UMA PROPOSTA DE TRANSFORMAÇÃO AUTOMATIZADA PARA GERAÇÃO DE PROGRAMAS PARALELOS A PARTIR DE LEIS ALGÉBRICAS APLICADAS A PROGRAMAS SEQUENCIAIS

Trabalho de Conclusão de Curso

Engenharia da Computação

Rafael Ferreira da Silva
Orientadora: Profª. Tarciana Dias

Universidade de Pernambuco
Escola Politécnica de Pernambuco
Graduação em Engenharia de Computação

RAFAEL FERREIRA DA SILVA

UMA PROPOSTA DE TRANSFORMAÇÃO AUTOMATIZADA PARA GERAÇÃO DE PROGRAMAS PARALELOS A PARTIR DE LEIS ALGÉBRICAS APLICADAS A PROGRAMAS SEQUENCIAIS

Monografia apresentada como requisito parcial para obtenção do diploma de Bacharel em Engenharia de Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Recife, maio 2012.

Escola Politécnica de Pernambuco
Graduação em Engenharia de Computação
Projeto de Final de Curso

POLI
ESCOLA POLITÉCNICA
DE PERNAMBUCO

MONOGRAFIA DE FINAL DE CURSO

Avaliação de defesa (cópia do aluno)

No dia 19 de 6 de 2012, às 10:00 horas, reuniu-se para deliberar a defesa da monografia de conclusão de curso do discente RAFAEL FERREIRA DA SILVA, orientado pelo professor Tarciana Dias da Silva, sob título Uma proposta de transformação auomatizada para geração de programas paralelos a partir de leis algébricas aplicadas a programas sequenciais, a banca composta pelos professores:

Maria Lencastre Pinheiro de Menezes Cruz

Tarciana Dias da Silva

Após a apresentação da monografia e discussão entre os membros da Banca, a mesma foi considerada:

☒ Aprovada ☐ Aprovada com Restrições* ☐ Reprovada

e foi-lhe atribuída nota: 9,5 (nove e meio)

*(Obrigatório o preenchimento do campo abaixo com comentários para o autor)

O discente terá _____ dias para entrega da versão final da monografia a contar da data deste documento.

Maria Lencastre

Maria Lencastre Pinheiro de Menezes Cruz

Tarciana Dias da Silva

TARCIANA DIAS DA SILVA

Aos meus pais.

Agradecimentos

Gostaria primeiramente de agradecer à minha mãe, meu pai, meu irmão, e a toda minha família, que estiveram sempre me apoiando e incentivando para que eu cumprisse mais este objetivo.

Agradeço à minha orientadora, professora Tarciana Dias, por sua dedicação, pelas suas idéias, conselhos e além de tudo, por ser uma ótima professora.

Agradeço a todos meus amigos pela constante preocupação e acompanhamento, durante esses anos de graduação.

Agradeço aos meus colegas, e também ao amigos de trabalho, pela compreensão, incentivo, divertimento e torcida.

Por fim, agradeço a todas as pessoas que, de alguma forma, contribuíram para que a realização deste trabalho fosse possível.

A todos, meu muito obrigado!

Rafael Ferreira da Silva

Resumo

A realização de computação paralela em sistemas computacionais sempre foi um desafio no desenvolvimento de código. Desenvolvedores e pessoas em geral pensam de forma sequencial e, normalmente, o objetivo maior na construção de um *software* é a sua adequação ou resolução de um problema num domínio de negócio ou contexto específico. Pouco se pensa em relação ao aproveitamento efetivo pelo *software* dos recursos computacionais disponíveis para sua execução. Além disso, mesmo que o desenvolvedor se disponha à construção de um código paralelizável, considerando as limitações para um compilador ou interpretador em fazê-lo, há a dificuldade da transformação manual do código sequencial para uma forma paralelizada, e ausência de mecanismos que o façam sem alterar a semântica dos códigos transformados. Este trabalho propõe uma abordagem de transformação automatizada de um código sequencial em paralelizado, baseada em estratégias de normalização e paralelização de código, construídas a partir de leis algébricas. Houve também uma preocupação em relação à não alteração da semântica original na transformação de tal código, mas esta limitou-se à realização dos mesmos testes de *benchmarking* a que foram submetidos códigos originais (sem a transformação).

Palavras-chave: Paralelização, Leis Algébricas, Normalização, Automatização, Java.

Abstract

The use of parallel computing in computer systems have always been a challenge in code development. Systems developers and people in general usually think sequentially and, normally, the main goal in software development is its suitability or solving a problem in a business domain or specific context. In general, system developers do not consider the effective use of the computational resources provided to a particular software execution. Furthermore, even if the systems developer is willing to build a parallel code, considering the compiler or interpreter limitations, it is difficult to transform manually a sequential code into a parallel code, and the lack of mechanisms to do it without changing the semantics of the transformed codes. In this work is proposed an approach to automatically transform a sequential code into a parallel one, based on normalization and code parallelization strategies developed using algebraic laws. Moreover, in order to check the consistence of the transformed code (i.e the semantic of the sequential and parallel code is the same), it was executed the same benchmark tests for both strategies.

Keywords: Parallelization, Algebraic Laws, Normalization, Automation, Java.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

ATL - Atlas Transformation Language

CPU - Central Processing Unit

EMF - Eclipse Modeling Framework

GFLOPS - Giga Floating-point Operations Per Second

IDEA - International Data Encryption Algorithm

JGB - Java Grande Benchmark Suite

JIT - Just-In-Time

JRPM - Java Runtime Parallelizing Machine

MDE - Modelagem de Domínio Específico

Capítulo 1

Introdução

Normalmente escolhidos como a melhor opção para computação de alto desempenho, os sistemas de processamento paralelo se mostram com dificuldades para serem inseridos no cotidiano do desenvolvimento de *software*. Com várias arquiteturas paralelas criadas [1] [2], e mesmo com uma otimização de desempenho nessas arquiteturas, o uso de processamento paralelo aparece com pouca frequência no desenvolvimento, mesmo com o aumento dos processadores e o ótimo custo/benefício proporcionado. Parte deste problema se deve à dificuldade de implementação por parte do programador nesse tipo de sistema.

O capítulo introdutório da monografia está dividido em quatro seções: a Seção 1.1 mostra a motivação da realização deste trabalho, além de apresentar o problema abordado pelo mesmo. Na Seção 1.2 é mostrado trabalhos similares ao apresentado nesta dissertação. Em seguida, na Seção 1.3 é apresentada a possível solução do problema e os objetivos deste trabalho. Por fim, na Seção 1.4 a estrutura do documento é descrita.

1.1 Motivação e Caracterização do Problema

Durante muitos anos, microprocessadores baseados em uma única unidade central de processamento (CPU), tais como os da família Intel Pentium e a família AMD Opteron, lideraram o aumento de desempenho e as reduções de custos na computação, com a possibilidade de execução de bilhões de operações de pontos flutuantes por segundo (GFLOPS – Giga Floating-point Operations Per Second) [3]. Esta evolução começou a estagnar por conta de limitações como o consumo de energia e dissipação de calor, o que limita a quantidade de atividades que podem ser feitas em um ciclo de *clock* dentro de uma única CPU. Dessa forma, a indústria tem mudado para modelos onde unidades de processamento múltiplo são usadas em cada chip para aumentar o poder de processamento.

Tradicionalmente, a maioria das aplicações de software é escrita como programas sequenciais, e a execução dos programas pode ser facilmente entendida navegando-se passo-a-passo pelos programas. Historicamente, durante a era dos microprocessadores baseados em uma única CPU, usuários e desenvolvedores estavam acostumados ao aumento de desempenho de suas aplicações ocorrer naturalmente a cada nova geração de microprocessadores, o que não é mais válido nos dias de hoje pelos motivos já apresentados acima. Os usuários, por sua vez, demandam ainda mais melhorias e poder de processamento criando assim um ciclo positivo para a indústria da computação.

Logo, são os programas paralelos que irão de fato usufruir dos ganhos de *performance* a cada nova geração dos microprocessadores de hoje. É observada uma clara evolução no poder de processamento computacional com a consolidação das arquiteturas *multicore* poderosas, *grids* computacionais, entre outros. O incentivo ao desenvolvimento de programas paralelos tem sido chamado como a revolução da concorrência [4].

Graças ao poder alcançado com a computação paralela, há uma tendência cada vez maior no aumento da demanda computacional. Como exemplos de aplicações que exigem alto poder de computação, temos as renderizações tridimensionais, previsão de movimentos de corpos celestes, estudos de sequenciamento genético, sísmicos e meteorológicos. Com isso, diversas possibilidades computacionais foram criadas para a resolução destes problemas, tais como a utilização de unidades de processamento gráfico, estruturas de *clusters* de computadores, a criação de supercomputadores, e até de computadores pessoais que possuem mais de um núcleo de processamento para processamento paralelo de processos.

Durante o desenvolvimento de programas, em geral, o foco maior é nas suas funcionalidades, no negócio da aplicação, na legibilidade do programa. Desta forma, os programas são desenvolvidos e testados de forma sequencial. Questões como a sua paralelização ou otimização de seu processamento e, consequentemente, melhor aproveitamento dos recursos disponíveis, nem sempre são levados em consideração. Isto é deixado a cargo dos compiladores das respectivas linguagens

e/ou máquinas virtuais, que muitas vezes, ficam bastante limitados a conseguir fazer uma paralelização efetiva.

A principal questão que vem à tona nesse aspecto é a semântica do programa (se esta vai ser ou não alterada). Desenvolver aplicações concorrentes e paralelas é, portanto, um desafio para os desenvolvedores, mesmo porque, tais programas podem apresentar problemas clássicos como *race conditions*, não-determinismo, *deadlocks* e *livelocks*, que não são observados em sua forma sequencial.

Portanto, há uma clara motivação para processos, técnicas e ferramentas, que ofereçam aos desenvolvedores um suporte sistematizado e, preferencialmente, automático, sem que os mesmos se preocupem com estas questões específicas de programação concorrente e paralela.

1.2 Objetivos

O objetivo deste trabalho é desenvolver uma solução inicial que venha a auxiliar os desenvolvedores em efetuar a paralelização de códigos, diminuindo o tempo com estas atividades, e deixando-os com um maior foco no projeto, e não na implementação das técnicas de paralelização.

1.2.1 Objetivos Gerais

Fazer a automatização da aplicação das leis algébricas, que possui como meta paralelizar um programa Java normalizando o seu código, e aplicando em seguida leis de paralelização. A também um cuidado em não prejudicar a produtividade do desenvolvedor, nem mudando a forma que eles desenvolvem programas hoje, e fazendo com que os recursos computacionais, disponíveis em arquiteturas distribuídas e *multicore*, possam ser aproveitados da melhor forma pelo *software*. A verificação da manutenção da semântica do programa original é feita a partir da realização dos mesmos testes de *benchmarking* nos programas originais e nos programas transformados. Vale salientar que não é o foco deste trabalho atestar de forma inquestionável que não houve alteração em absoluto da semântica dos programas. Isto requer um requinte maior no que diz respeito à formalização da solução a fim de se elaborar provas formais ou matemáticas que possam atestar

isso. Além disso, um estudo mais aprofundado da semântica de Java seria necessário – pelo que consta na literatura ainda não há uma definição padronizada da semântica de Java – e isto requer um tempo de dedicação bem superior ao disponível para um trabalho de conclusão de curso.

1.2.2 Objetivos Específicos

Para isto, os seguintes objetivos específicos foram traçados:

- Tornar a nossa abordagem um apoio a automatização o processo de normalização e paralelização;
- Utilizar algoritmos pré-definidos como estudos de caso;
- Projetar e desenvolver uma solução que seja capaz de efetuar a transformação de código sequencial em paralelo, de acordo com estudos de caso;
- Analisar os resultados para validação, repetindo os testes de *benchmarking* no código gerado pela nossa abordagem, a fim de atestar que de fato não houve mudança em relação ao comportamento do programa original.

1.3 Estrutura da Monografia

Este documento foi dividido em cinco capítulos, contando com este, tendo os restantes resumidos a seguir:

- **Capítulo 2: Fundamentação Teórica**

Reúne os principais conceitos necessários para a fundamentação teórica e compreensão do trabalho proposto. Para tal, são explicados os conceitos de paralelização, e os principais conceitos acerca das leis algébricas. Também será visto o conceito das leis algébricas de normalização e de paralelização, que são utilizadas para efetuar a transformação do código sequencial em paralelo.

- **Capítulo 3: Transformação Automática de Código**

Neste capítulo é apresentada uma abordagem para incorporação das leis algébricas de normalização e de paralelização em um programa sequencial, a fim de ter como resultado o mesmo programa de forma paralela.

- **Capítulo 4: Estudo de Caso e Resultados**

Contém todos os resultados, com os estudos de caso realizados, que visaram a validação do modelo desenvolvido. Os resultados serão analisados para extração de conclusões.

- **Capítulo 5: Conclusões e Trabalhos Futuros**

Apresenta a conclusão do trabalho, enfatizando as contribuições realizadas e enunciando possíveis trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo consiste na apresentação dos conceitos básicos necessários para o entendimento do estudo realizado. Primeiramente, apresentaremos alguns trabalhos relacionados à nossa abordagem. Também será exposto como se dá a paralelização, seguindo com o seu processo e forma de se mensurar o desempenho de tal paralelização. Também será introduzido a forma de paralelização que foi feito nesse trabalho, através das leis algébricas de normalização e paralelização.

2.1 Trabalhos Relacionados

Algumas abordagens têm sido propostas para explorar paralelismo implícito em Java, tais como [5] [6] [7] [8], cujo foco é o nível de código fonte e *bytecode*, introduzindo-se código extra na aplicação, ou através de ferramentas adicionais em tempo de execução. Mais detalhes desses tipos de ferramentas, serão explicados na seção 2.2.3.

O trabalho descrito em [9] aborda a paralelização de programas sem modificar a semântica do programa, sendo aplicada em cima de um algoritmo pré-existente. Tendo assim, como foco, permitir que o programa seja estruturado de maneira correta para a paralelização, com base nas leis de normalização, e então, aplicar as regras de paralelização que transformará o código apenas em sua forma sintática, fazendo com que a paralelização seja possibilitada. Esse trabalho utiliza leis de transformação para converter um programa Java em uma forma normal que utiliza um conjunto restrito de recursos da linguagem. A partir de um programa na forma normal, são utilizadas regras de transformação focadas em introduzir paralelismo. Após a aplicação dessas regras, de acordo com a estretégia desenvolvida, um programa paralelo é produzido. Dois casos de estudo foram realizados para validar a abordagem: cálculo de séries de Fourier e o algoritmo de criptografia IDEA. Ambos os códigos foram obtidos do *Java Grande Benchmark* (JGB). Nosso trabalho pode ser considerado uma extensão do trabalho descrito em

[9], no sentido de automatizar a aplicação das leis algébricas que é feita no referido trabalho de forma inteiramente manual, diminuindo assim a probabilidade de erros durante o processo de transformação, já que a interferência humana é reduzida. A execução dos estudos de caso permite avaliar o êxito da abordagem em melhorar a *performance* do código original, assim, utilizaremos os mesmos estudos de caso da proposta manual, pois com isso, teremos os códigos como referências para futuras comparações entre uma forma manual, e uma maneira automatizada na implementação das leis de normalização e paralelização, que é a nossa proposta.

2.2 Paralelização

Computação paralela ou processamento paralelo constitui-se na exploração de eventos computacionais concorrentes, através de unidades de processamento que cooperam e comunicam-se entre si [10]. Buscar um melhor desempenho é basicamente a tarefa do processamento paralelo, principalmente para aplicações que necessitam de uma maior potência computacional, mas muitas vezes se encontram em sua forma sequencial, não aproveitando todos os recursos disponíveis.

Existem vários tipos de exploração de eventos concorrentes, começando pelo hardware, onde o paralelismo pode existir nas unidades funcionais que compõem a CPU. Já sob as instruções de máquina, o paralelismo pode ser obtido através de ambientes de paralelização automática, onde a partir de um programa sequencial um programa paralelo é gerado, tendo o compilador como responsável por isso. O paralelismo pode ser explorado também, em um nível intermediário, pelo uso de procedimentos de programas paralelos, que serão executados concorrentemente.

O aumento de desempenho no processo computacional, utilizando o paralelismo em níveis de instruções de máquina e procedimentos, é algo factível. Porém, uma solução altamente empregada é a divisão do trabalho a ser realizado, em um nível mais alto, em tarefas, que serão executadas concorrentemente. Tendo isso em mente, basicamente, uma aplicação paralela é um conjunto de tarefas que interagem entre si para realizar um determinado trabalho.

Em termos gerais, essa paralelização pode aparecer de três formas [11]:

- Paralelização explícita, onde o programador explicita as tarefas a serem executadas em paralelo, e a forma como elas devem cooperar entre si;
- Paralelização implícita, onde o paralelismo é restrito à semântica de alguns comandos e construções. Neste caso, o programador não precisa descrever como se dará a sequencialização;
- Paralelização automática, onde o programador utiliza uma linguagem sequencial tradicional, e o compilador é responsável por efetuar automaticamente o paralelismo.

Cada uma dessas formas ainda apresentam fatores que não foram resolvidos satisfatoriamente. A paralelização explícita, define de forma manual como se dará a paralelização, em outras palavras, o programador irá considerar fatores para definir a arquitetura paralela do sistema, e muitas vezes, acaba se afastando da própria lógica do programa. Apesar de se ter um maior controle com o uso da paralelização explícita, há um custo para isso: o desenvolvimento acaba sendo mais lento, e muitas vezes, quando o algoritmo possui alguma modificação em curso, a validade dessa paralelização, acaba sendo prejudicada.

Com o uso restrito à semântica de alguns comandos e construções, a paralelização implícita exige que o programador aprenda uma linguagem nova, e descreva o seu algoritmo nessa linguagem. Em grande parte, a linguagem escolhida não possui uma eficiência tão boa, principalmente fora do contexto na qual é usada para especificar uma arquitetura paralela. Além do que, as duas formas apresentadas possuem um grande problema, de que um programa que já está consolidado e foi testado em sua forma sequencial, possivelmente não poderá ser levado para sua forma paralela, se mantendo com a mesma estabilidade. Com isso, temos o conceito de paralelização automática, que tenta resolver este problema. Porém, os métodos de paralelização automática não se difundiram, por não terem apresentado resultados com grandes efeitos. Mas ainda representa uma área promissora, e é onde o contexto do nosso trabalho se insere.

2.2.1 Processos de paralelização

Etapas chave podem ser definidas no processo de paralelização: primeiramente, é necessário fazer a detecção e extração das dependências de

dados de um programa, que por ventura, irão impedir que o paralelismo possa ser alcançado; em seguida, deve-se eliminar qualquer tipo de dependência desnecessária, tendo em vista que irá diminuir o acoplamento, facilitando a paralelização. Um grande número de técnicas já foram propostas na literatura [12] que se propõem com sucesso eliminar algumas das dependências, que em grande parte, ocorrem devido a falhas na própria implementação dos algoritmos. Por fim, é necessário analisar se o programa em questão será ou não paralelizado, de acordo com as informações obtidas até então.

O processo de paralelização tenta, por fim, fazer com que o programa aproveite os múltiplos processadores de uma máquina, diminuindo o tempo despendido.

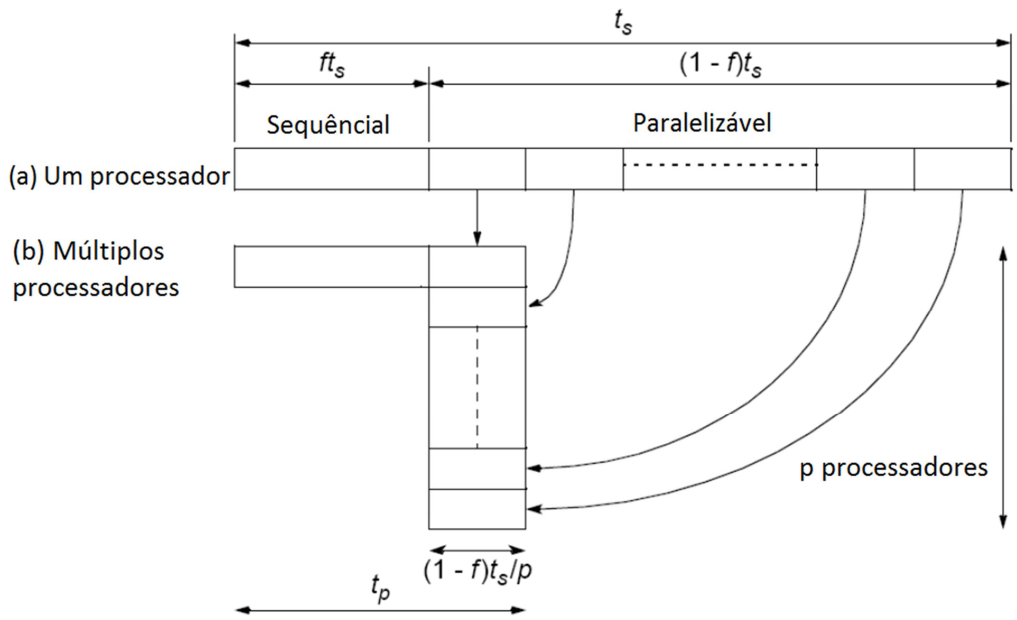


Figura 1. Instruções seriais versus paralelas

Na figura 1, vemos a execução de um código, que possui parte sequencial, e parte cabível de paralelização. Em (a) a execução ocorre com apenas um processador, ficando de forma sequencial toda a execução, mesmo que parte daquele código pudesse ter sido executado em paralelo, gastando com isso um tempo t_s. Já em (b), utilizando múltiplos processadores, vemos que a parte que era factível de paralelização, teve o seu tempo de execução bastante reduzido, de acordo com os processadores utilizados, p.

Porém, há casos em que as dependências existentes não permitem a execução de parte do programa de forma paralela, nesses casos o programa deverá ser executado sequencialmente.

2.2.2 Desempenho em programas paralelos

Uma das formas de se definir se um programa deve ser executado em sua forma paralela ou sequencial é avaliar o desempenho e verificar qual o melhor. O cálculo do *speedup* é uma das formas de se fazer isso, ele nada mais é do que a razão do tempo de execução de um algoritmo sequencial, executado em um processador de máquina paralela T(n), pelo tempo de execução do algoritmo paralelo em p processadores da máquina paralela T(n,p), como visto na Figura 2.

$$S(n,p) = \frac{T(n)}{T(n,p)}$$

Figura 2. Cálculo de speedup

O *speedup* nos dá um indicador da velocidade por usarmos uma máquina paralela. De forma geral, temos: $0 < S(n,p) \leq p$.

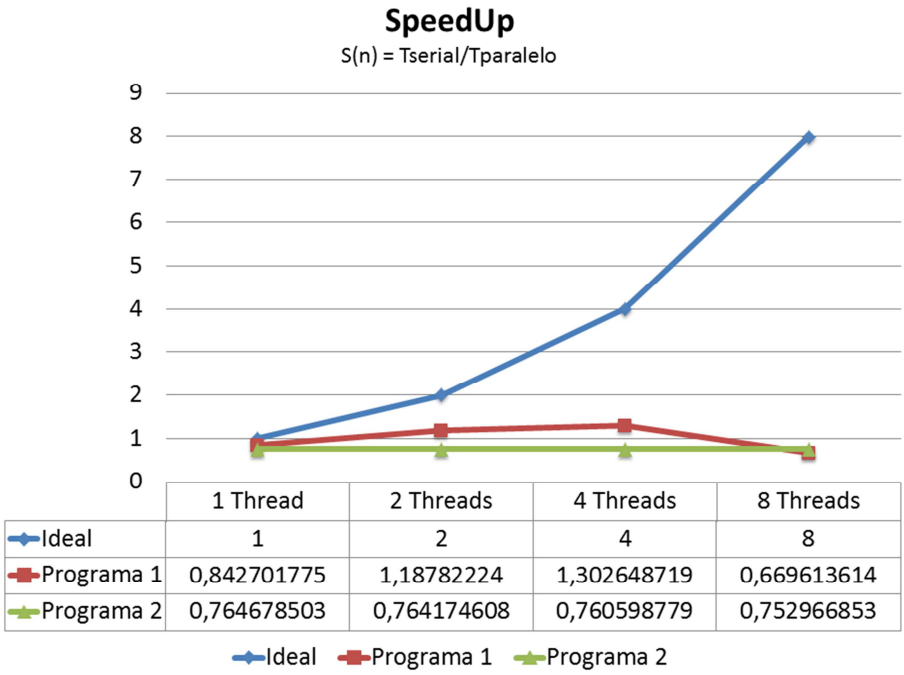
Se S(n,p) = p, teremos um *speedup* linear, que ocorre raramente, pois grande parte das soluções paralelas colocam algum tipo de sobrecarga na distribuição de carga e comunicação entre processos.

Se houver grande sobrecarga da paralelização teremos o chamado *slowdown*, uma situação indesejável, pois haveria um melhor desempenho com uma forma sequencial, com execução T(n), sendo menor que a execução de forma paralela T(n,p). Teriamos T(n) < T(n,p) e S(n,p) < 1.

Em suma, temos para os *speedups*:

- S(n; p) < 1, *slowdown*, situação indesejável;
- 1 < S(n; p) < p, sublinear, comportamento geral;
- S(n; p) = p, linear, ideal, não existe sobrecarga;

- S(n; p) > p, supralinear, situação possível;



	1 Thread	2 Threads	4 Threads	8 Threads
Ideal	1	2	4	8
Programa 1	0,842701775	1,18782224	1,302648719	0,669613614
Programa 2	0,764678503	0,764174608	0,760598779	0,752966853

Figura 3. Gráfico de *speedup*

Na Figura 3, temos três cenários de execução de programas paralelos, que refletem de forma sucinta e visual, o que se foi explicado até agora. Em azul, e marcado como círculo, o chamado “Ideal”, temos a situação em que de acordo com o número de *threads* utilizadas para a execução, temos um maior ganho no *speedup* sem nenhuma perda, seria o *speedup* linear, porém, essa situação nem sempre reflete a realidade na execução dos programas paralelos. Na segunda situação, temos em vermelho, e marcado como quadrado, a execução do chamado “Programa 1”, que mostra um aumento de *speedup* de acordo com o número de threads, até um certo ponto, após isso, o desempenho pode ficar, por vezes, pior do que a execução sequencial do programa (0,66 para 8 threads e 0,84 para 1 thread). Por fim, temos o “Programa 2”, em vermelho, e marcado como triângulo, que não possui uma grande variação de *speedup*, mesmo com o aumento na quantidade de *threads*.

Outras medidas que podem ser utilizadas, são: eficiência e tomadas de tempo, que aprimoram a utilização dos processos em um programa paralelo em relação a um programa sequencial, e avalia o desempenho de programas, com um tipo de cronômetro.

Com esse tipo de análise, foi formulada a lei de Amdahl [13], mostrando que o *speedup* obtido ao se paralelizar um programa é limitado. Segundo a lei, o ganho de desempenho que pode ser obtido melhorando uma determinada parte do sistema é limitado pela fração de tempo que essa parte é utilizada pelo sistema durante a sua operação. Em forma de exemplo, para um melhor entendimento, faremos uma analogia com a situação do pintor de estacas. Na situação, temos três passos para a pintura das estacas: a primeira, é preparar apenas a tinta, gastando-se 30 segundos; na segunda, efetuar a pintura de fato, gastando mais 300 segundos; e na terceira e última, 30 segundos para esperar a tinta secar. Nota-se que o passo 2, é factível de ser executado em paralelo, ao contrário dos passos 1 e 3, que só podem ser efetuados de forma sequencial.

Tabela 1. Exemplificação da lei de Amdahl

Pintores	Tempo	Speedup
1	30 + 300 + 30 = 360	1.0
2	30 + 150 + 30 = 210	1.7
10	30 + 30 + 30 = 90	4.0
100	30 + 3 + 30 = 63	5.7
∞	30 + 0 + 30 = 60	6.0

Analisando a tabela 1, vemos que, mesmo elevando o número de pintores para as estacas, existem etapas que independente da quantidade de pintores, terão sempre o mesmo tempo de execução, são as partes sequenciais. Da mesma forma acontece no número de threads utilizadas em um programa. Tendo f como a fração de operações em uma computação que deve ser executada sequencialmente, onde $0 \leq f < 1$, temos o seguinte gráfico com o *speedup* máximo, segundo a lei de Amdahl:

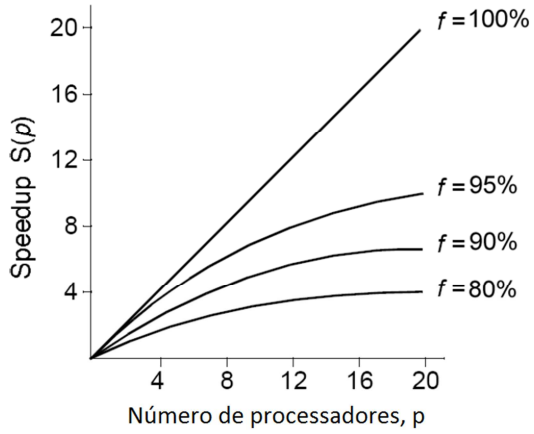


Figura 4. Speedup máximo segunda a lei de Ahmdal

Com isso, a lei de Amdahl desencoraja a utilização massiva de paralelismo. Em [14] é visto que, para programas rodando com um certo número de núcleos, não é recomendado dividir tal programa em um número de processos maior que o de núcleos.

2.2.3 Ferramentas de paralelização

Diversas são as ferramentas que automatizam o processo de transformação de um algoritmo que está em sua forma sequencial em uma forma paralelizável. Os níveis em que tais ferramentas trabalham são diversos, desde refatorações até transformações de *bytecode*.

Uma proposta semelhante ao do presente estudo é o da utilização da biblioteca j.u.c. (java.util.concurrent). Ela faz a transformação automática para o programador de um código sequencial em um paralelizável [15]. Aborda também fatores como a otimização da escalabilidade, e *threads* seguras, para deixar um programa concorrente, além da utilização de algoritmos *divide-and-conquer*. Porém, de antemão, os programadores necessitariam alterar a arquitetura do código existente para usar o j.u.c., tarefa que acarretaria na mudança de muitas linhas de código, e grande chance de erro, pois os programadores poderiam fazer mal uso de tal biblioteca. No estudo, não seria necessário o uso de “anotações” no código por parte do programador, utilizaria-se uma heurística de procura por blocos de códigos específicos, transformando-os para a utilização do j.u.c., fazendo a paralelização internamente em seu código.

Existem situações em que a modificação ocorre em um nível mais baixo, como em [7], com a utilização de técnicas baseadas em transformação de *bytecode* e execução paralela de métodos. No trabalho, enfatiza-se que só funciona com alguns tipos de aplicação, com tarefas livres de sincronização (*loosely-synchronous tasks*). O termo “semi”, do estudo referido diz respeito à necessidade de escolha de quais das regras de transformação serão aplicadas (se é necessário a criação de *thread* ou não, que tipos de métodos devem ser paralelizados, paralelização de for, etc). O que é feito é a inserção de código quando a classe está sendo carregada pelo JVM, manipulando o bytecode. O estudo também deixa claro que só é eficiente em aplicações que são naturalmente paralelizáveis (i.e. *well-engineered object-oriented applications,* com uma estrutura modular, encapsulamento). Programas com estruturas complexas precisam sofrer modificações no código fonte, sendo preparadas para a paralelização. O trabalho cita como limitação o processo de descoberta das classes e métodos que precisam ser paralelizados (e os respectivos erros nessas descobertas).

Paralelização dinâmica em nível de *hardware* é outro tipo de proposta [8], utilizando o Jrpm (Java runtime parallelizing machine), que é utilizado com um multiprocessador com suporte a *thread* (SLT). Com isso, em tempo real, são analisados os melhores *loops* (cada *loop* viraria uma *thread*) e trechos de códigos para serem paralelizados. Quando são selecionados, tal parte do código é recompilada dinamicamente para rodar em paralelo, sem interferência do programador. Opera em nível *assembly* (trabalhando com registradores), e o trabalho também cita que seria possível utilizar formas de marcar manualmente partes do código para serem paralelizados, assim como será abordado mais adiante neste trabalho.

O estado da arte se encontra em *Application Programming Interface’s* (API’s), conjunto de rotinas e padrões, como o *OpenMP (Open Multi-Processing).* É constituído por várias diretivas de compilador, rotinas de biblioteca e variáveis de ambiente. Elas expressam o paralelismo, e modificam em tempo de execução, no comportamento da aplicação. O *OpenMP* permite acrescentar simultaneidade aos programas escritos em C, C++ e Fortran, sobre a base do modelo de execução *fork-join*. Além disso, é um modelo de programação portável e escalável que proporciona

aos programadores uma interface simples e flexível para o desenvolvimento de aplicações paralelas.

2.3 Leis de normalização e paralelização

A normalização e paralelização, em forma de leis, tenta formalizar a aplicação das mesmas no contexto do desenvolvimento de *software*. Tais leis são baseadas em leis algébricas, que definem equações que estabelecem equivalências entre elementos de uma linguagem, o que é bastante útil na concepção de provas e verificação de sistemas. Leis algébricas para linguagens de programação adotam um princípio também utilizado na matemática [16], onde a teoria é feita de acordo com leis axiomáticas. Exemplos podem ser vistos na aritmética, como a simetria na utilização das operações de multiplicação ($x \times y = y \times x$) e adição ($x + y = y + x$). A normalização é uma estratégia de redução que transforma um programa em uma forma normal, que utiliza um conjunto limitado de características de uma linguagem. As leis de paralelização fornecem meios de aumentar o desempenho de um algoritmo sequencial, explorando o uso dos múltiplos processadores disponíveis.

Como já foi dito, este trabalho é uma extensão do trabalho feito em [9], portanto, as leis utilizadas para a elaboração deste estudo, foram retiradas do mesmo trabalho, que já sofreu adaptações de outros trabalhos, como a abordagem [17]. As leis seguem uma convenção que obedece às condições exigidas por cada uma delas, chamadas de provisos. Condições marcadas com (↔) precisam ser seguidas quando a transformação é efetuada em ambas as direções; Condições marcadas com (→), precisam ser seguidas quando a transformação ocorre da esquerda para a direita, e as marcadas com (←), quando ocorrer da direita para a esquerda.

Conceitos de atributos, construtores e declaração de métodos, também são representadas nas leis, por *ads*, *cnds* e *mds*, respectivamente. Em seguida, é utilizada a notação *cds* para a representação de um conjunto de classes, assim como *Main* é a única classe que possui o método *main* no sistema. Para representar um tipo a letra T é utilizada, enquanto que para representar a relação de subtipo entre classes o símbolo ≤ é utilizado. É considerado que todas as declarações internas de classes são feitas dentro de um pacote padrão, analogamente a Java.

Para demonstrar como se dão essas representações, será apresentado um exemplo com a lei 1 do trabalho [9]. O objetivo da lei é fazer eliminação ou introdução de uma classe, representada por cd_1, mostrada a seguir:

Lei 1. (eliminação / introdução de classe)
cds cd_1 *Main* = *cds Main*

provisos

(→) A classe declarada em cd_1 não é referenciada em *cds* ou *Main*;
(←) (1) O nome da classe declarada em cd_1 é diferente de todas as classes declaradas em cds;
(2) A superclasse de cd_1 é *Object* ou alguma declarada em *cds*.

Na lei apresentada, a eliminação da classe ocorrerá da esquerda para a direita, enquanto a introdução da classe ocorrerá da direita para a esquerda. Porém, para que elas aconteçam os provisos devem ser satisfeitos. No primeiro caso, uma condição apenas é apresentada, a de que para eliminar cd_1 ela não deve ter sido referenciada no conjunto de classes *cds*. Enquanto que para a segunda situação existem duas condições: (1) não deve haver uma classe, no conjunto de classes *cds* ou *Main,* que possua o mesmo nome da classe que está sendo inserida, no caso cd_1; (2) A superclasse da classe a ser inserida cd_1, deve ser ou *Object* ou alguma que já tenha sido declarada no conjunto de classes *cds.* O *Object* está sendo considerado como uma classe válida, pois é a classe padrão utilizada por Java.

Capítulo 3

Automatização das leis

Este capítulo aborda a principal contribuição deste trabalho, ou seja, como foi dada a automação das leis de normalização e paralelização, com base em Java. Ela foi escolhida para tal transformação por se tratar da linguagem base para concepção das leis, e dos estudos de caso escolhidos para a validação também terem sido implementados em Java. A linguagem também possui várias características [18] que se mostram pertinentes ao nosso estudo, como:

- Simples e orientado a objetos – a orientação a objetos é o paradigma dominante atualmente, o que torna essa característica muito importante;
- Robusto – compila, e em tempo de execução garante a confiabilidade do programa;
- Distribuído e seguro – Java foi criado para ser executado em ambientes distribuídos, por isso já incorpora diversos aspectos de segurança;
- Interpretado, portável e arquitetura neutra – utiliza uma máquina virtual para executar o *bytecode* (o código compilado de java), que é independente de plataforma, e provê a portabilidade para ser executado em diferentes plataformas;
- Alto desempenho – técnicas como a compilação *just-in-time* (JIT), permite que códigos críticos de desempenho sejam compilados para código nativo, acelerando a sua execução e diminuindo os *overheads* causados pela interpretação;
- Dinâmico – classes são carregadas apenas quando necessárias, e novas classes podem ser adicionadas de acordo com a evolução do sistema;
- Processamento paralelo – possui um suporte embutido para programação concorrente, permitindo o desenvolvimento de processos executando paralelamente.

3.1 Abordagens para transformação

Uma das abordagens analisadas, para a execução do presente trabalho, foi a de se trabalhar com os conceitos de modelo e metamodelo utilizando o Eclipse Modeling Framework (EMF), que proporciona o desenvolvimento de aplicações envolvendo modelagem de domínio específico (MDE).

O EMF é um *framework* de auxílio na modelagem e geração de código para o desenvolvimento de aplicação e ferramentas baseadas em modelos bem estruturados. Um ponto importante é que sua plataforma é *open-source*, dando oportunidade aos desenvolvedores criarem novas ferramentas para integrá-las ao *framework* [19].

Os modelos podem ser descritos também usando uma sintaxe de Java com anotações ou documentos XML. Um destaque para essa plataforma é que ela fornece uma série de padrões que permitem uma grande interoperabilidade entre as diversas ferramentas que podem ser integradas ao framework EMF. A modelagem dos metamodelos se dá através de uma das ferramentas que dão suporte a sua criação, chamada Xtext. Nela, a linguagem é definida através de um metamodelo, onde este é descrito usando uma sintaxe concreta que se assemelha com a descrição de uma gramática. A linguagem e o motor de transformação para a implementação das regras de transformação para a execução das transformações, são feitos através de ATL (Atlas Transformation Language). Porém, essa proposta não foi implementada neste projeto, devido à falta de experiência com esse tipo de solução, onde seria necessário um estudo prévio além do tempo previsto, se caracterizando como um possível tópico de pesquisa futura.

Outra abordagem analisada, e também a que foi escolhida, foi a de se trabalhar em cima do *parser* existente da linguagem Java. A escolha dessa abordagem se deu pelo simples fato da experiência em se trabalhar com compiladores, onde além de tornar a implementação da proposta viável, nos daria uma maior possibilidade de referências por parte da comunidade acadêmica, assim como foi mostrado anteriormente na seção 2.2.3, onde foram apresentadas abordagens que utilizam a fase de compilação para a transformação. Para efetuar a geração do *parser*, foi obtida a gramática do Java com a Sun, e então foi escolhido um gerador de analisador sintático aberto para a linguagem Java, o JavaCC.

3.1.1 JavaCC

O programa JavaCC é um gerador de analisador sintático que produz código Java. Ele permite que uma determinada linguagem seja definida de maneira simples, por meio de uma notação semelhante à EBNF. Como saída produz o código-fonte de algumas classes Java, que implementam os analisadores léxico e sintático para aquela linguagem. Provê também maneiras de incluir, junto à definição da linguagem, código Java para por exemplo, construir-se a árvore de derivação do programa analisado.

Apesar disso, para nossa solução, iremos apenas resgatar o código que é retornado, após a geração da árvore. O JavaCC define uma linguagem própria para descrição, em um único arquivo, do analisador léxico e do analisado sintático. Iniciando com o analisador léxico, esta linguagem permite que cada *token* seja definido na forma de uma expressão regular. Um arquivo de especificação JavaCC contém três blocos: opções do JavaCC, classes do *parser* e gramática da linguagem, com ações semânticas associadas.

O primeiro bloco, opcional, contém um conjunto de opções que definem como o JavaCC irá gerar o *parser,* como: se será sensível ao *case*, se gerará todas as classes usuais, se haverá checagem de ambiguidade na gramática, etc. O bloco seguinte define qual é o nome da classe do parser e implementa sua chamada. O restante da especificação JavaCC conterá a especificação dos *tokens* da linguagem e da sua gramática. O JavaCC produz então o analisador léxico e sintático do compilador, não necessitando de uma implementação ou de ferramentas extras.

Entretanto, o JavaCC gera analisadores sintáticos descendentes, o que o limita às classes gramaticais LL(k) (excluindo, por exemplo, recursividade à esquerda). Porém, nosso trabalho não chega ao nível de complexidade que necessitaria de tal transformação. Como já foi dito, apesar de gerar também uma árvore, a saída do JavaCC que nos interessa é a geração do próprio código em Java, que, ao passar pelas modificações feitas no *parser* com os conceitos das leis algébricas, nos retornará a saída da transformação automática, que será levada para os testes de *benchmarking* para comparação com as outras abordagens já efetuadas com os estudos de caso.

3.2 Implementação

Para efetuar a implementação da transformação foi seguida uma heurística, definindo quais leis deveriam ser aplicadas no contexto do caso de uso escolhido para validação. Algumas dessas leis não são aplicáveis nesse contexto, o que nos leva a não implementá-las, por ora, e logo, não serão descritas. Na transformação manual, foram analisadas várias dessas leis que não teriam transformação com os estudos de caso, porém, por se tratar de uma proposta manual, foram feitas modificações que não seguem o emprego de algumas leis da forma que é especificada pelos provisos, incluindo ou abstraindo alguns.

No caso de uma proposta automatizada, tais mudanças a priori não são factíveis, o que nos coloca em uma abordagem seguindo com conformidade a implementação das leis, que muitas vezes não nos proporciona a otimização desejada, que poderia ser feita manualmente. Em contrapartida, o tempo despendido para empregar as leis em uma proposta manual é maior com relação a uma abordagem automatizada já implementada, o que nos colocaria em uma zona de impasse, sobre qual das propostas utilizar em uma aplicação real. Tal escolha se deverá ao fato de qual fator irá predominar como mais crítico, se o tempo de execução do algoritmo ou o tempo de emprego para a implementação desse algoritmo. A ideia base da implementação foi tentar transformar cada lei que seria empregada, dentro dos conceitos da linguagem Java, que foi escolhida como padrão para o nosso estudo. Assim, seria feita uma análise de como se daria a implementação de cada lei, dentro do Java. Isso foi facilitado devido à forma com que as leis, tanto de normalização, quanto as de paralelização, são expressas, bastante próximo do que seria aproveitado para a implementação da solução de fato. Em base para explicar a abordagem das leis nos algoritmos, utilizaremos a implementação do estudo de caso IDEA e suas classes, que serão melhor apresentadas no capítulo 4, como analogia para exemplificação.

3.2.1 Leis de Normalização

Primeiramente será seguida uma estratégia de normalização, com as leis de normalização vistas em [9], para depois se seguir com as leis de paralelização. Temos a aplicação da lei 1, vista na seção 2.2, inserindo uma nova classe chamada

de *_Object*. A lei 2, aplicada em cima dos provisos, visa modificar a hierarquia das classes utilizando a nova classe criada, como uma nova superclasse raiz, onde todas as classes que não herdarem de outra classe explicitamente, ou seja, que herdam de *Object,* irão agora herdar de *_Object.* Para o nosso estudo de caso, a classe que se enquadra nessa situação é a *IDEATest*, que agora terá uma nova superclasse.

Após introduzir uma nova superclasse raiz, todos os atributos terão suas visibilidades alteradas para públicas, ou seja, aquelas que englobam as visibilidades: *default, protected* e *private*, agora possuirão a visibilidade *public,* respectivamente correspondem as leis 3, 4 e 5. Todas as três leis foram implementadas, porém, para o estudo de caso, não haveria necessidade da aplicação da lei 4, já que não existe nenhum atributo com a visibilidade *protected*, e logo, não foi aplicada, porém ainda foi implementada. Um cuidado ao ter sido aplicada a lei 5 é um proviso com relação à visibilidade *private*, de acordo com a herança dada pela classe onde o atributo está inserido, que vai limitar o escopo de visibilidade dentro dessa herança. A modificação no *parser* que permitiu a implementação de tais leis, foi com uma verificação feita no retorno dos *tokens* de visibilidade para os atributos, retornando a visibilidade *public* na leitura desses *tokens,* mas, apenas para os atributos que pertencem a classe, ou seja, globais.

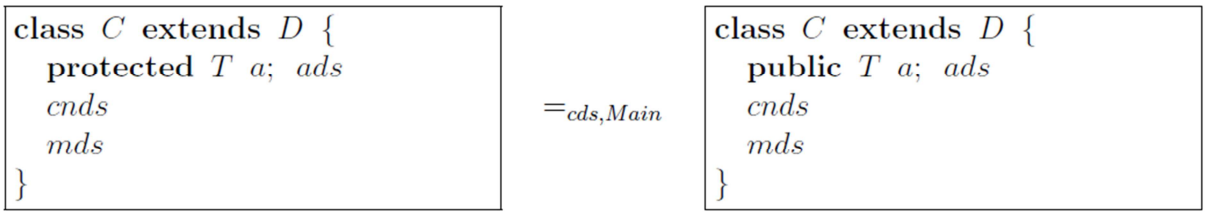


Figura 5. Lei 4 – Alteração de visibilidade do atributo de *protected* para *public*

A lei 6, passa os atributos de uma subclasse para a sua respectiva superclasse, até chegar na superclasse raiz, que na nossa situação é a nova classe criada anteriormente, *_Object.* Um cuidado ao se implementar essa lei, aos provisos em ambos os sentidos, se dá na preocupação que o atributo que for movido da subclasse para a superclasse (ou vice-versa, dependendo do sentido) não poderá se encontrar na respectiva superclasse, deixando o algoritmo inconsistente em termos de escopo semântico.

As leis 8, 9 e 10, ocorrem de maneira análogas às leis 3, 4 e 5, vistas anteriormente, porém, ao invés de se manipular a visibilidade dos atributos, essas leis manipulam a visibilidade dos métodos. A lei 8 foi aplicada para os únicos métodos com modificador *default*, *buildTestData*, *freeTestData* e *Do.* A lei 9 é implementada, porém não é aplicada pela ausência da visibilidade *protected* nos métodos. Já a lei 10 foi aplicada normalmente. O enquadramento do parser se deu da mesma forma de como foi feita com os atributos, fazendo a verificação de como o *token* de visibilidade seria lido pelo parser, e retornado para a árvore sintática.

A lei 14 faz a mudança dos métodos de uma classe, movendo-as para a sua superclasse, tendo cuidado mais uma vez, com relação ao escopo de nomes, que não podem possuir o mesmo nome de método, a não ser nos casos de sobrecarga (métodos com o mesmo nome e assinatura diferente). Para o estudo de caso, foi feita essa passagem apenas da classe *IDEATest*, para a classe *_Object*. Com isso, a classe *IDEATest* agora não possuirá nenhum método ou atributo, tendo em vista a aplicação das leis 6 e 14. Uma análise pertinente aos seus provisos diz respeito à criação de um novo proviso, englobando a situação para quando os métodos fossem derivados de uma interface ou de uma classe abstrata, e como tal proviso não existe, os métodos não podem ser passados para cima, foi o caso do *IDEARunner* e do *JGFCryptBench*. Essa análise foi feita na proposta manual, entre várias outras situações que foram efetuadas para a aplicação manual das leis, tendo em vista a adequação ao estudo de caso para um melhor resultado. Tais modificações não entram como escopo para este trabalho, pois nele as leis estão sendo implementadas sem modificações com relação aos provisos, o que pode vir a influenciar na comparação dos resultados, que será visto adiante. Para efetuar a aplicação dessas leis, foi criada uma nova classe temporária, sem conteúdo, e deixando-a como a nova *IDEATest,* em contrapartida, o *_Object* fica com os dados do antigo *IDEATest*, e dos atributos das outras classes que foram passados para o mesmo.

3.2.2 Leis de Paralelização

Iniciada pela lei 45, que prega a modificação da ordem dos comandos, é um passo bastante útil para agrupar comandos relacionados, colocando-os em sua ordem de dependência, facilitando sua execução em diferentes *threads*. Entretanto,

para aplicar esta lei os comandos precisam ser independentes, pois assim poderemos garantir uma maior consistência semântica na execução das threads. Para o nosso caso, a ordem dos comandos já estava numa seqüência apropriada. Se mudássemos a ordem implicaria na mudança de semântica do programa, já que os comandos dependiam de outros comandos imediatamente anteriores.

Laços que possuem comandos independentes no seu corpo podem manter uma paralelização, tendo duas formas de se iniciar tal paralelização: paralelizar todo o laço, ou paralelizar cada iteração. Dividi-lo em dois laços, cada um contendo comandos relacionados, aumenta a possibilidade de serem executados em threads múltiplas.

$$\boxed{\begin{array}{l} for(init;\ cond;\ incr)\ \{ \\ \quad cmds_1; \\ \quad cmds_2 \\ \} \end{array}} = \boxed{\begin{array}{l} for(init;\ cond;\ incr)\ \{ \\ \quad cmds_1 \\ \} \\ for(init;\ cond;\ incr)\ \{ \\ \quad cmds_2 \\ \} \end{array}}$$

Figura 6. Lei 46 – Fatoração de laço

Na figura 7, os provisos deixam claro que, na transformação da esquerda para a direita, o comando 1 ($cmds_1$) deverá ser independente do comando 2 ($cmds_2$), assim como ambos os comandos, deverão ser independentes da condição do laço (*cond*). Porém, a lei 46 não foi aplicada, pois não há independência de comandos dentro do *loop*. Essa lei também não foi implementada, devido à dificuldade em se destacar comandos independentes de uma maneira automatizada, e ainda mesmo, de maneira manual, com alguma forma de marcação.

Ainda em se tratando de laços, a lei 47 divide o laço original, onde cada novo laço criado irá executar uma parte da iteração do laço original.

$$\boxed{\begin{array}{l} for(int\ i = K;\ i < F;\ i = i{+}inc) \\ \quad cmds_i \end{array}} = \boxed{\begin{array}{l} for(int\ i = K;\ i < J;\ i = i{+}inc) \\ \quad cmds_i \\ for(int\ j = J;\ j < F;\ j = j{+}inc) \\ \quad cmds_i \end{array}}$$

Figura 7. Lei 47 – Divisão das iteraçõs do laço

Para implementar a lei 47 foi criada uma heurística para dividir o laço em dois novos laços. O primeiro laço irá ser efetuado até à primeira metade da condição, enquanto o segundo laço será efetuado a partir do fim da primeira metade, até à segunda metade da condição. Na figura 7, se tivéssemos K = 0 e F = 8 no primeiro laço, teríamos J = 8/2, fazendo com que a divisão dos laços se dê de forma correta.

A lei 48 especifica a transformação necessária para executar comandos concorrentemente. A introdução dessa lei precisa ser feita de forma cuidadosa, para uma execução concorrente, por isso não foi implementada para ser aplicada de forma automática. O processo de descoberta, de quais comandos iriam ser aplicados por essa lei, é algo que não vem ao propósito desse estudo, e que é inclusive uma das principais dificuldades na paralelização de algoritmos. Para fazer essa aplicação de forma semi-automática, nós fizemos uso do recurso de marcação no código, para deixar visível para a transformação quais blocos serão aplicados pela lei.

Inicialmente se pensou em fazer tal marcação com o uso do *annotation* do Java, que faz a demarcação de algo que pode ser posteriormente interpretadas por um compilador ou pré compilador que irá realizar alguma tarefa pré definida. Porém, o uso de *annotation* é restrito ao escopo de classes e métodos, o que nos deixa aquém da aplicação da lei, para comandos ou blocos específicos. Com isso, foi pensada em uma maneira de se marcar o código, sem se fazer qualquer influência no código, e sem alterar o *parser*, com o uso de comentários com uma marcação específica, aos mesmos moldes do *annotation*, utilizando o caractere ‘@’, seguido da string identificadora, no nosso caso, *BeginParallelize* e *EndParallelize*. O *parser* original do Java, gerado inicialmente, e que foi modificado por esse estudo, eliminava todo o tipo de comentário existente, o que já era esperado no processo de compilação. Como a utilização do *parser* será realizada mais uma vez na execução do algoritmo, desta vez com o código gerado por nosso estudo, aplicado ao *parser* original do Java, não há problemas em se manter os comentários originais do algoritmo, tendo em vistas que eles serão eliminados no processo de compilação “real”, na execução do algoritmo. Com isso, conseguimos definir mais uma contribuição do estudo, fazendo com que, em uma fase de pré-compilação, seja possível definir o que deve ser paralelizado em cada algoritmo que venha a ser transformado utilizando nossa abordagem.

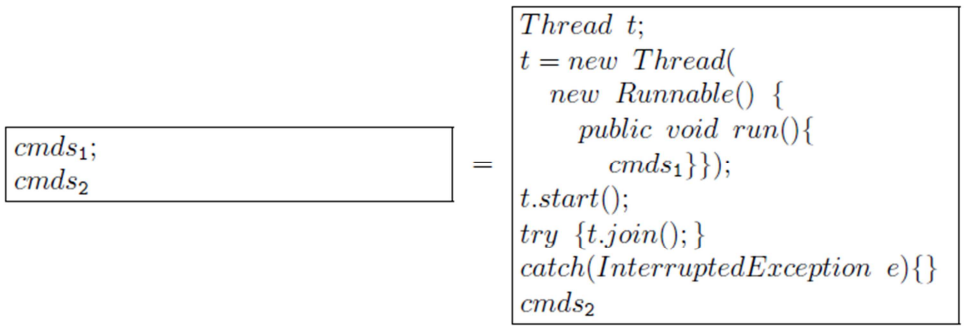


Figura 8. Lei 48 – fork - join

3.2.3 Dificuldades

Um grande problema em implementar as leis aplicando a abordagem, foi a dificuldade que se teve com a complexidade do *parser* de uma linguagem como java. Um primeiro problema, foi o de como se trabalhar individualmente com cada classe, pois o *parser* trabalha em cima de uma classe. Como os estudos de casos possuíam mais de uma classe que interagiam entre si, foi selecionada a classe que seria trabalhada, inicialmente fora do *parser*, e então foi passada para o parser, gerando um novo código. Isso significa dizer que foi necessário definir, de forma manual, cada classe que iria passar pela transformação do *parser*, mostrando que, apesar de ser uma proposta automatizada, há uma interação manual com o desenvolvedor, definindo tais classes, como ocorreu tanto no estudo de caso IDEA, com as classes *IDEATest* e IDEARunner, como para o estudo de caso de séries de Fourier, com as classes *SeriesTest* e *SeriesRunner.*

Outra dificuldade foi em manter a formalidade na construção da solução. Com a formalidade das leis algébricas, a abordagem que foi utilizada de alteração no *parser* nos proporcionou um grande grau de liberdade na implementação, porém, sem um formalismo que nos garantiria uma maior abrangência e generalização da solução.

Com relação às dificuldades na implementação, o que se verificou, e já havia sido visto em outros estudos de automatização, é que algumas modificações não são triviais de serem feitas de maneira automática, e acabam tendo que ser feitas de forma manual, como ocorreu na aplicação da lei 48.

3.3 Resultados

A partir da execução da nossa proposta automatizada, foi gerado um novo conjunto de classes, pós aplicação das leis algébricas, em conformidade com o que foi visto até então no estudo. Com a geração das classes já transformadas, foi feita a comparação de quatro abordagens distintas feitas para os estudos de caso selecionados, o algoritmo IDEA e as séries de Fourier.

Com o primeiro estudo de caso, as classes *JFGTimer*, *JGFInstrumentor* e *JGFCryptBenchSizeA* permaneceram inalteradas, pois fazem parte do contexto de testes do algoritmo, e não da implementação do algoritmo propriamente dito. Logo, a alteração dos mesmos não indicaria ganho no estudo, sendo assim, não houve a necessidade de modificá-los. Com isso, após a transformação, temos todas as classes de saída produzidos como resultado desse projeto na Figura 9, possibilitando uma análise com a Figura 15, que será vista no próximo capítulo, que contém o diagrama das classes de entrada do algoritmo IDEA.

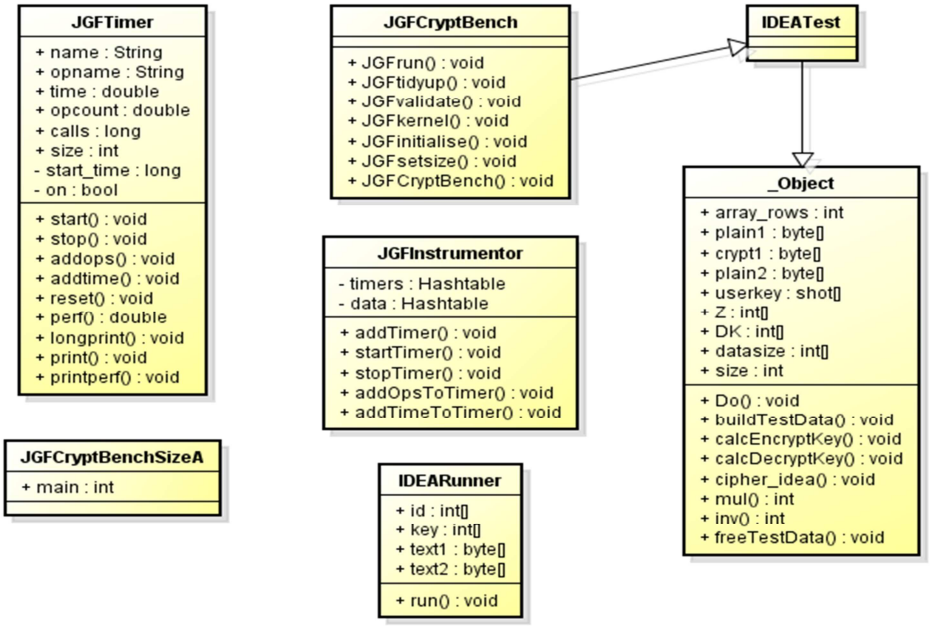


Figura 9. Diagrama das classes de saída para o algoritmo IDEA

Na Tabela 2, e nas Figuras 10 e 11, apresentaremos os resultados, com suas respectivas médias de execução, em segundos, variância, e *speedups* obtidos, além de gráficos para uma melhor visualização dos resultados. Pode se observar tanto na tabela, como nas figuras, que os resultados obtidos pela LANP AUTOMATIZADO, a

proposta deste trabalho, possui menor valor de *speedup* do que as outras formas paralelizadas, porém, com um maior com relação a forma sequencial.

Tabela 2. Dados de execução do *benchmark* IDEA

IDEA						
Forma do Algoritmo	*Execuções*	*Soma(s)*	*Média(s)*	*Variância*	*Speedup*	*Speedup(%)*
JGB SERIAL	10	2,332	0,233	1,50667E-05	1,000	0
JGB PARALELO	10	1,441	0,144	3,69889E-05	1,618	61,83%
LANP MANUAL	10	1,596	0,160	2,40444E-05	1,461	46,11%
LANP AUTOMATIZADO	10	1,764	0,176	2,27111E-05	1,322	32,19%

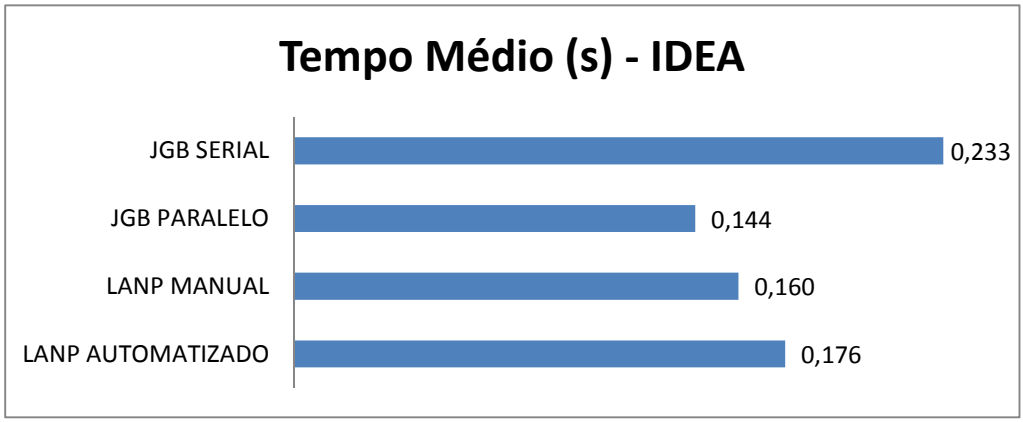


Figura 10. Tempo de execução médio nas diferentes implementações do IDEA

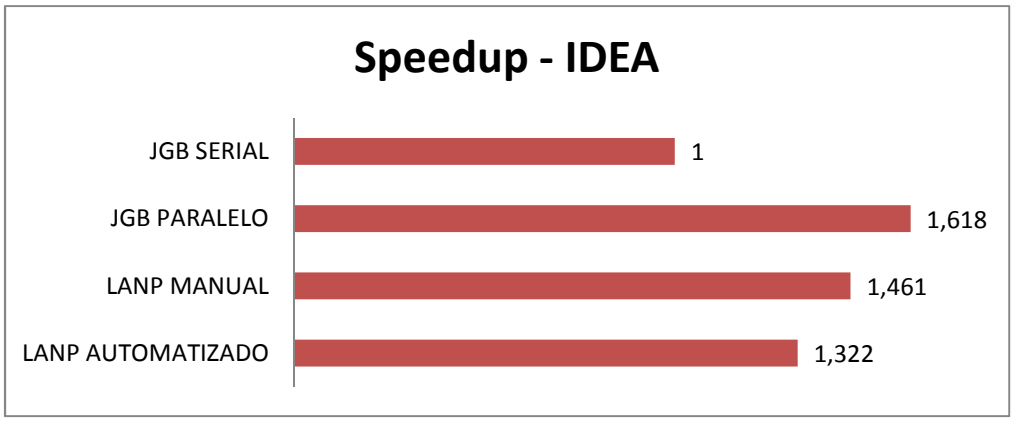


Figura 11. Speedups nas diferentes implementações do IDEA

Para o algoritmo das séries de Fourier, as classes JFGTimer, JGFInstrumentor e JGFSeriesBenchSizeA permaneceram inalteradas, também por fazerem parte do contexto de testes do algoritmo, e não da implementação do algoritmo propriamente dito. Logo, a alteração dos mesmos não indicaria ganho no

estudo, sendo assim, não houve a necessidade de modificá-los. Da mesma forma como foi feito para o primeiro estudo de caso, após a transformação, temos todas as classes de saída produzidos como resultado desse projeto na Figura 12, possibilitando uma análise com a Figura 16, que será vista no próximo capítulo, contendo o diagrama das classes de entrada das séries de Fourier.

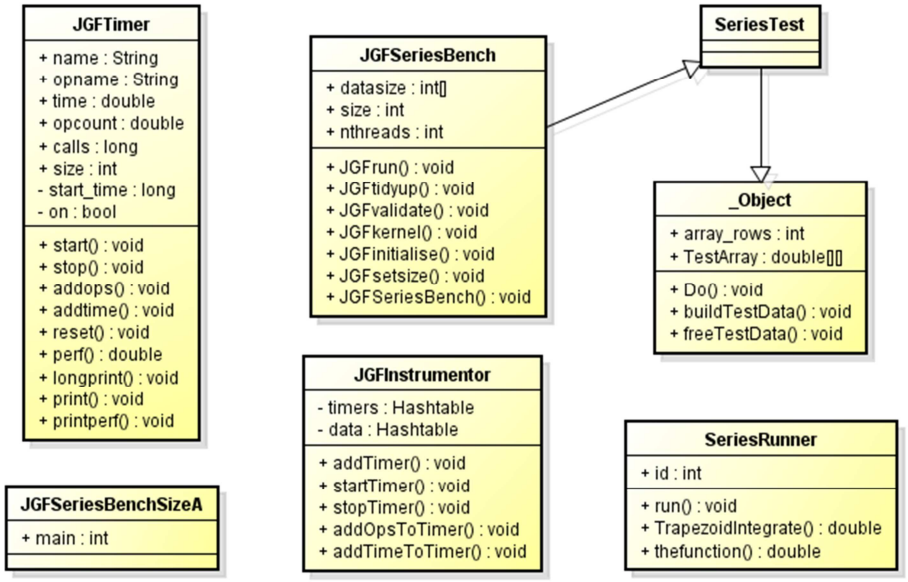


Figura 12. Diagrama das classes de saída para as séries de Fourier

Também serão apresentados os resultados, com suas respectivas médias de execução, em segundos, variância, e speedups obditos, além de gráficos para uma melhor visualização dos resultados. Da mesma maneira que o estudo de caso anterior possuiu um menor *speedup,* o estudo de caso do cálculo das séries de Fourier também obteve um pior resultado com relação as outras formas paralelizadas, porém, com um resultado mais significativo.

Tabela 3. Dados de execução do benchmark séries de Fourier

Séries de Fourier						
Forma do Algoritmo	*Execuções*	*Soma(s)*	*Média(s)*	*Variância*	*Speedup*	*Speedup(%)*
JGB SERIAL	10	162,09	16,209	3,168192	1,000	0
JGB PARALELO	10	90,748	9,075	0,367079511	1,786	78,61%
LANP MANUAL	10	95,772	9,577	0,3318944	1,692	69,24%
LANP AUTOMATIZADO	10	97,775	9,778	0,8630425	1,658	65,77%

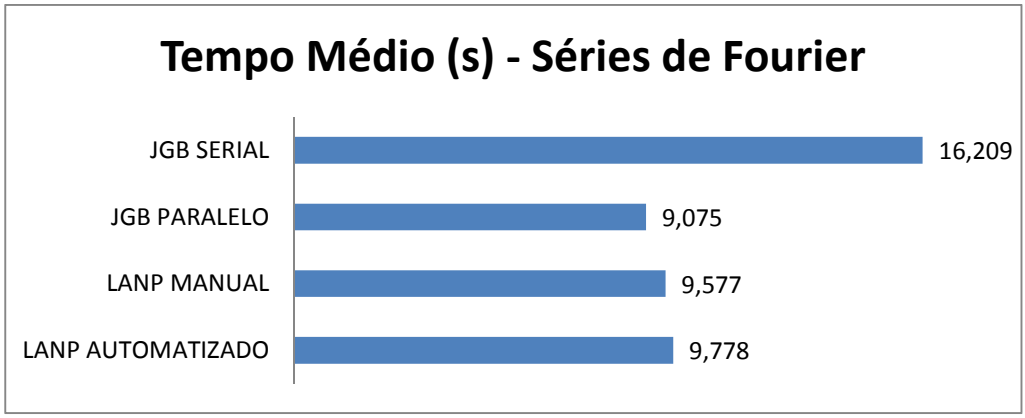


Figura 13. Tempo de execução médio nas diferentes implementações das séries de Fourier

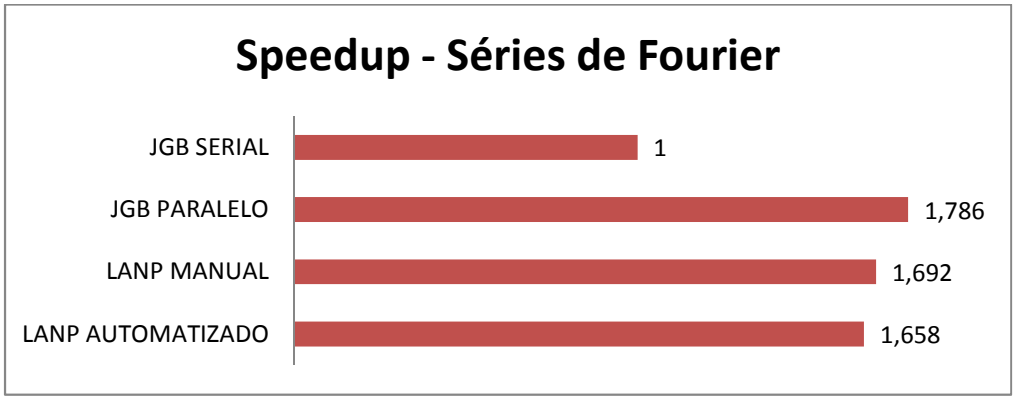


Figura 14. Speedups nas diferentes implementações das séries de Fourier

Como pôde ser observado nas figuras e tabelas, nossa solução obteve piores resultados de *speedups,* tanto com relação a proposta manual, aplicando as leis algébricas, quanto com a versão paralela do algoritmo disponibilizado pelo JGB, porém, melhores resultados que a versão sequencial. Para o primeiro estudo de caso, com o algoritmo IDEA, houve uma diferença significante de speedup da nossa proposta (32,19%), com relação a proposta manual (46,11%), entretanto, para o segundo estudo de caso, com as séries de Fourier, houve uma maior proximidade da nossa solução automatizada (65,77%), à manual (69,24%).

Outro ponto a ser ressaltado são as modificações feitas nas leis e provisos em uma proposta manual, tentando se adequar aos diversos algoritmos, o que não ocorre na abordagem automática deste estudo, podendo então, influenciar nos resultados, devido as aplicações das leis não estarem exatamente iguais.

Com esses resultados, nós acreditamos que o menor *speedup* encontrado por nossa solução é compensado pelo menor esforço que será necessário para paralelizar os algoritmos, levando em conta já a implementação que foi feita por nossa abordagem automatizada, com relação a uma proposta manual.

Capítulo 4

Estudos de Caso

Este capítulo mostra a metodologia e os estudos de caso utilizados para validar a transformação efetuada por nossa abordagem. O primeiro estudo de caso escolhido foi o algoritmo de encriptação e decriptação IDEA (International Data Encryption Algorithm) [20], e o segundo, o cálculo de coeficientes de uma série de Fourier. Os códigos foram obtidos do Java Grande Benchmark (JGB) [21], que apresenta seções de *benchmarks*, alguns deles contendo versões sequenciais e paralelas. Os resultados provenientes dos experimentos foram comparados aos obtidos pelo próprio JGB, e pela proposta de transformação manual.

4.1 Metodologia

Para realizar o estudo de caso, inicialmente obtivemos o mesmo código de entrada que foi utilizado na proposta manual [9], a fim de comparar os resultados. O algoritmo possui três diferentes tamanhos de entrada de dados para o teste de *benchmark*, mas foi escolhido o mesmo da proposta manual, o menor, também para uma melhor visualização de comparação.

Os algoritmos foram utilizados como entrada para a aplicação das leis de normalização e paralelização, que foi feita de forma automática. Como já era esperado, a aplicação dessas leis em um algoritmo de automatização se mostrou muito complicado em alguns casos, assim como foi visto na implementação manual desse algoritmo.

Para checar se a transformação está preservando o comportamento correto do algoritmo, foi utilizado o mecanismo de validação baseado em teste, do próprio JGB, que compara os resultados com um conjunto de valores, e caso eles não correspondam, uma mensagem de erro é exibida. No algoritmo executado em paralelo foi necessário escolher o número de *threads* que são executadas. Como será descrito mais à frente, a máquina que efetuará o processamento possui dois núcleos, por isso, com os conceitos vistos em [14], nós executaremos os algoritmos

em duas *threads*, o que nos dará uma visualização não tão longe de uma execução sequencial, e um bom parâmetro de comparação.

Após finalizada a transformação do código, foi feita uma comparação de acordo com o tempo de execução dentre as várias versões do código. Foram consideradas quatro fontes de código para a comparação: (i) o código original, em sua forma sequencial (JGB SERIAL), (ii) a versão paralela (JGB PARALELO), ambos fornecidos pelo JGB, (iii) a proposta paralela utilizando as leis algébricas de normalização e paralelização de forma manual (LANP MANUAL) [9], e (iv) a proposta deste trabalho, utilizando leis algébricas de normalização e paralelização de forma automatizada (LANP AUTOMATIZADO).

Para obter o tempo de execução de cada um foi utilizado o cronômetro do próprio JGB, que conta o tempo que foi despendido para a execução do algoritmo. Para uma melhor visualização, será apresentado os *speedups* de cada execução com relação ao código original, que está na forma sequencial. Para calcular o tempo de execução, cada versão do código foi executado dez vezes, e com os valores obtidos por cada versão, foi efetuada uma média aritmética que serviu de base para a comparação das versões.

Os testes foram executados em um Intel Core i3 2310m, com 8GB de RAM, rodando o Ubuntu 12.04, e o JDK 7. O ambiente foi mantido da mesma forma para a execução das quatro versões do algoritmo testado, para evitar a influência de fatores externos.

4.2 Algoritmo IDEA

O *benchmark* IDEA executa o algoritmo de encriptação e decriptação IDEA (International Data Encryption Algorithm) em um *array* de três milhões de *bytes* gerados aleatoriamente. Seguindo a estratégia de normalização utilizada neste estudo, foi introduzida uma superclasse *_Object*, fazendo com que a classe *IDEATest* herdasse dessa nova classe. Para a execução dos testes foi utilizada a classe *JGFCryptBenchSizeA*, além de outras duas classe úteis *JGFTimer* e *JGFInstrumentor.*

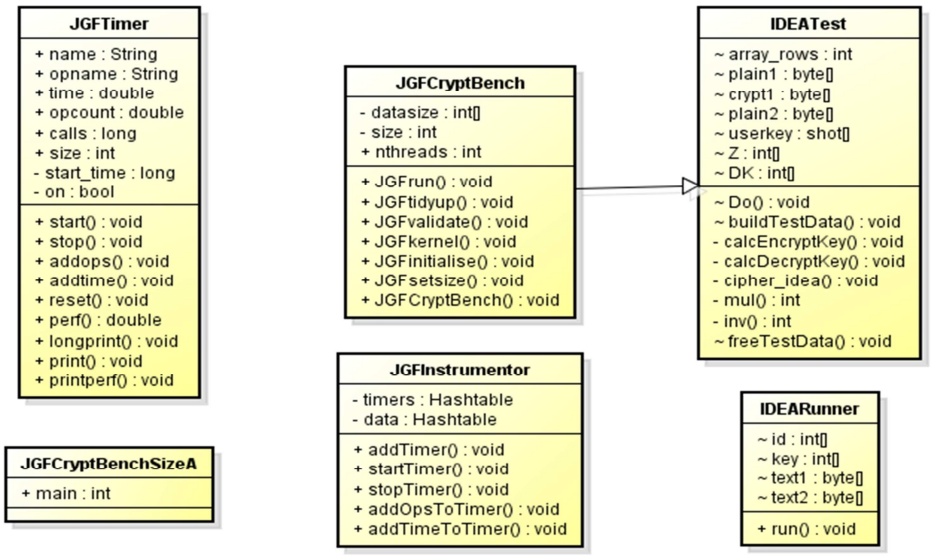


Figura 15. Diagrama das classes de entrada do *benchmark* IDEA

4.3 Séries de Fourier

No estudo de caso seguinte, foi feito com o *benchmark* séries de Fourier, que calcula os primeiros 10.000 coeficientes de Fourier, com base na função $f(x) = (x + 1)^x$, no intervalo 0-2. Analogamente ao feito no primeiro estudo de caso, foi introduzida uma superclasse _Object, fazendo com que a classe *SeriesTest* herdasse dessa nova classe. Também para a execução dos testes, foi utilizada a classe *JGFSeriesBenchSizeA*, além de outras duas classe úteis, *JGFTimer* e *JGFInstrumentor.*

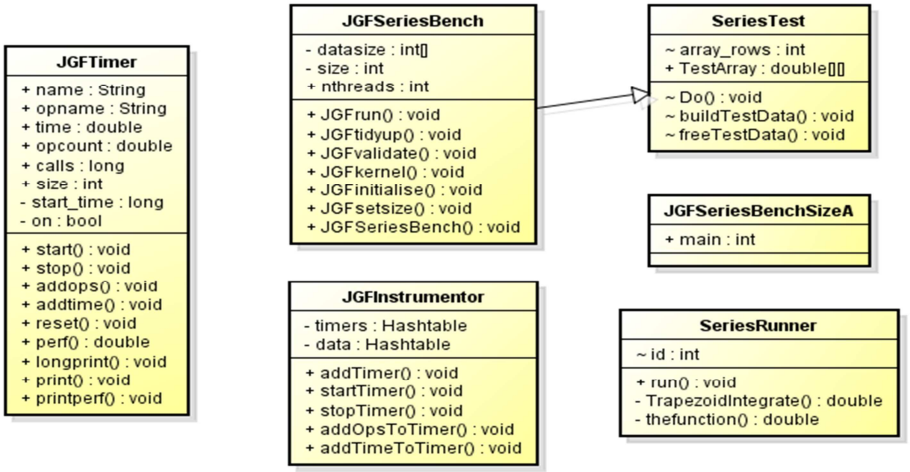


Figura 16. Diagrama das classes de entrada do benchmark séries de Fourier

Capítulo 5

Conclusão e Trabalhos Futuros

Neste último capítulo, serão apresentadas todas as contribuições realizadas a partir deste trabalho, conclusões finais sobre os resultados da abordagem proposta e, por fim, apresentar possíveis melhorias e trabalhos futuros.

5.1 Considerações Finais

Neste trabalho, abordamos como um programa sequencial em Java pode ser transformado em um paralelo automaticamente, através das leis algébricas. Em nosso trabalho foi visto que a utilização de tal transformação automatizada, com leis algébricas aplicadas em Java, se mostrou bastante complicada. A estrutura dos códigos de entrada facilitou o trabalho de transformação, assim, trabalhar com código de forma controlada, facilitou a automação da mesma, mas que de qualquer forma, vicia a implementação, de acordo com o estudo de caso.

Dentro dos estudos de caso, com o algoritmo de encriptação e decriptação IDEA, e o de cálculo das séries de Fourier, foram comparados os resultados de *speedup* e tempo de execução, as propostas implementadas manualmente, tanto as providas pelo JGB, como seguindo a mesma abordagem desse estudo, através de leis algébricas. Como foi visto, nossa proposta teve um pior desempenho com relação às duas propostas manuais apresentadas, mas teve um resultado significante com relação à proposta sequencial do algoritmo. Vale salientar, que os testes foram efetuados limitando o número de *threads* a serem executadas, e que com um maior número de *threads* possivelmente resultados melhores seriam alcançados, mas não traria a mesma visibilidade, já que a forma sequencial não teria proveito de tal número de threads.

5.2 Trabalhos Futuros

A exploração da proposta através do *parser* foi uma das muitas abordagens que poderia ter sido levada a diante para desenvolver uma solução para o estudo. Essa abordagem garante uma customização, que talvez outras não nos desse em tão pouco tempo, porém o formalismo alcançado ainda não é o desejado para o estudo.

A extensibilidade da solução é um ponto que deve ser seguido como guia para visar as pesquisas futuras, pois é algo que é almejado na implementação de uma solução automatizada, partindo de antemão com estudos de caso. Porém, que idealmente seja extensível inicialmente para algoritmos que tenham características semelhantes aos estudos de caso, e futuramente, abranger um maior número de tipos de algoritmos. Trabalhar com correções e o melhoramento do algoritmo, além desenvolver uma nova alternativa mais formal, para ser comparada com a desenvolvida nesse estudo, são possíveis pesquisas futuras.

Linguagem formal como Circus [22], fornece um melhor mecanismo para se trabalhar com transformações. Semelhante a isso foi feito com OhCircus [23], trabalhando-se com conceitos como engenharia de modelos, onde se faz uso se diversas ferramentas auxiliares, que ajudam na verificação da consistência e corretude dos modelos. Como visto na seção 4.1., uma das ferramentas, o xtext, nos dá suporte na criação de modelos, enquanto outra ferramenta chamada ATL, faz a descrição e automação das regras de refinamento. Todas essas ferramentas podem ser integradas ao *Eclipse Modeling Framework* (EMF), que é uma plataforma que auxilia o desenvolvimento de tais modelos, de aplicações envolvendo modelagem de domínio específico (MDE).

Com trabalhos já feitos utilizando essa abordagem, obtendo sucesso com tal formalismo, e com o crescimento da utilização de linguagens formais como Z e CSP, nos leva a crer que o próximo passo seria partir para uma abordagem utilizando refinamentos formais, que permitem a criação de sistemas ditos corretos por construção.

Bibliografia

[1] R. W. Hockey and C. R. Jesshope, “Parallel Computer,” 1988.

[2] R. N. Ibbett and N. P. Tophan, “Architecture of High Performance Computers II,” pp. 1-5, 83-108, 141-68, 1989.

[3] D. KIRK and W. HAW, “Programming Massively Parallel Processors, A Hands-on Approach”, 2010.

[4] H. SUTTER and J. LAURUS, “Software and the Concurrency Revolution,” p. 54–62, 2005.

[5] B. BRADEL, J. BRADEL, and S. TAREK, “Automatic trace-based parallelization of java programs,” *In ICPP ’07: Proceedings of the 2007 International Conference on Parallel Processing, p. 26, Washington, DC, USA, 2007. IEEE Computer Society.*, 2007.

[6] B. CHAN and S. TAREK, “Run-time support for the automatic parallelization of java programs,” pp. 91-117, 2004.

[7] P. FELBER, “Semi-automatic parallelization of java applications,” *In Robert Meersman, Zahir Tari, and Douglas C. Schmidt, editors, CoopIS/DOA/OD-BASE, volume 2888 of Lecture Notes in Computer Science*, pp. 1369-1383, 2003.

[8] M. K. Chen, “The Jrpm System for Dynamically Parallelizing Java Programs Kunle Olukotun,” June, p. 9-11, 2003.

[9] R. M. Duarte, “Parallelizing Java Program Using Transformation Laws,” *MSc thesis*, CIn, UFPE, 2008.

[10] G. Almasi and A. Gottlieb, “Highly Parallel Computing,” *Benjamin/Cummings Publishing Company Inc.*, 1994.

[11] R. H. PERROTT, “Parallel Languages and Parallel Programming,” *Parallel Computing 89. North-Holland: Elsevier Science Publishers B.V.*, pp. 47-58, 1990.

[12] F. C. MOKARZEL and J. PANETTA, “Reestruturação Automática de Programas Sequenciais para Processamento Paralelo,” *II Simpósio Brasileiro de Arquitetura de Computadores - Processamento Paralelo (II -SBAC-PP)*, 1988.

[13] G. Amdahl, “Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities,” *AFIPS Conference Proceedings*, pp. 483-485, 1967.

[14] J. Kwiatkowski and R. Iwaszyn, “Automatic Program Parallelization for Multicore Processors,” 2002.

[15] D. Dig, J. Marrero, and M. D. Ernst, “Refactoring sequential Java code for concurrency via concurrent libraries,” *2009 IEEE 31st International Conference on Software Engineering*, pp. 397-407, 2009.

[16] H. C. A. R. et al., “Laws of programming,” *Commun. ACM*, pp. 672-686, 1987.

[17] A. Sampaio and P. Borba, “Transformation laws for sequential object-oriented programming,” *In Lecture Notes in Computer Science : Refinement Techniques in Software Engineering*, pp. 18-63, 2006.

[18] A. Garrido and J. Meseguer, “Formal speci cation and veri cation of java refactorings,” *SCAM*, pp. 165-174, 2006.

[19] J. et al. BEZEVIN, “Bridging the MS/DSL Tools and the Eclipse Modeling Framework,” *ATLAS Goup*.

[20] B. Schneier, “The idea encryption algorithm,” *Dr. Dobb’s Journal*, pp. 50-56, 1993.

[21] L. A. Smith, J. M. Bull, and J. Obdrzálek, “A parallel java grande benchmark suite,” *In Supercomputing ’01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, 2001.

[22] A. Cavalcanti, A. Sampaio, and J. Woodcock, “A United Language of Classes and Processes,” *In St Eve: State-Oriented vs. Event-Oriented Thinking in Requirements Analysis, Formal Specification and Software Engineering, Satellite Workshop at FM’03*, 2003.

[23] P. R. G. Antonino, “Transformações Automatizadas para Herança de Processos em OhCircus,” *BSc*, 2011.