Instituto Superior de Engenharia do Porto

Suporte à Computação Paralela e Distribuída em Java: API e Comunicação entre Nós Cooperantes

Rui Miguel Brandão Rodrigues

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Arquitetura, Sistemas e Redes

Orientador: Prof. Doutor Luís Lino Ferreira
Coorientador: Eng.º Cláudio Maia

Júri:
Presidente:
Doutora Maria de Fátima Coutinho Rodrigues, ISEP
Vogais:
Doutor Jorge Manuel Neves Coelho, ISEP
Doutor Luís Miguel Moreira Lino Ferreira, ISEP
Eng.º Cláudio Roberto Ribeiro Maia, ISEP

Porto, Outubro 2012

Aos meus pais…

Resumo

Este trabalho é uma parte do tema global "Suporte à Computação Paralela e Distribuída em *Java*", também tema da tese de Daniel Barciela no mestrado de Engenharia Informática do Instituto Superior de Engenharia do Porto. O seu objetivo principal consiste na definição/criação da interface com o programador, assim como também abrange a forma como os nós comunicam e cooperam entre si para a execução de determinadas tarefas, de modo a atingirem um único objetivo global.

No âmbito desta dissertação foi realizado um estudo prévio relativamente aos modelos teóricos referentes à computação paralela, assim como também foram analisadas linguagens e *frameworks* que fornecem suporte a este mesmo tipo de computação. Este estudo teve como principal objetivo a análise da forma como estes modelos e linguagens permitem ao programador expressar o processamento paralelo no desenvolvimento das aplicações.

Como resultado desta dissertação surgiu a *framework* denominada *Distributed Parallel Framework for Java (DPF4j)*, cujo objetivo principal é fornecer aos programadores o suporte para o desenvolvimento de aplicações paralelas e distribuídas. Esta *framework* foi desenvolvida na linguagem *Java.* Esta dissertação contempla a parte referente à interface de programação e a toda a comunicação entre nós cooperantes da *framework DPF4j*.

Por fim, foi demonstrado através dos testes realizados que a *DPF4j*, apesar de ser ainda um protótipo, já demonstra ter uma performance superior a outras frameworks e linguagens que possuem os mesmos objetivos.

Abstract

The present thesis is part of the main theme "Parallel and Distributed Computing Support for Java". Its main goals are the definition and creation of an API for the framework, and the comprehension of the way nodes communicate and cooperate with each other in order to perform certain tasks to achieve a common goal.

In the scope of this thesis, a previous study was conducted about the theoretical models and frameworks that target the parallel computation domain. This study focused on the analysis of how these models and languages allow programmers to express parallelism in the development of applications.

As a result of this thesis a new framework was implemented, named Distributed Parallel Framework for Java (DPF4j), which main goal is to provide support to programmers in the development of parallel and distributed applications. The framework was developed using the Java programming language. This thesis is focused on the Application Programming Interface (API) and the communication process between all nodes that use the framework.

Finally, it was demonstrated that the DPF4j framework, although it is only a prototype, it already presents a good performance, judjing by the results obtained in the tests phase.

Agradecimentos

Para a realização desta dissertação várias pessoas contribuíram de diferentes formas e, como tal, merecem ser reconhecidas.

Em primeiro lugar, gostaria de agradecer aos meus pais pelo esforço que fizeram para eu conseguir concluir este mestrado e pelo apoio que me deram durante este tempo todo. Agradeço também à minha querida irmã, que é a melhor irmã do mundo, ao meu tio e à minha avó pelo carinho e força.

Gostaria ainda de agradecer à Tânia Rodrigues, a minha namorada, pela ajuda e pela paciência e aos seus pais pela força e por todo o apoio dado.

Não me posso esquecer de agradecer especialmente ao Daniel Barciela, que concebeu parte do trabalho comigo, pela sua capacidade e pelo seu profissionalismo.

Para além do Daniel, também quero agradecer ao Vítor Rodrigues e ao Miguel Ferreira, amigos que tive a oportunidade de conhecer e com os quais fiz projetos em equipa e passei muitos e bons momentos juntos.

Um especial agradecimento ao meu orientador, Prof. Doutor Luís Lino Ferreira, e ao meu coorientador, Eng.º Cláudio Maia por todo o tempo, paciência e empenho demonstrado.

Agradeço a todos os docentes do Mestrado em Engenharia Informática pela boa formação que me foi dada e pela disponibilidade de atendimento fornecida.

Por fim, para não estar a referir ainda mais nomes, fica então aqui um agradecimento geral a todos aqueles que me apoiaram e me deram atenção durante estes anos.

O meu muito obrigado.

Porto, Outubro de 2012

Rui Rodrigues

Índice

1 Introdução ... 1

1.1 Objetivos e Contribuições ... 2

1.2 Resumo da Solução Proposta ... 3

1.3 Estrutura da Tese ... 4

2 Modelos de Programação Paralela ... 7

2.1 Modelo Fork-Join ... 7

2.2 Modelo Divide and Conquer ... 10

3 Programação Paralela ... 13

3.1 .Net Framework 4 ... 13
3.1.1 Parallel Loops ... 14
3.1.2 Parallel Tasks ... 15

3.2 Cilk ... 16

3.3 OpenMP (Open Specifications for Multi-Processing) ... 17
3.3.1 Diretivas ... 18
3.3.2 Work-sharing ... 19

3.4 Intel® Threading Building Blocks ... 20
3.4.1 parallel_reduce ... 21
3.4.2 parallel_while ... 23

3.5 Ateji PX ... 24
3.5.1 Paralelização básica ... 24
3.5.2 Paralelização Recursiva & Paralelização Especulativa ... 24
3.5.3 Ciclos Paralelos ... 25
3.5.4 Parallel Reductions ... 26
3.5.5 Código Gerado ... 26

3.6 Java (JSR-166) ... 27
3.6.1 ExecutorService ... 28
3.6.2 ForkJoinPool ... 28
3.6.3 ForkJoinTask ... 29

3.7 Conclusões ... 29
3.7.1 .Net ... 30
3.7.2 Cilk ... 30
3.7.3 OpenMP ... 30
3.7.4 TBB ... 31
3.7.5 AteJi PX ... 31
3.7.6 Java ... 31

4 Sistemas Distribuídos ... 33

4.1 Cliente/Servidor ... 34

4.2 Peer-to-Peer ... 35

4.3 Híbrido ... 35

5 Modelos Híbridos Paralelos e Distribuídos 37
5.1 HPJava 37
5.2 Titanium 39
5.3 JavaParty 40
5.4 Conclusão 41
5.4.1 HPJava 41
5.4.2 Titanium 42
5.4.3 JavaParty 42
6 DPF4j 43
6.1 Arquitetura do Sistema (Distribuído) 45
6.2 Arquitetura da Framework 47
6.2.1 DPF Daemon 48
6.2.2 DPF User Layer 48
6.2.3 DPF Scheduler Layer 50
6.2.4 DPF Services 50
6.2.5 DPF System 51
6.3 DPF Profiler 54
6.4 Distribuição DPF4j 56
7 API 59
7.1 Ciclos 59
7.1.1 ParallelFor 61
7.1.2 ParallelForEach 64
7.2 Parallel 65
7.2.2 Execução de um ParallelFor 66
7.2.3 Execução de um ParallelForEach 69
8 Comunicação entre Nós Cooperantes 73
8.1 Segurança 73
8.2 Descoberta 73
8.2.1 Fases da Descoberta 74
8.2.2 Modos de descoberta 83
8.2.3 Configuração 83
8.3 Execução 84
8.4 Desassociação 84
9 Resultados 87
9.1 Caso 1 – Multiplicação de Matrizes 87
9.1.1 Objetivo e Explicação do Algoritmo 87
9.1.2 Dados 88
9.1.3 Ambiente de Testes 88
9.1.4 Resultados 89
9.1.5 Conclusões 94
9.2 Caso 2 – Sudoku 95

9.2.1 Objectivo e Explicação do Algoritmo 95
9.2.2 Dados 96
9.2.3 Ambiente de testes 96
9.2.4 Resultados 98
9.2.5 Conclusões 99

10 Conclusões 101

10.1 Resumo do Trabalho 101

10.2 Trabalho Futuro 102

11 Referências 105

Lista de Figuras

Figura 1 - Arquitetura da *Framework* ... 2
Figura 2 -Arquitetura Exemplo ... 3
Figura 3 - Arquitetura da *Framework* ... 4
Figura 4 - Modelo *Fork-Join* ... 7
Figura 5 - Exemplo de um ambiente com dois *cores* ... 8
Figura 6 - Exemplo de um ambiente com quatro *cores* ... 9
Figura 7 - Exemplo de um ambiente com oito *cores*. ... 10
Figura 8 - Esquema do cálculo do número de *Fibonacci* utilizando o Modelo *Divide and Conquer* ... 11
Figura 9 - *Stack* da *Framework .Net* [8] ... 13
Figura 10 - Lógica interna do *Parallel.invoke* ... 15
Figura 11 - Esquema de subprocedimentos no *Cilk* ... 17
Figura 12 - Esquema da partilha de memória entre processadores ... 18
Figura 13 - Exemplo da master thread no *OpenMP* ... 18
Figura 14 - *TBB Parallel Reduce* ... 21
Figura 15 - Exemplo de um *parallel_reduce* sobre um *blocked_range<int>(0,20,5)* ... 23
Figura 16 - Exemplo do tipo de arquitetura Cliente/Servidor ... 34
Figura 17 - Exemplo da arquitetura *Peer-to-Peer* ... 35
Figura 18 - Exemplo de um sistema distribuído híbrido ... 35
Figura 19 - Arquitetura do Sistema ... 46
Figura 21 - Diagrama de prioridades da configuração da *framework DPF4j* ... 52
Figura 22 - Resultado do *DPF Profiler* ... 55
Figura 23 - Diretório da distribuição *DPF4j* ... 57
Figura 24 - Diagrama de classes da *DPF4j API* ... 61
Figura 25 - Diagrama de fluxo do processo de criação de subtarefas na execução de um *ParallelFor* ... 67
Figura 26 - Diagrama exemplo da criação de blocos de iterações ... 69
Figura 27 - Diagrama de fluxo da execução de um *ParallelForEach* ... 70
Figura 28 - Exemplo de *Broadcast* ... 76
Figura 29 - Exemplo de *Multicast* ... 76
Figura 30 - Fluxo do tratamento de um *DiscoveryRequest* ... 78
Figura 31 - Exemplo da fase da descoberta - Procura ... 79
Figura 32 - Fluxo do tratamento de um *AssociateRequest* ... 80
Figura 33 - Exemplo da fase de descoberta - Associação ... 81
Figura 34 - Exemplo da fase de descoberta - Registo ... 82
Figura 35 - Diagrama de sequência de todo o processo de descoberta ... 82
Figura 36 - Caso de teste 1 - Gráficos de resultados referentes à máquina i7-1 ... 89
Figura 37 - Caso de teste 1 - Gráfico de resultados referente à máquina i5 ... 90
Figura 38 - Caso de teste 1 - Gráfico de resultados referente à máquina C2D-1 ... 92
Figura 39 - Caso de teste 2 - Infraestrutura ... 97
Figura 40 - Caso de teste 2 - Gráfico de resultados ... 98

Lista de Tabelas

Tabela 1 - Comparativo do código para cálculo do número de *Fibonacci* em *Cilk* sequencial e paralelo 16
Tabela 2 - *Parallel For* 21
Tabela 3 - *TBB Parallel Reduce* 22
Tabela 4 - *Splitting Constructor* 23
Tabela 5 - TBB Parallel While 24
Tabela 7 - Definição dos construtores da classe *ParallelFor* 62
Tabela 8 - Definição dos parâmetros do construtor *ParallelFor* 62
Tabela 9 - Definição dos construtores da classe *ParallelForEach* 66
Tabela 10 - Atributos do objeto *DiscoverRequest* 77
Tabela 11 - Configuração do processo de descoberta 83
Tabela 12 - Caso de teste 1 - Listagem de máquinas 88
Tabela 13 - Tabela de resultados obtidos pela máquina i7-1 para o caso de teste 1 90
Tabela 14 - Tabela de resultados obtidos pela máquina i5 para o caso de teste 1 91
Tabela 15 - Tabela de resultados obtido na máquina C2D-1 para o caso de teste 1 92
Tabela 16 - Resultados relativos à quantidade de dados transferidos 94
Tabela 17 - Registo do Problema na base de dados 96
Tabela 18 - Caso de teste 2 – Listagem de máquinas 96
Tabela 19 - Caso de teste 2 - Resultados 98
Tabela 20 - Caso de teste 2 - Resultado do processamento distribuído 99

Lista de Código

Código 1 - Exemplo de *ParallelFor* da *DPF4j* 4
Código 2 - Ciclo *for* em *.NET* 14
Código 3 - Assinatura do método *Parallel.For* em *.NET* 14
Código 4 - Exemplo de uso do *Parallel.for* em *.NET* 14
Código 5 - Execução de dois métodos sequencialmente em *.NET* 15
Código 6 - Execução de dois métodos em paralelo em *.NET* 15
Código 7 - Sintaxe de uma diretiva OpenMP 18
Código 8 - Ciclo for paralelo em OpenMP 19
Código 9 - Exemplo da utilização de *sections* no *OpenMP* 19
Código 11 - Corpo do *parallel_for* em *TBB* 20
Código 12 - Invocação do *parallel_for* em *TBB* 20
Código 13 - Exemplo de redução em *C++* 21
Código 14 - Exemplo de *parallel_reduce* em *TBB* 22
Código 15 - Invocação de *parallel_reduce* em *TBB* 22
Código 16 - Exemplo de parallel_while em TBB 23
Código 17 - Exemplo de código sequencial em *Java* 24
Código 18 - Exemplo de código paralelo em *Ateji PX* 24
Código 19 - Implementação recursiva do cálculo do número de *Fibonacci* em *Java* 25
Código 20 - Exemplo de paralelização especulativa em *Ateji PX* 25
Código 21 - Exemplo de for paralelo em *Ateji PX* 25
Código 22 - Exemplo de redução paralela em *Ateji PX* 26
Código 24 - Métodos da interface *ExecutorService* 28
Código 25 - Exemplo de multiplicação de matrizes em *HPJava* 37
Código 26 - Exemplo de multiplicação de matrizes em *HPJava* com comunicação entre processos 38
Código 27 - Exemplo da criação de um domínio retangular 40
Código 28 - For each em *Titanium* 40
Código 29 - Exemplo de remote class [17] 40
Código 30 - Exemplo de *ParallelFor* em *DPF4j* 44
Código 31 - Arrancar e desligar do DPF Daemon 48
Código 32 - Classe Conta 49
Código 33 - *For each* em *Java* 49
Código 34 - For each em *DPF4j* 49
Código 35 - Excerto da ajuda do comando *Java* 52
Código 36 - Exemplo de configuração do *workgroup* testGroup 53
Código 37 - Exemplo da execução de uma *DPFTask* 60
Código 38 - Definição da classe *DPFRunnable* 60
Código 39 - Exemplo de um ciclo for em *Java* 62
Código 40 - Exemplo de um ciclo for (*ParallelFor*) em *DPF4j* equivalente ao Código 39 62
Código 41 - Definição do objeto *MatrixContainer* 63
Código 42 - Exemplo da utilização do *DPF4j ParallelFor* 63
Código 43 - Definição do objeto Conta 64

Código 44 - Exemplo de um *foreach* em *Java* 64
Código 45 - Exemplo de um *foreach* (*ParallelForEach*) em *DPF4j* 65

Acrónimos

AES Advanced Encryption Standard

API Application Programming Interface

CDI Context and Dependency Injection

CPU Central Processing Unit

CSV Comma-separated Values

D&C Divide and Conquer

DPF Distributed Parallel Framework

DPF4J Distributed Parallel Framework for Java

HP High Performance

HPC High Performance Computing

HPF7 High Performance Fortran

IDE Integrated Development Environment

IEEE Institute of Electrical and Electronics Engineers

IP Internet Protocol

JAR Java Archive

JDK Java Development Kit

JMX Java Management Extensions

JRE Java Runtime Environment

JSR Java Specification Request

JVM Java Virtual Machine

LAN Local Area Network

MPI Message Passing Interface

RAM Random Access Memory

RMI Remote Method Invocation

SPMD Single Process Multiple Data

TBB Thread Building Blocks

TCP Transmission Control Protocol

UDP User Datagram Protocol

URL Uniform Resource Locator

VM Virtual Machine

1 Introdução

Um dos aspectos mais importantes nos sistemas computacionais é a *performance*. A *performance* de um sistema depende dos mais variados aspectos, desde o *hardware*, ao sistema operativo, qualidade e complexidade das aplicações, entre outros factores.

Durante anos a evolução da *performance* esteve centrada no aumento da velocidade de relógio dos *CPUs (Central Processing Unit)*. Assim que esta técnica começou a ser ineficiente, começaram a ser explorados os sistemas multiprocessador e *multi-core*, de forma a aumentar a *performance* através do paralelismo e ainda reduzir custos de infraestruturas e gastos energéticos. Adicionar mais *CPUs*/*cores* é uma forma eficiente de aumentar as capacidades do sistema, mas é necessário que o programador consiga retirar o máximo partido da arquitetura. Consequentemente, existe a necessidade das linguagens de programação se adaptarem a estas novas arquiteturas de uma forma transparente e, ao mesmo tempo, eficiente.

Atualmente, as linguagens de programação fornecem suporte aos programadores para desenvolverem aplicações paralelas. Estas linguagens/*frameworks* implementam paradigmas da computação para darem resposta a estas necessidades, tais como: o modelo *Fork-Join* [1] e o modelo *Divide and Conquer* [2]. Estes modelos partem do princípio que a solução para um determinado problema pode ser realizada em várias partes, e que estas partes podem ser processadas separadamente e em paralelo. Várias linguagens/*frameworks* implementaram estes modelos teóricos como é o caso do *Java*, do *OpenMP* [3], do *.Net* [4], do *Cilk* [5], do *Thread Building Blocks (TBB)* [6] e do *AteJi PX* [7].

Com a velocidade das redes atuais (*Gigabit*, *IEEE 802.11n*, etc.) é possível explorar a paralelização distribuída misturando o conceito anterior, com o conceito de sistemas distribuídos, onde um conjunto de máquinas comunicam e cooperam entre si para realizar determinadas tarefas ou ações que levam o sistema no seu global atingir os seus propósitos.

Este conceito é denominado de computação paralela distribuída e consiste em uma ou várias máquinas delegarem trabalho noutras máquinas (distribuição), também elas *multi-core* e por isso podem executar o *software* em paralelo, enquanto elas próprias continuam o seu processamento (paralelismo).

Esta dissertação propõe uma *framework* centrada no aproveitamento de técnicas desenvolvidas durante anos para facilitar a programação paralela e estudar formas de as aplicar num ambiente distribuído.

O trabalho sobre "Suporte à Computação Paralela e Distribuída em Java", foi na sua maioria feito em conjunto entre os alunos de tese no mestrado de Engenharia Informática do Instituto

Superior de Engenharia do Porto, Daniel Barciela e Rui Rodrigues. O primeiro especializou-se em “Distribuição e Execução de Trabalho”, tendo como foco os módulos *DPF Scheduler Layer* e *DPF Classloaders* da *framework*. O segundo (esta dissertação) focou-se na “*API* e Comunicação entre os Nós Cooperantes” cuja implementação incidiu sobre os módulos *DPF User Layer* ou *DPF4j API* e *DPF Services* da *framework* (Figura 1).

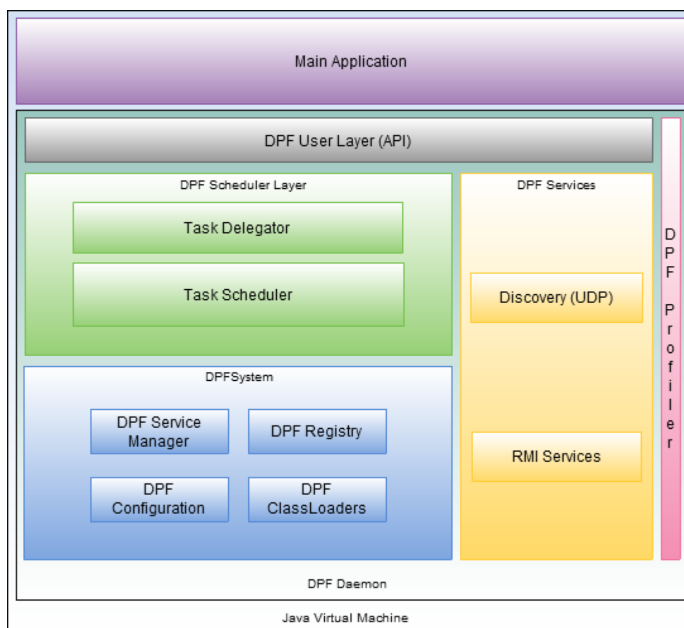

Figura 1 - Arquitetura da *Framework*

1.1 Objetivos e Contribuições

Na procura de desenvolver uma *framework* inovadora, dentro do que é a realidade tecnológica atual, foram definidos um conjunto de objetivos de forma a tornar a *framework* um produto usável e com perspectivas de trabalho futuro. Para alcançar estes objetivos foram colocadas alternativas e foram realizados estudos e testes para garantir a sua viabilidade.

O principal objetivo é criar uma *framework* que os programadores queiram utilizar, para isto como foi possível constatar não basta a *framework* ser eficaz e alcançar os objetivos de *performance* a que se propõe mas necessita de ser de fácil utilização e ter um curto período de aprendizagem. A utilização de tecnologias e conceitos conhecidos é um passo nesta direção.

Esta *framework* também pretende alcançar o maior público-alvo possível, para isso existe o requisito de ser altamente flexível e configurável mas sem que isso interfira com o objetivo anterior e a torne complexa.

Aproveitando muito trabalho realizado especialmente na área dos sistemas paralelos, esta *framework* pretende ter uma componente de abstração muito elevada e ser capaz de abstrair o programador dos detalhes de paralelismo e distribuição e aproximar estes modelos de programação o máximo possível do que é a programação sequencial.

Finalmente existe o objetivo de que a *Distributed Parallel Framework for Java* seja um projeto com futuro e de desenvolvimento continuo, para isso este terá que ser extensível e adaptável. O projeto deverá no futuro ser aberto à comunidade podendo assim ser uma mais-valia para todos, contribuindo com as suas capacidades e com o conhecimento inerente ao seu desenvolvimento e ainda beneficiando do conhecimento e experiência da comunidade para a sua evolução e aperfeiçoamento.

O desenvolvimento e concepção desta *framework* foi realizada por duas pessoas, devido ao facto dos seus requisitos serem compostos por um nível consideravelmente complexo e exigente.

A presente dissertação abrange todos os aspectos referentes à API da framework criada e a todas as comunicações realizadas entre os nós cooperantes. Tendo como principal objetivo estudar e conceber uma simples forma de o programador a utilizar, sem que este precise de se preocupar com as questões de distribuição que a própria framework fornece. Relativamente às comunicações realizadas entre todos os nós que compõem o sistema distribuído, é também realizado o estudo e a criação de mecanismos que permitam a que os nós se conheçam e consigam comunicar de forma a que seja possível distribuírem trabalho entre si.

1.2 Resumo da Solução Proposta

À *framework* desenvolvida chamou-se de *DPF4j* (*Distributed Parallel Framework for Java*) (*DPF4j*), que como o nome indica foi desenvolvida em *Java*.

A *framework* consiste numa biblioteca *Java standard* não necessitando de qualquer alteração ao ambiente de desenvolvimento ou execução além das dependências desta.

A *DPF4j* consegue de forma transparente para o programador distribuir o processamento por múltiplas máquinas, chamadas de nós, que pertencem a um mesmo grupo de trabalho, ou *workgroup* (Figura 2).

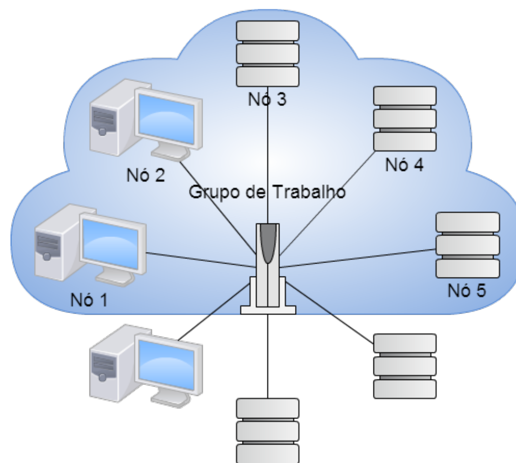

Figura 2 -Arquitetura Exemplo

Esta biblioteca disponibiliza uma *API* composta por um conjunto de métodos destinados a exprimir o paralelismo/distribuição do processamento, e por outro lado, métodos que representam operações que por norma são sequenciais mas que a *framework* consegue paralelizar e distribuir, como é o exemplo dos ciclos *for* (Código 1), *foreach,* entre outros.

```
Parallel.exec(new ParallelFor(0, 16, 1) {
public void run() {
System.out.println("Iteration number: "+ i);
}});
```

Código 1 - Exemplo de *ParallelFor* da *DPF4j*

A *framework* dispensa qualquer tipo de configuração e tem a capacidade de descobrir as outras máquinas presentes na rede (intranet e Internet) para poder distribuir trabalho. No entanto, se assim o desejar, o programador ou utilizador pode parametrizar praticamente qualquer aspecto relativo ao funcionamento da *framework* e afiná-la de acordo com as suas necessidades.

No desenvolvimento de toda a *framework* houve um cuidado de manter as várias funcionalidades devidamente separadas (Figura 3) e de desenvolver os módulos de forma extensível, com o objetivo de que possa evoluir e que possam ser adicionados módulos personalizados pelo programador, como por exemplo escalonadores, serviços ou aspectos de segurança.

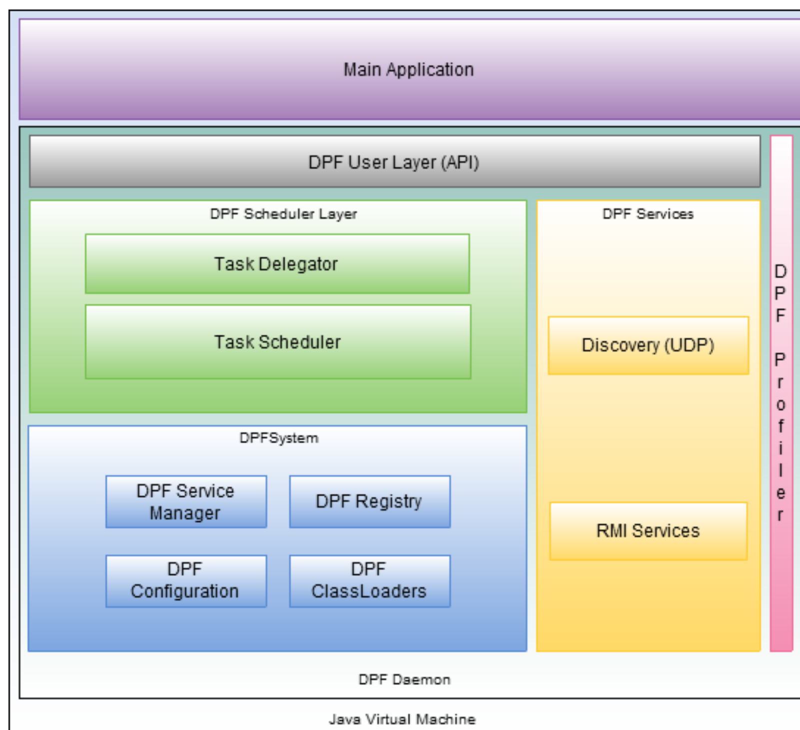

Figura 3 - Arquitetura da *Framework*

1.3 Estrutura da Tese

Esta dissertação está dividida em dez capítulos da seguinte forma:

O segundo, terceiro, quarto e quinto capítulos abordam o estudo efectuado sobre o tema para o desenvolvimento desta dissertação.

O **segundo capítulo** aborda os modelos teóricos existentes para o desenvolvimento de aplicações cujo processamento é feito de forma paralela. O **terceiro capítulo** descreve a investigação efectuada sobre as tecnologias que existem em termos de linguagens e *frameworks* que implementam os modelos abordados no capítulo anterior e a forma como estas são utilizadas. O **quarto capítulo** dá uma visão global do que são e como são utilizados os sistemas distribuídos demonstrando as suas vantagens e limitações e o **quinto capítulo** apresenta um conjunto de trabalhos já efectuados dentro da área de modelos híbridos, i.e. paralelos e distribuídos.

O **sexto capítulo** apresenta a *framework* desenvolvida de forma global, justificando as opções tecnológicas tomadas e de que forma foi possível alcançar os objetivos propostos, e demonstra ainda como a *framework* poderá ser utilizada, também é explicada a arquitetura de um sistema distribuído *DPF4j* e detalhados os vários módulos que compõem a *framework* e o respetivo funcionamento. Neste capítulo são apresentadas todas as entidades que compõem um sistema distribuído utilizando a *framework*.

No **sétimo capítulo** é feita a apresentação da *API* da *framework* apresentando ao pormenor a sua utilização e o seu comportamento. São também feitos exemplos na linguagem *Java* e é feita a comparação com os exemplos utilizando a *framework DPF4j*.

O **oitavo capítulo** descreve a forma como a *framework* lida com a questão das comunicações entres nós, desde o momento em que o nó inicia até ao momento que encerra. São apresentadas as fases do ciclo de vida do nó que utiliza o *daemon* da *framework*.

O **nono capítulo** demonstra os resultados obtidos nos testes efetuados utilizando várias linguagens ,tais como, *Java*, *AteJi PX* e *DPF4j*.

Para além dos testes realizados para comprovar os valores do tempo de execução para o mesmo fim, foram também feitos outros testes relativos à quantidade de dados enviada por rede quando é feita a distribuição de tarefas, tempos de associação dos nós, com o objetivo de justificar a viabilidade do sistema.

Por fim, com base nos resultados obtidos, as conclusões do trabalho efectuado estão apresentadas no **décimo capítulo**, onde também é sugerido trabalho futuro tendo em conta as limitações existentes e ideias de novas funcionalidades.

2 Modelos de Programação Paralela

Programação paralela é um conceito no qual a linguagem de programação, *framework*, ou ferramenta, está direcionada à programação e compilação de aplicações capazes de paralelizar o seu processamento. Para diferentes problemas, diferentes soluções de paralelismo podem ser aplicadas de forma a aumentar a eficiência da sua resolução, este capítulo aborda duas destas técnicas que se podem adaptar à programação paralela e distribuída. Neste capítulo irão ser abordados os algoritmos *Fork-Join* [1] e *Divide and Conquer* [2].

2.1 Modelo Fork-Join

Este modelo assenta na distribuição de trabalho em subtarefas, mais vulgarmente chamadas pelo termo inglês na informática como *tasks*. Uma *task* define um conjunto de trabalho a ser realizado num determinado processador/*core* ou nó quando se trata de sistemas distribuídos.

A principal vantagem deste modelo consiste na redução da carga de trabalho em trabalho fragmentado, ou seja, se o trabalho a realizar tem um tamanho considerável, então o melhor é executa-lo de forma repartida (em subtarefas). Caso o sistema computacional seja composto por mais de uma unidade central de processamento, isto permite que estes vários sub-trabalhos sejam executados em simultâneo. Tirando partido disto, o tempo de execução do trabalho pode ser reduzido a uma fracção do que demoraria se fosse executado por uma única *task*. No final da execução dos sub-trabalhos existe um momento de *join* em que os vários resultados são agregados na solução do trabalho inicial (Figura 4).

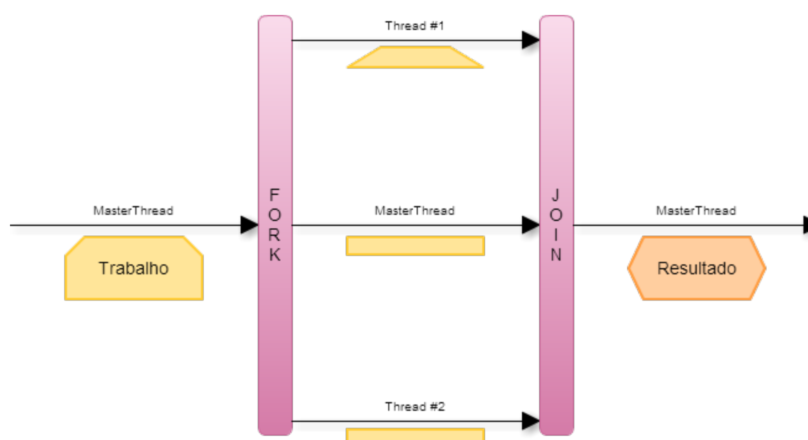

Figura 4 - Modelo *Fork-Join*

Dando o exemplo da soma de dois *arrays* de inteiros com um tamanho de 1000 posições, para realizar a soma destes *arrays* teria de se percorrer as 1000 posições e somar as duas posições dos *arrays* para cada iteração. Se o trabalho total demorasse, por exemplo, 100 segundos e fosse dividido em blocos de 200 posições demoraria aproximadamente 20 segundos a ser executado (se houvesse capacidade de executar os 5 blocos em simultâneo).

Este modelo poderá ser aplicado a este problema, dividindo o processamento por 5 *tasks*, da seguinte forma:

- *Task* 1, soma da posição [0,199];
- *Task* 2, soma da posição [200,399];
- *Task* 3, soma da posição [400,599];
- *Task* 4, soma da posição [600,799];
- *Task* 5, soma da posição [800,999];

Num ambiente com dois *cores*, teria-se algo como é apresentado na Figura 5.

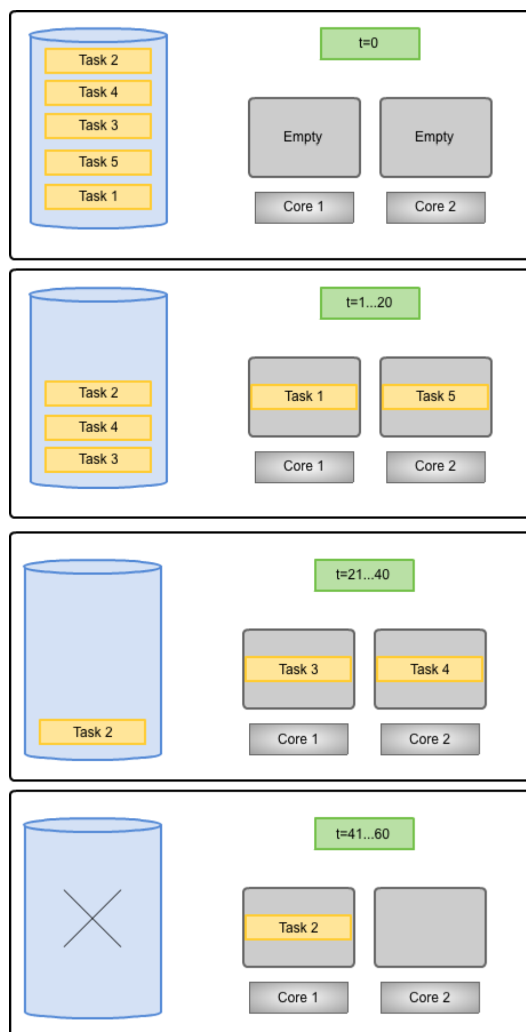

Figura 5 - Exemplo de um ambiente com dois *cores*

Nos primeiros 20 segundos seriam processadas a *Task1* e *Task5*, por exemplo. Nos próximos 20 segundos seriam processadas a *Task3* e *Task4*, e por fim, a *Task2* (A ordem de execução das tasks não importa, é matéria relacionada com escalonamento). Neste cenário o tempo de processamento total do trabalho seria de 60 segundos, conseguindo desta forma reduzir 40% do tempo gasto.

Para um ambiente com quatro *cores*, teria-se algo como é apresentado na Figura 6.

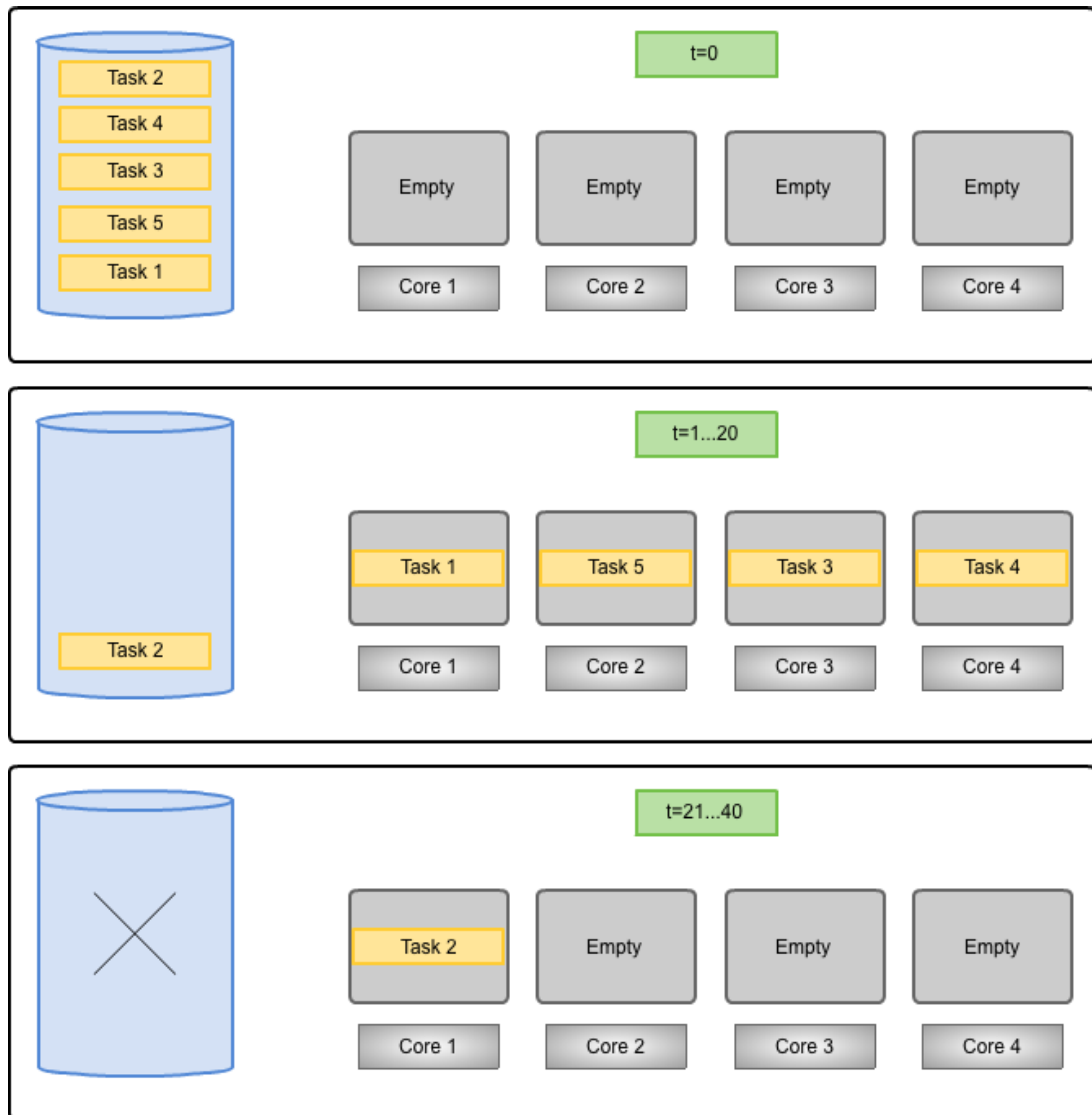

Figura 6 - Exemplo de um ambiente com quatro *cores*.

Nos primeiros 20 segundos as *tasks* 1, 5, 3, 4 seriam executadas e nos próximos 20 segundos seria executada a *Task2*, Desta forma reduz-se para 20 segundos em relação ao ambiente com 2 *cores*.

Por fim, para um ambiente com oito *cores* teria-se algo como é apresentado na Figura 7.

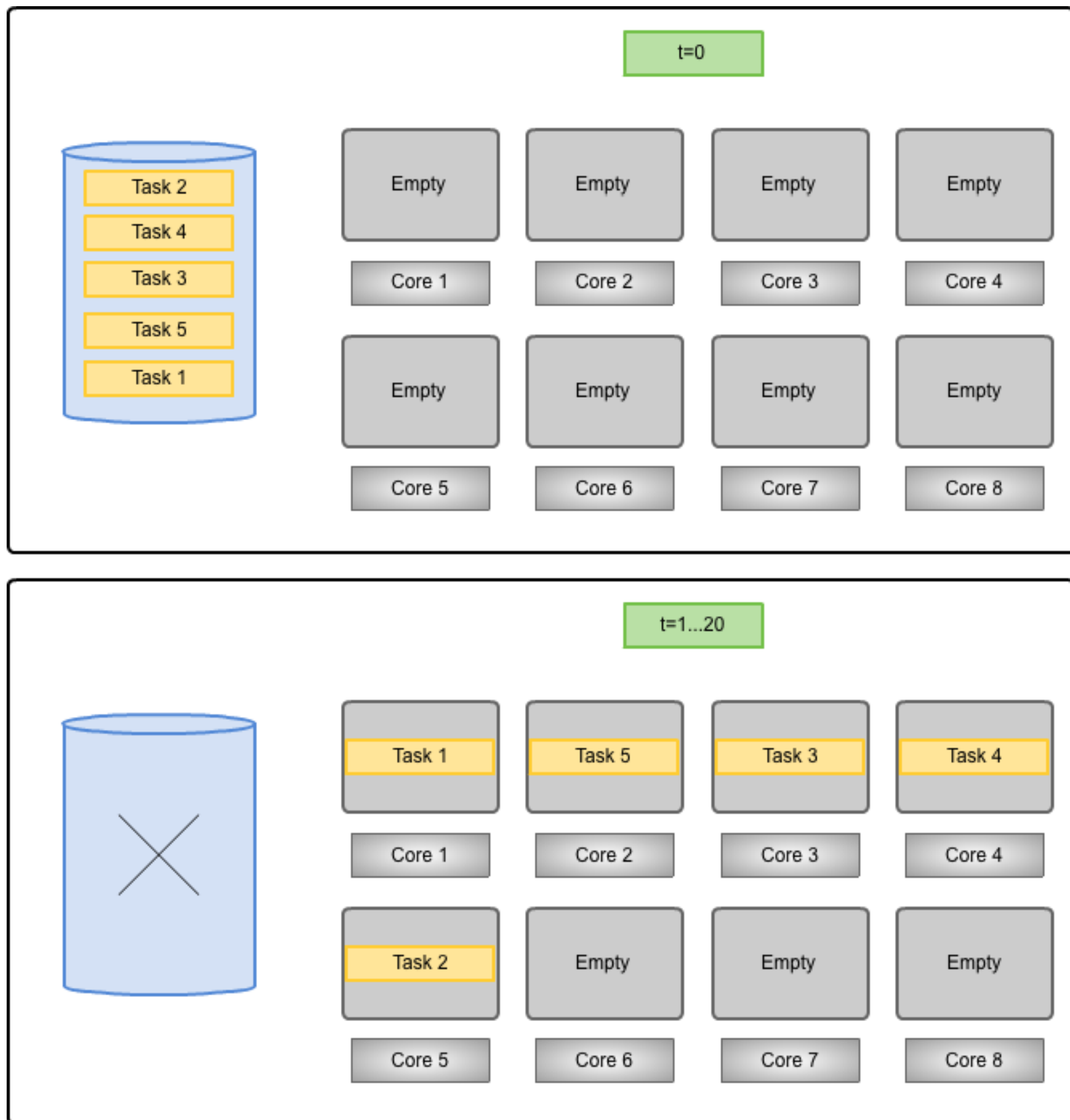

Figura 7 - Exemplo de um ambiente com oito *cores*.

Nos primeiros 20 segundos as 5 *tasks* eram executadas, ficando ainda livres 3 cores, e como resultado consegue-se reduzir o tempo do trabalho total para 20 segundos. No entanto, se existir algum mecanismo de divisão de trabalho que tenha em conta os recursos da máquina cuja execução vai ser realizada, ainda se poderá ter melhores tempos de execução. Após a conclusão de todas estas tarefas existirá o momento de *join,* no qual os seus resultados irão ser agregados.

2.2 Modelo Divide and Conquer

O modelo de *Divide and Conquer* é um modelo muito conhecido no que diz respeito à recursividade e ao paralelismo.

Este modelo baseia-se simplesmente em dividir um problema em dois ou mais sub-problemas e cada um destes sub-problemas são divididos em dois ou mais sub-problemas e assim sucessivamente até se conseguir obter a solução diretamente. Na Figura 8, encontra-se um exemplo deste modelo para o cálculo do número de *fibonacci* de 5, é assumido que o caso de paragem é quando o valor a calcular é igual a 0.

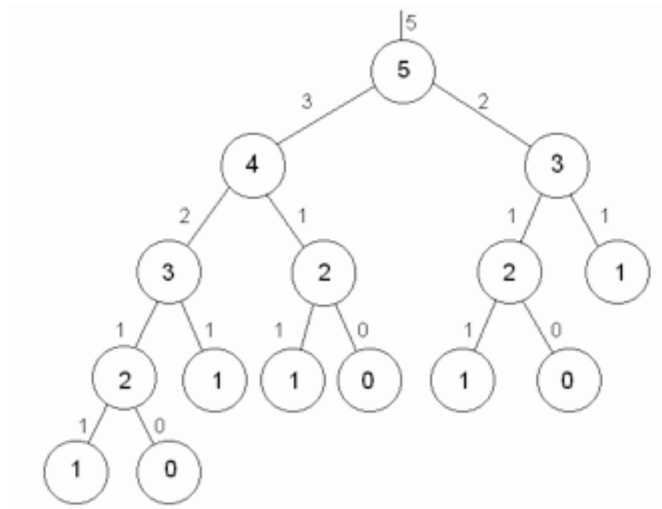

Figura 8 - Esquema do cálculo do número de *Fibonacci* utilizando o Modelo *Divide and Conquer*

O diagrama da Figura 8 facilita o entendimento deste modelo e o seu funcionamento. O modelo *divide and conquer* é eficiente e vulgarmente usado para problemas matemáticos e em algoritmos de ordenação de elementos, especialmente coleções de grandes dimensões. Exemplos vulgares da utilização deste modelo são o algoritmo do cálculo de *fibonacci* e os algoritmos *quick sort* e *merge sort* (algoritmos de ordenação).

Relativamente ao paralelismo, este modelo pode ser facilmente adoptado em sistemas de memória partilhada, se o programador tiver o cuidado de fazer a divisão dos vários problemas de forma a usarem zonas de memória diferentes pode evitar a sincronização.

O algoritmo de *divide and conquer* também tem algumas limitações, nomeadamente o número de chamadas d funções que pode originar devido à sua natureza recursiva, que por sua vez podem aumentar em muito a *stack* de uma aplicação.

3 Programação Paralela

Nesta secção são abordadas algumas das *frameworks* que implementam os modelos teóricos descritos na secção anterior, nomeadamente o *OpenMP*, o *Java 7*, o *.NET*, o *Cilk* e o *AteJi Px* que se baseiam nos modelos descritos anteriormente.

O objetivo é comparar estas *frameworks*, analisando as estruturas mais comuns, com o objetivo de se poderem vir a implementar na framework *DPF4j*.

Para cada *framework* é feita uma apresentação do seu funcionamento, focando essencialmente as suas capacidade para suportar paralelismo. No final da secção é feita uma comparação entre as *frameworks*.

3.1 .Net Framework 4

.Net é uma *framework* desenvolvida pela *Microsoft* que inclui uma grande quantidade de bibliotecas e que permite a combinação de várias linguagens de programação (interoperabilidade de linguagens), isto é, numa linguagem de programação invocar código escrito numa linguagem diferente.

A versão 4 desta *framework* adicionou uma biblioteca para programação paralela que integra conceitos como tarefas e ciclos paralelos (Figura 9).

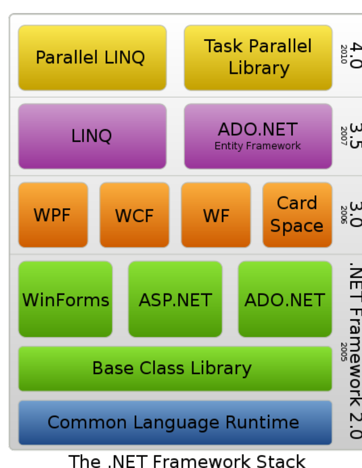

Figura 9 - *Stack* da *Framework .Net* [8]

Neste capítulo é feita a descrição da forma como a *Microsoft* adaptou a sua *framework* de modo a suportar o desenvolvimento de aplicações multitarefa.

3.1.1 Parallel Loops

Os ciclos são uma forma intuitiva de inferir paralelismo, uma vez que se trata duma prática da programação que por norma é feita de forma sequencial e numa grande parte dos casos pode ser paralelizada.

Um ciclo paralelo é muito similar a um ciclo sequencial, com as particularidades de ser paralelo e com a importante diferença da ordem de execução das iterações ser aleatória. No caso do *.NET*, a parametrização do paralelismo não necessita de ser programada (pode ser através de um objecto *Partitioner*), a própria *framework* é responsável por distribuir as várias iterações pelos processadores disponíveis, sendo que um ciclo paralelo executado num sistema uniprocessador demora aproximadamente o mesmo que um ciclo sequencial.

No *.NET* existem dois ciclos paralelos, *Parallel.For* e *Parallel.ForEach*.

A implementação destes ciclos no *.NET* é feita através de métodos que juntamente com a possibilidade de recorrer a expressões *lambda*, do ponto de vista de sintaxe, ficam muito semelhantes a um ciclo vulgar (Código 2, Código 3 e Código 4).

Em seguida é apresentado o ciclo for sequencial e paralelo.

Sequencial:

```
int n = ...
for (int i = 0; i < n; i++)
{
// ...
}
```

Código 2 - Ciclo *for* em *.NET*

Paralelo (assinatura):

```
Parallel.For(int fromInclusive,int toExclusive,Action<int> body);
```

Código 3 - Assinatura do método *Parallel.For* em *.NET*

Exemplo (exemplo):

```
int n = ...
Parallel.For(0, n, i =>
{
// ...
});
```

Código 4 - Exemplo de uso do *Parallel.for* em *.NET*

O método anterior é a forma mais vulgar de *Parallel.For*, sendo a mais semelhante com o for sequencial, mas existem vários *overloads* deste método [9] como por exemplo `Parallel.For(int fromInclusive, int toExclusive, Action<int, ParallelLoopState> body);` que é especialmente útil quando se quer utilizar operações de paragem. O *ParallelLoopState* permite executar métodos verificação e de controlo do estado do ciclo como *break* (para a *thread* e todas as *threads* de índice superior) e *stop* (para todas as

threads), é importante ter em atenção que ao utilizar um destes métodos é possível que certas *threads* de índice elevado tenham terminado antes da ordem de paragem.

Do ponto de vista de memória, as variáveis têm um comportamento também semelhante ao de um ciclo sequencial, ou seja, qualquer variável declarada fora do ciclo é partilhada pelas várias iterações, o que pode causar problemas de concorrência pois estas variáveis são partilhadas entre *threads*.

Quanto ao controlo de exceções, o comportamento dos ciclos paralelos implica parar a inicialização de novas iterações assim que a primeira exceção não controlada acontece e o método de paralelismo lança uma exceção (*AggregateException*).

3.1.2 Parallel Tasks

Uma *task*, ou tarefa, é uma série de operações sequenciais. No *.NET* qualquer método pode ser considerada uma *task, tal como é apresentado a seguir.*

Sequencial:

```
DoLeft();
DoRight();
```

Código 5 - Execução de dois métodos sequencialmente em *.NET*

No caso apresentado anteriormente, se se admitir que *DoLeft* e *DoRight* são tarefas independentes, estas podem ser executadas paralelamente, para isso basta recorrer ao método *invoke* da classe *Parallel*.

Paralelo:

```
Parallel.Invoke(DoLeft, DoRight);
```

Código 6 - Execução de dois métodos em paralelo em *.NET*

Este exemplo demonstra a forma mais simples de paralelizar tarefas. O método *Parallel.invoke* é responsável por criar as tarefas a partir dos métodos (*DoLeft* e *DoRight*, neste caso) e dar início à sua execução. De seguida é apresentado o exemplo de como o programador poderia fazer o trabalho do *Parallel.invoke* explicitamente (foram adicionados parâmetros aos métodos *doLeft* e *doRight* apenas para demonstrar que não é limitado a métodos sem parâmetros)

Paralelo:

```
Task doRight = Task.Factory.StartNew(() => doRight(dummyParameterA));
Task doLeft= Task.Factory.StartNew(() => doLeft(dummyParameterB));
Task.WaitAll(doRight, doLeft);
```

Figura 10 - Lógica interna do *Parallel.invoke*

3.2 Cilk

Cilk é uma linguagem de programação *multi-thread* desenvolvida no *MIT* (*Massachusetts Institute of Technology*) [5] e tem como principal objetivo fazer com que o programador se concentre na estruturação dos seus programas para explorar e usufruir do paralelismo, deixando o sistema encarregue de executar os seus programas de forma eficiente numa dada plataforma. O sistema de *runtime* tem em conta os seguintes aspectos:

- *Load balancing;*
- Sincronização
- Comunicação (Protocolos de comunicação).

A versão *CILK-5* fornece suporte *multi-thread* para linguagens como *ANSI C*, *Cilk++* e *C++*.

Em seguida na Tabela 1 é mostrado um exemplo de um programa em *Cilk* (*Fibonacci*),

Fibonacci Sequencial:

```
int fib (int n)
{
        if (n<2) return n;
        else{
                int x, y;
                x = fib(n-1);
                y = fib(n-2);
                return (x+y);
        }
}

int main (int argc, char *argv[])
{
        int n, result;
        n=atoi(argv[1]);
        result = fib(n);
        printf("Result:     %d\n",
result);
        return 0;
}
```

Fibonacci Paralelo:

```
cilk int fib (int n)
{
        if (n<2) return n;
        else{
                int x, y;
                x =spawn  fib(n-1);
                y =spawn fib(n-2);
                sync;
                return (x+y);
        }
}

int main (int argc, char *argv[])
{
        int n, result;
        n=atoi(argv[1]);
        result = spawn fib(n);
        sync;
        printf("Result:     %d\n",
result);
        return 0;
}
```

Tabela 1 - Comparativo do código para cálculo do número de *Fibonacci* em *Cilk* sequencial e paralelo

Do lado esquerdo é mostrado o cálculo do número de *Fibonacci* na forma sequencial e do lado direito encontra-se a aplicação do paralelismo.

Existem várias palavras-chave introduzidas pela linguagem *Cilk* relacionadas com o processamento paralelo, que são: ***cilk, spawn, sync, inlet, abort, shared, private, synched***. A palavra ***cilk*** identifica um procedimento da linguagem *Cilk*. Nesta linguagem é possível associar uma determinada ação a um determinado procedimento do tipo ***cilk*** quando este termina a sua execução. Este tipo de ação conhecida como sendo um procedimento do tipo ***inlet*** é definida como sendo parte de um procedimento e é executada automaticamente depois do procedimento associado fazer o *return*.

Cada procedimento/função pode invocar *N* subprocedimentos de forma paralela. Isto é possível utilizando a função ***Spawn***, que torna possível executar determinados procedimentos de forma paralela, como se pode verificar no exemplo apresentado acima (Tabela 1)**.** Um procedimento apenas termina quando todos os "subprocessos" terminarem, isto é a função da *keyword* ***sync***, é colocada antes da instrução *return* de cada procedimento.

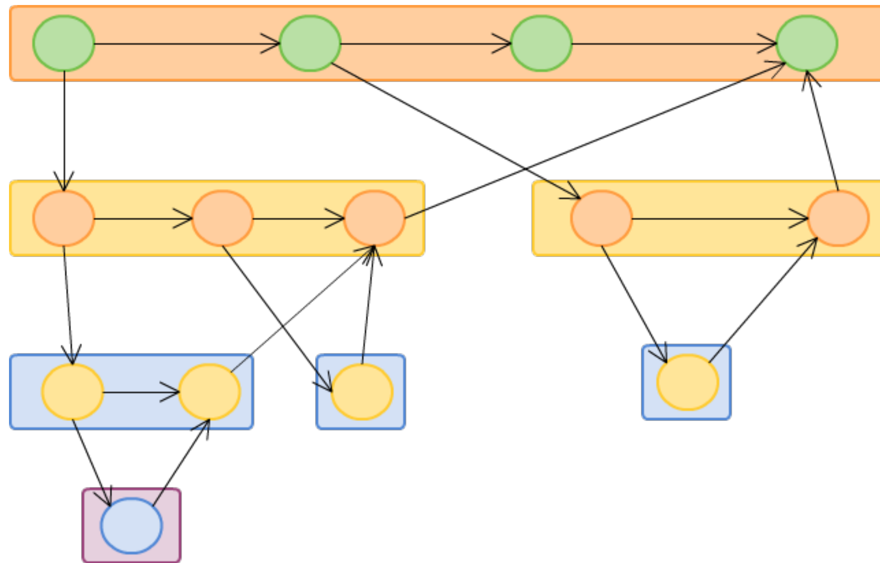

Figura 11 - Esquema de subprocedimentos no *Cilk*

Na Figura 11 é apresentado um exemplo onde é feito a criação de vários subprocedimentos a três níveis e cada processo que tem subprocedimentos aguarda pelo fim de cada um.

O próprio compilador coloca a instrução *sync* antes de todos os *returns*, caso este não exista, para garantir que um dado procedimento termina apenas depois de todos os seus procedimentos terem concluído o seu trabalho.

Esta linguagem também permite cancelar a execução de subprocedimento lançado pela função ***spawn***, sendo necessário utilizar a *keyword* ***abort***. Esta *keyword* é muito utilizada quando se utiliza execução especulativa. Relativamente aos processos/procedimentos filhos, um procedimento pai consegue obter informação acerca do processamento dos seus filhos. Para isso é necessário utilizar a keyword ***synched***.

3.3 OpenMP (Open Specifications for Multi-Processing)

É uma *API* destinada à programação paralela, onde o paralelismo é feito através de *threads* e memória partilhada. É de referir que o *OpenMP* não é uma nova linguagem, mas sim, uma especificação que se baseia na utilização de diretivas de compilação. Estas diretivas permitem ao programador desenvolver aplicações segundo o conceito de paralelismo, de forma simples, permitindo ter algum controlo sobre alguns aspectos, por exemplo definir as variáveis como partilhadas ou privadas, número de *threads* a serem utilizadas, etc. É feita uma descrição mais detalhada em seguida.

O *OpenMP* é uma implementação do modelo teórico *Fork-Join*, anteriormente descrito, e pode ser utilizado em linguagens tais como: *C\C++* e *Fortran*.

O código é executado pelos vários processadores/*cores* existentes, sendo a interação e sincronização feita através de memória partilhada (Figura 12).

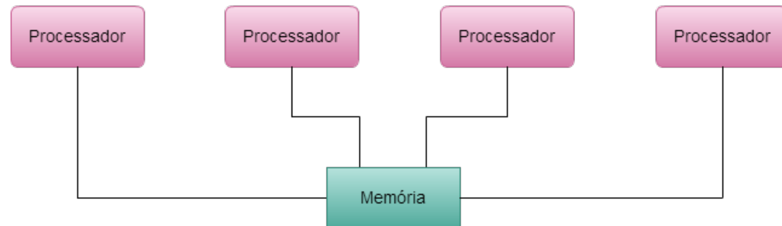

Figura 12 - Esquema da partilha de memória entre processadores

A *master* thread pode em determinadas situações criar novas threads para realizar sub-trabalhos, é aqui que se expressa o paralelismo. Na Figura 13 é apresentado esquema referente à *master thread.*

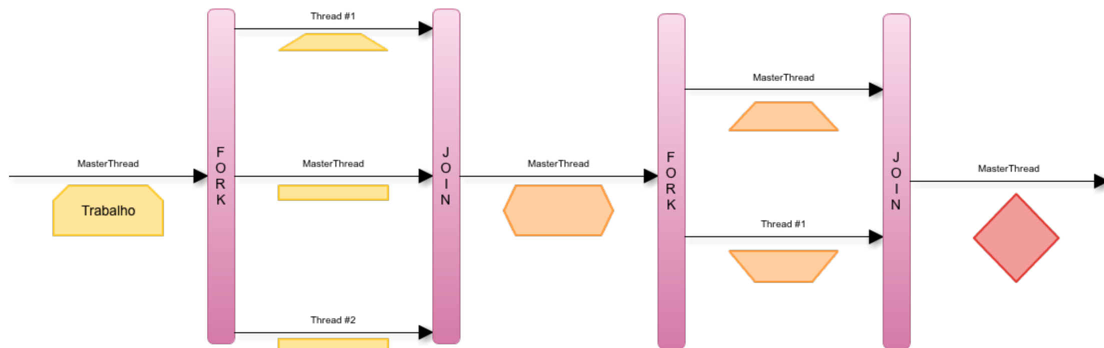

Figura 13 - Exemplo da master thread no *OpenMP*

3.3.1 Diretivas

As diretivas têm a seguinte sintaxe:

```
#pragma omp directive-name [clause, ...] newline
```

Código 7 - Sintaxe de uma diretiva OpenMP

Estas diretivas permitem aos programadores definirem a forma como desejam que o paralelismo seja realizado. A diretiva *parallel* indica que o bloco de código deve ser executado em paralelo e a diretiva *clause* é opcional e pode conter os seguintes valores:

`f(exp):` Executa se a condição *exp* for válida.

`private(var1, var2,....)`: Define as variáveis var1,var2,.. como privadas para cada thread e não são inicializadas.

`firstprivate(var1, var2,....)`: Permite que as variáveis sejam inicializadas.

`shared(var1, var2,....)`: Define as variáveis como partilhadas entre todas as threads, este é o valor por omissão.

`default(shared|none)`: Define o tipo de variável.

`copyin(var1, var2,....)`: Especifica que os dados da master thread serão copiados para variáveis privadas de cada thread.

`reduction(operator: var1, var2,....)`: Permite operar sobre as variáveis.

As instruções *private*, *shared*, *firstprivate* e *default* têm como principal objetivo permitir ao programador definir o controlo sobre as variáveis na zona paralela.

3.3.2 Work-sharing

Chama-se *work-sharing* às várias formas de definir a divisão de trabalho pelas *threads* existentes, os vários tipos de *work-sharing* implementados são os seguintes:

DO/For: partilha iterações de um ciclo por várias *threads*. Para expressar paralelismo num ciclo *for* apenas é necessário usar a seguinte diretiva antes das instruções do ciclo *for:*

```
#pragma omp parallel for [clauses] new-line
for loop
```

Código 8 - Ciclo for paralelo em OpenMP

Sections: divide trabalho por secções distintas de código, criando um momento de sincronização no final de cada uma destas secções. Exemplo de *sections*:

```
#pragma omp parallel section [clauses] new-line
        {
        [#pragma omp section new-line]
                      structured block;
        [#pragma omp section new-line]
                      structured block;
        [........]
        }
```

Código 9 - Exemplo da utilização de *sections* no *OpenMP*

Single: define que o trabalho em questão só pode ser realizado por uma *thread*.

Destes tipos de *work-sharing*, com base nos objetivos desta dissertação, o mais relevante é o *Do/For* visto que as sections não são facilmente aplicadas a outras linguagens. O *Do/For* aceita uma cláusula schedule que define como as iterações são divididas pelas threads. Esta cláusula pode definir os seguintes tipos de divisão:

Static: as iterações são atribuídas em bocados (*chunks*) de forma estática.

Dynamic: as iterações são atribuídas dinamicamente em bocados (*chunks*), quando uma thread termina recebe dinamicamente outro *chunk*.

Guided: o tamanho dos bocados vai diminuindo sucessivamente até atingir um tamanho mínimo definido

Runtime: a divisão é feita em runtime a partir da variável de ambiente *OMP_SCHEDULE*.

3.4 Intel® Threading Building Blocks

Thread Building Blocks (*TBB*) é uma biblioteca para programação paralela desenvolvida em *C++*. [6]

A biblioteca é totalmente *standard*, não dependendo de quaisquer compiladores ou linguagens alternativas. A biblioteca faz um uso intensivo de interfaces genéricas e define uma série de requisitos para os tipos com que ela lida.

Ao utilizar *TBB* deve se programar orientado a *tasks* sendo a própria *framework* totalmente responsável pela criação das *threads* e atribuição das *tasks*, ou seja, o programador nunca deve explicitamente criar *threads* para executar o seu código, deve sempre criar e submeter as suas tarefas ("*tasks*") para que o *TBB* as possa executar quando for mais oportuno, com a possibilidade de reutilizar *threads* e evitando a criação excessiva destas.

Sequencial:

```
void SerialApplyFoo(float a[], size_t n ){
        for( size_t i=0; i<n; ++i ) Foo(a[i]);
}
```

Código 10 - Exemplo de for em *C++*

Paralelo:

```
#include "tbb/blocked_range.h"
class ApplyFoo {
        float *const my_a;
public:
        void operator()( const blocked_range<size_t>& r ) const {
        float *a = my_a;
        for( size_t i=r.begin(); i!=r.end(); ++i )
                      Foo(a[i]);
        }
        ApplyFoo( float a[] ) :
        my_a(a)
        {}
};
```

Código 11 - Corpo do *parallel_for* em *TBB*

Invocação:

```
#include "tbb/parallel_for.h"
void ParallelApplyFoo( float a[], size_t n ) {
        parallel_for(blocked_range<size_t>(0,n,IdealGrainSize),
ApplyFoo(a) );
}
```

Código 12 - Invocação do *parallel_for* em *TBB*

O método *ParallelApplyFoo* (Código 12) vai fazer a invocação do *parallel_for*. O parâmetro do tipo *blocked_range* é um tipo de bloco e serve para definir o modo como o ciclo será dividido. Neste caso irá executar iterações de 0 a n-1 e irá dividir as iterações em blocos até um tamanho mínimo de *IdealGrainSize*.

A Tabela 2 apresenta os requisitos para a implementação de um *parallel_for*:

Pseudo - Assinatura	Semântica
Body:: Body (const Body&)	Construtor cópia
Body:: ~Body()	Destrutor
void Body::operator()(Range& range) const	Operação, código a ser executado pelo bloco

Tabela 2 - *Parallel For*

3.4.1 parallel_reduce

O *parallel_reduce* é mais uma forma de paralelização de ciclos, o método de funcionamento é parecido com o de um *parallel_for*, a transação é dividida recursivamente em blocos de iterações mais pequenos, mas no *parallel_reduce* no final de cada sub-transação existe um join com a transação que lhe deu origem.

Na Figura 14 é apresentado o fluxo que é feito no caso do *parallel_reduce*.

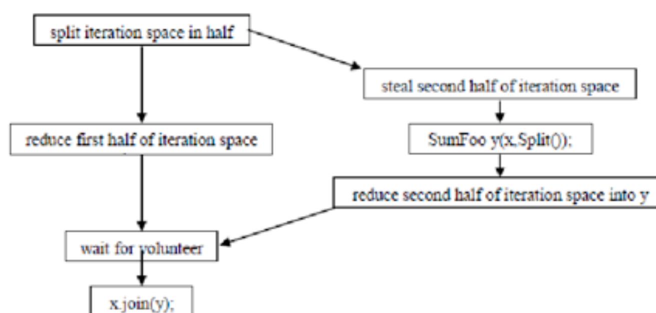

Figura 14 - *TBB Parallel Reduce*

A nível de código as principais diferenças em relação ao *parallel_for* é o facto de o *operator* não ser *const* (porque atualiza a variável *sum*) e ter um *splitting constructor*.

Sequencial:

```
float SerialSumFoo( float a[], size_t n ) {
        float sum = 0;
        for( size_t i=0; i!=n; ++i )
        sum += Foo(a[i]);
        return sum;
}
```

Código 13 - Exemplo de redução em *C++*

Paralelo:

```
class SumFoo {
float* my_a;
public:
float sum;
void operator()( const blocked_range<size_t>& r ) {
float *a = my_a;
for( size_t i=r.begin(); i!=r.end(); ++i )
  sum += Foo(a[i]);
}
SumFoo( SumFoo& x, split ) : my_a(x.my_a), sum(0) {}
void join( const SumFoo& y ) {sum+=y.sum;}
SumFoo(float a[] ) :
my_a(a), sum(0){}
};
```

Código 14 - Exemplo de *parallel_reduce* em *TBB*

Invocação:

```
float ParallelSumFoo(const float a[], size_t n ) {
        SumFoo sf(a);
parallel_reduce(blocked_range<size_t>(0,n,IdealGrainSize), sf );
        return sf.sum;
}
```

Código 15 - Invocação de *parallel_reduce* em *TBB*

Na Tabela 3 são apresentadas as assinaturas dos métodos.

Pseudo - Assinatura	Semântica
Body::Body (Body&, split)	Construtor Splitting. Deve ser possível executar concorrentemente com operator() e o método join
Body:: ~Body()	Destrutor
void Body::operator()(Range& range);	Operação, acumula os resultados do sub-bloco
void Body:: join(Body& rhs);	Junta os vários resultados

Tabela 3 - *TBB Parallel Reduce*

Um *splitting constructor* é um construtor que permite uma instância ser dividida em duas partes. A execução deste construtor deve criar um novo objeto, representante de metade do original e alterar o original para representar a outra metade. Este construtor é invocado pela *framework* para dividir objetos, chamados de *splitable*, e fazer processamento paralelo.

A Figura 15 mostra o resultado de invocações sucessivas do *splitting constructor* de um *parallel_reduce* por parte de um *blocked_range<int>(0,20,5).*

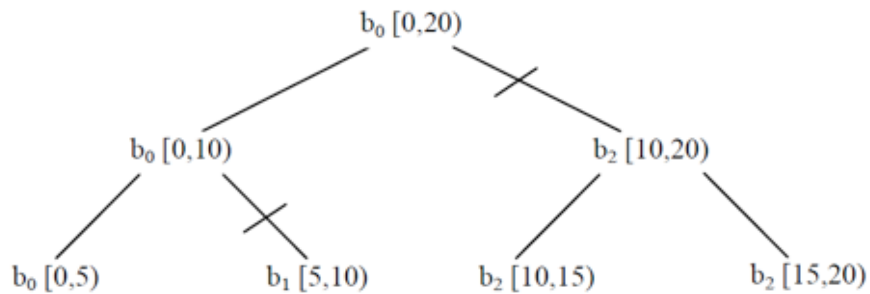

Figura 15 - Exemplo de um *parallel_reduce* sobre um *blocked_range<int>(0,20,5)*

Programaticamente o *splitting constructor* é representado por um construtor com 2 argumentos, um referência do objeto original e o outro argumento é do tipo Split e é simplesmente um *dummy* para distinguir este construtor do construtor cópia. A Tabela 4 apresenta a assinatura do *splitting constructor*.

Pseudo - Assinatura	Semântica
X::X (X& x, Split)	Parte x em x e devolve um novo objeto

Tabela 4 - *Splitting Constructor*

3.4.2 parallel_while

O *TBB* foi a única *framework* estudada que implementa o *parallel_while*. O *parallel_while* ao contrário dos anteriores não é uma função *template*, mas sim uma classe, e para a sua utilização é necessária a criação de dois objetos. Um objecto é o *stream* de items a tratar que tem de ter o método *pop_if_present(v)* (este método retorna *true* ou *false* mediante a existência de mais items e guarda em *v* o próximo item, ver primeiro bloco de código, Código 16), o segundo objeto define o método *operator* e o *argument_type* (segundo bloco de código, Código 16).

```
class ItemStream {
Item* my_ptr;
public:
bool pop_if_present( Item*& item ) {
if( my_ptr ) {
  item = my_ptr;
  my_ptr = my_ptr->next;
  return true;
} else {
 return false;
}
};
ItemStream( Item* root ) : my_ptr(root) {}
}
class ApplyFoo {
public:
void operator()( Item* item ) const {
Foo(item->data);
}
typedef Item* argument_type;
};
```

Código 16 - Exemplo de parallel_while em TBB

Na Tabela 5 é apresentada a especificação do *Parallel_while*.

Pseudo - Assinatura	Semântica
bool S::pop_if_present(B:: argument_type& item)	Obtém o próximo item
B::operator()(B::argument_type& item) const	Construtor por omissão
B::argument_type()	Processa o item. parallel_while pode invocar concorrentemente para o próprio mas com um item diferente
B::argument_type(const B:: argument_type&)	Construtor de cópia
~B::argument_type()	Destrutor

Tabela 5 - TBB Parallel While

3.5 Ateji PX

Ateji PX [7] é uma extensão à linguagem *Java*, adicionando primitivas de paralelismo. A programação é feita em ficheiros do tipo *APX*, que são posteriormente utilizados pelo *Integrated Development Environment (IDE)* para gerar ficheiros de código Java standard.

3.5.1 Paralelização básica

Para expressar paralelismo nesta linguagem as instruções a paralelizar são colocadas entre parênteses retos "[","]" e são utilizados dois *pipes* "| |" para as separar.

Sequencial (*Java*):

```
a=a+1;
b=b+1;
```

Código 17 - Exemplo de código sequencial em *Java*

Paralelo (*Ateji PX*):

```
[ a=a+1; || b=b+1;]
```

Código 18 - Exemplo de código paralelo em *Ateji PX*

3.5.2 Paralelização Recursiva & Paralelização Especulativa

Esta biblioteca fornece a possibilidade de se aplicar o paralelismo à recursividade (paralelização recursiva). Em seguida é apresentado o exemplo da função para o cálculo do número de *Fibonacci*(Código 19).

```
Int fib(int n){
        If(n<=1) return 1;
        Int fib1, fib2;
        [
        fib1 =  fib(n-1);
        fib2 = fib(n-2);
        ]
   return fib1 + fib2;
```

Código 19 - Implementação recursiva do cálculo do número de *Fibonacci* em *Java*

Neste tipo de paralelização é aplicado o modelo teórico *Divide and Conquer* explicado anteriormente, neste caso o paralelismo é feito a cada sub-problema criado.

No que diz respeito à paralelização especulativa, esta consiste na chamada de várias *tasks* em paralelo e apenas o resultado da primeira *task* é considerado, sendo os restantes descartados. Por exemplo: para a ordenação de um *array*, invocam-se dois algoritmos, neste caso utiliza-se o *quick sort* e *merge sort*. De seguida é apresentada uma imagem com o excerto de código referente à paralelização especulativa(Código 20).

Speculative Parallelism:

```
int[] sort(int[] array) {
        [
        || return mergeSort(array);
        || return quickSort(array);
        ]
}
```

Código 20 - Exemplo de paralelização especulativa em *Ateji PX*

3.5.3 Ciclos Paralelos

No que diz respeito a iterações, a biblioteca disponibiliza vários tipos, que são apresentados a seguir.

3.5.3.1 Parallel for

No que diz respeito a simples ciclos *for*, com a utilização desta *framework* é muito simples criar um ciclo *for* paralelo. Em seguida é mostrado um exemplo muito simples que consiste na soma de 1 a cada valor, de cada posição de um *array*.

```
for || (int i:N) array[i] ++;
```

Código 21 - Exemplo de for paralelo em *Ateji PX*

Como se pode verificar, apenas com uma única linha de código verificar criar um ciclo for paralelo.

3.5.4 Parallel Reductions

A *framework AteJi PX* fornece um mecanismo para a aplicação de *parallel reduction*, em que a sua utilização é tão simples quanto ao exemplo anterior.

Parallel reduction:

```
int sumOfSquares = + for || (int i:N) (i*i);
```

Código 22 - Exemplo de redução paralela em *Ateji PX*

Este exemplo demonstra como se poderia fazer a soma de todos os quadrados dos valores de 0 a N.

3.5.5 Código Gerado

A programação desta linguagem é feita com um *plugin* do *IDE*, o *plugin* possibilita a criação de ficheiros *.apx* onde é feito o desenvolvimento. Sempre que se efetua a gravação do código *IDE* gera um ficheiro equivalente em *Java*. Pode-se concluir que esta linguagem apenas traduz o código desenvolvido pelo programador em código *Java*.

De seguida (Código 23) encontra-se o código da classe gerada pelo *Ateji PX* para o exemplo citado em cima `([ a=a+1; | | b=b+1])`:

```
public static void main(String[] args) {
 int a;
 int b;
 // parallel assignment
 {
  final apx.lang.gen.MutableReferenceInt b0 = new
apx.lang.gen.MutableReferenceInt();
  final apx.lang.gen.MutableReferenceInt a0 = new
apx.lang.gen.MutableReferenceInt();
  {
   final java.util.List<apx.lang.gen.Branch> branches = new
java.util.ArrayList<apx.lang.gen.Branch>();
   final apx.lang.gen.Parallel parallelBlock = apx.lang.gen.Parallel
     .getParallelBlock();
   {
    apx.lang.gen.Branch branch = new apx.lang.gen.Branch() {
     public @java.lang.Override
     void run() throws java.lang.Throwable {
      a0.ref = 1;
     }
    };
    branches.add(branch);
   }
   {
    apx.lang.gen.Branch branch0 = new apx.lang.gen.Branch() {
     public @java.lang.Override
     void run() throws java.lang.Throwable {
      b0.ref = 2;
     }
    };
    branches.add(branch0);
   }
```

```
          final apx.lang.gen.ExitStatus exitStatus = parallelBlock
            .run(branches);
          a = a0.ref;
          b = b0.ref;
          if (exitStatus.hasReturned()) {
           return;
          }
          {
           final Throwable throwable = exitStatus.thrownException();
           if (throwable != null) {
            if (throwable instanceof RuntimeException)
             throw (RuntimeException) throwable;
            if (throwable instanceof Error)
             throw (Error) throwable;
           }
          }
        }
      }
    }
```
Código 23 - Código gerado pelo *Ateji PX*

Após uma leitura e interpretação do código, conclui-se que o que é feito é a criação da cópia da variável *a* e *b*. Em seguida são criados dois *branches* (*threads* do ponto de vista da linguagem *Java multi-thread*) que são adicionados a um objecto do tipo *ParallelBlock*. Por fim é executado o método *run* do objecto *ParallelBlock* que executa o método run de cada objecto *Branch* que lhe foi adicionado. Por fim, as variáveis a e b são atualizadas e é feita uma validação em relação a existência de *exception* durante o processamento do método *run* do objecto *ParallelBlock*.

Como se pode verificar este código é gerado segundo o conceito *multi-thread* para se obter o paralelismo desejado, no entanto, a linguagem tem a sua própria camada desenvolvida para esse fim, mas internamente utiliza a biblioteca que o *Java* disponibiliza para este fim.

3.6 Java (JSR-166)

A *release* 5 do *Java* introduziu o *JSR166* com utilitários de paralelismo (coleções *thread-safe*, fabricas de *threads*, *executores*, entre outros). Nas versões seguintes este *JSR* foi tendo várias adições, nas quais se inclui a *framework Fork/Join* incluída na versão 7 [10]. Esta *framework* consiste numa implementação da interface *ExecutorService* (introduzida no *Java* 5 em simultâneo com as primeiras partes do *JSR166*) que permite tirar partido de múltiplos processadores. Esta *framework* está optimizada para algoritmos em que o trabalho pode ser recursivamente dividido em parcelas mais pequenas.

Ao contrário da maioria dos *ExecutorService*, esta *framework* usa um algoritmo de "*Work-stealing*", ou seja, assim que um *worker* deixa de ter trabalho para fazer, rouba trabalho a um *worker* que esteja ocupado.

A *framework* é muito simples de utilizar e a sua utilização consiste, essencialmente, na utilização de uma *ForkJoinPool* e na criação de uma classe do tipo *ForkJoinTask*

3.6.1 ExecutorService

ExecutorService é uma interface do Java que serve para definir classes executoras capazes de executar tarefas de modo síncrono e assíncrono [11].

Os métodos mais importantes das classes deste tipo são os seguintes (Código 24):

```
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
<T> T invokeAny(Collection<? extends Callable<T>> tasks)
<T> Future<T> submit(Callable<T> task)
```

Código 24 - Métodos da interface *ExecutorService*

O método submit serve para submeter tarefas de forma assíncrona, este recebe tarefas do tipo *Callable* (interface que define classes com um método `call()` que pode ter qualquer tipo de retorno) e devolve um *Future*. O *Future* é um objecto que representa o resultado da tarefa, este tem um método `get()` que quando invocado bloqueia a thread até que um resultado seja obtido e nessa altura devolve o resultado da tarefa.

Os métodos *invokeAll* e *invokeAny* são métodos para invocação de *Callables* de forma síncrona, isto é, a *main thread* espera pelo resultado das tarefas. O método *invokeAll* aceita uma coleção de tarefas e devolve uma coleção de resultados. O método *invokeAny* serve para fazer paralelização especulativa, isto é, aceita uma coleção de tarefas, mas apenas devolve o resultado da primeira a terminar o seu trabalho.

3.6.2 ForkJoinPool

A *ForkJoinPool* é um *ExecutorService* e é o objecto responsável por invocar as tarefas ou ações. Neste objeto é que se encontra a lógica de *work-stealing* mencionada anteriormente. A *ForkJoinPool* por omissão cria um número de *workers* equivalente ao número de processadores presentes na máquina mas é possível alterar este comportamento através do seu construtor.

A execução de um *fork/join* por norma é iniciada através da invocação do método `ForkJoinPool.invoke(ForkJoinTask)` que dá inicio à execução da tarefa enviada por parâmetro e devolve o resultado no final do processamento (existe ainda o método *invokeAll* que recebe uma lista de tarefas e devolve uma lista de resultados). No caso de execuções assíncronas existem ainda duas alternativas, o método `execute(ForkJoinTask)` e o método `submit(ForkJoinTask)`. O primeiro existe para tarefas cujo resultado não é importante (é *void*) e o segundo devolve um objecto do tipo *Future* que serve para saber quando a tarefa está terminada e para obter o seu resultado.

Os métodos mencionados servem para iniciar a execução de uma tarefa do tipo *fork/join* por parte de um cliente não *fork/join*, isto é, quando a invocação é feita por uma *thread* que não está a executar uma *ForkJoinTask*. Quando se pretende fazer uma invocação a partir de uma *ForkJoinTask* (criação de uma nova tarefa dentro da tarefa atual para efetuar a divisão de

trabalho) devem ser utilizados os métodos da classe *ForkJoinTask* presentes na coluna da direita da Tabela 6.

	A partir de clientes não *fork/join*	A partir de computações *fork/join*
Execução assíncrona	execute(ForkJoinTask)	ForkJoinTask.fork()
Esperar e obter resultado	invoke(ForkJoinTask)	ForkJoinTask.invoke()
Executar e obter Future	submit(ForkJoinTask)	ForkJoinTask.fork()

Tabela 6 - Métodos para execução de tarefas *fork/join*

3.6.3 ForkJoinTask

ForkJoinTask é uma classe abstrata, a *framework* de *fork/join* inclui duas implementações deste tipo de tarefas: *RecursiveAction* e *RecursiveTask*.

A criação de uma classe que estende um dos dois tipos anteriores apenas obriga ao *override* do método `compute(),` onde deverá ser feito o processamento do trabalho, ou no caso deste ainda ser significativo, deverá ser feita a divisão deste em novas *tasks* a serem processadas pela *ForkJoinPool*. Para criação de novas tarefas a partir da tarefa em execução, no método `compute()`, cria-se uma nova instância do tipo de tarefa a executar, representativa duma parcela do seu trabalho, e de seguida invoca-se um dos seguintes métodos da nova *ForkJoinTask*:

`invoke()`/`invokeAll(ForkJoinTask t1, ForkJoinTask t2)`: adiciona a(s) nova(s) tarefa(s) para execução por parte da *ForkJoinPool* e só devolve o controlo quando tiver o resultado da(s) tarefa(s).

`fork()`: adiciona a tarefa para execução assíncrona por parte da *ForkJoinPool*. Este método devolve a própria instância da *ForkJoinTask* como retorno, isto acontece porque as *ForkJoinTasks* são do tipo *Future* e assim sendo, através destas é possível saber quando o trabalho está pronto (`isDone()`) e o seu resultado (`get()`). As *ForkJoinTasks* também incluem o método `join()`, este método é semelhante ao `get()`, ambos esperam que a tarefa termine e no fim devolvem o resultado da tarefa. A única diferença é o modo como lidam com exceções na tarefa executada: o método join devolve *RuntimeExceptions* e *Error* enquanto o método `get()` controla os erros lançando uma *ExecutionException*.

3.7 Conclusões

Quase todas as linguagens fornecem uma interface parametrizável no que diz respeito à forma como vai ser feita a paralelização do processamento, por exemplo numa soma de *arrays* é possível definir a granularidade, isto é, definir quantos valores cada *thread* vai ser responsável por somar. Na maioria das linguagens isto é feito na assinatura do método, ao contrário do *OpenMP* que é feito usando *pragmas*. Um recurso vulgar nos sistemas para programação de aplicações paralelas é a presença de ciclos paralelos e *APIs* para submissão

de tarefas. Algumas destas linguagens também introduzem dois conceitos importantes no que toca à facilidade de programação, sendo estes a tarefa, como representativo da unidade de código que é paralelizada, e o *worker* que abstrai o programador do conceito de *thread* e o sistema fica responsável por gerir definir se esse *worker* é uma thread isolada ou o reaproveitamento de outra thread de forma a tornar o sistema mais eficiente.

3.7.1 .Net

Esta *framework* consegue tornar a programação paralela muito idêntica à programação sequencial evitando assim tempo de adaptação ao programador.

Tem a grande vantagem de conseguir abstrair o programador dos detalhes do paralelismo e fornece a possibilidade de parametrizar a forma como o paralelismo pode ser feito.

A *framework .NET* é amplamente utilizada no mercado, e tem a vantagem de incluir as funcionalidades de paralelismo completamente integradas na distribuição da *framework* não obrigando à utilização de bibliotecas externas.

As principais desvantagens desta são estar muito agarrada a tecnologias *Microsoft* (proprietárias) e que a sua simplicidade de uso assenta em mecanismos indisponíveis nas outras linguagens (expressões *lambda*).

3.7.2 Cilk

Após a análise a esta linguagem pode-se concluir que o seu modo de utilização é simples.

O tempo de adaptação a esta linguagem por parte dum programador de linguagens baseadas em *C* deve ser curto visto que esta tecnologia não altera drasticamente a forma de programar nestas linguagens, apenas são utilizadas novas *keywords* para se desenvolver aplicações multitarefas, como *Cilk*, *Spawn* e *Sync*.

3.7.3 OpenMP

O *OpenMP* é uma *API* muito "personalizável" na forma como se expressa o paralelismo nas aplicações, devido ao uso de *pragmas*, e tem muitos pontos a seu favor. É uma API multiplataforma tal como o *Java*, não necessitando de uma *Virtual Machine* (*VM*), visto que é uma linguagem nativa. A nível de desenvolvimento é considerada uma linguagem de alto nível, eficiente e escalável. Do ponto de vista de evolução de aplicações é muito fácil de utilizar, ou seja, programar.

O *OpenMP* apesar de não ser uma linguagem, tem aspectos a ter em conta, por exemplo a forma com se pode dividir um ciclo por X iterações. Os seus conceitos podem ser de forma geral preciosos para o estudo em questão, embora o facto da forma como os *pragmas* são interpretados, neste caso em fase compilação, não é um ponto forte no que toca à possibilidade de aplicar esta *API* a outras tecnologias.

3.7.4 TBB

O *TBB* é programado em *C++* em que a principal vantagem em relação às outras é o facto de não depender de nenhum compilador ou linguagem. Tal como o *.Net*, permite definir a forma como o paralelismo é feito e de forma transparente para o programador. O método de programação é relativamente distinto da programação sequencial.

3.7.5 AteJi PX

O *Ateji PX* é uma tecnologia de muito fácil utilização, sendo apenas necessário a utilização de dois *pipes* || para se expressar paralelismo na aplicação. Existem um ponto forte a favor desta ferramenta, que é a linguagem *Java*. No entanto, o que a ferramenta faz é a transformação do código desenvolvido pelo programador para código *Java multithreading*. A principal desvantagem desta tecnologia é a criação de um ambiente de desenvolvimento dedicado devido à utilização de um pré-compilador e o facto de alterar uma linguagem, tornando o código criado incompatível com o compilador *standard*.

3.7.6 Java

A principal vantagem desta abordagem é o facto de ser totalmente *standard*, não sendo necessário recorrer a bibliotecas. Tendo em conta que a *framework* proposta será desenvolvida nesta linguagem, algumas das interfaces podem ser reutilizadas, como por exemplo os objetos de tipo *Future* para representar o resultado de uma invocação remota. O método de programação do *Fork/Join* leva a que várias *threads* sejam criadas apenas para dividir trabalho. No ambiente distribuído a divisão de trabalho pode ser uma tarefa demasiado elementar para justificar os tempos perdidos em comunicação.

4 Sistemas Distribuídos

"*A collection of independent computers that appears to its users as a single coherent system*" [A. Tanenbaum]

A distributed systems is "*one in which components located at networked computers communicate and coordinate their actions by message passing*" [G. Coulouris]

Pode-se definir um sistema distribuído como um grupo de computadores ligados por uma rede de comunicações que trabalham em conjunto e comunicam entre si através da passagem de mensagens.

Os sistemas distribuídos têm por definição um grande conjunto de vantagens e desvantagens relativamente aos sistemas centralizados e a maior parte das vantagens deste tipo de sistemas vem acompanhada de um maior nível de complexidade que as relaciona diretamente com as desvantagens.

Os sistemas distribuídos sendo compostos por várias máquinas, obrigam a que existam mecanismos de descoberta.

A cooperação entre sistemas implica a troca e duplicação de informação, a duplicação de informação é uma mais-valia em situações de falha, no entanto, para que a cooperação seja eficaz isto levanta o problema da validade da informação, se houver estado (dados em memória) nos vários nós envolvidos no sistema é preciso garantir que a informação está sincronizada e que não são executadas operações sobre dados desatualizados ou que haja sobreposição de dados processados por máquinas diferentes no caso de haver algum dispositivo de armazenamento comum.

A distribuição de processamento tira proveito da utilização de recursos remotos e por isso tende a melhorar a performance de um sistema, no entanto, este facto apenas se verifica se a carga computacional justificar a redução de performance devido às comunicações e devido ao aumento de complexidade da solução distribuída.

A duplicação de serviços também é vantajosa tornando o sistema mais tolerante a falhas, aumentando a sua disponibilidade e capacidade através de técnicas de *load balancing* e duplicação, sendo uma opção viável do ponto de vista *performance*/preço, no entanto esta abordagem adiciona muita complexidade no que toca à administração do dito sistema.

Os sistemas distribuídos têm ainda a grande vantagem de ter a capacidade de aumentar ou diminuir os seus recursos à medida das necessidades.

Do ponto de vista de desenvolvimento, os sistemas distribuídos introduzem problemas mais complexos em relação aos sistemas centralizados, nomeadamente no que se refere à segurança, à programação de operações com forte dependência temporal dado que não existe um relógio global.

Para fins desta dissertação o estudo de sistemas distribuídos focou-se em três modelos de sistemas distribuídos: cliente/servidor, *peer-to-peer, híbrido, ...*

4.1 Cliente/Servidor

O modelo cliente/servidor é um modelo de sistema distribuído que envolve uma ou mais máquinas que fornecem um recurso ou serviço, chamados de servidores, e existe um grupo de máquinas que utilizam esse mesmo serviço, chamados de clientes (Figura 16). Neste tipo de modelo as conexões são sempre iniciadas pelo cliente, e todas as conexões no sentido contrário limitam-se às respostas dos pedidos feitos pelo primeiro.

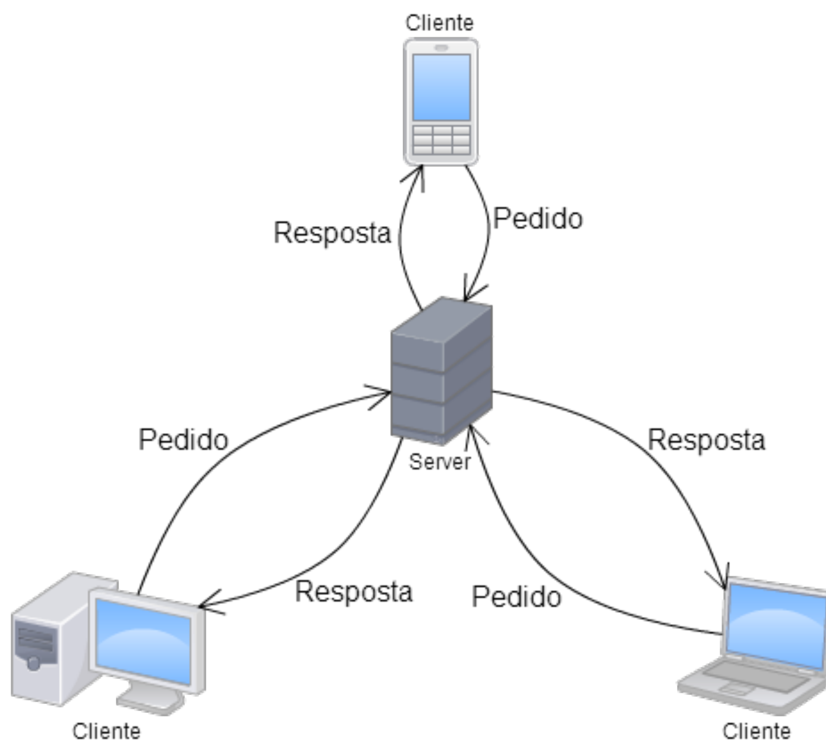

Figura 16 - Exemplo do tipo de arquitetura Cliente/Servidor

Um tipo particular de modelo cliente servidor que merece ser referenciado é o *thin client*/servidor [12] que consiste na utilização de uma máquina de baixos recursos que utiliza um servidor remoto para fazer tarefas de processamento. Neste modelo o servidor pode ser responsável por 100% das tarefas de processamento do sistema, sendo o cliente apenas a interface do sistema.

4.2 Peer-to-Peer

Ao contrário do modelo apresentado anteriormente, o modelo *peer-to-peer* envolve duas ou mais máquinas que disponibilizam recursos ou serviços e consomem recursos ou serviços de máquinas remotas, ou seja, os intervenientes são simultaneamente clientes e servidores (Figura 17).

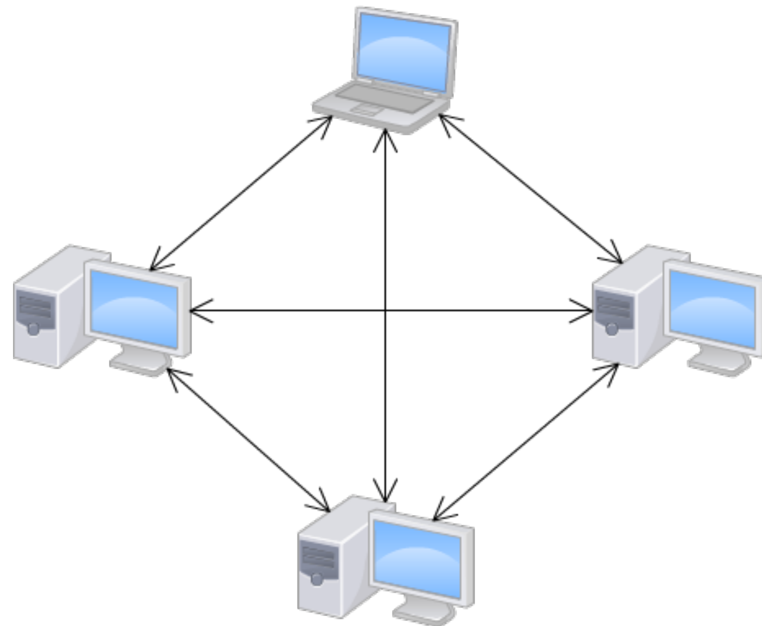

Figura 17 - Exemplo da arquitetura *Peer-to-Peer*

4.3 Híbrido

Um sistema distribuído híbrido é um sistema que tem simultaneamente relações cliente/servidor e *peer-to-peer* [12], isto é um modelo muito comum quando várias máquinas trabalham em conjunto para disponibilizar um serviço para um conjunto de clientes, como acontece no caso dos *clusters*. Na Figura 18 é apresentado um exemplo de um sistema distribuído híbrido.

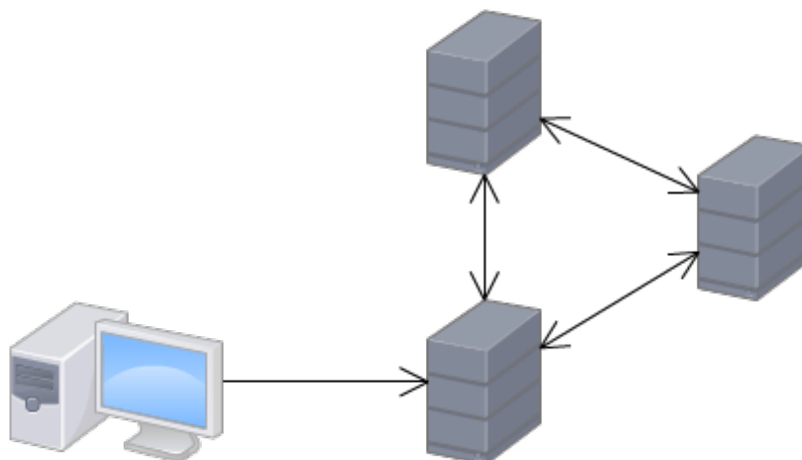

Figura 18 - Exemplo de um sistema distribuído híbrido

5 Modelos Híbridos Paralelos e Distribuídos

Esta é uma área que ainda tem muito para explorar, já existem vários trabalhos realizados dos quais será feita uma abordagem detalhada neste capítulo.

5.1 HPJava

O *HPJava* [13] é um ambiente para o desenvolvimento de aplicações *Java* paralelas com especial foco para o ramo científico. O *HPJava* baseou-se muito no modelo *HPF (High Performance Fortran*) [14] que se trata duma extensão que adicionou construtores de paralelismo ao *Fortran*. O *HPJava* chama ao seu modelo de programação *HPspmd que é uma conjunção do HPF com SPMD (Single Process, Multiple Data* - vários executores a executarem o mesmo processo sobre dados diferentes [15]).

HPJava é uma extensão ao *Java* na qual foram adicionadas novas sintaxes e algumas classes, como por exemplo classes para representação de *arrays* distribuídos. Também foram adicionados três novos construtores de controlo: *overall*, at e *on*. O *HPJava* adicionou uma nova funcionalidade à linguagem que consiste no conceito de arrays multidimensionais com propriedades semelhantes aos *arrays* utilizados na linguagem *Fortran* [16].

De seguida (Código 25) é apresentado um exemplo da utilização do *HPJava* para a multiplicação de matrizes.

```
Procs2 p= new Procs2(P,P);
on(p){
Range x= new BlockRange(N, P.dim(0));
Range y= new BlockRange(N, P.dim(1));

double [[-,-]] c = new double [[x,y]] on p;
double [[-,*]] a = new double [[x,N]] on p;
double [[*,-]] b = new double [[N,y]] on p;

overall(i=y for :)
 overall(j=y for :){
        double sum=0;
        for(int k=0; k<N ; k++){
               sum+=a[i,k]*b[k,i];
        }
        c[i,j]=sum;
 }
}
```

Código 25 - Exemplo de multiplicação de matrizes em *HPJava*

Neste exemplo começa-se por instanciar a variável *p* do tipo *Procs2* que é uma subclasse da classe base *Group* que representa um grupo de processos *HPJava* que é semelhante a um grupo *MPI*. Neste caso, o grupo de processos será igual a PxP. O bloco de código presente dentro da instrução *on(p)* apenas é executado pelo grupo de processos *p*. A classe *Range* representa um índice distribuído, existindo várias classes para este efeito com diferentes propriedades, neste exemplo é utilizada a classe *BlockRange* que consiste na definição de um bloco de índices distribuído em que o primeiro parâmetro é o tamanho total do *array* e o segundo é o tamanho a distribuir. Este exemplo não tem qualquer comunicação entre processos em *runtime*. Em seguida é apresentado um novo exemplo com o mesmo objetivo, multiplicação de matrizes, mas sendo um método parametrizável em que existe comunicação entre os processos do grupo.

```
void matmul(double [[-,-]] c, double [[-,-]] a,double [[-,-]] b){
 Group p=c.grp();

 Range x=c.rng(0);
 Range y=c.rng(1);

 int N = a.rng(1).size();

 double [[-,*]] ta = new double [[x,N]] on p;
 double [[-,*]] tb = new double [[N,y]] on p;

 Ablib.remap(ta,a);
 Adlib.remap(tb,b);

 on(p){
 overall(i = x for :)
       overall(j = y for :){
             double sum=0;
             for(int k=0; k<N ; k++)
                   sum+= ta[i,k] * tb[k,j];
c[i,j]=sum;
}
}
}
```

Código 26 - Exemplo de multiplicação de matrizes em *HPJava* com comunicação entre processos

Neste exemplo (Código 26) é feita um cópia do bloco a distribuir utilizando a função `remap()` da biblioteca *Adlib*. Este exemplo também apresenta a forma como se pode manipular *arrays* com o *HPJava* no formato distribuído.

Nos exemplos acima é utilizada uma biblioteca chamada *Adlib*, que é a biblioteca que o *HPJava* utiliza para efetuar comunicação. Esta biblioteca, de alto nível, foi desenvolvida sobre a biblioteca *mpjdev* (baixo nível) com o principal objetivo que de existir portabilidade a nível de rede e eficiência na paralelização de *hardware*.

As versões anteriores da biblioteca de comunicação do *HPJava* fornecem um conjunto de operações para interagir/manipular os arrays distribuídos. No entanto a nova versão suporta a utilização de dados primitivos do Java assim como objecto também.

5.2 Titanium

Titanium é um sistema concebido para o desenvolvimento de aplicações científicas de alta *performance*. Um dos objetivos desta tecnologia é fornecer aos seus utilizadores uma nova forma de utilizar o modelo de programação orientado a objetos e expressar de forma explícita o desenvolvimento de aplicações paralelas.

A linguagem *Titanium* é baseada em *Java* e tem como principais objetivos a *performance*, a segurança e a legibilidade, por esta ordem de prioridades. O processo de compilação desta linguagem consiste na geração de código *C* a partir do código *Titanium*.

Comparativamente ao *Java*, o *Titanium* introduz alguns novos conceitos, sendo eles:

Classes imutáveis: Este é um tipo mais limitado de classe, que não pode estender ou ser estendida e todos os seus atributos não estáticos são obrigatoriamente "*final*". Isto permite ao compilador passar objetos deste tipo de classes por valor e não por referência como é normal no *Java*. As passagens por referência originam um nível maior de complexidade no controlo de objetos, e dificulta especialmente a destruição destes, prejudicando assim a performance global da aplicação.

Sincronização global: A linguagem introduz o conceito de barreiras, uma barreira é uma instrução que faz com que um processo espere até que todos os outros atinjam a mesma barreira. Além da introdução do conceito, o compilador de *Titanium* garante ainda que o programador não se engana e faz código que levaria vários processos a atingirem as barreiras por ordem diferente (e assim originar *dead-locks*).

Referências locais e globais: A memória associada a um processo *Titanium* é chamada de região. Por omissão, as variáveis em *Titanium* são globais e um processo pode ter uma referência para uma variável de uma região que não a sua, mas através do modificador "*local*" o programador pode dizer que uma variável só está disponível dentro da sua região e assim beneficiar de um uso mais eficiente dessa variável (melhor *performance* e menos memória utilizada).

Comunicação: A comunicação entre processos é feita via *reads* e *writes* de atributos e cópia de objetos ou *arrays*. Existem dois tipos de comunicação: *Broadcast*, ou seja, de um processo para todos; *Exchange*, de todos para todos.

Estes dois modos de comunicação implicam a existência de um mecanismo global de sincronização, o que faz com que cada processo no final de executar uma determinada operação tenha de invocar uma das operações acima indicadas. A sincronização processo-a-processo é controlada como no Java utilizando métodos sincronizados "*syncronized*" ou com blocos de código protegidos contra *race-conditions* utilizando o "*syncronized*".

Modelo de consistência: Está em estudo a aplicação de modelos de consistência no *Titanium*, mas devido à complexidade de implementação, neste momento, a responsabilidade de garantir race-conditions está totalmente do lado do programador.

Gestão de memória: A memória é dividida por zonas e o programador tem a capacidade de definir em que zona uma variável deve ser guardada e limpar zonas inteiras de memória através da operação *delete-zone*. O sistema garante no entanto que não são eliminadas zonas nas quais existem variáveis referenciadas.

Arrays, pontos e domínios: Os *arrays* em *Titanium* são muito diferentes de *Java*. Os *arrays* são multidimensionais e englobam domínios. Por sua vez, os domínios são conjuntos de pontos que por sua vez são tuplos de inteiros.

```
Point<2> l = [1, 1];
Point<2> u = [10, 20];
RectDomain<2> r = [l : u];
double [2d] A = new double[r];
```

Código 27 - Exemplo da criação de um domínio retangular

No Código 27 é criado um domínio rectangular (o número de inteiros por tuplo é igual) e depois é criado um array de 2 dimensões (2d) com esse domínio. *Titanium* não tem operadores sobre arrays devido à dificuldade de optimizar esses tipo de operações mas fornece uma forma simples de os percorrer recorrendo ao *foreach* (Código 28).

```
foreach (p in A.domain()) {
A[p] = 42;
}
```

Código 28 - For each em *Titanium*

Na criação do sistema *Titanium* houve uma preocupação de o tornar o máximo compatível com a linguagem *Java*. No entanto, este é incompatível com *threads* devido a dar esta liberdade ao programador complicar a sincronização global do sistema.

5.3 JavaParty

JavaParty é uma extensão da linguagem *Java* criando uma vertente para ambientes distribuídos, neste caso *clusters*. O sistema *JavaParty* funciona a partir de um pré-compilador que gera código *Java standard* e tem como principal característica a adição de *remote classes*, tal como é apresentado no exemplo seguinte (Código 29).

```
public remote class HelloJP {
    public void hello() {
        System.out.println("Hello JavaParty!");
    }
    public static void main(String[] args) {
        for (int n = 0; n < 10; n++) {
            HelloJP world = new HelloJP();
            world.hello();
        }
    }
}
```

Código 29 - Exemplo de remote class [17]

Este modificador torna possível o acesso às instâncias deste tipo a partir de outras máquinas no *cluster*.

A sintaxe desta solução é puramente à la *Java* e o modificador *remote* é a única extensão feita à linguagem, criando desta forma o conceito de objetos remotos. Assim com a utilização destes objetos remotos é criado o conceito de memória distribuída em redes heterogéneas.

O *JavaParty* tem como requisito mínimo a versão 1.4 do *JDK* (*Java Development Kit).*

Uma aplicação *Java* pode ser transformada facilmente numa aplicação *JavaParty*, com o objetivo de se tornar numa aplicação cujo processamento seja distribuído. Para tal, é apenas necessário identificar os objetos/classes e *threads* que poderão fazer parte da distribuição, adicionando-lhes a palavra *remote* na definição da classe. O *JavaParty* fornece um espaço de memória partilhado, ou seja, os objetos remotos podem ser acedidos mesmo que se encontrem em diferentes máquinas, isto é feito de forma transparente para o utilizador.

No que diz respeito a comunicações, todos os mecanismos referentes a este aspecto são transparentes ao utilizador. Quando se trata de protocolos de comunicação não explícitos, cabe ao utilizador efetuar o respectivo tratamento.

Sendo transparente à localização, o programador não necessita mapear os objetos/*threads* aos nós da rede, o compilador e o sistema de runtime fica encarregue de lidar com estas questões de localização e comunicação. Para o conseguir, são utilizadas estratégias de distribuição no momento em que são criadas novas instâncias, podendo estas ser alteradas em runtime. Uma instância de uma *remote class* está sempre apenas num nó e os restantes têm apenas um *proxy* para essa instância, a distribuição de processamento funciona assim com a criação de objetos remotos (ou migração de objetos locais) e invocação dos métodos dos mesmos. A utilização deste sistema implica a existência de um nó mestre que conhece todos os nós do *cluster* e sabe onde se localizam todas as instâncias de classes remotas. Este nó mestre consiste numa aplicação chamada "*Java Party Runtime Manager*" e as várias máquinas que compõem o *cluster* encontram este nó através da utilização de mensagens multicast ou então manualmente através da adição dos argumentos -*host* e -*port* ao comando de arranque das aplicações *JavaParty*:

```
jp -host servidor1.local -port 1099  <classe a executar>
```

5.4 Conclusão

Como se pode verificar já existe um trabalho notável no que diz respeito à computação distribuída. No entanto, existem pontos fortes e pontos fracos relativamente a estas três tecnologias analisadas. Estas três tecnologias, todas elas baseadas na linguagem *Java*, implicam a utilização de um novo compilador ou uma nova fase de pré compilação.

5.4.1 HPJava

HPJava é uma extensão à linguagem *Java*, sendo-lhe adicionadas novas instruções e novas classes, como por exemplo grupos de processos. Esta tem um forte vertente para o controlo de acesso relativamente a grupos de processos e utiliza *MPI* para comunicação entre processos. A principal desvantagem do *HPJava* são as muitas diferenças da linguagem *Java* e implica um tempo de aprendizagem significativo para se ser capaz de utilizar esta tecnologia.

Como vantagem aponta-se as capacidades avançadas de controlo da distribuição de trabalho e comunicações mas estas trazem uma quantidade excessiva de complexidade. Não se enquadrando com os objetivos que estão traçados para a *DPF4j*.

5.4.2 Titanium

Em relação à tecnologia *Titanium*, apesar de ser baseada na linguagem *Java*, esta consiste num sistema totalmente novo sendo depois convertida para *C* e compilada. Apesar de isto trazer ganhos de *performance* ao sistema, esta abordagem torna as aplicações *Titanium* incompatíveis com as aplicações puramente Java que estão a executar dentro de uma *JVM*. À semelhança da *HPJava* o *Titanium* introduz muitos conceitos novos e também implica uma fase de aprendizagem superior ao que seria desejável, criando resistência à sua adesão. O facto de ter um compilador próprio implica a preparação de um ambiente de desenvolvimento dificultando a experimentação da tecnologia. Como vantagens a tecnologia demonstra muitas preocupações a nível de *performance* e uma gestão de memória inteligente.

5.4.3 JavaParty

JavaParty tem o conceito de objetos remotos alterando a linguagem *Java* de base. Uma desvantagem deste facto é implicar alterações ao ambiente de desenvolvimento do programador visto que o *JDK* instalado não é suficiente. Outro ponto fraco relativamente a esta tecnologia é o facto ter que existir obrigatoriamente nó mestre que basicamente conhece todos os outros nós que pertencem ao *cluster*. Os principais pontos fortes desta linguagem são a facilidade de migração e aprendizagem visto as alterações à linguagem serem mínimas. Outra grande vantagem é a capacidade de abstrair o utilizador das questões de ligações entre máquinas e ter um sistema de descoberta automático.

6 DPF4j

DPF4j é o acrónimo de *Distributed Parallel Framework for Java*. Tal como o próprio nome indica é uma *framework* cujo principal objetivo é facilitar o desenvolvimento de aplicações paralelas e distribuídas, isto é, aplicações capazes de paralelizar o seu processamento não só localmente (entre processadores ou cores) mas também de forma distribuída, delegando processamento para um grupo de máquinas com a *framework DPF4j*.

As grandes linguagens candidatas à implementação desta *framework* eram o *Java* e o *.NET*. Ambas as linguagens têm uma comunidade ampla, são linguagens de fácil aprendizagem que fazem parte do plano curricular da maior parte dos estudantes de informática. A linguagem *.NET* pode trazer algumas facilidades no que toca à definição da *API* graças à utilização de expressões lambda (*lambda expressions*), no entanto também é previsto que o *Java* beneficie desta capacidade na versão 8 [18]. As duas linguagens são orientadas a objetos o que possibilitará à *framework* ser mais flexível no que toca a extensões e modificações. A natureza *open-source* da comunidade *Java* também é uma mais-valia, pois existe a intenção de que o projeto tome esta filosofia no futuro para conseguir aumentar a sua funcionalidade e aperfeiçoar-se em vários aspectos. A linguagem *Java* também proporciona algumas facilidades no que toca a implementação da *framework* devido à grande quantidade de bibliotecas incluídas na linguagem e a possibilidade de carregar código dinamicamente em tempo de execução. No entanto, a principal razão de se optar pela linguagem *Java* dá-se pelo facto desta ser multiplataforma podendo assim a aplicação distribuir processamento por várias máquinas correndo diferentes sistemas operativos e sendo possível no futuro migrar a *framework* para outras arquiteturas como por exemplo para sistemas móveis.

A simplicidade de uso e a facilidade de aprendizagem são dois dos pontos fortes desta *framework* relativamente às *frameworks* já existentes, mas tentando manter todas as capacidades de configuração e funcionalidades avançadas destas.

No que diz respeito à distribuição, como representação de rede a *DPF4j* identifica cada máquina como um nó e introduz o conceito de *workgroup*, ou grupo de trabalho, que representa um grupo de nós que confiam uns nos outros e podem partilhar trabalho. Um nó pode ser qualquer máquina ligada à rede (local ou não), que esteja a executar uma aplicação *DPF4j* ou um simples *daemon* e pode pertencer a um ou mais *workgroups*.

Existem dois modos de utilização da *framework*, sendo eles:

- o *DPF4j Daemon* executa em modo *stand-alone* servindo apenas os nós pertencentes aos mesmos worgroups;
- o *DPF4j embedded* que se trata de quando uma aplicação é desenvolvida utilizando a framework, o programador pode arrancar o *daemon* explicitamente, executando em paralelo com a main thread da aplicação do utilizador.

De forma similar ao *cloud computing*, as aplicações *DPF4j* podem ser extremamente escaláveis devido ao conceito de workgroups, visto que novos nós podem-se juntar ou desassociar a qualquer momento, sendo assim possível gerir os recursos de forma eficiente de acordo com as necessidades. Assim sendo, se forem necessários mais recursos, basta adicionar um novo nó à rede, e este através de um processo de descoberta automático ou manual irá conhecer e dar-se a conhecer aos restantes nós do *workgroup* e ficar assim disponível para ajudar os outros nós a processar o trabalho que estão a executar.

A *framework* tenta facilitar a programação abstraindo ao máximo o programador dos detalhes de distribuição e até mesmo de paralelismo. Um exemplo desta abstração é a disponibilização de ciclos paralelos (e distribuídos) na sua *API*, que basicamente parte do princípio que cada uma das suas iterações ou grupos destas (*chunks*) são independentes e portanto podem ser paralelizadas/distribuídas. Uma utilização da *DPF4j* pode ser tão simples como o for paralelo que se segue:

```
Parallel.exec(new ParallelFor(0, 16, 1) {
public void run() {
System.out.println(“Iteration number: “+ i);
}});
```

Código 30 - Exemplo de *ParallelFor* em *DPF4j*

Este exemplo (Código 30) irá executar 16 iterações do método *run*, e cada uma delas irá simplesmente imprimir o seu número de iteração.

Se o grupo de trabalho for apenas constituído pelo nó local, então a *DPF4j* irá paralelizar o ciclo entre os cores ou processadores da máquina. No entanto, se o grupo de trabalho for composto por mais de um nó, algumas das iterações serão enviadas para os restantes nós para serem executadas. Neste caso a mensagem será impressa no *standard output* dos nós cuja tarefa é executada. A *API* fornece um conjunto de parâmetros que são passados para a interface principal, o *Parallel.exec*(...), que permite uma customização da forma como as iterações serão processadas. Utilizando o mesmo exemplo acima, as iterações podem ser todas processadas individualmente ou então, pode ser definido um bloco, mais conhecido como *chunk*, para a execução ser feita por blocos. A grande vantagem desta funcionalidade, é permitir ao utilizador ajustar o processamento das iterações conforme as suas necessidades. No caso de tarefas muito pequenas, os tempos de transferência por rede podem ultrapassar o tempo de execução da tarefa justificando assim a utilização desta funcionalidade.

A *framework* pode ser altamente parametrizável, mas como foi demonstrado, se assim o desejar, o programador pode-se abstrair completamente de qualquer questão relacionada com a divisão do ciclo ou mesmo a sua distribuição, focando-se apenas na implementação do algoritmo.

A *DPF4j* permite ao programador não só se abstrair da distribuição de processamento como é também responsável pela distribuição automática de *bytecode*. Desta forma, a distribuição é totalmente dinâmica e apenas é necessário que o *DPF4j Daemon* esteja a ser executado no próprio nó. Este *daemon* é responsável por aceitar novo trabalho e ajudar os outros nós a desempenharem as suas tarefas.

A framework *DPF4j* também tenta, dentro das limitações da linguagem *Java*, que o código resultante seja limpo de forma a aumentar a sua legibilidade.

No que toca à distribuição, um factor muito importante é o tipo de rede onde os nós estão localizados. O conceito paralelo distribuído e a *DPF4j* podem tanto ser implementados numa rede local como na Internet, e a única diferença consiste no modo como os vários nós se vão descobrir.

Duma perspectiva de requisitos, é importante manter o conceito independente do tipo de infraestrutura para que a *framework* possa ser aplicada em um grande número de ambientes diferentes, possibilitando assim a agregação de mais recursos e até mesmo criando a possibilidade de distribuir código e dados para ambientes de *cloud computing*.

Como exemplo da eficácia deste conceito imagine-se que em vez de escrever uma mensagem na consola, o ciclo anteriormente apresentado teria no seu conteúdo uma tarefa complexa que demoraria aproximadamente 20 segundos a ser executada por um core. Se se tiver uma rede composta por 4 nós *quad-core* e o ciclo *ParallelFor* criar uma tarefa por cada iteração (criando um total de 16 tarefas), este ciclo poderia demorar pouco mais de 20 segundos, comparativamente com os 320 segundos que demoraria se fosse utilizado um simples for sequencial, ou aproximadamente 80 segundos no caso de ser paralelo. Neste caso teria-se um ganho de performance na ordem dos 1500% (tendo em conta que demoraria quase 300 segundos a menos que um for vulgar). É de notar que neste exemplo os tempos de comunicação e sincronização foram considerados desprezáveis devido à dimensão temporal da tarefa (20 segundos).

6.1 Arquitetura do Sistema (Distribuído)

O *DPF4j* baseia o seu funcionamento em *workgroups*, em que cada *workgroup* é composto por vários nós.

Um nó pode ser qualquer máquina ligada à rede (local ou não), que esteja a executar uma aplicação *DPF4j* ou um simples o *DPF4j Daemon*. Para cada nó, qualquer outro nó é denominado como nó remoto, esteja ele na mesma rede ou não. Quando um nó recebe trabalho de um nó remoto, esse trabalho nunca é reencaminhado para outro nó, isto evita demasiados saltos de rede e garante a privacidade. Apenas o trabalho do nó atual é que pode ser enviado para nós remotos.

Um *workgroup* define um conjunto de computadores (nós) que comunicam e partilham recursos entre si. Este conceito é utilizado nos sistemas operativos da *Microsoft*, com o objetivo de permitir aos computadores de uma determinada rede, e pertencentes ao mesmo *workgroup*, partilhar recursos (impressoras, *scanners*, unidades de rede) entre si. Este

conceito foi adaptado neste âmbito porque na realidade os nós partilham os seus recursos, mais concretamente os seus recursos de processamento (*CPU*, memória) e as aplicações em si. Neste caso, um nó pode pertencer a mais que um *workgroup*, de forma a possibilitar aos administradores do sistema a possibilidade de modular a rede e os recursos da forma que preferirem podendo isolar ou disponibilizar máquinas com base em questões lógicas e de segurança. Do ponto de vista de segurança um *workgroup* representa um grupo de nós que confiam uns nos outros. Para isto, os *workgroups* implementam uma camada de segurança com base no conceito de chaves partilhadas, no entanto, numa rede fechada e segura estes mecanismos podem ser desligados para aumentar o desempenho e evitar encriptações desnecessárias.

A Figura 19 representa uma visão geral da arquitetura de um sistema composto por vários nós e *workgroups*. Como se pode ver trata-se de uma rede local com acesso à Internet, existindo desta forma nós localizados na Internet, i.e. para além da *firewall* local.

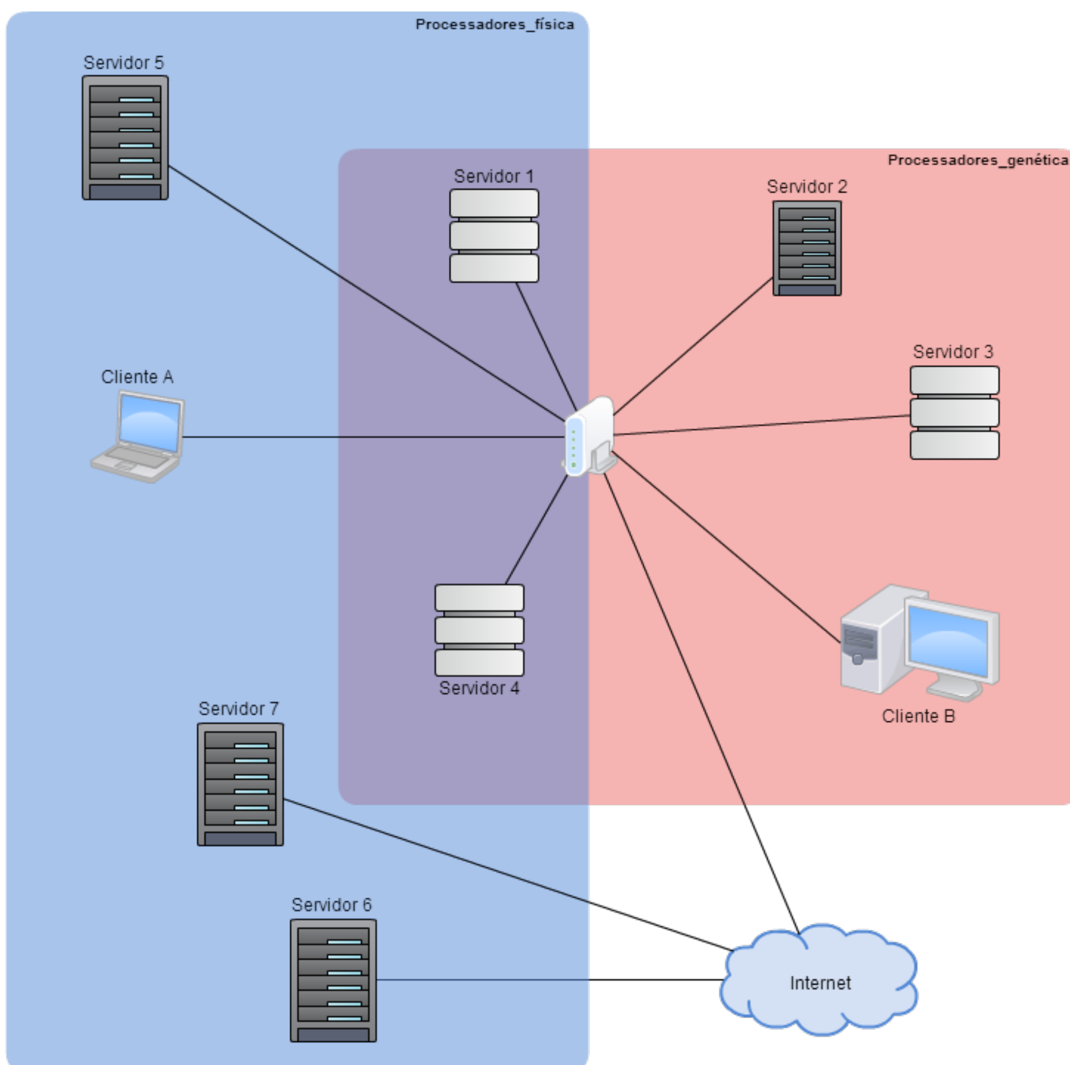

Figura 19 - Arquitetura do Sistema

Como se pode ver na Figura 19, o *workgroup* de nome "Processadores_física" alastra-se à Internet, sendo que a localização dos nós é transparente para o sistema e abre a possibilidade

do Cliente A delegar trabalho para os servidores 6 e 7. Outra capacidade que está representada nesta figura é a possibilidade de um nó pertencer a mais que um *workgroup* como acontece com os servidores 1 e 4 que pertencem em simultâneo ao *workgroup* "Processadores_física" e ao *workgroup* "Processadores_genética" partilhando assim os seus recursos com todas as máquinas presentes na figura.

Devido à vertente configurável da *DPF4j*, como sistema distribuído, pode funcionar num modelo cliente/servidor, *peer-to-peer*, ou num modelo híbrido visto que os nós podem ser clientes e servidores ao mesmo tempo mas existe a possibilidade de desativar qualquer uma destes modos e um determinado nó passar apenas a delegar trabalho ou a aceitar trabalho (caso do *DPF Daemon*).

Por exemplo, quando um nó está a executar em modo *stand-alone*, neste caso trata-se de um servidor que apenas vai servir os nós que delegam trabalho nele. No caso de ser uma aplicação desenvolvida utilizando a *DPF4j*, que delega trabalho, mas também executa trabalho de outros nós, trata-se de um nó que é servidor e cliente ao mesmo tempo.

6.2 Arquitetura da Framework

A *framework DPF4j* é composta por um conjunto de módulos que fazem com que o seu funcionamento seja ajustável às necessidades do programador. A Figura 20 representa a arquitetura global da *framework*, onde se encontram definidos cada um dos módulos que a compõem.

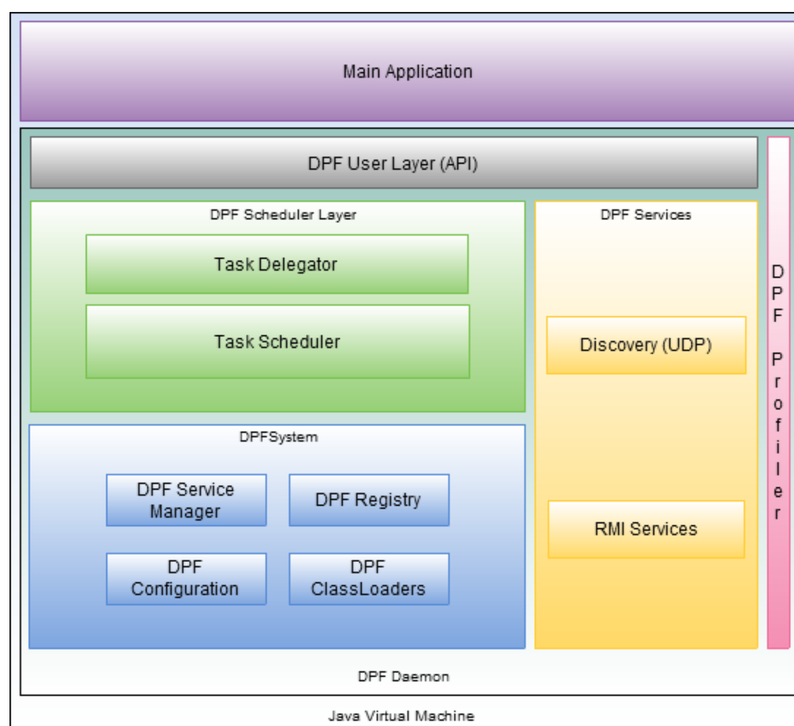

Figura 20 - Arquitetura da *Framework*

Cada um destes módulos é essencial para o funcionamento global da *framework*. Relativamente à *Main Application* representada na arquitetura da *framework*, esta representa a aplicação ou as aplicações desenvolvidas utilizando a *DPF4j*.

6.2.1 DPF Daemon

DPF Daemon é o nome dado ao sistema que corre em *background* e que realiza todas as funções da *framework* que são transparentes ao utilizador. O *Daemon* em tempo de execução é constituído por uma série de *threads* que realizam as várias operações da *framework*. O *DPF Daemon* tem dois modos de operação. No modo *stand-alone* serve apenas os nós pertencentes aos mesmos *worgroups*, especialmente útil para ser colocado em máquinas servidoras destinadas a fornecer poder de processamento aos clientes e o modo *DPF4j embedded* que trata do caso em que uma aplicação é desenvolvida utilizando a *framework* e automaticamente fica apta a delegar e aceitar trabalho. É de notar no entanto que se o utilizador o desejar pode limitar estas capacidades e por exemplo passar apenas a delegar trabalho e não a aceitar.

Relativamente ao arranque do daemon, este é feito de forma diferente para cada um dos modos. No que diz respeito ao modo *embedded*, o seu arranque é feito programaticamente pelo utilizador e com esta capacidade também vem a responsabilidade de encerrar o sistema corretamente no final. O utilizador é que decide quando e onde é que deve ser iniciado. De seguida é apresentado um exemplo da inicialização e paragem do daemon em modo *embedded* (Código 31).

```
DPFDaemon daemon = new DPFDaemon();
daemon.startDaemon(false);
...
daemon.stopDaemon();
```

Código 31 - Arrancar e desligar do DPF Daemon

Em relação ao modo *stand-alone*, esta ação é feita a nível da distribuição da *framework*, sendo necessário executar o *launcher* que se encontra no diretório bin da distribuição.

6.2.2 DPF User Layer

A *DPF User Layer*, também conhecida como *DPF4j API*, fornece um conjunto de interfaces que permitem a exploração do ambiente paralelo e distribuído. A *DPF4j API* além de abstrair o utilizador de todas as questões de distribuição tenta também em certos casos abstrair também as questões de paralelismo e para isso, numa fase inicial, estão disponíveis ciclos básicos tais como: *for* e *foreach.*

De seguida é mostrado um exemplo de um ciclo *foreach* para o cálculo da soma de dois valores (Código 32).

Partindo do princípio que existe uma classe do tipo Conta que contém os atributos *valA*, *valB* e *resultado*. Para além destes atributos fornece um método somaValores que faz a soma de

valA e *valB* e coloca o resultado na variável resultado. Em seguida é apresentado a respectiva definição do objecto.

```
public class Conta {

 private int valA;
 private int valB;
 private int resultado;

 public Conta(int valA, int valB) {
        super();
        this.valA = valA;
        this.valB = valB;
 }

 public void somaValores(){
        resultado=valA+valB;
 }

 public int getResultado() {
        return resultado;
 }

}
```

Código 32 - Classe Conta

Assumindo que existe uma *Array* do tipo Conta com várias posições, de seguida é apresentado um exemplo do for each na linguagem *Java*(Código 33).

```
List<Conta> contas = new ArrayList<>();
...
for (Conta conta : contas) {
 conta.somaValores();
}
...
```

Código 33 - *For each* em *Java*

No que diz respeito à utilização de ciclos na linguagem *Java*, esta é feita de forma simples e fácil. Visto isto, a *DPF4j* segue o mesmo conceito para que os utilizadores não sintam grandes mudanças nesse aspecto. Em seguida é apresentado o mesmo exemplo, mas desta vez utilizando a framework *DPF4j* (Código 34).

```
...
List<Conta> contas = new ArrayList<>();
...
Parallel.exec(new ParallelForEach<Conta>(contas) {
     private static final long serialVersionUID = 1L;

     public void run() {
             currentValue.somaValores();
}
...
});
```

Código 34 - For each em *DPF4j*

Como resultado do processamento, neste caso que não é passado o *chunk size* para o *Parallel*, as iterações serão todas paralelizadas. Como se pode verificar a sua utilização é muito simples.

Outro factor muito importante é o facto de no caso de já existirem aplicações *Java* desenvolvidas seguindo os paradigmas sequenciais ou paralelos oferecidos pela linguagem, estas podem ser facilmente migradas para a *framework*. Como se pode verificar nos exemplos apresentados a migração seria muito simples.

A classe *Parallel* é o ponto de partida para a utilização da *framework*, recebendo como parâmetro as várias implementações dos ciclos.

Desta forma os utilizadores desta podem usufruir deste mecanismo e migrarem o processamento atual das suas aplicações permitindo-o assim ser feito de paralela e distribuída.

6.2.3 DPF Scheduler Layer

Este módulo da *framework* tem como principal função a distribuição de trabalho, tanto a nível local (pelos vários *cores*) como remotamente pelos vários nós remotos. Este módulo é muito importante no que diz respeito a distribuição de processamento. Todas as *tasks* executadas pela *framework*, são tratadas pelo escalonador, sendo este o componente que valida se a *task* pode ser executada localmente, remotamente ou então não podendo ser executada fica bloqueada até existirem recursos disponíveis. Quando uma *task* vai ser enviada para um nó remoto para ser executada, esta é enviada para o *DPF Task Delegator* que está encarregue do tratamento relativo ao envio da *task* para um nó remoto e fazer o respectivo tratamento da resposta do seu processamento. O *DPF Task Delegator* acaba por também ser um escalonador de certa forma pois distribui o trabalho pelas máquinas remotas com base num algoritmo de decisão e rejeita as tarefas quando todos os recursos remotos estão ocupados.

O *DPF Scheduler* é o escalonador do *DPFSystem* e obedece à interface *ExecutorService*. Numa aplicação, um programador pode utilizar quantos escalonadores quiser e submeter trabalho para eles como desejar, mas por omissão a *framework* utilizará sempre o escalonador referenciado pelo *DPFSystem*. No caso de se desejar alterar o escalonador da *framework* por outra implementação basta para isso substituir a referência presente no *DPFSystem* por uma instância de qualquer classe que implemente a interface "*ExecutorService*". Como qualquer classe que implemente a interface *ExecutorService*, o *DPFScheduler* é capaz de aceitar tarefas através dos métodos *submit*, *invokeAll* e *invokeAny*. O primeiro (*submit*) serve para execuções assíncronas e devolve objetos do tipo *Future* (semelhante ao já verificado no *.NET*) que representam o resultado da tarefa. O método *invokeAll* serve para fazer execução síncrona, executa um conjunto de tarefas e devolve um conjunto de resultados quando termina. O método *submitAny* é também síncrono e também recebe múltiplas tarefas, mas serve para fazer paralelização especulativa, isto é, apenas o resultado da primeira tarefa a terminar é retornado.

6.2.4 DPF Services

Tal como do *DPF Scheduler*, este módulo é muito importante no que diz respeito à distribuição de *tasks*, sendo ele o módulo que está exclusivamente dedicado às comunicações

entre nós. Este módulo é utilizado desde que o *daemon* arranca até ao momento em que este encerra.

No que diz respeito á implementação dos serviços, foram utilizadas duas tecnologias de comunicação: *UDP* e *Java RMI*.

Os *DPFServices* dividem-se em duas partes. A primeira destas partes são os serviços de descoberta que, tal como o *JavaParty*, inclui um modo de descoberta automático baseado em *multicast* (*UDP*) em que o nó avisa todos os nós da rede que se ligou, seguido depois de uma fase de negociação em *Java RMI* que autentica a relação entre os dois nós com base numa chave partilhada. O serviço de descoberta tem um modo manual que basicamente segue os mesmos passos do anterior excluindo o momento de *multicast* e recorrendo a portas e endereços IP explícitos no ficheiro de configuração. Finalmente ainda neste grupo inclui-se o serviço de desassociação que os nós utilizam para informar a máquina remota de que vão sair do grupo de trabalho.

O segundo grupo de serviços são os serviços de *runtime* que inclui os serviços responsáveis por aceitar e cancelar a execução de tarefas vindas de outras máquinas, os serviços utilizados pelos *classloaders* remotos quando estes não encontram determinada classe relacionada com a tarefa submetida e outros serviços utilitários.

É de notar que a *framework* foi criada para que qualquer módulo seja adaptável às condições ou exigências do ambiente em que está inserido. Apesar de serem utilizadas estas duas tecnologias para a camada de comunicações, a *framework* foi desenvolvida de maneira a que sejam usados outros mecanismos de comunicação, sendo necessário apenas seguir um determinado contrato (interface).

6.2.5 DPF System

O módulo *DPFSystem* é o core da *framework*, sendo composto por vários sub-módulos. Cada um destes sub-módulos tem um papel importante no sistema, tal como será descrito nas próximas secções. Este módulo pode ser visto como o motor da *framework*, viste que tudo é dependente do seu funcionamento, desde a configuração, escalonamento até à execução remota de tarefas.

6.2.5.1 DPF Configuration

O sistema *DPF4J* é totalmente configurável pelo utilizador. Foi dado um ênfase muito grande ao sistema de configurações porque existe o objetivo de tornar a *framework* o mais configurável possível ao nível das funcionalidades, da performance e da segurança, de forma a se adaptar ao máximo de ambientes e abranger assim um público-alvo amplo. O módulo que centraliza todas estas configurações chama-se de *DPF Configuration* e pode ser obtido através do objecto *DPF System*. Este módulo não se limita a ser um simples repositório de configurações, implementando uma série de algoritmos para que a grande quantidade de configurações necessária para tornar a *framework* versátil a todos estes níveis não resulte num maior esforço de aprendizagem e dificulte o processo da sua utilização, o que foi

identificado como um dos principais defeitos da maior parte dos trabalhos desenvolvidos nesta área e poderá ter uma grande influência no seu sucesso.

O sistema de configurações segue uma cadeia hierárquica para que o utilizador possa configurar o estritamente essencial estando assim preparado tanto para utilizadores mais leigos como para utilizadores avançados que pretendem afinar a *framework* ao máximo para as suas necessidades.

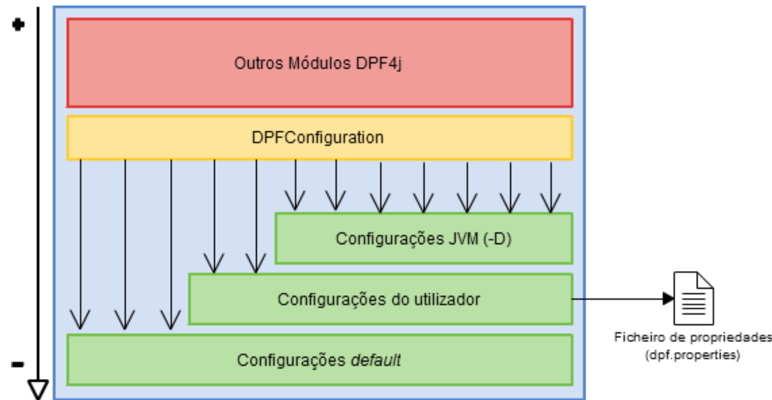

Figura 21 - Diagrama de prioridades da configuração da *framework DPF4j*

A hierarquia de configuração é a seguinte (da mais prioritária para a menos prioritária tal como é apresentado na Figura 21:

Propriedades da JVM: O nível mais alto a que se pode configurar um certo parâmetro são as propriedades da máquina virtual (argumentos começados por -D na execução do comando *Java*, Código 35, ou seja, se um utilizador executar uma aplicação *DPF4j* com uma definição ao nível da máquina virtual esta irá prevalecer sobre qualquer configuração feita num dos outros níveis. Este tipo de configuração é útil para quando se deseja alterar uma definição para uma execução em particular.

```
/>java -help
Usage: java [-options] class [args...]
           (to execute a class)
   or  java [-options] -jar jarfile [args...]
           (to execute a jar file)
where options include:
    -d32          use a 32-bit data model if available
    -d64          use a 64-bit data model if available
    -server       to select the "server" VM
    -hotspot      is a synonym for the "server" VM  [deprecated]
                  The default VM is server.

    -cp <class search path of directories and zip/jar files>
    -classpath <class search path of directories and zip/jar files>
                  A ; separated list of directories, JAR archives,
                  and ZIP archives to search for class files.
    -D<name>=<value>
                  set a system property
```

Código 35 - Excerto da ajuda do comando *Java*

Ficheiro de Propriedades: A principal forma de configuração da *DPF4j* é através de um ficheiro de propriedades. Este ficheiro de propriedades deverá ter o nome de "dpf.properties" e deverá estar incluído na raiz do *classpath* da aplicação. Este ficheiro será detalhado mais à frente neste capítulo.

***Defaults*:** Para evitar configurações existe um esforço para que todas tenham um valor por omissão e permitir assim que qualquer programador comece a utilizar a *framework* com o mínimo de esforço de configuração e por consequência, de aprendizagem.

O sistema de configuração com base em ficheiro de propriedades foi fortemente baseado no modo de configuração da *framework log4j* [19] devido ao uso generalizado da mesma e da familiarização dos programadores de Java com esta framework. Tal como foi apresentado anteriormente e tal como acontece na *framework log4j* o ficheiro de configuração não tem de ser explicitamente declarado pelo utilizador mas em vez disso este deverá ter um nome predefinido (dpf.properties) e estar presente na raiz do *classpath*.

Internamente o dpf.properties é um ficheiro de propriedades normal à parte da configuração de *workgroups*, que funciona de forma muito semelhante à configuração de *appenders* do log4j [20]. Chama-se *appender* ao componente responsável por fazer output dos *logs* para determinado destino (pe. o objeto da framework que escreve os *logs* para ficheiro ou para a consola). Existe uma propriedade chamada `dpf.workgroups` onde deverão ser definidos todos os *workgroups* a que o nó pertence separados por vírgulas, e a partir daí o sistema de configuração dinamicamente procurará todas as propriedades começadas por `dpf.workgroup.<nome do workgroup>` por configurações específicas de cada grupo de trabalho. No Código 36 pode-se ver a configuração do *workgroup* testGroup com a *password* "123321" e que irá enviar o "test.jar" e o conteúdo do diretório "c:/temp/classes" para todos os nós que descobrirem esse grupo.

```
# workgroups ativos
dpf.workgroups=testGroup


#definição de um workgroup
dpf.workgroup.testGroup.password=123321
dpf.workgroup.testGroup.classpath=c:/temp/test.jar;c:/temp/classes
```

Código 36 - Exemplo de configuração do *workgroup* testGroup

6.2.5.2 DPF Registry

É no módulo *DPF Registry* que é feito o registo de todas as ligações do nó local com todos os outros nós remotos e ainda a listagem de todos os *workgroups* ativos a que pertence. O *DPF Registry* é como uma lista de contactos de um nó e é utilizado para vários fins.

A manipulação desta informação é feita orientada aos *workgroups*, tendo cada *workgroup* a si associada uma lista de nós remotos. Este registo é utilizado pelo escalonador da *framework* para obter todos os nós conhecidos no início da execução de trabalhos, é utilizado pelo sistema de descoberta para dar a conhecer aos nós remotos os nós conhecidos pelo nó local pertencentes ao mesmo *workgroup* e é ainda utilizado para obter qualquer informação acerca dos nós e *workgroups* conhecidos como por exemplo as chaves de encriptação.

Esta forma de gerir os nós é muito útil para o sistema de descoberta, tornando-o mais eficaz e eficiente.

Este módulo permite ainda o registo de *listeners* podendo informar qualquer entidade que esteja à escuta da associação ou desassociação de nós em tempo de execução.

6.2.5.3 DPF Classloaders

A *framework* utiliza vários *classloaders* para permitir a execução de *tasks* em nós remotos sem que haja a necessidade de se fazer a instalação manual de binários nesses e evitando a instalação de binários desnecessários limitando-se às classes estritamente essenciais. Outra grande vantagem é que todo este processo de transferência de código (e respectivas dependências) é totalmente transparente ao utilizador no processo de delegação de trabalho entre nós.

Outra vantagem de utilizar este sistema é a possibilidade de carregar classes em tempo de execução em vez de implicar a preparação de um ambiente de execução prévio à execução da aplicação. Esta abordagem faz com que a primeira execução de uma tarefa cujas classes são desconhecidas seja mais lenta mas se assim o desejar é possível definir uma série de classes/*jars* que o nó que submete a tarefa pode enviar no arranque da aplicação. Tendo em conta os testes da *framework* desenvolvida isto só se justifica em casos de *classpath* extenso visto que o tempo de transferência de uma classe individual é praticamente imediato.

Existe um sistema de cache que faz com que a performance aumente quando se trata de delegação de tarefas repetidas, ou seja, o mesmo código, as mesmas dependências mas com dados distintos. O que acontece é que o sistema guarda o código e as suas dependências em *cache* no sistema de ficheiros para que nas próximas execuções não haja a necessidade de pedir estas dados ao nó que está a delegar trabalho.

6.2.5.4 DPF Service Manager

Todas as questões relativas à gestão dos serviços são feitas no módulo *DPF Service Manager*.

Neste momento este módulo apenas é utilizado no *Daemon* durante as operações de arranque e desligar, para fazer o respectivo arranque e encerramento dos serviços utilizados no sistema. No entanto ele já foi desenvolvido com a intenção de no futuro integrar uma interface *JMX* [21] para gestão remota do funcionamento do *DPF Daemon*. *JMX* é uma tecnologia que permite criar soluções modulares para monitorização, gestão de aplicações e dispositivos através de serviços remotos.

6.3 DPF Profiler

O *DPF Profiler* não é propriamente um módulo da *framework* mas sim uma ferramenta que o administrador do sistema pode utilizar para tirar métricas do comportamento da *framework* para a poder afinar de forma a ter a melhor performance para a situação a que ela está aplicada. O *DPF Profiler* cria um ficheiro *CSV* com detalhes da execução de vários módulos da *DPF4j* incluindo tempos, erros, avisos e dimensão de artefactos. A opção de escrever estes dados em *CSV* facilita a importação destes dados para folhas de cálculo ou bases de dados

para de seguida fazer cálculos ou *data mining* para extrair informações sobre o desempenho e retirar conclusões.

O *DPF Profiler* extrai informações de desempenho das seguintes funcionalidades:

- *DPF Daemon*
- Sistema de descoberta automático
- Serialização e desserialização de objetos
- Encriptação e desencriptação de dados
- *Classloading*
- Execuções remotas/Rede

É de notar que a ativação deste sistema tem algum peso no sistema, e tendo em conta que este tem escrita em ficheiro (que é uma operação síncrona) os tempos apresentados por este sistema serão mais elevados do que se o mesmo estivesse desativado daí este sistema ser uma ferramenta de *tuning* que deve ser utilizada apenas para configuração do sistema e não deve estar ligado de forma constante.

A ativação deste sistema é feita através da propriedade de configuração `dpf.system.profiler.file`, nesta propriedade deve-se indicar a localização para onde a *DPF4j* deve escrever os detalhes de execução (a localização aconselhada é ${DPF4J_HOME}/log/dpf4j.profiler.csv). Na ausência desta configuração o *DPF Profiler* ficará descativado.

O formato de saída do ficheiro pode ser alterado através da configuração `dpf.system.profiler.pattern` através da introdução de um formato de "*MessageFormat*" [22] em que cada uma das colunas é identificada pelos números presentes entre parênteses rectos na explicação das colunas que se apresenta de seguida. O formato *default* é `"{0};{1};{2};{3}-{4};{5};{6};{7};{8}"`, o que dará origem a um ficheiro de saída com seguinte formato (Figura 22):

```
<id do nó>;<tempo (ms)>;<categoria>;<ação>-<fase>;<identidade>;<erro>;<tamanho>;<notas>
```

	A	B	C	D	E	F	G	H
1	NodeID	Time	Category	Action-Phase	Identity	Error	Size	Notes
2	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835690435	System	Daemon-start	-1	false	-1	
3	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835690790	Task	Decrypt-start	934449077	false	176	
4	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835690905	Task	Decrypt-end	934449077	false	176	
5	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835690906	Task	Deserialize-start	934449077	false	172	
6	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835690917	Task	Deserialize-end	934449077	false	172	
7	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835690925	Task	Serialize-start	1834466260	false	-1	
8	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835690926	Task	Serialize-end	1834466260	false	172	
9	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835690926	Task	Encrypt-start	1834466260	false	172	
10	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835690926	Task	Encrypt-end	1834466260	false	172	
11	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835700633	Task	Scheduling-start	1293126184	false	-1	
12	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835700645	Task	Scheduling-start	1293126184	false	-1	
13	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835700646	Task	Scheduling-start	1293126184	false	-1	
14	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835700646	Task	Scheduling-start	1293126184	false	-1	
15	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835700647	Task	Scheduling-start	1293126184	false	-1	
16	b371a365-519c-4a54-9b24-debbbaa5c5d1	1348835700647	Task	Scheduling-start	1293126184	false	-1	

Figura 22 - Resultado do *DPF Profiler*

Id do nó [0]: Identificação do nó que escreve a mensagem. Esta informação é muito importante para quando se combina dois ficheiros conseguir saber a que nós pertencem os

dados, porque deve existir o cuidado de não fazer cálculos com dados provenientes de nós diferentes a menos que haja a garantia de que os relógios se encontram sincronizados.

Tempo (ms) [1]: Momento em que a mensagem foi escrita em milissegundos.

Categoria [2]: Este elemento serve apenas para fins de organização e identifica a funcionalidade ou módulo que escreveu os detalhes.

Ação [3]: Ação que estava a ser efectuada e levou à escrita dos detalhes.

Fase [4]: Fase em que a ação se encontrava, por norma início ou fim.

Identidade [5]: Identidade do objecto sobre o qual estava a ser efectuada a ação. Este detalhe é muito importante no que toca a combinar dados.

Erro [6]: Se existiram erros durante a execução da tarefa.

Tamanho (bytes) [7]: Tamanho do artefacto sobre o qual a ação foi efectuada (importante em serializações, encriptações e transferências).

Notas [8]: Qualquer tipo de informação sobre a execução que o programador pretenda adicionar.

Como é visível na figura acima por norma é utilizado o valor -1 quando um certo valor numérico não é aplicável à ação em questão e o cabeçalho da tabela é automaticamente gerado também com base no formato configurado.

Um exemplo de uso deste ficheiro, utilizando a figura presente acima, seria extrair o tempo de desencriptação do pacote de resposta ao *discover* que na tabela se apresenta como o primeiro "Decrypt-start" logo após ao "Daemon-start". Para se obter o tempo de execução teria de se subtrair o tempo do "Decrypt-start" ao "Decrypt-end" que tenham a mesma correspondência a nível de "NodeId", que representa o nó que efetuou a tarefa, e "Identity", que identifica o objeto que foi desencriptado. No caso duma análise em massa seria depois interessante relacionar os tempos obtidos deste tipo de operação com os dados presentes na coluna "Size" que apresenta o tamanho do objecto que foi desencriptado em *bytes*.

6.4 Distribuição DPF4j

Relativamente à distribuição da *framework* esta foi concebida de maneira a que seja executada em várias plataformas existentes. Nomeadamente foi testada em *Ubuntu*, *Mac OS X*, *Microsoft Windows* 7 e *Microsoft Windows* 8.

Na Figura 23 está apresentada a estrutura do diretório da distribuição da *framework*:

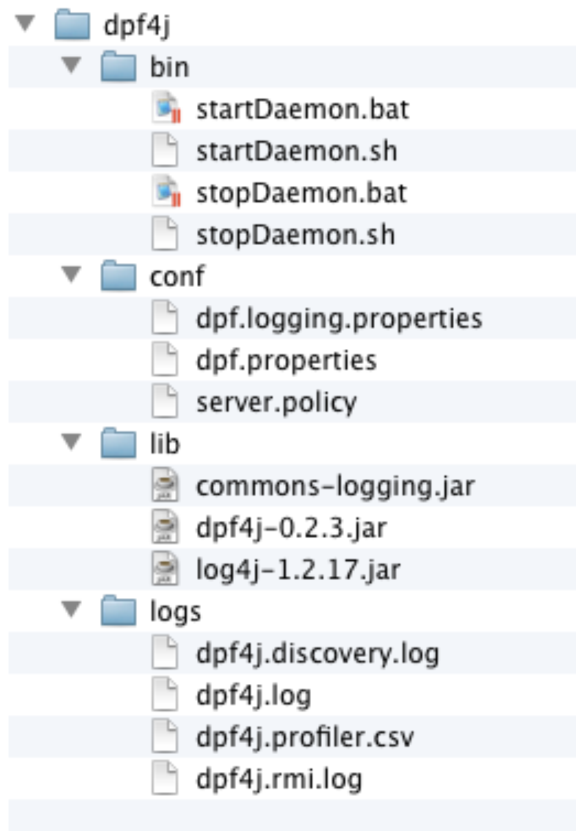

Figura 23 - Diretório da distribuição *DPF4j*

Em seguida é feita uma breve descrição de cada diretório pela ordem que se encontra na Figura 23.

Bin: O diretório bin tem os *launchers* que executam ou param o *DPF4j Daemon* para as várias plataformas, o *.bat* (*batch*) para sistemas *Windows* e o *.sh* (*Shell Script*) para sistemas variantes do *Linux/Unix*.

Conf: Neste diretório encontram-se todas as configurações referentes à *framework*. É de notar que o ficheiro server.policy também se encontra neste diretório. O server.policy ao contrário dos outros dois ficheiros presentes neste diretório não diz respeito diretamente à *framework* mas sim à máquina virtual. Este ficheiro contém definições de permissões relativamente à máquina virtual de *Java*. Podem ser definidas permissões a vários níveis tais como: acessos ao sistema de ficheiros, serialização de objetos, comunicações, *classloaders*, etc. [23] [24].

Lib: O *jar* da *release* da *framework DPF4j* encontra-se neste diretório, assim como todas as dependências como é o exemplo da *commons-logging*.jar e do *log4j-1.2.17.jar* (opcional). É de notar que a única real dependência da *DPF4j* é o *commons-logging* porque existiu uma grande preocupação em não utilizar bibliotecas de terceiros, tanto por questões de licenciamento, como por questões de facilitar qualquer utilizador utilizar a *framework* sem aumentar drasticamente os requisitos dos seus projetos.

Logs: Todos os *logs* referentes à *framework* são criados neste diretório. Sendo os ficheiros com extensão .log referentes ao *logging* do sistema e os *.csv* referentes ao *DPF Profiler*.

7 API

Sendo a *DPF4j* uma *framework* para desenvolvimento de aplicações paralelas/distribuídas, a sua interface com o utilizador é um factor muito importante para o sucesso da mesma. Um aspecto muito forte desta *framework* é o facto de ela ser *Java standard*, não estendendo a linguagem, tanto a nível sintáctico como semântico, o que é uma mais-valia visto que o nível de dificuldade de aprendizagem dos utilizadores pode influenciar o sucesso da sua adoção. Relativamente a questões de migração de código, as alterações são mínimas, sendo apenas necessário alterar os ciclos *for* para o equivalente da *DPF4j*, não sendo um processo moroso nem complexo visto que esta oferece alguma compatibilidade do ciclo.

Neste capítulo é apresentada a forma como foi concebida a interface com o utilizador e o seu funcionamento.

O ponto de partida para o uso da *DPF4j* é a classe *Parallel*, que tem um conjunto de métodos que permitem executar vários tipos de ciclos, neste caso paralelos/distribuídos. Neste fase a *DPF4J* fornece a implementação de dois ciclos: *for* e o *foreach*.

Na próxima secção é feita uma abordagem geral em relação aos ciclos da *DPF4j* e em seguida são apresentadas de forma detalhada as implementações dos ciclos que a *framework DPF4j* fornece ao programador.

7.1 Ciclos

Os ciclos, ou iteradores, nas linguagens de programação são essenciais para percorrer uma determinada lista de valores para que estes sejam manipulados com um determinado objetivo.

A grande maioria das linguagens de programação disponibilizam na sua *API* formas de iterar sobre listas. Visto que a *DPF4j* foi desenvolvida na linguagem *Java* é possível utilizar os ciclos nativos para o caso de operações com baixas necessidades computacionais e utilizar os ciclos fornecidos *pela DPF4j API*, caso o programador deseje, para situações computacionalmente exigentes.

Os ciclos fornecidos pela *API* da *DPF4j* têm como principal objetivo paralelizar e distribuir o processamento das iterações dos ciclos. Relativamente aos ciclos da *framework*, estes são denominados de *ParallelFor* (*for*) e *ParallelForEach* (*foreach*). Estas classes são apresentadas em pormenor nas próximas secções assim como serão também apresentados alguns exemplos práticos.

Para a utilização dos ciclos paralelos são necessários dois objetos, a classe *Parallel*, que é a classe que executa a implementação dos ciclos, mais a classe que implementa o próprio ciclo, podendo esta última ser do tipo *ParallelFor* ou *ParallelForEach*. Todas as implementações dos ciclos são classes abstratas que contêm propriedades específicas do ciclo e são uma extensão à superclasse abstrata *DPFRunnable*. Esta classe implementa as *interfaces Runnable,* nativa do *Java,* e a *interface DPFTask(DPF4j)*.

A *interface DPFTask* estende as *interfaces Serializable* e *Callable*. Visto que as tarefas no *DPF4j* podem ser distribuídas por rede, foi necessário implementar a *interface Serializable*. A *interface Serializable* do *Java* permite que um determinado objeto que a implemente seja convertido para *bytes*, neste caso com o objetivo de ser enviado por rede. A *interface Callable*, requer a implementação do método *call*(), o que na *framework* é utilizado para executar uma tarefa, quer remotamente ou local.

De seguida (Código 37) é apresentado um excerto de código que faz a execução de uma determinada *DPFTask*.

```
DPFTask<T> callable = …
T result = callable.call();
```

Código 37 - Exemplo da execução de uma *DPFTask*

Relativamente ao método *call()* implementado na classe abstrata *DPFRunnable*, esta simplesmente invoca o método *run*() definido pelo programador quando define o ciclo *DPF4j* como se pode verifica no excerto de código seguinte (Código 38).

```
public abstract class DPFRunnable<OUTOUT> implements Runnable,
DPFTask<OUTOUT> {

@Override
public OUTOUT call() throws DPFExecutionException {
run();
return null;
}

@Override
public void run() {
}
}
```

Código 38 - Definição da classe *DPFRunnable*

Na Figura 24 é apresentado o diagrama de classes onde é possível ter uma melhor percepção da relação entre as classes.

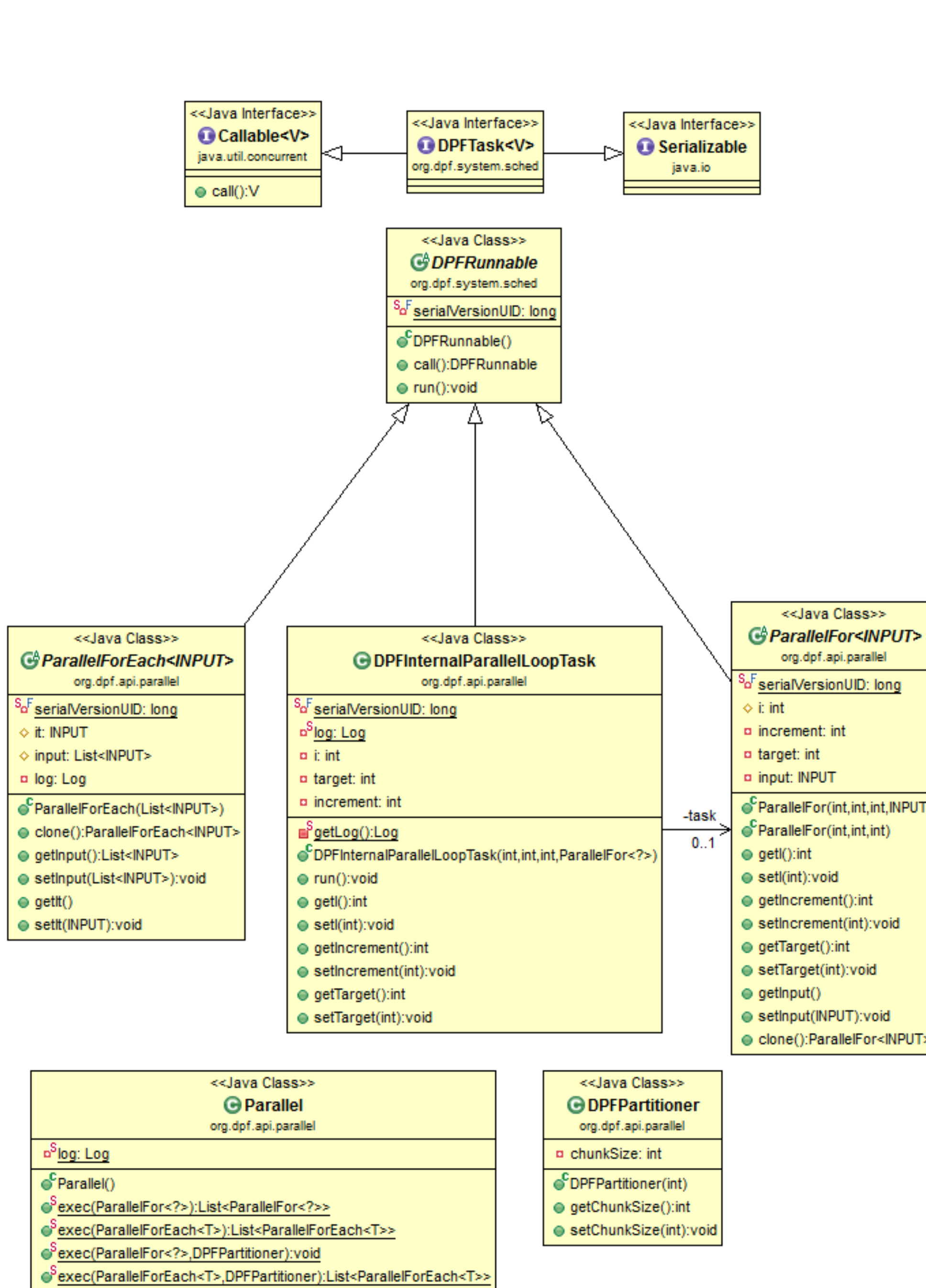

Figura 24 - Diagrama de classes da *DPF4j API*

Todas estas implementações requerem a implementação do método *run()* que representa o corpo do ciclo. Neste caso é utilizado um tipo especial de *DPFTask* chamado *DPFRunnable.* Este tipo tem a característica de que o seu resultado é a própria instância do objeto *DPFRunnable*.

Nas próximas secções são apresentadas as implementações para cada um dos ciclos fornecidos pela framework *DPF4j*.

7.1.1 ParallelFor

A classe *ParallelFor* é a representação do ciclo *for* fornecido nativamente pelo *Java*. O extrato Código 40 apresenta um exemplo da utilização do ciclo *ParallelFor*.

Java:

```
for(i=1;i<11;i++){
     System.out.println("Iteration number: "+ i);
}
```

Código 39 - Exemplo de um ciclo for em *Java*

DPF4j:

```
Parallel.exec(new ParallelFor(1, 11, 1) {
public void run() {
System.out.println("Iteration number: "+ i);
}});
```

Código 40 - Exemplo de um ciclo for (*ParallelFor*) em *DPF4j* equivalente ao Código 39

Visto que não foi feita qualquer extensão à linguagem *Java*, surgiu a necessidade de utilizar a programação orientada a objetos e a utilização dos genéricos do *Java* para se conceber o equivalente ao for nativo.

Relativamente à forma como o ciclo pode ser utilizado, na Tabela 7 encontram-se as definições das assinaturas dos seu construtores.

Assinatura	Descrição
ParallelFor(a,b,c)	**Construtor simples, apenas com os índices e incremento definidos.**
1.1.1 ParallelFor(a,b,c,d)	**Construtor que recebe o valor dos índices e incremento, mais o objecto que contém informação a ser manipulado no conteúdo do ciclo.**

Tabela 7 - Definição dos construtores da classe *ParallelFor*

Na Tabela 8 são apresentados os parâmetros que os construtores do *ParallelFor* recebem.

Parâmetro	Descrição
a	**Definição do valor do índice inicial do ciclo**
b	**Definição do valor do índice para paragem ciclo**
c	**Definição do valor a incrementar a cada iteração.**
d	**Objeto genérico.**

Tabela 8 - Definição dos parâmetros do construtor *ParallelFor*

Para manipulação de dados, a *framework* fornece o acesso ao número da iteração atual, através da variável interna *i*. Relativamente aos dados passados por parâmetro, a *framework* fornece o objeto *input* que representa a instância passada por parâmetro no construtor do objeto *ParallelFor*.

De seguida é apresentado um exemplo da multiplicação de matrizes. Inicialmente foi necessário criar um objeto que simplesmente vai conter os arrays que representam as matrizes e o respectivo número de linhas e colunas. Este objeto é passado por parâmetro no construtor do ciclo *ParallelFor*.

```
public class MatrixContainer implements Serializable {

  int[][] a, b, c;
  int nLinhas, nColunas;

public MatrixContainer(int[][] a, int[][] b, int[][] c, int lines, int
cols) {
     super();
     this.a = a;
     this.b = b;
     this.c = c;
     this.lines = lines;
     this.cols = cols;
}
}
```

Código 41 - Definição do objeto *MatrixContainer*

Partindo do princípio que já existe uma instância do objecto *MatrixContainer* (Código 41) com os respectivos atributos inicializados, a utilização do ParallelFor é apresentada no extrato de código 6:

```
int nLinhas=...
int nColunas=...
MatrixContainer matrixContainer=.....
….
List resultado = Parallel.exec(new
ParallelFor<MatrixBundle>(nLinhas,nColunas, 1, matrixContainer) {

public void run() {
for (int j = 0; j < getInput().getLines(); j++) {
                        for (int k = 0; k < getInput().getLines(); k++) {
                                    getInput().getC()[i][j] +=
getInput().getA()[i][k] * getInput().getB()[k][j];
}
          }
}
});
```

Código 42 - Exemplo da utilização do *DPF4j ParallelFor*

Como podemos verificar no exemplo (Código 42), a instância da iteração atual é obtida pelo `getInput()`, que depois de acedida oferece o acesso às matrizes definidas no objecto para a respectiva manipulação de valores. É de notar que o valor da iteração atual `i` é utilizada na primeira parte do cálculo do resultado, `getInput().getA()[i][k]`.

Por fim, no que diz respeito ao resultado do processamento de todas as iterações o método da classe *Parallel* retorna uma lista de objetos do tipo *ParallelFor*, que representam cada iteração efectuada.

No caso do exemplo acima (Código 42 - Exemplo da utilização do *DPF4j ParallelFor*), como o resultado do processamento teríamos o objecto resultado que é um *List* do tipo *ParallelFor*. Neste caso para se obter o resultado total, ou seja a matriz resultado, seria necessário

percorrer cada iteração do List e obter a linha referente no atributo *input* (instancia do objeto *MatrixContainer*) do *ParallelFor*.

7.1.2 ParallelForEach

A *DPF4j* fornece o *ParallelForEach* como implementação do *for each* nativo do *Java*. Tal como o *ParallelFor*, a sua utilização é muito simples. Utilizando o exemplo que se encontra na secção 6.2.2,temos o objeto *Conta* (Código 43) que contém os atributos *valA*, *valB* e *resultado*. Para além destes atributos o objeto Conta fornece o método somaValores que faz a soma de valA e valB e coloca o resultado na variável resultado. Em seguida é apresentada a respectiva definição do objecto (Código 43).

```
public class Conta {

 private int valA;
 private int valB;
 private int resultado;

 public Conta(int valA, int valB) {
        super();
        this.valA = valA;
        this.valB = valB;
 }

 public void somaValores(){
        resultado=valA+valB;
 }

 public int getResultado() {
        return resultado;
 }

}
```

Código 43 - Definição do objeto Conta

Partindo do principio que o *array* já se encontra devidamente preenchido com os respetivos valores, em seguida é apresentado um exemplo do *for each* na linguagem *Java* (Código 44).

```
...
List<Conta> contas = new ArrayList<>();
…
for (Conta conta : contas) {
 conta.somaValores();
}
...
```

Código 44 - Exemplo de um *foreach* em *Java*

O excerto de código apresentado em seguida representa o *foreach* equivalente utilizando a *framework DPF4j* .

```
...
List<Conta> contas = new ArrayList<>();
...

Parallel.exec(new ParallelForEach<Conta>(contas) {
   private static final long serialVersionUID = 1L;

   public void run() {
           it.somaValores();
}
...

});
```

Código 45 - Exemplo de um *foreach* (*ParallelForEach*) em *DPF4j*

Relativamente ao *ParallelForEach*, este apenas oferece um único construtor, `ParallelForEach<T>(a)` , em que *a* é a instância do objeto *List* que o programador deseja iterar e *T* é o tipo de objeto de que o *List* é composto (`List<T>`). No que diz respeito ao acesso à iteração atual, a *DPF4j* disponibiliza a variável `it` que representa o objeto que se encontra na respectiva iteração.

Por fim, para se obter o resultado da execução do *ParallelForEach*, é necessário efetuar um conjunto de operações semelhante ao caso do *ParallelFor*, no entanto, no *ParallelForEach*, é disponibilizada a iteração corrente, o `it`, logo no ciclo para percorrer o resultado apenas é necessário fazer o get da iteração que se encontra no *ParallelForEach*, o `getIt()`.

7.2 Parallel

A classe *Parallel* é uma simples classe com métodos estáticos cujo objetivo é realizar todo o trabalho referente à fragmentação de uma tarefa em várias subtarefas. A utilização da *framework* baseia-se simplesmente nesta classe, que por si delega todo o trabalho no *DPF Scheduler*. Para cada ciclo, esta classe, implementa um método *exec()* responsável pela execução de cada uma das implementações dos ciclos.

A *framework* permite também a definição de blocos para as iterações, tal como o *OpenMP*, *TBB* e *.Net*. Esta funcionalidade é definida utilizando o objecto *DPFPartitioner* (inspirado pelo objecto da *framework .NET*) que contém um atributo que permite ao programador definir o numero de iterações a serem executadas em bloco. Para cada método de execução da classe *Parallel*, existe um outro método com assinatura semelhante mas com mais o um parâmetro, o *DPFPartitioner*. Desta forma, o programador informa a *framework* que deseja que a divisão do trabalho total seja feita em blocos de *N* iterações.

Na Tabela 9 são apresentadas as assinaturas de todos os métodos *exec* que a classe *Parallel* disponibiliza para os programadores.

Assinatura		Descrição
2.1.1	List<ParallelFor<?>> exec(ParallelFor<?> parallelFor)	**Método de execução do ciclo ParallelFor em que para cada iteração será criada uma tarefa.**
2.1.2	public static <T> void exec(ParallelFor<?> parallelFor, DPFPartitioner partitioner)	**Método de execução do ciclo ParallelFor em serão criadas tarefas com N iterações cada uma.**
2.1.3	List<ParallelForEach<T>> exec(ParallelForEach<T> parallelForEach)	**Método de execução do ciclo ParallelForEach em que para cada iteração será criada uma tarefa.**
2.1.4	List<ParallelForEach<T>> exec(ParallelForEach<T> parallelForEach, DPFPartitioner partitioner)	**Método de execução do ciclo ParallelForEach em que serão criadas tarefas com N iterações cada uma.**

Tabela 9 - Definição dos construtores da classe *ParallelForEach*

Num ciclo sequencial, o código só prossegue com a sua execução após o fim do ciclo. Neste caso os ciclos paralelos seguem a mesma filosofia e como os anteriores, assim que uma *thread* entra num ciclo paralelo/distribuído, esta só terá o controlo de volta quando todas as iterações tiverem sido executadas. Isto é feito graças à natureza bloqueante do objeto *Future*, que coloca a *thread* em espera até que um resultado tenha sido alcançado.

Nas próximas secções é apresentada a forma como o *Parallel* efetua a criação de subtarefas e como as delega no *DPF Scheduler*.

7.2.2 Execução de um ParallelFor

Relativamente à execução do ciclo *ParallelFor*, a sua execução baseia-se na utilização da função clone oferecida pelo *Java*, que consiste na duplicação de uma instância.

Na Figura 25 é apresentado o diagrama de fluxo da execução do ciclo *ParallelFor*.

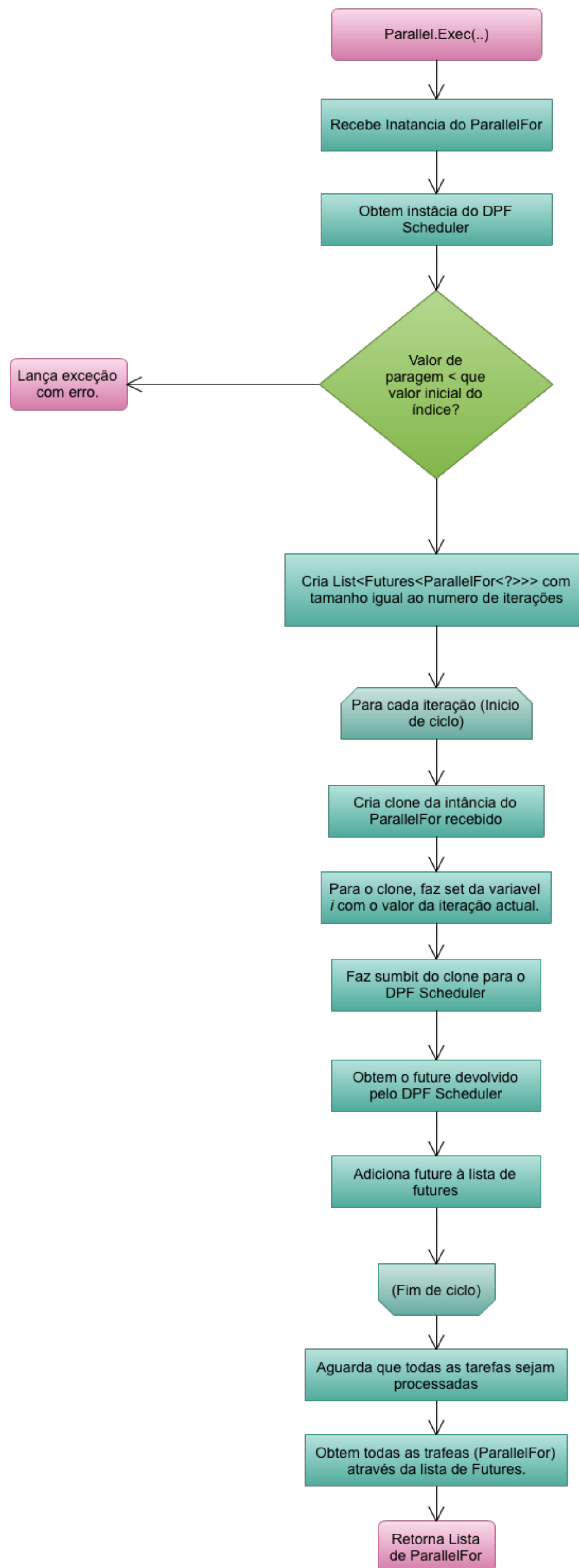

Figura 25 - Diagrama de fluxo do processo de criação de subtarefas na execução de um *ParallelFor*

O método *exec* quando recebe apenas como parâmetro uma instância de um objeto do tipo *ParallelFor*, significa que todas as iterações irão ser paralelizadas/distribuídas. O primeiro passo é a validação dos parâmetros de entrada. Primeiro é validado se o valor inicial do índice não é igual ou superior ao valor do índice final. O segundo passo é obter a instância do escalonador do sistema através do módulo *DPF System*. Após o passo anterior, é criado um objecto do tipo `List<Future<ParallelFor<?>>>` com o mesmo numero de iterações. A utilização deste tipo de objetos é essencial para fazer a sincronização de tarefas, ou seja, através deles é possível saber quando o processamento de uma determinada tarefa está concluído ou não. Realizadas todas as operações essenciais para se iniciar a divisão de trabalho, chega o momento de a realizar. A divisão de trabalho, neste caso trata-se de dividir cada iteração do ciclo numa tarefa. Dado isto, foi necessário criar um ciclo para iterar o *array* definido pelo programador para que as tarefas sejam criadas.. Para cada iteração do ciclo é feito um clone da instância do *ParallelFor* recebida por parâmetro, mas em vez de ser totalmente igual, o valor da iteração atual, a variável *i*, passa a ser igual ao número da iteração atual. Depois do clone ser colocado em execução e de ter o valor da iteração igual à iteração atual é feita a submissão da tarefa para o escalonador do sistema. Este devolve uma instância de um objecto *Future* que é adicionada à lista de *Futures* anteriormente criada.

Depois de todas as tarefas criadas e submetidas, a *thread* bloqueia e só retoma o controlo depois de todas as tarefas serem processadas. Depois de retomado o controlo, a lista `List<ParallelFor<?>>` com os resultados das tarefas executadas é obtida através do *list* de *Futures*. A lista `List<ParallelFor<?>>` é retornada para que o programador possa iterar sobre ela e obtenha os respectivos resultados por iteração. Está solução não é muito *user friendly*, tal como os programadores estão habituados a utilizar nos ciclos nativos oferecidos pelo *Java standard.* Esta questão deve-se ao facto de não existir memória partilhada, o que não permite a obtenção dos resultados apenas num único objeto resultado. Este é um dos aspetos está registado para trabalho futuro da *framework*.

Quando é passado por parâmetro o objecto *DPFPartitioner* o comportamento é semelhante ao anterior, apenas é alterada a forma como são criadas as tarefas. Neste caso uma tarefa não corresponde apenas a uma única iteração, mas sim a um conjunto de iterações, o chamado bloco de iterações.

Umas das diferenças, relativamente ao processo anterior, é o facto de ter que existir um passo adicional no cálculo dos sub-intervalos do índice para cada bloco de iterações.

Para realizar este processo de criação de blocos foi necessário criar um tipo de *wrapper* que vai executar esse mesmo bloco de iterações, o *DPFInternalLoopParallelTask*. Este tipo de *wrapper* é uma simples tarefa que tem como objetivo percorrer um bloco de iterações.

Na Figura 26 é apresentado um diagrama exemplo que representa este processo de criação de blocos feito pela *DPF4j*, para um ciclo de vinte iterações cujo processamento será efectuada em blocos de cinco iterações.

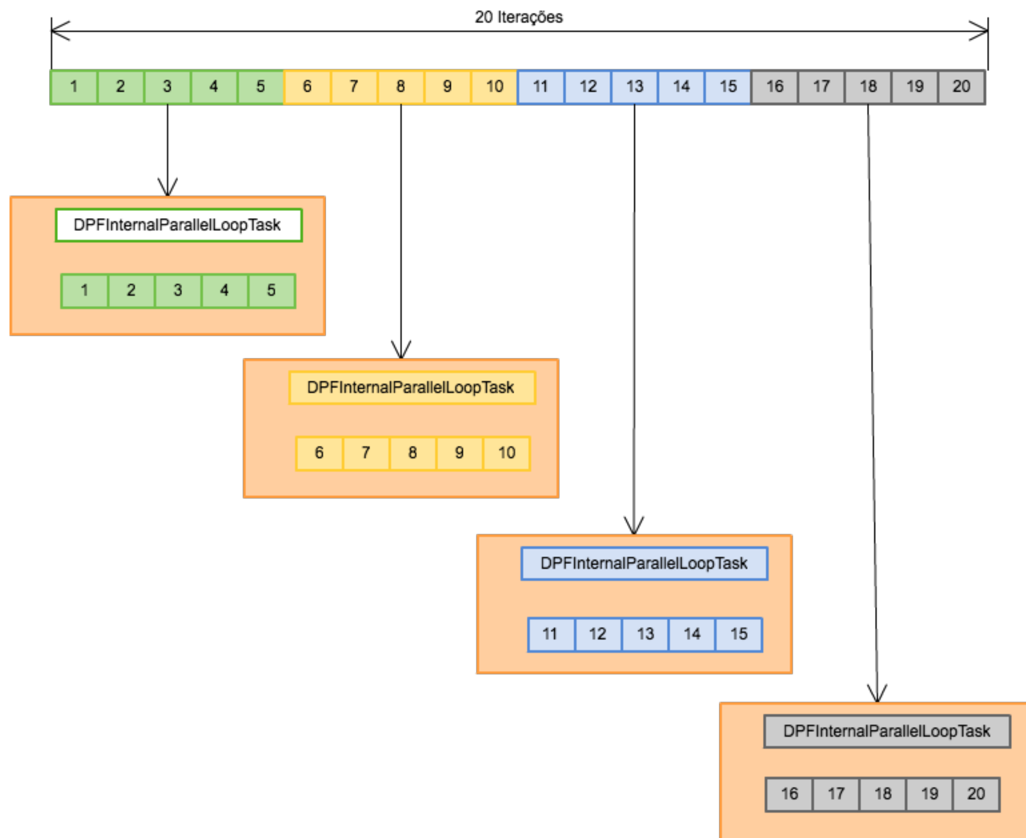

Figura 26 - Diagrama exemplo da criação de blocos de iterações

7.2.3 Execução de um ParallelForEach

Ao contrário do *ParallelFor*, o *ParallelForEach* não contém quaisquer índices, mas contem uma instância referente à iteração atual, que é representada pela variável `it` fornecida pela *framework*. A única diferença entre as duas implementações dos ciclos é mesmo este pormenor. Relativamente à criação de subtarefas, na Figura 27 é apresentado o diagrama de fluxo que representa todas as ações tomadas na implementação do método *exec* para a execução do ciclo *ParallelForEach*.

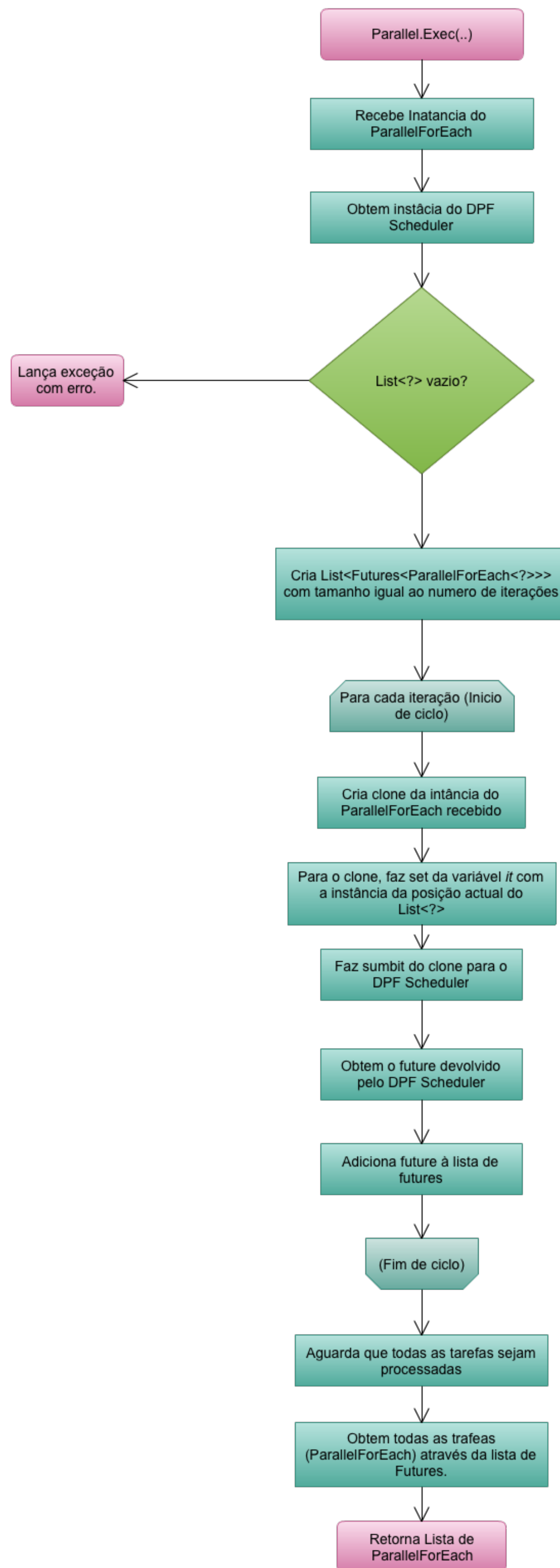

Figura 27 - Diagrama de fluxo da execução de um *ParallelForEach*

O método *exec,* neste caso, recebe apenas como parâmetro uma instância de um objeto do tipo *ParallelForEach*. O primeiro passo é dedicado a validações e o segundo passo corresponde à obtenção de uma instância do escalonador do sistema através do módulo *DPF System*. Após o passo anterior, é criada um objecto do tipo `List<Future<ParallelForEach<?>>>` com o mesmo número de iterações a serem executadas, tal como foi feito na execução do ciclo *ParallelFor*.

A divisão de trabalho, neste caso, como não é passado nenhum objecto do tipo *DPFPartitioner* significa que cada iteração corresponderá a uma tarefa. Após, é efectuado um ciclo para percorrer as iteração do `List<?>` que o programador pretende iterar. Para cada iteração é feito um clone da instância do *ParallellForEach* e é lhe atualizado o valor da variável `it` com a instância atual da iteração do ciclo. Realizado o passo anterior, a tarefa (o clone) é submetida para o escalonador do sistema e é devolvido como retorno desta ação uma instância de um objecto *Future* que é adicionada à lista de *Futures* anteriormente criada. Depois de todas as tarefas criadas e submetidas a *thread* atual bloqueia até que todas as tarefas sejam processadas.

Quando o processamento for totalmente concluído é devolvido um `List<ParallelForEach<?>>` para o programador, através do qual pode obter os resultados finais de cada tarefa e fazer a respectiva manipulação desses valores.

8 Comunicação entre Nós Cooperantes

Na área da computação distribuída, um dos principais aspectos a considerar é a forma como é composto o ambiente de computação e de que forma os nós que o compõem cooperam para executar um conjunto de operações para que determinado objetivo seja atingido.

A *DPF4j* também implica a existência de comunicações entre os nós cooperantes. O *DPF Daemon*, existente em cada nó, passa por três fases no seu ciclo de vida, sendo elas:

- Descoberta - Fase em que o nó arranca e se associa aos nós com *workgroups* em comum;
- Execução - Neste fase o nó encontra-se preparado para receber trabalho de outros nós e executa-lo;
- onde se encontra registado que se vai desligar.

Nas próximas secções são abordadas todas estas fases do ciclo de vida da *framework* assim como foi concebida a comunicação entre nós.

Relativamente a comunicações, há um factor muito importante a ter em conta neste tipos de aplicações/sistemas em que há transferência de dados entra várias máquinas, que é a segurança. A *DPF4j* tem alguns mecanismos para lidar com as questões de segurança.

8.1 Segurança

No que diz respeito a segurança, em todas as comunicações os dados que são considerados relevantes vão encriptados utilizando o algoritmo de encriptação *AES.* A chave de encriptação é definida na configuração de cada nó, relativamente a cada *workgroup*, ou seja, a chave não é distribuída por rede. Desta forma, só conseguem desencriptar os dados os nós do mesmo *workgroup*, ou seja, os nós que têm acesso à chave (*key*). É um mecanismo simples, mas é de notar que a *framework* está preparada para utilizar qualquer tipo de mecanismo de encriptação, sendo apenas necessário respeitar a *interface* especificada.

8.2 Descoberta

Nesta secção será apresentada a forma como a *framework* DPF4j descobre os outros nós cooperantes e quais as razões que levaram a seguir uma determinada solução/implementação.

8.2.1 Fases da Descoberta

O sistema de descoberta da *framework* é composto por três fases, sendo elas muito importantes para o funcionamento da *framework* no que diz respeito à distribuição, ou seja, se algo errado acontecer durante qualquer uma das fases pode comprometer o funcionamento desta, por exemplo, problemas de rede, etc.

Existem duas formas de utilização a *framework*, sendo elas:

- Utilizar apenas o *Daemon* para servir outros nós pertencentes aos mesmos *workgroups*;

- O *Daemon* pode arrancar automaticamente quando uma determinada aplicação for desenvolvida usando a *framework DPF4j*;

Ao longo da apresentação destas fases será apresentado um exemplo para que a percepção do funcionamento da *framework* seja entendido de maneira simples. De seguida são apresentadas as três fases que compõem o sistema de descoberta da *DPF4j*.

Para cada fase é explicado, tanto do lado do nó cliente, ou seja, o nó que está arrancar, como o nó servidor, neste caso qualquer nó já ligado ou em fase de arranque.

8.2.1.1 Procura

A fase da procura, tal como o próprio nome indica, consiste em procurar nós para trabalharem em conjunto. Tal como já foi definido antes, os nós agrupam-se por *workgroups*, logo cada nó procura nós que tenham *workgroups* em comum com ele.

O conceito da *DPF4j* tende em incluir mecanismo automatizados de forma a reduzir alguns processos extras por parte dos seus futuros utilizadores, como por exemplo definir a que nós se ligar, de forma estática. Na verdade, definir nós de forma estática é possível nesta *framework*, mas apenas para casos excepcionais. Voltando à questão dos mecanismos automatizados, estes são muito importantes no que diz respeito a esta fase. O principal objetivo é tornar transparente ao utilizador a forma como é feita a distribuição, ou seja, o utilizador não tem de se preocupar como é que vai ter de fazer para a sua aplicação distribuir trabalho com outros nós. A única preocupação que o utilizador deve ter para que o processamento da sua aplicação/sistema seja executado de forma distribuída é apenas garantir que existem nós homogéneos que sejam detectáveis pelo nó onde a sua aplicação está a ser executada.

No que diz respeito à comunicação, elemento essencial para que seja realizada uma interação seja por pessoas ou máquinas, foram analisados os vários tipos de comunicação possíveis para verificar quais é o que mais se adequa para este tipo de procura.

Existem vários tipos de comunicação, mas os mais conhecidos e os quais vão ser analisados são os seguintes:

- Um para todos:
- Um para alguns;
- Um para um;

O tipo um para todos, conhecido também como *broadcast*, consiste num determinado nó, neste caso, enviar uma mensagem para todos os nós que este consegue atingir.

De um para alguns, ou *multicast*, destina-se basicamente em enviar a mensagem apenas para um conjunto ou subgrupo de nós.

O tipo de comunicação um para um, ou ponto-a-ponto (*peer to peer*), consiste simplesmente em comunicar nó a nó, o que não tem grande vantagem num processo de descoberta, em que se parte do zero. Como não existe nenhum conhecimento à partida, logo este tipo de comunicação não é adequado para o que se pretende. No entanto, este tipo de comunicação é feito na fase de delegação de trabalho entre nós, o que será apresentado no respectivo capítulo.

Em seguida vamos apresentar a solução adoptada na implementação da *framework* para a fase da procura.

Num contexto de computação distribuída, partimos do princípio que um determinado nó está ligado a uma determinada rede, ou seja, podemos aproveitar as potencialidades da pilha *TCP/IP* para efetuar o mecanismo de descoberta da *framework*. Neste caso, a camada de rede e transporte tem um grande potencial para se conseguir construir o mecanismo de descoberta. Relativamente à camada de rede, podemos utilizar o *broadcast* ou *multicast* como tipo de comunicação. O *broadcast* consiste na comunicação de um para todos, cada um interpreta ou não a mensagem. Esta forma de comunicar é muito interessante para este caso, porque podem todos os outros nós pertencerem ao mesmo *workgroup*, ou apenas alguns, mas a mensagem é enviada para todos. A Figura 28 representa um exemplo do tipo de comunicação *broadcast*.

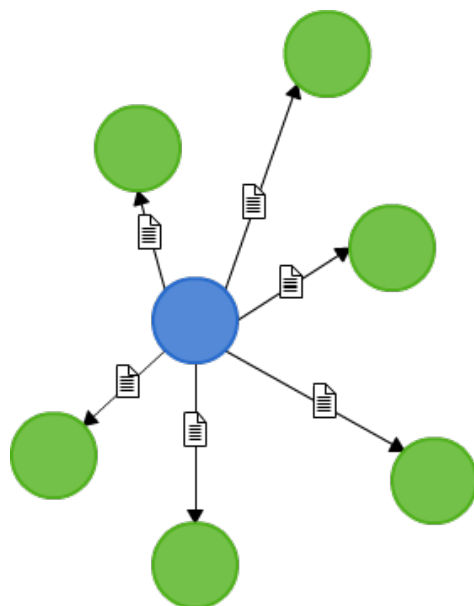

Figura 28 - Exemplo de *Broadcast*

No pior dos casos, alguns do nós podem não ser do mesmo *workgroup*, então nesse caso podem simplesmente ignorar a mensagem.

Relativamente ao tipo de comunicação *multicast*, existe a possibilidades para além da comunicação ser para todos, tal como no *broadcast*, esta pode ser feita de um para alguns, dai o multicast ser também conhecido por *broadcast* multiplexado. A Figura 29 seguinte apresenta um exemplo de comunicação *multicast*.

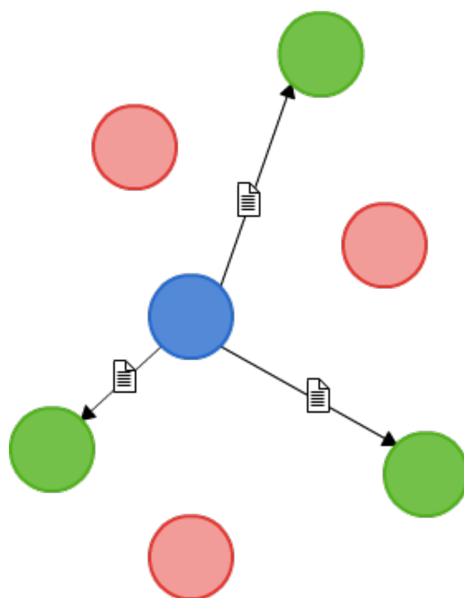

Figura 29 - Exemplo de *Multicast*

Visto que uma rede pode ter vários computadores a executar aplicações de diferentes contextos e diferentes necessidades, poderá não existir a necessidade de enviar a mensagem de procura para todos os computadores. Dado isto, foi optado o modelo de comunicação broadcast multiplexado para implementação da fase de procura, que permite também enviar

mensagem via broadcast, mais conhecido como *multicast ip*, onde o *IP* definido fornece diferentes gamas de nós que irão receber o datagrama. No entanto, o *multicast* utiliza a comunicação *UDP* da camada de transporte, em vez de *TCP*. Dado isto, o *UDP* tem várias desvantagens relativamente ao *TCP*, sendo elas por exemplo:

- Não orientado à ligação;
- Os pacotes são enviados individualmente;
- Não há garantia que os pacotes (dados) são entregues.
- etc.

Para compensar estas desvantagens da utilização das comunicações *UDP* foi criado um mecanismo que consiste no reenvio da mensagem *UDP.* Este mecanismo é totalmente configurável, sendo possível definir o número de vezes a reenviar o datagrama e o intervalo de tempo. Com este mecanismo não é garantido que o datagrama *UDP* seja entregue, mas a probabilidade de o ser aumenta conforme maior for o número de tentativas definido.

Relativamente aos dados que são enviados no datagrama *UDP*, quando o novo nó arranca, é enviado um objeto *Java* do tipo *DiscoveryRequest*. A Tabela 10 contém a definição do objeto.

Nome	Descrição
workgroups	**Lista de workgroups**
nodeId	**Id (único) do nó**
ip	**IP da máquina onde o nó está a ser executado.**
port	**Porta onde o serviço RMI está a ser executado.**
serviceName	**Nome do serviço para fazer o lookup do RMI.**

Tabela 10 - Atributos do objeto *DiscoverRequest*

Todas os dados enviados são obrigatórios para que a descoberta seja efectuada com sucesso.

A lista de *workgroups* é utilizada pelo nó que recebe o pedido para validar quais são os nós que tem em comum com o nó que enviou a mensagem via *multicast*. Todos os outros dados, são utilizados pela *framework* para obter o *stub RMI*, e depois fazer a chamada remota do método que faz a associação. Parte dos dados são utilizados na segunda fase do processo de descoberta, a associação, que é descrita na secção seguinte. Esta é a informação necessária para que se inicie a fase de descoberta da *framework DPF4j*.

O comportamento do *Daemon* quando recebe um *DiscocerRequest* via *multicast* é apresentado no diagrama apresentado de seguida (Figura 30).

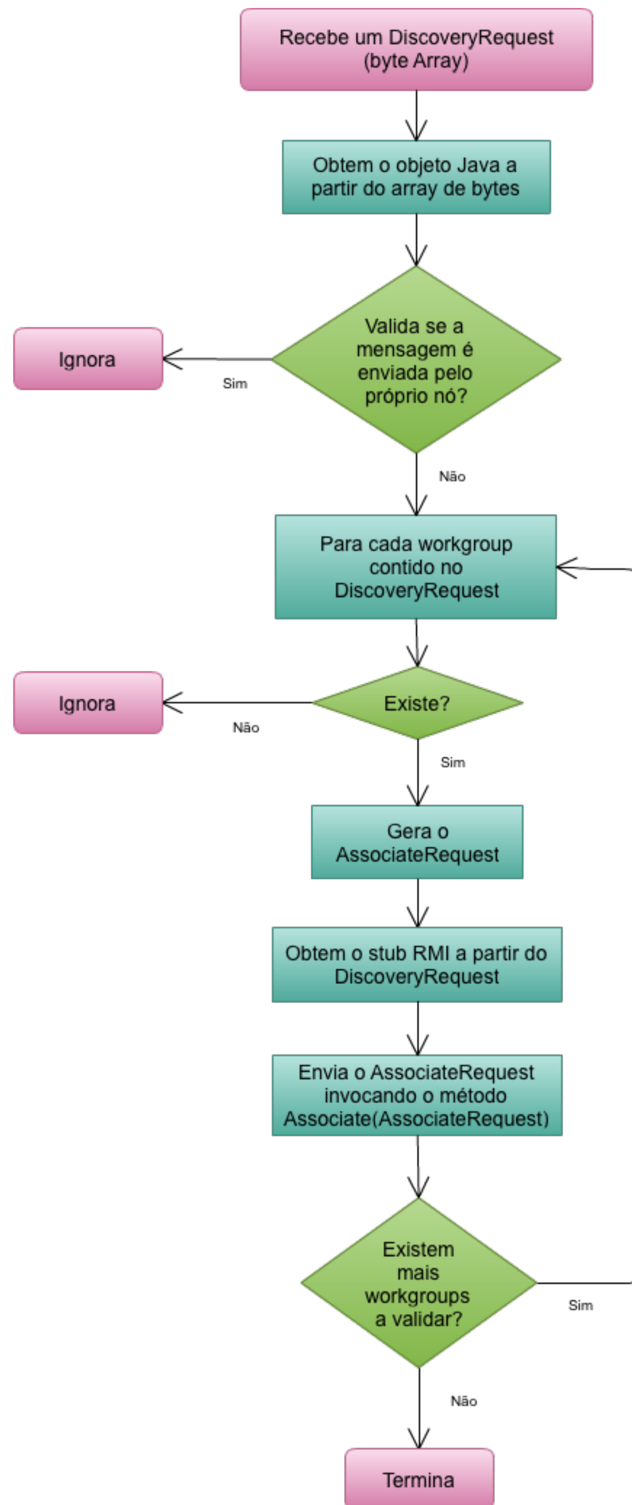

Figura 30 - Fluxo do tratamento de um *DiscoveryRequest*

Os dados são recebidos num *array* de *bytes*, sendo depois convertidos num objeto do tipo *DiscoveryRequest*. Depois de o array de *bytes* ser transformado, é feita a validação se o pedido é do próprio nó, por exemplo no *broadcast* a própria mensagem é recebida pelo emissor. Caso se verifique isto, o pedido é descartado, caso contrário são obtidos todos os *workgroups* que os nós têm em comum. Para cada *workgroup* em comum, é obtido o *stub RMI* para ser feita a invocação do método *associate* (*AssociateRequest*) em que é feita a associação.

8.2.1.2 Exemplo

Partindo do princípio que existem numa rede três *workgroups*: A, B e C. Por exemplo, o *workgroup* A está a executar um processo moroso e pesado o que levou à necessidade de se adicionar mais um recurso físico (nó) ao *workgroup* A. Para isto é necessário arrancar o *daemon* em modo *stand-alone* com as respectivas configurações. No momento do seu arranque é enviada a mensagem (*DiscoveryRequest*) via *multicast* para o ip "*224.0.0.1*". Este IP faz com que a mensagem seja enviada para todos os nós que se encontram no segmento de rede [15]. Tal como se pode visualizar na Figura 31, a mensagem é enviada para todos os nós, incluindo o próprio nó.

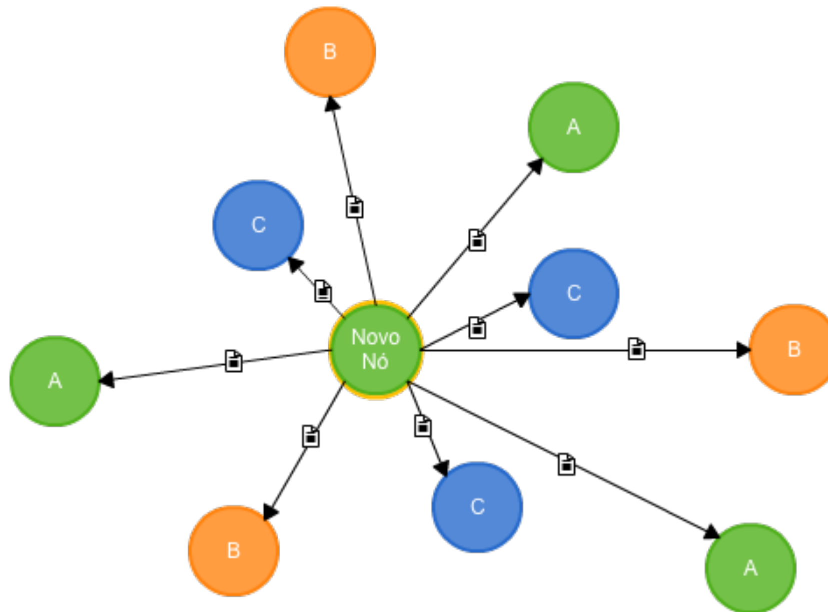

Figura 31 - Exemplo da fase da descoberta - Procura

Enviada a mensagem, o nó aguarda que os nós que pertençam ao mesmo *workgroup*, neste caso o A, façam a respectiva associação.

8.2.1.3 Associação

Na segunda fase do sistema de descoberta, a associação, consiste basicamente nos nós que receberam o datagrama *UDP* na fase da procura em fazer a associação com o nó que acabou de iniciar, ou seja, todos nós que fizeram a associação pertencem a pelo menos um dos *workgroups* a que o nó pertence. O objeto *AssociateRequest* que é utilizado nesta fase é composto pelo *workgroup* em questão e pelos dados necessários para obter o *stub RMI* do nó que fez a associação. Neste pedido também é enviada uma lista dos nós cujo *workgroup* é comum com o novo nó. Desta forma, é possível conhecer nós que não são descobertos pelo método local (*multicast*), por exemplo, se existir um conjunto de nós pertencentes ao mesmo grupo do novo nó e estes conhecerem um nó na Internet, o novo nó automaticamente fica-o a conhecer. Após obter a lista de nós, é feita uma iteração sobre essa lista e para cada nó é gerado o *stub RMI* e invocado o método remoto da associação. Neste caso, se o novo nó já conhecer nós cujo *workgroup* é comum com o nó em que se vais fazer a associação, estes são enviados no pedido.

Antes de se passar à fase de registo, o nó que recebeu o pedido de associação faz o respectivo registo do nó remoto para posteriormente poder delegar trabalho nesse mesmo nó. O registo do nó é feito no módulo *DPF Registry* da *framework*, que por sua vez informa o *DPF Scheduler* que foi registado um novo nó, ficando assim com mais um nó disponível para delegar trabalho. Após este processo, o próximo passo é o registo no nó que fez o pedido de associação. Utilizando os dados referentes ao serviço *RMI* que foram recebidos através do objeto *AssociateRequest*, é obtido o *stub* RMI e é feito o pedido de registo no nó remoto.

O diagrama apresentado na Figura 32 apresenta o fluxo executado pelo nó que recebe o *associateRequest*.

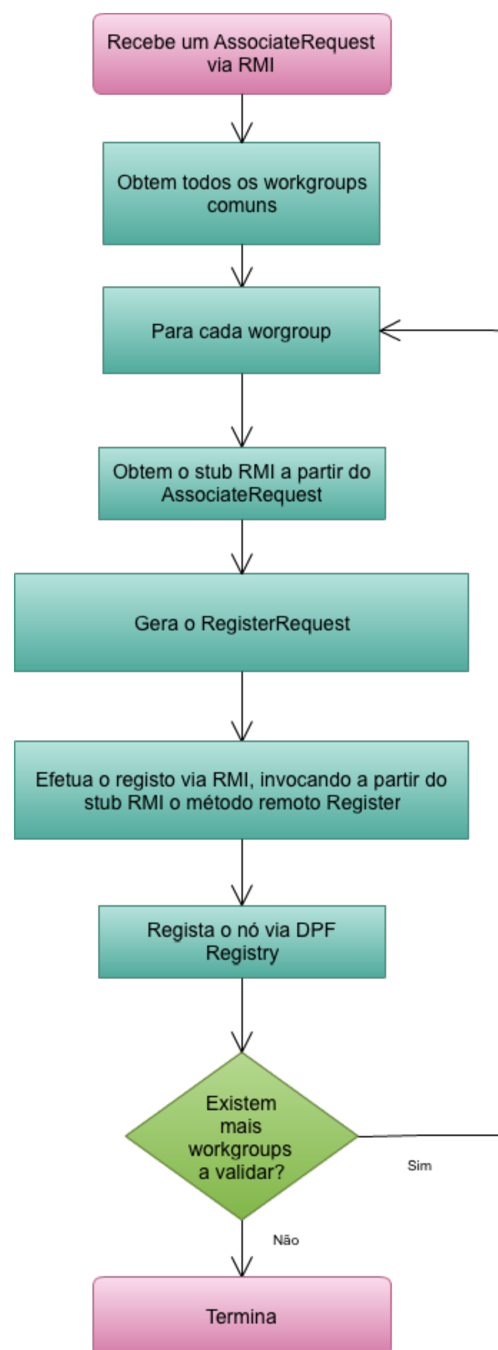

Figura 32 - Fluxo do tratamento de um *AssociateRequest*

Continuando o exemplo anterior, onde foi feito o envia da mensagem via *multicast*, neste momento partimos do princípio de que os nós que já se encontravam no *workgroup* A receberam a mensagem *DiscoveryRequest*, fizeram as respectivas validações e fizeram o respectivo associate utilizando o *stub RMI*. Na Figura 33 podemos verificar que apenas os nós do mesmo *workgroup* é que fazem a associação.

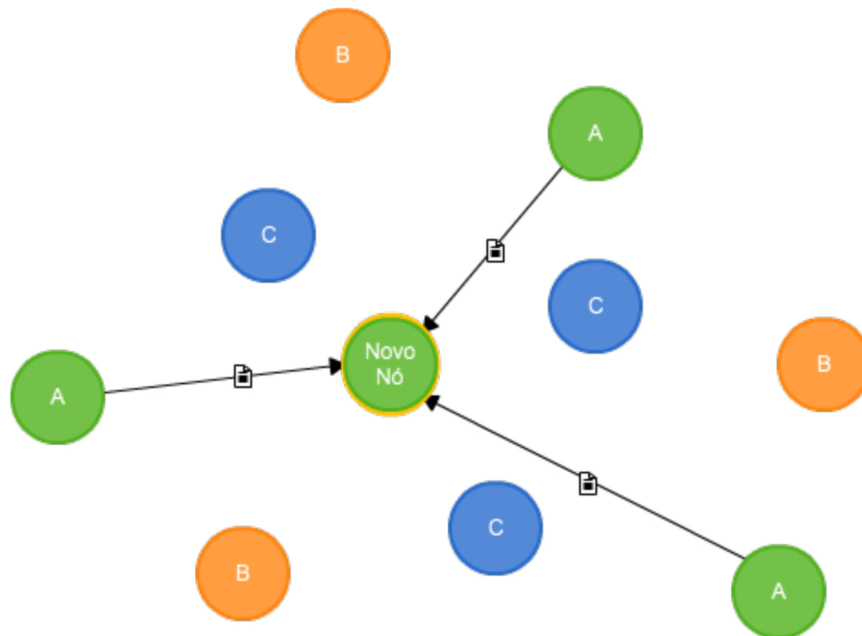

Figura 33 - Exemplo da fase de descoberta - Associação

Depois da associação ser efectuada apenas falta o processo de registo, que é explicado e exemplificado no capítulo seguinte.

8.2.1.4 Registo

Esta é a ultima fase do processo de descoberta e consiste no nó que acaba de iniciar, em fazer o registo nos nós que lhe fizeram a associação. Os nós que recebem o pedido de registo realizam o mesmo processo de registo que o nó que iniciou, quando recebeu os pedidos de associação, utilizando o *DPF Registry* para registar o nó.

Por fim, para terminar o exemplo que foi feito nas secções anteriores, de seguida é apresentado o exemplo da presente fase.

Visto que os nós que estão na rede e que pertencem ao mesmo *workgroup* do nó que se ligou, ou seja, o *workgroup* A são aqueles que fizeram a associação. Para cada pedido de associação recebido é feito o registo no respectivo nó. Na Figura 34 é apresentado um diagrama que representa o nó a enviar o pedido de registo aos nós remotos.

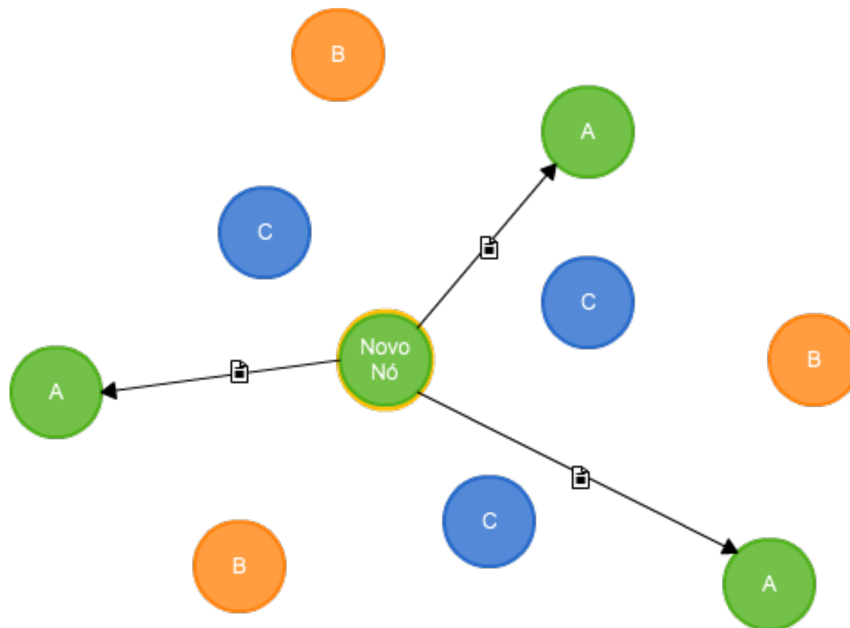

Figura 34 - Exemplo da fase de descoberta - Registo

Desta forma, depois da fase de registo estar concluída, este nó pode começar a delegar trabalho nos nós remotos, assim como os remotos podem delegar trabalho nele.

No diagrama da Figura 35 é apresentado um diagrama de sequência que engloba todo o processo de descoberta, desde a fase de descoberta até a fase de registo. Quando o nó representado do lado esquerdo do diagrama inicia, este envia o datagrama *UDP* via *multicast* para todos os nós que se encontram na rede. Como se pode verificar, apenas três fizeram a respetiva associação. Por fim, o nó requerente termina o processo, neste caso, com o registo nesses mesmos nós que fizeram a associação.

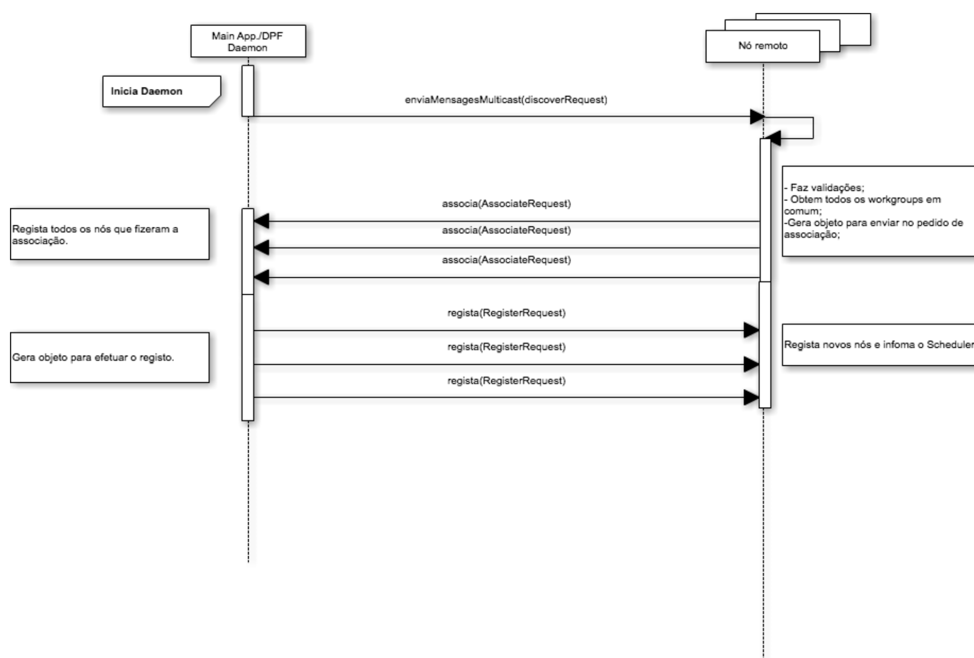

Figura 35 - Diagrama de sequência de todo o processo de descoberta

8.2.2 Modos de descoberta

O sistema de descoberta é composto por uma vertente automática e outra manual. Nas seguintes secções são abordados estes dois modos.

8.2.2.1 Modo Automático

O modo automático consiste basicamente na execução das três fases apresentadas na secção anterior. Tal como o próprio nome indica trata-se de um processo autónomo, a *framework* apenas necessita carregar toda a configuração relativa ao processo de descoberta e realizar as operações necessárias para descobrir novos nós e começar a interagir e cooperar com estes.

Este processo é despoletado pelo *daemon* no seu arranque, sendo enviado um datagrama *UDP* para todos os nós da rede em que o nó se encontra ligado, tal como explicado na secção da procura. Este modo é adequado para o uso da *framework* numa intranet por exemplo.

8.2.2.2 Modo Manual

O modo manual da *DPF4j* também é despoletado no seu arranque tal como no modo automático. Este modo basicamente apenas executa a ultima fase do processo de descoberta, o registo, sendo adequado para configurar nós que se encontram na Internet, por exemplo nós que se encontram na *cloud*. Para este modo ser executado apenas é necessário que o utilizador da *framework* configure os respectivos *IPs* dos nós que se encontram fora da rede intranet. É de notar que o modo manual não se restringe apenas a nós fora da rede que se encontra o nó, sendo possível também configurar nós específicos intranet.

8.2.3 Configuração

Neste capítulo são apresentadas as configurações necessárias para cada modo, e mais algumas configurações extras.

Relativamente ao modo automático apenas é necessário configurar os seguintes aspectos (Tabela 11):

Propriedade	Descrição
dpf.workgroups	**Lista de workgroups a que o nó pertence.**
dpf.workgroups.multicast.ip	**IP multicast para onde serão enviadas os datagramas.**
dpf.workgroups.remote.ports	**Lista de portas para quais devem ser enviadas os datagramas UDP.**
dpf.workgroups.multicast.port	**Porta em que o nó está à escuta para receber datagramas UDP.**
dpf.workgroups.bad.ips	**Lista de IPs a rejeitar.**

Tabela 11 - Configuração do processo de descoberta

Existe ainda a possibilidade de evitar ligações de vários nós, definindo uma lista de ips que serão ignorados na fase de procura. Para configurar esta funcionalidade da *DPF4j* é necessário colocar a *key* indicada na tabela acima no ficheiro de propriedades com a lista dos respectivos *IPs* a ignorar.

No que diz respeito ao modo manual apenas existe uma configuração que permite definir uma lista de ips de nós referentes a um determinado *workgroup*. Para efetuar essa configuração é necessário utilizar a key "`dpf.workgroup.{WorkGroupName}.nodes`".

Em seguida é apresentado um exemplo desta configuração:

```
dpf.workgroups=workgroup
dpf.workgroups.broadcast.remote.ports=17763,3213,1314,553
dpf.workgroups.multicast.ip=224.0.0.1
dpf.workgroups.multicast.port=17763
dpf.workgroups.bad.ips=192.168.158.35,192.168.158.36
dpf.workgroup.workgroup.nodes=10.0.4.10,10.20.36.47
```

8.3 Execução

Esta é a segunda fase do ciclo de vida do nó da *framework DPF4j*. Esta fase consiste basicamente em esperar que lhe seja delegado trabalho pelos outros nós que pertencem aos mesmos *workgroups*.

O nó que delega trabalho nos outros nós basicamente utiliza o respectivo *stub RMI* para invocar o método remoto *Execute*. Este método recebe como parâmetro um objecto do tipo *DPFPacket* que representa os dados que passam por rede. O *DPFPacket* é composto por três atributos: *workgroup*, o *ID* do nó que delegou trabalho e um *array* de *bytes* que representa a *DPFTask* encriptada. Por motivos de segurança, a *Task* que é representada pelo objeto *DPFTask* é encriptada com a chave do *workgroup* comum entre nós. Desta forma, caso algum nó receba uma *task*, ou mesmo intercepte uma tarefa por rede (*man-in-the-middle*), é garantido que esta apenas é decifrada usando a respectiva chave, caso contrário é despoletado um erro.

Quando o nó remoto recebe um pedido de execução, o primeiro passo é a obtenção do objeto *DPFTask*, que é decifrada utilizando a chave partilhada e posteriormente é feita a execução da respectiva *task*, caso a decifragem seja efectuada com sucesso.

Nesta fase também são invocados os métodos que fornecem as dependências necessárias para que o nó remoto seja capaz de executar a tarefa delegada.

8.4 Desassociação

A desassociação é o ultima fase do ciclo de vida de um nó. Quando um nó se encontra nesta fase do seu ciclo de vida significa que se está a desligar. A ação que o nó toma nesta situação é informar todos os nós em que se encontra registado para fazer o *unregister*. O *unregister* é invocado via *RMI* utilizando o *stub RMI* do nó em que está registado, sendo isto feito para todos os nós.

Relativamente ao nós nos quais este está registado, ao receberam o pedido de *unregister* validam se realmente o nó está registado e em seguida fazem o *unregister* informando o *DPFRegistry* que o nó em questão já não se encontra disponível para processar trabalho.

Este passo é muito importante no que diz respeito à eficiência do sistema, evitando existirem nós no *DPF Registry* associados a um determinado nó quando ele já não está ligado. Isto pode provocar um atraso no tempo total da execução de uma tarefa cujo processamento é feito de forma fragmentada. Caso esta fase não existisse permitia que os nós ficassem sempre com uma referência para os nós que se registaram e caso alguns se desligassem entretanto podiam surgir situações em que o nó tentava delegar tarefas nesses nós, mas como eles se encontram desligados iria surgir uma situação de *timeout* no que diz respeito a comunicações. A vantagem desta fase é o facto de não existir a possibilidade de ocorrerem estas situações.

9 Resultados

Neste capítulo são apresentados alguns dos testes que foram efetuados no final do desenvolvimento da *framework DPF4j*. Visto que com a utilização da *framework DPF4j*, o processamento das aplicações pode ser paralelizado e distribuído é importante ter em conta vários aspetos, tais como: a quantidade de dados a passar pela rede entre nós, tempo de arranque de um determinado nó até que este esteja disponível para executar trabalho, etc.

Foram realizados dois testes, cada um com objetivos diferentes, sendo eles:

A multiplicação de matrizes utilizando as *frameworks Java*, *Ateji PX*, *DPF4j* modo Paralelo e Distribuído de forma a obter os tempos de execução, quantidades de dados que são trocados pelos nós durante todos o processo;

Resolução de *Sudokus* de tamanho 9 por 9, com objetivo de fazer testes de processamento intensivo.

9.1 Caso 1 – Multiplicação de Matrizes

9.1.1 Objetivo e Explicação do Algoritmo

Este teste utiliza matrizes de várias dimensões para obrigar à utilização intensiva da rede, também é garantido que as máquinas remotas não têm as classes necessárias para a execução com a intenção de agravar este problema.

Neste teste foram executadas multiplicações em três *frameworks* diferentes, sendo elas: *Java*, *Ateji PX*, *DPF4j* (modo Paralelo) e *DPF4j* (modelo Distribuído).

No caso do *Java*, o processamento do cálculo da multiplicação das matrizes é feito de forma sequencial. Relativamente ao *Ateji PX*, o teste foi feito em modo paralelo apenas, porque o código sequencial não sofreria qualquer conversão e ficaria igual ao teste *Java standard*. Por fim, foram realizados testes utilizando a *framework DPF4j*, tanto em modo paralelo como em modo distribuído.

O algoritmo utilizado neste teste tem como objetivo o cálculo da multiplicação de matrizes e cada *Task* (*executor*) é responsável por calcular o resultado de uma linha.

No caso sequencial é utilizado um ciclo for para percorrer as linhas da matriz, e nos casos paralelos e distribuídos são utilizados os ciclos for paralelos das respetivas *frameworks*. No

caso da *framework DPF4j*, a multiplicação de matrizes foi feita recorrendo a um *ParallelFor* sem blocos que dará origem a um número de *tasks* igual ao número de linhas da matriz.

9.1.2 Dados

Todos os dados utilizados na construção das matrizes são gerados de forma aleatória com valores entre 0 e 1000. Os dados de entrada estão embrulhados num *DPFReusableObject* e são compostos por um objeto que encapsula duas matrizes de tamanho variável.

9.1.3 Ambiente de Testes

9.1.3.1 Máquinas

Neste caso de teste foram utilizadas as máquinas apresentadas na Tabela 12.

Máquina	CPU	Cores	HyperThreading (CPUs lógicos)	RAM
i7-1	Intel Core i7 1.6Ghz	4	Sim (8)	4GB 1333MHz
i5	Intel Core i5 2.4Ghz	2	Sim (4)	8GB 1333MHz
C2D-1	Intel Core2Duo 2.53 Ghz	2	-	4GB 1067 MHz
C2D-2	Intel Core2Duo 2.4 Ghz	2	-	2GB 1067 MHz

Tabela 12 - Caso de teste 1 - Listagem de máquinas

9.1.3.2 Tecnologias

No presente teste foram utilizadas três *frameworks*: Java (sequencial), *AteJi PX* e por fim, a framework *DPF4j* nos dois modos, paralelo e distribuído.

9.1.3.3 Pressupostos

A nível de infraestrutura de rede, todas as ligações entre máquinas são efectuadas via WI-FI.

A *framework DPF4j* executou sem um ciclo *warm-up*, isto é, os resultados dos testes contabilizam todas as execuções, incluindo a primeira. Isto tem as seguintes consequências nos testes que envolvem a *DPF4j*:

- os tempos incluem o arranque da *framework*, isto incluí arranque do *daemon* e dos serviços;
- o tempo do processo de descoberta está contabilizado;
- os processos de descoberta e arranque consomem recursos que tornam a execução global mais lenta;
- os processos de arranque e descoberta são paralelos à execução da aplicação principal por isso as primeiras execuções executam num modo paralelo, não distribuído, devido ao processo de descoberta ainda estar a encontrar os nós remotos;

- os testes *DPF4j* paralelos foram executados com os recursos para distribuição ativados mas sem máquinas remotas a aceitar pedidos.

As classes relativas às tarefas e aos dados não se encontram no nó remoto, sendo necessário que o nó remoto peça estas dependências ao nó que está a delegar trabalho, sendo estas enviadas por rede.

9.1.4 Resultados

Visto que foram utilizadas várias *frameworks* para efetuar os testes irão ser comparados os valores obtidos entre estas, e os resultados serão apresentados orientados à maquina que efetuou os testes. No final são apresentados os valores obtidos para a *framework DPF4j* no modo distribuído.

Nas figuras (Figura 36, Figura 37 e Figura 38) encontram-se os gráficos que demonstram os valores obtidos nos testes executados nas várias máquinas utilizadas para os testes.

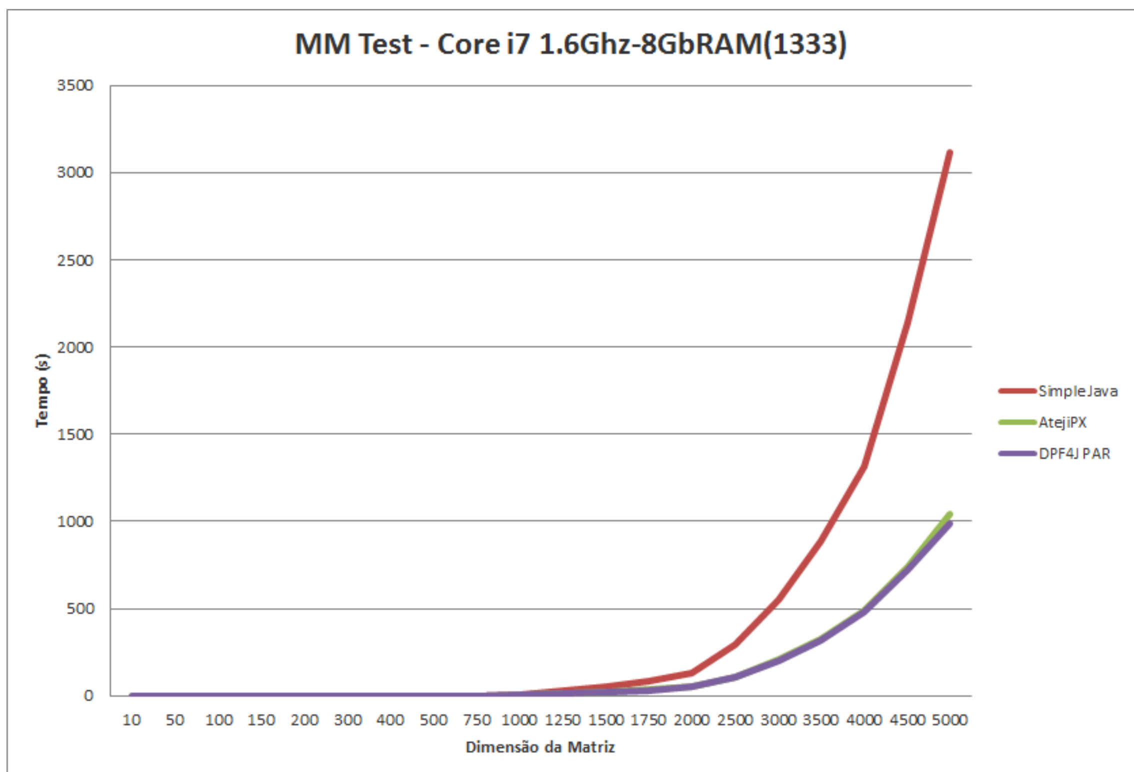

Figura 36 - Caso de teste 1 - Gráficos de resultados referentes à máquina i7-1

Os dados referentes ao gráfico apresentado na Figura 36 encontram-se na Tabela 13

Matrix Size	Simple Java	AtejiPX	DPF4J PAR
10	0.000	0.014	0.012
50	0.009	0.022	0.026
100	0.020	0.034	0.042
150	0.040	0.037	0.045
200	0.041	0.046	0.053
300	0.111	0.078	0.079
400	0.261	0.142	0.133
500	0.506	0.240	0.233
750	2.161	1.209	1.314
1000	9.320	3.914	4.258
1250	28.272	10.798	10.567
1500	50.745	19.563	20.043
1750	88.169	33.970	33.165
2000	130.175	50.075	49.501
2500	297.813	110.453	108.705
3000	552.237	205.727	198.025
3500	890.478	323.620	319.749
4000	1315.911	486.866	481.112
4500	2137.890	735.463	723.344
5000	3111.734	1040.401	989.413

Tabela 13 - Tabela de resultados obtidos pela máquina i7-1 para o caso de teste 1

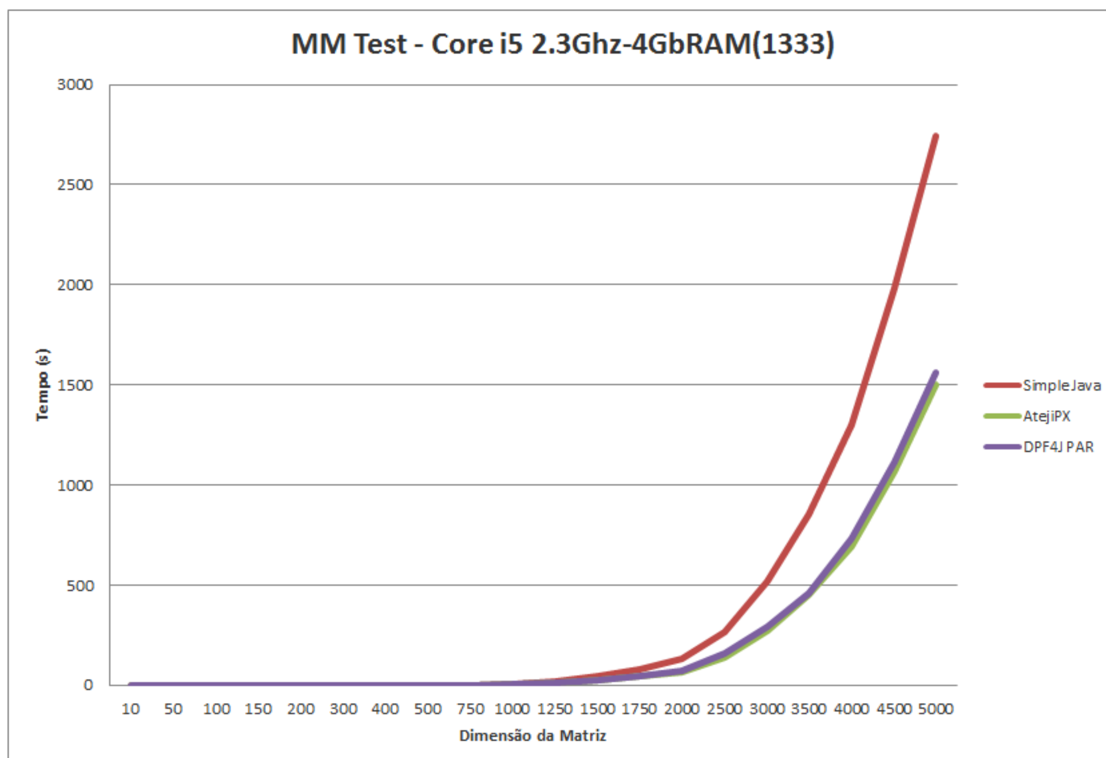

Figura 37 - Caso de teste 1 - Gráfico de resultados referente à máquina i5

O gráfico da Figura 38 mostra os tempos médios obtidos utilizando a máquina C2D-1 nos testes das *frameworks Java*, *Ateji Px* e *DPF4j* paralelo e distribuído. O teste distribuído foi feito em conjunto com a máquina i5. Na Tabela 14 são apresentadas as médias dos valores obtidos

para a máquina i5. Através dos valores da tabela é possível ter uma melhor percepção dos resultados obtidos.

Matrix Size	Simple Java	Ateji PX	DPF4J PAR
10	0.000	0.014	0.013
50	0.005	0.020	0.030
100	0.015	0.042	0.061
150	0.031	0.058	0.060
200	0.037	0.057	0.066
300	0.079	0.095	0.095
400	0.166	0.132	0.144
500	0.324	0.221	0.245
750	1.788	1.536	1.700
1000	8.750	4.746	5.139
1250	20.812	14.233	14.855
1500	48.837	25.967	27.847
1750	78.385	42.628	48.841
2000	129.441	68.795	71.822
2500	267.036	136.884	158.960
3000	521.568	272.953	294.193
3500	852.434	452.715	458.328
4000	1301.523	690.162	733.928
4500	1985.192	1065.407	1111.811
5000	2741.224	1504.724	1562.614

Tabela 14 - Tabela de resultados obtidos pela máquina i5 para o caso de teste 1

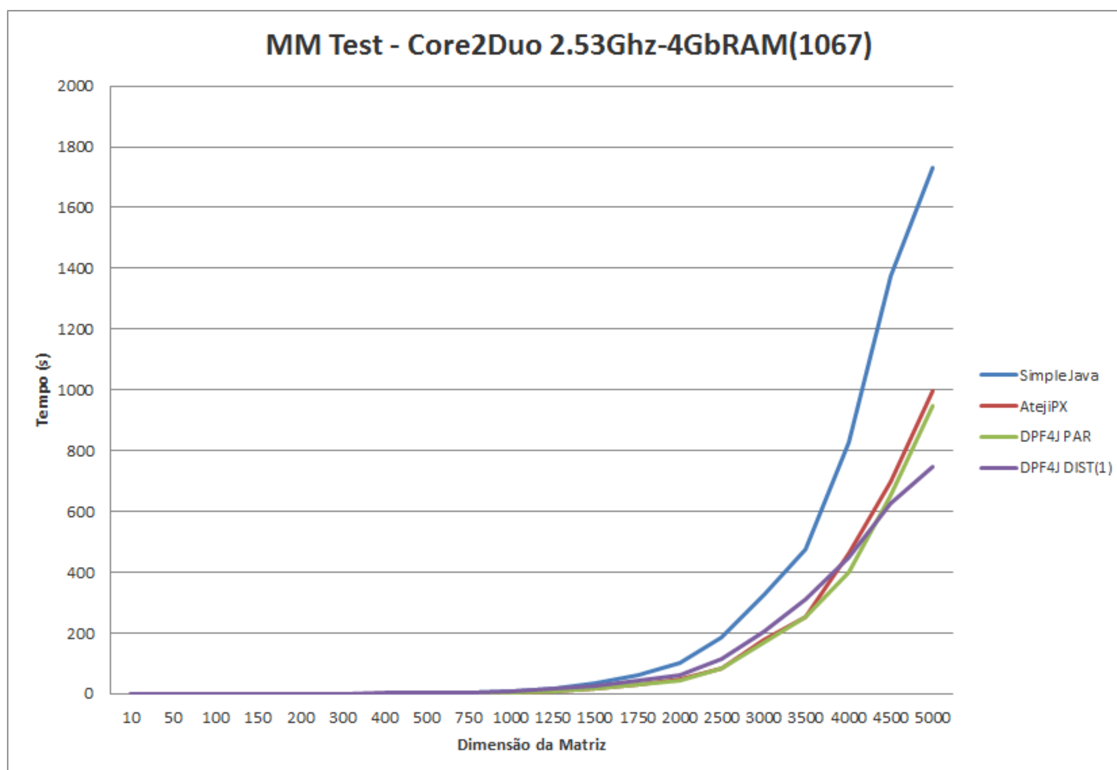

Figura 38 - Caso de teste 1 - Gráfico de resultados referente à máquina C2D-1

Na Tabela 15 são apresentados os valores referentes ao gráfico apresentado na Figura 38

Matrix Size	Simple Java	AtejiPX	DPF4J PAR	DPF4J DIST
10	0.000	0.018	0.018	0.030
50	0.008	0.026	0.026	0.061
100	0.021	0.053	0.053	0.083
150	0.042	0.068	0.068	0.184
200	0.061	0.079	0.079	0.679
300	0.150	0.146	0.146	0.615
400	0.345	0.267	0.267	1.962
500	0.737	0.457	0.457	2.357
750	2.857	1.599	1.877	4.359
1000	8.076	4.802	5.348	7.866
1250	19.057	9.962	9.931	17.284
1500	36.209	17.690	17.584	25.507
1750	62.673	32.392	29.659	43.685
2000	101.874	46.777	44.872	62.508
2500	187.500	83.272	84.689	115.151
3000	322.007	176.890	168.154	203.273
3500	476.335	253.476	251.906	312.254
4000	825.624	462.220	401.352	446.665
4500	1374.844	698.846	654.063	625.059
5000	1731.687	996.095	949.069	747.621

Tabela 15 - Tabela de resultados obtido na máquina C2D-1 para o caso de teste 1

Como se pode verificar os valores são semelhantes até à dimensão da matriz tomar o valor de 1000. Quando a dimensão da matriz excede o tamanho 1000 verifica-se um aumento

substancial do tempo de execuções realizadas usando a tecnologia *Java* sequencial. Relativamente à execução paralela, *Ateji PX* e *DPF4j* modo paralelo, os valores aumentam como é normal, mas a diferença relativamente ao *Java* sequencial é considerável como se pode verificar. Relativamente ao teste utilizando a *framework DPF4j* em modo distribuído, os valores a partir da dimensão 1000 da matriz são inferiores aos valores do *Java* sequencial, mas superiores aos valores obtidos para o *Ateji PX* e *DPF4j* modo paralelo. No entanto, este cenário mantem-se apenas até ao momento em que a matriz passa a ter dimensão de aproximadamente 4500. Quando este valor é ultrapassado verifica-se que os tempos obtidos para o *DPF4j* distribuído passam a ser menores que todas as outras tecnologias.

Os piores tempos que a *DPF4j* apresentou relativamente às tecnologias paralelas em matrizes de dimensão inferior a 4500 deve-se ao tempo de transferência de dados ser mais elevado que o tempo de processamento. A maior parte do tempo perdido está no envio dos dados de entrada (matriz) que é efectuado apenas uma vez, e enquanto não terminar bloqueia todas as tarefas que envolvam esses dados na máquina remota. Quer sejam criadas 1000 *tasks* ou 5000 devido ao uso do *DPFReusableObject* esta matriz só será enviada uma vez, no entanto o tempo de processamento de 5000 *tasks* será muito mais elevado, não só pela quantidade de *tasks* mas também pela dimensão dos dados processados (tamanho da matriz).

A tarefa submetida é do tipo *ParallelFor* e a instância em questão contém todos os atributos do *ParallelFor* mais o *DPFReusableObject* representante das duas matrizes. Dado isto, quando a tarefa é serializada para ser enviada por rede o seu tamanho é de 544 *bytes*. Esta tarefa é enviada uma vez por cada iteração executada remotamente, ainda que a única alteração seja o número da iteração, mas a *framework* não consegue detetar isso visto que a implementação é do programador.

Devido à utilização de *DPFReusableObjects* a matriz de dados só é transferida uma vez e o seu tamanho varia com a dimensão (ver Tabela 16)

Finalmente a resposta de cada tarefa, inclui o resultado da multiplicação e esta é representada por um objecto de tamanho variável (Tabela 16) em que ‘N’ equivale ao número de tarefas executadas remotamente).

Tamanho	Input - Transf. 1 vezes (bytes)	Output - Transf. N vezes (bytes)
10	1.136	624
50	21.136	784
100	82.128	992
150	183.136	1.184
200	324.128	1.392
300	726.128	1.792
400	1.288.128	2.192
500	2.010.128	2.592
750	4.515.136	3.584
1000	8.020.128	4.592
1250	12.525.136	5.584
1500	18.030.128	6.592
1750	24.535.136	7.584
2000	32.040.128	8.592
2500	50.050.128	10.592
3000	72.060.128	12.592
3500	98.070.128	14.592
4000	128.080.128	16.592
4500	162.090.128	18.592
5000	200.100.128	20.592

Tabela 16 - Resultados relativos à quantidade de dados transferidos

9.1.5 Conclusões

Através dos resultados demonstrados pode-se concluir que no que diz respeito ao processamento paralelo os valores obtidos tanto pela *framework DPF4j* como pelo *Ateji PX*, começam a ser inferiores a partir do momento em que a matriz toma como valor da sua dimensão o valor 300. Até a matriz tomar esse valor, os valores obtidos pelo *Java*, *Ateji PX* e *DPF4j* são muitos aproximados. Isto deve-se ao facto das *frameworks* que permitem a programação paralela, como é o caso do *Ateji PX* e do *DPF4j*, (o *Java* também o permite, mas neste caso o teste foi realizado no modo sequencial), terem um *atraso* relativamente às linguagens de programação sequenciais. Este *atraso* está relacionado com o facto de as linguagens terem um esforço extra para criar novas threads com o objetivo de paralelizar o processamento. Em algumas situações, como por exemplo para matrizes de tamanho reduzido, não é vantajoso utilizar processamento paralelo visto que o tempo de execução em modo sequencial é talvez inferior ou igual ao tempo que o processamento paralelo demora a criar as tarefas/*threads*. Este *atraso* começa a ser desprezável a partir de determinado tamanho da matriz, devido ao tempo de processamento da tarefa ser mais elevado.

Relativamente ao *Ateji PX* e *DPF4j* (modo paralelo), surpreendentemente estes testes demonstram que mesmo só fazendo paralelismo a *DPF4j* tem tempos muito semelhantes ao *Ateji PX* e em alguns casos consegue ser mais eficiente que o *Ateji PX*, este não era um resultado esperado devido ao peso da infraestrutura da *DPF4j* (escalonador, serviços, etc.) em relação ao *Ateji PX* que é apenas código gerado dedicado ao paralelismo.

No que toca à comparação destes com o *DPF4j* em modo distribuído, vemos que apesar de teoricamente o poder de processamento ter sido aumentado com mais máquinas, o desempenho diminuiu. Isto deve-se aos tempos relativos à transferência de dados serem muito elevados, criando *atrasos* de (até) vários segundos nas primeiras tarefas (visto que todas as tarefas de uma determinada máquina esperam pelo *download* do *DPFReusableObject*).

Este teste também demonstrou que quando existe uma grande quantidade de dados em memória terá de haver um peso de processamento muito elevado para fazer com que compense distribuir trabalho.

9.2 Caso 2 – Sudoku

9.2.1 Objectivo e Explicação do Algoritmo

O objetivo deste teste é fazer a comparação da *DPF4j* em modo paralelo e modo distribuído na execução de uma aplicação *CPU bound*, isto é, uma aplicação onde a maior parte do tempo perdido é em processamento.

Este problema trabalha sobre uma fonte de dados partilhada, que se trata duma base de dados *MySQL* disponível na rede, no entanto, nunca há duas máquinas a trabalhar no mesmo registo da base de dados visto cada problema ser resolvido por apenas uma máquina. Esta questão é importante para garantir que não há bloqueios na base de dados o que causaria ruído nos resultados do teste.

A base de dados tem apenas uma tabela que contém objetos serializados representativos de problemas de *Sudoku* 9 por 9, e tem uma coluna onde é suposto a aplicação colocar a solução do problema.

Para solucionar o problema recorreu-se a uma técnica de força bruta. Em traços gerais o algoritmo é o seguinte: É carregado um registo da base de dados e o objeto com o problema é desserializado. Para resolver o problema é atribuída à primeira célula livre o valor 1 e é feita a sua validação, ou seja, é validado se esta célula não repete nenhum valor na mesma linha, coluna ou região. No caso de uma das regras ser quebrada o valor é incrementado, e as regras são revalidadas. Este procedimento é feito até atingir um valor que seja válido para a célula em questão e o algoritmo passa para a célula seguinte. No caso do valor 9 (máximo) ser atingido e este quebrar as regras, significa que não existe nenhum valor válido para a célula e então a célula é deixada em branco e o algoritmo volta à última célula modificada e incrementa o valor dessa. No final de toda a matriz estar preenchida com valores válidos, o objeto que a representa é serializado e guardado na base de dados.

9.2.2 Dados

Os dados de entrada estão registados numa base de dados localizada na rede local. A base de dados consiste numa tabela com mais de 100.000 problemas de *Sudoku* diferentes com o formato 9 por 9: 9 linhas, 9 colunas, 9 regiões.

Sudoku
id:INT
problem:BLOB
solution:BLOB

Tabela 17 - Registo do Problema na base de dados

Os objetos do tipo problema, e respetivas soluções, quando serializados ocupam 455 *bytes*.

9.2.3 Ambiente de testes

9.2.3.1 Máquinas

Nestes casos de teste foram utilizadas as máquinas apresentadas na Tabela 18.

Máquina	CPU	Cores	HyperThreading	RAM
i7-1	Intel Core i7 1.6Ghz	4	Sim(8)	4GB 1333MHz
i7-2	Intel Core i7 3.5Ghz	4	Sim(8)	8GB 1600MHz
i5	Intel Core i5 2.4Ghz	2	Sim(4)	8GB 1333MHz
C2D-1	Intel Core2Duo 2.53 Ghz	2	-	4GB 1067 MHz
C2D-2	Intel Core2Duo 2.4 Ghz	2	-	2GB 1067 MHz

Tabela 18 - Caso de teste 2 – Listagem de máquinas

9.2.3.2 Tecnologias

No presente teste foram utilizadas duas *frameworks: Java (sequencial)* e a *framework DPF4j* nos dois modos, paralelo e distribuído. O *Ateji PX* foi deixado de fora deste teste visto ter uma performance similar ao *DPF4j* paralelo e o objetivo deste teste é comparar a performance entre os modos paralelo e distribuído da *DPF4j* (usando o Java sequencial como referência).

9.2.3.3 Infraestrutura

No que diz respeito à infraestrutura de rede utilizado no teste da utilização da *DPF4j* no modo distribuído, este é apresentada na Figura 39.

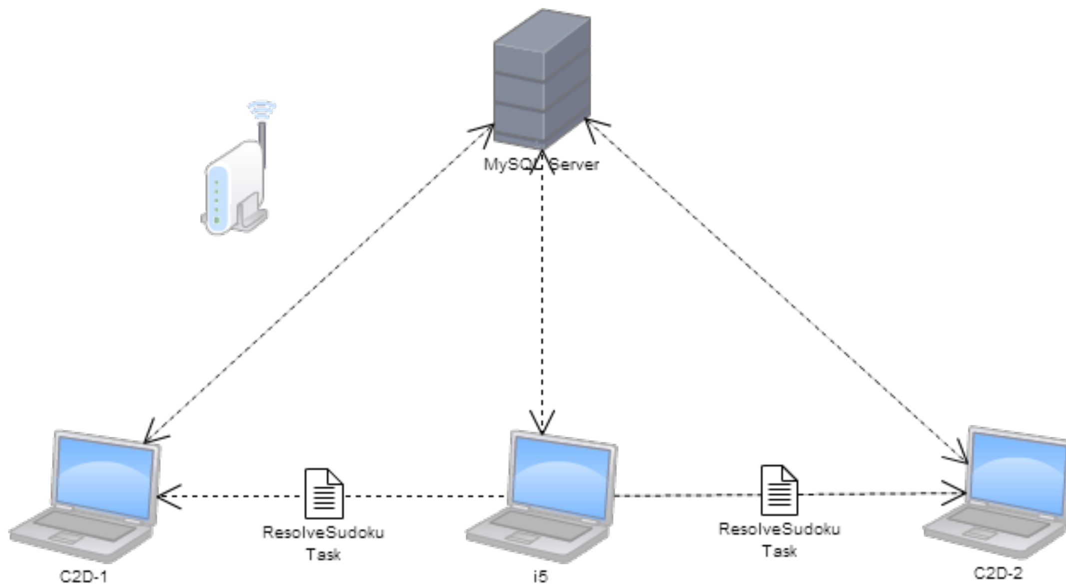

Figura 39 - Caso de teste 2 - Infraestrutura

9.2.3.4 Pressupostos

As ligações entre máquinas são todas via *WI-FI*.

A *framework DPF4j* executou sem um ciclo *warm-up*, isto é, os resultados dos testes contabilizam todas as execuções, incluindo a primeira. Isto tem as seguintes consequências nos testes que envolvem a *DPF4j*:

- os tempos incluem o arranque da *framework*, isto inclui arranque do *daemon* e dos serviços;
- o tempo do processo de descoberta está contabilizado;
- os processos de descoberta e arranque consomem recursos que tornam a execução global mais lenta;
- os processos de arranque e descoberta são paralelos à execução da aplicação principal por isso as primeiras execuções executam num modo paralelo, não distribuído, devido ao processo de descoberta ainda estar a encontrar os nós remotos;
- os testes *DPF4j* paralelos foram executados com os recursos para distribuição ativados mas sem máquinas remotas a aceitar pedidos.

9.2.4 Resultados

Na Tabela 19 são apresentados os tempos obtidos em milissegundos (*ms*) para a resolução do problema em questão para um tamanho de 5000. Os valores estão ordenados pelo tempo de execução em ordem decrescente.

	Tempo Total(ms)	Tempo Médio (ms) (Total/5000)
Simple Java: C2D-1	1.227.705	245.54
Simple Java: C2D-2	1.175.794	235.16
Simple Java: i7-1	1.002.341	200,47
DPF4j: i7-1	812.953	162,59
Simple Java: i5	970.957	194,19
Simple Java: i7-2	786.845	157,37
DPF4j: C2D-1	645.963	129.19
DPF4j: C2D-2	502.697	100.54
DPF4j: i5	371.311	742,62
DPF4j: i7-2	360.903	74,26
DPF4j: i7-1* + i7-2	356.919	71,38
DPF4j: i5* + C2D-1 + C2D-2	176.194	35,24

Tabela 19 - Caso de teste 2 - Resultados

*Máquina que delega trabalho

Na Figura 40 é apresentado um gráfico com os resultados obtidos para que se possa ter uma melhor perspetiva e fazer a comparação dos valores obtidos.

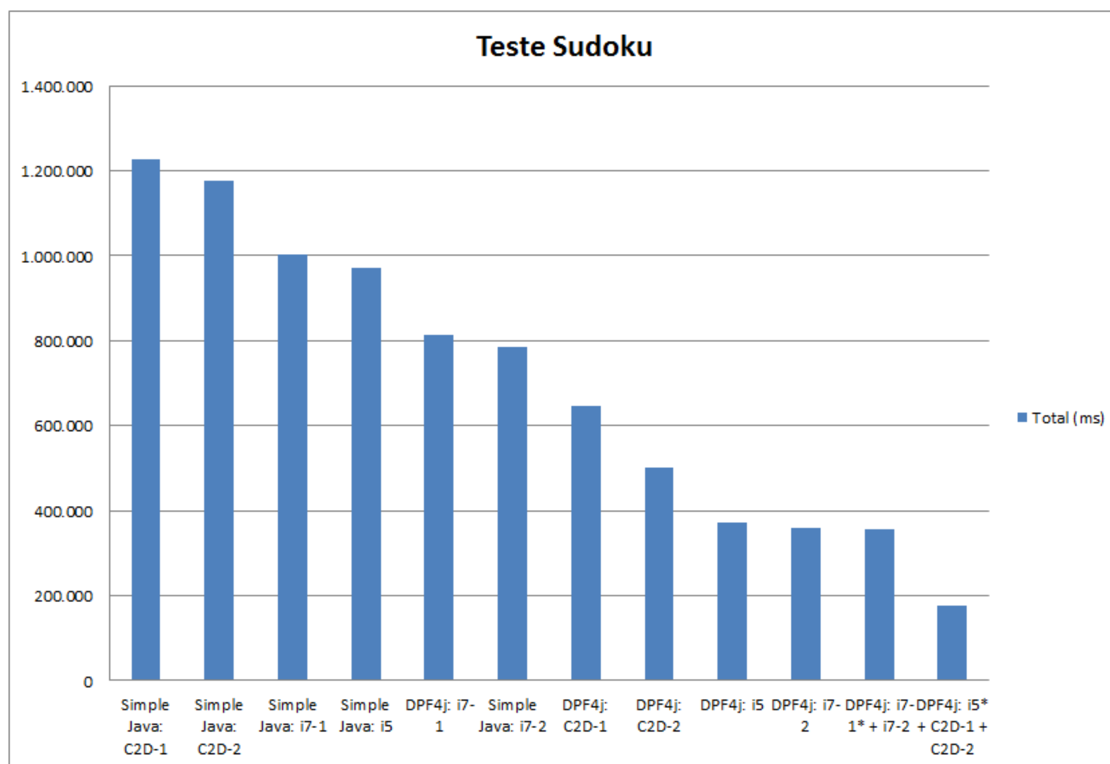

Figura 40 - Caso de teste 2 - Gráfico de resultados

Foi também realizado um teste, para um caso extremo, que é a resolução de 100.000 problemas utilizando a *framework DPF4j* em modo distribuído. Este teste tem o objetivo de demonstrar que o tempo médio por problema será inferior a um teste com menos iterações e comprovar que quanto maior for o processamento mais irrelevantes serão os tempos perdidos com a infraestrutura (arranque da aplicação, *classloading*, etc.). O valor obtido é apresentado na Tabela 20.

	Time (ms)	Tempo médio por problema (ms)
DPF4j: i5* + C2D-1 + C2D-2	3.332.515	33,33

Tabela 20 - Caso de teste 2 - Resultado do processamento distribuído

9.2.5 Conclusões

Como se pode ver o tempo médio por problema resolvido *DPF4j* nas três máquinas foi mais elevado no problema de 5000 do que de 100.000, apesar da diferença ser mínima, isto era esperado devido aos tempos de arranque da *DPF4j* que com a dimensão maior do problema começam a tornar-se mais desprezáveis. Da mesma forma também aumenta a percentagem de tempo com três máquinas a executar em simultâneo.

No caso da execução da máquina i7-1* + i7-2 reparamos que a máquina i7-1 mais do que duplicou a sua performance ao distribuir para a máquina i7-2, no entanto, também se nota que a diferença deste valor para o valor da i7-2 são apenas 4 segundos, isto deve-se ao facto da máquina que está a delegar trabalho, não ocupar os 8 *CPUs* lógicos da máquina remota, estando a delegar apenas 4 tarefas de cada vez.

10 Conclusões

No desenvolvimento desta dissertação foi possível aplicar todos os conhecimentos adquiridos ao longo da licenciatura e do mestrado. Durante o desenvolvimento da *framework* foram adquiridos conhecimento mais aprofundados na área da computação paralela e distribuída.

10.1 Resumo do Trabalho

O objetivo principal do trabalho consistiu na concepção de uma *framework* para dar suporte à computação paralela e distribuída na linguagem *Java*. Esta dissertação desenvolveu parte desse trabalho, tendo-se focado na *API* da *framework* e em toda a comunicação entre os nós cooperantes.

Todos os objetivos inicialmente definidos foram atingidos com sucesso. Foi possível criar a *framework*, ainda protótipo, que fornece o suporte ao desenvolvimento de aplicações paralelas e/distribuídas utilizando a linguagem de programação *Java*.

Através da sua utilização, o programador pode criar ciclos do tipo *for* e *foreach*, onde o seu processamento é paralelizado e distribuído de forma transparente, apenas sendo necessário ter a *framework* instalada nas máquinas cooperantes. A *API* da *framework* foi concebida recorrendo à programação orientada a objetos, visto que a *framework* não é uma extensão da *framework Java*, mas sim uma biblioteca de suporte ao desenvolvimento de aplicações paralelas e distribuídas. Dado isto, a utilização de objetos para foi um aspecto fundamental para a criação da API.

No que diz respeito à distribuição de trabalho, a forma como esta é feita é talvez o ponto mais forte da *framework*, visto que todo o processo de associação de nós pertencentes ao grupo que vai distribuir trabalho é realizado de forma automática sem que seja necessária a intervenção humana, para nós que pertençam à mesma rede local. A associação de nós que se encontram na Internet, obriga a que estes sejam devidamente configurados. Desta forma apenas é necessário configurar apenas um nó que conheça nós que se encontram na Internet e o próprio mecanismo de descoberta automático faz com que esses nós sejam também conhecidos por todos os restantes nós existentes na rede, pertencentes ao mesmo *workgroup*.

Como prova disso, temos ótimos resultados obtidos na fase de testes. Este trabalho é uma mais-valia para a área da computação paralela distribuída, apesar de já existiram alguns trabalhos nesta mesma área.

10.2 Trabalho Futuro

Apesar de avançada, a *framework* encontra-se ainda num estado protótipo e necessita ainda de mais trabalho para que se possa garantir que a sua funcionalidade total possa ser disponibilizada à comunidade de software aberto.

Para esta dissertação foi implementado um sistema de segurança básico baseado em chaves partilhadas que apenas garante a segurança do *workgroup*, mas não a segurança dos nós individuais, mas para o seu uso em sistemas abertos será necessário permitir a utilização de protocolos de autenticação e encriptação mais robustos como por exemplo a utilização de chaves simétricas para garantir a segurança nas relações entre os nós.

Outro ponto a melhorar é a administração do *DPF Daemon*. Este *daemon* muitas vezes irá correr em servidores e a sua gestão e administração deverão ser feitas remotamente, para isso está planeado implementar interfaces *JMX* para gerir e monitorizar o *DPF Daemon* no geral e mais precisamente o *DPF ServiceManager* (arranque/desligar serviços) e a *DPFConfiguration*. Além da gestão, as interfaces JMX também poderão servir para monitorizar estatísticas de utilização e até obter dados capturados pelo *DPF Profiler*.

Do ponto de vista de utilização, o principal fator diferenciador entre a DPF4j e outras plataformas de programação paralela e distribuída, é que esta solução não se baseia em memória partilhada distribuída. No futuro deverá ser estudada a implementação deste modelo de programação e qual o seu impacto na performance do sistema. Uma solução poderá recorrer à duplicação e sincronização de memória entre nós ou "whiteboards" (zonas de memória na rede partilhadas pelos vários executores onde poderão ler e escrever).

No que diz respeito à *API* da *framework*, poderão ser implementadas novas *APIs* como por exemplo a criação do ciclo *reduce*.

Relativamente à transferência de dados por rede, poderão ser estudados mecanismos de compressão de pacotes para reduzir o tempo de transferência de dados e código entre nós. Isto para os casos em que o tempo de compressão justificar os ganhos no tempo de transferência.

O escalonamento também poderá ser melhorado, por exemplo se o controlo de execução e distribuição de tarefas conseguir cancelar tarefas de nós remotos quando o nó local está sem trabalho e poder ser o próprio a realizá-las. Os nós também deverão conseguir recolher estatísticas de nós conhecidos, nomeadamente tempos de resposta e capacidade de processamento de modo a melhorar as decisões de escalonamento, no momento de decidir para que nó enviar determinada tarefa. Também poderão ser implementados mais escalonadores para dar mais opções ao programador, por exemplo, a existência de escalonadores *soft real-time*.

No futuro também seria interessante adicionar algum tipo de injeção de dependências de forma a permitir alterar as implementações a utilizar nos vários módulos de forma declarativa. Injeção de dependências é um padrão de desenvolvimento de software que tem como objetivo manter um baixo nível de acoplamento entre os diferentes módulos, tornando as dependências uma questão de configuração em vez de uma questão e compilação.

Para isso poderá fazer uma implementação própria ou incorporar uma biblioteca de terceiros como *Spring* [25] ou *Weld* (CDI) [26].

A *framework* precisa também de muito trabalho de testes e *profiling* de forma a conseguir-se aumentar a já aceitável performance e robustez geral da solução.

11 Referências

[1] D. Lea, “A Java Fork/Join Framework”.

[2] T. H. Cormen, C. E. Leiserson e R. L. Rivest, Introduction to Algorithms, MIT Press, 2000.

[3] P. Graham, OpenMp: A Parallel Programming Model for Shared Memory Architectures, 1999.

[4] C. Campbell, Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures, Microsoft Press, 2010.

[5] MIT Laboratory for Computer Science, Cilk 5.4.6: Reference Manual, 1998.

[6] Intel Corporation, [Online]. Available: http://threadingbuildingblocks.org/. [Acedido em 03 02 2012].

[7] P. Viry, Ateji PX for Java: Parallel Programming made Simple, 2010.

[8] Soumyasch, “The .NET Framework stack,” 20 10 2007. [Online]. Available: http://en.wikipedia.org/w/index.php?title=File:DotNet.svg&page=1. [Acedido em 15 03 2012].

[9] Microsoft, “Parallel Class,” [Online]. Available: http://msdn.microsoft.com/en-us/library/system.threading.tasks.parallel.aspx. [Acedido em 03 02 2012].

[10] Oracle, “Fork/Join,” [Online]. Available: http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html. [Acedido em 09 01 2012].

[11] Oracle, “Interface ExecutorService,” [Online]. Available: http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ExecutorService.html. [Acedido em 31 01 2012].

[12] A. Tanenbaum e M. Van Steen, Distributed Systems: Principles and Paradigms, 2007.

[13] B. Carpenter, “HPJava Home Page,” [Online]. Available: http://www.hpjava.org/. [Acedido em 06 02 2012].

[14] C. Koelbel. [Online]. Available: http://hpff.rice.edu/. [Acedido em 06 01 2012].

[15] B. Barney, “Introduction to Parallel Computing,” [Online]. [Acedido em 01 02 2012].

[16] D. Pigott, The Encyclopedia of Computer Languages., Murdoch University, 2006.

[17] “JavaParty - Java's Companion for Distributed Computing,” [Online]. Available: http://svn.ipd.kit.edu/trac/javaparty/wiki/JavaParty/QuickTour. [Acedido em 05 02 2012].

[18] Oracle, “Java™ Platform, Standard Edition 8 Early Access with Lambda Support,” [Online]. Available: http://jdk8.java.net/lambda/. [Acedido em 19 01 2012].

[19] Apache Software Foundation, “Apache log4j™ 1.2,” [Online]. Available: http://logging.apache.org/log4j/1.2/index.html. [Acedido em 09 06 2012].

[20] C. Gülcü, “Short introduction to log4j,” 03 2002. [Online]. Available: http://logging.apache.org/log4j/1.2/manual.html. [Acedido em 18 06 2012].

[21] Oracle, “Java Management Extensions (JMX) Technology,” [Online]. Available: http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html. [Acedido em 20 08 2012].

[22] Oracle, “Class MessageFormat,” [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/text/MessageFormat.html. [Acedido em 12 06 2012].

[23] Oracle, “Permissions in Java™ SE 7 Development Kit (JDK),” [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/security/permissions.html. [Acedido em 18 08 2012].

[24] Oracle, “Default Policy Implementation and Policy File Syntax,” [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html. [Acedido em 18 08 2012].

[25] VMware, “Springsource Community,” [Online]. Available: http://www.springsource.org/. [Acedido em 01 10 2012].

[26] Red Hat, Inc., “Weld Home,” [Online]. Available: http://seamframework.org/Weld. [Acedido em 01 10 2012].

[27] Intel Corporation. Intel(R), Threading Building Blocks: Reference Manual, 2007.

[28] Intel Corporation. Intel(R), Threading Building Blocks: Tutorial, 2007.

[29] Microsoft, “System.Threading.Tasks Namespace,” 25 01 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/dd235608.aspx.

[30] Oracle, “Interface ExecutorService,” [Online]. Available: http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ExecutorService.html. [Acedido em 13 01 2012].

[31] A. Moreira, “Introdução,” [Online]. Available: http://www.dei.isep.ipp.pt/~andre/documentos/redes-introducao.html. [Acedido em 13 05 2012].

[32] A. Moreira, “"User Datagram Protocol" (UDP),” [Online]. Available: http://www.dei.isep.ipp.pt/~andre/documentos/udp.html. [Acedido em 18 05 2012].

[33] A. Moreira, “Transmission Control Protocol (TCP),” [Online]. Available: http://www.dei.isep.ipp.pt/~andre/documentos/tcp.html. [Acedido em 18 05 2012].

[34] G. Coulouris, J. Dollimore e K. Kinberg, Distributed Systems: Concepts and Design., Addison-Wesley/Pearson Eduction, 2005.