Instituto Superior de Engenharia do Porto

Suporte à Computação Paralela e Distribuída em Java: Distribuição e Execução de Trabalho

Daniel Duarte Ribeiro Barciela

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

Nos últimos anos começaram a ser vulgares os computadores dotados de multiprocessadores e *multi-cores*. De modo a aproveitar eficientemente as novas características desse hardware começaram a surgir ferramentas para facilitar o desenvolvimento de software paralelo, através de linguagens e *frameworks*, adaptadas a diferentes linguagens.

Com a grande difusão de redes de alta velocidade, tal como *Gigabit Ethernet* e a última geração de redes *Wi-Fi*, abre-se a oportunidade de, além de paralelizar o processamento entre processadores e cores, poder em simultâneo paralelizá-lo entre máquinas diferentes. Ao modelo que permite paralelizar processamento localmente e em simultâneo distribuí-lo para máquinas que também têm capacidade de o paralelizar, chamou-se "modelo paralelo distribuído".

Nesta dissertação foram analisadas técnicas e ferramentas utilizadas para fazer programação paralela e o trabalho que está feito dentro da área de programação paralela e distribuída. Tendo estes dois factores em consideração foi proposta uma *framework* que tenta aplicar a simplicidade da programação paralela ao conceito paralelo distribuído.

A proposta baseia-se na disponibilização de uma *framework* em Java com uma interface de programação simples, de fácil aprendizagem e legibilidade que, de forma transparente, é capaz de paralelizar e distribuir o processamento. Apesar de simples, existiu um esforço para a tornar configurável de forma a adaptar-se ao máximo de situações possível.

Nesta dissertação serão exploradas especialmente as questões relativas à execução e distribuição de trabalho, e a forma como o código é enviado de forma automática pela rede, para outros nós cooperantes, evitando assim a instalação manual das aplicações em todos os nós da rede.

Para confirmar a validade deste conceito e das ideias defendidas nesta dissertação foi implementada esta *framework* à qual se chamou *DPF4j* (*Distributed Parallel Framework for JAVA*) e foram feitos testes e retiradas métricas para verificar a existência de ganhos de performance em relação às soluções já existentes.

Abstract

In the last years, multiprocessor and multi-core computers have become common in the computer industry. In order to take advantage of these new features, several tools including programming languages and frameworks have appeared to ease the parallelization of the applications' workload.

With the rise of high-speed networks such as Gigabit networks and the latest generation of Wi-Fi, there is now the opportunity, in addition of parallelizing the processing between processors, to be able to simultaneously parallelize workloads between different machines, that is to distribute the workload. The concept of parallel processing considering local execution and simultaneous distribution to other machines that also can parallelize workloads is called "distributed parallel concept".

This thesis analyzed the techniques and tools used to make parallel programming, the work that is done within the area of parallel and distributed programming, and proposed a framework that tries to apply the simplicity of the parallel programming to the distributed parallel concept.

The proposal is based on providing a Java framework with a simple API that is easy to learn and easy to read, which seamlessly will be able to parallelize and distribute the application workload. Although simple, there was an effort to make it configurable in order to adapt it to the maximum possible situations.

This thesis will address issues related to the execution and distribution of work, and the way that the code is automatically transferred between machines without the need of manual installation in the several networked nodes.

To validate this concept and the ideas exposed in this thesis, it was implemented a framework which was called DPF4j (Distributed Parallel Framework for JAVA) and tests were made to withdrawn metrics to check for performance gains compared to the existing solutions.

Agradecimentos

Ao fim desta dissertação gostaria de agradecer a todas as pessoas que durante este tempo me apoiaram e tornaram a sua concretização possível.

Quero agradecer principalmente ao Rui Rodrigues que desenvolveu grande parte deste trabalho comigo, pelo seu empenho, excelente trabalho e sentido de responsabilidade.

De forma muito especial quero agradecer às pessoas que me deram apoio incondicional apesar de eu não ter tempo de o retribuir e me mantiveram motivado e contribuíram para a minha felicidade ao longo do último ano, nomeadamente aos meus pais Álvaro e Emília Barciela, à Catarina, à minha família e aos meus amigos.

Agradeço ao meu orientador Prof. Doutor Luís Lino Ferreira e ao meu coorientador e ex-colega de curso Eng.º Cláudio Maia por todo o tempo dispensado e conhecimento técnico que partilharam comigo.

Gostaria de dar o meu grande obrigado aos meus colegas de curso Rui Rodrigues (mais uma vez), Vítor Rodrigues e Miguel Ferreira que contribuíram para o meu sucesso académico, profissional e pessoal desde o início da licenciatura.

Estou também grato aos colaboradores da I2S, especialmente aos que se cruzaram comigo na minha equipa com os quais aprendi muito do que foi aplicado neste trabalho.

Finalmente, a todos que direta ou indiretamente contribuíram para o sucesso da dissertação.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Código

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
HPF 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

Ao longo de toda história da informática houve uma pesquisa intensiva e uma evolução notável no que toca a melhorar a performance dos sistemas. A performance de um sistema depende dos mais variados aspetos, desde o *hardware*, ao sistema operativo, qualidade e complexidade das aplicações, entre outros fatores.

Durante anos a evolução da performance esteve centrada no aumento da velocidade de relógio dos *CPUs* (*Central Processing Unit*). Assim que os ganhos através desta técnica começaram a ser menos significativos, começaram a ser explorados os sistemas multiprocessador e *multicores*, 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 simples 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 começaram a fornecer 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* [6] e o modelo *Divide and Conquer* [7]. Estes modelos partem do princípio que a resolução de 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* [4], do *.Net* [8], do *Cilk* [5], do *Thread Building block (TBB)* [9] e do *AteJi PX* [10].

Nos nossos dias o material informático está muito mais acessível aos particulares e empresas e a obsolescência de *hardware* deu origem à existência de muitos equipamentos armazenados e não utilizados. Esta abundância de equipamentos juntamente 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/ações que levam o sistema no seu global a atingir os seus propósitos.

Este novo conceito é chamado 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.

Devido à sua complexidade e dimensão o trabalho sobre "Suporte à Computação Paralela e Distribuída em Java", foi na sua maioria feito em conjunto entre os alunos Daniel Barciela e Rui Rodrigues inscritos no Mestrado de Engenharia Informática do Instituto Superior de Engenharia do Porto. O aluno Daniel Barciela especializou-se em "Distribuição e Execução de Trabalho", tendo como foco os módulos *DPF Scheduler Layer* e *DPF Classloaders*. O aluno Rui Rodrigues focou-se na interface de programação e Comunicação entre os Nós Cooperantes, cuja implementação incidiu sobre os módulos *DPF User Layer* ou *DPF4j API* e *DPF Services* (Figura 1).

Esta dissertação irá incidir com maior foco nas questões de escalonamento, distribuição e execução de trabalho, e na forma como a *framework* distribui o código pela rede de forma totalmente dinâmica e automática, com o objetivo de evitar a instalação manual do código referente às tarefas que são delegadas nos vários nós da rede.

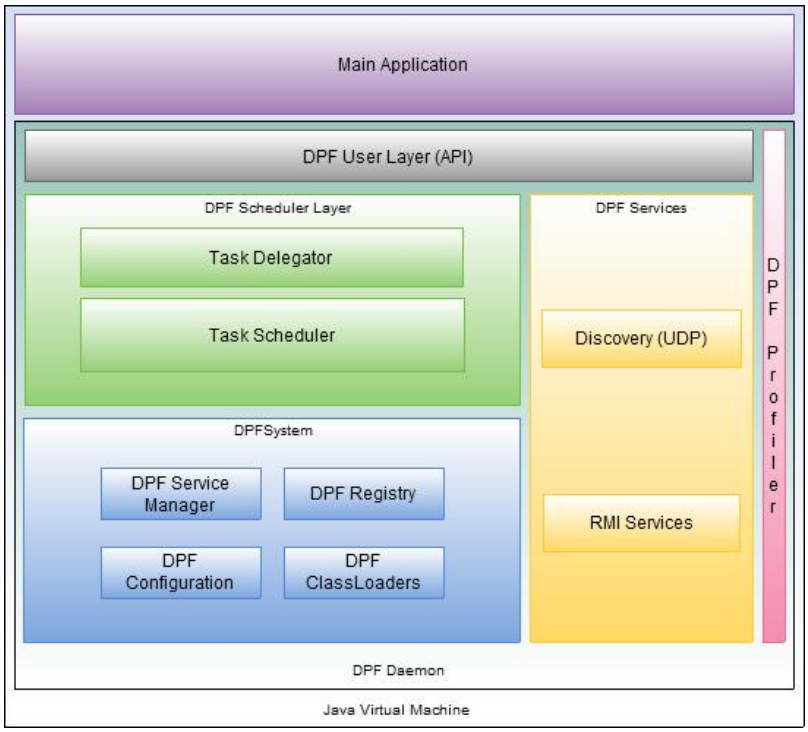

Figura 1 - Arquitetura da Framework

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 perspetivas de trabalho futuro. Para alcançar estes objetivos foram colocadas alternativas e foram realizados estudos e testes para garantir a sua validade.

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.

1.2 Resumo da Solução Proposta

À *framework* desenvolvida chamou-se de *DPF4j* (*Distributed Parallel Framework for Java*), 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.

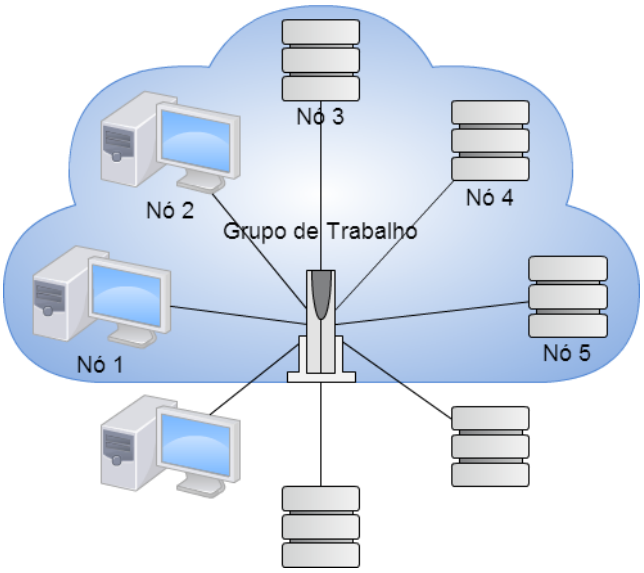

Figura 2 -Arquitetura Exemplo

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).

```
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 aspeto 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 1) 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 aspetos de segurança.

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 efetuado 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 efetuada 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 efetuados dentro da área de modelos híbridos paralelos e distribuídos.

O **sexto capítulo** apresenta a *framework* desenvolvida de forma global, justificando as opções tecnológicas tomadas, 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 as apresentadas todas as entidades que compõem um sistema distribuído utilizando a *framework*.

O **sétimo capítulo** apresenta detalhadamente como é definida uma tarefa na *framework* e como a *framework* lida com as tarefas no que toca a escalonamento tanto para as *threads* locais como para as máquinas remotas e todos os procedimentos inerentes ao processo.

O **oitavo capítulo** explica como é que a *framework* transfere *bytecode* entre as várias máquinas intervenientes no processo em tempo de execução.

O **nono capítulo** demonstra os resultados obtidos nos testes efetuados utilizando várias linguagens e *frameworks*, 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 enviados 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 efetuado estão apresentadas no **décimo capítulo**, onde também é sugerido algum 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* [6] e *Divide and Conquer* [7]

2.1 Modelo Fork-Join

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

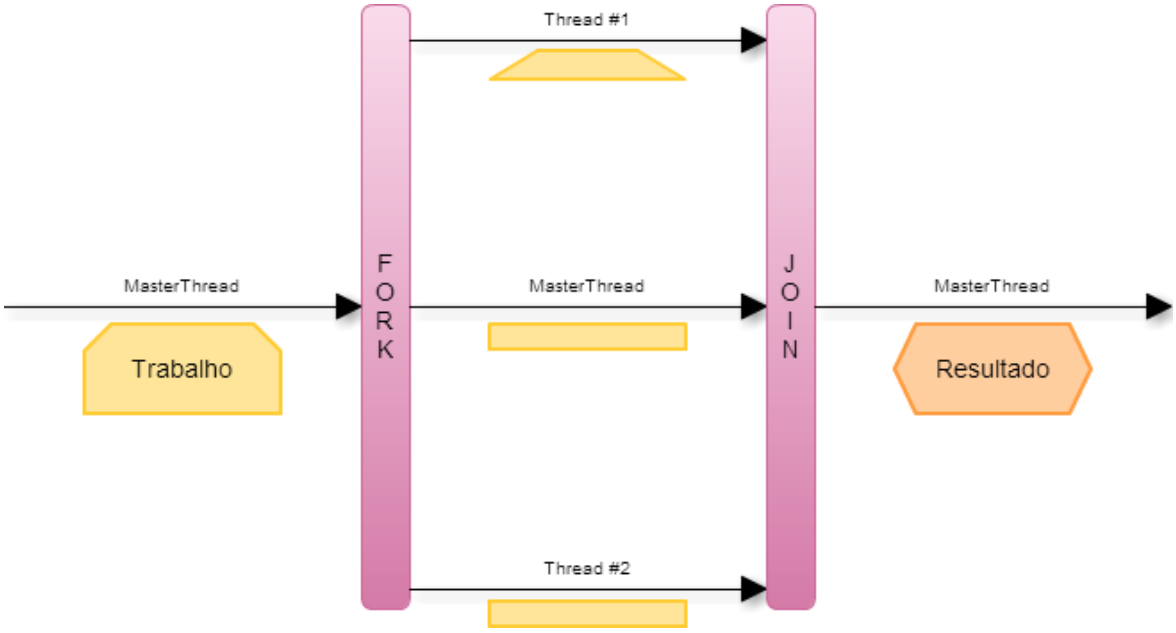

Figura 3 - Modelo *Fork-Join*

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 sub-trabalhos). 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 fração do que demoraria se fosse executado por uma única *thread*. 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 3).

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 4.

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 é decidida em tempo de execução pelo escalonador do sistema operativo). Neste cenário o tempo de processamento total do trabalho seria de 60 segundos, conseguindo desta forma reduzir 40% do tempo gasto.

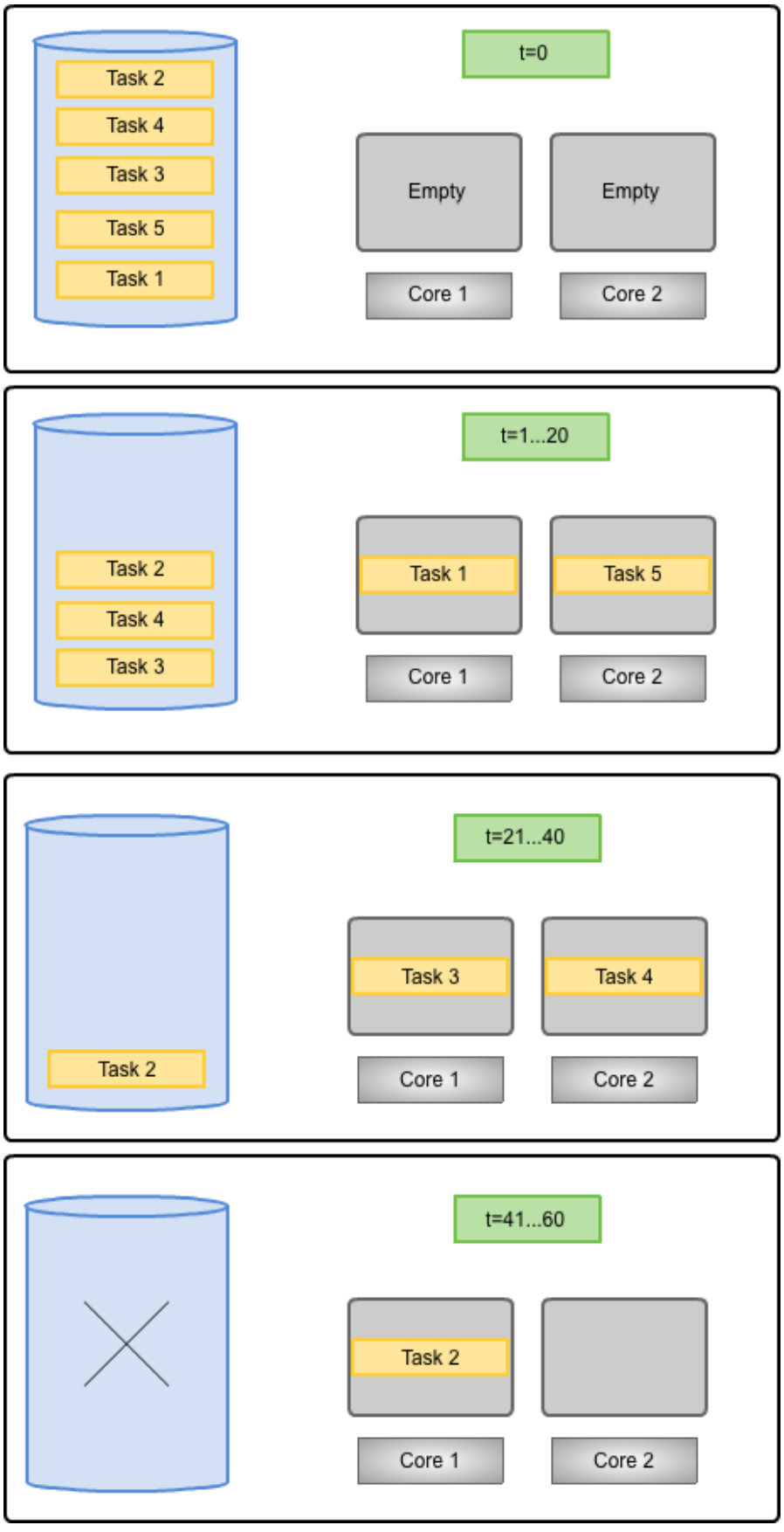

Figura 4 - Exemplo de um ambiente com dois *cores*

Para um ambiente com quatro *cores*, teria-se algo como é apresentado na Figura 5.

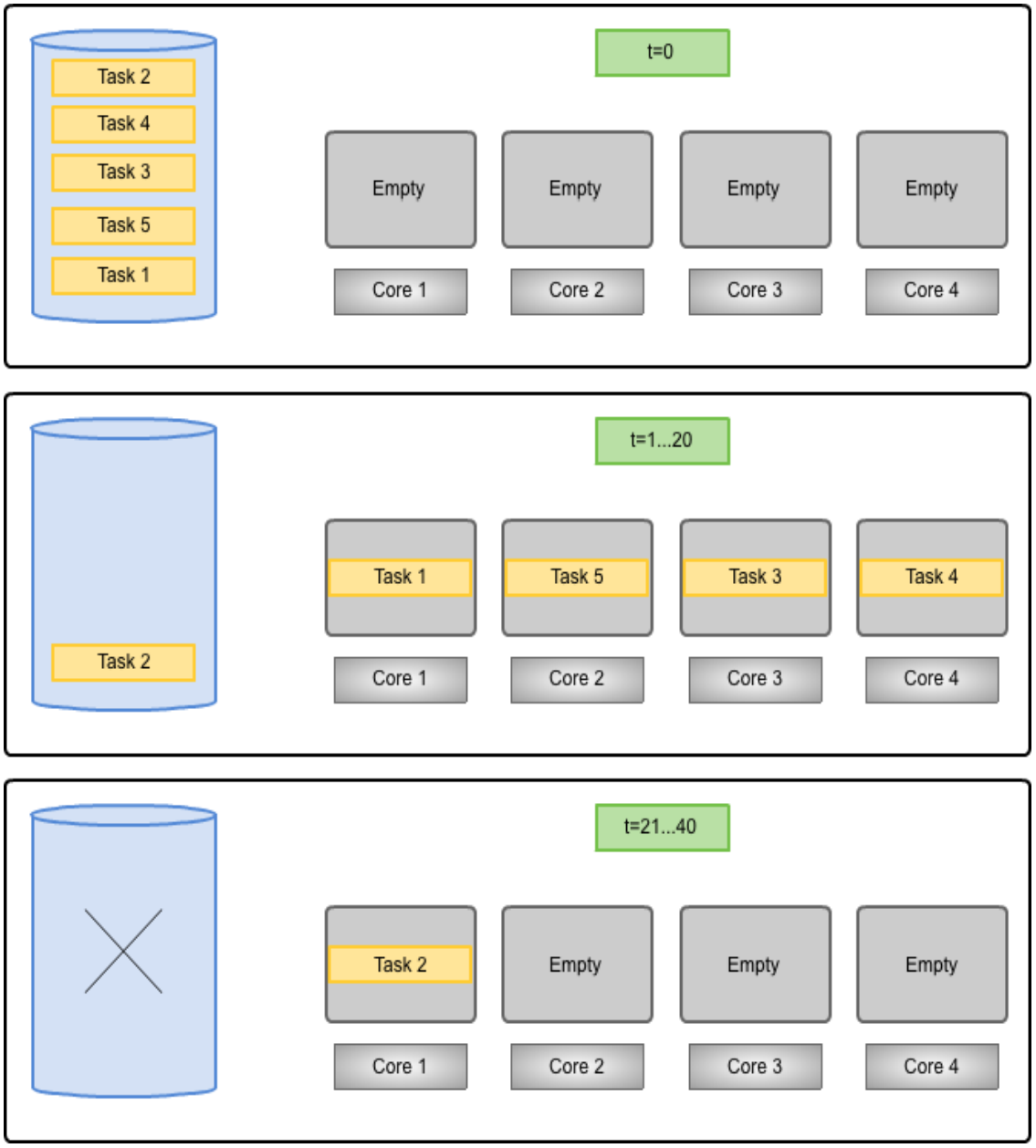

Figura 5 - Exemplo de um ambiente com quatro *cores*.

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 6.

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.

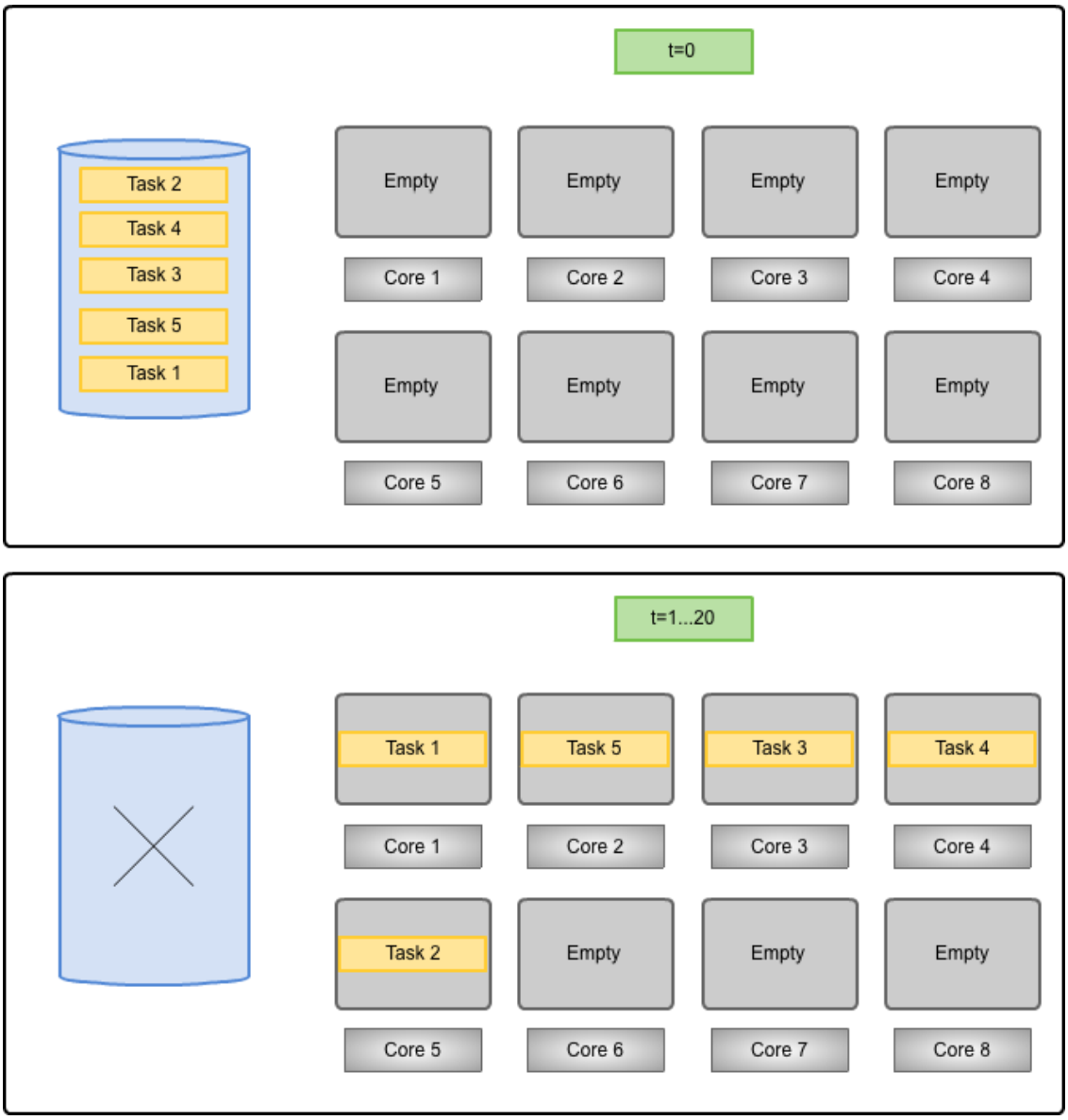

Figura 6 - Exemplo de um ambiente com oito *cores*.

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 7 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.

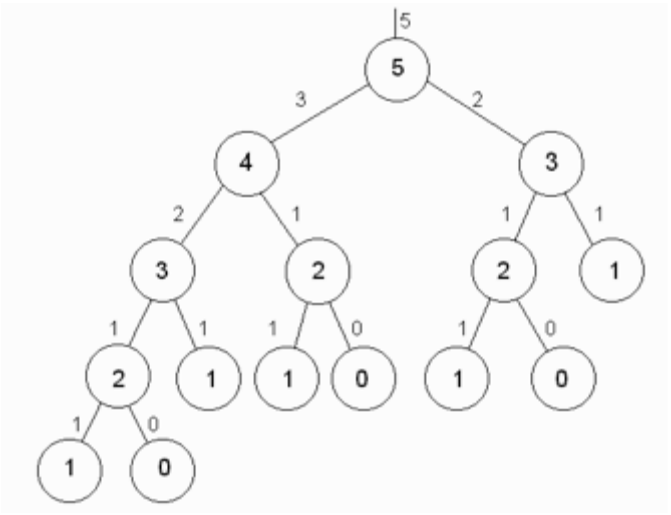

Figura 7 - Esquema do cálculo do número de Fibonacci utilizando o Modelo *Divide and Conquer*

O diagrama da Figura 7 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 adotado 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 de funções que pode originar devido à sua natureza recursiva, que por sua vez pode aumentar em muito a *stack* de uma aplicação.

3 Programação Paralela

Neste capítulo são abordadas *frameworks* que implementam os modelos teóricos descritos no capítulo anterior, nomeadamente o *OpenMP*, o *Java 7*, o *.NET*, o *Cilk* e o *AteJi Px*.

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 do capítulo 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 8).

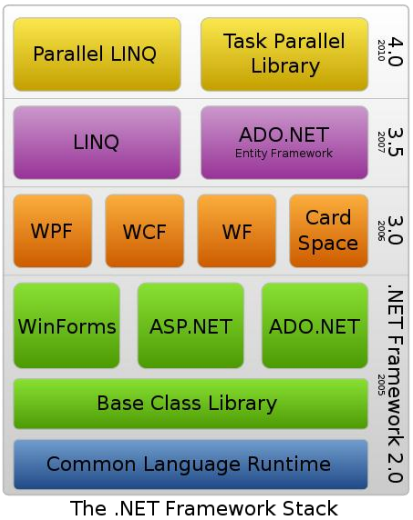

Figura 8 - *Stack* da *Framework .Net* [11]

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

3.1.1 Parallel Loops

Os ciclos paralelos 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 objeto *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 [12] 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 isto 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 9 - 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 aspetos:

- *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.

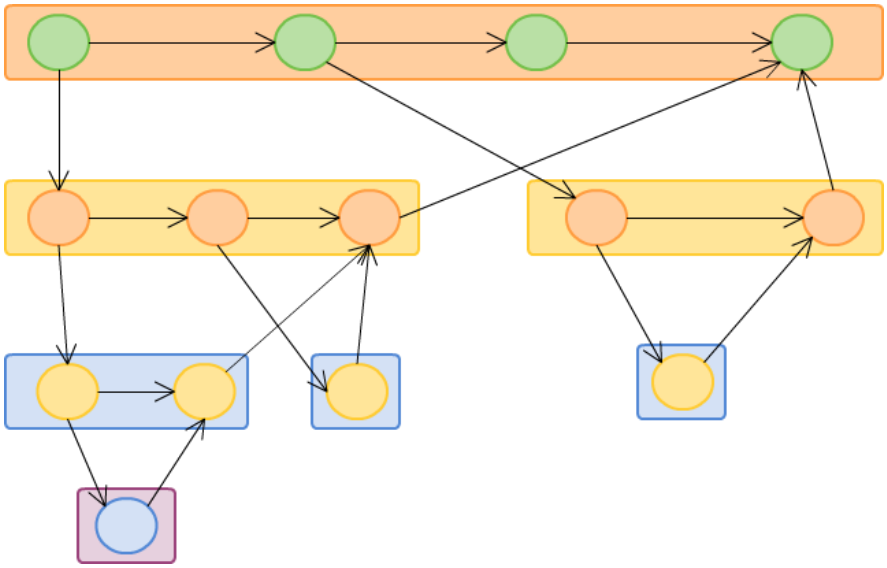

Figura 10 - Esquema de subprocedimentos no Cilk

Na Figura 10 é apresentado um exemplo onde é efectuada 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 um 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)

O OpenMP 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 aspetos, 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 11).

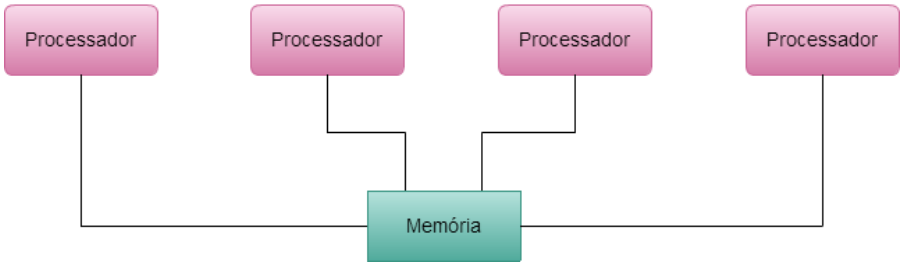

Figura 11 - Esquema da partilha de memória entre processadores

A *master* thread pode em determinadas situações criar novas threads para realizar sub-trabalhos, é aqui que se expressa o paralelismo. Na Figura 12 é apresentado esquema referente à *master thread.*

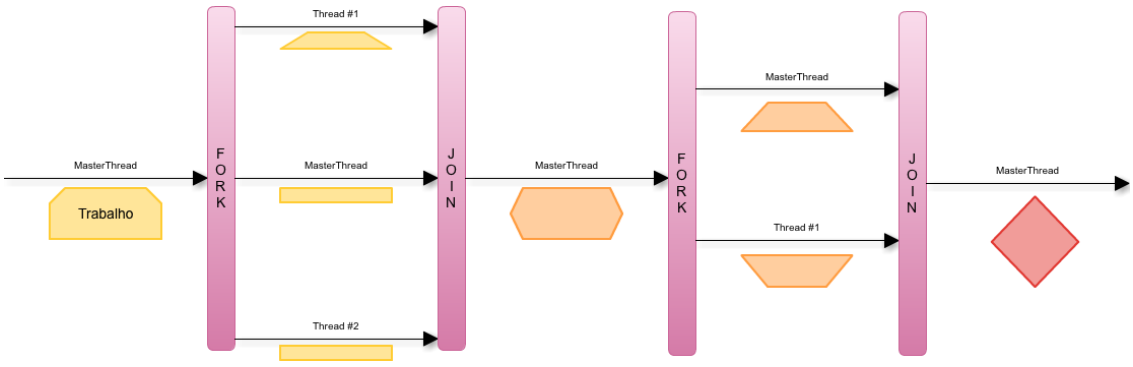

Figura 12 - Exemplo da master thread no OpenMP

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++* [9].

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 13 é apresentado o fluxo que é feito no caso do *parallel_reduce*.

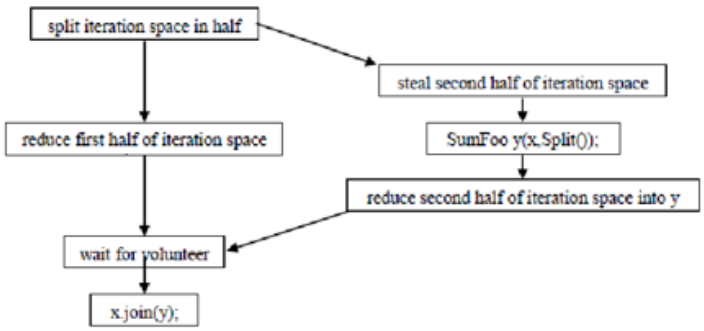

Figura 13 - TBB Parallel Reduce

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

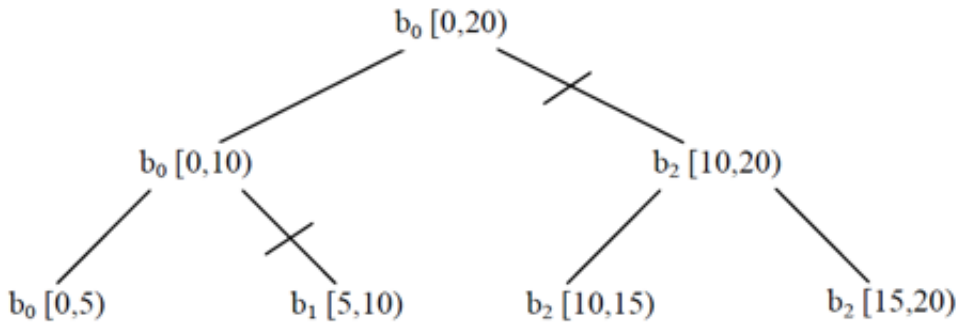

Figura 14 - Exemplo de um parallel_reduce sobre um blocked_range<int>(0,20,5)

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 objeto 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 itens 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 [10] 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). De 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.

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

De seguida (Código 23) encontra-se o código da classe gerada pelo *Ateji PX* para o exemplo citado anteriormente (`[ 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*

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*.

Após uma leitura e interpretação do Código 23, 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 multithread*) que são adicionados a um objeto do tipo *ParallelBlock*. Por fim é executado o método *run* do objeto ParallelBlock que executa o método *run* de cada objeto *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 objeto *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*, fábricas 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 [13]. 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 otimizada 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 [14].

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 objeto 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 objeto 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 objeto 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 t*hread* 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 *frameworks* fornecem uma interface parametrizável no que diz respeito à forma como é 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 *frameworks* 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 *frameworks* 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 *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*, no entanto não necessitando de uma máquina virtual. A nível de desenvolvimento a especificação é suportada por linguagens de alto nível, que por sua vez são eficientes e escaláveis. 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 aspetos 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.

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 15). 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.

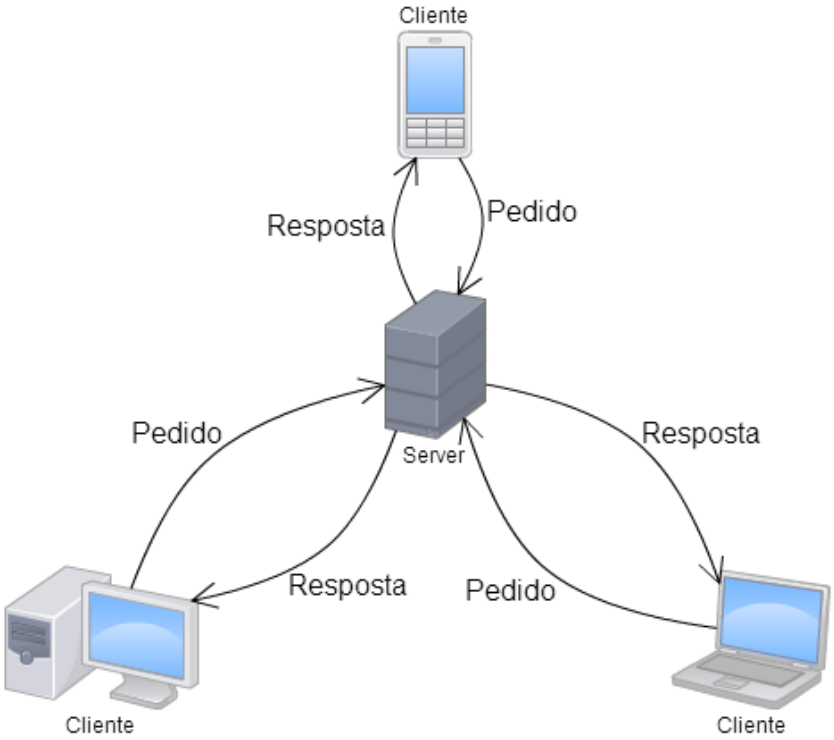

Figura 15 - Exemplo do tipo de arquitetura Cliente/Servidor

Um tipo particular de modelo cliente servidor que merece ser referenciado é o *thin client*/servidor [15] 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.

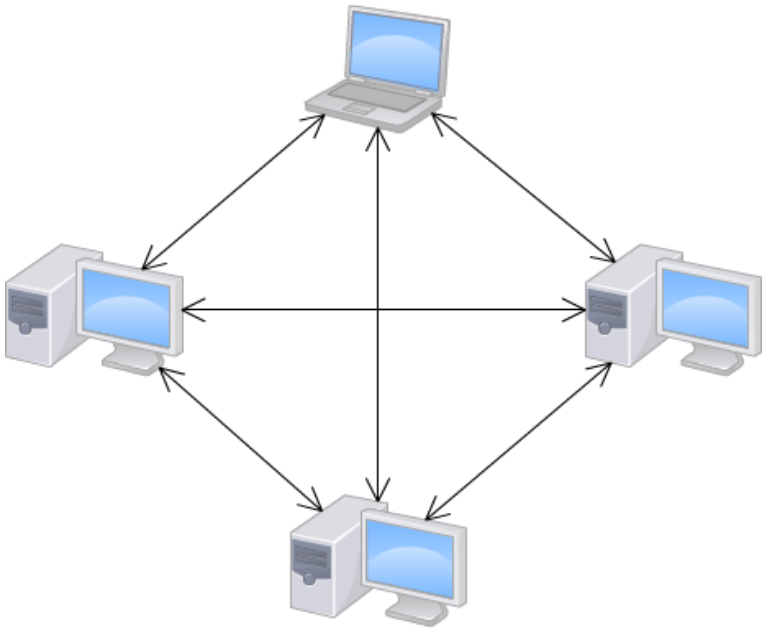

Figura 16 - Exemplo da arquitetura *Peer-to-Peer*

4.3 Híbrido

Um sistema distribuído híbrido é um sistema que tem simultaneamente relações cliente/servidor e *peer-to-peer* [15], 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 17 é apresentado um exemplo de um sistema distribuído híbrido.

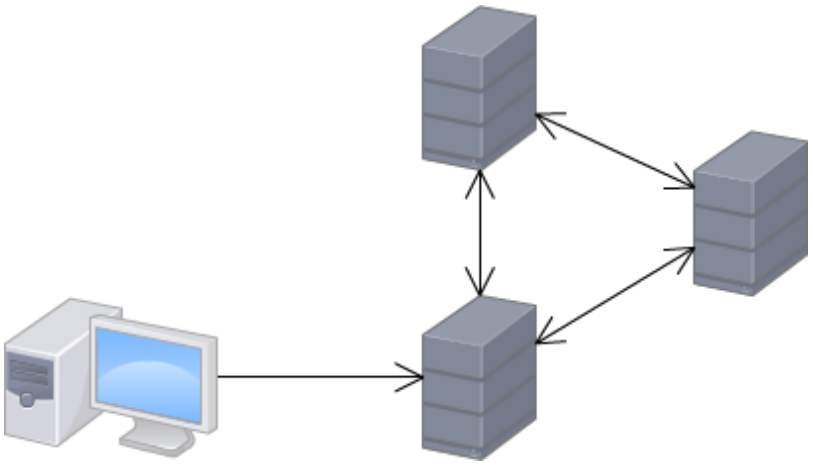

Figura 17 - Exemplo de um sistema distribuído híbrido

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* [16] 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*) [17] 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 [18]).

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* [19].

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 (Código 26) 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 anteriores é utilizada a 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 objeto 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 retangular (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 otimizar 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 - foreach 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* [20]

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, objetos remotos podem ser acedidos mesmo que se encontrem em diferentes máquinas, isto é feito de forma transparente ao utilizador.

No que diz respeito a comunicações, todos os mecanismos referentes a este aspeto são transparentes ao utilizador. Quando se trata de protocolos de comunicação não explícitos, cabe ao utilizador efetuar o respetivo 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 as estas três tecnologias analisadas. Estas três tecnologias, todas elas são 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 por um grupo de máquinas com a *framework DPF4j*.

As grandes candidatas à implementação desta *framework* eram a linguagem *Java* e a *framework .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 [21]. 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 nos vários aspetos. A linguagem *Java* também proporcionar 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 optar pela linguagem Java dá-se pelo facto de esta 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* que 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 permitem 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 fator 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 perspetiva 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 poderem 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 18 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.

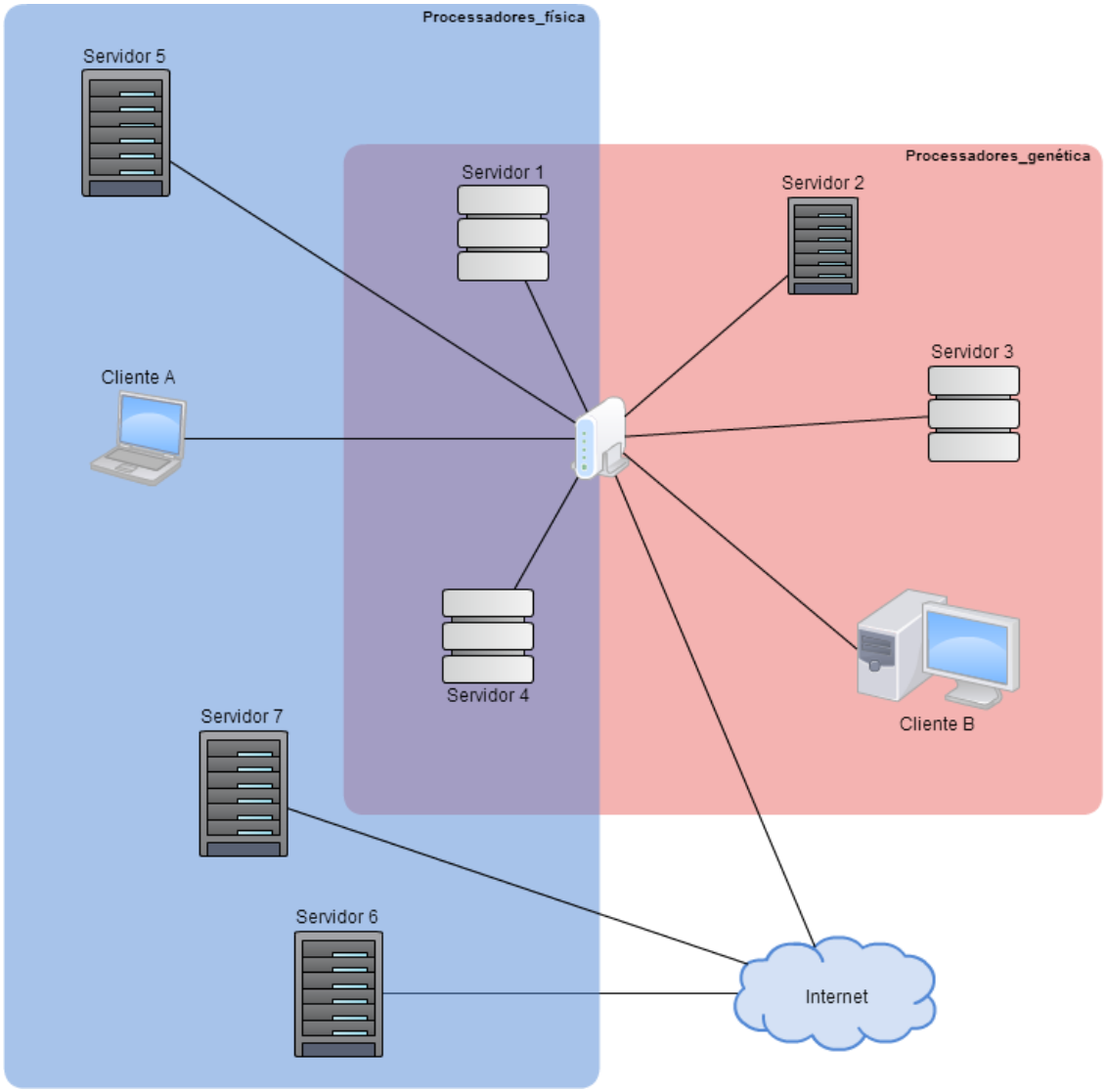

Figura 18 - Arquitetura do Sistema

Como se pode ver na Figura 18, 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 um 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 19 representa a arquitetura global da *framework*, onde se encontram definidos cada um dos módulos que a compõem.

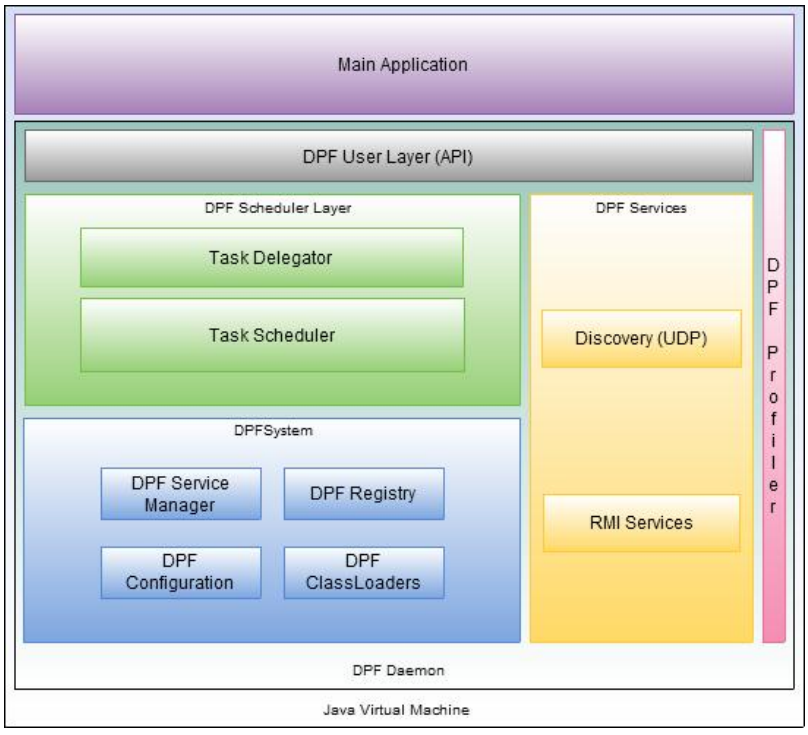

Figura 19 - Arquitetura da Framework

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 e não a aceitar trabalho.

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 o iniciar. 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 na pasta bin da distribuição.

6.2.2 DPF User Layer

A *DPF User Layer*, também conhecida como *DPF4j API*, fornece uma 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 respetiva definição do objeto.

```
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 (Código 33) é apresentado um exemplo *do for each* na linguagem *Java*.

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

…
for (Conta conta : contas) {
 conta.somaValores();
}

..
```

Código 33 - *foreach* 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/ideia para que os utilizadores não sintam grandes mudanças nesse aspeto. De seguida (Código 34) é apresentado o mesmo exemplo, mas desta vez utilizando a *framework DPF4j*.

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

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

Código 34 - *foreach* 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 fator 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 forma 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 à 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* é 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 respetivo 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 núcleo 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 uma 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.

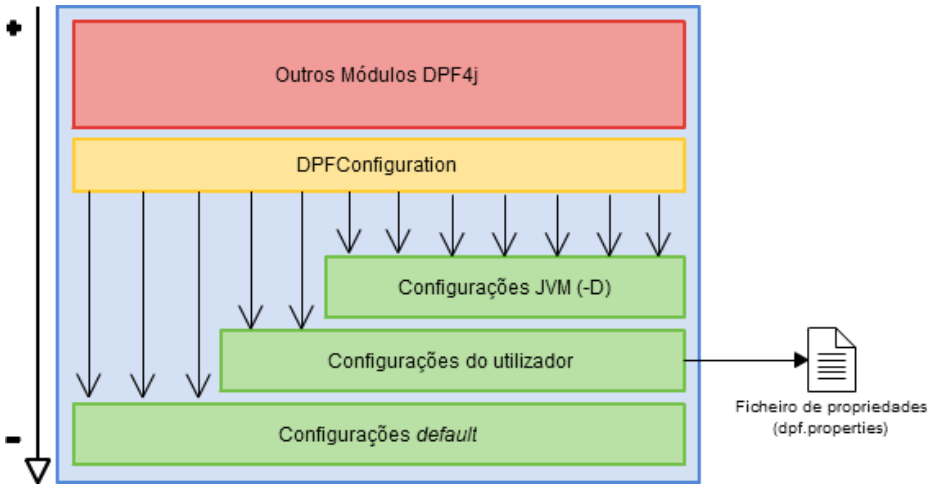

Figura 20 - Diagrama de prioridades da configuração da *framework DPF4j*

A hierarquia de configuração é a seguinte (da mais prioritária para a menos prioritária tal como é apresentado na Figura 20:

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* [22] 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 [23]. 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 descobrir desse grupo.

```
# workgroups activos
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 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 respetivas 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 respetivo arranque e encerramento dos serviços utilizados no sistema. No entanto ele já foi desenvolvido com a intenção de no futuro integrar uma interface *JMX* [24] para gestão remota do funcionamento do *DPF Daemon*. *JMX* é uma tecnologia que permite criar soluções modulares para monitorização e 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 cativação 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 21):

```
<id do nó>;<tempo (ms)>;<categoria>;<acção>-<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 21 - 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 efetuada e levou à escrita dos detalhes.

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

Identidade [5]: Identidade do objeto sobre o qual estava a ser efetuada 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 efetuada (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 objeto 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 esta foi testada em *Ubuntu*, *Mac OS X*, *Microsoft Windows* 7 e *Microsoft Windows* 8.

Na Figura 22 está apresentada a estrutura do diretório da distribuição da *framework*:

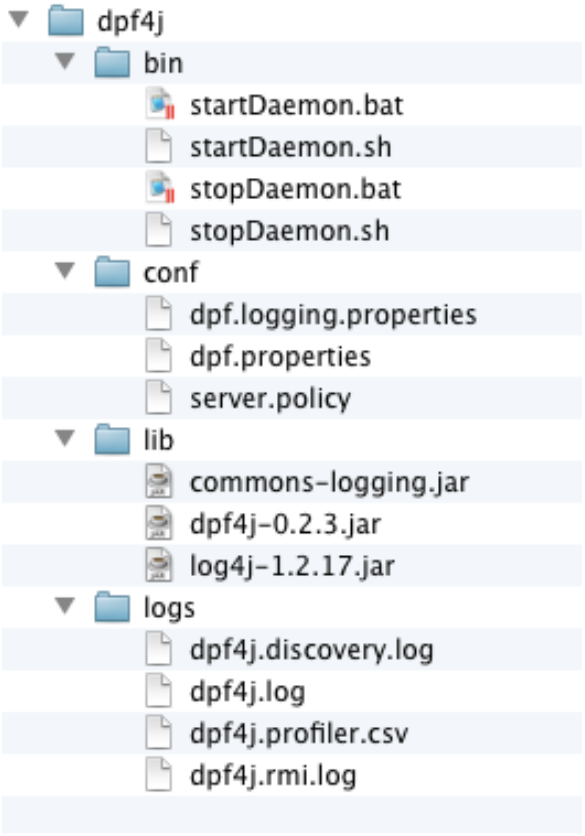

Figura 22 - Diretório da distribuição *DPF4j*

Em seguida é feita uma breve descrição de cada diretório pela ordem que se encontra na Figura 22.

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 Distribuição e Execução de Trabalho

Um ponto fulcral na *DPF4j* é a capacidade de distribuir e executar trabalho. Para isso a *DPF4j* define formas de como o programador deverá criar uma tarefa. Este capítulo irá abordar o formato dessas tarefas e depois como podem ser introduzidas no sistema de escalonamento e distribuição. Também existem formas de diminuir o tráfego de rede recorrendo a objetos de dados que podem ser reutilizados, e finalmente será abordada a implementação do escalonador, a forma como o trabalho é delegado para outras máquinas e como é que o código é propagado pelos vários nós.

7.1 DPFTask

Tal como na implementação do *fork-join* incluída na versão 7 do *Java*, foi desenvolvido um formato de representação de tarefa para as execuções distribuídas e paralelas, ao qual se chamou de *DPFTask* (em contraste com a *ForkJoinTask*).

Os grandes candidatos para o formato de tarefa seriam as *interfaces Runnable* e *Callable*, mas esta última tem mais potencialidade devido ao método *call* ser muito genérico e pode ser utilizado para a maior parte das situações, este método pode retornar qualquer tipo de objeto e lançar exceções.

```
public interface Callable<V> {
    V call() throws Exception;
}
```

Código 37 - Interface *Callable*

A interface *Callable* (Código 37) surgiu no *Java* 5 juntamente com as novas funcionalidades de paralelismo, sendo assim compatível com os executores lançados nessa versão e posteriores. Isto dá muitas possibilidades ao programador visto que poderá facilmente migrar sistemas já existentes para utilizarem *DPF4j* ou utilizar as tarefas deste tipo não só com a *DPF4j* de forma a distribuir trabalho, mas também usar as mesmas implementações com as classes já existentes do *JRE* para as executar de forma paralela, agendada, etc.

```
public interface Runnable {
    void run();
}
```

Código 38 - Interface *Runnable*

No caso da *interface Runnable* (Código 38), e mais propriamente o método *run()*, iriam surgir vários problemas no que toca a distribuir trabalho, esta interface é muito mais limitada, não tendo retornos nem podendo lançar exceções, todo o trabalho feito por este método só poderia ter algum impacto se o seu resultado for escrito em algum tipo de memória. Num ambiente distribuído o armazenamento de dados constitui um problema porque não existe memória (RAM) partilhada no *Java standard*. Se algum objeto for alterado na máquina remota, este não será sincronizado com a máquina local. O facto do *Callable* ser uma interface para especificar tarefas com retorno pode ser a solução para transferir o resultado de uma tarefa entre máquinas diferentes, mantendo aberta a possibilidade de que as *DPFTasks* sejam também executadas por uma série de executores já existentes que aceitem objetos deste tipo (Callable), nos quais se incluem os objetos do tipo *ExecutorService* cuja importância irá ser explicada na abordagem ao *DPF Scheduler*.

Tendo em conta as vantagens da interface *Callable* sobre a interface *Runnable, uma DPFTask* não é nada mais que um *Callable* serializável.

```
public interface DPFTask<V> extends Serializable, Callable<V> {}
```

Código 39 - Interface *DPFTask*

Como se pode ver no Código 39, as classes do tipo *DPFTask* serão *Callables* com retorno do tipo *V* (genérico, a definir pelo programador) e são serializáveis dada a necessidade que estes objetos têm de ser transferidos entre máquinas.

Podemos assim concluir que a *DPFTask* é o formato ideal para quando se quer executar algo e obter um retorno. No entanto, sendo o *Java* uma linguagem orientada a objetos, muitas vezes o resultado de uma tarefa não é um retorno mas sim a transformação da própria instância, isto é, em vez de existir um método que retorna o resultado, os atributos da instância na qual está incluído o método de processamento são modificados. Os objetos do tipo *Runnable*, que como foi visto anteriormente não têm retorno, usam muitas vezes este tipo de abordagem. Num ambiente distribuído não existe memória partilhada, logo, a menos que exista algum tipo de sincronismo, uma alteração de uma instância remota não será refletida na instância local. A DPF4j não inclui nenhum sistema de sincronismo de objetos mas consegue amenizar este problema através de um tipo de tarefa chamada *DPFRunnable*.

A *DPFRunnable* é uma classe abstrata que representa um tipo especial de *DPFTask* que é em simultâneo um *Runnable* e uma *DPFTask*.

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

Código 40 - Interface *DPFRunnable*

Como é possível ver na definição anterior (Código 40), o *DPFRunnable* implementa a interface *Runnable* e portanto irá obrigar o programador a implementar o método *run(),* outra característica é que esta classe implementa uma *DPFTask* que terá como retorno um *DPFRunnable*. Isto é porque esta classe é uma *DPFTask* que irá executar o método *run()* e irá se retornar a si própria quando terminar a sua execução (Figura 23, Código 41).

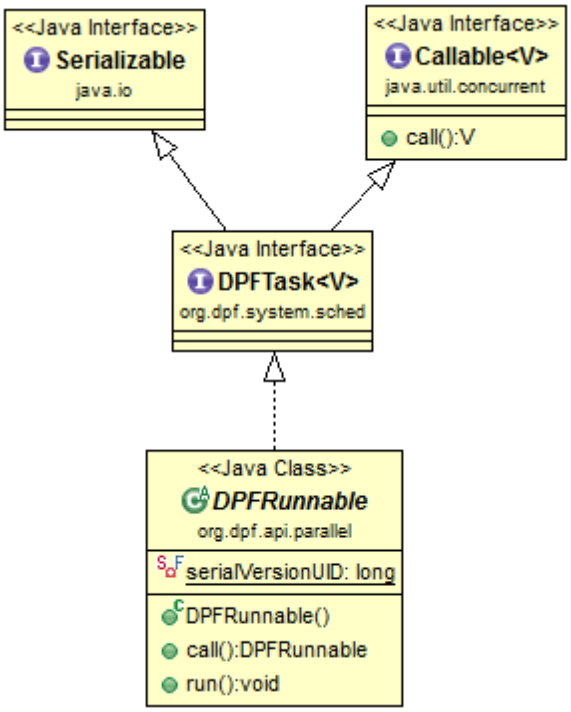

Figura 23 - Hierarquia da classe *DPFRunnable*

```
public DPFRunnable call() throws DPFRunnableException {
	run();
	return this;
}
```

Código 41 - Método *call()* da classe *DPFRunnable*

Isto permite a um programador submeter uma tarefa, e obter a instância da mesma como resultado, no entanto, qualquer sincronização que seja desejada por parte do programador, terá de ser implementada por ele, copiando os atributos da instância resultado para dentro da instância local.

As transferências de *DPFTasks* e respetivos resultados estão protegidas recorrendo a *DPFPackets* que serão detalhados na secção de segurança.

Do ponto de vista de escalonamento uma *DPFTask* passa por três estágios no seu ciclo de vida (Figura 24):

1. Em escalonamento ou espera: Fase em que a tarefa terminou de ser submetida para o escalonador e está à espera de ser atribuída a um executor;

2. Em comunicação (apenas tarefas executadas remotamente): Fase que engloba os procedimentos de serialização, encriptação, desencriptação e desserialização da tarefa e ainda o tempo em que a tarefa se encontra a ser transferida;

3. Em execução: Momento em que está a ser executado o método *call()* da *DPFTask*.

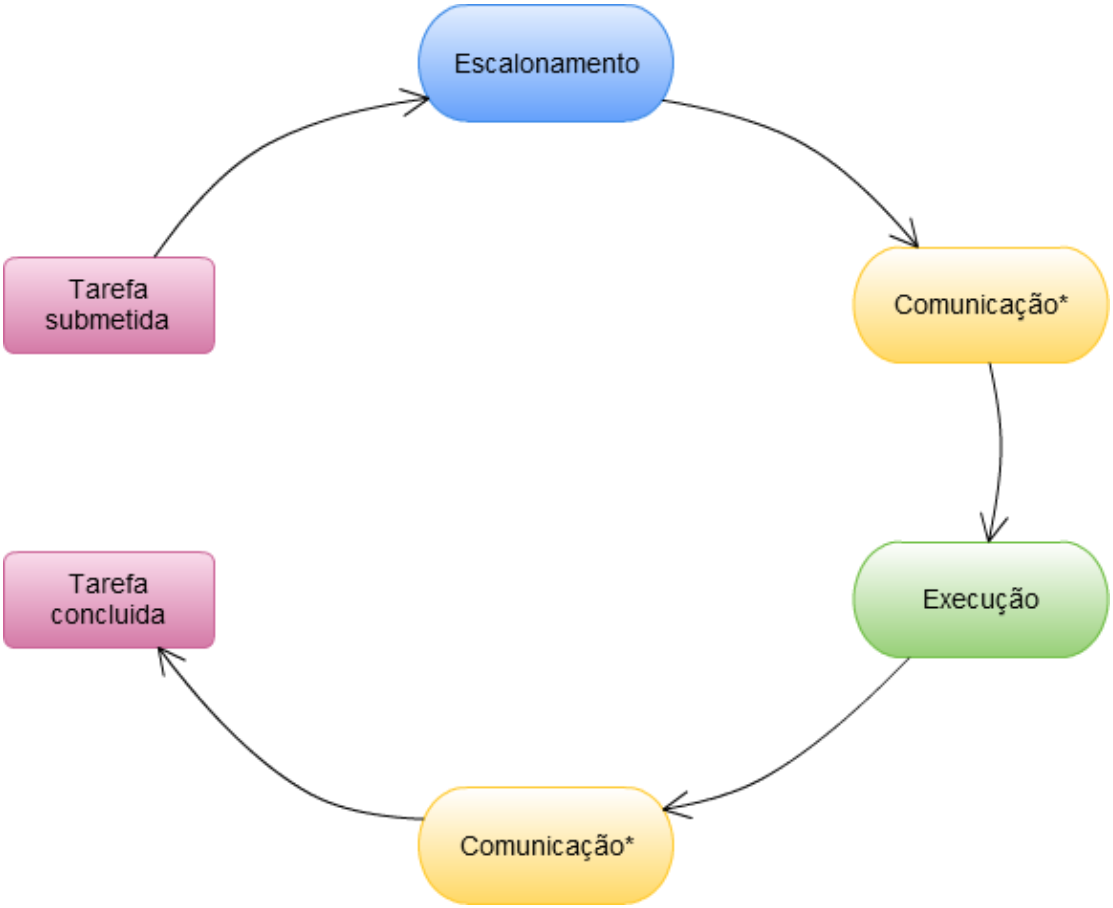

Figura 24 - Ciclo de vida de uma *DPFTask*

*aplicável apenas a tarefas executadas remotamente

7.2 DPFReusableObjects

Sempre que uma *DPF Task* é serializada, todo o seu conteúdo é serializado. Este é o mecanismo que a *DPF4j* tem para transferir dados entre máquinas, ou seja, todos os dados em memória que são necessários para a execução de determinada tarefa devem estar referenciados pela instância da *DPFTask* para assim serem serializados juntamente com esta. Isto cria um problema inevitável quando se está a trabalhar com grandes quantidades de dados devido à sua transferência por rede. No entanto, este problema pode ainda ser agravado quando estes objetos são utilizados em alguma API representativa de um ciclo que pode obrigar um objeto de grandes dimensões a ser transferido múltiplas vezes.

Por exemplo, imaginemos uma operação repetitiva sobre um *array* bidimensional de dimensões significativas, na ordem dos milhares de linhas por milhares de colunas. No caso de serem criadas várias *DPFTasks* sobre este *array*, potencialmente ele iria ser serializado várias vezes, transferido várias vezes, e desserializado várias vezes. Isto significa que possivelmente o *array* iria ser enviado várias vezes para a mesma máquina remota, e essa máquina iria desserializar várias vezes esse *array*. Esta situação ainda agrava mais o problema devido à

desserialização em Java gerar sempre um novo objeto, característica esta que é muitas vezes utilizada pelos programadores de Java para clonar objetos não clonáveis (serializando, e desserializando de seguida o objeto em questão), e no caso de dados de grandes dimensões pode consumir grandes quantidades de memória replicando o objeto e em casos extremos provocando um *OutOfMemoryError*. O *OutOfMemoryError* é um erro que ocorre quando a máquina virtual não consegue alocar um objeto por não ter memória suficiente, e o *garbage collector* não conseguir libertar mais.

Para combater este problema, foi criado o conceito *DPFReusableObjects*. Um *DPFReusableObject* será um embrulho (*wrapper*) que poderá referenciar qualquer objeto e garantir que este objeto apenas é enviado uma vez para a máquina remota.

No caso de um programador querer reutilizar um objeto que poderá ser utilizado por várias DPFTasks, de forma a aumentar a sua performance, deverá alterar a sua *DPFTask* para em vez de referenciar o objeto em questão diretamente, o armazenar dentro dum DPFReusableObject (Código 42).

```
DPFReusableObject arrayEmbrulhado = new DPFReusableObject(array);
String[][] arrayX = arrayEmbrulhado.getObj();
```

Código 42 - Embrulhar e desembrulhar *array*

Este objeto contém três atributos, o objeto embrulhado, que é *transient,* ou seja, não será serializado, o identificador do nó de origem, e o identificador do objeto (Figura 25). Assim que um destes objetos é criado, este é automaticamente guardado num repositório local. Este repositório utiliza *weak references*, isto é, assim que não houver mais referências para o objeto este será excluído no ciclo do *garbage collector*. Enquanto houver tarefas ativas que usem o objeto o *garbage collector* não o irá eliminar porque a instância do *DPFReusableObject* irá segurar uma referência deste.

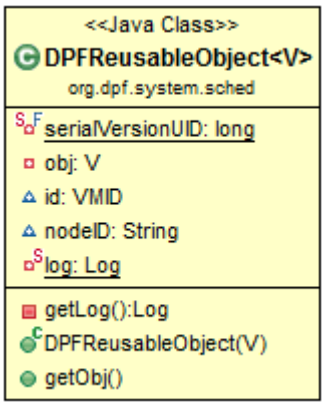

Figura 25 - Classe DPFReusableObject

Quando este objeto é enviado dentro duma *DPFTask*, este irá ser serializado e o objeto de dados (atributo "obj" na Figura 25) será perdido devido à sua natureza *transient*. A máquina remota irá tentar desembrulhá-lo através do método *getObj*() que ao verificar que o objeto de dados não está na instância irá a um repositório local procurar pelo objeto através do seu ID e

nó de origem e no caso de não o encontrar, irá pedir o objeto ao nó de origem que o fornecerá através duma consulta ao repositório mencionado anteriormente.

Na máquina remota, os objetos também são armazenados com *weak reference* o que significa que se uma máquina estiver muito tempo sem receber uma tarefa sobre o dito objeto, este irá automaticamente ser destruído.

As transferências de *DPFReusableObjects* estão protegidas recorrendo a *DPFPackets* que serão detalhados na secção de segurança.

7.3 DPF Scheduler

O *DPF Scheduler* é o módulo responsável por escalonar e distribuir as várias *tasks* que vão sendo criadas pela *framework*. Por omissão, todas as *tasks* executadas pela *framework*, são tratadas pelo *DPF Scheduler* referenciado pelo *DPF System*. Por sua vez, o *DPF Scheduler*, irá decidir se a tarefa deve ser executada localmente, remotamente ou mesmo se deverá esperar até existirem recursos disponíveis.

É chamado de *DPF Scheduler* ao escalonador que está referenciado pelo *DPF System*. Numa aplicação um programador pode instanciar quantos escalonadores quiser, pode fazer as suas implementações e pode submeter trabalho para eles como desejar, mas por omissão a *framework* utilizará sempre o escalonador referenciado pelo *DPF System*.

O *DPF Scheduler* obedece à interface *ExecutorService* do package *concurrent* do *JRE (java.util.concurrent)*, isto permite que facilmente um programador possa implementar um novo *scheduler* e possa substituir o *scheduler* utilizado pela *framework*, ou utilizar uma das existentes implementações de *ExecutorService* do *JRE* (que não irá distribuir trabalho). Para alterar o escalonador da *framework* por outra implementação basta para isso substituir a referência presente no *DPF System* por uma instância de qualquer classe que implemente a interface *ExecutorService* (Código 43).

```
DPFSystem.getInstance().setDPFScheduler(meuEscalonador);
```

Código 43 - Substituição do *DPF Scheduler*

É de notar que qualquer tarefa já submetida, continuará a executar segundo o escalonador anterior a esta redefinição.

Da interface *ExecutorService* (Figura 26) são de referenciar os métodos de execução *submit*, *invokeAll* e *invokeAny*.

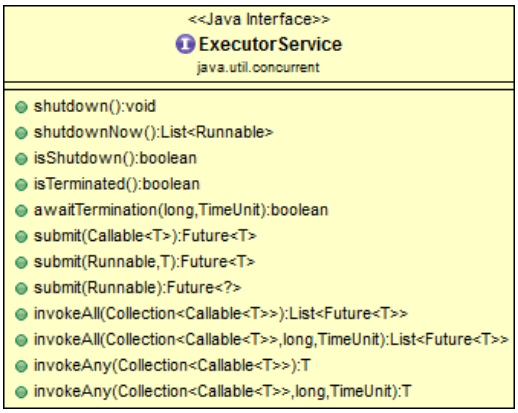

Figura 26 - Interface ExecutorService

O método *submit* recebe um *Callable* para ser executado de forma assíncrona e devolve um Future. O Future é um objeto muito importante no que toca a execuções assíncronas, este objeto é uma representação do que será o resultado de uma tarefa em execução, é a partir deste objeto que é possível saber quando uma tarefa terminou e obter o seu resultado. A implementação da classe do tipo Future é ainda responsável por fazer a espera passiva pela conclusão da tarefa e bloquear a execução da *thread* que requisitou o resultado (método *get()*). No caso de existir alguma exceção na execução da tarefa, uma exceção a encapsular esta será lançada no momento do método *get*().

O método *invokeAll* serve para fazer execuções síncronas (a *master thread* espera pelo resultado), este método recebe uma coleção de *Callables* e retorna uma lista de resultados.

Relativamente ao método *invokeAny* (também síncrono), este tem uma utilização muito mais particular, num ambiente distribuído pode ser especialmente interessante para algoritmos de pesquisa visto que recebem múltiplas tarefas, mas devolvem apenas o resultado da tarefa que terminar primeiro. Este método trata-se de um mecanismo de paralelização especulativa, como foi demonstrado no estudo ao *Ateji PX*. No que toca a este método é de ter em conta que o algoritmo de escalonamento utilizado atualmente na *framework*, que será apresentado mais à frente, dá preferência à execução local. Caso o ambiente de execução seja homogéneo e as várias tarefas do *invokeAny* sejam iguais, há uma tendência da máquina local terminar antes de qualquer outra e resultar assim num desperdício de recursos ao enviar a tarefa para outras máquinas.

Nesta fase a *framework* tem apenas uma implementação de escalonador, chamada de *DPFSimpleScheduler*.

7.4 DPFSimpleScheduler

O *DPFSimpleScheduler* é a primeira implementação de um *scheduler* para a DPF4j. Como explicado anteriormente o *DPFSimpleScheduler*, é composto por uma fila de tarefas, uma *ThreadPool* (com a sua própria fila) e um *Task Delegator*. A Figura 27 demonstra os componentes do *DPFSimpleScheduler* por onde as *DPFTask* passam (filas de espera, e executores).

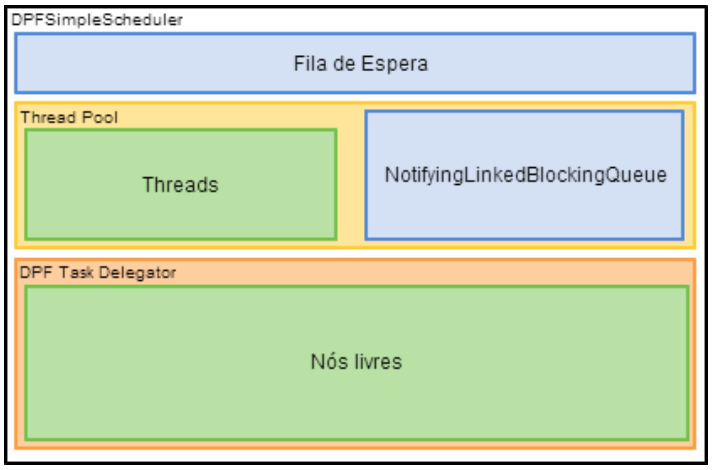

Figura 27 - Componentes DPFSimpleScheduler

Como explicado anteriormente, o *DPFSimpleScheduler* é um *ExecutorService*, e como tal tem que devolver *Futures* representativos dos resultados das *tasks* escalonadas por este, ou seja, assim que uma tarefa é submetida tem que ser imediatamente devolvido um *Future*.

O tipo de *Future* devolvido pelo *DPFSimpleScheduler* é uma implementação chamada "*FutureWrapper*". Devido a este escalonador lidar com duas entidades que já devolvem *futures* (*ThreadPool* e *TaskDelegator*), foi implementado o *FutureWrapper* que encapsula os *Futures* devolvidos por uma dessas entidades. A necessidade desta implementação dá-se pelo facto que estes *Futures* só são criados no momento em que a tarefa deixa de estar num estado de escalonamento ou espera e entra num estado de execução no qual o seu executor já é conhecido. Este *FutureWrapper* pode no entanto ser devolvido mal a tarefa entra em escalonamento e só dá a tarefa como concluída após um Future ter sido atribuído a si e esse tiver o resultado.

O funcionamento do escalonador é garantido por uma série de *threads* (quantidade configurável) que retiram tarefas presentes na fila de espera e de seguida executam um algoritmo de escalonamento. Estas *threads* encontram-se num estado de espera enquanto a fila está vazia, ou não existem recursos disponíveis, no entanto é possível forçar as *threads* a periodicamente acordarem para verificar se existem tarefas para executar (*ForceWakePeriod*). Os eventos que desbloqueiam as *threads* de escalonamento para executarem o algoritmo de escalonamento são as seguintes:

- Chegada de tarefa;
- Notificação de recursos livres por parte de um executor (*NotifyingLinkedBlockingQueue* ou *TaskDelegator*);
- Fim de período (*ForceWakePeriod*).

O algoritmo de escalonamento utilizado pode ser descrito através do fluxograma que se segue (Figura 28):

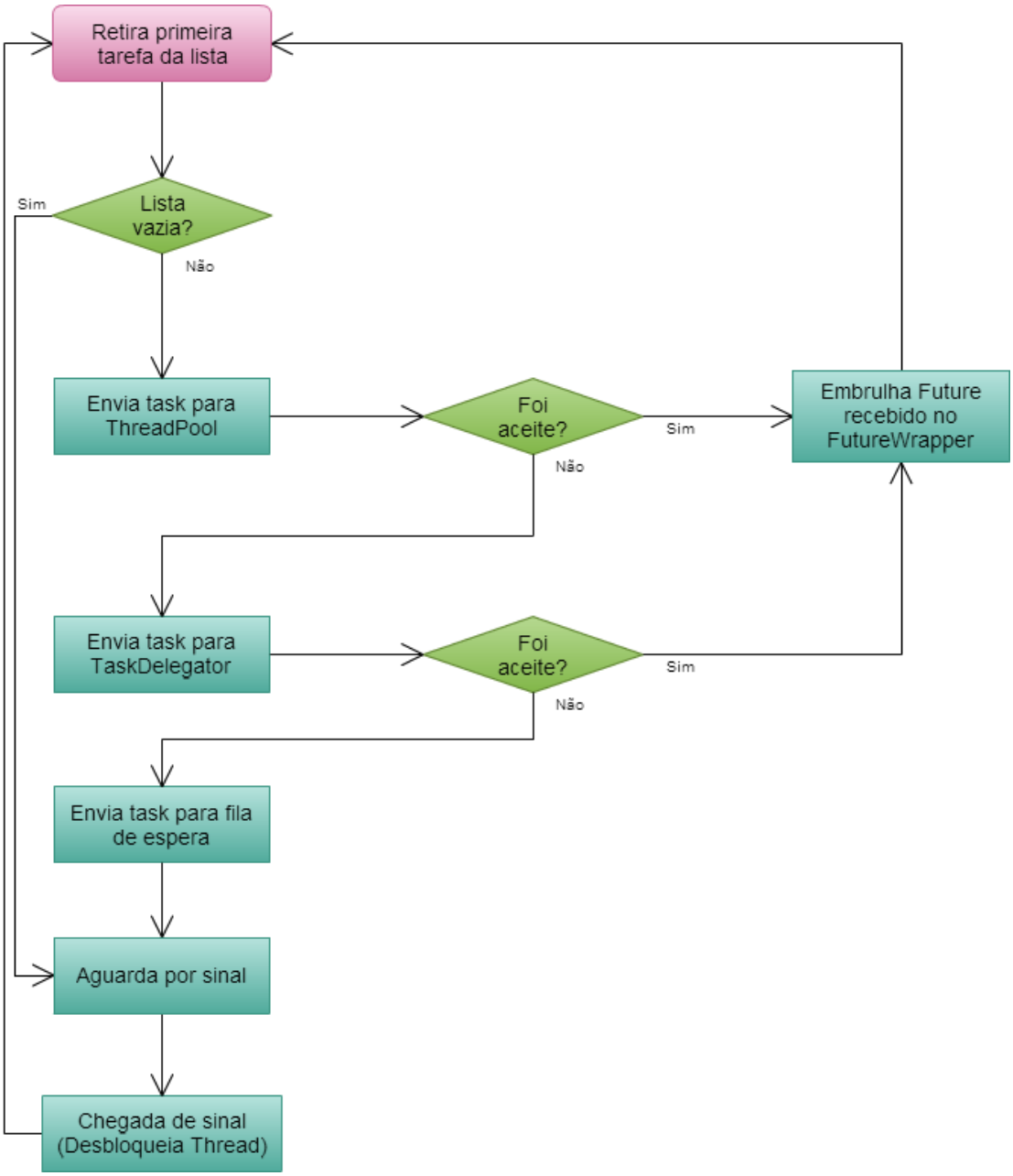

Figura 28 - Fluxo de uma *thread* de escalonamento

Sempre que uma nova *task* é retirada da fila de espera, o escalonador irá enviá-la para a *ThreadPool*. A *ThreadPool* irá colocar a tarefa numa fila de espera local, que nesta implementação tem por omissão um tamanho igual ao dobro do número de processadores físicos da máquina. Assim que uma *thread* fique livre, esta irá fazer "*work stealing*" desta fila. Esta fila encontra-se nesta zona do processo de escalonamento para garantir que os processadores locais nunca ficam desocupados com tarefas a serem delegadas para nós remotos (que por norma seria ineficiente).

Assim que chega um número de tarefas capazes de ocupar todas as *threads* e as várias posições da fila local, as tarefas começam a ser enviadas para o *Task Delegator* que é responsável por as enviar para nós remotos.

O *Task Delegator* é basicamente o escalonador dos nós remotos. O *Task Delegator* é o módulo responsável por enviar uma *task* para ser executada remotamente, este artefacto lida com todos os nós conhecidos (é um *listener* do *DPFRegistry*). O Task Delegator tem uma lista de nós livres, e um limite de *tasks* que cada nó poderá executar simultaneamente e assim definir o que é um nó não livre.

Quando uma tarefa chega ao *TaskDelegator* este serializará a mesma e envia-a ao primeiro nó livre da lista, e no caso de o nó atingir o seu limite de tarefas simultâneas este será removido da lista de nós livres. No final da tarefa o nó voltará à lista de nós livres e os *listeners* do *TaskDelegator* (pe. *DPFSimpleScheduler*) serão avisados de que existem recursos livres para assim aceitar mais trabalho (Figura 29).

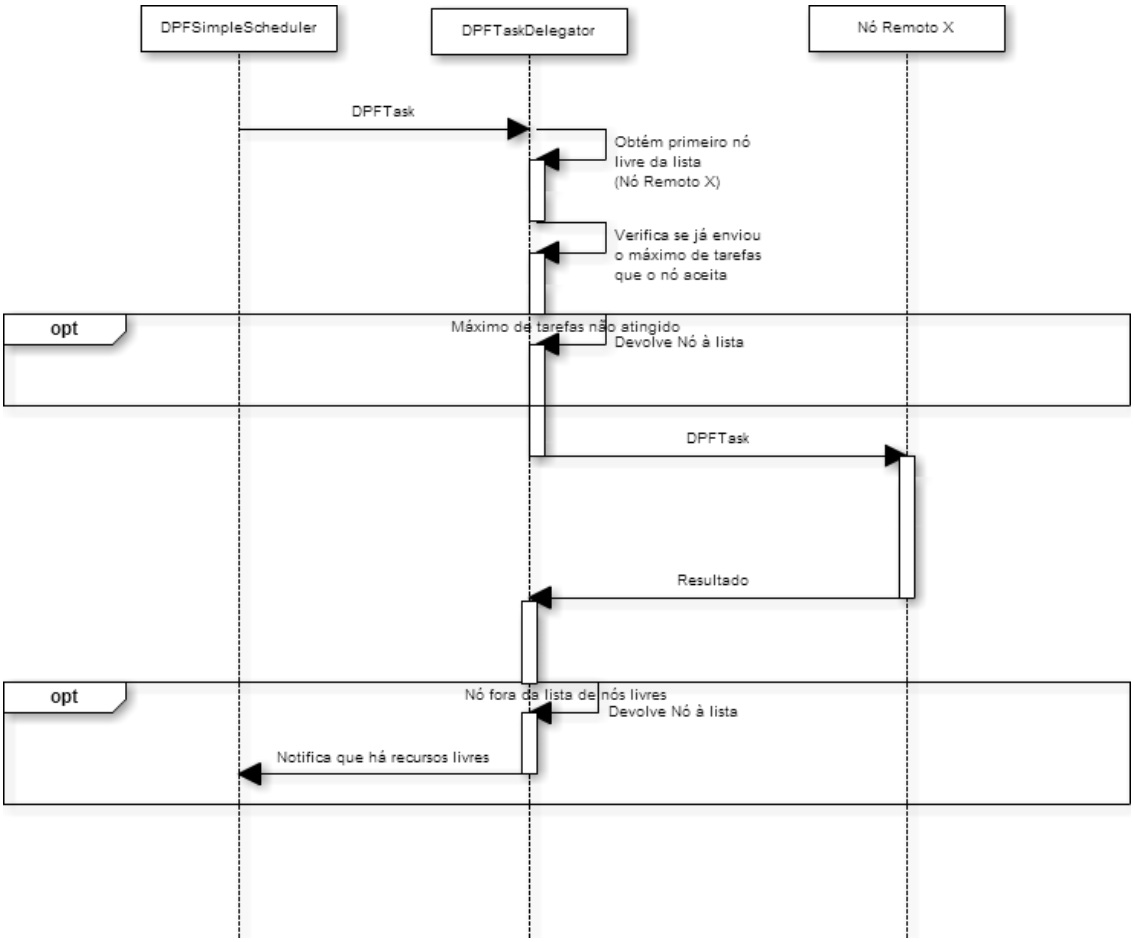

Figura 29 - Diagrama de sequência do *Task Delegator*

Finalmente, se todos os nós estiverem ocupados, as tarefas serão colocadas na fila de espera global e a *thread* ficará em espera até receber um sinal para acordar. Como é visível no fluxograma, não existe "passo" de fim, pois este algoritmo repete-se até que o sistema seja desligado.

7.5 ThreadPool

A *ThreadPool* utilizada pelo DPFSimpleScheduler, é uma implementação *java.util.concurrent. ThreadPoolExecutor,* e é composta por uma *pool* de *threads*, com uma capacidade igual ao número de processadores físicos da máquina e uma fila bloqueante com o dobro dessa capacidade. A fila bloqueante é uma fila que bloqueia quando é feito um *pull* sobre ela e esta se encontra vazia. A *ThreadPool* pode ter o comportamento que se desejar na ocasião de chegar uma tarefa e a capacidade desta estiver lotada, o comportamento por omissão, e utilizado neste caso, é recusar a tarefa, o que indicará ao *scheduler* que este recurso está ocupado através de uma *RejectedExecutionException*.

O número de *threads* é igual ao número de processadores físicos e este número é suficiente para o Java conseguir distribuir o trabalho pelos vários processadores, de forma a utilizá-los ao máximo, quanto à fila com o dobro deste valor foi uma decisão tomada porque se parte do princípio que a execução local é mais rápida que a execução remota, devido à não necessidade de serialização, encriptação e transferência, e é uma forma de garantir que os processadores locais nunca ficam desocupados à espera de trabalho remoto.

O funcionamento da *ThreadPool* no contexto do *DPFSimpleScheduler* pode ser facilmente explicado através do seguinte diagrama de sequência (Figura 30):

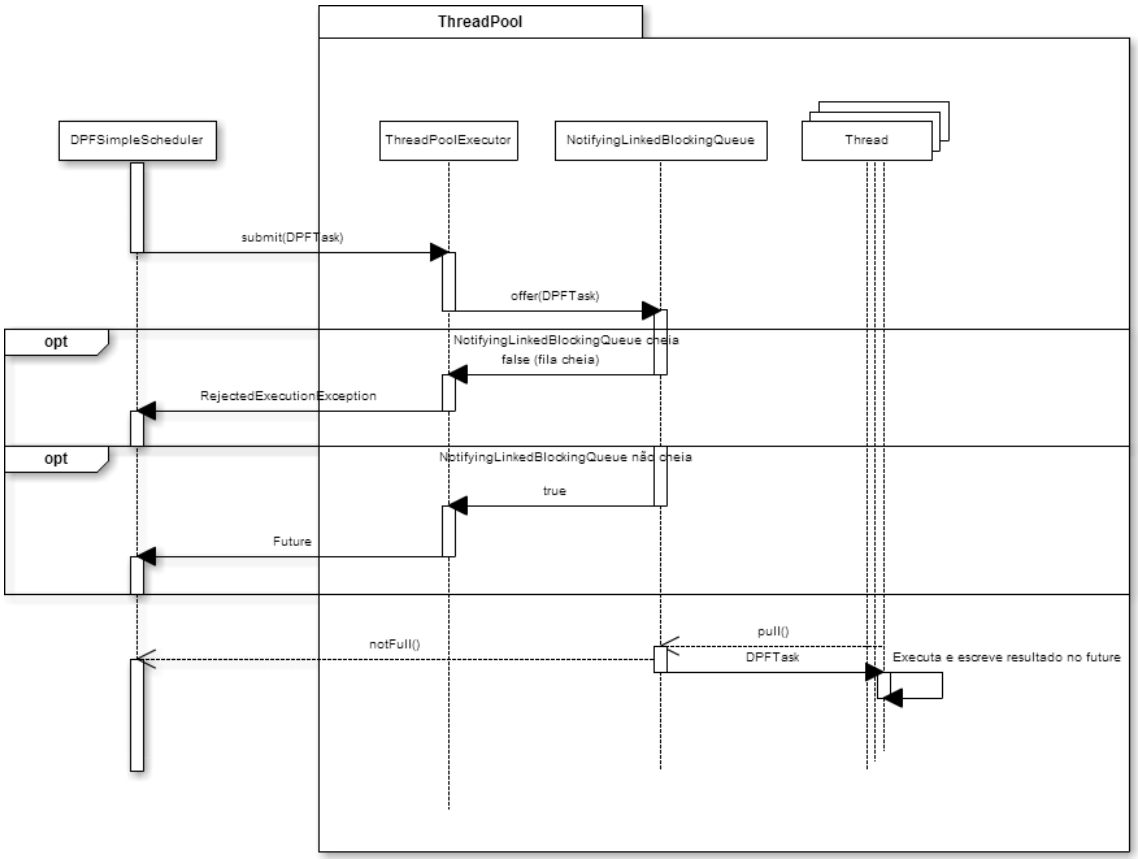

Figura 30 - Execução de uma tarefa por parte da *ThreadPool*

A *ThreadPool* aceita que lhe seja atribuída uma fila bloqueante para ser utilizada em vez da sua fila por omissão, para a *DPF4j* foi implementada uma fila bloqueante chamada

NotifyingLinkedBlockingQueue que aceita o registo de *listeners* que são avisados assim que a fila deixe de estar cheia, informando, assim, que passam a existir recursos livres.

Assim que chega uma tarefa, esta é guardada na fila bloqueante, no caso de a fila estar cheia, a tarefa será rejeitada, caso contrário, esta será aceite e um Future será devolvido.

A tarefa ficará então em fila de espera até que uma *thread* fique livre para a executar. Nesse momento, uma *thread* irá à fila buscar uma nova tarefa, o que irá despoletar a ação da fila notificar todos os seus *listeners* de que não se encontra cheia. Finalmente a *thread* irá executar a tarefa, e o resultado desta irá ser atribuído ao Future criado inicialmente.

Se por acaso uma *thread* ficasse livre e não houvesse tarefas na fila, a *thread* iria ficar num estado de *wait* assim que fizesse *pull* à fila bloqueante.

7.6 DPF TaskDelegator

O *DPFTaskDelegator* é o módulo responsável por enviar as *DPFTasks* para os outros nós para serem executadas, pode ser visto como um outro escalonador, que escalona o trabalho entre os vários nós. O *DPFTaskDelegator* também tem a sua própria *ThreadPool* mas esta não tem limites na quantidade de trabalho que aceita porque as suas *threads* passam a maior parte do tempo num estado *wait*, o trabalho destas *threads* limita-se ao envio das *tasks* para nós remotos e esperar pelo seu resultado fazendo muito pouco processamento.

A nível de escalonamento o *DPFTaskDelegator* possui uma fila que inclui todos os nós disponíveis, um nó é considerado não disponível quando está a executar um certo número de *DPFTasks* em simultâneo. Esta fila é abastecida no arranque da aplicação, e depois, através de um *listener* que o *DPFScheduler* tem no *DPFRegistry*, este consegue saber quando um novo nó se liga ou desliga e pode assim ir atualizando a sua fila. A fila dos nós disponíveis, é a fila utilizada para escalonar trabalho, os nós são retirados desta lista um a um à medida que vão sendo enviadas tarefas e no caso destes nós ainda terem capacidade para aceitar mais são devolvidos ao fim da fila, caso contrário ficam fora desta até terminarem uma das tarefas em execução. A capacidade de um nó é determinada por dois limites, o limite de tarefas por nó remoto definido na máquina local, e o máximo de tarefas que o nó aceita definido na máquina remota, assim que um destes limites é atingido o nó é dado como não disponível. Definir que um nó tem um limite de zero tarefas, é definir um nó que apenas delega trabalho, isto é, um nó que nunca realizará trabalho para outros nós (modo especialmente interessante para *thin-clients*).

O envio de uma tarefa é feito através da serialização desta, criação de um *DPFPacket*, e invocação do serviço de execução no nó remoto.

O nó remoto limitar-se à a desencriptar e desserializar a tarefa usando a chave do *workgroup*, executar o método *call()* e devolver o seu resultado dentro de outro *DPFPacket*. Este resultado é finalmente colocado dentro no *Future* correspondente que desbloqueará qualquer *thread* que esteja á espera dele.

Este fluxo pode ser resumido à seguinte figura (Figura 31):

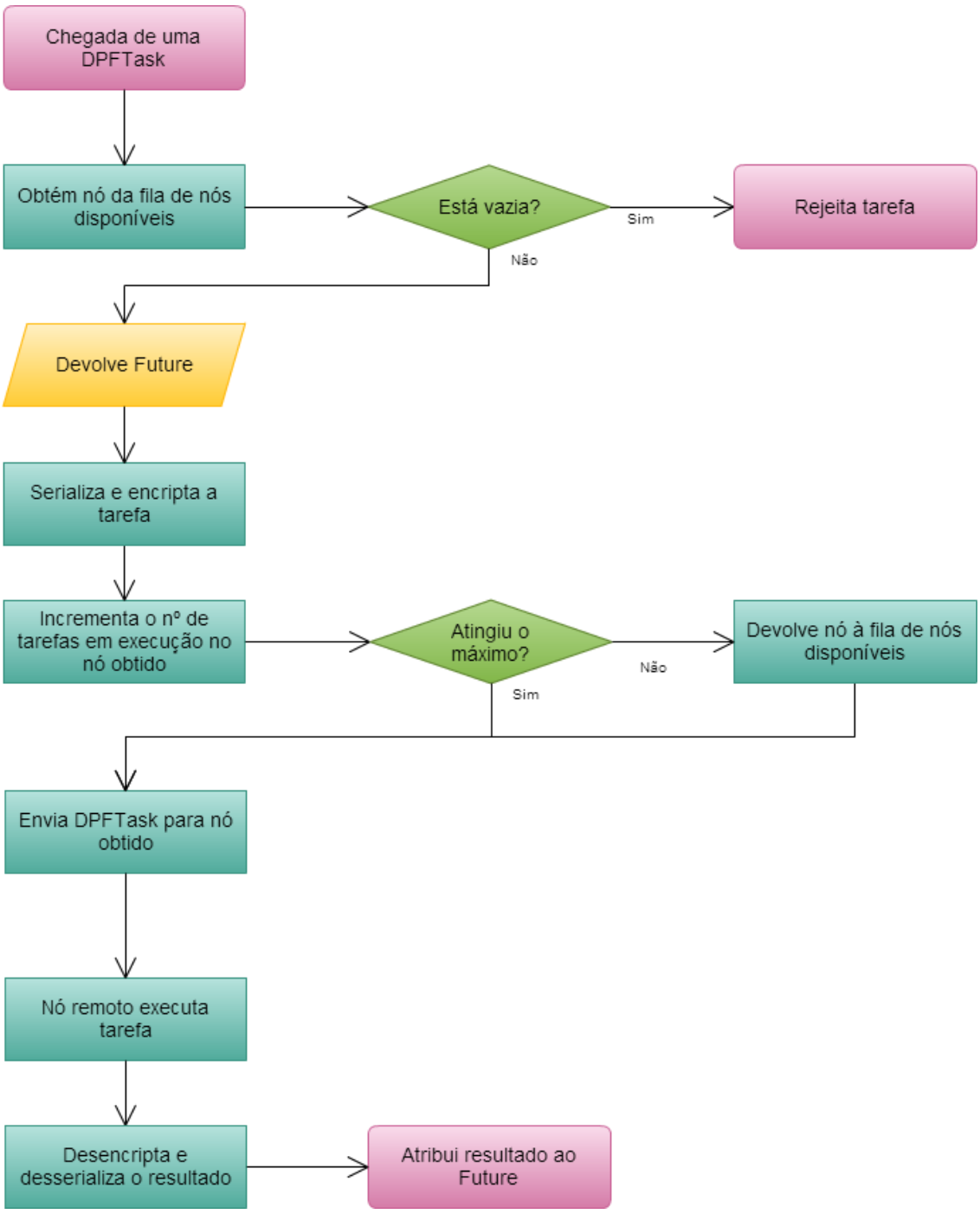

Figura 31 - Fluxo de execução de uma tarefa no *TaskDelegator*

7.7 Configuração

De seguida apresentam-se as propriedades de configuração do DPF *Scheduler*, com respetiva descrição e valor por omissão:

Propriedade	Descrição
`dpf.scheduler.executorThreads`	Número de *threads* na *ThreadPool*
	Default: nº processadores
`dpf.scheduler.schedulerThreads`	Número de *threads* do escalonador
	Default: 3
`dpf.scheduler.forceWakeTimer`	Período máximo que uma *thread* do escalonador pode estar em espera (ms)
	Default: 60000
`dpf.scheduler.disableRemote`	Desativar a execução remota
	Default: *false*
`dpf.scheduler.disableLocal`	Desativar a execução local
	Default: *false*

Tabela 7 - Configurações do DPF Scheduler

7.8 Segurança

No que se refere à segurança na transferência de dados e tarefas, a *DPF4j* limita-se a garantir a privacidade do *workgroup*, e a garantir que não é executado código proveniente de *workgroups* desconhecidos ou de nós mal autenticados.

Esta segurança é garantida através do uso de uma chave partilhada. Todos os nós, pertencentes ao mesmo *workgroup*, devem na sua configuração ter definido os *workgroups* a que pertencem e as respetivas chaves. Estas duas informações juntas (nome do *workgroup* e chave) são a base de toda a confiança entre os dois nós.

Para envio e receção deste tipo de dados (tarefas, resultados, *DPFReusableObjects*) são utilizados objetos de nome *DPFPackets*.

O *DPFPacket* consiste num objeto que inclui o nome de um *workgroup* comum aos dois nós, a identificação do nó que criou o *DPFPacket* e um *array* de *bytes*, que consiste na serialização encriptada do objeto *DPFTask*.

A encriptação dos objetos serializados é feita através do algoritmo *AES* [25] com uma *hash* gerada a partir da chave do *workgroup*.

Este objeto é então enviado para o nó remoto através de um serviço *RMI* (por exemplo o de execução de tarefas) e o nó remoto irá desencriptar e desserializar o objeto usando a chave do *workgroup* cujo nome vem indicado no *DPFPacket*. Se esta desencriptação não funcionar significa que, ou a chave introduzida localmente está errada, ou então a chave que o nó remoto utilizou está errada. Em qualquer um dos casos, o pacote será descartado e será lançada uma *RemoteException* para o nó invocador.

Visto os *DPFPackets* usarem um algoritmo de chave partilhada, não garantem apenas que o nó envia a tarefa pertence ao *workgroup*, como também protege a tarefa, seus dados e resultado de possíveis atacantes que estejam à escuta, é de notar no entanto que qualquer nó que pertença ao *workgroup* conhece esta chave e seria capaz de desencriptar qualquer um destes pacotes.

Numa rede fechada e segura, de forma a aumentar a performance do sistema, é possível introduzir uma chave vazia e os pacotes não serão encriptados.

8 Distribuição de Código

Quando se desenvolve e pretende utilizar uma aplicação *DPF4j* apenas é necessário instalá-la numa máquina para poder utilizar as potencialidades da *framework*, isto supondo que existirem mais máquinas com a *framework* base instalada. Esta característica torna possível utilizar a *framework* facilmente ainda mesmo durante o desenvolvimento e *debugging* pois torna possível a execução de qualquer aplicação a partir do ambiente de desenvolvimento do programador sem a necessidade de propagar e instalar o código desenvolvido pelas várias máquinas envolvidas no processo.

Para tornar esta funcionalidade possível é necessário que o código compilado (*bytecode*) seja transferido de umas máquinas para as outras. Todas as tarefas da *framework* são representadas por classes que implementam a interface *DPFTask* e o *bytecode* destas classes tem de ser enviado para as máquinas que estão a executar as tarefas para estas serem capazes de o fazer.

Com a DPF4j existem três formas de distribuir código entre máquinas:

1. Manual: Esta é a forma normal do Java funcionar. O utilizador distribui os *jars* e *classes* manualmente pelas várias máquinas e garante que estas são carregadas no *classpath* inicial da aplicação;

2. Automática: Quando uma máquina precisa duma classe que não tem para executar determinada tarefa, esta pede-a à máquina que submeteu a tarefa;

3. Forçada: Na configuração do *workgroup* é possível colocar uma lista de *URLs* com o *classpath* que deve ser enviado para as máquinas remotas após a associação com estas.

A *framework* consegue assim funcionar de forma segura e controlada utilizando apenas recursos disponibilizados manualmente, pode funcionar de forma totalmente automática e evitar carregamento de classes desnecessárias, e pode funcionar de forma automática mas forçando carregamentos no arranque para evitar possíveis impactos de performance que possam existir na transferência de classes em tempo de execução. É também possível utilizar qualquer combinação destas metodologias.

8.1 DPFClassLoader

A distribuição de código automática é feita através do *DPF Classloader*, a utilização de um *classloader* como ferramenta de propagação de código tem as seguintes vantagens:

- O processo torna-se totalmente invisível para o utilizador final e evita assim a complexidade de ter de transferir as classes e *JARs* necessários (evitando também possíveis esquecimentos);
- Só são transferidas as classes estritamente necessárias;
- A implementação de um *classloader* próprio permite o carregamento dinâmico de classes em tempo de execução.

A *DPF4J* utiliza vários *DPFClassloaders* (Figura 32), um por cada nó conhecido e responsável por carregar as classes utilizadas pelas tarefas submetidas por esse nó.

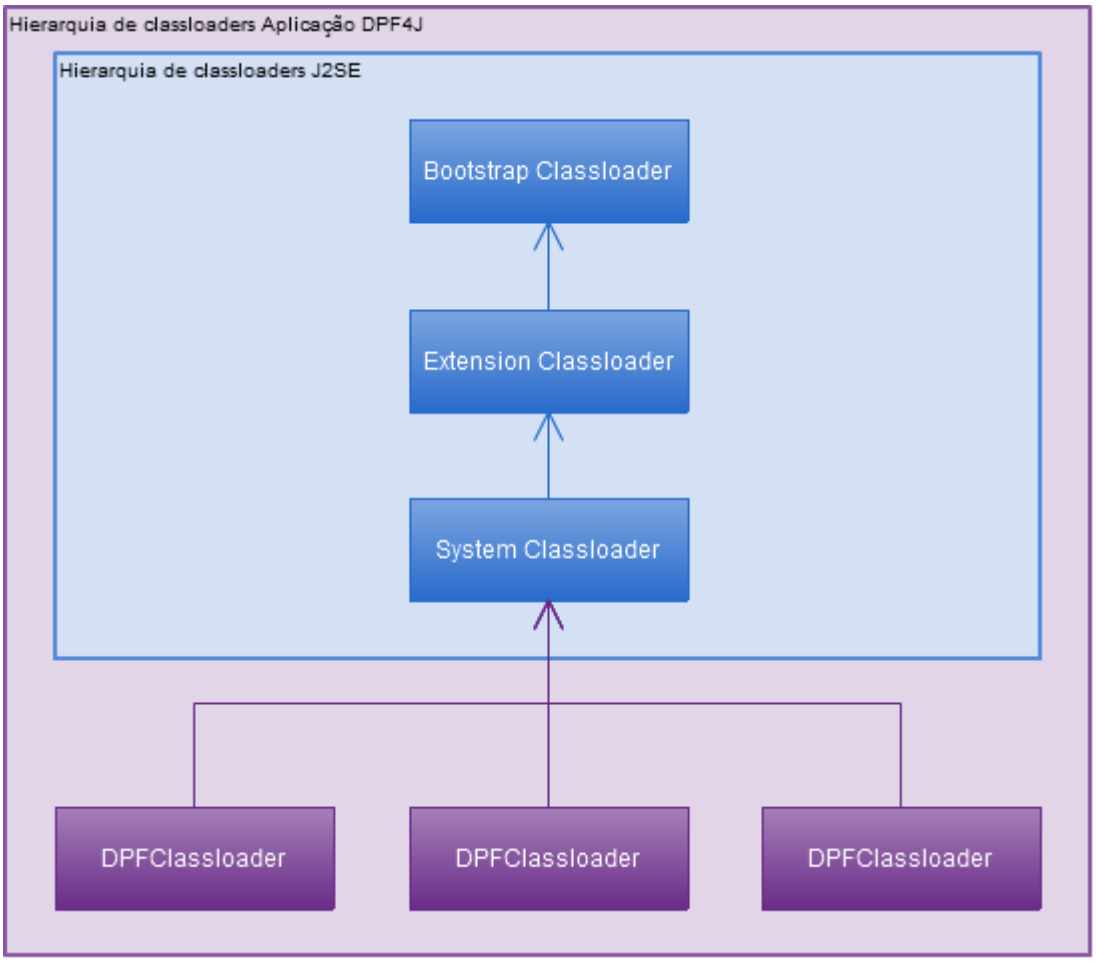

Figura 32 - Hierarquia de Classloaders DPF4j

O comportamento de *classloading standard* do *Java SE* é uma filosofia "pai primeiro" e é impossível a um *classloader* conhecer as classes dos seus filhos ou irmãos. No seguinte fluxograma é apresentado o comportamento normal de um *classloader* no momento em que uma classe lhe é requisitada (método *loadclass()*, Figura 33).

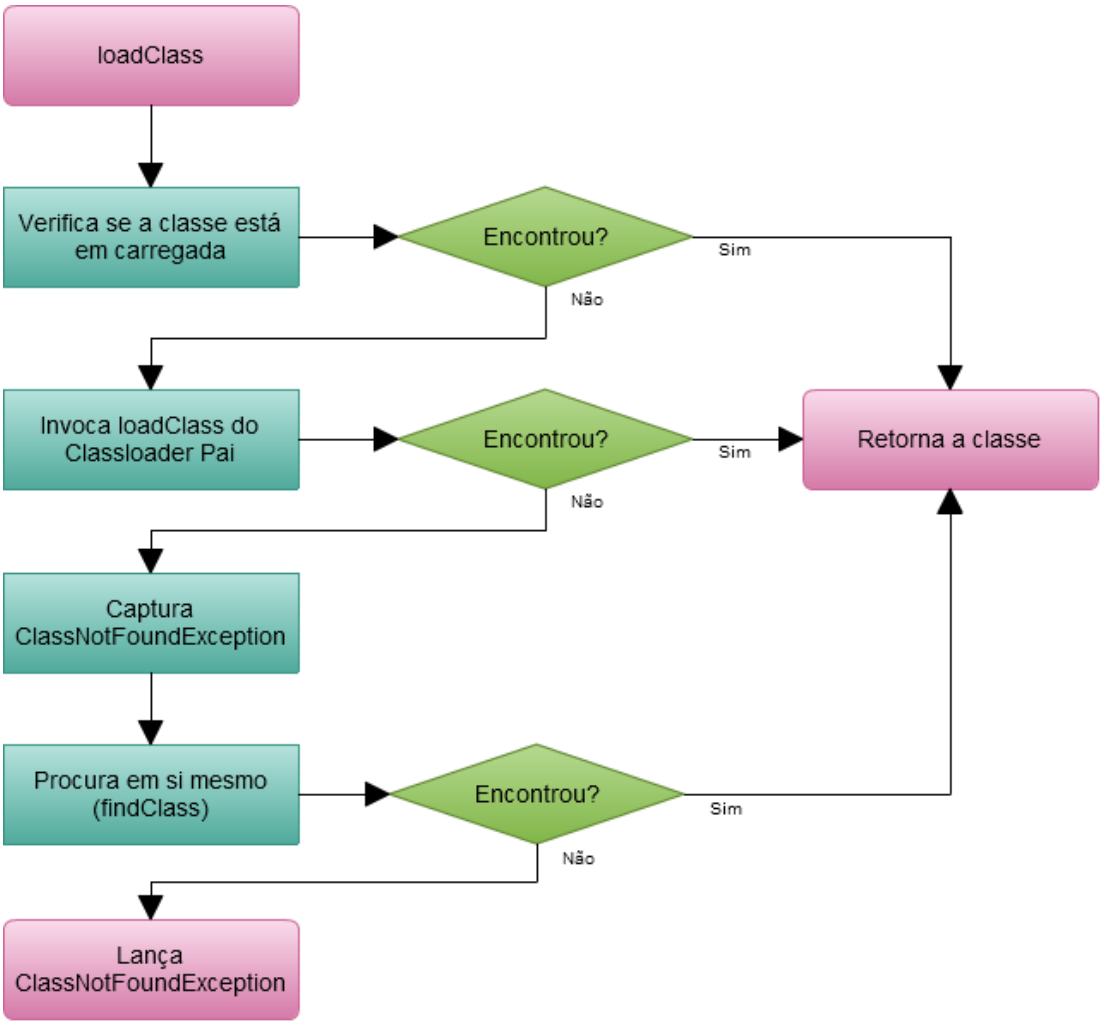

Figura 33 - Fluxo *loadclass()*

A existência de múltiplos *DPFClassloaders*, em vez de apenas um, permite que os vários nós tenham classes com o mesmo nome ou classes em diferentes versões sem criar conflitos entre si. Esta arquitetura tem no entanto as desvantagens de ser mais pesada do que a arquitetura com um *classloader* apenas, não só do ponto de vista de memória mas também porque exclui a utilização de recursos compatíveis já obtidos a partir da execução de tarefas provenientes de outros nós.

A implementação do *DPFClassloader* altera o comportamento normal de um *classloader* no momento em que o *classloader* procura pela classe em si mesmo (*findClass(),* Figura 34), verificando se esta já existe em cache devido a execuções anteriores ou então requisitando esta ao nó que submeteu a tarefa.

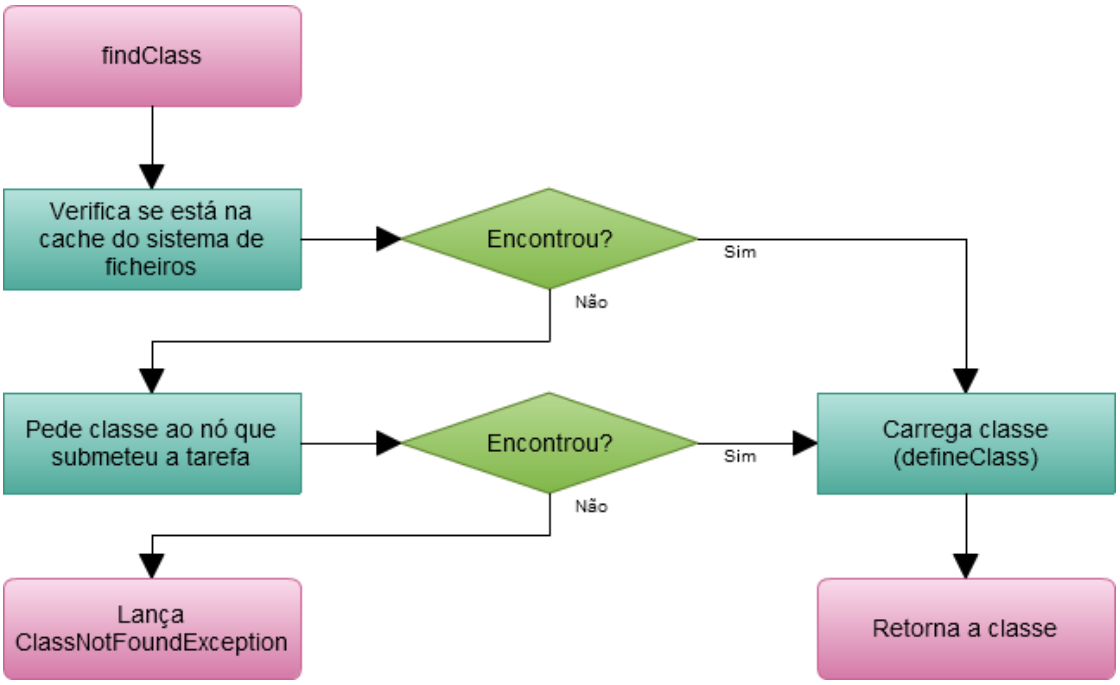

Figura 34 - Fluxo findClass()

Para melhorar a performance entre execuções, existe uma cache no sistema de ficheiros onde são arquivadas todas as classes utilizadas em determinada execução, isto permite que a máquina virtual possa ser reiniciada, sem que haja necessidade de requisitar todas as classes de novo na execução seguinte. As classes são guardadas num diretório de opção do utilizador seguido por um subdiretório com o nome do nó de origem da classe (opcional, ver Configuração), seguido da lista de diretórios correspondentes ao package da classe.

O pedido ao nó remoto é feito através da invocação da camada de serviços desse nó, o serviço em questão é chamado de *retrieveClassByName* e é responsável por descobrir onde se encontra a classe em questão através do *classloader* onde a DPF4j está presente (por norma será o *System Classloader* e evita assim qualquer classe que possa ter vindo de outro nó) e obtém esta no seu formato de *bytecode* de forma a transferi-la para o nó cliente. É obrigatória a passagem da classe em *bytecode* e não já definida (formato objeto) porque neste caso conteria referências aos *classloaders* locais e ficaria num estado inconsistente ao ser utilizada na outra máquina.

Está planeado que a opção de usar uma arquitetura (múltiplos *DPFClassloaders*) ou outra (um único *DPFClassloader*) seja passada para o utilizador da *framework* e assim configurável através da *DPFConfiguration*. No entanto por omissão será sempre a utilização de múltiplos *classloaders* devido a ser uma solução mais fiável. Neste momento se se desejar reaproveitar recursos é possível apaziguar o redescarregar de recursos através da propriedade "dpf.classloader.cache.global" que configura a existência de apenas uma cache global em vez de uma cache por nó.

8.1.1 Configuração

De seguida apresentam-se as propriedades de configuração do *DPFClassloader*, com respetiva descrição e valor por omissão:

Propriedade	Descrição
`dpf.workgroup.<nomedoworkgroup>.classpath`	Define uma lista de URLs de classes e JARs separados por ponto e virgula que serão enviados para as máquinas remotas após o momento de associação
	Default: n.d.
`dpf.classloader.cache.path`	Diretório no sistema de ficheiros onde a cache deve ser armazenada
	Default:.dpf_cache/
`dpf.classloader.cache.global`	Define se deve ser utilizada uma cache global ou uma cache por nó.
	Default: *false*
`dpf.classloader.cache.cleanuponboot`	Limpa todos os ficheiros do tipo .class encontrados no diretório de cache e subdiretórios.
	Default: *false*
`dpf.classloader.noremoteclassoading`	Desativa o *classloading* remoto
	Default: *false*

Tabela 8 - Configurações do *DPFClassloader*

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 por 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 9.

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 9 - 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 efetuadas 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 á maquina que efetuou os testes. No final são apresentados os valores obtidos para a *framework DPF4j* no modo distribuído.

Nas figuras (Figura 35, Figura 36 e Figura 37) encontram-se os gráficos que demonstram os valores obtidos nos testes executados nas várias máquinas utilizadas para os testes.

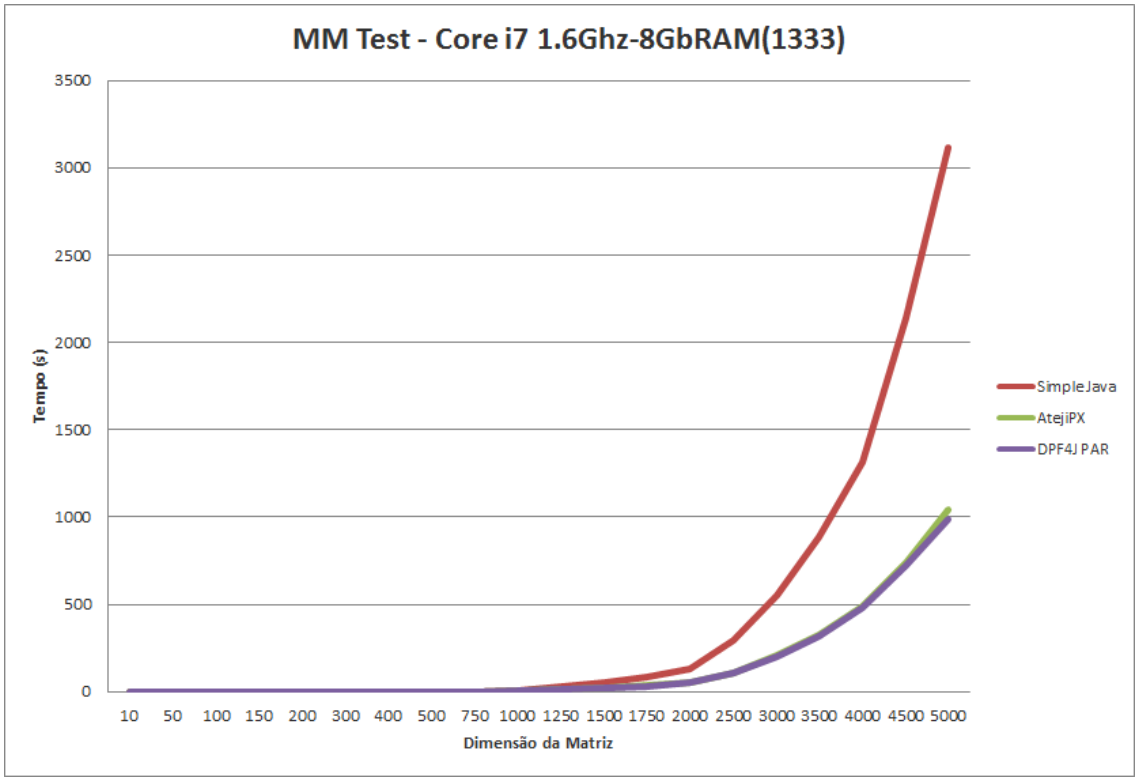

Figura 35 - Caso de teste 1 - Gráficos de resultados referentes à máquina i7-1

Os dados referentes ao gráfico apresentado na Figura 35 encontram-se na Tabela 10

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

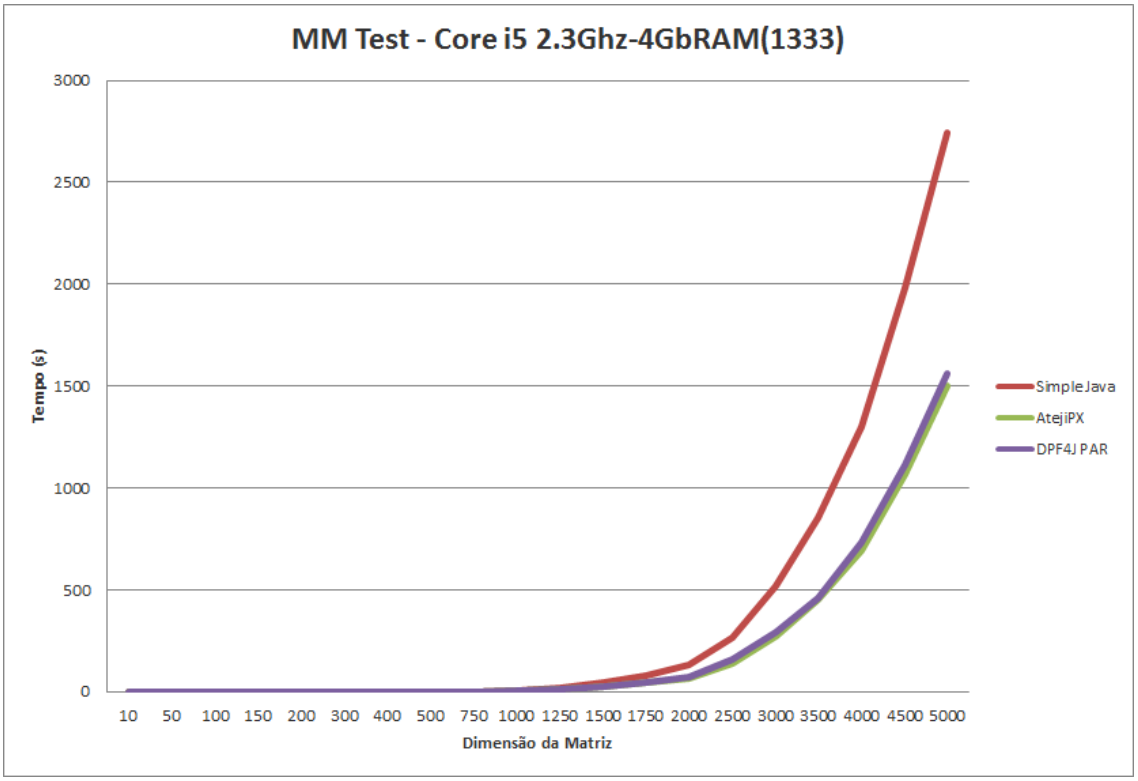

Figura 36 - Caso de teste 1 - Gráfico de resultados referente à máquina i5

O gráfico da Figura 37 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 11 são apresentadas as médias dos valores obtidos para a máquina i5. Através dos valores da tabela é possível ter uma melhor perceção 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 11 - Tabela de resultados obtidos pela máquina i5 para o caso de teste 1

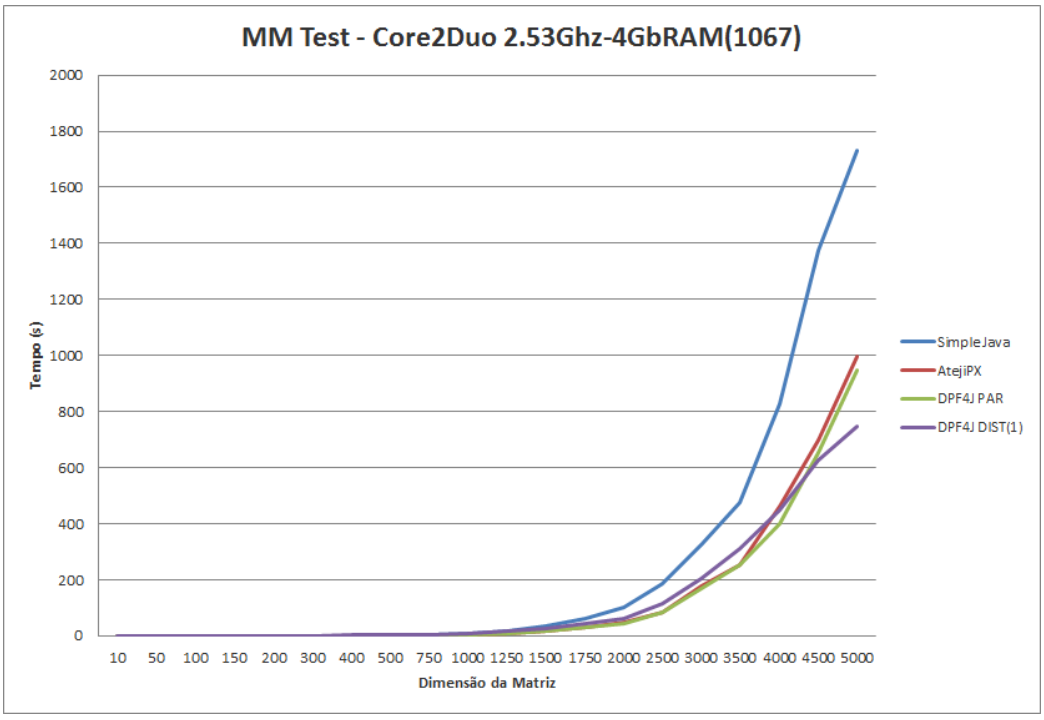

Figura 37 - Caso de teste 1 - Gráfico de resultados referente à máquina C2D-1

Na Tabela 12 são apresentados os valores referentes ao gráfico apresentado na Figura 37

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 12 - 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 efetuado 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 13)

Finalmente a resposta de cada tarefa inclui o resultado da multiplicação e esta é representada por um objeto de tamanho variável (ver Tabela 13 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 13 - Resultados relativos à quantidade de dados transferidos

9.1.5 Conclusões

Através dos resultados demonstrados pode-se concluir que 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 Objetivo 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 14 - 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 15.

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 15 - 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 38.

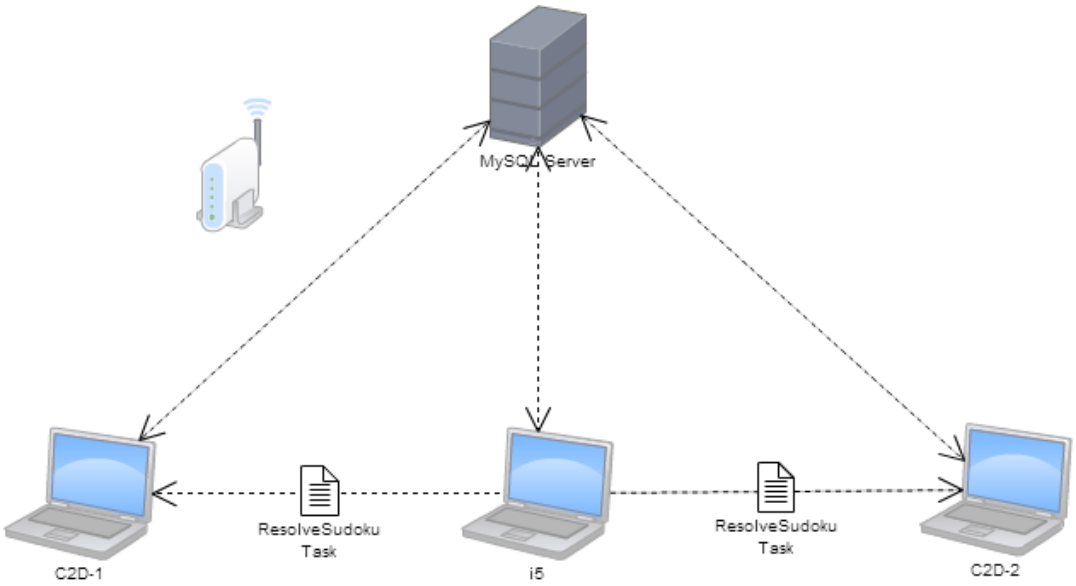

Figura 38 - Caso de teste 2 - Infraestrutura

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 16 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 16 - Caso de teste 2 - Resultados

*Máquina que delega trabalho

Na Figura 39 é apresentado um gráfico com os resultados obtidos para que se possa ter uma melhor perspetiva e fazer a comparação dos valores obtidos.

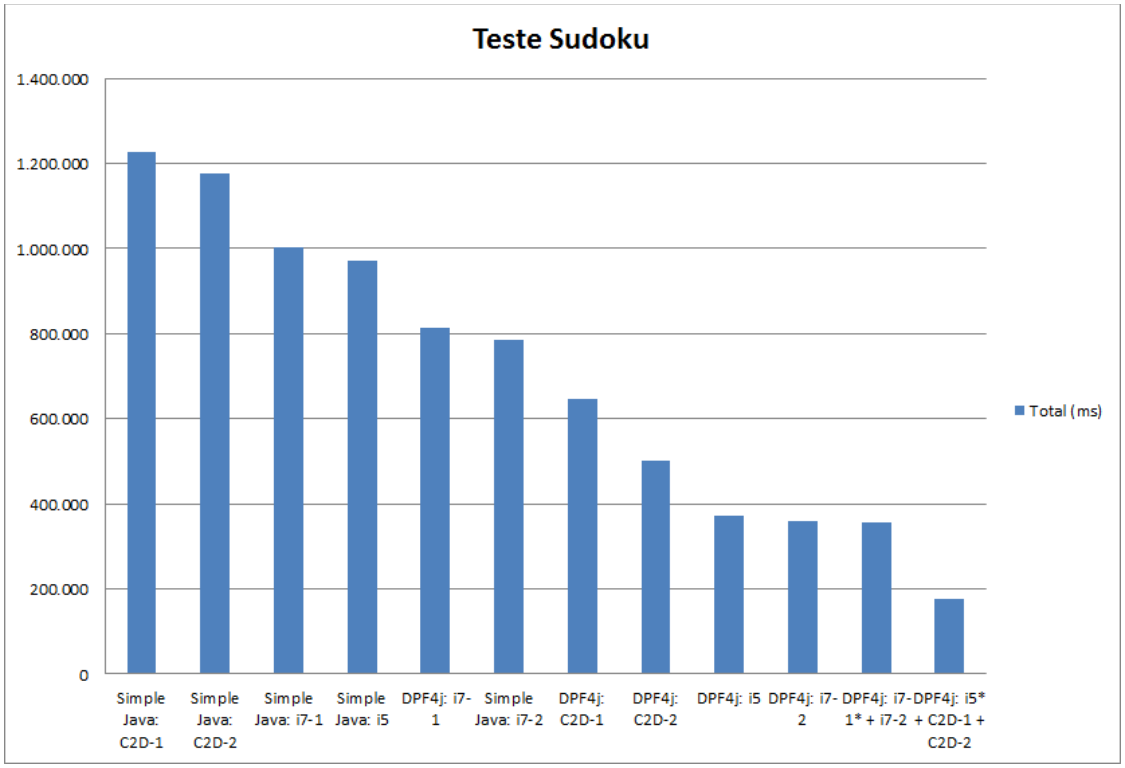

Figura 39 - Caso de teste 2 - Gráfico de resultados

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 17.

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

Tabela 17 - 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

Esta dissertação constituiu uma grande oportunidade para aprofundar conhecimentos sobre programação paralela e distribuída. Foi possível tomar conhecimento de vários trabalhos dentro duma área que se apresenta inovadora e promissora.

O desenvolvimento da *framework* apresentou inúmeros desafios no desenvolvimento dos seus vários componentes. Na *API* existiam as limitações impostas pela linguagem Java devido a fornecer uma interface o mais semelhante possível à utilizada na programação sequencial, mas lidando com problemas da distribuição como: i) a inexistência de memória partilhada, aos desafios trazidos pela dupla capacidade de paralelizar e distribuir e as formas como se poderia fazer isto de forma eficiente *(DPF Scheduler*); ii) à transferência e propagação de código de forma totalmente automática para aproximar o uso e instalação das aplicações *DPF4j* ao uso e instalação de vulgares aplicações sequenciais (*DPF ClassLoader*); iii) até à capacidade da *framework* se adaptar às necessidades dos seus utilizadores e se manter simples (*DPF Configuration*).

A nível da distribuição e execução de trabalho, foi possível a criação de um escalonador e mecanismos de transferência de trabalho eficientes e com características modulares que permitirão que futuros desenvolvimentos a possam melhorar e adaptar aos cenários de utilização.

Para a distribuição de código foi desenvolvida uma solução dinâmica com base num *classloader* que permitirá facilitar significativamente a instalação, manutenção e uso das aplicações que usem a *framework*.

A *DPF4j* conseguiu dar provas da validade de todo o estudo efetuado, atingindo os objetivos definidos e demonstrando que nos dias de hoje, com equipamentos de gama doméstica, já é possível tirar proveito do processamento paralelo e distribuído.

10.1 Trabalho Futuro

Apesar de avançada, a *framework* encontra-se ainda num estado protótipo e necessita ainda de trabalho para poder ser dispobilizada para a comunidade

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*, o *DPF Daemon* muitas vezes irá correr em servidores e a sua gestão e administração deverá ser feita 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 que a *DPF4j* tem no que toca à programação distribuída e programação paralela, é a inexistência de memória partilhada na primeira. No futuro deverá ser implementado alguma forma de memória partilhada, baseada em 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, irá ser estudado e implementado um mecanismo 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.

No que toca a *scheduling* terá que se melhorar o controlo de execução e distribuição de tarefas, por exemplo, cancelar tarefas de nós remotos quando o nó local está sem trabalho e pode realizá-las ele próprio. Os nós também deverão conseguir recolher estatísticas dos nós conhecidos, nomeadamente tempos de resposta e capacidade de processamento de forma a conseguir otimizar o processo de escalonamento, no momento de decidir para que nó enviar determinada tarefa. Também deverão ser implementados mais escalonadores para dar mais opções ao programador, por exemplo, a existência de escalonadores *soft-realtime*.

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] United States National Institute of Standards and Technology (NIST), Announcing the ADVANCED ENCRYPTION STANDARD (AES), 2001.

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

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

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

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

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

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

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

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

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

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