
**João André Figueiredo Gonçalves Saramago**

Licenciatura em Engenharia Informática

# Um Middleware para Computação Paralela em Clusters de Multicores

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : Prof. Doutor Hervé Miguel Cordeiro Paulino, Prof. Auxiliar, Universidade Nova de Lisboa

Júri:

| | |
|---|---|
| Presidente: | Prof. Doutora Maria Armanda S. Rodrigues Grueau |
| Arguente: | Prof. Doutor João Coelho Garcia |
| Vogal: | Prof. Doutor Hervé Miguel Cordeiro Paulino |


FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

**Junho, 2012**

**Um Middleware para Computação Paralela em Clusters de Multicores**

*Para os meus pais.*

# Agradecimentos

Quero expressar a minha sincera gratidão a todas as pessoas que acompanharam a realização desta dissertação. Agradeço especialmente ao meu orientador, o professor Doutor Hervé Paulino, por todo o apoio, orientação e disponibilidade desde o primeiro dia. A sua ajuda foi essencial para atingir os objetivos desta dissertação, tendo uma grande paciência para todas as revisões de escrita deste documento.

A todos os meus colegas e amigos de curso que acompanharam nesta jornada, nomeadamente Diogo Mourão, Vasco Pessanha, Ricardo Alves, João Vaz, Eduardo Marques, Tiago Vale e Valter Balegas.

Aos meus pais e irmã por todo o apoio e carinho que sempre me deram.

# Resumo

A boa relação custo/performance dos aglomerados (*clusters*) de processadores *multi-core* popularizou este tipo de plataforma no âmbito da *High Performance Computing* (HPC) - Computação de Alta Performance.

No entanto, a programação de *clusters* é complexa, requerendo a consciência da sua arquitetura, o que prejudica o desenvolvimento, portabilidade e manutenção das aplicações. No caso particular dos *clusters* de *multi-cores*, esta complexidade aumenta quando os nós que compõem o *cluster* não são uniformes, quer no número de *cores*, quer na hierarquia de memórias *cache*. Este tipo de arquitetura já foi apelidada de *Non-Uniform Cluster Computing* (NUCC) [CGS+05], pois conjuga os conceitos de *clustering* e acesso não uniforme à memória.

Neste contexto, é essencial o desenvolvimento de ferramentas que proporcionem níveis de abstração mais elevados, escondendo os detalhes subjacentes à arquitetura e às tecnologias necessárias à comunicação, escalonamento de tarefas, consistência de memória, entre outros. O trabalho desenvolvido pretende contribuir para o avanço do estado da arte nessa área. Propõe-se um *middleware* para programação paralela especialmente vocacionado para *clusters* de *multi-cores*, cujo desenho é inspirado no de Sistemas de Operação, no sentido de que, além de definir uma interface para as aplicações, define outra para o desenvolvimento de novos módulos (*drivers*) que permitem a especialização das suas funcionalidades para uma dada arquitetura alvo. O trabalho partiu de uma base já existente para memória partilhada, tendo sido estendido para arquiteturas de memória distribuída. Para tal, foram especializadas as camadas de interface, de suporte base e drivers para suportar estas arquiteturas, nomeadamente *clusters* de *multi-cores*.

**Palavras-chave:** computação paralela, clusters de multi-cores, middleware

# Abstract

The good ratio between cost and performance in multi-core clusters has made it a popular architecture in the HPC (High Performance Computing) environment.

However, cluster programming is complex, requiring an awareness of its architecture, which affects application development, portability and maintenance. In the specific case of multi-core clusters, this complexity increases when cluster nodes are not uniform, be it in number of cores or memory cache hierarchy. This type of architecture has been called NUCC (Non-Uniform Cluster Computing) [CGS+05], as it combines the concepts of clustering and non-uniform memory access.

With this in mind, the development of tools that provide higher levels of abstraction is essential, while also hiding the underlying details of the architecture, communication, job scheduling, memory consistency, among others. This work aims at contributing to the advancement of the state of the art in this area. We propose a middleware for parallel programming specially devoted to multi-core clusters, with a design inspired by the Operating System, in the sense that it not only defines an interface for applications, but also defines another for the development of new modules (drivers) which allows the specialization of features for a given target architecture. This work started from an existing base for shared memory, which has been extended to distributed memory architectures. To this end, there was a specialization of the interface, basic support and driver layers in order to support this architecture, namely multi-core clusters.

**Keywords:** parallel computing, multi-core clusters, middleware

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

# Listagens

# 1

# Introdução

## 1.1 Motivação

A *High-performance computing* (HPC), ou computação de alta performance, consiste na utilização de supercomputadores ou aglomerados (*clusters*) de computadores na resolução de problemas computacionais complexos, com recurso a um processamento paralelo, eficientemente, com tolerância a falhas e de alto desempenho. De entre os muitos problemas tratados com recurso a HPC encontram-se a dinâmica de fluídos [Can02], a previsão do clima [WOS08] e o sequenciamento genético [CSG+03]. As arquiteturas utilizadas no âmbito deste tipo de computação atingem, atualmente, performances na ordem dos teraflops [HLW00], ou seja, $10^{12}$ operações em vírgula flutuante por segundo. No global, a HPC integra a administração destes sistemas e a sua programação no contexto da computação paralela.

A aglomeração é a organização *standard* desde há muito tempo. Naturalmente com o aparecimento dos multi-cores, a aglomeração deste tipo de arquiteturas passou a ser o *standard*, em deterioramento dos supercomputadores, devido ao seu rácio custo/performance ser baixo. Atualmente, 82.20% do top dos melhores 500 supercomputadores do mundo são *clusters* e mais de 90% são *multi-cores*[1]. Uma investigação realizada no âmbito da IBM identifica os *clusters* de nós *Symmetric Multiprocessing* (SMP) compostos de processadores *multi-core* como sendo o paradigma emergente para a organização de estações de computação para HPC [CGS+05].

Este tipo de organização de recursos computacionais pode ser heterogéneo, combinando as características dos *clusters* com as características das arquiteturas com acessos

[1] Fonte: `http://top500.org` (Julho de 2011)

não uniformes a memória, que resultam da conjunção de hierarquias de memória não uniformes. Este facto é ainda mais acentuado na aglomeração de nós *Non-Uniform Memory Access* (NUMA), dada a não uniformidade do acesso à memória estender-se ao interior do nó. No entanto, antevê-se que os nós NUMA ganhem popularidade no contexto destas arquiteturas, pois o aumento do número de *cores* nos processadores expõe claramente as limitações de escalabilidade das arquiteturas SMP. O Intel® QuickPath [Cor08] é um exemplo de um tecnologia para a interligação de processadores numa arquitetura NUMA.

Além da heterogeneidade no acesso à memória, existe também a heterogeneidade ao nível da capacidade do processamento. Antes da introdução dos *multi-cores*, esta devia-se essencialmente à diferença da velocidade do seu relógio dos processadores. Atualmente, nas arquiteturas *multi-cores*, a heterogeneidade devem-se principalmente às diferenças no número de cores e na capacidade e níveis da hierarquia de memória *cache*. Estas características têm de ser consideradas na conceção dos algoritmos de escalonamento de tarefas.

A arquitetura dos *clusters* de *multi-cores* é bastante popular devido as altas performances destas sem *hardware* especializado. Como se trata de num aglomerado de máquinas, esta arquitetura é bastante escalável e com baixo custo de aquisição, manutenção e de expansão, ao contrario dos supercomputadores tradicionais. Este tipo de infra-estrutura é utilizada tanto ao nível académico, como na industria (por exemplo, industria automóvel, farmacêutica, aeroespacial e finanças). Os principais utilizadores deste tipo de infra-estruturas são programadores que, além de terem conhecimento da área do problema a resolver, têm de ser especialistas em programação paralela e distribuída devido à complexidade de desenvolvimento destas arquiteturas.

A linguagem Java tem assumido um papel particularmente importante nestas arquiteturas. A sua popularidade tem aumentado consistentemente, à medida que a HPC vem sido lentamente adotada em computação uso geral [TTD09]. A diferença de desempenho entre o Java e o código nativo foi consideravelmente reduzida, permitindo que as vantagens da linguagem se destaquem, como por exemplo, a interoperabilidade entre plataformas, um modelo de programação amplamente aceite com construções simples para multi-threading e uma API abrangente para comunicação remota.

Esta tendência fomentou a proposta de vários sistemas para HPC baseados em Java, que vão desde a implementação de bibliotecas de troca de mensagens, como por exemplo o *standard* MPI com o MPIJ [CGJ+00]; sistemas de *middleware*, salientando Proactive [HCB04], Parallel Java [Kam07] e MapReduce [DG08]; extensões de linguagem, como o Titanium [YSP+98], a novas linguagens concorrentes inspiradas em Java, sendo uma delas o X10 [CGS+05].

Com a exceção do Proactive, todos os modelos de programação mencionados possuem um problema fundamental: não existe espaço para o compilador gerar código especifico aos processadores ou para o sistema de execução se adaptar às características do *hardware*.

## 1.2 Problema

O desenvolvimento de software para arquiteturas de memória distribuída é complexo pois requer o particionamento dos dados da aplicação pelas memórias locais do vários nós que compõem o ambiente de execução. Tal obriga o programador a ter consciência da arquitetura (infraestrutura *hardware*). As ferramentas disponíveis para a programação deste tipo de arquitetura são orientadas para especialistas, expondo toda a complexidade subjacente. Apesar de terem sidos propostos alguns *middlewares*, as ferramentas mais tradicionais, nomeadamente o *Message Passing Interface* (MPI) [SOW+95], continuam a ser o *standard* de-facto para a comunicação entre nós em ambientes distribuídos.

Nos *clusters* de *multi-cores*, além da comunicação e gestão de paralelismo entre nós, são necessárias ferramentas para explorar o paralelismo em memória partilhada, como por exemplo o OpenMP [CJP07] ou *threads*. Nestas arquiteturas, dada a heterogeneidade do *hardware* existente, nomeadamente no que se refere à hierarquia de memória, número de cores e número de máquinas, existe uma tendência para que as aplicações se adaptem ao hardware devido a questões de performance, o que torna as aplicações não portáveis e difíceis de manter. De entre as questões que contribuem para a complexidade da programação deste tipo de arquiteturas, salientam-se:

**Escalonamento e balanceamento da carga** A distribuição de trabalho, tendo em vista o balanceamento de carga, é um problema crítico neste tipo de arquiteturas, principalmente quando os nós são heterogéneos no número de *cores*. Sem esta não é possível explorar a totalidade dos processadores. Uma outra vantagem dos escalonadores é a distribuição automática de trabalho.

**Comunicação** A comunicação entre fluxos de execução num ambiente distribuído requer trocas de mensagens explícitas, o que introduz um grau extra de complexidade relativamente ao modelo de comunicação via memória partilhada. No caso especial dos *clusters* de *multi-cores* é necessário ter a consciência da não-uniformidade no acesso a memória, para que a comunicação seja ótima.

**Consistência de memória** O suporte a um modelo de memória partilhada neste tipo de arquiteturas requer protocolos que garantam que as escritas em posições de memória virtualmente partilhadas sejam repercutidas em todos os fluxos de execução. Para tal existem vários protocolos que garantem esta consistência da memória, de forma mais ou menos estrita. A utilização de modelos estritos em ambientes distribuídos é impraticável, devido a limitações físicas, ou seja, é necessário que existam modelos mais relaxados.

**Heterogeneidade** Dada a heterogeneidade do *hardware* existente, nomeadamente ao que se refere à hierarquia de memória, número de *cores* e número de máquinas, existe uma tendência para que as aplicações se adaptem ao *hardware* devido a questões de performance, o que torna as aplicações não portáveis e difíceis de manter.

No caso dos *clusters* de *multi-cores*, e especialmente nos *clusters* de máquinas heterogéneas, este problema ainda é mais evidente, pois é necessário fazer uma gestão hierárquica dos recursos, principalmente da memória e dos processadores, com os seguintes níveis: dentro do processador, dentro dos nós e entre os nós.

## 1.3 Proposta de solução

Devido à complexidade inerente à programação de arquiteturas de memória distribuída, é essencial o desenvolvimento de ferramentas que forneçam uma abstração da arquitetura alvo, escondendo detalhes desta, como por exemplo, comunicação, escalonamento de tarefas e consistência de memória. A nossa abordagem ao problema da heterogeneidade passa por delegar a adaptação ao hardware alvo no sistema de execução, em vez de nas aplicações. Com este objetivo, pretendemos contribuir para o avanço no estado da arte nesta área. A nossa estratégia passa por estender o *middleware* desenvolvido por Mourão [Mou11] para arquiteturas de memória partilhada, de forma a que possa suportar arquiteturas de *clusters* de *multi-cores*.

### 1.3.1 *Middleware*

O estado de arte na área dos sistemas de execução e *middlewares* genéricos para programação paralela para arquitetura de *clusters* focam-se essencialmente na problemática de submissão remota de trabalhos, escalonamento e balanceamento de carga.

Nesta medida, o *middleware* desenvolvido por Mourão apresenta-se como uma máquina virtual onde o *hardware*, é abstraído do programador por uma interface que apresenta uma visão uniforme e independente da arquitetura. O desenho deste é inspirado nos sistemas operativos, na medida em que, para além de especificar a interface para o desenvolvimento de aplicações, também especifica uma interface, baseada na noção de *driver*, para suportar as funcionalidades oferecidas. Este desenho permite delegar em tecnologias já existentes a implementação concreta de uma dada funcionalidade, remetendo para o núcleo do *middleware* a lógica independente das tecnologias. A arquitetura do *middleware* divide-se em três camadas, como se pode verificar na figura 1.1.

Com o objetivo de oferecer uma interface genérica para o programador, esta é centrada na abstração de localidade (ou *place*) e de serviço. Uma localidade representa um processador virtual, no qual se pode adjudicar trabalho sob a forma de tarefas. Esta, pode abstrair um processador, um nó com vários processadores ou vários nós, o que permite ao programador gerir a afinidade entre as computações e os recursos disponíveis. Cada localidade possui uma *pool* de *threads* prontas a executar (trabalhadores) e um espaço de endereçamento que é partilhado pelas suas *threads*. Sobre as localidades serão executados serviços implementados pelos utilizadores. Um serviço fundamenta-se no modelo de objeto ativo [LS96] para disponibilizar um componente modelar para a construção de aplicações paralelas escaláveis. A comunicação entre serviços é feita através da invocação

de métodos.


Figura 1.1: Arquitetura do *middleware*

### 1.3.2 Trabalho a Desenvolver

Esta dissertação tem como objetivo principal estender o *middleware* existente para arquiteturas de memória distribuída, nomeadamente *clusters* de *multi-cores*. A figura 1.2 apresenta a arquitetura da extensão ao *middleware*. O processo de extensão incide sobre as três camadas do sistema, tendo em conta os seguintes objetivos:

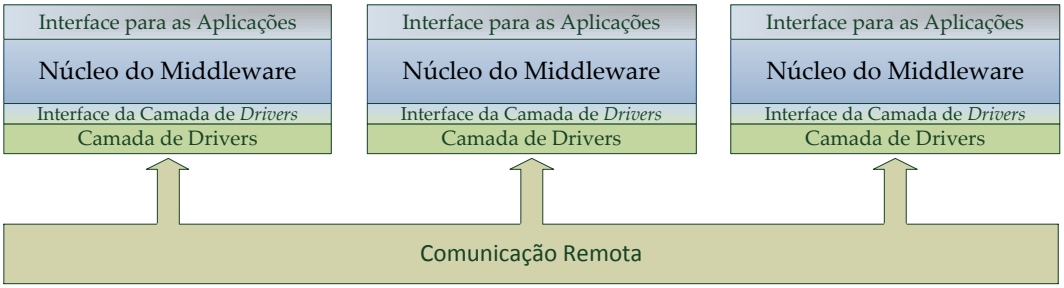



Figura 1.2: Arquitetura do *middleware* num *cluster*

a). Propor construções para a composição hierárquia de serviços para que estes explorem a natureza das arquiteturas de *cluster* de *multi-cores*. Estas construções deverão ser eficientes e expostas ao utilizador de uma forma simples.

b). Implementar um prototipo, com base no *middleware* desenvolvido por Mourão, para este tipo de arquiteturas;

c). Aferir a flexibilidade e performance do modelo de paralelismo de dados ao nível do método (DMR) [Mar12] (mais detalhes na secção 3.2.1) em arquiteturas de memória partilhada e distribuída.

O trabalho de implementação terá de ser precedido pelo levantamento dos requisitos e pela identificação das alterações nas camadas de *drivers* e do núcleo relativamente à versão implementada por Mourão.

## 1.4 Contribuições

As contribuições previstas por esta dissertação são as seguintes:

- Disponibilizar uma plataforma adaptável para computação paralela em Java para arquiteturas de memória distribuída;
- Desenho e implementação de um mecanismo de agregação de serviços dirigido à computação neste tipo de ambientes distribuídos;
- Desenvolvimento de uma prototipo funcional, que concretiza a camada de adaptabilidade do *middleware* para um dado conjunto de tecnologias;
- Uma primeira avaliação do modelo DMR em arquiteturas de memória partilhada e distribuída no âmbito do middleware proposto.

## 1.5 Estrutura do documento

Esta dissertação está organizadas nos seguintes capítulos:

**Capitulo 1 - Introdução:** Corresponde ao capítulo atual, onde é introduzida a motivação para a realização desta dissertação e uma proposta de solução para o problema identificado.

**Capitulo 2 - Estado de Arte:** Neste capítulo são abordados temas da área de investigação onde esta dissertação se encontra inserida. Descrevem-se as dimensões que caracterizam os modelos de programação paralela/distribuída e as suas implementações. O foco deste estudo encontra-se no suporte dos modelos para arquiteturas de memória distribuída, mais especificamente, arquiteturas de *clusters* de multi-cores.

**Capitulo 3 - O *middleware*:** Descreve-se o desenho da arquitetura do *middleware*. Apresentando a interface para as aplicações, descrevendo os módulos e os *drivers*.

**Capitulo 4 - Implementação:** Faz-se uma descrição dos detalhes da implementação atual do *middleware*, descrevendo a comunicação a agregação de serviços.

**Capitulo 5 - Avaliação Inicial:** Realiza-se uma avaliação de desempenho que permite avaliar a performance da nossa implementação do modelo DMR.

**Capitulo 6 - Conclusões e Trabalho futuro:** Descrevem-se as conclusões final desta dissertação, sendo também apresentadas algumas sugestões para trabalho futuro.

# 2

# Estado de Arte

Este capítulo faz um levantamento da área onde se insere o trabalho realizado no âmbito desta dissertação - a elaboração de um *middleware* para computação paralela em *clusters* de multi-core. São abordados temas relevantes para esta dissertação, como por exemplo, conceitos básicos de programação concorrente, arquiteturas paralelas, modelos de programação e *middleware* existentes.

## 2.1 Programação concorrente e paralela

### 2.1.1 Programação concorrente

A concorrência é uma propriedade dos sistemas onde vários fios de execução são executados em simultâneo. A palavra simultâneo não quer dizer ao mesmo tempo. No caso das arquiteturas uni-processadores não é possível ter mais que um programa a ser executado. Para que a concorrência possa ser aplicada a este tipo de arquiteturas é necessário usar um mecanismo de *time sharing*. O *time sharing* consiste na execução de vários fios de execução, atribuindo tranches de tempo de utilização do processador pré-definidas, dando a impressão que os vários fios de execução estão a executar em simultâneo.

A programação concorrente explora este conceito, onde um programa cria vários fios de execução concorrentes para que o problema possa ser decomposto (ver subsecção 2.3.1) em vários subproblemas para facilitar a programação e, eventualmente, aumentar a performance.

Sendo que vários fluxos de execução podem colaborar no cumprimento de um objetivo comum, é normal que tenham de sincronizar a sua execução e, mesmo, trocar informação. Estes dois tipos de comunicação constituem o principal desafio da programação

concorrente. A sua implementação concreta pode ser realizada sobre memória partilhada ou troca de mensagens. Ambos serão descritos detalhadamente na Subsecção 2.3.2.

### 2.1.2 Programação paralela

A programação paralela é uma especialização do modelo da programação concorrente onde a aplicação é desenvolvida assumindo a execução efetivamente paralela de todos (ou de apenas alguns) os fluxos de execução. O foco principal é a performance, e para tal é normalmente efetuado uma decomposição do problema de tal forma que o mapeamento entre as tarefas e o número de processadores seja o mais eficiente possível. Em teoria [Amd67] a decomposição de um problema de forma a que possa ser executado por $N$ fluxos em paralelo poderá ter um ganho de performance de $N$, ou seja executar $N$ vezes mais rápido. Na prática, existe uma percentagem do código que tem de executar sequencialmente, por exemplo acesso a disco. Com este problema em mente, o ganho de performance usando $N$ fluxos em paralelo será inferior a $N$.

Os desafios para programação paralela são os mesmos que a programação concorrente, ou seja, as comunicações. Mas para além deste problema, também existe o de particionar e distribuir os dados e as tarefas pelos vários fios de execução, e de encontrar todas as partes de um problema (ou zonas de código de um programa) que podem ser efetivamente paralelizáveis.

## 2.2 Taxonomia de Flynn

Em 1972, Flynn [Fly72], propôs uma taxonomia para a classificação de arquiteturas de computadores, tendo duas dimensões de classificação, os dados e as instruções, dando origem à Tabela 2.1.

| | Single Instruction | Multiple Instruction |
|---|---|---|
| Single Data | **SISD** | **MISD** |
| Multiple Data | **SIMD** | **MIMD** |

Tabela 2.1: Taxonomia de Flynn

**SISD** Esta categoria refere-se às arquitetura de computadores onde existe um único fio de execução que opera sobre um único conjunto de dados. É aplicado em processadores sequenciais, como, por exemplo, os *cores* das arquiteturas *multi-core*.

**SIMD** Esta categoria refere-se às arquitetura de computadores onde uma mesma instrução é executada simultaneamente sobre vários conjuntos de dados. A instrução é mapeada em diversas unidades de processamento, obtendo-se paralelismo nos dados.

A instanciação ao nível arquitetural consiste na incorporação no conjunto de instruções do processador, capazes de operar sobre vários elementos simultaneamente, tal

como acontece no CRAY-1 [Rus78]. O CRAY-1 foi uma máquina vetorial criada nos anos 70 pela *Cray Research* com o objetivo de aumentar a performance. Esta máquina possuía algumas características interessantes, como instruções SIMD, registos escalares e vetoriais, palavras de memória de 64 bits.

Os processadores que equipam os computadores pessoais atualmente também contêm instruções com capacidade vetorial. Um exemplo disto é a família de instruções *Streaming SIMD Extensions* (SSE) e MMX dos processadores Intel [Sie10]. As instruções MMX foram introduzidas na arquitetura IA-32 nos processadores da família *Pentium* e nas SSE na família *Pentium III*.

Mais recentemente os *General-Purpose Graphics Processing Units* (GPGPU) também têm sido utilizados para computação vetorial, tendo sido propostas API's (e.g., *Compute Unified Device Architecture* (CUDA) [NVI10]) e de linguagens (e.g., OpenCL [Khr09]) para a programação destes. Estes evoluíram a partir de um hardware especializado (*Graphics Processing Units* (GPU)) no processamento gráfico para hardware capaz de realizar computações de cariz mais geral, daí o termo GPGPU. Na última década os GPU aumentaram a sua performance até níveis que ultrapassaram os *Central processing unit* (CPU) para determinados tipos de aplicações [BB09]. No entanto, os GPU foram desenhados para uma classe de aplicações particular [OHL+08], e.g., processamento de um *stream* de vídeo em tempo real.

**MISD** Esta categoria caracteriza arquiteturas de computadores, onde diversos processadores executam instruções diferentes sobre o mesmo conjunto de dados. Mas, até aos nossos dias, não foram desenvolvidos exemplos deste modelo de arquitetura.

**MIMD** Esta categoria é a mais comum em computadores paralelos. Caracteriza arquiteturas de computadores onde diversos processadores executam instruções diferentes sobre um conjunto de dados diferente.

Este tipo de arquiteturas pode ser de memória partilhada ou de memória distribuída. No caso das arquiteturas de memória partilhada podem ser, por exemplo, SMP ou NUMA, no caso das de memória distribuída há, por exemplo, *clusters* ou *Massive parallel processing* (MPP)

Uma arquitetura SMP (figura 2.1) consiste num conjunto de processadores idênticos que partilham um *bus* comum com outros recursos (e.g., memória e disco). Nesta categoria estão incluídas as tecnologias *multi-core*.

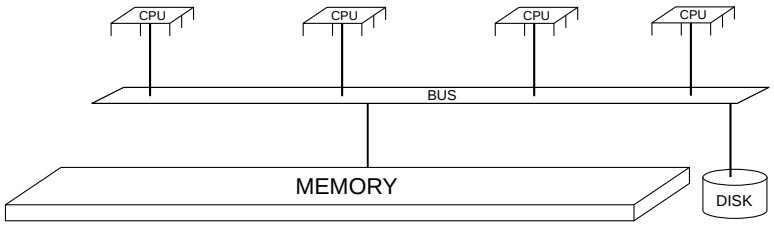



Figura 2.1: Arquitectura SMP [Wik12]

Nas arquiteturas NUMA, cada processador possui uma memória local (ver figura 2.2), onde o acesso é mais rápido, mas pode aceder a qualquer memória do sistema. O fundamento assenta na atenuação da diferença de velocidade entre os processadores e a memória, e na eliminação da contenção no *bus* existente nas arquiteturas SMP. No entanto, o desenho tem como consequência o facto de o tempo de acesso às várias memórias não ser uniforme.

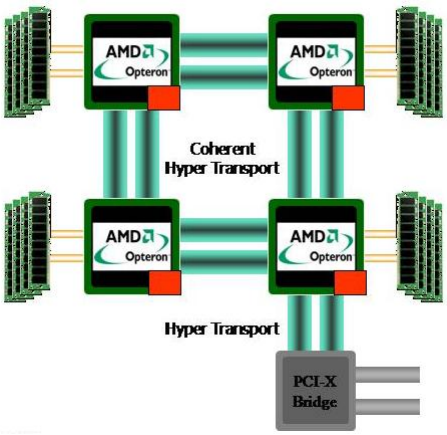



Figura 2.2: Arquitetura NUMA da AMD [Fer12]

A arquitetura MPP consiste num aglomerado de módulos contendo memória, processador e sistema de *Input/Output* (I/O), na mesma máquina. Estas máquinas possuem *hardware* especializado, incluindo tecnologias de rede especializadas para conectar os módulos.

A arquitetura de *cluster* é bastante semelhante à arquitetura de MPP. Esta consiste num aglomerado de máquinas de arquiteturas de memória partilhada, como por exemplo SMP ou NUMA, interligadas por uma rede, de preferência de alta performance. As principais características desta arquitetura são a alta disponibilidade e o possível balanceamento de carga. Esta arquitetura poderá ser homogénea ou heterogénea. No primeiro caso todas as máquinas que compõem o aglomerado são iguais. Basta uma máquina diferente, para que esta arquitetura seja heterogénea.

Nesta dissertação foca-se em arquiteturas de *clusters* de multi-cores, tanto *clusters* de SMP como de *clusters* de NUMA. Num futuro próximo temos como objetivo incluir *clusters* de arquiteturas híbridas, suportando GPU's.

## 2.3 Modelos de Programação Paralela

Esta secção descreve vários modelos de suporte a programação paralela. Um modelo de programação paralela define como os vários fios de execução comunicam entre si e quais as operações de sincronização disponíveis. O levantamento foca inicialmente as várias dimensões dos modelos, nomeadamente decomposição, comunicação, modelo de execução e instanciação.

### 2.3.1 Decomposição

A decomposição, em programação paralela, é o ato de analisar, repartir e mapear nos fios de execução um algoritmo paralelo, com o objetivo de explorar a performance. Esta pode ser realizada em duas vertentes: paralelismo de dados e paralelismo de tarefas. O restante conteúdo desta subsecção é fortemente baseado no artigo *Models for Parallel Computing: Review and Perspectives* [KK07].

**Paralelismo de dados** Computação *Data parallel* consiste na aplicação simultânea de uma computação escalar a vários elementos de um ou mais vetores, resultando num novo vetor. As computações aplicadas a cada elemento deverão ser independentes entre si e passíveis de ser executadas em qualquer ordem, em paralelo ou em *pipeline*. Uma operação *data parallel* corresponde, na programação sequencial, a um ciclo que percorre todos os elementos dos vetores a operar e o vetor resultado. Esta funcionalidade é suportada na maior parte das linguagens *data parallel* na forma de um ciclo paralelo: o ***forall***. Em tempo de execução, o *forall* é responsável por distribuir os dados pelas várias *threads*, ou processos disponíveis, e executar as operações do ciclo em cada um deles. Uma das aplicações deste modelo é o processamento de imagem, onde uma imagem a processar pode ser dividida em várias partes sendo aplicado o algoritmo desejado.

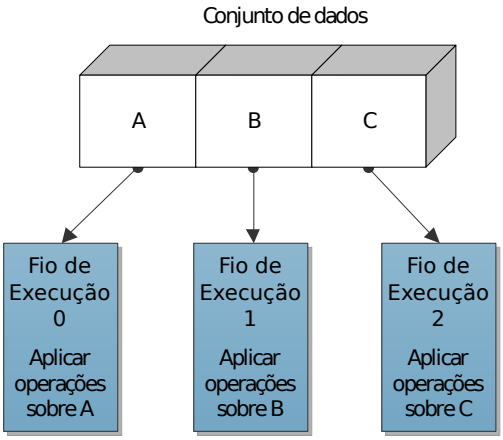



Figura 2.3: Paralelismo de dados [Mou11]

A vantagem do paralelismo de dados sobre decomposição baseado em tarefas (mais pormenores na Subsecção 2.3.1), é a facilidade de análise e de fazer *debug*. Este facto deve-se à existência de um estado único de controlo do programa, o que contrasta com o paralelismo de tarefas, onde os programas podem seguir fluxos de execução diferentes. Por outro lado, o modelo da computação *data parallel* por si só é um pouco restritivo. Apesar de se encaixar na maioria das computações numéricas é, em muitos casos, demasiado inflexível. O algoritmo 2.1 mostra um pequeno exemplo *data parallel*, a soma de

todos os elementos de um vetor.

```
int soma (int[] x){
  int sum = 0;
  forall(k in x){
    sum += k;
  }
  return sum;
}
```

Listagem 2.1: Algoritmo *data parallel* da soma de todos os elementos de um vetor

Um caso particular da computação *data parallel* é a computação vetorial. Neste modelo, as operações *data parallel* são limitadas a operações pré-definidas do tipo vetoriais como, por exemplo, a adição elemento-a-elemento. Qualquer operação do tipo vetorial pode ser reescrita por um ciclo *data parallel*, sendo que o contrário não é verdade, porque as operações de *data parallel computing* são normalmente mais gerais.

O modelo *data parallel* pode ser implementado ao nível das linguagens, e.g., ZPL [CCL+98], Sequoia [FHK+06] e OpenMP [CJP07]. O modelo de computação vetorial pode ser implementado ao nível das linguagens, e.g., Intel Array Building Block [GSC+10] e OpenCL [Khr09].

**Paralelismo de tarefas** Muitas aplicações podem ser consideradas como um conjunto de tarefas, cada uma resolvendo uma parte do problema. Uma tarefa pode gerar outras tarefas dinamicamente, usando o mecanismo de *fork-join* (ver secção 2.4.1). Estas aplicações podem ser representadas por um grafo, onde os nós representam tarefas e os arcos representam comunicação entra as mesmas, i.e., dependência de dados. O escalonamento de um grafo de tarefas envolve a ordenação e o mapeamento de tarefas nos vários processadores.

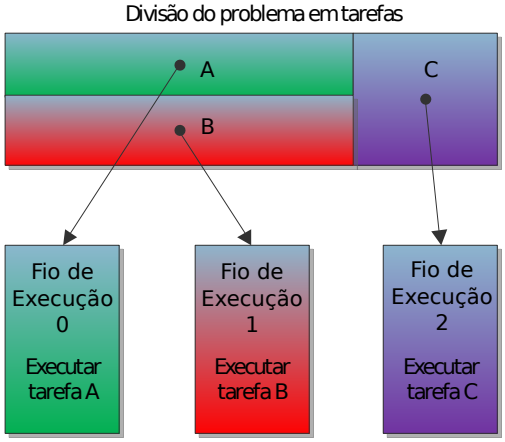



Figura 2.4: Paralelismo de tarefas [Mou11]

Exemplo de linguagens (ou extensões de linguagens) que implementam este modelo são: o pSystem [LS97], Cilk [FLR98], *Cascade High Productivity Language* (CHAPEL) [CCZ04], o X10 [CGS+05] e o OpenMP [CJP07], este último com um suporte rudimentar. A listagem 2.2 mostra um programa que aplica uma estratégia de paralelismo de tarefas.

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

Listagem 2.2: Algoritmo do Fibonacci em *task parallel*

No nosso trabalho estamos particularmente interessados neste mecanismo ao nível da linguagem, mas ele pode ser implementado em termos de arquitetura, e.g., processadores super-escalares. Um processador super-escalar explora o paralelismo ao nível da instrução, executando mais que uma instrução por ciclo de relógio. Para isso, tem de detetar as dependências do fluxo de dados e de controlo de um fluxo linear de instruções, para distribuir as instruções pelas várias unidades funcionais. Para que estas arquiteturas funcionem, precisam de mecanismos para guardar o estado na ordem correta. Estes mecanismos têm de manter a aparência de uma execução sequencial.

O modelo *Dataflow* também faz uso do paradigma do paralelismo baseado em tarefas. Proporciona uma máquina de execução com o grafo de fluxo de dados da aplicação, onde os nós representam blocos de instruções. Finalmente, os grafos de tarefas são também usados em *grid computing* [QZHZ05], onde cada nó representa uma computação com um tempo de execução de horas ou dias. Aqui, as unidades de execução são coleções de computadores geograficamente distribuídos.

### 2.3.2 Comunicação

A comunicação no contexto das aplicações paralelas é fundamental, pois é um requisito para a cooperação entre fios de execução. Como referido na Subsecção 2.1, esta pode ser realizada através de zonas de memória ou troca de mensagens.

**Memória partilhada** Comunicação por memória partilhada é um mecanismo em que vários fluxos de execução comunicam entre si através de um espaço de endereçamento partilhado por todos.

Um dos problemas deste modelo é a ocorrência de *dataraces*. Estas ocorrem quando pelo menos dois fios de execução acedem à mesma localização de memória, sem restrições de ordenação e se pelo menos um deles é uma escrita [CLL+02]. Estes são erros de programação que são especialmente difíceis de fazer depurar, porque podem apresentar

comportamentos diferentes com os mesmos valores de entrada [OC03] e, além disso, podem depender do escalonamento efetuado pelo Sistema Operativo. Ou seja, a ocorrência de um *datarace* depende do estado inicial do CPU, do sistema operativo e das decisões de escalonamento do sistema operativo ao longo do tempo de vida do programa. É, portanto, natural que estes erros apresentem um comportamento não determinístico.

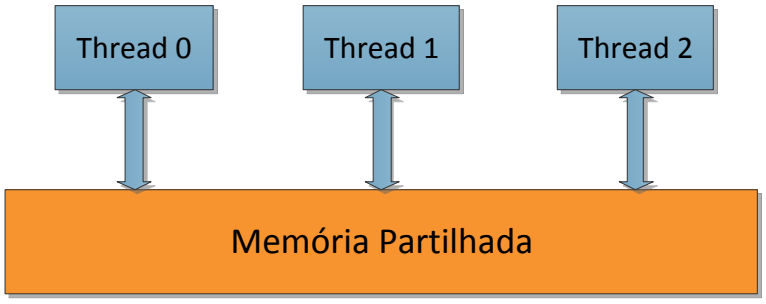



Figura 2.5: Memória partilhada [Mou11]

A comunicação via memória partilhada está intrinsecamente ligada às arquiteturas de memória partilhada. No entanto, também pode ser utilizada em arquiteturas de memória distribuída, desde que suportada por uma camada especializada de *software*. Exemplos de sistemas que implementam este modelo em arquiteturas de memória partilhada são o OpenMP [CJP07] e as bibliotecas de *threads* (e.g., Posix Threads [But97]).

A implementação de uma camada de memória partilhada virtual requer um sistema de execução (ou *middleware*) que abstraia o acesso das posições de memória remotas. O propósito destes sistemas é esconder das camadas superiores da aplicação a comunicação através de troca de mensagens (ver secção 2.3.2), necessária à implementação do modelo de memória partilhada em arquiteturas de memória distribuída. Exemplos de modelos construídos em cima da memória partilhada virtual são o Linda [LW89], as *Distributed shared memory* (DSM), e.g., TreadMarks [ACD+96] e o *Partitioned Global Address Space* (PGAS), e.g., X10 [CGS+05] e UPC [CDC99].

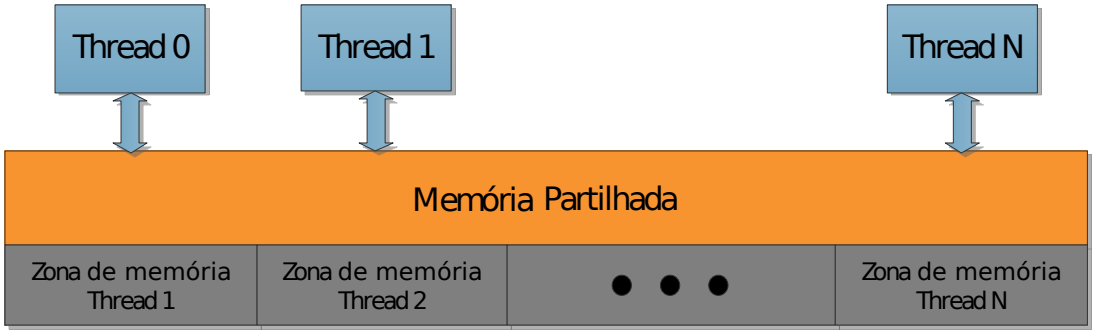



Figura 2.6: Memória partilhada virtual

Um dos problemas inerente à utilização de mecanismos de memória partilhada virtual é a abstração da localidade da memória, porque estes sistemas dão acesso transparente às memórias remotas, o que faz com que as aplicações sofram de alguma latência e limitando a sua escalabilidade.

Para resolver este problema, foi introduzido o modelo do PGAS. Este modelo não é mais que uma extensão do modelo de memória partilhada virtual com o conceito de afinidade. O conceito de afinidade [Den05] é importante, porque dá ao programador uma melhor visão sobre o espaço de endereços partilhado, sabendo onde se encontram cada objeto e distinguindo um acesso local de um acesso remoto, otimizando a performance. Neste modelo, cada fio de execução pode aceder a qualquer objeto que se encontra no espaço de endereços, seja ele local ou remoto, usando um estilo semelhante ao do modelo de memória partilhada. Cada fio de execução contribui com uma zona de memória para o espaço de endereçamento global, mantendo uma parte da sua memória para uso privado. Para que este modelo possa funcionar, existem cinco requisitos a que a camada de comunicação terá de obedecer [ULB07, YBC+07]:

- Suporte para *Remote Memory Access* (RMA)
- Baixa latência para pequena granularidade
- Comunicação não bloqueante
- Suporte para acessos concorrentes e arbitrários a memória remota
- Suporte para comunicação coletiva e mecanismos de sincronização

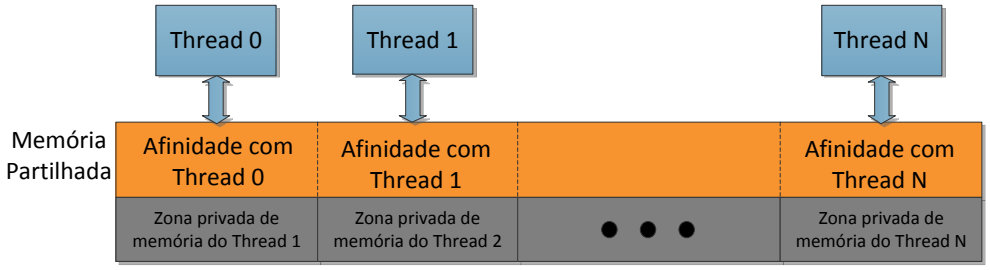



Figura 2.7: Modelo PGAS [Mou11]

**Troca de mensagens** Neste modelo, cada fio de execução só possui acesso direto à memória local e um meio de comunicar com os outros fios, através da troca de mensagens, sendo que estas podem ser ponto-a-ponto ou coletivas (enviadas para um grupo de participantes). Este modelo segue uma filosofia diferente do da memória partilhada, no sentido em que esconde as alterações do estado resultantes da troca de mensagens. A

troca de mensagens pode ser síncrona ou assíncrona, tanto no emissor como no recetor. O envio síncrono permite que os vários fios de execução possam sincronizar-se entre si.

Este modelo é, por norma, menos natural para os programadores de aplicações sequenciais, porque requer uma análise do programa para detetarem as zonas de comunicação. Por outro lado, é mais flexível [HW10].

A troca de mensagens é ortogonal à arquitetura. Está normalmente associado às arquiteturas de memória distribuída.

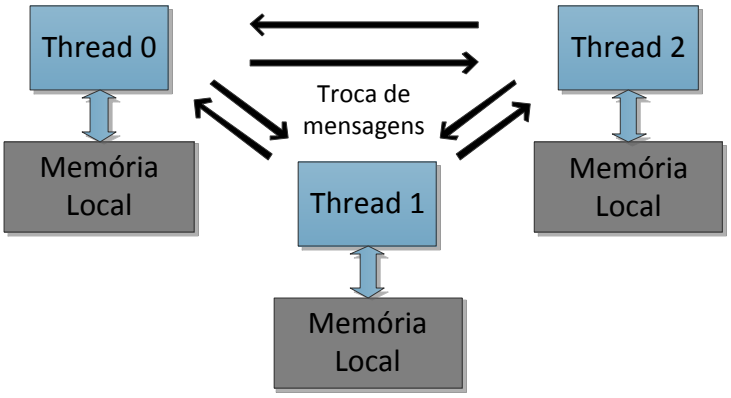



Figura 2.8: Modelo de Troca de mensagens [Mou11]

Este modelo é usado em vários níveis, tanto em alto nível como em baixo nível. Em baixo nível, este modelo é implementado em *sockets* de rede, em filas de mensagens do *UNIX System V* [HGS99] e no MPI [SOW+95] (standard de facto para comunicação via troca de mensagens em computação paralela). A alto nível, este modelo é implementado em *Remote Method Invocation* (RMI) [Sun98], em sistemas de *Remote Procedure Call* (RPC) [Sri95] e na comunicação entre serviços Web [NMM+03].

## 2.4 Modelos de Consistência de Memória

Nesta subsecção iremos falar sobre modelos de consistência de memória, sendo que não suportamos memória virtualmente partilhada. Estes são necessários nos sistemas de memória partilhada para definir a ordem das escritas e das leituras, de maneira que a memória de todos os processadores fique consistente [Man07]. O programador tem de conhecer o modelo de consistência de memória usado para garantir a correção do programa. Existem diversos modelos, desde os mais restritos até aos relaxados. Este estudo aborda quatro modelos mais relevantes:

- Consistência Sequencial;

- Consistência Local;
- Consistência Fraca;
- *Entry Consistency*.

**Consistência Sequencial** Do ponto de vista do programador, o modelo de consistência sequencial é o mais simples de perceber, estendendo este o modelo do uni-processador, e portanto, segue as premissas da memória em programas sequenciais. Lamport [MRZ95] definiu que um sistema é sequencialmente consistente se:

> O resultado de qualquer execução é o mesmo que resulta da execução por uma ordem sequencial das operações de todos os processadores e as operações de cada processador aparecem nesta sequência na ordem do programa.

O sistema assegura que todos os acessos à memória partilhada a partir de processadores diferentes, entrelaçam-se de uma determinada ordem, de forma a que a execução se comporte como se os acessos forem executados numa ordem sequencial. Garante ainda que cada escrita é vista imediatamente por todos os processadores. Esta propriedade é obtida à custa de uma quantidade significativa de mensagens para manter esta consistência, aumentando a latência de cada escrita [Cha02].

**Consistência Local** A consistência local permite que as escritas de diferentes processadores sejam vistas com ordens diferentes. Porém, as escritas de um processador têm de ser executadas na ordem em que ocorreram. Operações de sincronização explícita têm de ser usadas para acessos que devem ser ordenados globalmente.

**Consistência Fraca** Os modelos de consistência fraca não requerem que as alterações à memória sejam visíveis em todos os processadores ao mesmo tempo. Quando ocorre uma sincronização, todas as escritas anteriores têm de ser vistas na ordem em que ocorrem no programa. Existem duas aproximações conhecidas para implementar a consistência fraca: *Release Consistency* e *Lazy-Release Consistency*.

***Release Consistency*** Neste tipo de sistemas existem duas operações especiais de sincronização: *libertar* e *adquirir*. Antes de fazer uma escrita num objeto em memória, um processador tem de adquirir o objeto e depois libertá-lo, definindo uma região crítica. No *Release Consistency*, as operações de escrita só são vistas por outros processadores depois da operação de libertar. A vantagem deste tipo de consistência é que só atualiza a memória quando se executa a operação libertar. Portanto, as atualizações só ocorrem quando são necessárias e são reduzidas mensagens desnecessárias. Todavia, a maior parte dos sistemas que usam este modelo necessitam que o programador use as operações de libertar e adquirir explicitamente.

***Lazy-Release Consistency*** O *Lazy-Release Consistency* é muito semelhante ao *Release Consistency*, com a diferença de que as atualizações só ocorrem quando se executa a operação de adquirir. Quando executada, o processador que quer adquirir o objeto da memória vai determinar que modificações necessita de fazer para que a memória fique consistente.

***Entry Consistency*** Na *Entry Consistency*, os dados partilhados têm de ser explicitamente declarados e associados a um objeto de sincronização para proteger o acesso, definindo um ponto de entrada com acesso exclusivo a estes. A *Entry Consistency* tira partido da relação entre os objetos de sincronização que protegem regiões críticas e os dados partilhados dessas. Depois de um processador completar um adquirir, a *Entry Consistency* assegura que o processador vê a última versão dos dados, porque não existe outro processador a aceder à região crítica.

**Comparação crítica** No contexto dos DSM, o custo da comunicação é bastante cara, o que limita a escalabilidade dos DSM e cria outros problemas. Devido a esta característica, a granularidade está limitada a um intervalo de valores para prevenir a falsa partilha ou excessivas trocas de mensagens. No entanto, a inflexibilidade da granularidade tem um efeito negativo sobre a computação de alguns programas com requisitos de alta comunicação.

A consistência sequencial é impraticável, devido a limitações físicas, ou seja, é necessário que existam modelos mais relaxados. No caso dos modelos mais relaxados, o *Entry Consistency* é melhor que o *Lazy-Release Consistency* se a unidade de coerência for maior que uma página. Se a unidade de coerência no *Entry Consistency* for menor que uma página, então o *Entry Consistency* é melhor que o *Lazy-Release Consistency* se existir uma falsa partilha. O *Lazy-Release Consistency* é melhor que o *Entry Consistency* se existir uma localidade na memória, resultando num efeito de *prefetch* [Cha02].

| **Nível de restrição** | **Modelos de consistência** |
|---|---|
| Mais restrita | Consistência sequencial |
| | Consistência local |
| | Consistência fraca |
| | *Release consistency* |
| Menos restrita | *Entry consistency* |

Tabela 2.2: Quadro Comparativo entre os modelos de consistência de memória

### 2.4.1 Modelo de Execução

Um modelo de execução é um padrão para estruturar a execução dos programas. De seguida, vamos analisar os seguintes modelos para aplicações paralelas: *Fork/Join*, SPMD e MPMD.

***Fork/Join*** Os programas que aplicam o modelo *Fork/Join* apenas paralelizam partes do programa. Quando têm de paralelizar, criam novos fios de execução (*fork*) e o fio de execução principal vai esperar por estes (*join*). Normalmente não são criados novos fios de execução, mas o trabalho é adjudicado a um conjunto de trabalhadores em prol da aplicação. Este pode ser aplicado a programas com paralelismo de dados e a aplicações com paralelismo de tarefas.

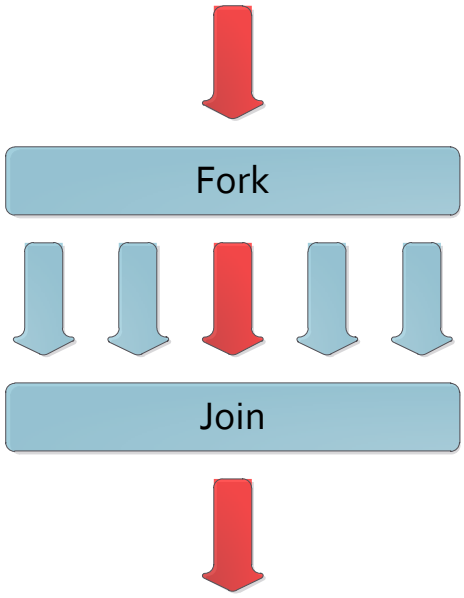



Figura 2.9: Modelo *Fork/Join*

Em memória partilhada, este modelo pode ser implementado usando uma *pool* de executores, onde quando é feito o *fork*, o trabalho é distribuído para os vários executores. Em memória distribuída, pode ser implementado com um conjunto de trabalhadores distribuídos, onde os dados e o código têm de ser enviados na operação *fork* e os sub-resultados têm de ser juntados na operação *join*.

**SPMD** Os programas SPMD executam o mesmo código em diferentes dados em paralelo. Este modelo é bastante utilizado em aplicações com paralelismo de dados. Este modelo é da responsabilidade do programador, de definir a distribuição dos dados pelos diversos fios de execução.

Este modelo em memória partilhada pode ser implementado usando vários executores com o mesmo código, mas atribuindo dados diferentes. Em memória distribuída a implementação é semelhante à de memória partilhada, mas no final é necessário juntar os sub-resultados de todos os executores.

**MPMD** O modelo MPMD define que os programas executem vários pedaços de código diferentes sobre diferentes dados. Este é bastante aplicado em programas com paralelismo de tarefas e para simular o comportamento de *Fork/Join* em arquiteturas de

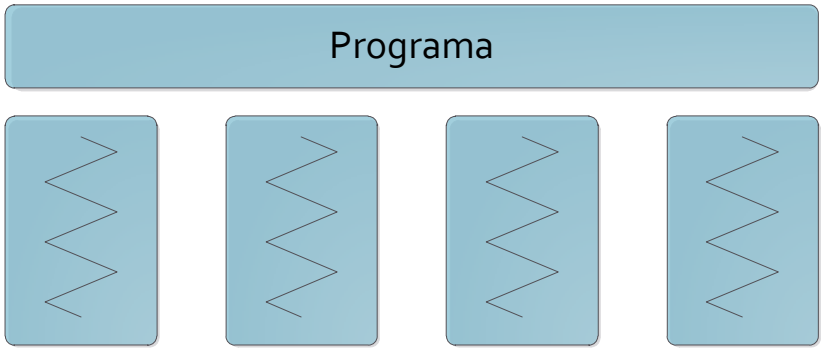



Figura 2.10: Modelo SPMD

memória distribuída.

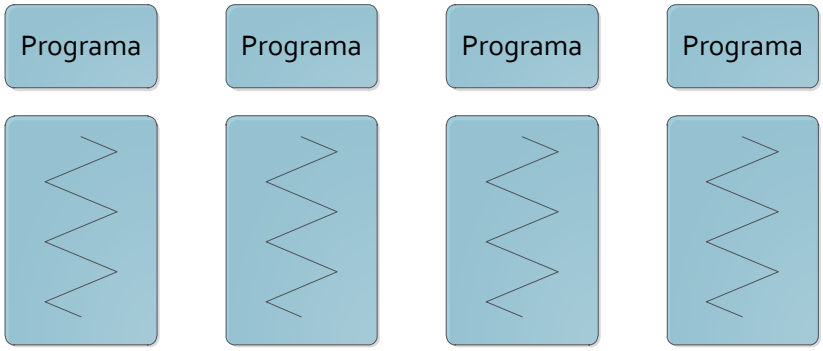



Figura 2.11: Modelo MPMD

Este modelo pode ser implementado através do modelo SPMD, atribuindo um identificador a cada trabalhador. Com este, podemos fazer distinções e executar um código diferente em cada trabalhador.

### 2.4.2 Paralelismo Implícito e Explícito

Todos os modelos descritos até este ponto se encaixam na categoria de paralelismo explicito. Este refere-se ao caso em que programador tem de anotar o paralelismo explicitamente, utilizando para tal funções de biblioteca ou construtores de linguagem. Neste contexto, o programador tem de lidar com os seguintes problemas: atribuir (possivelmente) as tarefas aos processadores; controlar a execução da tarefas; controlar a sincronização das tarefas.

A vantagem do paralelismo explícito é a de que os programadores experientes conseguem produzir soluções eficientes para problemas específicos. As desvantagens são a delegação no programador da responsabilidade por todos os detalhes da execução e o facto de, em certo casos, as soluções não serem portáveis entre arquiteturas.

O paralelismo implícito refere-se a quando um compilador ou interpretador fazem

transformações automáticas no código, para adicionar paralelismo sem que o programador intervenha. Para isso, o compilador tem de tratar dos seguintes problemas: detetar o potencial paralelismo no programa; atribuir as tarefas para a execução em paralelo; controlar e sincronizar a execução.

A grande vantagem do paralelismo implícito é libertar o programador dos detalhes da execução paralela, contribuindo para um desenvolvimento mais simples e rápido. A desvantagem prende-se com o facto de ser difícil para o compilador criar uma solução eficiente para todos os casos de aplicação.

### 2.4.3 Implementação

Um modelo de programação paralela pode ser instanciado de diferentes formas, nomeadamente bibliotecas e linguagens. Esta secção debruça-se sobre ambas.

**Biblioteca** Em desenvolvimento de *software*, uma biblioteca é uma coleção de sub-rotinas que podem ser ligadas, dinamicamente ou estaticamente, a um programa. No contexto da programação paralela, as bibliotecas são bastante úteis, pois permitem a reutilização de mecanismos de paralelização desenvolvidos por outros. Esta é a abordagem das Posix Threads [But97].

Nesta categoria também se inserem os *middlewares*: uma camada de *software* que interliga outras camadas, especialmente camadas de alto nível com camadas de baixo nível, virtualizado as últimas. Este sistema é muito usado em arquiteturas *Service Oriented Architecture* (SOA) [soa]. Uma vantagem desta abordagem é que para atualizar o *middleware* não é necessário atualizar os programas que usa, desde que as interfaces não sejam alteradas, sendo mais modular.

**Linguagem** Em computação paralela, as linguagens de programação podem conter mecanismos de paralelização embutidos. Estas linguagens podem ser divididas nas seguintes categorias:

**Linguagem paralela:** As linguagens paralelas são linguagens específicas com construções para lidar com o paralelismo. Exemplos deste tipo de linguagens são o Chapel [CCZ04], X10 [CGS+05] e o ZPL.

**Extensão de linguagem:** As extensões de linguagem têm como base uma linguagem sequencial existente, e adicionam construções para lidar com o paralelismo. Um exemplo deste tipo de linguagens é o Titanium [YSP+98], o *Unified Parallel C* (UPC) [CDC99], Sequoia [FHK+06], pSystem [LS97] e o OpenMP [CJP07].

Uma desvantagem das linguagens é que requerem aprendizagem da sintaxe e das funcionalidades destas, ao contrário das bibliotecas, que apenas requerem a aprendizagem das funcionalidades. As extensões de linguagem minimizam este problema, já que é só necessário aprender a extensão e não a linguagem completa, diminuindo a curva de aprendizagem.

### 2.4.4 Quadro Comparativo

Nesta subsecção é apresentando um quadro comparativo que resume os sistemas enumerados nesta secção e os modelos de decomposição, comunicação, execução e de implementação que aplicam.

| Sistemas | Decomposição | | Comunicação | Modelo de Execução | Implementação |
|---|---|---|---|---|---|
| | Dados | Funcional | | | |
| X10 | X | X | PGAS | MPMD | Linguagem |
| ZPL | X | | Memória partilhada | SPMD | Linguagem |
| Sequoia | X | | | SPMD | Extensão ao C |
| OpenMP | X | X | Memória partilhada | *Fork/Join* | Anotações e biblioteca |
| Intel Array Building Block | X | | Memória partilhada | *Fork/Join* | Biblioteca |
| OpenCL | X | | Memória partilhada | SPMD | Linguagem, próxima do C |
| pSystem | | X | Memória partilhada | MPMD | Extensão ao C |
| Cilk | X | | Memória partilhada | *Fork/Join* | Extensão ao C |
| Chapel | X | X | PGAS | MPMD | Linguagem |
| Posix Threads | X | X | Memória partilhada | *Fork/Join* | Biblioteca |
| UPC | X | | PGAS | SPMD | Extensão ao C |
| Titanium | X | | PGAS | SPMD | Extensão ao Java |

Tabela 2.3: Quadro Comparativo entre os modelos e os sistemas

## 2.5 Java em *Clusters*

Nesta subsecção iremos abordar alguns *middlewares*, como por exemplo, Parallel Java e o ProActive. Este estudo tem como objetivo compreender o conjunto de funcionalidades de *middlewares* que dão suporte a execução de aplicações paralelas.

### 2.5.1 cJVM/Jessica

O cJVM [AFT99] é uma implementação da *Java Virtual Machine* (JVM) para *clusters*. Por trabalhar no nível da JVM, é capaz de explorar otimizações baseadas na semântica do Java, e.g., migração de *threads*. Este sistema usa o MPI para comunicar. Mas as principais contribuições deste sistema são:

- Uma arquitetura que disponibiliza uma *Single System Image* (SSI) de uma JVM num *cluster*;
- Um modelo de memória distribuída;

- Um novo modelo de objetos, que distingue entre a vista da aplicação de uma classe de um objeto e a sua implementação, tendo conhecimento do uso de um objeto em específico para aumentar a performance;
- Uma implementação de *threads* que suporta de maneira transparente *stacks* distribuídas.

O Jessica também explora o mesmo conceito de SSI do cJVM [MWL00], mas com uma pequena diferença de implementação. Enquanto o cJVM para aceder a objetos remotos usa um mecanismo de referências remotas, o Jessica usa um subsistema de DSM para guardar os objetos remotos. Este sistema usa o Treadmarks para comunicar, fornecendo um espaço DSM. Por sua vez, o Treadmarks usa MPI para comunicar.

### 2.5.2 Parallel Java

O Parallel Java [Kam07] é uma biblioteca e *middleware* para computação paralela e distribuída, funcionando em computadores SMP, em *clusters* de uni-processadores e em *clusters* de SMP's. O desenvolvimento deste foi inspirado no OpenMP e no MPI, unificando as funcionalidades deste sistemas para aplicações Java. Usando uma única *Application Programming Interface* (API) é possível desenvolver aplicações paralelas nas várias arquiteturas.

O *middleware* do Parallel Java é responsável por gerir a fila de trabalhos no *cluster* e lançar os processos nas máquinas do *cluster*,i.e., para o desenvolvimento de aplicações em Parallel Java para máquinas SMP não é necessário o *middleware*. Os nós neste *middleware* são classificados por *Frontend Processor*, que é o nó principal, e por *Backend Processors*, que são os outros nós. Este *middleware* está dividido em quatro componentes:

***Job Scheduler Daemon*** - Este componente é executado no *Frontend Processor* e é responsável por vigiar o estado dos *Backend Processors*, manter a fila de trabalhos e por uma interface Web para disponibilizar o estado do *cluster*;

***Job Frontend Process*** - É o representante da aplicação que está a ser executada no *cluster* no *Frontend Processor*;

***Job Launcher Daemon*** - O *Job Launcher Daemon* é responsável por lançar o *Job Backend Process* por cada trabalho;

***Job Backend Process*** - É responsável pelas classes necessárias para executar o trabalho, os argumentos de consola do trabalho, executar o trabalho e de retransmitir o *standard input* , o *standard output* e o *standard error* para o *Job Frontend Process*.

### 2.5.3 ProActive

O ProActive [HCB04] é um *middleware* em Java para computação concorrente, paralela e distribuída que oferece serviços de alto nível, e.g., migração, comunicação em grupo

e segurança. A implementação atual pode usar três tipos de comunicações: RMI, Jini (para descoberta) e um protocolo baseado em *Extensible Markup Language* (XML). O seu modelo de programação baseia-se em componentes, sendo estes um conjunto de objetos ativos.

Uma aplicação concorrente ou distribuída elaborada usando ProActive, é composta por um número de entidades chamados objetos ativos. Cada objeto ativo tem o seu *thread* de controlo e decide qual é a ordem em que trata as chamadas a métodos recebidas que são automaticamente guardadas numa fila de pedidos pendentes. As chamadas a métodos de objetos ativos são sempre assíncronas com o retorno de um objeto transparente, chamado *future*, e escondem as comunicações remotas. A sincronização é tratada por um mecanismo chamado *wait-by-necessity*. Todos os objetos ativos a correr numa JVM pertencem a um nó que disponibiliza uma abstração para a sua localização física. A qualquer momento, a JVM possui um ou vários nós.

Este distingue-se dos outros por permitir a composição hierárquica de componentes. Sendo este feito no nível de integração em vez de ao nível da programação. Os componentes são definidos num ficheiro XML que especifica a interface do componente, composição e requisitos.

No geral, ProActive desenvolve alguns princípios básicos e simples [TC10]: as atividades são distribuídas, objetos acessíveis remotamente; Interações são feitas através de chamadas de métodos assíncronas; Resultados das interações são futuros; Os evocadores podem esperar pelos resultados usando um mecanismo chamado de *wait-by-necessity*.

### 2.5.4 Jaroodi

Em [AJMJS03], Jarrodi descreve um *middleware* para desenvolver aplicações paralelas e distribuídas em *clusters* e redes heterogéneas. Este sistema fornece uma infraestrutura desenvolvida inteiramente em Java, no modelo de memória partilhada, que torna portável, segura e com a capacidade de correr em vários modelos de programação, e.g., JOPI [MAJJS02] ou o modelo *Distributed shared object* (DSO) (ver figura 2.12).

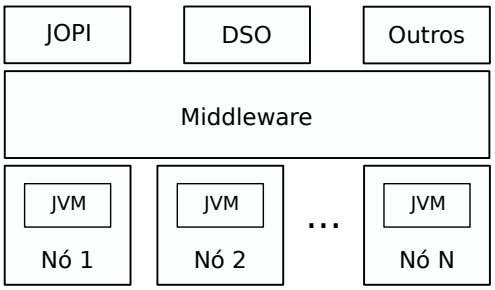



Figura 2.12: Infraestrutura do *middleware* [AJMJS03]

Este sistema utiliza o conceito de agente, para fornecer os serviços flexíveis e expansíveis. Os agentes são responsáveis por lançar,escalonar, suportar a execução de código

Java paralelo/distribuído e também de gerir, controlar, monitorizar e escalonar os recursos disponíveis num *cluster* ou numa coleção de sistemas heterogéneos. Quando uma aplicação paralela Java é submetida, um agente executa as seguintes tarefas:

- Examinar os recursos disponíveis e escalonar o trabalho para execução, tendo em conta o balanceamento da carga;
- Converter as classes do utilizador em *threads*, carregando estas e executa-as em máquinas remotas;
- Monitorizar e controlar os recursos, disponibilizando funções de monitorização e controlo para o utilizador.

### 2.5.5 Virtualized Self-Adaptive Heterogeneous High Productivity Computers Parallel Programming Framework

*Virtualized Self-Adaptive Heterogeneous High Productivity Computers Parallel Programming Framework* (VAPPF) [CCS+09] é um *middleware* orientado para *clusters* de heterogéneos, como por exemplo, *multi-cores*, SMP e *Field-programmable gate array* (FPGA), criando uma camada intermédia, virtualizando o *hardware* e permitindo o paralelismo de tarefas e de dados.

O *runtime* está repartido em dois componentes, o *Node-Level Virtual Machine Monitor* (NVMM) e o *System-Level Virtual Infrastructure* (SVI). O NVMM encontra-se em todos os nós que fazem parte do *middleware* e é responsável por executar, monitorizar e migrar as máquinas virtuais e da comunicação entre máquinas virtuais. O SVI é o componente principal do *middleware*, existindo um para todo o sistema. Este é responsável por gerir e escalonar os NVMM. Para poder mapear os processadores virtuais e os dados em processadores específicos, o SVI tem de manter o estado global do sistema, disponibilizando um SSI. Este componente pode automaticamente descobrir novos recursos do *middleware*.

O modelo de programação tem como objetivo os seguintes requisitos: produtividade, escalabilidade e facilidade de programação. Este modelo é a *interface* entre o *runtime* e as aplicações, oferecendo uma máquina virtual de armazenamento de dados e de execução aos programadores.

### 2.5.6 FlexPar

O FlexPar [UMG08] é um *middleware* baseado em componentes para o desenvolvimento de aplicações paralelas em ambientes heterogéneos. Este suporta vários paradigmas de programação paralela, mas no prototipo desenvolvido apenas foi desenvolvido o paradigma *Communicating Sequential Processes* (CSP), mais concretamente, occam-pi e JCSP. Este paradigma ajuda o desenvolvimento, já que evita problemas de concorrência, como por exemplo *deadlocks*.

Este *middleware* é desenvolvido em Java, o que o torna portável. A arquitetura é composta por três camadas: i) um *kernel* do *runtime* ii) componentes para extensão, e iii) as

aplicações. O *kernel* é mínimo e só instancia as extensões quando necessárias e destrói-as quando estas deixam de ser necessárias. As extensões incluem um *loader* e um *binder*. O *loader* implementa os mecanismos para inicializar e instanciar um determinado processo CSP, enquanto o *binder* cria uma determinada conexão entre dois processos CSP.

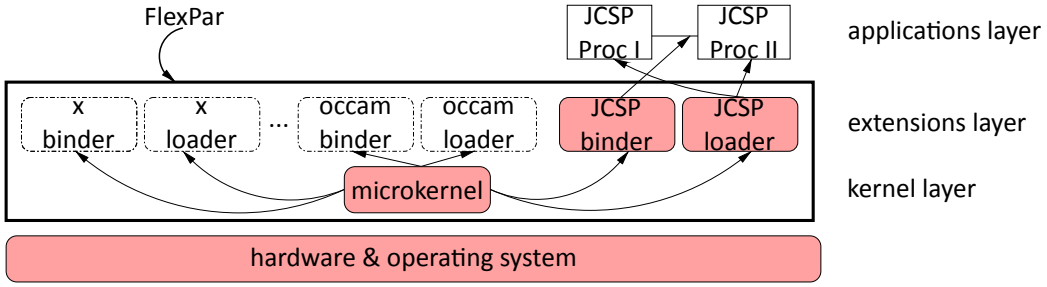



Figura 2.13: Infraestrutura do *middleware* FlexPar [UMG08]

## 2.6 Sistemas de Execução de linguagens paralelas para *clusters*

Para termos um maior conhecimento sobre a construção de sistemas de execução de linguagens de computação paralela para *clusters* fizemos um levantamento de algumas linguagens e dos seus sistemas de execução.

### 2.6.1 Linguagens PGAS

Nesta subsecção iremos abordar alguns sistemas que execução de linguagens que instanciam o modelo PGAS, como por exemplo, UPC e o Titanium.

**UPC:** O UPC é uma extensão à linguagem C para arquiteturas HPC em larga escala. Disponibiliza um modelo de programação unificado para arquiteturas de memória partilhada ou distribuída, baseado no modelo PGAS. O UPC segue o modelo de computação SPMD, onde o nível de paralelismo é fixo e definido na inicialização da aplicação.

O principal objetivo do UPC é libertar o programador dos detalhes do paralelismo e do acesso remoto dos dados, de maneira a explorar a localidade dos dados, para uma melhor performance [CDC99].

O nosso trabalho não irá abordar questões de consistência de memória. No entanto relativamente à implementação, o UPC recorre ao *Global-Addressing-Space Networking* (GASNET) no seu sistema de comunicação e implementa a consistência local para manter a memória consistente.

**Titanium:** O Titanium é uma linguagem para o desenvolvimento de aplicações Java de HPC para multi-processadores de memória partilhada e distribuída que instância o modelo PGAS. Não é mais que uma extensão do Java 1.4, que utiliza um compilador próprio para traduzir o código fonte em C. Não utiliza, portanto, a JVM como sistema de execução.

Atualmente, o sistema de execução do Titanium suporta diversas plataformas, incluindo arquiteturas uni-processador, multi-processadores de memória partilhada, *clusters* de memória distribuída e supercomputadores. Para comunicação, o Titanium, como o UPC, usa GASNET. Como modelo de consistência de memória, este sistema implementa consistência local.

### 2.6.2 Linguagens APGAS

As linguagens *Asynchronous Partitioned Global Address Space* (APGAS) estendem o modelo PGAS, introduzindo a distribuição de tarefas. O modelo APGAS organiza as computações numa coleção de *places* lógicos. Um *place* encapsula dados e uma ou mais tarefas que operam sobre os dados. Contudo, este tipo de linguagens apresentam um problema: a localização da computação é explícita, sendo que estas possuem pouca escalabilidade, devido ao facto de o escalonamento ser realizado manualmente e a sua complexidade crescer com a complexidade da computação. Nesta subsecção iremos analisar duas linguagens que implementam este modelo, o CHAPEL e o X10.

**Chapel:** O CHAPEL é uma linguagem desenhada para *High-End Computing* (HEC) com o foco na produtividade de desenvolvimento, combinando a performance com a facilidade de desenvolvimento. O design desta linguagem tem como áreas chave o *multithreading*, *locality-awareness*, orientação por objetos e programação genérica [CCZ04]. A arquitetura base desta linguagem tem como alvo sistemas de *petaflops*.

Numa fase inicial, a implementação do CHAPEL transformava a linguagem para C, usando bibliotecas portáveis de comunicação, por exemplo, MPI ou *Aggregate Remote Memory Copy* (ARMCI), para aumentar o número de arquiteturas paralelas que o CHAPEL possa executar. Numa segunda fase, a implementação não só gera código C, como também gera código Assembly .

Para comunicar, a implementação do CHAPEL usa GASNET, ARMCI, MPI e *Parallel Virtual Machine* (PVM) e para manter a memória consistente, implementa Consistência Sequencial.

**X10:** O X10 é uma linguagem de programação desenvolvida pela IBM Research em colaboração com vários parceiros académicos. Esta é *type-safe*, paralela e orientada a objetos. O seu alvo são as arquiteturas NUCC.

Para que o sistema seja produtivo e com boas performances, os principais objetivos do X10 são a segurança, a analisabilidade, escalabilidade e flexibilidade [CGS+05].

Este sistema usa *sockets*, MPI, *Low-level application programming interface* (LAPI) e *Active Messages* para comunicar e para manter a memória consistente usa o *Entry Consistency*.

### 2.6.3 Sequoia

A linguagem Sequoia foi desenvolvida pela universidade de Stanford com uma abordagem diferente para o problema da distribuição de dados num *cluster* [FHK+06]. Em vez da tradicional partição horizontal dos dados, o Sequoia particiona os dados verticalmente, tendo em conta a hierarquia de memória.

O Sequoia é uma linguagem de programação que desenha para assistir o programador na estruturação de programas paralelos *bandwidth-efficient*, de forma a que facilmente sejam portáveis para novas máquinas. O desenho do sistema foca-se nos seguintes aspetos: introdução da noção de hierarquia de memória no modelo de programação, para ganhar portabilidade e performance; o uso da abstração de tarefa, isto é, unidades de computação encapsuladas, que incluem informação relevante, por exemplo, informação sobre a comunicação.

Para comunicar entre os vários nós do *cluster*, o sistema Sequoia usa MPI-2.

### 2.6.4 pSystem/di_pSystem

O pSystem é um sistema de programação paralela baseado no modelo de memória partilhada. É uma extensão à linguagem C, onde o paralelismo é expresso através de um mecanismo de anotações explícitas, que requerem pequenas mudanças de sintaxe do programa original. Em tempo de compilação, o pré-processador vai fazer *parse* as anotações e gerar o programa C correspondente, contendo chamadas ao pSystem para ativar a paralisação para um *subset* das funções do programa.

O escalonamento de trabalhos paralelos entre as várias unidades de processamento num sistema de programação paralela, têm uma elevada importância para atingir altas performances na execução de aplicações paralelas. Usualmente, os vários sistemas de programação paralela resolvem o problema do escalonamento com escalonamento estático (normalmente baseado em análise de compilador) ou com alocação de trabalho dinâmico (i.e., *runtime*) [LS97]. O pSystem permite o desenvolvimento e introdução de várias heurísticas dinâmicas de escalonamento, tendo sido utilizado para o seu estudo em termos de propriedades e de performance.

O di_pSystem [SPL99] é uma extensão ao pSystem para arquiteturas de memória distribuída. O di_pSystem usa o MPI para comunicar.

Em termos de consistência de memória, o di_pSystem usa consistência sequencial ou fraca, sendo que a fraca que usa é uma variante do *Lazy-Release Consistency*, mas que o programador tem de explicitamente indicar que vai adquirir o recurso mas não tem que indicar quando o liberta.

### 2.6.5 Dryad

Dryad [IBY+07] é um *middleware* para computação paralela, suportando apenas paralelismo de dados. As aplicações desenvolvidas sobre o Drayd são compostas por vértices computacionais e por canais de comunicação, sendo que estes formam um grafo acíclico

de fluxo de dados. O Drayd corre a aplicação ao executar os vértices num conjunto de computadores disponíveis, comunicando por ficheiros, por canais *Transmission Control Protocol* (TCP) ou por filas *First In, First Out* (FIFO) de memória partilhada. Os vértices são simples e normalmente escritos como programas sequenciais, sem criação de *threads* ou utilização de *locks*. A concorrência é conseguida a partir do escalonamento dos vértices para serem executados em simultâneo com vários computadores, ou em múltiplos *cores*.

A arquitetura deste *middleware* está dividida em 3 partes (ver figura 2.14): i) *Job Manager* (JM), ii) *Name Server* (NS), iii) *Daemon* (D). O *Job Manager* é responsável pela execução da aplicação no *cluster*, constrói o grafo da comunicação e escalona os trabalhos pelos os recursos disponíveis. Este pode ser executado na maquina do cliente ou no *cluster*. O *Name Server* disponibiliza a lista de recursos disponíveis no *cluster* e a topologia da rede, de maneira a que o escalonador tenha em conta a localização dos recursos. O *Daemon* que está a executar em cada nó do *cluster*, é responsável por criar os processos em nome do *Job Manager*. Na primeira vez que um vértice é executado num computador, o binário é enviado do *Job Manager* para o *Daemon*.

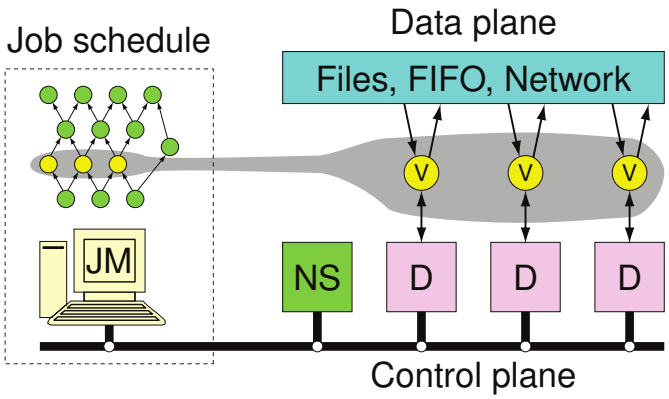



Figura 2.14: Infraestrutura do *middleware* Dryad [IBY+07]

## 2.7 Discussão

Após o estudo do estado de arte relacionado com computação HPC em Java para *clusters* de *multi-cores*, identificamos alguns problemas nos modelos de programação estudados, sendo que o principal problema, com a exceção do Proactive, é a não existência de espaço para o compilador gerar código especifico aos processadores ou para o sistema de execução se adaptar às características do *hardware*:

- Titanium apresenta um modelo SPMD estático e horizontal que força o programador a usar outros mecanismos para explorar o paralelismo intra-nós;
- Parallel Java requer uma programação híbrida de troca de mensagens/memória partilhada;

- MapReduce é limitado a problemas sem comunicação e sincronização;
- X10 delega para o programador o escalonamento de tarefas, que compromete a eficiência da modularidade e escalabilidade.

O modelo de programação do Proactive é mais adequado a estes ambientes, nomeadamente porque este usa paralelismo de dados hierárquico, popularizado por o Sequoia [FHK+06]. No entanto, a maior parte deste esforço é realizado ao nível da integração do sistema ao invés de ao nível da programação.

Em tempo de execução, todos estes sistemas dependem da JVM para tratar da heterogeneidade da plataforma. As diferenças da configuração dos vários nós têm de ser identificadas e geridas pelo programador. Além disso, não existe suporte para heterogeneidade dos vários nós, como por exemplo, o uso de *Graphic Processing Units* (GPUs). Apesar de este trabalho não pretender fornecer o suporte completo para este tipo de processadores, todo o desenho da interface e da arquitetura do sistema de execução tem em mente a posterior adição do suporte a GPU's. O trabalho realizado em adicionar suporte a GPU em sistemas e linguagens existentes evidenciou as limitações de sobre-expor o programador ao modelo de execução subjacente [Dav11]. Na nossa opinião, estes problemas deverão ser tratados pelo compilador e sistema de execução, o programador apenas deverá ser responsável pela identificação do que deverá ser paralelizado e não sobrecarregá-lo com o "como".

Dado este estado da arte, a nossa opinião é a de que é possível fazer contribuições ao nível do modelo de programação e do sistema de execução. Portanto, propomos um *middleware* em Java para o desenvolvimento de aplicações paralelas através de uma vasta diversidade de arquiteturas, nomeadamente máquinas de memória partilhada, *clusters* de máquinas e GPUs[1]. Para este efeito, fornecemos uma interface de programação de alto nível baseada na noção de serviço e localidades, processadores virtuais que podem ser mapeados nos nós ou cores disponíveis.

O modelo de programação é apoiado por um sistema de execução com uma arquitetura inspirada nos sistemas operativos, no sentido de que não só define uma interface para o programador, mas também uma interface que permite a integração de diferentes tecnologias, permitindo a adaptação do *middleware* à natureza da arquitetura alvo. Esta camada de adaptação suporta vários níveis de aplicação: processador, localidade, *cluster* e inter-*cluster*. Assim, uma dada configuração pode ter impacto no sistema em diferentes níveis.

[1] O suporte para GPUs será tratado em trabalho futuro

# 3

# O *Middleware*

Este capítulo apresenta um middleware para computação paralela em *clusters* de multicores, nomeadamente o modelo de programação que oferece e a arquitetura do seu sistema de execução. O trabalho realizado partiu de uma base desenvolvida no âmbito de uma dissertação de mestrado [Mou11]. Assim sendo, o capitulo termina com a discriminação do trabalho realizado nesta dissertação.

## 3.1 Objetivos

O *middleware* desenvolvido no âmbito desta dissertação tem como objetivo servir de suporte ao desenvolvimento de aplicações paralelas, tanto em arquiteturas de memória partilhada e distribuída. Para tal oferece uma interface e sistema de execução independentes da arquitetura alvo. Apresenta-se como um intermediário entre as aplicações e a arquitetura *hardware*, virtualizando a possível natureza heterogénea da última. O *middleware* fornece várias funcionalidades comuns ao desenvolvimento de aplicações paralelas e concorrentes, tais como: paralelismo de tarefas e de dados, comunicação, gestão da concorrência e sincronismo. As suas características diferenciadoras são:

- Fornecer uma plataforma simples de desenvolvimento de aplicações paralelas através de uma interface de alto nível, que abstrai a gestão do paralelismo, aumentando a produtividade do programador;
- Fornecer ao programador primitivas para expressar a afinidade entre a as computações e os dados, otimizando o desempenho;
- Fornecer suporte a vários modelos de programação paralela para que possa ser utilizado como uma *framework* de aplicações paralelas e concorrentes ou como suporte

a sistemas de execução de linguagens de programação;

- Ser independente da plataforma, fornecendo uma plataforma onde as aplicações possam executar em várias arquiteturas, tais como arquiteturas de memória partilhada, distribuída ou híbridas.

## 3.2 Desenho

O *middleware* apresenta-se como um máquina virtual que abstrai a possível natureza heterogénea da arquitetura alvo numa interface bem definida, apresentando uma visão uniforme e independente da plataforma. Tal permite que uma mesma aplicação possa executar em arquiteturas diferentes, com por exemplo *multi-cores*, *clusters* ou por uma máquina com um acelerador *hardware*, o que permite que as aplicações possam executar sem que a sua estrutura não seja alterada. O programador pode optar por negligenciar a natureza distribuída do *hardware* e portanto aumentando o seu nível de abstração, ou estar ciente da localização das computações nos recursos disponíveis. A figura 3.1 apresenta a visão geral do *middleware*.

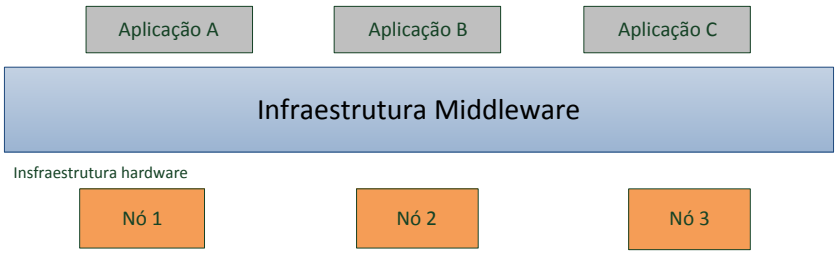



Figura 3.1: Visão geral do *middleware* de memória partilhada [Mou11]

Cada nó representa uma máquina com arquitetura partilhada e executa uma instância do *middleware*, que denominamos por versão *base*. A interligação destas instâncias via rede fornece a infraestrutura de suporte à execução de aplicações sobre arquiteturas de memória distribuída.

### 3.2.1 Interface de Programação

A interface de programação do *middleware* é baseada no conceito de serviço. Um serviço fundamenta-se no modelo de objeto ativo [LS96] para disponibilizar um componente modelar para a construção de aplicações paralelas escaláveis. Apresenta-se como um modulo de *software* opaco que disponibiliza as suas funcionalidades através de uma interface previamente definida. Este paradigma distribuído mapeia de forma natural nas arquiteturas de memória distribuída e mesmo nos *multi-cores*, onde o aumento gradual do número de cores faz com que estas arquiteturas partilhem, cada vez mais, propriedades intrínsecas às arquiteturas de memória distribuída.

Os serviços executam em localidades (figura 3.2), um processador virtual que pode ser mapeado num ou mais núcleos (cores) de um nó. Por omissão, uma localidade é mapeada um nó. Este conceito é parecido ao que pode ser encontrado nas linguagens de programação X10 [CGS+05] ou Chapel [CCZ04], embora estes definam um *place* (ou localidade) como uma porção do espaço de endereçamento global mais um conjunto de fluxos de execução que operam sobre esse espaço. O *middleware* proposto não oferece um espaço particionado nativamente. O conceito de localidade oferecido é mais próximo do proposto em [YVPH08], virtualizando os processadores disponíveis, comportando-se como recetor para as execuções dos serviços, contendo um ou mais fluxos de execução. Os fluxos de execução executam as tarefas dos serviços de forma concorrente e estes podem aceder à zona de memória privada do serviço para comunicar entre si.

Os serviços podem ser autónomos, ou seja, podem realizar computações autonomamente, ou podem ser passivos, executando computações apenas como resposta a um estimulo externo (invocação de um método na interface do serviço). Os serviços autónomos executam concorrentemente. Além disso, a execução de métodos do serviço pode ser assíncrona dentro do espaço de endereçamento da localidade, ou seja, o espaço de endereçamento é partilhado pelos vários fluxos de execução.

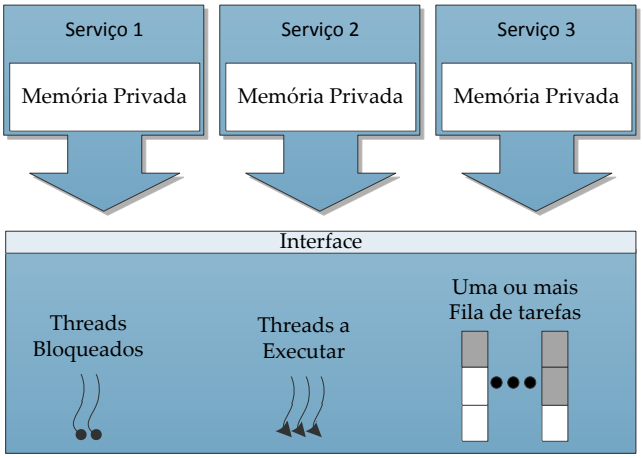



Figura 3.2: Localidade

Uma aplicação é composta por um conjunto de serviços, autónomos e passivos, que são distribuídos pelas localidades de acordo com uma política de escalonamento pré-definida. Um dos objetivos da linha de investigação onde este trabalho se insere é providenciar uma interface que seja independente da arquitetura, para que a aplicação seja facilmente portada de uma arquitetura *multi-core* para *clusters*. Com este objetivo em mente, o *middleware* pode ser usado como uma biblioteca para as aplicações em arquiteturas de memória partilhada, ou instalado num *cluster*, onde pode suportar a execução de várias aplicações concorrentemente.

**Serviço** A interface de programação do *middleware* assume a forma de uma API em Java. O paralelismo é expresso por uma coleção de anotações Java ou por invocação de métodos na classe ServiceProvider (listagem 3.1), a classe base para a implementação de um serviço. Os serviços autónomos têm de estender a classe ActiveServiceProvider, uma sub-classe de ServiceProvider que implementa a interface java.lang.Runnable. O método run() é automaticamente executado quando o serviço é submetido.

```
public class ServiceProvider implements Service {

  // Invocações assíncronas
  public <R> IFuture<R> invoke(String methodName, Object... args)
      throws PlaceNoSuchMethod;
  public <T, P, R> IFuture<R> distReduce
    (Distribution<T> distr, Reduction<P, R> red, String methodName, Object... args)
       throws PlaceNoSuchMethod;


  // Sincronização - operações sobre monitores
  protected IMonitor createMonitor();
  public void beginAtomic();
  public void endAtomic();
  public ICondition newCondition(ConditionCode code);

  // Sincronização - operações sobre barreiras
  protected IBarrier createBarrier();
  protected IBarrier createBarrier(IBarrier b);

  // Operações de gestão
  protected <T> T copy(T elem);
  public UUID getApplicationId();
  public UUID getID();
  public void cancel();
  public void setAffinity(IPlace p);
  public List<UUID> getAffinity():
  public Level getLevel();
  public Place getPlace();
}
```

Listagem 3.1: Interface do serviço

A interface do serviço está dividida em três tipos de primitivas: i) paralelismo, ii) sincronização e iii) gestão. As primitivas de paralelismo são invoke e distReduce, que oferecem os mecanismos para suportar o paralelismo de tarefas e de dados, respetivamente. Ambas recebem como parâmetro o método do serviço a invocar e o seus argumentos. Na primitiva distReduce é ainda necessário indicar uma política de distribuição e redução (detalhado mais à frente).

Com as primitivas de sincronização é possível indicar blocos atómicos (com as primitivas beginAtomic e endAtomic), criar condições no monitor por omissão do serviço (newCondition), criar novos monitores (createMonitor) e criar barreiras (createBarrier).

As primitivas de gestão destinam-se a dar suporte à execução dos serviços. Estas incluem o retorno do identificador do serviço e da aplicação onde este se insere (detalhes sobre o conceito de aplicação serão dados na secção 3.2.1.1), o cancelamento da execução

do serviço (cancel), definição e retorno de afinidades entre serviços (setAffinity e getAffinity, mais detalhes na secção 4.3) e o retorno da localidade que o serviço se encontra a executar e o nível a que este executa (getLevel e getPlace, mais detalhes na secção 3.2.2.2). Os níveis em que um serviço pode executar são: nó e *cluster*, mas antevê a inclusão de novos níveis.

**Anotações** Pretende-se que a interface para as aplicações seja o mais simples e concisa possível, permitindo ao programador focar-se na lógica da aplicação em vez de detalhes de gestão do paralelismo em geral, ou seja, criação de *threads*, sincronização e comunicação. A interface descrita anteriormente já oferece abstrações de alto nível para expressar paralelismo, mas esta obriga o programador a mudar de paradigma.

Para minimizar o esforço do programador, desenvolvemos um mecanismo de anotações que tem como base a filosofia do OpenMP [CJP07], em que o programador anota o código para expressar paralelismo. Estas anotações são interpretadas quando aplicadas a classes que estendem ServiceProvider e têm como objetivo adicionar meta-dados ao código para posterior instrumentalização. O código será alterado por forma a introduzir chamadas a métodos da interface do serviço em tempo de ligação ou criar novas classes auxiliares em tempo de compilação.

As Listagens 3.2 e 3.3 mostram a especificação e estrutura da implementação de um serviço que fornece operações matemáticas.

```
public interface Math extends Service{
  int max(int[] vector);
  int sum(int[] vector);
  int[][] mult(int[][] m1, int[][] m2);
  int[][] translate(int[][] m);
}
```

Listagem 3.2: Especificação da interface do serviço Math

```
public class MathProvider extends ServiceProvider implements Math{
  int max(int[] vector) { [...] }
  int sum(int[] vector) { [...] }
  int[][] mult(int[][] m1, int[][] m2) { [...] }
  int[][] translate(int[][] m) { [...] }
}
```

Listagem 3.3: Estrutura da implementação do serviço Math

**Comunicação** Os serviços comunicam através da invocação de métodos das interfaces que disponibilizam. Em prol da uniformidade da interface em ambientes de memória partilhada e distribuída, todos os parâmetros deveriam ser passados por valor em arquiteturas de memória partilhada e distribuída. No entanto, muitas das vezes tal impõe um custo elevado em arquiteturas de memória partilhada. Assim, em beneficio da performance, serviços que se encontram a executar na mesma localidade podem passar

apontadores para objetos partilhados que, implicitamente, define um outro meio de comunicação. Note que é restrito a serviços que estejam a executar numa mesma localidade. Todas as comunicações entre serviços de localidades distintas copia todos os parâmetros para o espaço de endereçamento do recetor. O sistema de execução tal como está implementado ainda não suporta a passagem de referências para argumentos e portanto ainda existe uma quebra na uniformização do interface. Esta questão será abordada no futuro e discutimo-la em mais detalhe na secção 6.1.

Foi definida a anotação @Copy, que permite ao programador forçar a passagem de parâmetros por valor dentro de uma localidade. Para isso, basta ao programador adicionar a anotação aos parâmetros que quer que sejam passados por valor na definição do método, como no exemplo apresentado na listagem 3.4.

```
public int[][] matrixMult(@Copy int[][] m1, @Copy int[][] m2){
  [...]
}
```

Listagem 3.4: Exemplo de uma cópia de argumentos utilizando a anotação

Neste exemplo, as variáveis m1 e m2 são copiadas, fazendo com que este método aceda apenas a variáveis locais.

**Paralelismo de tarefas** Os métodos que compõem a interface do serviço podem ser explicitamente anotados como tarefas (*tasks*), para serem executadas concorrentemente dentro do espaço de endereçamento do serviço. A anotação do método é transparente ao invocador, que não tem de modificar a chamada ao método. Note que um agente Java (Java *agent*) modifica tanto o invocador como o invocado de todos os métodos anotados com @Task, introduzindo implicitamente futuros. Esta aproximação sobrepõe, enquanto for possível, a execução do invocador com a do método invocado, i.e., até que o primeiro aceda ao resultado calculado pelo segundo.

O objetivo da anotação é permitir ao programador expressar paralelismo através de meta-dados a um método que foi programado sequencialmente. O código do método é instrumentalizado para que seja encapsulado numa tarefa e para que esta seja submetida para execução paralela (invocando o método invoke da interface do serviço). A execução do método será feita assincronamente, retornando um futuro (objeto que implementa o interface java.util.concurrent.Future), sendo que todos os acessos à variável que originalmente deveria conter o resultado do método são substituídos pela obtenção síncrona do resultado computado (invocação do método get sobre o Future). Na listagem 3.5 encontra-se um exemplo de aplicação desta anotação.

Como se pode ver, o programador desenvolveu o método para o cálculo do número de Fibonacci de forma sequencial e anotou-o como uma tarefa para que seja executada em paralelo. As invocações recursivas (linhas 7 e 8) também são realizadas em paralelo. O resultado destas são guardadas nas variáveis locais x e y, que são transformadas em variáveis do tipo Future. Uma referência a qualquer uma destas variáveis é necessária,

```
1  @Task
2  public int fib(int n){
3    int x,y;
4
5    if (n <= 1) return n;
6
7    x = fib(n-1);
8    y = fib(n-2);
9    return x + y;
10 }
```

Listagem 3.5: Método que calcula o número de Fibonacci com anotações

ou seja no retorno do método, será chamada a primitiva bloqueante get do Future. A utilização desta anotação simplifica o código, pois elimina a criação e submissão explícita de tarefas.

**Paralelismo de dados** O paralelismo de dados é também disponibilizado ao nível do método. O modelo de programação segue um paradigma de Distribute-Map-Reduce (DMR) [Edu12] que permite a aplicação paralela de métodos sequenciais a partições distintas dos parâmetros de entrada. Este paradigma é constituído pelos seguintes passos (figura 3.3):

**Distribuir:** aplica uma distribuição no parâmetro de entrada fornecido para obter uma coleção de elementos do mesmo tipo que o parâmetro.

**Mapear:** aplica a operação do serviço a cada elemento do vetor criado anteriormente.

**Reduzir:** aplica a redução para combinar todos os resultados gerados pelos passo anterior para computar o resultado final.

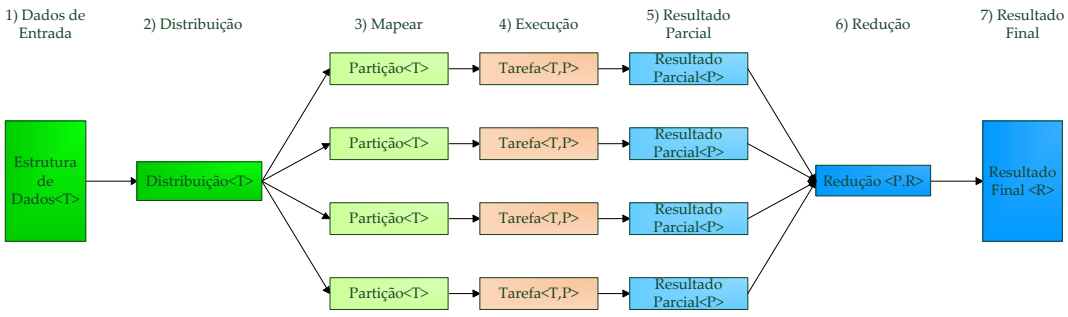



Figura 3.3: Paralelismo de dados [Mou11]

As computações DMR no *middleware* são tarefas que são redefinidas para usarem esta estratégia de paralelismo na implementação do serviço. Para esse efeito as mesmas devem ser: inicialmente anotadas com @DistRedTask na interface do serviço, e depois decoradas com as anotações @DistributionPolicy e @ReductionPolicy na implementação

do serviço. Estas duas ultimas anotações recebem como argumento o tipo da classe que implementa a distribuição/redução e os seus argumentos, se existirem. A listagem 3.6 exemplifica a aplicação deste paradigma para encontrar o valor máximo num vetor. A distribuição parte o vetor de entrada e a redução determina o máximo global de todos os máximos locais computados pelas tarefas.

```
@ReductionPolicy(reduction=ArrayMaxReduction.class)
public int max(
   @DistributionPolicy(distribution=ArrayDistribution.class) int[] array) {

  int max=Integer.MIN_VALUE;
  for(int i : array)
    if (i > max) max = i;
  return max;
}
```

Listagem 3.6: Aplicação do paradigma *Distribute-Map-Reduce*

O âmbito da anotação @ReductionPolicy encontra-se ao nível do método, sendo que a classe que implementa a estratégia a aplicar tem que respeitar a interface Reduction (listagem 3.7). A anotação @DistribuitionPolicy é aplicada ao nível dos argumentos do método. A interface a respeitar neste caso é @Distribution (listagem 3.8).

```
public interface Reduction <P,R> {
  public R reduce(Iterator<P> results);
}
```

Listagem 3.7: A interface Reduction

```
public interface Distribution <T> {
  public T[] distribution();
  public void setPartitions(int length);
}
```

Listagem 3.8: A interface Distribution

A interface Distribution especifica dois métodos: setPartitions, para definir o número de partições a criar, e distribution, para aplicar a estratégia implementada. A interface Reduction especifica um método, reduce, que recebe um iterador de todos os resultados parciais (do tipo P) e retorna o resultado do mesmo tipo que o retorno do método anotado.

De momento, as anotações para expressar o paralelismo de tarefas apenas estão definidas para o nível do *cluster* (mais detalhes na secção 4.6.1). Atualmente para definir o paralelismo de dados usa-se o método distReduce da classe ServiceProvider, sendo que está em vista para trabalho futuro a expressão deste tipo de paralelismo intra-localidade com estas anotações.

**Sincronização** Os principais mecanismos de sincronização oferecidos pelo *middleware* são os monitores e as barreiras. No entanto, apenas foram definidas anotações para criar

métodos atómicos, usando a anotação @Atomic, sendo que falta anotações para condições e para barreiras. Portanto, a sua utilização tem de ser realizada a através das interfaces oferecidas IMonitor e IBarrier (Listagens 3.9 e 3.10).

```
public interface IMonitor {
  void beginAtomic();
  void endAtomic();
  ICondition newCondition(ConditionCode conditionTest);
}
```

Listagem 3.9: Interface dos monitores

A interface IMonitor disponibiliza três métodos, dois para definir blocos atómicos (beginAtomic e endAtomic) e um para a criação de variáveis de condição [Hoa74]. Como parâmetro ao método de criação de condições, é necessário fornecer uma variável do tipo ConditionCode, que contem um método boolean call(), onde é analisada a expressão booleana associada à condição.

```
public interface IBarrier {
  public void await();
  public void register();
}
```

Listagem 3.10: Interface das barreiras

As barreiras são hierarquias, semelhantes aos *phasers* do Habanero Java [CZSS11]. Estas fornecem duas primitivas: o register, que permite registar um fio de execução na barreira, e await que bloqueia até que todos os fios de execução cheguem até à barreira. A listagem 3.11 mostra um exemplo de utilização de barreiras, onde se evidencia que parte do código do calculo do problema NBodies [BH86]. O algoritmo aplicado tem um fase inicial em que calcula as forças envolvidas e uma segunda fase em que calcula as novas posições dos corpos.

```
@Task
public void nbody(Body[] bodies,IBarrier b){
  b.register();
  for(Body body:bodies){
    body.calculateForces();
  }
  b.await();
  for(Body body:bodies){
    body.calculatePosition();
  }
}
```

Listagem 3.11: Exemplo do uso das barreiras

Cada serviço e localidade tem associado um monitor por omissão, o que permite que se possa sincronizar tarefas no âmbito de um serviço ou sincronizar serviços a correr na mesma localidade. Ambos, serviço e localidade, permitem a criação de novos monitores

para permitir elementos de sincronização distintos. O monitores do *middleware* disponibilizam os seguintes mecanismos:

**Blocos atómicos:** Estes são suportados pelos monitores, usando as primitivas beginAtomic e endAtomic. Para facilitar o uso, criamos uma anotação, @Atomic, para ser utilizada ao nível do método para expressar a atomicidade deste. A listagem 3.12 mostra a simplicidade do uso desta anotação.

**Condições:** Estas, como habitualmente, disponibilizam pré-condições para a entrada nos monitores. A criação destas é feita com a primitiva newCondition no serviço, em que se deve passar como argumento um objeto do tipo ConditionCode. A classe ConditionCode é uma classe abstrata onde se deverá implementar o método Boolean call() e neste é definida a condição.

```
@Atomic
public void update(int v){
  int x = this.val;
  x = x + v;
  this.val= x;
}
```

Listagem 3.12: Exemplo do uso da anotação @Atomic

#### 3.2.1.1 Computação em *Clusters*

Quando é usado num ambiente de *cluster*, o *middleware* tem a habilidade de executar várias aplicações concorrentemente. As aplicações têm de ser encapsuladas num serviço autónomo especial denominado Application. Uma aplicação tem de conter todos os serviços necessários para a sua execução. O seu objetivo é servir como veiculo para o envio destes serviços para o *middleware* a executar no *cluster* e para a associação destes serviços à JVM cliente. Nesta associação são também incluídos *stubs* para os canais *standard* da JVM cliente, permitindo que aplicações sejam desenvolvidas da mesma forma do que se fossem desenvolvidas em máquinas individuais.

Para ajudar à construção de aplicações para *clusters* escaláveis, providenciamos um meio para compor serviços hierarquicamente através da classe ServiceAggregator (listagem 3.13). Esta apresenta um aspeto verboso pois recorre às variáveis de tipo (genéricos) do Java para garantir, em tempo de compilação, a compatibilidade dos tipos. Em geral pretende-se garantir que só se podem agregar serviços e em particular pretende-se restringir os tipos de serviços a agregar, como será detalhado na descrição do comportamento de *Memória particionada*. Estas composições agregam serviços de acordo com um comportamento pré-definido. O resultado final é um novo serviço que herda as interfaces dos serviços que o compõem. Sendo elas próprias um serviço, as agregações podem também elas tomar parte de novas agregações, criando uma árvore de agregações que a compõem. Atualmente existem quatro comportamentos disponíveis.

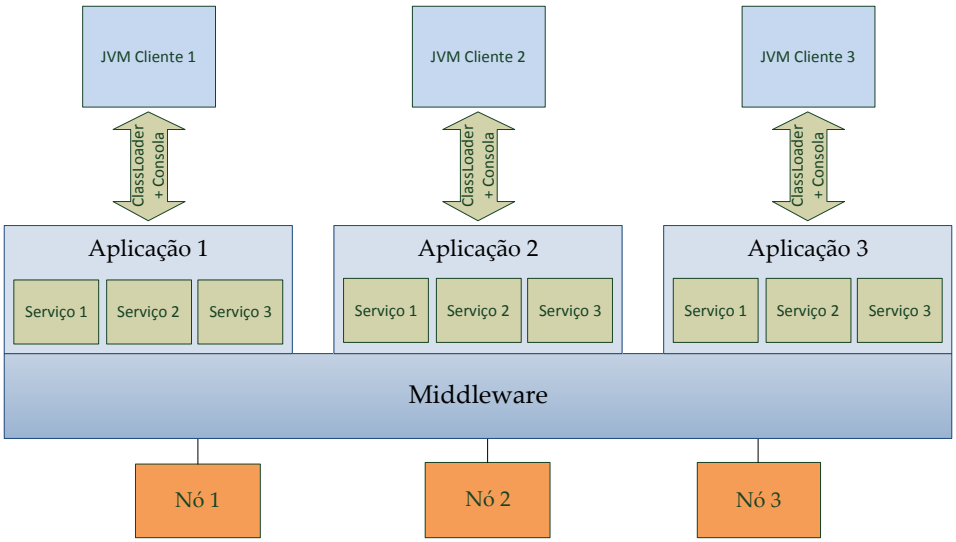



Figura 3.4: Redireção da consola e *class loader*

```
public class ServiceAggregator {
  public static <I extends Service, T extends Service> I distRed(T service, int
      number);
  public static <I extends Service, T extends Service> I distRed(Class<T> service,
      int number);
  public static <I extends Service> I distRed(I[] services);

  public static <I extends Service> I placePool(Class<? extends I> service,int
      number);
  public static <I extends Service> I placePool(IServiceScheduler s, Class<?
      extends I> service, int number);
  public static <I extends Service> I placePool(I[] services);
  public static <I extends Service> I placePool(IServiceScheduler s, I[] services);

  public static <K, V, I extends Service & PartitionedMem<K, V>> I partMem(
      IPartitioner<K,V> p, Class<? extends I> service, int number);
  public static <K, V, I extends Service & PartitionedMem<K, V>> I partMem(Class<?
      extends I> service, int number);
  public static <K, V, I extends Service & PartitionedMem<K, V>> I partMem(I[]
      services);
  public static <K, V, I extends Service & PartitionedMem<K, V>> I partMem(
      IPartitioner<K,V> p, I[] services);

  public static <I extends Service, J extends Service> Facade<I, J> facade(I s1, J
      s2);

}
```

Listagem 3.13: ServiceAggregator

***Distribute-Map-Reduce*** Este comportamento aplica o paradigma DMR pelos serviços, permitindo o uso destas estratégia de paralelismo hierarquicamente. O primeiro nível distribui o trabalho pelos serviços e no segundo nível, o trabalho é distribuído pelos fios

de execução dentro de cada serviço. Ou seja, aplica a operação da figura 3.3 entre os serviços e dentro de cada serviço aplica a mesma operação (figura 3.5). Este comportamento usa as mesmas anotações do paralelismo de dados para o primeiro nível, sendo que o segundo nível é expressado pelo método distReduce.

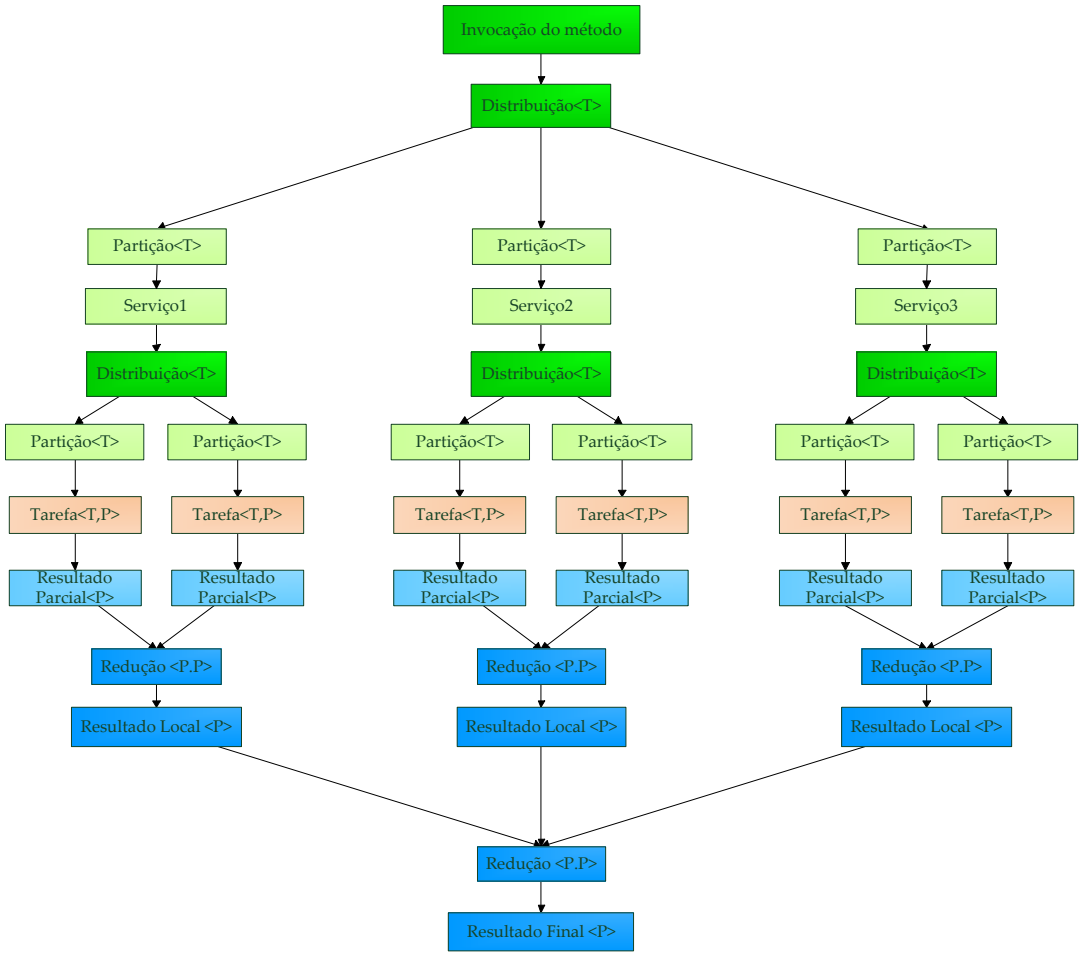



Figura 3.5: *Distribute-Map-Reduce*

A listagem 3.14 ilustra um exemplo de uso do mecanismo de DMR, onde é calculado o valor de $\pi$ através do método de monte carlo. Podemos ver o primeiro nível a ser expressado com as anotações @ReductionPolicy e @DistributionPolicy e o segundo a ser expressado no interior do método. Neste caso, as reduções terão de ser diferentes que as do primeiro nível, para retornar o valor de $\pi$ temos que dividir o número de pontos dentro da circunferência pelo número de rondas e no segundo nível só temos de somar o número de pontos.

***Pool* de Serviços** Este comportamento transpõe o conceito de *pool* de *workers* para o âmbito dos serviços, onde todas as invocações dos métodos é intersectada por um escalonador que decide que serviço será escolhido para tratar do pedido. Este escalonador

```
@ReductionPolicy(reduction=ReductionPI.class,params={"rounds"})
public double getPI(@DistributionPolicy(distribution=DistributionPI.class,params={"
    rounds"}) double rounds) {
  Distribution<Double> distr = new DistributionPI(rounds);
  Reduction<Double, Double> red = new InnerReductionPI();
  return distReduce(distr, red, new PITask()).get();
}
```

Listagem 3.14: Exemplo de uso do mecanismo de DMR

tem de implementar a interface IServiceScheduler (listagem 3.15), que especifica dois métodos: setSize que recebe o número de serviços a escalonar, e getIndex que retorna o índice do próximo serviço que irá tratar do pedido.

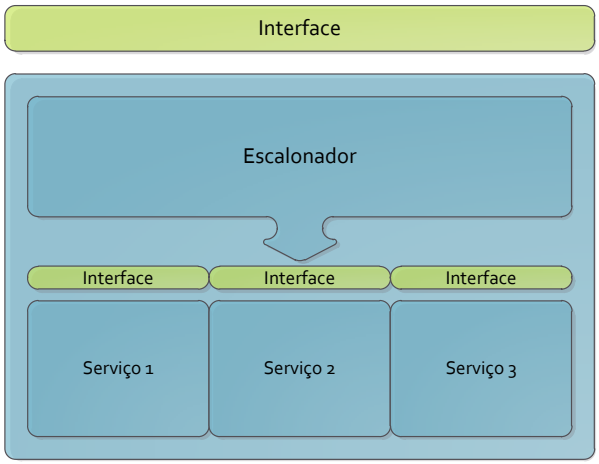



Figura 3.6: *Pool* de Serviços

```
public interface IServiceScheduler {
  int getIndex();
  void setSize(int length);
}
```

Listagem 3.15: Interface IServiceScheduler

Se não for indicada uma política de escalonamento, por omissão é utilizada uma política de *round-robin*. A listagem 3.16 mostra um pequeno exemplo de utilização deste comportamento, usando o escalonador por omissão. A figura 3.6 mostra visualmente este comportamento usando o serviço MathProvider que é um serviço de funções matemáticas. Após agregar é criado um novo serviço com a mesma interface dos serviços que o componham, sem que seja necessário alterar o serviço a agregar.

**Memória particionada** Este comportamento distribui uma chave de uma estrutura de dados entre a memória privada dos serviços agregados. Um particionador determina a localização dos dados. Para isso, na interface do serviço tem de estender um serviço

```
Math m = ServiceAggregator.placePool(MathProvider.class,3);
System.out.println(m.fib(10));
```

Listagem 3.16: Exemplo de criação e utilização do *pool* de Serviços

especial, o PartitionedMemService<K,V>, onde K é o tipo da chave e V o tipo do valor, e a implementação do serviço tem de estender AbstractPartitionedMemService<K, V>. Na interface do serviço, para cada método, é necessário indicar que parâmetro irá ser usado como chave, através da anotação @Key. Na implementação, os valores da estrutura de dados são manipuláveis a partir dos métodos getValue e setValue da classes AbstractPartitionedMemService<K, V>. Nas Listagens 3.17 e 3.18 apresenta-se um exemplo de um serviço Banco que usa este comportamento. Neste exemplo usamos como chave das contas bancárias um inteiro, ou seja, o número da conta, e como valor o seu saldo.

```
public interface Banco extends PartitionedMemService<Integer,Double>{
  public void depositar(@Key int conta, double valor);
  public void levantar(@Key int conta, double valor);
  public void transferencia(@Key int contaOrigem, int contaDestino, double valor);
  public double getSaldo(@Key int conta);
}
```

Listagem 3.17: Serviço Banco

**Façade** Este comportamento cria um façade automático de dois serviços, sendo também um serviço. Pode ser composto hierarquicamente de maneira a podermos compor mais que dois serviços, oferecendo-os através de uma única referencia. Este mecanismo tem como principal objetivo a possibilidade de expor para o exterior um conjunto de serviços em execução num dado *cluster*. As Listagens 3.19 e 3.20 ilustra um exemplo de criação e utilização dum Façade respetivamente, em que são agregados os serviços Math, Buffer e Banco. O façade oferece uma única operação, get, que permite ao cliente obter a referencia para o serviço pretendido.

### 3.2.2 Arquitetura

A arquitetura do *middleware* é inspirada nas arquiteturas clássicas de sistemas operativos, na medida em que este está dividido em três camadas (figura 3.9):

**Interface para as Aplicações:** centrada na noção de serviço e localidade. Esta camada é a API Java e serve como *framework* para o processamento das anotações. Pretende oferecer, de uma forma simples, as funcionalidades para o desenvolvimento de aplicações paralelas e independentes da plataforma.

**O Núcleo:** providencia o suporte base para a execução dos serviços e localidades em

```
public class BancoProvider extends MapPartitionedMemService<Integer, Double>
    implements Banco{

  public void depositar(int conta, double valor) {
    Double aux = this.getValue(conta);

    aux+=valor;

    this.setValue(conta,aux);
  }

  public void levantar(int conta, double valor) throws NotEnoughFundsException{
    Double aux = this.getValue(conta);

    if(aux>=valor){
      aux-=valor;
      this.setValue(conta,aux);
    }else{
      throw new NotEnoughFundsException();
    }
  }

  @Atomic
  public void transferencia(int contaOrigem, int contaDestino, double valor) throws
       NotEnoughFundsException{
    this.levantar(contaOrigem, valor);
    this.depositar(contaDestino, valor);
  }

  public double getSaldo(int conta) {
    return this.getValue(conta);
  }
}
```

Listagem 3.18: Implementação do serviço banco

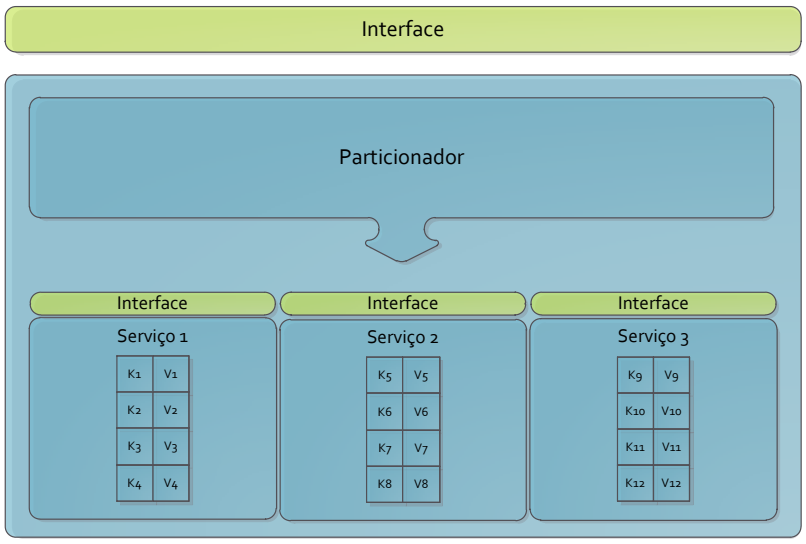



Figura 3.7: Memória particionada entre Serviços

```
Math math = new MathProvider();
Buffer<String> buf=new BufferProvider<String>();
Banco b = new BancoProvider();

Facade<Math,Buffer<String>> facade1=ServiceAggregator.facade(math, buf);
Facade<Facade<Math,Buffer<String>>,Banco> facade2=
                              ServiceAggregator.facade(facade1, buf);
```

Listagem 3.19: Exemplo de criação do façade

```
System.out.println(facade2.get(Math.class).fib(6));
facade2.get(Buffer.class).put("teste");
facade2.get(Banco.class).depositar(10,100);
```

Listagem 3.20: Exemplo de uso do façade

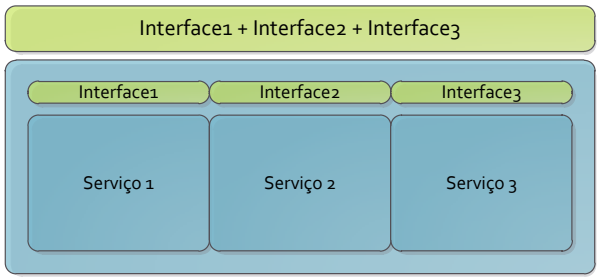



Figura 3.8: Façade de Serviços

arquiteturas partilhadas e distribuídas. Está dividido em módulos, que serão abordados com mais detalhe na Subsecção 3.2.2.1. Estes fornecem as varias funcionalidades para a execução de aplicações paralelas, independentes da tecnologia e arquitetura.

**A Camada de Abstração da Tecnologia (CAT):** uma camada de adaptação que permite a integração de tecnologias distintas para o suporte à execução de um ou mais módulos da camada de cima. O âmbito de um *driver* pode ser local a um nó (por exemplo a política de gestão da *pool* de trabalhadores) ou global ao *middleware* (por exemplo o protocolo de comunicação entre localidades). Este conceito permite adaptar o *middleware* às características do ambiente de execução, nomeadamente ao *hardware* e ao *software* disponível.

#### 3.2.2.1 Módulos

Esta secção descreve a camada nuclear, dando uma visão mais pormenorizada do sistema. A camada do núcleo contem toda a lógica do *middleware* que é independente da tecnologia. Esta é composta por vários módulos, cada um com a sua funcionalidade especifica. De seguida são apresentados e descritos os módulos que o constituem:

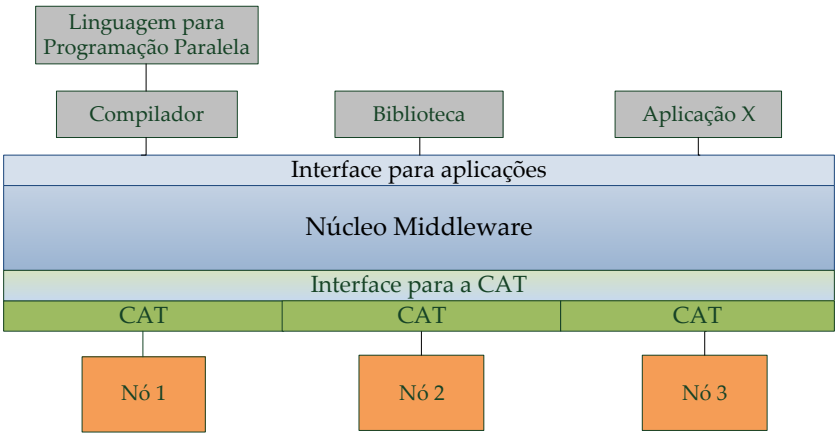



Figura 3.9: Visão global do *middleware* [Mou11]

**Gestor de inicialização** Este módulo inicializa o sistema em cada localidade, lançando uma instância do *middleware* nos nós especificados, com a CAT especificada. A configuração da CAT é definida num ficheiro XML de nome Config.xml. A sua estrutura e conteúdo são validados por um XML Schema.

**Gestor de aplicação** Este é responsável por lançar as aplicações nos *clusters*. Providencia isolamento para a sua execução de forma a não interferirem entre elas. Para tal, disponibiliza a cada aplicação um *class loader* dedicado e ligações ao cliente (suporte para o redirecionamento do *standard input*, *output* e *error*). Este *class loader* é responsável por pedir à JVM cliente todas as classes necessárias para a execução da aplicação.

**Gestor de serviços** A localização física do serviços pode ser transparente para o utilizador. Quando assim é, estes podem ser migrados para melhorar o balanceamento de carga sem que a funcionalidade da aplicação seja afetada. Este módulo é responsável por determinar a localização inicial dos serviços via uma heurística de escalonamento e de migrá-los, de acordo com uma heurística de balanceamento de carga que tem como objetivo nivelar a carga entre os nós. As implementações das heurísticas são da responsabilidade de *drivers* da camada CAT.

**Gestor de tarefas** Este gestor é responsável pela execução das tarefas dos serviços dentro das localidades. O *middleware* executa as tarefas dos serviços assincronamente, escalonando-as entre uma, ou mais, *pools* de trabalhadores. A implementação destas *pools* é delegada na camada CAT.

**Comunicação** A comunicação entre serviços geograficamente dispersos é transparente para a aplicação. O compilador gera *stubs* para cada um destes serviços necessários pela aplicação, enquanto o *middleware* garante que todos os clientes têm na sua localidade uma

instância do serviço ou um *stub* para este. A comunicação entre o *stub* e o fornecedor do servidor é realizada através de troca de mensagens, de acordo com o protocolo e tecnologia definido pela camada CAT.

**Sincronização** O módulo de sincronização redireciona os pedidos de criação de monitores e barreiras para a camada CAT, funcionando como uma fábrica destes.

**Distribuição e Redução** Este módulo é responsável por suportar o paradigma DMR. Sendo que a sua implementação está delegada na camada CAT, permitindo assim várias implementações sobre diversos tipos de *hardware*, nomeadamente *clusters*, CPU's e GPU's.

**Estatística** Este módulo é responsável por guardar as estatísticas sobre o estado global do sistema. Este módulo é decisivo para a implementação dos *drivers* de escalonamento e de balanceamento de carga. Estas são transportadas em todas as mensagens trocadas entre localidades.

Existem relações de dependência entre alguns módulos. Estas estão ilustradas na figura 3.10 em que o sentido da seta denota a relação de dependência.

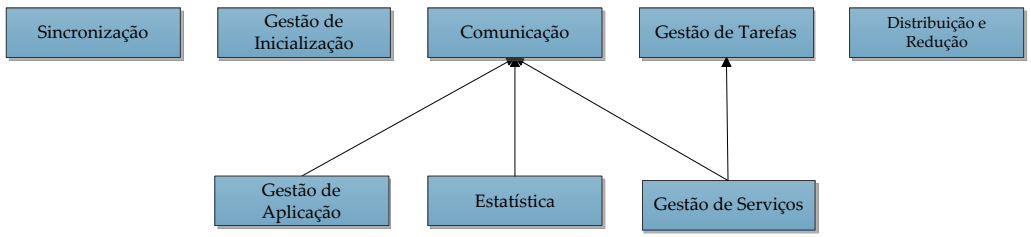



Figura 3.10: Dependências dos módulos

#### 3.2.2.2 *Drivers*

Esta secção descreve de forma mais pormenorizada a camada CAT. A camada do CAT contem as implementações da lógica dependentes da arquitetura ou tecnologias. A Tabela 3.1 apresenta e descreve os *Drivers* que constituem a camada CAT. A cada *driver* são associados níveis onde este executa. No caso em que o *middleware* se encontra em execução numa só máquina, os *drivers* do nível de *cluster* não são declarados. No caso de um *driver* estar associado a dois ou mais níveis, como por exemplo a Distribuição e Redução, então será necessário declarar o *driver* para cada nível, sendo que poderão ser iguais ou diferentes, tendo uma implementação especifica e especializada para cada nível.

A figura 3.11 apresenta as dependências entre módulos e *drivers*.

| Driver | Nível | Descrição |
|---|---|---|
| *Ranking* de Nós | *Cluster* | Seleciona e ordena os nós para o escalonamento da aplicação |
| Escalonador | *Cluster* | Escalona os serviços da aplicação nos nós do *cluster* |
| Balanceador da carga | *Cluster* | Balanceia a carga no sistema, migrando serviços |
| Executor de Tarefas | Nó | Executa as tarefas geradas pelos serviços |
| Comunicação | *Cluster* | Realiza a comunicação entre os nós do *middleware* |
| Monitores | Nó | Fornece uma fabrica de monitores |
| Barreiras | Nó | Fornece uma fabrica de barreiras |
| Distribuição e Redução | Nó e *Cluster* | Implementa o mecanismo de DMR |

Tabela 3.1: *Drivers* da camada CAT

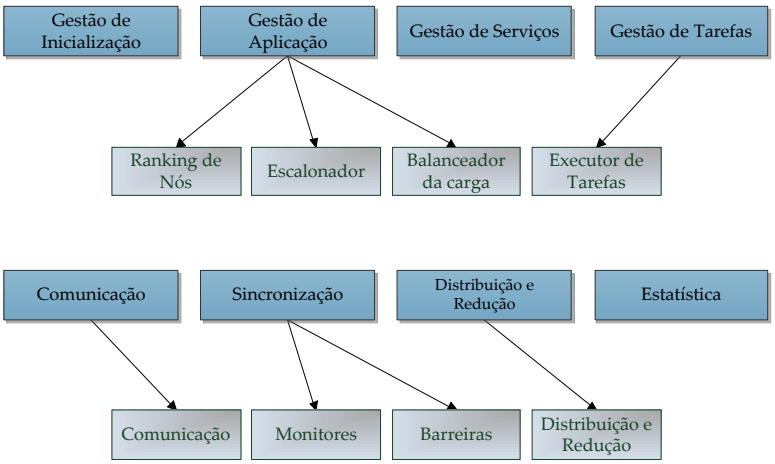



Figura 3.11: Dependência entre os módulos e *drivers*

## 3.3 Funcionalidades implementadas

Como será detalhado no próximo capítulo, a transição de ambientes de memória partilhada para ambientes de memória distribuída incidiu essencialmente na composição de instâncias base desenvolvidas por Mourão [Mou11].

A implementação realizada por Mourão não possuía qualquer suporte para memória distribuída. A sua versão do *middleware* comportava-se como uma biblioteca, no sentido em que não possuía uma interface de programação remota e este estava embutida nas aplicações, sendo que, existia uma instância do *middleware* por aplicação. Atualmente o *middleware* suporta a execução de várias aplicações, garantido o isolamento de execução e de carregamento de classes. Para tal oferece mecanismos para o lançamento das aplicações num cluster, através de uma interface de programação remota. O *driver* de comunicação não estava implementado e não existia um protocolo de comunicação entre

nós.

Apesar da implementação de Mourão focar apenas ambientes de memória partilhada, a arquitetura por si proposta já contemplava ambientes de memória distribuída. No entanto, durante o trabalho realizado no âmbito desta tese optou-se por efetuar algumas alterações. Inicialmente não existia o conceito de Serviço, no seu desenho unicamente existiam localidades e interfaces, que poderiam migrar entre nós. Estas localidades eram uma mistura das atuais localidades com os serviços, onde possuíam um espaço de endereçamento privado, uma ou mais filas de tarefas prontas a executar e um conjunto finito de fluxo de execução disponíveis. Estas poderiam ser estendidas com operações especificadas pelo utilizador, definindo uma localidade especializada (figura 3.12). Outro conceito que não existia era o de Aplicação, uma vez que na sua versão anterior o *middleware* executava uma aplicação por instância do *middleware* e atualmente uma instância pode executar várias aplicações.

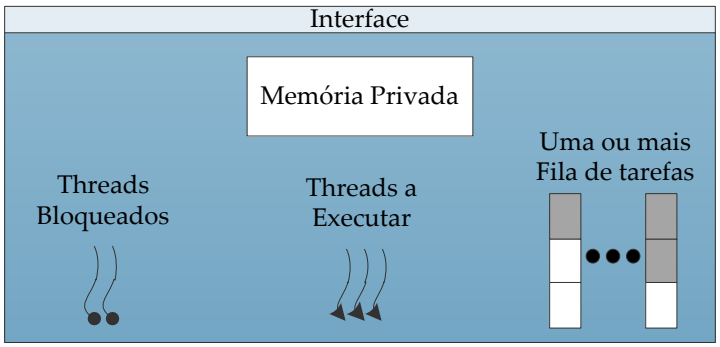



Figura 3.12: Desenho inicial da localidade [Mou11]

O desenho, apesar de já ter sido elaborado com o intuito de suportar ambientes distribuídos heterogéneos, não consagrava os módulos de Gestão de Aplicação, Gestão de Serviços e de Estatística e os drivers de *Ranking* de Nós e Escalonador.

O executor de tarefas foi redesenhado para que pudesse suportar diferentes implementações ao nível da camada CAT. Anteriormente a sua implementação estava incluída no núcleo. Foram também introduzidas as barreiras, o que requereu alterações na interface base do serviço e a adequação do *middleware* para o seu suporte.

# 4

# Implementação

Este capitulo descreve de forma pormenorizada o trabalho realizado para implementar o *middleware* em arquiteturas de memória distribuída. Este começa por abordar a questão da mudança de paradigma, o lançamento de uma aplicação, a comunicação, a sincronização e a agregação de serviços.

## 4.1 Requisitos de um sistema de execução distribuído

Para implementar um sistema de execução distribuído, é necessário que este tenha uma boa performance e que abstraia a distribuição de trabalho. Para desenvolver esta camada existem três abordagens [AFT99]:

- Criar uma camada por cima do sistema de execução existente, usando bibliotecas de terceiros. Esta abordagem faz com que a execução num *cluster* não seja totalmente transparente;
- Construir um sistema de execução por cima de uma infraestrutura para *clusters*, por exemplo, PGAS. Nesta abordagem consegue-se apresentar um visão do *cluster* como um sistema SSI, mas é incapaz de usar a semântica da linguagem para aumentar o desempenho;
- Construir um sistema de execução que tem embutido o conceito de *cluster* e que é executada por cima deste, mas que o abstrai da aplicação.

Após uma analise destas abordagens de desenvolvimento desta camada de software, detetamos um conjunto de requisitos para o desenvolvimento dos modelos de programação paralela para este tipo de arquiteturas. Existem vários requisitos necessários para que

possamos implementar sistemas desta natureza: comunicação remota, gestão de consistência de memória, escalonamento, performance e tolerância a falhas.

**Comunicação remota:** a colaboração entre processos a executar em máquinas diferentes requer primitivas de comunicação remota. Estas permitem aos processos sincronizarem-se e trocarem dados entre si. O assunto é desenvolvido em detalhe no apêndice A, onde serão analisadas e comparadas várias bibliotecas de comunicação. As primitivas de comunicação remota podem ser classificadas em dois grupos *Two-sided* e *One-sided*.

**Gestão de consistência de memória:** Como foi descrito na subsecção 2.3.2, o modelo de memória partilhada pode ser implementado através do modelo de troca de mensagens, sendo que não implementamos este mecanismo. No entanto, uma verdadeira abstração do acesso às memórias partilhadas requer a existência de um sistema de gestão de consistência de memória. Os modelos de consistência de memória fornecem uma especificação formal de como o sistema de memória se apresenta ao programador, abstraindo o comportamento real do sistema de memória [AG96]. O middleware a implementar não fornece uma camada de memória virtualmente partilhada. Esta questão será abordada em trabalho futuro.

**Escalonamento:** Um sistema que faça uma distribuição automática do trabalho entre os vários nós do *cluster* requer um escalonador. Existem dois tipos de escalonadores, os de curto prazo e de longo prazo [SGG05]. Os de curto prazo, ou também chamados de escalonadores de CPU, são responsáveis por selecionar da fila de processos prontos, o próximo processo a ser executado. Este tipo de escalonadores está embutido no sistema operativo, não necessitando, normalmente, de implementações especializadas.

Por outro lado, o escalonador de longo prazo, ou escalonador de admissão, fica a cargo do sistema de execução a implementar. Este decide que trabalhos são admitidos na fila de processos prontos de um nó, ou seja, num *cluster* este tipo de escalonador é responsável por decidir que nó vai executar um dado trabalho.

**Performance:** Um requisito fundamental neste tipo de sistemas é a performance. Existem vários fatores que podem prejudicar a performance e estes têm de ser alvo de implementações otimizadas, como por exemplo, comunicação e escalonador.

Na comunicação, podemos explorar a performance usando uma primitiva de comunicação com baixa latência[1]. Estas são importantes, principalmente em sistemas com paralelismo nos dados, porque o grão[2] da comunicação é bastante fino e, se a comunicação for de alta latência, a penalização é bastante pesada.

[1] A latência é a diferença de tempo entre o início de um evento e o momento em que os seus efeitos se tornam percetíveis, ou seja, no caso das comunicações, é o tempo que um bit está em trânsito.

[2] Neste contexto, o grão é o rácio entre a computação e a quantidade de comunicação.

No escalonador podemos aumentar a performance utilizando técnicas de balanceamento de carga. Este tipo de sistemas pode gerar trabalhos dinamicamente, com cargas computacionais diferentes, o que leva a diferenças notáveis no tempo de execução. Para aumentar a eficiência dos vários processadores, os vários trabalhos têm de ser balanceados entre os processadores. Uma técnica de balanceamento muito utilizada pela sua natureza distribuída é o *work stealing*. Esta consiste numa técnica onde um processador parado ou prestes a parar "roube"trabalho a outro que tenha vários trabalhos em espera [DLS+09].

**Tolerância a falhas:** Este requisito em muitos casos não é considerado, pois penaliza a performance. Sendo que nestes sistemas, a performance tem mais importância do que a tolerância a falhas. Para implementar a tolerância a falhas, frequentemente recorre-se à replicação. Replicação é o conceito chave para a eficiência de um sistema distribuído que providencia, além da tolerância a falhas, alta disponibilidade. A replicação de dados consiste na manutenção de várias cópias em diversos computadores [CDK01].

### 4.1.1 Discussão

No trabalho realizado, focamos a nossa atenção sobre a comunicação, visto ser uma parte fulcral no desenvolvimento de sistemas. Para escolher a melhor tecnologia para este requisito, realizamos um estudo que se encontra no apêndice A. Além deste requisito, demos suporte com uma implementação rudimentar ao escalonamento, sendo que os outros requisitos não endereçámos, devido ao facto de não termos implementado um mecanismo de memória partilhada nem tolerância a falhas.

## 4.2 Mudança de paradigma

A primeira tarefa realizada na implementação foi a alteração de paradigma do *middleware* de memória partilhada para memória distribuída. Primeiro desacoplámos as aplicações do *middleware*, através da criação de uma interface de programação remota. Para as aplicações, esta camada está encapsulada numa interface independente da tecnologia (listagem 4.1).

```
public class Launcher {
  public static void init(String server) throws MalformedURLException;
  public static void init();
  public static void stop(Application app) throws RemoteException;
  public static void deploy(Application app) throws RemoteException;
}
```

Listagem 4.1: Interface de programação do *middleware*

Quando utilizado em ambientes de memória distribuída o *middleware* é um sistema

independente da JVM cliente que lança a aplicação. Assim sendo, é necessário fornecer-lhe o código da aplicação (na forma de *bytecode*). Para tal foi desenvolvido um mecanismo baseado num *class loader* remoto. Este mecanismo é transparente para os utilizadores, estando embutido na classe Launcher (apêndice B).

O objetivo deste trabalho é caminhar no sentido de um modelo único para a programação de aplicações em arquiteturas de memória partilhada e distribuída. Com esse objetivo em mente, desenhámos e implementámos o mecanismo de interação com o cliente de tal forma a que seja uniforme em ambos os contextos. Para tal, qualquer operação sobre os canais das entradas e saídas *standard* (*standard* I/O) é redirecionada para a consola do cliente. Ou seja, as escritas para a consola via System.out aparecem na consola da JVM cliente tal como se a aplicação estivesse a executar localmente. Este mecanismo é suportado pela associação de um contexto de execução a cada aplicação, sendo que os canais *standard* do *middleware* são alterados de maneira a detetar que a aplicação efetuou uma operação sobre estes.

Para que um serviço possa aceder transparentemente a outros serviços que se encontram em nós diferentes, implementámos um mecanismo de representantes, ou *stubs*. Estes são gerados em tempo de compilação, tirando partido da ferramenta *Annotation Processing Tool* (APT) [Blo04] do Java. O APT é uma ferramenta que permite, em tempo de compilação, aceder a um subconjunto da *Abstract Syntax Tree* (AST) do Java e processá-la. Esta ferramenta foi inicialmente criada para o processamento de anotações (ou metadados) e criar novas classes a partir destas. Mas como possuímos acesso a um subconjunto da AST, podemos fazer outros processamentos. No nosso caso, criámos *stubs* para todas as classes que estendem a classe ServiceProvider.

O *middleware* garante que todos os serviços têm na sua localidade uma instância ou um *stub* para todos os serviços que precisa de invocar. Para tal é necessário substituir os serviços pelo seu *stub*, o que é efetuado em tempo de execução através da API de reflexão do Java.

Internamente a camada de programação remota é implementada usando RMI, sendo que a sua interface encontra-se na listagem 4.2. Mesmo com problemas de performance, escolhemos o RMI devido à sua facilidade de utilização. Não vemos a performance como um problema devido a esta camada ser apenas usada para o lançamento das aplicações, sendo utilizada uma única vez por aplicação. Esta interface possui três operações: deploy, addClassLoader e shutdown. A operação addClassLoader regista um *class loader* remoto, indicando o identificador da aplicação. A operação shutdown termina a execução de uma aplicação no *cluster*. A operação deploy inicializa a computação da aplicação. Recebe como parâmetros o identificador da aplicação e aplicação serializada num vetor de bytes. O envio da aplicação como um vetor de bytes, em vez de um objeto, evita que o sistema de execução do RMI tente deserializar a aplicação e que lance uma exceção por ausência de classes no processo. Desta forma, o processo de desserialização e consequente carregamento das classes é controlado pelo *middleware*. Mais detalhes serão dados na secção 4.3.

Para podermos identificar unicamente as várias aplicações, os serviços, os futuros e do seus *stubs*, recorremos a identificadores únicos *Universally Unique IDentifier* (UUID) [LMS05]. Estes identificadores são compostos por 128 bits e a probabilidade de colisão é desprezável.

```
public interface IServerInterface extends Remote{
  public IRemoteFuture<Void> deploy(UUID id, byte[] app) throws RemoteException;
  public void addClassLoader(UUID clientID,IRemoteClassLoader remoteClassLoader)
      throws RemoteException;
  public void shutdown(UUID id) throws RemoteException;
}
```

Listagem 4.2: Interface remota do *middleware*

## 4.3 Lançamento de uma Aplicação

Ao longo desta secção (seguindo a estrutura da listagem 4.3) iremos explicar o processo de lançamento de uma aplicação num *cluster*. Após receber uma aplicação, esta é deserializada e o seu lançamento no *middleware* é da responsabilidade do método deploy.

**Detenção das Afinidades** Entre as linhas 4 e 6 realiza-se a deteção de afinidades implícitas entre serviços. Estas afinidades são essenciais para as várias políticas de escalonamento, pois as políticas de escalonamento do *middleware* são baseadas em afinidades. No escalonamento existem dois tipos de afinidades: i) explícito; ii) implícito. A afinidade explicita é definida pelo utilizador usando a primitiva setAffinity da classe ServiceProvider; a afinidade implícita é estabelecida no sistema a partir das referências entre serviços inferiras a partir do código dos mesmos. A figura 4.1 exemplifica um possível esquema de afinidades numa aplicação, onde as afinidades a tracejado são implícitas e as restantes são explícitas.

Para detetar as referências dentro de um serviço, usamos o Java Reflection API. Ou seja, examinamos todos os serviços para determinar se possuem referências para outros serviços. A listagem 4.4 mostra o código de deteção de afinidades implícitas.

**Escalonar a aplicação** A linha 9 escalona a aplicação pelos nós dos *cluster*. O processo de escalonamento no *middleware* está dividido em duas partes: i) classificação e seleção dos nós; ii) escalonamento dos serviços pelos nós escolhidos.

A primeira fase de escalonamento de uma aplicação no *cluster* determina em que nós os serviços irão executar, de maneira a manter o balanceamento da carga. Para tal, o *driver* que implementa esta funcionalidade recorre ao módulo de estatística para obter o estado mais recente do sistema. A sua implementação deve respeitar a interface SchedulingRankDriver (listagem 4.5), que especifica um só método, sort, que, dado uma lista de todos os nós disponíveis no sistema, irá retornar a lista ordenada, sendo que esta lista

```
1  private static SchedulingModule sch;
2
3  public static IFuture<Void> deploy(Application app) {
4    for (Service p : app.getPlaces()) {
5      p.detectAffinity();
6    }
7
8    List<IFuture<Void>> mains=new ArrayList<IFuture<Void>>();
9    SchedulingInfo info=sch.schedule(app);
10
11   if(info==null){
12     Middleware.registerErrConsole(app.getAppID(), app.getErrConsole());
13     Middleware.registerInConsole(app.getAppID(), app.getInConsole());
14     Middleware.registerOutConsole(app.getAppID(), app.getOutConsole());
15     mains.add(Middleware.startApp(app));
16   }else{
17     info.finish();
18     for (Place n : info.getPlaces()) {
19       Application a = new Application(app.getAppID());
20       a.setSchedulingInfo(info);
21       for (ServiceProvider p : info.getSchedule(n)) {
22         a.addService(p);
23       }
24       for (ServiceStub p : info.getStubSchedule(n)) {
25         a.addService(p);
26       }
27
28       if(!n.isLocal()){
29         a.setErrConsole(app.getErrConsole());
30         a.setInConsole(app.getInConsole());
31         a.setOutConsole(app.getOutConsole());
32         a.setRemoteClassLoader(app.getRemoteClassLoader());
33         mains.add(comm.sendApplication(a, n));
34       }else{
35         Middleware.registerErrConsole(app.getAppID(), app.getErrConsole());
36         Middleware.registerInConsole(app.getAppID(), app.getInConsole());
37         Middleware.registerOutConsole(app.getAppID(), app.getOutConsole());
38         mains.add(Middleware.startApp(a));
39         List<Message<Object[]>> list=tasksWainting.remove(a.getAppID());
40         if(list!=null){
41           for (Message<Object[]> message : list) {
42             invokeService(message, message.getSending_host(), a);
43           }
44         }
45       }
46     }
47   }
48   return new MultipleFutures<Void>(mains);
49 }
```

Listagem 4.3: Lançamento de uma aplicação

denota os nós selecionados. Foi desenvolvida apenas uma implementação deste driver, o RandomRanking, que retorna a lista dada ordenada de forma aleatória.

A segunda fase do escalonamento distribui os serviços pelos *places*. Como já foi descrito, o escalonamento é baseado no conceito de afinidade. As políticas de escalonamento

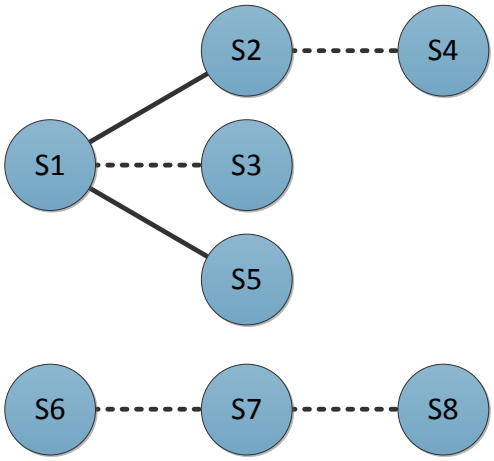



Figura 4.1: Exemplo de afinidades entre Serviços

```
public void detectAffinity() {
  Class<? extends ServiceProvider> c = this.getClass();
  for (Field f : c.getDeclaredFields()) {
    try {
      f.setAccessible(true);
      Object o = f.get(this);
      if (o instanceof ServiceProvider) {
        ServiceProvider p = (ServiceProvider) o;
        if (!this.exAffinities.contains(p) && !this.imAffinities.contains(p)) {
          this.setImpAffinity(p);
        }
      }
    } catch (IllegalArgumentException e) {
      // Nunca é lançada esta exceção
    } catch (IllegalAccessException e) {
      // Nunca é lançada esta exceção
    }
  }
}
```

Listagem 4.4: Deteção das afinidades implícitas

```
public interface SchedulingRankDriver {
  List<Place> sort(List<Place> places);
}
```

Listagem 4.5: Interface SchedulingRankDriver

no *middleware* têm de ser implementadas via *driver*, concretizando a interface SchedulingDriver (listagem 4.6).

```
public interface SchedulingDriver {
  SchedulingInfo schedule(Application app, List<Node> sort);
}
```

Listagem 4.6: Interface SchedulingDriver

O driver de escalonamento recebe como parâmetros uma aplicação e uma lista de nós ordenados, obtida através do driver de *ranking*, e retorna o escalonamento, através da classe SchedulingInfo. Esta classe contém a associação entre os serviços e os *places*. Para o núcleo do *middleware*, esta classe disponibiliza os seguintes métodos: i) finish; ii) getNodes; iii) getSchedule e iv) getStubSchedule. A primitiva finish gera os *stubs* para cada *place*. O método getNodes retorna uma lista de *places* que irá receber a aplicação, a qual poderá ser diferente da lista retornada pelo *driver* de *ranking*. Os métodos getSchedule e getStubSchedule retornam uma lista de serviços e *stubs* respetivamente para um determinado nó.

Para este protótipo inicial foi implementado apenas um *driver* de escalonamento baseado na política de *round-robin* [Ras70], sendo que apenas suporta afinidade explicita. Este suporte de afinidade garante que serviços afins sejam escalonados para o mesmo *place*.

**Envio do serviços para os nós** Entre as linhas 11 e 47 da listagem 4.3 encontra-se o envio dos serviços para os *places*. Este envio está dividido em duas partes: modo *cluster* e modo *multi-core*. Em modo *multi-core*, toda a aplicação irá correr na instância local do *middleware* (linhas 12 a 15). Mesmo a correr localmente, é necessário registar os *stubs* das consolas para manter a compatibilidade de versões. No modo *cluster*, a execução do método recolhe a informação de escalonamento contida em info (instância da classe ScheduleInfo). Para cada nó, é criada um nova instância da classe Application que conterá os serviços e *stubs* que lhe foram destinados (linhas 19 a 26). Além destes, são adicionados os *stubs* para as consolas *standard* e para o *class loader* do cliente. No caso particular dos serviços a escalonar no nó corrente (linhas 35 a 44), são registadas as consolas, o *class loader* e iniciar a aplicação localmente (linha 38).

O processo distribuído de inicialização de uma aplicação tem de garantir que a partir de um dado instante todos os serviços estão disponíveis para atender pedidos de invocação do seu interface. Tal pode ser facilmente resolvido através de um ponto de sincronização global. Para evitar o custo associado a tal sincronização, permite-se o inicio desfasado da aplicação o que pode dar origem a cenários em que um serviço invoca um método noutro que ainda não está em execução. Nesse caso (listagem 4.7, linhas 11 a 17), o pedido é colocado num mapa (tasksWainting). A receção e consequente execução de uma aplicação num dado nó verifica se já existem pedidos de invocação pendentes. Se existirem tarefas, essas foram geradas por serviços remotos que contêm *stubs* para os serviços locais.

Caso uma aplicação não se destine ao nó corrente, esta é enviada para o nó destino através do *driver* de comunicação (linha 33). Este envio tem duas etapas: primeiro é enviado o *stub* do *class loader*, para garantir que os nós conseguiram obter o código de todas as classes, e depois é enviado a aplicação serializada num vetor de bytes. Os *stub* são compostos apenas por uma String que contém o endereço de rede do cliente e um inteiro, que contém a porta do *class loader*.

```
1  private static Map<UUID, Application> applications =
2                                                  new HashMap<UUID, Application>();
3  //Método que processa mensagens de rede
4  public static void processMessage(Message msg) {
5    switch (msg.getType()) {
6      [...]
7      case ADDTASK:{
8        Object[] aux = (Object[])msg.getContent();
9        Application app = applications.get((UUID)aux[0]);
10       if(app==null){
11         List<Message<Object[]>> list=tasksWainting.get((UUID)aux[0]);
12         if(list==null)
13           list=new ArrayList<Message<Object[]>>();
14
15         list.add((Message<Object[]>)msg);
16
17         tasksWainting.put((UUID)aux[0], list);
18       }else{
19         invokePlace((Message<Object[]>)msg, msg.getSending_host(), app);
20       }
21     }
22     [...]
23   }
24 }
```

Listagem 4.7: Receção de uma tarefa remota

## 4.4 Comunicação

A comunicação entre *places* és suportada pelo *driver* de comunicação, cuja interface especifica três métodos (listagem 4.8): init, send e registerEvent. O método init é invocado aquando da inicialização do driver e é responsável por efetuar todas as inicializações necessárias ao *driver*, que normalmente envolve em abrir um porto. O método send serializa e envia uma mensagem pela rede.

```
public interface CommunicationDriver {
  void init(Config conf);
  <C> void send(Message<C> message);
  void registerEvent(UUID oid, IComEvent event);
}
```

Listagem 4.8: Interface do *driver de comunicação*

O *driver* implementa uma lógica orientada a eventos para a receção de mensagens. As mensagens podem destinar-se ao *place* em geral ou a um objeto em particular. Para que um objeto possa receber mensagens tem de implementar a interface IComEvent (listagem 4.9) e registar-se diante do driver através do método registerEvent, fornecendo um identificador único.

Uma mensagem (listagem 4.10) contém toda a informação que precisa para o seu envio, ou seja: as localidades origem e destino, o objeto destino e origem, o tipo de mensagem, o conteúdo e as estatísticas atuais do nó que a envia. A atribuição do valor null

```
public interface IComEvent {
  public <C> void processMessage(Message<C> message);
}
```

Listagem 4.9: Interface IComEvent

aos campos relativos aos objetos origem e destino indica que a mensagem é destinada à localidade.

```
public class Message<C> implements Externalizable{
  private UUID destination_object;
  private Node destination_host;
  private UUID sending_object;
  private Node sending_host;
  private MessageTag tag;
  private C content;

  private Statistics sta;
}
```

Listagem 4.10: Mensagem

Como já foi mencionado na secção 4.2, para cada serviço contido na aplicação é gerado um *stub* respetivo em tempo de compilação. A sua construção recorre a templates para a geração do código geral da classe (listagem 4.12) e para os seus métodos (listagem 4.13). Estes templates são genéricos, sendo também utilizados no contexto de algumas agregações de serviços (secção 4.6). Um exemplo de um *stub* gerado encontra-se no apêndice C. A Tabela 4.1 apresenta uma breve descrição das várias etiquetas que podem aparecer nos templates.

```
public class ServiceStub implements Service,Serializable{
  protected Node location;
  protected UUID clientid;
  protected UUID id;

  [...]

  public <R> IFuture<R> invoke(String methodName, Object... args) throws
      NoSuchMethodException {
    return new RemoteFuture<R>(new Handler<R>(this, methodName, args), this.
        location, this.clientid).init();
  }
}
```

Listagem 4.11: Classe ServiceStub

Para facilitar a construção dos *stubs* de forma a que os pedidos de invocação sejam encaminhados para o serviço alvo, fatorizámos as funcionalidades comuns num *stub* base que contem uma implementação do método invoke (da interface do serviço - listagem 4.11) que encaminha as invocação para o serviço alvo.

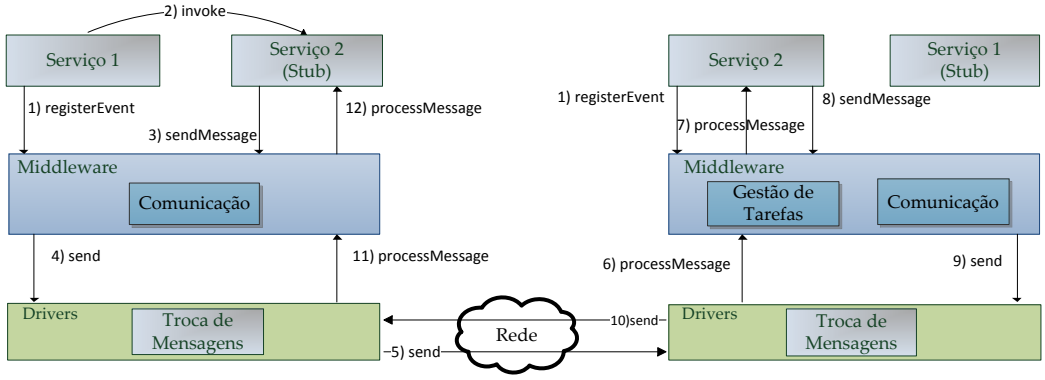



Figura 4.2: Protocolo de comunicação para a execução de tarefas remotas

A implementação do método invoke faz uso da classe RemoteFuture, uma implementação do interface Future que permite representar o resultado de uma computação assíncrona a executar noutro *place*. Este tem a responsabilidade, conjuntamente com o *driver* de comunicação, de enviar uma tarefa remota e de receber o seu resultado. Na figura 4.2 encontra-se o protocolo de comunicação do RemoteFuture. Antes de poder receber alguma mensagem remota, uma instância da classe RemoteFuture tem de se registar no *driver* de comunicação.

```
<package>

/*GENERATED CLASS - DO NOT EDIT*/

import instrumentation.definitions.Generated;
<imports>

@Generated
public class <class_name> <type_parameters> extends <super_class> <interfaces>{

  <constructor>

  <methods>
}
```

Listagem 4.12: Template da classe

```
<modifiers> <type_parameters> <return_type> <name> (<params>) <throws> {
  try{
    IFuture<<Return_type>> aux = this.invoke("<name>",new Object[]{<params_vars>});
    <return>
  }catch (NoSuchMethodException e) {
    <return_default>
  }
}
```

Listagem 4.13: Template dos métodos

| Campo | Descrição |
|---|---|
| <package> | Pacote da classe. |
| <imports> | *Imports* necessários para a classe. |
| <class_name> | Nome de classe. |
| <type_parameters> | Tipos paramétricos da classe ou do método. |
| <super_class> | Nome da super classe. |
| <interfaces> | Interfaces da classe (se existirem). |
| <constructor> | Construtor da classe. |
| <methods> | Métodos da classe. |
| <modifiers> | Modificadores do método. |
| <return_type> | Tipo do retorno do método, este inclui tipos primitivos. |
| <Return_type> | Tipo do retorno do método, não inclui tipos primitivos. Se este for do tipo primitivo é traduzido para o seu objeto equivalente. |
| <name> | Nome do método. |
| <params> | Parâmetros do método com as declarações de tipo. |
| <params_vars> | Parâmetros do método sem as declarações do tipo. |
| <throws> | Exceções que este método pode lançar. |
| <return> | Retorno do método. Se o retorno for void, então o resultado deste campo será aux.get(), caso contrário o resultado será return aux.get(). |
| <return_default> | Retorno por defeito do método. Se o retorno for um numérico, então retorna 0, caso contrário, retorna null. |

Tabela 4.1: Descrição dos campos do template

Após um estudo de várias tecnologias (apêndice A), optámos por implementar este driver recorrendo ao Java *New Input/Ouput* (NIO). Esta biblioteca possibilita escutar vários canais através do uso de *selectors* usando um só fio de execução, ao contrário da biblioteca de *sockets* do Java, serializando o atendimento dos pedidos. Este problema poderá ser atacado no futuro com uma *pool* de *threads* dedicada à comunicação.

Ao receber os dados, que serão sempre do tipo Message, estes vêm como um vetor de bytes, que são acedidos através de um buffer do tipo ByteBuffer. Após receber os dados todos, estes terão de ser deserializados, para que possamos reconstruir a instância de Message.

## 4.5 Sincronização

No que se refere aos mecanismos de sincronização acrescentámos o mecanismo de barreiras. Para que estas barreiras sejam o mais flexíveis possível, o número de fios de execução a sincronizar não é pré-determinado, mas sim definido dinamicamente através do método register. Além desta primitiva (listagem 4.14), a interface inclui ainda a primitiva await que serve para sincronizar os fios de execução.

Como já foi indicado na subsecção 3.2.1, estas barreiras são hierarquias. Esta decisão de desenho deve-se ao facto de que estas puderam ser combinadas com a agregação

```
public interface IBarrier {
  public void await();
  public void register();
}
```

Listagem 4.14: Interface das barreiras

de serviços. Esta é uma barreira que trabalha ao nível do *cluster* e que agrega todas as barreiras dos nós. Com este mecanismo poupamos as comunicações remotas.

## 4.6 Agregação de Serviços

Esta secção aborda a implementação dos vários tipos de agregação de serviços suportados para memória distribuida. Dos quatro comportamentos apresentados, o único que não recorre a serviços gerados em tempo de compilação é o *Façade*. Dos restantes, os que recorrem a anotações para indicar o seu comportamento são o Memória particionada e o *Distribute-Map-Reduce*.

### 4.6.1 *Distribute-Map-Reduce*

A implementação deste comportamento é a mais complexa porque requer uma inspeção mais profunda e detalhada do código da aplicação, nomeadamente os imports, as invocações a métodos e tipos de dados usados. Para tal foram utilizadas as bibliotecas APT e o *Compiler Tree API* (CTA). O CTA permite aceder à AST do código Java em modo de leitura e disponibiliza ferramentas para processá-la. A geração do código para este comportamento divide-se em 2 partes.

**Leitura de todas as políticas de distribuição e redução e verificação dos seus tipos paramétricos** Antes de podermos fazer qualquer análise aos serviços que usam este comportamento, temos de verificar todas as políticas de distribuição e redução, para descobrir com que tipos concretos foram instanciadas as classes com tipo genéricos. Este problema agrava-se devido à hierarquia de classes, sendo que esta análise é feita tendo em conta esta hierarquia. A análise é necessária para realizar a verificação estática de tipos entre a distribuição, o método e a redução.

**Geração do código** Para todos os métodos que usam este comportamento, primeiro verifica-se se existe uma política de distribuição e redução. Se uma destas não existir é dado um erro. Com esta informação, e sabendo que tipos de dados é que as políticas tratam, podemos gerar o código da agregação. Esta geração é feita em dois passos: i) geração da tarefa que irá ser executada nos nós e ii) geração da agregação. Gerar a tarefa consiste em retirar o código do método e encapsulá-lo no método call de uma classe que implemente a interface ITask. A listagem 4.15 apresenta o exemplo do método getPI da classe MathProvider que implementa o serviço Math.

```
1  @DistRed
2  public class MathProvider extends ServiceProvider implements Math{
3    [...]
4    @ReductionPolicy(reduction=ReductionPI.class,params={"rounds"})
5    public double getPI(@DistributionPolicy(distribution=DistributionPI.class,params
         ={"rounds"}) double rounds) {
6      Distribution<Double> distr = new DistributionPI(rounds);
7      Reduction<Double, Double> red = new InnerReductionPI();
8      return distReduce(distr, red, new PITask()).get();
9    }
10 }
```

Listagem 4.15: Classe MathProvider

A partir deste código, o *middleware* gera a tarefa da listagem 4.16 que contém o código do método e o serviço agregador da listagem 4.17 que usa o mecanismo de DMR com as políticas de distribuição e redução indicadas nas anotações.

```
1  @Generated
2  public class MathProvider_getPITask extends DistrRedTask<java.lang.Double, java.
       lang.Double> {
3
4    private MathProvider parent;
5
6    public java.lang.Double call() {
7      this.parent = (MathProvider)this.service;
8      double rounds = popArgument();
9      Distribution<Double> distr = new DistributionPI(rounds);
10     Reduction<Double, Double> red = new InnerReductionPI();
11     return this.parent.distReduce(distr, red, new PITask()).get();
12   }
13
14 }
```

Listagem 4.16: Tarefa de exemplo gerada com o *middleware*

A tarefa gerada contém o código do método anotado. Neste exemplo, as linhas 6 a 8 da listagem 4.15 são iguais às linhas 9 a 11 da listagem 4.16. A única diferença incide no objeto alvo da invocação do método distReduce. A operação tem de ser invocada no serviço original e portanto o código gerado tem de deter uma referência para esse mesmo serviço (no campo parent). Para resolver este problema, guardamos uma referencia para o serviço invocador e usamos o método dessa referência. Como estamos a distribuir o parâmetro rounds, então temos de receber a partição deste utilizado uma pilha de argumentos que evita a criação de várias instâncias da tarefa (para mais detalhes consulte [Mou11]). Para isso é usado a primitiva popArgument (linha 8 listagem 4.16). As linhas 7 e 8 da listagem 4.17 são geradas a partir das das anotações da listagem 4.15.

### 4.6.2 *Pool* de Serviços

A implementação do *Pool* de Serviços é semelhante à dos *stubs*, na medida em que utiliza os mesmos templates (Listagens 4.12 e 4.13). As diferenças no código gerado centram-se

```
1  @Generated
2  public class MathProviderDistRed extends place.aggregator.DistRedPlace implements
       services.MathService.Math, java.io.Serializable{
3
4    [...]
5
6    public double getPI (double rounds) {
7      core.collective.Distribution<java.lang.Double> dist = new services.MathService.
           pi.DistributionPI(rounds);
8      core.collective.Reduction<java.lang.Double,java.lang.Double> red = new services
           .MathService.pi.ReductionPI(rounds);
9
10     return distReduce(dist, red,new MathProvider_getPITask()).get();
11   }
12 }
```

Listagem 4.17: Agregador exemplo gerado por o *middleware*

no construtor e no método invoke. O construtor do ServicePool recebe dois parâmetros, o vetor de serviços a agregar e uma política de escalonamento. O método invoke recorre ao escalonador para determinar qual o serviço alvo da invocação. O apêndice D inclui um exemplo com o código gerado para este comportamento.

```
public abstract class ServicePool extends ServiceProvider implements Serializable{

  private Service[] services;
  private IServiceScheduler scheduler;

  public ServicePool(Service[] services,IServiceScheduler scheduler){
    super();
    this.level=Level.Cluster;
    this.services=services;
    this.scheduler=scheduler;
    this.scheduler.setSize(services.length);
  }
  public <R> IFuture<R> invoke(String methodName, Object... args) throws
      NoSuchMethodException{
    int i=scheduler.getIndex();
    return services[i].invoke(methodName, args);
  }
  [...]
}
```

Listagem 4.18: Classe ServicePool

### 4.6.3 Memória particionada

A implementação deste comportamento é semelhante ao do *Pool* de Serviços na medida em que utiliza templates para a geração da agregação, com a diferença do template do método. Esta agregação para os métodos utiliza o template da listagem 4.19.

Como se pode constatar no template, a especificação do método invoke inclui um parâmetro extra para indicar o valor da chave a utilizar (listagem 4.20). No apêndice

```
<modifiers> <type_parameters> <return_type> <name> (<params>) <throws> {
  try{
    IFuture<<Return_type>> aux = this.invoke(<key>,"<name>",new Object[]{<
        params_vars>});
    <return>
  }catch (NoSuchMethodException e) {
    <return_default>
  }
}
```

Listagem 4.19: Serviço Banco

E encontra-se o código gerado para este comportamento para o serviço Banco (listagem 3.18).

```
public class PartitionedMemService<K,V> extends ServiceProvider implements
    Serializable{

  private IPartitioner<K,V> partitioner;

  public PartitionedMemPlace(AbstractPartitionedMemService<K, V>[] services,
      IPartitioner<K,V> partitioner){
    super();
    this.level=Level.Cluster;
    this.partitioner=partitioner;
    this.partitioner.setServices(services);
    for (AbstractPartitionedMemPlace<K, V> p : services) {
      p.setPartitioner(partitioner);
    }
  }

  public <R> IFuture<R> invoke(K key,String methodName, Object... args) throws
      NoSuchMethodException{
    AbstractPartitionedMemService<K, V> service=this.partitioner.getService(key);
    return service.invoke(methodName, args);
  }
}
```

Listagem 4.20: Classe PartitionedMemService

### 4.6.4 Façade

Este comportamento é o mais simples, pois não depende de código gerado. Este usa apenas o mecanismo de tipos genéricos do Java e a API de reflexão. O façade apenas agrega dois serviços, sendo que pode agregar outros facades, criando uma árvore de serviços agregados. Este disponibiliza uma primitiva get. Esta retorna a instância de um tipo passado por parâmetro que esteja contida neste ou em outros façades que este possa conter. Portanto esta primitiva tem de contemplar os seguintes casos:

1. O serviço está em T1, sendo que T1 é uma folha;

2. O serviço está em T2, sendo que T2 é uma folha;

3. O serviço está na sub-árvore T1;

4. O serviço está na sub-árvore T2;

```
public class Facade<T1, T2> extends ServiceProvider implements Service {

  private T1 service1;
  private T2 service2;

  [...]

  public <T> T get(Class<T> Tclass) throws FacadeTypeNotFound {
    if (service1.getClass().equals(Tclass)||contains(service1.getClass().
        getInterfaces(), Tclass))
        return (T)service1;
    else {
      if (service2.getClass().equals(Tclass)||contains(service2.getClass().
          getInterfaces(), Tclass))
          return (T)service2;
      else
        if (service1 instanceof Facade)
          try {
            return (T)((Facade)service1).get(Tclass);
          }
          catch (FacadeTypeNotFound e) {
            if (service2 instanceof Facade)
              return (T)((Facade)service2).get(Tclass);
          }
        else {
          if (service2 instanceof Facade)
            return (T)((Facade)service2).get(Tclass);
        }
    }
    throw new FacadeTypeNotFound();
  }

  private boolean contains(Class<?>[] interfaces,Class<?> i){
    for (Class<?> class1 : interfaces) {
      if(class1.equals(i))
        return true;
    }
    return false;
  }

}
```

Listagem 4.21: Classe Facade

## 4.7 Hierarquia de drivers

Na inicialização do *middleware*, todos os *drivers* são carregados e inicializados. Para os *drivers* que podem ser definidos em vários níveis, o que é o caso do *driver* de Distribuição e Redução, estes são guardados num mapa onde a chave é o nível que está associado o *driver*.

Como já foi mencionado, para todos os serviços está associado um nível de execução. Usando este nível, podemos entregar aos serviços os *drivers* definidos para o seu nível de execução.

```
public class DistReduce<T,P,R>{
  [...]

  public IFuture<R> getResult(Distribution<T> distr, Reduction<P,R> red,
      DistrRedTask<T,P> task, Service s)
  {

    DistRedDriver<T,P,R> distRed=Drivers.getDistRedDriver(s.getLevel());

    return distRed.getResult(distr, red, task, place);
  }
}
```

Listagem 4.22: Módulo de Distribuição/Redução

Como podemos ver na listagem 4.22, quando um serviço invoca o mecanismo de DMR, o método getResult do módulo apresentado na listagem é invocado. Neste método, é pedido o *driver* de distribuição/redução associado ao nível em que o serviço se encontra a executar.

Com este mecanismo podemos adaptar a camada de *drivers* às características do ambiente de execução, nomeadamente ao hardware e ao software disponível. Neste momento, apenas suportamos dois níveis, nó e *cluster*, mas antecipamos mais níveis, como por exemplo, o nível inter-*cluster*.

## 4.8 Conclusões

Nesta dissertação foi implementado um *middleware* adaptável à plataforma, que oferece uma interface genérica centrada no conceito de serviço. Este fornece vários modelos de programação, desde paralelismo de dados e de tarefas ao nível do método. Implementámos um mecanismo de expressão de paralelismo através de anotações Java. Disponibilizamos mecanismos para a agregação de serviços com 4 comportamentos predefinidos: *Distribute-Map-Reduce*, *Pool*, Memória particionada e Façade.

Concretizamos a camada de adaptabilidade, desenvolvendo *drivers* para suportar as funcionalidades específicas para ambientes de memória distribuída, como por exemplo, comunicação, escalonamento e distribuição/redução para estes ambientes.

Com este prototipo vamos realizar uma avaliação inicial ao mecanismo de DMR para arquiteturas de memória partilhada e distribuída, analisando a utilidade deste modelo e a performance da nossa implementação.

# 5

# Avaliação

Este capítulo apresenta duas avaliações de desempenho relativamente à utilização da implementação atual do *middleware* como sistema de execução de aplicações paralelas, sendo que estas focam-se no mecanismo de DMR. A primeira avaliação insere-se num desafio de programação paralela, SICSA MultiCore Challenge[1], onde era pedido a implementação de dois problemas, a Concordância e o N-Nody. A segunda avaliação tem como objetivo a avaliação mais pormenorizada do mecanismo DMR em ambientes de memória distribuída, para isso usamos alguns *benchmarks* do *Java Grande Benchmark Suite*.

## 5.1 SICSA MultiCore Challenge

O SICSA MultiCore Challenge é um desafio de programação paralela organizada pela *Scottish Informatics and Computer Science Alliance* (SICSA). O objetivo deste desafio é comparar várias abordagens para a programação paralela em termos de desempenho e de facilidade de programação. Este desafio encontra-se ativo deste 2010 e conta com duas fases. Na primeira fase, que decorreu em 2010, pedia-se aos participantes o desenvolvimento de um gerador de concordância num ficheiro de texto e na segunda fase, que decorreu em 2011, pedia-se um simulador planetário usando o problema N-body. No inicio de 2012, lançam uma fase especial, fase esta que será publicada numa edição especial da revista *Concurrency and Computation: Practice and Experience*, em que se pede aos participantes que apresentem soluções para um dois dos problemas das fases anteriores. Para este desafio, adaptámos implementações Java *multi-threaded* já existentes dos problemas para o *middleware*.

[1] `http://www.macs.hw.ac.uk/sicsawiki/index.php/MultiCoreChallenge`

A nossa participação deu-se apenas na fase especial. Para tal adaptámos para o *middleware* implementações de Jeremy Singer da Concordância[2] e a adaptação de Nicholas Chen da implementação Lonestar do algoritmo Barnes-Hut[3].

### 5.1.1 Os problemas

O problema da Concordância consiste em encontrar num ficheiro todas as sequências de palavras com tamanho até a um dado $N$ num ficheiro. Por exemplo, o resultado deste exemplo usando um $N = 2$ do texto

*As armas e os Barões assinalados*

produz a seguinte lista:

```
As:0          Barões assinalados:4   assinalados:5     os:3
As armas:0    armas:1                e:2               os Barões:3
Barões:4      armas e:1              e os:2
```

A implementação deste problema que realizámos usa o mecanismo de DMR. Para o problema da Concordância, o *input* que era pedido para a realização do estudo da performance era sempre com $N = 4$ e com os seguintes ficheiros:

| Ficheiro | Número de palavras | Número de linhas | Tamanho em MBytes |
|---|---|---|---|
| `WaD.txt` | 268429 | 27743 | 1,42 |
| Bíblia (versão inglesa) | 801541 | 79443 | 4,57 |

Tabela 5.1: Ficheiros de *input* da Concordância

O problema do N-Body é uma simulação de um sistema dinâmico de partículas sobre a influência de forças físicas. Um caso especial deste problema é a previsão do movimento de um grupo de objetos celestes que interagem gravitacionalmente uns com os outros. Para resolver este problema, é dado como entrada um ficheiro contendo as massas, as posições e velocidades dos corpos num espaço tridimensional. Para este problema, o ficheiro de entrada fornecido continha 1024 corpos e era pedido que realizássemos 20 iterações, cada uma com intervalo de tempo de 0.001. Neste problema realizámos 3 versões: i) baseado em tarefas, ii) baseado em DMR com a distribuição de limites e iii) baseado em DMR com a distribuição de partições.

### 5.1.2 Implementação dos problemas

Nesta subsecção iremos apresentar os algoritmos dos dois problemas e as suas implementações

[2] `http://code.google.com/p/sicsaconcordance/`
[3] `https://wiki.engr.illinois.edu/display/transformation/Barnes-Hut+Java`

#### 5.1.2.1 Concordância

A implementação de referência do problema da Concordância segue a seguinte abordagem:

```
1: linhas ← ler_ficheiro()
2: regioes ← particionar_linhas(linhas)
3: atribuir_regioes_a_tabalhadores(trabalhadores, regioes)
4: para todos trabalhadores fazer
5:     trabalhador.pesquisarFrases(tabela_dispersao)
6: fim para
7: imprimirResultado(tabela_dispersao)
```

**Versão *multi-core*** A implementação do *middleware* foca-se nos passos 2,3 e 4, adaptando-os para o paradigma DMR. Para isso, especificámos a interface ConcordService que contem um único método: createConcordance (listagem G.1). A sua implementação delimita o trabalho atribuído aos *threads* na implementação de referência (listagem G.2).

Os passos 2, 3 e 4 são automaticamente tratados por o módulo de DMR, desde que o método seja anotado com as políticas de distribuição e redução. Neste caso particular, nenhuma redução é realizada, pois o método trabalha diretamente na tabela de dispersão (um HashMap) que irá guardar os resultados finais. A política de distribuição apresentada na listagem G.3 gera um vetor de zonas distintas do ficheiro de entrada (regiões), que serão processadas por várias execuções do método createConcordance.

**Versão de *cluster*** Para transpor esta aplicação para arquiteturas de *cluster*, a solução mais natural é agregar implementações de ConcordService usando o paradigma DMR. No entanto, os serviços agregados não podem ser instâncias de ConcordProvider (listagem G.2), uma vez que este recorre a memória partilhada, partilha o ficheiro de entrada e a tabela de dispersão dos resultados. Então temos que:

- distribuir o texto entre os serviços que componham a agregação em vez de distribuir os índices sobre dados partilhados - listagem G.5, e;
- reduzir as *hash tables* parciais produzidas por cada instância do serviço - listagem G.6.

A redução envolve fundir uma coleção de *hash tables*, que não é nativamente suportada para a API standard do Java. Este problema é agravado pelo facto de o conjunto das chaves das tabelas não ser disjunto. A ocorrência da mesma sequência identificada por serviços diferentes tem de ser junta.

Como será discutido na secção 5.3 esta redução limita altamente a execução paralela da aplicação, já que estrangula toda a computação.

#### 5.1.2.2 N-Body

Para implementar este problema, selecionamos o algoritmo Barnes-Hut [BH86], que compreende aos seguintes passos:

1: $dados \leftarrow ler\_dados()$
2: $corpos \leftarrow particionar\_corpos(dados)$
3: $atribuir\_corpos\_a\_tabalhadores(trabalhadores, corpos)$
4: **para todos** $trabalhadores$ **fazer**
5: $\quad trabalhador.calcular\_aceleracao\_e\_velocidade\_dos\_corpos()$
6: $\quad trabalhador.calcular\_nova\_posicao\_dos\_corpos()$
7: **fim para**
8: $imprimirResultado()$

A computação de cada iteração dos trabalhadores requer a perceção da criação de uma *oct-tree*. A implementação de referência delega esta tarefa ao fio de execução principal. Assim, existem três pontos de comunicação/sincronização: 1. todos os trabalhadores têm de esperar que a *oct-tree* seja gerada por o fio de execução principal; 2. todos os trabalhadores têm de completar o cálculo da aceleração antes de avançar para o cálculo da posição, e; 3. o fio de execução principal tem de ser notificado quando todos os trabalhadores terminarem a sua iteração, para que este possa gerar uma nova *oct-tree*.

**Versões *multi-core*** Para demonstrar a flexibilidade do *middleware* implementámos três versões diferentes do algoritmo. Apresentamos estas versões de menor para maior grau de abstração.

Todas as versões implementam a interface NBodyService descrita na listagem F.1. Mais uma vez, apenas define um ponto de entrada para a execução da simulação.

**Versão baseada em tarefas:** A primeira versão (listagem F.2) é puramente baseada em tarefas. O método runSimulation() particiona uma coleção de corpos (passo 2) na linha 23 invocando o método getLowerandUpperBounds() e executa o algoritmo à partição recebida (passo 4) na linha 24 recorrendo à tarefa privada computePartition(). A atribuição das partições é tratada pelo *middleware* com a invocação de computePartition() na linha 24.

Para sincronizar a execução das tarefas, recorremos a duas barreiras e a uma condição. Estas são criadas nas linhas 8 a 15, e são usadas no método computePartition(), nas linhas 41 a 62.

**Versão DMR com partição de limites:** A segunda versão usa o paradigma de DMR para executar os passos 2 a 4. Para esta versão, implementamos uma nova classe para o serviço (listagem F.3) e uma política de distribuição (listagem F.4), separando a distribuição da lógica do algoritmo. O política implementa o mesmo particionamento que o método getLowerAndUpperBoundsFor() da versão anterior.

O método computePartition() trabalha diretamente na variável body, partilhada entre todas as instâncias da classe e o cliente do serviço. Assim, não há necessidade de uma política de redução.

**Versão DMR baseado em partições:** Finalmente, a terceira versão distribui o conjunto de corpos, em vez de limites de um vetor partilhado. Nesta versão também aplicámos a estratégia de DMR mas com uma política de distribuição diferente da versão anterior (listagem F.6). Com isto, abstraímos a execução paralela do algoritmo, já que a tarefa não tem conhecimento que está a trabalhar num subconjunto do vetor dos corpos. Este facto é percetível nas linhas 22 e 28 na implementação desta versão (listagem F.5) quando comparado com as mesmas linhas da versão anterior (listagem F.3). Se negligenciarmos as primitivas de sincronização, na verdade estamos a aplicar a versão sequencial do algoritmo.

**Versão de *cluster*** Nesta versão recorremos outra vez à agregação de serviços com o paradigma DMR. Note que os passos 5 e 6 são precedidos e seguidos por pontos de comunicação e sincronização. Nas versões *multi-core*, a comunicação era implícita na variável partilhada body. Nesta versão, o vetor dos corpos tem de ser dividido entre as instâncias dos serviços agregados e reunido ao fim de cada um destes passos. Para isso, definimos uma nova interface de serviço, ComputePartitionService (listagem F.8), que especifica os métodos computeForce() e advance() (as computações dos passos 5 e 6). As listagens F.7 e F.9 ilustram, respetivamente, as implementações de NBodyService e ComputePartitionService, sendo que o primeiro é cliente do último. Esta implementação introduz um padrão de *fork-join* em dois passos com pontos comunicação e sincronização entre cada iteração.

Como esta implementação usa instâncias de serviços distribuídos, temos de reduzir os resultados gerados localmente. A redução, apresentada na listagem F.10, recebe um iterado de vetores de corpos e funde-os. A política de distribuição é igual a que é aplicada na terceira versão de *multi-core* (listagem F.6).

### 5.1.3 Ambiente experimental

Para o desafio, todas as medições foram realizadas num *cluster Beowulf* com 32 nós, localizado na Universidade de Heriot-Watt. Cada nó contem as seguintes características:

**Processador** Intel Xeon E5504 a 2.00GHz com quatro cores com Hyper-Threading;

**Memória RAM** 12 GBytes de capacidade;

**Sistema Operativo** Fedora com a versão do Kernel 2.6.18-274;

**JDK** Java SE Development Kit 7, 64 bits.

### 5.1.4 Resultados

Neste estudo, apenas analisámos os resultados das versões *multi-core*. Para a análise de performance destes problemas, medimos a execução de cada versão mais que 30 vezes, descartando $20\%$ dos melhores e piores resultados, calculando a média dos resultados restantes. Todas as medições se focam unicamente na execução do algoritmo, não tendo em conta a leitura dos ficheiros de *input* ou o *output* dos resultados computados.

**Concordância** Para este problema, apresentamos os tempos medidos para os dois cenários: a figura 5.1b e a tabela 5.2 contêm os tempos de execução para o ficheiro de *input* `WaD.txt`, e a figura 5.2b e a tabela 5.3 contêm os da Bíblia.

Testámos com um número de *threads* (para a implementação de referência) e trabalhadores (tamanho da *pool* de *threads* do *middleware*) de 1 até 32, ou seja, 8 vezes mais que o número de cores e 4 vezes mais que o número de *threads* lógicas. A performance das duas implementações são similares. Com o ficheiro de *input* `WaD.txt`, o *speed-up* é baixo, menos de 2. Este problema tem a ver com a contenção da redução de vários *hash-map*.

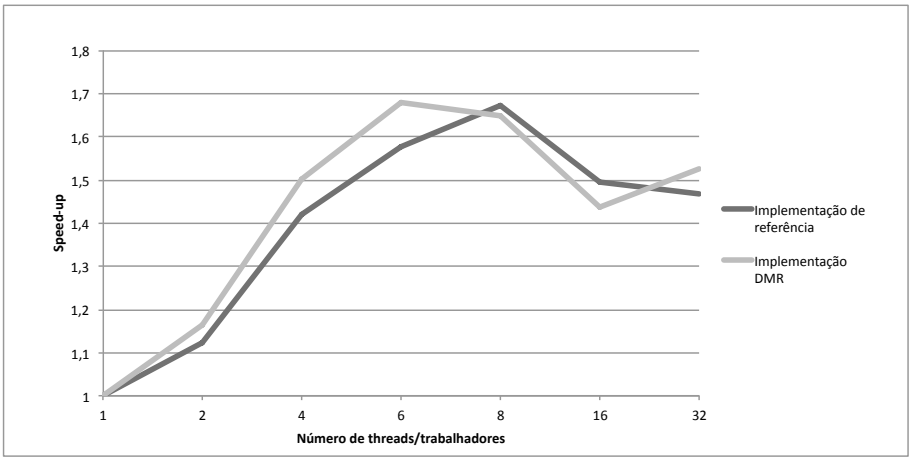
Contudo, a versão do *middleware* comporta-se um bocado melhor acima dos 6 trabalhadores mas a implementação de referência escala melhor (acima dos 8 trabalhadores), enquanto que a versão do *middleware* tem uma ligeira diminuição no desempenho de 6 a 8 trabalhadores. No entanto, com o aumento de trabalhadores, a versão do *middleware* ultrapassa versão de referência. A figura 5.1a apresenta os *speed-up's* obtidos em cada versão.

| Número de *threads*/trabalhadores | Implementação de referência ($R$) | Implementação Proposta ($P$) | *Speed-up* sobre a implementação de referência ($1 - P/R$) |
|---:|---:|---:|---:|
| 1 | 1285,15 ms | 1326,42 ms | 3% |
| 2 | 1145,64 ms | 1140,70 ms | 0% |
| 4 | 904,56 ms | 882,81 ms | 2% |
| 6 | 815,61 ms | 789,82 ms | 3% |
| 8 | 768,48 ms | 803,80 ms | -5% |
| 16 | 860,49 ms | 922,09 ms | -7% |
| 32 | 874,97 ms | 869,81 ms | 1% |

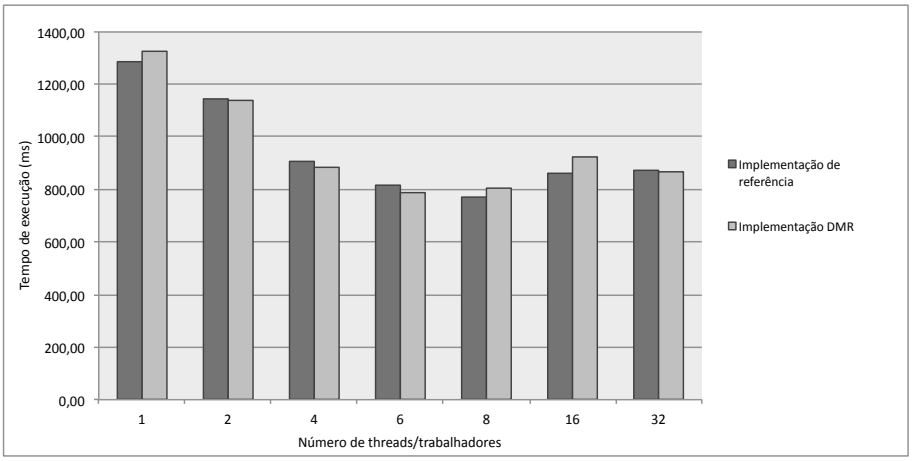
Tabela 5.2: Tempos de execução da Concordância (`WaD.txt`)

Com a Bíblia como ficheiro de *input*, as duas implementações escalam linearmente até um *speed-up* de 2,3. Desempenho obtido com 16 trabalhadores (figura 5.2a). Relativamente ao *speed-up* obtido pela versão do *middleware* sobre a versão de referência, varia de $-2\%$ a $1\%$ entre 2 e 16 trabalhadores. No entanto, a partir deste ponto, nomeadamente entre 24 e 32 trabalhadores, a versão do *middleware* comporta-se muito melhor, melhorando ligeiramente a performance sobre 16 trabalhadores, enquanto a implementação de referência duplica o seu tempo de execução. Esta informação encontra-se na coluna 4 da tabela 5.3.

**N-Body** A figura 5.3a e a tabela 5.4a apresentam os tempos medidos, a tabela 5.4b apresenta o *speed-up* sobre a implementação de referência e a figura 5.3b descreve os *speed-up's*




(a) Speed-Up



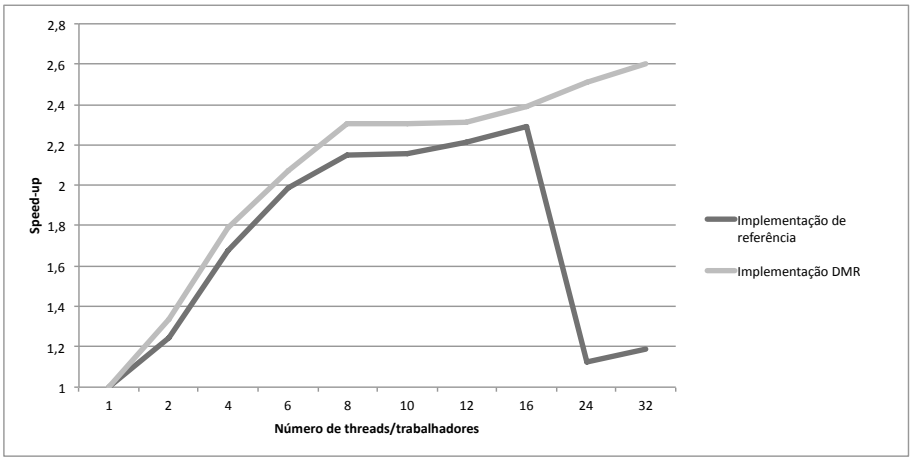

(b) Tempos de execução

Figura 5.1: Resultados da Concordância (`WaD.txt`)

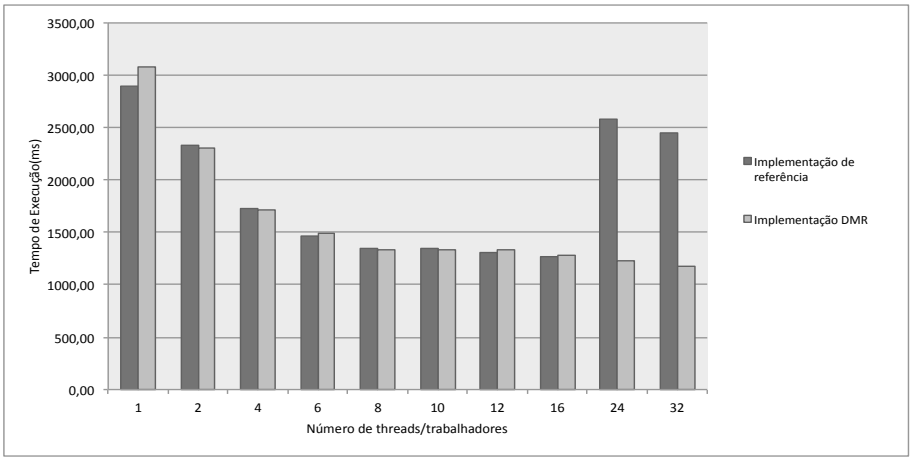
| Número de *threads*/trabalhadores | Implementação de referência ($R$) | Implementação Proposta ($P$) | *Speed-up* sobre a implementação de referência ($1 - P/R$) |
|---|---|---|---|
| 1 | 2897,16 ms | 3077,82 ms | -6% |
| 2 | 2333,15 ms | 2306,22 ms | 1% |
| 4 | 1731,38 ms | 1717,33 ms | 1% |
| 6 | 1459,97 ms | 1485,96 ms | -2% |
| 8 | 1348,09 ms | 1336,87 ms | 1% |
| 10 | 1342,74 ms | 1336,59 ms | 0% |
| 12 | 1308,47 ms | 1331,72 ms | -2% |
| 16 | 1266,10 ms | 1286,53 ms | -2% |
| 24 | 2579,01 ms | 1226,80 ms | 52% |
| 32 | 2444,63 ms | 1181,34 ms | 52% |

Tabela 5.3: Tempos de execução da Concordância (Bíblia)

de todas as versões sobre a execução de um trabalhador. Novamente, os tempos de execução são parecidos, e o *speed-up* é baixo. Deve-se ao facto de o algoritmo base ter uma complexidade de $n \log(n)$ e de a simulação levar apenas 130ms para executar, possuindo três pontos de sincronização por iteração, limitando, claramente, a performance esperada.




(a) Speed-Up




(b) Tempos de execução

Figura 5.2: Resultados da Concordância (Bíblia)

Observando mais detalhadamente, a implementação de referência apresenta uma linha de *speed-up* mais suave e possui o seu ponto mais alto com 6 trabalhadores, com um *speed-up* de 1,47. No entanto, é a versão do *middleware* baseada em tarefas que atinge a melhor performance: 85,18ms com 6 trabalhadores. Isto faz um *speed-up* de 1,64 sobre a versão com 1 trabalhador e um *speed-up* de 1,53 sobre a versão de referência com um fio de execução. Esta análise só vai até 8 trabalhadores, porque nenhuma das versões escala acima desse limite.

## 5.2 Avaliação do mecanismo DMR

Para avaliar o mecanismo DMR em ambiente distribuído, escolhemos os problemas *Java Grande Benchmark Suite* (JGF). O JGF define um conjunto de aplicações que foi desenvolvido pela comunidade *Java Grande Forum*. O objetivo desta *suite* de *benchmarks* é providenciar uma maneira de medir e comparar ambientes de execução Java alternativos de forma a que sejam importantes para aplicações *Grande*. Uma aplicação *Grande* é a que usa

| Número de *threads*/trabalhadores | Implementação referência ($R$) | Implementação com tarefas ($P1$) | Implementação DMR com limites ($P2$) | Implementação DMR com partições ($P3$) |
|---|---|---|---|---|
| 1 | 130,73 ms | 139,79 ms | 139,58 ms | 141,85 ms |
| 2 | 101,78 ms | 95,55 ms | 102,03 ms | 99,76 ms |
| 4 | 89,43 ms | 85,18 ms | 88,60 ms | 89,40 ms |
| 6 | 89,08 ms | 94,47 ms | 107,28 ms | 97,55 ms |
| 8 | 98,33 ms | 100,44 ms | 103,99 ms | 104,18 ms |

(a) Tempos de execução

| Número de *threads*/trabalhadores | Implementação com tarefas ($1 - P1/R$) | Implementação DMR com limites ($1 - P2/R$) | Implementação DMR com partições ($1 - P3/R$) |
|---|---|---|---|
| 1 | -7% | -7% | -9% |
| 2 | 6% | 0% | 2% |
| 4 | 5% | 1% | 0% |
| 6 | -6% | -20% | -10% |
| 8 | -2% | -6% | -6% |

(b) Speed-ups sobre a implementação de referência

Tabela 5.4: Resultados do N-Body

grandes quantidades de processamento, I/O, largura de banda de rede, ou memória. Estas não incluem apenas aplicações em ciência e engenharia, mas também, por exemplo, bancos de dados corporativas e simulações financeiras.

### 5.2.1 Ambiente experimental

Para esta avaliação, todas as medições foram realizadas num *cluster* com 8 nós, localizado na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa. Sendo que cinco nós contêm as seguintes características:

**Processador** AMD Quad-Core AMD Opteron 2376 a 2.30GHz com quatro cores e oito threads;

**Memória RAM** 16 GBytes de capacidade;

**Sistema Operativo** Debian com a versão do Kernel 2.6.26-2;

**JDK** Java SE Development Kit 7, 64 bits.

E os outros três nós contem as seguintes características:

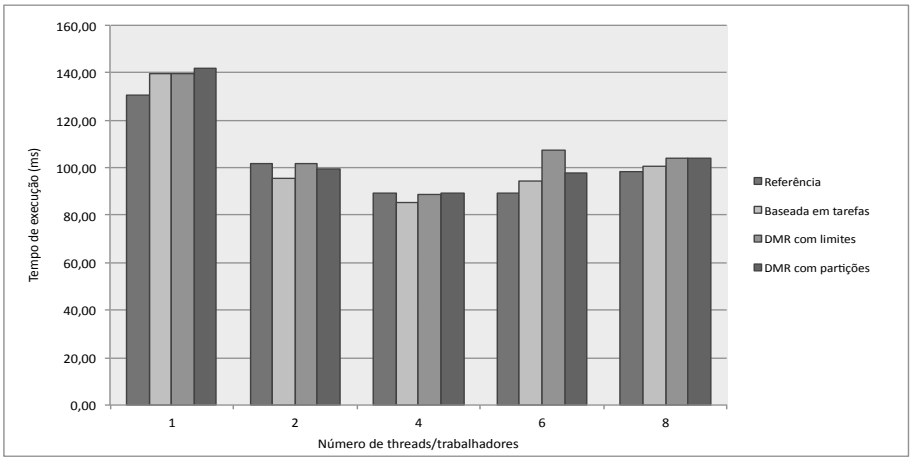
**Processador** Intel Xeon X3450 a 2.67GHz com quatro cores, com Hyper-Threading;

**Memória RAM** 8 GBytes de capacidade;

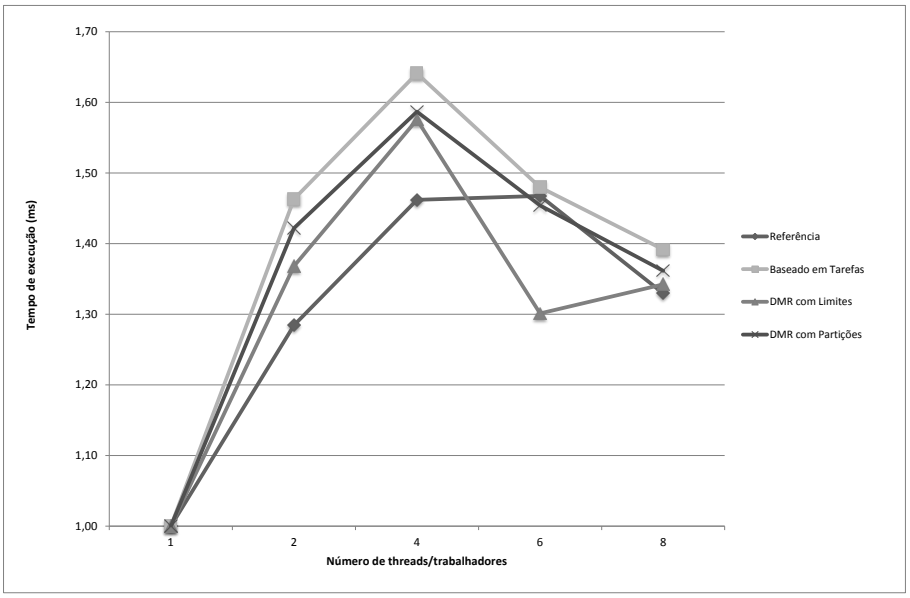
**Sistema Operativo** Debian com a versão do Kernel 2.6.26-2;

**JDK** Java SE Development Kit 7, 64 bits.

Para simularmos outro nível de heterogeneidade, foi definido um ficheiro de configuração diferente nas máquinas Intel, de forma a termos um número de trabalhadores menor que nas máquinas AMD.




(a) Tempos de execução




(b) Speed-up

Figura 5.3: Resultados do N-Body

### 5.2.2 Resultados

Como na análise anterior, medimos a execução de cada versão mais de 30 vezes, descartando $20\%$ dos melhores e piores resultados calculando a média dos resultados restantes. Estes tempos apenas medem a execução do algoritmo, isto é, não inclui o tempo de inicializarão do *middleware*. Só desta forma é que se consegue realizar uma análise junta entre os tempos obtidos pela mesma aplicação que é suportada por ambientes distintos.

O *benchmark* JGF além de fornecer as aplicações, também fornece três classes de parametrizações associadas a cada aplicação, em que cada uma destas representa uma dimensão do problema em termos de dados de entrada. Estas classes são identificadas por pequena (A), média (B) e grande (C) (Tabela 5.5).

Das várias aplicações que compunham este *benchmark*, implementámos o Crypt, o Series e o MonteCarlo[4]. O Crypt encripta e desencripta um vetor com dimensão $N$ usando o algoritmo *International Data Encryption Algorithm* (IDEA). O Series calcula os primeiros $N$ coeficientes de Fourier da função $f(x) = (x + 1)^x$ no intervalo 0,2. Este aplica fortemente funções transcendentais e trigonométricas. O MonteCarlo é uma simulação financeira, que usa técnicas de Monte Carlo para preços de produtos derivados de um ativo. O código gera $N$ amostras de tempo com a mesma média e flutuação como uma série de dados históricos.

| | Series | Crypt | MonteCarlo |
|---|---|---|---|
| A | 10.000 | 3.000.000 | 2.000 |
| B | 100.000 | 20.000.000 | 60.000 |
| C | 1.000.000 | 50.000.000 | - |

Tabela 5.5: Dimensões do problema

**Series** Além dos valores de referência, decidimos testar esta aplicação com outros valores de entrada, nomeadamente, 100 e 1000. Com estes valores, pretendemos diminuir o problema para avaliar o peso da comunicação. A tabela 5.6 apresenta os tempos medidos, com 1 a 5 localidades, sendo que cada localidade tem à sua disposição 8 trabalhadores. As tabelas 5.7a e 5.7b, e correspondentemente a figuras 5.4a e 5.4b apresentam o *speed-up* obtido sobre a versão sequencial e sobre a execução de uma só localidade. Para as execuções mais pequenas, podemos verificar o *overhead* da comunicação, sendo que este não tem tanto impacto sobre as execuções mais pesadas.

Nas execuções cuja dimensão é menor, nota-se o peso da comunicação. Este problema é mais evidente com o $N = 100$, onde o *speed-up* máximo é atingido com 3 localidades, sobre uma só localidade. Mas para problemas de maior dimensão, este *overhead* é minimizado tornando as aplicações escaláveis. Todas as curvas de *speed-up* apresentam um comportamento linear, exceto com o $N = 1.000.000$, onde a linha apresenta um *speed-up* super-linear.

| *N* | Sequencial | 1 *place* | 2 *places* | 3 *places* | 4 *places* | 5 *places* |
|---|---|---|---|---|---|---|
| 100 | 97,42 | 29,25 | 38,67 | 29,54 | 33,19 | 33,24 |
| 1.000 | 981,30 | 188,25 | 133,14 | 120,95 | 97,67 | 82,53 |
| 10.000 | 9.977,67 | 1.353,74 | 763,41 | 575,71 | 523,46 | 442,76 |
| 100.000 | 102.914,96 | 12.929,11 | 6.578,76 | 4.479,69 | 3.486,29 | 2.914,50 |
| 1.000.000 | 1.661.674,04 | 271.719,15 | 123.371,37 | 83.554,80 | 55.887,25 | 39.358,20 |

Tabela 5.6: Tempos de execução do Series em milissegundos

**Crypt** Como fizemos para o Series, testamos este *benchmark* com outros valores, nomeadamente 100, 1.000, 50.000, 100.000.000 e 200.000.000. Com valores mais baixos podemos

[4]Para mais detalhes sobre estas aplicações consulte `http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads/contents.html`

| *N* | 1 *place* | 2 *places* | 3 *places* | 4 *places* | 5 *places* |
|---|---|---|---|---|---|
| 100 | 3,33 | 2,52 | 3,30 | 2,94 | 2,93 |
| 1.000 | 5,21 | 7,37 | 8,11 | 10,05 | 11,89 |
| 10.000 | 7,37 | 13,07 | 17,33 | 19,06 | 22,54 |
| 100.000 | 7,96 | 15,64 | 22,97 | 29,52 | 35,31 |
| 1.000.000 | 6,12 | 13,47 | 19,89 | 29,73 | 42,22 |

(a) Speed-up sobre a versão sequencial

| *N* | 2 *places* | 3 *places* | 4 *places* | 5 *places* |
|---|---|---|---|---|
| 100 | 0,76 | 0,99 | 0,88 | 0,88 |
| 1.000 | 1,41 | 1,56 | 1,93 | 2,28 |
| 10.000 | 1,77 | 2,35 | 2,59 | 3,06 |
| 100.000 | 1,97 | 2,89 | 3,71 | 4,44 |
| 1.000.000 | 2,20 | 3,25 | 4,86 | 6,90 |

(b) Speed-up sobre uma localidade

Tabela 5.7: Speed-up da aplicação Series

explorar o peso da comunicação e com valores elevados podemos diluir estes peso e testar melhor o *middleware*. A tabela 5.8 apresenta dos tempos de execução com 1 a 5 localidades, e como a aplicação anterior, cada localidade possui 8 trabalhadores. As tabelas 5.9a e 5.9b e as figuras 5.5a e 5.5b apresentam o *speed-up* sobre a versão sequencial e sobre a execução de uma só localidade.

Como $N < 30.00.000$ verificamos que *overhead* da comunicação, da distribuição e redução é maior que o custo da computação, onde o *speed-up* entre localidades é inferior a 1. Sendo que para $N \geq 20.000.000$ o problema escala linearmente.

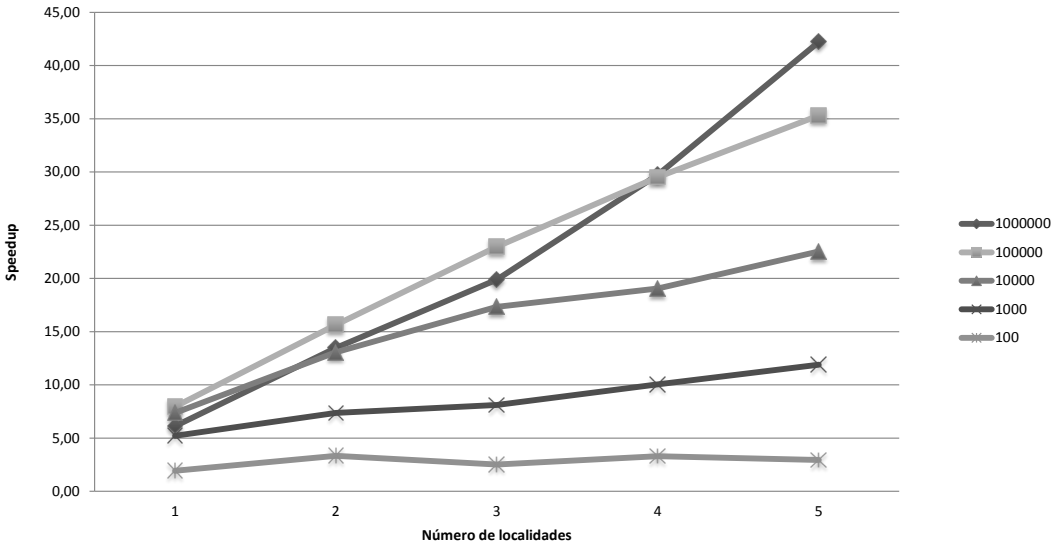
| *N* | Sequencial | 1 *place* | 2 *places* | 3 *places* | 4 *places* | 5 *places* |
|---|---|---|---|---|---|---|
| 100 | 0,50 | 1,94 | 22,44 | 26,22 | 28,61 | 30,41 |
| 1.000 | 0,50 | 1,34 | 14,62 | 19,65 | 23,76 | 30,36 |
| 50.000 | 3,67 | 3,51 | 23,00 | 28,12 | 30,29 | 36,47 |
| 3.000.000 | 202,66 | 74,10 | 58,86 | 56,42 | 54,56 | 60,08 |
| 20.000.000 | 1.350,93 | 283,50 | 188,64 | 157,06 | 137,26 | 123,06 |
| 50.000.000 | 3.373,56 | 548,45 | 367,03 | 294,80 | 248,50 | 205,06 |
| 100.000.000 | 6.746,66 | 995,53 | 675,61 | 476,44 | 406,17 | 364,37 |
| 200.000.000 | 13.485,52 | 1.878,26 | 1.045,40 | 842,77 | 693,34 | 593,01 |

Tabela 5.8: Tempos de execução do Crypt em milissegundos

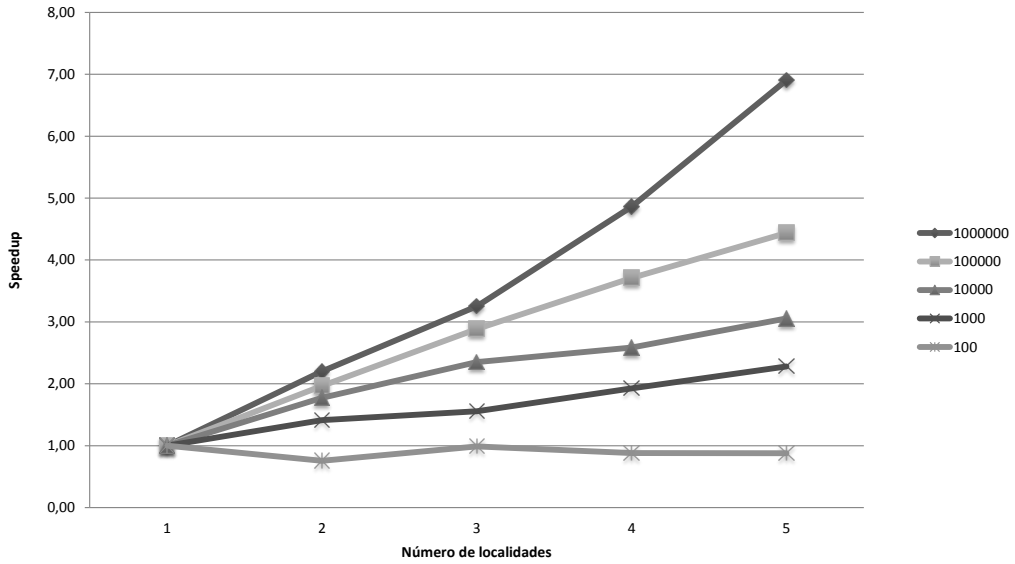
**MonteCarlo** Como o JGF apenas define as classes A e B para o MonteCarlo, definimos a nossa classe "C"com $N = 120.000$. Com este valor queremos testar esta aplicação com valores maiores. A tabela 5.10 apresenta os tempos de execução e os *Speed-ups* com 1 e 2 localidades. Com estes dados, conseguimos ver que a aplicação não escala para várias localidades.

Para tentar perceber a razão dos valores de *speed-up* baixos, decidimos fazer um pequeno *profiling* às várias fases da aplicação. Após o *profiling* conseguimos obter a tabela 5.13 com os tempos das várias fases. Salientamos o peso da comunicação, principalmente do envio, da leitura e da serialização. Com isto, concluímos que existe espaço para a otimização da comunicação, mais concretamente, no *driver* de comunicação.

Com este resultado, decidimos modificar a aplicação MonteCarlo para que após cada




(a) Speed-up sobre a versão sequencial




(b) Speed-up sobre uma localidade

Figura 5.4: Speed-up da aplicação Series

trabalhador efetue as suas tarefas, eles não enviem o resultado, mas sim um booleano a indicar que realizaram a computação. A tabela 5.12 apresenta os tempos de execução com 1 a 8 localidades, sendo que cada localidade possui 8 trabalhadores. As tabelas 5.13a e 5.13b e as figuras 5.6a e 5.6b apresentam o *speed-up* sobre a versão sequencial e sobre a execução de uma só localidade.

| *N* | 1 *place* | 2 *places* | 3 *places* | 4 *places* | 5 *places* |
|---|---|---|---|---|---|
| 100 | 0,26 | 0,02 | 0,02 | 0,02 | 0,02 |
| 1.000 | 0,37 | 0,03 | 0,03 | 0,02 | 0,02 |
| 50.000 | 1,05 | 0,16 | 0,13 | 0,12 | 0,10 |
| 3.000.000 | 2,73 | 3,44 | 3,59 | 3,71 | 3,37 |
| 20.000.000 | 4,77 | 7,16 | 8,60 | 9,84 | 10,98 |
| 50.000.000 | 6,15 | 9,19 | 11,44 | 13,58 | 16,45 |
| 100.000.000 | 6,78 | 9,99 | 14,16 | 16,61 | 18,52 |
| 200.000.000 | 7,18 | 12,90 | 16,00 | 19,45 | 22,74 |

(a) Speed-up sobre a versão sequencial

| *N* | 2 *places* | 3 *places* | 4 *places* | 5 *places* |
|---|---|---|---|---|
| 100 | 0,09 | 0,07 | 0,07 | 0,06 |
| 1.000 | 0,09 | 0,07 | 0,06 | 0,04 |
| 50.000 | 0,15 | 0,12 | 0,12 | 0,10 |
| 3.000.000 | 1,26 | 1,31 | 1,36 | 1,23 |
| 20.000.000 | 1,50 | 1,81 | 2,07 | 2,30 |
| 50.000.000 | 1,49 | 1,86 | 2,21 | 2,67 |
| 100.000.000 | 1,47 | 2,09 | 2,45 | 2,73 |
| 200.000.000 | 1,80 | 2,23 | 2,71 | 3,17 |

(b) Speed-up sobre uma localidade

Tabela 5.9: Speed-up da aplicação Crypt

| *N* | Sequencial | 1 *place* | | 2 *places* | |
|---|---|---|---|---|---|
| | | Tempo de Execução | *Speed-up* | Tempo de Execução | *Speed-up* |
| 2.000 | 609,89 | 143,05 | 4,26 | 286,34 | 2,13 |
| 60.000 | 17.362,21 | 3.310,13 | 5,25 | 10.562,44 | 1,64 |
| 120.000 | 36.347.66 | 7.420,58 | 4,90 | 20.335,13 | 1,79 |

Tabela 5.10: Tempos de execução (em milissegundos) e *Speed-ups* do MonteCarlo

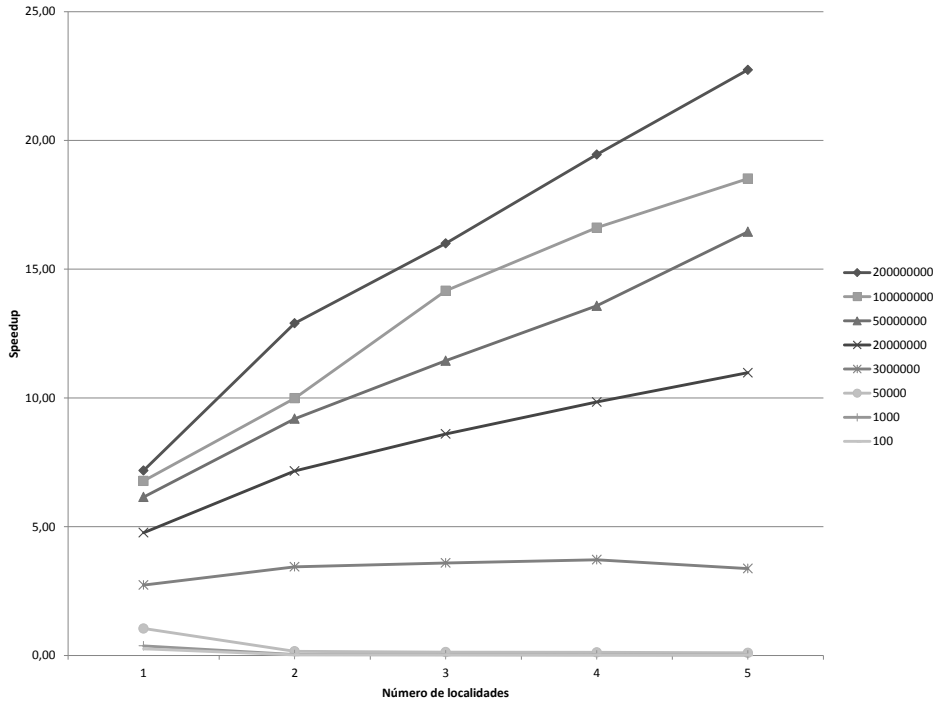
| | Distribuição | Redução | Tarefa | Comunicação | | | |
|---|---|---|---|---|---|---|---|
| | | | | Envio | Leitura | Serialização | Deserialização |
| 2.000 | 10,72 | 0,47 | 334,71 | 62,63 | 231,43 | 33,14 | 27,13 |
| 60.000 | 79,51 | 1,23 | 3.962,24 | 3.134,20 | 2.970,91 | 1.389,10 | 166,58 |
| 120.000 | 138,26 | 2,02 | 7.284,25 | 6.183,00 | 5.713,71 | 2.798,04 | 307,60 |

Tabela 5.11: *Profiling* da aplicação MonteCarlo em milissegundos

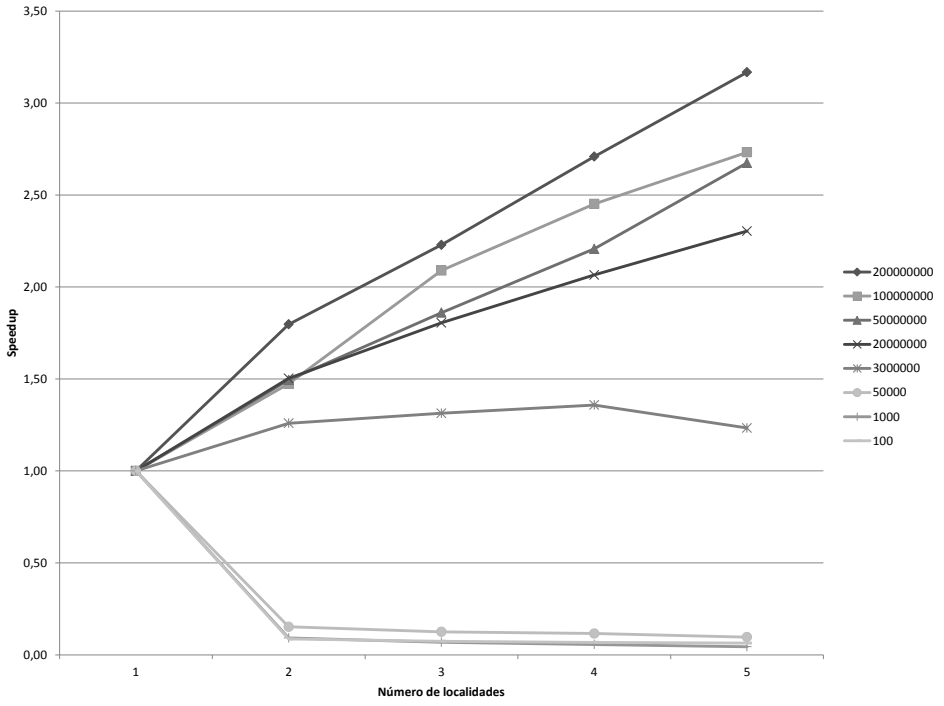
Com $N = 2.000$ verificamos que o peso da computação não compensa o peso da comunicação, mas com valores maiores, conseguimos tempos interessantes. Esta versão tem como objetivo medir a escalabilidade em termos da computação.

| *N* | Sequencial | 1 *place* | 2 *places* | 3 *places* | 4 *places* | 5 *places* | 6 *places* | 7 *places* | 8 *places* |
|---|---|---|---|---|---|---|---|---|---|
| 2.000 | 202,63 | 121,38 | 111,08 | 85,83 | 72,96 | 70,08 | 72,52 | 58,17 | 58,19 |
| 60.000 | 16.864,38 | 2.279,08 | 1.303,78 | 964,15 | 791,58 | 674,84 | 601,84 | 509,55 | 474,65 |
| 120.000 | 34.221,80 | 4.727,61 | 2.514,54 | 1.768,84 | 1.424,79 | 1.213,86 | 1.019,58 | 920,70 | 802,55 |

Tabela 5.12: Tempos de execução do MonteCarlo em milissegundos




(a) Speed-up sobre a versão sequencial




(b) Speed-up sobre uma localidade

Figura 5.5: Speed-up da aplicação Crypt

| *N* | 1 *place* | 2 *places* | 3 *places* | 4 *places* | 5 *places* | 6 *places* | 7 *places* | 8 *places* |
|---|---|---|---|---|---|---|---|---|
| 2.000 | 1,67 | 1,82 | 2,36 | 2,78 | 2,89 | 2,79 | 3,48 | 3,48 |
| 60.000 | 7,40 | 12,93 | 17,49 | 21,30 | 24,99 | 28,02 | 33,10 | 35,53 |
| 120.000 | 7,24 | 13,61 | 19,35 | 24,02 | 28,19 | 33,56 | 37,17 | 42.64 |

(a) Speed-up sobre a versão sequencial

| *N* | 2 *places* | 3 *places* | 4 *places* | 5 *places* | 6 *places* | 7 *places* | 8 *places* |
|---|---|---|---|---|---|---|---|
| 2.000 | 1,09 | 1,41 | 1,66 | 1,73 | 1,67 | 2,09 | 2,09 |
| 60.000 | 1,75 | 2,36 | 2,88 | 3,38 | 3,79 | 4,47 | 4,80 |
| 120.000 | 1,88 | 2,67 | 3,32 | 3,89 | 4,64 | 5,13 | 5,89 |

(b) Speed-up sobre uma localidade

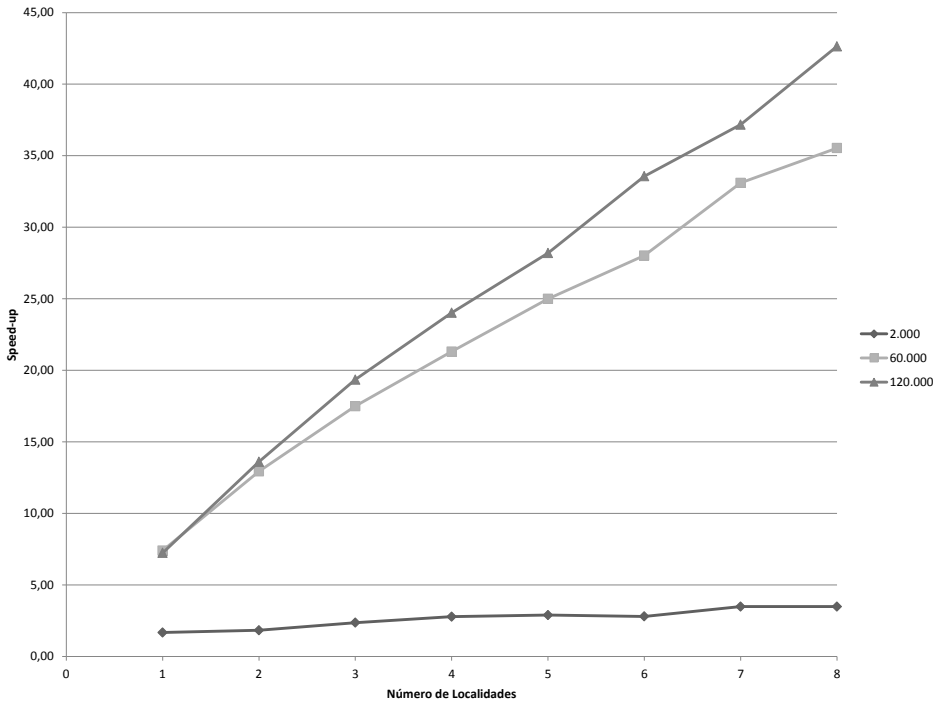
Tabela 5.13: Speed-up da aplicação MonteCarlo

## 5.3 Discussão

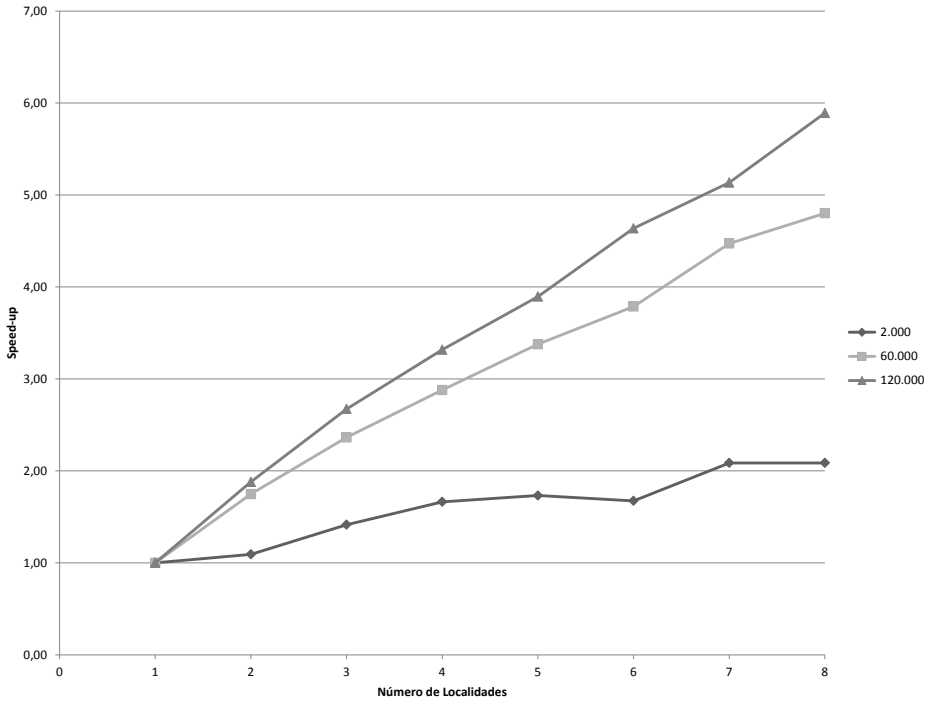
Em relação aos resultados apresentados no SICSA MultiCore Challenge, os resultados obtidos estão perto, ou melhores, do que a implementação de referência. Estes são encorajadores uma vez que oferecemos construções de alto nível para a expressão de paralelismo. No entanto os *speed-ups* globais não são muito altos. O algoritmo da Concordância está condicionado pela contenção no *hash-map*. Este seria, em teoria, um bom cenário para a aplicação do paradigma DMR, onde cada tarefa executaria sobre uma tabela de dispersão privada, eliminando a contenção. O problema é a etapa de redução, a fusão das tabelas de parciais é bastante complexa e computacionalmente pesada. Este restringe toda a aplicação, perdendo mais tempo do que a execução do algoritmo. Esta é a principal razão por detrás da pouca performance da implementação em *cluster*.

Sobre o problema do N-Body, a complexidade deste, $n\log(n)$, limita à partida a performance esperada. Naturalmente que teríamos melhores resultados em algoritmos com complexidades maiores, como por exemplo, $n^2$. A transição deste problema para um ambiente distribuído acentua algumas limitações do mecanismo de DMR, tal como está desenhado e implementado: conseguimos distribuir a computação entre um conjunto de serviços, conseguimos distribuir uma estrutura de dados entre um conjunto de serviços, mas não conseguimos fazer uma combinação dos dois. A versão de *cluster* atual distribui o vetor dos corpos entre os serviços agregados em cada ponto de sincronização, o que torna o *overhead* insuportável para a dimensão do problema.

Sobre os resultados em ambiente distribuído, salientaram-se alguns problemas na nossa abordagem. A execução de aplicações com pouca computação não é a mais indicada para este ambiente, devido ao *overhead* da comunicação. Reparámos que a implementação deste paradigma poderia ser melhorado, visto que a fase de redução é feita sequencialmente. Posto isto, era interessante a implementação de um *driver* de Distribuição/Redução que implementasse a fase de redução de forma hierárquica e paralela.




(a) Speed-up sobre a versão sequencial




(b) Speed-up sobre uma localidade

Figura 5.6: Speed-up da aplicação MonteCarlo

# 6

# Conclusões e Trabalho Futuro

Há largos anos que a aglomeração de elementos computacionais são *standard* na industria, devido ao aumento de desempenho que se dá com o aumento de núcleos de processamento em vez da velocidade de relógio. Esta tendência também se verifica na construção de infraestruturas de HPC. Com o aparecimento das arquiteturas multi-core é natural que os *cluster* atuais sejam compostos por este tipo de arquiteturas. O problema com este paradigma arquitetural é que este poderá ter uma natureza heterogénea, tanto ao nível do número de *cores*, como na velocidade dos processadores e na hierarquia de memória. Para tirar partido deste tipo de arquiteturas, é necessário que o programador tenha consciência desta e de que as ferramentas atuais são de uso complexo, expondo toda a complexidade.

Neste contexto, o trabalho desenvolvido nesta dissertação consistiu no desenvolvimento de um *middleware* para *clusters* de multi-cores, tendo como base uma biblioteca para computação paralela em arquiteturas de memória partilhada desenvolvida por Mourão. O nosso objetivo é a execução de aplicações paralelas, independentes da plataforma, em arquiteturas de memória partilhada e distribuída. Para isso, oferecemos abstrações de alto nível para a simplificação do processo de desenvolvimento de aplicações, fornecendo um conjunto de funcionalidades indispensáveis para a computação paralela. O *middleware* abstrai o *hardware* e a gestão do paralelismo através de uma interface independente da plataforma, sendo que esta assenta no conceito serviço.

O suporte para arquiteturas de memória distribuída requereu extensões ao nível da interface de programação e do sistema de execução. No primeiro salientamos a composição hierárquica de serviços. Esta consiste em compor serviços de forma simples e eficiente em ambientes de *cluster*, para que as aplicações possam explorar totalmente estes ambientes. A forma de agregar serviços é com base em comportamentos, estes são

pré-determinados e definem a funcionalidade da agregação. Definimos quatro comportamentos, o *pool* de serviços, a memória particionada, o façade e o *Distribute-Map-Reduce*.

No segundo, foi modificada a interface de programação para que esta fosse remota, sendo que esta suporta o envio de serviços, o acesso às consolas *standard* e *class loader* remoto para disponibilizar as classes dos clientes. Foram criados mecanismos para o isolamento de execução, de forma a que a execução de uma aplicação não afete uma outra, tanto em termos de exceções, como ao nível de carregamento de classes. Foi criado um *driver* com suporte à comunicação entre instâncias do *middleware* de forma eficiente, juntamente com um protocolo. Para aumentar o nível da transparência da localização dos serviços, foi criado um mecanismo de *stubs*, sendo estes uma referência para um serviço que se encontra numa localidade remota. Para que as aplicações possam executar em várias localidades, criou-se um mecanismo de escalonamento baseado em afinidades, usando uma política de *round robin*.

A premissa da adaptabilidade foi obtida resenhando o mecanismo de drivers existente, introduzindo hierarquia de *drivers* para que estes se adaptem à arquitetura alvo, sendo que os *drivers* poderão ser diferentes nas várias localidades de um *cluster*. Além da introdução da hierarquia de *drivers*, foi definindo e implementando drivers para *ranking* de nós, escalonamento, comunicação e barreiras.

Foi aferido a utilidade do modelo DMR em arquiteturas de memória partilhada e distribuída, implementando vários *benchmarks* de referência, incluindo os problemas do SICSA MultiCore Challenge. Além disso, foi efetuado um estudo de performance a este, tendo sido promissor, o que era de esperar, devido ao estudo anterior efetuado por Mourão. Este tinha concluído que o mecanismo DMR obtinha bons resultados para um determinado tipo de aplicações em memória partilhada, ou seja, com distribuições e reduções simples e com bastante computação.

## 6.1 Trabalho Futuro

Em termos de trabalho futuro existem alguns pontos que podem ser desenvolvidos. Esse pontos focam-se nas três camadas do *middleware*, sendo estes:

- Fornecer uma interface para que as aplicações possam interagir com os serviços presentes no *middleware*;
- Melhorar a análise estática, de geração das agregações, introduzindo verificações de tipos;
- Desenvolver anotações para expressar o modelo DMR dentro de uma localidade, via *Java Agent*;
- Fornecer mecanismos de sincronização entre as várias localidades;
- Desenvolver suporte para balanceamento de carga no *middleware*, migrando serviços para instâncias com pouca carga;

- Implementar mecanismos para combinar vários comportamentos de agregações de serviços, de forma que se possa agregar serviços usando dois ou mais comportamentos;
- Implementar um *driver* para a distribuição/redução de forma a otimizar a fase de redução de forma que esta seja feita de forma hierárquica.

# Bibliografia


[ACD+96] Cristiana Amza, Alan L. Cox, Hya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, e Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29:18–28, 1996.

[AFT99] Yariv Aridor, Michael Factor, e Avi Teperman. cJVM: A Single System Image of a JVM on a Cluster. In *Proceedings of the 1999 International Conference on Parallel Processing*, ICPP '99, pág. 4–, Washington, DC, USA, 1999. IEEE Computer Society.

[AG96] Sarita V. Adve e Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1996.

[AJMJS03] Jameela Al-Jaroodi, Nader Mohamed, Hong Jiang, e David R. Swanson. Middleware Infrastructure for Parallel and Distributed Programming Models in Heterogeneous Systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(11):1100–1111, 2003.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pág. 483–485, New York, NY, USA, 1967. ACM.

[BB09] Peter Bui e Jay Brockman. Performance analysis of accelerated image registration using GPGPU. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pág. 38–45, New York, NY, USA, 2009. ACM.

[BD04] Dan Bonachea e Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *Int. J. High Perform. Comput. Netw.*, 1:91–99, Agosto 2004.

[BDV94] Greg Burns, Raja Daoud, e James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pág. 379–386, 1994.

[BH86] Josh Barnes e Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, Dezembro 1986.

[Blo04] Joshua Bloch. JSR 175: A Metadata Facility for the Java Programming Language. `http://jcp.org/en/jsr/detail?id=175`, Setembro 30, 2004.

[But97] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[Can02] Stewart Cant. High-performance computing in computational fluid dynamics: progress and challenges. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 360(1795):1211–1225, 2002.

[CCL+98] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Lawrence Snyder, W. Derrick Weathersby, e Calvin Lin. The Case for High-Level Parallel Programming in ZPL. *IEEE Comput. Sci. Eng.*, 5:76–86, Julho 1998.

[CCS+09] Hua Cheng, Zuoning Chen, Ninghui Sun, Fenbin Qi, Chaoqun Dong, e Laiwang Cheng. A Virtualized Self-Adaptive Parallel Programming Framework for Heterogeneous High Productivity Computers. *Parallel and Distributed Processing with Applications, International Symposium on*, 0:543–548, 2009.

[CCZ04] David Callahan, Bradford L. Chamberlain, e Hans P. Zima. The Cascade High Productivity Language. In *HIPS*, pág. 52–60, 2004.

[CDC99] William W. Carlson, Jesse M. Draper, e David E. Culler. Introduction to UPC and Language Specification, 1999.

[CDK01] George Coulouris, Jean Dollimore, e Tim Kindberg. *Distributed systems: concepts and design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2001.

[CGJ+00] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, e Geoffrey Fox. Mpj: Mpi-like message passing for java. *Concurrency - Practice and Experience*, 12(11):1019–1038, 2000.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, e Vivek Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, 40:519–538, Outubro 2005.

[Cha02] Chingwen Chai. Consistency Issues in Distributed Shared Memory Systems. 2002.

[CJP07] Barbara Chapman, Gabriele Jost, e Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[CLL+02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, e Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pág. 258–269, New York, NY, USA, 2002. ACM.

[Cor08] Intel Corporation. Intel® QuickPath Architecture - A new system architecture for unleashing the performance of future generations of Intel® multicore microprocessors. Relatório técnico, 2008.

[CSG+03] Jack R Collins, Robert M Stephens, Bert Gold, Bill Long, Michael Dean, e Stanley K Burt. An exhaustive DNA micro-satellite map of the human genome using high performance computing. *Genomics*, 82(1):10 – 19, 2003.

[CZSS11] Vincent Cave, Jisheng Zhao, Jun Shirako, e Vivek Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 8th International Conference on Principles and Practice of Programming in Java, PPPJ 2011*. ACM, 2011.

[Dav11] Dave Cunningham and Rajesh Bordawekar and Vijay Saraswat. GPU programming in a high level language: compiling X10 to CUDA. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pág. 8:1–8:10. ACM, 2011.

[Den05] Peter J. Denning. The locality principle. *Commun. ACM*, 48:19–24, Julho 2005.

[DG08] Jeffrey Dean e Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[DLS+09] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, e Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pág. 53:1–53:11, New York, NY, USA, 2009. ACM.

[Edu12] Eduardo Marques and Hervé Paulino. Single Operation Multiple Data - Data Parallelism at Subroutine Level. In *14th IEEE International Conference on High Performance Computing & Communication, HPCC 2012, Liverpool, UK, June 25-27, 2012*. IEEE Computer Society, 06 2012.

[FBD01] Graham E. Fagg, Antonin Bukovsky, e Jack Dongarra. HARNESS and fault tolerant MPI. *Parallel Computing*, 27(11):1479–1495, 2001.

[Fer12] Massimo Re Ferrè. Intel, AMD, VMware and... Aircrafts. `http://it20.info/2007/10/intel-amd-vmware-and-aircrafts/`, 2012.

[FHK+06] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, e Pat Hanrahan. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[FLR98] Matteo Frigo, Charles E. Leiserson, e Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *In Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, pág. 212–223, 1998.

[Fly72] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, Setembro 1972.

[GCD+02] Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger, e Mitchel W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pág. 77–83, New York, NY, USA, 2002. ACM.

[GLT99] William Gropp, Ewing Lusk, e Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[GSC+10] Anwar Ghuloum, Amanda Sharp, Noah Clemons, Stefanus Du Toit, Rama Malladi, Mukesh Gangadhar, Michael McCool, e Hans Pabst. Array Building Blocks: A Flexible Parallel Programming Model for Multicore and Many-Core Architectures. *Dr. Dobbs Go Parallel*, 210. `http://drdobbs.com/go-parallel/article/showArticle.jhtml?articleID=227300084`.

[GWS05] Richard L. Graham, Timothy S. Woodall, e Jeffrey M. Squyres. Open MPI: A Flexible High Performance MPI. In *In The 6th Annual International Conference on Parallel Processing and Applied Mathematics*, 2005.

[HCB04] Fabrice Huet, Denis Caromel, e Henri E. Bal. A High Performance Java Middleware with a Real Application. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pág. 2–, Washington, DC, USA, 2004. IEEE Computer Society.

[HGS99] K. Haviland, D. Gray, e B. Salama. *UNIX system programming: a programmer's guide to software development*. Addison-Wesley, 1999.

[HLW00] Adolfy Hoisie, Olaf Lubeck, e Harvey Wasserman. Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications. *International Journal of High Performance Computing Applications*, 14(4):330–346, 2000.

[Hoa74] Hoare, C. A. R. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, Outubro 1974.

[HW10] Georg Hager e Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2010.

[IBM11] IBM Corporation. What is a socket? `http://publib.boulder.ibm.com/infocenter/zos/v1r12/topic/com.ibm.zos.r12.cbcpx01/ovsock.htm#ovsock`, Junho 2011.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, e Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pág. 59–72, New York, NY, USA, 2007. ACM.

[Kam07] Alan Kaminsky. Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pág. 1–8. IEEE, 2007.

[KGK+03] Rainer Keller, Edgar Gabriel, Bettina Krammer, Matthias S. Müller, e Michael M. Resch. Towards Efficient Execution of MPI Applications on the Grid: Porting and Optimization Issues. *Journal of Grid Computing*, 1:133–149, 2003. 10.1023/B:GRID.0000024071.12177.91.

[Khr09] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.33*, Abril 2009.

[KK07] Christoph Kessler e Jörg Keller. Models for Parallel Computing: Review and Perspectives. In *PROCEEDINGS, PARS*, pág. 13–29, 2007.

[KWSS03] Dawid Kurzyniec, Tomasz Wrzosek, Vaidy Sunderam, e Aleksander Slominski. RMIX: A Multiprotocol RMI Framework for Java. *Parallel and Distributed Processing Symposium, International*, 0:140, 2003.

[LMS05] P. Leach, M. Mealling, e R. Salz. RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace, Julho 2005. `http://www.ietf.org/rfc/rfc4122.txt`.

[LS96] R. Greg Lavender e Douglas C. Schmidt. Active object: An object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*, pág. 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[LS97] Luís M. B. Lopes e Fernando M. A. Silva. Thread- and Process-Based Implementations of the pSystem Parallel Programming Environment. *Softw., Pract. Exper.*, 27(3):329–351, 1997.

[LW89] Jerrold S. Leichter e Robert A. Whiteside. Implementing Linda for distributed and parallel processing. In *Proceedings of the 3rd international conference on Supercomputing*, ICS '89, pág. 41–49, New York, NY, USA, 1989. ACM.

[MAJJS02] Nader Mohamed, Jameela Al-Jaroodi, Hong Jiang, e David Swanson. JOPI: a Java object-passing interface. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, JGI '02, pág. 37–45, New York, NY, USA, 2002. ACM.

[man01] Efficient Java RMI for parallel programming. *ACM Trans. Program. Lang. Syst.*, 23:747–775, Novembro 2001.

[Man07] Jenny Mankin. CSG280: Parallel Computing Memory Consistency Models: A Survey in Past and Present. Dezembro 2007.

[Mar12] Eduardo Marques. Single Operation Multiple Data - Paralelismo de Dados ao Nível da Subrotina. Tese de Mestrado, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2012.

[Mou11] Diogo André Ribeiro Mourão. Um middleware independente da plataforma para computação paralela. Tese de Mestrado, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2011.

[MRZ95] Masaaki Mizuno, Michel Raynal, e James Zhou. Sequential consistency in distributed systems. In Kenneth Birman, Friedemann Mattern, e André Schiper, editores, *Theory and Practice in Distributed Systems*, volume 938 of *Lecture Notes in Computer Science*, pág. 224–241. Springer Berlin / Heidelberg, 1995. 10.1007/3-540-60042-6_16.

[MWL00] Matchy J. M. Ma, Cho-Li Wang, e Francis C.M. Lau. JESSICA: Java-Enabled Single-System-Image Computing Architecture, 2000.

[NMM+03] Henrik F. Nielsen, Noah Mendelsohn, Jean J. Moreau, Martin Gudgin, e Marc Hadley. SOAP version 1.2 part 1: Messaging framework. W3C recommendation, W3C, Junho 2003.

[NVI10] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2010. Version 3.2.

[OC03] Robert O'Callahan e Jong-Deok Choi. Hybrid dynamic data race detection. *SIGPLAN Not.*, 38:167–178, Junho 2003.

[OHL+08] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, e J.C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879 –899, Maio 2008.

[PHN00] Michael Philippsen, Bernhard Haumacher, e Christian Nester. More efficient serialization and rmi for java, 2000.

[Qua01] Quadrics. *SHMEM Programming Manual*. Quadrics Supercomputers World Ltd, Bristol, UK, 2001.

[QZHZ05] Weiguang Qiao, Guosun Zeng, An Hua, e Fei Zhang. Scheduling and Executing Heterogeneous Task Graph in Grid Computing Environment. In *GCC*, pág. 474–479, 2005.

[Ras70] Philip J. Rasch. A Queueing Theory Study of Round-Robin Scheduling of Time-Shared Computer Systems. *J. ACM*, 17:131–145, Janeiro 1970.

[Rus78] Richard M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21:63–72, Janeiro 1978.

[SGG05] Abraham Silberschatz, Peter Baer Galvin, e Greg Gagne. *Operating System Concepts*. Wiley Publishing, 7th edition, 2005.

[SGI11] SGI. intro_shmem - Introduction to the SHMEM programming model. `http://docs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=linux&db=man&fname=/usr/share/catman/man3/intro_shmem.3.html&srch=intro_shmem`, Junho 2011.

[Sie10] Sam Siewert. Using Intel® Streaming SIMD Extensions and Intel® Integrated Performance Primitives to Accelerate Algorithms. `http://software.intel.com/en-us/articles/`, 2010.

[SM09] Aamir Shafi e Jawad Manzoor. Towards efficient shared memory communications in MPJ express. In *IPDPS*, pág. 1–7, 2009.

[soa] Reference Architecture Foundation for Service Oriented Architecture. Relatório técnico, OASIS.

[SOW+95] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, e Steven Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[SPL99] Fernando M. A. Silva, Hervé Paulino, e Luís M. B. Lopes. Di_pSystem: A Parallel Programming System for Distributed Memory Architectures. In *PVM/MPI*, pág. 525–532, 1999.

[Sri95] R. Srinivasan. RPC: remote procedure call protocol specification version 2. 1995.

[Su10] Hung-Hsun Su. A Study of GASNet Communication System Overview, Evaluation, and Usage. 2010.

[Sun98] Sun Microsystems. *Java Remote Method Invocation Specification, JDK 1.2*. Sun Microsystems, Mountain View, Calif., Outubro 1998.

[Tab09] Guillermo López Taboada. *Design of Efficient Java Communications for High Performance Computing*. Tese de Doutoramento, University of A Coruña, Maio 2009.

[TC10] The OASIS Research Team e ActiveEon Company. *ProActive Programming - Get Started*, Janeiro 2010.

[TTD09] Guillermo L. Taboada, Juan Touriño, e Ramon Doallo. Java for high performance computing: assessment of current research and practice. In Ben Stephenson e Christian W. Probst, editores, *PPPJ*, pág. 30–39. ACM, 2009.

[ULB07] Keith D. Underwood, Michael J. Levenhagen, e Ron Brightwell. Evaluating NIC hardware requirements to achieve high message rate PGAS support on multi-core processors. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pág. 36:1–36:10, New York, NY, USA, 2007. ACM.

[UMG08] J. Ueyama, E.R.M. Madeira, e P. Grace. FlexPar: Reconfigurable Middleware for Parallel Environments. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pág. 312 –316, Maio 2008.

[vNMW+05] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesińska, Rutger F. H. Hofman, Ceriel J. H. Jacobs, Thilo Kielmann, e Henri E. Bal. Ibis: a flexible and efficient Java-based Grid programming environment: Research Articles. *Concurr. Comput. : Pract. Exper.*, 17:1079–1107, Junho 2005.

[Wik12] Wikipedia. Symmetric multiprocessing. `http://en.wikipedia.org/wiki/Symmetric_multiprocessing`, 2012.

[WOS08] Michael Wehner, Leonid Oliker, e John Shalf. Towards Ultra-High Resolution Models of Climate and Weather. *International Journal of High Performance Computing Applications*, 22(2):149–165, 2008.

[YBC+07] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Welco Michael, e Tong Wen. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, PASCO '07, pág. 24–32, New York, NY, USA, 2007. ACM.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, e Alex Aiken. Titanium: A High-Performance Java Dialect. In *In ACM*, pág. 10–11, 1998.

[YVPH08] Nobuko Yoshida, Vasco Thudichum Vasconcelos, Hervé Paulino, e Kohei Honda. Session-based compilation framework for multicore programming. In Frank S. de Boer, Marcello M. Bonsangue, e Eric Madelain, editores, *Formal Methods for Components and Objects, 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures*, volume 5751 of *Lecture Notes in Computer Science*, pág. 226–246. Springer, 2008.

# Estudo de bibliotecas de comunicação

Neste apêndice descrevemos algumas primitivas de comunicação utilizadas na implementação de sistemas de execução para arquiteturas de memória distribuída. Em particular, incidimos sobre dois tipos de comunicação: *Two-sided* e *One-sided*. Interessa-nos essencialmente as primitivas disponíveis na linguagem Java.

## A.1 *Two-sided*

As primitivas de comunicação *Two-sided* são caracterizadas por uma cooperação entre o recetor e o emissor, definindo pontos de envio e de receção de mensagens, indicado explicitamente parâmetros, e por não haver uma separação entre a comunicação e a sincronização, sendo que estas são implementadas sobre as mesmas primitivas.

### A.1.1 *Sockets*

Os *Sockets*, também designados por *Internet sockets* ou *network sockets*, são um ponto de entrada de fluxo de comunicação bidirecional de uma rede de computadores sobre *Internet Protocol* (IP) [IBM11]. Estes constituem um mecanismo para entregar pacotes de dados para um apropriado fio de execução, identificado por uma combinação de um endereço IP/porta local e remota. Os protocolos de transporte suportados pelos *sockets* são o *User Datagram Protocol* (UDP) e o TCP. O UDP é um protocolo de transporte não-fiável, ou seja, não garante a ordem e entrega dos datagramas[1]. Por outro lado, o TCP é

[1] As mensagens no UDP designam-se por datagramas.

um protocolo de transporte fiável, onde garante a chegada e ordem dos pacotes[2].

### A.1.2 MPI

O MPI é um sistema de troca de mensagens estandardizado e portável, desenhado por um grupo de investigadores académicos e industriais, para funcionar numa grande variedade de computadores paralelos [SOW+95]. O *stardard* do MPI define a sintaxe e a semântica do core da biblioteca, para que os utilizadores possam desenvolver programas com troca de mensagens portáveis em Fortran 77 ou C. O MPI consiste apenas numa especificação de uma biblioteca, ou seja, da sua interface e da sua funcionalidade, existindo várias implementações, como por exemplo: OpenMPI [GWS05], LAM/MPI [BDV94], LA-MPI [GCD+02], FT-MPI [FBD01] e PACX-MPI [KGK+03].

### A.1.3 RMI

O RMI é uma protocolo que permite ao utilizador criar aplicações distribuídas através da utilização de objetos Java remotos. A invocação de métodos nestes objetos é efetuada da mesma forma do que em objetos locais, pelo que a sua utilização é transparente.

O RMI é, essencialmente, o sistema de RPC do Java. No entanto, o facto de ser implementado em Java faz com que transporte consigo o poder da segurança e da portabilidade da linguagem, para a computação distribuída. Entre as diversas vantagens sobre os tradicionais sistemas de RPC, salienta-se: a) o facto de ser orientado a objetos; b) a segurança; c) a facilidade de desenvolvimento e de utilização, e; d) o *garbage collection* distribuído.

### A.1.4 Java NIO

O NIO é uma API alternativa às primitivas *stardard* do Java. O foco desta API é a alta performance, disponibilizando mecanismo de baixo nível, assim aproximando a interface de programação com a interface do subsistema de I/O dos sistemas operativos. Os principais mecanismos que esta API disponibiliza são os *buffers*, *channels* e os *selectors*. Os *buffers* permitem realizar transferências de dados; estes representam uma zona contigua de memória. Tipicamente, este mecanismo pode usar a mesma zona de memória das operações de I/O nativas, permitindo uma transferência de dados mais direta e elimina a necessidade de uma cópia adicional dos dados. Em grande parte dos sistemas operativos, se a área de memória das operações de I/O tiver as propriedades adequadas, a transferência de dados pode ser realizada sem a intervenção do CPU, usando *Direct Memory Access* (DMA). Os *channels* disponibilizam canais de comunicação que proporcionam transferências de dados, de e para os *buffers*. Estes *channels* são análogos aos *file descriptors* dos sistemas operativos baseados em UNIX. Os *selectors* disponibilizam mecanismos para esperar por eventos nos *channels*, como por exemplo, a chegada de dados.

[2] As mensagens no TCP designam-se por pacotes.

Quando vários *channels* estão registados num *selector*, este bloqueia a execução até que um dos *channels* esteja pronto a ser usado, ou até uma interrupção. Embora este comportamento de multiplexagem poderiam ser implementados usando vários fios de execução, os *selectors* disponibilizam este comportamento usando construtores de baixo nível, aumentando a performance. Este comportamento encontra-se em sistemas operativos POSIX, permitindo aos *selectors* usar primitivas do sistema operativo.

## A.2 *One-sided*

As primitivas de comunicação *One-sided* são caracterizadas por usar o paradigma de RMA, pela comunicação estar separada da sincronização.

### A.2.1 MPI2

Em 1997 a MPI Fórum introduziu o MPI-2, que é uma extensão do MPI-1, acrescentando as seguintes funcionalidades [GLT99]:

- *I/O* Paralelo
- RMA
- Gestão de processos dinâmicos

O *I/O* Paralelo, ou simplesmente MPI-IO, é um sistema de *I/O* que disponibiliza uma interface de alto nível com suporte para partição de ficheiros pelos vários processos, tal como interface coletiva que permite transferências de estruturas de dados globais entre as memórias dos processos e os ficheiros. Outras características deste sistema são *I/O* não bloqueante, acesso não contíguo a memória e ficheiros, uso de deslocamentos (*offsets*) explícitos e apontadores de ficheiros partilhados e individuais.

O RMA estende os mecanismos de comunicação do MPI permitindo que um processo especifique todos os parâmetros da comunicação, tanto para enviar como para receber. Este modo de comunicação facilita a programação de algumas aplicações que mudam dinamicamente o padrão de acesso a dados, onde a distribuição dos mesmos é fixa ou com poucas mudanças. Em qualquer caso, cada processo consegue calcular que dados precisa de aceder ou atualizar noutros processos. No entanto, os processos podem não saber que endereços da sua memória têm de ser acedidos por outros, nem conhecerem a sua identidade destes. Assim, os parâmetros da transferência estão disponíveis apenas num dos lados da operação. As operações de enviar/receber usuais requerem uma operação correspondente pelo emissor e o recetor. Para resolver o problema das operações correspondentes, uma aplicação precisa de distribuir os parâmetros de transferência. Isto pode requerer que todos os processos participem numa computação global dispendiosa, ou periodicamente verificar potenciais pedidos de comunicação. O uso de

mecanismos de comunicação RMA evita a necessidade de cálculos globais ou de verificações explícitas. O design do RMA separa as funções de comunicação da sincronização. Para comunicar são disponibilizados as seguintes funções:

**MPI_PUT** - Escrita remota;

**MPI_GET** - Leitura Remota;

**MPI_ACCUMULATE** - Atualização Remota.

A Gestão de processos dinâmicos permite a criação e a terminação cooperativa de processos, depois de uma aplicação MPI ser inicializada. Este modelo também providencia um mecanismo para estabelecer a comunicação entre duas aplicações existentes de MPI, mesmo que não tenham sido inicializadas ao mesmo tempo.

### A.2.2 Shmem

O Shmem [Qua01, SGI11] é uma API de programação que permite ao utilizador escrever aplicações paralelas com o modelo de memória partilhada, através de primitivas de comunicação de baixa latência e com elevada largura de banda. As funções do Shmem suportam transferência de dados remotos através da operação de *put*, que transfere dados para diferentes *Processing Elements* (PE), da operação de *get*, que transfere dados de um PE diferente e de apontadores remotos, que permitem referências diretas para dados pertencentes a outros PE. Outras operações suportadas são redução e *broadcast* colaborativos, barreiras de sincronização e operações sobre memória atómicas. Uma operação sobre memória atómica é uma operação de ler e atualizar atómica, na memória local ou remota.

### A.2.3 GASNet

O GASNET [Su10] é uma camada de comunicação de baixo nível, independente da linguagem, desenvolvida na Universidade de Berkeley que providencia primitivas de comunicação de alta performance independentes da rede, tendo como objetivo suportar o modelo de programação paralelo SPMD. Atualmente, o GASNET suporta a execução em vários tipos de redes, como por exemplo, UDP, MPI, Dolphin SCI, Cray XT Portals, Cray X1/SGI, Altix Shmem, Myrinet, Quadrics, Infiniband, IBM BlueGene/P DCMF e IBM LAPI. O GASNET é dividido em duas camadas: o GASNET *Core* API e o GASNET *Extended* API.

O *Extended* API é uma interface independente da rede que disponibiliza operações de médio/alto nível, como por exemplo, operações bloqueantes e não bloqueantes de acesso a memória partilhada remota (*put/get*). Sempre que possível, as funções do *Extended* API são implementadas diretamente por cima da API de interconexão (i.e., acesso direto) para maximizar a performance, em termos de latência e débito.

Por outro lado, o *Core* API é uma interface geral baseada no paradigma de *Active Message*, com implementações específicas para cada tipo de rede. Esta camada inclui funções para preparar e terminar a comunicação, como por exemplo, enviar/receber/executar *Active Messages* e definir o tamanho do espaço de memória partilhado.

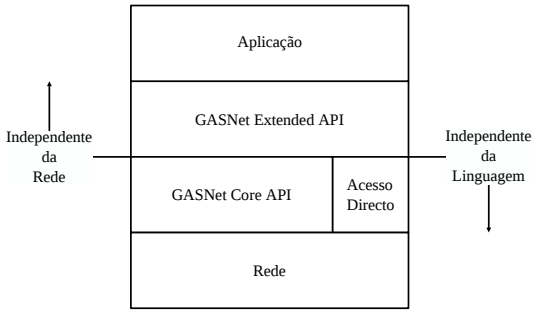



Figura A.1: Arquitetura do GASNet [Su10]

### A.2.4 Comparação crítica

Iremos fazer uma comparação crítica sobre as bibliotecas de comunicação *Two-sided*, focando-nos nas bibliotecas suportadas em Java, devido à não existência de bibliotecas de comunicação *One-sided* em Java.

O MPI 1 é suportado pelo Java usando o *Java Native Interface* (JNI) e por uma implementação em Java (MPJ) [SM09]. O JNI é uma *framework* que permite um programa em Java chamar aplicações e bibliotecas nativas escritas noutras linguagens, como o C, C++ ou o Assembly. O MPI, além de primitivas de comunicação, fornece um modelo de programação que, no nosso caso, não é interessante, uma vez que o nosso *middleware* não necessita do modelo de programação do MPI, já que este pretende fornecer um modelo próprio.

Infelizmente, nem o MPI 1, nem o MPI 2 providenciam comunicações adequadas para linguagens do tipo *Global Addressing Space* (GAS) [BD04]. Simular comunicação *one-sided* em MPI 1 é muito pesado, enquanto a comunicação *one-sided* do MPI 2 impõe um número significante de restrições no acesso à memória ao nível da linguagem. Este problema deve-se ao facto de o compilador não conseguir esconder os mecanismos de deteção de conflitos e de *alias* ao utilizador.

O RMI é bastante usado no mundo do Java em sistemas distribuídos. Sendo por um lado, simples de usar, por outro, tem uma performance fraca [Tab09]. Para resolver este problema, foram desenvolvidas algumas bibliotecas, usando a mesma interface do RMI, tendo estas melhor performance e a possibilidade de serem usadas em comunicação HPC em *clusters* de máquinas. Destas bibliotecas, as mais relevantes são KaRMI [PHN00], RMIX [KWSS03], Manta [man01] e o Ibis RMI [vNMW+05].

Após uma análise de performance com um simples teste de RTT aos vários mecanismos de comunicação (*Message Oriented Middleware* (MOM), MPI, MPJ, TCP, UDP, NIO)

chegamos ao gráfico A.2, onde, com pacotes interiores a 1500 bytes, o TCP demora 80 milissegundos.

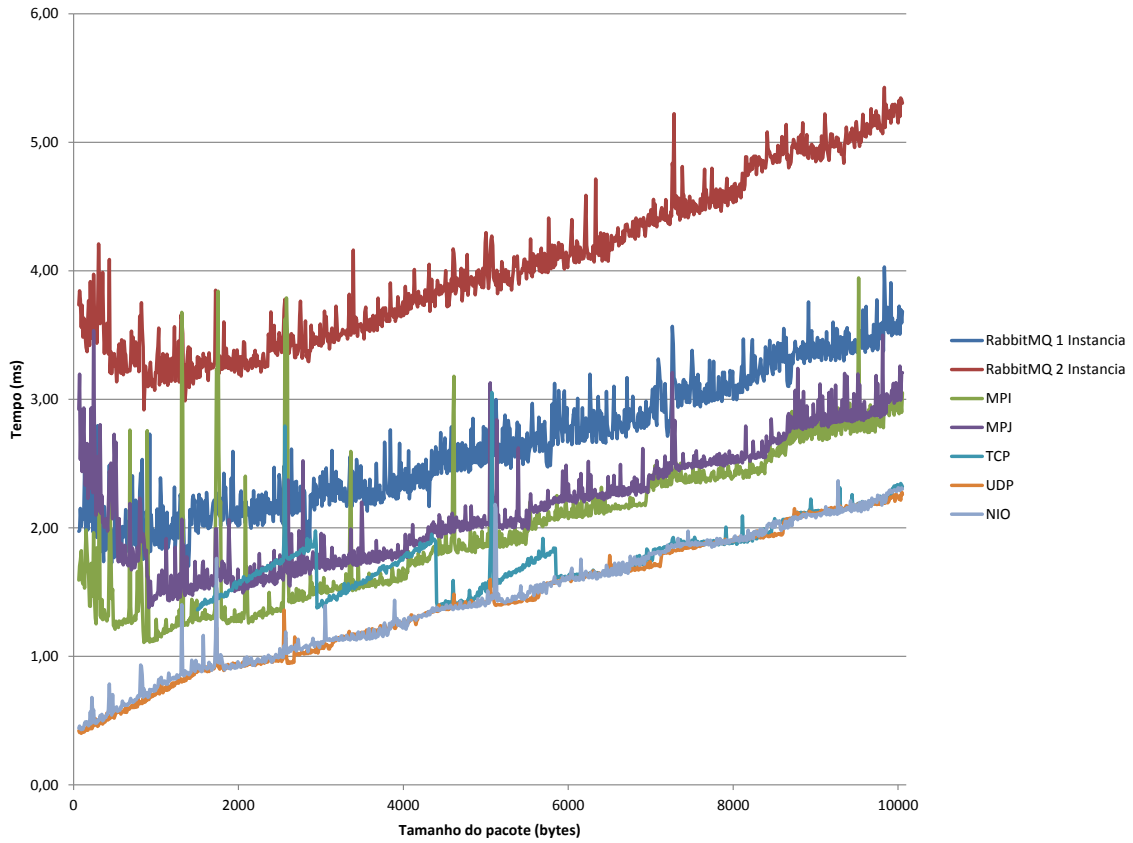



Figura A.2: Tempos em milissegundos do teste de RTT

Neste gráfico podemos verificar as seguintes afirmações:

- O NIO apresenta valores de RTT inferiores;
- O UDP apresenta bons valores de RTT, mas tem perda de pacotes;
- O TCP não é um para comunicação com grão fino.

Medimos também o consumo de CPU e de memória destes testes o que resultou na Tabela A.1.

| | **Utilização de CPU (max)** | **Utilização de Memória (max)** |
|---|---|---|
| MPI | 40% | 14Mb |
| MPJ | 79% | 13Mb |
| NIO | 45% | 35Mb |
| RabbitMQ | 19% | 19Mb |
| TCP | 19% | 250Mb |
| UDP | 29% | 13Mb |

Tabela A.1: Utilização de CPU e de Memória nos testes de RTT

Após a analise destes resultados decidimos escolher o Java NIO como primitiva de comunicação, devido à sua baixa latência e aceitável utilização de CPU e de memória.

# B

# *Class Loader* Remoto

```
class remoteclass implements IRemoteClassLoader, Runnable {

  private String ip;
  private ServerSocket server;
  private int port;

  public remoteclass( {
    ip = System.getProperty("java.rmi.server.hostname");
    try {
      server = new ServerSocket(0);
      port = server.getLocalPort(;
      server.setReuseAddress(true);
    } catch (IOException e) {
      e.printStackTrace(;
    }
  }

  public byte[ loadClass(String name) {
    try {
      String name = classs.replace('.', File.separatorChar) + ".class";
      InputStream stream = ClassLoader.getSystemClassLoader().getResourceAsStream(
          name);
      int size = stream.available();
      byte buff[] = new byte[size];
      DataInputStream in = new DataInputStream(stream);
      in.readFully(buff);
      in.close();
      return buff;
    } catch (IOException e) {
```

```
      e.printStackTrace();
    }
    return null;
  }

  public void run() {
    Socket aux;
    while (true) {
      try {
        aux = server.accept();
        DataOutputStream dos = new DataOutputStream(
            aux.getOutputStream());
        DataInputStream dis = new DataInputStream(aux.getInputStream());
        byte[] data = loadClass(dis.readUTF());
        dos.writeInt(data.length);
        dos.flush();
        dos.write(data);
        dos.flush();
        dos.close();
        dis.close();
        aux.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Listagem B.1: *Class Loader* Remoto

# Exemplo de um *stub*

```
package services.MathService;

/*GENERATED CLASS - DO NOT EDIT*/

import instrumentation.definitions.Generated;
import services.MathService.Math;

@Generated
public class MathProviderStub extends ServiceStub implements services.MathService.
    Math, java.io.Serializable{

  public MathProviderStub(java.util.UUID id, java.util.UUID clientid,core.Level l)
      {
    super(id,clientid,l);
  }

  public double getPI (double rounds) {
    try{
      IFuture<Double> aux = this.invoke("getPI",new Object[]{rounds});
      return aux.get();
    }catch (NoSuchMethodException e) {
      return 0;
    }
  }

  public int fib (int n) {
    try{
      IFuture<Integer> aux = this.invoke("fib",new Object[]{n});
      return aux.get();
```

```
    }catch (NoSuchMethodException e) {
      return 0;
    }
  }

  public void mergesort (int[] data, int first, int n) {
    try{
      IFuture<Void> aux = this.invoke("mergesort",new Object[]{data, first, n});
      aux.get();
    }catch (NoSuchMethodException e) {

    }
  }

  public void merge (int[] data, int first, int n1, int n2) {
    try{
      IFuture<Void> aux = this.invoke("merge",new Object[]{data, first, n1, n2});
      aux.get();
    }catch (NoSuchMethodException e) {

    }
  }
}
```

Listagem C.1: *Stub* para o serviço Math

# D

# Exemplo de a agregação com o comportamento *pool* de serviços

```
package services.MathService;

/*GENERATED CLASS - DO NOT EDIT*/

import instrumentation.definitions.Generated;

@Generated
public class MathPool extends ServicePool implements Service, Math{

  public MathPool(Service[] services,IServiceScheduler scheduler) {
    super(services,scheduler);
  }

  public double getPI (double rounds) {
    try{
      IFuture<Double> aux = this.invoke("getPI",new Object[]{rounds});
      return aux.get();
    }catch (NoSuchMethodException e) {
      return 0;
    }
  }

  public int fib (int n) {
    try{
      IFuture<Integer> aux = this.invoke("fib",new Object[]{n});
      return aux.get();
```

```
    }catch (NoSuchMethodException e) {
      return 0;
    }
  }

  public void mergesort (int[] data, int first, int n) {
    try{
      IFuture<Void> aux = this.invoke("mergesort",new Object[]{data, first, n});
      aux.get();
    }catch (NoSuchMethodException e) {

    }
  }
}
```

Listagem D.1: *Pool* de serviços para o serviço Math

# Exemplo de agregação com o comportamento de memória partilhada

```
package banco;

/*GENERATED CLASS - DO NOT EDIT*/

import instrumentation.definitions.Generated;

@Generated
public class BancoPartMem extends PartitionedMemPlace<java.lang.Integer,java.lang.
    Double> implements PartitionedMemService<java.lang.Integer,java.lang.Double>,
    Banco{

  public BancoPartMem(AbstractPartitionedMemPlace<java.lang.Integer,java.lang.
      Double>[] services,IPartitioner<java.lang.Integer,java.lang.Double>
      partitioner) {
    super(services,partitioner);
  }
  public void depositar (int conta, double valor) {
    try{
      IFuture<Void> aux = this.invoke(conta,"depositar",new Object[]{conta, valor})
          ;
      aux.get();
    }catch (NoSuchMethodException e) {
      e.printStackTrace();
    }
```

```
  }
  public void levantar (int conta, double valor) {
    try{
      IFuture<Void> aux = this.invoke(null,"levantar",new Object[]{conta, valor});
      aux.get();
    }catch (NoSuchMethodException e) {
      e.printStackTrace();
    }
  }
  public void transferencia (int contaOrigem, int contaDestino, double valor) {
    try{
      IFuture<Void> aux = this.invoke(contaOrigem,"transferencia",new Object[]{
          contaOrigem, contaDestino, valor});
      aux.get();
    }catch (NoSuchMethodException e) {
      e.printStackTrace();
    }
  }
  public double getSaldo (int conta) {
    try{
      IFuture<Double> aux = this.invoke(conta,"getSaldo",new Object[]{conta});
      return aux.get();
    }catch (NoSuchMethodException e) {
      e.printStackTrace();
      return 0;
    }
  }
}
```

Listagem E.1: Agregação com o comportamento de memória partilhada para o serviço Banco

# F

# Código do problema N-Body

```
public interface NBodyService extends Service {
  /**
   *
   * @param ntimesteps number of time steps to run
   * @param dtime length of one time step
   * @param eps potential softening parameter
   * @param tol tolerance for stopping recursion
   * @param dthf half of dtime
   * @param epssq softening factor
   * @param itolsq equal to 1.0 / (tol * tol)
   */
  @Task
  void runSimulation(int ntimesteps, double dtime, double eps, double tol, double
      dthf, double epssq, double itolsq);
}
```

Listagem F.1: Interface NBodyService

```
1  public class NBodyProviderTask extends ServiceProvider implements NBodyService{
2    public static OctTreeLeafNode body[]; // corpos
3    private IBarrier specialLatch;
4    private IBarrier barrier;
5    private static boolean partitionsDone;
6
7    public NBodyProviderTask(){
8      partitionsCondition = newCondition(new ConditionCode() {
9        public Boolean call() {
10         return partitionsDone;
11       }
12     });
13     barrier = createBarrier();
14     specialLatch = createBarrier();
15     specialLatch.register();
16   }
17
18   [...] //Funcao auxiliar
19
20   @Task
21   public void runSimulation(int ntimesteps, double dtime, double eps, double tol,
         double dthf, double epssq, double itolsq){
22     for (int i = 0; i < numberOfTasks; i++) {
23       int[] bounds = getLowerAndUpperBoundsFor(i, numberOfTasks);
24       this.computePartition(bounds[0], bounds[1]);
25     }
26
27     [...] // Para cada passo, e criada uma nova oct-tree e enviada para os
           trabalhadores
28   }
29
30   private int[] getLowerAndUpperBoundsFor(int count, int numberOfTasks) {
31     int[] results = new int[2];
32     int increment = body.length / numberOfTasks;
33     results[0] = count * increment;
34     results[1] = (count + 1 == numberOfTasks) ?
35           body.length :
36           (count + 1) * increment;
37     return results;
38   }
39
40   @Task
41   public void computePartition(int lowerBound, int upperBound){
42     barrier.register();
43     for (int step = 0; step < ntimesteps; step++) {
44       specialLatch.await();
45
46       // Calcula a forca primeiro
47       for (int i = lowerBound; i < upperBound; i++)
48         body[i].computeForce(root, diameter, itolsq, step, dthf, epssq);
49
50       barrier.await();
51
52       // Calcula a posicao quando todas as forcas forem calculadas
53       for (int i = lowerBound; i < upperBound; i++)
54         body[i].advance(dthf, dtime);
55
56       barrier.await();
57
58       // Notifica o fio de execucao principal que ja acabou a iteracao
59       partitionsDone = true;
60       partitionsCondition.condNotifyAll();
61     }
62   }
63 }
```



Listagem F.2: Classe NBodyProviderTask

```
public class NBodyProviderRangeDist extends ServiceProvider implements NBodyService
    {

  [...] // Campos e Construtor

  @Task
  public void runSimulation(int ntimesteps, double dtime, double eps, double tol,
      double dthf, double epssq, double itolsq){
    this.computePartition(new Integer[]{0, body.length});

    [...] // Para cada passo, e criada uma nova oct-tree e enviada para os
        trabalhadores
  }

  @DistRedTask
  private void computePartition(@DistributionPolicy(distribution=NBodyDistRange.
      class) Integer range[]) {
    barrier.register();
    for (int step = 0; step < ntimesteps; step++) {
      specialLatch.await();

      // Calcula a forca primeiro
      for (int i = range[0]; i < range[1]; i++)
        body[i].computeForce(root, diameter, itolsq, step, dthf, epssq);

      barrier.await();

      // Calcula a posicao quando todas as forcas forem calculadas
      for (int i = range[0]; i < range[1]; i++)
        body[i].advance(dthf, dtime);

      barrier.await();

      // Notifica o fio de execucao principal que ja acabou a iteracao
      partitionsDone = true;
      partitionsCondition.condNotifyAll();
    }
  }
}
```

Listagem F.3: Classe NBodyProviderRangeDist

```
1  public class NBodyDistRange implements Distribution<Integer[]> {
2    private int n_parts;
3
4    [...] //Construtor
5
6    public void setPartitions(int length) {
7      this.n_parts=length;
8    }
9
10   public Integer[][] distribution() {
11     Integer[][] out = new Integer[n_parts][2];
12     for (int i = 0; i < out.length; i++) {
13       int increment = body.length / n_parts;
14       out[i][0] = i * increment;
15       out[i][1] = (i + 1 == n_parts) ?
16         body.length :
17           (i + 1) * increment;
18     }
19     return out;
20   }
21 }
```

Listagem F.4: Classe NBodyDistRange

```
1  public class NBodyProviderBodyDist extends ServiceProvider implements NBodyService{
2
3    [...] // Campos e Construtor
4
5    @Task
6    public void runSimulation(int ntimesteps, double dtime, double eps, double tol,
         double dthf, double epssq, double itolsq){
7      this.computePartition(body);
8
9      [...] //  Para cada passo, e criada uma nova oct-tree e enviada para os
           trabalhadores
10   }
11
12   @DistRedTask
13   private void computePartition(@DistributionPolicy(distribution=NBodyDistBody.
         class) OctTreeLeafNode body[]){
14     barrier.register();
15     while (int step = 0; step < ntimesteps; step++) {
16       specialLatch.await();
17
18       // Calcula a forca primeiro
19       for (OctTreeLeafNode octTreeLeafNode : body)
20         octTreeLeafNode.computeForce(root, diameter, itolsq, step, dthf, epssq);
21
22       barrier.await();
23
24       // Calcula a posicao quando todas as forcas forem calculadas
25       for (OctTreeLeafNode octTreeLeafNode : body)
26         octTreeLeafNode.advance(dthf, dtime);
27
28       barrier.await();
29
30       // Notifica o fio de execucao principal que ja acabou a iteracao
31       partitionsDone = true;
32       partitionsCondition.condNotifyAll();
33     }
34   }
35 }
```

Listagem F.5: Classe NBodyProviderBodyDist

```
public class NBodyDistBody implements Distribution<OctTreeLeafNode[]> {

  private int n_parts;

  [...] //Construtor

  public void setPartitions(int length) {
    this.n_parts=length;
  }

  public OctTreeLeafNode[][] distribution() {
    OctTreeLeafNode[][] out= new OctTreeLeafNode[n_parts][body.length/n_parts];
    int aux=0;
    for (int i = 0; i < n_parts; i++) {
      for (int j = 0; j < body.length/n_parts; j++) {
        out[i][j] = body[aux];
        aux++;
      }
    }
    return out;
  }
}
```

Listagem F.6: Classe NBodyDistBody

```
public class NBodyClusterProvider extends ServiceProvider implements NBodyService{
  private ComputePartitionService p;

  [...] //Construtor

  @Task
  public void runSimulation(int ntimesteps, double dtime, double eps, double tol,
      double dthf, double epssq, double itolsq) {

    for (int step = 0; step < ntimesteps; step++) {
      computeCenterAndDiameter();

      // Cria a raiz da arvore
      OctTreeInternalNode root = OctTreeInternalNode.newNode(centerx, centery,
          centerz);
      double radius = diameter * 0.5;
      for (int i = 0; i < body.length; i++) {
        root.insert(body[i], radius);
      }
      curr = 0;
      curr = root.computeCenterOfMass(body, curr);
      body=p.computeForce(body,root,diameter,itolsq,step,dthf,epssq);
      body=p.advance(body,dthf,dtime);
    }
  }
}
```

Listagem F.7: Classe NBodyClusterProvider

```
public interface ComputePartitionService extends Service {

  @Task
  OctTreeLeafNode[] computeForce(OctTreeLeafNode[] body, OctTreeInternNode root,
      double diameter, double itolsq, int step, double dthf, double epssq);

  @Task
  OctTreeLeafNode[] advance(OctTreeLeafNode[] body, double dthf, double dtime);
}
```

Listagem F.8: Interface ComputePartitionService

```
public class ComputePartitionProvider extends ServiceProvider implements
    ComputePartitionService{

  @ReductionPolicy(reduction=NBodyClusterReduction.class,params="body.length")
  public OctTreeLeafNode[] computeForce(@DistributionPolicy(distribution=
      NBodyDistBody.class) OctTreeLeafNode[] body,OctTreeInternalNode root, double
      diameter, double itolsq, int step, double dthf, double epssq) {

     for (OctTreeLeafNode octTreeLeafNode : body) {
      if (octTreeLeafNode!=null)
       octTreeLeafNode.computeForce(root, diameter, itolsq, step, dthf, epssq);
    }
    return body;
  }

  @ReductionPolicy(reduction=NBodyClusterReduction.class,params="body.length")
  public OctTreeLeafNode[] advance(@DistributionPolicy(distribution=NBodyDistBody.
      class) OctTreeLeafNode[] body, double dthf, double dtime) {
    for (OctTreeLeafNode octTreeLeafNode : body) {
      if(octTreeLeafNode!=null)
        octTreeLeafNode.advance(dthf, dtime);
    }
    return body;
  }

}
```

Listagem F.9: Interface ComputePartitionProvider

```
public class NBodyClusterReduction implements
             Reduction<OctTreeLeafNode[], OctTreeLeafNode[]> {
  private int body_size;

  [...] //Construtor

  public OctTreeLeafNode[] getResult(Iterator<OctTreeLeafNode[]> res) {
    OctTreeLeafNode[] out = new OctTreeLeafNode[body_size];
    int i=0;
    for (OctTreeLeafNode[] node : res) {
      int j;
      for (j = node.length-1; j >=0; j--)
        if (node[j] != null)
          break;
      j++;
      System.arraycopy(node, 0, out, i, j);
      i+=j;
    }
    return out;
  }
}
```

Listagem F.10: Classe NBodyClusterReduction

# Código do problema da Concordância

```
public interface ConcordService extends Service {
  @Task
  ConcurrentHashMap<String, ConcurrentLinkedQueue<Integer>>
     createConcordance(Region r);
}
```

Listagem G.1: Interface ConcordService

```
public class ConcordProvider extends ServiceProvider implements ConcordService {
  private ArrayList<String> text;
  private int n;
  private ConcurrentHashMap<String,ConcurrentLinkedQueue<Integer>> concordance;

  [...] //Construtor

  @DistRedTask
  public ConcurrentHashMap<String, ConcurrentLinkedQueue<Integer>>
      createConcordance(@DistributionPolicy(distribution=ConcordDist.class) Region
      r) {
    int index = r.startIndex;
    int endIndex = r.endIndex;

    [...] // Trabalho atribuido aos threads na implementacao de referencia

    return concordance;
  }
}
```

Listagem G.2: Classe ConcordProvider

```
public class ConcordDist implements Distribution<Region> {
  private int n_parts;

  [...] //Construtor

  public void setPartitions(int length) {
    this.n_parts=length;
  }

  public Region[] distribution() {
    Region[] out = new Region[n_parts];
    int chunkSize = ((r.endIndex-r.startIndex)+n_parts) / n_parts;

    for (int i = 0; i < n_parts; i++) {
      int startIndex = (chunkSize * i)+r.startIndex;
      int endIndex = (chunkSize * (i + 1))+r.startIndex;
      out[i] = new Region(startIndex, endIndex);
    }
    return out;
  }
}
```

Listagem G.3: Classe ConcordDist

```
public class ConcordClusterProvider extends ServiceProvider implements
    ConcordService {

  @DistRedTask
  @ReductionPolicy(reduction=ConcordClusterRed.class)
  public Map<String, ConcurrentLinkedQueue<Integer>> createConcordance(
      @DistributionPolicy(distribution=ConcordClusterDist.class) Partition p) {
     ConcurrentHashMap<String, ConcurrentLinkedQueue<Integer>> concordance = new
         ConcurrentHashMap<String, ConcurrentLinkedQueue<Integer>>(p.text.size()
         /10,0.75f);

    int index = 0;
    int endIndex = p.text.size();

    [...] // Trabalho atribuido aos threads na versao de referencia

    return concordance;
  }
```

Listagem G.4: Classe ConcordClusterProvider

```
public class ConcordClusterDist implements Distribution<Partition> {
  Partition p;
  int n_parts;

  [...] //Constructor

  public void setPartitions(int length) {
    this.n_parts=length;
  }

  public Partition[] distribution() {
    Partition[] out = new Partition[n_parts];
    int chunkSize = (p.text.size()+n_parts) / n_parts;

    for (int i = 0; i < n_parts; i++) {
      int startIndex = (chunkSize * i);
      int endIndex = (chunkSize * (i + 1));
      endIndex += p.n;
      if(endIndex > p.text.size())
        endIndex = p.text.size();
      ArrayList<String> part = new ArrayList<String>(endIndex-startIndex);
      for (int j = startIndex; j < endIndex; j++)
        part.add(p.text.get(j));
      out[i] = new Partition(startIndex+p.startIndex, part, p.n);
    }
    return out;
  }
}
```

Listagem G.5: Classe ConcordClusterDist

```
public class ConcordClusterRed implements
  Reduction<Map<String, ConcurrentLinkedQueue<Integer>>,
            Map<String, ConcurrentLinkedQueue<Integer>>> {

  public Map<String, ConcurrentLinkedQueue<Integer>>
      getResult(Iterator<Map<String, ConcurrentLinkedQueue<Integer>>> res) {

    Map<String, ConcurrentLinkedQueue<Integer>> out = null;

    while (res.hasNext()) {
      Map<String, ConcurrentLinkedQueue<Integer>> map =  res.next();
      if (out==null) {
        out = map;
        continue;
      }
      for (Entry<String, ConcurrentLinkedQueue<Integer>> e : map.entrySet()) {
        if (!out.containsKey(e.getKey()))
          out.put(e.getKey(), e.getValue());
        else{
          ConcurrentLinkedQueue<Integer> q=out.get(e.getKey());
          for (Integer integer : e.getValue()) {
            if (!q.contains(integer))
              q.add(integer);
          }
        }
      }
    }
    return out;
  }
}
```

Listagem G.6: Classe ConcordClusterRed

# Acrónimos

## A

**APGAS** *Asynchronous Partitioned Global Address Space*
**API** *Application Programming Interface*
**APT** *Annotation Processing Tool*
**ARMCI** *Aggregate Remote Memory Copy*
**AST** *Abstract Syntax Tree*

## C

**CAT** Camada de Abstração da Tecnologia
**CHAPEL** *Cascade High Productivity Language*
**CPU** *Central processing unit*
**CSP** *Communicating Sequential Processes*
**CTA** *Compiler Tree API*
**CUDA** *Compute Unified Device Architecture*

## D

**DMA** *Direct Memory Access*
**DMR** *Distributed-Map-Reduce*
**DSM** *Distributed shared memory*
**DSO** *Distributed shared object*

## F

**FIFO** *First In, First Out*
**FPGA** *Field-programmable gate array*

## G

**GAS** *Global Addressing Space*
**GASNET** *Global-Addressing-Space Networking*
**GPGPU** *General-Purpose Graphics Processing Units*
**GPU** *Graphics Processing Units*

## H

**HEC** *High-End Computing*
**HPC** *High Performance Computing*

## I

**I/O** *Input/Output*
**IDEA** *International Data Encryption Algorithm*
**IP** *Internet Protocol*

## J

**JGF** *Java Grande Benchmark Suite*
**JNI** *Java Native Interface*
**JVM** *Java Virtual Machine*

## L

**LAPI** *Low-level application programming interface*

## M

**MOM** *Message Oriented Middleware*
**MPI** *Message Passing Interface*
**MPMD** *Multiple Program Multiple Data*
**MPP** *Massive parallel processing*

## N

**NIO** *New Input/Ouput*
**NUCC** *Non-Uniform Cluster Computing*
**NUMA** *Non-Uniform Memory Access*

## P

**PE** *Processing Elements*
**PGAS** *Partitioned Global Address Space*
**PVM** *Parallel Virtual Machine*

## R

**RMA** *Remote Memory Access*
**RMI** *Remote Method Invocation*
**RPC** *Remote Procedure Call*
**RTT** *Round Trip Time*

## S

**SICSA** *Scottish Informatics and Computer Science Alliance*
**SMP** *Symmetric Multiprocessing*
**SOA** *Service Oriented Architecture*
**SPMD** *Single Program Multiple Data*
**SSE** *Streaming SIMD Extensions*
**SSI** *Single System Image*

## T

**TCP** *Transmission Control Protocol*

## U

**UDP** *User Datagram Protocol*
**UPC** *Unified Parallel C*
**UUID** *Universally Unique IDentifier*

## V

**VAPPF** *Virtualized Self-Adaptive Heterogeneous High Productivity Computers Parallel Programming Framework*

## X

**XML** *Extensible Markup Language*