Diogo André Ribeiro Mourão

Licenciado em Engenharia Informática

Um Middleware Independente da Plataforma para Computação Paralela

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

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

Júri:

Presidente:	Prof. Doutor Vasco Miguel Moreira Amaral
Arguente:	Prof. Doutor Luís Manuel Antunes Veiga
Vogal:	Prof. Doutor Hervé Miguel Cordeiro Paulino

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2011

Um Middleware Independente da Plataforma para Computação Paralela

Para os meus pais.

Agradecimentos

Quero expressar a minha gratidão a todas as pessoas que, diretamente ou indiretamente, contribuíram para tornar esta dissertação possível.

Em primeiro lugar, quero agradecer ao meu orientador Professor Doutor Hervé Paulino por todo o apoio e orientação desde o primeiro dia. Esteve sempre disponível para me ajudar e teve sempre uma grande paciência para fazer todas as revisões à escrita da dissertação que foram efetuadas que ajudaram a melhorar a sua qualidade.

A todos os meus colegas e amigos de curso. Em especial ao Tiago Vale que partilhou comigo a maioria dos momentos vividos por mim na faculdade. Também ao Henrique Azevedo, Francisco Salvador, Bruno Preto, João Saramago, Ricardo Silva, Vítor Pereira, Adérito Batista, entre outros, que tive o prazer de conhecer e que considero meus amigos.

Aos meus pais agradeço todo o apoio e carinho que me deram e os valores que me transmitiram durante toda a minha educação. Também agradeço à restante família que sempre se preocupou comigo, sempre dando apoio e incentivo.

À Daniela Machado por todo o amor, carinho e apoio demonstrado ajudando-me sempre a ultrapassar todas as dificuldades.

Resumo

A adoção generalizada dos processadores com vários núcleos (*multi-core*) requer modelos de programação que permitam expressar paralelismo de uma forma simples, sem expor detalhes de baixo nível no que se refere à gestão da concorrência.

No entanto, apesar dos processadores *multi-core* se terem tornado o *standard* de-facto desde o ano de 2006, as linguagens de programação de uso generalizado e respetivos compiladores e sistemas de execução permanecem, na sua essência, inalterados. Este facto reflete-se não só ao nível do desempenho das aplicações como também ao nível de produtividade do seu desenvolvimento. É então necessário criar novas soluções que ofereçam abstrações de alto nível que permitam expressar o paralelismo de uma forma simples, ao mesmo tempo que permitam separar a lógica da aplicação da gestão da concorrência.

Neste contexto, esta dissertação propõe um *middleware* independente da plataforma que tem como objetivo o suporte à execução de aplicações paralelas fornecendo as funcionalidades mais comuns, tais como: paralelismo de tarefas e de dados, comunicação e controlo de concorrência. Pretende ser suficientemente genérico de forma a que possa ser utilizado como suporte ao desenvolvimento de uma grande variedade de aplicações concorrentes e paralelas, bem como servir de suporte a sistemas de execução de linguagens de programação. A sua arquitetura é inspirada na arquitetura dos sistemas de operação, na medida em que, para além de oferecer uma interface bem definida para o programador, centrada no conceito de localidade, também especifica uma interface, baseada em *drivers*, para o suporte de várias implementações das funcionalidades necessárias. De forma a simplificar o desenvolvimento de aplicações foi desenvolvido um mecanismo de anotações permitindo ao programador expressar o paralelismo nas suas aplicações com recurso às anotações oferecidas.

O *middleware* está atualmente concretizado para arquiteturas de memória partilhada apesar do seu desenho contemplar a sua extensão para arquiteturas de memória distribuída sobre a qual já existe trabalho em curso. A implementação atual é avaliada em

termos de desempenho através dos *benchmarks* NAS Parallel Benchmarks [NAS] e Java Grande Benchmark Suite [EPC]. Os resultados obtidos confirmam a existência de um *overhead* associado à utilização do *middleware*, como era expectável antes da realização deste estudo. Porém, este é compensado pelo facto de se oferecerem abstrações que simplificam o desenvolvimento de aplicações paralelas.

Palavras-chave: Programação Paralela, *Middleware*, Arquiteturas *multi-core*

Abstract

The widespread adoption of multi-core processors requires programming models that allow to express parallelism in a simple way, without exposing low level details about concurrency management.

Nevertheless, despite multi-core processors becoming the de-facto standard since 2006, commonly used programming languages and their respective compilers and runtime systems remain the same. This is reflected not only at the applications performance level but also at their development productivity. As such, it is then desirable to create new solutions that offer high level abstractions allowing to express parallelism in a simple way, and to separate the application logic from the concurrency management at the same time.

In this context, this thesis proposes a independent middleware platform to support parallel applications execution by providing common functionality such as: task and data parallelism, communication and concurrency control. It aims to be sufficiently generic to be able to support the development of a great variety of concurrent and parallel applications, as well as supporting programming languages runtime systems. Its architecture is inspired in that of operating systems, in the sense that, not only having a well defined interface to the programmer, based on places, it also defines an interface, based on driver concept, to support multiple implementations of the needed functionalities to execute parallel applications. In an effort to ease application development, an annotation mechanism is available which allows the programmer to express parallelism in his applications by just annotating the source code.

Our middleware already implements support for shared memory architectures but its design allows future support for distributed memory architectures, to which work is already being done. The present implementation is evaluated performance wise through the NAS Parallel Benchmarks [NAS] and Java Grande Benchmark Suite [EPC] benchmarks. The results confirm that using our middleware incurs in some overhead, as expected. However, it is compensated by the abstractions offered which greatly simplify

parallel application development.

Keywords: Parallel Programming, Middleware, Multi-core architectures

Conteúdo

Lista de Figuras

Lista de Tabelas

Listagens

1

Introdução

1.1 Motivação

As áreas da programação concorrente e paralela têm sido alvo de particular interesse com a generalização dos processadores *multi-core*, cada vez mais o paradigma arquitetural de eleição dos computadores pessoais. Tal mudança tem consequências ao nível do desenvolvimento de aplicações, pois apenas as aplicações concorrentes aproveitam verdadeiramente todo o potencial oferecido por estas arquiteturas.

A lei de Moore [Moo65] tem sido comprovada ao longo das últimas décadas, com o número de transístores por *chip* a crescer exponencialmente. Esse conjunto de novos transístores tem sido utilizado para aumentar o desempenho dos processadores, o que, até há pouco tempo, se traduzia em grande medida no aumento da velocidade do relógio. Consequentemente, cada nova geração de processadores executava as aplicações sequenciais mais rapidamente. No entanto, devido essencialmente a limitações físicas, como o aumento da energia sob a forma de calor, ao consumo de energia, entre outros, o aumento da velocidade do relógio tornou-se incomportável como fator de aumento do desempenho dos processadores, como pode ser observado no gráfico da Figura 1.1.

Foi então necessário encontrar outras tecnologias para aumentar o seu desempenho. Desta forma, surgiram os processadores *multi-core*, que consistem em integrar, no mesmo chip, várias unidades de processamento independentes habitualmente designadas por núcleos (*cores*).

As aplicações com um único fluxo de execução não aproveitam verdadeiramente de todo o potencial oferecido por este tipo de processadores porque a sua execução ocorre num só núcleo. Além disso, não beneficiam do aumento de desempenho destes processadores pois este é obtido a partir da adição de núcleos, sendo que o desempenho dos

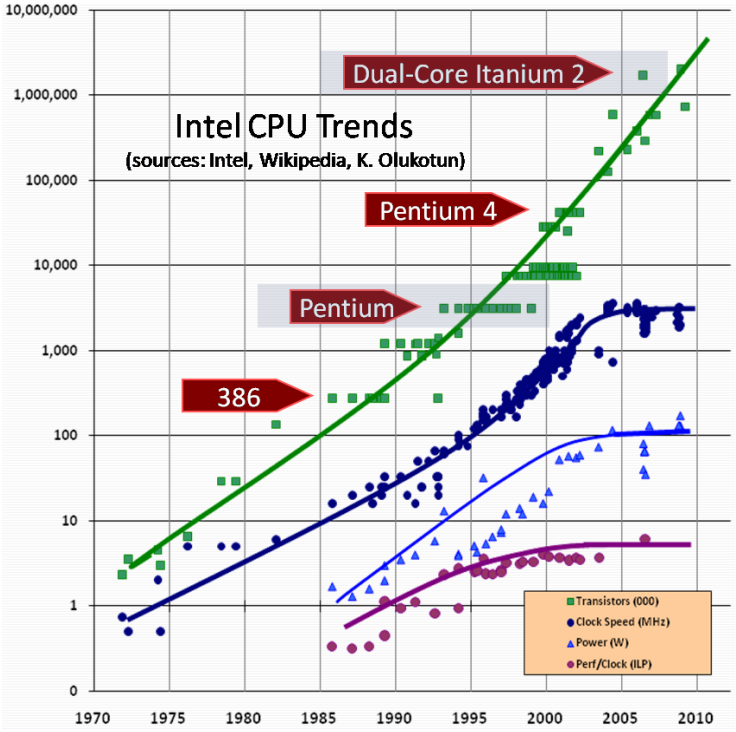


Figura 1.1: Relação entre número de transístores com a velocidade do relógio [Sut05].

núcleos por si só, muitas vezes, não aumenta. A arquitetura destes processadores impõe então que as aplicações sejam desenvolvidas de forma a que dividam o seu trabalho pelos vários núcleos, de forma a beneficiar efetivamente da natureza paralela do hardware. Desta forma, apenas as aplicações que fazem uso da concorrência usufruem totalmente das vantagens dos processadores *multi-core* e por essa razão é necessário que cada vez mais as aplicações sejam desenvolvidas tendo em conta essa dimensão.

Porém, a construção de aplicações paralelas acarreta uma complexidade acrescida quando comparada com o desenvolvimento sequencial habitual. Além da lógica da aplicação, é necessário também ter em consideração os seguintes desafios [MW08]:

1. Encontrar possível paralelismo numa aplicação. A dificuldade surge em como dividir o trabalho em tarefas, ou dividir uma estrutura de dados, para que possam ser executadas em paralelo;
2. Escalonar as tarefas nos processadores disponíveis no sistema;
3. Aproximar os dados dos processadores para uma maior rapidez de execução. Esta propriedade é designada por princípio da localidade;
4. Suportar escalabilidade, tanto por parte de hardware como de software;
5. Utilização de mecanismos de sincronização de baixo nível, evitando *deadlocks* e conflitos de acesso a zonas de memória partilhada;

6. Recuperação de erros e tolerância a falhas;
7. Suporte para boas práticas de engenharia de software: paralelismo incremental, reutilização de código, etc.;
8. Suportar a portabilidade de uma aplicação para várias arquiteturas paralelas, não diminuindo o seu desempenho, utilizando os modelos de programação mais adequados;

A complexidade do processo de depuração também é incrementada devido ao seu comportamento não determinístico. No entanto, apesar de todas estas nuances, a natureza do hardware atualmente disponível nos computadores pessoais obriga à sua utilização. Consequentemente, para diminuir a carga sobre o programador, os suportes existentes terão que, idealmente, oferecer mecanismos de alto nível para a gestão da concorrência por parte do programador.

1.2 Descrição do Problema e Objectivos do Trabalho

Apesar dos processadores *multi-core* terem-se afirmado como o *standard* de-facto desde o ano de 2006, as linguagens de programação de uso generalizado e os respetivos compiladores e sistemas de execução permanecem, na sua essência, inalterados. Os modelos de programação existentes pecam pela ausência de construtores de alto nível que permitam expressar paralelismo de forma simples e eficiente, e as técnicas de compilação atuais não conseguem extrair a informação necessária para realizar as otimizações necessárias. Os modelos existentes exigem que o programador seja confrontado com aspetos que não estão apenas relacionados com as funcionalidades das aplicações, como a utilização e gestão de múltiplos *locks*. Isto expõe o programador ao erro, resultando numa tarefa árdua, em sistemas complexos, para tornar as aplicações corretas. Ao longo dos últimos anos, vários modelos de programação foram propostos, no entanto, nenhum destes modelos conquistou por completo a comunidade científica.

É portanto necessário criar novas soluções que ofereçam níveis de abstração elevados que permitam expressar paralelismo de uma forma simples, ao mesmo tempo que permitam separar a lógica funcional da aplicação da gestão da concorrência. Dessa forma, será possível que a produtividade de aplicações concorrentes e paralelas seja mais elevada do que atualmente é. Este facto torna a área de investigação no desenho e implementação de modelos de programação concorrente e paralela muito importante e com grande impacto.

Uma área a explorar é o desenvolvimento de camadas de software intermédias (*middlewares*) que forneçam abstrações de alto nível que ofereçam todas as funcionalidades necessárias para o desenvolvimento de aplicações concorrentes e paralelas, que possam recorrer a tecnologias já existentes para o suporte dessas funcionalidades. Os *middlewares* existentes para programação paralela [Gen01, FTL+02] concentram-se essencialmente na

submissão global de trabalho em ambientes distribuídos, no âmbito das áreas da computação em *grid* e computação na *cloud* [1]. No entanto, com a proliferação dos processadores *multi-core*, os *middlewares* terão cada vez mais de considerar a computação em arquiteturas *multi-core*, em que cada um dos núcleos, juntamente com a sua *cache*, pode ser visto com um nó numa arquitetura distribuída. Por outro lado, os sistemas de execução para o suporte de linguagens para programação paralela [FLR98, ZSS06, CGS+05, Xin06, HPR+08] estão apenas vocacionados para as linguagens que suportam e portanto não têm como objetivo ser suficientemente genéricos para poderem suportar vários modelos de programação.

Neste contexto, esta dissertação tem como base propor um *middleware* genérico e independente da plataforma para programação paralela, que fornece as funcionalidades mais comuns para o suporte à execução de aplicações concorrentes e paralelas.

1.3 Solução Apresentada

O *middleware* proposto tem como objetivo principal o suporte à execução de aplicações paralelas sobre as arquiteturas paralelas atuais, fornecendo as funcionalidades mais comuns para o seu suporte, nomeadamente: paralelismo de tarefas e de dados, comunicação e controlo de concorrência. O *middleware* pretende ser independente da plataforma permitindo assim que as aplicações sejam portáveis para vários tipos de arquiteturas, tais como arquiteturas de memória partilhada, arquiteturas de memória distribuída e arquiteturas híbridas. O *middleware* tem também como objetivo ser suficientemente genérico de forma a que possa ser utilizado como sistema de execução para o suporte de várias linguagens de programação ou como suporte à execução de aplicações paralelas. Um objetivo do *middleware* é fornecer à comunidade que utiliza o Java como linguagem de programação meios para usufruir das arquiteturas paralelas atualmente existentes.

O *middleware* apresenta-se como uma máquina virtual onde o conjunto de máquinas, que constitui a plataforma de execução, é abstraído do programador através de uma interface bem definida que apresenta uma visão uniforme independentemente da arquitetura alvo. A sua arquitetura é composta por três entidades principais: (i) Interface para as aplicações; (ii) Núcleo do *middleware*; (iii) Interface para múltiplas especializações das funcionalidades fornecidas.

O desenho da sua arquitetura é inspirado na arquitetura dos sistemas de operação, na medida em que, para além de oferecer uma interface bem definida para as aplicações, centrada no conceito de localidade, e de fornecer as funcionalidades necessárias, também oferece uma interface, baseada na noção de *driver*, para a especialização de cada funcionalidade. A implementação (*driver*) de uma funcionalidade permite delegar em tecnologias já existentes a implementação de uma dada funcionalidade. Por exemplo, será possível recorrer ao *standard* MPI para suportar a comunicação por troca de mensagens. Permite

[1] Em [FZRL08] é apresentada uma visão geral das áreas da programação em *grid* e na *cloud* e dos últimos avanços nesta área, nomeadamente na área dos *middlewares*.

também adequar o *middleware* a uma dada arquitetura, podendo-se recorrer a diferentes implementações das funcionalidades consoante a arquitetura subjacente. A abordagem seguida para o seu desenho faz com que o *middleware* atue como um intermediário entre as aplicações e as tecnologias necessárias à especialização de cada uma das funcionalidades disponíveis.

O recurso à noção de localidade para compor a interface para as aplicações permite oferecer uma interface genérica aos programadores. Uma localidade pode ser vista como processador virtual sobre a qual se adjudica trabalho, na forma de tarefas, podendo abstrair, por exemplo, um CPU, um GPU ou uma máquina com vários processadores. A localidade permite dar controlo ao programador da localização da computação, podendo assim gerir a afinidade entre a computação e os recursos computacionais existentes. A gestão da afinidade é um aspeto importante pois é possível ao programador distinguir um acesso remoto de um acesso local, podendo assim otimizar o desempenho. A sua utilização também tem como objetivo apresentar a mesma interface para a programação em arquiteturas de memória partilhada e memória distribuída.

Uma localidade possui uma interface, o seu próprio espaço de endereçamento de memória e um conjunto de *threads* ativos que executam as tarefas. Durante a execução das tarefas, os *threads* executam concorrentemente e podem aceder ao espaço de endereçamento da localidade. Desta forma, a comunicação entre si é realizada sob o paradigma de memória partilhada. Uma localidade pode então ser vista como um processador virtual com a sua memória privada.

As aplicações são mapeadas sobre um conjunto de localidades, fazendo com que cada aplicação seja composta por um conjunto de localidades. Estas interagem entre si para submeter trabalho realizando invocações sobre as operações especificadas na sua interface. Adicionalmente, a interface de uma localidade pode ser estendida de forma a oferecer novas operações especificadas pelo programador, comportando-se assim como um objeto sobre o qual se podem invocar operações.

As localidades presentes numa computação estão distribuídas pelos vários nós (máquinas físicas), sendo que cada nó executa uma instância do *middleware* que suporta um subconjunto dessas localidades. O núcleo do *middleware* é responsável pela gestão do trabalho adjudicado nas localidades, pela comunicação entre as localidades, por oferecer mecanismos de sincronização entre tarefas a executar numa localidade, pela gestão da memória partilhada e pelo balanceamento da carga. A Figura 1.2 ilustra a visão geral da infraestrutura *middleware*.

Com o objetivo de simplificar a tarefa do programador no desenvolvimento de aplicações paralelas, foi desenvolvido um conjunto de anotações que têm como base a filosofia seguida pelo OpenMP [CJP07]. O objetivo é permitir ao programador anotar o código das suas aplicações de forma a expressar o paralelismo. As anotações atualmente existentes permitem ao programador especificar os métodos que cuja execução pode ser realizada em paralelo, os métodos que cuja execução tenha que ser executada de forma atómica e os argumentos passados numa invocação que devem ser passados por cópia.

Figura 1.2: Visão geral da infraestrutura *middleware*.

O volume de trabalho necessário para desenhar e implementar por completo esta arquitetura não se coaduna com o âmbito e prazos definidos para uma dissertação de mestrado. Assim sendo, esta dissertação tem como objetivo definir o desenho geral do *middleware* e da sua interface e implementá-lo para arquiteturas de memória partilhada. A realização desta última requer uma implementação da lógica do *middleware* e de uma base da camada de *drivers* para o seu suporte neste tipo de arquiteturas, como arquiteturas *multi-core*.

1.4 Contribuições previstas

As contribuições desta dissertação são:

- Desenho de um *middleware* genérico e independente da plataforma que fornece as funcionalidades necessárias para o suporte à execução de aplicações paralelas com as seguintes características:
  - Suporte a vários modelos de programação;
  - Oferecer uma interface bem definida centrada no conceito de localidade. A interface é composta por um conjunto de localidades e a sua utilização permite dar ao programador o controlo da localização da computação, podendo desta forma otimizar o desempenho;
  - Camada de *drivers* que permite extensibilidade, utilização de tecnologias já existentes e a sua escolha de acordo com a arquitetura alvo e adequar cada instância do *middleware* de acordo com a arquitetura do nó em que está a executar.
- Implementação do *middleware* para arquiteturas de memória partilhada, o que incluí a implementação base do núcleo e dos *drivers* necessários para a sua execução neste tipo de arquiteturas;
- Implementação de um mecanismo de anotações que suporta um conjunto de anotações que simplificam a tarefa do programador no desenvolvimento das aplicações;

- Construção de uma suite de *benchmarks* utilizada pelo *middleware*:
  - Adequação de alguns *benchmarks* NAS [NAS] e JGF [EPC] recorrendo à interface para as aplicações oferecida;
  - Implementação de mais um conjunto de aplicações que recorrem a Java "puro" e à interface para as aplicações oferecida.
- Realização de uma avaliação de desempenho do *middleware* numa arquitetura *multicore* utilizando os *benchmarks*.

1.5 Estrutura do documento

A preparação da dissertação está dividida nos seguintes capítulos:

- Introdução - Corresponde ao atual capítulo onde se descreve o contexto e as motivações para a realização desta dissertação, o problema colocado, onde se apresenta uma síntese da solução para o problema e as contribuições previstas no fim desta dissertação.
- Trabalho relacionado - Descrevem-se os tópicos relacionados com a área de investigação onde se insere esta dissertação, onde se passa em revista alguns dos modelos de programação concorrente/paralela estudados para a identificação de algumas funcionalidades comuns entre os vários modelos para o desenvolvimento do *middleware*.
- Conceção e Desenho - Descreve-se o desenho da arquitetura do *middleware*. Apresenta a interface para as aplicações e descreve os módulos que integram o núcleo do *middleware*.
- Implementação - Faz-se uma descrição dos detalhes da implementação atual do *middleware*, descrevendo de que forma o mecanismo de anotações foi implementado e que tecnologias foram utilizadas e a implementação de cada módulo e as tecnologias utilizadas.
- Avaliação - Faz-se uma avaliação de desempenho que permitem avaliar a utilização do *middleware* como suporte à execução de aplicações paralelas.
- Conclusões e Trabalho Futuro - Descrevem-se as conclusões finais do trabalho realizado nesta dissertação, sendo também efetuadas algumas sugestões para trabalho futuro.

2

Trabalho relacionado

O trabalho a ser desenvolvido no âmbito desta dissertação centra-se nas áreas da programação concorrente e paralela. Nesse sentido, este capítulo apresenta os tópicos mais relevantes dessas áreas para o contexto desta dissertação. Na Secção 2.1, é descrito o que é a programação concorrente, mencionando os seus desafios e os mecanismos de sincronização e comunicação existentes; na Secção 2.2, é descrito o que é a programação paralela, mencionando os seus desafios e modelos de programação paralela e respetivo suporte em termos de linguagens ou bibliotecas; na Secção 2.3, é feita uma visão e uma descrição dos sistemas de execução existentes para o suporte de alguns dos modelos apresentados; e por fim na Secção 2.4, onde são descritos alguns dos *middlewares* existentes para programação paralela que foram estudados.

2.1 Programação Concorrente

A programação concorrente é um paradigma de programação que permite que vários processos sejam executados progressivamente, no mesmo intervalo de tempo, competindo e partilhando os recursos disponíveis. Os processos concorrentes contêm um ou mais fluxos de execução (*threads*) que são executados progressivamente num certo período de tempo. Os fluxos de execução podem ser executados em qualquer ordem sem alterar o resultado final. Estes podem ser executados num único processador, intercalando blocos de instruções de cada, decidido pelo sistema de operação (escalonamento de processos) ou poderão ser executados simultaneamente em várias unidades de processamento. Para que seja possível cooperar os fluxos de execução precisam, de alguma forma, de comunicar e sincronizar entre si.

Por motivos de simplicidade, a partir daqui, será assumido que o *thread* é a unidade

básica de concorrência.

2.1.1 Desafios

Os principais desafios para a cooperação de *threads* numa aplicação concorrente são assegurar a comunicação entre *threads* para a troca de informação e a sincronização para evitar conflitos no acesso a dados partilhados ou no controlo de execução.

Comunicação A comunicação entre *threads* é importante para que estes troquem informação de forma a atingirem o estado final de uma computação de forma correta e eficiente. A comunicação pode ser realizada através de duas formas: memória partilhada e troca de mensagens. Estes dois modelos de comunicação serão discutidos na Subsecção 2.1.2.

Sincronização A sincronização entre *threads* é fulcral em programação concorrente. Um problema fundamental na programação concorrente é o problema da região crítica. Uma região crítica é um segmento de código presente numa aplicação, em que os *threads* devem executar de forma atómica, garantindo exclusão mútua. Para garantir que apenas um *thread* execute na sua região crítica, é preciso que hajam mecanismos de sincronização. Se a exclusão mútua não for garantida, isto é, se dois ou mais *threads* estiverem a executar na sua região crítica e partilhem dados, poderá levar a inconsistência de dados (*race condition*).

A sincronização não é apenas importante para prevenir conflitos de escrita/leitura. Uma outra forma de sincronização de *threads* é a criação de um *rendezvous*, isto é, criar um ponto da execução em que os múltiplos *threads* em execução fiquem bloqueados até que todos os *threads* concorrentes cheguem a esse ponto de encontro. Assim, existe coordenação sob a forma de sincronização, sem haver necessariamente o problema da região crítica. Os mecanismos de sincronização existentes serão discutidos na Subsecção 2.1.3.

2.1.2 Modelos básicos de comunicação

Memória Partilhada A comunicação entre *threads* pode ser realizada através de uma zona de memória partilhada, física ou lógica, onde esta é concorrentemente acedida por vários *threads* com a intenção de trocarem dados. Para evitar conflitos de acesso, os dados partilhados deverão ser acedidos usando mecanismos de controlo de concorrência, como por exemplo *locks*/semáforos, que serão abordados na Subsecção 2.1.3.

Este modelo é usual em arquiteturas de memória partilhada onde os processadores acedem à mesma memória física, mas no entanto, este modelo também é utilizado em arquiteturas de memória distribuída, desde que haja uma camada de software que simule memória partilhada sobre memória distribuída.

A existência de múltiplas memórias *cache* e memórias locais aos processadores, coloca

o problema de como garantir a coerência de *cache* de cada processador. Existem protocolos de coerência de *cache* ao nível de hardware e software.

Troca de Mensagens A comunicação entre *threads* poderá também ser efetuada através de troca de mensagens, onde os *threads* não partilham uma zona de memória, tendo cada um a sua zona de memória privada. Este mecanismo de comunicação é particularmente interessante em sistemas com arquitetura de memória distribuída. No entanto, também pode ser usado em sistemas de memória partilhada. A troca de mensagens pode ser síncrona ou assíncrona, no emissor e no recetor da mensagem. Este modelo poderá garantir confiabilidade, ordenação das mensagens e comunicação multi-ponto.

2.1.3 Modelos básicos de sincronização

Semáforos Um semáforo é um mecanismo de sincronização entre *threads* criado por Edsger W. Dijkstra e é apenas aplicável quando existe memória partilhada. O objetivo do semáforo é disciplinar o acesso a recursos partilhados em exclusão mútua ou para sincronização utilizando variáveis de condição. Um semáforo é definido por uma variável especial não negativa partilhada por todos os *threads* concorrentes que é manipulada através de duas operações atómicas: *acquired* (ou simplesmente P) e *release* (ou simplesmente V). Estas duas operações são apresentadas na Figura 2.1.

Algoritmo: P()
se $permits > 0$ **então**
| $permits \leftarrow permits - 1$
senão
| $wait()$

Algoritmo: V()
$permits++$

Figura 2.1: Semáforo

Os semáforos binários são usados para garantir exclusão mútua no acesso a um determinado recurso. Para além dos semáforos binários, existem semáforos em que a variável possa atingir valores maiores que um. Estes são particularmente interessantes para que dois ou mais *threads* possam adquirir um determinado recurso.

O semáforo é, tipicamente, um mecanismo de sincronização sem espera ativa. Bloquear um *thread* significa, normalmente, retirá-lo do estado de execução e colocá-lo numa fila de espera, utilizando uma operação atómica designada por *wait()*. Os *threads* inseridos na fila de espera aguardam até que haja outro *thread* que os notifique de um evento, desta forma passam da fila de espera do sistema de operação para a fila de *threads* prontos do sistema de operação para serem executados pelo CPU. Sendo este um mecanismo de baixo nível, o seu uso em sistemas de grande dimensão dificulta a garantida das propriedades desejadas na sincronização. Entre os erros possíveis podem estar a origem de *deadlocks* ou a não garantia da execução em exclusão mútua.

Monitores Um monitor é uma construção de sincronização de alto nível que foi desenvolvido para que as sincronizações fossem programadas mais facilmente e menos suscetíveis a erros. Sendo os semáforos mecanismos de baixo nível, a probabilidade de erro em sistemas de grande dimensão é muito alta. Desta forma era necessário criar uma estrutura/objeto que fosse capaz de encapsular a garantia de correção da sincronização entre *threads*. É um mecanismo particularmente interessante nas linguagens orientadas a objetos pois nestas linguagens o monitor é visto como uma classe que encapsula dados e métodos para a sincronização. O monitor garante que apenas um *thread* está ativo dentro do monitor, ou seja, garante exclusão mútua, implícita, no acessos aos dados pertencentes ao monitor. No entanto, em algumas aplicações concorrentes é necessário indicar explicitamente em que condições os *threads* podem executar dentro do monitor. Estas condições são denominadas por variáveis de condição ou eventos.

Locks Um *spinlock* é um mecanismo de sincronização semelhante a um semáforo binário, mas com a diferença de que os *threads* ficam bloqueados em espera ativa, ao contrário do que acontece usando semáforos, onde os *threads* são inseridos numa fila de espera. Os *locks* garantem exclusão mútua e são particularmente interessantes em linguagens orientadas a objetos, pois é possível indicar qual o dono do *lock* e assim é possível fazer *lock* recursivo (poder fazer duas vezes seguidas *lock()*, sem haver *deadlock*). Este tipo de *lock* recursivo denomina-se de *lock re-entrante*.

Barreiras Uma barreira é um mecanismo de sincronização onde os *threads* se encontram num *rendezvous* coletivo. A ideia é criar um ponto de sincronização no programa onde um *thread* fica bloqueado nesse ponto até todos os outros *threads* concorrentes atingirem essa barreira. Uma barreira é implementada recorrendo a um inteiro que será um contador do número de chegadas até à barreira e por dois semáforos, um para garantir que o contador é incrementado em exclusão mútua e o outro semáforo é utilizado para bloquear os *threads* até que todos cheguem.

2.2 Programação Paralela

A programação paralela é um subconjunto da programação concorrente em que os *threads* são executados em simultâneo. Dois *threads* que são executados em paralelo demorarão, em teoria e no limite, metade do tempo do que se forem executados sequencialmente. Assim, um programa desenhado de forma a ter em conta os desafios da programação concorrente para sistemas mono-processador, correrá sem alterações em sistemas multi-processador ou *multi-core*.

Um programa paralelo é escrito em função do tipo de arquitetura em que este será executado. Flynn propôs em 1972 uma taxonomia para classificação de arquiteturas de computador [Fly72], em que classifica as arquiteturas em quatro categorias, como está retratado na Tabela 2.1.

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

Tabela 2.1: Taxonomia de Flynn

SISD Refere-se às arquiteturas de computadores com um único *thread* que opera sobre um único conjunto de dados, nomeadamente aos computadores mono-processador ou computadores sequenciais.

SIMD Refere-se às arquiteturas de computadores em que uma mesma instrução é executada em várias unidades de processamento em simultâneo e que operam sobre um conjunto de dados diferente. Estas máquinas exploram uma forma de paralelismo denominada por paralelismo nos dados. As GPUs[1] utilizam este modelo de execução, onde um conjunto de núcleos do GPU, a dado momento, executa uma mesma instrução, mas cada um opera sobre um conjunto diferente de dados.

MISD Refere-se às arquiteturas de computadores com vários processadores em que executam instruções diferentes mas sobre um mesmo conjunto de dados. No entanto, não existem exemplos concretos de computadores com esta arquitetura.

MIMD Refere-se ao tipo mais comum de arquiteturas de computadores paralelos. Os computadores incluídos nesta categoria são constituídos por vários processadores e que operam sobre um conjunto de dados diferente. Nesta categoria inserem-se as arquiteturas de: memória partilhada (onde os computadores *multi-core* se inserem) e memória distribuída.

2.2.1 Desafios

Os desafios da programação paralela engloba também os desafios da programação concorrente pois ela é uma sub-categoria da programação concorrente. Para além de ter em conta os desafios da programação concorrente, também existe o desafio da partição e distribuição dos dados/tarefas pelos múltiplos *threads*. O desafio é encontrar todas as porções da execução que possam ser paralelizáveis e escaloná-las nas unidades de processamento disponíveis da forma mais eficiente possível.

Apesar do paralelismo permitir que uma aplicação, que seja possível de paralelizar, seja executada mais rapidamente do que se fosse executada sequencialmente, o ganho de desempenho por dividir o trabalho pelo número de unidades de processamento pode não equivaler a um ganho linear, isto é, quantos mais processadores houverem, mais rápida é a execução. De facto, isto acontece pois existe um *overhead* associado à criação e gestão de *threads*, mudança de contexto quando um *thread* sai de execução para dar lugar

[1] Graphics Processing Unit

a outro, sincronização entre *threads* e contenção no *bus* acesso à memória. Além deste *overhead*, o desempenho está sempre limitado pela parcela sequencial da aplicação. Para determinadas aplicações, devido ao seu perfil, é possível tornar estes custos desprezáveis para o cálculo do desempenho à medida que o volume de dados aumente, mas no entanto esta particularidade não se coloca a todas as aplicações.

2.2.2 Modelos de Programação Paralela

Um modelo de programação paralela especifica como é que os vários *threads* de um programa a serem executados em paralelo comunicam entre si e quais as operações de sincronização disponíveis para a coordenação das respetivas atividades.

Muitos dos modelos de programação existentes utilizam um modelo de execução *Single Program, Multiple Data* (SPMD). Este modelo de execução consiste em executar o mesmo programa em múltiplos processadores de forma independente operando, possivelmente, sobre dados distintos. Um modelo de execução mais geral que o modelo SPMD é o modelo de execução *Multiple Program, Multiple Data* (MPMD). Consiste em executar programas diferentes em múltiplos processadores operando, possivelmente, sobre dados distintos.

2.2.2.1 Memória Partilhada

Neste modelo de programação múltiplos *threads* de execução partilham um espaço de endereçamento comum, comunicando através de dados partilhados. Este modelo de programação está muito próximo do modelo de programação sequencial, sendo esta uma vantagem deste modelo. No entanto, este tipo de arquiteturas não é escalável devido à partilha do *bus* de acesso à região de memória. O suporte para este modelo é dado através de bibliotecas, como a biblioteca PosixThreads [But97], mas também pode ser suportado através de um conjunto de diretivas para o compilador, como o OpenMP [CJP07].

PThreads é um padrão POSIX para *threads* tendo uma API especificada para criação e gestão explícita de *threads*. A API fornece igualmente primitivas para sincronização entre *threads*.

O OpenMP é uma API que define um modelo de programação paralela segundo um modelo de *multi-threading*, abstraindo do programador a criação e gestão de múltiplos *threads*, tirando partido de arquiteturas SMP[2]. Ou seja, em comparação com a programação utilizando a biblioteca de POSIXThreads, o OpenMP fornece uma abstração de alto nível para o programador, facilitando o desenvolvimento de aplicações. Consiste num conjunto de diretivas para o compilador da linguagem (pré-processador), numa biblioteca de funções e variáveis de ambiente, que permite explicitar o paralelismo. As diretivas podem ser adicionadas a aplicações sequenciais já existentes escritas em C, C++ ou Fortran, com alterações mínimas, de forma a beneficiar de arquiteturas paralelas de

[2]Multi-processamento Simétrico

memória partilhada, sendo o programador responsável pela deteção do potencial paralelismo indicando-o ao compilador através das diretivas. O programador é responsável por explicitar a sincronização e coordenação entre *threads* de forma a evitar conflitos. É também responsável por garantir a não existência de *deadlock*, visto que, o sistema de execução de suporte não garante a não existência.

O modelo de memória é muito restritivo e não é implementado eficientemente em ambientes distribuídos, pois não tem a noção de separação entre acesso uniforme a memória e acesso não-uniforme a memória. Outra das desvantagens é o facto de as aplicações terem um limite de escalabilidade, decorrente das arquiteturas SMP.

2.2.2.2 Troca de Mensagens

Neste modelo de programação a comunicação entre *threads* é realizada explicitamente, pois não existe qualquer região de memória partilhada, fazendo com que o modelo de programação seja mais complexo relativamente ao modelo de programação em memória partilhada. Porém, estas arquiteturas têm a vantagem de serem escaláveis.

O PVM [GBD+94] consiste num ambiente de execução e numa biblioteca de comunicação bi-direcional via troca de mensagens. Permite criar uma rede de computadores heterogénea para que possam colaborar na execução de aplicações paralelas, onde essa rede é vista como uma única máquina paralela virtual. O programador submete as tarefas a executar nessa máquina virtual e as várias máquinas físicas comunicam entre si utilizando troca de mensagens, onde a comunicação é abstraída do programador. É suportada nas linguagens de programação C, C++ e Fortran.

O MPI [SOHL+98, GLT99] é o *standard* de-facto para programação em arquiteturas de memória distribuída, que surgiu no seguimento do PVM. O código de uma aplicação que utilize MPI tem a vantagem de ser portável, pois existe uma grande variedade de plataformas que suportam a especificação MPI. A computação está organizada num conjunto de *threads* que comunicam entre si trocando mensagens, onde esta comunicação pode ser ponto-a-ponto ou coletiva. Cada *thread* tem a sua zona de endereçamento privada que é inacessível pelos outros *threads* resultando na não existência de uma representação de um apontador remoto na memória, pois um apontador no espaço de endereços do *thread* aponta apenas para memória alocada desse *thread*. O MPI não utiliza obrigatoriamente um modelo de execução SPMD, podendo utilizar um modelo de execução MPMD, porém a maioria das aplicações MPI são aplicações SPMD utilizando paralelismo nos dados. O MPI, tal como o OpenMP, envolve paralelismo explicito deixando ao cargo do programador a forma como dividir o processamento pelos vários nós de computação, dando uma melhor performance à aplicação.

A programação neste modelo é propensa a erros e é difícil de fazer *debug*, originando que o código seja difícil de manter num estado correto. Ao contrário do OpenMP, é mais difícil paralelizar uma aplicação sequencial existente devido às várias alterações que serão precisas para transformar uma aplicação sequencial numa aplicação paralela

distribuída.

2.2.2.3 Paralelismo de tarefas

Paralelismo de tarefas é um modelo de programação paralela em que a computação é dividida em diferentes tarefas que podem ser executadas concorrentemente (Figura 2.2). As tarefas a executar num determinado instante são adicionadas a uma fila de tarefas que é consumida por um conjunto de trabalhadores (usualmente *threads*). Cada trabalhador executa apenas uma tarefa de cada vez e cada tarefa está adjudicada a um só *thread* de cada vez. Para utilizar este modelo de paralelismo é necessário que o programador identifique explicitamente as regiões do código fonte (tarefas) que poderão ser executadas em paralelo.

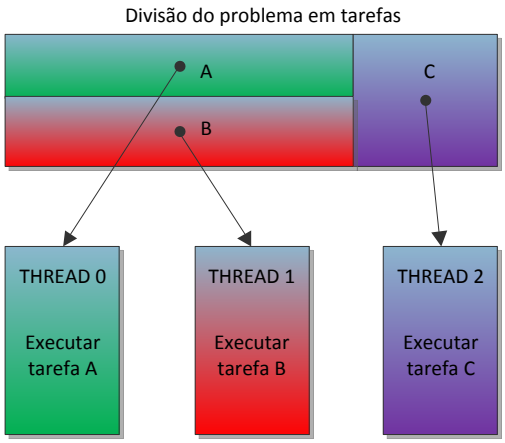


Figura 2.2: Paralelismo de tarefas.

O Cilk [BJK+95] é uma extensão à linguagem C sendo uma das linguagens de referência para paralelismo de tarefas. O Cilk introduz construções de linguagem para criação e sincronização de tarefas concorrentes e num sistema de execução de suporte. O sistema de execução de suporte foi desenhado para executar eficientemente em arquiteturas de memória fisicamente partilhada. O construtor da linguagem spawn *task* cria e adiciona uma tarefa à fila de tarefas a executar. O construtor sync é utilizado para sincronizar os *threads*, resultando numa barreira de sincronização. Uma das grandes contribuições do Cilk para a comunidade foi a introdução de um mecanismo de *work-stealing* (roubo de trabalho) no escalonador. A ideia por detrás do *work-stealing* é o roubo de tarefas, por parte de um *thread* que fica sem trabalho para realizar, das filas privadas dos outros *threads* para executá-las.

Em 2005 foi desenvolvido o JCilk [DLL05] que é uma extensão à linguagem Java com a mesma semântica da linguagem Cilk original para paralelismo. O Cilk original foi abandonado por alguns anos, mas com o surgimento dos processadores *multi-core* voltou

novamente a ser uma linguagem a ter em consideração, devido às suas características. Em consequência, recentemente a Intel Corp. adquiriu os direitos sobre o Cilk e dessa forma criou o Intel Cilk Plus [Corb]. O Intel Cilk Plus é parte integrante do Intel Parallel Building Blocks [Corc] que é uma coleção de três soluções para computação paralela para *multi-cores*. As outras duas são o Intel Threading Building Blocks [Rei07] e o Intel Array Building Blocks [Cora].

A biblioteca Intel Threading Building Blocks [Rei07] é baseada num conjunto de *templates* C++ que se concentra em definir tarefas ao invés de definir *threads* como no Cilk, permitindo ao programador evitar as complicações de criação e gestão de *threads*, de forma a obter portabilidade e escalabilidade nas aplicações paralelas. O TBB inclui algoritmos paralelos genéricos, primitivas de sincronização de baixo nível e escalonador de tarefas.

A linguagem pSystem [LS97], à semelhança do Cilk, é também uma extensão à linguagem C e consiste num ambiente de programação paralela para arquiteturas de memória partilhada que utiliza um modelo de paralelismo de tarefas, adicionando novas construções de linguagem para expressar paralelismo. O pSystem tem como vantagem permitir parametrizar a política de escalonamento a utilizar. A linguagem di(pSystem) [SPL99] é uma extensão do sistema de execução pSystem para arquiteturas de memória distribuída.

2.2.2.4 Paralelismo nos dados

Paralelismo de dados é um modelo de programação paralela em que um conjunto de dados, tipicamente uma estrutura de dados, é dividida em pequenas partes, atribuindo a cada *thread* uma parte para ser operado concorrentemente, como pode ser observado na Figura 2.3. Tipicamente, o mesmo conjunto de operações é aplicado concorrentemente sobre os dados. Desta forma, não é necessária a sincronização entre os *threads* durante o processamento desses dados, pois todos eles operam sobre dados diferentes. No entanto, terá que existir sincronização antes e depois do processamento para que seja possível distribuir os dados e juntar os resultados das operações. Em arquiteturas de memória partilhada todos os *threads* podem aceder à estrutura de dados na zona de memória global, enquanto em arquiteturas de memória distribuída a estrutura é dividida e é enviada uma parte para cada nó.

Uma linguagem pioneira na aplicação deste modelo foi a linguagem High Performance Fortran [MC97]. A linguagem é suportada por um conjunto de diretivas para o compilador que permite expressar o paralelismo em aplicações implementas na linguagem Fortran e num conjunto de rotinas implementadas em bibliotecas. A linguagem está assente numa sintaxe para manipulação de *arrays* introduzida no Fortran, suportando a noção de *arrays* distribuídos. Um dos aspetos mais importantes da linguagem é o facto das aplicações serem portáveis entre máquinas de arquiteturas de memória diferentes, como em arquiteturas de memória partilhada ou arquiteturas de memória distribuída.

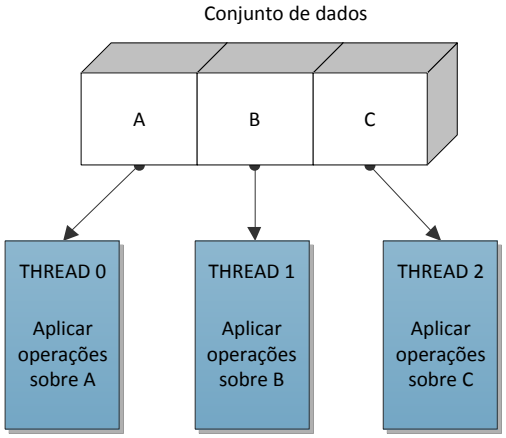


Figura 2.3: Paralelismo de dados.

Desta forma, a linguagem esconde os detalhes de comunicação entre *threads* em arquiteturas de memória distribuída, escondendo do programador a implementação da comunicação usando troca de mensagens. O compilador da linguagem gera um programa SPMD onde o código gerado pelo código do programador em conjunto com o sistema de execução contém os detalhes de implementação dos *arrays* distribuídos e da comunicação entre *threads*.

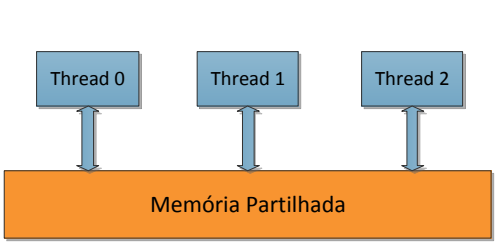
A linguagem ZPL [CCL+98] é uma linguagem para programação paralela baseada em expressões para manipulação de *arrays*. Como o HPF, o ZPL suporta uma visão global da computação em *arrays* distribuídos onde os detalhes de comunicação são gerados pelo compilador e tratados pelo sistema de execução usando uma implementação com modelo SPMD. O paralelismo de dados é obtido através da distribuição de regiões entre as várias unidades de processamento do sistema. Uma região representa um conjunto de índices de um *array* distribuído.

A linguagem Intel Array Building Blocks, que faz parte da colecção Intel Parallel Building Blocks, enfatiza igualmente a programação baseada em paralelismo de dados.

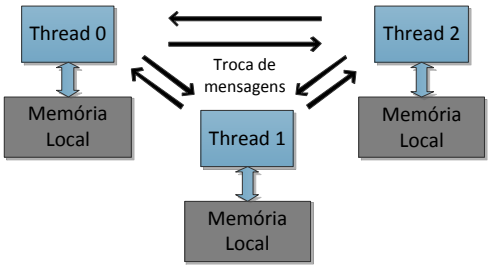
2.2.2.5 *Partitioned Global Address Space* (PGAS)

O PGAS é um modelo de programação que explora o melhor do modelo de programação em memória partilhada e do modelo de programação em memória distribuída. O modelo de memória do PGAS está desenhado em volta de um espaço de endereços global (memória partilhada), onde o espaço de endereços global está logicamente particionado, onde cada partição é local a cada *thread* (memória distribuída). A grande vantagem deste modelo é o facto de expor ao programador a localização dos recursos podendo este gerir explicitamente a afinidade entre a computação e os recursos. Tal é importante pois permite distinguir um acesso remoto de um acesso local permitindo que o programador

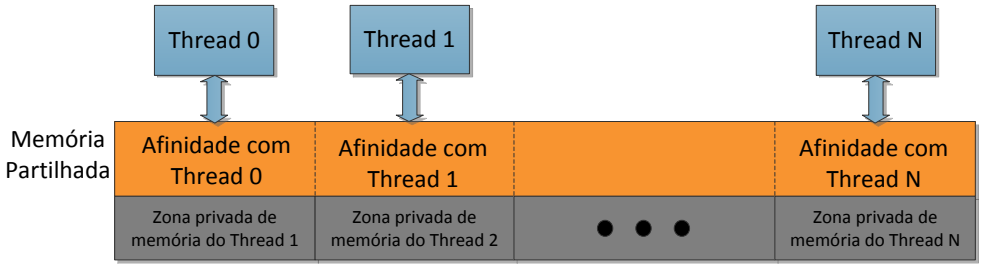
possa otimizar desempenho no acesso aos dados. Uma comparação entre os três modelos pode ser vista na Figura 2.4.



(a) Memória Partilhada



(b) Troca de Mensagens



(c) PGAS

Figura 2.4: Memória Partilhada vs Troca de Mensagens vs PGAS.

A partição da memória partilhada associada a cada *thread*, diz-se que tem *afinidade* com esse *thread*, explorando desta forma a localidade por referência [Den05]. O conceito de afinidade é importante, pois a partir dele, o programador tem noção da localização dos objetos. Sendo o espaço de endereços global, cada *thread* pode aceder a qualquer objeto presente no espaço de endereços, esteja ele localmente ou remotamente, utilizando um estilo semelhante ao usado num modelo de programação em memória partilhada. Cada *thread* para além de possuir uma zona de memória para os dados globalmente partilhados, tem também uma zona privada de memória para dados locais.

Em relação à comunicação, cinco requisitos terão que ser suportados para camadas de comunicação subjacentes às linguagens que instanciam este modelo:

- Suporte para acesso remoto a memória (RMA);
- Baixa latência para acessos remotos a pequenas porções de dados;
- Comunicação não bloqueante;
- Suporte para acessos concorrentes e arbitrários a memória remota;
- Implementação de suporte para comunicação coletiva e mecanismos de sincronização.

Os dados remotos são acedidos através de comunicação uni-direcional e fiável de baixa latência. Desta forma, a comunicação no espaço de endereços global virtual terá que ser implementada recorrendo ao uso de bibliotecas de comunicação unilateral, como GASNet [Bon02], ARMCI [NC99] ou MPI 2 [GLT99]. O sistema de execução de suporte fornece operações para a comunicação, onde a implementação das operações no sistema de execução são delegadas para a biblioteca de comunicação utilizada. As linguagens que instanciam este modelo utilizam um modelo de execução SPMD. A utilização deste modelo de execução implica que o número de *threads* prontos a executar seja afixado na inicialização da aplicação. As linguagens UPC [CCY+99], Titanium [YSP+98] e Co-Array Fortran [NR98] instanciam o modelo de programação PGAS.

A linguagem UPC é uma extensão do C que instancia o modelo PGAS através da utilização da declaração do tipo de acesso das variáveis. Uma variável declarada como shared é uma variável que está no espaço global de endereços, tendo o programador conhecimento que o acesso a essa variável pode incorrer uma penalização no desempenho. Uma variável declarada como private é uma variável local ao *thread*. A linguagem fornece primitivas de sincronização de controlo tais como barreiras, barreiras não bloqueantes, *locks* e cercas. Fornece igualmente primitivas de sincronização de dados como cercas e modelos de consistência relaxado e restrito. Os modelos de consistência referem de que forma os acessos a dados por parte de um *thread* são visíveis a outros *threads*. Uma cerca força a conclusão de acessos, seja de leitura ou escrita, sobre a zona de memória partilhada. Num acesso restrito, é introduzido uma cerca antes do acesso para assegurar que o acesso atual não irá ocorrer antes que outros processos anteriores. Numa escrita restrita uma cerca é colocada depois do acesso também para garantir que nenhum acesso ocorre até que a operação de escrita esteja concluída. Em acessos relaxados não são requeridas cercas. O modo de acesso relaxado conduz a uma melhoria da performance, pois não é necessária a comunicação com os restantes *threads* para a comunicação da alteração do dado alterado.

A linguagem Titanium é uma extensão da linguagem Java 1.4, beneficiando da programação orientada por objetos e das várias características do Java como o *garbage collector*, mas na compilação o código é gerado para C, que depois é traduzido para código máquina pelo compilador de C e *linkado* para as bibliotecas do sistema de execução de suporte. A linguagem CAF é uma extensão da linguagem Fortran 90.

O UPC, o Titanium e o CAF aplicam em grande medida os mesmos conceitos em linguagens diferentes. Sendo o UPC a linguagem mais utilizada das três, ela foi escolhida para uma descrição mais detalhada.

2.2.2.6 *Asynchronous* PGAS (APGAS)

O modelo Asynchronous PGAS [SAB+] estende o modelo de programação PGAS fornecendo um modelo de execução mais rico do que o modelo SPMD, que é aplicado pelo

modelo PGAS. Ele introduz o conceito de localidade como forma de apresentar o modelo PGAS e introduz paralelismo dinâmico através da possibilidade de criar atividades assíncronas dinamicamente.

Localidades Uma localidade é uma porção do espaço de endereços global sendo definida como uma coleção de objetos mutáveis e das atividades que operam sobre esses objetos. Todas as atividades operam apenas numa localidade, mas uma localidade pode ter várias atividades ativas simultaneamente. Se uma atividade pretender executar numa localidade remota, a atividade terá que criar uma atividade na localidade remota. Desta forma, as localidades terão que fornecer a funcionalidade de criar atividades que sejam pedidas remotamente. Uma propriedade importante das localidades é o facto de que todas as localidades presentes numa aplicação não terem que ser obrigatoriamente homogéneas. Desta forma, é possível que uma localidade seja mapeada num CPU, enquanto outra localidade seja mapeada num GPGPU[3].

Atividades assíncronas Uma atividade representa uma pequena parte da computação como uma tarefa. Pode ser vista como um Java *thread*. Quando uma atividade é lançada, localmente ou numa localidade remota, ela pertencerá a essa localidade durante todo o seu tempo de vida, isto é, não será migrada. Existem limitações na forma como as atividades poderão referenciar dados remotos, e portanto quando uma operação numa localidade remota é desejada, é obrigatório que a atividade lance uma atividade nessa localidade para realizar a operação. Uma atividade não é apenas utilizada para realizar computação numa localidade remota mas poderá também ser lançada para enviar dados de uma localidade para outra. Para o controlo da execução das atividades ativas foi introduzido o conceito de *finish*, que é uma construção de sincronização onde a atividade que lançou novas atividades (atividade pai), espere até que todas as atividades lançadas (atividades filho) sejam concluídas.

A linguagem de programação X10 [CGS+05] é uma extensão da linguagem Java que instancia o modelo APGAS e foi escolhida para ilustrar o modelo. Desta forma, o espaço de endereços da computação é particionado, pois as atividades e os objetos são associados a uma única localidade. Assim, o conjunto de todos as localidades fará o espaço de endereço global e compõe o modelo de execução. A linguagem suporta uma semântica GALS (*Globally Asynchronous Locally Synchronous*) nas leituras e escritas nos objetos mutáveis. Qualquer tentativa por parte de uma atividade de modificar um dado remoto resultará num erro, sendo que terá que lançar atividades na localidade onde esse dado pertence para o modificar.

A linguagem fornece uma primitiva de sincronização semelhante às barreiras utilizadas em programas SPMD, mas com algumas nuances diferentes, chamado de *clocks*. Os *clocks* em X10 permitem que as atividades se sincronizem repetidamente, possivelmente

[3]General-purpose Computing on Graphics Processing Units

em diferentes localidades. As atividades registadas num *clock* estarão sincronizadas no fim de cada etapa. A linguagem X10 permite a declaração de blocos atómicos, isto é, conjunto de instruções indivisíveis. O programador, ao utilizar os blocos atómicos num contexto *multi-threading*, garante que um bloco de instruções é realizado em exclusividade sem se preocupar com aspetos relacionados com mecanismos de *locking*, pois os detalhes de implementação são delegados no sistema de execução. De referir, que todos os acessos a memória referidos no bloco atómico terão que ser, obrigatoriamente, locais à localidade associada à atividade que é executada.

A linguagem permite a criação de futuros numa determinada localidade. Quando uma atividade `A` quer enviar uma atividade para uma localidade `P`, é criada uma atividade `B` que executará na localidade `P` a expressão pretendida e terminará a sua execução dando lugar a um futuro. Quando a atividade `A` quer saber o resultado da operação, invocará uma operação *force()* no futuro e bloqueará até que o resultado fique disponível.

A linguagem Chapel [CCZ04, CCZ07] instancia também o modelo APGAS, e tal como o X10, divide o espaço de endereços global em localidades. O detalhe destas duas linguagens não é abordado pois o foco é apenas na linguagem X10 como referência do modelo APGAS, por ser a mais relevante das linguagens que instanciam este modelo e porque alguns dos sistemas de execução de suporte à linguagem X10 serão abordados na Secção 2.3.

2.2.2.7 Programação Explicita em Hierarquia de Memória

Um modelo de programação que surgiu na necessidade de responder à utilização eficiente de arquiteturas de *stream*, como o processador Cell Broadband Engine [CRDI05], em que se pretende obter um maior desempenho e eficiência expondo a hierarquia de memória presente na arquitetura, para que os dados sejam movidos por software entre os vários níveis de memória. Os modelos de programação apresentados anteriormente não são os mais adequados para a utilização deste tipo de arquiteturas, pois estes descrevem a distribuição e a comunicação horizontal entre os vários nós do sistema, e não endereçam o problema da comunicação vertical entre os níveis de memória, sendo este um aspeto crítico neste tipo de arquiteturas. Desta forma foi proposto um modelo de programação que proporciona o controlo explícito na forma como os dados são verticalmente distribuídos e como são transferidos entre os vários níveis de memória. Este modelo fornece uma abstração de múltiplos níveis de memória, em que a representação abstrata da máquina é vista como uma árvore de memória onde os módulos de memória mais lentos ficam no topo da árvore junto à raíz, enquanto as memórias mais rápidas ficam mais abaixo e onde as unidades de processamento ficam nas folhas dessa árvore representativa.

A linguagem Sequoia [FHK+06] instancia este modelo. Está desenhada para facilitar o desenvolvimento de aplicações em que seja exposto ao programador uma noção de abstração da hierarquia de memória, em que este é o aspeto central. O programador é responsável por alocar e mover os dados entre os diferentes níveis de memória com

recurso a construções da linguagem, ao mesmo que tempo que estas sejam facilmente portáveis entre várias máquinas de hierarquias de memória diferentes. Escrever uma aplicação Sequoia envolve descrever abstratamente uma árvore de tarefas que será mapeada na hierarquia de memória da arquitetura alvo. A motivação para representar a arquitetura em árvore está relacionada com aspetos de portabilidade e minimização de complexidade de programação. A comunicação entre os níveis de memória é feita através de chamadas a tasks, que é a construção básica desta linguagem. Uma task é executada no nível de memória em que é invocada. No entanto esta task poderá invocar subtasks, que são as tasks dos níveis inferiores, e estas poderão invocar recursivamente outras subtasks. Desta forma, o nível superior comunica com o nível inferior. O retorno da invocação é a comunicação entre o nível inferior e superior.

2.3 Sistemas de Execução de Linguagens Paralelas

Um sistema de execução (*runtime*) é destinado ao suporte da execução de aplicações desenvolvidas numa determinada linguagem de programação. Tipicamente, um sistema de execução está vocacionado apenas para a linguagem que suporta, não permitindo que outras linguagens ou aplicações possam também ser suportadas.

O estudo dos sistemas de execução das linguagens de programação paralela, que é descrito nesta secção, tem como objetivo ajudar a perceber qual o conjunto de funcionalidades obrigatórios para que seja possível o suporte à execução de aplicações paralelas.

2.3.1 Cilk

Na implementação da linguagem Cilk apresentada em [FLR98], cada processador, que na terminologia Cilk é designado por *worker*, tem uma fila de tarefas a executar associada, e os *workers* consumirão essa fila para executar as tarefas. O sistema de execução implementa um escalonador para permitir que o programador possa expor potencial paralelismo sem ter noção dos detalhes da arquitetura alvo, onde a sua função é decidir, em tempo de execução, como mapear as tarefas definidas pelo programador nos *workers* disponíveis. Outra das funções importantes do sistema de execução é gerir o balanceamento de carga, onde este recorre a uma estratégia de roubo de trabalhos. Quando o *worker* realizar todo o seu trabalho, uma tarefa é retirada do inicio da fila de outro *worker*, escolhido aleatoriamente, impedindo assim que fique sem trabalho.

O sistema de execução apresentado em [Dan05] dá suporte à execução de aplicações escritas na linguagem JCilk apresentada na página 16. A implementação do escalonador e do roubo de trabalho é adaptada do Cilk original mas orientada para Java.

2.3.2 pSystem

O sistema de execução pSystem [LS97], à semelhança do sistema de execução do Cilk também utiliza um conjunto de *workers*. No entanto, os *workers* podiam ser *threads* ou

processos. Além disso permite que o escalonador de tarefas seja parametrizável. É possível implementar vários escalonadores cada um usando uma heurística diferente.

O sistema de execução di_pSystem [Pau98] por sua vez adapta o sistema de execução pSystem para arquiteturas de memória distribuída. O sistema de execução implementa o conceito de agentes que são responsáveis pela execução da aplicação, onde cada máquina presente no sistema corre um agente. Os agentes comunicam entre si para enviar tarefas para serem processadas por outros agentes e para receberem os resultados da sua execução. Um agente é dividido em dois *threads*: *thread* de computação e *thread* de comunicação. O *thread* de computação é responsável pela criação, execução e suspensão de tarefas. Este interage com o escalonador de forma a que se determine se uma tarefa será executada localmente ou remotamente. Se for remotamente, então o *thread* de computação interage com o *thread* de comunicação para que este envie a tarefa para o agente respetivo. Assim, torna-se óbvio qual a função do *thread* de comunicação: receber e enviar pedidos.

A distribuição do trabalho é feita tendo em conta a carga do trabalho local e global. Consoante a informação obtida, o escalonador decide para onde enviar as tarefas a executar. Se o agente não tiver trabalho suficiente, este poderá pedir trabalho a outros agentes.

2.3.3 MuPC

MuPC[ZSS06, Zha07] é um sistema de execução portável de suporte para a linguagem UPC. A implementação deste sistema de execução está baseado no recurso às bibliotecas de implementação de MPI e POSIX Threads, fazendo que este seja portável para qualquer plataforma que suporte MPI e PThreads.

Cada *thread* UPC é mapeado, em tempo de execução, para dois *threads*: *thread* de comunicação e *thread* de computação. O *thread* de computação é responsável por executar o código de execução da aplicação, sendo este, que conterá as chamadas ao sistema de execução MuPC. Todas as tarefas de comunicação são delegadas para o *thread* responsável pela comunicação. O *thread* de comunicação tem dois papeis fundamentais: 1) Trata dos pedidos de escrita/leitura pelo *thread* UPC a executar; 2) Trata dos pedidos de escrita/leitura de *threads* UPC remotos. Os dois *threads* sincronizam-se utilizando *locks* de exclusão mútua e variáveis de condição e comunicam entre si utilizando *buffers* partilhados.

A utilização de uma biblioteca MPI para a implementação do sistema de execução resulta numa implementação simples pois garante, à partida, a ordenação das mensagens em comunicação ponto-a-ponto. O MPI foi escolhido para implementação da API do sistema de execução por duas razões importantes:

- A garantia de portabilidade do MPI;
- As bibliotecas MPI fornecem primitivas de alto nível, ajudando no rápido desenvolvimento dos sistemas escondendo os detalhes das trocas de mensagens.

No entanto, o MPI não cumpre todos os cinco requisitos de comunicação para as linguagens Global Address Space (GAS). Especificamente, não cumpre o requisito em que é requerido comunicação unilateral, no qual a especificação MPI 1.1 não fornece comunicação unidirecional. No entanto a especificação MPI 2.0 fornece, mas existem algumas restrições que fazem com que seja incompatível com os requisitos da linguagem UPC. O sistema de execução MuPC está implementado com a especificação MPI 1.1, mas a comunicação unilateral é simulada no sistema de execução através das funções fornecidas pelo MPI utilizando espera ativa para a receção de pedidos. Como algumas implementações MPI não são *thread-safe*, o sistema de execução encapsula todas as chamadas a funções MPI no *thread* de comunicação.

A função de inicialização do sistema de execução tem como função alocar a fila de receção de mensagens remotas, alocar a fila para pedidos de envio de mensagens, invocar a inicialização do MPI e lançar os *threads* de computação e comunicação.

A localização de um objeto na memória partilhada é dada pelo endereço e pela sua afinidade. Assim, o sistema de execução apenas precisa do endereço e da afinidade para ler ou escrever dados partilhados. Leituras de dados presentes em zonas de memória remotas resulta numa invocação à função get do sistema de execução. O sistema de execução implementa esta função utilizando um protocolo de duas fases:

1. Inicia e faz o pedido;
2. Espera pela resposta, fazendo com que a função get seja síncrona.

Escritas em dados remotos resultam na invocação da operação put no sistema de execução. O único passo que a função faz é a criação de um pedido de envio que resultará numa mensagem que será enviada para o *thread* UPC remoto, sendo a operação put assíncrona. No acesso a dados partilhados, o sistema de execução está otimizado de forma a distinguir acessos a dados partilhados locais dos acessos a dados partilhados em local remoto. Para cada acesso a dados partilhados é testado se a referência à variável é local ao sistema de execução e se o acesso for local, então o acesso é transformado num acesso simples à memória, evitando invocar as respetivas funções de acesso do sistema de execução.

Os mecanismos de sincronização da linguagem UPC são fornecidos através da interface. Implementa a barreira recorrendo a um algoritmo baseado numa árvore binária, e fornece um mecanismo de wait e notify. Existem duas variantes de barreiras:

- Barreiras indexadas, onde cada uma tem um identificador e terá que corresponder apenas a barreiras em outros *threads* com o mesmo identificador;
- Barreiras anónimas, que é a típica barreira.

Fornece igualmente operações de *lock* para operações de baixo nível. Recorre aos *mutexes* da biblioteca Pthreads e num *array* partilhado com o tamanho igual ao número de threads UPC para a implementação do tipo *lock*.

Implementa operações atómicas, que oferecem um acesso livre de *locks* sobre a memória partilhada, tais como *Compare and Swap*, *Double Compare and Swap*, *Fetch and Operate* e *Masked Swap*. Quando uma operação atómica é realizada sobre uma variável partilhada, todas as restantes operações atómicas de outros *threads* sobre a mesma variável irão falhar. O sistema de execução inicia uma operação atómica criando uma mensagem MPI de pedido para ser enviada ao *thread* com afinidade com a região de memória acedida. O recetor recebe este pedido de operação e insere-o numa fila para ser tratada. Usando uma fila, garante-se que os acessos concorrentes serão serializados. O primeiro pedido a chegar será efetuado com sucesso. Todos os outros pedidos presentes na fila que acedam a dados modificados pela operação efetuada com sucesso, falharão. O recetor depois de efetuar a operação enviará uma mensagem MPI a confirmar o sucesso ou a falha da operação.

2.3.4 X10 - Sistema de execução base

A implementação base do X10, apresentada em [CGS+05], opera numa única máquina virtual X10 e todas as localidades X10 são executadas em cima da única instância da máquina virtual X10 a executar. O suporte está escrito em Java e é executado numa máquina virtual Java. Desta forma, possui todas as vantagens da programação orientada a objetos e das funcionalidades da máquina virtual Java, onde fornece todo o suporte para as construções e funcionalidades da linguagem X10.

Nesta implementação, uma localidade atua como um executor que lança atividades individuais utilizando uma *pool* de *threads* Java, em que um *thread* disponível na *pool* é atribuído para a execução das operações da atividade. O modelo *pool* de *threads* faz com que seja possível reutilizar um *thread* para múltiplas atividades X10. Contudo, se uma atividade X10 bloquear, o *thread* responsável pela execução das operações da atividade não estará disponível até que a operação acabe. Nesse caso, um novo *thread* Java será criado para responder a mais atividades. Desta forma, o número de *threads* que podem ser criados não tem qualquer limite, podendo resultar numa diminuição de performance da aplicação.

Os blocos atómicos estão implementados utilizando um *lock* de exclusão mútua por localidade. Enquanto esta estratégia de implementação limita o paralelismo, esta garante a ausência de *deadlock*, desde que nenhuma computação necessite de adquirir o *lock* em mais que uma localidade.

Uma outra implementação do sistema de execução [BCD+06] foi efetuada utilizando o pacote *java.util.concurrent* do Java para estudo comparativo. A motivação para a utilização do pacote JUC é dada por duas razões importantes:

- Desempenho - Utiliza uma *pool* de *threads* para atribuir *threads* Java para a execução das atividades, ao invés de criar um novo *thread* Java por cada atividade;
- Simplicidade - A sincronização de baixo nível está encapsulada na JUC na implementação da *pool* e de outros serviços.

A comparação foi feita entre a implementação X10 para SMP utilizando o pacote JUC com uma implementação Java para SMP utilizando o pacote JUC também. Os resultados obtidos foram encorajadores, onde a conjugação dos níveis de abstração do X10 com o pacote JUC são uma mais valia para a produtividade de aplicações paralelas.

2.3.5 CX10 - Sistema de execução distribuído para a linguagem X10

O sistema de execução CX10 [Xin06] é um sistema de execução distribuído baseado na implementação base do X10 em que é baseada numa única máquina virtual. O objetivo deste sistema de execução é permitir que o código gerado pelo compilador utilizado na implementação base possa também ser executado em cima de plataformas distribuídas, como *clusters*. A ideia geral é colocar várias instâncias da máquina virtual X10 (implementação base em cada máquina presente no *cluster*) a trabalhar em conjunto para concetualmente formarem um único sistema de execução que será visualizado pelo programador.

Na inicialização do sistema de execução é necessário especificar o número de máquinas do *cluster* que serão utilizadas, quantas máquinas virtuais X10 a executar e como mapeá-las para as máquinas físicas e mapear as localidades para as VMs. As máquinas sem VMs configuradas podem mais tarde executar dinamicamente VMs, e VMs sem qualquer localidade configurada podem mais tarde ser usadas para criar dinamicamente novas localidades.

Em relação às atividades assíncronas existem diferenças relativamente à implementação base pois as atividades podem ser executadas remotamente, isto é, numa localidade em outra máquina virtual. Dessa forma, é necessário verificar se a localidade está na mesma máquina virtual em que foi lançada. Se sim, então o procedimento utilizado na implementação base é seguido; caso contrário é necessário serializar o objeto e enviar para a máquina virtual que conterá a localidade alvo.

Em relação aos *clocks* existem, naturalmente, diferenças apesar da semântica ser mantida. Numa configuração distribuída, as atividades de localidades de VMs diferentes podem-se sincronizar num mesmo *clock*. O *clock* sendo um objeto X10 pertence a apenas uma localidade. Dessa forma, existem objectos opacos que representem o *clock* nas várias localidades, em VMs distintas. Estes objetos opacos contêm apontadores para a localidade (e respetiva VM) na qual o *clock* pertence. Desta forma, existe um protocolo de comunicação que envolve a troca de mensagens para proceder às operações sobre o clock e disseminação do estado do *clock*. O conjunto dos objetos opacos e da localidade origem do *clock* forma conceptualmente o *clock* visto pelo programador.

2.3.6 Sequoia

Um dos desafios para a implementação do sistema de execução do Sequoia para um *cluster* [FHK+06] é como representar a abstração da hierarquia de memória do sistema, visto

que a raíz da árvore é um nível virtual de memória, sendo este a agregação da memória de todos os nós do sistema. Dessa forma, é necessário implementar e representar este nível de memória através de um passo de mapeamento em tempo de compilação. Um único nó do *cluster* é responsável por executar o código Sequoia, começando pela raíz da árvore de tarefas, até encontrar paralelismo. Quando encontrar paralelismo, este é responsável por notificar os restantes nós para realizar as sub-tarefas em paralelo. A comunicação e sincronização é implementada com recurso ao MPI. A comunicação é implementada via troca de mensagens não bloqueante MPI, onde cada nó do *cluster* tem uma instância do sistema de execução a executar. O input para o compilador Sequoia é o código da aplicação escrito em Sequoia e um ficheiro de configuração com a especificação do mapeamento para a arquitetura alvo. O compilador gera código C em que este utiliza a interface do sistema de execução especifico da plataforma para a qual a aplicação irá executar. Contudo, para que a aplicação Sequoia seja portável para várias plataformas, é necessário que seja implementado um *backend* especifico para cada plataforma para o compilador, de forma a que o código gerado possa ser executado na plataforma alvo. Ou seja, com esta aproximação, cada nova arquitetura requer que seja criado um *backend* especifico.

Para resolver este problema, em [HPR+08] foi apresentado um interface para o sistema de execução única para qualquer hierarquia de memória, que permite que o compilador gere código que utilize as operações definidas nesta interface, e que para cada arquitetura, um sistema de execução que implemente esta interface seja construído. Desta forma, não é necessário implementar um novo *backend* para cada nova arquitetura. A interface deste sistema de execução tem a particularidade de permitir que a implementação de um sistema de execução para uma hierarquia de memória multi-nível seja composta por vários sistema de execução de sub-níveis, isto é, um sistema de execução para um *cluster* de SMPs pode ser implementado à custa da agregação do sistema de execução para o *cluster* com o sistema de execução para a arquitetura SMP. Em [HPR+08] é apresentado um sistema de execução para cluster e um sistema de execução para SMP, em que ambos implementam a mesma interface, e um sistema de execução destinado para clusters de SMP em que os dois sistema de execuções especifico são combinados. Um *cluster* apresenta uma hierarquia de memória de dois níveis. O agregado da memória de todos os nós perfaz o nível superior, sendo um nível virtual. A memória individual de cada nó perfaz o nível inferior, onde cada nó é filho do nível superior.

O sistema de execução para o *cluster* foi implementado utilizando uma biblioteca PThreads e uma biblioteca de MPI-2. Na inicialização do sistema de execução o nó *master* é responsável pela execução das funções do sistema de execução do nível superior. Os outros nós (nós *slaves*) são inicializados como nós pertencentes ao nível inferior e desta forma esperam invocações do nó *master*. Semelhante ao MuPC e ao di_pSystem, cada nó inicializa duas *threads*: *thread* de comunicação e *thread* de execução. O *thread* de execução executa as instruções das tarefas e o *thread* de comunicação trata das transferências de dados, sincronização e chamadas às tarefas no *cluster*. Um SMP apresenta também uma

hierarquia de memória de dois níveis. O sistema de execução SMP é implementado com recurso à biblioteca de *threads* do POSIX.

2.3.7 Quadro Comparativo

A Tabela 2.2 pretende estruturar em módulos as funcionalidades oferecidas pelos sistemas de execução de suporte aos modelos de programação estudados. Esta tabela resume a identificação de alguns requisitos de cada um dos modelos de programação, dando uma visão geral sobre o suporte e foi a base para o desenho do *middleware* a desenvolver. Os módulos a desenvolver são:

Inicialização: Módulo responsável pela inicialização do sistema de execução.

Comunicação: Módulo responsável pela comunicação entre *threads*. A comunicação pode ser unidirecional/assíncrona ou bidirecional/síncrona.

Sincronização: Módulo responsável por fornecer os mecanismos de sincronização necessários para a execução concorrente de *threads*.

Gestão de tarefas: Módulo responsável por criar tarefa e adicioná-la na fila de tarefas do serviço alvo.

Gestão de memória partilhada: Módulo responsável por gerir o valor dos dados partilhados à vista de cada thread aquando de uma leitura. Se for relaxado, cada *thread* pode ver um valor diferente; se for restritivo os *threads* verão exatamente o mesmo valor.

Distribuição e Redução: Módulo responsável por fornecer operações de distribuição de dados e operações de redução.

2.4 Middlewares

Middleware é uma camada intermédia de software que é utilizada como intermediária entre as aplicações e o sistema operativo, no qual permite o desacoplamento entre si, fornecendo interoperabilidade entre as aplicações. O objetivo de um *middleware* é facilitar o desenvolvimento de aplicações e a colaboração entre aplicações, fornecendo abstrações comuns às aplicações para computação e comunicação, escondendo a heterogeneidade da plataforma e os detalhes de programação de baixo nível.

Os sistemas de execução estudados estão muito vocacionados para as próprias linguagens e por isso o estudo para o trabalho relacionado foi alargado para considerar *middlewares* já existentes para programação paralela.

Módulos/S.E.	MuPC	cX10	Cilk/pSystem	di_pSystem
Inicialização	Iniciar cada nó: - Inicializa MPI. - Inicializa mutexes e buffers. - Lança *threads* de computação e comunicação.	Iniciar cada localidade: - Inicia *pool* de *threads*. - Inicia filas de actividades.	Inicia cada *worker*: - Inicializa filas de tarefas. Inicia escalonador.	Iniciar cada *worker*: - Inicializa fila de tarefas. - Lança threads de computação e comunicação. Inicia escalonador em cada máquina.
Comunicação	Unidirecional.	Unidirecional.	Bidirecional.	Bidirecional.
Sincronização	Barreiras. *Locks*. Cercas	Blocos atómicos. *Clocks*. Finish.	Barreiras. Cercas.	Barreiras. Cercas.
Gestão de tarefas	Mensagens propagadas com pedidos de acesso.	Mensagens activas para lançar atividades.	Tarefa adicionada na fila respetiva.	Mensagens activas para lançar tarefas.
Gestão de Memória Partilhada	Modelos de consistência relaxado e restritivo.	Acesso sequencial.	Acesso sequencial.	Modelo de consistência relaxado.
Distribuição e Redução	Distribuição estática.	Distribui estruturas de dados por localidades. Não tem redução.	Não existe.	Não existe.

Tabela 2.2: Quadro comparativo entre funcionalidades oferecidas por cada sistema de execução de suporte.

2.4.1 ARTS

O ARTS [BNSP99] é um *middleware*, implementado em C++, para computação paralela e distribuída orientado a objetos que fornece serviços básicos para a execução paralela de aplicações num espaço global de objetos. O espaço global é composto por domínios que contêm objetos ativos e objetos passivos. Os domínios (semelhantes às localidades no X10) são processadores virtuais que controlam o acesso aos objetos encapsulados. Cada domínio pode ter o seu próprio espaço de endereçamento ou partilhar um espaço de endereçamento com outros domínios. O modelo de objetos ativos formam a base do ARTS visto que fornecem uma maneira simples de obter paralelismo, visto que os objetos ativos [LS96] introduzem concorrência/paralelismo sobre as operações de um objeto abstraindo a execução dos métodos da sua invocação.

2.4.2 Jaroodi

Em [AJMJS03] é apresentando um *middleware* que fornece serviços para suporte de vários modelos de programação paralela e distribuída e também para desenvolvimento de aplicações paralelas e distribuídas em *clusters* e ambientes heterogéneos. Estes serviços são comuns a vários modelos de programação, como escalonamento de tarefas, monitorização do sistema, balanceamento da carga e sincronização. Desta forma, o *middleware* pode suportar diferentes modelos de programação, como pode ser visto na Figura 2.5.

O *middleware* está implementado em Java e é constituído por agentes distribuídos, cada um a residir em cada nó do sistema, que comunicam entre si para realizar as suas funções. A função de cada agente é criar, escalonar e executar cada tarefa, controlar e monitorizar os recursos disponíveis no sistema para escalonar da melhor forma as tarefas.

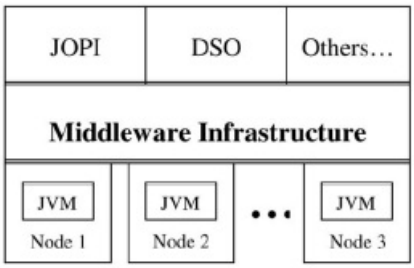


Figura 2.5: Representação do *middleware* como suporte a vários modelos de programação paralela e distribuída [AJMJS03].

2.4.3 ProActive

O ProActive [BBC+06, HCB04] é uma API Java suportada por um *middleware*, para programação paralela e distribuída. Oferece abstrações de alto nível para simplificar a programação de aplicações para serem executadas num *SMP*, *clusters* ou em *grids*, oferecendo serviços como migração, comunicação em grupo e segurança. Em relação à comunicação, a comunicação pode ser realizada por três formas: RMI, JINI (para descoberta de serviços) e um através de um protocolo baseado em XML.

Uma aplicação concorrente ou distribuída é composta por um conjunto de objetos ativos. Cada objeto ativo contém o seu próprio fio de execução e oferece uma interface para a invocação de métodos que são transformados em pedidos e colocados numa fila para serem processados. As invocações no objeto ativo são sempre assíncronas e são representadas do lado do invocador por um *futuro*. Do lado do invocador, o objeto ativo é representado por um *proxy* e este *proxy* retorna futuros, para representação do resultado da invocação, sempre que uma invocação é gerada. O *proxy* contém exatamente a mesma interface do objeto ativo mas a implementação dos métodos é diferente, onde normalmente a implementação é delegada para o objeto ativo, onde a comunicação entre si é realizada através do módulo de comunicação. Quando o invocador necessitar do resultado da invocação, ele invocará um método bloqueante no futuro que só retornará o resultado apenas quando este estiver disponível, sendo esta sincronização implícita não necessitando de nenhuma operação especial por parte do programador.

2.4.4 Parallel Java

Parallel Java [Kam07] é uma biblioteca totalmente desenvolvida em Java para programação paralela em arquiteturas de memória partilhada, distribuída ou híbridas, sendo suportada através de um *middleware*.

O seu desenho foi inspirado nas características do OpenMP e do MPI, unificando-as numa única API. Esta unificação permite que uma aplicação possa ser executada num SMP, cluster ou numa arquitetura híbrida. As abstrações oferecidas ao programador permitem que o programador defina regiões paralelas, que são equivalentes a tarefas, grupos de *threads* que executarão as regiões paralelas e ciclos paralelos.

Os nós para computação presentes no sistema são compostos pelo nó principal e pelos nós secundários. Ambos podem ser vistos com o nó *master* e nós *slaves* respetivamente. O *middleware* é responsável pela gestão das tarefas entre os nós do sistema e pelo lançamento dos processos nos nós dos sistema. Desta forma, o *middleware* contém um módulo que é responsável pelo lançamento dos processos nos nós secundários, ficando estes à espera de trabalho que será enviado pelo processo a executar nó principal. O *middleware* contém também um módulo que é responsável pela gestão da fila de tarefas, escalonando as tarefas pelos nós secundários e é também responsável por manter informação do estado global do *cluster*.

2.4.5 Dryad

Dryad [IBY+07] é uma infraestrutura *middleware* que permite a execução paralela de aplicações que recorram ao paradigma de paralelismo de dados. Foi concebido com o objetivo de realizar a computação em plataformas de execução como pequenos *clusters* até grandes centros de dados. O modelo de execução combina um determinado número de vértices computacionais com arestas de comunicação formando um grafo acíclico de fluxo de dados. A computação das aplicações é então realizada por executar os vértices num conjunto de computadores onde as arestas formam a comunicação entre os vértices recorrendo a ficheiros, protocolos de rede ou filas partilhadas.

O programador é responsável apenas por fornecer uma implementação dos vértices que são tipicamente escritos como programas sequenciais. O sistema de execução por sua vez faz o mapeamento dos vértices pelos recursos disponíveis na plataforma de execução, podendo ser executados num *cluster* ou numa máquina *multi-core*. A comunicação pode estar implementada com diferentes tecnologias, ficando o sistema de execução responsável por escolher a implementação consoante a localização dos vértices, ou seja, se dois vértices adjacentes não partilharem uma mesma máquina a comunicação será realizada através da rede recorrendo ao protocolo TCP.

Para a especificação dos vértices, os autores desenvolveram uma linguagem declarativa onde os programadores são responsáveis pela construção conceptual do grafo. No entanto, de forma a evitar o esforço do programador para aprender uma nova linguagem, os autores desenvolveram também abstrações de alto nível a partir da linguagem C++.

O sistema de execução contém um processo designado por *Job Manager* que é responsável por inicializar os vértices e construir o grafo. Após o grafo ser construido, a comunicação é feita através de *peer-to-peer*, ficando o *Job Manager* responsável apenas por evitar engarrafamentos.

2.4.6 Flexpar

FlexPar [UMG08] é um *middleware* que tem como objetivo a execução de aplicações paralelas em ambientes heterogéneos. É um *middleware* baseado em componentes reconfiguráveis que permite suportar vários modelos de programação. No protótipo realizado foi

implementado o suporte para implantar processos CSP (*Communicating sequential processes*) em múltiplos ambientes (e.g. sensores, sistemas embebidos, etc.).

A sua arquitetura é constituída por três componentes: (i) Núcleo do sistema de execução; (ii) Conjunto de componentes de extensão; (iii) Interface para as aplicações.

O núcleo segue uma arquitetura *micro-kernel* o que permite que ele seja instalado numa grande variedade de plataformas, incluindo em dispositivos com recursos computacionais muito limitados. Ou seja, o núcleo apenas implementa as funcionalidades mínimas necessárias para a programação paralela em determinado ambiente. O *micro-kernel* está implementado em Java o que permite a sua portabilidade para uma grande variedade de plataformas.

O conjunto de componentes de extensão permite adicionar funcionalidades ao núcleo. Existem dois tipos de componentes de extensão: (i) *Loaders*; (ii) *Binders*. Os *loaders* encapsulam a complexidade de carregamento de processos CSP em ambientes heterogéneos. Cada *loader* suporta um mecanismo de carregamento diferente permitindo que uma grande variedade de estilos CSP possam usufruir da plataforma. Os *binders* implementam mecanismos de comunicação (*channels* na terminologia CSP) entre processos CSP. É assim possível ter diferentes implementações da comunicação entre os processos CSP, onde esta forma de comunicação pode ser diferente devido às características do ambiente de execução. No protótipo implementado pelos autores foram implementados *loaders* e *binders* que permitem dois tipos de processos CSP: JCSP (Java CSP) e Ocamm-pi.

2.4.7 Virtualized Self-Adaptive Parallel Programming Framework

O artigo [CCS+09] apresenta um *middleware* independente da plataforma para programação paralela em ambientes heterogéneos. O *middleware* é apresentado como sendo composto por dois níveis: o seu núcleo e o modelo de programação que oferece. O modelo de programação oferece uma visão uniforme ao programador independentemente da plataforma de hardware. O modelo é apresentado como um SSI (*Single System Image*) em que o conjunto de nós que constitui a plataforma de execução é apresentada como uma única máquina virtual. Ou seja, as computações interagem via memória virtualmente partilhada, não existindo qualquer noção da localização da computação.

O núcleo é composto por duas entidades: *Node-Level Virtual Machine Monitor* (NVMM) e *System-Level Virtual Infrastructure* (SVI). Cada nó que constitui a plataforma de execução tem um NVMM. Cada NVMM tem uma máquina virtual associada que é responsável por virtualizar os recursos. O NVMM é responsável por executar o trabalho, comunicar com outros NVMMs e monitorizar a carga no nó. O SVI é o componente principal de toda a operação do sistema sendo responsável pela gestão global do sistema. Entre as suas responsabilidade encontra-se o escalonamento do trabalho pelos NVMMs e melhorar o balanceamento da carga podendo mesmo migrar a computação num nó para outro, migrando o NVMM para outro nó.

O modelo de programação oferece um modelo de paralelismo de dados e de tarefas

através da interface, escondendo do programador a divisão e mapeamento do trabalho pelas unidades de execução. Os programadores preocupam-se apenas em identificar o paralelismo lógico das suas aplicações sequenciais, assumindo que cada um dos blocos de código será executado em paralelo. O modelo também oferece mecanismos de sincronização nomeadamente é possível especificar que partes do código devem ser executadas de forma atómica através de transações.

2.4.8 Análise Crítica

Nesta subsecção serão discutidos criticamente os *middlewares* estudados mediante as informações que foram possíveis obter a partir da leitura dos artigos que os descrevem.

Relativamente aos modelos de programação suportados, tanto o ARTS, como o descrito em [AJMJS03] e o Virtualized oferece um modelo de programação em memória partilhada onde a memória partilhada é virtual, ou seja, a memória é fisicamente distribuída existindo uma camada de software que simula a memória partilhada. Portanto, este modelo tem como desvantagem o facto de não ser dado ao programador o controlo da computação, não podendo ele distinguir os tipos de acesso dados (se locais ou remotos). O Virtualized oferece um paradigma de paralelismo de dados e de tarefas, sendo que o paralelismo de dados é obtido através da submissão de várias tarefas que operam em zonas distintas da memória virtual partilhada. Relativamente à interface, o programador necessita de anotar as partes do código que podem ser executadas em paralelo, mas não se descreve de que forma isso é possível.

O FlexPar e o Pro-Active oferecem um modelo de programação orientado a troca de mensagens. O FlexPar baseia-se no modelo teórico CSP, em que os processos comunicam entre canais comuns, onde esta comunicação é síncrona, suportando apenas paralelismo de tarefas. O Pro-Active baseia-se no modelo orientado a objetos distribuídos, como encontrado no .Net Remoting, RMI e CORBA. Este modelo é instanciado através da utilização de objetos ativos. Um objeto ativo contém o seu próprio fluxo de execução e a sua interface. O programador é responsável pela criação de uma classe que corresponderá à interface do objeto ativo e também pela própria criação do objeto ativo recorrendo às primitivas oferecidas.

O Parallel Java oferece um modelo de programação em memória partilhada e distribuída. Ele não oferece uma interface uniforme independentemente da arquitetura, sendo esta a sua desvantagem. A sua interface é uma unificação das primitivas oferecidas pelo OpenMP e pelo MPI, fazendo com que o programador esteja ciente da arquitetura alvo não sendo possível desta forma obter portabilidade das aplicações.

O Dryad oferece apenas um paradigma de paralelismo nos dados em arquiteturas de memória distribuída. A sua interface tem como desvantagem o facto de ser necessário aprender a sintaxe de uma linguagem declarativa criada pelos autores de forma a especificar o conjunto de unidades de execução e as dependências relativas à comunicação entre eles.

Relativamente às arquiteturas suportadas, todos eles têm como objetivo suportar ambientes heterogéneos.

3

Conceção e Desenho

Neste capítulo é apresentado o desenho do *middleware* que foi desenvolvido no âmbito desta dissertação. Na Secção 3.1 são descritos os objetivos a atingir pelo *middleware*; na Secção 3.2 é descrita a visão geral da computação; por último, na Secção 3.3 é descrita a arquitetura do *middleware* onde se apresenta a interface para as aplicações que é oferecida ao programador e onde se apresentam os módulos que constituem a infraestrutura para o suporte das funcionalidades oferecidas.

3.1 Objetivos

O trabalho desenvolvido nesta dissertação visa propor uma camada de software (*middleware*) que tem como objetivo o suporte ao desenvolvimento e execução de aplicações concorrentes e paralelas. O *middleware* é utilizado como intermediário entre as aplicações e a plataforma de hardware que constituí a arquitetura alvo, fornecendo as funcionalidades mais comuns para o desenvolvimento de aplicações, tais como: paralelismo de tarefas e de dados, comunicação e controlo de concorrência. O *middleware* é responsável por fazer toda a gestão da concorrência necessária para a execução das aplicações, escondendo todos os seus detalhes do programador.

O *middleware* desenvolvido destaca-se por:

- Oferecer simplicidade na programação de aplicações paralelas através de uma interface que oferece abstrações de alto nível que reduzem o mais possível a tarefa do programador, permitindo aumentar a sua produtividade;
- Oferecer ao programador o controlo na localização da computação, permitindo que

este possa gerir a afinidade entre a computação a realizar e os recursos computacionais existentes, permitindo assim que possa otimizar o desempenho;

- Dar suporte a vários modelos de programação de forma a que possa ser utilizado como suporte ao desenvolvimento de um grande leque de aplicações concorrentes e paralelas, bem como servir de suporte a sistemas de execução de linguagens de programação;
- Ser independente da plataforma permitindo assim obter portabilidade das aplicações para vários tipos arquiteturas, tais como arquiteturas de memória partilhada, arquiteturas de memória distribuída ou arquiteturas híbridas.

3.2 Visão geral

O *middleware* proposto apresenta-se como uma máquina virtual onde o hardware, que é constituído pelo conjunto de todos os nós (máquinas físicas) do sistema, é abstraído do programador a partir de uma interface bem definida que apresenta uma visão uniforme independentemente da arquitetura alvo. Assim é possível que uma mesma aplicação execute em diferentes tipos de arquiteturas sem que a sua estrutura tenha que ser modificada, ao contrário do que acontece no desenvolvimento de aplicações que recorrem a bibliotecas para o suporte de funcionalidades para arquiteturas específicas. Por exemplo, uma aplicação desenvolvida em C que recorra apenas à biblioteca PThreads para introduzir paralelismo na computação, terá que alterar a sua estrutura caso se pretenda que a aplicação execute num ambiente distribuído.

O *middleware* tem como objetivo esconder os pormenores da computação paralela do programador, por meio da interface, tornando o desenvolvimento de aplicações menos complexo. A Figura 3.1 apresenta a visão geral do sistema.

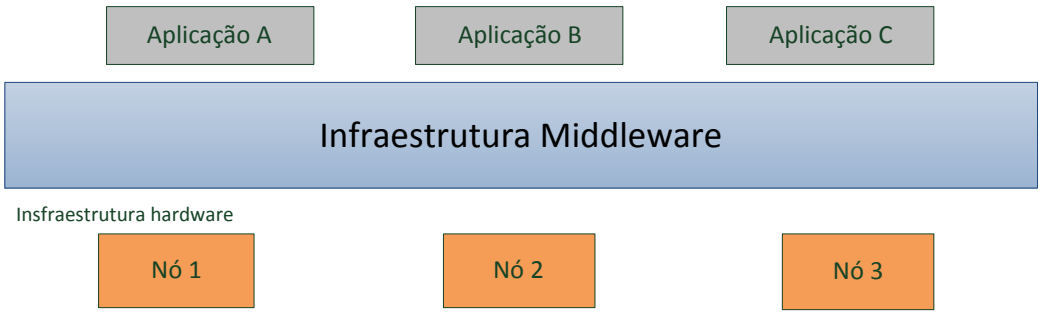


Figura 3.1: Visão geral do sistema.

Cada um dos nós representa uma máquina com arquitetura de memória partilhada e executa uma instância do *middleware* à qual se refere como implementação base para arquiteturas de memória partilhada. A composição de um conjunto de implementações

base interligadas pela rede fornece a infraestrutura de suporte para arquiteturas de memória distribuída, nomeadamente *clusters*. Quer sobre a implementação base quer sobre o suporte para ambientes distribuídos, podem executar várias aplicações, não sendo necessário iniciar uma nova instância do *middleware* para cada aplicação.

3.3 Arquitetura

A arquitetura do *middleware* é constituída por três entidades principais: (i) Interface para as aplicações; (ii) Núcleo do *middleware*; (iii) Interface para os *drivers*.

O desenho do *middleware* é inspirado no desenho dos sistemas de operação, na medida em que, para além de oferecer a interface para as aplicações, também oferece uma interface, baseada na noção de *driver*, para o suporte de múltiplas especializações das várias funcionalidades suportadas. Portanto, a abordagem seguida para o desenho do *middleware* faz com que este atue como um intermediário entre as aplicações e as tecnologias necessárias para a implementação de cada funcionalidade oferecida. A Figura 3.2 retrata a visão global da computação.

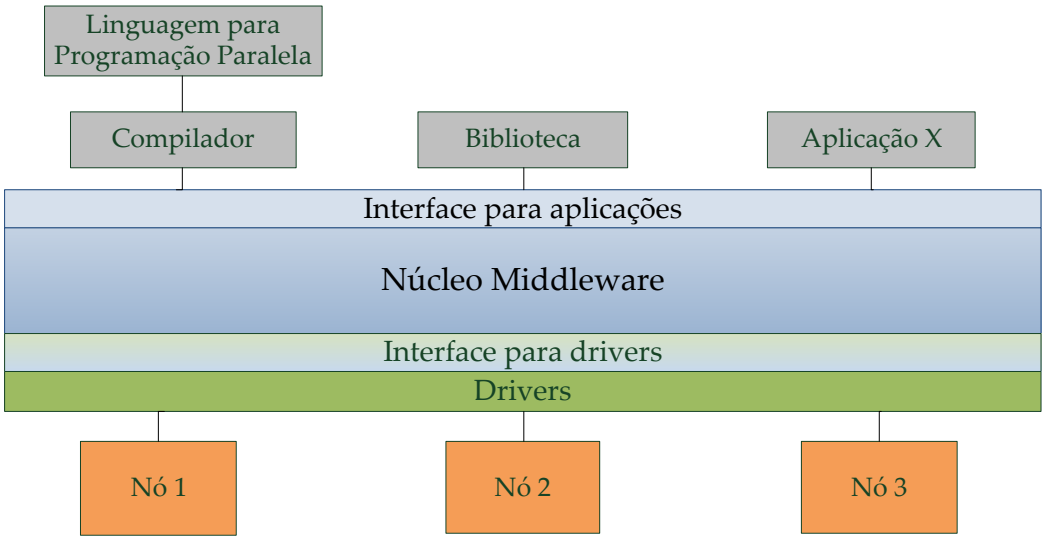


Figura 3.2: Visão global da computação.

A interface para as aplicações, que será abordada em detalhe na Subsecção 3.3.1, pretende oferecer, de uma forma simples e concisa, as funcionalidades necessárias para o desenvolvimento de aplicações paralelas independentes da plataforma. As funcionalidades são oferecidas pela interface como abstrações de alto nível que abstraem o programador da gestão da concorrência permitindo-lhe concentrar-se o mais possível na lógica das suas aplicações.

O núcleo do *middleware*, que será abordado com mais detalhe na Subsecção 3.3.2, é

constituído por um conjunto de funcionalidades (ou serviços), encapsuladas em módulos, que tornam possível a execução paralela de computações. É responsável pela gestão de todas as interações necessárias para a realização da computação submetida.

O recurso ao conceito de *driver* permite desacoplar a implementação do núcleo, permitindo por isso que se possa adequar cada instância do *middleware* à arquitetura subjacente. Permite também delegar em tecnologias já existentes a implementação efetiva de uma dada funcionalidade. Por exemplo, no que se refere à comunicação via troca de mensagens, esta pode fazer uso de uma implementação do *standard* MPI ou de filas de mensagens Unix System V, entre outros, bastando que para tal exista um *driver*. De realçar o facto de como uma dada funcionalidade pode ser implementada à custa de tecnologias já existentes, pode acontecer o caso de o *middleware* ter que integrar outros *middlewares*, o que é prática comum em muitos sistemas de execução para programação paralela. Por exemplo, o facto do módulo de comunicação poder ser implementado através de um *driver* que utiliza uma implementação de MPI, poderá ter que ser necessário inicializar o sistema de execução do MPI para que este possa ser utilizado.

3.3.1 Interface para aplicações

A interface para as aplicações oferecida está centrada à volta do conceito de localidade. Este conceito, que será descrito com mais pormenor na Subsecção 3.3.1.1, tem como base um modelo com múltiplos espaços de endereçamento, o que vai de encontro a propostas de linguagens como o X10 e o Chapel, que têm favorecido este tipo de modelo para a programação em arquiteturas *multi-core*. Conceptualmente, uma localidade pode ser vista como um multi-processador de memória partilhada virtual, isto é, uma unidade computacional que contém um número finito de fluxos de execução e uma região de memória partilhada que é uniformemente acedida por estes.

A motivação para a escolha do conceito de localidade para ser a base da interface deve-se essencialmente ao facto de este induzir o conceito de localização da computação. O recurso a este conceito dá ao programador o controlo sobre a localização da computação. Além disso, mapeia-se de forma natural tanto em arquiteturas de memória distribuída como arquiteturas *multi-core*, pois estas últimas podem ser vistas como uma arquitetura de memória distribuída, na medida em que, cada núcleo e a sua *cache* é visto como um nó num ambiente distribuído.

Relativamente ao modelo de execução, a computação é composta por um conjunto de localidades. Portanto, o programador desenha a sua aplicação como um conjunto de localidades sobre as quais pode submeter trabalho para ser executado em paralelo. Existem dois tipos de localidade: passivas e ativas. Uma aplicação terá de conter, pelo menos, uma localidade ativa. A localidade ativa servirá como ponto de entrada de toda a computação a realizar. Ela tem como característica diferenciadora o facto de ter um método *main* que é invocado assim que a localidade é lançada. A execução desse método executa concorrentemente com o resto da aplicação.

Apesar desta interface ser simples e concisa, ela requer uma mudança de paradigma relativamente à programação sequencial. Para suavizar esta transição foi desenvolvido um conjunto de anotações que permitem simplificar o desenvolvimento de aplicações.

3.3.1.1 Localidade

Uma localidade denota um processador virtual sobre o qual se adjudica explicitamente trabalho. Uma localidade abstraí uma unidade de execução, podendo por exemplo abstrair um CPU, um GPU ou um nó com vários CPUs, não sendo obrigatório que as localidades que compõem uma computação sejam homogéneas. Esta característica permite ao programador beneficiar de hardware especial que possa pertencer à plataforma de execução, como aceleradores. Por exemplo, poderá realizar uma computação que executará sob o paradigma de paralelismo de dados numa localidade mapeada num GPU de forma a obter um melhor desempenho em comparação com a sua execução num CPU.

A localidade é constituída pela sua interface, pelo seu próprio espaço de endereçamento de memória, por uma ou mais filas de tarefas prontas a executar e por um conjunto finito de fluxos de execução disponíveis. A sua representação é retratada na Figura 3.3.

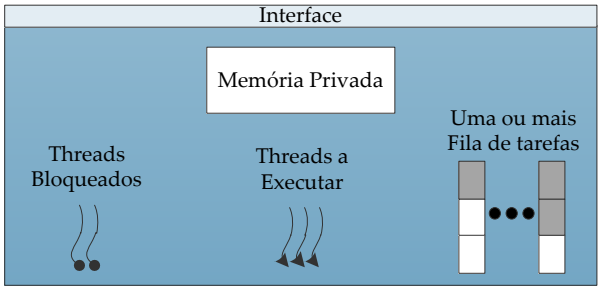


Figura 3.3: Representação de uma localidade.

A computação é composta por um conjunto de localidades onde o programador interage (adjudica trabalho) com estas através da invocação das operações especificadas na sua interface. Portanto, a comunicação entre localidades é obtida através da invocação das operações. A semântica seguida mapeia de forma natural nas atividades do X10 e da criação e submissão para execução de tarefas no Cilk.

Adicionalmente, a interface da localidade pode ser estendida para acomodar novas operações especificadas diretamente pelo programador, comportando-se como um módulo sobre o qual se podem invocar operações específicas. Este mecanismo permite estender o âmbito para suportar linguagens baseadas em objetos ativos, atores, agentes ou serviços.

Os fluxos de execução executam as tarefas de forma concorrente e poderão aceder à zona privada de memória da localidade para comunicarem entre si, sendo necessário que

a localidade ofereça mecanismos que tenham como objetivo a garantia de coerência dos dados partilhados.

A região de memória local de cada localidade diz que tem afinidade com a localidade, explorando desta forma a localidade por referência. A noção de afinidade é vantajosa pois o programador tem conhecimento da localização dos dados e terá a noção de que lançar uma computação na mesma localidade terá um custo menor, em termos de desempenho devido à comunicação, em comparação com a execução dessa computação numa localidade diferente, podendo desta forma otimizar o desempenho.

API da Localidade

A interface da localidade pode ser observada na Listagem 3.1, seguindo-se uma descrição das funcionalidades e da utilização das primitivas oferecidas, deixando para a Subsecção 3.3.2 a descrição das etapas internas ao *middleware* na resposta à invocação das operações.

```
public class Place implements PlaceInterface{

  public <R> Future<R> spawn(ConcurrentTask<R> task);
  public <R> Future<R> [] spawnTasks(ConcurrentTask<R> [] tasks);
  public <R> Future<R> invoke(String methodName, Object ...args)
              throws PlaceNoSuchMethod;

  protected <T> T copy(T elem);

  protected MonitorInterface createMonitor();
  public void beginAtomic();
  public void endAtomic();
  public Condition newCondition(ConditionCode code);

  public <T,P,R> R distReduce(Distribution<T> distr, Reduction<P,R> red,
            DistrRedTask<P> task);
  public <T,P,R> R distReduce(Distribution<T> distr, Reduction<P, R> red,
                        String methodName, Object ...args)
                        throws PlaceNoSuchMethod;
  public <T, R> void dist(Distribution<T> distribution, DistrRedTask<P> task);
}
```

Listagem 3.1: API da Localidade.

O programador para usufruir da API pode utilizar diretamente as localidades ou pode criar classes que estendam a classe `Place` para que cada uma usufrua das primitivas oferecidas. Contudo, existem algumas primitivas declaradas como `protected`. Estas primitivas não foram declaradas como públicas porque não faria sentido, que o programador no contexto de uma localidade, realizasse invocações dessas primitivas em outras localidades. Na descrição de cada primitiva declarada como `protected` será indicada a razão de ela não ser pública.

Gestão de Tarefas A primitiva `spawn` implementa o modelo de paralelismo nas tarefas, ou seja permite ao programador decompor as suas aplicações em tarefas e submete-las para execução na plataforma. Esta semântica é semelhante ao encontrado no X10 e no Chapel, em que o programador não cria e gere explicitamente *threads*, tendo apenas que criar e submeter as tarefas para execução. Desta forma, o programador preocupa-se apenas em expressar o paralelismo lógico da aplicação através de tarefas, deixando a gestão do paralelismo a cargo do *middleware*. Ao gerir explicitamente *threads*, o programador ficaria responsável pela correspondência entre o paralelismo da aplicação (tarefas) com os recursos disponíveis no sistema.

A invocação da primitiva *spawn* é não bloqueante, ou seja, o fluxo invocador não fica bloqueado até ser retornado o resultado da invocação. Em vez de bloquear, é retornado um `Future` explícito [BH77] ao programador. Um `Future` é um objeto que representa a execução assíncrona da computação e que atua como um *proxy* para o seu resultado, fornecendo uma primitiva bloqueante (`get()`) que será invocada pelo programador para obter o resultado da computação, apenas quando este for necessário.

Para criar uma tarefa, o programador deve criar uma classe que estenda a classe abstrata `ConcurrentTask<R>` e implementar o método `R call()` que será invocado aquando da sua execução. O tipo abstrato `R` representa o tipo de retorno da execução da tarefa. Na Listagem 3.2 é apresentado um exemplo da utilização desta primitiva que é um extrato da implementação do cálculo do número de Fibonacci, seguindo-se uma breve explicação.

```
1  class FibTask extends ConcurrentTask<Integer> {
2    //...
3    private Place place;
4    //...
5    public Integer call() {
6      //...
7      Future<Integer> f1 = place.spawn(new FibTask(n-1));
8      Future<Integer> f2 = place.spawn(new FibTask(n-2));
9      return f1.get() + f2.get();
10   }
11 }
```

Listagem 3.2: Exemplo de uma tarefa para o cálculo do número de Fibonnacci.

Como pode ser observado, o programador criou a classe `FibTask` que estende a classe abstrata `ConcurrentTask<Integer>` implementando o método `call()` que contém o código para realizar o cálculo. Portanto, na linha 7 e 8 é lançada a computação do número de Fibonacci `n-1` e `n-2` assincronamente, ficando o fluxo invocador à espera dos resultados na linha 9 para retornar o resultado da adição. Neste ponto da descrição da interface não se tem como objetivo realizar otimizações de desempenho em relação aos exemplos ilustrados.

A primitiva `spawnTasks` tem o mesmo comportamento do `spawn`, tendo apenas como diferença o facto de poder passar como argumento várias tarefas e de retornar um

array de objetos do tipo `Future`. Ela apenas existe por uma questão de simplicidade na programação.

Como foi referido na Subsecção 3.3.1.1, a interface de uma localidade pode ser estendida com novas operações. A primitiva `invoke` permite ao programador invocar uma dessas operações. Para isso, o programador deve passar como argumento o nome do método e a sequência de argumentos para a invocação. Por omissão, os argumentos são passados por referência, no entanto o programador poderá realizar passá-los por cópia. Caso o método não esteja definido na interface da localidade, uma exceção (`PlaceNoSuchMethod`) é lançada de forma a ser apanhada no código da aplicação. A desvantagem desta primitiva é o facto de não ser possível fazer a verificação de tipos em tempo de compilação, de forma a validar a correspondência entre os tipos passados como argumento e os tipos dos parâmetros definidos pelo método. Tal como na primitiva `spawn`, o retorno da invocação é um objeto `Future` onde o tipo abstrato `R` é o tipo de retorno da operação. Assumindo a existência de um método `fib` na interface da localidade referenciada por `place`, a Listagem 3.3 ilustra como esse método pode ser invocado.

Portanto, o programador com recurso à primitiva `invoke` passou como argumento o nome do método (`"fib"`) e o argumento da operação a invocar (`40`, que indica que o número a obter é o do índice 40 da sequência de Fibonacci), ficando com o resultado invocando explicitamente a primitiva bloqueante `get` do `Future` retornado.

```
try {
  Future<Integer> future = place.invoke("fib", 40);
  int res = future.get();
} catch (PlaceNoSuchMethod e) {
    //...
}
```

Listagem 3.3: Exemplo de uma invocação de uma operação especificada na interface.

A primitiva `copy` permite ao programador fazer uma cópia dos argumentos passados numa invocação, pois por omissão os argumentos são passados por referência. Para fazer a cópia, o programador deve invocar a primitiva passando como argumento o objeto a copiar. Na Listagem 3.4 é apresentado um exemplo de como o programador pode realizar uma cópia dos argumentos.

```
public int[][] matrixMult(int[][] m1, int[][] m2)
{
  m1 = copy(m1);
  m2 = copy(m2);
  //...
}
```

Listagem 3.4: Exemplo de uma cópia de argumentos.

Portanto, neste exemplo, o programador pretende fazer uma cópia dos argumentos (*arrays* `m1` e `m2`) que são passados na invocação do método `matrixMult`. Desta forma,

o corpo do método terá apenas acesso a uma cópia dos argumentos, ficando o fluxo invocador com o seu valor original. A razão pela qual esta primitiva está declarada como `protected` deve-se ao facto de não fazer sentido que uma localidade invoque esta primitiva noutra localidade para fazer uma cópia de objetos.

Controlo de concorrência O mecanismo de base para controlo de concorrência oferecido para a sincronização dentro de uma localidade é o monitor [Hoa74]. Através do monitor o programador define blocos de código que terão que ser executados atomicamente (blocos atómicos) e pode definir as pré-condições para a execução dos blocos atómicos. A sua interface é apresentada na Listagem 3.5.

```
interface Monitor {
  void beginAtomic();
  void endAtomic();
  Condition newCondition(ConditionCode code);
}
```

Listagem 3.5: Interface do monitor.

O programador para definir blocos atómicos deve recorrer às primitivas do monitor `beginAtomic` e `endAtomic`. A primitiva `beginAtomic` define o inicio de uma região atómica e a primitiva `endAtomic` define o fim dessa região. À semelhança da *keyword* `synchronized` do Java, esta abordagem permite flexibilidade na definição de partes de um método que tenham que ser executadas atomicamente.

Uma das alternativas para definir regiões atómicas poderia ser a invocação de uma primitiva, por exemplo, de nome `atomic` passando como argumento a implementação dessa região, como exemplificado na Listagem 3.6.

```
atomic(
  new Code<Integer>() {
    public Integer call() {
        //....
    }
  }
);
```

Listagem 3.6: Alternativa para definir um bloco atómico.

Porém, esta alternativa exigia a criação de objetos que executassem a região crítica, originando um grande impacto sobre o desempenho, se por exemplo essa região fosse um cálculo muito simples.

A definição de variáveis de condição é obtida através da primitiva `newCondition`, em que se deve passar como argumento um objeto do tipo `ConditionCode` que define a pré-condição de acesso à região atómica. O programador deve criar uma classe que estenda a classe abstrata `ConditionCode` e deverá implementar o método `Boolean call()` para especificar a condição.

Por omissão, cada localidade tem um monitor associado, à semelhança de um objeto na linguagem Java. Por uma questão de simplicidade de acesso, a interface da localidade oferece as primitivas que operam sobre esse monitor. No entanto, a estratégia de associar apenas um monitor a cada localidade limita o paralelismo dentro da localidade, pois apenas um fluxo de execução poderá executar dentro do monitor. Para aumentar o grau de paralelismo o programador pode criar novos monitores utilizando primitiva `createMonitor`, sendo-lhe devolvido um objeto `Monitor`.

Na Listagem 3.7 é apresentado um exemplo da definição de um bloco atómico, assumindo a existência de um método `update` na localidade.

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

Listagem 3.7: Exemplo da definição de um bloco atómico.

O método altera o valor do campo `val` de forma atómica. A região atómica engloba a leitura do valor corrente e pela sua modificação adicionando o valor do argumento que é passado.

Distribuição e Redução A interface para além de oferecer um modelo de paralelismo de tarefas, oferece também um modelo de paralelismo de dados. Este modelo é obtido a partir da execução de uma mesma tarefa por vários fluxos de execução concorrentemente, onde cada um opera sobre dados distintos.

A primitiva `<T,P,R> R distReduce` permite aplicar este modelo de paralelismo sobre uma mesma tarefa criada pelo programador ou na invocação de uma operação especificada na interface. O programador fica apenas responsável pela implementação das estratégias de distribuição e redução, ficando a cargo do *middleware* a realização das computações e do cálculo do resultado final. Este mecanismo é equivalente a um Map-Reduce [DG04].

A operação de distribuição decompõe o(s) argumento(s) de entrada da tarefa em sub-conjuntos do mesmo tipo. A tarefa é então aplicada em paralelo a cada sub-conjunto onde o resultado parcial obtido por cada computação é colecionado de forma a ser aplicado uma função de redução sobre esses resultados de forma a obter o resultado final. O tipo abstrato `T` indica o tipo de dados a decompor, o tipo abstrato `P` indica o tipo do resultado de cada computação e o tipo abstrato `R` indica o tipo do resultado final que é obtido a partir da combinação de resultados do tipo `P`. A interface a implementar pelo programador para definir esta estratégia é apresentada na Listagem 3.8.

A estratégia de distribuição, portanto, passa por aplicar a função de partição a uma estrutura de dados do tipo T que retorna um vetor de partições do tipo T.

```
interface Distribution<T> {
  T [] distribution();
}
```

Listagem 3.8: Interface da estratégia de distribuição.

A estratégia de redução passa por aplicar uma operação de combinação a todos os resultados parciais obtidos. A interface a implementar pelo programador para definir esta estratégia é apresentada na Listagem 3.9.

```
interface Reduction<P,R> {
  R getResult(Iterator<P> res);
}
```

Listagem 3.9: Interface da estratégia de redução.

O programador irá receber um iterador de resultados parciais do tipo P e irá implementar uma operação que combinará os resultados de forma a gerar um resultado final do tipo R.

De forma a exemplificar de que forma o programador deve utilizar o mecanismo de distribuição e redução será apresentado um exemplo da sua aplicação em que se mostrará as etapas que o programador deverá realizar. Este exemplo ilustra o cálculo do valor de π utilizando o método de Monte Carlo[1]. O programador em primeiro lugar deve criar a tarefa. Para tal, o programador deve então criar uma classe que estende a classe abstrata `DistrRedTask<P>`, que por sua vez estende o comportamento base da tarefa (classe `ConcurrentTask`) e implementar o método `call`. Na Listagem 3.10 é apresentada parte da implementação da tarefa.

```
1 public class PITask extends DistrRedTask<Double> {
2   public Double call() {
3     double iters = (Double) popArgument();
4     for (double i = 0; i <= iters; i++) {
5       //...
6     }
7     //...
8   }
9 }
```

Listagem 3.10: Tarefa utilizada no mecanismo de distribuição e redução para o cálculo de PI.

Em comparação com a implementação sequencial do método só existe uma diferença: a forma como se obtém o número de iterações a realizar. Se na versão sequencial o número de iterações é passado, por exemplo, como argumento no método que faz o cálculo, na tarefa o número de iterações a realizar é obtido através da primitiva `popArgument` (linha 3) definida na classe abstrata `DistrRedTask`. Esta primitiva permite obter uma das

[1] `http://www.stanford.edu/group/pandegroup/folding/education/montecarlo/`

partições geradas, em que para este exemplo em específico cada partição corresponde ao número de iterações a realizar pela tarefa.

Após a criação da tarefa, o programador deve então definir a estratégia de distribuição dos dados implementando uma classe que implementa a interface `Distribution` e definir o método `distribution`. A estratégia de distribuição (Listagem 3.11) dividirá um determinado número de iterações (`rounds`) em `n_parts` sub-conjuntos de iterações, como já foi referido. Neste exemplo, cada partição corresponde exatamente ao mesmo número de iterações.

```
public class DistributionPI implements Distribution<Double> {
  private double rounds;
  private int n_parts;

  public DistributionPI(double rounds, int n_parts){
    this.rounds = rounds;
    this.n_parts = n_parts;
  }

  public Double[] distribution() {
    Double [] partitions = new Double[n_parts];
    double iters = rounds / n_parts;
    for(int i = 0; i < n_parts; i++)
        partitions[i] = iters;
    return partitions;
  }
}
```

Listagem 3.11: Estratégia de distribuição a aplicar sobre os dados de entrada.

O programador também deve também definir a estratégia de redução (Listagem 3.12). A estratégia de redução passa aglomerar os resultados parciais de cada computação, cada um obtido pelo `Iterator` e pelo seu somatório, e fazer o cálculo do resultado final.

```
public class ReductionPI implements Reduction<Double, Double> {
  private double rounds;

  public ReductionPI(double rounds) {
    this.rounds = rounds;
  }

  public Double getResult(Iterator<Double> it) {
    double total_hits = 0;
    while(it.hasNext())
      total_hits+=it.next();

    return (4.0 * total_hits/rounds);
  }
}
```

Listagem 3.12: Estratégia de redução a aplicar para o cálculo final do valor de PI.

Após o programador ter definido as estratégias de distribuição e redução e ter implementado a tarefa, deverá então invocar a primitiva `distReduce` e obter o resultado final da forma como é ilustrado na Listagem 3.13.

```
public double getPI(double rounds, int n_parts) {

  Distribution<Pair<Double, Double>> dst = new DistributionPI(rounds, n_parts);
  Reduction<Double, Double> red = new ReductionPI(rounds);

  return distReduce(dst, red, new PITask());
}
```

Listagem 3.13: Aplicação do mecanismo de distribuição e redução para o cálculo do PI.

3.3.1.2 Anotações

A interface para as aplicações deve ser o mais concisa e simples possível e de preferência que não exija do programador uma mudança de paradigma na estruturação dos seus algoritmos. Apesar da interface descrita anteriormente oferecer abstrações de alto nível para expressar o paralelismo, ainda é exigido ao programador uma mudança no paradigma de programação a que está habituado. Nomeadamente, é lhe exigido que coloque o código que pode ser executado em paralelo dentro de uma tarefa e submete-la para execução, ou que defina as regiões do código que devem ser executadas em exclusão mútua sem se esquecer de colocar a primitiva de fim de região atómica.

Para minimizar o esforço do programador, foi desenvolvido um mecanismo de anotações que tem como base a filosofia do OpenMP, na medida em que, o programador anota o código da sua aplicação para expressar o paralelismo. As anotações têm como objetivo adicionar meta-dados ao longo do código de forma a que estes sejam posteriormente interpretados. As anotações ao serem interpretadas farão com que o código compilado pelo programador seja instrumentalizado de forma a adicionar as invocações na interface descrita anteriormente para expressar o paralelismo. Portanto, restringe-se a utilização destas anotações a classes que estendam o comportamento base da localidade (classe `Place`).

De seguida apresentam-se e descrevem-se as anotações atualmente existentes juntamente com alguns exemplos ilustrativos, deixando para o Capítulo 4 os detalhes de funcionamento do mecanismo de anotações e de implementação de cada uma das anotações.

@Task

Esta anotação deve ser utilizada para anotar métodos cuja execução pode ser realizada em paralelo na localidade corrente. O objetivo é permitir ao programador criar um método cujo fluxo de execução seja sequencial e anotá-lo caso ele possa ser executado em paralelo. O código original será instrumentalizado de forma a que o método seja convertido numa tarefa e que seja feita uma invocação à primitiva `spawn` para submeter esta

última para execução em paralelo. Desta forma, a invocação do método será realizada de forma assíncrona, ficando o seu retorno associado a um `Future`. Quando o resultado da invocação for necessário (por exemplo, numa leitura a uma variável local que guarda o valor da invocação) será invocada a primitiva bloqueante no `Future` para obter o resultado.

De forma a poder comparar a vantagem da utilização desta anotação, é apresentado na Listagem 3.14 um exemplo de forma a fazer uma comparação com o exemplo da Listagem 3.2, o esforço feito pelo programador para obter o mesmo resultado final.

```
1 @Task
2 public int fib(int n) {
3   //...
4   x = fib(n-1);
5   y = fib(n-2);
6
7   return x + y;
8 }
```

Listagem 3.14: Método que calcula o número de Fibonacci anotado.

Como pode ser observado, o programador desenvolveu o cálculo do número de Fibonacci de forma sequencial e anotou o método indicando que pode ser realizado em paralelo. As invocações recursivas presentes no corpo do método (linhas 4 e 5) são elas também realizadas em paralelo. O resultado obtido por cada invocação e que é guardado numa variável local é transformado num `Future`. Quando o resultado é necessário (na altura do retorno do método) é invocada a primitiva bloqueante `get` para cada `Future` guardado. A utilização desta anotação simplifica a tarefa do programador pois não lhe é exigido a criação e submissão da tarefa.

Os métodos originais anotados que cuja execução não retorne qualquer valor (`void`) são casos especiais. No caso dos métodos que tenham retorno, quando o seu valor é obtido por exemplo através da leitura de variáveis locais, em que a leitura é bloqueante, existe a garantia de que a execução acabou. No caso dos métodos que não tenham qualquer retorno não existe forma de tomar conhecimento de quando é que a execução termina. Portanto, nestes métodos, é necessário que o programador especifique quando é que o fluxo invocador deve verificar se a computação do método já completou e bloquear-se caso a computação ainda esteja em curso. Para tal, o programador deve invocar a primitiva `sync` que pertence à localidade. Ela permite ao programador indicar quando é que o fluxo invocador deve ficar à espera da terminação de todas as invocações anteriormente lançadas. A primitiva não está descrita na interface da localidade por não ser uma questão de desenho mas sim uma questão de implementação. A sua razão deve-se ao facto de não ser possível anotar os corpos dos métodos e portanto foi criada esta primitiva para ter o mesmo efeito.

Na Listagem 3.15 é apresentado um exemplo da utilização da primitiva `sync`, onde o exemplo apresenta a implementação (parcial) do algoritmo de ordenação *MergeSort*.

```
1  @Task
2  public void mergesort(int[ ] data, int first, int n){
3    //...
4    if (n > 1)
5    {
6      //...
7      mergesort(data, first, n1);
8      mergesort(data, first + n1, n2);
9
10     sync();
11
12     merge(data, first, n1, n2);
13   }
14 }
```

Listagem 3.15: Método que executa o algoritmo MergeSort anotado.

Como pode ser observado, nas linhas 7 e 8 o algoritmo divide o vetor em duas partes e faz duas invocações recursivas do algoritmo para cada um das partes. Na linha 12, a execução do método `merge` só pode ser realizada quando as invocações anteriormente lançadas tenham efetivamente terminado. Para tal, o programador colocou a invocação ao método `sync` na linha 10 para garantir que as invocações assíncronas lançadas terminem antes da invocação ao método `merge`.

@Atomic

Esta anotação destina-se apenas a métodos cuja execução tenha que ser realizada em exclusão mútua. Na Listagem 3.16 é apresentado um exemplo da utilização desta anotação. Comparativamente com o exemplo da Listagem 3.7 o programador simplesmente se preocupou na implementação da lógica do método e anotou-o para indicar que ele deve ser executado em exclusão mútua, eliminado a questão do programador poder omitir o `endAtomic`.

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

Listagem 3.16: Exemplo da especificação de um método atómico utilizando a anotação Atomic.

Se o programador quiser flexibilidade de forma a sincronizar apenas uma parte do método, deverá então recorrer às primitivas de sincronização ou colocar essa parte do método num novo método anotado e invocá-lo no método original.

@Copy

Como referido na Subsecção 3.3.1, o programador ao passar os argumentos para a invocação de uma operação da localidade poderá copiar os argumentos passados. Para isso, o programador deverá anotar os parâmetros na definição do método que será invocado, como no exemplo apresentado na Listagem 3.17.

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

Listagem 3.17: Exemplo de uma cópia de argumentos utilizando a anotação.

Neste exemplo, as variáveis locais `m1` e `m2` são copiadas fazendo com que o método aceda apenas às cópias das variáveis.

3.3.1.3 Inicialização da computação

O programador para construir uma aplicação, recorrendo às localidades, tem duas formas de o realizar: interagir diretamente com as instâncias das localidades ou criar classes que estendam o comportamento base da localidade (classe `Place`) de forma a ter localidades com funcionalidade específicas. Considere o exemplo da criação de duas classes da aplicação, apresentado na Listagem 3.18, que permite demonstrar como é que o programador deve estender as localidades com funcionalidades especificas e como as deve utilizar.

```
public class MathProvider extends Place implements Math {
  int fib(int n) {
    //...
  }
}
-----------------------------------------------------------------
public class Client extends ActivePlace {
  private PlaceInterface service;
  //...
  public Client(PlaceInterface service) {
    this.service = service;
  }

  public void main() {
    try {
      Future<Integer> future = service.invoke("fib", n);
      int fib = future.get();
    } catch (PlaceNoSuchMethod e) {
      //...
    }
  }
}
```

Listagem 3.18: Definição de duas localidades.

O programador definiu uma classe `MathProvider` que estende a base da localidade com métodos definidos na interface `Math`. Como tal, esta classe vai comportar-se como um serviço.

O programador definiu igualmente uma classe `Client` que por sua vez estende a classe `ActivePlace`. Esta classe, que ainda não tinha sido descrita, estende a classe `Place` mas que obriga à implementação de um método `main` por parte do programador. O método `main` funcionará como o ponto de entrada da aplicação iniciando toda a computação. No exemplo, a implementação do método `main` realiza a invocação do método "`fib`" definido na interface da localidade `MathProvider`. Portanto, é necessário que a fase de inicialização das localidades presentes numa computação que na instanciação de cada localidade lhe sejam passadas as referências de todas as localidades sobre as quais ela irá fazer invocações.

Para ser possível inicializar a computação, nomeadamente inicializar o *middleware* e adicionar as localidades que farão parte da computação, é necessário recorrer à interface apresentada na Listagem 3.19.

```
interface Launcher {
  void init();
  void init(int n_workers);
  int addPlace(PlaceInterface place);
  void stop();
}
```

Listagem 3.19: Interface do Launcher.

A primitiva `init` deve ser invocada quando o objetivo é inicializar toda a plataforma de execução. A primitiva que tem um argumento definido indica que o *middleware* irá suportar uma *pool* de *threads* com `n_workers` que será partilhada pelas localidades. A primitiva `addPlace` adiciona uma localidade à computação. A primitiva verifica se a localidade é uma localidade ativa (`ActivePlace`) e caso seja, invoca o método `main` para ser executado concorrentemente. Caso contrário, não precisa de realizar qualquer invocação. A primitiva `stop` encerra toda a plataforma de execução.

3.3.2 Middleware

O *middleware* tem como objetivo esconder a heterogeneidade (hardware, sistemas de operação, protocolos de comunicação, etc.) e fazer a gestão do paralelismo, escondendo os detalhes de baixo nível do programador de toda a computação realizada. O núcleo do *middleware* encontra-se dividido por módulos (Figura 3.4) em que cada um é responsável por suportar uma determinada funcionalidade. O conjunto de módulos e o conjunto de ações que realizam perfazem o suporte à execução de aplicações paralelas.

Cada um dos módulos que integra o núcleo suporta uma funcionalidade e recorre à camada de *drivers* para a sua implementação. Desta forma, cada módulo pode ter dependências ao nível dos *drivers*. Na Figura 3.5 podem ser observadas as dependências dos módulos em relação aos *drivers*.

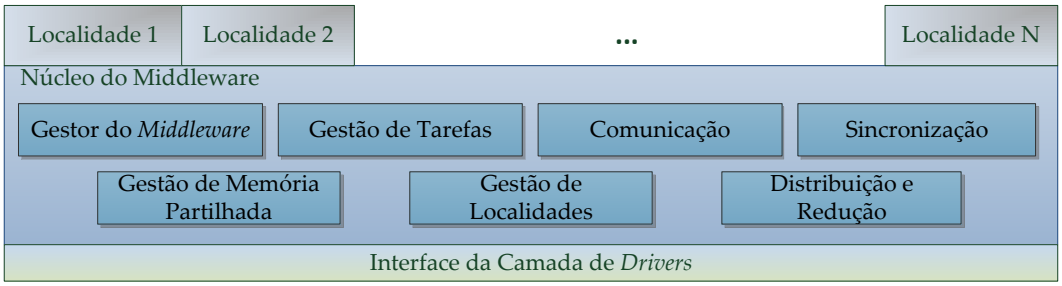


Figura 3.4: Núcleo do *middleware*.

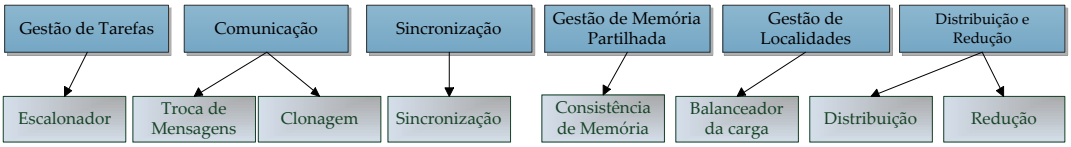


Figura 3.5: Dependência dos módulos ao nível dos *drivers*.

De seguida são apresentados e descritos os módulos que constituem a infraestrutura *middleware* que suportam as funcionalidades necessárias.

Gestor do *Middleware*

Este módulo é responsável por todas as inicializações necessárias para a construção do ambiente de execução e por adicionar e remover nós em tempo de execução. A fase de inicialização do sistema requer a definição dos nós que irão constituir a computação, a inicialização de todos os módulos em cada instância do *middleware* que irá executar em cada nó e o mapeamento das localidades nesses nós.

Gestão de Localidades

A localização física das localidades é transparente para o programador. O *middleware* poderá mesmo migrar as localidades, sem repercussões na estrutura das aplicações, de forma a melhorar o balanceamento da carga. A responsabilidade deste módulo passa por avaliar a carga do sistema e consoante uma heurística pré-definida, ele poderá migrar a computação de nó, sendo que uma computação está sempre associada a uma localidade em particular. O sistema é paramétrico relativamente ao algoritmo de balanceamento, sendo este incorporado via *driver*, onde o algoritmo atua consoante a heurística pré-definida.

Gestão de Tarefas

Este módulo tem como responsabilidade fazer toda a gestão do trabalho submetido (na forma de tarefas) para execução por parte das localidades. Uma tarefa é uma unidade de trabalho que está desacopulada do fluxo de execução que a irá executar e pode ser executada em paralelo independentemente das outras tarefas presentes. Este módulo torna transparente para as localidades, e consequentemente para as aplicações, os recursos computacionais existentes e de que forma eles são utilizados para a execução das tarefas.

Uma localidade tem associada uma *pool* de *threads* que pode ser partilhada com as outras localidades. Caso seja partilhada com outras localidades, ela será suportada pela instância do *middleware* que suporta essas localidades. A *pool* de *threads* tem uma fila de tarefas associada. A fila será consumida e será atribuída uma tarefa a um *thread* disponível presente na *pool* para a sua execução.

A interface do módulo de gestão de tarefas é apresentada na Listagem 3.20.

```
interface TaskManager {
  <R> Future<R> executeTask(TaskInterface<R> task);
  <R> Future<R>[] executeTasks(TaskInterface<R>[] tasks);
  boolean freeThreads();
  void shutdown();
}
```

Listagem 3.20: Interface do módulo de gestão de tarefas.

Este módulo será utilizado pelos métodos presentes na interface da localidade para submeter trabalho. A primitiva `spawn` e o `invoke` invocam a primitiva `executeTask` passando-lhe um objeto que implemente a interface `TaskInterface<R>`. Na primitiva `spawn`, como já foi referido na Subsecção 3.3.1.1, o programador submete um objeto que estende o comportamento de `ConcurrentTask`. Este por sua vez implementa a interface `TaskInterface` que obriga à definição do método `call`. Em relação à primitiva `invoke`, é criado internamente um objeto que implementa a interface `TaskInterface` em que o seu código realiza a invocação pretendida. Esta primitiva recebe a tarefa e submete-a para execução, retornando o `Future` à localidade para ser entregue ao programador.

A primitiva `spawnTasks` definida na localidade invoca a primitiva `executeTasks` passando um *array* de tarefas. O `executeTasks` tem o mesmo comportamento do `executeTask` tendo apenas a diferença de submeter várias tarefas para execução.

A primitiva `freeThreads` permite saber se existem *threads* disponíveis na *pool*. Apesar de não estar contemplado no desenho da localidade, esta primitiva é invocada pela localidade. A sua utilidade será descrita no Capítulo 4.

A primitiva `shutdown` será invocada pelo `Launcher` quando toda a infraestrutura for encerrada. O encerramento do módulo de gestão de tarefas faz com que ele rejeite novas tarefas para execução, mas continua a executar todas as tarefas pendentes.

Sincronização

Este módulo é responsável por oferecer mecanismos de sincronização entre tarefas dentro de uma localidade. É necessário oferecer mecanismos de sincronização pois a região de memória privada da localidade pode ser concorrentemente acedida pelas várias tarefas a executar nessa localidade. Dessa forma, é necessário que o programador defina uma disciplina de acesso aos dados partilhados. Para tal, o programador deve recorrer ao monitor que é fornecido pela interface, como foi referido na Subsecção 3.3.1.1.

A interface do módulo é apresentada na Listagem 3.21, seguindo-se uma descrição do método.

```
interface SyncModule {
  Monitor createMonitor();
}
```

Listagem 3.21: Interface do módulo de sincronização.

A primitiva `createMonitor` será invocada pela localidade sempre que é necessário criar e associar um monitor a essa localidade. Na fase de inicialização da localidade, a primitiva será invocada no seu construtor para que lhe seja associado um monitor por omissão. Ela também será também invocada pela localidade sempre que o programador deseje associar novos monitores à localidade. Neste último caso, a interface da localidade oferece a primitiva `createMonitor` que por sua vez irá invocar a primitiva definida pelo módulo.

Esta primitiva retorna um objeto que é uma implementação do monitor, fazendo com que este módulo funcione como uma fábrica (*factory*) de monitores. A sua implementação é delegada no *driver* de sincronização o que permite ter várias implementações do monitor em que cada implementação pode recorrer a diferentes mecanismos, como por exemplo *locks* ou memórias transacionais.

Comunicação

Este módulo é responsável por suportar os meios de comunicação necessários para a cooperação entre tarefas a executar em localidades distintas.

Caso uma localidade pretenda comunicar com outra que partilha o mesmo espaço de endereçamento físico, então a comunicação entre si é realizada através do mecanismo de invocação de métodos da linguagem, não sendo necessário recorrer a este módulo. Na Figura 3.6 é apresentada a invocação da primitiva `spawn` numa localidade que partilha o mesmo espaço de endereçamento físico, ilustrando como é que essa comunicação é efetuada.

Caso as localidades não partilhem o mesmo espaço de endereçamento físico, a comunicação entre elas terá que ser efetuada pela rede via troca de mensagens. Independentemente de uma localidade ser remota ou não, ela apresenta a mesma interface ao programador. Isto é obtido a partir da geração de *stubs* para a representação das localidades remotas. Um *stub* atua como um *proxy* entre a localidade representante e as

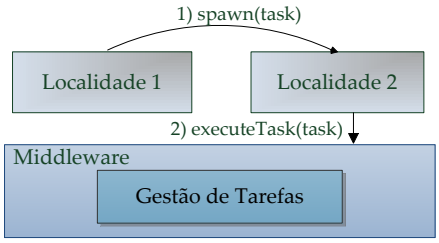


Figura 3.6: Comunicação entre localidades no mesmo nó.

localidades remota. Como tal, ela tem a mesma interface da localidade mas a implementação das operações reencaminha simplesmente os pedidos (toda a informação de uma invocação, como o nome do método e respetivos argumentos) para a localidade remota ficando à espera do resultado, caso exista. Este mecanismo é semelhante ao encontrado nas tecnologias RPC/RMI [Wal98].

O *stub* recorre ao módulo de comunicação para que este suporte a comunicação entre ele e a localidade que representa. O módulo fornece as primitivas necessárias para a realização dessa comunicação. A interface do módulo é apresentada na Listagem 3.22.

```
interface CommunicationModule {
  void init();
  <C> Message<C> receiveMessage();
  <C> void sendMessage(C content, MessageTag tag, int destination);
}
```

Listagem 3.22: Interface do módulo de comunicação.

A primitiva `init` é invocada pelo *middleware* para realizar todas as inicializações necessárias para iniciar o módulo, como inicializar um *daemon* que irá tratar da receção de mensagens. A primitiva `sendMessage` é invocada pelo *stub* de forma a enviar uma mensagem para a localidade representada por este. Os seus argumentos são compostos pelo conteúdo da mensagem, pelo seu tipo e pelo identificador da localidade de destino da mensagem. A primitiva `receiveMessage` será invocada pelo *middleware* de forma a que o módulo inicie o processo de receção de mensagens de forma a reencaminhar os pedidos para as localidades.

O módulo de comunicação faz uso de um *driver* de comunicação por troca de mensagens permitindo assim que a comunicação possa ser implementada com recurso a diferentes bibliotecas como por exemplo MPI ou JMS [MHC01]. O módulo antes de iniciar qualquer comunicação obtém um objeto que implementa a comunicação para a localidade desejada, implementando as primitivas definidas na interface do *driver*. A interface deste *driver* pode ser vista na Listagem 3.22.

A primitiva `init` é invocada pelo módulo de forma a fazer todas as inicializações necessárias do *driver*. A primitiva `send` é invocada pelo módulo de forma a enviar a mensagem à localidade de destino. A mensagem, que é um objeto do tipo `Message`, é

constituída por uma *tag* que define o seu tipo, pelo seu conteúdo e pelo identificador do destinatário. A primitiva `receive` é invocada pelo módulo para implementar o processo de receção de mensagens.

```
interface CommunicationDriver {
  void init();
  <C> void send(Message<C> message);
  <C> Message<C> receive();
}
```

Listagem 3.23: Interface do *driver* de comunicação.

A Figura 3.7 retrata o exemplo de uma submissão de uma tarefa para execução numa localidade remota, através da invocação da primitiva `spawn`, ilustrando todas as interações necessárias para a comunicação entre as localidades remotas.

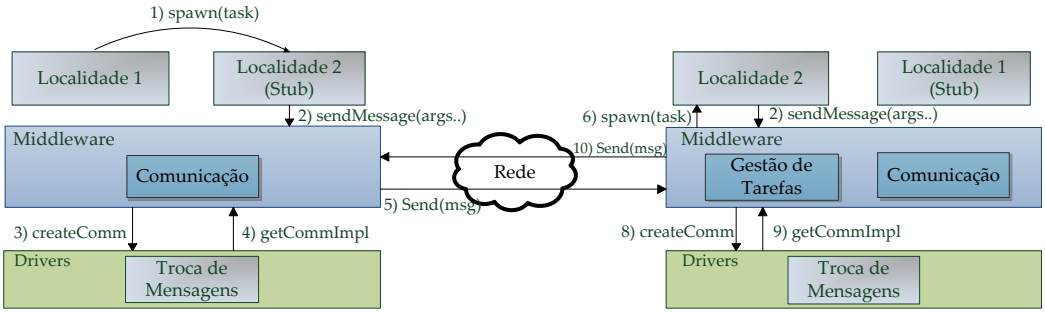


Figura 3.7: Comunicação entre localidades residentes em nós distintos.

Como pode ser observado pela figura, a *Localidade 1* pretende submeter trabalho na *Localidade 2*. A *Localidade 2*, sendo uma localidade remota, é representada por um *stub* no espaço de endereçamento da localidade invocadora. Portanto, o *stub* deve recorrer ao módulo de comunicação para que a tarefa seja enviada para a localidade remota. O módulo de comunicação, por sua vez, faz utilização do *driver* para que a implementação disponível envie a mensagem com a tarefa.

Por omissão, os argumentos são passados por referência quando o emissor e recetor partilham um mesmo espaço de endereçamento físico, sendo passados por cópia caso contrário. Para tal o *middleware* recorre ao *driver* de clonagem para criar uma cópia do argumento. A motivação para relegar a clonagem para a camada de *drivers* prende-se com a possibilidade de permitir diferentes estratégias como: *deep-copy*, *shallow-copy* ou *lazy-copy*. O *deep-copy* consiste em fazer uma cópia dos valores que compõem o argumento. Por exemplo, caso o argumento seja um objeto, os valores de todos os campos que o compõem são efetivamente copiados. O *shallow-copy* consiste em copiar apenas as referências para os objetos e não o valor. Usando este tipo de cópia, tanto como a referência original como a copiada ficam a apontar para o mesmo valor. O *lazy-copy* consiste em

fazer uma cópia dos valores caso estes sejam efetivamente modificados durante a computação. Caso os argumentos sejam apenas acedidos para leitura, então não será realizada qualquer cópia do argumento. A interface do *driver* de cópia é apresentado na Listagem 3.24.

```
interface CloningDriver {
  <T> T copy(T object);
}
```

Listagem 3.24: Interface do *driver* de clonagem.

A implementação de cada estratégia deve implementar esta interface, fornecendo uma definição do método `copy`, aplicando a estratégia sobre o objeto passado como argumento.

Gestão de Memória Partilhada

Uma localidade pode conter uma região de memória acessível a todas as suas tarefas ativas. Como tal, a comunicação entre as tarefas é realizada sob o paradigma de memória partilhada. Esta região de memória pode ser fisicamente partilhada mas também pode ser logicamente partilhada pois uma localidade pode estar distribuída por um ou mais nós. Como as posições de memória serão acedidas e possivelmente modificadas concorrentemente por vários fluxos de execução é necessário definir um modelo de consistência de memória.

Um modelo de consistência define a ordenação das escritas e leituras sobre os dados partilhados que será visto por todos os fluxos de execução, de forma a manter a coerência dos dados. O modelo de consistência sequencial garante que o próximo valor lido de uma determinada variável é o valor da última escrita efetuada. Embora este modelo seja o habitual em multi-processadores de memória partilhada física, esta garantia não existe em ambientes distribuídos devido a limitações físicas. Desta forma, terão que existir modelos de memória mais relaxados do que o sequencial. Os modelos mais relaxados permitem que os vários fluxos de execução possam ver, até certo ponto, valores inconsistentes tendo o programador conhecimento de que isso poderá acontecer.

Na situação onde a região de memória é logicamente partilhada é necessário ter uma implementação que simule essa região de memória, oferecendo primitivas que implementem as operações de consulta e atualização de variáveis partilhadas e que aplique um modelo de consistência relaxado. Este módulo será responsável por fazer a gestão dessa memória oferecendo primitivas para a leitura e escrita das variáveis. Para tornar isso possível foi criado o conceito de `SharedVariable`. Apesar de não estar descrita na interface da localidade por não estar implementada, será oferecida ao programador uma primitiva que lhe permite encapsular os dados partilhados das localidades num objeto `SharedVariable`, associando-o à localidade. Ele tem como objetivo aplicar um modelo de consistência de memória dos acessos de escrita e leitura consoante a natureza do ambiente. Portanto a sua interface, terá primitivas para ler e modificar o seu valor, como é

ilustrado na Listagem 3.25.

```
interface SharedVariable<V> {
  void write(V val);
  V read();
}
interface SharedVariable2d<V> {
  void write(V val, int x, int j);
  V read(int x, int j);
}
```

Listagem 3.25: Interface do SharedVariable.

Como pode ser observado, existem duas interfaces: `SharedVariable` para encapsular variáveis de tipos simples e `SharedVariable2d` que servirá para encapsular variáveis como *arrays*. O programador terá em mãos objetos deste tipo e terá que invocar a primitiva `get` para obter o valor e invocar a primitiva `set` para modificar o valor.

As primitivas definidas no `SharedVariable` irão recorrer ao módulo para que faça a gestão da variável. A interface do módulo é apresentada na Listagem 3.26.

```
interface SharedMemoryModule {

  <V> void registSharedVariable(SharedVariable<V> sv, Place place);
  <V> void registSharedVariable(SharedVariable2d<V> sv, Place place);

  <V> V get(SharedVariable<V> sv);
  <V> V get(SharedVariable2d<V> sv);

  <V> void set(SharedVariable<V> sv, V value);
  <V> void set(SharedVariable2d<V> sv, V value);
}
```

Listagem 3.26: Interface do módulo de gestão de memória partilhada.

A primitiva `registSharedVariable` será invocada pela localidade para registar a variável partilhada passada como argumento. A referência da localidade também terá que ser passada para que o módulo construa um mapa onde associa para cada localidade todas as variáveis partilhadas registadas. A primitiva `get` será invocada pelo `SharedVariable` de forma a obter o valor mais recente[2]. A primitiva `get` será invocada pelo `SharedVariable` de forma a modificar o valor da variável.

O módulo aplica um modelo de consistência de memória consoante a natureza do estado da localidade. Se o estado da localidade for fisicamente partilhado ele aplica um modelo mais restrito; caso contrário aplica um modelo mais relaxado. Esse modelo é delegado no *driver* respetivo, permitindo desta forma implementar vários modelos mais estritos ou mais relaxados.

[2]Pode não corresponder ao valor da última atualização realizada.

Distribuição e Redução

Este módulo é responsável por suportar um paradigma de paralelismo nos dados, sendo esta uma dimensão fulcral na decomposição paralela de problemas, através de um mecanismo de distribuição e redução ao nível da tarefa. O seu uso é vantajoso em situações onde o programador deseje aplicar uma tarefa sobre um grande aglomerado de dados. Um exemplo clássico é uma pesquisa (tarefa) do número de ocorrências de cada palavra de um texto sobre um grande conjunto de ficheiros (dados).

O *middleware*, recorrendo a este módulo, abstraí o programador da gestão da computação e do cálculo do resultado final. Como foi referido na secção 3.3.1.1, o programador apenas terá que indicar a estratégia de distribuição dos dados, a estratégia de redução dos resultados parciais obtidos e a tarefa, ou a operação, a executar. A interface deste módulo é apresentado na Listagem 3.27.

```
interface DistReduce<T,P,R>
{
  R getResult(Distribution<T> distr, Reduction<P,R> red,
              DistRedTask<P> task, PlaceInterface place);
}
```

Listagem 3.27: Interface do módulo de distribuição e redução.

Os métodos `distReduce` definidos na interface da localidade irão realizar uma invocação sobre a primitiva especificada pelo módulo. A primitiva recebe as estratégias de distribuição e redução a aplicar, a tarefa a executar sobre cada partição e a localidade sobre o qual o mecanismo irá ser aplicado.

Na Figura 3.8 são apresentadas as etapas realizadas pelo mecanismo, seguindo-se uma descrição de cada uma das etapas:

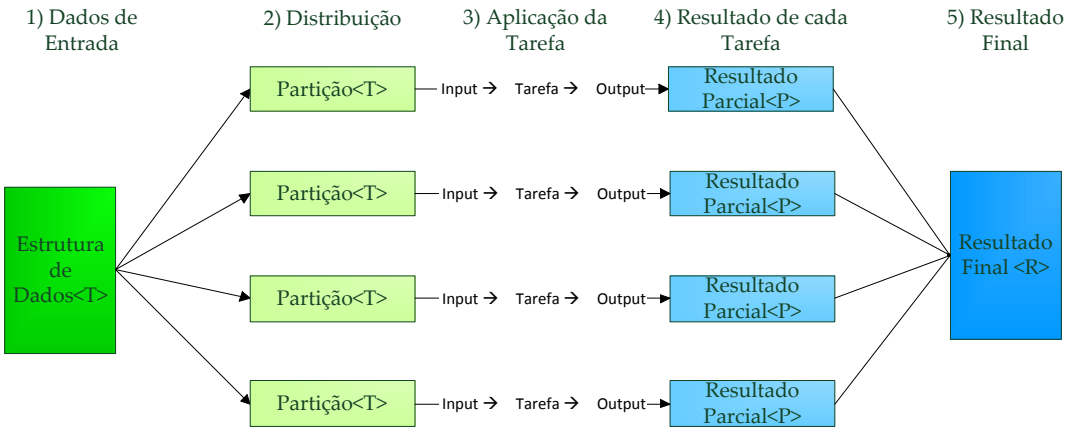


Figura 3.8: Mecanismo de Distribuição e Redução.

1) Os dados de entrada, sendo tipicamente uma estrutura de dados, são particionados a partir de uma função definida pela estratégia de distribuição especificada por parte do programador;

2) A aplicação da estratégia de distribuição resulta na obtenção de vários sub-conjuntos do mesmo tipo dos dados de entrada (cada sub-conjunto é designado por partição);

3) Um conjunto de tarefas irá executar paralelamente e de forma independente entre si, onde cada tarefa irá operar sobre partições distintas não sendo necessário, portanto, qualquer sincronização durante a sua computação;

4) Quando cada uma das computações terminar, o resultado parcial obtido pela sua execução será colecionado num repositório;

5) A partir do repositório de resultados, os resultados parciais são obtidos e é lhes aplicada uma função de combinação que foi definida pela estratégia de redução especificada por parte do programador.

O mecanismo recorre a um *driver* que implementa estas etapas sobre uma determinada localidade. Desta forma, o *driver* pode ter várias implementações, podendo mesmo ter uma implementação deste mecanismo utilizando GPUs. A interface que cada especialização terá que implementar é apresentada na Listagem 3.28.

```
interface DistrReduce<T,P,R>
{
  R getResult(Distribution<T> distr, Reduction<P, R> red,
          DistrRedTask<P> task, PlaceInterface place);
}
```

Listagem 3.28: Interface do *driver* de distribuição e redução.

4

Implementação

Neste capítulo é descrita a implementação do *middleware* que pretende atingir os objetivos propostos no contexto desta dissertação. Na Secção 4.1 são descritas as razões que fizeram com que a linguagem Java fosse escolhida para a implementação da plataforma; na Secção 4.2 é descrita a implementação dos vários módulos que constituem a infraestrutura e a sua concretização atual para arquiteturas de memória partilhada; por último, na Secção 4.3 são descritos todos os detalhes da implementação do mecanismos de anotações oferecido.

4.1 Introdução

A escolha da linguagem para instanciar a plataforma recaiu na linguagem Java (Versão 6). A principal razão que levou à escolha desta linguagem prende-se com o facto de esta ser uma linguagem de referência para o desenvolvimento de aplicações. Por essa razão, pretende-se contribuir para a sua comunidade com meios para que os seus programadores possam beneficiar das arquiteturas atuais sem uma grande curva de aprendizagem.

O recurso a esta linguagem teve também vantagens não só no desenho que foi pensado numa lógica orientada a objetos, como também na própria implementação do *middleware*. O facto de o Java ser uma linguagem que instancia o paradigma de programação orientado a objetos, permitindo estruturar as aplicações por módulos e podendo re-aproveitar código, é importante pois permitiu a implementação do *middleware* por módulos e a possibilidade de incorporar várias especializações das funcionalidades suportadas. Outro dos aspetos que se revela essencial é o facto de a linguagem ter suporte para *multi-threading*, permitindo assim que vários fluxos de execução possam coexistir

durante as computações. Outra das vantagens é o facto de as suas aplicações serem independentes da plataforma, o que se revela importante pois desta forma o *middleware* pode ser instalado em diferentes máquinas. A portabilidade obtém-se a partir da utilização de uma máquina virtual Java (JVM) que interpreta código intermédio (Java *Bytecode*), que é gerado pelo compilador, sendo este independente da plataforma. A integração da JVM no ambiente de execução permite que o *middleware* usufrua dos mecanismos oferecidos pelos sistemas de operação, tais como escalonamento de processos/*threads*, gestão de memória virtual e gestão de *caches*, o que é essencial para os fins pretendidos. Outra das grandes vantagens da JVM é o facto de possuir um *garbage collector* o que permite a gestão automática da memória num nó.

4.2 Middleware

A implementação atual do *middleware* contempla o seu suporte apenas para arquiteturas de memória partilhada, apesar do seu desenho, que foi descrito no Capítulo 3, contemplar o seu suporte para ambientes distribuídos. O suporte para arquiteturas de memória partilhada resulta de uma implementação base da camada de *drivers* que especializam as funcionalidades indispensáveis para o seu suporte neste tipo de arquiteturas. O suporte para ambientes distribuídos, como *clusters*, encontra-se atualmente em desenvolvimento[1]. Será realizada a implementação dos módulos de comunicação e gestão de memória partilhada, apesar de o respetivo desenho estar considerado, como foi apresentado no Capítulo 3. Também será realizada uma extensão à camada de *drivers*, atualmente existente, para incorporar os *drivers* necessários, nomeadamente os *drivers* de comunicação, modelo de consistência de memória, balanceamento de carga e escalonador global de trabalho.

Na Figura 4.1 é apresentado o desenho do núcleo do *middleware* onde se pode se pode observar os módulos que o constituem e os respetivos *drivers* que constroem o seu suporte. De seguida serão descritos os detalhes de implementação dos módulos que constituem o *middleware* e dos *drivers* concretizados para a sua execução.

Gestor do *Middleware*

Atualmente o ambiente de execução contempla apenas uma JVM, que suporta uma instância do *middleware*. Por sua vez, cada instância do *middleware* suporta um número finito de localidades que serão usadas para a computação. O processo de inicialização passa por iniciar todos os módulos que constituem o *middleware* e de todas as localidades. Este módulo será também responsável por adicionar e remover nós em tempo de execução, mas como o âmbito desta dissertação dirige-se apenas a arquiteturas de memória partilhada o seu suporte não foi implementado.

[1] `http://citi.di.fct.unl.pt/postgrad/postgrad.php?id=459`

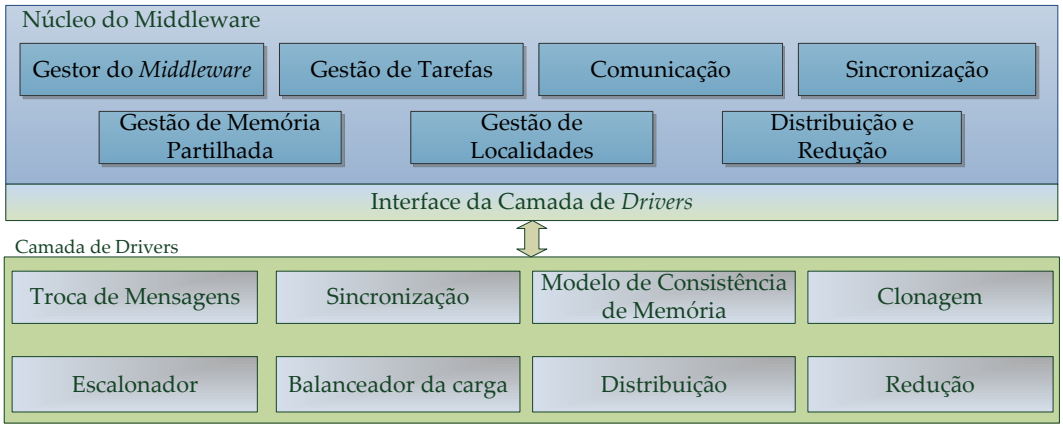


Figura 4.1: Desenho do núcleo do *middleware*.

A inicialização dos módulos, na prática, passa por obter os *drivers* escolhidos que serão utilizados pelo módulo e inicializá-los. Os *drivers* escolhidos para fazerem o suporte do *middleware* para determinado ambiente de execução serão escolhidos através da definição de um XML. A razão pela qual se recorre à definição de um XML, em detrimento de outras possíveis abordagens como a criação de um ficheiro *properties* do Java, deve-se ao facto de ser possível restringir o conteúdo do mesmo. O XML terá que ser validado perante um XML *Schema* que foi definido e que restringe o seu conteúdo. Isto permite que não se possam utilizar *drivers* que não estejam disponíveis, para além de evitar erros sintáticos no ficheiro. Quando um determinado *driver* for implementado e puder ser utilizado, então o XML *Schema* terá que ser alterado para que no XML se possa indicar o seu uso. Na Listagem 4.1 é apresentado um exemplo de um XML que define quais os *drivers* a utilizar.

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="sable" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                                  xsi:schemaLocation="sable␣config.xsd">
  <drivers>
    <communication>org.sable.middleware.drivers.comm.MPIImpl</communication>
    <synchronization>org.sable.middleware.drivers.sync.Locks</synchronization>
    <copy>org.sable.middleware.drivers.cloning.Deep</copy>
    <consistencyModel>org.sable.middleware.drivers.sm.Strict</consistencyModel>
    <distrReduce>org.sable.middleware.drivers.dr.Base</distrReduce>
  </drivers>
</config>
```

Listagem 4.1: Exemplo de um XML que especifica que *drivers* devem ser utilizados.

Com pode ser observado, cada *driver* é especificado pelo nome absoluto da classe de implementação. Neste exemplo, o *driver* de comunicação a utilizar é o *driver* que recorre a um *binding* de MPI para Java, para o de sincronização é escolhida a implementação do monitor baseada em *locks*, o de cópia recorre à implementação da estratégia de *deepCopy*,

para o modelo de consistência de memória escolhe-se o modelo estrito e finalmente para a implementação do mecanismo e distribuição é escolhida uma implementação base.

Para a inicialização de toda a computação, terá que existir um passo de inicialização das localidades que constituirão a computação. Para isso, recorre-se à interface da classe `Launcher`, que foi apresentada no Capítulo 3, para iniciar a infraestrutura *middleware* e para adicionar as localidades. Na Listagem 4.2 é apresentado um exemplo de inicialização que pretende inicializar as localidades definidas pelo programador apresentadas na Listagem 3.18.

```
public static void main(String[] args) {
  Launcher.init(N_WORKERS);

  Place math = new MathProvider();
  ActivePlace client = new Client(math);

  Launcher.addPlace(math);
  Launcher.addPlace(client);

  Launcher.stop();
}
```

Listagem 4.2: Inicialização da computação.

Como pode ser observado, foi definido um método `main` do Java para que seja possível inicializar toda a plataforma de execução. A primitiva `init` definida no `Launcher` é então invocada para se iniciar o *middleware*. O argumento passado (`N_WORKERS`) indica que existirá uma *pool* de *threads* partilhada pelas localidades com `N_WORKERS` *threads* disponíveis. Após a inicialização do *middleware*, as localidades presentes na computação são instanciadas e são adicionadas ao *middleware* com recurso à primitiva `addPlace` também definida no `Launcher`. A implementação do `addPlace` adiciona a localidade e verifica se a localidade passada como argumento é uma localidade ativa. Se for, ele invoca a primitiva `main` de forma a ativar a localidade. A primitiva `stop` é invocada para encerrar toda a plataforma de execução.

Gestão de Tarefas

Como foi referido na Subsecção 3.3.1.1, o processo de submissão de trabalho pode resultar na invocação de uma operação específica dessa localidade ou pode ser explícito, no qual o programador cria e submete uma tarefa para ser executada em paralelo. Em qualquer dos casos, o objeto passado para o gestor tem que implementar a interface `TaskInterface`. A interface `TaskInterface` estende a interface `Callable` do pacote `java.util.concurrent` e portanto herda o método `call` definido pelo `Callable`.

Na invocação de uma operação, é gerado um *handler* que representa a descrição da invocação do método pretendido, no qual se incluí o nome do método e os respetivos argumentos. Para a sua concretização é utilizada uma classe `Handler` que implementa

a interface `TaskInterface<R>`, onde o tipo abstrato `R` representa o tipo de retorno da operação. A implementação do `R call()` recorre ao pacote Reflection API [Mic] oferecido pelo Java. Este pacote torna possível, por meio da sua interface, a inspeção a classes, interfaces, campos e métodos em tempo de execução sem conhecimento prévio destes em tempo de compilação. Também torna possível instanciar novos objetos, invocar métodos e modificar campos de classes, tudo em tempo de execução. A implementação do `call`, com o recurso à Reflection API e com a descrição do método a invocar, torna possível a invocação desejada na interface da localidade, retornando o resultado obtido pela invocação.

Na submissão explicita de trabalho, o programador submete um objeto que é uma extensão da classe abstrata `ConcurrentTask` e que implementa o método `call()`. Esta classe abstrata comporta uma implementação base da tarefa e implementa a interface `TaskInterface` e portanto o `call` é herdado dessa interface.

Atualmente apenas existe uma fila de tarefas que está associada à *pool*, onde as tarefas são retiradas da fila para serem executadas por um *thread* disponível presente na *pool*. Portanto, a vida de um *thread* presente na *pool* é: 1) Obter uma tarefa da fila. Caso a fila esteja vazia, ele adormece até a fila conter tarefas; 2) Executar a tarefa; 3) Voltar a 1).

O recurso ao padrão *pool* de *threads* (Figura 4.2) tem vantagens em relação à criação de um novo *thread* para a execução cada nova tarefa submetida. O re-uso de *threads* em vez da criação de novos *threads* traz ganhos no desempenho pois não existirá o *overhead* associado à criação e destruição de *threads* à medida que surge uma tarefa para executar. É ainda mais grave, se a tarefa for de granularidade fina, pois o *overhead* influenciará em grande medida o tempo total de execução da tarefa. Para além do problema do *overhead*, se um grande número de tarefas for submetido para execução, o grande número de *threads* criado numa única JVM pode causar graves falhas no sistema devido ao *thrashing* de recursos. O número de *threads* da *pool* é fixo e é definido aquando da sua construção podendo assim prevenir o *thrashing* de recursos. Quando não existirem *threads* disponíveis na *pool*, as tarefas são colocadas numa fila para serem executadas posteriormente.

Figura 4.2: *Pool* de *threads*.

A *pool* está atualmente implementada com recurso à classe `ThreadPoolExecutor`

do pacote `java.util.concurrent`. Na inicialização do módulo, o número de *threads* da *pool* requerido é criado e será sempre o mesmo, ou seja não existe dinamismo na criação de novos *threads* mesmo que a fila fique cheia. Os objetos submetidos para o executor têm que implementar a interface `Callable` e por essa razão a interface `TaskInterface` estende essa interface.

Apesar da escolha da *pool* de *threads* ter as vantagens já referidas em relação ao desempenho global do sistema, a escolha do tipo de fila e respetiva disciplina é também importante para manter o bom desempenho. A escolha da fila residiu numa fila limitada pois garante uma gestão mais estável dos recursos. As filas ilimitadas têm como desvantagem o facto de, se a taxa de chegadas de novas tarefas for superior à taxa de tarefas completadas existirá uma degradação de desempenho progressiva até que deixe de haver memória suficiente para guardar as tarefas. Usando uma fila limitada e com uma taxa de chegadas de tarefas superior à taxa de tarefas completadas, a fila ficará cheia em determinada altura, e portanto será necessário tratar das tarefas que não caibam na fila. Para resolver este problema foi utilizada uma política em que uma nova tarefa que não possa ser adicionada na fila será executada pelo fluxo invocador. A implementação utilizada da *pool* de *threads* permite adicionar essa política. Para tal, no seu construtor foi indicada essa política através de uma instância da classe `CallerRunsPolicy`.

Paralelismo Recursivo A primitiva `freeThreads` permite saber se existem fluxos de execução disponíveis. A sua utilização é vantajosa quando uma computação recorre ao paralelismo recursivo. O paralelismo recursivo segue o mesmo principio das invocações recursivas, mas onde estas são realizadas de forma assíncrona. O exemplo do cálculo do número de Fibonacci apresentado na Listagem 3.2 é um exemplo de paralelismo recursivo. A implementação da tarefa realiza duas invocações assíncronas da mesma tarefa, mas com argumento diferente, ficando à espera do seu resultado antes de realizar a operação de adição. Cada uma destas execuções irá ocupar um fluxo de execução e este ao ficar bloqueado à espera dos resultados faz com que não possa realizar trabalho útil. Portanto, uma maneira de contornar este problema é perguntar à localidade se existem fluxos de execução disponíveis para a execução de cada nova computação. Se não existirem, então a tarefa deve invocar a versão sequencial do método e assim é possível ter vários fluxos de execução a realizar trabalho útil. A desvantagem desta abordagem é que o programador, para além de criar a tarefa e submete-la para execução, tem também que criar a versão sequencial do método. Para tornar mais clara a explicação, considere o exemplo da Listagem 4.3 que alterou o código apresentado na Listagem 3.2.

Como pode ser observado, antes de serem submetidas as tarefas nas linhas 11 e 16, nas linhas 8 e 13 a primitiva é invocada de forma a saber se existem fluxos de execução disponíveis naquele momento. Caso não existam, a versão sequencial do método é invocada nas linhas 9 e 14, assumindo a existência do método dentro da classe `FibTask`.

```
1  public Integer call()
2  {
3    int x = 0, y = 0;
4    Future<Integer> f1 = null;
5    Future<Integer> f2 = null;
6    //...
7
8    if(!place.freeThreads())
9      x = seqFib(n-1);
10   else
11     f1 = place.spawn(new FibTask(n-1));
12
13   if(!place.freeThreads())
14     y = seqFib(n-2);
15   else
16     f2 = place.spawn(new FibTask(n-2));
17   //...
18 }
```

Listagem 4.3: Exemplo da utilização da primitiva para o cálculo do número de Fibonacci.

Comunicação

Como foi referido no Capítulo 3, a comunicação entre localidades é realizada através da invocação de operações. Como atualmente apenas é utilizada uma única JVM, a comunicação entre localidades é realizada através do mecanismo de invocações de métodos do Java, não sendo necessário a existência de qualquer *driver* em especifico para a comunicação entre localidades a executar no mesmo nó. No âmbito da extensão do *middleware* para arquiteturas de memória distribuída, este módulo terá que ser totalmente implementado. Também terá que ser implementado, pelo menos, um *driver* de troca de mensagens para ser utilizado pelo módulo para suportar a receção e emissão de mensagens.

Na invocação de uma operação o programador, se assim o desejar, poderá declarar que argumentos pretende passar por cópia/valor. A implementação dos tipos de cópia está delegado num *driver* de clonagem que recorre à biblioteca Java-Deep-Cloning Library [Kou], na qual a biblioteca oferece duas primitivas para fazer *shallowCopy* e *deepCopy* de qualquer objeto.

Sincronização

A implementação do monitor é delegada num *driver* de sincronização. A motivação para delegar a implementação do mecanismo de sincronização num *driver* advém da possibilidade de se aplicar uma abordagem otimista ou pessimista no acesso a variáveis partilhadas através de implementações otimistas ou pessimistas.

A forma tradicional de disciplinar o acesso a variáveis partilhadas é o recurso a uma abordagem pessimista com o recurso a *locks*. A única implementação do mecanismo

de sincronização recorre à estratégia de *locking* através da classe `ReentrantLock` presente no pacote `java.util.concurrent.locks`. Cada monitor tem um *lock* associado fazendo com que a implementação das operações `beginAtomic` e `endAtomic` sejam facilmente mapeadas nas primitivas *lock* e *unlock* respetivamente, garantido a atomicidade das operações encapsuladas pelo monitor. Uma das vantagens da utilização do `ReentrantLock` é a possibilidade da criação de várias variáveis de condição associadas ao *lock*, com recurso à primitiva `newCondition`, permitindo assim que seja possível definir as pré-condições de acesso ao monitor.

Em alternativa, apesar de não existir atualmente uma implementação, é possível implementar o *driver* usando uma abordagem otimista recorrendo a uma implementação de memória transacional. Segundo esta implementação, cada execução atómica seria englobada numa transação e seria executada concorrentemente com as outras execuções transacionais. O gestor de transações ficaria responsável pela verificação de conflitos de escrita/leitura, aplicando *commit* das modificações realizadas caso não haja conflitos, ou abortar e re-executar caso contrário.

Um mapeamento possível das primitivas do monitor nas primitivas usuais das bibliotecas de memória transacional seria: a primitiva `beginAtomic` seria mapeada num `beginTransaction` e a primitiva `endAtomic` seria mapeada num `commit`. Em relação às variáveis de condição, o `wait` resultaria num `rollback` ou num `commit`. No entanto, antes de ser realizada a sua implementação, terá que ser realizado um estudo para determinar como é que este *driver* poderá ser implementado à custa de memórias transacionais, pois o mapeamento descrito é apenas uma visão conceptual. Existe atualmente trabalho que estuda como é que um monitor pode ser implementado à custa de memórias transacionais [CCC+06].

Gestão de Memória Partilhada

Atualmente este módulo não está implementado pois o âmbito do middleware nesta dissertação limita-se a arquiteturas de memória partilhada. Como o módulo não está implementado, também não existe qualquer *driver* que implemente um modelo de consistência de memória, sendo a gestão de memória partilhada delegada para o modelo de memória do Java [MPA05].

No entanto, na extensão do *middleware* para ambientes distribuídos, este módulo será implementado e terá que existir, pelo menos, uma implementação do *driver* que aplique um modelo de consistência relaxado para a garantia da coerência dos dados partilhados neste ambiente.

Distribuição e Redução

A implementação corrente recorre a um *driver* que dá a noção de uma caixa negra, no sentido em que o *driver* abstraí as etapas realizadas pelo mecanismo, retornando apenas

o resultado final da computação. Este fica responsável por aplicar a distribuição, aplicar a tarefa sobre cada partição e pela aplicação da estratégia de redução aos resultados parciais obtidos.

As N partições obtidas a partir da aplicação da estratégia de distribuição, por parte de um fluxo de execução (tipicamente designado por *master*) dão origem a N execuções da mesma tarefa por parte dos vários *threads* disponíveis (tipicamente designados por *workers*).

Devido ao facto de o método `call` definido na tarefa não receber quaisquer argumentos teve que ser encontrada uma forma de os passar para o contexto da tarefa. Uma forma possível era estes serem fornecidos através do estado da tarefa, o que implicava a clonagem da tarefa. Porém, esta estratégia penaliza em grande parte o desempenho das aplicações pois as N cópias a realizar têm grande impacto tanto em termos de tempo de execução como de gasto de memória. O impacto é ainda mais visível se este mecanismo for utilizado diversas vezes durante a computação. Visto que o objetivo é aplicar o mesmo código a diferentes partições e minimizar o impacto no desempenho, então optou-se por recorrer a uma pilha de argumentos na tarefa. Isto implica que o *master* empilhe as N partições obtidas, tendo os *workers* acesso à mesma tarefa onde cada um desempilha a partição sobre a qual irá operar na execução do `call` (Figura 4.3). Naturalmente, terá que ser garantida uma disciplina de acesso visto que vários fluxos de execução irão aceder à pilha.

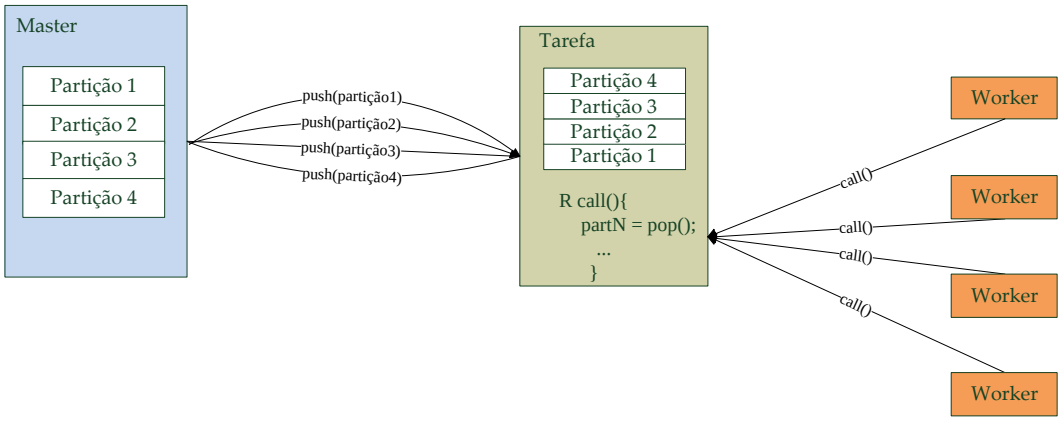


Figura 4.3: Pilha de argumentos.

4.3 Mecanismo de Anotações

As anotações Java foram introduzidas na versão 5 do Java para permitir que os seus programadores possam adicionar meta-dados ao código da sua aplicação, de uma maneira simples, de forma a serem interpretadas pelo compilador ou pela máquina virtual aquando do carregamento de classes ou durante a execução. A inserção de anotações

no código não altera a semântica da aplicação mas poderá alterar a semântica de execução se a interpretação das anotações resultar na modificação do código original da classe [TSDNP02]. As anotações Java apenas podem ser usadas para anotar classes, campos da classe, métodos e parâmetros. As anotações são utilizadas como modificadores nas declarações, tais como `public`, `final`, entre outros e sua utilização é indicada pelo símbolo @ seguido pelo seu nome.

Como foi descrito no Capítulo 3, no contexto da interface oferecida, são oferecidos novos tipos de anotação Java que permitem ao programador especificar:

- os métodos cuja execução pode ser realizada em paralelo através da anotação `Task`;
- os métodos cuja execução tenha que ser realizada atomicamente através da anotação `Atomic`;
- os argumentos de uma invocação que deverão ser passados por cópia através da anotação `Copy`.

A leitura desta secção pressupõe conhecimentos básicos da arquitetura de uma máquina virtual Java e do seu conjunto de instruções.

4.3.1 Middleware Agent

O recurso ao mecanismo de anotações, para que o programador expresse o paralelismo nas suas aplicações, traz a vantagem de não ter que ser necessário de realizar uma modificação na gramática da linguagem para suportar novas construções, e consequentemente no seu compilador, nem a modificação da implementação da máquina virtual. Isto permite que uma aplicação que usufrua do *middleware* possa ser compilada com um compilador Java sem modificações.

Quando uma aplicação é submetida para execução, um agente Java (*JavaAgent*) interceta todas as classes utilizadas para a sua execução antes de serem carregadas pela JVM. Com a ajuda de uma biblioteca externa para instrumentalização de código o agente pode modificar o *bytecode* dessas classes, caso contenha as anotações suportadas, antes de serem carregadas.

Para iniciar o agente é necessário especificar a classe que representa o agente aquando da inicialização da JVM. É necessário definir um JAR que contém a classe que especifica o agente e classes auxiliares, caso existam. O JAR deve também possuir um ficheiro *MANIFEST* onde este especifica qual a classe que representa o agente. Usando a linha de comandos para inicializar a JVM, a especificação do agente é indicada pela opção da JVM `-javaagent` em que se deve indicar o caminho relativo, ou absoluto, do JAR. Na Listagem 4.4 é ilustrado um exemplo da forma como se inicializa o agente a partir da linha de comandos.

```
java -javaagent:middlewareAgent.jar Main args...
```

Listagem 4.4: Inicialização de um JavaAgent.

Um agente é definido através de uma classe que conterá um método com a assinatura "`public static void premain(String arglist,Instrumentation inst)`", onde este principio é semelhante à definição do ponto de entrada de uma aplicação a partir do método `main`. O método `premain` implementado será invocado antes da invocação do método `main` da aplicação. O parâmetro `arglist` representa os argumentos que poderão ser passados para o agente, à semelhança do que acontece com o método `main`. De frisar o facto de não ser passado como argumento um *array* de strings. O programador terá que definir uma forma para representar os argumentos (por exemplo através um ponto e virgula entre cada argumento) e saber como interpretar a *string* recebida. O parâmetro `inst` é uma instância da classe `Instrumentation` que permite registar os transformadores que serão utilizados pelo agente para alterar o código original da classe.

A classe que define um transformador tem obrigatoriamente que implementar a interface `ClassFileTransformer` e implementar o método `transform`. Este método tem acesso à representação binária de cada classe e pode-a modificar antes da classe ser carregada pela JVM. O agente irá invocar este método especificado em cada transformador registado para transformar cada classe carregada.

Para cada classe que será carregada pela JVM, o agente realizará três etapas antes da classe ser efetivamente carregada:

1. Pesquisar anotações e retirar toda a informação relevante para a instrumentalização posterior do código da classe, caso contenha qualquer anotação;
2. Instrumentalizar o código da classe, caso ela contenha alguma anotação;
3. Carregar o código da classe transformada em vez do código da classe original.

De seguida serão descritas as etapas que permitiram a instrumentalização do código.

4.3.2 Instrumentalização

ASM Framework

Para instrumentalizar o *bytecode* das classes foi utilizada a biblioteca ASM [Con, Kul07, Bru07] que é largamente adotada para a manipulação de *bytecode*. Ela pode ser utilizada para modificar classes existentes ou gerar dinamicamente novas classes.

A biblioteca utiliza o padrão de desenho (*design pattern*) Visitor [Gam95] para processar os dados da representação binária das classes, evitando assim que os programadores manipulem diretamente as instruções, escondendo assim toda a complexidade da interpretação e escrita de *bytecode*. A biblioteca fornece duas APIs para a geração e transformação de classes:

Core API Esta fornece uma representação das classes baseada em eventos. A estrutura de uma classe é portanto apresentada como uma sequência destes eventos ordenados pela sua ordem de ocorrência no código. A API está organizada em volta de um produtor de eventos (*parser*) e num consumidor de eventos (*writer*). O primeiro gera um evento por cada elemento da classe que é intercetado e o segundo

gera código a partir de uma sequência destes eventos. Entre o produtor e o consumidor existirá um filtro que poderá modificar a sequência de eventos gerada pelo produtor que é entregue ao consumidor.

TreeAPI Esta fornece uma representação da classe como um árvore de objetos. Cada objeto representa um elemento da estrutura da classe como a própria classe, um campo, um método, uma instrução, etc.. Cada objeto contém apontadores para os objetos que fazem parte da sua constituição (ex: um objeto que representa a declaração de um método terá apontador para um objeto que representa a sua lista de instruções). Esta modelo converte a sequência de eventos de uma classe numa árvore de objetos representando a mesma classe e vice versa.

O ASM fornece as duas APIs porque não existe uma que seja muito melhor a que a outra. Ambas têm vantagens e desvantagens. Em comparação com a Tree API, a Core API requer menos memória visto não ser necessário guardar em memória qualquer estrutura acerca da estrutura da classe. Contudo, a implementação das transformações utilizando a Core API é mais complexo visto que não existe a noção de ordenação dos elementos, onde esta noção de ordenação por vezes é vantajosa na implementação de uma determinada transformação.

A interação baseada em eventos entre produtores e os consumidores de eventos é definida por várias interfaces: `ClassVisitor`, `FieldVisitor`, `MethodVisitor` e `AnnotationVisitor`. Os produtores de eventos como o `ClassReader` lançam várias invocações `visit*()` a estas interfaces, como por exemplo invocar `visitMethod` a todos os eventos que correspondam à definição de métodos da classe. Os consumidores de eventos como os *writers* (como o `ClassWriter`, `MethodWriter`, `FieldWriter`, `AnnotationWriter`) e os *adapters* (como o `ClassAdapter` e o `MethodAdapter`), implementam as interfaces `*Visitor`.

A tarefa do programador consiste no desenvolvimento de classes que alteram o fluxo dos eventos (modificar elementos, eliminar elementos, adicionar elementos, etc.) recebido a partir de um `ClassReader`, antes de delegar esse fluxo para os *writers*. De forma a tornar mais claro, na Listagem 4.5 é apresentado um pequeno exemplo do ponto de vista do programador, para a implementação de uma transformação, seguindo-se uma breve explicação.

```
ClassReader cr = new ClasReader(bytecode);
ClassWriter cw = new ClassWiter(ClassWriter.COMPUTE_MAXS);
MyClassAdapter cv = new MyClassAdapter(cw);
cr.accept(cv, ClassReader.EXPAND_FRAMES);
byte [] transformed = cw.toByteArray();
```

Listagem 4.5: Exemplo da utilização do ASM.

Neste exemplo, em primeiro lugar uma instância da `ClassReader` lê o *bytecode* de uma classe denotado pela variável `bytecode`. A classe `ClassReader` implementa a

interface `ClassVisitor` e foi desenvolvida pelos autores da biblioteca. A implementação do seu método `accept` faz as invocações a todos os `visit*` sobre o objeto do tipo `ClassVisitor` (`MyClassAdapter`) que é passado como argumento na sua invocação.

A classe `MyClassAdapter`, que foi implementada pelo programador, irá receber uma *stream* de eventos e realizará as alterações no *stream* para fazer a transformação desejada. Depois de alterar o stream de eventos, ele é passado para o `ClassWriter`. A classe implementada pelo programador pode ser vista como um filtro entre o produtor (o `ClassReader`) e o consumidor (`ClassWriter`). O `ClassWriter`, por sua vez, recebe essa transformação e o *bytecode* modificado estará disponível através da primitiva `toByteArray()`, na forma de um *array* de bytes.

Implementação

As classes relevantes para compor as aplicações desenvolvidas são as classes que estendem a classe `Place` (incluindo as que estendem `ActivePlace` que por sua vez também estende `Place`). Portanto, todas as classes que não estendam `Place` serão ignoradas pelo agente durante a etapa de pesquisa. Para todas as outras, o agente pesquisa as anotações e guarda a informação relevante para a instrumentalização posterior do código da classe (Listagem 4.6).

```
// Bytecode interpreter
ClassReader creader = new ClassReader(bytecode);
// Only classes that extends Place are needed
if(!creader.getSuperName().equals("userInterface/Place")
    && !creader.getSuperName().equals("userInterface/ActivePlace"))
  return null;
// Collect all information about annotated elements
AnnotatedMethodCollector amc = new AnnotatedMethodCollector(cname);
// ClassReader fire *visit calls to Collector
creader.accept(amc, ClassReader.SKIP_FRAMES);
```

Listagem 4.6: Colecionador de elementos da classe anotada.

Como pode ser observado na listagem anterior, depois da leitura do *bytecode* por parte do `ClassReader` é criada uma instância da classe `AnnotatedMethodCollector`. Esta classe é uma `ClassVisitor` e tem como função principal a pesquisa de anotações na classe a inspecionar e guardar toda a informação relevante. Portanto, a classe irá implementar o método "`visitMethod(int access, String name, String dscrip, String sign, String[] exceptions)`" e irá guardar toda a informação do método (acesso, nome, assinatura e exceções) caso ele esteja anotado. Para tal, foi criado uma classe `MethodCollector` que implementa `MethodVisitor` que irá inspecionar o conteúdo do método e irá inspecionar o método à procura de anotações que ele possa ter associadas.

A instrumentalização do código resultante da interpretação de cada anotação presente na classe resultará na invocação de métodos definidos na interface da localidade.

Para auxiliar o desenvolvimento foi utilizada a ferramenta *Javap*, um desassemblador incluído na distribuição Java da *Sun*/Oracle. Esta ferramenta tem como objetivo interpretar e apresentar o código binário de uma classe de forma legível. Esta ferramenta foi extremamente útil pois a partir dela foi possível obter o *bytecode* da classe contendo anotações e o *bytecode* da classe sem anotações com o objetivo de verificar as diferenças entre si, de forma a identificar o *bytecode* extra a produzir para a interpretação de cada anotação em determinado contexto. A instrumentalização resultante para cada anotação existente será descrita de seguida.

A instrumentalização do código recorre principalmente à CoreAPI devido ao seu melhor desempenho em comparação com a TreeAPI. Porém, existem algumas operações mais complexas, nomeadamente operações em que é necessário ter uma noção clara da ordenação das instruções, e por essa razão a TreeAPI é também utilizada devido à simplicidade de utilização.

@Task

De uma forma geral, a implementação desta anotação requer que seja criada dinamicamente uma classe que representa a tarefa e a modificação da classe original de forma a que a tarefa seja instanciada e seja submetida para execução. De seguida serão descritos os detalhes mais relevantes da implementação da interpretação desta anotação, onde se irá descrever as alterações realizadas no código da classe original e de que forma a tarefa é implementada.

Modificar classe original A modificação da classe original resulta em quatro etapas fundamentais: (i) Clonar o método original; (ii) Criar versão assíncrona do método; (iii) Modificar o próprio método original; (iv) Criar *getters* e *setters* para todos os campos da classe que são acedidos pelo método original.

1. Clonar método original Em primeiro lugar, o método original é sempre clonado, ficando a classe sempre com uma versão sequencial do método. A sua utilidade advém na transformação de um método que utilize paralelismo recursivo. A ideia geral é invocar este método caso não existam fluxos de execução para realizar as novas computações em paralelo, onde esta informação é dada pela invocação da primitiva `freeThreads` na localidade corrente. Mais detalhes serão descritos quando forem abordadas as modificações realizadas no código original. A sua assinatura só difere do método original no nome que lhe é atribuído para que não existam dois métodos com a mesma assinatura.

2. Criar versão assíncrona É também criado um novo método que ao ser invocado resulta na execução assíncrona do método original. Tal é necessário para que a execução do método original possa ser executada em paralelo. Relativamente à sua assinatura, o seu nome é irrelevante para o contexto, os seus parâmetros são exatamente os mesmos do método original e o seu retorno é um `Future`.

A sua implementação é composta pela instanciação da tarefa, pela sua submissão na localidade corrente, através da primitiva `spawn`, e pelo retorno do `Future` obtido na invocação da primitiva. Na instanciação da tarefa, os argumentos a serem passados para a tarefa são exatamente os mesmos que são passados no método, mais uma referência para a instância da classe mapeada na localidade (classe da aplicação).

Na Listagem 4.7 é ilustrado um exemplo da constituição deste método, baseado no exemplo do cálculo do número de Fibonnacci.

```
public Future<Integer> get_future_fib(int n){
  return spawn(
      new ConcurrentTask<Integer>(n, this) {
        public Integer call() {
          //...
        }
      }
  );
}
```

Listagem 4.7: Método que representa o método original retornando um Future.

Como pode ser observado pela listagem, o método tem como parâmetro o mesmo parâmetro do método original e a sua implementação é constituída pela instanciação da tarefa (classe aninhada anónima) onde os seus argumentos são compostos pelo argumento passado no método e por uma referência para a instância da classe. Após a instanciação da tarefa, a primitiva `spawn` é invocada para submeter a tarefa para execução na localidade corrente, sendo retornado o `Future` que é retornado pela invocação do `spawn`.

3. Modificar método original A ideia geral da utilização desta anotação é substituir todas as invocações síncronas a este método pela invocação da sua versão assíncrona. No entanto, por dificuldades de implementação, esta substituição só é feita no contexto da própria classe. Ou seja, as classes externas que invoquem o método original não terão a sua invocação modificada pela sua versão assíncrona. No entanto, no trabalho futuro, tem-se como objetivo estender essa substituição em todas as classes que serão carregadas que invoquem o método original.

O método original é então alterado para realizar a invocação à versão assíncrona e obter logo o resultado a partir do `Future` obtido, como é ilustrado na Listagem 4.8.

```
public int fib(int n) {
  return this.get_future_fib(n).get().intValue();
}
```

Listagem 4.8: Modificação do método original.

As classes externas que invoquem o método original continuarão a realizar a invocação ao método original, o que desta forma faz com que a sua invocação seja igualmente síncrona.

4. Criar *getters* e *setters* É também necessário criar métodos *get* (para obter valor) e *set* (modificar valor) para cada um dos campos que são acedidos pelos métodos originais. Portanto, foram colecionados todos os campos da classe que são lidos e foi gerado um método *get* para cada um deles. Foram também colecionados todos os campos alvos de modificação e foi gerado um método *set* para cada um deles.

Criar tarefa A criação da tarefa envolve a criação dinâmica de uma classe aninhada anónima, pertencente à localidade, que estende a classe abstrata `ConcurrentTask<R>`. A sua instanciação é efetuada através da super-classe ficando os seus argumentos acessíveis a partir da super-classe. A tarefa terá um campo em que o seu valor será a referência para instância da localidade (classe da aplicação que estende `Place`) que contém o método anotado.

A implementação efetiva da tarefa requer a implementação do método `R call()`, como visto na Secção 4.2. O tipo do seu retorno só não é o mesmo que o tipo do retorno do método original se este for um tipo primitivo. A razão para este facto resulta na obrigatoriedade de o método `call` retornar sempre um objeto e portanto, caso o método retorne um tipo primitivo, será retornado um objeto que é um *wrapper* do tipo primitivo.

O corpo do método é constituído por uma fase de inicialização e pelo código, que na sua essência, é o mesmo que o do método original mas com algumas alterações que não alteram a sua semântica. Estas alterações advêm do facto não serem impostas várias limitações/restrições quanto ao código do método desenvolvido pelo programador e portanto é necessário alterar o seu conteúdo quando o mesmo é colocado na tarefa devido à sua mudança de contexto.

1. Inicialização dos dados A fase de inicialização dos dados requer a criação de variáveis locais ao método `call` que representem os argumentos passados no método original de forma a serem utilizados na computação. Também é inicializada a referência para a instância da localidade. Naturalmente, esta última inicialização poderia ser realizada no construtor da tarefa mas a sua inicialização no `call` exigiu menos esforço de programação.

2. Alterações ao código original do método O método original pode, naturalmente, fazer acessos a campos que constituem o estado da localidade para ler ou modificar o seu conteúdo. No entanto, ao colocar o seu código na tarefa, o seu contexto irá obviamente mudar e portanto o código para realizar acessos aos campos estaria incorreto, fazendo com que o verificador de classes da JVM invalidasse a classe. Por exemplo, o *bytecode* original de uma leitura do conteúdo de um campo (entenda-se leitura por colocar o seu valor no topo da pilha) é o seguinte:

```
1 aload_0   //this
2 getfield  #14; //Field val:I
```

Listagem 4.9: *Bytecode* do acesso a um campo da classe.

Na linha 1 é colocado no topo da pilha o conteúdo da variável `this` que, neste contexto, é a referência para a própria instância da localidade; na linha 2 é colocado no topo da pilha o conteúdo do campo representado pelo identificador `val` na *constant pool* da classe do tipo inteiro (`I`). Se este código ficasse inalterado ao ser colocado na tarefa, o verificador de classes iria gerar um erro e invalidar a classe pois o campo `val` não pertence à classe da tarefa. Portanto, teve que ser encontrada uma solução para que o conteúdo do campo pudesse ser acedido pela tarefa.

A solução passou pela criação dinâmica de métodos *getters* e *setters* na localidade, como já foi referido. Assim, o código original do método será alterado para que, no contexto da tarefa, faça invocações sobre esses métodos quando for necessário fazer acessos aos campos da classe. Desta forma, o código apresentado no exemplo da Listagem 4.9 seria modificado de forma a dar origem ao código apresentado na Listagem 4.10.

```
1 aload_0   //this
2 getfield  #35; //Field mathprovider:Ltests/services/MathService/MathProvider;
3 invokevirtual #38; //Method tests/services/MathService/MathProvider.get_val:()I
```

Listagem 4.10: *Bytecode* do acesso a um campo da classe a partir da tarefa.

Na linha 1 o conteúdo da variável `this`, que é uma referência para a própria tarefa, é empilhado; na linha 2 o conteúdo do campo, que pertence à tarefa, identificado por `mathprovider` na *constant pool*, é colocado no topo da pilha. O seu conteúdo é a referência para a instância da localidade aplicação que contém o campo desejado; na linha 3 é realizada a invocação ao método *get()*, que obtém o valor do campo da classe, sobre a referência que estava presente no topo da pilha, deixando o seu resultado no topo da pilha. Apesar de o exemplo demonstrar a leitura do valor do campo, o mesmo processo seria aplicado se a operação desejada fosse a de modificação do seu valor, invocando naturalmente o respetivo método *set*.

O método original pode também fazer invocações sobre outros métodos da classe. A única restrição imposta é de que os únicos métodos que poderão ser invocados são apenas os métodos cujo acesso é público. Porém, esta limitação será ultrapassada em trabalho futuro com a alteração do acesso dos métodos privados para públicos ou com a criação de novos métodos públicos cujo corpo seja exatamente o mesmo dos métodos privados. Mais uma vez, se o código ficasse inalterado o verificador de classes iria rejeitar a classe da tarefa pois esta não contém qualquer método para além do `call`. Desta forma, a invocação terá que ser sobre o objeto que representa a instância da localidade e portanto é necessário empilhar a sua referência antes de realizar a invocação. O transformador irá, no entanto, distinguir as invocações que são realizadas a métodos anotados com a anotação `Task` dos restantes.

A motivação para diferenciar a invocação dos métodos com esta anotação surge da possibilidade de transformar a invocação desses métodos que originalmente resultam em invocações síncronas pela invocação assíncrona, possibilitando a execução paralela de trabalho durante a execução do método. Para ilustrar a motivação e a vantagem desta

transformação, considere o exemplo do cálculo do número de Fibonnacci (Listagem 3.14) que foi apresentado no Capítulo 3. Nas linhas 4 e 5, são efetuadas duas chamadas recursivas ao método `fib` em que a invocação de cada uma delas resulta numa invocação síncrona. De forma a possibilitar a realização das duas execuções em paralelo, o código é transformado de forma a substituir a invocação deste método pelo correspondente assíncrono, resultando na tradução para o código apresentado na Listagem 4.11.

```
1 Future<Integer> x_1 = null;
2 Future<Integer> y_1 = null;
3 //...
4 x_1=  get_future_fib(n-1); //x = fib(n-1);
5 y_1 = get_future_fib(n-2); //y = fib(n-2);
```

Listagem 4.11: Versão assíncrona do cálculo do número de Fibonacci.

Como pode ser observado na Listagem 4.11, nas linhas 4 e 5 são efetuadas as invocações ao método que realiza a invocação assíncrona (`get_future_fib`), ficando o resultado de cada computação acessível através do `Future` retornado.

A outra alteração prende-se com a obtenção do resultado da computação. Se na invocação síncrona, existe a garantia de que o seu resultado estará disponível ao invocador antes de este executar a próxima instrução, relativamente à invocação assíncrona não existe essa garantia. A computação é lançada independentemente do invocador e o resultado poderá não estar imediatamente disponível após a invocação e este terá que ser obtido através de um intermediário. Portanto, é necessário transformar o código de forma a substituir todas as ocorrências da necessidade do resultado da computação pela invocação do método bloqueante `get` do `Future`. A partir do mesmo exemplo da Listagem 3.14, na linha 7 o resultado de ambas as computações terá que estar disponível para que a operação de adição seja efetuada. Para tal, o resultado é obtido através da invocação da primitiva bloqueante `get` do `Future` para obter o resultado de cada uma, como pode ser observado na Listagem 4.12.

```
return x_1.get() + y_1.get(); //return x + y;
```

Listagem 4.12: Obtenção dos resultados das invocações assíncronas.

No caso dos métodos que não tenham qualquer retorno (`void`) o programador deve especificar quando é que o fluxo de execução deve bloquear, se necessário, à espera que a execução assíncrona do método acabe. Portanto, o código do método original é inspecionado de forma a encontrar todas as ocorrências da invocação da primitiva `sync`. Quando encontradas, cada invocação ao método `sync` é substituída no código da tarefa pela invocação da primitiva `get` de todos os `Future` presentes na tarefa até à altura da invocação do `sync`.

@Atomic

A implementação desta anotação não é tão complexa em comparação com a implementação da anotação `Task`. Para definir um método cuja execução tenha que ser executada atomicamente, utilizando as primitivas oferecidas pela localidade, bastará apenas adicionar a invocação `beginAtomic()` à entrada do método e a adicionar a invocação `endAtomic()` à saída (antes de um `return` ou `athrow`) do método.

A biblioteca oferece uma classe abstrata `AdviceAdapter` que pode ser utilizada para inserir código à entrada e à saída do método de uma forma simples. Para tal, é apenas necessário estender a classe e fornecer uma implementação desejada aos métodos `onMethodEnter` e `onMethodExit`. Realizada a instrumentalização, o *bytecode* de um método atómico terá a estrutura ilustrada na Listagem 4.13, seguindo-se uma explicação da estrutura.

```
1 aload_0 //load this
2 invokevirtual #76; //Method beginAtomic:()V
3 //...method body
4 aload_0 //load this
5 invokevirtual #85; //Method endAtomic:()V
6 //...return
```

Listagem 4.13: Exemplo do *bytecode* de um método atómico.

Em primeiro lugar é necessário acrescentar o código, à entrada do método, para a invocação do `beginAtomic`. Para tal, na linha 1 o conteúdo da variável `this` é carregada para o topo da pilha de execução sendo necessário para que a invocação do `beginAtomic` na linha 2 seja realizada sobre o objeto que está no topo da pilha. Na linha 3, segue-se o conteúdo do método original que estará englobado num bloco atómico. À saída do método é necessário acrescentar o código para a invocação do `endAtomic`. Na linha 4 é carregada novamente o valor da variável `this` de forma a realizar a invocação do método na linha 5. Nas restantes linhas seguem-se as operações de retorno ou de lançamento de exceções.

Naturalmente esta anotação pode ser utilizada em conjunto com a anotação Task. Para a sua concretização o código original que foi alterado para adicionar estas duas invocações será colocado no contexto da tarefa.

@Copy

Para ser possível copiar o valor dos argumentos passados numa invocação, bastará adicionar a invocação ao método `copy` definida interface da localidade, passando como argumento na sua invocação o valor do argumento do método a copiar.

Para a implementação desta anotação foi necessário primeiramente colecionar a informação de todos os parâmetros anotados pertencentes a cada método. O resultado da pesquisa originou a criação de uma tabela onde para cada método inspecionado existe

uma lista associada ao método com a informação dos parâmetros anotados. A informação necessária relativa a cada parâmetro é constituída pelo seu índice dentro do conjunto de variáveis locais do método e pelo seu tipo. Para acrescentar o código à entrada do método, à semelhança do `Atomic`, foi utilizada a classe abstrata `AdviceAdapter`. Na Listagem 4.14 é apresentado o *bytecode* que adiciona a invocação `copy` para cada argumento, resultante da implementação do método `onMethodEnter`.

Na linha 1 o conteúdo da variável `this` é empilhado, na linha 2 o conteúdo da variável local `m1` é também empilhado, seguindo-se a invocação do método `copy` onde o seu argumento é o valor que está no topo da pilha, deixando o seu retorno no topo da pilha. Na linha 4 é feito um *cast* para o tipo da variável local. Isto é necessário porque o método retorna um `Object` e portanto é necessário fazer *cast* para o tipo respetivo. Na linha 5 o valor é guardado na variável local `m1`. Da linha 6 à 10 (inclusive) o processo é exatamente o mesmo mas a variável local copiada é a variável `m2`. Nas linhas seguintes segue-se o código do método original. Pode constatar-se nas linhas 2 e 7, e 4 e 9, respetivamente que o índice e o tipo de cada parâmetro são necessários para que seja possível obter o conteúdo das variáveis através do *opcode* `ALOAD` e para guardar o valor na mesma variável.

```
1  aload_0  //load this
2  aload_1  //load m1, index: 1
3  invokevirtual #25; //Method copy:(Ljava/lang/Object;)Ljava/lang/Object;
4  checkcast #29; //class "[[I"
5  astore_1 //store top stack m1
6  aload_0  //this
7  aload_2 //load m2, index: 2
8  invokevirtual #25; //Method copy:(Ljava/lang/Object;)Ljava/lang/Object;
9  checkcast #29; //class "[[I"
10 astore_2 //store top stack m2
11 //... //method body
```

Listagem 4.14: Exemplo do *bytecode* da cópia de argumentos.

5

Avaliação

Neste capítulo é apresentada uma avaliação de desempenho relativamente à utilização da implementação atual do *middleware* como suporte à execução de várias aplicações paralelas. Esta avaliação tem como objetivo principal perceber o impacto introduzido pelo *middleware* nos tempos de execução e no gasto de memória na execução de aplicações paralelas.

5.1 Avaliação de Desempenho

Os testes realizados no âmbito desta avaliação têm como objetivo comparar o desempenho de aplicações que recorrem ao *middleware* como se de uma biblioteca tratasse, face às mesmas aplicações que recorrem diretamente às funcionalidades oferecidas pelo Java para concorrência. O estudo realizado tem como foco a obtenção dos tempos de execução e do gasto de memória das duas implementações de cada aplicação, de forma a realizar uma análise crítica em relação aos resultados obtidos. Os resultados obtidos pretendem demonstrar o desempenho global do *middleware* em termos do *overhead* expectável adicional, das dimensões estudadas, na execução das aplicações.

Para a elaboração desta avaliação foram utilizados os *benchmarks* para programação paralela *NAS Parallel Benchmarks* [NAS] e *Java Grande Benchmark Suite* [SBO01, EPC]. A escolha destas suites de *benchmarks* deve-se ao facto de serem uma referência na avaliação de desempenho de máquinas paralelas e também ao facto de fornecerem uma implementação na linguagem Java. Para além da utilização destas duas suites, foram também utilizadas outras aplicações que foram desenvolvidas durante o desenvolvimento do *middleware* para aferir a usabilidade da interface. Segue-se uma breve descrição das suites de *benchmarks* utilizadas.

NAS Parallel Benchmarks O *NAS Parallel Benchmarks* (NAS) define um conjunto de aplicações independentes da linguagem e foi desenvolvido pelo *NASA Advanced Supercomputing* (NAS) *Division*. Do conjunto das aplicações fornecidas na linguagem Java foram escolhidas três aplicações para a realização destes testes:

FFT (*Fast Fourier Transform*) - Resolve uma equação diferencial tridimensional usando o algoritmo da Transformação Rápida de Fourier.

CG (*Conjugate Gradient*) - Estima o menor valor próprio de uma matriz esparsa simétrica usando o método do gradiente conjugado.

MG (*Multi-Grid*) - Aproxima a solução de uma equação discreta de Poisson tridimensional utilizando o método *multi-grid*.

A descrição detalhada de cada aplicação é apresentada em [FSJY02].

Java Grande Benchmark Suite O *Java Grande Benchmark Suite* (JGF) também define um conjunto de aplicações que foi desenvolvido pela comunidade *Java Grande Forum*. As aplicações utilizadas foram as aplicações:

Crypt - Aplica o algoritmo de cifra simétrico IDEA (*International Data Encryption Algorithm*) para cifrar e decifrar um *array* de *bytes* que pode corresponder a um texto em claro.

Series - Calcula os primeiros N coeficientes de Fourier da função $f(x) = (x+1)^x$ no intervalo 0,2.

SOR (*Successive Over-Relaxation*) - Executa 100 iterações do método de longo relaxamento sucessivo numa matriz $N \times N$.

A descrição detalhada de cada é apresentada em [EPC].

Outras aplicações As outras aplicações desenvolvidas e utilizadas são:

PI - Realiza o cálculo do valor de pi com recurso ao método de Monte Carlo;

MatrixMult - Aplica uma multiplicação de duas matrizes $N \times N$;

Fib - Realiza o cálculo de um número da sequência de Fibonacci;

MergeSort - Aplica o algoritmo de ordenação *Merge Sort* sobre um *array* de números inteiros.

Cada aplicação tem classes de parametrização associadas, em que cada uma destas representa uma dimensão do problema em termos de dados de entrada. A notação seguida para fazer referência a uma determinada execução é dada pelo nome da aplicação, seguido de um ponto final e de uma letra que identifica a classe de parametrização utilizada na execução. Por exemplo, a referência à execução da aplicação FFT que teve como dados de entrada os definidos na classe de parametrização A é identificada por FFT.A.

O NAS tem definidas cinco classes de parametrização para cada aplicação: S, W, A, B e C. A classe S define o menor volume de dados de entrada, ao passo que a classe C define o maior. As classes utilizadas para a realização dos testes foram todas exceto a classe S pois o seu volume de dados é demasiado pequeno e portanto não é muito relevante para os testes, e a classe C para as aplicações FFT e MG devido a restrições na capacidade de memória. O JGF tem definidas três classes de parametrização para cada aplicação: A, B e C. Da mesma forma que no NAS, a classe A define o menor volume de dados e a classe C define o maior. Todas as classes de parametrização definidas foram utilizadas para a realização dos testes. No que se refere às outras aplicações desenvolvidas, as classes utilizadas bem como a sua definição são apresentadas na Tabela 5.1.

Aplicação/Classe	A	B	C	D	E
PI (rondas)	100 Milhões	300 Milhões	800 Milhões	2000 Milhões	5000 Milhões
MM ($Nlin \times Mcols$)	1024×1024	2048×2048	2500×2500	3000×3000	3500×3500
Fib (índice na seq.)	42	43	44	45	46
MergeSort (N elems)	1 Milhão	10 Milhões	100 Milhões	200 Milhões	-

Tabela 5.1: Classes de parametrização para as aplicações PI e MatrixMult.

A razão pela qual esses valores foram escolhidos para o estudo da avaliação, em detrimento dos outros que foram também testados, deve-se essencialmente ao tempo de execução que eles proporcionam. Os valores que representam dimensões do problema abaixo dos que estão definidos originam computações com baixo tempo de execução, podendo existir uma maior probabilidade de instabilidade dos tempos obtidos, devido a fatores externos como os serviços a executar do sistema de operação, fazendo com que os tempos não sejam o mais precisos. Os valores que representam dimensões do problema superior aos definidos no MatrixMult e no MergeSort não puderam ser testados devido a restrições na capacidade de memória necessária para a sua execução.

Para a realização deste estudo, cada uma das aplicações das suites de *benchmarks* utilizada foi reescrita de forma a que a sua implementação recorra à interface para as aplicações oferecida pelo *middleware*. Relativamente às outras aplicações, foram desenvolvidas três versões: 1) Versão sequencial; 2) Versão que recorre a Java "puro" que recorre a uma *pool* de *threads* para a submissão de trabalho; 3) Versão que recorre à interface do *middleware*.

Ambiente de Avaliação Os testes foram executados numa máquina com as seguintes características:

- **Processador** - Intel Core i5-760 a 2.80GHz com 8MB de cache L2 (QuadCore);
- **Memória RAM**: 4 GB de capacidade;
- **Sistema de Operação** - Windows 7 Professional 64 bits;
- **JDK** - Java SE Development Kit 6 Update 27, 64 bits.

Para a realização destes testes, de forma a minimizar possíveis inconsistências dos valores obtidos, a maioria dos processos utilizador foram encerrados, ficando sobretudo os serviços do sistema de operação a executar ao mesmo tempo que as aplicações.

Para a realização dos testes foram colocados disponíveis, para a execução do trabalho adjudicado pelas aplicações, o mesmo número de fluxos de execução (*threads*/*workers*) que de núcleos (*cores*) do processador: quatro.

De modo a obter uma maior confiança nos tempos de execução obtidos foram efetuadas várias medições para cada par aplicação-classe. Dessas medições foram retirados os três tempos mais distantes da gama de valores maioritariamente obtidos, ficando o tempo de execução de uma determinada execução igual à média das restantes medições. A obtenção de várias amostragens é importante devido à possível instabilidade dos tempos obtidos devido aos serviços a correr no sistema de operação (escalonamento de processos, gestão de memória virtual, etc.) que podem influenciar o tempo final da execução.

De seguida serão apresentados e discutidos criticamente os resultados obtidos e efetuada uma conclusão final acerca do desempenho obtido.

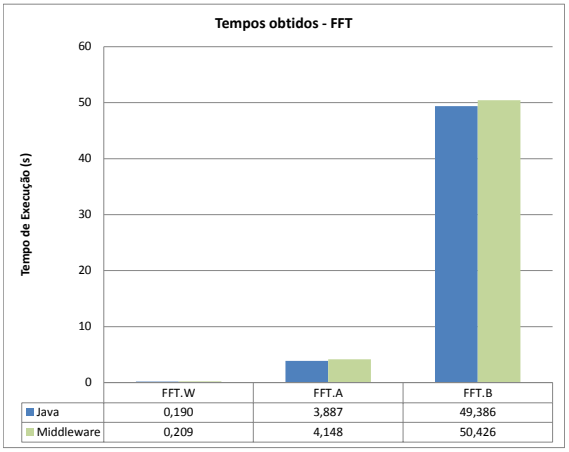
5.1.1 Tempos de execução obtidos

Os tempos de execução obtidos correspondem ao tempo total da computação, isto é, o tempo começa a contar imediatamente antes da invocação do método que origina a computação e acaba imediatamente a seguir à obtenção do seu resultado. Só desta forma é possível fazer uma comparação justa entre os tempos obtidos pela mesma aplicação que é suportada por ambientes distintos. Relativamente às aplicações que recorrem ao *middleware*, é importante referir que o *middleware* é colocado a executar imediatamente antes do inicio de execução da aplicação. Ou seja, o tempo de execução das aplicações não contempla o tempo de inicialização do *middleware*.

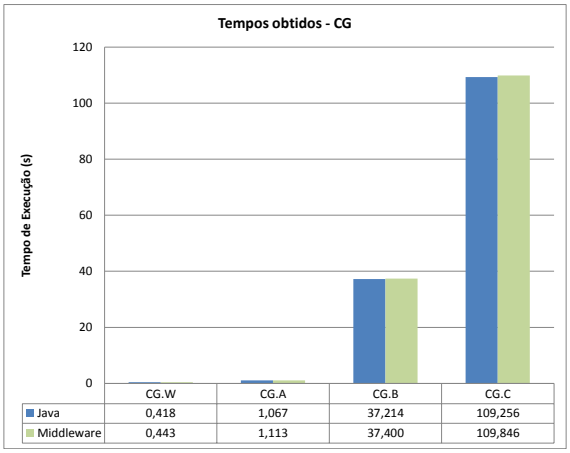
É expectável que os tempos de execução das aplicações que recorram ao *middleware* sejam superiores aos tempos de execução das aplicações que recorrem ao Java "puro". A razão pela qual se espera que isso aconteça deve-se ao *overhead*, introduzido pelo *middleware*, que está associado ao suporte às abstrações oferecidas na interface e à gestão interna do trabalho adjudicado. Portanto, para tentar confirmar esta expectativa, foram obtidos os tempos de execução de cada par aplicação-classe para as duas implementações e a partir deles foram elaborados os gráficos ilustrados nas Figuras 5.1 e 5.2 [1]. A partir dos gráficos é possível observar o tempo de execução para cada par aplicação-classe em cada uma das implementações, sendo também possível verificar a diferença entre si.

Como pode ser observado pelos gráficos, os tempos, na sua generalidade, são de facto superiores, confirmando-se a expectativa. O *overhead* associado tem mais impacto no conjunto de aplicações da suite NAS devido ao perfil das suas aplicações. As aplicações presentes nesta suite requerem que seja utilizado o mecanismo de distribuição e

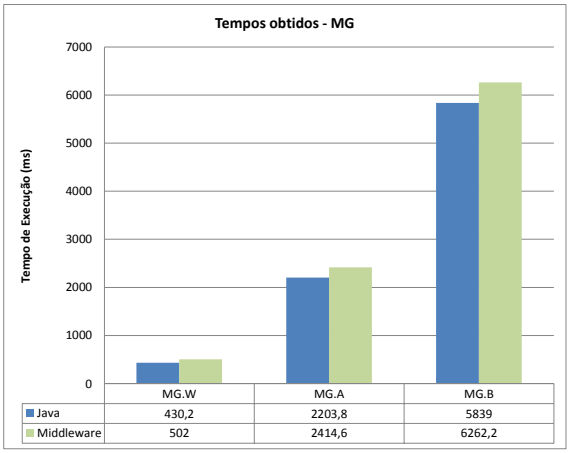
[1] Os dez gráficos não foram apresentados na mesma figura devido ao facto de estes não poderem ser convenientemente ilustrados na mesma página devido ao seu tamanho.



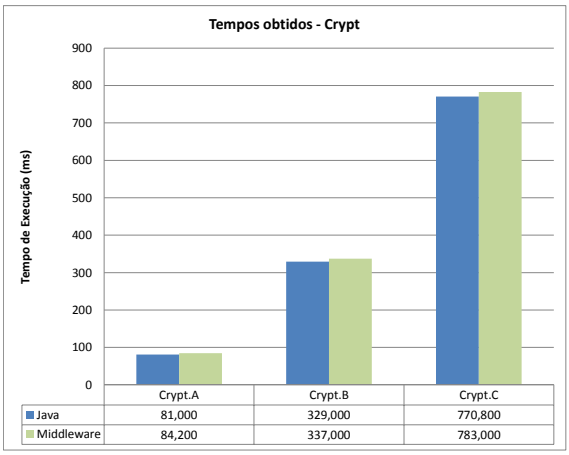
(a) FFT - Tempo de execução em segundos.



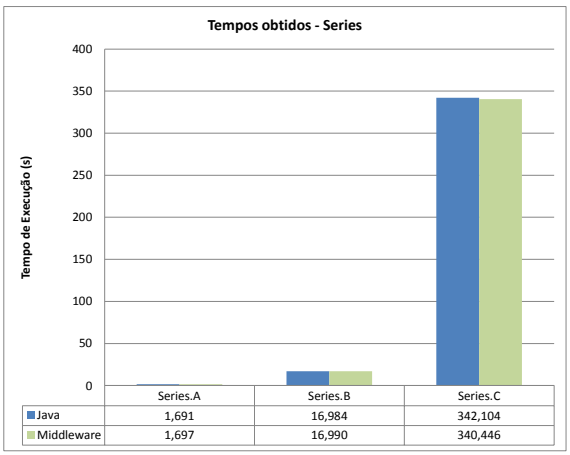
(b) CG - Tempo de execução em segundos.



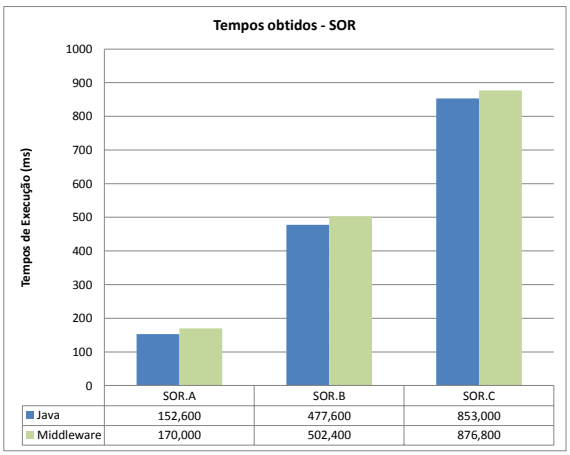
(c) MG - Tempo de execução em milisegundos.



(d) Crypt - Tempo de execução em milisegundos.



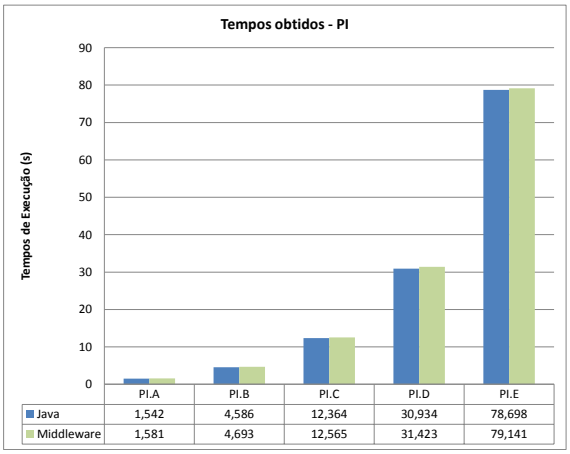
(e) Series - Tempo de execução em segundos.



(f) MG - Tempo de execução em milisegundos.

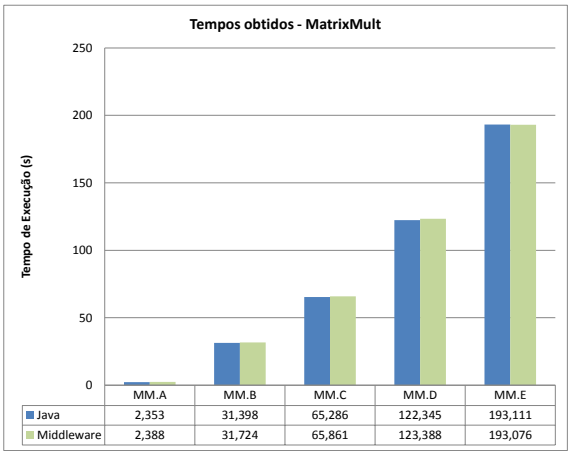
Figura 5.1: Tempos de execução obtidos na execução das aplicações das suites NAS e JGF.

redução por diversas vezes (distribuição, tarefa e redução sempre diferentes) durante a sua execução, o que faz com que sejam criados vários objetos, como as várias instâncias da distribuição e da redução e que sejam criadas várias tarefas. O facto de serem criados



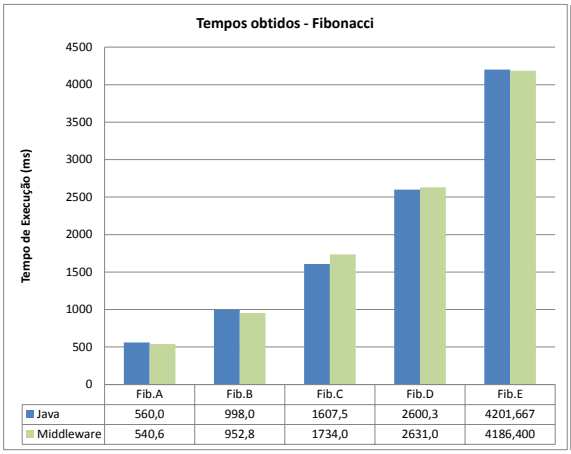
	PI.A	PI.B	PI.C	PI.D	PI.E
Java	1,542	4,586	12,364	30,934	78,698
Middleware	1,581	4,693	12,565	31,423	79,141

(a) PI - Tempo de execução em segundos.



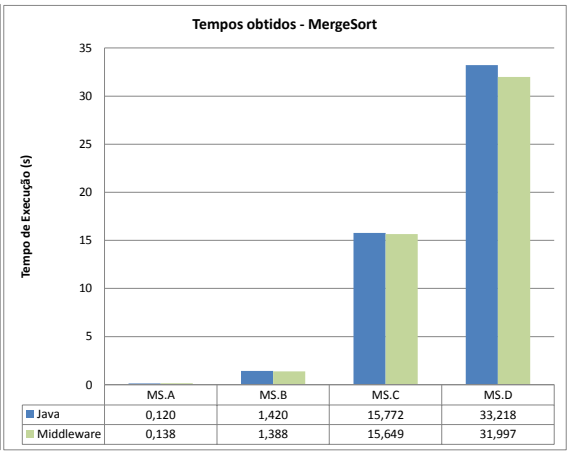
	MM.A	MM.B	MM.C	MM.D	MM.E
Java	2,353	31,398	65,286	122,345	193,111
Middleware	2,388	31,724	65,861	123,388	193,076

(b) MatrixMult - Tempo de execução em segundos.



	Fib.A	Fib.B	Fib.C	Fib.D	Fib.E
Java	560,0	998,0	1607,5	2600,3	4201,667
Middleware	540,6	952,8	1734,0	2631,0	4186,400

(c) Fib - Tempo de execução em milisegundos.



	MS.A	MS.B	MS.C	MS.D
Java	0,120	1,420	15,772	33,218
Middleware	0,138	1,388	15,649	31,997

(d) MergeSort - Tempo de execução em segundos.

Figura 5.2: Tempos de execução obtidos na execução das restantes aplicações utilizadas.

e geridos vários objetos aliado ao facto de os vários fluxos de execução terem que se sincronizar para obter, a partir da pilha de argumentos da tarefa, as partições sobre as quais irão operar, resulta num grande impacto nos tempos de execução.

É, no entanto, expectável que à medida que a dimensão do problema aumente, o impacto do *overhead* no tempo de execução seja cada vez menor. Na subsecção seguinte serão discutidos os valores dos *overheads* obtidos.

5.1.2 *Overhead*

O cálculo do *overhead* obtido na utilização do *middleware* permite aferir com mais precisão o peso que este introduz no tempo total de execução das aplicações. Tal dado é importante pois permite identificar os fatores mais relevantes que contribuem para a diminuição do desempenho em relação ao desempenho máximo expectável que uma implementação em Java consegue obter.

Como já foi referido na Subsecção 5.1.1, é expectável que o impacto do *overhead* introduzido seja desprezável à medida que a dimensão do problema aumente. Portanto, pode-se também esperar que o peso do *overhead* tenha muito mais impacto nas aplicações que tenham um tempo de execução muito pequeno, influenciando em grande medida o seu tempo final.

Antes de se apresentar os resultados obtidos, é importante introduzir a fórmula de cálculo utilizada. Para este cálculo foi utilizada a seguinte fórmula:

$$Overhead = \frac{T_{Middleware}}{T_{Java}}$$

onde $T_{Middleware}$ indica o tempo de execução da aplicação que recorre ao *middleware* e T_{Java} indica o tempo de execução da mesma aplicação que utiliza Java "puro".

A partir dos tempos obtidos, foram gerados os gráficos apresentados nas Figuras 5.3 e 5.4 onde se apresenta os *overheads* obtidos na execução das várias aplicações, ilustrando a tendência do *overhead* à medida que a dimensão do problema aumenta.

De facto, pode-se comprovar que o *overhead* tem muito maior impacto nas computações com tempos de execução muito reduzidos e que, em geral, o *overhead* associado é inversamente proporcional à dimensão do problema como era expectável. Para computações até 1 segundo a percentagem máxima foi de 16,6%, entre 1 e 2,3 segundos foi de 9,5%, entre 2,3 e 5,9 segundos foi de 7,24% e acima de 5,9 segundos foi de 1,3%. De frisar que os *overheads* de 16,6%, 9,5% e 7,24% foram todos obtidos na execução da aplicação MG. O perfil desta aplicação requer a aplicação do mecanismo de distribuição e redução por diversas vezes fazendo com que o *overhead* associado a cada distribuição e redução acumule ao longo da aplicação e que no final da mesma se faça notar em grande medida no tempo final.

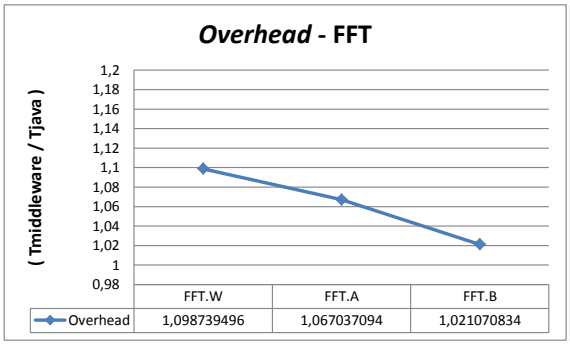
O facto de, em geral, o *overhead* decrescer à medida que se aumenta a dimensão do problema, permite expectar que o ganho de desempenho da aplicação que recorre ao *middleware* face à versão sequencial tenda para o valor do desempenho máximo possível a obter. Na próxima subsecção serão abordados os resultados obtidos relativamente ao ganho de desempenho.

5.1.3 *Speedup*

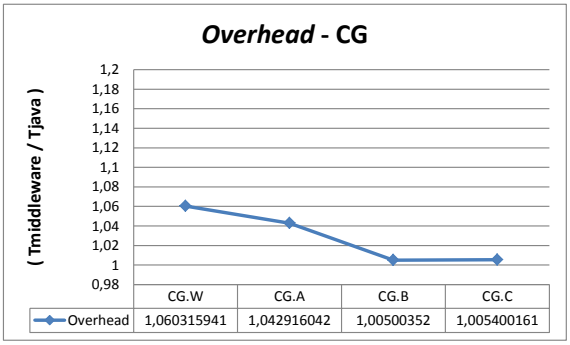
Para se determinar o ganho no desempenho de uma aplicação sequencial quando transformada em paralelo, calcula-se o seu *speedup*. O *speedup* corresponde à fração entre o tempo de execução da versão sequencial da aplicação e o tempo de execução da versão paralela da aplicação. Portanto, o *speedup* é obtido pela seguinte fórmula:

$$Speedup = \frac{T_{Sequencial}}{T_{Paralela}}$$

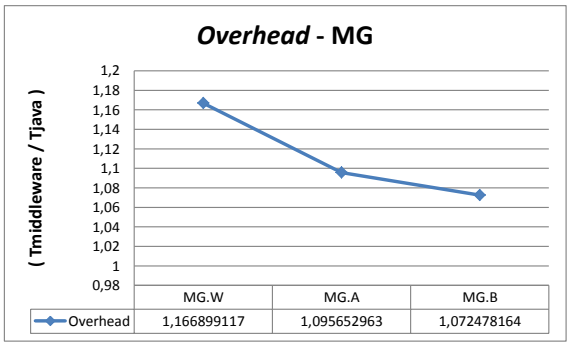
onde $T_{Sequencial}$ é o tempo de execução da versão sequencial e $T_{Paralela}$ é o tempo de execução de uma versão paralela.



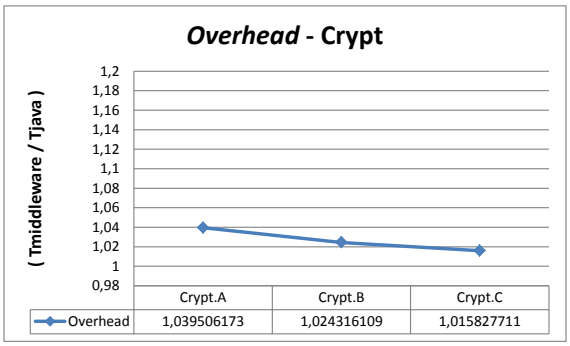
(a) *Overhead* - FFT



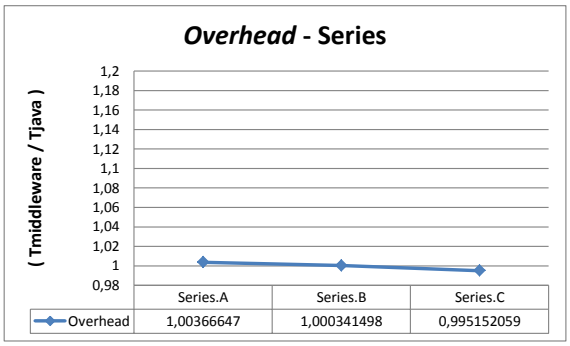
(b) *Overhead* - CG



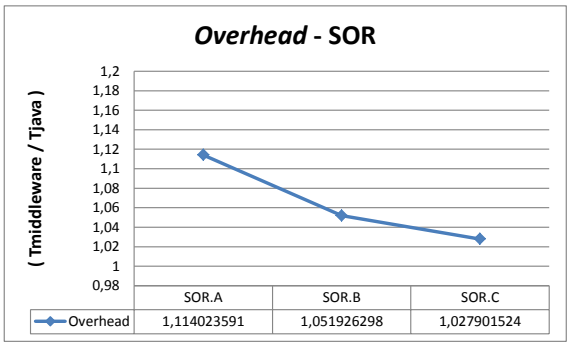
(c) *Overhead* - MG



(d) *Overhead* - Crypt



(e) *Overhead* - Series



(f) *Overhead* - SOR

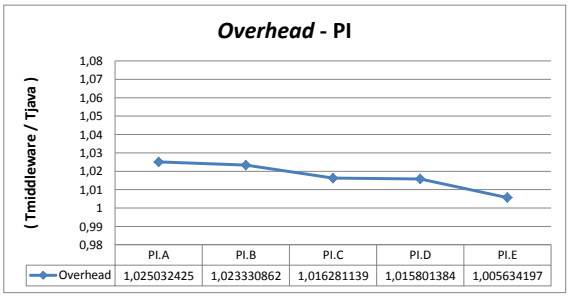
Figura 5.3: *Overhead* obtido na execução das aplicações das suites NAS e JGF.

Teoricamente, o *speedup* ótimo é igual ao número de unidades de execução utilizadas na computação paralela. No entanto, a Lei de Amdahl [Amd67] afirma que o *speedup* ótimo de uma versão paralela de um algoritmo estará sempre limitado pela região sequencial do algoritmo. Desta forma, o speedup ótimo é dado pela fórmula:

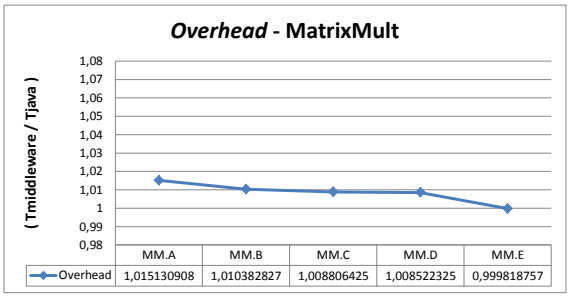
$$Speedup = \frac{1}{f}$$

onde f é a fração sequencial do algoritmo.

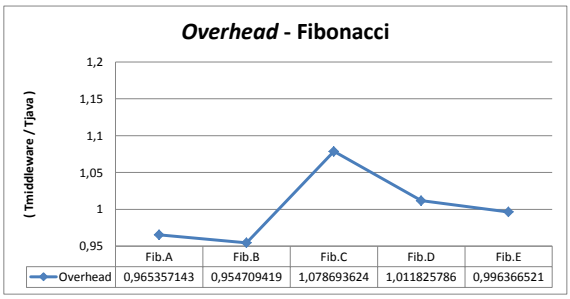
No entanto, esta fórmula não se aplica para todas as aplicações paralelas. No caso das



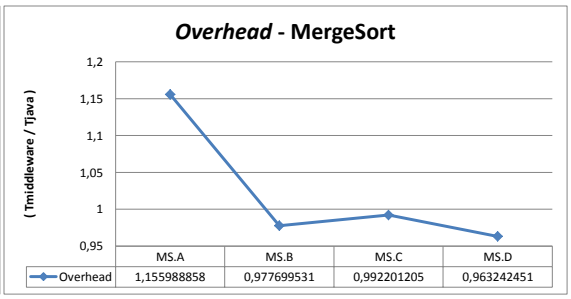
(a) *Overhead* - PI



(b) *Overhead* - MatrixMult



(c) *Overhead* - Fibonacci



(d) *Overhead* - MergeSort

Figura 5.4: *Overhead* obtido na execução das restantes aplicações.

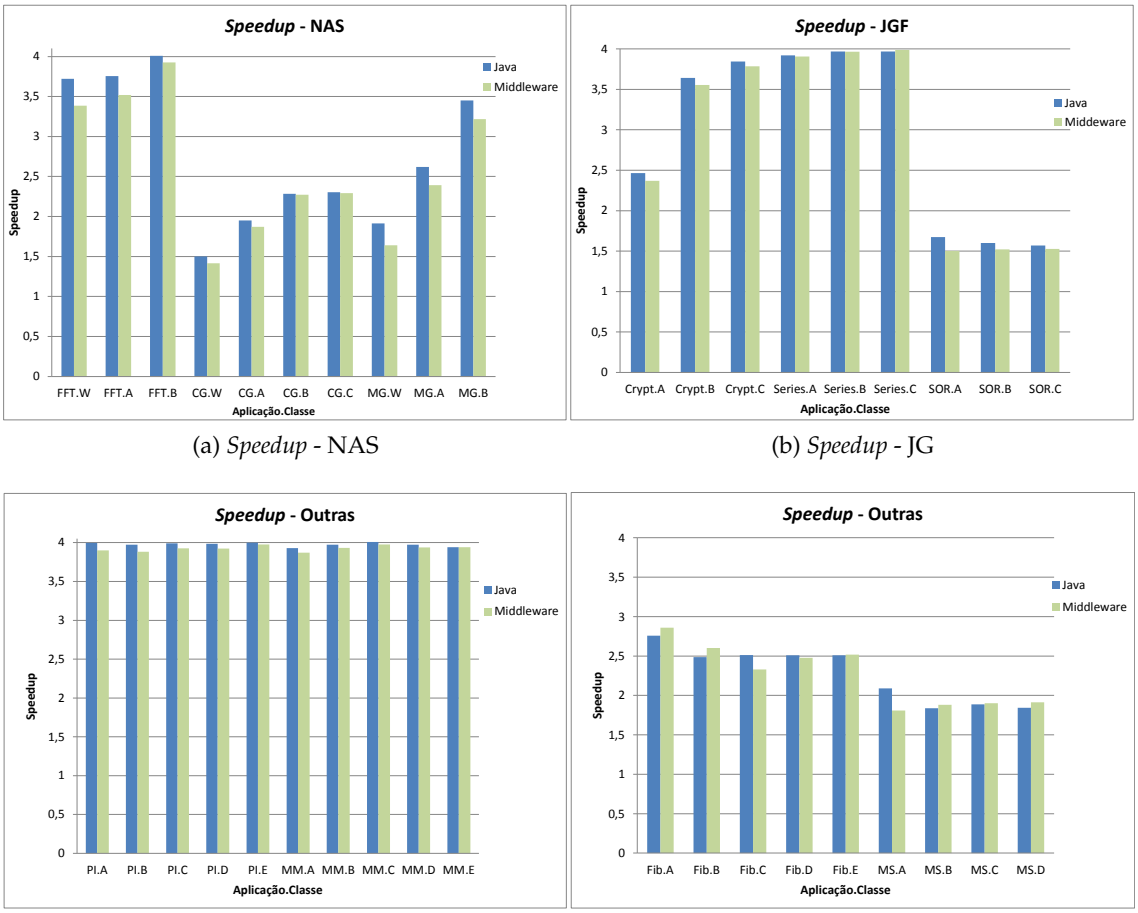
aplicações que se enquadrem nas computações embaraçosamente paralelas, isto é, a fração de execução paralela é muito superior à fração sequencial, à medida que a dimensão do problema aumenta, menos peso terá a fração sequencial do problema no desempenho. Consequentemente, neste tipo de aplicações, à medida que a dimensão do problema aumenta, o speedup tende para comportamento linear.

O facto de, em geral, o *overhead* decrescer à medida que se aumenta a dimensão do problema, permite expectar o *speedup* obtido tenda para o valor do desempenho máximo que a aplicação consegue obter. Para tentar comprovar este facto foram realizadas medições dos tempos de execução das respetivas aplicações sequenciais de forma a calcular o *speedup* obtido na execução das aplicações paralelas que utilizam Java "puro" e na execução das aplicações que utilizam o *middleware*. Relativamente às aplicações da suite NAS foram utilizadas as versões sequenciais fornecidas. Relativamente às aplicações da suite JGF optou-se por executar cada aplicação paralela apenas com um *thread*.

Na Figura 5.5 são apresentados os gráficos do *speedup* obtido para cada par aplicação-classe. Como se pode verificar, em geral, à medida que a dimensão do problema da aplicação que faz uso do *middleware* aumenta, o *speedup* máximo obtido por ela também aumenta, aproximando-se cada vez mais do valor do *speedup* obtido pela versão Java "puro".

Na maioria das aplicações o *speedup* obtido ficou longe do valor do *speedup* linear. Tal deve-se à fração não paralelizável do código e aos *overheads* do sistema de execução do Java (JVM), nomeadamente o *garbage collector*, e às mudanças de contexto, sincronização e acesso à memória por parte dos fluxos de execução. No contexto do *middleware*, a tudo

isso é necessário acrescentar também o *overhead* associado à sua utilização.



(a) *Speedup* - NAS (b) *Speedup* - JG

(c) *Speedup* - Outras (d) *Speedup* - Outras

Figura 5.5: *Speedups* obtidos.

5.1.4 Gasto de Memória

A avaliação do gasto de memória pretende contribuir com mais um importante fator para o estudo da avaliação de desempenho. A memória adicional que é alocada durante a execução de uma aplicação pode influenciar em grande medida o seu desempenho. Se uma computação necessitar de alocar uma grande quantidade de dados e estes não couberem todos em memória RAM, o mecanismo de memória virtual do sistema de operação entrará em ação com mais frequência. Desta forma, o mecanismo de memória virtual realiza com mais frequência a operação de *swapping*, que consiste em guardar em disco partes do conteúdo da memória RAM (relativas a um processo) de forma a ter mais RAM disponível. Esta é uma operação pesada fazendo com que ela tenha um grande impacto no desempenho das aplicações.

Relativamente a este teste, o gasto de memória obtido para cada computação refere-se ao tamanho máximo da *heap* Java utilizado nessa computação. Para determinar esse valor foi utilizado o *profiler* do IDE Netbeans [Com]. O *profiler* permite analisar dinamicamente

uma computação de forma a medir a utilização total do processador e o uso de memória, com o objetivo de permitir ao programador encontrar pontos críticos que influenciam negativamente o desempenho. Tal é importante, pois permite ao programador otimizar as suas aplicações. Ao contrário de outros *profilers* que ficam a executar em *background* analisando quaisquer computações submetidas durante o seu tempo de execução, o *profiler* Netbeans permite analisar especificamente uma execução. Na prática, o *profiler* Netbeans é um agente Java que tem como objetivo modificar o *bytecode* de cada classe que compõe a aplicação de forma a adicionar as invocações às APIs Java Virtual Machine Profiling Interface (JVMPI) e Java Virtual Machine Tool Interface (JVMTI) de forma a obter toda a informação de análise para fornecer ao programador.

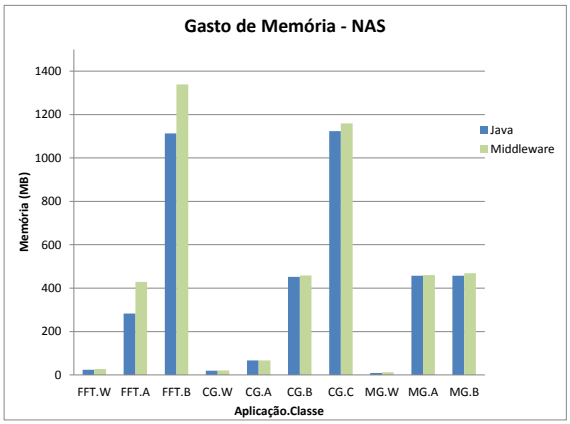
A execução de cada par aplicação-classe foi analisada e foram obtidos os gastos de memória apresentados nos gráficos da Figura 5.6. É expectável que o gasto de memória das aplicações que recorrem ao *middleware* seja superior ao das aplicações que recorrem a Java "puro". É esperado devido à criação de vários objetos por parte do *middleware* e pela criação, por parte do programador, de objetos que correspondem às tarefas e, caso seja necessário, das estratégias de distribuição e redução. De facto, essa expectativa é confirmada pois em todas as aplicações que recorreram ao *middleware* têm sempre um maior gasto de memória, mesmo sendo ligeiro.

Em quase todas as aplicações, à medida que a dimensão do problema aumenta o gasto de memória também aumenta. As exceções são as aplicações PI e Fib, como pode ser observado no gráfico da Figura 5.6c. Na aplicação PI a dimensão do problema refere-se a um número de iterações de um cálculo a realizar por cada fluxo de execução e na aplicação Fib a dimensão refere-se apenas ao índice do número a obter da sequência de Fibonacci. Portanto, o aumento de dimensão não resulta na alocação de mais objetos na *heap*.

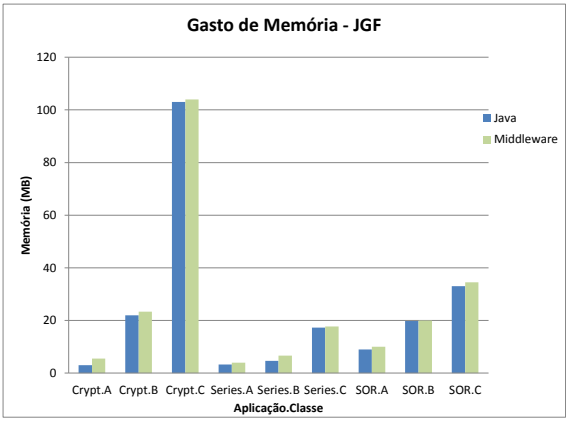
5.1.5 Apreciação final

Como era expectável antes da realização deste estudo, o desempenho das aplicações que recorrem ao *middleware* para o seu suporte sofre de um *overhead* associado à sua utilização. Porém, os resultados obtidos foram animadores pois os valores obtidos do *overhead* não têm um grande impacto no tempo de execução das aplicações. Este facto é mais evidente, nas computações que operam sobre grandes dimensões de problemas onde o *overhead* associado é praticamente desprezável. Relativamente ao gasto de memória, ele existirá sempre, apesar dos resultados obtidos não serem desanimadores. Em trabalho futuro, poderá procurar-se encontrar maneiras de forma minimizar o gasto de memória.

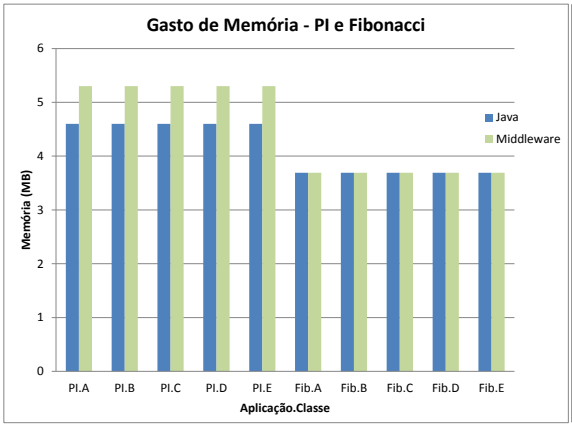
Apesar de existir sempre este *overhead* associado, por mais pequeno que seja, acreditamos que o seu impacto é mitigado pela maior produtividade que é possível obter no desenvolvimento de aplicações recorrendo à interface para as aplicações oferecida pelo *middleware*.



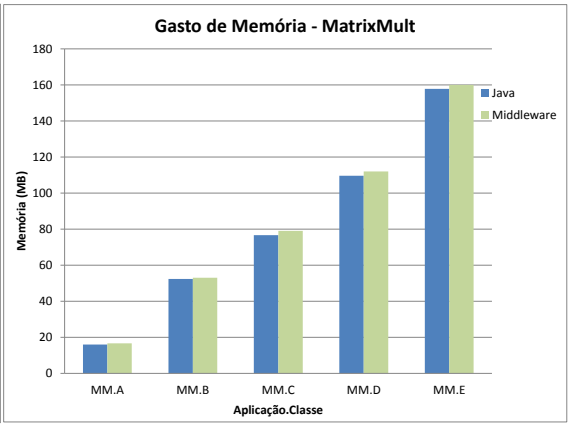
(a) Gasto de Memória - NAS.



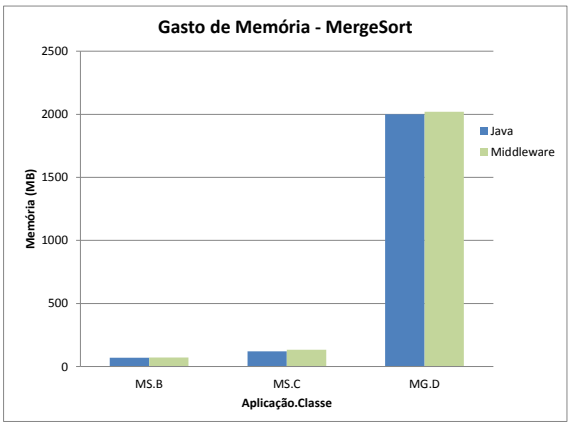
(b) Gasto de Memória - JGF.



(c) Gasto de Memória - PI e Fibonacci.



(d) Gasto de Memória - MatrixMult.



(e) Gasto de Memória - MergeSort.

Figura 5.6: Comparação dos gastos de memória usando Java puro versus *middleware*.

6

Conclusões e Trabalho Futuro

A tendência atual do aumento de desempenho dos processadores dá primazia ao aumento de núcleos de processamento em vez do aumento da velocidade de relógio. Consequentemente, apenas as aplicações que fazem uso da concorrência podem beneficiar do aumento de desempenho deste tipo de processadores. No entanto, os modelos de programação atuais não são os mais apropriados para o desenvolvimento de aplicações concorrentes e paralelas pois não oferecem construtores de alto nível que permitam expressar o paralelismo de uma forma simples. Eles obrigam a que o programador lide com aspetos que não estão apenas relacionados com a lógica das aplicações. É então necessário criar novas soluções que permitam expressar o paralelismo de uma forma simples de forma a ser possível separar os aspetos relacionados com a lógica das aplicações dos aspetos relacionados com a gestão da concorrência, de forma a aumentar a produtividade.

Neste contexto, o trabalho desenvolvido nesta dissertação consistiu no desenvolvimento de uma infraestrutura *middleware* que tem como objetivo principal a execução de aplicações paralelas independentes da plataforma, oferecendo para isso abstrações que simplificam o processo de desenvolvimento de aplicações. Para tal, o *middleware* suporta um conjunto de funcionalidades indispensáveis para tornar possível a computação paralela. O *middleware* abstrai o hardware da plataforma de execução e a gestão do paralelismo através de uma interface bem definida para as aplicações que é independente da arquitetura alvo. Pretende também ser uma plataforma suficientemente genérica para que possa ser utilizado como suporte ao sistema de execução de linguagens de programação ou como biblioteca para o desenvolvimento de várias aplicações.

A interface para as localidades é centrada no conceito de localidade. Uma localidade apresenta uma interface bem definida e pode ser vista como uma unidade de execução independente sobre a qual se adjudica trabalho. Uma computação é então composta

por um conjunto de localidades onde o programador interage com estas a partir da sua interface. Uma das vantagens na sua utilização advém do facto de se dar o controlo ao programador relativamente à localização da computação, podendo ele gerir a afinidade entre o trabalho e os recursos disponíveis. A outra vantagem prende-se com o facto de ser possível oferecer a mesma interface para a programação em arquiteturas de memória partilhada e memória distribuída.

Com o objetivo de minimizar o esforço de desenvolvimento por parte do programador, foi também desenvolvido um mecanismo de anotações que lhe permitem anotar o código da aplicação para expressar o paralelismo. O programador com recurso às anotações atualmente disponíveis pode definir que métodos podem ser executados em paralelo, os métodos que devem ser executados atomicamente e quais os argumentos de uma invocação devem ser passados por cópia.

Para além da interface para as aplicações, o *middleware* oferece também uma interface para a integração de várias especializações das funcionalidades suportadas. Isto permite delegar em tecnologias já existentes o suporte de uma dada funcionalidade. A integração de várias implementações permite ao *middleware* adequar o conjunto de especializações a utilizar consoante a arquitetura alvo.

Relativamente aos modelos de programação suportados, a interface oferecida permite oferecer um modelo de programação em memória partilhada e um modelo de programação em memória distribuída (troca de mensagens). O modelo de programação de memória partilhada é obtido através da adjudicação de trabalho numa localidade, em que a localidade tem o seu próprio estado e um conjunto de fluxos de execução que executam concorrentemente onde estes podem comunicar através do seu estado. O modelo de programação em memória distribuída é obtido através da invocação das operações entre localidades distribuídas. A interface oferece também dois modelos mais refinados de programação paralela: paralelismo de tarefas e paralelismo de dados.

Uma implementação base da camada de *drivers* foi realizada de forma a que o *middleware* seja suportado em arquiteturas *multi-core*, podendo ser estendida para que o *middleware* seja suportado em outros tipos de arquitetura, como *clusters* de *multi-cores*. A implementação atual foi avaliada essencialmente em termos de desempenho. Esta avaliação consistiu na comparação dos tempos de execução e de gasto de memória de aplicações que recorrem a Java "puro", face às mesmas aplicações que recorrem à interface para as aplicações oferecida pelo *middleware*. As aplicações utilizadas pertencem às suites de *benchmarks* de referência para programação paralela *NAS Parallel Benchmarks* e *Java Grande Benchmark Suite*. Cada uma das aplicações utilizadas foi reescrita de forma a recorrer ao *middleware* para o seu suporte, o que de certa forma contribuiu também para avaliar o seu funcionamento e aferir a usabilidade da interface oferecida. A comparação das dimensões estudadas teve como objetivo confirmar o *overhead*, que era expectável antes da realização do estudo, na execução das aplicações que são suportadas pelo *middleware*. No entanto, apesar de existir sempre um *overhead* associado, este é compensado pelo facto de se oferecerem abstrações que simplificam o desenvolvimento de aplicações, obtidos

através da interface da localidade e pelo conjunto de anotações oferecido atualmente.

Em termos de trabalho futuro existem alguns pontos que podem ser desenvolvidos. Esses pontos incluem-se na implementação do núcleo do *middleware*, na interface para as aplicações e na avaliação. Relativamente ao núcleo do *middleware*, os pontos a desenvolver são:

- Estender o *middleware* para arquiteturas de memória distribuída de forma a que ele possa ser suportado em *clusters* de *multi-cores*:
  - Implementar a lógica dos módulos que não estão atualmente implementados, nomeadamente os módulos de comunicação e gestão de memória partilhada;
  - Popular a camada de *drivers* com as especializações que tornam possível o suporte do *middleware* para este tipo de arquiteturas.
- Implementação do *driver* de sincronização (monitor) com recurso a uma biblioteca de memória transacional;
- Implementação de um *driver* de distribuição e redução que recorra a GPUs.

Relativamente à interface para aplicações, os pontos a desenvolver são:

- Oferecer uma primitiva na localidade que indique a sua localização física. Isto permitiria ao programador submeter trabalho nas localidades que lhe incorressem uma menor penalização de desempenho;
- Oferecer uma primitiva na interface da localidade que permite devolver uma localidade que esteja mapeado num GPU presente no nó sobre o qual a localidade está suportada. A ideia é que a localidade que esteja mapeada num GPU possa ser vista como uma localidade filha, tendo a mesma interface da localidade pai;
- Colocar a primitiva *spawn* associada à tarefa e não à localidade. Isto permitiria que a tarefa pudesse ser executada em qualquer localidade, sendo que um escalonador tomaria a decisão de em que localidade a tarefa devesse ser executada, como acontece no Cilk e no pSystem/di_pSystem. Atualmente o programador especifica explicitamente em que localidade cada tarefa deve executar;
- Implementar mais um conjunto de anotações que possibilitem a utilização simplificada do mecanismo de distribuição e redução e para definir as variáveis de condição de acesso ao monitor.

Relativamente à avaliação seria interessante realizar uma análise de produtividade. O objetivo era realizar uma comparação entre o desenvolvimento de aplicações recorrendo à interface para aplicações oferecida pelo *middleware* e o desenvolvimento das mesmas aplicações utilizando outras abordagens para programação paralela em Java. Por exemplo, poderia-se realizar a reescrita das aplicações que foram utilizadas na avaliação de

desempenho recorrendo à interface dos *middlewares* Parallel Java e Pro-Active e verificar o esforço realizado no desenvolvimento da mesma aplicação utilizando interfaces distintas.

Bibliografia

[AJMJS03] J. Al-Jaroodi, N. Mohamed, H. Jiang, e D. Swanson. Middleware infrastructure for parallel and distributed programming models in heterogeneous systems. *Parallel and Distributed Systems, IEEE Transactions on*, 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.

[BBC+06] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, e Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[BCD+06] Rajkishore Barik, Vincent Cave, Christopher Donawa, Allan Kielstra, Igor Peshansky, e Vivek Sarkar. Experiences with an SMP implementation for X10 based on the Java Concurrency Utilities. In *Workshop on Programming Models for Ubiquitous Parallelism (PMUP 2006)*, September 2006.

[BH77] Henry C. Baker, Jr. e Carl Hewitt. The incremental garbage collection of processes. *SIGART Bull.*, pág. 55–59, August 1977.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, e Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING*, pág. 207–216, 1995.

[BNSP99] Lars Buttner, Jorg Nolte, e Wolfgang Schroder-Preikschat. Arts of peace: A high-performance middleware layer for parallel distributed computing. *J. Parallel Distrib. Comput.*, 59:155–179, November 1999.

[Bon02] Dan Bonachea. Gasnet specification, v1.1. Relatório técnico, Berkeley, CA, USA, 2002.

[Bru07] E. Bruneton. Asm 3.0 a java bytecode engineering library. *URL: http://download. forge. objectweb. org/asm/asmguide. pdf*, 2007.

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

[CCC+06] Brian D. Carlstrom, JaeWoong Chung, Hassan Chafi, Austen McDonald, Chi Cao Minh, Lance Hammond, Christos Kozyrakis, e Kunle Olukotun. Executing java programs with transactional memory. *Sci. Comput. Program.*, 63:111–129, December 2006.

[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, July 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.

[CCY+99] W. Carlson, D. Culler, K. Yellick, E. Brooks, e K. Warren. Introduction to UPC and language specification. 1999.

[CCZ04] David Callahan, Bradford L. Chamberlain, e Hans P. Zima. The cascade high productivity language. In *in Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04*, pág. 52–60, 2004.

[CCZ07] B.L. Chamberlain, D. Callahan, e H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, August 2007.

[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, October 2005.

[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.

[Com] NetBeans Community. Netbeans profiler. `http://netbeans.org/features/java/profiler.html`.

[Con] OW2 Consortium. Asm framework. http://asm.ow2.org/.

[Cora] Intel Corporation. Intel array building blocks. http://software.intel.com/en-us/articles/intel-array-building-blocks/.

[Corb] Intel Corporation. Intel cilk plus. http://software.intel.com/en-us/articles/intel-cilk-plus/.

[Corc] Intel Corporation. Intel parallel building blocks. http://software.intel.com/en-us/articles/intel-parallel-building-blocks/.

[CRDI05] T. Chen, R. Raghavan, J. N. Dale, e E. Iwata. Cell broadband engine architecture and its first implementation. https://www.ibm.com/developerworks/power/library/pa-cellperf/, 11 2005.

[Dan05] John S. Danaher. The jcilk-1 runtime system. Tese de Mestrado, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Junho 2005.

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

[DG04] Jeffrey Dean e Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pág. 10–10, 2004.

[DLL05] John S. Danaher, I-Ting Angelina Lee, e Charles E. Leiserson. The JCilk language for multithreaded computing. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, California, Outubro 2005.

[EPC] EPCC. Java grande benchmark suite. http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html.

[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, 2006.

[FLR98] Matteo Frigo, Charles E. Leiserson, e Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pág. 212–223, Montreal, Quebec, Canada, Junho 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

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

[FSJY02] Michael A. Frumkin, Matthew Schultz, Haoqiang Jin, e Jerry Yan. Implementation of the nas parallel benchmarks in java. Relatório técnico, 2002.

[FTL+02] J. Frey, T. Tannenbaum, M. Livny, I. Foster, e S. Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[FZRL08] I. Foster, Yong Zhao, I. Raicu, e S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pág. 1 –10, 2008.

[Gam95] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongorra, Weicheng Jiang, Robert Manchek, e Vaidy Sunderman. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, 1994.

[Gen01] W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pág. 35–36. IEEE, 2001.

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

[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.

[Hoa74] Charles Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[HPR+08] Mike Houston, Ji-Young Park, Manman Ren, Timothy Knight, Kayvon Fatahalian, Alex Aiken, William Dally, e Pat Hanrahan. A portable runtime interface for multi-level memory hierarchies. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pág. 143–152, 2008.

[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.

[Kou] Kostas Kougios. Java-deep-cloning library. `http://code.google.com/p/cloning/`.

[Kul07] E. Kuleshov. Using asm framework to implement common bytecode transformation patterns. In *Aspect-Oriented Software Development (AOSD) Conf*, 2007.

[LS96] R. Greg Lavender e Douglas C. Schmidt. Active object – an object behavioral pattern for concurrent programming, 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.*, 27:329–351, March 1997.

[MC97] John Merlin e Barbara Chapman. High performance fortran 2.0. In *in Proc. Sommerschule uber Moderne Programmiersprachen und Programmiermodelle, Technical University of Hamburg-Harburg, Sept 15–19*, 1997.

[MHC01] R. Monson-Haefel e D.A. Chappell. *Java message service*. O'Reilly Media, 2001.

[Mic] Sun Microsystems. Java reflection api. `http://download.oracle.com/javase/tutorial/reflect/index.html`.

[Moo65] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[MPA05] J. Manson, W. Pugh, e S.V. Adve. *The Java memory model*, volume 40. ACM, 2005.

[MW08] Tim Mattson e Michael Wrinn. Parallel programming: can we please get it right this time? In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pág. 7–11, New York, NY, USA, 2008. ACM.

[NAS] NASA. Nas parallel benchmarks. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[NC99] Jarek Nieplocha e Bryan Carpenter. Armci: A portable remote memory copy libray for ditributed array libraries and compiler run-time systems. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pág. 533–546. Springer-Verlag, 1999.

[NR98] Robert W. Numrich e John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17:1–31, 1998.

[Pau98] Hervé Paulino. Desenho e implementacão do psystem para arquitecturas de memória partilhada. Tese de Mestrado, Faculdade de Ciências da Universidade do Porto, 1998.

[Rei07] J. Reinders. *Intel threading building blocks*. O'Reilly, 2007.

[SAB+] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, e O. Tardieu. The Asynchronous Partitioned Global Address Space Model.

[SBO01] L.A. Smith, J.M. Bull, e J. Obdrzalek. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pág. 8–8. ACM, 2001.

[SOHL+98] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, e Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[SPL99] Fernando M. A. Silva, Hervé Paulino, e Luis M. B. Lopes. Di_psystem: A parallel programming system for distributed memory architectures. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pág. 525–532, 1999.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[TSDNP02] Éric Tanter, Marc Ségura-Devillechaise, Jacques Noyé, e José M. Piquer. Altering java semantics via bytecode manipulation. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, GPCE '02, pág. 283–298, London, UK, 2002. Springer-Verlag.

[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.

[Wal98] J. Waldo. Remote procedure calls and java remote method invocation. *Concurrency, IEEE*, 6(3):5–7, 1998.

[Xin06] Bin Xin. Cx10: Distributed runtime for x10. `http://www.cs.purdue.edu/homes/xinb/cx10/CX10Report/CX10Report.html`, 08 2006.

[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.

[Zha07] Z. Zhang. A performance model for unified parallel c. Relatório técnico, Michigan Technological University, 2007.

[ZSS06] Zhang Zhang, Jeevan Savant, e Steven Seidel. A upc runtime system based on mpi and posix threads. In *Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2006.