Daniel Filipe Ventura Ferreira

Licenciado em Engenharia Informática

Execução paralela de métodos em Scala

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Co-orientadores: Artur Miguel Dias, Prof. Auxiliar,
Faculdade de Ciências e Tecnologia da
Universidade Nova de Lisboa
Hervé Paulino, Prof. Auxiliar,
Faculdade de Ciências e Tecnologia da
Universidade Nova de Lisboa

Júri

Presidente: Vítor Manuel Alves Duarte
Arguente: Vasco Fernando de Figueiredo Tavares Pedro
Vogal: Artur Miguel de Andrade Vieira Dias

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Junho, 2019

Execução paralela de métodos em Scala

Copyright © Daniel Filipe Ventura Ferreira, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.
A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Agradecimentos

A realização desta dissertação de mestrado contou com importantes apoios pelos quais estarei eternamente grato.

Ao professor Artur Miguel Dias pelo total apoio e disponibilidade, opiniões construtivas, palavras de incentivo e total colaboração no solucionar de dúvidas problemas que foram surgindo ao longo do processo.

Ao professor Hervé Paulino pela orientação, saber que transmitiu, sempre fundamentadas criticas e pela constante procura de excelência.

Aos meus colegas de faculdade pela companhia em sessões de trabalho e estudo, sempre cheias de animação e partilha de conhecimento.

Aos meus amigos pelas suas constantes palavras de incentivo e pela compreensão nas alturas em que não pude participar em diversas atividades. Ainda assim, obrigado por estarem sempre prontos para partilhar momentos de diversão e intimidade.

Aos meus colegas de trabalho por despertarem em mim um grande interesse por programação funcional que me fez apreciar ainda mais o trabalho e conhecimento adquirido no processo de realização desta dissertação.

Um agradecimento especial aos meus pais pelo seu apoio incondicional, incentivo e paciência demonstrados. Obrigado pela total ajuda na superação dos obstáculos que foram surgindo ao longo desta caminhada, tanto profissionais como pessoais.

Por último, mas não menos importante, obrigado à minha namorada Joana Veríssimo pelo amor, carinho e companheirismo que me acompanharam durante todo o processo.

Resumo

A programação não tem acompanhado de uma forma simplificada as evoluções de *hardware* verificadas ao longo dos anos. Os recursos disponíveis nos atuais processadores multinúcleo e *clusters* de alta performance ainda não estão a ser aproveitados na sua totalidade por programadores não especialistas na área de computação paralela.

O modelo de programação de alto nível *Single Operation Multiple Data (SOMD)* apresenta uma nova ideia que tem o objetivo de generalizar a execução de aplicações de forma paralela. O modelo pretende, com o acrescento de algumas anotações, ser capaz de executar em sistemas multinúcleo e distribuídos sub-rotinas previamente escritas e pensadas de forma sequencial.

O objetivo principal do projeto desta dissertação é a criação de uma implementação em *Scala*, usando a *framework* de programação *Akka*, do modelo *SOMD*. A solução será capaz de executar de forma paralela código sequencial que seja suscetível de tratamento usando a técnica de divisão e conquista. Pretende-se explorar as duas formas clássicas de paralelismo: computação local usando memória partilhada (nível multinúcleo) e computação distribuída usando troca de mensagens (nível multimáquina).

Palavras-chave: Paralelismo, Sistemas distribuídos, SOMD, Akka, Multinúcleo, Cluster

Abstract

Programming has not accompanied in a simplified way the evolutions of hardware verified over the years. The features available in today's multi-core processors and high-performance clusters are still not being fully exploited by non-specialized programmers.

The high-level Single Operation Multiple Data (SOMD) programming model introduces a new idea that aims to vulgarize the execution of applications in parallel. The model intends, with the addition of some annotations, to be able to execute in multi-core and distributed systems subroutines previously written and thought sequentially.

The main objective of the project of this dissertation is the creation of an implementation in Scala, using the Akka programming framework, of the SOMD model. The solution will be able to run in parallel sequential code that is amenable to treatment using the technique of divide and conquer. It is intended to explore the two classic forms of parallelism: local computing using shared memory (multi-core level) and distributed computing using message exchange (multi-machine level).

Keywords: Parallel computing, Distributed computing, SOMD, Akka, Multi-Core, Cluster

Índice

Lista de Figuras xv

Lista de Tabelas xvii

Listagens xix

Glossário xxi

Siglas xxiii

1 Introdução 1
- 1.1 Contexto 1
- 1.2 Motivação 1
- 1.3 Problema 2
- 1.4 SOMD 2
  - 1.4.1 Modelo de execução 2
- 1.5 Objetivos 4
- 1.6 Contribuições 4
- 1.7 Estrutura do documento 4

2 Estado da Arte 7
- 2.1 Introdução 7
- 2.2 Modelo APGAS 7
  - 2.2.1 X10 8
    - 2.2.1.1 Exemplo 10
    - 2.2.1.2 Outras implementações 10
  - 2.2.2 Chapel 11
    - 2.2.2.1 Exemplo 12
- 2.3 MapReduce 12
  - 2.3.1 Modelo 12
  - 2.3.2 Distribuição 14
  - 2.3.3 Hadoop 16

2.3.4 Spark 16
2.3.4.1 Exemplo 18
2.4 PCT 18
2.4.1 Modelo 18
2.4.2 Exemplo 19
2.5 Considerações finais 19

3 Akka **21**
3.1 Atores 21
3.1.1 Referência 22
3.1.2 Estado 22
3.1.3 Caixa de Entrada 22
3.1.4 Supervisão 23
3.1.5 Terminação 23
3.2 Akka Cluster 23
3.3 Exemplo 24

4 Scala/SOMD **27**
4.1 Modelo 27
4.2 Descrição semântica 28
4.3 Partições 29
4.3.1 Partição automática 30
4.3.1.1 *BogoMips* 30
4.3.1.2 *Des3* 30
4.3.2 Comunicação entre trabalhadores 31
4.3.2.1 Passagem por valor 31
4.3.2.2 Passagem por referência 32
4.4 Implementação modular recursiva 33
4.5 Tradução 35
4.6 Arquitetura de atores 36
4.6.1 Ator mestre 36
4.6.2 Paralelismo local 37
4.6.3 Paralelismo distribuído 37
4.7 Paralelismo embaraçoso 38

5 Avaliação **39**
5.1 Eficiência e Escalabilidade 39
5.2 *Speedup* 41
5.3 Facilidade de programação 42
5.3.1 Nota 44

6 Conclusões e trabalho futuro **47**

6.1 Cache 47
6.2 Serializador 48
6.3 Atores Akka 48
6.4 Anotações 50
6.5 Bibliotecas internas 51
6.6 Paradigma funcional em *Scala* adaptado a computações de alta performance 51

Bibliografia **53**

Lista de Figuras

1.1 Execução do modelo *SOMD* [12] 3

2.1 PGAS [13] 9
2.2 Execução do MapReduce [6] 14
2.3 Execução Paralela do MapReduce [6] 15

5.1 Testes artificiais 40
5.2 SOMD vs coleções paralelas do *Scala* 42

Lista de Tabelas

4.1 Tradução de elementos pelo *Scala/SOMD* 35

5.1 Comparação de *speedups* e eficiências entre ambientes de teste 41

Listagens

1.1 Soma de dois *arrays* em *Scala/SOMD* 4
2.1 Somatório de matriz em X10 11
2.2 Multiplicação de matrizes em Chapel 13
2.3 Pseudocódigo MapReduce [6] 14
2.4 Somatório MapReduce 17
2.5 Somatório em Spark 18
2.6 Multiplicação de matrizes em MATLAB 19
3.1 PingPong em Akka 25
3.2 Pong remoto em Akka 25
4.1 Soma de dois *arrays* em *Scala/SOMD* 28
4.2 Array1Partition 29
4.3 Soma de dois *arrays* em *Scala/SOMD* usando passagem por valor 31
4.4 Soma de dois *arrays* em *Scala/SOMD* usando passagem por referência . . 32
4.5 Interface *ChunkK* para coleções 34
4.6 Tradução da soma de dois *arrays* efetuada pelo *scala/SOMD* 35
4.7 Ausência de paralelismo embaraçoso 38
5.1 Soma de dois *arrays* em *Scala* 42
5.2 Soma de dois *arrays* em *Scala/SOMD* 43
5.3 Soma de dois *arrays* paralelos *Scala* 43
5.4 Soma dos elementos de uma árvore binária em *Scala/SOMD* 44
5.5 Soma de dois *arrays* em *Spark* 44
5.6 Soma de dois *arrays* em *Scala/SOMD* 45
5.7 Soma dos elementos de uma àrvore binária em *Scala/SOMD* 45
6.1 somdActor 49
6.2 Simplificação de uma chamada à solução para a soma de dois *Arrays* . . . 50

Glossário

Ator	Unidade primitiva de computação, algo que recebe uma mensagem e efetua, isoladamente, alguma computação nela baseada.
Chunk	Em *scala/SOMD*, *chunk* é o nome dado a cada parte de uma coleção já repartida.
Cluster	Computadores ligados entre si que trabalham em conjunto, tanto que, em alguns aspetos, podem ser considerados como um único sistema.
Núcleo	Unidade de processamento que recebe instruções de forma a executar cálculos ou ações. Um conjunto destas instruções é o que permite a um programa executar uma função. Um processador contém um mais núcleos.
P. Funcional	Paradigma de programação que trata a computação como uma avaliação de funções matemáticas, em contraste com a programação imperativa, que se baseia em mudanças de estado.
Thread	Parte de um programa que é executada de forma independente e concorrente com o resto do programa.

Siglas

JVM	Java virtual machine.
PCT	Parallel Computing Toolbox da linguagem MATLAB.
SOMD	Single Operation Multiple Data.
SPMD	Single Program Multiple Data.

Introdução 1

1.1 Contexto

Já desde a década de 60 que se ouve falar em computação concorrente, paralela e distribuída. Nessa altura apareceram as primeiras máquinas multiprocessador (e.g. *Multics*) e os primeiros exemplos de processamento distribuído (e.g. *ARPANET*), mas tratavam-se ainda de situações experimentais fora da prática da maioria dos centros de processamento de dados.

A computação distribuída é cada vez mais importante nos dias de hoje. Normalmente, este tipo computação é associado a dois tipos de aplicações: servidores que têm de aceitar várias ligações em simultâneo e computação de grandes quantidades de dados.

A computação paralela com memória partilhada também se tem tornado cada vez mais importante, especialmente com o aparecimento recente, mas já generalizado, de processadores multinúcleo.

Infelizmente, a programação paralela continua a ser vista como um ramo da informática complicado e apenas direcionado para programadores especializados. A situação é complicada para o programador comum que queira tirar partido de todos os recursos disponíveis de uma máquina ou *cluster*.

1.2 Motivação

Atualmente já existem alguns modelos de programação de alto nível que pretendem oferecer algum tipo de abstrações ao programador para facilitar a escrita de aplicações paralelas. Alguns exemplos mais comuns destas abstrações são ao nível de transmissão de mensagens e gestão de acesso a endereços em sistemas de partilha de memória. O objetivo de acrescentar mais abstrações nestas ferramentas é permitir que estas possam ser usadas por um programador não especialista no tema da programação paralela.

Mesmo com o aparecimento destes modelos de programação paralela, ainda existem poucas construções linguísticas que permitam ao programador tirar facilmente partido do *hardware* paralelo que tem à sua disposição. Acontece que, apesar da mudança de paradigma para as arquiteturas de computadores atuais (*CPUs*, *GPUs*, *Clusters*) já ter

acontecido há alguns anos, a programação paralela ainda não se tornou comum para programadores não especializados na área.

1.3 Problema

O problema da escrita de programas paralelos é na sua essência complexo e para ele não há soluções miraculosas. No entanto na programação do dia a dia surgem frequentemente situações do chamado *código embaraçosamente paralelo*, sendo este o alvo dos esforços deste trabalho.

Portanto, faz falta um mecanismo linguístico simples que permita executar de forma paralela código sequencial suscetível de tratamento usando a técnica de divisão e conquista. Idealmente seria explorada a computação local usando memória partilhada e computação distribuída usando troca de mensagens.

1.4 SOMD

O modelo *Single Operation Multiple Data* (*SOMD*) [12] apresenta uma metodologia que permite que uma sub-rotina previamente escrita para funcionamento sequencial seja executada de forma paralela apenas com o acrescento de algumas anotações para orientação da partição e redução dos dados.

O conceito de paralelismo é normalmente divido em dois tipos distintos, paralelismo de tarefas e de dados. Geralmente os modelos que trabalham com paralelismo de tarefas baseiam-se em oferecer ao programador algumas sub-rotinas (funções, métodos, etc) que podem ser executadas por um grupo de *threads* que trabalham para a aplicação. Já o paralelismo de dados é habitualmente oferecido através de ciclos *for*, em que os dados são divididos pelas *threads* trabalhadoras e todas elas executam individualmente o mesmo código.

O modelo *SOMD* trabalha com paralelismo de dados, mas ao nível das sub-rotinas. No contexto do *SOMD*, são criadas várias instâncias de uma sub-rotina de modo a que o código seja aplicado em paralelo a diferentes porções dos dados. Estas instâncias são executadas em conformidade com uma variação do modelo de execução *Single Program Multiple Data* (*SPMD*), de onde vem o nome *SOMD*. Esta abordagem é o que vem permitir que a utilização deste modelo passe por uma simples adição de anotações em sub-rotinas sequenciais inalteradas.

1.4.1 Modelo de execução

O modelo de execução do *SOMD* baseia-se na criação de várias instâncias de um método que se pretende paralelizar, em que cada instância opera sobre uma parte dos dados iniciais.

Neste modelo, a invocação é separada da execução, uma característica que faz com que o paralelismo que é alcançado com a chamada do *SOMD* seja transparente ao utilizador, isto é, a invocação é tratada como sendo síncrona. Assim, tal como num método normal de linguagens conhecidas como o *Java*, a aplicação fica à espera de uma resposta.

O modelo é apresentado como um paradigma *Distribute-Map-Reduce (DMR)* que partilha algumas ideias com o *MapReduce*:

Distribute Particiona os dados a serem usados em coleções de dados do mesmo tipo. Por exemplo, se estivermos a trabalhar com uma lista de inteiros, esta fase vai repartir a lista inicial em listas mais pequenas. Podem ser declarados no mesmo método vários parâmetros e variáveis para serem distribuídos.

Map Aplica uma instância da sub-rotina a cada porção dos dados.

Reduce Combina os resultados parciais do *Map* de modo a poder computar o resultado final.

É esta a formula que permite que uma sub-rotina não tenha de ser alterada para poder ser executada de forma paralela. Dado um argumento de um tipo *T*, a distribuição a efetuar será uma função *T* -> *List* <*T*>. A redução aplicada a um método que devolva um tipo R é uma função de tipo *List* <*R*> -> *R*. Assim, as instâncias de uma sub-rotina a ser usada pelo *SOMD* mantêm-se iguais à original. Cada instância passa a receber uma parte da coleção distribuída (List <T>) e a devolver um valor do tipo *R*. A Figura 1.1 mostra um exemplo abstrato da execução do modelo *SOMD*.

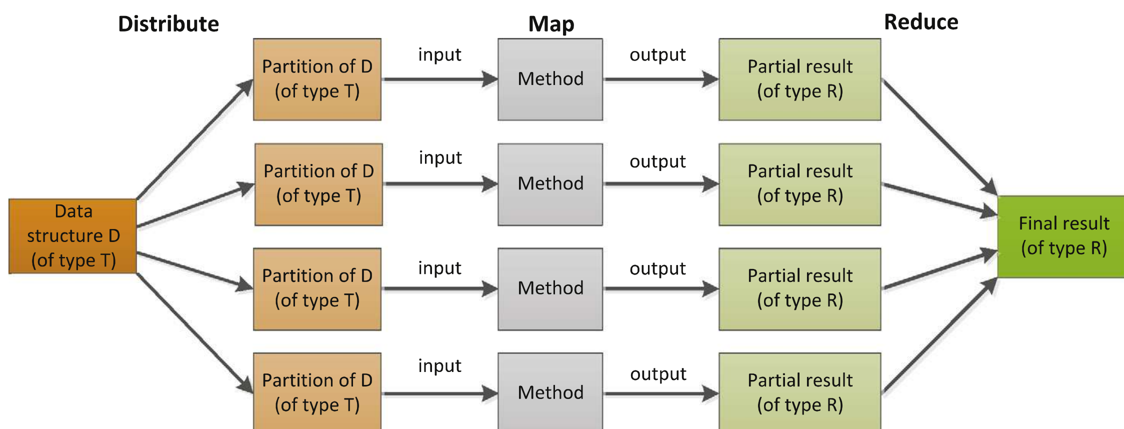


Figura 1.1: Execução do modelo *SOMD* [12]

1.5 Objetivos

A listagem 1.1 mostra um exemplo de utilização da ferramenta *Scala/SOMD*. Esta permite expressar, a partir de anotações, uma sub-rotina que se pretende que seja executada de forma paralela ou distribuída por uma ou mais máquinas multinúcleo, de acordo com o modelo *SOMD* 1.4.

Listagem 1.1: Soma de dois *arrays* em *Scala/SOMD*

```
1  @COMBINE[ParallelList]
2  @REDUCE( (partials : Array[Array[Double]]) => partials.flatten )
3  def arrayAddition (@DIST[Array1PartitionRef] a : Array [ Double ],
4      @DIST[Array1PartitionVal] b : Array [ Double ]) : Array[Double] =
         {
5    assert(a.length == b.length)
6    val res = new Array[Double](a.length)
7    for ( i <- a.indices )
8      res(i) = a(i) + b(i);
9    res
10 }
```

Apesar da solução ter tomado como ponto de partida uma implementação protótipo do modelo *SOMD* para computações locais sobre memória partilhada, pretende-se estender a implementação para sistemas distribuídos e descobrir como criar uma boa integração entre os dois tipos de paralelismo.

Pretende-se também aperfeiçoar, flexibilizar e generalizar os mecanismos existentes para o seu campo de aplicação e facilidade de utilização.

1.6 Contribuições

A primeira contribuição desta dissertação consiste numa implementação dos conceitos do modelo *SOMD* usando paralelismo híbrido: computação distribuída usando troca de mensagens (nível multimáquina) e computação local usando memória partilhada (nível multinúcleo).

A segunda contribuição será a determinação dos ganhos de eficiência face a execuções sequenciais básicas de alguns problemas, assim como à sua implementação usando ferramentas concorrentes.

1.7 Estrutura do documento

A presente dissertação está organizado da seguinte forma.

No Capítulo 2 são apresentados alguns modelos (e suas implementações) de programação de sistemas paralelos. Tal como o *SOMD*, estes modelos são de muito alto nível pois a procura incidiu sobre implementações que apresentam uma maior simplicidade de uso para o programador.

O Capítulo 3 introduz a *framework Akka* e o respetivo modelo de atores. Esta ferramenta é usada na solução deste projeto. A própria implementação previamente existente *Scala/SOMD* já fazia uso desta ferramenta.

O Capítulo 4 descreve a ferramenta *Scala/SOMD*, uma implementação do modelo *SOMD* para sistemas multinúcleo e multimáquina. Trata-se da solução encontrada nesta dissertação.

O Capítulo 5 apresenta alguns resultados obtidos na utilização da versão final do *Scala/SOMD* obtida nesta dissertação. A partir destes resultados, a solução é avaliada em alguns aspetos como escalabilidade e ganhos em relação a execuções sequenciais.

Concluindo, o Capítulo 6 apresenta uma reflexão sobre a qualidade de metodologias e ferramentas usadas na solução apresentada. Esta reflexão, para além de esclarecedora, pode servir como motivação para, no futuro, melhorar a solução.

Estado da Arte 2

2.1 Introdução

O *SOMD* é capaz de executar de forma paralela funções escritas de forma sequencial. O principal foco é permitir a escrita de programas capazes de utilizar os recursos disponíveis, tanto em máquinas multinúcleo como em *clusters*, de uma maneira que tenha pouco impacto nos hábitos de escrita do programador.

Neste Capítulo, são estudados alguns modelos de programação paralela de muito alto nível que, não obstante, presumem que seja o utilizador a controlar a forma como os dados são repartidos e reagrupados. Atualmente, este parece ser um dos melhores balanços entre a facilidade de programação e o uso eficiente dos recursos disponíveis.

No contexto da programação distribuída, o número de diferentes exemplos que se podem estudar é reduzido. Os modelos de distribuição dos exemplos deste Capítulo cobrem quase todas as soluções existentes. Para cada implementação dos modelos é apresentado um exemplo onde é possível observar alguns dos construtores existentes que permitem obter distribuição.

2.2 Modelo APGAS

Existem alguns modelos de programação de alto nível que lidam de diferentes formas com um dos problemas da programação paralela, a gestão de memória.

O *MPI* [11] continua a ser o modelo padrão quando falamos em grandes sistemas distribuídos. Trata-se de um modelo baseado em transmissões de mensagens entre as várias máquinas de um sistema, em que as memórias locais de cada máquina são privadas. O *MPI* é normalmente usado para implementações de sistemas distribuídos *SPMD*, onde cada máquina executa sequencialmente a sua tarefa.

Normalmente usado para programação local multinúcleo, o *OpenMP* [5] é um modelo de memória partilhada. Aqui os processos não comunicam por mensagens mas sim através de um espaço de endereços de memória comum onde ficam guardados os dados com que todo o sistema está a trabalhar.

Com base nestes dois modelos começaram a aparecer implementações de sistemas paralelos de duas fases, usando o *MPI* para distribuir os dados pelas máquinas e o *OpenMP* para que se possa usar todos os núcleos e processadores de cada máquina.

Começaram também a surgir modelos de mais alto nível de memória virtualmente partilhada como é o caso do *TreadMarks* [9]. Nestes modelos, cada máquina executa o código como se de um modelo de memória partilhada se tratasse, porém os dados podem não estar guardados localmente. Estes modelos pecam exatamente por isso, quando se está a tentar aceder a um endereço de memória não se sabe qual o local onde este está guardado. Em grandes sistemas, aceder várias vezes a locais remotos pode levar a um *overhead* de largura de banda.

O *PGAS (Partitioned Global Address Space)* [1] pretende resolver este problema estendendo o modelo de memória partilhada para um modelo que possa ser eficientemente usado em sistemas paralelos de dois níveis. No *PGAS* cada processo, ou *thread*, tem um espaço de endereços privados, porém, o conjunto total de endereços é unificado, isto é, a memória privada de um processo pode ter um endereço que aponta para outro endereço da memória privada de outro processo. Assim, no *PGAS* uma estrutura de dados pode ser alocada de forma privada ou global. As estruturas de dados globais encontram-se distribuídas pelos diferentes espaços de endereços, normalmente, sob o controlo do programador. Independentemente da divisão efetuada, qualquer processo pode aceder a dados marcados como globais de outro processo através de mensagens de rede, sendo que a divisão inicial serve apenas para garantir uma maior eficiência de todo o sistema. A Figura 2.1 ilustra o formato de gestão de memória do *PGAS* em comparação com o do *MPI* e o do *OpenMP*.

O modelo *PGAS* torna-se um modelo mais conveniente de ser usado em implementações de sistemas paralelos em *clusters*. Neste modelo, em contraste com um modelo de memória virtualmente partilhada, é o próprio programador que define onde estão e quais são os dados que se querem partilhados.

O *PGAS* é um modelo síncrono e ideal para problemas *SPMD*. O modelo *APGAS* [13] vem acrescentar ao *PGAS* o conceito de não sincronia permitindo assim o aparecimento de dois pontos importantes: variedade de *hardware* entre processos e capacidade de lidar com o começo, de forma dinâmica, de novas atividades (*threads*) em tempo de execução.

2.2.1 X10

Desenvolvido como uma forma de demonstração do modelo *APGAS*, *X10* é uma linguagem de programação de alta performance que procura aumentar a produtividade em sistemas de memória partilhada com distribuição de larga escala.

Apesar do aspeto inovador de ser uma linguagem de programação orientada para o paralelismo, o *X10* é implementado sobre alguns conceitos já conhecidos de outras linguagens correntes: é baseado em classes, fortemente tipificado, orientado a objetos e tem um sistema de coleção de lixo.

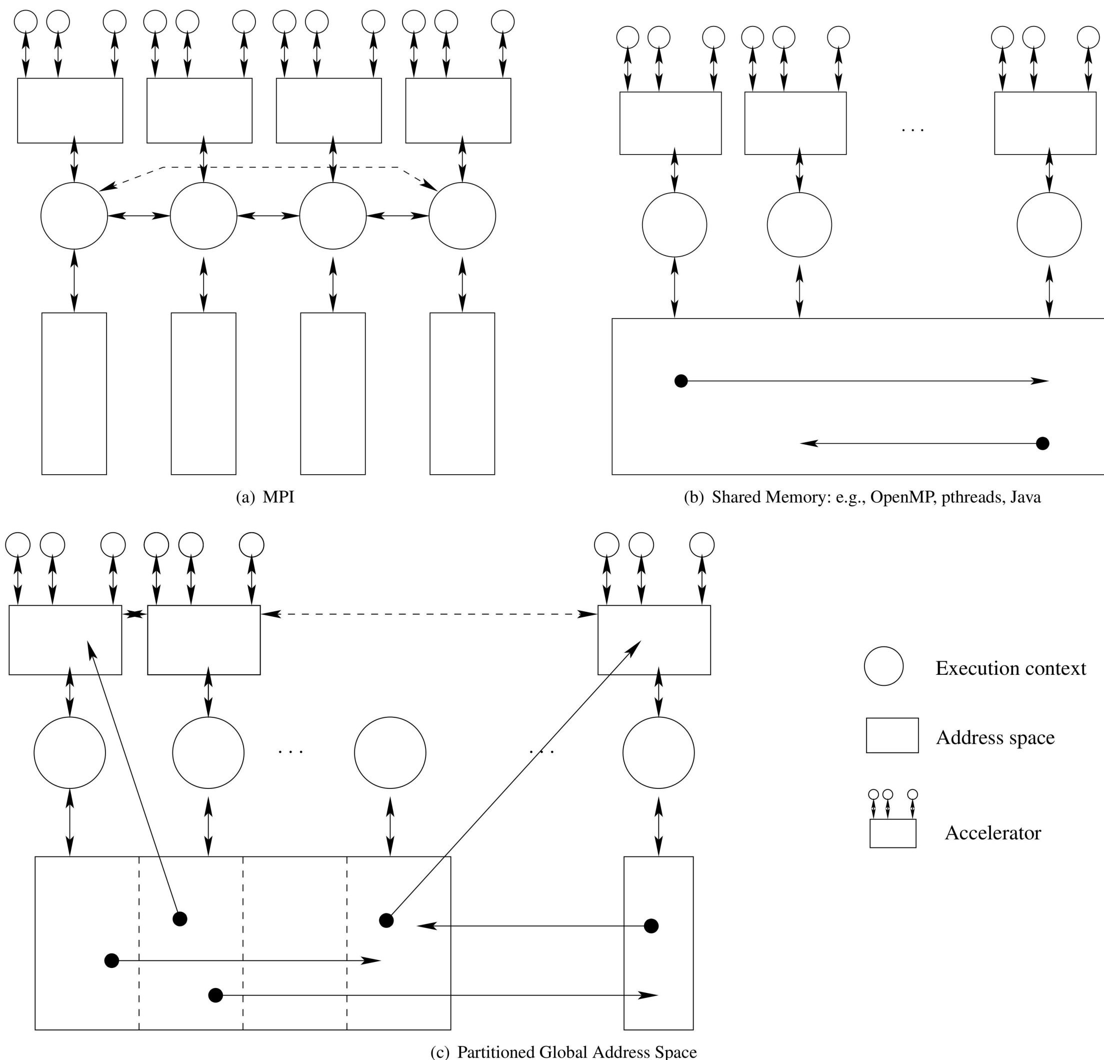


Figura 2.1: PGAS [13]

O *X10* implementa o modelo *APGAS* através de dois novos conceitos: *Places* e *Asyncs*.

Place Nome dado a cada porção do espaço de endereços. Contém tanto os respetivos dados dessa porção como as atividades (*threads*) que sobre eles trabalham. Os dados não são migráveis entre *Places* e consistem em objetos mutáveis. Em *X10*, normalmente, uma computação é composta por vários *Places*.

Mesmo depois da divisão do sistema em *Places*, cada *Place* pode também ser *multi-threaded*, isto é, múltiplas atividades podem correr sobre o mesmo *Place*. Os *Places* têm também a capacidade de correr em qualquer tipo de arquitetura de processadores. Apesar do número de *Places* ser fixo logo a partir do momento em que o sistema começa a executar, existe a opção de serem criados de forma hierárquica.

Async Anotação que permite expressar uma atividade a ser executada num *Place*. Normalmente utilizada em conjunto com o termo *at* que permite especificar qual o *Place* de destino para ser executada a atividade.

As atividades não têm de corresponder diretamente a um processo ou *thread*, isto é, são simples tarefas que podem ser chamadas local ou remotamente. No modelo *APGAS* uma atividade pode mudar de *Place* a meio de uma execução. Porém, no *X10* esta funcionalidade apresenta-se de uma forma mais restrita em que uma atividade apenas pode começar outra atividade (*async*) num *Place* remoto de modo a poder ter acesso aos seus dados.

Num uso normal do *X10* o programador recorre ao mecanismo *finish* que espera que todas as atividades acabem as suas execuções em todos os *Places*. Esta sincronização torna-se possível pois as atividades são organizadas em árvore, em que a atividade do nível anterior corresponde à que começou a execução da do nível atual.

2.2.1.1 Exemplo

O exemplo da Listagem 2.1 mostra o quão o *X10* é baseado em *Java* e *C++*, destacando desde logo o facto de todo o programa de somatório de uma matriz estar contido na classe *DistArray2DSum*.

O método *main* cria uma matriz *DistArray_BlockBlock_2* pronta para ser distribuída pelos *Places* existentes, com uma partição em blocos.

No método *sumOpt* o ciclo *for* navega pelos *places* existentes com recurso às denotações já descritas *at* e *async*. Cada *place* calcula a soma dos valores da matriz local e o resultados são juntos num *SumReducer* que calcula a soma total. A denotação *finish* garante que o processo não avança até a obtenção do resultado desejado, ou seja, até todos os *places* devolverem as suas computações.

2.2.1.2 Outras implementações

O *X10*, não obstante de ser uma linguagem, também tem bibliotecas que permitem usar os seus conceitos em *Java*.

Ainda assim, durante a sua história de desenvolvimento, houve um momento em que os desenvolvedores abandonaram o *Java* concentrando-se apenas na versão para linguagem *C++*. Aqui começaram a aparecer outras soluções, baseadas nas versões *X10* anteriores a este abandono, como o *Habanero-Java* [2]. Isto também fez com que o *X10* começasse a aparecer na linguagem *Scala*.

Listagem 2.1: Somatório de matriz em X10

```
public class DistArray2DSum {
  static N = 9000;

  static def sumOpt(a:DistArray_BlockBlock_2[Double]):Double {

    val sum = finish(Reducible.SumReducer[Double]()) {
      for(p in a.placeGroup()) at(p) async {
        var localSum:Double = 0;
        for(pt in a.localIndices()) localSum += a(pt);
        offer localSum;
      }
    };
    return sum;
  }

  public static def main(Rail[String]) {
    val v = new DistArray_BlockBlock_2[Double](N, N, (i:Long,j:Long)=>
    (i+j) as Double);
    Console.OUT.println("Somatório:␣" + sumOpt(v));
  }
}
```

2.2.2 Chapel

O nome *Chapel* vem de *Cascade High Productivity Language*. Ao contrário do *X10*, que apesar de ser uma nova linguagem, funciona sobre o *Java*, o *Chapel* é totalmente independente, não construído sobre qualquer linguagem. Apesar de ir buscar alguns elementos de sintaxe a linguagens de programação conhecidas como o *C* ou o *Fortran*, o *Chapel* é construído de raiz com o objetivo de ser uma linguagem altamente produtiva para aplicações de paralelismo.

Os próprios autores do *Chapel* esclarecem que era necessária uma linguagem diferente das usadas atualmente para não fazer com que um programador volte aos hábitos de programação sequencial. De facto, o modo ideal de programar aplicações paralelas não passa por fazer uma tradução de um código sequencial mas sim escrever uma versão com os princípios de paralelismo em mente logo de inicio.

Em *Chapel* dá-se o nome *Locale* às unidades de memória partilhada. Tal como em *X10*, numa arquitetura de sistemas distribuídos, cada nó vai corresponder a um *locale*. Podem existir ter múltiplas *threads* a operar sobre cada *locale*.

Mesmo trabalhando numa única máquina cada programa *Chapel* começa automaticamente com um *array* de *locales* com tamanho dependente do tipo de processador ou processadores da máquina. Cada *locale* corresponde a uma parte da máquina (processadores, núcleos, etc) onde vão ser executados os processos associados.

Num código *Chapel* o paralelismo acontece onde aparecem declarações como *forall*, *begin* e *cobegin* ou variáveis paralelas como *single*, que não é mutável, e *synch* que pode variar de valor. O *forall* indica ao compilador que as instruções nele contidas podem ser

executadas de forma independente para cada iteração. O *begin* inicia uma nova computação que pode ser executada em paralelo com o processo que a inicia. O *cobegin* serve como forma de criar vários processos ao mesmo tempo ao estilo do *begin*.

2.2.2.1 Exemplo

A Listagem 2.2 mostra um exemplo de multiplicação de matrizes distribuídas em *Chapel*.

São criadas três matrizes 24x24, duas preenchidas de forma aleatória e uma que servirá para guardar os resultados finais.

A multiplicação de matrizes multiplica uma linha da primeira matriz com uma coluna da segunda. No código, a matriz A é distribuída pelas suas linhas e B pelas colunas. O *Chapel* não tem uma forma automática de fazer estas distribuições mas os valores *boundingBox* e *targetLocales* junto com, neste caso, *BlockDist*, permitem manualmente efetuar qualquer divisão.

Com a linguagem *Chapel* consegue-se sempre criar aplicações distribuídas muito simplificadas, porém, esta linguagem tem uma grande curva de aprendizagem devido ao elevado número de conceitos existentes.

2.3 MapReduce

O *MapReduce* [6], proposto pela empresa *Google, Inc.*, trata-se de um modelo de programação e respetiva implementação. O nome *MapReduce* provém de duas funções de alto nível conhecidas do mundo da programação, o *Map* e o *Reduce*. Este modelo surge pela constatação que muitas das funções que eram utilizadas, neste caso, dentro da *Google, Inc.* se podiam traduzir numa sequência de operações *Map/Reduce*.

Map é uma função que trata de aplicar uma outra função (recebida) a todos os elementos de uma coleção. O método devolve uma nova coleção, normalmente da mesma ordem mas em que cada índice pode passar a corresponder a um ou mais valores, por exemplo, se a função a ser aplicada devolver uma lista.

A função *Reduce* é utilizada para reduzir uma estrutura de dados, normalmente, num único valor aplicando uma outra função (recebida) à estrutura. Imaginando que uma função *Map* devolve uma lista de listas, o *Reduce* é capaz de combinar os valores de todas as listas internas devolvendo uma única lista de elementos.

2.3.1 Modelo

Este modelo tem como objetivo processar um conjunto de pares chave/valor num novo conjunto com novos valores calculados a partir da aplicação das funções *Map* e *Reduce*, expressas pelo utilizador, ao conjunto inicial. A Figura 2.2 mostra um exemplo de execução.

Listagem 2.2: Multiplicação de matrizes em Chapel

```
use Random;
use BlockDist;

config const N = 24;
config const K = 24;

const Space = {1..N, 1..K};

var Aview = {0..#numLocales, 1..1};
var Atarget: [Aview] locale = reshape(Locales, Aview);
const Aspace: domain(2) dmapped Block(boundingBox=Space,
          targetLocales=Atarget) = Space;

var A : [Aspace] int;
fillRandom(A, 100);

var Bview = {1..1, 0..#numLocales};
var Btarget: [Bview] locale = reshape(Locales, Bview);
const Bspace: domain(2) dmapped Block(boundingBox=Space,
          targetLocales=Btarget) = Space;

var B : [Bspace] int;
fillRandom(B, 100);

const Rspace: domain(2) dmapped Block(boundingBox=Space,
          targetLocales=Atarget) = Space;
var R : [Rspace] int;

forall i in 1..N do {
  forall j in 1..K do {
    forall k in 1..N do {
      R[i,j] += A[i,k] * B[k,j];
    }
  }
}

writeln(R);
```

Cada tarefa *Map* recebe parte da coleção inicial e produz um conjunto intermédio de pares chave/valor.

Em seguida, a biblioteca *MapReduce* agrupa todos os resultados intermédios com a mesma chave criando uma estrutura chave/lista de valores. Estas estruturas são enviadas para tarefas *Reduce*.

A função *Reduce* aceita os os valores de uma só chave de modo a poder conjugá-los num conjunto mais pequeno, normalmente de tamanho zero ou um.

A Listagem 2.3 corresponde ao pseudocódigo de um exemplo clássico do *MapReduce*, a contagem do número de ocorrências de cada palavra num texto. A função *Map* recebe o texto e para cada palavra (chave) encontrada devolve o valor um, assim cada chave vai corresponder a uma lista destes valores. A função *Reduce* recebe o par chave/lista de valores e devolve o somatório dos valores da lista, neste caso, o número de ocorrências da

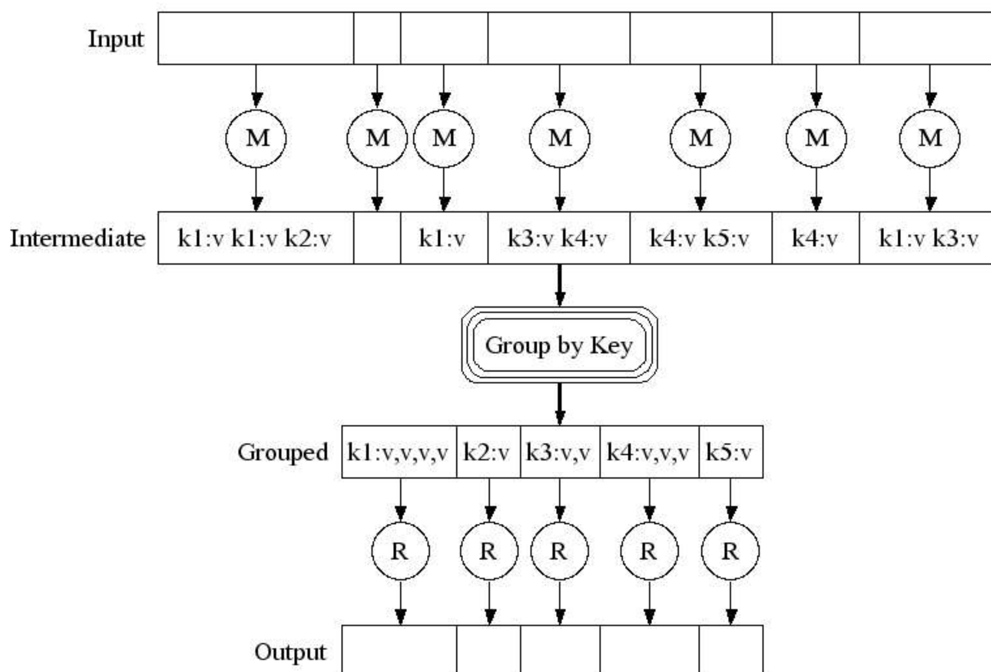


Figura 2.2: Execução do MapReduce [6]

Listagem 2.3: Pseudocódigo MapReduce [6]

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

chave no texto. O resultado final da chamada do *MapReduce* será um conjunto de pares palavra/número de ocorrências.

2.3.2 Distribuição

As funções *Map* e *Reduce*, para além de independentes uma da outra, são independentes entre as suas próprias instâncias.

Uma tarefa designada para executar *Map* recebe uma parte dos dados iniciais e sem

precisar de mais informações devolve os pares chave/valor encontrados no bloco de ficheiro recebido. Podemos ter tantas tarefas *Map* quanto desejado.

Em seguida, cada tarefa *Map* terá de saber a qual tarefa, designada a executar *Reduce*, terá de enviar os valores intermédios calculados. Para isto, o *MapReduce* usa uma função de partição sobre as chaves intermédias. Por omissão, é usada uma função *hash* que, para além de olhar para as próprias chaves, tem como objetivo dividir o trabalho de forma justa pelas tarefas *Reduce*. A função de partição pode também ser dada pelo utilizador para casos em que são necessários outros tipos de divisões, por exemplo, se as chaves corresponderem a *urls* podemos querer uma divisão por domínio. De referir que cada uma das diferentes tarefas *Reduce* devolve os resultados num ficheiro diferente.

O modelo de execução distribuída do *MapReduce* pode ser observado na Figura 2.3.

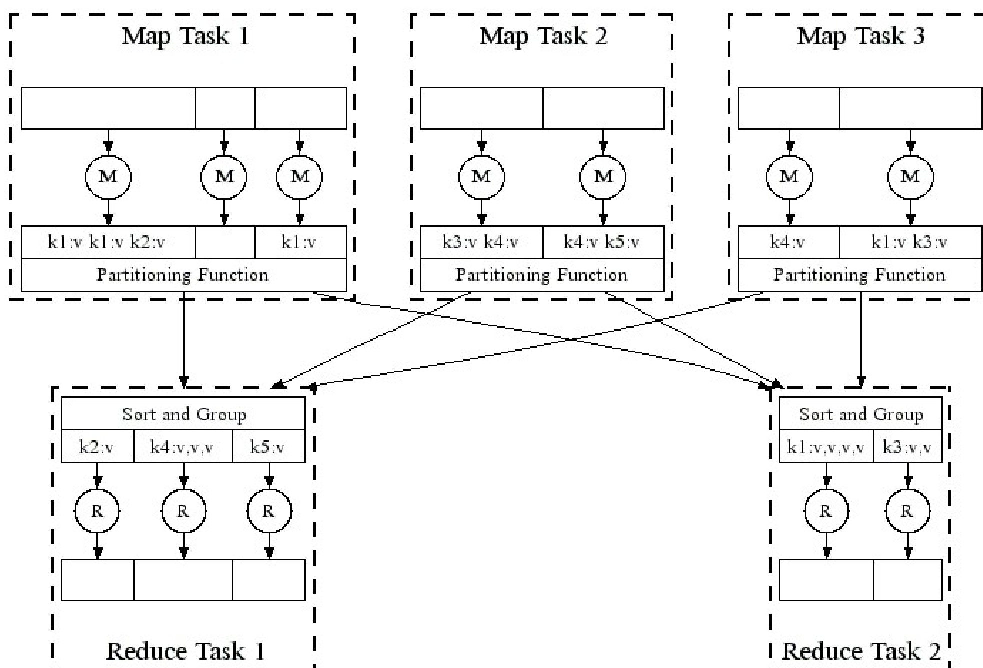


Figura 2.3: Execução Paralela do MapReduce [6]

Apesar de ser o utilizador que define o número de tarefas que devem ser executadas, é sempre aconselhado uma divisão de maior granularidade, isto é, muito mais tarefas do que o número de máquinas. Os autores do modelo apresentam como ordem ideal uma de 200.000 tarefas *Map* e 5.000 tarefas *Reduce* para um sistema com 2.000 máquinas. Com esta granularidade estamos a diminuir o tempo de tratamento de falhas e a melhorar o balanço da divisão dos dados.

2.3.3 Hadoop

A Listagem 2.4 mostra um exemplo de utilização do *MapReduce* para o somatório de todos os valores encontrados no ficheiro de entrada. O exemplo foi escrito usando o *Hadoop*, uma implementação do *MapReduce* em *Java*.

De notar que logo na função *main* são necessárias algumas configurações para se poder trabalhar com este modelo. Estas configurações dão mais controlo ao programador para poder adaptar o *MapReduce* ao seu sistema mas tornam todo o processo mais complicado.

O método *map*, inserido na classe *TokenizerMapper*, calcula logo a soma dos valores correspondentes ao seu bloco de ficheiro recebido. Todas as funções *Map* devolvem o resultado na mesma chave pois os valores são todos dependentes entre si, têm de ser somados.

O método *reduce*, inserido na classe *IntSumReducer*, soma todos os valores recebidos e é executado apenas uma vez pois só queremos um único resultado final.

2.3.4 Spark

O sistema *Spark* [20] é uma implementação do conceito *Resilient Distributed Datasets* (RDDs) apresentado pelos mesmos autores. Baseado no *MapReduce*, que já apresentou várias abstrações ao nível da utilização dos recursos de cada nó de um sistema, o *Spark* vem acrescentar mais abstrações no que toca ao uso de memória distribuída. Os RDDs são de estruturas de dados paralelas que permitem guardar os dados intermédios em memória e que oferecem ao utilizador operações de persistência e de particionamento, isto é, os dados intermédios guardados em RDDs podem ser reutilizados.

Apesar de também poder ter um uso geral, o *Spark* apresenta-se como um sistema direcionado para aplicações que possam usufruir de memória partilhada, isto é, aplicações que precisam de reutilizar os cálculos intermédios nas suas computações. Os algoritmos onde é mais comum este tipo de uso da memória são os de aprendizagem automática e os que assentam sobre grafos, por exemplo, o *PageRank*, o *K-means* e a *Logistic Regression*.

Em *MapReduce*, a implementação de todos estes casos requer que os dados que se pretendem reutilizar sejam escritos em memória não volátil, usando um sistema de ficheiros partilhado, o que torna o sistema muito pesado e lento devido à quantidade de operações de leitura e escrita.

Para além deste direcionamento para aplicações de memória partilhada, o *Spark* apenas trás vantagens às aplicações que trabalhem com grandes quantidades de dados e que se baseiem em efetuar a mesma operação a todos os elementos de um conjunto de dados. Isto porque os RDDs, ao contrário do que normalmente é usado em sistemas de partilha de memória, estão desenhados num modelo *coarse-grained*. Isto significa que as operações são efetuadas em grandes conjuntos de dados de uma só vez e que a informação que fica guardada para backup não são os próprios dados mas sim o historial de transformações sobre estes, o que permite uma recuperação de dados mais eficiente.

Listagem 2.4: Somatório MapReduce

```
public class Sum {

  public static class TokenizerMapper extends Mapper<Object, Text, Text,
      IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text("sum");

    public void map(Object key, Text value, Context context) throws
        IOException, InterruptedException {

      StringTokenizer itr = new StringTokenizer(value.toString());
      int intermediate_sum = 0;
      while (itr.hasMoreTokens()) {
        intermediate_sum+=Integer.parseInt(itr.nextToken());
      }
      context.write(word, new IntWritable(intermediate_sum));
    }
  }

  public static class IntSumReducer extends Reducer<Text,IntWritable,Text,
      IntWritable> {

    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {

      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(new Text("somatório"), result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "sum");
    job.setJarByClass(Sum.class);
    job.setNumReduceTasks(1);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Apesar de todas estas limitações de uso, o *Spark* apresenta grandes melhorias de performance para os casos que tem como objetivo, chegando a ser, segundo os autores, 100x mais rápido que o *hadoop* em casos de *Logistic Regression*.

2.3.4.1 Exemplo

A Listagem 2.5 mostra um exemplo de somatório dos valores contidos no ficheiro de entrada.

Apesar de neste exemplo não se precisar de reutilizar dados intermédios, podemos ver que o *Spark*, até por estar implementado em *Scala*, torna o exemplo mais simples.

Neste caso as somas são todas feitas no *Reduce* e o *Map* apenas tem a função de juntar os valores todos na mesma chave.

Listagem 2.5: Somatório em Spark

```
1  object Sum {
2    def main(args: Array[String]) {
3      val inputFile = args(0)
4      val outputFile = args(1)
5      val conf = new SparkConf().setAppName("Sum")
6      val sc = new SparkContext(conf)
7      val input =  sc.textFile(inputFile)
8      val words = input.flatMap(line => line.split("␣"))
9      val counts = words.map(word => ("sum", word.toInt))
10       .reduceByKey{case (x, y) => x + y}
11     counts.saveAsTextFile(outputFile)
12   }
13 }
```

2.4 PCT

O *MATLAB* é um exemplo de uma linguagem de programação que contém ferramentas que permitem criar sistemas paralelos. A ferramenta *Parallel Computing Toolbox* permite, através de operações paralelas como ciclos *for* e troca de mensagens, criar sistemas de paralelismo tanto de tarefas como de dados. O interesse no *PCT* para esta dissertação aparece no seu caso de uso que trata de distribuição de *arrays*.

2.4.1 Modelo

Existem duas anotações em *MATLAB* que permitem distribuir um *array* pelas atividades existentes, *distributed* e *codistributed*.

Distributed pode ser usado diretamente no espaço de cliente da aplicação, ou seja, no mesmo local do resto do código da aplicação, porém o programador não tem controlo sobre a forma como o *array* é distribuído. *Distributed* distribui sempre o *array* pela última dimensão, por exemplo, um *array* bidimensional é repartido pelas suas colunas.

A anotação *codistributed* apenas é usada dentro de um ambiente *SPMD* que é declarado em *MATLAB* através das anotações *spmd* e *end*. O *codistributed* já permite ao programador definir a partir de que dimensão deve o *array* ser repartido.

2.4.2 Exemplo

A Listagem 2.6 apresenta um exemplo de multiplicação de matrizes distribuídas em *MATLAB*.

Inicialmente são criadas quatro atividades que funcionarão como trabalhadores disponíveis para a distribuição. São definidas duas matrizes 24*24, *m1* e *m2*, preenchidas com valores aleatórios.

É criado um novo ambiente *SPMD* que é onde irá acontecer a distribuição das matrizes e operações sobre elas. A matriz *m1D*, criada a partir de *m1*, é distribuída pelas suas linhas enquanto que *m2D* pelas suas colunas. Estas distribuições são as ideais para o problema de multiplicação de matrizes pois queremos multiplicar as linhas da primeira matriz pelas colunas da segunda.

Listagem 2.6: Multiplicação de matrizes em MATLAB

```
1  matlabpool open 4
2  m1 = rand(24); m2 = rand(24);
3  spmd
4    m1D = codistributed(m1, codistributor1d(1));
5    m2D = codistributed(m2, codistributor1d(2));
6    res = matmul(m1D,m2D);
7  end
8  matlabpool close
```

2.5 Considerações finais

Atendendo a que o modelo *SOMD* teve a mesma inspiração dos conceitos do modelo *MapReduce*, é interessante fazer-se uma comparação entre ambos. Para além de não existir controlo por parte do utilizador de como são efetuadas as partições dos ficheiros, no *MapReduce* cada tarefa *Map* executa alguns cálculos de modo a retornar um par chave/valor. Isto faz com que o seu âmbito de utilização seja reduzido a problemas em que esta técnica faça sentido. O *SOMD* tem soluções automáticas para lidar com os aspetos de preparação e finalização das atividades paralelas. Para além disso, no *SOMD* é passada para cada tarefa *Map* toda a sub-rotina implementada pelo utilizador.

Apesar desta partilha de ideias com o modelo *MapReduce*, são as linguagens *X10* e *Chapel* que têm uma maior proximidade, ao nível da própria execução da aplicação, com o *SOMD*. Apesar do modelo *APGAS* se destacar por ser totalmente assíncrono, o que não se passa no *SOMD*, esta parecença vem do facto de em ambos os casos se declarar

diretamente as coleções a serem distribuídas e como distribui-las. A grande diferença destas linguagens em relação ao modelo *SOMD* encontra-se nestas próprias declarações. Em *SOMD* o programador usa as coleções sequenciais habituais no resto do programa pois a distribuição só acontece nas sub-rotinas designadas a executar de forma paralela. Nas linguagens apresentadas, a distribuição acontece no momento da criação das coleções o que faz com que estas só possam ser usadas para os pedaços da aplicação que efetivamente se quer que executem no modelo distribuído.

A ferramenta *PCT* do *MATLAB* foi a solução estudada mais fácil de ser usada pelo programador. Isto porque esta ferramenta tem um âmbito de utilização muito mais restrito em comparação com as outras soluções e com o *SOMD*. Para além do *PCT* apenas trabalhar com *arrays* e matrizes, este é especificamente desenhado para funcionar sobre as funções biblioteca da linguagem. Apesar desta facilidade de uso, em relação à própria implementação de aplicações, o *PCT* usa, tal como as restantes soluções estudadas, um esquema programático enquanto que o *SOMD* usa um esquema declarativo.

Akka 3

O *Akka* é uma *framework* implementada através de um conjunto organizado de bibliotecas de código publico que pretendem simplificar a programação de sistemas paralelos. Disponíveis nas linguagens *Scala* e *Java*, estas bibliotecas têm como principal foco facilitar a construção de sistemas escaláveis e resilientes. O *Akka* vem permitir que o programador possa escrever sistemas complexos numa linguagem de alto nível que sejam igualmente eficientes em questões como o comportamento, tolerância a falhas e performance.

O uso do *Akka* passa pela utilização do modelo de atores, que o define. São os atores que fazem uso das bibliotecas *Akka* e que permitem ao programador usá-las de forma consistente. Tal como num modelo orientado a objetos tudo é um objeto, no modelo *Akka* tudo é um ator.

O *Akka* apresenta desde logo uma grande vantagem para se trabalhar com sistemas paralelos: nesta *framework* as comunicações entre atores existentes em diferentes sistemas (máquinas) e entre atores locais são escritas da mesma forma e totalmente implementadas e controladas pelas bibliotecas. Eliminando logo à partida uma potencial fonte de distinção, não existe no *Akka* o conceito de partilha de memória pois cada ator é executado de forma isolada. Contudo, um ator é para todos os efeitos um objeto e o *Scala* não proíbe a partilha de dados entre dois objetos. Se o programador resolver ir contra a filosofia do *Akka* e precisar de partilhar recursos entre dois atores terá de ter especial cuidado.

3.1 Atores

Um ator é uma unidade primitiva de computação, algo que recebe uma mensagem e efetua alguma computação nela baseada.

Podemos dizer que a ideia de ator é similar à de objeto de uma linguagem orientada a objetos, isto é, um objeto recebe uma mensagem (uma chamada de um método) e computa algo dependente da mensagem. A diferença é que os atores funcionam de forma isolada entre si permitindo assim a concorrência desejada. Devido a este isolamento total entre atores, um modelo de atores é sempre assíncrono, ou seja, um ator não fica à espera da resposta de outro. Aqui, as mensagens funcionam de modo reativo, quando são recebidas, são processadas. Dito isto, no caso do *Akka*, o modelo de atores pode ser usado num

contexto híbrido pois, estando implementado em *Java* e *Scala*, uma aplicação pode usar outras bibliotecas que possam permitir partilha de memória.

É importante perceber que apesar de múltiplos atores poderem estar a ser executados ao mesmo tempo, um ator processa uma mensagem de forma sequencial, por exemplo, se enviarmos três mensagens a um ator este vai processar uma de cada vez, se quisermos uma execução concorrente teremos de criar três atores, um por mensagem.

No *Akka*, um ator é definido pelo seu estado, comportamento mediante mensagens, caixa de entrada, filhos e estratégia de tolerância a falhas. Toda esta estrutura é privada e apenas pode ser contactada pelo exterior através de uma classe de referência.

3.1.1 Referência

Um objeto ator é representado por um objeto *Actor Reference* que contém informação sobre o endereço local ou remoto do ator e que pode ser chamado pelo exterior sem qualquer tipo de restrição.

A divisão entre o interior e exterior de um ator permite obter um alto nível de transparência em operações como as de reiniciar um ator, colocar um ator num certo sistema remoto e transmitir mensagens entre atores que façam parte de diferentes aplicações.

3.1.2 Estado

O estado de um ator é definido pelas variáveis que nele constam. Para além das variáveis definidas pelo utilizador, um ator contém outras que dizem respeito ao seu estado relativo ao sistema, por exemplo, a caixa de entrada. Estas variáveis são o que dão valor a um objeto ator e devem ser protegidas de interferência externa.

No *Akka* não é programador que se tem de preocupar com esta questão de proteção de estado. Apesar dos atores usarem *threads* normais para executar o código e vários atores poderem até partilhar a mesma *thread*, o sistema *Akka* trata de toda a sincronização entre execuções.

3.1.3 Caixa de Entrada

O comportamento de um ator é sempre definido pelas mensagens que recebe, isto é, o programador define no ator qual o procedimento a tomar para cada mensagem recebida.

Cada ator tem apenas uma caixa de entrada que trata de organizar e guardar em lista as mensagens recebidas. Normalmente as mensagens são organizadas por ordem de chegada (*FIFO*), mas não nos podemos esquecer que um sistema *Akka* é totalmente assíncrono pelo que a ordem de chegada não é obrigatoriamente igual à ordem de partida, exceto quando as mensagens são enviadas pelo mesmo ator.

É possível definir uma ordem de prioridade em relação aos tipos das mensagens, por exemplo, uma mensagem que corresponda a um erro pode ser considerada prioritária e

no caso de existirem várias mensagens de erro na caixa de entrada estas ficam organizadas pela ordem de chegada.

De modo a que esta metodologia possa ser usada em qualquer sistema é dada a opção ao programador de criar a sua própria caixa de entrada para uma organização personalizada.

3.1.4 Supervisão

Os atores têm sempre um supervisor, quer seja externo ao sistema *Akka* quer seja outro ator. Esta supervisão é fixa e é da competência de quem criou o ator. Cada ator tem guardada a lista dos filhos (atores que criou) que é alterada apenas com operações de criação de novos atores e terminação destes. Estas operações são também elas executadas de forma assíncrona de modo a não bloquear a execução do ator supervisor.

Cabe ao supervisor lidar com as falhas dos filhos. É o supervisor que recebe uma mensagem de erro quando existe uma falha e cabe a ele decidir o procedimento a ser tomado. Por omissão, os atores funcionam de maneira independente neste tipo de situações e o supervisor apenas reinicia o ator em causa. Para sistemas em que os atores dependam uns dos outros é possível definir que, em caso de erro de um ator, o supervisor reinicia todos os filhos, ou apenas os que têm dependências entre si. De referir que a caixa de entrada de um ator não é alterada quando existe uma ordem de reiniciação deste, assim, a melhor metodologia para limpar a caixa de mensagens é terminar por completo o ator e criar uma nova cópia.

3.1.5 Terminação

Exceto ordem em contrário ou falhas, um ator nunca termina. A execução de um ator pode ser vista como um ciclo em que esperamos a receção de uma mensagem, executamos a função correspondente à mensagem e voltamos a esperar por novas mensagens.

Uma atenção a ter com esta metodologia é que as mensagens de terminação entre o supervisor e os filhos não são mensagens normais pelo que não entram na fila da caixa de entrada. Isto faz com que o ator possa terminar sem ter acabado de executar as mensagens que constam na sua caixa de entrada. Para resolver este problema, se a nossa aplicação realmente prevê que seja o supervisor a decidir quando um filho termina, podemos enviar uma mensagens normal que pede ao filho que se termine a si próprio, depois de ter lido todas as mensagens ou quando seja desejável.

3.2 Akka Cluster

Com o modelo de atores é possível desenhar qualquer tipo de sistema paralelo. Não obstante, de forma a facilitar a implementação de alguns dos tipos sistemas usados com maior frequência, no *Akka* estão incluídas algumas bibliotecas que estendem o modelo de atores com alguns conceitos como o de *cluster*.

O *Akka Cluster* permite organizar atores em grupos. Os atores passam a ser definidos por mais um valor indicativo de a que grupo pertencem. Os atores de um grupo comunicam entre si através de um protocolo *gossip*. Cada grupo tem um ator líder que tem as funções de gerir as entradas e saídas do grupo e tratar de falhas dos atores do grupo.

Junto com o *Akka Cluster* aparece também o conceito de distribuição de dados em *Akka*. Trata-se da possibilidade de atores do mesmo grupo poderem trabalhar com os mesmos dados, mais precisamente, uma réplica dos dados existente em cada ator. A replicação funciona sobre um modelo de subscrição em que cada ator é notificado sempre que outro ator do grupo modifica a sua réplica.

O *Akka Cluster* é usado para, por exemplo, organizar um sistema paralelo em que cada máquina contém um grupo de atores.

3.3 Exemplo

A Listagem 3.1 mostra um simples programa de troca de mensagens entre atores. As classes *Ping* e *Pong* representam atores e cada uma delas define o método *receive* que é ativado quando uma nova mensagem chega à caixa de entrada. As mensagens são transmitidas com a anotação *!*, por exemplo *pong ! PingMessage*, onde *PingMessage* é um objeto que fará com que, no ator recetor *Pong*, seja executado o código correspondente à condição *case PingMessage*.

Neste caso são os próprios atores que se terminam a si próprios. O ator *Ping* controla o contador de paragem e, quando este chega ao fim, avisa *Pong* para terminar.

Para trabalharmos com atores remotos, o objeto *Ping*, que é quem começa a execução, tem de saber a porta, o endereço, o nome do sistema e o nome de *Pong*. A Listagem 3.2 mostra como se pode estabelecer a ligação inicial com o ator remoto *Pong*. É usada a função *preStart()* de modo a garantir que os atores se encontram antes da execução principal começar.

Listagem 3.1: PingPong em Akka

```
class Ping(pong: ActorRef) extends Actor {
  var count = 0
  override def receive = {
    case StartMessage =>
      incrementAndPrint
      count+=1
      pong ! PingMessage

    case PongMessage =>
      incrementAndPrint
      if (count > 99) {
        sender ! StopMessage
        println("ping stopped")
        context.stop(self)
      } else {
        sender ! PingMessage
      }
  }
}

class Pong extends Actor {
  override def receive = {
    case PingMessage =>
      sender ! PongMessage

    case StopMessage =>
      println("pong stopped")
      context.stop(self)
  }
}

object PingPong extends App {
  val system = ActorSystem("PingPongSystem")
  val pong = system.actorOf(Props[Pong], name = "pong")
  val ping = system.actorOf(Props(new Ping(pong)), name = "ping")

  ping ! StartMessage
}
```

Listagem 3.2: Pong remoto em Akka

```
override def preStart(): Unit = {
  pong = context.system.actorSelection("akka.tcp://" sistema + "@" +
    ip + ":" + port + "/user/" + name)
  println("Remote found:" + pong)
}
```

Scala/SOMD 4

Este Capítulo para apresenta e discute alguns dos detalhes mais importantes de implementação da solução do problema desta dissertação. Trata-se de uma descrição do produto final desenvolvido a partir de uma implementação já existente do modelo *SOMD* 1.4 em memória partilhada.

Apesar do ênfase principal se basear em computações distribuídas, também foram desenvolvidos alguns melhoramentos à solução de paralelismo local anteriormente desenvolvida.

Ao longo desta descrição serão também discutidas algumas decisões que foram tomadas e as suas implicações no projeto.

4.1 Modelo

O modelo de programação funciona à base de diretivas, em forma de anotações, que permitem ao programador expressar quais os parâmetros de uma qualquer função *Scala* que devem ser distribuídos e qual o algoritmo de redução que deve ser aplicado, se necessário.

As anotações *@DIST* e *@REDUCE* são usadas para expressar o modelo de distribuição a ser aplicado a cada parâmetro e o algoritmo de redução dos valores intermédios. Ao contrário de outros modelos baseados em *MapReduce*, no *SOMD* a partição dos dados de entrada a ser usada é definida por parâmetro. Isto permite a existência de mais uma anotação, o *@COMBINE*. Esta aparece quando existem dois ou mais parâmetros e permite especificar quais os elementos de cada coleção que cada instância da função irá receber.

A Listagem 4.1 mostra um exemplo de utilização do *Scala/SOMD* para uma função que soma os elementos com o mesmo índice de dois *arrays*. No *Scala/SOMD* as anotações *@DIST*, *@REDUCE* e *@COMBINE* têm de ser instanciadas com implementações concretas que podem ou não existir na biblioteca existente. Neste caso, *@DIST* e *@COMBINE* são definidos por classes existentes no sistema enquanto *@REDUCE* é definido pelo utilizador.

As partição usada, *Array1PartitionRef* implementa uma estratégia de partilha dos dados. O combinador *ParallelList* representa uma combinação dos múltiplos *chunks* de cada *array*, dois neste caso, ao nível dos índices dos elementos. A redução usada junta os resultados intermédios num único *array* final.

Listagem 4.1: Soma de dois *arrays* em *Scala/SOMD*

```
1 @COMBINE[ParallelList]
2 @REDUCE( (partials : Array[Array[Double]]) => partials.flatten)
3 def arrayAddition (@DIST[Array1PartitionRef] a: Array[Double],
4        @DIST[Array1PartitionRef] b: Array[Double]): Array[Double] = {
5   val res = new Array[Double](a.length)
6   for (i <- a.indices)
7     res(i) = a(i) + b(i)
8   res
9 }
```

4.2 Descrição semântica

O sistema *Scala/SOMD* envolve três componentes: uma biblioteca de partições e combinações, um sistema de funcionamento em tempo de execução e um *plugin* para o compilador, que trata das anotações.

A biblioteca de partições e combinações foi desenhada de forma a facilitar a sua extensão com novos tipos, que são implementados de forma independente. Torna-se assim uma ferramenta de uso geral pois podem existir todo o tipo de partições e combinações de modo a contemplar todos os casos precisos para diferentes problemas.

No estado atual do sistema *Scala/SOMD* já existem algumas partições disponíveis, nomeadamente para *arrays* de uma e duas dimensões, intervalos (*ranges*) de inteiros e árvores binárias. Para as combinações, existem duas opções disponíveis: *ParallelList*, combinação ao nível dos índices dos elementos dos *arrays*, e *CombinationsPair* que distribui os elementos a partir do cálculo do produto cartesiano de dois *arrays*, ficando cada partição com um elemento de cada *array*. Apesar de não ter sido dada prioridade ao processador de anotações e, atualmente, a tradução das anotações ser ainda feita manualmente, são estes tipos definidos desde logo no sistema que irão permitir que a solução seja usada a partir destas diretivas.

O sistema de *runtime* tem como fundação a classe abstrata *SOMD*. Esta classe está prepara para ser executada com diferentes tipos de trabalhadores: atores *Akka* locais, atores *Akka* remotos e trabalhadores que computam as partições de forma sequencial. O principal comportamento da classe abstrata *SOMD*, ou seja, de todo o processo, baseia-se em quatro passos:

1. Enviar os componentes da partição e a função a executar aos diversos trabalhadores existentes no sistema.

2. Esperar os resultados dos cálculos intermédios feitos pelos trabalhadores e organizar todos os valores.

3. Aplicar à coleção de valores intermédios obtida a função de redução dada pelo utilizador.

4. Retornar o resultado final.

4.3 Partições

A biblioteca de partições do sistema *Scala/SOMD* consiste num conjunto de classes que definem a metodologia a usar para dividir o conjunto inicial de dados a processar em coleções mais pequenas que serão posteriormente distribuídas pelos trabalhadores existentes.

Internamente, para além do conjunto de partições e combinações que é dado a conhecer aos utilizadores, o sistema faz uso de outra biblioteca modular e extensível que trata dos tipos de coleções de dados a serem usados num processo *SOMD*. Com base nas diretivas dadas pelo utilizador, o sistema é capaz de transformar coleções *Scala* (ou *Java*) originais em tipos internos que contêm funções dedicadas a facilitar o processo de partição e redução de dados.

Para cada tipo de dados reconhecido pelo sistema, tem de ser definida uma forma de partição dos mesmos em conjuntos mais pequenos.

Esta divisão interna entre partições e coleções permite separar a lógica de partição de dados, o que facilita a sua implementação. Assim, as partições definem como partir o conjunto de dados recebido e as coleções (os próprios dados) criam conjuntos de dados mais pequenos do mesmo tipo do conjunto original.

A Listagem 4.2 contém uma simplificação da classe abstrata *Array1Partition* que representa uma partição para *Arrays* de uma dimensão. Esta classe faz uso da lógica já existente de partição de *Ranges*.

Listagem 4.2: Array1Partition

```
1  abstract class Array1Partition[T, P] extends Partition {
2    type Chunk = Array1[T]
3    type PartialResult = P
4    def build: List[Chunk] =
5      for (ck <- rangesPartition.chunks)
6        yield {
7          val sel = ck(0)
8          val loc = ck.location.toInt
9          createArray1(a, sel, loc)
10       }
11
12   lazy val chunks: List[Chunk] = build
13
14   lazy val partials: PartialResults = Array1[P](lengths)
15 }
```

O objetivo da uma partição é definir, a partir do argumento inicial recebido, um conjunto de *chunks* mais pequenos a serem enviados para os trabalhadores existentes no sistema, assim como uma coleção *partials* que servirá para guardar o resultado devolvido pelos mesmos de forma a aplicar uma redução final.

4.3.1 Partição automática

Um dos objetivos do sistema *Scala/SOMD* é o de se conseguir, de forma automática, adaptar a *clusters* onde exista heterogeneidade de qualidade das máquinas e mesmo heterogeneidade dos próprios núcleos de cada processador. Para isso, o sistema precisa de alguma forma de conhecer e avaliar os recursos disponíveis.

4.3.1.1 *BogoMips*

A primeira metodologia implementada foi a de usar os valores *BogoMips* de uma máquina (obtidos por cada núcleo) para conseguir fazer uma distinção entre processadores. Deste modo, estaríamos a aproveitar um cálculo já efetuado pelo sistema operativo (*Linux*).

Ora, apesar ser uma solução direta para o problema, esta não é ideal. O valor *BogoMips* refere-se ao número milhões de vezes por segundo em que um núcleo de um processador consegue computar *absolutamente nada*. Como foi possível verificar em testes sobre o sistema *Scala/SOMD* após a implementação do uso do valor *BogoMips* para distribuir os dados pelos núcleos de processador conhecidos pelo sistema, este valor apenas poderia servir como solução para processadores que tenham a mesma arquitetura.

4.3.1.2 *Des3*

Como alternativa ao incompleto *BogoMips*, visto não existir nenhum valor pré-definido de onde se possa deduzir uma comparação direta entre quaisquer núcleos ou processadores, existe na solução final desta dissertação um cálculo manual para o efeito.

O algoritmo *des3*, que executa três vezes uma função de cifragem por cada bloco de dados, foi o escolhido para este efeito. Ao usar este algoritmo de modo a forçar o uso de um só núcleo em cada máquina, os resultados podem ser usados tanto para comparar a qualidade dos processadores entre as máquinas como entre os núcleos de cada processador.

A implementação do algoritmo *des3* no sistema *Scala/SOMD* destina-se apenas a máquinas *Linux*, porém, caso houvesse necessidade, seria relativamente simples abranger outros sistemas operativos, por exemplo, traduzindo a implementação para a linguagem *Scala* usada pelo resto da solução.

4.3.2 Comunicação entre trabalhadores

A forma como está desenhada a biblioteca de partições do sistema *Scala/SOMD* permite que as partições a serem acrescentadas possam definir também a forma como as argumentos são passados para os trabalhadores: por valor (ie. através de cópia) ou por referência (ie. usando partilha de dados).

4.3.2.1 Passagem por valor

Listagem 4.3: Soma de dois *arrays* em *Scala/SOMD* usando passagem por valor

```
1 @COMBINE[ParallelList]
2 @REDUCE( (partials : Array[Array[Double]]) => partials.flatten)
3 def arrayAddition (@DIST[Array1PartitionVal] a: Array[Double],
4     @DIST[Array1PartitionVal] b: Array[Double]): Array[Double] = {
5   val res = new Array[Double](a.length)
6   for (i <- a.indices)
7     res(i) = a(i) + b(i)
8   res
9 }
```

A passagem de argumentos entre atores por valor é o caso base do sistema *Scala/-SOMD*.

Tal como é exemplo a linguagem de programação *Scala* usada neste projeto, o paradigma de programação funcional é cada vez mais usado em sistemas de computação paralela e distribuída pela sua garantia de consistência de dados. Ora, apesar da ferramenta *Scala/SOMD* trabalhar sobre métodos implementados exteriormente por um utilizador, que podem ou não seguir este paradigma, é possível implementar ideias funcionais sobre a comunicação de argumentos entre atores.

Na solução *Scala/SOMD* uma coleção que trabalha com passagem de argumentos por valor é identificada com o sufixo *Val*, por exemplo *Array1Val*. Este tipo de coleções, apesar de poderem ser mutáveis, não são alteradas pelo sistema.

Para argumentos *Val* a partição é implementada seguindo um paradigma funcional onde cada ator irá receber uma nova e mais pequena coleção do mesmo tipo da original. No fim, após o ator mestre receber os resultados de cada trabalhador, estes são agrupados numa outra nova coleção de acordo com o tipo esperado.

Paralelismo distribuído

Para coleções do tipo *Val* é pouca a diferença lógica entre paralelismo local e distribuído. Visto que a partição destas coleções cria outras mais pequenas e independentes entre si, estas podem ser enviadas e processadas em qualquer local sem gerar problemas de memória.

Escrita de resultados

Por uma questão de simplicidade de organização do código e de performance, apesar da implementação do fluxo orientado a computações por passagem de valor partir de ideias funcionais, a escrita de resultados intermédios efetuada por trabalhadores locais é também um processo paralelizado onde cada um dos atores acede diretamente à mesma coleção *partials* em memória.

Isto permite que os resultados não tenham de ser agrupados num só ator mestre antes de poderem ser escritos e que o código possa ser igual ao do fluxo de computação que atua sobre passagem por referência.

4.3.2.2 Passagem por referência

Listagem 4.4: Soma de dois *arrays* em *Scala/SOMD* usando passagem por referência

```
1 @COMBINE[ParallelList]
2 @REDUCE( (partials : Array[Array[Double]]) => partials.flatten)
3 def arrayAddition (@DIST[Array1PartitionRef] a: Array[Double],
4     @DIST[Array1PartitionRef] b: Array[Double]): Array[Double] = {
5   val res = new Array[Double](a.length)
6   for (i <- a.indices)
7     res(i) = a(i) + b(i)
8   res
9 }
```

Identificadas pelo sufixo *Ref*, as coleções implementadas na solução *Scala/SOMD* podem também seguir um modelo de passagem de argumentos entre atores por referência.

A possibilidade de passar argumentos entre trabalhadores por referência permite obter ganhos de performance em muitos exemplos de computações. Tal como existem problemas onde uma implementação imperativa obtém grandes ganhos em relação a uma funcional, o mesmo acontece na ferramenta *Scala/SOMD*.

A partição de argumentos *Ref* usa apenas índices da própria coleção inicial para transmitir aos trabalhadores qual a sua área de trabalho. Aqui, o sistema não gasta tempo e memória em criar novas coleções mais pequenas pois permite que vários atores possam aceder à coleção original dentro dos limites criados para cada um.

Outra vantagem de se poder implementar coleções deste tipo é a de se poder oferecer ao utilizador a possibilidade de, na ferramenta *Scala/SOMD*, usar um método imperativo na mesma forma que o faria numa solução sequencial normal, onde o argumento recebido é mutável e alterado pela função com os resultados.

Paralelismo distribuído

Apesar de se perder a vantagem inicial de performance obtida por uma metodologia de passagem de argumentos por referência, na solução *Scala/SOMD* é também possível usar coleções *Ref* para computações distribuídas.

Para este tipo de computações, os trabalhadores usam localmente a mesma metodologia de passagem de argumentos por referência entre atores locais e, no final, o utilizador consegue obter o mesmo resultado esperado onde a solução da computação pode ser escrita diretamente na coleção inicial recebida.

A ferramenta *Scala/SOMD* está preparada para ser usada com coleções de grande dimensão, sendo que nunca é desejado transmitir remotamente os argumentos iniciais na sua integridade. Por outro lado, uma coleção *Ref* é dividida entre atores apenas pelos seus índices o que faz com que esta não seja efetivamente partida em coleções mais pequenas. Assim, para computações remotas, uma coleção *Ref* tem de ser transformada em *Val* de acordo com os diferentes índices obtidos na partição. Esta transformação cria novas coleções *RefRemote* com o tamanho exato que cada trabalhador remoto precisa para fazer a sua parte das computações.

A classe *RefRemote* é uma subclasse de *Ref* que guarda no seu interior uma variável de instância do tipo *Val*. Fornece uma representação alternativa duma coleção *Ref*, preparada para transmissão e receção remotas.

A solução *Scala/SOMD* está assim preparada para que possa ser acrescentada lógica extra às coleções *Ref* de modo a que estas possam ser transmitidas entre trabalhadores remotos. Para isto, deve ser criada uma coleção *RefRemote*, que, a partir do argumento *Ref* original, é capaz de gerar um *Val* correspondente. Para além disso, como a ferramenta *Scala/SOMD* trabalha com comunicações assíncronas, existe uma *cache* já definida capaz de guardar o endereço da coleção *Ref* original para que, após a processo remoto estar concluído, se possa voltar à metodologia de passagem por referência original.

Um argumento estar identificado como sendo do tipo *RefRemote* é também uma indicação que cada trabalhador remoto deve efetuar os cálculos locais seguindo a metodologia normal de um processamento com passagem por referência *Ref*. O resultado da computação terá novamente de ser enviado usando as propriedades do *RefRemote*.

Tal como detalhado na Secção 4.4, todo este processo é gerido pelas próprias coleções de modo a que se possa manter uma solução modular e extensível sem ter de alterar a lógica do fluxo principal de computação.

4.4 Implementação modular recursiva

De modo a manter as propriedades de modularidade e extensibilidade da ferramenta *Scala/SOMD*, a biblioteca de coleções e partições tem interfaces base que permitem que cada novo tipo de dados acrescentado ao sistema possa ter a sua própria lógica para funcionalidades como a sua transmissão entre trabalhadores ou a forma como este pode ser clonado. Isto também permite que, ao acrescentar novas coleções e partições, estas possam implementar apenas os métodos que sejam necessárias para os seus casos de uso.

A Listagem 4.5 contém a interface sobre a qual todas as coleções da solução são implementadas.

Listagem 4.5: Interface *ChunkK* para coleções

```
trait ChunkK extends Serializable {
  val location: Location
  override def toString: String =
    getClass.getName + "{" + toStringContents + "/" + location.toString + "}"
  def toStringContents: String
  def prepareToSend(): Unit = {}
  def makeNet: ChunkK = this
  def copyBack(): Unit = {}
  def getRef: List[ChunkK] = List()

}
```

Por exemplo, os métodos *makeNet*, *prepareToSend* e *copyBack* são todos de implementação obrigatória quando se pretende que uma coleção *Ref* possa ser processada num modelo distribuído:

makeNet Permite que as coleções *Ref* se possam transformar nas suas extensões *RefRemote*.

prepareToSend É utilizado pela maior parte das coleções pois permite que estas possam apagar alguns dados que já não sejam necessários, tornando assim as mensagens remotas mais pequenas. Para a funcionalidade de usar *Ref* num modelo distribuído, este método é também usado para guardar em *cache* a coleção original que queremos manter até ao final da computação.

getRef Usado nos trabalhadores remotos para obter o argumento real com que irão trabalhar. Normalmente serve para que quando a coleção é do tipo *RefRemote*, esta possa ser lidada como um *Ref* normal quando o sistema está a processar uma computação local.

copyBack Permite que se possa voltar a obter as coleções originais *Ref* já atualizadas com os resultados remotos obtidos.

Outra propriedade deste tipo da abordagem é que, naturalmente, cada um destes métodos é chamado em cada um dos níveis que possam existir numa coleção *Scala/SOMD* de forma recursiva. Por exemplo, o argumento sobre o qual a ferramenta está a trabalhar pode ser uma *ListK* de *Array1* onde o método *prepareToSend* atua tanto sobre a lista como para cada um dos elementos da lista.

Este tipo de design permite que o sistema base possa juntar todos os possíveis casos de coleções numa só implementação, onde cada um destes métodos será sempre chamado e cabe às próprias coleções saber como lidar com o seu caso especifico.

4.5 Tradução

Analisemos o processo de tradução efetuado pelo *Scala/SOMD* para o exemplo da Listagem 4.1. A Listagem 4.6 mostra o aspeto final da tradução.

O primeiro passo da tradução da função consiste em extrair alguns elementos essenciais provenientes das definições dadas na função original. A Tabela 4.1 mostra o tipo calculado de cada elemento e de onde provém a informação.

Elemento traduzido	**Origem da informação**
InputData0 = Array[Double]	tipo de a
InputData1 = Array[Double]	tipo de b
Partition=ParallelPair[Array1PartitionRef , Array1PartitionRef]	determinado a partir das anotações
Chunk = ListK[Array1[Double]]	determinado a partir da partição
PartialResult = Array[Double]	tipo do retorno da função original
PartialResults = Array1[Array[Double]]	D[PartialResult]. D depende de Partition
FinalResult = Array[Double]	tipo de retorno do método de redução

Tabela 4.1: Tradução de elementos pelo *Scala/SOMD*

O segundo passo trata de converter a função original numa função *Scala* não anotada. As versão da função traduzida acaba por ficar parecida com a original com algumas exceções como o nome e o tipo dos parâmetros.

Por último é gerada a nova função *translation*. Os parâmetros mantêm-se iguais aos originais. O corpo da função define uma instância da classe *SOMD*. O método de redução *reduce* é praticamente copiado do original definido na anotação *@REDUCE*. O resto é gerado usando os elementos obtidos no primeiro passo.

Listagem 4.6: Tradução da soma de dois *arrays* efetuada pelo *scala/SOMD*

```
1  abstract class SOMD {
2
3    type Chunk = ListK[Array1[Double]]
4    type PartialResult = Array[Double]
5    type PartialResults = Array1[Array[Double]]
6    type FinalResult = Array[Double]
7
8    def process(chunk: Chunk): PartialResult = {
9      val (ckA, ckB) = (chunk(0), chunk(1))
10     val res = new Array[Double](ckA.length)
11     for( i <- ckA.indices )
12       res(i) = ckA(i) + ckB(i)
13     res
14   }
15
16   def reduce(partialResults: PartialResults): FinalResult =
```

```
17      Array1.flatten(partialResults)
18
19    def getSubPartition(chunk: Chunk): ParallelList[Array1[Int], Array[Int]] =
20      ParallelList[Array1[Int], Array[Int]](
21        Array1PartitionVal(chunk(0), 2, parts),
22        Array1PartitionVal(chunk(1), 2, parts)
23      )
24  }
```

Internamente, uma partição é representada por uma lista de componentes do tipo *Chunk*. Cada *chunk* guarda as coordenadas de localização na coleção original.

O paralelismo acontece na nova função *process*, que é executada uma vez em cada trabalhador (quatro neste caso). Cada uma das execuções recebe uma lista com dois *arrays* correspondentes aos *chunks* definidos para serem tratados por cada trabalhador. Os resultados de cada *process* são guardados num tipo *PartialResult* e mais tarde são todos agrupados numa coleção do tipo *PartialResults = Array1[Array[Double]]*.

A função *reduce*, que foi definida pelo programador, é executada apenas uma vez e recebe todos os resultados parciais obtidos.

4.6 Arquitetura de atores

À possível complexidade inicial existente no sistema para permitir partições automáticas é acrescida toda a lógica de procura de máquinas vizinhas na rede. Por esta razão, a ferramenta *Scala/SOMD* foi desenhada de modo a ser um sistema que está em constante processamento em *background*.

A solução é lançada em todas as máquinas que se pretende usar e, quando não está a executar um processamento *SOMD*, está a gerir a rede e outras propriedades. Deste modo, quando o cliente invoca um processo, a ferramenta pode executá-lo imediatamente.

4.6.1 Ator mestre

Numa execução *SOMD* existe sempre um ator principal, lançado em todas as máquinas, que efetua a tradução do código original para um que seja suscetível de uma computação *SOMD* e que calcula, após os passos paralelos estarem concluídos, o resultado final.

Ainda assim, em termos conceptuais, a arquitetura de atores remotos usada na implementação final não segue um modelo de cliente-servidor. Não existe nenhuma diferença entre atores lançados em diferentes máquinas, ou seja, qualquer um destes trabalhadores pode iniciar uma execução *SOMD*, tanto num modelo distribuído como local.

Toda a solução passa por mensagens assíncronas entre atores. Assim, podem até existir diferentes execuções, lançadas na mesma máquina ou em máquinas distintas, distribuídas ou locais, a serem independentemente processadas em paralelo por todo o sistema.

4.6.2 Paralelismo local

No fluxo de paralelismo local, a primeira responsabilidade do ator mestre passa por perceber o número de núcleos existentes no processador da máquina onde está a ser executado.

Este valor é importante para o outro lado da solução, as partições. Depois do ator mestre receber os dados (e o seu tipo) a serem usados, estas coleções passam por um processo de partição de acordo com a capacidade da máquina.

Após os cálculos da partição, o ator mestre acaba com uma lista de coleções de dados mais pequenas (do mesmo tipo das originais). A partir do tamanho desta lista, o ator mestre consegue criar o número necessário de atores filho.

A partir da lista de sub-coleções é também possível criar, desde já, uma lista vazia de resultados parciais. Esta coleção é guardada em memória partilhada e é onde os atores filhos irão escrever os resultados calculados.

No momento da criação de um ator filho, é-lhe imediatamente enviado, pelo ator mestre, a sub-coleção de dados com que o ator filho irá trabalhar, assim como a função que a solução está a processar.

Após o ator aplicar a função traduzida ao seu subconjunto de dados, este escreve os resultados diretamente na partição. A localização (índice) para esta escrita é também recebida como argumento. Ainda assim, o ator filho devolve uma mensagem vazia ao ator mestre que serve como forma de notificar que o processo terminou.

Por último, após todos os atores filhos terem terminado a sua parte, o ator mestre aplica a redução definida nas anotações à lista de resultados parciais, obtendo assim o valor final desejado.

4.6.3 Paralelismo distribuído

O fluxo distribuído de uma execução usa a totalidade da solução encontrada para paralelismo local. Desde logo, os atores descritos até agora, mestre e filho, são os únicos tipos de atores existentes na solução.

O paralelismo distribuído acrescenta alguma responsabilidade ao ator mestre. Aqui, para além de controlar o fluxo local, este ator tem o trabalho de, assíncrona e periodicamente, procurar outros atores iguais existentes na rede local usada pela máquina. Uma máquina participa numa computação se nela existir um ator *SOMD* ativo.

Neste caso, o valor indicativo do número de sub-coleções enviado à partição é o número de núcleos mas sim o número de máquinas vizinhas conhecidas na rede.

Após a partição estar concluída de acordo com a capacidade de cada máquina, tal como no modelo de paralelismo local, o ator mestre local envia aos atores vizinhos os subconjuntos de dados correspondentes, porém, neste caso, é necessário enviar não só a função a ser aplicada aos dados mas também o resto da tradução inicialmente calculada.

Agora que todos os atores mestre têm acesso a toda a informação da execução, estes podem iniciar o processo de paralelismo local com memória partilhada. No final, após

reduzir os dados a um único resultado, este é enviado para a máquina onde se iniciou todo o processo.

Quando o ator mestre local recebe todos os resultados de todas as máquinas, este executa o redução final, obtendo assim o valor final desejado.

4.7 Paralelismo embaraçoso

Apesar de, felizmente, muitas das situações que surgem na programação permitirem uma aplicação automática de um modelo de divisão-e-conquista como o *SOMD*, existem alguns casos em que tal não é possível.

A Listagem 4.7 mostra duas funções que são exemplos de casos onde não é possível a aplicação do modelo *SOMD*. Ambos os casos dizem respeito à necessidade de utilização do índice para os cálculos desejados, no primeiro na própria soma e no segundo para uma condição. O que acontece é que, depois da distribuição, um processo não tem acesso ao índice original de um elemento na coleção.

Listagem 4.7: Ausência de paralelismo embaraçoso

```
1  def sum2(a: Array[Int]): Int = {
2    var res = 0
3    for( i <- a.indices )
4      res += i * a(i)
5    res
6  }
7
8  def sum3(a: Array[Int]): Int = {
9    var res = 0
10   for( i <- a.indices)
11     if( i % 2 == 0 )
12     res += a(i)
13   res
14 }
```

Avaliação 5

A solução foi avaliada no *cluster* do Departamento de Informática da FCT-UNL. Este *cluster* é ideal para testar estes tipos de sistemas pois é composto por treze (mais um que serve como *front-end*) nós com processadores de seis arquiteturas diferentes.

Desde já é importante referir que existe uma opção de execução do programa que não será explicitamente referenciada pelos testes. Trata-se de qual o nó escolhido para ser o nó-mestre (repartidor de trabalho). A escolha desta máquina não é interessante de ser mencionada pois, em qualquer um dos casos, a escolha do nó-mestre não afeta os resultados.

5.1 Eficiência e Escalabilidade

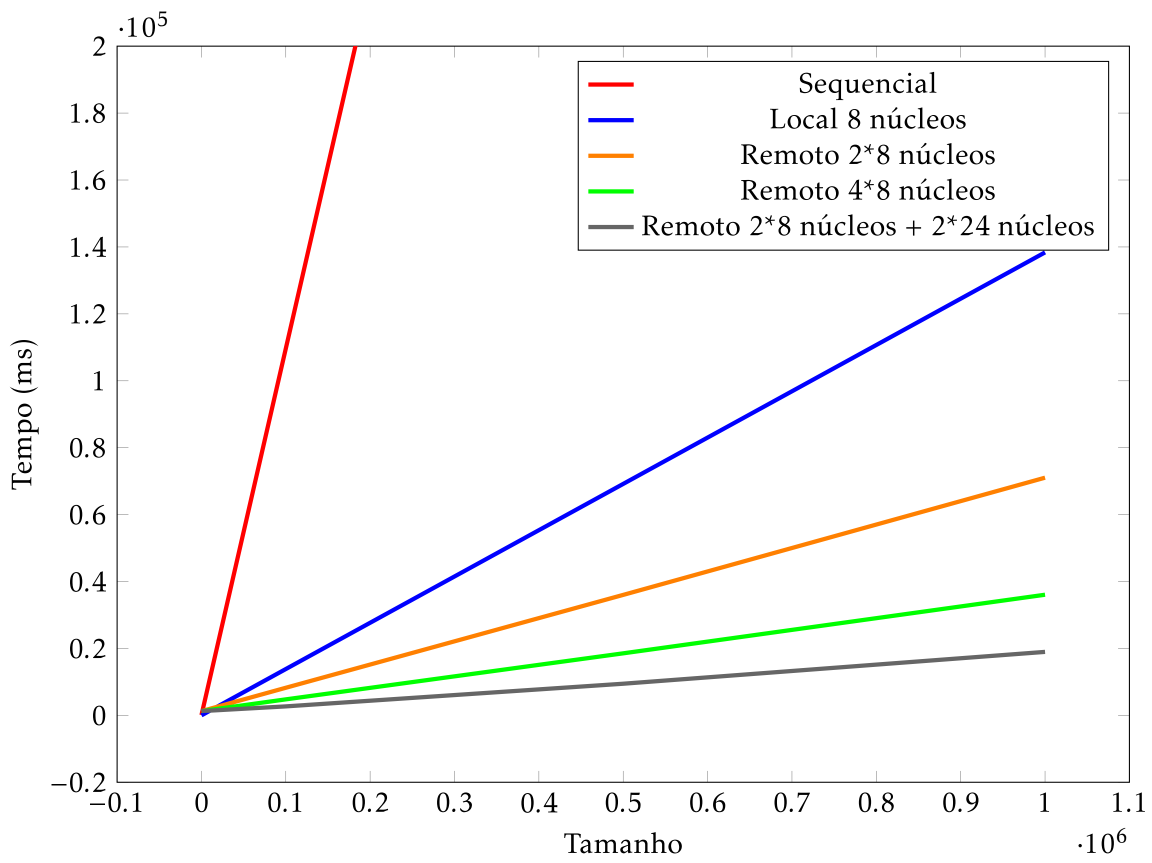
De modo a testar a eficiência e a escalabilidade da solução *Scala/SOMD* foi criado um teste artificial que permite simular vários tamanhos de argumentos recebidos e diferentes tempos de computação por argumento.

O Gráfico 5.1 mostra a relação de tempos de execução entre diferentes ambientes possíveis no contexto do *cluster* disponível, desde a execução sequencial à utilização de várias máquinas com diferentes características, para um problema onde o tempo artificial de computação por cada argumento de uma coleção é de 500 nanossegundos.

O cálculo da eficiência entre os os vários ambientes de *cluster* em relação à execução sequencial ajuda a perceber com exatidão a eficiência da solução:

$$\text{Eficiência} = \frac{\text{Tempo de execução sequencial}}{\text{N}^{\text{o}}\text{ de processadores} \times \text{Tempo de execução paralela}} = \frac{\text{Speedup}}{\text{N}^{\text{o}}\text{ de processadores}}$$

A tabela 5.1 mostra os valores calculados para o *speedup* e eficiência de cada um dos ambientes de execução. Para este cálculo foram apenas considerados tamanhos de problemas suficientemente grandes para que uma execução remota já não seja afetada por *overheads* de comunicação. Os resultados são os esperados. Os valores são muito altos mas é possível observar que se vão afastando das referências ideais conforme se aumenta a qualidade dos recursos disponíveis. No último caso, onde são introduzidos dois processadores em média 4 vezes superiores aos usados inicialmente é onde se observa a



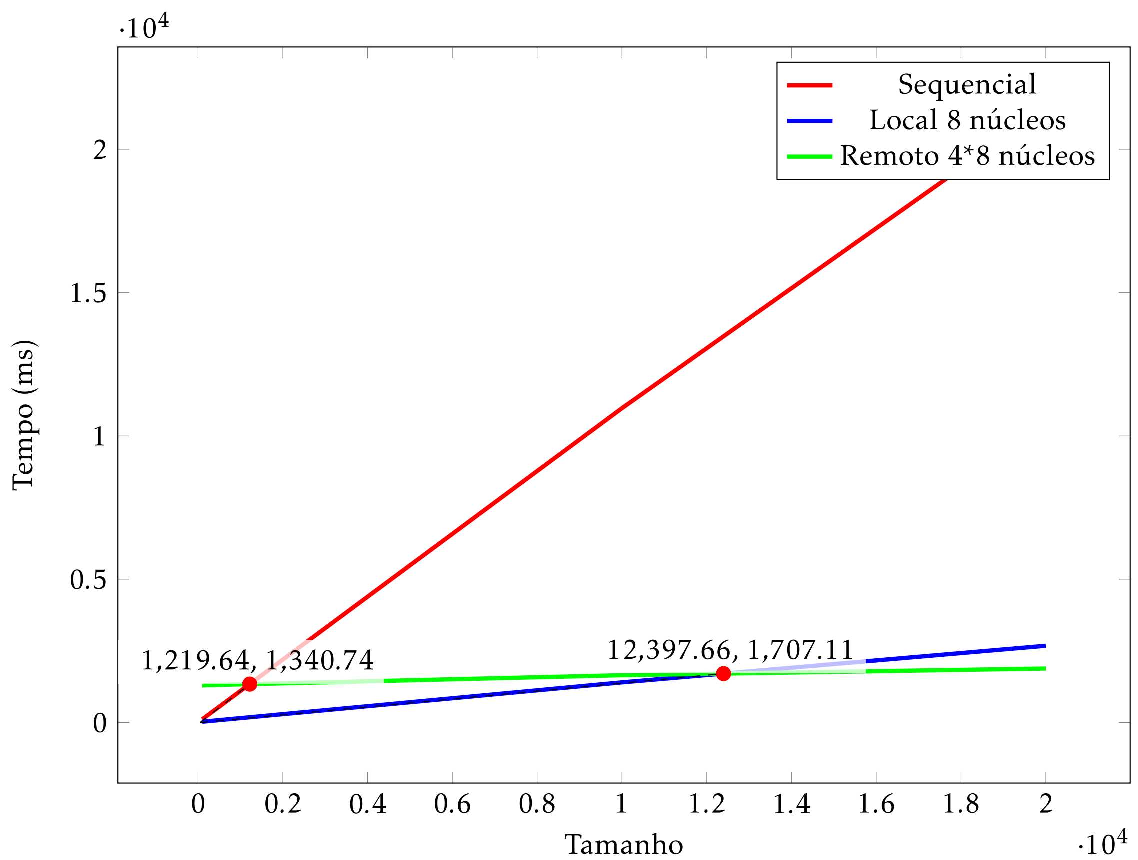


Figura 5.1: Testes artificiais

maior descida de eficiência, ainda que com tempos de execução mais baixos que os outros ambientes. É importante referir que estamos a avaliar um caso de uso artificial que reflete uma computação otimizada para computações paralelas.

Computação	***Speedup*** (ideal teórico)	**Eficiência**
Local 8 núcleos	7.92 (8)	0.99
Remoto 2*8 núcleos	15.42 (16)	0.96
Remoto 4*8 núcleos	30.25 (32)	0.94
Remoto 2*8 + 2*24 núcleos	57.75 (64)	0.90

Tabela 5.1: Comparação de *speedups* e eficiências entre ambientes de teste

O segundo Gráfico da Figura 5.1 trata-se de uma ampliação dos mesmos resultados numa escala mais curta onde é possível verificar os pontos aproximados onde se ganha vantagem em passar para uma execução remota.

5.2 *Speedup*

Para se conhecer os *speedups* para um problema real utilizando a ferramenta *Scala/SOMD*, é interessante não só que o valor seja calculado a partir de uma solução sequencial na linguagem *Scala* pura (ao invés de usar o modo sequencial da solução), mas também que se possa comparar o resultado com as ferramentas que esta linguagem já oferece sobre coleções paralelas.

A listagem 5.2 compara um procedimento de adição de dois *Arrays* índice a índice, resultando num *Array* final. Para além de ser um problema que, dependendo do tamanho dos argumentos, pode ganhar ao ser paralelizado, este também consegue tirar partido de ferramentas que permitam o uso de memória partilhada. Ora, os resultados não corresponderam na totalidade ao esperado.

A solução *SOMD* conseguiu obter sempre melhor performance do que o código sequencial escrito em *Scala*. Apesar de ser um ponto muito positivo, é interessante o facto de que os *overheads* referentes à partição de coleções não sejam suficientemente grandes para a ferramenta ter algum tipo de perda, mesmo para argumentos mais pequenos.

As coleções paralelas do *Scala* não conseguiram obter resultados convincentes a nível de performance. Para além disso, ainda que para desenhar o Gráfico se tenha sempre trabalhado com os melhores valores encontrados, estas coleções demonstraram ser muito inconstantes nos tempos de computação obtidos, muitas vezes tornando-se até mais lentas que a contraparte sequencial, mesmo para problemas com argumentos na ordem de milhões de dados.

A solução *Scala/SOMD* obtém assim um *speedup* médio de ***2.35*** enquanto que as coleções paralelas um de ***1.06***.

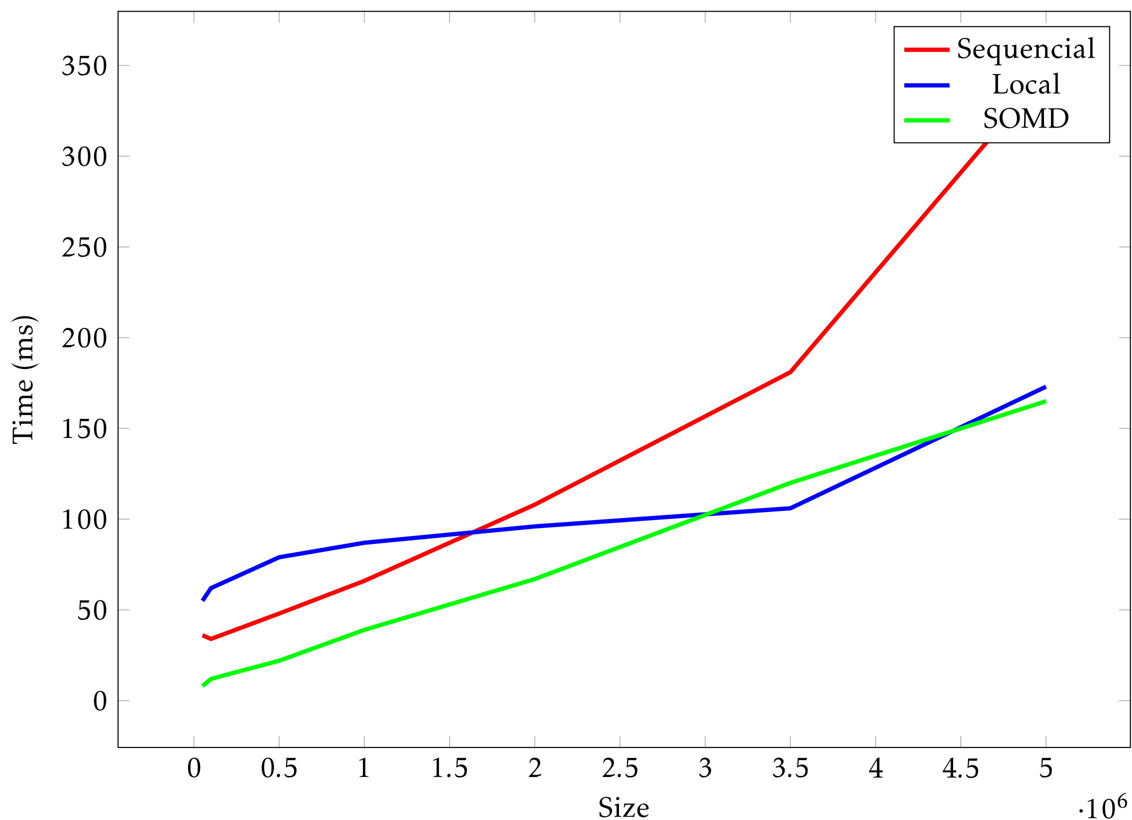


Figura 5.2: SOMD vs coleções paralelas do *Scala*

5.3 Facilidade de programação

O principal objetivo desta dissertação passa pela implementação de uma ferramenta capaz de, a partir de algumas diretivas, executar uma função escrita de forma sequencial segundo um modelo paralelo e distribuído. Desta forma, a solução permite que um programador possa fazer uso de todos os recursos que tem a seu dispor sem precisar de estudar uma nova *framework* ou linguagem de programação.

Este subcapítulo compara a facilidade de escrita de um processo usando a solução *Scala/SOMD* em relação à sua implementação sequencial e a outras ferramentas existentes.

A listagem 5.1 apresenta um simples exemplo de uma adição de dois *Arrays* na linguagem *Scala*.

Listagem 5.1: Soma de dois *arrays* em *Scala*

```
1 def arrayAddition(a: Array[Double], b: Array[Double]): Array[Double] = {
2   val res = new Array[Double](a.length)
3   for ( i <- a.indices )
4     res(i) = a(i) + b(i);
5   res
6 }
```

Sendo que um dos objetivos da ferramenta *Scala/SOMD* é o de permitir executar problemas paralelos e distribuídos a partir de código sequencial, para se transformar o problema de adição de *Arrays Scala* num processo *SOMD*, apenas se precisa de acrescentar algumas diretivas, demonstradas na listagem 5.2.

Listagem 5.2: Soma de dois *arrays* em *Scala/SOMD*

```
@COMBINE[ParallelList]
@REDUCE( (partials: Array1[Array[Double]]) => partials.flatten )
def arrayAddition (@DIST[Array1PartitionRef] a : Array[Double],
    @DIST[Array1PartitionRef] b : Array [Double]) : Array[Double] = {
  val res = new Array[Double](a.length)
  for(i <- a.indices)
    res(i) = a(i) + b(i)
  res
}
```

Apesar do facto de um utilizador da ferramenta *Scala/SOMD* apenas precisar de conhecer a biblioteca de partições e combinações, a própria linguagem *Scala* oferece algumas ferramentas que permitem exprimir problemas como o de adicionar *Arrays* por índice numa forma mais simples, demonstrada na listagem 5.3. A coleção *ParArray* faz uso de memória partilhada e de *threads* para percorrer coleções de forma paralela de acordo com o número de núcleos de uma máquina.

Listagem 5.3: Soma de dois *arrays* paralelos *Scala*

```
def arrayAddition(a: Array[Double], b: Array[Double]): Array[Double] = {
  val res = new ParArray[Double](a.length)
  val pA: ParArray[Double] = a.par
  val pB: ParArray[Double] = b.par
  for ( i <- pA.indices )
    res(i) = pA(i) + pB(i)
  res.toArray
}
```

A vantagem da solução *Scala/SOMD* é a sua versatilidade, modularidade e extensibilidade. Por exemplo, a listagem 5.4 descreve uma utilização da ferramenta para um problema de soma dos valores de uma árvore binária num modelo recursivo.

Outras ferramentas mais complexas como as estudadas no capitulo 2 também têm como objetivo a versatilidade de problemas possíveis de resolver. Porém, para obter esta caraterística, estas soluções passam por extensões a linguagens existentes ou mesmo linguagens criadas de raiz.

Listagem 5.4: Soma dos elementos de uma árvore binária em *Scala/SOMD*

```
1 @REDUCE( (partials: Array1[Int]) => partials.sum )
2 def binTreeAddition (@DIST[BinTreePartition] t: BinTree[Int]): Int =
3   if( t.isEmpty ) 0
4   else t.value + binTreeAddition(t.left) + binTreeAddition(t.right)
```

A listagem 5.5 mostra o exemplo de soma de dois *Arrays* usando *Spark*. Aqui o utilizador controla todo o processo de forma manual e a baixo nível. Por exemplo, os *Arrays Java* têm de ser transformados em coleções específicas *Spark*, o resultado tem de ser preparado para se poder, manualmente, ordenar os índices recebidos de acordo com os originais e, para se obter como resultado um *Array Java*, tem de se transformar novamente a coleção.

Listagem 5.5: Soma de dois *arrays* em *Spark*

```
1  private static JavaPairRDD<Long, Integer> arrayAdd(JavaRDD<Integer> array1,
2      JavaRDD<Integer> array2) {
3    JavaPairRDD<Long, Integer> a1 = array1.zipWithIndex()
4      .mapToPair(Tuple2::swap);
5    JavaPairRDD<Long, Integer> a2 = array2.zipWithIndex()
6      .mapToPair(Tuple2::swap);
7
8    return a1.join(a2).mapToPair(e -> new Tuple2<>(e._1, e._2._1 + e._2._2));
9  }
10
11 public Integer[] addArray(Integer[] a, Integer[] b){
12   JavaPairRDD<Long, Integer> result = arrayAdd(
13     sc.parallelize(Arrays.asList(a1)),
14     sc.parallelize(Arrays.asList(a2))
15   );
16   List<Tuple2<Long, Integer>> finalList = result.sortByKey().collect();
17
18   Integer[] array = new Integer[finalList.size()];
19   for(int i=0; i < l.size(); i++){
20     array[i] = l.get(i)._2;
21   }
22   return array;
23 }
```

5.3.1 Nota

A biblioteca de partições e coleções e a de combinações existentes na solução *Scala/SOMD* tratam-se, pelo menos numa versão inicial, de um aglomerado das metodologias mais usadas em processos de computação paralela para distribuir coleções e combinar as suas partições. Assim, para além do utilizador não ter de estar constantemente a reescrever estas fórmulas, apenas as terá de conhecer por nome.

Listagem 5.6: Soma de dois *arrays* em *Scala/SOMD*

```
@COMBINE[ParallelList]
@REDUCE(ArrayFlatten)
def arrayAddition (@DIST[Array1PartitionRef] a : Array[Double],
    @DIST[Array1PartitionRef] b : Array [Double]) : Array[Double] = {
  val res = new Array[Double](a.length)
  for(i <- a.indices)
    res(i) = a(i) + b(i)
  res
}
```

Listagem 5.7: Soma dos elementos de uma àrvore binária em *Scala/SOMD*

```
@REDUCE(ArraySum)
def binTreeAddition (@DIST[BinTreePartition] t: BinTree[Int]): Int =
  if( t.isEmpty ) 0
  else t.value + process(t.left) + process(t.right)
```

Neste sentido, apesar de não existir nenhum exemplo implementado, seria natural a criação de uma biblioteca adicional para reduções. As listagens 5.6 e 5.7 mostram qual seria o aspeto final utilizando uma biblioteca deste tipo.

Sendo que, tal como são exemplos estes dois casos, na linguagem *Scala* as diferentes coleções partilham muitos métodos entre si, poderia-se ainda generalizar as reduções descritas para *Flatten* e *Sum*.

Conclusões e trabalho futuro 6

Este Capítulo descreve alguns pontos importantes de serem lembrados para uma possível continuação de desenvolvimento da aplicação, tanto a nível de tecnologias usadas, como a nível da própria linguagem.

6.1 Cache

Apesar de não ter sido explicitamente referida nenhuma metodologia de *caching* ao longo do documento, foram pensadas algumas formas de como se poderia aproveitar uma *cache* na solução *SOMD*.

Numa primeira tentativa de uso prático de *caching* de resultados finais de um processo, foi pensado que se poderia guardar, para todos os casos, o último resultado de uma computação *SOMD*, assim como a sua classe tradução. Deste modo, no começo de uma computação, o ator mestre verificaria se a classe tradução *SOMD* recebida é igual à da última execução de maneira a perceber se já conhece o resultado final, guardado no objeto *Cache*.

O problema desta abordagem é que todo o processo de execução *SOMD* trabalha com tipos explícitos, isto é, por exemplo, o tipo do resultado final é conhecido e único, não podendo este ser representado por *Any*. Para os casos em que um resultado final é um *ChunkK*, por exemplo, uma multiplicação de *Array2*, a *cache* funciona e devolve os resultados esperados. Porém, esta *cache* não está preparada para trabalhar com tipos base, como um inteiro ou uma *String*.

Uma solução para este problema seria o projeto implementar os seus próprios tipos base, por exemplo, existir uma classe *Int* interna que estenderia *ChunkK*. Ainda assim, esta é uma forma algo forçada de resolver o problema sendo que, provavelmente, existem melhores alternativas.

6.2 Serializador

O serializador de mensagens usado atualmente pela solução, o *Protocol Buffer* [17] é simples e eficaz pois trabalha com *arrays* de *bytes*, o que permite obter uma grande liberdade do tipo de mensagens a usar.

Ainda assim, acontece que no processo de serialização é efetuada uma cópia, para memória, de todos os dados traduzidos em *bytes*. Isto torna-se um problema pois é esperado que a ferramenta *Scala/SOMD* seja usada para tratamento de grandes quantidades de informação.

Para grandes tamanhos de mensagens, seria interessante que a solução aceitasse enviá-las por *Streams*. A biblioteca *Akka*, já usada em grande parte da solução, contém uma funcionalidade *Akka Streams* que facilita a introdução de *Streams* nos seus atores.

6.3 Atores Akka

Cada linguagem tem o seu mundo, sem o qual esta perde qualquer tipo de importância na informática. O mundo de uma linguagem baseia-se nos seus utilizadores, nos seus contribuidores e nas suas bibliotecas. Ora, a biblioteca e *framework Akka*, em todos os seus aspetos mas mais precisamente no seu caso base (*Akka Actors*), é um tema algo controverso no mundo da linguagem *Scala*. Este Capítulo descreve alguns pontos desta controvérsia e, no caso da solução desta dissertação, destina-se apenas aos atores *Akka* usados para computações paralelas locais, pois os atores remotos executam operações extra para além das de processamento de dados.

Na documentação da biblioteca *Akka* não existe uma descrição explicita do que é um ator. Ainda assim, é possível ler a descrição de como os definir.

> *Actors are implemented by extending the Actor base trait and implementing the receive method. The receive method should define a series of case statements (which has the type PartialFunction[Any, Unit]) that defines which messages your Actor can handle, using standard Scala pattern matching, along with the implementation of how the messages should be processed.*
>
> `https://doc.akka.io/docs/akka/2.5/actors.html`

Ora, a linguagem *Scala* existe como uma forma de, na *JVM*, permitir implementar programas com tipos definidos e únicos de uma maneira mais facilitada. A acrescentar a este objetivo, grande parte dos utilizadores da linguagem usam-na como forma de implementar soluções segundo um paradigma funcional e grande parte dos contribuidores tenta trazer funcionalidades da linguagem puramente funcional *Haskell* para a *JVM*.

A citação acima, proveniente da documentação da biblioteca *Akka*, explica que um ator trata-se de um aglomerado de funções parciais de *Any* para *Unit* (*PartialFunction[Any, Unit]*). Esta está longe de ser uma metodologia que agrade aos programadores que tentam

usar ao limite o paradigma funcional. Para além dos tipos de entrada e saída abstratos, estas funções são parciais, o que indica que podem ou não executar operações com efeitos secundários.

Numa tentativa de melhorar o tipo desta função para um que se adeque ao nosso problema, podemos começar por relembrar que a solução só trabalha com dados de tipo *ChunkK* e que executa, sobre estes dados, funções *process* (também recebidas como argumento) que devolvem o tipo *PartialResult*. Assim, poderíamos descrever um ator *SOMD* como uma simples função:

PartialFunction[(ChunkK, Partition, ChunkK => PartialResult), Unit].

O tipo de resultado do ator mantêm-se *Unit* pois, para além de executar um processo sobre os dados recebidos, queremos que este escreva o resultado diretamente na partição de modo a que esta escrita seja feita em paralelo pelos múltiplos atores, usando memória partilhada. Ainda assim, esta é uma versão simplificada para demonstração, por exemplo, se for esperado que possam existir erros, para estes poderem ser tratados, o ator deve devolver o tipo *Either[Error, Unit]*.

Relembrando o que define uma função puramente funcional:

1. Total. Para cada valor de argumentos, retorna sempre um valor.

2. Determinística. Para o mesmo argumento, retorna o mesmo valor.

3. Pura. O único efeito secundário é o cálculo do valor de retorno.

Não podemos afirmar que a solução acima seja uma função puramente funcional, porém, não está longe de o ser. Se não estivéssemos a trabalhar com partilha de memória, a função devolveria o mesmo *PartialResult* para os mesmos *ChunkK* e método sobre *ChunkKs* recebidos. Neste caso, poderíamos mudar a terminologia da função de *PartialFunction* para apenas *Function*.

A Listagem 6.1 mostra uma implementação de exemplo do que poderia ser a forma do *SOMD* executar um problema em paralelismo local.

Listagem 6.1: somdActor

```
1 def somdActor(chunk: ChunkK, partition: Partition, process: ChunkK =>
    PartialResult): Future[Unit] =
2   Future.successful {
3     partition.partials(chunk.location) = process(chunk)
4   }
```

Em *Scala*, o tipo *Future* representa algo a ser executado assincronamente. Tal como em atores *Akka*, não é possível definir explicitamente um núcleo do processador para executar

estas operações. A linguagem permite também, de seguida, juntar vários *Futures* num só de modo a que a espera de resultados seja apenas uma. Se não existisse esta possibilidade, a solução teria de esperar por um ator de cada vez, o que não seria o ideal.

Concluindo, para o caso de computações *SOMD* locais, a utilização de *Futures*, para além de permitir obter um melhor sistema de tipos, permitiria que a solução não tivesse de lidar com os pesados *ActorSystems* que fazem parte da utilização de atores *Akka*. No geral, a implementação poderia ficar um pouco mais leve e mais simples para o programador.

Ainda assim, este Capítulo não se trata de uma forma de retirar qualquer tipo de crédito à biblioteca *Akka*. Esta biblioteca permite criar programas completamente assíncronos de maneira muito facilitada. Mesmo utilizando *Futures* na solução, os atores remotos continuariam a ser os da biblioteca *Akka*. O Capítulo resume o problema de, apesar de num modelo de atores tudo ser um ator, isto não quer dizer que o programador tenha de criar atores para tudo.

6.4 Anotações

As anotações são uma parte importante da implementação do modelo *SOMD*. Apesar desta funcionalidade não ter sido pensada para ser desenvolvida nesta fase e dissertação, uma possível continuação de evolução do projeto terá a implementar.

Com a tradução de funções segundo diretiva de anotações implementada, será possível obter resultados reais da diferença de performance entre a ferramenta simplista *Scala/SOMD* e as concorrentes linguagens explícitas de paralelismo local e distribuído.

Ainda que não tenha sido realizada esta tradução, a partir da solução obtida é possível provar que efetivamente se consegue obter os dados desejados apenas a partir das anotações previamente pensadas: *@DIST*, *@REDUCE* e *@COMBINE*. A Listagem 6.2 demonstra uma simplificação da chamada original ao sistema que foi possível implementar com a versão atual da solução.

Listagem 6.2: Simplificação de uma chamada à solução para a soma de dois *Arrays*

```
1  SOMD.newSOMD(
2    args = List(arg0, arg1),
3    finalPartition = ParallelList[Array1[Int], Array[Int]](
4      Array1PartitionVal(arg0, 2),
5      Array1PartitionVal(arg1, 2)
6    )
7  )(
8    map = ck => {
9      val (ckA, ckB) = (ck(0), ck(1))
10     val res = new Array[Int](ckA.length)
11     for (i <- ckA.indices)
12       res(i) = ckA(i) + ckB(i)
13     res
```

```
14    },
15    reduce = ps => Array1.flatten(ps)
16  )
```

Na primeira secção de parâmetros os argumentos são recebidos e é desde logo definido qual o tipo interno que queremos usar sobre estes. A segunda secção define qual o processamento a efetuar sobre os dados recebidos e qual a forma de os agrupar para obter o resultado final.

6.5 Bibliotecas internas

A solução *Scala/SOMD* demonstra ser um balanço ideal entre versatilidade de uso e facilidade de programação em relação a outras soluções existentes. Isto deve-se à modularidade e extensibilidade da sua biblioteca de coleções, partições e reduções.

Apesar dos tipos de dados e outras funcionalidades a usar terem de ter uma implementação já existente no sistema, acontece que muitos dos problemas de computação paralela partilham metodologias de partição e redução dos seus argumentos.

Esta propriedade do sistema implementado onde, seguindo as ideias de programação funcional, todos os tipos de dados são desde logo conhecidos, permite que uma chamada ao *Scala/SOMD* possa ser realmente feita a partir de apenas algumas diretivas, independentemente do processamento que queremos aplicar aos dados.

6.6 Paradigma funcional em *Scala* adaptado a computações de alta performance

O paradigma de programação funcional é cada vez mais usado em sistemas de computação paralela e distribuída pela sua garantia de consistência de dados. Por esta razão, a linguagem *Scala* tem ganho alguma popularidade, principalmente a nível empresarial, por permitir desenhar ferramentas funcionais sobre a *JVM*, onde é possível usar de bibliotecas *Java* já existentes e consagradas.

Assim, é natural que muitos dos esforços atualmente existentes sobre a linguagem se debrucem sobre ferramentas e *frameworks* aplicadas à criação e gestão de servidores *Web*. Isto para além da dedicação contínua de trazer mais funcionalidades de linguagens puramente funcionais como *Haskell* para o *Scala*.

A solução *Scala/SOMD* demonstra como ferramentas como o *Akka*, normalmente usada em servidores pela facilidade em criar sistemas assíncronos, podem ser adaptadas para trabalhar sobre computações de alta performance. Assim, o *Scala/SOMD* implementa um sistema modular, onde cada utilizador tem a escolha sobre o paradigma que quer usar, sobre a biblioteca funcional *Akka*, usada para toda a gestão de trabalhadores do sistema.

Bibliografia

[1] G. Almasi. "PGAS (Partitioned Global Address Space) Languages". Em: *Encyclopedia of Parallel Computing*. 2011, pp. 1539–1545. DOI: 10.1007/978-0-387-09766-4_210. URL: https://doi.org/10.1007/978-0-387-09766-4_210.

[2] V. Cavé, J. Zhao, J. Shirako e V. Sarkar. "Habanero-Java: The New Adventures of Old X10". Em: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. PPPJ '11. Kongens Lyngby, Denmark: ACM, 2011, pp. 51–61. ISBN: 978-1-4503-0935-6. DOI: 10.1145/2093157.2093165. URL: http://doi.acm.org/10.1145/2093157.2093165.

[3] B. Chamberlain, D. Callahan e H. Zima. "Parallel Programmability and the Chapel Language". Em: *The International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312. DOI: 10.1177/1094342007078442. eprint: https://doi.org/10.1177/1094342007078442. URL: https://doi.org/10.1177/1094342007078442.

[4] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun e V. Sarkar. "X10: An Object-oriented Approach to Non-uniform Cluster Computing". Em: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: ACM, 2005, pp. 519–538. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094852. URL: http://doi.acm.org/10.1145/1094811.1094852.

[5] L. Dagum e R. Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". Em: *IEEE Comput. Sci. Eng.* 5.1 (jan. de 1998), pp. 46–55. ISSN: 1070-9924. DOI: 10.1109/99.660313. URL: https://doi.org/10.1109/99.660313.

[6] J. Dean e S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". Em: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004. URL: https://research.google.com/archive/mapreduce.html.

[7] A. M. Dias e H. Paulino. *Scala/SOMD implementation guide*. Jan. de 2018.

[8] D. Grove, J. Milthorpe e O. Tardieu. “Supporting Array Programming in X10”. Em: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY’14. Edinburgh, United Kingdom: ACM, 2014, 38:38–38:43. ISBN: 978-1-4503-2937-8. DOI: 10.1145/2627373.2627380. URL: http://doi.acm.org/10.1145/2627373.2627380.

[9] P. Keleher, A. L. Cox, S. Dwarkadas e W. Zwaenepoel. “TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems”. Em: *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*. WTEC’94. San Francisco, California: USENIX Association, 1994, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1267074.1267084.

[10] E. Marques e H. Paulino. “Single Operation Multiple Data - Data Parallelism at Subroutine Level”. Em: *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*. 2012, pp. 254–261. DOI: 10.1109/HPCC.2012.42.

[11] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*. Specification. 2009. URL: http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[12] H. Paulino e E. Marques. “Heterogeneous Programming with Single Operation Multiple Data”. Em: *Journal of Computer and System Sciences* 81 (2015), pp. 16–37. URL: http://citi.di.fct.unl.pt/publication/article.php?id=173.

[13] 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*. Rel. téc. Toronto, Canada, 2010. URL: http://www.cs.rochester.edu/u/cding/amp/papers/full/The%20Asynchronous%20Partitioned%20Global%20Address%20Space%20Model.pdf.

[14] V. A. Saraswat, O. Tardieu, D. Grove, D. Cunningham, M. Takeuchi e B. Herta. *A Brief Introduction To X10 (For the High Performance Programmer)*. 2012. URL: http://x10-lang.org/documentation/intro/latest/html/.

[15] P. Suter, O. Tardieu e J. Milthorpe. “Distributed Programming in Scala with APGAS”. Em: *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*. SCALA 2015. Portland, OR, USA: ACM, 2015, pp. 13–17. ISBN: 978-1-4503-3626-0. DOI: 10.1145/2774975.2774977. URL: http://doi.acm.org/10.1145/2774975.2774977.

[16] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, M. Vaziri e W. Zhang. “X10 and APGAS at Petascale”. Em: *ACM Trans. Parallel Comput.* 2.4 (mar. de 2016), 25:1–25:32. ISSN: 2329-4949. DOI: 10.1145/2894746. URL: http://doi.acm.org/10.1145/2894746.

[17] K. Varda. *Protocol Buffers: Google's Data Interchange Format*. Rel. téc. Google, jun. de 2008. URL: http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html.

[18] S. Voulgaris, D. Gavidia Simonetti e M. van Steen. "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays". English. Em: *Journal of Network and Systems Management* 13.2 (2005). steen2005.03, pp. 197–217. ISSN: 1064-7570. DOI: 10.1007/s10922-005-4441-x.

[19] M. Weiland. *Chapel , Fortress and X10 : novel languages for HPC*. UoE HPCx Ltd., out. de 2007. URL: http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0706.pdf.

[20] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker e I. Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". Em: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28. ISBN: 978-931971-92-8. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.