Uma implementação paralela do AIRS em Scala

Filipe Ferraz Salgado

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO
DE
MESTRE EM CIÊNCIAS

Programa: **Ciência da Computação**
Orientador: **Prof. Dr. Francisco Carlos da Rocha Reverbel**

Durante a realização deste trabalho o autor recebeu apoio financeiro da CAPES.

São Paulo, março de 2012

Uma implementação paralela do AIRS em Scala

Esta versão definitiva da dissertação
contém as correções e alterações sugeridas pela
Comissão Julgadora durante a defesa realizada
por Filipe Ferraz Salgado em 15/09/2010.

Comissão Julgadora:

Prof. Dr. Francisco Carlos da Rocha Reverbel (orientador) - IME-USP

Prof. Dr. Alair Pereira do Lago - IME-USP

Profa. Dra. Graça Bressan - POLI-USP

Agradecimentos

Em primeiro lugar, gostaria de agradecer à minha família por sempre me apoiar, aconselhar e incentivar para que eu alcançasse mais esse objetivo. Agradeço também à minha noiva Cristina por estar sempre ao meu lado disposta a tudo para me ver bem. E claro, a todos os meus familiares e amigos, próximos e distantes, que torceram por mim e que também serviram de ombros amigos para me ouvir algumas vezes.

Meus agradecimentos também vão ao professor Francisco Carlos da Rocha Reverbel, meu orientador, pela ajuda prestada para que eu pudesse aprender um pouco mais e desenvolver esse trabalho. Estendo os agradecimentos aos professores Marco Dimas Gubitoso e Alair Pereira do Lago que fizeram parte da banca de qualificação e me ajudaram a melhorar o trabalho. Agradeço ainda ao aluno Marcelo de Rezende Martins por ter colaborado com um script para execução dos testes.

Finalmente, agradeço à CAPES por ter me apoiado financeiramente durante meus estudos.

Resumo

Com o avanço tecnológico dos últimos anos passou a ser normal vermos microprocessadores com múltiplos núcleos (*cores*). A expectativa é de que o crescimento da quantidade de núcleos passe a ser maior do que o crescimento da velocidade desses núcleos. Assim, além de se preocuparem em otimizar algoritmos sequenciais, os programadores começaram a dar mais atenção às possibilidades de aproveitamento de toda a capacidade oferecida pelos diversos *cores*.

Existem alguns modelos de programação que permitem uma abordagem concorrente. O modelo de programação concorrente mais adotado atualmente é o baseado em *threads*, que utiliza memória compartilhada e é adotado em Java. Um outro modelo é o baseado em troca de mensagens, no qual as entidades computacionais ativas são denominadas atores. Nesse trabalho, estudamos a linguagem Scala e seu modelo de atores. Além disso, implementamos em Scala uma versão paralela de um algoritmo de classificação que simula o sistema imunológico dos animais, o AIRS paralelo, e comparamos seu desempenho com a versão em Java.

Palavras-chave: Scala, AIRS, Java, programação orientada a objetos, programação funcional, paralelo, ator, imunológico, concorrente, WEKA.

Abstract

With the technological advance of the last years it has been normal to see microprocessors with multiple cores. The expectation is that the growth of number of cores becomes greater than the growth of the speed of these cores. This way, besides worrying about optimizing sequential algorithms, developers started to give more attention to the possibilities of profiting from all capacity offered by the cores.

There exists a few programming models that allow a concurrent approach. In these days, the most adopted concurrent programming model is the one based on *threads*, which uses shared memory and is adopted in Java. Other model is based on message passing, on which the active computational entities are called actors. In this project, we studied Scala language and its model based on actors. Besides that, we implemented in Scala a parallel version of a classification algorithm that simules the immune system of the animals, parallel AIRS, and compared its performance with the Java version.

Keywords: Scala, AIRS, Java, object oriented programming, functional programming, parallel, actor, immune, concurrent, WEKA.

Sumário

1 Introdução

Até pouco tempo atrás, os programadores estavam acostumados a pensar nas aplicações como uma sequência de tarefas. Isso implicava em um código sequencial, onde era necessário esperar o término de um trecho do programa para que outro pedaço de código pudesse ser executado. Em grande parte, isso ocorria devido à limitação física, afinal de contas os computadores vinham com processadores que possuíam apenas um núcleo. Além disso, não havia a demanda que se tem hoje. Era raro um site que disponibilizava uma interface web para que várias pessoas realizassem requisições simultâneas aos servidores. Então, assuntos como paralelização e concorrência não eram abordados com muita frequência. Os próprios desenvolvedores não se preocupavam com esses aspectos enquanto programavam.

Entretanto, com a evolução da tecnologia o cenário se modificou. A chegada da Internet tornou usual o acesso simultâneo de várias pessoas a uma aplicação e, consequentemente, os servidores passaram a receber várias consultas ao mesmo tempo. Além disso, os processadores passaram a ter mais núcleos, já sendo comum encontrarmos dois núcleos por processador nos dias atuais. Com isso, os programadores passaram a se preocupar mais com os acessos concorrentes à memória compartilhada efetuados pelos processos ativos e também em como aproveitar melhor todos os núcleos disponíveis visando o aumento da produtividade.

Atualmente, Java [21, 24] é a segunda linguagem mais utilizada pelos desenvolvedores de software [35]. Nessa linguagem pode-se utilizar o modelo baseado em *threads*, que utiliza memória compartilhada. É necessário que os desenvolvedores cuidem da sincronização dos dados, já que esses podem ser acessados pelos vários processos[1] concorrentemente. Caso o acesso aos dados não seja cuidadosamente controlado, o comportamento da aplicação como um todo passa a ser imprevisível [17].

Outro modelo que tem sido bastante discutido recentemente, apesar de não ser novo, é o baseado em troca de mensagens. Nesse modelo, a comunicação é feita através de mensagens, como o nome sugere, ao invés de uma memória compartilhada. As entidades responsáveis por essa troca de mensagem são comumente denominadas atores [2, 20]. Em consequência do recebimento de uma mensagem, um ator pode: *(i)* enviar mensagens a outros atores, *(ii)*

[1]Neste texto usamos o termo processo para nos referir a "processos leves" (*lightweight processes*), como *threads* Java ou processos de Erlang.

criar outros atores e *(iii)* definir o comportamento que ele adotará quando receber a próxima mensagem. Esse modelo elimina a necessidade do controle de acesso concorrente aos dados compartilhados, pois não há compartilhamento de dados, mas pode adicionar uma sobrecarga (*overhead*) no tempo de execução da aplicação devido à comunicação.

Nesse contexto, linguagens funcionais como Erlang [1, 12] e Scala [27, 31] têm despertado grande interesse por apresentarem características que tornam o desenvolvimento de aplicações concorrentes menos complexo. Uma das principais características é a utilização do paradigma de troca de mensagens (modelo de atores) ao invés de memória compartilhada. Nessa dissertação, realizamos um estudo sobre a linguagem Scala e seu modelo de atores. Optamos por estudar Scala por ser uma linguagem com características tanto funcionais quanto de orientação a objetos. Além disso, ela é completamente interoperável com Java, rodando também na JVM, o que garante a sua portabilidade.

Como experiência do modelo de atores de Scala, optamos por trabalhar com o AIRS (*Artificial Immune Recognition System*) [39], um algoritmo de aprendizado de máquina bioinspirado que pode ser paralelizado. Tomamos como ponto de partida uma implementação já existente do AIRS paralelo escrita em Java para implementarmos uma versão desse algoritmo em Scala. Em seguida, realizamos experimentos com as duas versões do algoritmo para que pudéssemos comparar o desempenho entre elas. Por fim, integramos o algoritmo desenvolvido à bancada de ferramentas de aprendizado computacional WEKA [15, 40]. Tal integração evidencia a interoperabilidade entre Scala e Java, já que a WEKA é desenvolvida em Java, e aumenta as possibilidades de uso da bancada de ferramentas.

Em nossa implementação do AIRS paralelo em Scala, procuramos manter fidelidade ao algoritmo original. Isso quer dizer que optamos por não implementar no algoritmo as melhorias que já foram propostas na literatura [7]. Com isso, pretendemos avaliar apenas o desempenho das linguagens e dos respectivos modelos adotados para controle de acesso concorrente para um mesmo algoritmo.

1.1 Organização do texto

O restante deste texto está organizado da seguinte forma: no Capítulo 2 apresentamos os trabalhos relacionados. Em seguida, no Capítulo 3 é apresentada uma visão geral da linguagem Scala e do seu modelo de atores. No Capítulo 4 detalhamos o algoritmo AIRS. Já no Capítulo 5, descrevemos nossa implementação do AIRS paralelo em Scala e no Capítulo 6 detalhamos a análise de desempenho feita entre as versões Java e Scala do algoritmo. Por fim, o Capítulo 7 apresenta nossas conclusões, os trabalhos futuros e as contribuições do nosso estudo.

2 Trabalhos relacionados

Neste capítulo apresentamos alguns trabalhos que foram estudados durante o desenvolvimento do nosso projeto. Dividimos esse capítulo em duas partes: a primeira aborda estudos relacionados a duas linguagens que são frequentemente comparadas a Scala (Java [21] e Erlang [12]). Na segunda parte, iremos mostrar um trabalho relacionado à parte da implementação que realizamos (CLONALG [8]).

2.1 Linguagens

Entre os estudos feitos durante esse trabalho, estão as linguagens que possuem características semelhantes às de Scala. Como veremos, essas semelhanças podem ser identificadas em vários pontos, como na sintaxe, no modelo utilizado para tratar o acesso concorrente aos dados, entre outros. Nas Seções 2.1.1 e 2.1.2, fazemos uma pequena análise de duas dessas linguagens, frequentemente comparadas a Scala, e no Capítulo 3 explicamos o motivo pelo qual escolhemos Scala.

Queremos deixar claro que nossa intenção não é a de analisar as linguagem em sua totalidade, mas simplesmente identificar algumas de suas características. Tais características serviram de parâmetros para podermos optar por Scala.

2.1.1 Java

A linguagem Java é uma das primeiras a ser lembrada quando falamos de Scala. Isso talvez seja devido ao fato de que na própria página inicial de Scala [31] já existam referências dizendo que programadores de Java podem ser mais produtivos se utilizarem Scala e que o código pode ser reduzido em duas e até três vezes [31]. Além disso, Scala tem algumas semelhanças com Java como mostraremos mais adiante.

Java é considerada uma linguagem orientada a objetos [22]. Isso significa que as entidades são modeladas através de objetos, representados pelas classes. A comunicação entre esses objetos é feita através de chamadas de métodos das classes. Além disso, ela é estaticamente tipada. Ou seja, o tipo de cada variável e parâmetro são determinados pelo programador, de

modo que fiquem explícitos para o compilador antes da compilação. Na figura 2.1 vemos um exemplo simples de código Java evidenciando sua sintaxe:

```
package br.ime.usp

import java.util.Calendar;

public class GetTempoEmMilissegundos {
   public static void main(String[] args) {
      Calendar cal = Calendar.getInstance();
      System.out.println("A data atual representa :" + cal.getTimeInMillis()
         + " milissegundos");
   }
}
```

Figura 2.1: Classe Java que obtém a data atual e a imprime em milissegundos.

Com relação ao controle de acesso aos dados compartilhados, Java utiliza o modelo de *threads*. Nesse modelo, cada *thread* representa um dentre os vários processos que podem ser executados concorrentemente acessando variáveis compartilhadas na memória principal. Caso o controle de acesso a essas variáveis não seja bem feito, o comportamento da aplicação passa a ser inesperado. Isso se deve ao fato de que o valor das variáveis lido por cada processo dependerá da ordem que cada processo acessar as variáveis em questão [17].

De acordo com [24], para realizar tal controle, de modo que um determinado processo possa realizar suas operações sem interferência de outros processos, costuma-se utilizar um mecanismo de sincronização, que é implementado através de monitores. Nesse mecanismo, cada objeto é associado a um monitor. Esse monitor possui operações de trava (*lock*) e destrava (*unlock*). Uma *thread* pode travar um determinado objeto quantas vezes quiser. Enquanto um objeto estiver travado por alguma *thread*, qualquer outra *thread* que tentar obter a trava daquele objeto terá que esperar numa fila, até que esse seja liberado. Para liberar um objeto, é utilizada a operação *unlock*. Através do modificador `synchronized`, é possível delimitar um bloco de código (que pode ser o corpo de um método) cuja execução requer uma trava, a qual será requisitada quando se entra no bloco e liberada na saída do bloco. Em consequência disso, a *thread* que fez a requisição fica aguardando até que se complete a execução do bloco de código que se deseja sincronizar, por exemplo um método. Apesar de existirem outros mecanismos, como leitura e escrita de variáveis voláteis e outras classes disponíveis no pacote `java.util.concurrent`, este é o mais utilizado. Na figura 2.2 vemos um trecho de código em Java que mostra como pode ser controlado o acesso aos dados pelos vários processos utilizando o modificador `synchronized`:

```
public class Contador {
...
   public synchronized void incrementar() {
      c = c + 1;
   }
...
}
```

Figura 2.2: Método `incrementar` da classe `Contador` com acesso controlado.

Para evitar que várias *threads* chamem o método `incrementar` ao mesmo tempo, podendo causar valores indesejáveis na variável `c`, o método é sincronizado. Na figura 2.3, uma *thread* em Java que poderia ser usada para acessar o método `incrementar`:

```
public class ExemploThread extends Thread {

   private Contador contador = null;

   public ExemploThread(Contador contador) {
      this.contador = contador;
   }

   public void run() {
      for(int i = 0; i < 1000; i++) {
         contador.incrementar();
      }
   }

   public static void main(String args[]) {
      Contador contador = new Contador();
      (new ExemploThread(contador)).start();
      (new ExemploThread(contador)).start();
   }
}
```

Figura 2.3: Exemplo de *thread* em Java usando memória compartilhada.

Ao executar o método `ExemploThread.main()` da figura 2.3, uma instância da classe `Contador` é criada e essa mesma instância é passada para duas instâncias da *thread* `ExemploThread`, que são inicializadas. Com a inicialização, cada uma das *threads* executa o método `ExemploThread.run()`, que realiza 1000 iterações chamando o método `Contador.incrementar()`. Como o método `Contador.incrementar()` é sincronizado, o valor da variável `c` na instância de `Contador` não será inesperado, apesar do compartilhamento de memória.

2.1.2 Erlang

Outra linguagem que também é lembrada quando começamos a estudar Scala é Erlang [1]. A principal semelhança entre essas linguagens são os modelos de atores adotados por elas.

Ao contrário de Java, Erlang é uma linguagem funcional [6]. Em linguagens funcionais, um programa é definido por funções, sem que haja o conceito de classe. Essas funções podem receber parâmetros, inclusive outras funções, e devolvem um valor de retorno, que também pode ser uma função. Seus criadores tinham em mente projetar uma linguagem que resolvesse vários dos problemas que um sistema operacional deveria resolver: suporte a sistemas grandes, não precisar interromper sistemas para poder fazer manutenção, bom controle de concorrência, *garbage collection* eficiente, ... [4]

De acordo com [30], Erlang é uma linguagem dinamicamente tipada. Os tipos de dados de Erlang podem ser divididos em duas categorias. A primeira é formada pelos números, átomos, identificadores de processos e referências. A segunda categoria é constituída pelos tipos compostos, que são as listas e as tuplas. Na figura 2.4, podemos ver a sintaxe de Erlang:

```
-module(math).
-export([fac/1]).

fac(N) when N > 0 -> N * fac(N-1);
fac(0)            -> 1.
```

Figura 2.4: Exemplo de cálculo de fatorial em Erlang.

Repare que o módulo (`module`) é equivalente ao pacote (`package`) de Java, mas não há o conceito de classes. Após a definição do módulo, declaram-se as funções que serão exportadas pelo módulo e a quantidade de parâmetros delas. O simples exemplo calcula o fatorial de um número maior do que zero.

Em Erlang, a comunicação entre os processos ativos é feita através de envio de mensagens assíncronas. Isso quer dizer que, ao invés de usar uma memória compartilhada, os projetistas optaram pelo modelo no qual não existe um estado global compartilhado pelos processos em execução. Ao invés disso, esses processos trocam mensagens entre si.

Os processos de Erlang são como as *threads* de Java, porém não necessitam das travas, uma vez que se comunicam por mensagens. Além disso, são processos mais leves. Na figura 2.5 podemos ver a criação de um processo concorrente através da função `spawn`. Esse processo executará a função `loop` do módulo `counter` utilizando a lista de parâmetros `[0]`. Em seguida, a mensagem `increment` é enviada para esse processo.

```
...
Pid = spawn(counter, loop, [0]),
Pid ! increment,
...
```

Figura 2.5: Criação de um novo processo por meio da função `spawn` passando como parâmetros o módulo (`counter`) ao qual a função pertence, a função que será executada (`loop`) e os parâmetros para a função (`[0]`). Em seguida, é enviada a mensagem `increment` ao processo criado.

O fato da comunicação ser feita através de envio de mensagens evita que o programador seja afetado pelos problemas típicos do modelo de *threads* e memória compartilhada [13]:

- sincronização de recursos que não precisariam ser sincronizados.
- esquecimento de sincronização de algum recurso.
- utilização de muitas travas em uma arquitetura paralela.

Na figura 2.6, mostramos o corpo do processo criado quando chamamos `spawn` no exemplo anterior.

```
-module(counter).
-export([loop/1]).

loop(Val) ->
   receive
      increment ->
         loop(Val+1);
      {From, Val} ->
         From ! {self(), Val},
         loop(Val);
      stop ->
         true;
      Other ->
         loop(Val)
   end.
```

Figura 2.6: Processo em Erlang utilizando casamento de padrões para tratar as mensagens recebidas.

A função `loop` define as mensagens que ela irá receber. Essa definição utiliza casamento de padrões. Seguindo o exemplo anterior, onde a mensagem `increment` foi enviada para o processo criado, tal mensagem seria tratada no primeiro caso.

2.2 CLONALG Paralelo

Baseando-se no sistema imunológico dos animais vertebrados, os pesquisadores começaram a desenvolver algoritmos que, na medida do possível, imitam o comportamento de tais sistemas. Esses algoritmos passaram a ser chamados de Sistemas Imunológicos Artificiais (*Artificial Immune Systems (AIS)*). Os *AIS* [3] são algoritmos de aprendizado de máquina cujo objetivo é aproveitar a eficiência do sistema imunológico animal para resolver problemas complexos. Algoritmos de aprendizado de máquina são aqueles que tentam encontrar padrões a partir de dados fornecidos. Quanto mais dados o algoritmo tiver, melhor será.

Um desses algoritmos é o *Clonal Selection Algorithm* (CLONALG [8,10]). Ele foi baseado na teoria da seleção por clones (*clonal selection theory* [14]) cuja ideia é de que aquelas células que são boas no reconhecimento dos antígenos devem ser selecionadas e propagadas. Inicialmente o CLONALG foi chamado de CSA (*clonal selection algorithm*) e somente após ser reformulado recebeu o atual nome. Em [37] os autores apresentam uma visão geral do algoritmo antes de paralelizá-lo:

1. Inicialização: Cria uma população aleatória P de N anticorpos e uma M de células memória.

2. Apresentação Antigênica: Para cada padrão antigênico em S faça:

   a) Expansão por Clones: Determina a afinidade de cada anticorpo em P ao padrão antigênico dado. Seleciona os *n1* melhores anticorpos para produzir clones.

   b) Maturação de Afinidade: Modifica os clones baseado nas afinidades.

   c) Seleção por Clones: Seleciona o melhor clone e substitui uma célula memória em M se o clone tiver melhor afinidade.

   d) Metadinâmica: Substitui *n2* anticorpos em P com células geradas aleatoriamente.

3. Ciclo: Repete passo 2 para um número fixo de gerações ou até as células memórias em M terem convergido para um limite.

No algoritmo, N é o tamanho da população P, *n1* é número de anticorpos selecionados para mutação e clonagem durante cada passo da apresentação antigênica; e *n2* é o número de células aleatórias geradas para substituir células em P após cada apresentação antigênica. O conjunto S de padrões corresponde aos antígenos que devem ser reconhecidos.

Pensando biologicamente, o objetivo do CLONALG é obter um conjunto de anticorpos. Esses anticorpos seriam capazes de identificar novos antígenos que tentassem invadir o sistema

biológico. Em computação, os anticorpos representam o conjunto de boas soluções para o problema que se deseja resolver. Já os antígenos, seriam novas instâncias ainda desconhecidas do problema.

A sua paralelização veio de maneira trivial. Como não há comunicação entre as células, cada processador poderia cuidar do desenvolvimento de cada célula. Assim, os dados de entrada são divididos entre os processadores e, terminados os cálculos, um processo central reúne os resultados. O CLONALG se mostrou bastante eficiente em vários problemas nos quais foi aplicado.

2.2.1 Relação com o nosso trabalho

Os bons resultados apresentados pelos experimentos com o CLONALG [37] paralelo estimularam a pesquisa sobre a possibilidade de se paralelizar outros algoritmos AIS, como o AIRS. A possibilidade se comprovou através do AIRS2 paralelo [38]. Em nosso trabalho, pretendemos utilizar o AIRS2 paralelo para mostrar algumas características de Scala e comparar a versão Scala desse algoritmo com sua versão Java. Além disso, o CLONALG é um precursor do AIRS como veremos no Capítulo 4.

3 Scala

O sugestivo nome Scala (*scalable language*) [28, 31] reflete quais eram os propósitos iniciais dos projetistas dessa linguagem. Ela foi construída com o intuito de permitir que os sistemas fossem realmente baseados em componentes e, consequentemente, reaproveitáveis, assim como os computadores são feitos de circuitos integrados. Scala começou a ser desenvolvida em 2001 possuindo características tanto funcionais quanto de orientação a objetos. Do aspecto funcional, vieram os conceitos de que funções são valores de primeira classe e que alguns objetos podem ser decompostos por casamento de padrões. Rod Burstall definiu em seu artigo [9] que para um valor ser de primeira classe, é necessário que ele possa ser:

- atribuído a uma variável;
- passado como argumento para uma função;
- devolvido por uma função;
- armazenado em uma estrutura de dados;
- criado em qualquer contexto.

A decomposição por casamento de padrões permite que seja encontrada uma estrutura específica a partir de padrões preestabelecidos. Em seguida, pode-se fazer o devido tratamento para a estrutura encontrada. Uma explicação mais detalhada é dada na Seção 3.4.

Do ponto de vista da orientação a objetos, vem o fato de que todo valor é um objeto e toda operação é uma chamada de método [29]. Além disso, os criadores de Scala quiseram aproveitar o fato de grande parte das aplicações feitas atualmente ser desenvolvida em Java e C#. Com isso em mente, resolveram fazer com que a nova linguagem fosse interoperável com aquelas duas para que todo o trabalho já desenvolvido não fosse perdido e também a fim de facilitar a adoção de Scala.

Pensando nesses aspectos, eles chegaram à conclusão de que seriam necessários três tipos de abstrações de linguagem de programação para fazer com que os componentes fossem reaproveitáveis: membros abstratos que são tipos; tipo próprio (*selftype*); e composição [29].

3.1 Interoperabilidade com Java e C#

Inicialmente Scala foi desenvolvida para funcionar apenas com Java, não como uma extensão, pois é uma linguagem independente, mas podendo interoperar de maneira muito boa com Java. Os programas feitos em Scala são compilados em JVM bytecodes e rodam na JVM. Além disso, é possível importar bibliotecas Java para utilizá-las nas aplicações feitas em Scala, com isso garante-se o reaproveitamento de toda a tecnologia existente. Mais tarde, os criadores de Scala fizeram a implementação para compilar para .NET. Dessa maneira passou a ser possível também a utilização da nova linguagem com C# [32]. Contudo, a vertente que mantém o suporte a Java é mais ativa, uma vez que fazia parte da ideia inicial do projeto e, pelo menos por enquanto, parece não haver muitos interessados em manter atualizada a parte do projeto direcionada à plataforma .NET.

Um exemplo simples dessa interoperabilidade com Java é mostrado na figura 3.1. Todas as classes do pacote `java.lang` são importadas automaticamente. Repare que é possível utilizar o método `System.out.println()` de maneira direta. Para utilizar classes de outras bibliotecas é necessário importar de maneira explícita através da expressão `import`, de maneira semelhante ao que se faz em Java.

```
package hello

object HelloWorld
{
   def main(args: Array[String]): Unit = {
      System.out.println("Hello world from Scala")
   }
}
```

Figura 3.1: `Object` Scala chamando método Java.

3.2 Abstração

Abstração de dados é um mecanismo que permite esconder a implementação dos componentes, mostrando apenas aquilo que o componente necessita para seu funcionamento. Scala suporta as duas grandes formas de abstração nas linguagens de programação: parametrização e membros abstratos. A primeira é tipicamente funcional enquanto que a segunda é tipicamente orientada a objetos [27].

3.2.1 Parametrização

A abstração funcional, conhecida em Java como *generics*, permite que se abstraia o tipo de valor que será utilizado na implementação da classe. Assim, pode-se deixar para fornecer o tipo no instante em que a classe for utilizada, ou ainda, que ele seja determinado automaticamente pelo compilador Scala a partir do valor passado. Também é possível utilizar essa abstração na declaração dos parâmetros dos métodos. Isso permite que não seja necessário estipular um determinado tipo para o parâmetro, deixando-o genérico, e, ao mesmo tempo, mantendo a consistência interna da função [29]. Essas características estão representadas na figura 3.2.

```
class Celula[T](valorInicial: T) {
   private var valor: T = valorInicial
   def get: T = valor
   def set(x: T): Unit = { valor = x }
}

def trocar[T](x: Celula[T], y: Celula[T]): Unit = {
   val t = x.get; x.set(y.get); y.set(t)
}

def main(args: Array[String]) = {
   val x: Celula[int] = new Celula [int](1); //val x = new Celula(1)
   val y: Celula[int] = new Celula [int](2); //val y = new Celula(2)
   trocar[int](x, y) //trocar(x,y)
}
```

Figura 3.2: Abstração funcional em Scala e exemplo da inferência automática no código comentado.

Primeiro, podemos ver a abstração do tipo da classe `Celula`, representada pelo parâmetro `T` que simboliza um tipo genérico. Em seguida, podemos observar que no método `trocar` são utilizados os parâmetros que são tipos (*type parameters*) que garantem o correto funcionamento interno mesmo sem especificar de quais tipos devem ser as classes `Celula` dos parâmetros x e y, bastando que eles sejam iguais. Por fim, é possível ver no código comentado a possibilidade do uso sem a necessidade de informar qual o tipo utilizado já que o compilador Scala realiza a inferência dos tipos.

Outra característica dessa abstração é a capacidade que se tem para limitar o tipo desejado. Ou seja, fazer com que as possíveis classes sejam apenas aquelas que forem subtipos ou supertipos de outra [29].

3.2.2 Membros Abstratos

Típico modelo de abstração orientado a objetos, os membros abstratos são fundamentais para a construção de sistemas baseados em componentes. Para efeito de comparação, podemos perceber que, diferentemente da abstração funcional, essa forma de abstração não utiliza nem parâmetros que são tipos nem valores. Ao invés disso, é utilizada uma classe abstrata com um membro abstrato que é um tipo `T` e um membro abstrato que é um valor. Assim, no instante da utilização, é necessário indicar implementações concretas desses elementos abstratos. A figura 3.3 ilustra a utilização dos membros abstratos e evidencia as diferenças em relação à abstração funcional.

```
abstract class Celula{
   type T
   val valorInicial: T
   private var valor: T = valorInicial

   def get: T = valor

   def set(x: T): Unit = { valor = x }
}

val celula = new Celula { type T = int; val valorInicial = 1 }
celula.set(celula.get * 2)
```

Figura 3.3: Membros abstratos em Scala.

A partir das definições das classes `Celula` dos exemplos anteriores percebemos que é possível transformar uma abstração funcional (*generics*) em uma orientada a objetos (membros abstratos). Para tal, temos que fazer com que os parâmetros da classe `C` do modelo funcional passem a ser os membros abstratos declarados como `type` no modelo orientado a objetos. Além disso, no momento da instanciação é necessário substituir `new C[TIPO]` por `new C {type T = TIPO}`. Caso `C[TIPO]` apareça como um construtor de uma superclasse, ainda há a necessidade de se declarar `type t = TIPO` na classe que estiver utilizando a herança [27].

O caminho inverso (transformar uma abstração orientada a objetos em uma funcional) é bem mais difícil e requer a reescrita do programa como um todo, ao invés de uma sequência de transformações simples.

Um novo conceito utilizado em Scala é o tipo dependente de caminho (*path-dependent type*). Ele permite que, mesmo sem saber o tipo concreto do valor, certas operações possam ser realizadas, desde que sejam respeitados os tipos envolvidos na operação. Um exemplo simples é listado na figura 3.4:

```
def reiniciar (c: Celula): Unit = c.set(c.valorInicial);
```

Figura 3.4: O tipo `c.T` é um tipo dependente de caminho em Scala.

No exemplo da figura 3.4 vemos que o método `reiniciar` só é válido pois a função `c.set` recebe como parâmetro um objeto do tipo `c.T` que é justamente o tipo da expressão `c.valorInicial`. O tipo `c.T` é um tipo dependente de caminho. No caso geral, um tipo dependente de caminho tem a forma $x_1.x_2...x_n$.t, onde x_1, $x_2...x_n$ são valores imutáveis e t é um tipo que é membro de x_n. Note que todo o caminho $x_1.x_2...x_n$ deve ser imutável, caso contrário a consistência do sistema de tipos não é assegurada.

Assim como no modelo funcional de abstração, também é possível impor restrições às classes, fazendo com que as possíveis especializações da classe abstrata sejam aquelas limitadas superiormente ou inferiormente pelo tipo determinado [29].

```
abstract class Grafo {
   type No <: NoBase;

   class NoBase requires No {
      def conectarA(n: No): Aresta = new Aresta(this, n);
   }

   class Aresta(de: No, para: No) {
      def origem() = de;

      def destino() = para;
   }
}
```

Figura 3.5: Uso do limitante (<:) e da anotação `requires` em Scala.

Na figura 3.5 podemos ver, além do uso do limitante (<:), a palavra `requires`. O limitante restringe as classes do tipo `No` que poderão ser utilizadas para apenas aquelas que sejam subtipos da classe `NoBase`. Já a anotação `requires` serve para determinar qual o tipo que será utilizado quando aparecer a palavra `this`, por isso essa expressão é chamada de anotação de tipo próprio (*selftype annotation*).

A necessidade dessa construção vem do fato de que o `this` não se refere, necessariamente, à classe atual. No exemplo em questão, se retirássemos a expressão `requires No`, teríamos um erro de compilação, pois o tipo do parâmetro esperado pelo construtor da classe `Aresta` seria `No` enquanto que o tipo passado seria `NoBase`. Uma alternativa seria tirar a expressão `requires No`, substituir `this` por `self` e definir um método `self` que devolveria um objeto

da classe `No`. Assim, quem fosse definir uma classe concreta a partir da abstrata `Grafo`, teria que implementar também o método `self` na "concretização" do tipo `No`. Por se mostrar tão recorrente esse processo, foi criada a expressão `requires`.

A princípio surge a dúvida se qualquer classe é aceita como tipo próprio de outra, ou seja, se pode ser usada na anotação de tipo próprio. Isso é possível graças a duas condições. A primeira é que o tipo próprio de uma classe deve ser subtipo dos tipos próprios de todas as suas classes bases (ver definição de classe base na Seção 3.3). A segunda condição é que quando uma classe é instanciada através da expressão `new`, ocorre uma verificação se o tipo próprio dessa classe é um supertipo do tipo do objeto que está sendo criado [27].

3.3 Composição

Antes de começarmos a falar sobre o modelo de composição adotado em Scala, fundamental para a ideia de criação de sistemas baseados em componentes, vamos apresentar algumas definições básicas.

Trait: Forma especial de classe abstrata. Pode ser utilizada em qualquer situação onde se utilizaria uma classe abstrata comum. Entretanto, apenas `trait` pode ser usada como *mixin*.

Mixin: Dada a seguinte declaração da classe C: `class C extends` C_0 `with` C_1 `with ... with` C_n. Uma classe *Mixin* é aquela que aparece associada à palavra `with` na declaração.

Super classe: Dada a seguinte declaração da classe D: `class D extends` D_0 `with` D_1 `with ... with` D_n. Uma super classe é aquela que aparece associada à palavra `extends` na declaração.

Object: Quando se usa a expressão `object`, define-se uma classe com uma única instância, comumente chamada de *singleton*.

Classes Base da classe C: São todas as classes que podem ser alcançadas partindo-se de determinada classe C até chegar ao topo da hierarquia de classes de Scala.

Composição em Scala é feita através de um modelo que é uma forma de herança múltipla. Denominada composição *mixin* esse modo de composição permite que seja aproveitada apenas a diferença entre as classes herdadas, ou seja, serão utilizadas apenas as definições de métodos das classes *mixin* que não estiverem na super classe. Para que esse modelo funcione é necessário que a ordem das classes bases esteja bem estabelecida.

3.3.1 Linearização de Classes

O conceito de linearização de classes em Scala resolve o problema de ordenação citado anteriormente. Consideremos a seguinte declaração da classe C:

`class C extends` B_0 `with` B_1 `with ... with` B_n

A linearização de classe de C, *L(C)*, é definida pela equação 3.1.

$$L(C) = \{C\} +^{>} L(B_n) +^{>} ... +^{>} L(B_0) \tag{3.1}$$

Na equação 3.1, $+^{>}$ significa a concatenação dos conjuntos com substituição dos elementos do operando da esquerda pelos elementos do operando da direita, caso eles se repitam. Em outras palavras, sua linearização de classe começaria com a linearização da classe B_0, constituindo a parte final de *L(C)*. Em seguida, precedendo *L(B_0)*, seriam colocadas todas as classes da linearização de B_1, menos aquelas que já apareceram em *L(B_0)*. Assim sucessivamente até adicionar a própria classe C como primeira classe de sua linearização de C.

Com esse conceito podemos esclarecer várias questões relacionadas à hierarquia. Olhando para a figura 3.6 podemos observar que a classe `Iterador` herda métodos tanto da classe `IteratorDeString` quanto da `IteratorComForeach`. Isso nos leva à dúvida sobre como Scala resolve a questão de *overriding*, ou seja, qual, entre os possíveis métodos, é realmente utilizado. A solução encontrada é parecida com a adotada em Java e é baseada nos conceitos de membros concretos e abstratos.

Odersky e Zenger [29] definiram precisamente esses conceitos. Para eles, membros concretos de uma classe C são todas as definições concretas M em C e suas classes base, exceto se já houver alguma definição concreta de um membro igual a M em uma classe base precedente em *L(C)*. Por outro lado, membros abstratos de uma classe C são todas definições abstratas M em C e suas classes base, exceto se C já possuir alguma definição concreta de um membro igual a M ou se já houver uma definição abstrata de um membro igual a M em uma classe base precedente.

Com essas definições pode-se concluir que definições concretas sempre sobrescrevem definições abstratas e que para duas definições M e M', ambas concretas ou abstratas, M sobrescreve M' se M aparecer em uma classe, em *L(C)*, que precede a classe na qual M' é definida.

Outra questão a ser considerada é o modo como a chamada *super* é tratada em Scala. A classe acessada através dessa expressão não é decidida de maneira estática. Por exemplo, se tivermos a seguinte definição da classe C:

`class C extends B with` B_1 `with` B_2

de modo que a classe *mixin* B_1 tenha uma chamada *super*, então ela irá acessar o método correspondente na super classe B, seguindo a linearização de C. Ou seja, para que esteja correto, é necessário que referencie estaticamente algum membro de uma super classe. Mas,

```
trait IteradorAbstrato[T] {
   def hasNext: boolean

   def next: T
}

trait IteradorComForeach[T] extends IteradorAbstrato[T] {
   def foreach(f: T => unit): Unit =

   while (hasNext) f(next)
}

class IteradorDeString(s: String) extends IteradorAbstrato[char] {
   private var i = 0

   def hasNext = i < s.length

   def next = { val x = s charAt i; i = i + 1; x }
}

object Teste {
   def main(args: Array[String]): Unit = {
      class Iterador extends IteradorDeString(args(0))
            with IteradorComForeach[char]

      val iter = new Iterador
      iter foreach System.out.println
    }
}
```

Figura 3.6: Exemplo em Scala que imprime os caracteres de uma *string* usando um iterador.

mais do que isso, é necessário conhecer a linearização da classe que está sendo definida na composição para que se saiba qual classe está sendo realmente acessada [27].

3.4 Atores

O modelo de atores de Scala é muito semelhante ao utilizado em Erlang [12]. Sua principal característica está no fato da comunicação entre os processos ser feita através de troca de mensagens. Esse paradigma é mais seguro para o desenvolvedor quando comparado ao método baseado em *threads*, que utiliza travas para controlar os acessos concorrentes aos dados compartilhados. Devido ao atual desenvolvimento do hardware, com os processadores tendo mais núcleos, e ao crescimento do interesse dos programadores por aplicações distribuídas, esse

modelo baseado na troca de mensagens passou a ganhar mais espaço.

Alguns aspectos identificados por Haller e Odersky [19] que fazem com que a programação concorrente seja facilitada são:

- O modelo baseado em mensagens é mais seguro, no que diz respeito à concorrência, do que o baseado em memória compartilhada. O programador não precisa se preocupar com travas.
- Comparando-se com *threads*, os atores Scala podem ser bem mais leves. De acordo com os autores, mais de 1.200.000 atores puderam ser rodados ao mesmo tempo nos sistemas que suportam 5.000 *threads* ativas da máquina virtual.
- Cada *thread* da máquina virtual é considerada como um ator. Isso faz com que as *threads* se integrem bem ao ambiente de atores.

Outra funcionalidade da linguagem que teve importância fundamental na criação da biblioteca de atores Scala foi o casamento de padrões. Conforme veremos na Seção 3.4.1, o reconhecimento das mensagens recebidas pelos atores depende totalmente dessa funcionalidade.

3.4.1 Casamento de Padrões

Para construir uma calculadora com números inteiros e operação de soma utilizando linguagens orientadas a objetos costuma-se fazer algo parecido com o mostrado na figura 3.7. Essa modelagem torna a adição de novos métodos cansativa e sujeita a erros, já que faz com que todas as classes sejam mudadas ou herdadas. Um dos problemas que se tem com essa abordagem é que a implementação está espalhada pelo código o que dificulta o entendimento e a correção. Nas linguagens funcionais o que se tem é uma separação dos tipos de dados algébricos e das operações realizadas, que são realizadas de acordo com padrão encontrado.

Conforme descrito em [27], Scala fornece um mecanismo que simplifica a criação dos dados estruturados. Através do modificador `case`, é criado automaticamente um método idêntico ao construtor da classe que está utilizando o modificador. Além disso, Scala possui um esquema de casamento de padrões (*pattern matching*) que utiliza os métodos que são gerados automaticamente através do `case`.

O casamento de padrões é feito através da expressão `x match { case padrao1 => e1 case padrao2 => e2 ... }`. Essa expressão verifica se x coincide com o `padrao1` ou com o `padrao2` e assim sucessivamente. Para coincidir com determinado padrão, x precisa ser uma

```
abstract class Termo {
   def calcular: int
}

class Numero(x: int) extends Termo {
   def calcular: int = x
}

class Soma(esquerda: Termo, direita: Termo) extends Termo {
   def calcular: int = esquerda.calcular + direita.calcular
}
```

Figura 3.7: Calculadora com operação de soma seguindo a modelagem orientada a objetos.

instância da classe gerada a partir do `case` daquela classe. Caso o `padraoi` coincida, o respectivo lado direito da expressão (`ei`) é calculado. O exemplo da figura 3.8 ilustra como pode ser implementada uma calculadora com casamento de padrões:

```
abstract class Termo
case class Numero(x: int) extends Termo
case class Adicionar(esquerda: Termo, direita: Termo) extends Termo

object Interpretador {
   def avaliar(termo: Termo): int = termo match {
      case Numero(x) => x
      case Adicionar(esquerda, direita) => eval(esquerda) + eval(direita)
   }
}
```

Figura 3.8: Calculadora com operação de soma seguindo a modelagem de casamento de padrões.

No contexto dos atores fica fácil entender como esse mecanismo é útil. Como o modelo é baseado em troca de mensagens, o casamento de padrões permite que sejam determinadas quais mensagens serão reconhecidas pelo ator e o que ele irá fazer após o reconhecimento delas. Veja um exemplo na figura 3.9.

3.5 Atores em Scala

Já sabemos que o modelo de atores adotado em Scala utiliza troca de mensagens como forma de comunicação. Agora iremos conhecer um pouco melhor o funcionamento e utilização desse modelo.

```
case object Ping
case object Stop

class Pong extends Actor {
   def act() {
      var pongCount = 0
      while (true) {
         receive {
            case Ping =>
               if (pongCount % 1000 == 0)
                  Console.println("Pong: ping "+pongCount)
               pongCount = pongCount + 1
            case Stop =>
               Console.println("Pong: stop")
               exit()
         }
      }
   }
}
```

Figura 3.9: Ator Scala com casamento de padrões.

Uma mensagem é enviada através da expressão `dest ! msg`, onde `dest` representa o ator que receberá a mensagem `msg`. A operação de envio de mensagem é assíncrona, o que permite que o ator continue em execução normalmente após enviar as mensagens. Cada ator possui uma caixa de mensagens (*mailbox*) que recebe todas as mensagens enviadas por outros atores. Além de colocar uma mensagem na caixa do destinatário, a operação de envio (!) acorda o ator que está recebendo a mensagem caso este esteja em estado de espera. No exemplo da figura 3.9 vimos o mecanismo para retirar as mensagens da *mailbox*. Através do casamento de padrões, a primeira mensagem que coincidir com um dos padrões (percorridos na ordem determinada) é retirada. Caso uma mensagem não seja reconhecida, ela forma uma espécie de lixo na caixa. Por isso é comum escrever um `case` padrão para reconhecer qualquer outra forma que não interesse, simplesmente para não deixar mensagens na caixa [19, 31].

Há ainda outras construções que facilitam a troca de mensagens. Por exemplo, `dest !? msg` que envia `msg` para o ator `dest`, aguarda a resposta e a devolve como resultado. Outros exemplos são `sender, reply(msg)` e `dest forward msg`. O primeiro exemplo serve para referir ao ator que enviou a última mensagem ao objeto `self`. O segundo é utilizado quando se deseja mandar uma mensagem `msg` de volta para `sender`. Já o último manda uma mensagem `msg` para `dest` usando `sender` como ator remetente ao invés de `self` [19].

O modelo de atores adotado em Scala é uma unificação do modelo baseado em *threads* e de um modelo baseado em eventos sem inversão de controle. Através dessa unificação os

programadores podem fazer uma troca entre a eficiência e flexibilidade. O primeiro modelo é geralmente mais fácil de usar, porém menos eficiente, enquanto que o segundo é mais eficiente, mas difícil de aplicar em grandes projetos. Para entender como foi feita essa integração dos modelos, vamos primeiro entendê-los melhor separadamente.

Para implementar os atores baseados em *threads*, bastou fazer um mapeamento de cada ator para sua *thread*, considerando que o ambiente possui um modelo de *threads*. Assim, as operações das *threads* passam a ser equivalentes nos atores, ou seja, para fazer com que os atores voltem a ser executados ou entrem em estado de espera é preciso executar a *thread* ou fazer com que ela entre em estado de espera. Em Scala, para fazer esse mapeamento na JVM, a opção escolhida foi criar uma subclasse de `Thread` [19].

A comunicação nesse modelo é trivial. No processo de envio, a mensagem fica armazenada na caixa de mensagens do destinatário. Caso ele esteja em estado de espera e possa tratar a mensagem recebida, ele volta a ser executado. No recebimento, as mensagens da caixa são percorridas em ordem de chegada. Caso encontre alguma que ele possa tratar, o ator aplica uma função sobre aquela mensagem e devolve o resultado. Se não encontrar alguma mensagem que possa ser tratada, entra em estado de espera.

Ao invés de utilizar *threads* bloqueantes para representar a espera por mensagens, o modelo baseado em eventos utiliza o conceito de fechamentos. Um ator que aguarda num `receive` não corresponde a uma *thread* bloqueada e sim, a um fechamento que representa a "continuação" do trabalho do ator. O fechamento é executado assim que uma mensagem chega à caixa do ator e coincide com um dos padrões determinados na operação `receive`. Se o fechamento termina de ser executado, o controle volta para o processo que enviou a mensagem. Entretanto, se a execução do fechamento parar em outro `receive`, uma exceção é lançada. Essa exceção é tratada pelo método de envio de mensagens (!) e o controle volta para o processo que enviou a mensagem [18].

A implementação desse modelo foi feita de tal modo que o `receive` nunca retornasse normalmente. Essa necessidade surgiu do fato de haver a possibilidade de um ator entrar em estado de espera em um `receive` quando executar uma continuação através do recebimento de uma mensagem. Caso isso ocorra, o ator que mandou a mensagem também seria bloqueado porque o controle ainda não teria voltado para ele. Isso iria contra a ideia de que o envio de mensagens é assíncrono. A solução foi justamente guardar o resto da computação do `receive` e lançar uma exceção. Porém, num caso como o da figura 3.10, essa solução não é suficiente.

Nesse caso, seria necessário guardar informações sobre o que vem após o `receive` e não apenas o que está dentro dele. Isso não é possível, pois a JVM não oferece operações de gerenciamento de pilha de execução. Para evitar isso, o método `receive` devolve o tipo

```
val x = receive { case y => f(y) }
g(x)
```

Figura 3.10: Exemplo em que não basta guardar o resto da computação do `receive` e lançar uma exceção para se ter mensagens assíncronas.

`Nothing` em sua declaração e lança uma exceção que faz com que qualquer instrução que venha depois dela nunca seja executada. Então, o exemplo da figura 3.10 poderia ser reescrito como na figura 3.11.

```
receive { case y => x = f(y); g(x) }
```

Figura 3.11: Solução encontrada para que o envio das mensagens seja assíncrono.

Devido à mudança semântica, a nova versão da operação `receive` passou a ser chamada de `react` [18, 19].

A unificação dos dois modelos discutidos permite a utilização de operações bloqueantes, típicas do modelo baseado em *threads*, e ao mesmo tempo fornece atores leves e escaláveis.

Haller e Odersky [19] descrevem o funcionamento do modelo unificado, que é baseado em uma coleção de *worker threads*. À medida que uma nova tarefa é criada, uma *worker thread* que estiver disponível nessa coleção fica responsável pela execução daquela tarefa. As tarefas são criadas em três situações:

- Quando se cria um ator usando a construção `actor { corpo }`, é gerada uma tarefa que processa `corpo`.
- Quando é feita uma chamada para `react`, onde a mensagem recebida pode ser tratada imediatamente pelo ator, é gerada uma tarefa que processa a mensagem.
- Quando uma mensagem faz com que um ator em estado de espera em um `react` volte a ser executado, é gerada uma tarefa que processa tal mensagem.

Sendo assim, caso os atores usem apenas as operações típicas do modelo baseado em eventos, não haverá problemas com o número de *threads* disponíveis, pois a implementação garante que a *worker thread* seja liberada para uso de outro ator após a execução. Por outro lado, se forem operações bloqueantes como `receive` ou entrada e saída de sistema, a quantidade de *worker threads* na coleção pode acabar. Portanto, seria necessário expandir a coleção para terminar de executar as tarefas que estiverem pendentes.

Para evitar essa situação onde todas as *worker threads* fiquem ocupadas, e bloqueadas, e ainda existam tarefas pendentes, foi feito um mecanismo para aumentar a coleção. Um regulador de *threads* verifica se todas as *threads* estão sem atividade a um determinado tempo e se ainda existe alguma tarefa na fila. Caso isso ocorra, uma nova *worker thread* é criada, aumentando a coleção de *threads*, para executar a próxima tarefa e tentar fazer com que as outras tarefas terminem.

Ainda de acordo com [19], não existe uma forma segura de verificar, através de código disponível na JVM, se uma *thread* está bloqueada. Portanto, a biblioteca de atores implementa uma heurística conservadora que verifica se a marcação do tempo (*time-stamp*) da última atividade é recente. Essa marcação do tempo é comparada com um valor global que é atualizado a cada utilização da biblioteca (cada envio, recebimento,...).

Na figura 3.12 temos o mesmo exemplo da figura 3.9, agora usando `react`:

```
case object Ping
case object Stop

class Pong extends Actor {
   def act() {
      var pongCount = 0
      loop {
         react {
            case Ping =>
               if (pongCount % 1000 == 0)
                  Console.println("Pong: ping "+pongCount)
               pongCount = pongCount + 1
            case Stop =>
               Console.println("Pong: stop")
               exit()
         }
      }
   }
}
```

Figura 3.12: Ator em Scala usando `react`.

4 AIRS

Desde que começaram a ser estudados, os sistemas imunológicos artificiais (*Artificial Immune Systems (AIS)* [3]) eram utilizados para determinar características semelhantes entre os dados analisados (*clustering*) ou para identificação de algum tipo de anomalia, como invasão em um sistema. Ultimamente os AIS passaram a ser utilizados em outras áreas também, como em otimização de funções e classificação, que é o caso do AIRS (*Artificial Immune Recognition System*) [36].

Os AIS se baseiam no funcionamento do sistema imunológico dos animais e na tentativa de imitá-lo para poder resolver problemas complexos. Nessa tradução do sistema imunológico natural para o artificial, muitos elementos são desconsiderados. Na verdade, são levadas em conta apenas algumas ideias que se mostram eficientes no mundo real. Assim, espera-se que tais ideias também sejam eficientes quando utilizadas nos algoritmos. Essas expectativas têm sido atingidas, como mostrado na Seção 2.2 e como mostraremos no caso do AIRS.

4.1 Características

Um algoritmo de classificação tem por objetivo prever se uma instância de determinado objeto pertence ou não a uma classe específica. Caso seja supervisionado, ele precisará ser, primeiramente, treinado com algumas instâncias conhecidas. Esse treinamento irá servir para que o algoritmo aprenda quais características um objeto deve ter para que ele pertença, ou não, à classe sendo estudada. Após o treinamento, é possível utilizar o algoritmo com essa base de treinamento para fazer a predição de novas instâncias.

Sendo um algoritmo de classificação supervisionado, o AIRS também segue esse modelo. Antígenos são usados durante uma fase de treinamento e, de acordo com a sua afinidade com os anticorpos, esses anticorpos passam a formar um conjunto especial (células de memória). Afinidade é uma medida de semelhança entre dois elementos que estão sendo comparados. No caso, dois anticorpos ou antígenos. Seu valor fica no intervalo [0,1] e é obtido através da distância Euclideana. Ao final do processo, esse conjunto de células pode ser usado para avaliar a que categoria novos antígenos provavelmente pertencem.

No contexto do nosso trabalho, anticorpos e antígenos possuem a mesma representação. Ambos são vetores de características do elemento que queremos modelar e que estão relacionados com suas classes (categoria a qual o vetor pertence). A diferença é que tais vetores são chamados de anticorpos quando fazem parte de ARBs (*Artificial Recognition Balls*) ou de células de memória (ver Seção 4.2). Por outro lado, são chamados de antígenos quando estão sendo apresentados para as ARBs para estimulação. Um vetor de características é um sequência de valores onde cada posição representa uma característica diferente e, portanto, possui um intervalo de valores próprio.

Por ser criado com o objetivo específico de ser um algoritmo de classificação, o AIRS apresenta algumas características importantes citadas em [7,36]. Durante o treinamento, o próprio algoritmo aprende a arquitetura que será usada. Assim, não é necessário que o usuário indique a topologia, fato comum em outros algoritmos. Outra característica apresentada pelo AIRS é que ele se mostrou bem competitivo quando comparado a alguns dos melhores classificadores ficando entre os 8 melhores na maioria das bases utilizadas [39]. Para fazer a classificação de novos antígenos após o término do treinamento, o AIRS utiliza um número reduzido de dados, ao invés de todo o conjunto de dados do treinamento, sem perder informações relevantes. Essa redução dos dados é uma das características principais do algoritmo, já que diminui o número de dados com os quais os cálculos serão feitos diminuindo o tempo total gasto. Uma última característica que vale a pena citar é a grande quantidade de parâmetros que o AIRS possui. Através deles é possível que o usuário customize os atributos do algoritmo da melhor maneira possível de modo que os resultados sejam melhores.

4.2 AIRS1

Aproveitando estudos feitos por outros pesquisadores, Watkins e Boggess elaboraram em 2001 um sistema imunológico artificial que chamaram de AIRS, conhecido como AIRS1 [36]. Uma das principais características é que esse classificador é baseado em sistemas com recursos limitados. Isso quer dizer que, ao contrário de outros classificadores da época, o AIRS1 levava em conta o número de células produzidas e tentou, desde o começo, controlar essa produção.

O funcionamento do algoritmo é simples. Ele é dividido em quatro fases, sendo que a primeira funciona como um pré-processamento, ou seja, é executada apenas uma vez no início do algoritmo. As outras três fases são executadas em ciclo até que uma condição de parada seja atingida. Nessa seção iremos apenas comentar as etapas da primeira versão do algoritmo (AIRS1), sem entrar em detalhes. Deixaremos para explicar melhor o funcionamento do algoritmo quando formos falar sobre o AIRS2, pois o AIRS1 hoje já é considerado obsoleto mesmo

tendo se mostrado eficiente quando comparado a bons classificadores. Entretanto, algumas modificações no algoritmo fizeram com que o AIRS2 fosse a versão padrão.

O pré-processamento é constituído pela normalização dos vetores de características. Essa normalização é feita para que as distâncias entre antígenos e ARBs ou entre duas ARBs fiquem no intervalo [0,1]. Uma ARB, também conhecida como B-célula (*B-cell*), é uma possível solução e representa um anticorpo. Também ocorre nessa etapa a inicialização de alguns valores. Após o pré-processamento, os outros três passos se repetem enquanto existirem antígenos. Primeiro, são geradas ARBs a partir de uma população de células de memória levando-se em conta a classe do antígeno em treinamento e as classes das células de memória e também as afinidades entre eles. Em seguida, ocorre uma competição pelos recursos disponíveis entre as ARBs geradas e aquelas que possuírem uma maior estimulação em relação ao antígeno tornam-se candidatas a células de memória. Finalmente, algumas candidatas são promovidas a células de memória que servirão como classificadoras para antígenos ainda não expostos ao algoritmo. Portanto, célula de memória nada mais é do que uma ARB que possui maior estimulação em relação a um dado antígeno.

Com essa breve descrição do algoritmo, podemos notar que o AIRS possui algumas qualidades. Além de ser um classificador com mecanismo para aprendizado supervisionado, ele trata os recursos como sendo uma quantidade abstrata limitada no sistema e não apenas como a quantidade de B-células. Além disso, as classes dos antígenos e dos anticorpos possuem grande importância. Caso as B-células sejam da mesma classe que o antígeno, elas são recompensadas na alocação dos recursos, no momento de gerar as ARBs e durante a promoção a candidata a célula de memória.

4.3 AIRS2

As alterações feitas na primeira versão do algoritmo por Watkins, Timmis e Boggess [39] não foram tão grandes. Entretanto, simplificaram o código e apresentaram resultados satisfatórios. A medida dos resultados foi feita levando em conta as duas principais características do AIRS: a acurácia da classificação e a redução dos dados (número de células geradas). Com relação ao desempenho, observou-se que a segunda versão continuou tão boa quanto a primeira. Então, o grande motivador da criação do AIRS2 foi a redução do número de células de memória geradas durante o processo. Vale lembrar que são essas células que são usadas para classificar os antígenos, então qualquer redução na quantidade delas, sem perder informações, é valiosa.

Além disso, como a classificação utiliza uma abordagem kNN[1], que é baseada nos vizinhos mais próximos, quanto menos células existirem melhor será, pois o cálculo será mais rápido. Na Seção 4.3.1 descrevemos o algoritmo para, em seguida, vermos as principais diferenças entre as duas versões do AIRS.

4.3.1 Algoritmo

Como vimos na Seção 4.1, o AIRS é capaz de criar um número reduzido de células de memória capazes de classificarem antígenos novos, ou seja, que ainda não foram expostos ao classificador. Nesta seção iremos nos aprofundar no algoritmo. O AIRS pode ser dividido em cinco fases: inicialização, treinamento dos antígenos, competição por recursos limitados, seleção de células memórias e classificação. O diagrama da figura 4.1 mostra uma visão geral do fluxo do algoritmo, passando por essas cinco fases [7].

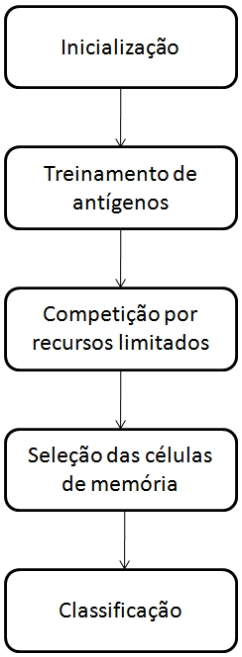


Figura 4.1: Fluxo do algoritmo do AIRS.

- Inicialização: Durante a inicialização ocorre a preparação de algumas variáveis e dos dados

[1] O primeiro passo da abordagem kNN é calcular os k vizinhos mais próximos da instância que está sendo treinada. Em seguida, cada um desses vizinhos selecionados compara seus atributos com os atributos da instância em treinamento para saber se ela pertence, ou não, à mesma classe que ele. Feita essa comparação, passa-se para a etapa final, que é a classificação propriamente dita. A classe que tiver tido maior votação será a classe da instância em treinamento.

que serão usados durante o treinamento. Para garantir que os valores das distâncias entre duas ARBs ou entre uma ARB e um antígeno estejam sempre no intervalo [0,1] é calculada a variável afinidade, definida como:

$$affinity = \frac{dist}{maxDist} = \frac{\sqrt{\sum_{i=1}^{n}(u_i - v_i)^2}}{\sqrt{\sum_{i=1}^{n}(r_i^2)}} \tag{4.1}$$

No cálculo da afinidade, *dist* representa a distância Euclideana entre dois elementos cuja afinidade queremos calcular. No cálculo da distância, n representa o número de atributos usados, já u e v são dois elementos cuja afinidade queremos encontrar. O denominador, *maxDist*, representa a distância máxima entre dois vetores de dados, onde r_i é o comprimento do intervalo admissível para o atributo i. Outro passo que ocorre durante a inicialização, que é optativo, é a escolha aleatória de antígenos para começarem como sementes do reservatório de células de memória. O último passo é o cálculo do limiar de afinidade (*affinityThreshold, ou AT*). Ele é obtido através do valor médio da afinidade entre os antígenos de um subconjunto do conjunto de treinamento ou de todo o conjunto de treinamento (coleção de antígenos).

- Treinamento dos antígenos: Nessa etapa, cada antígeno é exposto às células que estão no reservatório de células de memória. Assim, para cada uma é possível calcular um valor de estimulação em relação àquele antígeno da seguinte maneira:

$$stim = 1 - affinity \tag{4.2}$$

Para cada antígeno, a célula que tiver maior valor de estimulação é selecionada para o processo de maturação de afinidade. Nesse processo, as células selecionadas sofrem clonagem e mutações. Dentre os clones modificados apenas alguns serão selecionados, levando-se em conta suas classes e suas afinidades em relação ao antígeno em treinamento, para serem colocados no reservatório de ARBs. Nessa etapa, a quantidade de clones modificados que pode ser gerada é calculada através da seguinte fórmula:

$$numClones = stim * clonalRate * hypermutationRate \tag{4.3}$$

onde *stim* é a estimulação entre a melhor célula de memória e o antígeno. O parâmetro *clonalRate* é um inteiro definido pelo usuário para determinar o número de clones modificados que uma ARB pode tentar produzir. O parâmetro *hypermutationRate* é um

número inteiro definido pelo usuário para determinar o número de clones modificados que uma célula de memória pode produzir.

- Competição por recursos limitados: Depois de adicionar alguns clones modificados ao reservatório de ARBs, começa a geração de ARBs e a competição entre as ARBs geradas.

  Usada para controlar o tamanho do reservatório de ARBs e promover aquelas ARBs com maior estimulação (portanto afinidade) ao antígeno em treinamento, a competição por recursos limitados para somente quando um critério de parada é atingido. A condição de parada desse processo de refinamento das ARBs ocorre quando a média da estimulação normalizada das ARBs de cada classe é maior que o limiar de estimulação. Esse limiar é um parâmetro definido pelo usuário cujos valores ficam no intervalo [0,1]. Caso tal critério não seja atingido, novas ARBs são geradas da mesma maneira que foram criados os clones da célula original. O número de clones criados nessa etapa, caso a condição de parada não seja atingida, é calculado da seguinte maneira:

$$numClones = stim * clonalRate \tag{4.4}$$

  Em seguida, ocorre uma nova competição entre as ARBs pelos recursos e o ciclo se completa. Durante a competição por recursos, há uma alocação de recursos às ARBs, onde a quantidade de recursos alocados para cada ARB é proporcional à sua estimulação, conforme mostrado na equação abaixo:

$$resource = normStim * clonalRate \tag{4.5}$$

  onde *normStim* é o valor de estimulação normalizado. *clonalRate* é o mesmo parâmetro utilizado no cálculo do número de clones que podem ser produzidos.

  O algoritmo realiza uma verificação para garantir que o total de recursos alocados às ARBs durante a competição pelos recursos seja menor que o limite máximo determinado pelo usuário através do parâmetro *total resources*. Em primeiro lugar, as ARBs são ordenadas de forma descendente de acordo com as suas afinidades em relação ao antígeno. Caso o total alocado seja maior que o permitido, são retirados recursos das ARBs do fim da lista ordenada até que se tenha um total de recursos alocados menor que o permitido. Em seguida, aquelas ARBs que não possuírem recursos alocados também são removidas do reservatório.

- Seleção das células de memória: Após terminar a fase de competição por recursos limita-

dos, a ARB que tiver uma maior afinidade em relação ao antígeno será selecionada como candidata a célula de memória. Caso o valor de estimulação da candidata seja maior do que o da célula que originou os clones, a ARB é copiada para o reservatório de células de memória. Além disso, caso a afinidade entre a candidata a célula de memória e a célula original seja menor que um valor de corte, então a célula original é removida do reservatório de células de memória. Esse valor de corte é definido da seguinte maneira:

$$cutOff = affinityThreshold * affinityThresholdScalar \tag{4.6}$$

Na equação 4.6, o *affinityThreshold* é a variável preparada na inicialização e a magnitude do limiar de afinidade (*affinityThresholdScalar, ou ATS*) é um parâmetro definido pelo usuário que varia no intervalo [0,1] e serve para limitar o valor de corte.

- Classificação: Quando o processo de treinamento acaba, o classificador está pronto para ser usado. Os vetores de dados (ARBs) que ficam no reservatório de células de memória serão utilizados para classificar novos antígenos. Esses vetores podem ser utilizados como estão ao final do treinamento ou seus valores podem ser denormalizados antes da classificação. A classificação utiliza a técnica *kNN* onde, após selecionar as k células mais próximas do antígeno não classificado, a classe é determinada por voto majoritário.

4.3.2 AIRS1 x AIRS2

Apesar de não termos descrito o algoritmo do AIRS1 em detalhes, com base no AIRS2 podemos identificar algumas diferenças [39] entre essas versões. Tais mudanças foram responsáveis pela melhora nos resultados de redução do número de células geradas e por manter o desempenho do AIRS1.

- As classes das ARBs afetam diretamente a alocação dos recursos. As ARBs que possuírem maior afinidade em relação ao antígeno em treinamento (e que forem da mesma classe) receberão mais recursos.

- Não há mais a preocupação em se manter a diversidade de classes dentro do reservatório de ARBs. Pode-se, portanto, semear apenas o reservatório de células de memória durante a inicialização, ao invés de semear tanto o reservatório de células de memória quanto o reservatório de ARBs com exemplos de várias classes.

- Enquanto que o AIRS1 permitia que as ARBs mudassem de classe durante a clonagem para popular o reservatório de ARBs, o AIRS2 não oferece tal possibilidade, uma vez

que apenas os clones da mesma classe que o antígeno são levados em conta.

- Durante a maturação da afinidade (mutação), o nível de estimulação da célula é levado em consideração. A quantidade de mutações de um clone é proporcional à sua afinidade em relação ao antígeno.
- Para o cálculo do critério de parada do treinamento, o AIRS2 considera apenas os valores da estimulação das ARBs da mesma classe que o antígeno para o cálculo do critério de parada do treinamento. O AIRS1, por outro lado, considerava os valores das ARBs de todas as classes.

4.4 AIRS2 Paralelo

Baseando-se no caso da implementação do CLONALG paralelo, onde foi demonstrado que é possível aplicar técnicas de paralelização em algoritmos AIS, Watkins e Timmis paralelizaram o AIRS [38].

A ideia geral do algoritmo paralelo é a seguinte:

1. Um processo central é responsável pela leitura dos dados.
2. Esses dados são divididos entre múltiplos processos.
3. Em cada subprocesso é executado o algoritmo sequencial:
   a) Compare uma instância em treinamento (antígeno) com todas as células (anticorpos) da mesma classe e encontre a célula de memória que possuir maior afinidade com essa instância. Chamaremos essa célula de memória de cm_{match}.
   b) Clone e modifique a célula cm_{match} proporcionalmente à sua afinidade a fim de criar um reservatório de B-células.
   c) Calcule a afinidade de cada B-célula com a instância em treinamento.
   d) Aloque recursos para cada B-célula de acordo com a sua afinidade.
   e) Remova as B-células com menor afinidade até que o número de recursos atinja o limite estipulado.
   f) Caso a afinidade média das B-células restantes esteja acima de certo limite, vá ao passo 'g'. Caso contrário, clone e modifique as B-células restantes de acordo com as suas afinidades e retorne ao passo 'c'.
   g) Escolha a melhor B-célula como uma candidata a célula de memória (cm_{cand}).

h) Caso a afinidade da célula candidata (cm_{cand}) em relação à instância em treinamento seja maior do que a afinidade da (cm_{match}) em relação à essa instância, então adicione (cm_{cand}) ao reservatório de células de memória. Além disso, se a afinidade entre (cm_{cand}) e (cm_{match}) estiver dentro de um certo limite, então remova (cm_{match}) do reservatório de células de memória.

i) Repita o processo desde o passo 'a' até que todas as instâncias em treinamento tiverem sido testadas.

4. As células geradas em cada processo são devolvidas ao processo principal.

5. O processo principal une as células de memória obtidas de cada processo em um reservatório para serem usadas como classificador.

Apesar do tempo total de execução do algoritmo ter diminuído devido à paralelização, o número de células de memória ao final do processo principal cresceu em relação à versão serial. Esse crescimento ocorreu quando foi utilizada uma abordagem simples para agrupar as células de memória criadas pelos processos. Tal abordagem, denominada concatenação, simplesmente reúne os conjuntos de células gerados pelos processos para formar um único conjunto de células de memória no processo principal. O motivo pelo qual o número de células cresceu foi o fato de que cada processo criado calcula o seu próprio conjunto de células de memória. Assim, supondo que cada processo tenha uma instância de cada classe, o número mínimo de células de memória em cada processo será o número de classes (nc), ou seja, uma célula de memória para cada classe. Se considerarmos todos os processos, a quantidade mínima de células de memória será $nc * np$, onde np representa a quantidade de processos criados. Por outro lado, na versão serial são necessárias, no mínimo, apenas nc células de memória no classificador final. Como a redução do número de células de memória era uma das melhores características do AIRS serial, foram estudadas novas técnicas para pelo menos manter essa redução.

Uma das tentativas para reduzir o número final de células de memória foi aplicar uma técnica baseada na afinidade. Depois de reunir as células geradas pelos diversos processos, elas são separadas pelas classes. Em seguida, é calculada a afinidade entre as células de cada classe. Se a afinidade entre duas células for menor do que o limiar de afinidade (AT) vezes a magnitude do limiar de afinidade (ATS), então uma das células é excluída.

Como as abordagens baseadas na concatenação e na afinidade não apresentaram bons resultados, Watkins e Timmis fizeram uma nova tentativa fazendo uma modificação na versão baseada na afinidade. Eles repararam que o crescimento das células era logarítmico em relação ao crescimento da quantidade de processadores utilizados. Então, decidiram adicionar um fator de correção logarítmico em função do número de processadores. O resultado obtido

com essa técnica foi inconclusivo. Houve teste em que o número de células geradas diminuiu. Entretanto, houve situação em que o número permaneceu semelhante e, até mesmo, cresceu.

Resumindo, a versão paralela do AIRS melhorou os tempos de execução do algoritmo em relação à versão serial. Por outro lado, ela produz mais células de memória do que a versão serial. As abordagens vistas em [38] não solucionaram esse problema, mas, assim como o sistema imunológico serviu de base para o AIRS e outros algoritmos, a solução para o controle da geração de células de memória pode estar nesse mesmo sistema.

5 A implementação de uma versão paralela do AIRS em Scala

Durante nosso estudo implementamos o algoritmo AIRS2 paralelo em Scala com o objetivo de analisar o aproveitamento dos núcleos em máquinas *multicore* através de programas desenvolvidos em Scala e de fazer uma avaliação qualitativa da facilidade que o programador passa a ter quando desenvolve programas concorrentes utilizando o paradigma de troca de mensagens adotado por essa linguagem.

Os seguintes motivos nos levaram a escolher o AIRS para trabalhar:

- A aplicabilidade do AIRS a problemas importantes de mineração de dados e, em particular, ao problema da detecção de fraudes em transações com cartões de crédito [16].
- A existência de uma implementação *multi-threaded* do AIRS em Java [41], com a qual poderemos fazer comparações de desempenho.
- A experiência com o AIRS já existente em nosso grupo de pesquisa e o interesse do grupo por esse algoritmo.

A implementação do AIRS em Scala que desenvolvemos foi baseada na versão em Java, que pode ser obtida no site *http://wekaclassalgos.sourceforge.net*. A versão em Java já estava integrada à bancada de ferramentas de aprendizado computacional WEKA. Com o nosso projeto fizemos com que a nossa versão também fosse capaz de ser executada na WEKA. Essa integração facilitou tanto a realização de experimentos comparativos entre as versões Java e Scala do algoritmo quanto permitiu o eventual uso do classificador por outras pessoas. Isso foi possível devido à interoperabilidade entre Scala e Java, uma vez que a WEKA é desenvolvida em Java.

5.1 WEKA

Desenvolvida na Universidade de Waikato na Nova Zelândia, a WEKA [40] (*Waikato Environment for Knowledge Analysis*) é uma bancada de ferramentas de aprendizado computa-

cional para mineração de dados. Criada com o intuito de processar dados obtidos a partir da agricultura neozelandesa, a WEKA está disponível como software livre. Com o passar do tempo e com o aumento de sua utilização percebeu-se que era possível empregá-la em contextos mais amplos. Hoje em dia, ela é utilizada em várias áreas, como bioinformática [15].

A WEKA possui ferramentas para preprocessamento, classificação, regressão, agrupamento, regras de associação e visualização [7]. É possível utilizar essas ferramentas através de uma interface gráfica ou então programaticamente, fazendo acesso à API da WEKA por meio de código Java. A interface possui três componentes principais:

- Explorer: Um ambiente para explorar os dados
- Experimenter: Um ambiente para realizar experimentos e comparar diferentes algoritmos de aprendizado.
- KnowledgeFlow: Um ambiente similar ao Explorer, mas que suporta *drag-and-drop* para colocar em execução o processo de aprendizado.

Na figura 5.1 mostramos a tela inicial da WEKA. Já na figura 5.2 é possível ver o ambiente *Explorer* com o AIRS paralelo (em Java) selecionado como algoritmo classificador.

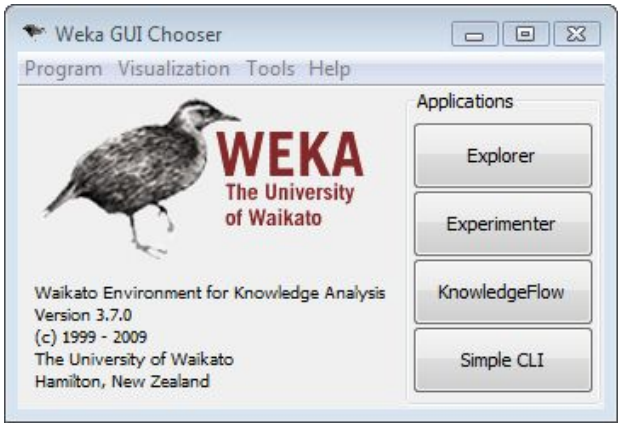


Figura 5.1: Tela inicial da WEKA.

Mais informações sobre como utilizar a ferramenta com os vários algoritmos, inclusive com o AIRS, podem ser encontradas em [40, 41].

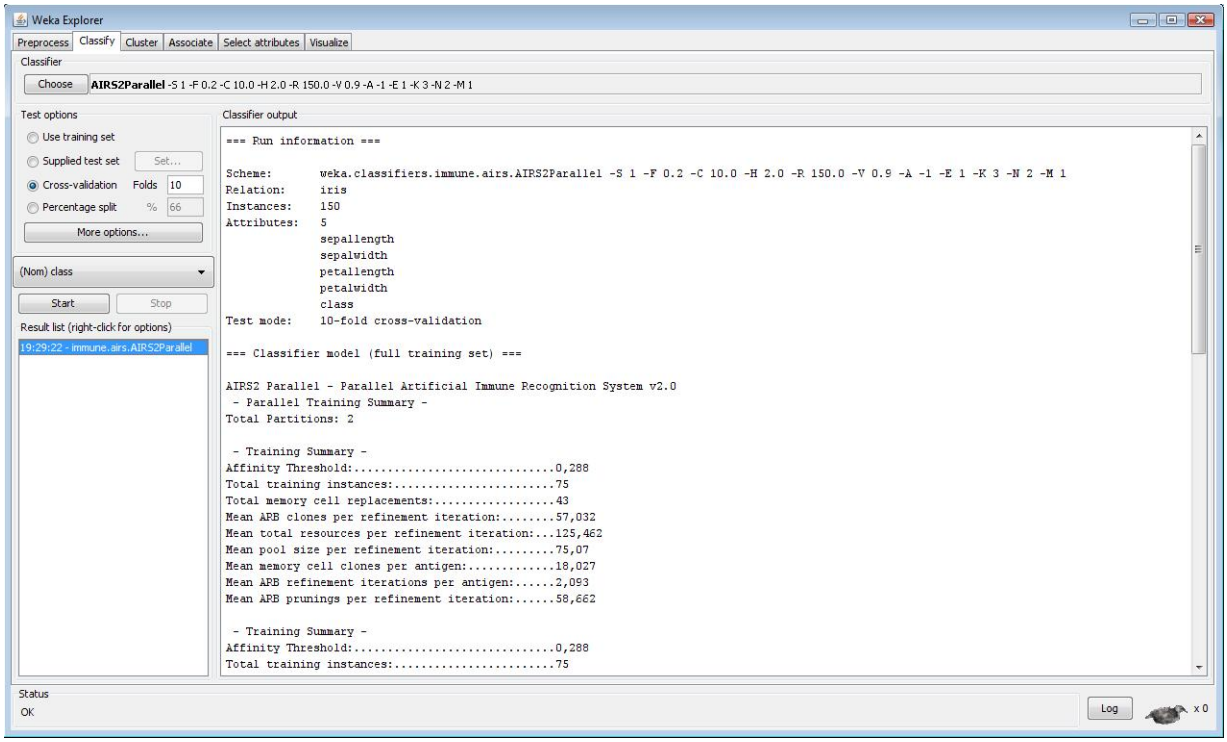


Figura 5.2: WEKA Explorer com o algoritmo AIRS paralelo selecionado.

5.2 Implementação

Assim como é possível selecionar o AIRS2 paralelo em Java para classificar dados na WEKA, fizemos com que fosse possível fazer o mesmo com a nossa implementação desse algoritmo em Scala. Como todo o restante da WEKA (interfaces e algoritmos) é desenvolvido em Java, a integração do AIRS2 paralelo em Scala ao ambiente da WEKA foi possibilitada pela inter-operabilidade entre as linguagens, permitindo inclusive comparações de desempenho entre as versões do algoritmo.

O nosso trabalho possuiu as seguintes etapas:

1. Identificar as classes Java relativas ao algoritmo AIRS paralelo. Dado que o algoritmo que utilizamos como base já estava integrado a WEKA, primeiramente tivemos que identificar as classes do algoritmo para que escrevêssemos apenas essas classes em Scala.

2. Escrever as classes identificadas em Scala. Após identificarmos as classes relativas ao algoritmo, reescrevemos essas classes em Scala. Tomamos cuidado para manter fidelidade ao código original, ou seja, tentamos não melhorar o algoritmo de nenhuma forma.

Tentamos também não depender de classes escritas em Java.

3. Integrar com a WEKA. Após reescrever o algoritmo em Scala, fizemos com que fosse possível utilizá-lo na WEKA como qualquer outro classificador.

5.2.1 Identificação das classes do AIRS paralelo

A primeira etapa serviu para definirmos o escopo do nosso projeto dentro da WEKA. Definidas as classes do algoritmo, pudemos nos concentrar apenas na reescrita delas. As outras classes puderam ser acessadas normalmente como mostrado no Capítulo 3, mesmo estando em Java.

Listamos abaixo as principais classes relativas ao AIRS paralelo:

- `weka.classifiers.immune.airs.algorithm.merge.ConcatenateMerge`:
  Classe responsável pela concatenação dos dados calculados nos múltiplos processos criados.

- `weka.classifiers.immune.airs.algorithm.merge.PruneMerge`: Classe responsável pela concatenação dos dados calculados nos múltiplos processos criados.

- `weka.classifiers.immune.airs.algorithm.AIRS2ParallelTrainer`:
  Classe responsável pelo treinamento dos dados (normalizar os dados, calcular o limiar de afinidade, criar processos, juntar resultados). Também é responsável pela geração do classificador.

- `weka.classifiers.immune.airs.algorithm.AIRS2Trainer`:
  Classe responsável pelo treinamento no algoritmo AIRS serial. Cada processo criado na versão paralela utiliza essa classe para realizar o treinamento de maneira serial no conjunto de dados designado àquele processo.

- `weka.classifiers.immune.airs.algorithm.AIRSProcess`: Classe que implementa um processo (*thread*). Executa as operações do AIRS serial.

- `weka.classifiers.immune.airs.algorithm.MemoryCellMerger`: Interface que define o método para unir as células dos processos (ver `weka.classifiers.immune.airs.algorithm.merge.ConcatenateMerge` e `weka.classifiers.immune.airs.algorithm.merge.PruneMerge`).

- `weka.classifiers.immune.airs.AIRS2Parallel`: Classe principal do algoritmo. Inicia a classe responsável pelo treinamento e obtém os resultados.

- `weka.classifiers.immune.airs.AIRSParameterDocumentation`: Classe responsável pela documentação dos parâmetros do algoritmo AIRS.

5.2.2 Reescrita das classes em Scala

Após identificarmos as classes responsáveis pelo algoritmo AIRS paralelo, começamos a reescrever o código em Scala. Nessa etapa, apesar da sintaxe das linguagens serem diferentes, encontramos muita semelhança, o que vai de acordo com o intuito dos projetistas de Scala (ver Capítulo 3).

As *threads* de Java que eram utilizadas para executarem os múltiplos processos criados pelo algoritmo paralelo foram substituídas pelos atores de Scala. Na figura 5.3 podemos ver a versão Java desses processos. Essa figura mostra uma classe que implementa a interface `Runnable`.

O exemplo da figura 5.4 mostra o ator que na versão em Scala desempenha a mesma função da *thread* mostrada na figura 5.3.

Repare que os códigos das versões Java e Scala são bem parecidos, diferindo basicamente em sua sintaxe. A grande diferença está no modo como os processos sinalizam que terminaram a sua tarefa. No caso da versão Java, foi usada a classe `CountDownLatch` da biblioteca `java.util.concurrent` para sinalizar quando um processo havia terminado suas tarefas. Através dessa classe é possível fazer com que uma *thread* principal espere até que vários processos terminem para que, só então, ela continue sendo executada. Na versão Scala foi usado o modelo de atores baseado em troca de mensagens. Ao terminarem de executar suas tarefas, os atores enviam uma mensagem para um ator representado pelo processo principal (`waitForAllActors ! Finished`). Então, o processo principal fica esperando todos os atores enviarem suas mensagens informando que terminaram para que ele possa continuar sua execução.

5.2.3 Integração a WEKA

A última fase da implementação foi alcançada de maneira quase que automática. Após terminarmos a reescrita do código já era possível executar testes utilizando a WEKA programaticamente. Para aproveitar os recursos gráficos da ferramenta tivemos apenas que incluir o arquivo `scala-library.jar` no instante da criação do novo jar. Com isso, a utilização do algoritmo na WEKA será como a do AIRS paralelo em Java, como podemos ver na figura 5.5.

```
protected final class AIRSProcess implements Runnable
{
   protected final AIRS2Trainer algorithm;
   protected final Instances instances;
   protected final Normalize normalise;
   protected AISModelClassifier classifier;

   public AIRSProcess(
      AIRS2Trainer aAlgorithm,
      Instances aInstances,
      Normalize aNormalise)
   {
      algorithm = aAlgorithm;
      instances = aInstances;
      normalise = aNormalise;
   }

   private long startTime = Long.MIN_VALUE;
   private long totalTime = Long.MAX_VALUE;

   public void run()
   {
      startTime = System.currentTimeMillis();

      // set the affinity threshold manually
      algorithm.setAffinityThreshold(affinityThreshold);
      // run training
      try
      {
         classifier = algorithm.internalTrain(instances, normalise);
      }
      catch(Exception e)
      {
         throw new RuntimeException("Failed to prepare classifier partition." , e);
      }
      finally
      {
         // finished
         latch.countDown();
         totalTime = System.currentTimeMillis() - startTime;
      }
   }

   public long getStartTime()
   {
      return startTime;
   }

   ...
```

```
   ...

   public long getTotalTime()
   {
      return totalTime;
   }

   public LinkedList<Cell> getCells()
   {
      if(classifier != null) {
         CellPool cellPool = classifier.getModel();
         return cellPool.getCells();
      }

      return null;
   }

   public String getTrainingSummary()
   {
      return algorithm.getTrainingSummary();
   }
}
```

Figura 5.3: `Thread` Java que executa cada subprocesso do algoritmo AIRS paralelo.

```
class AIRSProcessScala
(
   aAlgorithm: AIRS2Trainer,
   aInstances: Instances,
   aNormalise: Normalize,
   aAffinityThreshold: Double,
   aWaitForAllActors: Actor) extends Actor
{
   protected val algorithm: AIRS2Trainer = aAlgorithm
   protected val instances: Instances = aInstances
   protected val normalise: Normalize = aNormalise
   protected val affinityThreshold: Double = aAffinityThreshold
   protected var classifier: AISModelClassifier = null
   val waitForAllActors = aWaitForAllActors

   private var startTime: Long = Long.MinValue
   private var totalTime: Long = Long.MaxValue

   ...
```

```
   ...

   def getCells(): LinkedList[Cell] =
   {
      var cells: LinkedList[Cell] = null;
      if(classifier != null) {
         var cellPool: CellPoolScala = classifier.getModel();
         cells = cellPool.getCells();
      }
      cells
   }

   def getTrainingSummary(): String =
   {
      algorithm.getTrainingSummary();
   }

   def getStartTime(): Long =
   {
      startTime;
   }

   def getTotalTime(): Long =
   {
      totalTime;
   }

   def act() {
      startTime = System.currentTimeMillis

      // set the affinity threshold manually
      algorithm.setAffinityThreshold(affinityThreshold);
      // run training
      try
      {
         classifier = algorithm.internalTrain(instances, normalise)
      }
      catch
      {
         case e: Exception => throw new RuntimeException("Failed to prepare"
                + " classifier partition." , e);
      }
      finally
      {
         waitForAllActors ! Finished
         totalTime = System.currentTimeMillis - startTime
      }
   }
}
```

Figura 5.4: Ator Scala que executa cada subprocesso do algoritmo AIRS paralelo.

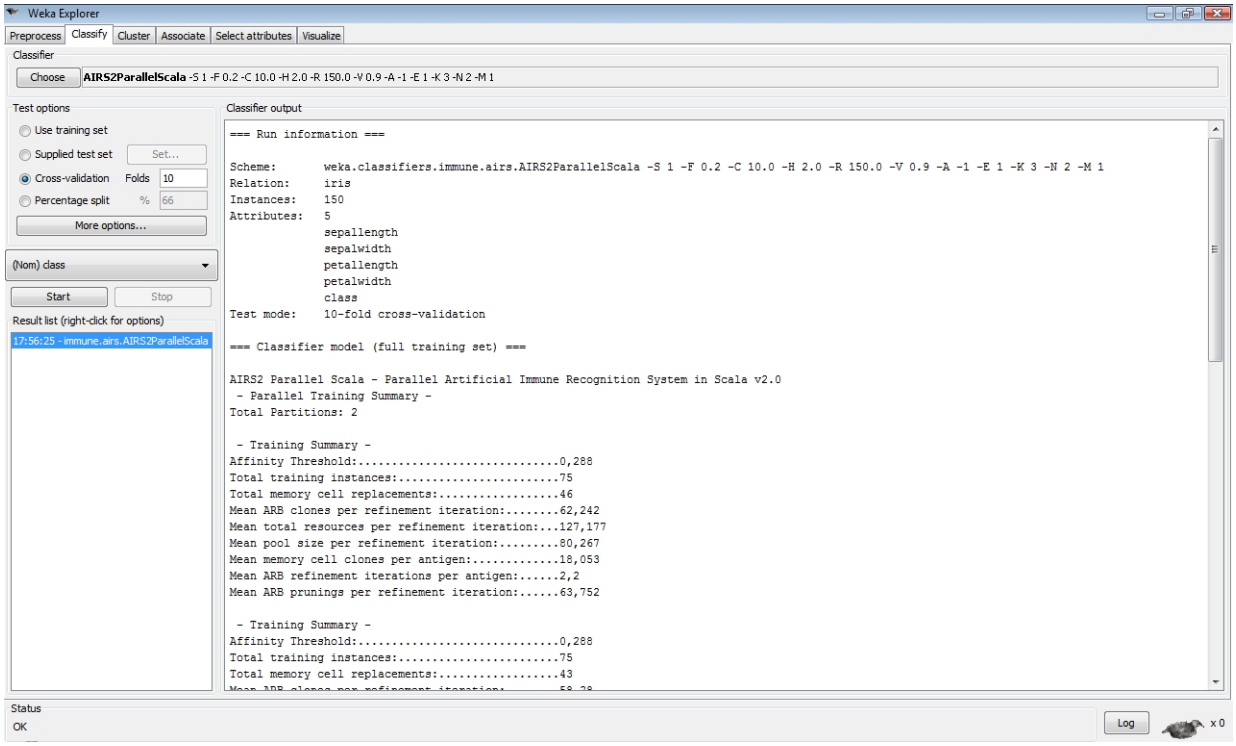


Figura 5.5: WEKA Explorer após uma execução do algoritmo AIRS paralelo em Scala.

6 Resultados experimentais

No decorrer desse capítulo apresentamos os resultados obtidos com experimentos realizados com a versão Scala do algoritmo AIRS que implementamos durante esse projeto. Dividimos o capítulo em duas seções. A Seção 6.1 faz uma comparação entre o desempenho da versão Java com o de Scala focando em critérios de concorrência. A Seção 6.2 apresenta uma avaliação do modelo baseado na troca de mensagens adotado por Scala.

6.1 Comparação de desempenho

Scala propõe um modelo de programação concorrente baseado na troca de mensagens que facilita o trabalho dos desenvolvedores. Mas esse modelo não seria útil se os sistemas desenvolvidos com ele passassem a ter um desempenho pior do que os sistemas desenvolvidos através do modelo tradicional de memória compartilhada. Então, para sabermos se a nova versão do algoritmo AIRS possuía um desempenho maior, menor ou tão bom quanto o desempenho da versão Java, comparamos:

- O tempo médio total de execução das versões.
- A aceleração entre as duas versões do algoritmo.
- A eficiência entre as duas versões do algoritmo.

Além disso, fizemos uma comparação entre as versões serial e paralela, tanto de Java quanto de Scala, para confirmarmos qual delas terminava os testes em menor tempo.

6.1.1 O ambiente de testes

Durante nossos testes utilizamos uma máquina equipada com dois processadores Intel(R) Xeon(R) Quad Core CPU E5440 de 2.83GHz totalizando oito núcleos disponíveis para nossos experimentos. O sistema operacional adotado foi o Linux versão 2.6.28-15-server.

Utilizamos a versão 6 do Sun Java SE Development Kit (JDK) [23] e a versão 2.7.6.final do plugin de Scala [33] para a IDE Eclipse [11] cuja versão utilizada foi a 3.4. Apesar do plugin

ainda possuir algumas falhas, preferimos trabalhar com ele ao invés de utilizarmos um editor de textos e o *shell* de Scala, ou ainda, o plugin [34] para o NetBeans [26]. Optamos pelo plugin do Eclipse devido ao conhecimento prévio sobre a IDE e dos recursos que poderiam facilitar o desenvolvimento. As falhas existentes não chegaram a atrapalhar pois puderam ser contornadas. O principal problema encontrado com o plugin foi o fato de ele não remover da tela as indicações visuais de erro correspondentes a erros de compilação já corrigidos, dando a ideia de que algo ainda estava errado. Tivemos também que atualizar a WEKA para a versão 3.7 por ser mais compatível com o Java 6 do que a versão da WEKA utilizada no código original obtido em *http://wekaclassalgos.sourceforge.net*.

Finalmente, retiramos alguns comandos de impressão para não interferirem na medição dos tempos e adicionamos os comandos necessários para guardar os tempos em um arquivo de saída para compararmos as versões. Também alteramos o código das versões paralelas para que elas aceitassem a chamada do algoritmo com apenas uma *thread*, já que a versão original aceitava apenas duas ou mais *threads*.

6.1.2 Metodologia

Para comparar o desempenho entre as versões do AIRS, executamos cada uma dessas versões sobre quatro bases de dados conhecidas na área de aprendizagem computacional e que foram usadas para avaliar originalmente o AIRS: *ionosphere.arff*, *iris.arff*, *sonar.arff* e *diabetes.arff* [5]. Cada experimento foi rodado 200 vezes. Apesar do tempo da primeira execução sempre ser maior que os demais, não descartamos seu tempo no cálculo do tempo total médio.

Primeiramente, calculamos o tempo da versão Java serial utilizando as quatro bases para podermos comparar com o tempo da versão paralela em Java quando executada em apenas um processador. Fizemos o mesmo para a versão Scala. Em seguida, para cada base, calculamos o tempo da versão paralela em Java utilizando dois, quatro e oito processadores. Também fizemos o mesmo para a versão Scala.

Os parâmetros utilizados nas execuções do AIRS (ver figura 6.1) foram extraídos a partir de um script fornecido pelo mestrando Marcelo de Rezende Martins. Esse script calcula bons valores dos parâmetros do AIRS, de modo que a eficácia do algoritmo seja alta. Utilizamos esse script para não precisarmos rodar os experimentos com parâmetros aleatórios. Em primeiro lugar, rodamos o script com a versão Java, tanto serial quanto paralela (com um, dois, quatro e oito processadores). Em seguida, usamos os parâmetros obtidos numa mesma execução do script nos experimentos das versões Java e Scala.

	ionosphere.arff	*iris.arff*	*diabetes.arff*	*sonar.arff*
Quantidade de testes (T)	200	200	200	200
Semente usada para gerar números aleatórios (S)	1	1	1	1
Magnitude do limiar de afinidade (F)	0	0	0	0
Clonal Rate (C)	15	1	13	1
Hypermutation Rate (H)	10	1	1	1
Quantidade total de recursos que podem ser alocados (R)	1	1	1	1
Função de estimulação (V)	1	0.01	0	0.98
Total de instâncias de treinamento para calcular o limiar de afinidade (A). O valor -1 significa que todo o conjunto de treinamento será usado.	1	-1	-1	-1
Tamanho inicial do reservatório de ARBs (E)	10	1	10	9
K-Nearest Neighbour. Quantidade de vizinhos que serão considerados na hora de realizar uma classificação de instâncias não vistas (K)	6	1	5	1
Quantidade de processos (N)	1/2/4/8	1/2/4/8	1/2/4/8	1/2/4/8
Modo de junção dos reservatórios de células de memória gerados pelos subprocessos (M). O valor 1 corresponde à concatenação simples dos dados.	1	1	1	1

Figura 6.1: Parâmetros do AIRS utilizados na execução dos testes.

6.1.3 Serial X Paralela

O primeiro experimento serviu para compararmos o desempenho entre a versão serial e a versão paralela, executada em apenas um processador, das implementações Java e Scala do algoritmo. Para isso, estabelecemos a seguinte definição retirada de [25]:

- Definição 1 Tempo Sequencial: é o tempo gasto pelo programa quando executado em um único processador da máquina em estudo.

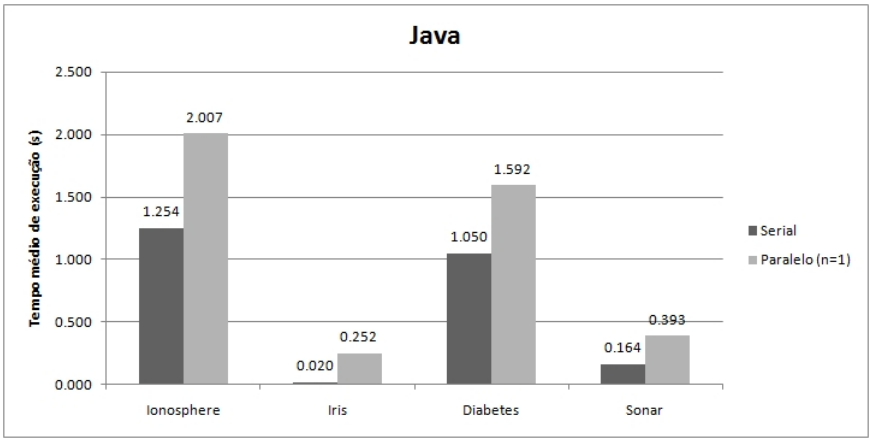
  Notação: T(1)

Na figura 6.2a podemos ver algumas estatísticas das versões serial e paralela (executada em um processador) da implementação em Java após rodarmos 200 testes em cada uma das quatro bases de dados. Nessa figura, podemos notar um menor desvio padrão na versão serial em relação à versão paralela, o que leva a um menor intervalo de confiança (I.C.).

De acordo com a figura 6.2a podemos ver que em todas as bases utilizadas a versão serial do algoritmo teve um melhor desempenho do que a versão paralela. Isso fica mais evidente se olharmos para a figura 6.2b, que mostra graficamente a diferença dos tempos para cada base de dados. O melhor desempenho da versão serial era de se esperar, dado que a versão paralela possui um preprocessamento dos dados e a divisão (*fork*) em n processos (no caso, n=1) antes

		Ionosphere	Iris	Diabetes	Sonar
Java serial	Média	1.254	0.020	1.050	0.164
	Máximo	1.71	0.36	1.66	0.48
	Mínimo	1.24	0.02	1.03	0.16
	Desvio padrão	0.033	0.026	0.045	0.024
	I.C. (95%)	[1.249;1.258]	[0.016;0.024]	[1.044;1.056]	[0.161;0.168]
Java paralelo (n=1)	Média	2.007	0.252	1.592	0.393
	Máximo	5.07	1.12	4.12	2.15
	Mínimo	1.34	0.06	1.08	0.19
	Desvio padrão	0.595	0.218	0.450	0.287
	I.C. (95%)	[1.925;2.09]	[0.222;0.282]	[1.53;1.655]	[0.353;0.433]

(a) Estatísticas das versões Java serial e paralela (n=1) do AIRS executadas em somente um processador nas quatro bases de dados.



(b) Tempo médio das versões Java serial e paralela (n=1) do AIRS executadas em somente um processador nas quatro bases de dados.

Figura 6.2: Comparação entre as implementações Java serial e paralela executadas em 1 processador.

de realizar a parte relativa à versão serial. Depois que os n processos terminam seus cálculos para criação das células de memória, os resultados são reunidos no processo principal (ver figura 6.3). Essas etapas justificam o acréscimo no tempo gasto da implementação paralela em relação à serial.

Vale ressaltar que no caso da *iris.arff* aparentemente não adianta paralelizar o algoritmo, pois o tempo médio obtido com o algoritmo paralelo (n=1) possui aproximadamente 90% de sobrecarga em relação ao tempo médio obtido com a versão serial. Sendo assim, mesmo que a paralelização reduza o tempo médio, a sobrecarga deixaria o tempo médio total elevado. Entretanto, continuaremos utilizando a base *iris.arff* nos testes para analisar o seu comportamento.

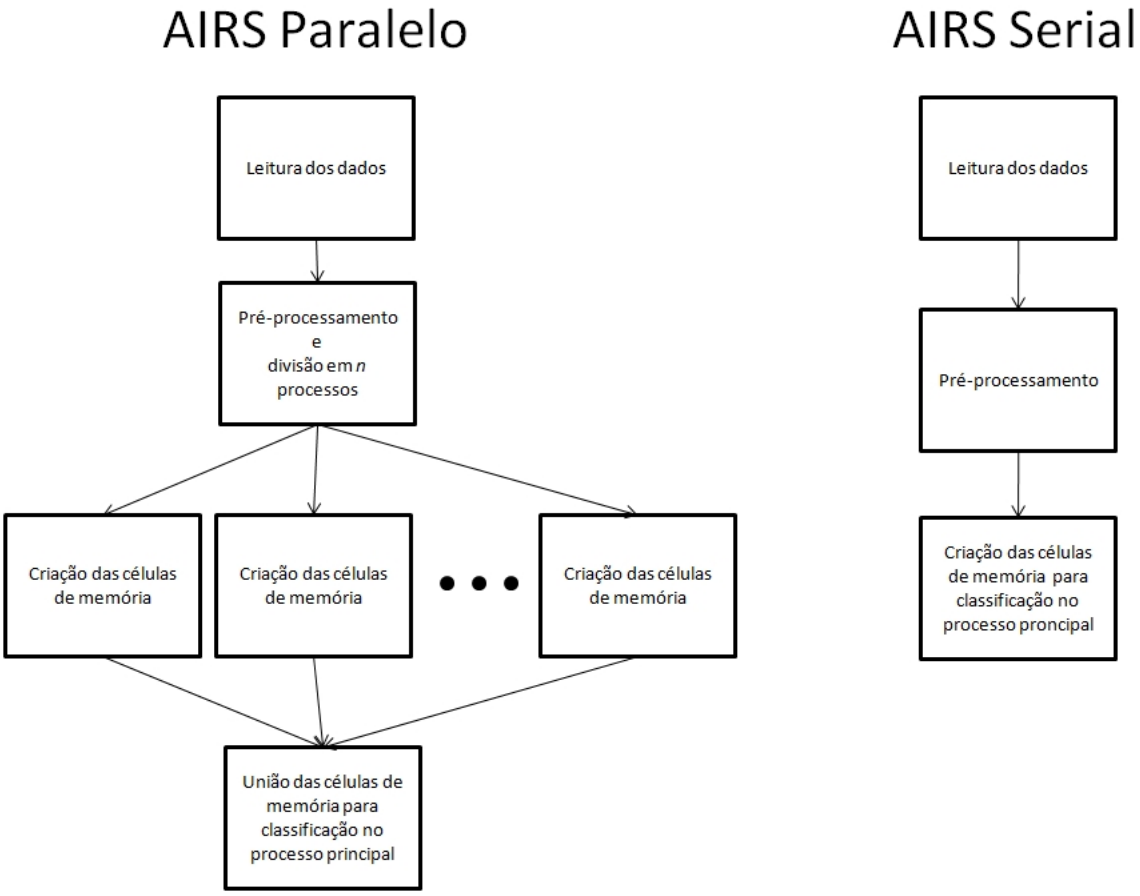


Figura 6.3: Diferença entre os fluxos de execução das versões serial e paralela.

	Ionosphere	Iris	Diabetes	Sonar
Instâncias	351	150	768	208
Atributos	35	5	9	61

Figura 6.4: Quantidade de instâncias e atributos das bases de dados utilizadas.

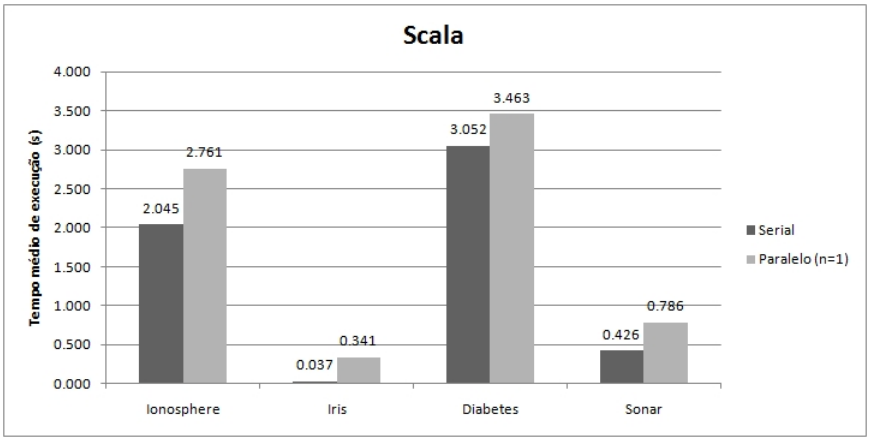
Ainda de acordo com a figura 6.2b é possível notar que o tempo médio gasto pelo algoritmo para classificar cada base está relacionado ao número de instâncias e de atributos que tal base possui. Cada instância é uma amostra do material em estudo e cada atributo representa alguma característica desse material. Por exemplo, no caso da *iris.arff*, um atributo é a largura da pétala. De acordo com a figura 6.4 a base *ionosphere.arff* tem 351 instâncias e 35 atributos. Já a base *iris.arff* contém 150 instâncias e 5 atributos. A diminuição na quantidade de instâncias e atributos fez com que o tempo médio caísse consideravelmente. Isso sugere que quanto menos instâncias e atributos, menos dados (instâncias) terão que ser analisados pelo algoritmo utilizando uma menor quantidade de parâmetros (atributos). Portanto, mais rápida será sua execução. Entretanto, se olharmos para a base *diabetes.arff*, que possui 768 instâncias e 9 atributos, vemos que o seu tempo está entre o tempo obtido na classificação da *iris.arff* e da *ionosphere.arff*. Ou seja, apesar do número de instâncias ter mais do que dobrado em relação ao da *ionosphere.arff*, o fato da quantidade de atributos ter diminuído 5 vezes fez com que o tempo gasto para classificar os dados fosse menor. Por outro lado, tanto a quantidade de instâncias quanto de atributos da base *diabetes.arff* ficaram acima das quantidades de instâncias e de atributos da base *iris.arff*, respectivamente, fazendo com que o tempo da *diabetes.arff* fosse superior. Finalmente, se compararmos a base *sonar.arff* com a base *iris.arff*, veremos que tanto a quantidade de atributos quanto a quantidade de instâncias da *sonar.arff* são superiores às quantidades da *iris.arff*, fazendo com o que tempo obtido seja superior. Mas, se comparamos a base *sonar.arff* com a base *diabetes.arff*, notaremos que apesar da quantidade de atributos da *sonar.arff* ser mais de 6 vezes maior, a quantidade de instâncias é mais de 3 vezes menor. Isso sugere que a quantidade de instâncias influencia mais no tempo do que a quantidade de atributos, uma vez que o tempo obtido com a *sonar.arff* foi menor do que aquele obtido com a *diabetes.arff*.

De maneira semelhante, a figura 6.5a nos mostra as estatísticas para a versão Scala do algoritmo. Repare que, assim como em Java, todos os tempos da versão serial foram mais baixos dos que os da versão paralela. A mesma justificativa dada na análise dos tempos de Java se aplica aqui. Há uma sobrecarga que é acrescida na versão paralela. E quando executamos o algoritmo paralelo em apenas um processador, não aproveitamos o fato do algoritmo estar paralelizado para diminuir o tempo de execução. Isso porque a implementação paralela irá desempenhar a mesma função que a versão serial, porém com a sobrecarga já descrita anteriormente.

Assim como ocorreu com a versão em Java do algoritmo, o tempo médio obtido com a versão paralela do algoritmo em Scala quando utilizada com a base *iris.arff* teve uma sobrecarga de aproximadamente 90% em relação ao tempo médio obtido com a versão serial. Mesmo assim,

		Ionosphere	Iris	Diabetes	Sonar
Scala serial	Média	2.045	0.037	3.052	0.426
	Máximo	2.65	0.35	4.00	0.83
	Mínimo	2.02	0.03	2.94	0.41
	Desvio padrão	0.045	0.025	0.082	0.031
	I.C. (95%)	[2.039;2.052]	[0.034;0.041]	[3.041;3.064]	[0.422;0.431]
Scala paralelo (n=1)	Média	2.761	0.341	3.463	0.786
	Máximo	6.57	3.79	6.30	3.62
	Mínimo	2.30	0.08	2.82	0.46
	Desvio padrão	0.536	0.380	0.538	0.448
	I.C. (95%)	[2.687;2.835]	[0.288;0.394]	[3.389;3.538]	[0.724;0.848]

(a) Estatísticas das versões Scala serial e paralela (n=1) do AIRS executadas em somente um processador nas quatro bases de dados.



(b) Tempo médio das versões Scala serial e paralela (n=1) do AIRS executadas em somente um processador nas quatro bases de dados.

Figura 6.5: Comparação entre as implementações Scala serial e paralela executadas em 1 processador.

continuaremos analisando a base *iris.arff* para sabermos como a base se comportará.

Se olharmos para a figura 6.5b notaremos que é um gráfico semelhante ao da versão Java (ver figura 6.2b). As duas principais diferenças são: os tempos, tanto da versão serial quanto da paralela, são maiores em Scala do que em Java e a diferença entre os tempos das versões serial e paralela é menor na implementação de Scala do que na de Java. A primeira diferença mostra um melhor desempenho de Java em relação a Scala em ambas as versões (serial e paralela). Por outro lado, a segunda diferença mostra que a implementação em Scala teve uma menor sobrecarga após ser paralelizada.

6.1.4 Avaliação do tempo médio de execução

A segunda avaliação que fizemos foi em relação ao tempo que as implementações do algoritmo AIRS em Java e em Scala consumiram até concluírem os experimentos. Na figura 6.6 podemos ver os tempos médios, em segundos, que as duas implementações levaram para rodar cada experimento. Podemos notar que os tempos de Scala foram, em geral, acima dos tempos obtidos em Java, principalmente nos experimentos realizados com a base *sonar.arff*. Além dos tempos, podemos ver outras estatísticas como máximo, mínimo, desvio padrão e intervalo de confiança (I.C.). O desvio padrão mostra que a dispersão dos dados em Java é semelhante à dispersão em Scala, apesar dos tempos de Java serem menores. Contudo, tal semelhança não é válida para a base *sonar.arff*, onde o desvio padrão de Scala é quase duas vezes maior que o de Java para 1 e 8 processos.

Se olharmos para a figura 6.7a veremos uma comparação entre os tempos médios dos testes rodados na base *ionosphere.arff* das versões Java e Scala utilizando 1, 2, 4 e 8 processos. Esses tempos aumentam conforme a quantidade de processadores utilizados cresce, ao contrário do que esperávamos. Ou seja, mesmo paralelizando o algoritmo, o tempo médio de execução para treinar e validar os dados aumenta a uma taxa crescente. Apesar dos resultados com a base *ionosphere.arff* não fazerem sentido, não conseguimos encontrar a razão para tal comportamento.

Na figura 6.7b podemos ver os tempos médios para a base *iris.arff* das implementações Java e Scala. Nesse caso, os tempos de Java também crescem à medida que o número de processos aumenta. Por outro lado, o desempenho de Scala vai melhorando e seus tempos vão diminuindo a ponto de ultrapassarem os de Java. No caso da base *diabetes.arff* ocorreu o inverso da *ionosphere.arff*. Tanto os tempos obtidos com Java quanto os obtidos com Scala decresceram à medida que a quantidade de processos aumentou. Mais uma vez os tempos de Java foram inferiores do que os de Scala, mas a queda dos tempos da versão Scala foi mais acentuada. A base de dados onde houve maior discrepância de tempo foi a *sonar.arff*. Nessa base a relação entre os tempos das implementações em Java e em Scala chegou a ser de 1:2 (n=1). E mesmo para outros valores de n, os tempos médios de ambas versões não se alteraram muito, mantendo certa constância na relação.

6.1.5 Avaliação da aceleração

Também avaliamos qual o ganho no tempo total de execução se adicionássemos mais processadores para executar a versão paralela. Para isso, precisamos definir a seguinte métrica:

Ionosphere		n=1	n=2	n=4	n=8
Java paralelo	Média	2.007	2.071	3.736	6.330
	Máximo	5.07	4.43	6.09	9.01
	Mínimo	1.34	1.58	3.17	4.64
	Desvio padrão	0.595	0.412	0.459	0.802
	I.C. (95%)	[1.925;2.09]	[2.014;2.129]	[3.672;3.8]	[6.218;6.441]
Scala paralelo	Média	2.761	2.661	3.595	6.916
	Máximo	6.57	4.70	6.16	10.49
	Mínimo	2.30	2.16	3.03	4.80
	Desvio padrão	0.536	0.392	0.481	0.877
	I.C. (95%)	[2.687;2.835]	[2.607;2.716]	[3.529;3.662]	[6.794;7.037]

(a) Comparação entre as versões paralelas de Java e Scala utilizando a base *ionosphere.arff* à medida que aumentamos o número de processos.

Iris		n=1	n=2	n=4	n=8
Java paralelo	Média	0.252	0.286	0.351	0.364
	Máximo	1.12	2.99	2.76	5.28
	Mínimo	0.06	0.06	0.06	0.08
	Desvio padrão	0.218	0.310	0.405	0.442
	I.C. (95%)	[0.222;0.282]	[0.243;0.328]	[0.295;0.407]	[0.303;0.426]
Scala paralelo	Média	0.341	0.353	0.314	0.318
	Máximo	3.79	4.62	2.78	2.55
	Mínimo	0.08	0.08	0.07	0.06
	Desvio padrão	0.380	0.420	0.305	0.378
	I.C. (95%)	[0.288;0.394]	[0.295;0.412]	[0.272;0.357]	[0.265;0.37]

(b) Comparação entre as versões paralelas de Java e Scala utilizando a base *iris.arff* à medida que aumentamos o número de processos.

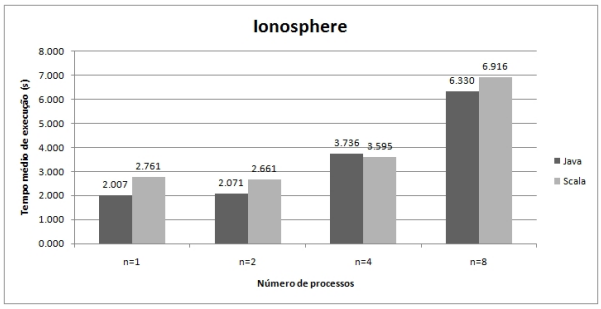
Diabetes		n=1	n=2	n=4	n=8
Java paralelo	Média	1.592	0.965	0.904	0.876
	Máximo	4.12	3.68	3.56	3.27
	Mínimo	1.08	0.57	0.50	0.50
	Desvio padrão	0.450	0.481	0.439	0.453
	I.C. (95%)	[1.53;1.655]	[0.899;1.032]	[0.843;0.965]	[0.814;0.939]
Scala paralelo	Média	3.463	1.614	1.281	1.261
	Máximo	6.30	4.04	3.66	3.67
	Mínimo	2.82	1.10	0.87	0.87
	Desvio padrão	0.538	0.440	0.410	0.369
	I.C. (95%)	[3.389;3.538]	[1.553;1.675]	[1.224;1.337]	[1.209;1.312]

(c) Comparação entre as versões paralelas de Java e Scala utilizando a base *diabetes.arff* à medida que aumentamos o número de processos.

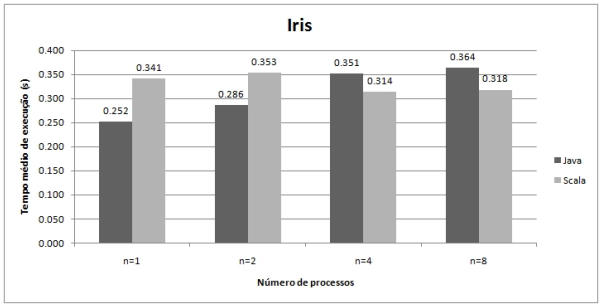
Sonar		n=1	n=2	n=4	n=8
Java paralelo	Média	0.393	0.348	0.400	0.393
	Máximo	2.15	2.85	3.16	2.50
	Mínimo	0.19	0.17	0.19	0.19
	Desvio padrão	0.287	0.306	0.356	0.311
	I.C. (95%)	[0.353;0.433]	[0.306;0.391]	[0.35;0.449]	[0.349;0.437]
Scala paralelo	Média	0.786	0.685	0.711	0.760
	Máximo	3.62	2.94	2.91	4.59
	Mínimo	0.46	0.41	0.39	0.39
	Desvio padrão	0.448	0.348	0.393	0.537
	I.C. (95%)	[0.724;0.848]	[0.637;0.733]	[0.656;0.765]	[0.686;0.834]

(d) Comparação entre as versões paralelas de Java e Scala utilizando a base *sonar.arff* à medida que aumentamos o número de processos.

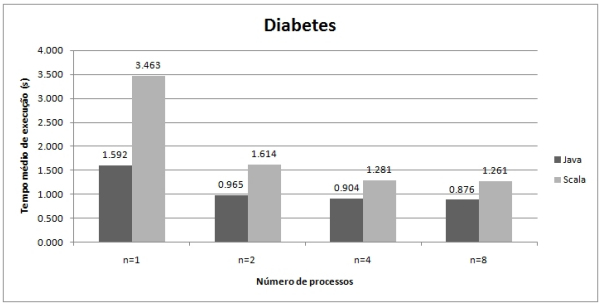
Figura 6.6: Estatísticas do tempo de execução das duas implementações do AIRS nas 4 bases de dados.



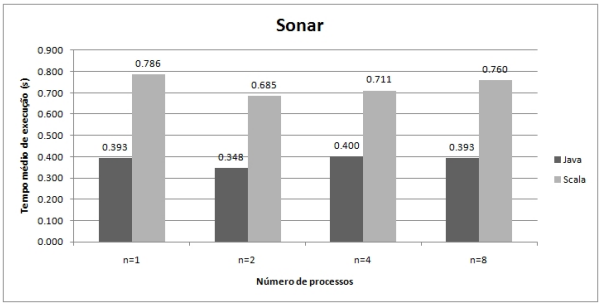
(a) Comparação dos tempos médios de Java e Scala utilizando 1, 2, 4 e 8 processos com a base *ionosphere.arff*.



(b) Comparação dos tempos médios de Java e Scala utilizando 1, 2, 4 e 8 processos com a base *iris.arff*.



(c) Comparação dos tempos médios de Java e Scala utilizando 1, 2, 4 e 8 processos com a base *diabetes.arff*.



(d) Comparação dos tempos médios de Java e Scala utilizando 1, 2, 4 e 8 processos com a base *sonar.arff*.

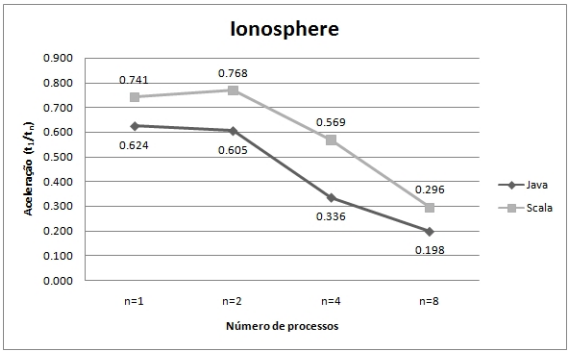
Figura 6.7: Gráficos dos tempos médios em segundos das duas implementações do AIRS nas 4 bases de dados.

- Métrica 1.1 Aceleração: a aceleração de n processadores é a relação entre o tempo sequencial e o tempo gasto pelo programa quando executado com n processadores.

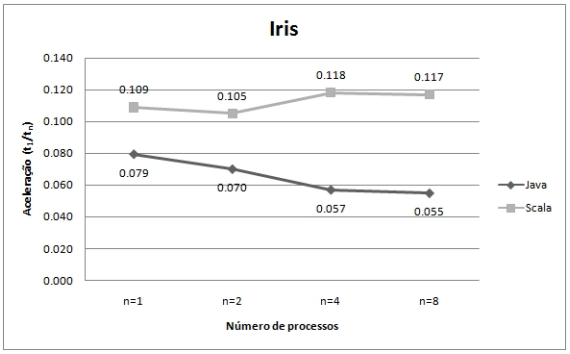
$$S(n) = \frac{T(1)}{T(n)} \tag{6.1}$$

O ideal é que a aceleração seja igual ao grau de paralelismo (número de processadores ativos), mas geralmente o valor é menor devido a sobrecargas por conta do paralelismo.

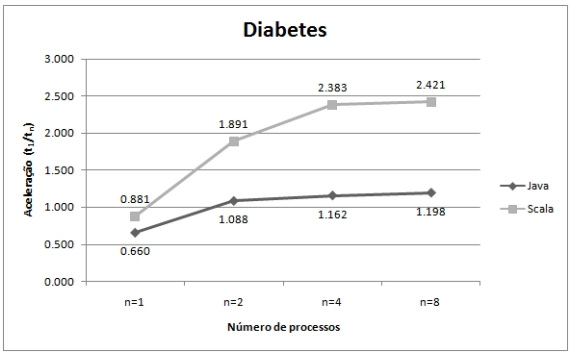
A figura 6.8 compara a aceleração da implementação Java paralela do algoritmo com a aceleração de Scala paralela para as quatro bases de dados quando utilizamos 1, 2, 4 e 8 processos. De acordo com essa figura, podemos ver que o ganho nos tempos médios de Scala é superior ao de Java em todas as bases apesar dos tempos médios de Java serem menores.



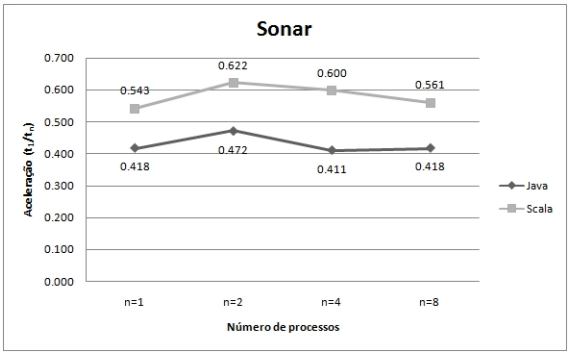
(a) Aceleração obtida para as versões Java e Scala do AIRS paralelo com a base *ionosphere.arff*.



(b) Aceleração obtida para as versões Java e Scala do AIRS paralelo com a base *iris.arff*.



(c) Aceleração obtida para as versões Java e Scala do AIRS paralelo com a base *diabetes.arff*.



(d) Aceleração obtida para as versões Java e Scala do AIRS paralelo com a base *sonar.arff*.

Figura 6.8: Gráficos das acelerações das duas implementações do AIRS nas 4 bases de dados testadas.

Se analisarmos a figura 6.8a, veremos que, em ambas as implementações, a aceleração diminuiu ao invés de crescer à medida que o número de processos utilizados aumentou. Isso se deve ao fato de os tempos médios terem aumentado conforme aumentamos a quantidade de núcleos utilizados, como foi visto na Seção 6.1.4. Como esse aumento dos tempos médios de Scala foi menos acentuado do que o crescimento obtido na versão Java, a relação $\frac{t_1}{t_n}$ ficou maior em Scala. Além disso, como a variação dos tempos médios das duas implementações foi similar, vemos duas linhas quase paralelas no gráfico da aceleração da *ionosphere.arff*.

Para a base de dados *iris.arff* (ver figura 6.8b) as acelerações de Scala e Java seguiram direções opostas, apesar do gráfico ter uma amplitude bem menor em relação ao gráfico da base *ionophere.arff*. À medida que aumentamos a quantidade de processos, o tempo médio da versão Scala diminuiu fazendo com que a aceleração aumentasse. Por outro lado, o tempo médio da implementação Java aumentou e, consequentemente, a aceleração diminuiu.

Como podemos ver na figura 6.8c, com a base de dados *diabetes.arff* a aceleração de Scala também foi maior do que a da versão Java para todos os valores de processos testados. Essa foi a única base na qual os tempos médios de ambas versões diminuiram conforme aumentávamos a quantidade de processos utilizados. O reflexo disso foi que tanto a aceleração de Scala quanto a de Java aumentaram à medida que o número de processos aumentava. Um detalhe sobre os testes dessa base é que nos experimentos de Scala para 1 e 2 processos (n=1 e n=2), os valores obtidos foram próximos àqueles ideais: 0.881 e 1.891. Como havíamos dito anteriormente, o ideal é que a aceleração seja igual ao grau de paralelismo, que nesse caso seria igual a 1 e 2, respectivamente. Assim, nesses dois casos podemos dizer que houve um bom ganho no tempo médio total de execução do algoritmo.

Finalmente, olhando para a figura 6.8d notamos que não houve uma grande variação nos valores das acelerações de Scala e de Java, como podemos notar através da pequena variação dos valores no eixo vertical. Isso é consequência da estabilidade dos tempos médios mostrados na Seção 6.1.4 para os testes realizados com a base *sonar.arff*. Podemos reparar também que, assim como a base *ionosphere.arff*, as curvas de aproximação das acelerações de Java e Scala são quase que paralelas, o que reflete a variação semelhante dos tempos médios de ambas as implementações.

Resumindo, apesar dos tempos médios dos experimentos com a implementação em Scala terem sido maiores do que os de Java, o ganho obtido nos tempos de Scala cada vez que aumentávamos o número de processos foi maior do que o ganho da implementação Java do algoritmo.

6.1.6 Avaliação da eficiência

De posse da aceleração dos n processadores e dos tempos seriais podemos calcular quão eficiente é a máquina quando roda cada uma das implementações do algoritmo. A eficiência pode ser calculada da seguinte maneira:

- Métrica 1.2 Eficiência: indica quão bem utilizada a máquina está sendo.

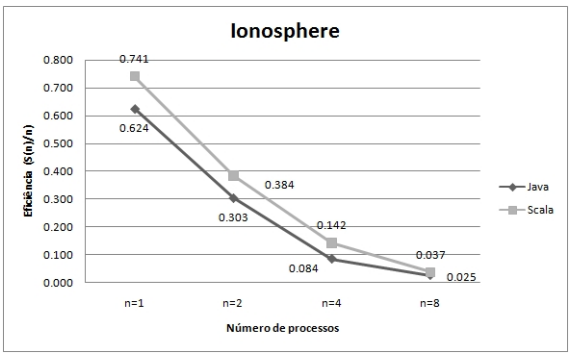
$$E(n) = \frac{T(1)}{n * T(n)} = \frac{S(n)}{n} \tag{6.2}$$

  Assim, quanto mais próximo de 1 estiver, mais eficiente será a máquina. Entretanto, esse número nunca poderá ser exatamente 1, pois existem partes do algoritmo que são necessariamente sequenciais, ou seja, a aceleração não será linear (S(n) nunca será igual a n).

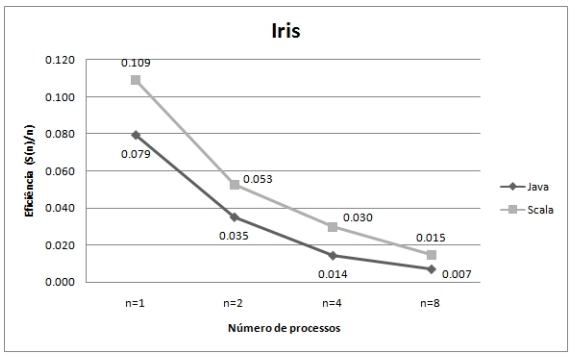
Na figura 6.9 podemos ver uma comparação entre as eficiências obtidas através das implementações de Scala e Java para as quatro base de dados que utilizamos. De um modo geral, todos os gráficos apresentam curvas decrescentes, tanto para Java quanto para Scala. Isso quer dizer que à medida que o número de processos definidos aumenta, o rendimento da máquina diminui.

Ainda na figura 6.9 podemos notar que nos testes de todas as bases a eficiência de Scala foi superior à de Java. Como pode ser visto na figura 6.9c, no caso da base *diabetes.arff*, a eficiência se aproximou bastante de 1, que seria o ideal. As bases *ionosphere.arff* e *sonar.arff* começaram com valores relativamente altos de eficiência (0.741 e 0.543, respectivamente, para Scala e 0.624 e 0.418 para Java). Entretanto, decaíram bastante conforme aumentávamos o número de processos utilizados pelo algoritmo chegando ao ponto de terem o primeiro algarismo significativo na segunda casa decimal (ver figuras 6.9a e 6.9d). A base de dados *iris.arff* foi a que teve menor eficiência. Seus valores, apesar de terem uma menor amplitude, o que representa um menor decréscimo, são mais baixos do que os valores das outras bases. Ou seja, estão mais próximos de 0 do que de 1, sendo portanto, menos eficiente.

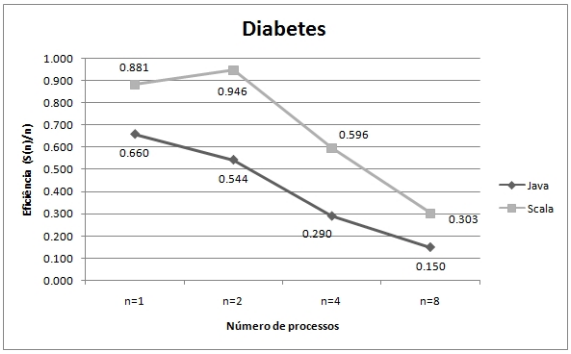
Concluindo, nos gráficos podemos ver as curvas de eficiência decrescerem em todos os testes mostrando que o rendimento diminui com o aumento da quantidade de processos utilizados. Além disso, vemos também que os valores das eficiências obtidos com a implementação em Scala foram de 1,5 a 2 vezes maiores que os valores obtidos em Java.



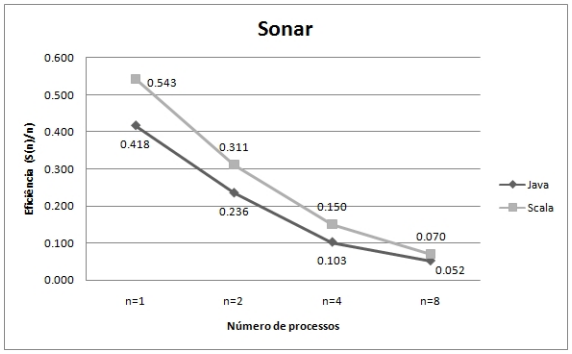
(a) Eficiência obtida para as versões Java e Scala do AIRS paralelo com a base *ionosphere.arff*.



(b) Eficiência obtida para as versões Java e Scala do AIRS paralelo com a base *iris.arff*.



(c) Eficiência obtida para as versões Java e Scala do AIRS paralelo com a base *diabetes.arff*.



(d) Eficiência obtida para as versões Java e Scala do AIRS paralelo com a base *sonar.arff*.

Figura 6.9: Gráficos das eficiências das duas implementações do AIRS nas 4 bases de dados testadas.

6.2 Avaliação

Esta seção apresenta uma avaliação qualitativa do modelo tradicional baseado em *threads* e adotado em Java e do modelo de atores adotado em Scala. A avaliação é feita no contexto do nosso projeto (paralelização do algoritmo AIRS).

A existência de uma biblioteca com classes específicas para programação concorrente (`java.util.concurrent`) faz com que o controle de acesso aos dados compartilhados em Java seja facilitado. Além disso, o exemplo escolhido, apesar de permitir que o código seja paralelizado, faz com que os processos criados sejam executados independentemente uns dos outros. Assim, eles têm que mandar seus resultados para o processo principal apenas no final do seu processamento. Dessa maneira, não há necessidade dos processos criados trocarem informações entre si e não se corre o risco de vários desses processos criados tentarem acessar

a mesma variável ao mesmo tempo, situações nas quais o programador que fosse utilizar a biblioteca concorrente de Java teria que ter um maior cuidado.

Apesar disso, a facilidade e os benefícios de se trabalhar com o modelo de troca de mensagens ficaram evidentes. Pudemos facilmente definir mensagens que um ator mandaria para o outro, ao invés de ter que usar métodos ou mensagens pré-definidas. Desse modo, a interpretação das mensagens ficou mais intuitiva. Além disso, o mecanismo de casamento de padrões no recebimento das mensagens também foi bastante útil, uma vez que serviu para especificar quais mensagens seriam tratadas pelo ator e como ele iria tratá-las.

7 Considerações Finais

Dividimos esse capítulo em 2 seções. A Seção 7.1 discorre sobre as contribuições do nosso trabalho. Já a Seção 7.2 indica possíveis direções para trabalhos futuros.

7.1 Principais Contribuições

Neste projeto identificamos algumas contribuições para as áreas de Engenharia de Software, Computação Paralela e Distribuída e áreas afins. Dividimos essa seção em duas partes. A primeira trata das contribuições relativas à implementação do algoritmo enquanto que a segunda discorre sobre as contribuições feitas através da análise de desempenho realizada entre as versões Java e Scala do algoritmo.

7.1.1 Implementação do algoritmo AIRS paralelo em Scala

Uma imediata contribuição do nosso trabalho foi ter fornecido uma implementação do AIRS em outra linguagem. Isso permitirá que outras pessoas comparem o código da nossa versão com outras versões do algoritmo ou, até mesmo, a reutilizem. Além disso, a implementação do algoritmo AIRS que realizamos durante nosso projeto nos permitiu tirar algumas conclusões:

- Reaproveitamento de códigos prontos: grande parte das aplicações atuais são escritas em Java. Com o nosso projeto, mostramos que, caso os projetistas não queiram reescrever os sistemas em Scala, poderão escrever apenas os módulos que forem mais convenientes em Scala ou em Java, uma vez que as linguagens são interoperáveis.
- Maior eficiência no desenvolvimento de aplicações concorrentes: através do nosso projeto verificamos que, de fato, o modelo de atores adotado em Scala pode facilitar o trabalho dos programadores no desenvolvimento de aplicações concorrentes. Com isso, as equipes de desenvolvimento só têm a ganhar uma vez que desenvolver sistemas paralelos (ou distribuídos) não é uma tarefa rápida. Mais do que isso, se feito de maneira errada, ou não feito, pode fazer com que o sistema funcione de maneira inesperada.

- Expansão das possibilidades de uso da WEKA: Como consequência da interoperabilidade entre Java e Scala, conseguimos fazer com que a nossa versão do AIRS paralelo em Scala também rodasse na WEKA, que é desenvolvida em Java. Assim, aumentaram as possibilidades de uso da bancada de ferramentas, já que os algoritmos existentes podem ser reescritos em Scala. Há ainda a opção de implementar em Scala apenas as partes dos algoritmos que justifiquem a mudança de linguagem.

7.1.2 Análise de desempenho entre as versões Java e Scala

Durante nosso projeto fizemos uma série de comparações entre a implementação original do AIRS e a que desenvolvemos em Scala. Para avaliar qual das implementações do algoritmo AIRS possuía um melhor desempenho, comparamos os tempos médios que elas levaram para realizar os experimentos conforme aumentávamos a quantidade de processos utilizados. Outra comparação que fizemos foi para saber qual implementação possuía um maior ganho em seu tempo médio de execução em relação ao tempo sequencial à medida que aumentávamos o número de processos. Por fim, analisamos quão bem-aproveitada a máquina foi por ambas versões do algoritmo. Tais comparações nos permitiram tirar as seguintes conclusões:

- Aproveitamento dos núcleos disponíveis: utilizando o paradigma de troca de mensagem, através do modelo de atores, Scala sugere que se tenha um maior aproveitamento dos núcleos disponíveis em máquinas *multicore* no que diz respeito a tempo de execução. Através do nosso projeto, não pudemos confirmar essa hipótese. Apesar de variar de base para base, os tempos obtidos nos experimentos com a versão Scala do AIRS foram, em geral, maiores que aqueles obtidos com a versão Java.

- Aceleração: apesar da versão Scala exibir tempos médios superiores aos de Java, a aceleração à medida que aumentávamos a quantidade de processos foi superior na versão em Scala. Os valores dos experimentos feitos com o modelo baseado na troca de mensagens ficaram mais próximos do valor ideal (grau de paralelismo) do que os valores com o modelo baseado em *threads*.

- Eficiência: outro fato que pudemos observar foi que a implementação feita em Scala fez com que a máquina empregada fosse utilizada mais eficientemente do que a versão em Java. Isso porque os valores da eficiência obtidos com a versão em Scala ficaram mais próximos de 1 do que os obtidos com a versão Java.

Vale ressaltar que, mesmo paralelizando as versões Scala e Java do algoritmo AIRS, o tempo médio obtido com a base *ionosphere.arff* aumentou. Entretanto, não conseguimos encontrar

um motivo para tal comportamento.

7.2 Trabalhos Futuros

Realização de novos testes. Uma possível direção para trabalho futuro identificada em nosso projeto é a realização de novos testes com outras bases. Em especial, a realização de testes com a base de cartão de crédito, que é utilizada pelo grupo de pesquisa em computação bioinspirada do IME-USP para detecção de fraudes em transações de cartão de crédito.

Implementação de melhorias no algoritmo. Em nosso trabalho, não realizamos nenhuma alteração no algoritmo original do AIRS quando implementamos a versão em Scala. Assim, outra possível direção que identificamos foi a implementação de melhorias no algoritmo, sejam elas as citadas em [7] ou quaisquer outras que tenham sido identificadas.

Utilização de outros exemplos. Nesse trabalho, utilizamos como exemplo as versões Java e Scala do algoritmo AIRS paralelo para podermos comparar o desempenho entre elas. Entretanto, como pudemos ver, não houve a necessidade de muita comunicação entre os processos. Portanto, uma terceira opção de caminho a se seguir é a realização dos testes com exemplos que necessitem de mais comunicação entre os processos.

Referências Bibliográficas

[1] Ericsson Telecom Ab. The development of Erlang. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 196–203. ACM Press, 1997.

[2] Gul A. Agha. *ACTORS: A Model for Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachusetts, 1986.

[3] Artificial Immune System. `http://www.artificial-immune-systems.org`. Sítio visitado em agosto de 2010.

[4] Joe Armstrong. Erlang - A survey of the language and its industrial applications. In *INAP'96 - The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, 1996.

[5] Bases de Dados. `http://archive.ics.uci.edu/ml/index.html`. Sítio visitado em agosto de 2010.

[6] Bird and Wadler. *Introduction to Functional Programming (2nd Edition)*. Prentice Hall PTR, April 1998.

[7] Jason Brownlee. Artificial Immune Recognition System (AIRS) - A Review and Analysis. Technical report, Centre for Intelligent Systems and Complex Processes (CISCP), Faculty of Information and Communication Technologies (ICT), Swinburne University of Technology, 2005.

[8] Jason Brownlee. Clonal Selection Theory and Clonalg - The Clonal Selection Classification Algorithm (CSCA). Technical report, Centre for Intelligent Systems and Complex Processes (CISCP), Faculty of Information and Communication Technologies (ICT), Swinburne University of Technology, 2005.

[9] Rod Burstall. Christopher Strachey - Understanding Programming Languages. *Higher Order and Symbolic Computation*, 13(1-2):51–55, 2000.

[10] Leandro N. de Castro and Fernando J. Von Zuben. Learning and optimization using the clonal selection principle. In *Evolutionary Computation, IEEE Transactions on*, volume 6, pages 239–251, 2002.

[11] Eclipse. `http://www.eclipse.org`. Sítio visitado em agosto de 2010.

[12] Erlang. `http://www.erlang.org`. Sítio visitado em agosto de 2010.

[13] Erlang IBM. `http://www.ibm.com/developerworks/java/library/j-cb04186.html`. Sítio visitado em agosto de 2010.

[14] DR Forsdyke. The origins of the clonal selection theory of immunity as a case study for evaluation in science. *The FASEB Journal*, 9:164–166, 1995.

[15] Eibe Frank, Mark Hall, Geoffrey Holmes, Richard Kirkby, Bernhard Pfahringer, Ian H. Witten, and Len Trigg. *Data Mining and Knowledge Discovery Handbook: A Complete Guide for Practitioners and Researchers*, chapter 1. Springer-Verlag, 2005.

[16] Manoel Fernando Alonso Gadi, Xidi Wang, and Alair Pereira do Lago. Credit Card Fraud Detection with Artificial Immune System. In *ICARIS*, pages 119–131, 2008.

[17] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[18] Philipp Haller and Martin Odersky. Event-Based Programming without Inversion of Control. In *Proc. JMLC 2006*, 2006.

[19] Philipp Haller and Martin Odersky. Actors That Unify Threads and Events. In *Proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION 2007)*, volume 4467 of *Lecture Notes in Computer Science*, pages 171–190, Paphos, Cyprus, June 2007. Springer.

[20] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI-73)*, pages 235–245, Stanford, California, USA, August 1973. William Kaufmann.

[21] Java Sun. `http://java.sun.com/`. Sítio visitado em agosto de 2010.

[22] Java - Orientação a Objetos. `http://java.sun.com/docs/books/tutorial/java/concepts`. Sítio visitado em agosto de 2010.

[23] Java Software Development Kit (SDK). `http://java.sun.com/javase/downloads`. Sítio visitado em agosto de 2010.

[24] David Holmes Ken Arnold, James Gosling. *Java Programming Language, The, 4/E*. Prentice Hall, 4 edition, August 2005.

[25] MAC431 - Introdução ao Processamento Paralelo e Distribuído. `http://mairinque.ime.usp.br/~gubi/cursos/431/apostila/index.html`. Sítio visitado em agosto de 2010.

[26] NetBeans. `http://netbeans.org`. Sítio visitado em agosto de 2010.

[27] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An Overview of the Scala Programming Language. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2006.

[28] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, 2008.

[29] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 41–57, New York, NY, USA, 2005. ACM.

[30] Dan Sahlin and Ericsson Telecom. The concurrent functional programming language Erlang - An Overview, 1996.

[31] Scala. `http://www.scala-lang.org`. Sítio visitado em agosto de 2010.

[32] Scala History. `http://www.scala-lang.org/node/239`. Sítio visitado em agosto de 2010.

[33] Plugin de Scala para Eclipse. `http://www.scala-lang.org/node/94`. Sítio visitado em agosto de 2010.

[34] Plugin de Scala para NetBeans. `http://wiki.netbeans.org/Scala`. Sítio visitado em agosto de 2010.

[35] Tiobe. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. Sítio visitado em agosto de 2010.

[36] Andrew Watkins. A new classifier based on resource limited Artificial Immune Systems. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC2002), IEEE*, volume 2, pages 1546–1551. Press, 2002.

[37] Andrew Watkins, Xintong Bi, and Amit Phadke. Parallelizing an Immune-Inspired Algorithm for Efficient Pattern Recognition. In *Intelligent Engineering Systems through Artificial Neural Networks: Smart Engineering System Design: Neural Networks, Fuzzy Logic, Evolutionary Programming, Complex Systems and Artificial Life*, pages 224–230. ASME Press, 2003.

[38] Andrew Watkins and Jon Timmis. Exploiting Parallelism Inherent in AIRS, an Artificial Immune Classifier. In *ICARIS*, pages 427–438, 2004.

[39] Andrew Watkins, Jon Timmis, and Lois Boggess. Artificial Immune Recognition System (AIRS): An Immune-Inspired Supervised Learning Algorithm. *Genetic Programming and Evolvable Machines*, 5(3):291–317, 2004.

[40] Weka Machine Learning Project. `http://www.cs.waikato.ac.nz/ml`. Sítio visitado em agosto de 2010.

[41] WEKA com AIRS. `http://wekaclassalgos.sourceforge.net`. Sítio visitado em agosto de 2010.