FRANCISCO VINHAS NETO

EXECUÇÃO PARALELA DE PROGRAMAÇÃO GENÉTICA UTILIZANDO MAPREDUCE

LAVRAS-MG
2013

FRANCISCO VINHAS NETO

EXECUÇÃO PARALELA DE PROGRAMAÇÃO GENÉTICA UTILIZANDO MAPREDUCE

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso para obtenção do título de Bacharel em Ciência da Computação

Orientador
Dra. Marluce Rodrigues Pereira

Co-Orientador
Renato Resende Ribeiro de Oliveira

LAVRAS – MG
2013

FRANCISCO VINHAS NETO

EXECUÇÃO PARALELA DE PROGRAMAÇÃO GENÉTICA UTILIZANDO MAPREDUCE

Monografia apresentada ao Colegiado do Curso de Ciência da Computação, para obtenção do titulo de Bacharel em Ciência da Computação

APROVADA em 16 de Agosto de 2013.

Prof. Dr. Denilson Alves Pereira	UFLA
Prof. Dr. Tales Heimfarth	UFLA
Renato Resende Ribeiro de Oliveira	UFLA

Dra. Marluce Rodrigues Pereira
(Orientador)

LAVRAS – MG

2013

AGRADECIMENTOS

Agradeço aos meus pais, Nilson e Cristiany, aos meus irmãos, André e Guilherme, pela confiança e amor incondicional, pelo apoio nos momentos de dificuldades. Essa conquista só foi possível graças a vocês!

Aos meus amigos Kayo e João Vitor, pela amizade e exemplo de vida.

Aos meus amigos de Lavras, Pablo, Bolina, Luiz, Bruno, Guilherme, Igor e amigos da república. Que sempre me apoiaram em todos os momentos.

RESUMO

A Programação Genética é uma técnica utilizada para geração automática de aplicações em Redes de Sensores sem Fio, que necessita realizar um certo número de simulações para um determinado problema para que se tenha um maior grau de confiança do resultado obtido pelo método. Dessa forma, o seu tempo de execução torna-se alto quando utilizando uma única máquina. Porém, existem oportunidades de paralelização dessas execuções que podem implicar na redução do tempo de execução e na melhoria da qualidade dos resultados obtidos. Neste trabalho, é realizado um estudo sobre o modelo de programação MapReduce adaptado para uma Programação Genética para geração automática de aplicações em Rede de Sensores sem Fio (RSSF), através da distribuição das execuções entre as máquinas de um cluster. É proposta uma implementação de uma Programação Genética para geração automática de aplicações em RSSF e utilizado um simulador de RSSF para avaliar a qualidade da solução. São analisados também os benefícios de se usar o framework MapReduce.

Palavras-chave: MapReduce; Programação Genética; Rede de Sensores sem Fio; Programação Paralela.

ABSTRACT

The Genetic Programming is a technique used for automatic generation of applications in Wireless Sensor Networks, which needs to perform a number of simulations for a given problem in order to have a greater degree of confidence of the result obtained by the method. Thus, its running time becomes high when using a single machine. However, there are opportunities for parallelization of these executions that might imply a reduction in execution time and improving the quality of the results. This work is a study on the MapReduce programming model adapted for a Genetic Programming to automatic generation of applications in Wireless Sensor Network (WSN), through the distribution of executions among the machines of a cluster. It proposed an implementation of a Genetic Programming to automatic generation of applications in WSN and used WSN simulator to evaluate the quality of the solution. This study also analyzes the benefits of using the MapReduce framework.

Keywords: MapReduce; Genetic Programming; Wireless Sensor Network; Parallel Programming.

SUMÁRIO

Lista de Figuras

Lista de Tabelas

1. INTRODUÇÃO

1.1. Contextualização

O avanço da tecnologia nas áreas de sensores, circuitos integrados e comunicação sem fio levaram a criação de redes de sensores sem fio (RSSF). Esse tipo de rede pode ser aplicado no monitoramento, rastreamento, coordenação e processamento em diferentes contextos. Por exemplo, pode interconectar sensores para fazer o monitoramento de certos eventos em cidades, florestas ou oceanos. Essas redes são formadas por diversos sensores, denominados nodos (elemento computacional com capacidade de processamento, memória, interface de comunicação sem fio e um ou mais sensores), tem restrições de energia, e devem possuir um sistema de auto-configuração, devido à perda de nodos e falhas de comunicação (LOUREIRO *et al.* 2003).

Para que a RSSF possa se auto-configurar, usa-se um *middleware* específico que irá abstrair a complexidade da especificação de auto-nível e da infraestrutura para aplicações cooperativas em uma RSSF, tratando a reprogramação da RSSF como um todo sem a intervenção humana. Assim, o usuário conectado em um ponto arbitrário da rede pode injetar tarefas na RSSF. Essas instruções podem ser feitas usando uma linguagem *script* de alto nível, onde o *script* será interpretado pelo *middleware* instalado nos nodos. Porém, *scripts* escritos em um paradigma imperativo requerem uma programação distribuída, trazendo cenas imprevistas, complicando a programação.

Apesar do programador não precisar especificar um código específico para cada nodo, ele precisa especificar a colaboração dos nodos como um todo. Wang e Cao (2008) apresentam trabalhos existentes sobre *middlewares* para RSSF. A utilização da Programação Genética (KOZA 1992) é feita para gerar e evoluir uma aplicação para RSSF utilizando uma linguagem *script* fornecida pelo

Middleware, que irá eficientemente resolver a tarefa designada pelo usuário. Existe a necessidade de utilizar um módulo de simulação de uma RSSF, pois a Programação Genética exige inúmeras execuções dos *scripts* nos sensores da RSSF para serem feitas as avaliações dos *scripts*, o que é inviável utilizar nós sensores reais para cada candidato à solução.

Sendo a Programação Genética um método estocástico, existe a necessidade de ser realizado um determinado número de simulações para um determinado problema para que se tenha um maior grau de confiança do resultado obtido pelo método. Devido ao tempo computacional gasto para executar uma Programação Genética, e a necessidade destas várias simulações, procura-se métodos para paralelizar e distribuir as execuções para acelerar o processo de geração automática de aplicações em RSSF.

Um modelo de programação e *framework* que vem se destacando é o *MapReduce* (DEAN E GHEMAWAT, 2004). Esse framework possui módulos para realizar a distribuição de tarefas entre os nós de um *cluster* de máquinas, o balanceamento de carga, a tolerância a falhas, entre outras funcionalidades. Dessa forma, fica a cargo do programador somente identificar o que deve ser paralelizado e inserir o código nesse *framework*.

1.2. Objetivo

O principal objetivo deste trabalho é estudar e adaptar um algoritmo utilizando Programação Genética para geração automática de aplicações para rede de sensores sem fio usando o *framework MapReduce*, onde será feita a distribuição de execuções de diferentes instâncias do problema.

O presente trabalho tem como objetivos específicos:

- Estudar e implantar o *framework MapReduce* em um *cluster* de máquinas.

- Implementar uma aplicação de RSSF para o *framework MapReduce* que distribua execuções de uma Programação Genética.
- Avaliar o desempenho do *framework* quanto ao tempo de execução e qualidade das soluções geradas pela Programação Genética.

1.3. Justificativa

O uso de técnicas de paralelização vem se tornando necessário devido à quantidade de computações que diversos sistemas requerem. A Programação Genética requer diversas iterações para que seja encontrada uma boa solução, gerando um volume de computações que podem ser paralelizadas e várias execuções para avaliar com maior grau de confiança a qualidade o método implementado. O tempo gasto para gerar essas várias execuções em uma única máquina é alto. Havendo um ambiente composto por várias máquinas e um framework que facilite a divisão das execuções entre as diversas máquinas o tempo pode ser reduzido e até a qualidade das soluções melhorada. Uma das técnicas que podem ser utilizadas para paralelizar uma Programação Genética é a utilização de um *framework* de paralelização que abstraia do desenvolvedor os detalhes do ambiente computacional utilizado. *O framework Hadoop MapReduce* (WHITE, 2009) fornece uma abstração na paralelização de algoritmos, facilitando sua implementação, e foi projeto para executar grandes volumes de dados.

1.4. Metodologia

Neste trabalho foi feita uma pesquisa bibliográfica para obter os conhecimentos necessários para a realização do mesmo, sobre algoritmos genéticos, rede de sensores sem fio, programação paralela e distribuída e *MapReduce*.

Depois de estudados artigos e livros sobre *MapReduce*, foi decidido implementar um algoritmo no modelo *MapReduce* para distribuir as execuções de uma programação genética específica.

A pesquisa realizada é aplicada e quantitativa, pois foi realizada uma análise estatística sobre os resultados. A partir de então foi medido o tempo de execução do algoritmo, para verificar a eficácia do método de paralelização usado. Foram analisadas também as soluções geradas pelo algoritmo, verificando a eficácia da programação genética utilizada.

1.5. Organização do texto

O restante do texto está organizado da seguinte forma. O Capítulo 2 apresenta o referencial teórico. O Capítulo 3 apresenta um *framework* para geração automática de aplicações em rede de sensores sem fio utilizando programação genética. O Capítulo 4 apresenta o algoritmo desenvolvido, com detalhes da implementação. O Capítulo 5 apresenta detalhes dos experimentos e os resultados obtidos e o Capítulo 6 apresenta conclusões e trabalhos futuros.

2. REFERENCIAL TEÓRICO

Este capítulo apresenta conceitos relacionados à *MapReduce*, Rede de Sensores Sem Fio e Programação Genética

2.1. MapReduce

De acordo com (LIN *et al.*, 2010), *MapReduce* tem sua origem na programação funcional. Uma característica chave das linguagens funcionais é o conceito de funções de ordem superior, em outras palavras, funções que aceitam outras funções como argumento. Duas funções comuns construídas em funções de ordem superior são *map* e *fold*.

Dada uma lista de valores, a função *map* recebe como argumento uma função f (que possui um único argumento) e a aplica a todos os elementos da lista.

Dada uma lista de valores, a função *fold* recebe como argumentos uma função *g* (que possui dois argumentos) e um valor inicial. A função *g* é aplicada ao valor inicial e ao primeiro item na lista. O resultado obtido é armazenado em uma variável intermediária. O valor da variável intermediária e o próximo item na lista são usados como argumentos para a segunda aplicação de g e os resultados obtidos são armazenados em uma variável intermediária. Esse processo se repete até que todos os itens da lista tenham sido consumidos. A função *fold* retornará o valor final da variável intermediária.

As funções *map* e *fold* podem ser utilizadas em conjunto. Por exemplo, para computar a soma dos quadrados de uma lista de inteiros, a função *map* irá elevar ao quadrado cada valor da lista e posteriormente a função *fold* irá somar os resultados gerados pela função *map* juntamente com o valor inicial zero.

A parte superior do diagrama da Figura 2.1 ilustra o passo realizado pela função *map*, onde cada círculo do primeiro nível do diagrama representa um elemento da lista inicial e cada círculo do segundo nível representa o resultado intermediário gerado a partir da aplicação da função *f*. A função *g* recebida como argumento pela função *fold* é aplicada ao valor inicial e aos valores intermediários (gerados pela função *f*), produzindo os resultados representados pelos quadrados. A função *fold* irá então retornar como saída final o último valor intermediário.

Uma observação que (LIN *et al.*, 2010) faz é que a aplicação da função *f* pode ser paralelizada de uma forma direta, onde podem-se distribuir essas operações em máquinas diferentes de um cluster uma vez que cada uma dessas operações acontece de forma isolada. Já a operação *fold* tem mais restrições nas localidades dos dados, uma vez que os elementos devem ser reunidos para serem aplicados na função *g*. Entretanto, em muitos problemas a função *g* não tem necessidade de se aplicar a todos os itens da lista.

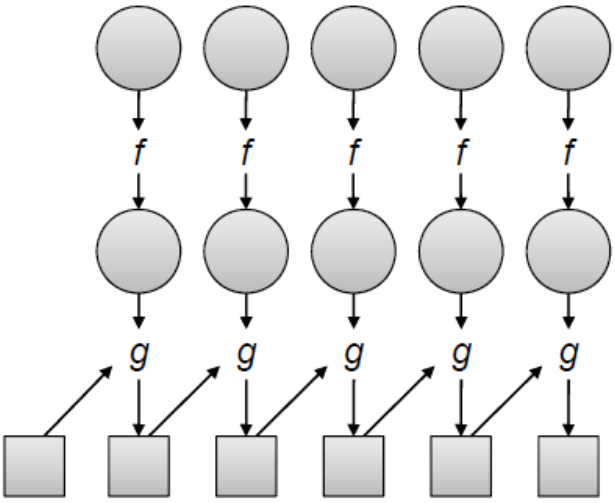

Figura 2.1: Funcionamento do conceito de funções de ordem superior (LIN *et al.*, 2010).

A fase *map* do MapReduce corresponde aproximadamente à operação *map* da programação funcional, enquanto que a fase *reduce* corresponde aproximadamente à operação *fold* da programação funcional.

Segundo (LIN *et al.*, 2010), pode-se definir *MapReduce* em três conceitos distintos, porém relacionados. Primeiro, *MapReduce* pode ser visto como um modelo de programação. Segundo, pode se referir como um *framework* de execução que coordena as execuções de programas escritos no modelo *MapReduce*. Terceiro, pode se referir a uma implementação de um *software* do modelo de programação e do *framework* de execução. A seguir são detalhados esses conceitos.

2.1.1 Modelo de programação *MapReduce*

Segundo (DEAN *et. al.,* 2004), no modelo de programação MapReduce a computação recebe um conjunto de pares no formato chave/valor como entrada e irá retornar um conjunto de pares também no formato chave/valor como saída. O usuário deverá escrever basicamente duas funções para realizarem essa computação: *Map* e *Reduce*.

A função *Map* é aplicada em toda a entrada do conjunto de dados, de forma paralela em diferentes nós de processamento, produzindo um conjunto de pares chave/valor intermediários que serão agrupados quanto à chave.

A função *Reduce* irá então receber esse pares intermediários agrupados, executar também em paralelo em diferentes nós de processamento, e gerar pares chave/valor finais, que serão a saída final da execução do programa.

Segundo (LIN *et al.*, 2010) os pares chave/valor formam a estrutura básica em *MapReduce.* As chaves e valores podem ser de tipos primitivos, como números inteiros, reais, *strings*, vetor de *bits*, etc., bem como podem ser estruturas mais complexas, como listas, tuplas, etc.

Segundo (DEAN *et. al.*, 2004) os tipos dos pares chave/valor das funções *map* e *reduce* são associados da seguinte forma:

map (c1,v1) →lista(c2,v2)
reduce (c2,lista(v2)) →lista(c3,v3)

Onde *map*, a partir dos pares chave c1 e valor c1, retorna uma lista de pares chave c2 e valor v2. A função *reduce* irá então, a partir desses pares agrupados quanto à chave c2, retornar a saída do programa que será uma lista de pares de chave c3 e valor v3.

Há dois elementos adicionais que completam o modelo de programação *MapReduce*: *partitioners* e *combiners* (DEAN *et. al.*, 2004), estes porém, não precisam, necessariamente, serem implementados pelo programador.

Os *partitioners* têm como objetivo classificar dados gerados pela função *map* que possuem a mesma chave antes de serem enviados para a execução da função *reduce*. Desta forma, dados que contêm a mesma chave serão enviados apenas a uma máquina, evitando fragmentação das informações.

Os *combiners* são responsáveis em agrupar valores de dados que contém a mesma chave, porém este passo é feito localmente, diminuindo a quantidade de dados que será trafegado pela rede do *cluster*. Este processo é feito antes da saída da função *map*.

A Figura 2.2 mostra o modelo do *MapReduce*, ilustrando o funcionamento de todos os elementos do modelo, *mapper, reducer, partitioners* e *combiners*. Primeiramente, a entrada é dividida e são realizadas várias tarefas *mapper*, que irão retornar pares chave/valor intermediários. A etapa *combiner* é então iniciada, agrupando valores de mesma chave localmente. No segundo *combiner* da figura verifica-se esse processo, onde os valores das duas ocorrências de c (3 e 6) são agrupados, gerando o único valor 9 . A etapa

partitioner é feita, classificando os pares intermediários de acordo com a chave, e também os ordena em relação à chave e os envia à próxima etapa, *reduce*, que irá processar o que foi implementado e retornar a saída final. Por exemplo, a chave a apareceu como saída do primeiro e do terceiro mapper. Os seus valores 1 e 5 são agrupados para serem enviados para o mesmo *reducer*.

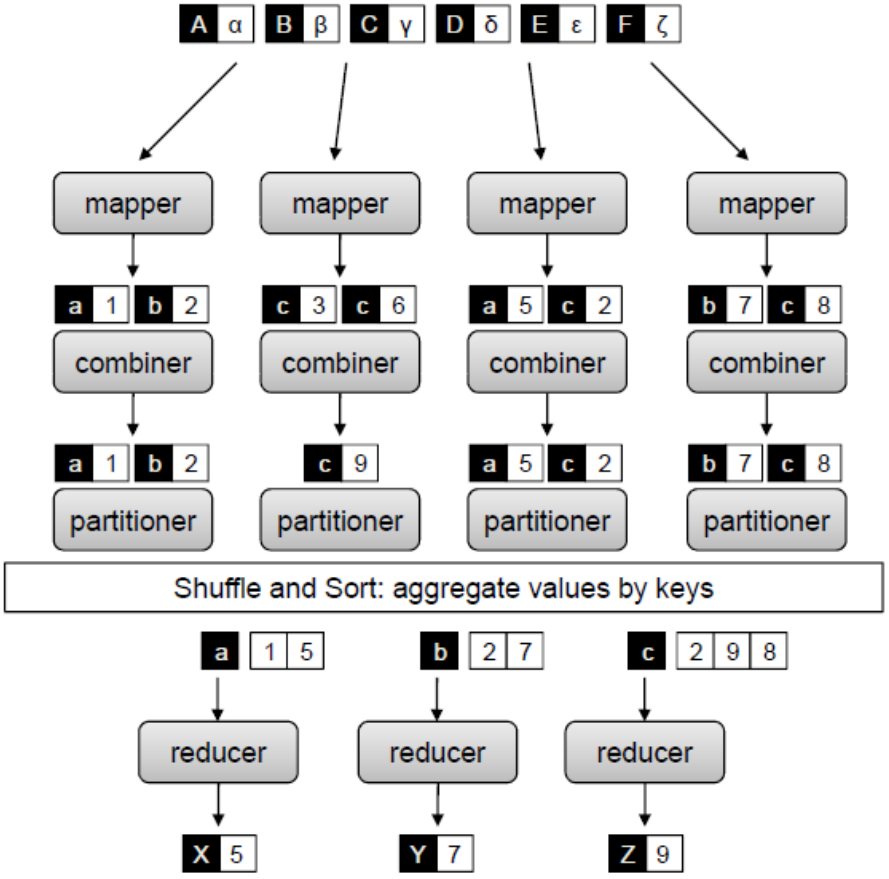

Figura 2.2: Funcionamento do modelo de programação *MapReduce* (LIN et al, 2010).

2.1.2 *Framework* de execução *MapReduce*

Existe a necessidade de saber como é feita a distribuição de processamento por trás do *MapReduce*. Um programa *MapReduce* é referido como *job*, e consiste do código que contém *mappers*, *reducers*, *combiners* e *partitioners*, e das configurações de parâmetros (LIN *et al.*, 2010). O desenvolvedor submete o *job* a um nó específico do *cluster* (esse nó é responsável em submeter o trabalho ao *cluster*). O *framework* é responsável por manter a transparência da paralelização e da divisão de dados de entrada para ser feita a distribuição.

Segundo (LIN *et al.*, 2010), o *framework* é responsável em dividir o *job* em partes menores, chamados *tasks*, e agendar cada tarefa *task* para as máquinas do *cluster*. Por exemplo, uma tarefa *task* de *map* é responsável por processar um determinado bloco de entrada. Em muitos casos o número das tarefas *task* é maior que o número de máquinas em um *cluster*, então o *framework* é responsável em definir quais tarefas serão realizadas primeiro, e distribuir novas tarefas às máquinas que terminarem alguma tarefa.

Uma característica importante do *framework* é que ele evita excessivo tráfego de dados entre as máquinas do *cluster*. Isso ocorre distribuindo o código de execução para as máquinas que contêm dados que não foram processados, iniciando uma *task* naquela máquina. Caso a máquina não possa iniciar uma nova *task*, se ela estiver executando muitas *tasks* por exemplo, o *framework* inicia a nova task em outra máquina, movendo dados pela rede do *cluster*.

O *framework* deve estar apto a lidar com erros e tratamentos de falhas, uma vez que ele executa em *clusters* de inúmeras máquinas, e falhas em *hardware* são comuns nestes ambientes. Além disso, dificilmente o programador irá cobrir erros de dados corrompidos no conjunto de dados, pois este conjunto de dados pode ser enorme.

2.1.3 *Hadoop MapReduce*

Existem algumas implementações baseadas no modelo de programação *MapReduce* e no *framework MapReduce*, como por exemplo a implementação de propriedade da *Google* e a implementação de código aberto *Hadoop* em Java (WHITE, 2009). Neste tópico é descrita a versão da implementação *Hadoop*, uma vez que foi esta utilizada neste trabalho. Utilizou-se o *framework Hadoop MapReduce* pois ele é de código aberto, vem sendo bastante utilizado fornecendo bons artigos e livros para referências.

Segundo (WHITE, 2009), o processo do *Hadoop MapReduce* funciona através de quatro entidades independentes: *JobClient, JobTracker, TaskTracker* e um sistema de arquivos de distribuídos.

O *JobClient* é o responsável por submeter o *MapReduce job.* O *JobTracker* é responsável por coordenar a execução do *MapReduce job* e o *TaskTracker* é responsável em executar as *tasks* do *MapReduce job*. Um sistema de arquivos distribuídos (será definido na próxima seção) é responsável por compartilhar os arquivos entre as entidades e as máquinas do *cluster*. A Figura 2.3 de (WHITE, 2009) ilustra o processo de execução da implementação do *Hadoop MapReduce*.

A partir do programa *MapReduce,* cria-se um *JobClient* que irá submeter um trabalho *job* ao *framework* (Passo 1). O *JobClient* irá então solicitar a um nó *JobTracker* um novo *job* ID, verificando as configurações de execução, por exemplo verificando diretórios de entrada e saída (Passo 2). O *JobClient* solicita ao sistema de arquivos distribuídos os recursos necessários para a execução, como arquivos de configuração, o arquivo JAR e os arquivos de entrada já divididos (Passo 3). A partir daí o *JobClient* submete ao *JobTracker* o *job* a ser executado (Passo 4). O *JobTracker* irá então iniciar o *job* (Passo 5). A inicialização consiste em criar um objeto que representa que foi iniciada a

execução do *job*, encapsula suas *tasks* e mantém as informações sobre as execuções das *tasks*. A lista de *tasks* a ser executada é criada a partir da entrada do sistema de arquivos dividida (Passo 6), e é criada uma *map task* para cada divisão, gerando várias *tasks*. A comunicação entre o *TaskTracker* e o *JobTracker* é feita através de um método que envia seu *status* periodicamente, chamado de *Heartbeat* (Passo 7). O *TaskTracker* irá então, através do sistema de arquivos, recuperar os arquivos necessários para executar a tarefa *task* que foi submetida (Passo 8). Com os arquivos necessários inicia-se a máquina virtual Java (Passo 9) que irá executar a tarefa (passo 10).

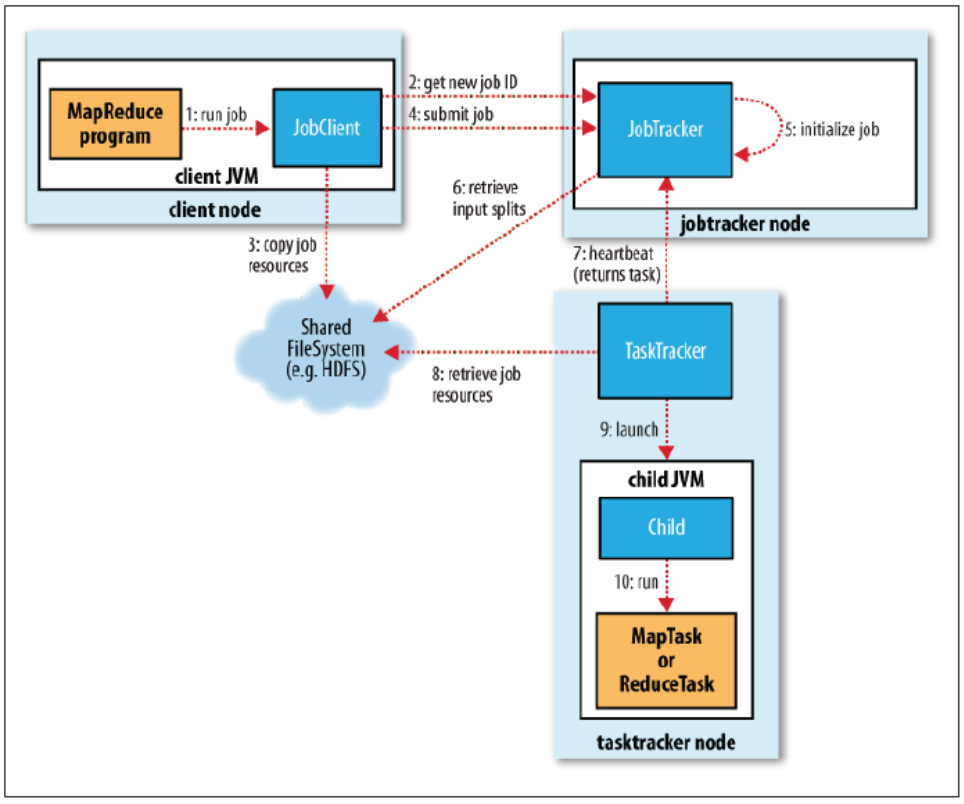

Figura 2.3: Execução de *job MapReduce* (WHITE, 2009).

2.2 HDFS

Hadoop Distributed Filesystem (HDFS) é um sistema de arquivos distribuídos para armazenar grandes quantidades de arquivos com transmissão de dados utilizando padrões de acesso, que pode ser executado em máquinas comerciais (WHITE, 2009).

Em um *cluster* HDFS existem dois tipos de nós operando no padrão mestre/escravo. Em uma máquina mestre opera o *namenode*, e nos escravos opera o *datanode*. O *namenode* é responsável em gerenciar o sistema de arquivos. O *namenode* mantém informações sobre quais *datanodes* os blocos de um determinado arquivo estão localizados. O *datanode* é responsável em armazenar e obter os blocos de dados definidos pelo *namenode*.

A Figura 2.4 de (LIN *et. al*, 2010) ilustra a arquitetura completa de um *cluster Hadoop*. Uma máquina é responsável pelo *namenode* e pelo *jobtracker*, que irão gerenciar as execuções nas máquinas escravas, que irão executar *tasktracker* e *datanode* em sistema de arquivos *Linux*.

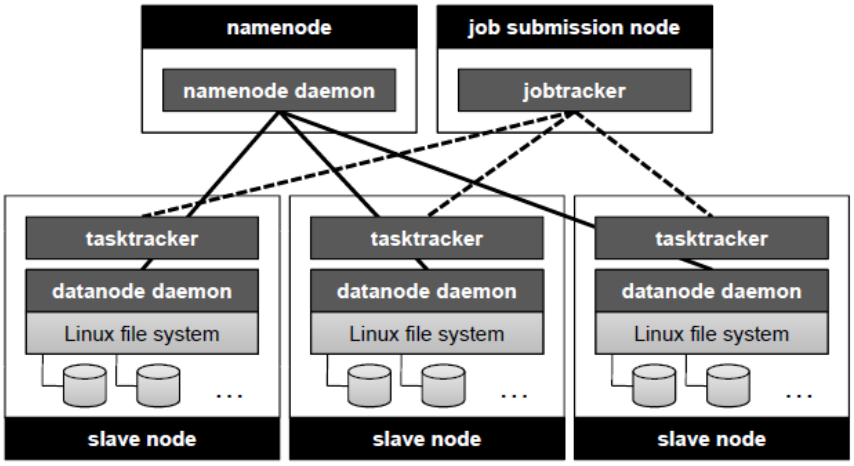

Figura 2.4: Arquitetura do *Hadoop MapReduce* (LIN *et. al*, 2010).

2.3 Redes de sensores sem fio

Segundo (LOUREIRO *et al*., 2003) uma rede de sensores sem fio (RSSF) é uma rede composta de vários elementos distribuídos denominados nodos (ou sensores). Essas redes diferem das redes tradicionais de computadores, pois elas possuem restrições de energia, devem possuir mecanismos de auto-configuração e adaptação, pois elas estão mais sujeitas a perdas de nodos e falhas em comunicação. Uma rede de sensores sem fio tende a ser mais autônoma e requer um alto grau de cooperação entre os nodos para executar uma tarefa definida para a rede. Isso requer que algoritmos distribuídos devam ser revistos para que se possa utilizá-los nesse tipo de rede.

A Figura 2.5 de (LOUREIRO *et al*., 2003) mostra os seguintes componentes básicos de um nodo sensor: transceptor, memória, bateria, processador, sensor. O transceptor é responsável pela comunicação entre os nodos. Segundo Loureiro *et al*. (2003) os transceptores têm uma largura de banda de 1kbits/s a 1Mbits/s e normalmente uma capacidade de memória de 128 kbytes a 1 Mbyte. A bateria varia de acordo com a necessidade da RSSF, considerando condições de clima, volume e condição inicial. Assim como a bateria há diversos processadores que poderão ser escolhidos de acordo com a necessidade da RSSF. Quanto aos sensores, um nodo pode apresentar mais de um sensor, podendo ser acústico, sísmico, infravermelho, vídeo-câmera, calor, temperatura e pressão.

Loureiro *et al.* (2003) descreve várias áreas que redes de sensores sem fio têm o potencial de serem empregados, como controle, ambiente, tráfego, segurança, medicina, militar.

Loureiro *et al.* (2003) apresenta também tarefas típicas de uma rede de sensores sem fio. Apesar de o objetivo de uma RSSF depender da aplicação, algumas tarefas são frequentemente encontradas nesse tipo de rede. As principais são: determinar o valor de algum parâmetro num dado local, detectar a ocorrência

de eventos de interesse e estimar valores de parâmetros em função do evento detectado, classificar um objeto detectado, rastrear um objeto.

A complexidade inerente de uma RSSFs dificulta a sua programação. Topologia dinâmica dos nós, restrição de recursos, falhas, programação de baixo nível e a complexidade inerente de um sistema massivamente distribuído impõem barreiras para o desenvolvimento de aplicações eficientes e livres de erros para esse tipo de rede. O suporte ao desenvolvimento de aplicações é extremamente necessário com a crescente demanda de programas resolvendo problemas complexos executando sobre RSSFs.

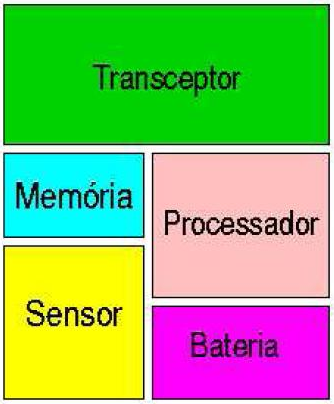

Figura 2.5: Hardware básico de um nodo sensor (LOUREIRO *et al*., 2003).

2.3.1. *Middlewares* para Rede de sensores sem fio

O grande desafio da programação de RSSFs advém da lacuna existente entre a especificação funcional de alto nível e a complexidade da infraestrutura subjacente. Isso motiva a busca por mecanismos que possam simplificar a programabilidade dessas aplicações, escondendo a complexidade mencionada

(WANG M. M., 2008). A complexidade supra mencionada se opõem diretamente aos requisitos das aplicações, como flexibilidade, reusabilidade e confiabilidade.

Assim, aplicações cooperativas de sensoriamento requerem um modelo de programação com abstrações adequadas para seu desenvolvimento. Na literatura diversos *middlewares* foram propostos a fim de prover essas abstrações, suportando os requerimentos de uma RSSFs (RÖMER AND MATTERN, 2004).

Uma característica importante que muitos m*iddlewares* costumam tratar é a reprogramação dinâmica da rede de sensores como um todo, um agregado e não somente como um conjunto de nós individuais. Isso significa que um usuário, conectado a rede em um ponto arbitrário, pode injetar tarefas para a rede de sensores executarem sem a intervenção humana (BOULISA *et. al*., 2007). Essas instruções podem ser declarativas, onde uma requisição da leitura de sensores é repassada para o sistema, que retorna as informações requisitadas acessando os nós corretos, agregando os dados e retornando ao usuário (YAO E GEHRKE, 2002). Pode também ser na forma de uma requisição imperativa, ou seja, um código em uma linguagem script de alto nível é interpretado pelo módulo de execução instalado nos nós sensores (BOULISA *et. al*., 2007). A ideia é que a linguagem *script* tenha poder suficiente para expressar diferentes aplicações distribuídas e ao mesmo tempo esconda os detalhes de baixo nível da rede.

Embora permitam uma maior expressividade escondendo os detalhes de baixo nível, *scripts* escritos em um paradigma imperativo ainda requerem a programação de um sistema distribuído, onde múltiplas interações entre os nós sensores podem ter comportamento imprevisto, complicando a programação. O desenvolvedor não precisa desenvolver um código específico para cada nó sensor, mas ainda necessita planejar a colaboração entre os diversos nós de uma forma explícita. O uso da Programação Genética para a construção de um m*iddleware* para RSSF é útil para otimizar os *scripts* que serão executados pelos nós da rede de sensores sem fio.

2.4. Programação Genética

Algoritmos Genéticos (AGs) são métodos de computação evolutiva inspirados em processos biológicos para conduzir a busca no espaço de soluções, (HOLLAND, 1975). Algoritmos Genéticos são geralmente usados como técnicas de otimização na busca do ótimo global de funções. Contudo este não é o seu único domínio de aplicação. Dois importantes domínios de aplicação alternativos para AGs são aprendizado de máquina baseados em algoritmos genéticos (AMBAG) e programação genética (PG), (SETTE E BOULLART, 2001).

A Programação Genética é um Algoritmo Genético (AG) aplicado a uma população de programas de computador, mas a PG aplica-se na otimização de estruturas muito mais complexas e pode, portanto, ser aplicado a uma maior diversidade de problemas. A representação de programas, ou geralmente estruturas, na PG tem uma forte influência sobre o comportamento e a eficiência do algoritmo. Na PG começa-se com uma descrição de alto nível de "o que precisa ser feito" e executa automaticamente um processo iterativo em uma tentativa de criar um programa de computador que faz o que é exigido (KOZA 1992).

Na programação genética, as populações de centenas ou milhares de programas de computador são geneticamente criadas. Esta criação é feita usando o princípio Darwiniano de sobrevivência e reprodução dos mais aptos, juntamente com uma recombinação genética (cruzamento) de operação adequada para o acasalamento de programas de computador. Um programa de computador que resolve (ou aproximadamente resolve) um determinado problema pode surgir a partir desta combinação de seleção natural Darwiniana e operações genéticas, (KOZA, 1992).

A Programação Genética inicia gerando uma população de programas de computador aleatoriamente compostos de funções e variáveis apropriadas para o problema a ser resolvido.

Cada um dos indivíduos da população é avaliado em termos de quão bem é seu desempenho no meio do problema em particular. Esta avaliação é chamada de avaliação *fitness*.

O princípio Darwiniano de reprodução e sobrevivência dos mais aptos e a operação de recombinação genética (*crossover*) são usados para criar uma nova população de programas individuais da população atual de programas. A operação de reprodução envolve a seleção, na proporção do *fitness*, de um programa da população atual dos programas para ser copiada para a próxima população, permitindo a sua sobrevivência. A operação de recombinação de dois indivíduos envolve a seleção, na proporção do *fitness*, onde são selecionadas, aleatoriamente, partes significativas dos pais para gerar uma prole que poderá ser ainda mais apta a resolver o problema, (KOZA, 1992).

Depois das operações de reprodução e de recombinação genética, a população atual é trocada pela nova população, então essa nova população é avaliada e o processo de reprodução e recombinação é repetido (KOZA, 1992).

Normalmente, a programação genética gera programas executando os seguintes passos:

(1) Gerar uma população de programas aleatórios com as funções e variáveis para o problema.
(2) Fazer uma iteração dos seguintes passos até que um critério de parada seja satisfeito:
    a. Executar cada programa e avaliar o quão bem este programa resolve o problema.

b. Criar uma nova população aplicando as duas seguintes operações primárias. Elas são aplicadas em programas escolhidos da atual população de acordo com uma probabilidade baseada no *fitness*.
    i. Copia o programa escolhido para a próxima população.
    ii. Cria um novo programa pela recombinação de partes de dois programas já existentes na população atual.

(3) O melhor programa encontrado em cada geração é designado como resultado da programação genética. Este resultado pode ser uma solução (ou uma solução aproximada) para o problema.

Koza (1992) diz que as operações genéticas são simples e não consumem muito tempo enquanto que a avaliação do *fitness* de cada indivíduo da população é tipicamente complicada e demorada. Isto motiva a busca de abordagens de paralelização em programação genética.

2.5. Trabalhos relacionados

Na literatura são encontrados estudos sobre o uso de MapReduce em algoritmos genéticos e programação genética. Jin C. et al (2008) apresentam uma adaptação do *MapReduce* para Algoritmos Genéticos Paralelos. A função *map* proposta recebe como chave o índice do individuo e valor o próprio individuo e executa a avaliação de *fitness* de cada individuo. Apresenta duas funções *reduce*. A primeira faz uma operação de seleção no ótimo local, e a segunda função *reduce* faz a operação de seleção no espaço global do problema a partir dos pares entrada gerado pelo primeiro *reduce*. Huang D. *et al* (2010) apresentam o uso do MapReduce para escalar populações de algoritmos genéticos para o problema do agendamento de lojas de trabalho. É implementado uma função *map* que distribui a avaliação do *fitness* e uma função *reduce* que faz a seleção e as operações de seleção e reprodução.

Em relação ao uso de algoritmos genéticos para resolver problemas em rede de sensores sem fio, Ferentinos e Tsiligiridis (2007), apresentam uma topologia da rede de sensores em que ela é otimizada utilizando um AG para determinar a potência de cada um dos rádios visando a redução do gasto de energia. Uma construção de *clusters* de redes móveis Ad hoc guiada por um AG é apresentada por Turgut et al. (2002). Nesse algoritmo, parâmetros como distância dos nós membros até o *clusterhead*, velocidade média dos nós e tempo de existência do *cluster* são utilizados na função de *fitness*. No trabalho de Bhondekar et al. (2009), a densidade da rede, conectividade e consumo de energia são otimizados utilizando um AG. O modo operacional dos nós sensores juntamente com esquemas de clusterização e potência de transmissão são escolhidos de forma a reduzir o consumo de energia.

PG aplicada à geração de protocolos distribuídos (não específico para RSSFs) é apresentada por Weise e Zapf (2009). Os métodos propostos foram utilizados para evolução de um algoritmo de eleição. O método denominado programação genética padrão com memória utilizou-se de PG baseada em genomas organizados em árvore. O código implementado é formado por vários procedimentos, entre eles um inicializador e um procedimento chamado no recebimento de mensagem. Uma memória local e uma global estavam à disposição dos procedimentos. Mensagens poderiam ser enviadas em broadcast para todos os nós vizinhos com uma expressão. Uma variante com memória indexada e declaração de memória também é apresentada. O artigo também apresenta uma variante que utiliza programação genética linear, além de uma variante que usa a linguagem *Fraglets*. Ainda o mesmo artigo apresenta duas versões de PG baseadas em regras, com diferença no tipo de memória utilizada. No artigo os resultados da aplicação de cada um dos métodos utilizando uma rede com topologia linear é apresentado.

3. PROGRAMAÇÃO GENÉTICA PARA REDE DE SENSORES SEM FIO

Esta seção apresenta um *framework* para geração de aplicações em RSSF composto por três camadas: um simulador de RSSF, um *middleware* que fornece uma linguagem *script* para programação em alto nível de aplicações para RSSF e um método baseado em Programação Genética para gerar automaticamente aplicações para este *middleware*, desenvolvida em (DE OLIVEIRA, 2013).

O componente principal desse sistema é o *middleware* que é responsável pela execução dos programas escritos na linguagem *script* em cada nó sensor. Para isso uma máquina virtual é implementada no *middleware*. A Figura 3.1 adaptada de (DE OLIVEIRA, 2013) apresenta esse cenário. Os nós sensores são distribuídos em uma área a ser monitorada executam tarefas que são codificadas na linguagem *script*. O *script* é executado por uma máquina virtual que é parte do *middleware*. Cada nó sensor da rede executa uma cópia local do *middleware*. O código *script* contém todas as informações necessárias para executar a tarefa especificada pelo usuário. O código é inserido na rede, replicado a todos os nós e executados pela máquina virtual.

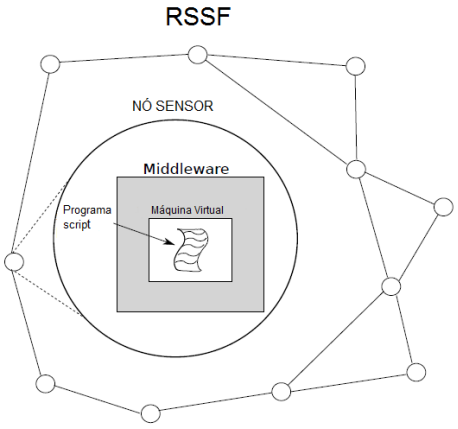

Figura 3.1: RSSF executando o *middleware*

O trabalho de (DE OLIVEIRA, 2013) inova ao apresentar um método de geração do código *script*, onde uma abordagem para geração automática do código baseada em PG é apresentada. A Figura 3.2 adaptada de (DE OLIVEIRA, 2013) mostra esse processo. O usuário tem apenas a tarefa de descrever uma função objetivo, que será utilizada para avaliar os *scripts* candidatos gerados pela PG (Figura 3.2 (a)). A função objetivo corresponde a uma função de *fitness* que leva em consideração os parâmetros da rede relacionados a custos de eventos perdidos, custo do número de mensagens circuladas na rede, custo de mensagens e ações realizadas prematuramente, custo de mensagens enviadas na direção errada e custo da distância do evento ocorrido em relação ao nó *sink*. O detalhamento dessa função *fitness* encontra-se no trabalho (DE OLIVEIRA, 2013).

A geração automática do código pela PG é mostrada em (Figura 3.2 (b)), onde um conjunto de populações é gerado e cada população contém um conjunto de indivíduos. Cada indivíduo é um *script* que pode estar no *middleware* instalado na RSSF. A ideia principal é desenvolver um conjunto inicial de *scripts* aleatórios para comporem um programa que possa eficientemente resolver uma tarefa desejada pelo usuário.

Em cada geração, cada *script* da população é avaliado usando a função objetivo definida pelo usuário. Uma vez que as avaliações utilizando uma rede de sensores física para cada solução candidata não é praticável, um módulo de simulação foi implementado nesse estágio. O objetivo é que esse módulo faça uma simulação significante, porém o mais rápido possível.

Inicialmente cada *script* candidato é copiado para todos os nós sensores dentro do simulador. Os nós no simulador têm instalado uma versão do *middleware*. Então, a simulação é executada e a função objetivo é usada para avaliar o desempenho do *script*.

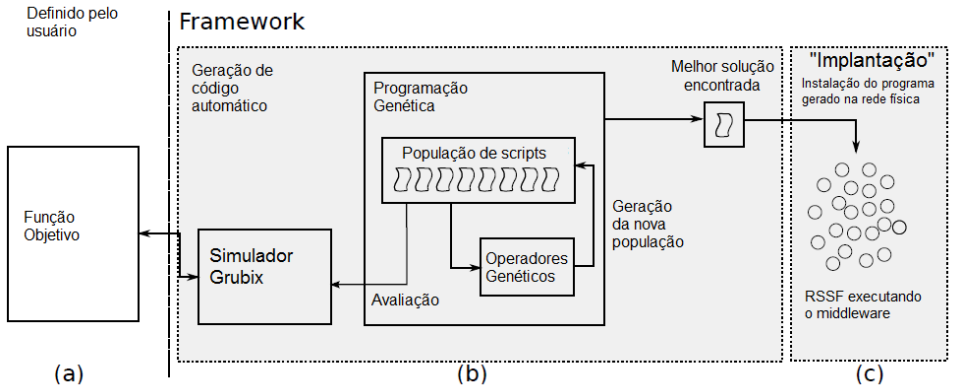

Figura 3.2: Visão geral do *framework*

Depois de todos os *scripts* da população ser avaliados uma nova geração de *scripts* é criada através dos operados genéticos, que combinam *scripts* da atual população (baseado na avaliação dos indivíduos). Assim, uma nova geração é criada. O processo é iniciado novamente com a nova população de indivíduos. Esse processo de evolução é realizado por um determinado período de tempo ou até que um determinado valor de *fitness* é encontrado, assim o melhor individuo encontrado será selecionado para a implantação real. A Figura 3.2 (c) ilustra a instalação final do *script* gerado nos nós sensores reais. Cada nó executado uma versão local do *middleware* que é responsável pela execução do código *script*.

3.1. Representação dos indivíduos

O indivíduo é representado como um programa de computador com *triggers* e comandos que serão executados pelo nó sensor. O propósito dessa representação dos indivíduos é estruturar um vetor de *triggers*, onde cada *trigger* é composta por um cabeçalho de instrução e uma lista de comandos. O cabeçalho de instrução é uma expressão booleana composta de dois termos e uma operação

lógica. Apenas os operadores *AND* e *OR* foram utilizados. Os termos podem ser eventos ativos representados por A1, A2, ..., A*n*, e eventos passivos com P1, P2, ..., P*m*.

Há três tipos de comandos: *up(<event>)*, *down(<event>)* e *send(<event>, <direction>)*. Os dois primeiros eventos modificam eventos passivos no nó sensor. O comando *send* modifica eventos passivos em um nó vizinho enviando uma mensagem. A Figura 3.3 mostra um possível indivíduo.

P1 **and** A3	A1 **or** A2	P3 **and** P2
send(P2,→)	down(P1)	up(P2)
down(P3)	up(P2)	send(P1,↓)
	send(P3,↑)	down(P1)
	down(P1)	

Figura 3.3: Possível indivíduo na PG (DE OLIVEIRA, 2013).

O comando *up* torna um evento passivo ligado e o comando *down* torna um evento passivo desligado na memória de eventos. O comando *send* recebe um evento passivo e uma direção para o nó destino. As possíveis direções são para cima (↑), para baixo (↓), para esquerda (←) e para direita (→). Este comando troca informações entre os nós sensores, tornando ligado o evento passivo no nó sensor da direção indicada. Nas redes de topologia em grade é fácil determinar qual vizinho está acima, abaixo, à esquerda e à direita. Entretanto, na topologia randômica pode haver mais de um vizinho na mesma direção. Quando isso ocorre o *middleware* envia a mensagem para o vizinho mais próximo naquela direção.

Por exemplo, na Figura 3.3, caso a condição A1 **or** A2 ocorra, os comandos *down*(P1), *up*(P2), *send*(P3, ↑) e *down*(P1) são executados.

A representação do individuo é periodicamente avaliada pela máquina virtual que lê os eventos e interpreta as várias ações codificadas no programa. A

mesma representação é usada por todos os nós sensores na rede, mas a memória de eventos em cada nó tem diferentes configurações sobre a simulação da rede. Logo, diferentes ações podem ser executadas por cada sensor usando o mesmo programa.

Os indivíduos são inicializados aleatoriamente considerando o domínio dos parâmetros usados na representação. O *fitness* é determinado quando o programa é executado na RSSF. Por isso, a representação é copiada para todos os nós sensores e executada pela máquina virtual em cada sensor. Todas as execuções nos sensores fornecem o comportamento da RSSF. Se o comportamento da rede alcança eficientemente o objetivo, um valor baixo de *fitness* é atribuído ao individuo, caso contrario um valor alto é atribuído.

3.2. Programação Genética

Esta subseção descreve a programação genética que compõe o framework proposto por (DE OLIVEIRA, 2013). É um algoritmo evolucionário baseado no algoritmo genético clássico com elitismo. A Figura 3.4 apresenta o pseudocódigo que De Oliveira (2013) propõe.

Inicialmente, o método inicializa aleatoriamente a população de indivíduos (Linha 2). Então os indivíduos são avaliados pelo simulador da RSSF (Linha 3), esta simulação é realizada por um módulo que simula o comportamento da RSSF. Este comportamento é guiado pelo *script* que cada indivíduo representa.

Um contador para a atual geração é inicializado como primeira população (Linha 4), então a evolução da população inicia (laço que vai das linhas 5 a 19). Este laço irá repetir até que o contador de gerações alcance um parâmetro que define o total de gerações.

```
1  begin
2      initialize(population);
3      simulationAndEvaluation(population);
4      actualGeneration ← 1;
5      while actualGeneration < totalGenerationsNumber do
6          matingPool ← ∅;
7          insert(bestIndividual(population), matingPool);
8          for i ← 2 to numberOfIndividuals do
9              (parent1, parent2) ← tournament(population);
10             if probability(crossoverRate) then
11                 newIndividual ← crossover(parent1, parent2);
12             else
13                 newIndividual ← parent1;
14             if probability(mutationRate) then
15                 newIndividual ← mutation(newIndividual);
16             insert(newIndividual, matingPool);
17         simulationAndEvaluation(matingPool);
18         population ← matingPool;
19         actualGeneration ← actualGeneration +1;
```

Figura 3.4: Pseudocódigo da Programação Genética (DE OLIVEIRA, 2013)

Uma variável irá receber os melhores indivíduos de cada geração, aplicando assim o conceito de elitismo. Inicialmente, essa variável está vazia (Linha 6), o melhor indivíduo da atual população é então inserido nesta variável (Linha 7). É iniciado então um laço que irá gerar novos indivíduos (linhas de 8 a 16), baseado em combinações dos indivíduos já existentes.

Os indivíduos selecionados para os operadores genéticos são selecionados por um torneio (Linha 9). O torneio seleciona o melhor indivíduo a partir de alguns indivíduos selecionados randomicamente. O número de indivíduos selecionados aleatoriamente é definido por um parâmetro da PG.

A seguir um teste de probabilidade é aplicado no parâmetro de taxa de *crossover*. Caso o teste obtenha sucesso, um novo indivíduo é obtido por um operador de *crossover* a partir dos indivíduos selecionados. Caso contrário, o primeiro indivíduo selecionado será o novo indivíduo (Linhas 10 a 13).

Depois da criação do novo indivíduo, um teste de probabilidade é aplicado no parâmetro de taxa de mutação da PG. Caso o teste obtenha sucesso um operador de mutação é aplicado ao novo indivíduo (Linhas 14 a 15).

Depois de criar uma nova população de mesmo tamanho da população inicial, esta população é simulada e avaliada (Linha 17). A seguir então, a nova população é copiada para a população principal (Linha 18), descartando a população inicial. O contador é então incrementado nesse ponto (Linha 19).

No final da execução da PG o melhor indivíduo encontrado é retornado como a melhor aplicação encontrada para o problema em RSSF.

3.3. Operadores Genéticos

Um total de 4 operadores de *crossover* e 7 operadores de mutação foi proposto por (DE OLIVEIRA, 2013) levando em conta os vários *triggers* e comandos em cada representação. As Figuras 3.5 e 3.6 ilustram os operadores *crossover*.

A Figura 3.5 (a) ilustra o primeiro operador *crossover*, onde o novo indivíduo é criado por um ponto de corte nos *triggers*. O novo indivíduo irá herdar todas as *triggers* de um pai até o ponto de corte e herdar do outro pai as *triggers* depois do ponto de corte.

O segundo operador *crossover* é ilustrado na Figura 3.5 (b), este também é baseado em um ponto de corte porém este corte é feito nos comandos de cada *trigger*. Os cabeçalhos das *triggers* são escolhidos aleatoriamente.

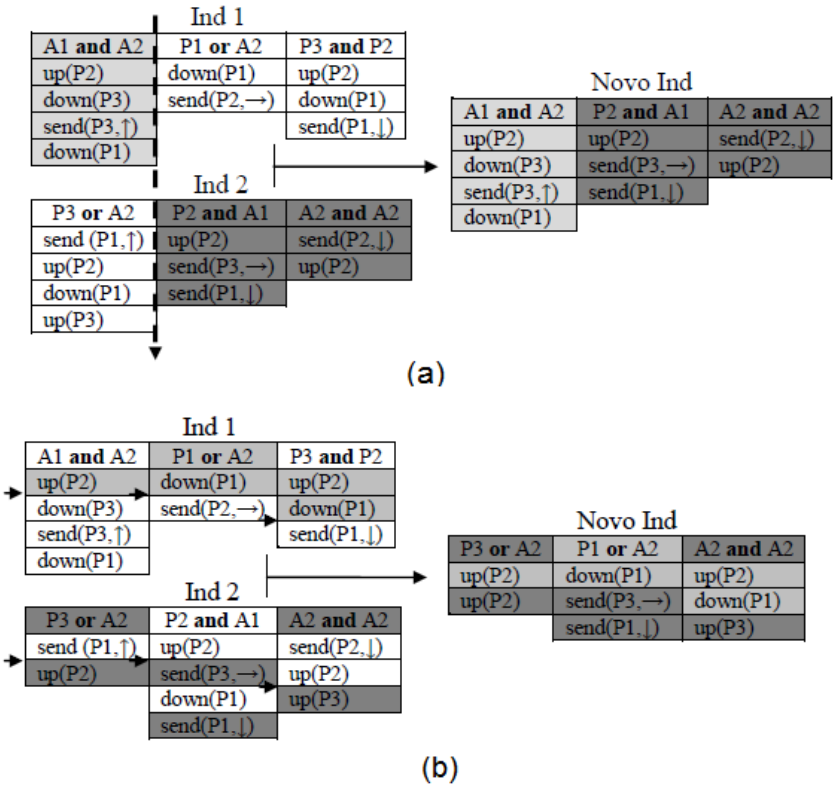

Figura 3.5: Tipos de operadores *crossover* (DE OLIVEIRA, 2013).

O terceiro operador é mostrado na Figura 3.6 (a), onde cada *trigger* do novo indivíduo é herdado de um pai escolhido de forma aleatória.

O ultimo operador, Figura 3.6 (b), trabalha de forma semelhante ao operador anterior, a diferença é que este compara cada comando em cada *trigger* selecionando um comando de um dos pais, se um dado *trigger* de um pai tem muitos comandos, os comandos excessivos ainda têm 50% de chance de serem incluídos na *trigger* do novo filho.

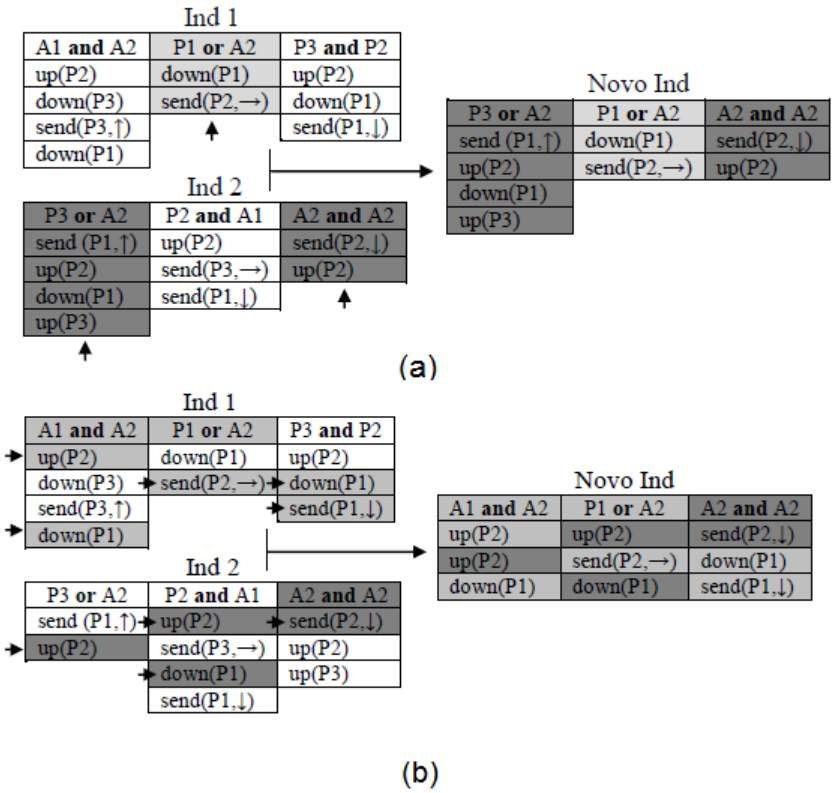

Figura 3.6: Tipos de operadores *crossover* (DE OLIVEIRA, 2013).

Um total de sete mutações é utilizado:

- Substitui Comando: sorteia um comando e troca este por outro comando gerado aleatoriamente.
- Reinicia Comandos: faz o mesmo que Substitui Comando, mas para todos os comandos de um mesmo *trigger*.
- Remove & Insere: sorteiam um comando em um *trigger* e insere em uma posição aleatória em outro *trigger*.
- Troca Ordem: sorteiam dois comandos dentro de um *trigger* e troca suas posições.

- Comando: sorteia um comando e modifica os dados internos como eventos e destinos.
- Troca *Triggers:* troca a ordem de dois *triggers* sorteados aleatoriamente.
- Cabeçalho: modifica os dados internos do cabeçalho de um *trigger* sorteado, alterando os operadores e operando.

4. ALGORITMO PROPOSTO

A paralelização e/ou distribuição de uma PG pode ser realizada usando duas abordagens principais: paralelizando o cálculo da função de *fitness* (parta da PG mais custosa computacionalmente) ou distribuindo as várias execuções da PG entre diferentes máquinas.

A primeira abordagem foi utilizada por (JIN C. *et. al*, 2008) e (HUANG D. *et. al*, 2010). (JIN C. *et. al*, 2008) propõem paralelizar o cálculo do *fitness* distribuindo a função de cálculo de *fitness* entre as máquinas do *cluster*. Em (HUANG D. *et. al*, 2010) é realizada uma paralelização do cálculo do *fitness*, onde a função *map* é implementada para realizar o cálculo da função *fitness* de cada indivíduo da população. Enquanto que a função *reduce* é implementada para gerar novas gerações da população. Uma função *map* adicional é implementada para gerar uma população inicial.

A segunda abordagem foi utilizada no algoritmo proposto neste trabalho.

A Figura 4.1 fornece uma visão geral do algoritmo proposto. O funcionamento consiste em distribuir as execuções do *framework* de RSSF para geração automática de aplicações para RSSF em *cluster*. Depois das execuções serem realizadas são feitos cálculos sobre os resultados obtidos emitindo então o valor do menor *fitness* obtido, o maior *fitness* obtido, a média dos *fitness* obtidos, o valor do desvio padrão dos *fitness* e o coeficiente de variação dos *fitness* obtido. A realização desses cálculos estatísticos facilita a coleta dos resultados e consequentemente sua análise.

O funcionamento do algoritmo proposto utilizando o modelo de programação *MapReduce* consiste em receber instâncias do problema e distribuir as execuções no *cluster* através da implementação da função *map*. As instâncias são arquivos no formato XML, que apresenta as características de uma RSSF,

contendo o número de nós, a topologia da rede, o nó sensor *sink*, o qual será simulado pelo módulo de simulação do *framework* descrito na seção anterior.

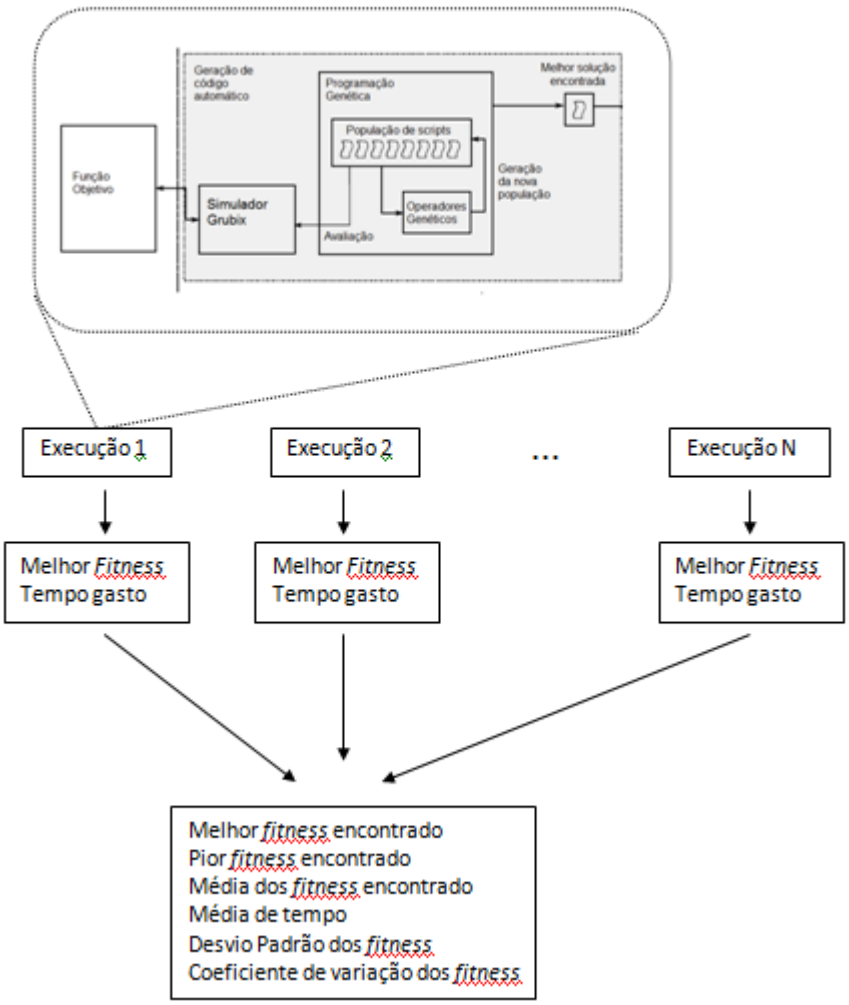

Figura 4.1: Visão geral do funcionamento do algoritmo proposto.

A função *map* irá retornar como chave o nome da instância que foi executada em uma *string*, e o valor será uma *string* contendo o *fitness* obtido pelo algoritmo genético e o tempo gasto para ser executado. A função *reduce* realiza o

cálculo do desvio padrão e do coeficiente de variação dos valores de *fitness* para cada instância, retornando como chave uma *string* contendo o nome da instância, e o valor uma *string* contendo o menor, o maior e a média dos *fitness* obtidos e valor de desvio padrão e coeficiente de variação desses *fitness*.

A Figura 4.2 ilustra o funcionamento de uma execução do algoritmo no *framework MapReduce*.

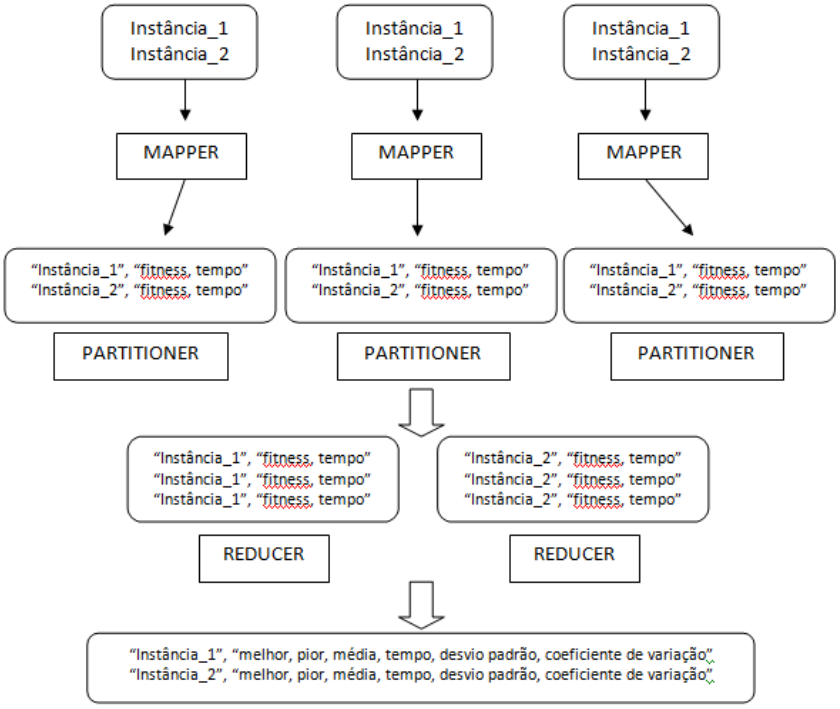

Figura 4.2: Ilustração da execução do algoritmo proposto.

O *framework* irá receber como entrada arquivos contendo as instâncias a que serão entrada para as tarefas *map* a serem executadas. Na Figura 4.2 três tarefas *map* são realizadas, onde estas são realizadas de forma paralela. Depois de a função *map* emitir os pares intermediários chave/valor, o *framework* executa a

classificação desses pares, realizando uma ordenação em relação às chaves dos pares e distribuindo o conjunto de pares de mesma chave para uma única tarefa *reduce.* Assim, a realização dos cálculos dos resultados obtidos serão feitos separados de acordo com a instância que a chave representa. Retornando então a saída final.

O algoritmo descrito acima foi implementado na linguagem C++, uma vez que o código do algoritmo genético utilizado estava implementado em C++. O *framework Hadoop MapReduce* aceita algumas linguagens de programação como Java, C++, Python, devendo usar as devidas bibliotecas de acordo com a linguagem usada. Para utilizar a linguagem C++ foi preciso utilizar a biblioteca *Hadoop Pipes* (WHITE, 2009). Segundo (WHITE, 2009) *Pipes* utiliza *sockets* como canal sobre o qual o *TaskTracker* comunica-se com o processo de execução das funções *map* e *reduce* escritas em C++.

Abaixo são apresentados trechos do pseudocódigo que se referem à implementação das funções *map* e *reduce*.

FUNÇÃO MAP

```
1    void map( HadoopPipes::MapContext& context ) {
2         string instancia = context.getInputValue( );
3         double fitness;
4         geneticnet::BatchEngine executionEngine(instancia);
5         executionEngine.execute( );
6         fitness = executionEngine.retornaFitness( );
7         context.emit(“ instancia”; “fitness, tempo”);
8    }
```

Figura 4.3: Pseudocódigo da função *map*.

A Figura 4.3 apresenta a função *map*. Na linha (2) o *framework* recebe como *string* o valor da entrada, que é o nome da instância que será executada. Na linha (4) inicia-se a execução do código da PG, recebendo a instância que será executada. Depois de executado, o *framework* irá retornar o nome da instância

como chave, e uma *string* contendo o valor do *fitness* obtido e o tempo gasto para executá-lo (linha (7)).

FUNÇÃO REDUCE

```
1   void reduce( HadoopPipes::ReduceContext& context ) {
2       while ( context.nextValue() ) {
3           std::vector<std::string> valores =
    HadoopUtils::splitString(context.getInputValue(), ",");
4           if (melhor > HadoopUtils::toFloat(valores[1])){
5               melhor = HadoopUtils::toFloat(valores[1]);
6           }
7           if (pior < HadoopUtils::toFloat(valores[1])){
8               pior = HadoopUtils::toFloat(valores[1]);
9           }
10          media = media + HadoopUtils::toFloat(valores[1]);
11          tempo = tempo + HadoopUtils::toFloat(valores[0]);
12          vfitness[i]= HadoopUtils::toFloat(valores[1]);
13          i++;
14      }
15      media = media / i;
16      tempo = tempo / i;
17      for (int j=0; j<i; j++){
18          somaquad = somaquad + pow(vfitness[j] - media, 2);
19      }
20      desviop = sqrt(somaquad/i);
21      coefvar = desviop/media*100;
22      context.emit(context.getInputKey(),saida );
23  }
```

Figura 4.4: Pseudocódigo da função *reduce*.

A Figura 4.4 apresenta a função *reduce*, onde seleciona-se o melhor e o pior valor de *fitness* obtido, calcula-se a média dos valores de *fitness* e de tempo, é calculado o desvio padrão e o coeficiente de variação dos valores de *fitness*. A função retorna então o nome da instância como chave final, e uma *string* contendo o melhor *fitness*, o pior *fitness*, a média dos *fitness*, o tempo médio de cada execução, o valor do desvio padrão e do coeficiente de variação.

5. EXPERIMENTOS E RESULTADOS

Para avaliar o desempenho do algoritmo proposto, foi usado o mesmo problema resolvido em (DE OLIVEIRA, 2013), que é um problema de detecção de eventos com as seguintes características:

- Não há comunicação na rede, os nós sensores devem coletar dados do ambiente em que está inserido;
- Quando um algum estado de ambiente é detectado, um evento é lançado;
- O evento gera uma mensagem de comunicação que deve ser lançado ao nó *sink*.

5.1. Ambiente de execução

O ambiente de execução do experimento é composto por quatro máquinas ligadas em rede. A tabela 5.1 apresenta as configurações das máquinas. As configurações para o funcionamento do *framework Hadoop MapReduce* foram feitas conforme (WHITE, 2009).

Tabela 5.1: Configuração do ambiente

Hardware	
CPU	AMD Athlon (TM)64 Processor 3500+ 2.200 MHz 512Kb Cache
Memória	1,3GB DDR RAM
Disco Rígido	5GB
Rede	10/100 Mbps Fast Ethernet
Software	
S.O.	Ubuntu 11.04 LTS
Apache	Hadoop MapReduce 1.0.4

5.2. Instâncias do problema

Foram utilizadas oito instâncias diferentes para avaliar o trabalho. Cada instância representa as posições dos nós da rede de sensores em um ambiente. A tabela 5.2, adaptada de (DE OLIVEIRA, 2013), apresenta as principais características de cada instância, número de nós sensores, dimensão do campo em que está inserida e sua topologia, de forma aleatória ou em grade.

Tabela 5.2: Característica das instâncias

Instância	Numero de nós	Dimensão do campo	Topologia
G25	25	40m x 40m	Grade
G49	49	60m x 60m	Grade
G225	225	140m x 140m	Grade
G625	625	240m x 240m	Grade
R25	25	40m x 40m	Randômica
R49	49	60m x 60m	Randômica
R225	225	140m x 140m	Randômica
R625	625	240m x 240m	Randômica

A figura 5.1 representa a instância G25. Os círculos indicam os nós sensores, e as linhas indicam as conexões de comunicação entre os nós.

Os nós rotulados com "*Event"* são os nós que detectam eventos ocorridos no ambiente. O nó rotulado com "*Sink Node"* é o nó que deve ser informado sobre a ocorrência do evento ocorrido no ambiente. Nota-se que os nós estão dispostos em forma de grade no ambiente. Esta instância é a mais simples dessa topologia.

A figura 5.2 de (DE OLIVEIRA, 2013) ilustra a instância R25. Onde a diferença está na disposição dos nós sensores, não há um padrão entre suas disposições. Esta instância é a mais simples dessa topologia.

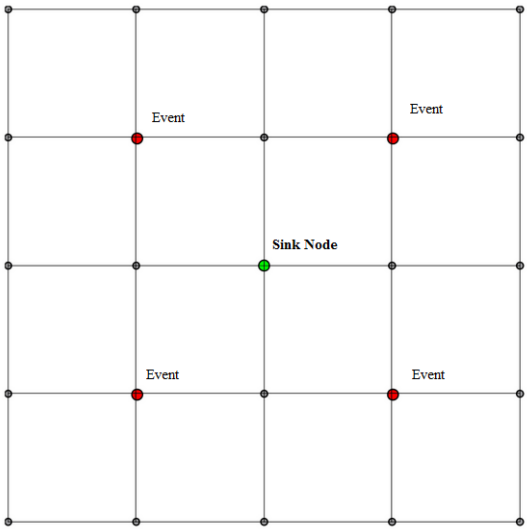

Figura 5.1: Topologia da instância G25

5.3. Simulações e Resultados Obtidos

O algoritmo foi executado para resolver as instâncias descritas acima. Os resultados são analisados em relação à qualidade dos resultados obtidos baseado em função *fitness* definida em (DE OLIVEIRA, 2013). Para cada instância o algoritmo foi executado 40 vezes, foi feito o cálculo do desvio padrão e do coeficiente de variância em relação aos resultados obtidos. O objetivo desses cálculos é verificar quanto varia as soluções obtidas, uma vez que uma PG é um método estocástico, não garantindo a ocorrência de uma solução ótima.

Os parâmetros da Programação Genética foram os mesmos utilizados em (DE OLIVEIRA, 2013).

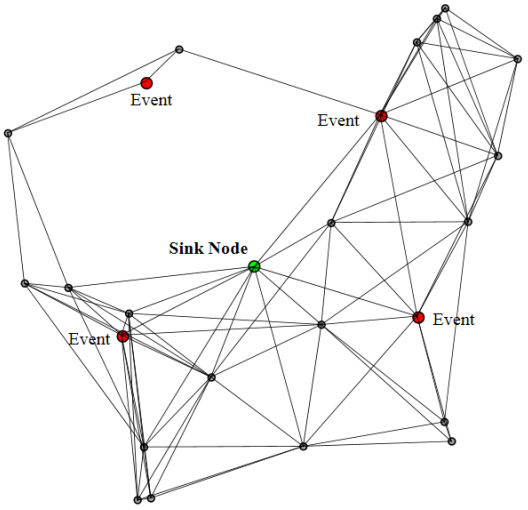

Figura 5.2: Topologia da instância R25

A tabela 5.3 mostra os resultados obtidos para as quarenta execuções. Na tabela são apresentados: o menor resultado de *fitness* obtido, que representa a melhor solução encontrada pelo algoritmo genético; o maior resultado de *fitness* obtido, que representa a pior solução encontrada pelo algoritmo genético; a média dos resultados de *fitness* obtidos e o coeficiente de variação realizado nos valores de *fitness* obtidos. O coeficiente de variação é uma medida de dispersão utilizada para estimar a precisão de experimentos como uma porcentagem.

A tabela 5.4 mostra os resultados obtidos em (DE OLIVEIRA, 2013). Comparando os resultados, nota-se que, para as instâncias G49, G225 e G625 executadas dez vezes em (DE OLIVEIRA, 2013), o número de execuções utilizado não é suficiente para se realizar uma análise precisa da eficiência a PG, uma vez que, quando executados quarenta vezes, a PG retorna possíveis soluções não encontradas em (DE OLIVEIRA, 2013). Assim, realizando mais execuções da PG, a conclusão sobre a eficiência do algoritmo será mais precisa. Para as outras instâncias a diferença de variação dos resultados obtidos em relação ao

número de execuções não é tão significativa, porém um estudo com mais execuções traz mais precisão ao trabalho.

Tabela 5.3: Resultados obtidos

Instância	Menor Fitness	Maior Fitness	Média dos Fitness	Coeficiente de variação
R25	30	104	39,1	50,39%
R49	43	199	93,45	62,18%
R225	376	2722	1093,2	82,04%
R625	757	4384	3047,575	37,10%
G25	20	27	23,425	13,60%
G49	29	68	31,075	21,78%
G225	61	552	78,075	104,50%
G625	101	1416	232,9	143,45%

Tabela 5.4: Resultados obtidos em (DE OLIVEIRA, 2013).

Instância	Menor Fitness	Maior Fitness	Média dos Fitness	Coeficiente de Variação
R25	30	53	32,9	22,22%
R49	43	189	84,8	70,75%
R225	376	2722	1157	77,64%
R625	461	4384	2756,2	47,27%
G25	20	27	22,6	69,74%
G49	29	29	29	0,00%
G225	61	61	61	0,00%
G625	101	101	101	0,00%

O *framework Hadoop MapReduce* oferece uma interface *web* que permite acompanhar o andamento das execuções. A interface mantém informações sobre as tarefas *map* e *reduce* que estão sendo executadas ou que foi finalizada. É possível saber também em quais máquinas do *cluster* essas tarefas foram executadas.

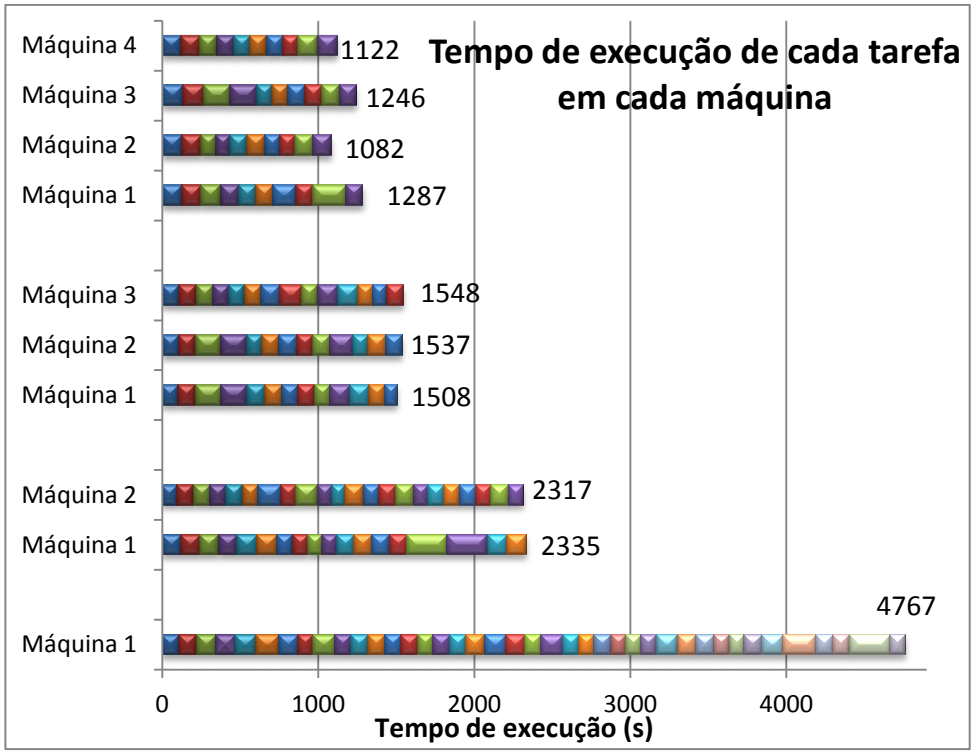

Figura 5.3: Tempos de execução

Na figura 5.3 é apresentado o tempo de execução, em segundos, de cada tarefa para a instância G25. Cada quadradinho representa o tempo de execução de uma tarefa. O tempo apresentado para a barra total em uma máquina é o tempo de execução das tarefas executadas naquela máquina. Por exemplo, para a execuções

em duas máquinas, o tempo total de execução foi (2.317+2.335=4.652 segundos). Foram realizadas execuções com uma, duas, três e quatro máquinas. Cada tarefa é a execução da função *map*, então todas as tarefas são iguais, uma vez que foi utilizada a mesma instância do problema. Os tempos de execução são diferentes para cada tarefa devido a inicialização da população de indivíduos com valores aleatórios que impacta no tempo de execução da programação genética para obter um indivíduo de melhor qualidade.

Quando os valores aleatórios são gerados em máquinas diferentes, há possibilidade de geração de indivíduos melhores do que quando gerado em uma mesma máquina.

Comparando a execução em uma máquina com a execução em duas máquinas percebe-se que na execução com duas máquinas o tempo total de execução em cada máquina é inferior à metade do tempo de execução com uma máquina. Isso ocorre, pois a média do tempo de execução de uma tarefa em apenas uma máquina foi 119,175 segundos, enquanto que a média do tempo de execução com duas máquinas foi de 116,3 segundos.

A média do tempo de execução das tarefas em três máquinas foi de 114,825, fazendo com que a execução utilizando três máquinas fosse 3,08 vezes mais rápida comparada com a execução utilizando uma máquina.

Ainda que a média do tempo de execução das tarefas em quatro máquinas fosse menor que em uma máquina, 118,425 segundos, a execução em quatro máquinas foi 3,70 vezes mais rápida comparando com a execução utilizando uma máquina. Analisando a Figura 5.3 é possível observar que algumas máquinas (Máquina 1 e Máquina 3) tiveram um maior tempo de execução, ainda que tenham realizado o mesmo número de tarefas, em relação às outras duas, impactando no tempo total de execução.

6. CONCLUSÃO

Uma Programação Genética necessita da realização de várias execuções para se ter maior grau de confiança quanto à sua eficiência da solução gerada. Dessa forma, são geradas muitas computações e gasto um tempo de execução alto. Uma forma de reduzir o tempo de execução de uma Programação Genética, gerando boas soluções, é realizando a paralelização e/ou distribuição dessa programação. Existem duas abordagens principais: distribuir as várias execuções (tarefas) da Programação Genética, para serem realizadas simultaneamente, e paralelizar o cálculo do *fitness*.

Uma aplicação existente para Programação Genética é a geração de código para os nós de uma Rede de Sensores Sem Fio.

Neste trabalho, foi proposto um algoritmo, utilizando o modelo de programação *MapReduce*, para distribuir as várias execuções da Programação Genética para uma Rede de Sensores Sem Fio entre as máquinas de um *cluster*.

A partir dos resultados obtidos, pode-se concluir que para algumas instâncias do problema analisado, a realização de mais execuções da Programação Genética trouxe resultados mais precisos que uma execução sequencial e em menor tempo de execução. Assim, é mostrada a necessidade de uma grande quantidade de execuções da Programação Genética para obter resultados mais precisos.

A utilização do *framework MapReduce* mostrou-se eficaz na realização da distribuição das execuções da Programação Genética, uma vez que o tempo médio de execução de cada tarefa pode ser reduzido e o número de tarefas executadas em um *cluster* de computadores pode ser grande. Como as populações na Programação Genética são geradas de forma aleatória, cada tarefa pode ter tempo de médio de execução diferente em cada execução, mas impactando em um tempo total de execução menor.

Como trabalhos futuros, pretende-se:

- Realizar experimentos executando um número maior de tarefas para obter resultados mais precisos;
- Realizar experimentos com um número maior de máquinas para analisar melhor o tempo médio de execução das tarefas e
- Comparar os resultados do algoritmo proposto com os resultados gerados por um algoritmo que realize a paralelização do cálculo do *fitness* no modelo de programação *MapReduce*.

REFERÊNCIAS BIBLIOGRÁFICAS

BHONDEKAR, A. P., VIG, R., SINGLA, M. L., GHANSHYAM, C., E KAPUR, P. (2009). Genetic algorithm based node placement methodology for wireless sensor networks. In Proceedings of the International MulticConference of Engineers and Computer Scientists.

BOULISA, A., HANB, C., SHEAB, R., E SRIVASTAVAB, M. B. (2007). Sensorware: Programming sensor networks beyond code update and querying. Pervasive and Mobile Computing, 3:386–412.

DEAN, J. E GHEMAWAT, S. (2004) MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December.

DE OLIVEIRA, R. R. R., HEIMFARTH, T., DE BETTIO, R. W., ARANTES, M. S., TOLEDO, C.F.M. (2013). A Genetic Programming Based Approach to Automatically Generate Wireless Sensor Networks Applications. 2013 IEEE Congress on Evolutionary Computation June 20-23, Cancún, México.

FERENTINOS, K. P. E TSILIGIRIDIS, T. A. (2007). Adaptive design optimization of wireless sensor networks using genetic algorithms. Comput. Netw., 51:1031 – 1051.

HOLLAND, J. H. (1975). Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, MI, USA.

HUANG D. E LIN J. (2010). Scaling Populations of a Genetic Algorithm for Job Shop Scheduling Problems using MapReduce. 2nd IEEE International Conference on Cloud Computing Technology and Science. Pag. 780 – 785. Washington DC, EUA.

JIN C.,VECCHIOLA C. E BUYYA R. (2008). MRPGA: An Extension of MapReduce for Parallelizing Genetic Algorithms. Fourth IEEE International Conference on eScience. Pag. 214 – 221. Washington DC, EUA.

KOZA, J. R. (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA.

LIN, JIMMY AND DYER, CHRIS. Data-Intensive Text Processing with MapReduce. Maryland, 2010. Final Pre-Production.

LOUREIRO, A. A. F., NOGUEIRA, J. M. S., RUIZ, L. B., DE FREITAS MINI, R. A., NAKAMURA, E. F., E FIGUEIREDO, C. M. S. (2003). Redes de sensores sem fio. In Anais do Simpósio Brasileiro de Redes de Computadores.

RÖMER, K. E MATTERN, F. (2004). The design space of wireless sensor networks. IEEE Wireless Communications, 11:54–61.

SETTE, S. E BOULLART, L. (2001). Genetic programming: principles and applications. Engineering Applications of Artificial Intelligence, 14(6):727 – 736.

TURGUT, D., DAS, S., ELMASRI, R., E TURGUT, B. (2002). Optimizing clustering algorithm in mobile ad hoc networks using genetic algorithmic approach. IEEE GLOBECOM02, Tapei.

WANG M.M., CAO, J. L. J. D. S. (2008). Middleware for wireless sensor networks: A survey. Journal of Computer Science And Technology.

WEISE, T. (2006). Genetic programming for sensor networks. Technical report, University of Kassel, University of Kassel.

WHITE, T. (2009). Hadoop: The Definitive guide. O‟Reilly.