UNIVERSIDADE ESTADUAL DO MARANHÃO
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
CENTRO DE CIÊNCIAS TECNOLÓGICAS

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO
CURSO DE MESTRADO PROFISSIONAL EM ENGENHARIA DE COMPUTAÇÃO E SISTEMAS

JACKSON AMARAL DA SILVA

Dissertação de Mestrado

UM GERADOR AUTOMÁTICO DE ALGORITMOS EVOLUTIVOS PARALELOS EM JAVA

São Luís
2013

JACKSON AMARAL DA SILVA

UM GERADOR AUTOMÁTICO DE ALGORITMOS EVOLUTIVOS PARALELOS EM JAVA

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia de Computação e Sistemas da Universidade Estadual do Maranhão, como parte das exigências para a obtenção do título de mestre em engenharia de computação e sistemas.

Orientador: Prof. Dr. Omar Andres Carmona Cortes.

SÃO LUIS - MA

2013

JACKSON AMARAL DA SILVA

UM GERADOR AUTOMÁTICO DE ALGORITMOS EVOLUTIVOS PARALELOS EM JAVA

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia de Computação e Sistemas da Universidade Estadual do Maranhão, como parte das exigências para a obtenção do título de mestre em engenharia de computação e sistemas.

APROVADA EM: 10 de dezembro de 2013.

Prof. Dr. Omar Andres Carmona Cortes (Orientador)
Departamento Acadêmico de Informática
Instituto Federal de Educação, Ciência e Tecnologia do Maranhão (IFMA)

Prof. Dr. Alexandre César Muniz de Oliveira
Departamento de Informática
Universidade Federal do Maranhão (UFMA)

Prof. Dr. Luís Carlos Costa Fonseca
Departamento de Engenharia da Computação
Universidade Estadual do Maranhão (UEMA)

Silva, Jackson Amaral da.
Um gerador automático de algoritmos evolutivos paralelos em Java / Jackson Amaral da Silva. – São Luis, 2013.
98f

Dissertação (Mestrado) – Curso de Engenharia de Computação e Sistemas, Universidade Estadual do Maranhão, 2013.

Orientador: Prof. Dr. Omar Andres Carmona Cortes.

1.Algoritmos evolutivos paralelos. 2.Geração automática de código. 3.Padrões de projeto. I.Título

CDU: 004.421

À minha família e amigos dos quais

sempre lembro em minhas orações.

AGRADECIMENTOS

Agradeço primeiramente a Deus, por sempre me abençoar e por me dar forças para superar todas as dificuldades encontradas no desenvolvimento deste trabalho. Sem ele nada seria possível pra mim.

Aos meus amigos que conheci durante a vida acadêmica tanto no IFMA quanto na UEMA, que sempre me apoiaram e compartilharam comigo seu conhecimento técnico-científico. Em especial ao meu irmão Jefferson que sempre foi um exemplo de pessoa e profissional pra mim.

A meu orientador Prof. Dr. Omar Andres, pela instrução, orientação e companheirismo desde a época em que eu cursava a graduação.

A minha noiva Hildejane que sempre me ofereceu apoio incondicional em todos os aspectos de minha vida e a todo momento me oferece palavras e gestos de carinho e ternura.

“As máquinas me surpreendem com muita frequência”

Alan Turing

RESUMO

Os Algoritmos Evolutivos (AEs) são capazes de encontrar boas soluções para problemas em diversos campos. Da mesma forma, os Algoritmos Evolutivos Paralelos (AEPs) também resolvem vários tipos de problemas, com a vantagem de que estes necessitam de menos tempo para concluir sua execução, característica essa que é vantajosa quando os problemas a serem resolvidos são muito grandes e complexos. Assim, podemos afirmar que os AEPs podem ser eficientes e mais rápidos do que um AE regular. Por outro lado, a programação paralela trás consigo novos problemas substanciais para os desenvolvedores, aumentando assim o tempo e esforço necessários para concluir a implementação. Nesse contexto, foi desenvolvido o *Java Parallel Evolutionary Algorithms Generator* (JPEAG), um *software Web* que cria automaticamente código Java que implementa Algoritmos Genéticos Paralelos e Estratégias Evolutivas Paralelas, a partir de parâmetros inseridos em sua interface gráfica. O usuário configura um AEP através da interface gráfica do JPEAG e seu núcleo utiliza essas informações para selecionar e processar um conjunto de templates e a partir deles gerar o código-fonte Java que implementa o AEP configurado pelo usuário. Assim o JPEAG reduz o tempo e esforço necessários para o desenvolvimento de AEPs. Foram implementados mecanismos para a geração de três modelos de paralelismo: ilhas, mestre-escravo e vizinhança, com paralelismo baseado em *threads*. Além disso, os padrões de projeto Strategy e Observer foram aplicados no código gerado, a fim de aumentar a sua flexibilidade e legibilidade, facilitando sua compreensão e manutenção. Foram realizados experimentos com AEPs gerados a partir de nossa ferramenta, com o objetivo de mostrar o aumento de velocidade proporcionado pelo paralelismo.

Palavras chave: Algoritmos Evolutivos Paralelos, Geração automática de código, Padrões de Projeto.

ABSTRACT

Evolutionary Algorithms (EAs) are able to find out solutions in many fields. In the same manner, Parallel Evolutionary Algorithms (PEAs) also solve many kinds of problems, with the advantage of requiring less time to complete its execution, which is a benefit when the problems being solved are very large and complex. Thereby, we can state that PEAs can be efficient and faster than a regular EA. On the other hand, parallel programming brings new substantial problems to the developers, increasing the programming time and efforts to complete the implementation. In this context, it was developed the *Java Parallel Evolutionary Algorithms Generator* (JPEAG), a web-based software that automatically creates Java code for parallel genetic algorithms and parallel evolutionary strategies, based on parameters given on its GUI. The user configures the PEA through the JPEAG GUI and its core uses these information to select and process a set of templates and then generates Java source code that implements the user-configured AEP. Doing so, JPEAG considerably reduces the required time and effort to develop this kind of application. Three parallel models were implemented, island, neighborhood and master-slave using thread-based parallelism. Further, the design patterns strategy and observer were applied in the generated code in order to increase the software maintainability and legibility, facilitating its understanding and maintenance. An experiment was conducted with AEPs generated from our tool in order to show the speedup provided by the parallelism.

Keywords: Parallel Evolutionary Algorithms, Automatic Code Creation, Design Patterns.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

AE - Algoritmo Evolutivo

AEP - Algoritmo Evolutivo Paralelo

AG - Algoritmo Genético

EE - Estratégias Evolutivas

JPEAG - *Java Parallel Evolutionary Algorithms Generator*

MIMD - *Multiple Instruction, Multiple Data*

MISD - *Multiple Instruction, Single Data*

MVC - *Model View Controller*

SIMD - *Single instruction, multiple data*

SISD - *Single Instruction, Single Data*

VTL - *Velocity Template Language*

SUMÁRIO

1 INTRODUÇÃO

Algoritmos Evolutivos (AEs) são algoritmos de busca baseados na evolução natural (EIBEN, 2003). Em geral, eles podem encontrar boas soluções em uma quantidade razoável de tempo. Contudo, quando eles são aplicados na resolução de problemas maiores e mais complexos, há um aumento significativo no tempo de execução necessário para encontrar soluções adequadas. Como consequência, houve vários esforços para tornar os AEs mais rápidos, sendo que uma das alternativas mais promissoras é a utilização de implementações paralelas (CANTÚ-PAZ, 2007). Ao dividir o processamento da população por vários elementos de processamento, os algoritmos evolutivos paralelos (AEPs), permitem alcançar resultados de alta qualidade em um tempo de execução razoável até mesmo na resolução de problemas complexos de otimização (NESMACHNOW, 2012)

O conceito de AEPs é razoavelmente simples, contudo, a implementação de programas paralelos trás desafios adicionais, como a correta sincronização entre as populações, a exploração adequada do paralelismo e um processo diferente de depuração, aumentando assim a curva de aprendizagem e os esforços para concluir a implementação. Além disso, a falta de experiência no desenvolvimento de aplicações paralelas também pode impactar de forma negativa na produtividade do desenvolvimento de aplicações evolutivas paralelas.

Nesse contexto, este trabalho apresenta o *Java Parallel Evolutionary Algorithms Generator* (JPEAG) (SILVA, 2013), uma aplicação *web* que implementa um gerador de código paralelo, baseado em *templates*, que produz código de AEPs escritos com Java. O JPEAG gera automaticamente grande parte do código fonte dos AEPs a partir de parâmetros informados pelo usuário através de sua interface gráfica, deixando a cargo do usuário escrever manualmente somente o código referente à função de avaliação do AE, pois essa é específica do problema que se deseja resolver utilizando o AEP. Apesar de não ser gerada automaticamente, essa função pode ser inserida diretamente na interface gráfica do JPEAG e assim o código estará pronto para ser compilado e executado. Atualmente o JPEAG é capaz de criar código para Algoritmos Genéticos (AG) e Estratégias Evolutivas (EE), ambos

utilizando representação real de indivíduos, nos modelos de paralelismo mestre-escravo, vizinhança e ilha, utilizando paralelismo baseado em *threads*. No entanto, extensões para gerar outros tipos de AEPs podem ser facilmente implementadas no *software*.

1.1 TRABALHOS CORRELATOS

O objetivo principal do desenvolvimento do JPEAG é fornecer uma forma simples de desenvolver aplicações paralelas, mais especificamente AEPs, aumentado assim a produtividade de seu desenvolvimento. A fim de superar as dificuldades encontradas no desenvolvimento de programas paralelos e reduzir seu nível de complexidade, que tendem a diminuir a produtividade, diversos autores apresentaram formas de auxiliar no desenvolvimento de aplicações paralelas, seja gerando código-fonte ou fornecendo bibliotecas que facilitam o seu desenvolvimento.

Jaquie (1999), Malacarne (2001), Passini & Dotti (2007) e Rodrigues et. al. (2012) apresentam ferramentas que produzem código paralelo automaticamente. No entanto, elas geram código referente apenas a aspectos gerais, comuns a maioria dos programas paralelos, como sincronização e comunicação, sendo que os usuários tem que implementar manualmente o código específico de sua aplicação. Hawick & Playne (2010) criaram uma abordagem específica para geração de código paralelo para resolver equações diferenciais parciais (PDEs).

Em Cahon et. al. (2004) é apresentado o ParadisEO. Um *framework* que fornece um conjunto de classes genéricas reutilizáveis que podem ser utilizadas para implementar metaheurísticas paralelas e distribuídas. O ParadisEO oferece suporte à diferentes modelos de paralelização utilizando as bibliotecas MPI, PVM e PThreads. Similarmente aos outros trabalhos citados previamente, neste, o usuário tem que implementar manualmente as classes específicas de sua aplicação e aprender a utilizar as classes fornecidas pelo *framework* da maneira correta. Pelo nosso conhecimento pode-se afirmar que não existem trabalhos que produzam automaticamente código-fonte paralelo para algoritmos evolutivos.

Nesse contexto, os demais capítulos deste trabalho estão organizados da seguinte forma: o Capítulo 2 apresenta uma introdução à computação paralela; o Capítulo 3 apresenta os conceitos básicos de algoritmos evolutivos, englobando os AGs, as EEs e os AEPs; o Capítulo 4 apresenta a teoria sobre a geração de código e aprofunda na estratégia de geração baseada em templates, o Capítulo 5 apresenta padrões de projeto e como estes foram utilizados para aumentar a qualidade do código-fonte dos AEPs. O Capítulo 6 trata do desenvolvimento do JPEAG e dos experimentos realizados. Finalmente, o Capítulo 7 apresenta as conclusões deste trabalho e o horizonte de trabalhos futuros que estão em mente.

2 INTRODUÇÃO A COMPUTAÇÃO PARALELA

A partir da década de 80 o desempenho dos microprocessadores baseados em apenas uma unidade de processamento aumentou rapidamente. Essa ascensão chegou ao seu limite por volta do ano de 2003, devido a problemas com dissipação de calor e consumo de energia, que limitaram o aumento do *clock* (KIRK, 2010). Dessa forma, em algum tempo, virtualmente todos os fabricantes de microprocessadores mudaram o modelo de seus produtos para formatos que utilizam múltiplas unidades de processamento (referidas como núcleos), chamados de processadores *multicore* (multi-núcleo). Dessa forma, o caminho para aumentar o desempenho dos processadores acabou indo na direção do paralelismo. Devido a esse fato o *hardware* paralelo se popularizou rapidamente e já se tornou comum há algum tempo. É difícil encontrar um computador (*notebook*, *desktop* ou servidor) atual, que não utilize pelo menos um processador *multicore* (PACHECO, 2011).

É nesse contexto que pode-se definir a computação paralela como sendo uma coleção de elementos de processamento que se comunicam e cooperam entre si e com isso resolvem um problema de maneira mais rápida (ALMASI, 1994). A ideia básica dos programas paralelos consiste em dividir uma tarefa em partes menores e executar essas partes simultaneamente utilizando várias unidades de processamento e dessa forma concluir a tarefa mais rapidamente (CANTÚ-PAZ, 1998).

2.1 HARDWARE PARALELO

Em computação paralela a taxonomia de Flynn é frequentemente utilizada para classificar as arquiteturas computacionais. Ela classifica os sistemas de acordo com o número de fluxos de execução e o número de fluxos de dados que eles podem gerenciar simultaneamente, conforme a listagem a seguir:

- *Single Instruction, Single Data* (SISD) - é um computador serial que é capaz de executar uma única instrução por vez sobre um único item de dados. Os clássicos sistemas de von Neumann se enquadram nessa classificação.
- *Single instruction, multiple data* (SIMD) - são sistemas paralelos que operam sobre múltiplos fluxos de dados, aplicando a mesma instrução a vários itens de dados.
- *Multiple Instruction, Single Data* (MISD) - são sistemas paralelos onde cada unidade de processamento opera de forma independente sobre um mesmo conjunto de dados, por meio de fluxos de instrução separados.
- *Multiple Instruction, Multiple Data* (MIMD) - são sistemas paralelos onde cada unidade de processamento pode executar uma instrução diferente e operar sobre um fluxo de dados diferente.

A maioria dos computadores paralelos atuais se enquadra na classificação MIMD (GOMES, 2009). Os sistemas MIMD tipicamente são constituídos de um conjunto de unidades de processamento ou núcleos totalmente independentes. Existem dois tipos principais de sistemas paralelos MIMD: sistemas de memória compartilhada e sistemas de memória distribuída (PACHECO, 2011).

Em sistemas de memória compartilhada, os processadores atuam independentemente, mas compartilham acesso a uma memória comum, em princípio, cada núcleo pode ler e escrever em qualquer local da memória (GOMES, 2009). Os sistemas de memória compartilhada mais amplamente disponíveis atualmente são computadores que utilizam um ou vários processadores *multicore* (PACHECO, 2011).

Em sistemas de memória distribuída, cada núcleo tem sua própria memória privativa e acessa a memória dos outros nós via algum tipo de rede de comunicação (GOMES, 2009). Os sistemas de memória distribuída mais amplamente disponíveis são os chamados *clusters*. Na verdade, os nós desses sistemas, são geralmente sistemas de memória compartilhada com um ou mais processadores *multicore*. Para distinguir esses sistemas dos sistemas de memória distribuída puros, eles são às vezes chamados sistemas híbridos. É nos sistemas híbridos que as grades de computadores se baseiam, estas são infra-estruturas que permitem a interligação e o compartilhamento de recursos computacionais, a fim de se otimizar a realização

de tarefas (CAMARGO, 2007). Na verdade, a grade computacional oferece a infra-estrutura necessária para transformar grandes redes de computadores distribuídos geograficamente em um sistema de memória distribuída unificado. (PACHECO, 2011).

Assim, levando em consideração os sistemas MIMD, quando se executa programas em ambiente de memória compartilhada, inicia-se um único processo e se bifurca a execução das tarefas em várias *threads*, que possuem cada uma seu próprio fluxo de execução. Por outro lado, quando nós executamos programas de memória distribuída, nós vamos iniciar vários processos diferentes (PACHECO, 2011).

2.2 SOFTWARE PARALELO

A utilização de *hardware* paralelo pode diminuir o tempo total de execução da aplicação, contudo para aproveitar o seu potencial é necessário *software* que faça uso desse paralelismo. Esse tipo de *software* traz consigo complexidades adicionais em relação ao *software* sequencial, como o controle de processos/*threads* em execução, controle de acesso à memória e comunicação entre os processos/*threads* (GOMES, 2009). Devido a isso, existe uma série de fatores que se deve observar ao projetar e desenvolver *softwares* que utilizem o paralelismo.

Um dos aspectos iniciais a ser considerados na criação de um programa paralelo diz respeito a como dividir o problema em "pedaços" de trabalho que possam ser executados por vários núcleos. Para tal existem duas abordagens largamente utilizadas (GOMES, 2009), (PACHECO, 2011):

- Paralelismo de tarefa - o programa é decomposto em várias tarefas que devem realizadas para resolver o problema e essas são executadas por diferentes unidades de processamento. Nesse caso os núcleos podem ou não trabalhar sobre o mesmo conjunto de dados (GOMES, 2009), (PACHECO, 2011).

- Paralelismo de dados - a mesma tarefa é alocada a vários núcleos, mas cada um deles trabalha sobre um conjunto de dados diferente do outro. O conjunto de dados utilizado na resolução do problema é subdividido e cada parte é tratada por um núcleo diferente (GOMES, 2009), (PACHECO, 2011).

Quando os vários núcleos podem trabalhar de forma independente, escrever um programa paralelo é similar a escrever um programa sequencial. No entanto, essa tarefa se torna bem mais complexa quando os núcleos precisam coordenar seus trabalhos entre si para resolver a tarefa principal. A coordenação de tarefas paralelas em geral envolve a comunicação e a sincronização (PACHECO, 2011) (BARNEY, 2010).

2.2.1 Comunicação Inter-Tarefas

Tarefas paralelas geralmente precisam trocar informações entre si, o objetivo dessa troca de informações pode ser, por exemplo, enviar o resultado parcial de sua execução ou manter o sincronismo entre as tarefas. Ao projetar a comunicação inter-tarefas é importante observar que toda comunicação tem um custo, essa praticamente sempre implica em *overhead* e utiliza ciclos de processador e recursos da máquina para transmitir dados, recursos esses que poderiam estar sendo usados para realizar a computação da tarefa (BARNEY, 2010).

Em sistemas com memória compartilhada, os processadores geralmente se comunicam implicitamente através do acesso a variáveis ou estruturas de dados compartilhadas. Por outro lado em sistemas de memória distribuída, os processadores geralmente se comunicam explicitamente através do envio de mensagens pela rede de interconexão que os liga. (PACHECO, 2011). Os programas que executam sobre esse modelo criam tarefas que encapsulam dados locais e interagem umas com as outras através de mensagens.

2.2.2 Sincronização

A coordenação entre tarefas em paralelo, muitas vezes está associada à sincronização. Essa geralmente é implementada por meio da criação de um ponto de sincronização dentro da aplicação, no qual a tarefa não pode prosseguir com seu processamento até que alguma determinada condição seja satisfeita (BARNEY, 2010). Esse ponto de sincronização é comumente chamado de barreira, sendo encontrado nas principais linguagens paralelas explícitas, onde o programador deve controlar a comunicação entre as tarefas.

Em ambientes de memória compartilhada pode ocorrer de tarefas paralelas executarem operações simultâneas no mesmo local de memória. Caso não haja a sincronização adequada, um código que cause essa situação pode até, por vezes, funcionar normalmente, mas pode não funcionar corretamente, e de forma imprevisível, em outros momentos, pois alguma atualização de memória pode ser sobrescrita ou alguma leitura de memória pode ser efetuada antes de uma atualização e assim acessar um valor inconsistente (MCCOOL, 2012). Os segmentos de código que atualizam regiões de memória compartilhada por diferentes tarefas são denominados seção crítica.

Uma forma de evitar inconsistências no acesso à seção crítica é fazer com que apenas uma tarefa possa acessá-la por vez, por meio da criação de áreas de exclusão mútua. Dessa forma as diferentes tarefas podem ser coordenadas de modo que elas se revezem no acesso à seção crítica e assim mantenham a consistência dos dados acessados dentro dela. A exclusão mútua é normalmente implementada com um *lock*, muitas vezes chamado de *mutex* (MCCOOL, 2012).

Mutex é abreviação de *mutual exclusion*, esse é um tipo especial de variável que possui apenas dois estados (aberto e fechado) e apenas duas operações (abrir e fechar) que alternam entre os estados (PACHECO, 2011). Quando uma *thread* tenta executar a operação fechar, em um *mutex* que está fechado, ela deve esperar até que esse esteja aberto. Após fechar o *mutex*, a *thread* executa a sessão crítica e após executá-la ela deve abrir o *mutex*, permitindo assim que uma outra *thread* feche novamente o *mutex* e acesse a seção crítica. As operações de abrir o *mutex* antes do início da sessão crítica e de fechar o *mutex* após o término da sessão

crítica garantem que as *threads* vão acessá-la cada uma por vez (MCCOOL, 2012). Assim um *mutex* pode ser utilizado para garantir que uma *thread* "exclui" todas as outras enquanto executa a sessão crítica, ou seja, garante o acesso mutuamente exclusivo da seção crítica. Embora existam linguagens de programação e bibliotecas que permitam a utilização dos *mutexes* de maneira simples, o programador fica responsável pelas chamadas das operações de *lock/unlock* e por definir condições que devem ser satisfeitas para que essas chamadas ocorram, o que pode ser suscetível a erros.

Os monitores são estruturas de mais alto nível que permitem a utilização de exclusão mútua, esses devem ser implementados pela linguagem de programação. Monitor é um tipo especial de módulo ou objeto cujos métodos só podem ser executados por uma *thread* de cada vez. Dessa forma, todo código que estiver dentro do monitor será acessado de forma mutuamente exclusiva (PACHECO, 2011). O monitor encapsula os aspectos referentes à implementação da exclusão mútua e o programador apenas define os segmentos de código que farão parte do monitor e terão acesso mutuamente exclusivo.

Quando a computação envolvendo tarefas paralelas é dividida em fases, muitas vezes é necessário assegurar que todas as *threads* completem todo o trabalho de uma determinada fase antes avançar para a fase seguinte (BARNEY, 2010), (MCCOOL, 2012). A barreira é uma forma de sincronização que pode assegurar isso, permitindo sincronizar várias *threads* de uma vez. Segundo MCCOOL (2012), o funcionamento de uma barreira clássica pode ser descrito da seguinte forma:

1. Define-se um ponto de sincronização onde será fixada a barreira.
2. Ao atingir a barreira, cada *thread* deve aguardar até que todas as demais a atinjam também.
3. Quando a última *thread* atinge a barreira, essa é liberada e todas as *threads* podem continuar sua execução.

Segundo Kirk (2010), a barreira é um método simples e popular de coordenar tarefas paralelas. Ele utiliza o seguinte exemplo do cotidiano para exemplificar a

utilização de sincronização por barreira para coordenar tarefas paralelas de um grupo de pessoas:

1. Quatro amigos vão a um *shopping* utilizando como transporte um único carro. Ao chegarem ao shopping, eles se dirigem a diferentes lojas para comprar roupas.
2. Ao concluírem suas compras, cada um deles retorna ao local onde está estacionado carro e espera até que todos os outros retornem também.
3. Após todos retornarem, eles entram no carro e deixam o *shopping*.

Executar essa atividade paralelamente é mais eficiente do que seria executá-la de maneira sequencial. Onde os amigos permaneceriam em grupo e visitariam as lojas sequencialmente para comprar as roupas de cada um. No entanto, a sincronização por barreira é necessária antes deles deixarem *shopping*, pois sem essa barreira, possivelmente alguém seria deixado para trás (KIRK, 2010).

2.3 GRANULARIDADE DO PARALELISMO

Uma decisão importante, que afeta diretamente o desempenho dos programas paralelos, diz respeito à granularidade do paralelismo (KIRK, 2010). A granularidade é definida pela quantidade de decomposição aplicada na paralelização de um algoritmo e pelo tamanho da unidade de decomposição, que é a quantidade de trabalho a ser executado de forma serial em cada processador (MCCOOL, 2012). Segundo Barney (2010) a definição da granularidade envolve também a quantidade de trabalho executado por cada processador entre os eventos de comunicação/sincronização. A granularidade mais eficaz depende do algoritmo que se está executando e do ambiente de *hardware* no qual ele é executado. Uma conhecida divisão dos níveis de granularidade é:

- Granularidade grossa - utilizam-se poucas tarefas paralelas, grandes e complexas. (JAQUIE, 1999). Uma quantidade relativamente grande de trabalho computacional é realizado entre os eventos de comunicação/sincronização (BARNEY, 2010). Às vezes, é mais vantajoso

colocar mais trabalho em cada tarefa e usar menos tarefas paralelas, tornando a granularidade mais grossa. Contudo se a granularidade for muito grossa, pode não haver tarefas paralelas o suficiente para fazer uso de todas as unidades de processamento disponíveis (MCCOOL, 2012).

- Granularidade fina - utiliza-se um grande número de tarefas paralelas pequenas e simples (JAQUIE, 1999). Uma quantidade relativamente pequena de trabalho computacional é realizado entre os eventos de comunicação/sincronização (BARNEY, 2010). Se a granularidade for muito fina, haverá muitas tarefas paralelas e o *overhead* causado por comunicação/sincronização pode suprimir a melhora de desempenho causado pela utilização do paralelismo (MCCOOL, 2012).

2.4 SPEEDUP E EFICIÊNCIA

O principal objetivo ao se escrever programas paralelos é aumentar o desempenho (ALBA, 2006), (PACHECO, 2011), dessa forma, torna-se fundamental utilizar métricas para medir o ganho de desempenho proporcionado pela utilização do paralelismo. Duas importantes métricas relacionadas ao paralelismo e seu desempenho são o *speedup* e a eficiência. O *speedup* relaciona o tempo necessário para resolver um problema computacional utilizando um processador (T_1), ou seja, executando um programa serial, e o tempo necessário para resolver um problema idêntico utilizando p processadores (T_p) conforme a equação 1 (GOMES, 2009) (MCCOOL, 2012).

$$speedup = \frac{T_1}{T_p} \quad (1)$$

Quando um algoritmo é executado p vezes mais rápido utilizando p processadores do que utilizando 1 processador (equação 2), é dito que sua versão paralela demonstrou *speedup* linear. Dessa forma o aumento da velocidade é diretamente proporcional ao aumento do número de processadores. Na prática, atingir o *speedup* linear é improvável, uma vez que a distribuição de tarefas por vários processos/*threads* quase sempre introduz algum *overhead* na aplicação, pois

em geral exige a execução de trabalho extra que não necessita ser realizado em programas seriais, como por exemplo, a comunicação e a sincronização, e que diminuem o *speedup* (BARNEY, 2010), (PACHECO, 2011), (MCCOOL, 2012).

$$T_p = \frac{T_1}{p} \tag{2}$$

A eficiência é usada para medir a qualidade de um algoritmo paralelo. Ela caracteriza como um algoritmo utiliza os recursos do *hardware* paralelo (GOMES, 2009). A eficiência relaciona o *speedup* e o número de processadores utilizados para obtê-lo (GOMES, 2009), (MCCOOL, 2012), conforme a equação 3. Quanto mais próximo o valor da eficiência for de 1, melhor será, uma eficiência igual a 1 (referido também como 100%), corresponde ao *speedup* linear (MCCOOL, 2012).

$$E = \frac{speedup}{p} = \frac{\left(\frac{T_1}{T_p}\right)}{p} = \frac{T_1}{p.T_p} \tag{3}$$

2.5 PROGRAMAÇÃO MULTITHREAD COM JAVA

A linguagem de programação Java fornece suporte nativo para desenvolvimento de aplicações *multithread*. A execução *multithread* é uma característica essencial da plataforma Java, ela foi projetada desde seu início para dar suporte a programação *multithread*. Além disso, a partir da sua versão 5.0, foi incluído o pacote *java.util.concurrent*, que possui um conjunto de APIs de alto nível que permitem tratar tarefas concorrentes (ORACLE, 2013).

Cada *thread* Java está associada a uma instância da classe *java.lang.Thread*. Uma aplicação que cria uma instância da classe *Thread* deve fornecer o código que será executado pela *thread*. Isso pode ser feito de três maneiras:

- Instanciar um objeto de uma classe que implemente a interface *java.lang.Runnable* e utilizá-lo como parâmetro para o construtor da classe *Thread*.
- Instanciar um objeto de uma classe que implemente a interface *java.concurrent.Callable* e submeter essa instancia para um objeto executor.

Segundo Oracle (2013) a interface *Callable* é similar a *Runnable,* a principal diferença entre as duas é que *Callable* permite que a *thread* retorne um valor ao término de sua execução enquanto *Runnable* não o permite.

- Criar uma subclasse da classe *Thread* e sobrescrever seu método *run*.

A primeira e a segunda formas permitem maior flexibilidade, pois uma classe *Runnable* ou *Callable* pode ser subclasse de outra classe qualquer, enquanto na terceira forma, ao se estender a classe *Thread*, fica-se impossibilitado estender outra classe, visto que Java não permite herança múltipla.

Há basicamente duas estratégias para utilizar objetos da classe *Thread*:

- Controlar diretamente a criação e gerenciamento das *threads*. Dessa forma o programador deve instanciar a classe *Thread* e utilizar o método *start* para iniciá-la.
- A partir da versão 5.0 da plataforma Java é possível delegar o gerenciamento das *threads* para um objeto executor. O programador submete objetos *Runnable* ou *Callable* para o executor e esse cuida da criação e inicialização das *threads*, além disso, o executor fornece uma série de métodos que permitem gerenciar as *threads*. Com a utilização do objeto executor é possível separar o gerenciamento das *threads* do restante da aplicação. Segundo Oracle (2013), a maioria das implementações de objetos executores permitem ainda utilizar *thread pools*, um recurso que é utilizado para executar múltiplas tarefas e que minimiza o *overhead* devido à criação de várias *threads*.

2.5.1 Sincronização em Java

Na linguagem de programação Java, os monitores podem ser utilizados por meio da utilização da palavra-chave *synchronized* (ORACLE, 2013). Todo objeto Java possui um monitor associado a ele, a utilização da palavra-chave *synchronized* sobre um bloco de código determina que esse bloco terá seu acesso controlado pelo monitor do objeto e assim será acessado por apenas uma *thread* por vez.

Quanto à criação de mecanismos de sincronização por barreira, esses podem ser implementados em Java com a utilização conjunta dos métodos *wait* e *notifyAll*, os quais podem ser invocados a partir de qualquer objeto, funcionando da seguinte forma:

- *wait* – a chamada desse método para um objeto faz com que a *thread* corrente aguarde no monitor desse objeto até que outra *thread* invoque os métodos *notify* ou *notifyAll* para esse objeto. Para poder invocar o método *wait* para um objeto, a *thread* necessita estar acessando o monitor desse objeto. Ao ser invocado, a *thread* libera o acesso ao monitor e espera até que outra *thread* a notifique através da chamada dos métodos *notify* ou *notifyAll*, para ela então continuar seu processamento. Existe também a possibilidade de definir um *timeout* que, ao expirar, faz *thread* deixar o estado de espera independentemente de receber ou não uma notificação dos métodos *notify* ou *notifyAll* (ORACLE, 2013).
- *notifyAll* – a chamada desse método para um objeto faz com que todas as *threads*, que estão aguardando no monitor desse objeto, saiam do estado de espera e continuem seu processamento. Esse método só pode ser invocado por uma *thread* que possui o acesso ao monitor do objeto.

2.6 CONSIDERAÇÕES FINAIS

Neste capítulo foi apresentada uma introdução sobre a computação paralela, englobando aspectos teóricos e práticos. Inicialmente foi apresentado o contexto que resultou na grande inserção de processadores *multicore* no mercado e consequentemente na popularização do *hardware* paralelo. A seguir foram abordadas características das arquiteturas de *hardware* paralelo, focando na arquitetura MIMD e nos sistemas de memória compartilhada e distribuída. Em sequencia foram abordados aspectos do *software* paralelo que o diferenciam do *software* serial, englobando as formas de divisão do problema, a comunicação e a sincronização entre as tarefas paralelas. O capítulo também apresenta duas importantes métricas utilizadas para avaliar o ganho de desempenho alcançado por

meio da aplicação do paralelismo, o *speedup* e a eficiência. Ao final do capítulo é feita uma introdução sobre a implementação de aplicações paralelas, baseadas na utilização de *threads*, utilizando a linguagem de programação Java.

3 ALGORITMOS EVOLUTIVOS

AEs são algoritmos estocásticos de busca que se inspiram no processo de evolução natural (EIBEN, 2003). Eles utilizam a capacidade de evoluir para otimizar soluções (YU, 2010). Segundo Eiben (2011), os AEs formam uma classe de métodos de busca heurística baseados em uma estrutura especial de algoritmos cujos componentes principais são: o operador de seleção, operadores de variação e uma população de indivíduos que representam cada um uma possível solução para o problema.

Os termos utilizados no campo dos AEs são análogos a termos utilizados na biologia. A seguir são apresentados alguns dos principais termos utilizados com breves descrições:

- População: conjunto de indivíduos manipulados pelo AE.
- Indivíduo ou Cromossomo: estrutura de dados que codifica uma possível solução para o problema. Eles são constituídos por genes.
- Gene: parâmetro codificado no indivíduo. Cada gene representa uma característica do problema a ser resolvido.
- Alelo: o valor atribuído a um gene.
- Geração: também chamado de ciclo ou iteração, na qual a população é manipulada.

Segundo Michalewicz (1996), para resolver um problema em particular, um AE deve possuir os seguintes componentes:

- Uma representação genética para as potenciais soluções do problema;
- Uma forma de criar uma população inicial de potenciais soluções;
- Uma função de avaliação que avalia as soluções em termos de sua aptidão;
- Operadores genéticos que alteram a composição dos indivíduos;
- Valores para vários parâmetros que o AE utiliza;

O funcionamento básico dos AEs consiste em tentar encontrar a melhor solução para um problema específico, dentro de um determinado espaço de busca, manipulando uma população de indivíduos por meio da aplicação dos operadores de

seleção e variação. Esse processo ocorre em várias gerações como pode ser visto na Figura 1.

```
Inicialização;
Avaliação;
Enquanto o critério de parada não for alcançado
      Seleção;
      Operadores de Variação;
      Avaliação;
Fim-Enquanto;
Retorna melhor solução ou indivíduo;
```

Figura 1 - Estrutura básica de um AE

O primeiro passo da execução de um AE clássico consiste em gerar uma população de indivíduos para iniciar o processo evolutivo, a população inicial, sendo que esse procedimento é chamado inicialização. Na maioria das situações inicia-se a população de forma aleatória, utilizando uma distribuição uniforme, pois não se tem informação de onde se encontram as melhores soluções dentro do espaço de busca (YU, 2010).

O parâmetro chamado tamanho da população define a quantidade de indivíduos que permanecerão nela. O tamanho da população influencia na capacidade do AE encontrar boas soluções e no tempo necessário para chegar a elas, logo consiste de um importante parâmetro de sua configuração. Se a população for muito pequena ela pode não ser o suficiente para cobrir o espaço de busca de forma satisfatória. Além disso, uma população pequena contribui para uma rápida perda de diversidade genética. Um problema associado a essa rápida perda de diversidade é a convergência prematura da população.

A convergência genética se traduz em uma população com baixa diversidade genética que, por possuir genes similares, não consegue evoluir, a não ser pela ocorrência de mutações. A convergência genética é um fenômeno inerente à evolução da população e necessário para encontrar boas soluções, contudo, quando esta ocorre prematuramente, a população deixa de evoluir antes de se aproximar adequadamente da solução ótima. Quanto menor for a diversidade, mais rápida será a convergência genética. Por outro lado, se a população for muito grande, o AE vai precisar processar muitos indivíduos, o que pode gerar o desperdício de recursos computacionais ao processar indivíduos desnecessários.

Após a criação da população inicial e a cada nova geração, todos os indivíduos da população são avaliados a fim de determinar o quanto cada um deles é apropriado para solucionar o problema, ou seja, o quanto ele está adaptado ao ambiente. A avaliação é o elo entre o AE e o ambiente em que ele está inserido, sendo realizada por meio de uma função objetivo que calcula e fornece um valor numérico, chamado aptidão (*fitness*), para cada indivíduo. Quanto melhor a avaliação de um indivíduo mais forte ele será dentro da população.

Após a avaliação da população inicial, começa a execução das gerações. Em cada geração, os operadores genéticos são aplicados sobre a população, estes são essencialmente modelos computacionais de processos que ocorrem na natureza durante a evolução das espécies (LINDEN, 2012). As gerações são executadas repetidas vezes até que um determinado critério de parada seja satisfeito, como por exemplo: (i) certo número de gerações, (ii) a população deixar de evoluir, ou seja, quando após um determinado número de gerações consecutivas não se observar melhoria na aptidão, ou (iii) o algoritmo alcance a solução ótima conhecida para o problema. A aplicação dos operadores genéticos sobre a população tende a melhorar as características genéticas dos indivíduos e adaptá-los ao ambiente, isto é, melhorar sua aptidão. Dessa forma a população evolui à medida que são criadas novas gerações, ou seja, seus indivíduos ficam mais próximos da solução ótima do problema (YU, 2010).

Para que os indivíduos possam ser manipulados computacionalmente, os mesmos precisam ser representados por uma estrutura de dados. Em outras palavras, essa estrutura de dados representará as possíveis soluções. A representação do cromossomo depende do tipo de problema que se deseja resolver e do que essencialmente se deseja manipular geneticamente. É necessário que a forma de representação permita representar todo o espaço de busca disponível. A forma de representação influencia tanto no desempenho do AE quanto na aplicação dos operadores genéticos, pois existem formas de operadores genéticos que são específicas para cada forma de representação. Normalmente, um indivíduo é representado por um vetor, no entanto, como já foi mencionado, a forma de representação está esta fortemente relacionada com o tipo de problema a ser

solucionado. Portanto outras representações como, por exemplo, na forma de matrizes, ou até mesmo árvores, podem ser encontradas na literatura.

Como já mencionado, a evolução se dá através da aplicação dos operadores genéticos. Os normalmente utilizados são: seleção, cruzamento e mutação. O mecanismo de seleção utilizado pelos AEs simula o processo de seleção natural. Segundo a teoria da evolução de Darwin, quanto melhor um indivíduo se adaptar ao seu meio ambiente, maior será sua chance de sobreviver e gerar descendentes. Da mesma forma, nos AEs a seleção é um processo de escolha, através do qual os indivíduos competem pela sobrevivência para a próxima geração ou pela chance de gerar descendentes (a partir da aplicação dos operadores genéticos) que propagarão seu material genético para a próxima geração (EIBEN, 2003). Isso dependerá do tipo de AE que está sendo executado. Quanto melhor for o *fitness* de um indivíduo, mais forte ele será dentro da população e, normalmente, maior será a probabilidade de ser selecionado.

O operador de cruzamento simula o processo de reprodução sexuada que ocorre na natureza, onde é necessária a participação de mais de um indivíduo. Com a utilização do operador de cruzamento, indivíduos pais trocam informações contidas em seus genes entre si, a fim de criar um ou mais descendentes, os quais possuirão características genéticas de ambos os genitores. Idealmente, quando dois indivíduos fortes trocam genes pelo cruzamento, os filhos gerados serão ainda mais fortes. Dessa forma o cruzamento acaba por contribuir para a evolução da população (HERRERA, 1998). Contudo, as repetidas aplicações do operador de cruzamento, à medida que novas gerações são criadas, tende a tornar a população cada vez mais homogênea e assim diminuir sua diversidade (CANTU-PAZ, 1998).

Diferentemente do operador de cruzamento, a mutação é um operador que envolve apenas um indivíduo, sua aplicação consiste basicamente em alterar os valores dos genes dos indivíduos de forma estocástica para gerar um novo indivíduo com código genético levemente modificado (EIBEN, 2003). A utilização desse operador contribui para o aumento da diversidade da população, pois insere material genético nos indivíduos que não está necessariamente presente em seus pais.

Diversos modelos de AEs são descritos na literatura. Os AEs clássicos como os algoritmos genéticos (AG) e as estratégias evolutivas (EE) (EIBEN, 2003), (YU, 2010), são abrangidos por este trabalho. AGs e EEs compartilham as mesmas características gerais citadas anteriormente, mas diferem em questões mais específicas.

3.1 ALGORITMOS GENÉTICOS

Algoritmos genéticos são algoritmos estocásticos de busca e otimização que se baseiam no processo de evolução de Darwin e na genética (CANTÚ-PAZ, 1998) (LINDEN, 2012). Segundo Linden (2012 p. 387), "a aplicabilidade dos AGs é praticamente infinita – sempre que houver uma necessidade de busca ou otimização, um AG pode ser considerado como uma ferramenta de solução". A Figura 2 mostra a estrutura básica de um AG.

```
Inicialização;
Avaliação;
Enquanto o critério de parada não for alcançado
      Seleção;
      Cruzamento;
      Mutação;
      Avaliação;
Fim-Enquanto;
Retorna melhor solução ou indivíduo;
```

Figura 2 - Estrutura básica de um AG

No algoritmo considerado, os AGs devem manter vários indivíduos em sua população. Cada indivíduo possui duas propriedades: o cromossomo (composto por genes) que define sua localização no espaço de busca e a sua aptidão (*fitness*). Originalmente ao AGs foram propostos com uma representação binária do cromossomo (AG *Binary-Coded*), para imitar a codificação genética dos organismos naturais (YU, 2010). Na codificação binária, os cromossomos assumem uma estrutura de vetor de valores binários, assim cada gene pode assumir apenas os valores "0" ou "1". Nessa forma de representação, frequentemente é necessária a utilização de codificação e decodificação da solução. A representação binária é de fácil manipulação e análise (PACHECO, 1999). Contudo, se o problema tem

múltiplos parâmetros necessitando um maior nível de precisão, será necessária a utilização de cromossomos longos, o que exigirá mais memória e mais processamento (FERREIRA, 2009).

Outra forma comum de representação das indivíduos nos AGs é a forma real (AG *Real-Coded*). Nessa forma de representação os cromossomos assumem a estrutura de vetor de valores reais, assim cada gene pode assumir valores reais limitados pelo domínio do problema. A representação real permite cromossomos menores, sendo de mais fácil compreensão (MICHALEWICZ, 1996) e requer uma quantidade menor de memória do que a representação binária (FERREIRA, 2009). Segundo Michalewicz (1996), um AG *real-coded* pode ser até nove vezes mais rápido que um AG *binary-coded*.

Os indivíduos manipulados pelos AGs são submetidos à seleção, cruzamento e mutação, nessa respectiva ordem. Após a aplicação desses operadores, os indivíduos gerados substituem os genitores e formam a próxima geração, assim, cada indivíduo sobrevive apenas durante uma geração. Uma abordagem alternativa é a utilização do elitismo, que consiste em armazenar os n melhores indivíduos da população atual para substituírem os n piores da próxima geração. Isso evita que indivíduos fortes sejam descartados rapidamente durante o processo evolutivo. Nesse contexto, as próximas seções tratam dos operadores aplicados no processo de evolução dos AGs.

3.1.1 Seleção nos AGs

Nos AGs, após ser executada a avaliação de todos os indivíduos é utilizado o operador de seleção para escolher os indivíduos que terão chance de participar do processo reprodutivo e gerar descendentes que farão parte da próxima geração (LINDEN, 2012). De maneira geral, indivíduos com melhor avaliação terão maior probabilidade de serem selecionados do que aqueles com pior avaliação. Dessa forma a informação contida em indivíduos fortes tem mais chances de serem preservadas e propagadas para a próxima geração. Os indivíduos selecionados se

tornam candidatos a pais, em geral dois pais são selecionados aleatoriamente dentre os indivíduos selecionados para cruzar e gerar um ou mais filhos (YU, 2010).

Apesar de favorecer os indivíduos mais aptos, o processo seletivo em geral não despreza completamente os indivíduos com aptidão baixa, pois até mesmo indivíduos com péssima aptidão podem ter características genéticas que sejam favoráveis à criação de um indivíduo, que pode eventualmente representar a melhor solução para o problema que se quer resolver. Além disso, caso sejam selecionados apenas os melhores indivíduos para participarem do processo reprodutivo, a população tenderá a ser formada por indivíduos cada vez mais semelhantes e assim faltará diversidade a essa população, o que pode impedir que a evolução prossiga de forma satisfatória (LINDEN, 2012). Dentre as diversas formas de seleção os métodos por roleta viciada e por torneio são os mais comuns, por isso ambas são utilizadas neste trabalho.

No método de seleção por roleta viciada, os indivíduos são selecionados utilizando um algoritmo que cria uma roleta, semelhante à roleta utilizada em jogos de azar (FERREIRA, 2009). Cada indivíduo ocupa uma porcentagem da roleta proporcional à sua aptidão, dessa forma, indivíduos com maior aptidão ocuparão uma parte maior da roleta e assim terão maior probabilidade de serem sorteados quando ela for girada. No entanto, mesmo indivíduos com baixa aptidão têm chance de serem selecionados. A Tabela 1 apresenta um exemplo de valores utilizados em uma roleta viciada.

Tabela 1 - Exemplo de valores utilizados em da Roleta Viciada

Indivíduo	Aptidão Absoluta	Aptidão Relativa	q_i
1	2	0,052631579	0,052631579
2	4	0,105263158	0,157894737
3	5	0,131578947	0,289473684
4	9	0,236842105	0,526315789
5	18	0,473684211	1
Total	38	1	

A aptidão relativa de um indivíduo é o resultado da divisão de sua aptidão absoluta pela soma das aptidões de toda a população. A soma de todas as aptidões relativas é igual a 1, o que representa os 100% da roleta. Para cada indivíduo é associado um valor q_i que é a soma das aptidões do indivíduo corrente e dos

anteriores posicionados na roleta. Para girar a roleta, é gerado um número aleatório R no intervalo $[0,1]$, esse número "aponta" para um ponto na roleta. O indivíduo que tiver o maior valor de q_i que seja menor ou igual a R é o indivíduo selecionado. A roleta é girada repetidas vezes até que a quantidade de indivíduos selecionados seja igual ao tamanho da população.

No método de seleção por torneio, são escolhidos $n > 1$ indivíduos aleatoriamente para participarem do torneio, sendo n o tamanho do torneio. Os indivíduos escolhidos competem diretamente pelo direito de ser selecionado para participar do processo reprodutivo. O indivíduo que tiver a melhor aptidão, dentre os participantes, vencerá o torneio e será selecionado. Quanto maior o tamanho do torneio maior é a chance de dominância do melhor indivíduo, pois esse terá mais chance de ser um participante do torneio e acabará sendo o vencedor. São disputados torneios até que a quantidade de vencedores seja igual ao tamanho da população.

3.1.2 Cruzamento nos AGs

Nos AGs o cruzamento é executado após o processo de seleção. Relacionado ao operador de cruzamento tem-se a taxa de cruzamento, um parâmetro de configuração dos AGs que determina a probabilidade de haverem cruzamentos. Se esse valor for muito alto, pode ocorrer uma convergência prematura. Por outro lado, se ele for muito baixo, o processo evolutivo poderá ser demasiadamente lento. O valor para esta taxa é normalmente alto, situando-se entre 50 e 80 por cento (GOLDBERG, 1989), (LACERDA, 1999). Embora nos AGs o cruzamento seja considerado o principal operador genético, os efeitos de sua aplicação podem levar a uma convergência prematura da solução, pois a população tende a ficar mais homogênea, devido às suas repetidas aplicações, à medida que são criadas novas gerações (LINDEN, 2012). As estratégias de cruzamento utilizadas neste trabalho são aplicáveis aos AGs *Real-Coded*, visto que apenas esses foram abrangidos por este trabalho, são elas: cruzamento de um ponto, cruzamento heurístico e cruzamento linear.

O cruzamento de um ponto foi o primeiro operador de cruzamento a ser utilizado (PINHEIRO, 2007). Para executar o cruzamento de um ponto, é escolhido um ponto onde os vetores de genes dos genitores serão divididos (ponto de corte). Um descendente é criado juntando-se os genes à esquerda do ponto de corte de um genitor e dos genes a direita do ponto de corte do outro genitor. As partes restantes dão origem ao segundo descendente. A Figura 3 ilustra a execução do cruzamento de um ponto.

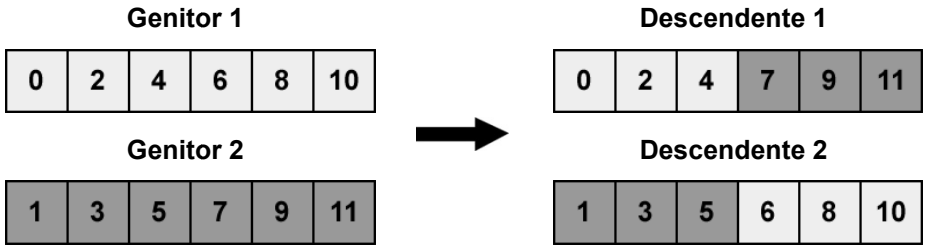

Figura 3 - Cruzamento de um ponto

No cruzamento linear dois indivíduos genitores P_1 e P_2 , geram três descendentes D_1, D_2 e D_3, conforme as Equações 4, 5 e 6, na qual d_i é o gene localizado no índice i do vetor de genes de um descendente D e p_i é o gene localizado no índice i do vetor de genes de um pai P.

$$d_i^1 = \frac{p_i^1}{2} + \frac{p_i^2}{2} \quad (4)$$

$$d_i^2 = \frac{3p_i^1}{2} - \frac{p_i^2}{2} \quad (5)$$

$$d_i^3 = -\frac{p_i^1}{2} + \frac{3p_i^2}{2} \quad (6)$$

Como se pode verificar, um dos descendentes irá ser a média dos dois pais, outro irá ser menor do que o menor dos pais, e o terceiro será maior que o maior dos pais (PINHEIRO, 2007). Dos descendentes gerados, apenas o com melhor aptidão fará parte da próxima geração, os restantes são descartados.

Ao utilizar o cruzamento heurístico, a partir de dois indivíduos genitoresP_1 e P_2, é gerado um único descendente D, de acordo com a Equação 7, na qual r é um número aleatório gerado a partir de uma distribuição uniforme no intervalo [0,1]. Por gerar apenas um descendente, o cruzamento heurístico tende a ser consideravelmente rápido.

$$d_i = r(p_i^1 - p_i^2) + p_i^1, se\ fitness(P_1) > fitness(P_2)$$
$$d_i = r(p_i^2 - p_i^1) + p_i^2, se\ fitness(P_2) > fitness(P_1) \quad (7)$$

3.1.3 Mutação nos AGs

Nos AGs, após a aplicação do operador de cruzamento, os indivíduos são submetidos ao operador de mutação, esse operador geralmente é considerado um operador secundário (CANTU-PAZ, 1998). A mutação é utilizada nos AGs com o objetivo de aumentar a diversidade da população que tende a se tornar homogênea com o passar das gerações, devido à aplicação do operador de cruzamento (CANTU-PAZ, 1998), (FERREIRA, 2009), dessa forma a utilização do operador de mutação tende a contribuir para evitar a convergência prematura da solução e para que o algoritmo saia de ótimos locais. Além disso, o operador de mutação permite que o AG alcance qualquer ponto do espaço de busca.

O intuito da mutação nos AGs é causar pequenas modificações nos indivíduos que foram gerados a partir da aplicação do operador de cruzamento (YU, 2010). Todos os genes de todos os indivíduos são passíveis de sofrerem mutação. A probabilidade de que ocorra mutação em um gene c_i é determinada por um parâmetro do AG chamado taxa de mutação, caso essa taxa seja muito alta, a busca acabará se tornando essencialmente aleatória. O valor atribuído à taxa de mutação é geralmente baixo, situando-se entre 1 e 10 por cento (GOLDBERG, 1989) (LACERDA, 1999). As estratégias de mutação utilizadas neste trabalho são aplicáveis aos AGs *Real-Coded*, são elas: mutação aleatória e mutação *creep*.

A mutação aleatória consiste em substituir o valor do gene a ser mutado por um valor aleatório, gerado a partir de uma distribuição normal em um intervalo definido pelo domínio do problema. Esse tipo de mutação pode causar uma grande mudança na posição do indivíduo dentro do espaço de busca, favorecendo a exploração, no entanto, próximo ao fim da execução do AG, quando se espera que os indivíduos já estejam próximos a boas soluções e não se deseja que eles se afastem dessas, pode ser mais eficiente utilizar uma forma de mutação que seja menos agressiva em termos de mudança (LINDEN, 2012).

A mutação *creep* consiste em somar ao valor do gene a ser mutado um valor gerado a partir de uma distribuição normal com média zero e desvio padrão baixo. Uma alternativa é multiplicar-se o valor do gene por um número aleatório próximo a 1. Esse tipo de mutação causa apenas uma pequena perturbação no indivíduo e auxilia na exploração local do espaço de busca. Parte da ideia de que se um indivíduo está perto de um valor ótimo, uma pequena alteração pode levá-lo até esse valor ótimo (SILVA, 2005).

3.2 ESTRATÉGIAS EVOLUTIVAS

As estratégias evolutivas (EE) foram concebidas como heurísticas de busca para a otimização de problemas de engenharia e demonstraram grande sucesso na área de otimização de parâmetros numéricos (LINDEN, 2012). Ao projetar as EE, seus criadores utilizaram números reais para representar os alelos de seus genes e uma mutação baseada em uma distribuição normal como a técnica de exploração mais importante sobre um espaço de busca (EIBEN, 2003), (YU, 2010). A estrutura básica de uma EE é apresentada na Figura 4.

```
Inicialização;
Avaliação;
Enquanto o critério de parada não for alcançado
      Mutação;
      Cruzamento;
      Avaliação;
      Seleção;
Fim-Enquanto;
Retorna melhor solução ou indivíduo;
```

Figura 4 - Estrutura básica de uma EE

Os indivíduos manipulados pelas EEs são representados por um par de vetores de números reais, na forma $v = (x, \sigma)$, onde x é o vetor de genes que representa o ponto no espaço de busca e σ o vetor de valores de desvio padrão associados a cada valor do vetor x, o qual será utilizado para determinar a mutação do gene (CORTES, 2000). Pode-se também utilizar um único valor σ na mutação de todos os genes do indivíduo ao invés de um valor para cada gene (YU, 2010). Nas

versões atuais das EEs, a descendência é obtida submetendo-se os indivíduos da população a dois operadores de variação: cruzamento e mutação (CORTES, 2000).

Observa-se que o parâmetro σutilizado na mutação dos genes, também está sujeito ao processo de evolução por meio da mutação. Esta é uma característica fundamental das EEs, que permite a auto-adaptação dos parâmetros de mutação (CORTES, 2000). Em geral a auto-adaptação significa que alguns parâmetros do AE variam durante a sua execução. Nas EEs os parâmetros de mutação são incluídos no cromossomo e estes co-evoluem juntamente com as soluções. As primeiras EEs não possuíam essa característica. A auto-adaptação de parâmetros foi inserida somente na década de 70, sendo utilizada na maioria das EEs modernas. (EIBEN, 2003). A auto-adaptação de parâmetros é uma das grandes vantagens das EEs, devido a essa característica, nas EEs não há necessidade de que se defina uma taxa de mutação fixa (LINDEN, 2012).

3.2.1 Mutação nas EEs

O operador de mutação produz um indivíduo mutante, em primeiro lugar mutando os valores de desvio padrão e, em seguida, mutando os genes do indivíduo genitor (PEREIRA, 2001). A mutação é considerada o principal operador de variação nas EEs, é um operador que envolve apenas um genitor e gera apenas um descendente (YU, 2010). A mutação dos genes dos indivíduos nas EEs consiste em criar uma perturbação no valor de cada gene, através da adição de um valor aleatório a cada um deles. Para obter esse valor aleatório, comumente é utilizada uma distribuição Gaussiana com média zero e desvio padrão σ, dessa forma, o valor de cada gene do indivíduo descendente é obtido pela Equação 8 (CORTES, 2000), (EIBEN, 2003) onde x_i^j é o gene na posição i do vetor de genes do indivíduo j da população e $N(0,\sigma)$ é um número aleatório Gaussiano com média 0 e desvio σ.

$$x_i^j = x_i^j + N(0,\sigma) \tag{8}$$

Cada gene possui um valor de σ associado a si, esse é um parâmetro do algoritmo que determina a extensão com que os valores dos genes são perturbados pelo operador de mutação. Dessa forma o parâmetro σ é frequentemente chamado

de *mutation step size* (EIBEN, 2003). Os valores dos parâmetros de mutação também estão sujeitos a mutação como ilustrado pela Equação 9 (CORTES, 2000), (PEREIRA, 2001).

$$\sigma_j^j = \sigma_j^j \times \exp\left(t'N(0,1) + tN_j(0,1)\right) \quad (9)$$

Na Equação 9, $\mathrm{N}(0,1)$ representa um número aleatório Gaussiano com média 0 e desvio padrão 1, esse número é gerado apenas uma vez para cada indivíduo. Dessa forma o fator $\mathrm{t}'\mathrm{N}(0,1)$ permite uma mudança global da mutabilidade do indivíduo. $\mathrm{N_j}\ (0,1)$ também representa um número aleatório Gaussiano, no entanto, esse deve ser gerado uma vez para cada gene do indivíduo. Dessa forma, o fator $\mathrm{tN_j}\ (0,1)$ permite mudanças individuais para cada valor σ_j (PEREIRA, 2001). Os valores t e t' foram sugeridos por Bäck (1997) como sendo $\mathrm{t} = (\sqrt[4]{4\mathrm{n}})^{-1}$ e $\mathrm{t}' = (\sqrt{2\mathrm{n}})^{-1}$, onde n é a quantidade de genes do indivíduo.

Outras formas de mutação nas EEs são possíveis, como por exemplo, a mutação Cauchy. A única diferença entre essa estratégia de mutação e a mutação Gaussiana, descrita acima, é que a Equação 8 é substituída pela Equação 10.

$$x_i^j = x_i^j + \sigma\delta \quad (10)$$

Na qual δ é um número aleatório de Cauchy com parâmetro escalar $\mathrm{t} = 1$. Esse valor é gerado uma vez para cada valor de j.

3.2.2 Cruzamento nas EEs

Nas EEs, o cruzamento é considerado uma operação secundária de exploração do espaço de busca (YU, 2010). A seleção dos indivíduos que participarão do cruzamento não é baseada no valor do *fitness* (diferentemente do que ocorre nos AGs), os indivíduos são selecionados aleatoriamente da população por meio de uma distribuição uniforme (EIBEN, 2003). Dessa forma, todos os indivíduos da população são candidatos a genitores e possuem as mesmas chances de gerar descendentes pelo cruzamento. São executados cruzamentos até que a quantidade de descendentes gerados seja igual a λ. O valor de λ frequentemente é

maior que a quantidade de indivíduos na população (YU, 2010). O esquema básico de cruzamento nas EEs envolve dois genitores que criam um único descendente (EIBEN, 2003). Existem diferentes mecanismos de cruzamento que podem ser utilizados pelas EEs.

Considerando os indivíduos da população no formato $v = (x, \sigma)$ onde x é um vetor de números reais que representa os genes do indivíduo eσ é um vetor de números reais que representa os valores de desvio padrão que serão utilizados para a mutação de cada gene, o cruzamento intermediário gera apenas um descendente x', a partir de dois genitores x_1 e x_2, tirando a média de seus valores, por meio da aplicação da Equação 11 (YU, 2010).

$$(x', \sigma') = \left(\frac{x_1 + x_2}{2}, \frac{\sigma_1 + \sigma_2}{2}\right) \quad (11)$$

Ao utilizar-se o cruzamento discreto, cada componente do indivíduo filho, ou seja, cada valor de x e σ, é herdado aleatoriamente a partir de um dos genitores (PEREIRA, 2001).

3.2.3 Seleção nas EEs

Nas EEs o objetivo da aplicação do operador de seleção é escolher os indivíduos que sobreviverão após o processo reprodutivo e farão parte da próxima geração, os indivíduos com melhor aptidão sobrevivem e os restantes são eliminados da população. A operação do operador de seleção é diretamente afetada pelo formato da EE. A seguir são apresentados alguns dos formatos comuns.

A primeira versão das EEs mantinha em sua população apenas um indivíduo e seu descendente, o que foi denominado $(1 + 1) - \mathrm{EE}$. Nesse esquema era utilizado apenas o operador de mutação, que gerava um descendente a partir de um genitor e por meio da aplicação do operador de seleção os dois competiam pela sobrevivência, apenas aquele com melhor avaliação sobrevivia e avançava para a próxima geração (LINDEN, 2012). Um aspecto negativo observado nas $(1 + 1) - \mathrm{EE}$ é a convergência lenta, além da busca ponto a ponto ser suscetível a estagnar em ótimos locais. Posteriormente outras versões de EE foram desenvolvidas com o

objetivo de resolver tais problemas. Estas estratégias são denominadas multi-indivíduos, onde o tamanho da população é maior que 1 (CORTES, 2000). Ao permitir mais membros na população passou-se também a introduzir a utilização do operador de cruzamento (LINDEN, 2012).

Na versão $(\mu + \lambda) - \mathrm{EE}$, μ indivíduos geram λ descendentes, a partir da aplicação dos operadores genéticos, os indivíduos genitores e seus descendentes permanecem em uma população temporária de tamanho $\lambda + \mu$, em seguida todos os indivíduos nessa população temporária competem pela sobrevivência para a próxima geração. O operador de seleção é executado após a aplicação dos operadores genéticos e elimina os indivíduos com pior avaliação, o que significa que esse formato de EE é inerentemente elitista desde sua concepção (LINDEN, 2012).

Na forma $(\mu,\lambda)-\mathrm{EE}$, μ indivíduos produzem λ descendentes, a partir da aplicação dos operadores genéticos, com $\mu < \lambda$. O operador de seleção é aplicado apenas sobre os λ descendentes eliminando os de pior avaliação. Os indivíduos pais são eliminados automaticamente e os descendentes selecionados formarão a próxima geração, dessa forma o período de vida de cada indivíduo é limitado a apenas uma geração. Esse tipo de estratégia tem bom desempenho em problemas onde o ponto ótimo é em função do tempo, ou onde a função é afetada por ruído (CORTES, 2000).

3.3 ALGORITMOS GENÉTICOS x ESTRATÉGIAS EVOLUTIVAS

Tendo apresentado os algoritmos evolutivos utilizados neste trabalho, é importante identificar as diferenças e similaridades existentes entre eles. Segundo Michalewicz (1999), Herrera (1998) e Cortes (2005), as diferenças básicas entre AGs e EEs podem ser resumidas em:

- Quanto à representação dos indivíduos - nos AGs (*real-coded*) um vetor de números reais é o suficiente para representar o indivíduo, enquanto nas EEs é necessário um segundo vetor, também de números reais, que contém valores de desvio padrão que serão utilizados no momento da mutação. Além

disso, nos AGs também é possível utilizar a forma de representação binária dos indivíduos (AG *binary-coded*).

- Quanto à aplicação dos operadores genéticos - embora AGs e EEs utilizem os mesmo operadores genéticos (seleção, cruzamento e mutação) nos AGs os indivíduos são primeiramente selecionados e a seguir submetidos à aplicação do cruzamento e da mutação, enquanto nas EEs a ordem é inversa, ou seja, utilizam-se primeiro o cruzamento e a mutação e em seguida é executada a seleção. Nos AGs o cruzamento é considerado o principal operador genético, enquanto nas EEs o principal operador é a mutação. A função do operador de seleção também é diferente nos os AGs e nas EEs, no primeiro sua função é escolher os indivíduos que terão chance de participar do processo reprodutivo, enquanto no segundo, a função é definir os indivíduos que irão sobreviver e passar para a próxima geração.
- Quanto à política de substituição dos indivíduos a cada nova geração - nos AGs, em geral, os indivíduos gerados pela aplicação dos operadores genéticos substituem os membros da atual geração quando o elitismo não é utilizado. Por outro lado, existe a opção de utilizar o elitismo. Isso evita que indivíduos fortes sejam descartados rapidamente durante o processo evolucionário. Diferentemente, nas EEs simplesmente são selecionados os indivíduos mais aptos para comporem a próxima geração, podendo essa seleção englobar apenas os indivíduos filhos, na estratégia $(\mu, \lambda) - EE$, ou os filhos juntamente com os pais, na estratégia $(\mu + \lambda) - EE$.

3.4 ALGORITMOS EVOLUTIVOS PARALELOS

Uma das virtudes frequentemente observadas dos AEs é seu paralelismo "natural", além disso, a crescente disponibilidade de *hardware* paralelo oferece uma oportunidade tentadora para explorar o poder do paralelismo nos AEs de forma a permitir que os mesmos resolvam classes maiores e mais complexas de problemas. (DE JONG, 2006).

Em geral os AEs são capazes de encontrar boas soluções em uma quantidade razoável de tempo. Contudo, à medida que eles são aplicados na

resolução de problemas maiores e mais complexos, há um aumento significativo no tempo de execução necessário para encontrar soluções adequadas. Uma das possibilidades mais promissoras para aumentar a sua velocidade é a utilização de implementações paralelas (CANTÚ-PAZ, 1998). A utilização do paralelismo permite um aumento da velocidade de processamento, pois aumenta a potência computacional.

A ideia principal por trás da computação paralela consiste em dividir uma tarefa em partes menores e resolver essas partes simultaneamente utilizando diferentes unidades de processamento. Nesse contexto, os AEs podem ser considerados, pois eles possuem paralelismo inerente, haja vista que eles se baseiam em manter e evoluir vários indivíduos simultaneamente (HE, 2006) (LINDEN, 2012). Ao dividir o processamento da população por vários elementos de processamento, os AEPs permitem alcançar resultados de alta qualidade em um tempo de execução razoável até mesmo na resolução de problemas complexos de otimização (NESMACHNOW, 2012). Os benefícios provenientes da utilização correta de paralelismo nos AEs incluem: encontrar soluções alternativas para o mesmo problema; busca simultânea em vários pontos no espaço e busca mais e busca mais rápida (ALBA, 2002).

O paralelismo pode ser aplicado de várias formas diferentes na implementação dos AEs, originando assim diversos modelos de AEPs. De acordo com Cantú-Paz (1998), De Jong (2006) e Linden (2012) existem três modelos básicos para aplicar o paralelismo na implementação dos AEs: mestre-escravo, vizinhança e ilha. Combinando esses modelos básicos em um mesmo algoritmo pode-se ainda obter modelos híbridos. Diferentes nomes podem ser atribuídos a esses modelos, no entanto eles mantêm as mesmas características. Neste trabalho foram implementados todos os três modelos de paralelismo supracitados.

Os diferentes modelos de AEP podem ser implementados de forma síncrona ou assíncrona. No primeiro caso, um processador espera até que os outros completem uma determinada etapa de sua execução para então prosseguir com sua própria execução. Por outro lado, no formato assíncrono, a execução de cada processador não depende do término da execução dos demais processadores envolvidos, assim não existe a necessidade dos processadores aguardarem uns

pelos outros para continuarem suas execuções. Todos os AEPs implementados neste trabalho são síncronos.

3.4.1 AEP Mestre-Escravo

Também conhecido como modelo global. Os AEPs do tipo mestre-escravo utilizam uma unidade de processamento principal (mestre) que mantém a população e que delega a avaliação dos indivíduos e/ou a aplicação dos operadores genéticos para serem executados em paralelo por unidades de processamento secundárias (escravos).

A maneira mais simples e comum de implementar AEPs mestre-escravo é distribuir a avaliação do *fitness* pelos escravos enquanto o mestre mantém a população e executa os operadores genéticos (CANTÚ-PAZ, 2007). Isso, pois a aptidão de um indivíduo independe do restante da população. Assim não há necessidade de comunicação entre as unidades de processamento durante a fase de avaliação (CANTÚ-PAZ, 1998). Se o AE está executando utilizando apenas uma unidade de processamento e a avaliação dos indivíduos é muito cara computacionalmente, uma configuração mestre-escravo, na qual os escravos são utilizados para calcular a aptidão dos indivíduos em paralelo, é uma maneira natural de acelerar o tempo de execução (DE JONG, 2006).

A Figura 5 ilustra a estrutura de um AEP mestre-escravo, com a paralelização da avaliação dos indivíduos, nesse caso, o processador mestre envia uma parcela da população para cada um dos escravos disponíveis, para que esses executem a avaliação dos indivíduos paralelamente e retornem os resultados das avaliações para o mestre. Os AEPs mestre-escravo nesse formato são interessantes, pois eles exploram o espaço de busca exatamente da mesma maneira que um AE serial, a única diferença é a sua velocidade de execução, dessa forma, as configurações de parâmetros utilizadas para um AE serial são diretamente aplicáveis, com os mesmos efeitos sobre a busca, em um AEP mestre-escravo desse tipo (CANTÚ-PAZ, 2007).

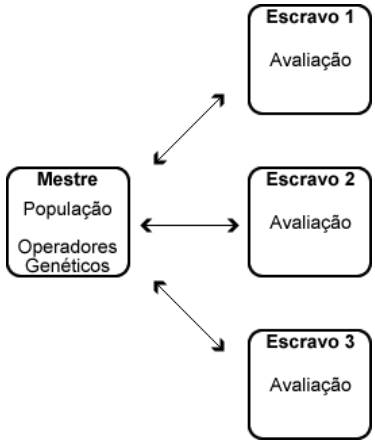

Figura 5 - AEP mestre-escravo com paralelização da avaliação dos indivíduos

3.4.2 AEP de Vizinhança

Também conhecido como modelo de granularidade fina ou modelo celular. Assim como o AEP mestre-escravo, o AEP de vizinhança evolui apenas uma população. Cada indivíduo é colocado em uma célula de uma grade de unidades de processamento e a aplicação da seleção e do cruzamento são restritas aos vizinhos na grade (LINDEN, 2012). Contudo, a sobreposição de vizinhanças permite uma maior interação entre os indivíduos. Os AEPs de vizinhança têm comportamento de busca diferente dos AEs seriais (CANTU-PAZ, 1998). Isso se dá devido à limitação da interação entre os indivíduos nesse modelo de paralelismo. Enquanto em um AE serial um indivíduo tem possibilidades de interagir com qualquer outro indivíduo da população, no modelo de vizinhança, cada indivíduo só pode interagir com seus vizinhos. A Figura 6 ilustra um AEP de vizinhança.

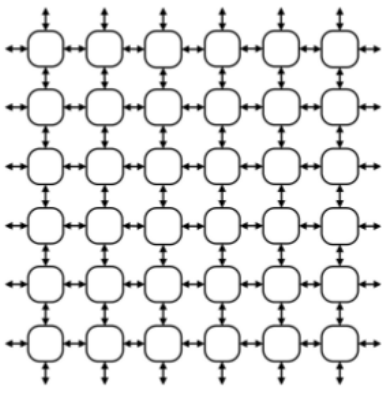

Figura 6 - AEP de vizinhança

A forma mais natural de implementar esse modelo de paralelismo nos AEs é tendo um único indivíduo em cada célula e um simples $(1 + 1) - AE$ executando em cada uma delas,no qual permanecem na população apenas um indivíduo e seu descendente, em cada geração é gerado apenas um descendente que condicionalmente substitui o indivíduo residente na célula (DE JONG, 2006). Para executar o cruzamento, o AE seleciona um indivíduo dentre os residentes nas células vizinhas para ser um dos genitores, sendo que o segundo genitor é o indivíduo residente na sua célula (LINDEN, 2012). Após a aplicação dos operadores genéticos, o descendente gerado substitui o indivíduo ocupante da célula (DE JONG, 2006). Diferentemente do AEP mestre-escravo (com paralelização da avaliação da aptidão) o AEP de vizinhança tem um comportamento diferente durante a busca se comparado a um AE serial, isso se dá devido limitação do alcance dos operadores de seleção e cruzamento aos vizinhos da grade.

No modelo de vizinhança e também importante que cada indivíduo tenha uma quantidade mínima de vizinhos, pois caso contrário poderá não haver uma diversidade satisfatória na população. Entretanto, diversos trabalhos apontam que o desempenho do AEP de vizinhança piora quando se aumenta de forma excessiva o tamanho da vizinhança (LINDEN, 2012).

O modelo de paralelismo de vizinhança, no formato que foi descrito previamente, é apropriado para ser utilizado em computadores maciçamente paralelos com uma grande quantidade de processadores (DE JONG, 2006) (LINDEN, 2012). Dessa forma, neste trabalho foi implementado também um modelo de paralelismo de vizinhança no qual cada célula mantém uma quantidade n de indivíduos e executa um AE que, ao aplicar os operadores genéticos, tem acesso aos seus indivíduos e aos indivíduos residentes nas células vizinhas. Ao final de cada geração, os descendentes gerados substituem a população local da célula, conforme a política de substituição utilizada pelo AE que estiver sendo executado. No restante desse trabalho e na ferramenta desenvolvida, esse modelo de paralelismo é referenciado como Vizinhança Multi-indivíduos.

3.4.3 AEP Ilha

Também conhecido como modelo de granularidade grossa. O modelo de ilha é o mais popular e sofisticado tipo de AE paralelo, ele consiste de várias sub-populações que evoluem paralelamente de forma independente e trocam indivíduos entre si (CANTÚ-PAZ, 2007). Cada ilha executa um AE serial sobre sua população particular e os indivíduos são capazes de interagir, por meio dos operadores de seleção de variação, apenas com os outros indivíduos da mesma ilha. Esse modelo de paralelismo define um operador adicional: a migração. Esse consiste basicamente em, ocasionalmente, alguns indivíduos serem trocados entre as ilhas (NESMACHNOW, 2012).

Ao dividir a população, os AEPs do tipo ilha conseguem atingir alta eficiência computacional devido à interação limitada entre os indivíduos e à redução do tamanho da população em cada ilha. No entanto, por normalmente utilizarem populações menores que a do algoritmo serial, as ilhas tendem a perder a diversidade mais rapidamente e a busca sofre mais facilmente o efeito de convergência prematura da solução, o que conduz a uma situação de estagnação da evolução da população. Em contrapartida, a utilização do operador de migração introduz uma nova fonte de diversidade em cada ilha (NESMACHNOW, 2012), ajudando assim, a prevenir uma possível convergência prematura da solução ou até mesmo a escapar de um ótimo local.

Com a introdução do operador de migração, novos parâmetros precisam ser definidos, são eles: o número de indivíduos que serão trocados entre as ilhas (taxa de migração), a frequência com que ocorrerão as migrações (frequência de migração) e a topologia que conecta as ilhas (topologia de migração), a qual define as ligações entre elas por onde ocorrerão as migrações (CANTÚ-PAZ, 1998), (ALBA, 2002). Geralmente nos AEPs do tipo ilha é utilizada a topologia de anel unidirecional (NESMACHNOW, 2012), outras possíveis topologias comuns são: anel bidirecional e malha. A Figura 7 mostra o esquema de um AEP do tipo ilha utilizando a topologia de anel unidirecional.

A configuração correta dos parâmetros de migração é fundamental para otimizar a eficiência do AEP do tipo ilha. Quanto à topologia, caso a conectividade

entre as populações seja muito densa, as boas soluções serão disseminadas mais rapidamente, mas haverá custos mais altos com comunicação. Por outro lado, se a conectividade for muito esparsa, as boas soluções se espalharão mais lentamente, mas esse maior isolamento das populações é propício para surgimento de uma variedade maior de soluções diferentes (FERREIRA, 2009).

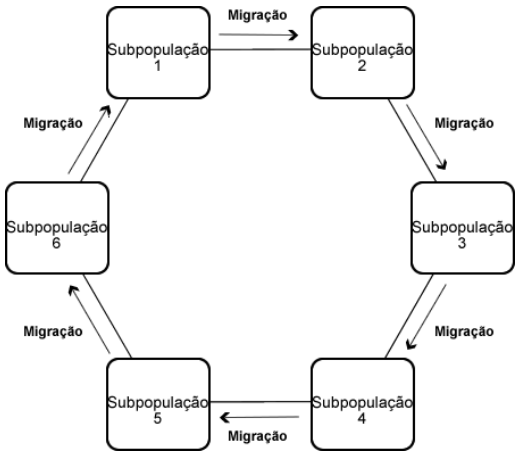

Figura 7 - Esquema de uma AEP do tipo ilha com topologia de anel unidirecional

Quanto à taxa e frequência de migração, muitas migrações de muitos indivíduos por migração podem resultar em uma população, que tenha indivíduos muito fortes, forçando outra a convergir para o mesmo ponto no espaço de busca onde a primeira está mais concentrada (FERREIRA, 2009). Além disso, pode-se perder o efeito da evolução em separado e a variabilidade genética pode desaparecer rapidamente (LINDEN, 2012). Por outro lado, poucas migrações de poucos indivíduos por migração podem não surtir efeito significativo no sentido de aumentar a diversidade e prevenir a convergência prematura da solução (FERREIRA, 2009).

Outro fator importante no projeto de um AEP do tipo ilha é a escolha da política de seleção dos indivíduos migrantes. Normalmente se escolhe os melhores indivíduos de cada ilha para migrar para as outras, o que garante que as melhores soluções de cada ilha serão disseminadas para as ilhas vizinhas. Contudo, essa estratégia tende a causar uma convergência genética mais veloz. Outra opção é escolher de forma aleatória os indivíduos que irão migrar (LINDEN, 2012).

O sentimento comum é que a utilização do modelo de paralelismo de ilha pode melhorar significativamente a capacidade de resolução de problemas dos AEs,

contudo é necessária uma compreensão profunda de seus parâmetros e particularidades, a fim de projetar esses AEPs de forma eficaz para aplicações específicas (DE JONG, 2006).

3.5 CONSIDERAÇÕES FINAIS

Este capítulo apresentou inicialmente os conceitos e características gerais dos AEs. A seguir foram abordados especificamente os AGs a as EEs, descrevendo suas principais características e o papel da seleção, cruzamento e mutação em cada um deles, também foram apresentadas as estratégias mais comuns de aplicação de cada um desses operadores em cada um dos algoritmos. Além disso, foram identificadas as diferenças e similaridades existentes entre esses dois modelos de AE, a partir daí, foi observado que esses dois modelos de AE possuem as mesmas características gerais, mas diferem em aspectos específicos. Um resumo dessas características pode ser visto na Tabela 2. Por fim, o capítulo abordou os AEPs, apresentando a ideia central de seu desenvolvimento e os modelos mais comuns de aplicação do paralelismo nos AEs.

Tabela 2 - Resumo comparativo entre AGs e EEs

	Representação dos Indivíduos	**Aplicação dos operadores genéticos**	**Composição das novas gerações**
AGs	Vetor de números reais (AG *Real-Coded*).	Primeiro os indivíduos são selecionados e a seguir são aplicados o cruzamento e a mutação. A função da seleção é selecionar candidatos a genitores. O operador mais importante é o cruzamento.	Os descendentes substituem a população atual (pode-se utilizar o elitismo).
EEs	Par de vetores de números reais.	Primeiro são aplicados mutação e cruzamento e a seguir os indivíduos são selecionados. A função da seleção é selecionar os indivíduos que	Os indivíduos mais aptos compõem a próxima geração. Esses são selecionados dentre os descendentes, ou dentre os descendentes

		avançarão para a próxima geração. O operador mais importante é a mutação.	juntamente seus genitores.

4 GERAÇÃO AUTOMÁTICA DE CÓDIGO

Geração de código pode ser definida como a técnica de construir e utilizar programas que escrevem outros programas A ideia central é criar código-fonte consistente e de alta qualidade rapidamente (HERRINGTON, 2003). Assumindo geradores de código como programas que escrevem outros programas podem-se enquadrar nessa categoria de *software* inclusive os compiladores, que a partir de um programa escrito em linguagem de mais alto nível, geram programas em código de máquina, no entanto, neste trabalho utiliza-se esse termo para descrever apenas programas que são capazes de gerar código fonte de outros programas escritos em linguagem de alto nível. Um exemplo concreto desse tipo de aplicação é um *software* que se conecta a uma base de dados e gera código-fonte que implementa uma camada de persistência para seu acesso.

A utilização de técnicas de geração automática de código-fonte pode prover uma série de benefícios ao desenvolvimento de *software*, tais como:

- Desenvolvimento ágil - geradores de código entregam o código-fonte mais rapidamente do que o faria uma pessoa que o escrevesse à mão, reduzindo também o custo do desenvolvimento. Além disso, espera-se que o código gerado seja livre de erros, com os quais o programador possivelmente se depararia e teria que lidar, caso escrevesse o código a mão.
- Consistência - geradores de código podem ser utilizados para tornar homogênea a aplicação de padrões, tanto no que diz respeito à aplicação de padrões de projeto quanto a convenções de codificação, evitando quebras desses padrões por parte dos programadores.
- Ponto central de conhecimento - uma mudança feita em um arquivo de definição utilizado para a geração de código pode ser propagada rapidamente para todos os arquivos criados a partir dessa definição apenas gerando o código novamente, enquanto em uma codificação manual, essas modificações teriam que ser feitas pelos programadores em todos os arquivos afetados, individualmente.

Segundo Herrington (2003), a construção de um gerador de código genérico pode seguir as seguintes etapas básicas de desenvolvimento:

- Implementar um código de teste: em primeiro lugar, deve-se desenvolver um código tal o qual se deseja que o gerador seja capaz de criar. Assim será definida qual saída se deseja obter com a execução do gerador.
- Projetar o gerador: deve-se definir o tipo de entrada que será utilizada pelo gerador e como ela será analisada para obter informações que permitam gerar uma saída formatada de acordo com as expectativas definidas previamente no passo anterior. Deve-se também esboçar o fluxo de execução do gerador.
- Criação do mecanismo que gere a saída desejada a partir das informações extraídas da entrada.

Ferramentas geradoras de código podem ser de uso específico e gerarem código para resolver apenas um determinado problema, sendo dificilmente empregáveis na resolução de problemas diferentes, ou, podem ser de uso geral, podendo ser utilizadas para resolver, em teoria, qualquer tipo de problema (LUCAS, 2005), contudo, os geradores de uso geral tendem a produzir código menos completo que os de uso específico. Nesse contexto, dentre as possíveis estratégias de geração de código, este trabalho utiliza a estratégia baseada em *templates*. A escolha dessa estratégia de geração se deu pelo fato de que a implementação de AGs e EEs e dos modelos de paralelismo abordados possuem uma quantidade significativa de código-fonte exatamente igual ou com pequenas diferenças, o que favorece a utilização de *templates*.

4.1 GERAÇÃO DE CÓDIGO BASEADA EM TEMPLATES

Um *template*, no contexto de geração de código, consiste de uma estrutura pré-definida que representa um artefato de *software* semi-concluído, com pontos em aberto que podem ser preenchidos através de variáveis especificadas pelo desenvolvedor (LUCRÉDIO, 2009), ou seja, um *software* inacabado que pode ser concluído utilizando variáveis. Segundo Velocity (2013), *template* é um arquivo

contendo código em alguma linguagem de *templates* que pode ser executado através de um processador de *templates* para produzir uma saída dinâmica.

Segundo Lucrédio (2009), uma solução para geração de código baseada em *templates* é composta de pelo menos três elementos principais:

1. Um formato de entrada.
2. O *template*.
3. O processador de *templates*.

A Figura 8 ilustra o processo de geração de código. Nela é possível observar que o *template* é similar à saída desejada, acrescido de anotações e instruções que permitem ao programador adicionar conteúdo dinamicamente para que assim esse se torne um código completo e forme a saída desejada.

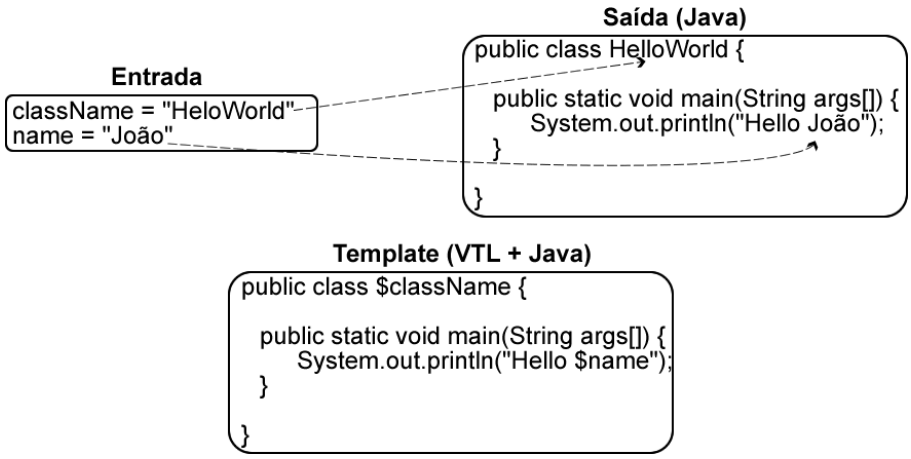

Figura 8 - Geração de código baseada em templates

Uma abordagem para geração de código baseada em *templates* funciona basicamente pela substituição de elementos presentes no *template* para gerar uma saída (MANOLESCU, 2006). Em outras palavras, um *software* substitui alguns elementos presentes no arquivo de *template* e gera o código final. Essa substituição é executada por um processador de *templates* que lê um conjunto de entradas e executa as substituições correspondentes gerando a saída.

Um *template* deve possuir construções que permitam realizar consultas em uma dada entrada. A informação resultante destas consultas é então utilizada como parâmetro para produzir código dinamicamente baseado no *template* (LUCREDIO, 2009). O processador de *templates* executa as consultas sobre os dados de entrada

e utiliza seu retorno para processar cada trecho do *template* e criar o arquivo de saída.

No exemplo mostrado na Figura 8 é ilustrada a geração de código utilizando o *framework* Apache Velocity (VELOCITY, 2013). A entrada é um objeto Java que contêm pares no formato$[chave, valor]$, o *template* é um arquivo de texto contendo código Java combinado com código VTL (*Velocity Template Language*), que é uma linguagem específica para codificação de *templates* que poderão ser processados pelo Apache Velocity. A saída resultante do processamento do *template* é um código Java puro. Percebe-se que o código Java presente no arquivo do *template* permanece inalterado após a fase de processamento sendo reproduzido no arquivo de saída.

4.1.1 Apache Velocity

O Apache Velocity é um *framework* que implementa um mecanismo de processamento de *templates*, baseado em Java, e define uma linguagem para a sua codificação (VTL – *Velocity Template Language*) (VELOCITY, 2013). Esse é desenvolvido e mantido pela *Apache Software Foundation* e licenciado sob a *Apache Software License*, que garante sua gratuidade para os usuários. O *Apache Velocity* fornece métodos que permitem definir os dados de entrada e utilizá-los para processar os *templates*, recebendo como saída código-fonte escrito em qualquer linguagem textual. Os métodos fornecidos pelo *Apache Velocity* podem ser acessados a partir de qualquer programa Java bastando para isso referenciar uma biblioteca encapsulada em um arquivo "*jar*".

Para utilizar as funcionalidades básicas fornecidas pelo *Velocity*, que permitem gerar código-fonte a partir de arquivos de *template*, levando em consideração um conjunto de entradas, é necessário seguir os seguintes passos a partir de um programa Java (VELOCITY, 2013):

1. Iniciar o Velocity;
2. Criar um objeto de contexto;
3. Adicionar dados ao contexto;

4. Escolher o *template*;
5. Mesclar o *template* e o contexto para produzir a saída;

A Figura 9 ilustra a execução dos passos supracitados, onde é possível identificar os objetos envolvidos e os métodos que são invocados durante esse processo:

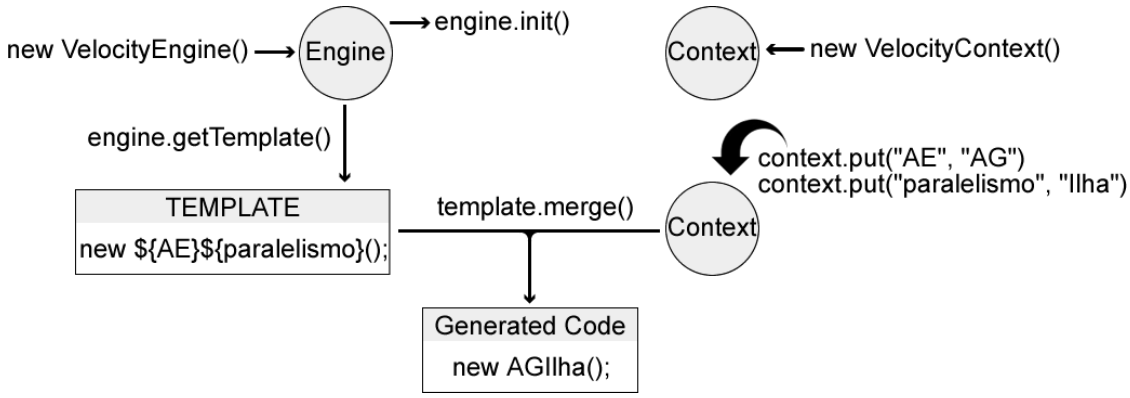

Figura 9 - Geração de código-fonte utilizando o Apache Velocity

Ainda considerando a Figura 9, os seguintes objetos podem ser identificados:

- *Engine* – Objeto da classe *Velocity Engine*. Ele representa a *engine* principal do *framework*. Utiliza-se o método *init*, para iniciá-la e posteriormente seu método *getTemplate(String)*, passando como parâmetro o caminho para o arquivo de *template*. O método *getTemplate* carrega o arquivo de *template* especificado para dentro de um objeto Java, seu retorno é um objeto da classe*Template* com o arquivo de *template* carregado.
- *Context* – Objeto que representa o contexto que será utilizado para processar o template. O conceito de objeto de contexto é central no Velocity, esse é essencialmente uma tabela *hash* que fornece métodos para definir e recuperar objetos. A ideia básica do objeto de contexto é transportar dados entre a aplicação Java e os templates (VELOCITY, 2013). O objeto de contexto é instanciado a partir da classe *VelocityContext* e seu método *put(String, Object)* é utilizado para inserir objetos de dados no contexto. O primeiro parâmetro do método *put* é um identificador que será utilizado no momento do processamento do *template* para recuperar o objeto de dados passado em seu segundo parâmetro.

- *Template* – Objeto que contém um arquivo de *template* carregado. Após o contexto estar preenchido com os dados da entrada, utiliza-se seu método *merge(Context, Writer)*, passando como parâmetro o objeto de contexto que será utilizado no processamento e um objeto da classe *Writer*. O método *merge* executa o processamento do *template* mesclando-o ao contexto que foi definindo e preenche o objeto *Writer* com o código-fonte resultante do processamento, esse objeto pode então ser utilizado, por exemplo, para escrever o código-fonte gerado em um arquivo de texto.

4.1.2 Velocity Template Language - VTL

A VTL é a linguagem de codificação de templates do Velocity, por meio de sua utilização é possível adicionar conteúdo dinamicamente aos *templates* (VELOCITY, 2013). As principais estruturas que a VTL permite utilizar são as referências e as diretivas (GRADECKI, 2003), (VELOCITY, 2013).

Os objetos inseridos no contexto do Velocity são acessíveis nos *templates* por meio das referências definidas com a VTL, o código das referências é identificado pelo caractere “$” em seu início. Os valores das referências são carregados dinamicamente no momento do processamento do *template*, a partir dos objetos dedados de entrada inseridos no contexto. O processador de *templates* do *Velocity* pesquisa no contexto por objetos que estejam associadas aos mesmos identificadores das referências e os carregam para dentro delas. Existem três tipos de referências no *Velocity*: variáveis, propriedades e métodos. As variáveis são carregadas com um valor do tipo *String* a partir de um objeto que esteja no contexto; é possível também definir seu valor diretamente no *template* com a própria VTL. As propriedades são estruturas que permitem acessar valores de atributos de objetos que estejam no contexto através de seus métodos *get,* os quais também serão convertidos para dados do tipo *String.* Os métodos permitem invocar qualquer método público dos objetos que estão no contexto, caso esses possuam algum retorno esse também será convertido para o tipo *String*.

As diretivas da VTL permitem ter controle sobre as referências e o fluxo de processamento do *template* (GRADECKI, 2003), elas dão suporte à definição de

estruturas de repetição (*foreach*), desvios condicionais (*if/else/elseif*), permitem incluir em tempo de execução outros arquivos de texto com ou sem o processamento dos mesmos (*parse* e *include* respectivamente), dentre outras possibilidades. O código referente às diretivas da VTL é identificado pelo caractere "#" em seu início.

No momento do processamento do *template* as referências da VTL são carregadas e as diretivas são executadas. Dessa forma o Velocity mescla o *template* ao contexto e gera a saída, a qual não conterá código VTL, pois em seu lugar será inserido o resultado de sua execução.

A utilização da abordagem baseada em *templates* utilizando o Velocity, neste trabalho, se mostrou eficiente e se deu pelo fato de que uma parcela significativa da implementação de AGs e EEs apresenta código-fonte exatamente igual ou com sutis diferenças entre si. O mesmo se observa ao se comparar o código de um AEP mestre-escravo, um AEP de vizinhança e um AEP ilha. A partir da identificação dessas diferenças e similaridades, se mantém o código comum inalterado nos *templates* e deixam-se pontos em aberto onde houver diferenças, para que esses pontos sejam preenchidos de acordo com o AEP que será gerado, dessa forma os mesmos *templates* podem ser utilizados para criar diferentes algoritmos. Além disso, a abordagem baseada em *templates* é simples de ser utilizada e permite alterar rapidamente qualquer parte do código gerado, apenas alterando o *template* correto e gerando o código novamente. Facilitando assim a manutenção de vários arquivos gerados a partir do mesmo *template*.

4.2 CONSIDERAÇÕES FINAIS

Este capítulo apresentou os conceitos básicos sobre a geração de código mostrando as vantagens de sua utilização. Em seguida detalhou-se a geração de código baseada em *templates*, que utiliza arquivos de texto com código-fonte incompleto que pode ser concluído dinamicamente para gerar um código completo como saída, pois essa foi a estratégia utilizada na implementação deste trabalho. Na sequência foi apresentado o *framework* Apache *Velocity*, que permite definir

templates e processá-los a partir de um programa Java. Nesse contexto, foram abordados os principais elementos do *framework* e os passos básicos que devem ser seguidos em sua utilização para gerar código-fonte em qualquer linguagem textual a partir de arquivos de *template*, levando em consideração um conjunto de entradas. Foi apresentada ainda a linguagem de definição de *templates* VTL, definida pelo próprio *framework*, abordando sua sintaxe básica e suas principais construções que permitem inserir dados dinamicamente nos *templates* e controlar o fluxo de seu processamento. Ao final do capítulo foram explicitados os motivos que levaram a utilização da estratégia de geração baseada em *templates* neste trabalho e que permitiram sua utilização de forma simples e eficiente.

5 PADRÕES DE PROJETO NOS AEPs

Independentemente do tipo de AE que se está implementando, encontrar a configuração ideal de seus parâmetros, que permita encontrar as melhores soluções para um dado problema específico, não é uma tarefa trivial. Isso, pois, tal configuração é encontrada, principalmente, com base no método de tentativa e erro testando-se várias configurações diferentes até encontrar a ideal (MARKOWSKA-KACZMAR, 2010). Tendo em vista a grande quantidade de combinações possíveis de parâmetros para um AE, encontrar a melhor configuração pode ser uma tarefa longa e cansativa. Assim, o código-fonte dos AEPs deve ser flexível, extensível e legível, permitindo que alterações sejam executadas de forma rápida e simples. A utilização de padrões de projeto em seu desenvolvimento pode garantir a presença dessas características no código-fonte. Por esses motivos optou-se por aplicá-los aos AEPs desenvolvidos no contexto deste trabalho.

Padrões de projeto, em engenharia de *software*, são soluções genéricas para problemas bem conhecidos da programação orientada a objetos. Todas essas soluções foram amplamente testadas, são comprovadamente eficazes e são consideradas elegantes pelos *designers* de *software* (GAMMA, 2008), (MARKOWSKA-KACZMAR, 2010), além disso, sua aplicação traz diversos benefícios para o código. Nesse contexto, foram aplicados dois padrões de projeto no código dos AEPs gerados neste trabalho: Strategy e Observer. Esses padrões foram utilizados por proverem características benéficas para o código dos AEPs, como a modularização, o baixo acoplamento e a notificação de alterações de estado, além disso, suas implementações são simples e assim não adicionam complexidade significativa ao código.

5.1 O PADRÃO *STRATEGY*

O padrão *Strategy* define uma família de algoritmos, encapsula-os por meio da utilização de interfaces e os torna intercambiáveis, permitindo assim variações no algoritmo independentemente dos clientes que o utilizam (FREEMAN, 2007)

(GAMMA, 2008). Com a sua aplicação, algoritmos podem ser expressos como objetos intercambiáveis e dessa forma, o algoritmo utilizado pode ser substituído por outro da mesma família apenas instanciando uma estratégia diferente (MARKOWSKA-KACZMAR, 2010). Segundo Gamma (2008), encapsular os algoritmos por meio da utilização do padrão *Strategy* diminui a complexidade do código, tornando mais fácil de compreendê-lo, alterá-lo e estendê-lo. A Figura 10 mostra o diagrama de classes básico do padrão considerado.

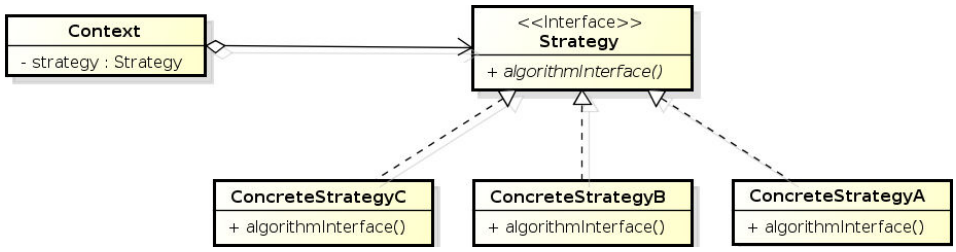

Figura 10 - Estrutura do Padrão Strategy

- *Strategy* define uma interface comum aos algoritmos de uma mesma família que será suportada.
- As classes *ConcreteStrategy* implementam o algoritmo utilizando a interface *Strategy*.
- *Context* é o cliente que utilizará os algoritmos. Ele mantém uma referência para um objeto que *Strategy*. O Algoritmo que será executado depende da *ConcreteStrategy* que for instanciada e adicionada nessa referência.

A implementação do padrão conforme a Figura 10 traz os seguintes efeitos benéficos para o código-fonte:

- O cliente (Context) conhece apenas a interface do algoritmo que será executado e assim esse pode ser substituído por outro da mesma família (ou seja, que implemente a mesma interface *Strategy*) apenas instanciando uma *ConcreteStrategy* diferente.
- Com a implementação dos algoritmos isoladas nas classes *ConcreteStrategy*, alterações nessas implementações podem ser executadas sem afetar outras partes do modelo.
- Novos algoritmos da mesma família podem ser facilmente inseridos, apenas criando novas classes que implementem a interface *Strategy* correspondente.

- As classes *ConcreteStrategy* podem ser reaproveitadas por diferentes clientes que utilizem algoritmos da mesma família.

Alternativas à aplicação do padrão Strategy incluem: (1) implementar os algoritmos diretamente no cliente, definindo qual será executado por meio de estruturas condicionais. (2) Criar uma estrutura de herança, onde se cria um cliente genérico e suas especializações se encarregam de implementar o comportamento do algoritmo que será utilizado. Essas duas soluções acarretariam em uma maior complexidade no código, pois seria necessário misturar o código dos clientes com os dos algoritmos, além disso, essa solução dificulta o reaproveitamento tanto do código do cliente quanto os dos algoritmos e a primeira ainda exige a utilização adicional de estruturas condicionais.

Segundo Gamma (2008), a aplicação do padrão *Strategy* é recomendável quando:

- Diversas classes relacionadas diferem apenas em seu comportamento;
- Um algoritmo usa dados dos quais o cliente não deve ter conhecimento;
- Um cliente agrega diversos comportamentos diferentes;

Levando em consideração as recomendações supracitadas, percebe-se a aplicabilidade do padrão Strategy nos AEs, pois várias classes relacionadas tem a mesma função, mas diferem na maneira de executar essa função, como por exemplo, várias classes diferentes que implementam cada uma um método de cruzamento diferente. Além disso, o AE necessita agregar uma série de comportamentos diferentes para concluir o processo evolutivo.

5.1.1 *Strategy* nos AEPs

Com a utilização do padrão *Strategy* na implementação dos AEs reduz-se o esforço necessário para mudar as operações do algoritmo. As interfaces utilizadas são simples e pode-se facilmente adicionar novas estratégias apenas criando classes que as implementem. Outro efeito dessa aplicação é a diminuição da

complexidade do código do AE e o aumento de sua flexibilidade (MARKOWSKA-KACZMAR, 2010).

No contexto dos AEPs, o padrão Strategy pode ser utilizado com eficiência, por exemplo, para prover modularização e esconder a implementação dos operadores genéticos e assim garantir que esses possam ser alterados sem afetar outras partes do código do AE (FENG, 2003), (MARKOWSKA-KACZMAR, 2010). Essa característica permite que se alterne o comportamento de um operador (por exemplo, mudar de seleção por roleta para seleção por torneio) apenas alterando a estratégia que foi instanciada. Além disso, esse padrão permite que as mesmas implementações de operadores genéticos sejam reaproveitadas por diferentes AEPs. Esses mesmos efeitos benéficos são também visíveis ao aplicar o padrão *Strategy* nos seguintes aspectos dos AEPs:

- Na implementação dos critérios de parada dos AEs.
- Na implementação das políticas de seleção de indivíduos migrantes (nos AEPs do tipo ilha).
- Na implementação das topologias de migração (nos AEPs do tipo ilha).
- Na implementação dos formatos de vizinhança (nos AEPs de vizinhança).

A Figura 11 mostra o diagrama de classes resultante da aplicação do padrão Strategy em um AE para encapsular os métodos de cruzamento utilizados.

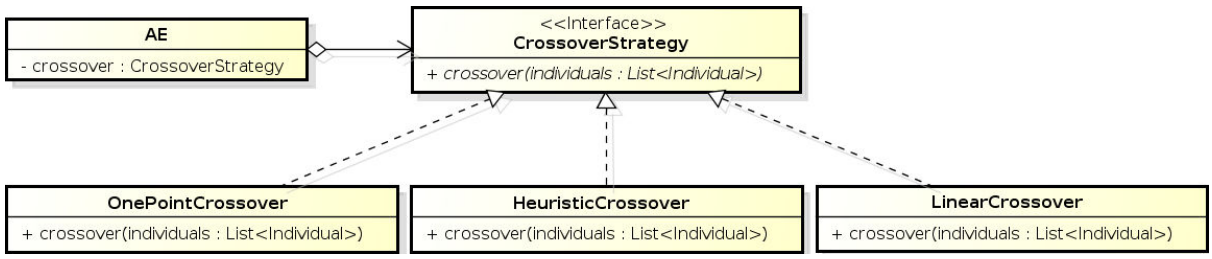

Figura 11 - Utilização do padrão *Strategy* para encapsular o operador de cruzamento

Todos os operadores genéticos, critérios de parada, políticas de seleção de indivíduos migrantes, topologias de migração e formatos de vizinhança, foram implementados na forma de estratégias. Permitindo assim que esses comportamentos sejam facilmente alternados, que novos comportamentos sejam facilmente adicionados e que os comportamentos sejam reutilizados.

5.2 O PADRÃO *OBSERVER*

O padrão *Observer* define uma relação de dependência de um para muitos entre objetos, com baixo acoplamento, de maneira que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente (FENG, 2003), (GAMMA, 2008). A relação entre objetos criada pela aplicação do padrão *Observer* consiste em um objeto observado por vários outros objetos observadores. Quando um objeto observado sofre uma alteração em seu estado, todos os seus observadores são notificados e seus estados são atualizados automaticamente. Com a utilização do padrão *Observer*, um conjunto de objetos cooperantes podem manter a consistência e a coerência entre si (GAMMA, 2008). A Figura 12 mostra o diagrama de classes básico do padrão considerado.

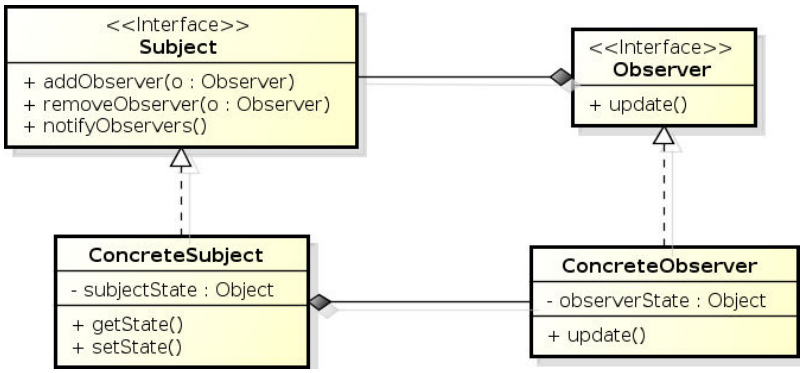

Figura 12 - Estrutura básica do padrão *Observer*

- *Subject* é a interface que define quais objetos podem ser observados por outros. Ela fornece uma interface para o provimento de métodos que permitem adicionar ou remover observadores e para enviar notificações de modificação para eles.
- *Observer* é a interface que define quais objetos terão a capacidade de observar o *Subject*. Ela fornece uma interface para o provimento de métodos que executam a atualização do objeto *Observer* a respeito de modificações em um *Subject*. Caso necessário um objeto *Observer* poderá consultar o *Subject* para obter informações essenciais para executar sua atualização.

O relacionamento entre *Subject* e *Observer* é baseado em que um conheça apenas a interface do outro e não suas implementações concretas. Dessa forma, o

acoplamento entre o *Subject* e os *Observers* é abstrato e mínimo (GAMMA, 2008). Segundo Gamma (2008) o padrão *Observer* é aplicável quando:

- Uma abstração tem dois aspectos dependentes entre si.
- Quando uma alteração no estado de um objeto exige mudanças em outros para manter a consistência do algoritmo e não se sabe, necessariamente, quantos objetos devem ser atualizados.
- Quando um objeto deva ser capaz de enviar notificações a outros sem, necessariamente, conhecer quem são esses objetos, ou seja, sem eles serem fortemente acoplados.

5.2.1 *Observer* nos AEPs

Neste trabalho, o padrão *Observer* foi utilizado para fins de implementar mecanismos que mantenham a sincronização entre vários objetos do AEP que executam tarefas paralelamente em *threads* diferentes. Nesse contexto, os objetos observadores são os sincronizadores e os objetos que serão observados por eles dependem do modelo de paralelismo considerado, da mesma forma, os eventos que disparam o mecanismo de atualização automática do padrão *Observer* diferem de um modelo de paralelismo para o outro.

Independentemente do modelo de paralelismo considerado, o mecanismo de atualização automática, implementado no objeto *Observer*, consiste basicamente em atualizar seu estado interno e a partir desse seu novo estado controlar a execução dos objetos que ele está observando, definindo se esses podem continuar seu processamento ou se devem permanecer em estado de espera até que um determinado evento ocorra. Dessa forma o *Observer* mantém a sincronização das tarefas paralelas do AEP. Os critérios que definem se um objeto vai continuar sua execução ou vai permanecer em estado de espera também são dependentes do modelo de paralelismo considerado.

Nesse contexto, pode-se observar que o padrão *Observer* oferece uma maneira simples, elegante e funcional de manter a sincronização entre as diferentes tarefas do AEP, garantindo assim a consistência da execução do algoritmo, com

baixo acoplamento entre os objetos observadores e os objetos observados. Maiores detalhes a respeito da sincronização e da forma de aplicação do padrão *Observer* para esse fim são apresentados na Seção 6.3.

5.3 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados os padrões de projeto e como estes podem ser utilizados para aumentar a qualidade do código-fonte de aplicações em geral e especificamente dos AEPs. Inicialmente foi feita uma introdução sobre os padrões de projeto e os objetivos gerais de sua aplicação, em seguida detalharam-se os padrões de projeto: *Strategy* e *Observer* mostrando suas características, diagramas de classe e situações em que eles são aplicáveis conforme pode ser visto na Tabela 3. Nesse contexto, foi explicado como esses padrões foram utilizados no desenvolvimento dos AEPs envolvidos neste trabalho e como sua aplicação contribuiu para tornar os códigos-fonte dos AEPs desenvolvidos mais legíveis, flexíveis e extensíveis.

Tabela 3 - Características resumidas dos padrões Strategy e Observer

	Características	**Aplicável quando**
Strategy	• Provê encapsulamento e modularização para algoritmos. • Torna algoritmos de uma mesma família intercambiáveis. • Facilita a adição, substituição, manutenção e reaproveitamento de algoritmos.	• Diversas classes relacionadas diferem apenas em seu comportamento. • Um algoritmo usa dados dos quais o cliente não deve ter conhecimento. • Um cliente agrega diversos comportamentos diferentes.
Observer	• Quando um objeto muda de estado, os objetos interessados são notificados e atualizados automaticamente. • Permite que um conjunto de objetos cooperantes	• Uma abstração tem dois aspectos dependentes entre si. • Quando uma alteração no estado de um objeto exige mudanças em outros para manter a consistência do algoritmo e não se sabe,

	mantenham a consistência e a coerência entre si.	necessariamente, quantos objetos devem ser atualizados. • Quando um objeto deva ser capaz de enviar notificações a outros sem, necessariamente, conhecer quem são esses objetos, ou seja, sem eles serem fortemente acoplados.

6 JPEAG: JAVA PARALLEL EVOLUTIONARY ALGORITHMS GENERATOR

Embora a ideia por trás dos AEPs seja simples, sua implementação não é uma tarefa trivial, pois ela envolve desafios provenientes do modelo de programação paralela, tais como problemas de comunicação/sincronização, a utilização adequada dos modelos/recursos paralelos e um processo de depuração diferente. Dessa forma, a falta de experiência com a computação paralela pode reduzir significativamente a produtividade no desenvolvimento de código de AEPs. Além disso, a programação paralela pode exigir o aprendizado de novas APIs, *frameworks*, novas bibliotecas ou até mesmo uma nova linguagem.

Como alternativa para melhorar a produtividade no desenvolvimento de AEPs, podem ser utilizadas ferramentas de geração de código paralelo de propósito geral, no entanto, ainda assim, o código específico da implementação do AE tem de ser escrito manualmente. Neste contexto, uma ferramenta específica para a criação automática de código de AEs paralelos, chamada JPEAG (*Java Parallel Evolutionary Algorithm Generator*), foi desenvolvida. Atualmente, dois algoritmos evolutivos estão disponíveis, AGs e EEs, ambos utilizando representação dos indivíduos em código real. Além disso, três modelos de paralelismo estão disponíveis: mestre-escravo, vizinhança e ilha.

O JPEAG é uma aplicação *web*, que foi desenvolvida em Java seguindo o padrão MVC (*Model View Controller*). Ela provê uma interface gráfica que permite configurar os parâmetros dos AEPs e gerar seu código-fonte, também escrito em Java. A maior parte do código do AEP é criado com esta aplicação, a exceção é o código referente à função de avaliação do AE, podendo esse ser inserido através interface gráfica da aplicação. O JPEAG insere o código referente à função de avaliação no local correto dentro do código gerado, produzindo assim um código completo, pronto para ser compilado e executado. O JPEAG foi desenvolvido obedecendo aos seguintes requisitos:

- Orientada a objetos - hoje em dia os conceitos de orientação a objetos são importantes, especialmente quando se planeja aplicar padrões de projeto no código.

- Amigável para o usuário - uma aplicação amigável para o usuário é fácil de se aprender e utilizar. Dessa forma pode-se aumentar ainda mais a produtividade.
- Ocultar a complexidade inerente ao modelo de programação paralela - usuários sem experiência com o desenvolvimento de programas paralelos serão capazes criar AEPs rapidamente, pois a aplicação esconde os aspectos da implementação envolvidos pelos modelos paralelos.
- Flexibilidade do código gerado - o JPEAG gera código-fonte Java salvo em arquivos de texto puro que os usuários podem facilmente modificar, adaptar ou estender.
- Utilização de padrões de projeto - aplicando padrões de projeto garantimos que estão sendo utilizadas soluções testadas, eficientes e largamente aceitas. Além disso, sua aplicação aumenta a qualidade do código gerado tornando-o mais flexível, extensível e legível.

6.1 ARQUITETURA DO JPEAG

A arquitetura do JPEAG é mostrada na Figura 13. Seu funcionamento ocorre da seguinte forma: o usuário acessa o JPEAG utilizando um *web browser* e através da sua interface gráfica configura as características do AEP que deseja gerar, incluindo a função de avaliação. Após terem sido definidos todos os parâmetros, ele submete essas informações para o núcleo do JPEAG. O núcleo é responsável por selecionar os *templates* apropriados na base de dados de templates, de acordo com as configurações inseridas pelo usuário, e utilizá-los para criar o código do AEP. A seguir, o código gerado é empacotado em um arquivo “zip” e enviado ao usuário por meio de *download*, visto o JPEAG ser uma aplicação *web*.

A Figura 14 mostra o relacionamento entre as principais classes que compõem o JPEAG. A classe *ConfiguracaoAE* contém um conjunto de atributos que são utilizados para armazenar a configuração do AEP que for inserida pelo usuário na interface gráfica. O processo de carregar as dados inseridos na interface gráfica em um objeto da classe *ConfiguracaoAE* é executado automaticamente pelo *framework* ZK (ZKOSS, 2013) através de requisições AJAX.

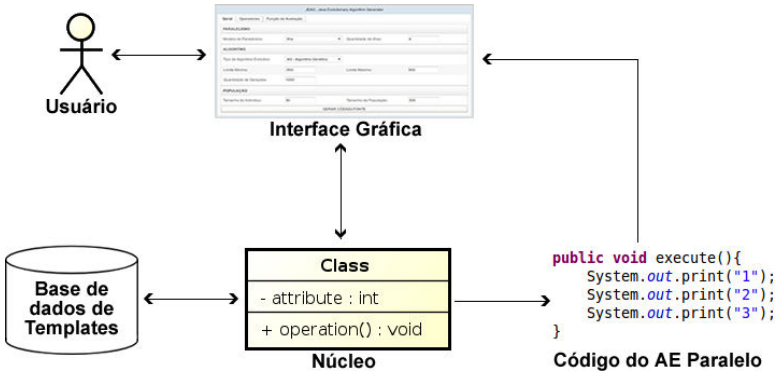

Figura 13 - Arquitetura do JPEAG

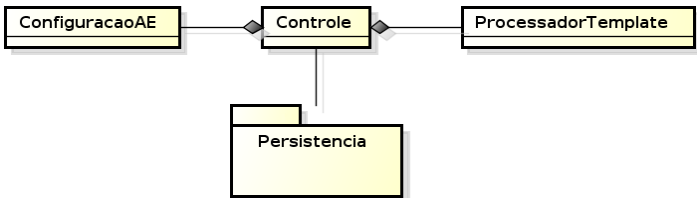

Figura 14 - Principais classes do JPEAG

A classe *ProcessadorTemplate* é responsável por processar os *templates* para criar o código-fonte paralelo, para tal ela utiliza o *framework* Apache *Velocity*, apresentado na Seção 4.1.1. Essa classe inicia a *engine* do *Velocity* em seu construtor e possui o método *processar(ConfiguracaoAE)* o qual recebe as configurações do AEP inseridas pelo usuário na interface gráfica. Em resumo, os passos executados pelo método *processar* são:

1. Instanciar um objeto de contexto do *Velocity* e inserir nele as informações de configuração contidas no objeto *ConfiguracaoAE* que foi recebido como parâmetro.
2. Criar os diretórios que formam a estrutura de pacotes das classes que serão geradas para o AEP que foi configurado.
3. Selecionar os *templates* apropriados para gerar o AEP configurado e utilizar os métodos do *Velocity* para carregá-los e processá-los, mesclando-os ao contexto, e recebendo como resultado o código-fonte do AEP.
4. Salvar as classes geradas em arquivos de texto, mantendo a estrutura de pacotes correta para o AEP, e empacotá-las em um arquivo zip.

A classe *Controlador* é responsável por implementar a camada de controle do JPEAG. Ela recebe todos os eventos gerados na camada de visualização e envia as respostas correspondentes de volta para essa camada. Essa classe mantém uma instancia da classe *ConfiguracaoAE* e da classe *ProcessadorTemplate* e, as utiliza para solicitar a geração do código-fonte e receber seu resultado, enviando-o em seguida para o usuário. Além disso, a classe *Controlador* se comunica com a camada de persistência para carregar a partir do SGBD os dados necessários para a execução do JPEAG, como por exemplo, os tipos de AE, os modelos de paralelismo e os operadores genéticos disponíveis.

As demais classes da implementação do JPEAG estão no pacote *persistencia* e implementam a camada de persistência da aplicação usando a *Java Persistence API* (JPA) e o *framework Hibernate*. Basicamente sua função é prover uma camada de persistência que salva e recupera informações sobre modelos de paralelismo, modelos de AE e operadores genéticos. Atualmente, toda a persistência é feita em uma base de dados relacional utilizando o SGBD PostgreSQL.

6.2 TEMPLATES DO JPEAG

Por utilizar o *framework* Apache *Velocity* para processar os *templates* e exportar código-fonte Java, os *templates* utilizados pelo JPEAG são arquivos de texto contendo código escrito em Java combinado com código escrito em VTL, o qual indica os pontos em aberto nos *templates* que serão preenchidos dinamicamente pelo JPEAG de acordo com os parâmetros definidos pelo usuário através da interface gráfica. Todo o código escrito com VTL é substituído quando o *template* é processado pelo núcleo do JPEAG e a saída é código Java puro. A Figura 15 mostra um trecho pertencente a um dos *templates* utilizados pelo JPEAG, no qual é possível identificar tanto código Java quanto código VTL.

As diretivas VTL são iniciadas pelo caractere "#" e são executadas no momento em que o *template* é processado, as referências são iniciadas com o caractere “$” e são carregadas, a partir do contexto, também no momento do processamento. Tomando como exemplo o trecho de código presente na Figura 15,

a diretiva "*#if*", está definindo que aquele bloco de código será processado somente se o usuário houver escolhido o modelo de paralelismo de ilha.

```
#if ($modeloParalelismo == "Ilha")
      TopologiaStrategy topologia = new $topologia();
      ${modeloAE}${modeloParalelismo} ae = new
      ${modeloAE}${modeloParalelismo}($qtdIlhas,
      $tamanhoPopulacao, $taxaMigracao,
      $frequenciaMigracao, topologia);
#end
ae.executar();
```

Figura 15 - Trecho de um *template* utilizado no JPEAG

Como resultado do processamento do trecho de código mostrado na Figura 15, o usuário recebe o código Java mostrado na Figura 16, no qual exibe-se, por exemplo, caso o usuário tenha selecionado gerar um AGP no modelo ilha, onde os parâmetros são a quantidade de ilhas (4), o tamanho da população (200 indivíduos), a taxa de migração (5 indivíduos por migração), a frequência de migração (a cada 100 gerações) e a topologia de migração (anel unidirecional).

```
TopologiaStrategy topologia = new AnelUnidirecional();
AGIlha ae = new AGIlha(4, 200, 5, 100, topologia);
ae.executar();
```

Figura 16 - Código-fonte gerado a partir do processamento de um *template* do JPEAG

A partir do carrregamento de dados do contexto para dentro do *template*, através das referências, e da execução das diretivas, o trecho de *template* mostrado na Figura 15 pode produzir como saída código Java pertencente à implementação tanto de AGs quanto de EEs, no modelo de paralelismo de ilha, com quaisquer configurações de quantidade de ilhas, tamanho da população, taxa de migração, frequência de migração e topologia de migração.

6.3 PARALELISMO E SINCRONIZAÇÃO NOS AEPs

Atualmente, o JPEAG é capaz de criar AEPs nos modelos mestre-escravo (com paralelização da função de avaliação), vizinhança (com alocação das células em grade) e ilha. Além disso, o paralelismo é baseado em *threads*, sem execução remota. No modelo mestre-escravo a avaliação dos indivíduos é executada em várias *threads* diferentes. No modelo de vizinhança cada célula é executada em uma *thread* diferente. Por fim, no modelo ilha, cada população evolui em uma *thread* diferente. Todos os modelos implementados são síncronos e a sincronização é implementada com a utilização de monitores e barreiras. Todo o processo de envio de notificações entre as *threads* e os objetos que cuidam as sincronização e as atualizações de estado das *threads* foram implementados seguindo o padrão de projeto *Observer*.

No que diz respeito à implementação gerada para o modelo mestre-escravo, existe uma *thread* principal (mestre) onde toda a população é mantida e na qual são executados os operadores genéticos. A execução da função de avaliação é delegada para *threads* secundárias (escravos), as quais executam como *daemons* e ficam à espera de indivíduos para avaliar. Quando a *thread* mestre necessita processar a fase de avaliação do AE, a população é dividida em segmentos e cada um é enviado para um escravo diferente, após efetuar o envio, a *thread* mestre entra em estado de espera e só volta a executar após todos os escravos completarem as avaliações. Assim que cada escravo termina sua tarefa, ele envia uma notificação para o objeto sincronizador, quando o sincronizador recebe essa notificação por parte de todos os escravos, ele notifica a *thread* mestre para que ela continue sua execução.

Quanto à implementação gerada para o modelo de vizinhança, cada célula executa um $(1+1)-AE$ em uma *thread* separada, no qual permanecem na população apenas um indivíduo e seu descendente. Cada célula tem acesso apenas ao seu indivíduo nativo e aos indivíduos das células vizinhas para executar os operadores genéticos. Antes de iniciar cada nova geração, cada célula envia uma notificação ao objeto sincronizador e entra em estado de espera. Quando o sincronizador é notificado por todas as células, ele envia uma notificação em

broadcast para que elas continuem sua execução. A implementação gerada para o modelo de vizinhança multi-indivíduos é similar ao modelo de vizinhança, a diferença básica é que ele mantém n indivíduos em cada célula ao invés de apenas 1 como no modelo de vizinhança.

Em relação à implementação gerada para o modelo ilha, cada ilha executa um AE completo e independente em uma *thread* separada e as ilhas trocam indivíduos entre si durante a fase de migração, respeitando a taxa e a frequência de migração definidas pelo usuário. Ao executar a fase de migração, são selecionados n indivíduos (sendo n a taxa de migração) de acordo com a política de seleção de migrantes selecionada, e esses são enviados de cada ilha para as suas vizinhas, respeitando a topologia de migração que foi definida. Os piores indivíduos da população são então substituídos pelos indivíduos recebidos das outras ilhas. Ao completar o processo de migração, cada ilha notifica o objeto sincronizador e entra em estado de espera. Quando o objeto sincronizador é notificado que todas as ilhas concluíram suas migrações, ele envia uma notificação em *broadcast* informando a todas as ilhas que elas podem continuar sua execução.

6.3.1 A CLASSE SINCRONIZADOR

Independente do modelo de paralelismo considerado, todas as *threads* compartilham uma única instancia da classe *Sincronizador* que é a responsável por implementar o mecanismo de sincronização utilizando monitores e barreiras. A Tabela 4 mostra seus atributos com suas respectivas descrições, para cada modelo de paralelismo gerado.

Tabela 4 - Atributos da classe Sincronizador

Mestre-escravo	Vizinhança/Vizinhança multi-indivíduos	Ilha	Descrição
qtdEscravos	qtdCelulas	qtdIlhas	Define a quantidade de threads que devem atingir a barreira para que essa possa ser liberada.
qtdAvaliaram	qtdCptGeracao	qtdMigraram	Armazena a quantidade

			de *threads* que concluíram a tarefa anterior à barreira. No caso do modelo mestre-escravo, a avaliação dos indivíduos, no modelo de vizinhança, o fim de uma geração e no modelo ilha, a migração.

O principal método da classe *Sincronizador* é o método *sincronizar*, ele é o responsável por controlar a barreira, esse método é invocado automaticamente por conta da implementação do padrão *Observer* e possui acesso mutuamente exclusivo, controlado por um monitor. Seu funcionamento consiste em incrementar a quantidade de *threads* que concluíram suas tarefas anteriores e que estão no ponto de sincronização, ou seja, que atingiram a barreira, e a partir daí executar uma das seguintes ações, de acordo com a quantidade de *threads* que atingiram a barreira até o momento em que ele é executado:

- Caso essa quantidade seja menor que o total de *threads* que o AEP executa: Nos modelos ilha, vizinhança e vizinhança multi-indivíduos, a *thread* corrente é colocada em modo de espera. No modelo mestre-escravo, apenas não é executada ação nenhuma, nesse modelo a *thread* corrente não é colocada em estado de espera, pois os avaliadores executam como *deamons*.
- Caso essa quantidade seja igual ao total de *threads* que o AEP executa. Todas as *threads* são liberadas para continuarem seu processamento, no modelo ilha são todas as ilhas, no modelo vizinhança são todas as células e no modelo mestre-escravo é o mestre. Logo após isso a barreira é reiniciada.

A implementação da classe Sincronizador possui a mesma estrutura em todos os modelos de paralelismo desenvolvidos, na verdade, a única diferença entre as implementações de cada modelo são os nomes dos identificadores utilizados.

6.4 O CÓDIGO-FONTE GERADO

Cada um dos modelos de paralelismo considerados neste trabalho reflete um diagrama de classes diferente ao ser gerado. A seguir são apresentados esses diferentes diagramas. É importante observar que os diagramas apresentados correspondem ao código-fonte gerado para algoritmos genéticos, dessa forma, o identificador de algumas classes é iniciado pelo prefixo “AG”, caso fossem gerados códigos para estratégias evolutivas, essas mesmas classes teriam seus identificadores iniciados pelo prefixo “EE”.

A Figura 17 apresenta o diagrama de classe contendo as principais classes geradas pelo JPEAG para um AEP mestre-escravo.

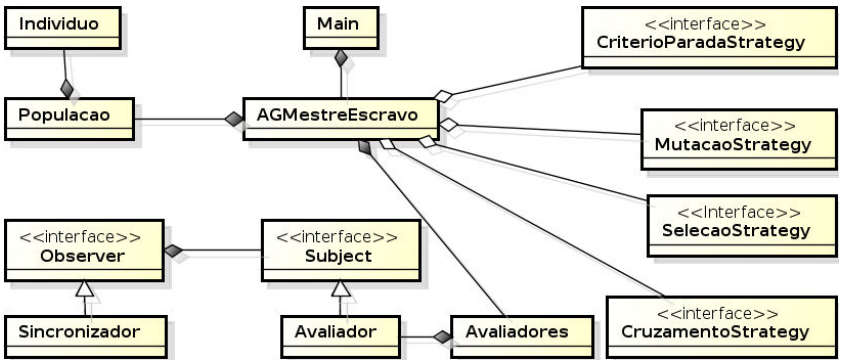

Figura 17 - Principais classes geradas para um AEP Mestre-Escravo

A tabela 5 descreve as principais classes contidas no diagrama mostrado na Figura 17 que são geradas exclusivamente para o modelo mestre-escravo: *AGMestreEscravo* e *Avaliadores*.

Tabela 5 - Descrição das principais classes geradas para um AEP mestre-escravo

Classe	Descrição
AGMestreEscravo	Implementa o núcleo do AE e representa o mestre. Essa classe mantém um conjunto de objetos que implementam os comportamentos que o AE utiliza para concluir o processo evolutivo, no formato estabelecido pelo padrão *Strategy*, e um objeto da classe *Avaliadores*, utilizado para fins de avaliação dos indivíduos. Além disso, essa classe mantém também a população que o AE manipula.
Avaliadores	Mantém uma estrutura de dados no formato de lista que mantém os avaliadores disponíveis. O mestre envia os

	indivíduos para um objeto da classe *Avaliadores* e esse por sua vez divide a população em partes e envia para os avaliadores. Além disso, essa classe é responsável por iniciar os avaliadores.

O diagrama de classe correspondente ao código-fonte gerado para os modelos de vizinhança e vizinhança multi-indivíduos é apresentado na Figura 18.

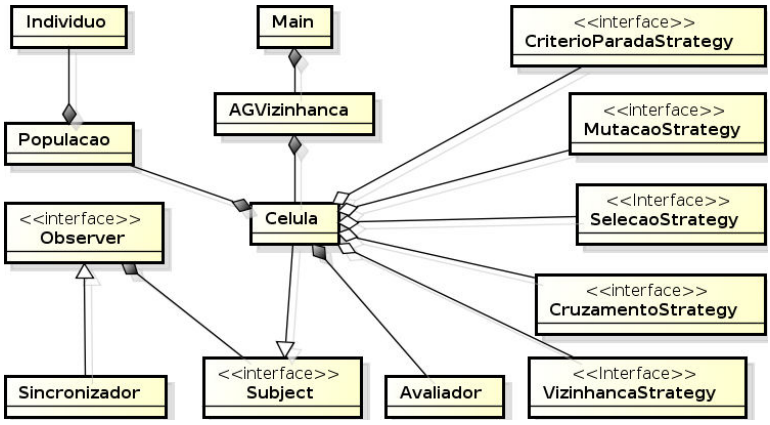

Figura 18 - Principais classes geradas para um AEP de vizinhança / Vizinhança multi-indivíduos

A tabela 6 descreve as principais classes contidas no diagrama mostrado na Figura 18 que são geradas exclusivamente para o modelo de vizinhança: *AGVizinhanca* e *Celula*.

Tabela 6 - Principais classes geradas para os AEPs de vizinhança

Classe	Descrição
AGVizinhanca	Mantém uma estrutura de dados no formato de matriz bidimensional, que representa uma grade, na qual cada nó é uma célula que executa um AE em uma *thread* diferente. Essa classe é responsável por criar as células, colocá-las na grade e iniciar suas execuções.
Celula	Mantém um conjunto de objetos que implementam os comportamentos que o AE utiliza para concluir o processo evolutivo, no formato estabelecido pelo padrão *Strategy*, e um objeto da classe *Avaliador*. Os operadores genéticos englobam apenas o indivíduo nativo da célula e os indivíduos das células vizinhas. Essa classe implementa a interface *java.util.concurrent.Callable*, permitindo assim que cada célula execute em uma *thread* diferente. A classe *Celula* implementa ainda a interface *Subject*, utilizada pelo padrão *Observer*, que permite que toda célula seja observada durante sua execução.

O diagrama de classe correspondente ao código-fonte gerado para o modelo de paralelismo de ilha é apresentado na figura 19.

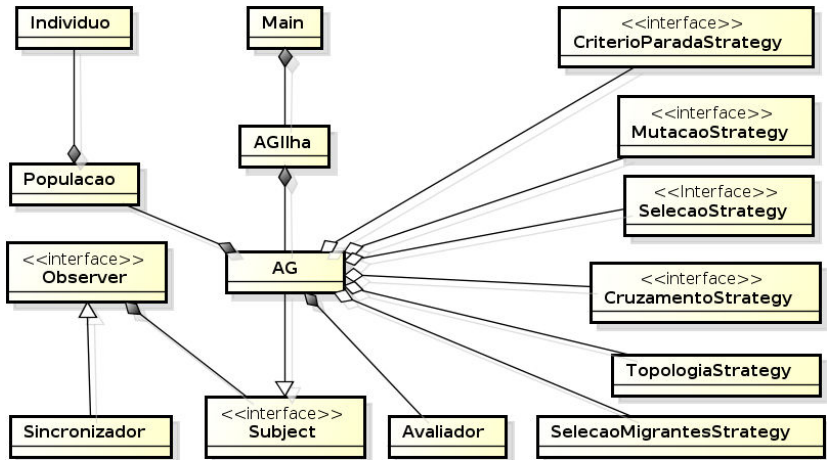

Figura 19 - Principais classes geradas para um AEP Ilha

A tabela 7 descreve as principais classes contidas no diagrama mostrado na Figura 19, que são geradas exclusivamente para o modelo ilha: AGIlha *e* AG.

Tabela 7 - Descrição das principais classes geradas para um AEP Ilha

Classe	Descrição
AGIlha	Mantém uma estrutura de dados similar a um grafo, na qual cada nó é uma ilha que executa um AE completo e independente. As ligações entre os nós definem como as ilhas irão se comunicar, ou seja, os caminhos por onde serão enviados os indivíduos migrantes. Essa classe inicia todas as ilhas, define suas ligações e inicia a execução de cada uma.
AG	Essa classe implementa o núcleo do AE e cada um de seus objetos representa um ilha. Essa classe mantém um conjunto de objetos que implementam os comportamentos que o AE utiliza para concluir o processo evolutivo, no formato estabelecido pelo padrão *Strategy*, e um objeto da classe *Avaliador*. Além disso, essa classe mantém a população que o AE manipula e implementa o mecanismo de migração. Essa classe implementa a interface *java.util.concurrent.Callable*, permitindo assim que cada ilha execute em uma *thread* diferente. Essa classe implementa ainda a interface *Subject*, utilizada pelo padrão *Observer*, que permite que toda ilha seja observada durante sua execução.

As classes descritas na tabela 8 são comuns a todos os modelos de paralelismo gerados, possuindo a mesma implementação em todos os modelos, ou implementações contendo pequenas variações entre um modelo e outro.

Tabela 8 - Descrição das classes comuns a todos os modelos de paralelismo gerados

Classe	Descrição
Main	Possui o método principal da aplicação (*main*). Sua função é instanciar o AEP e iniciar sua execução. A diferença em sua implementação de um modelo de paralelismo para o outro está apenas na classe que é instanciada internamente, a qual deve ser adequada a cada modelo.
Individuo	Implementa o indivíduo e possui a mesma implementação para todos os modelos de paralelismo suportados.
Populacao	Mantém uma estrutura de dados no formato de lista que contém os indivíduos manipulados pelo AE e fornece métodos que provêem acesso a esses indivíduos.
Avaliador	Implementa os métodos que utilizam a função de avaliação definida pelo usuário. Para os modelos ilha e célula essa classe possui a mesma implementação. Para o modelo mestre-escravo ela implementa a interface *java.lang.Runnable*, o que permite que cada avaliador, execute em uma *thread* diferente. Nesse modelo, essa classe implementa ainda a interface *Subject*, utilizada pelo padrão *Observer*, que permite que todo avaliador seja observado durante sua execução.
Sincronizador	Sua função é manter a sincronização entre as tarefas paralelas, utilizando monitores e barreiras. No modelo ilha, um objeto dessa classe coordena o processo de migração, no modelo mestre-escravo, o processo de avaliação e no modelo vizinhança / vizinhança multi-indivíduos, o início de novas gerações. Essa classe implementa a interface *Observer*, utilizada pelo padrão *Observer*, permitindo que ela observe outros objetos e seja notificada acerca de modificações em seus estados.

6.5 FUNCIONALIDADES DO JPEAG

O JPEAG foi concebido para ser uma aplicação simples de ser utilizada. Para utilizá-lo com destreza é necessário apenas conhecimentos básicos sobre a teoria dos AEs e AEPs, de modo a ser coerente com os valores que se está inserindo no formulário de descrição do AEP. É necessário ainda ter conhecimento sobre a linguagem Java o suficiente para escrever a função de avaliação, respeitando a sua assinatura que é definida pelo JPEAG.

Por ser uma aplicação *web*, seu acesso é por meio de um *web browser*. Ao acessar o JPEAG, o usuário visualiza um formulário que permite configurar o AEP que será gerado. As características dos AEPs, que podem ser definidas pelo usuário utilizando a interface gráfica do JPEAG, até o presente momento, são apresentadas na Tabela 9.

Tabela 9 - Parâmetros do AEP que podem ser configuradas no JPEAG

Parâmetro	**Valor de Preenchimento**
Modelo de Paralelismo	Mestre-Escravo Vizinhança Vizinhança multi-indivíduos Ilha
Quantidade de Tarefas Paralelas	Quantidade de ilhas, células ou escravos (de acordo com o modelo de paralelismo selecionado)
Domínio do problema	Intervalo de valores numéricos
Modelo de AE	Algoritmo Genético Estratégias Evolutivas
Tipo de Problema	Maximização Minimização
Tamanho do Indivíduo	Valor numérico
Tamanho da população (exceto para o AEP de vizinhança que utiliza um indivíduo por célula)	Valor numérico
Utilizar Elitismo / Tamanho da elite (Apenas para AGs)	Verdadeiro ou falso / Quantidade de indivíduos que comporão a elite.
Valor de λ (Apenas para EEs)	Valor Numérico
Operadores	Estratégia de Seleção, Cruzamento e Mutação
Critério de parada	Quantidade de gerações
Função de Avaliação	Código-fonte da função de avaliação (Escrito em Java)
Parâmetros exclusivos do modelo ilha	
Taxa de Migração	Valor numérico
Frequência de migração	Valor numérico
Topologia de Migração	Anel unidirecional Anel bidirecional Malha
Política de seleção de migrantes	Melhores indivíduos Aleatória
Parâmetros exclusivos do modelo de vizinhança/vizinhança milti-indivíduos	

Parâmetro	Valor de Preenchimento
Formato da vizinhança	Cruz
Tamanho da vizinhança	Valor numérico

A Figura 20 mostra a interface gráfica principal do JPEAG. Através da qual é possível configurar os parâmetros supracitados.

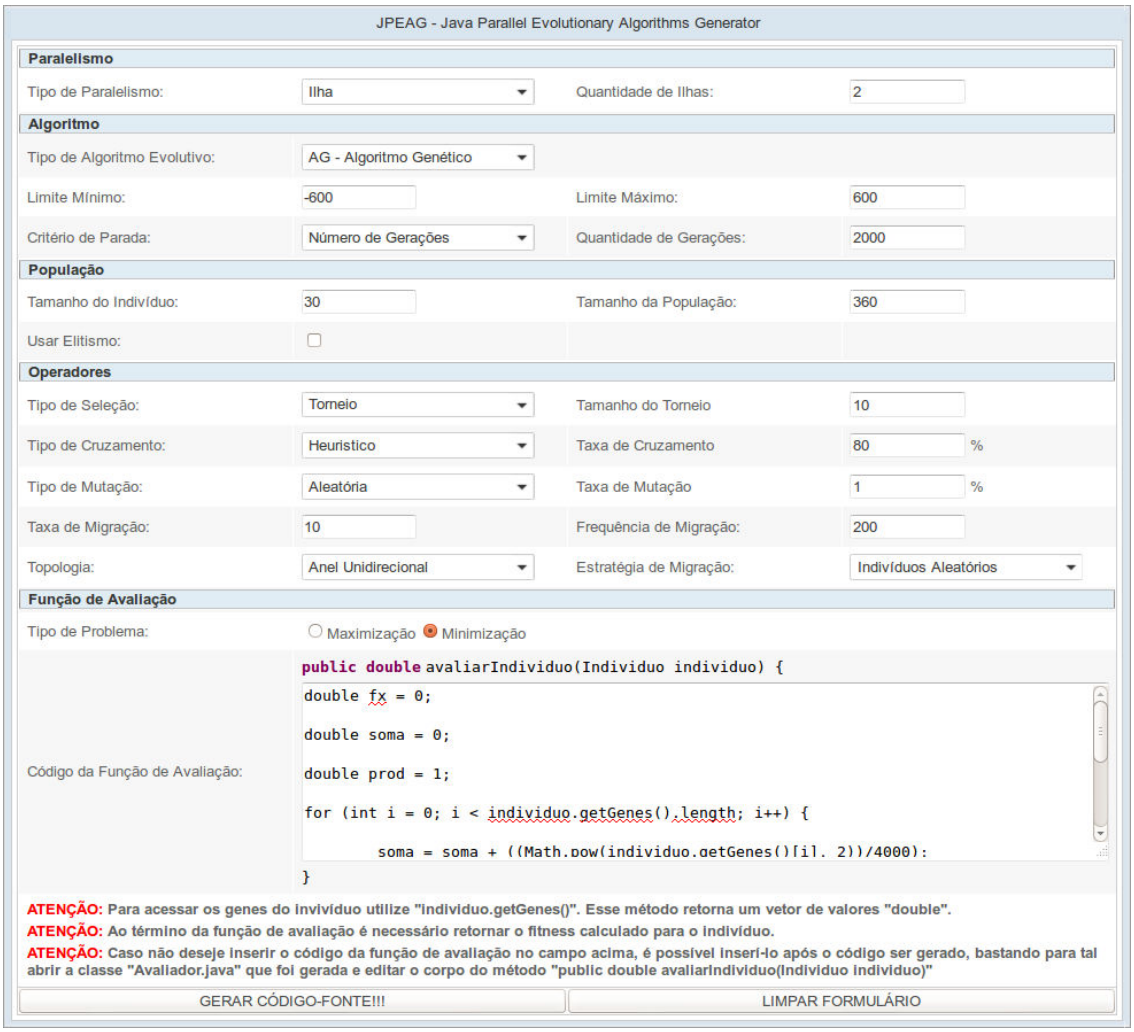

Figura 20 - Tela principal do JPEAG

A Tabela 10 mostra os operadores genéticos disponíveis para cada modelo de AE, até o presente momento.

Tabela 10 - Operadores genéticos disponíveis no JPEAG

AE	Operadores Genéticos	Métodos
Algoritmo Genético	Seleção	Roleta Torneio
	Cruzamento	Um ponto Heurístico

		Linear
	Mutação	Uniforme ou Aleatória Creep
Estratégias Evolutivas	Seleção	ES-(μ,λ) ES-(μ+λ)
	Cruzamento	Intermediário Uniforme
	Mutação	Gaussiana Cauchy

Após preencher o formulário, o usuário clica no botão “Gerar Código-Fonte” e nesse momento o JPEAG gera um conjunto de classes Java contendo o código-fonte do AEP configurado. Essas classes são então empacotadas em um arquivo no formato “*zip*” e enviadas ao usuário por meio de *download*.

6.6 EXPERIMENTOS

A fim de demonstrar o ganho de velocidade atingido com a utilização do paralelismo nos AEPs gerados pelo JPEAG, foram realizados experimentos baseados na otimização da função de Griewank, mostrada na Equação 12, onde $x_i \in [-600, 600]$ e n é igual a 30, representando um indivíduo com 30 genes com valores reais. Essa função foi introduzida em 1981 sendo bem conhecida na literatura (LOCATELI, 2003). A solução ótima para a função considerada é 0, ou seja, o melhor indivíduo para solucionar o problema de otimização dessa função tem sua avaliação igual a 0.

$$\min f(x) = \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \qquad (12)$$

Os experimentos foram conduzidos utilizando um processador Intel Core i7 3.4GHz, com quatro núcleos físicos, utilizando o sistema operacional Ubuntu Linux e o JDK 1.7.04. Foram testados os modelos de paralelismo mestre-escravo, ilha e vizinhança multi-indivíduos. Quanto ao modelo de vizinhança que mantém apenas um indivíduo em cada célula, esse não foi incluso neste experimento visto que ele é

apropriado para ser utilizado em computadores maciçamente paralelos com uma grande quantidade de processadores (DE JONG, 2006), (LINDEN, 2012), dessa forma, devido às limitações do *hardware* utilizado, e não possuindo acesso a *hardware* com essas características, optou-se por não incluir esse modelo nos experimentos. Para os experimentos foram gerados algoritmos genéticos com as configurações apresentadas na Tabela 11. Os primeiros 9 parâmetros são comuns a todos os modelos de paralelismo, os 3 seguintes são exclusivos do modelo ilha e os 2 restantes são exclusivos do modelo de vizinhança multi-indivíduos.

Tabela 11 - Parâmetros utilizados nos testes

PARÂMETRO	VALOR
Algoritmo Evolutivo	Algoritmo Genético
Operador de Seleção	Torneio
Tamanho do Torneio	10
Operador de Cruzamento	Heurístico
Taxa de Cruzamento	80%
Operador de Mutação	Aleatória
Taxa de Mutação	1%
Tamanho da População	360
Critério de Parada	2000 gerações
Modelo Ilha	
Taxa de Migração	10 indivíduos
Frequência de Migração	100 gerações
Topologia de Migração	Anel Unidirecional
Modelo de Vizinhança multi-indivíduos	
Formato da vizinhança	Cruz
Tamanho da vizinhança	1

6.6.1 Análise dos Tempos de Execução

O *speedup* e a eficiência obtidos ao executar os algoritmos gerados, utilizando 1, 2, 3 e 4 *threads,* são apresentados na Tabela 12. Os tempos mostrados nessa tabela são as médias dos tempos de execução atingidos em 31 execuções de cada algoritmo, dados em milissegundos, e os valores entre parêntesis mostram o desvio padrão. Para testar o desempenho das versões seriais foram utilizados os mesmos códigos das versões paralelas, mas executando em uma única *thread*.

Foram geradas 31 populações aleatoriamente e cada uma delas foi utilizada como população inicial por todos os algoritmos.

Tabela 12 - Speedup e Eficiência

Threads	Tempo(ms)	Speedup	Eficiência
Mestre-Escravo			
1	3071,774194 (42,49290896)	-	-
2	2318,774194 (40,34328501)	1,324740547	66,2370273%
3	2158,129032 (42,52038094)	1,423350573	47,4450191%
4	2152,774194 (49,43191255)	1,426891033	35,6722758%
Ilha			
1	2939,935 (40,76922)	-	-
2	1516,29 (36,70712)	1,93890021	96,9450105%
3	1069,871 (41,47348)	2,747934097	91,5978032%
4	866,4839 (39,7382)	3,392948213	84,8237053%
Vizinhança multi-indivíduos			
1	2940,419355 (50,47030427)	-	-
2	1554,870968 (47,05722894)	1,891101844	94,5550922%
3	1101,580645 (41,79774651)	2,669272893	88,9757631%
4	942,2903226 (51,34211627)	3,120502551	78,0125638%

Observa-se que todas as versões paralelas apresentaram tempo médio de execução menor que as versões seriais. De fato, de acordo com os experimentos, em todos os algoritmos, quanto maior o número de núcleos utilizados, maior é o *speedup*, por outro lado, menor é a eficiência, uma das causas disso é o fato de que quanto maior o número de núcleos utilizados, maior é o *overhead* causado pela comunicação e sincronização entre as *threads*.

Com base nos resultados dos experimentos é possível observar ainda que o modelo ilha supera o ganho de velocidade e a eficiência dos modelos mestre-escravo (com uma larga vantagem) e vizinhança multi-indivíduos (com uma vantagem menor). Em relação à vantagem adquirida sobre o modelo mestre-escravo, isso ocorre, pois esse tem que lidar com mais aspectos de comunicação e sincronização entre as *threads*, visto que, nesse modelo a sincronização ocorre em todas as gerações, enquanto no modelo ilha, ela ocorre somente quando é efetuada a migração, no caso dos experimentos executados neste trabalho, 1 vez a cada 100 gerações. Além disso, o modelo mestre-escravo deixa os núcleos disponíveis ociosos durante grande parte do processo evolutivo, pois ele utiliza os núcleos em sua totalidade apenas no momento da avaliação dos indivíduos, deixando-os ociosos no restante do processo evolutivo, enquanto o modelo ilha utiliza melhor os recursos paralelos, visto que ele utiliza os vários núcleos disponíveis durante todo o processo evolutivo.

Agora, levando em conta a vantagem obtida pelo modelo ilha em relação ao modelo de vizinhança multi-indivíduos, essa também se dá pelo fato do segundo ter que lidar com mais aspectos de comunicação e sincronização que o modelo ilha, pois também sincroniza a comunicação em todas as gerações, no entanto seu desempenho se aproxima mais do desempenho do modelo ilha que o modelo mestre-escravo, pois ele não desperdiça recursos paralelos deixando núcleos ociosos durante o processo evolutivo.

6.6.2 Análise da Qualidade das Soluções Encontradas

As somas, as médias e a variância das soluções encontradas nas 31 execuções de cada algoritmo testado são apresentadas na Tabela 13. A quantidade de *threads* utilizadas por cada algoritmo estão informadas entre parêntesis.

Tabela 13 - Resumo dos valores das soluções encontradas nos experimentos

Grupo	*Soma*	*Média*	*Variância*
Mestre-Escravo (1)	0,616688	0,019893	0,00063
Mestre-Escravo (2)	0,516084	0,016648	0,000259
Mestre-Escravo (3)	0,649395	0,020948	0,000432
Mestre-Escravo (4)	0,641077	0,02068	0,000638
Ilha (1)	0,577822	0,018639	0,000286
Ilha (2)	0,589862	0,019028	0,000391
Ilha (3)	0,766194	0,024716	0,000664
Ilha (4)	1,218188	0,039296	0,000704
Vizinhança multi-indivíduos (1)	0,645791	0,020832	0,000564
Vizinhança multi-indivíduos (2)	0,604648	0,019505	0,000259
Vizinhança multi-indivíduos (3)	0,531918	0,017159	0,000477
Vizinhança multi-indivíduos (4)	0,609692	0,019667	0,000408

A Tabela 14 mostra os dados resultantes do teste ANOVA aplicado sobre os valores das soluções encontradas nas 31 execuções de cada algoritmo.

Tabela 14 - Resultado do teste ANOVA sobre os valores das soluções encontradas

Fonte da variação	*SQ*	*gl*	*MQ*	*F*	*valor-P*	*F crítico*
Entre grupos	0,012245	11	0,001113	2,338288	**0,008662**	1,815287
Dentro dos grupos	0,171385	360	0,000476			
Total	0,18363	371				

Com base no resultado do teste ANOVA e considerando um nível de significância de 5%, pode-se afirmar que existem diferenças significantes entre os algoritmos testados, no que diz respeito às soluções encontradas por eles. A Tabela 15 mostra os dados resultantes da aplicação do Teste de Tukey sobre os valores das soluções encontradas, esse teste compara os resultados obtidos pelos algoritmos agrupando-os em pares.

Tabela 15 - Resultado do Teste de Tukey aplicado sobre os valores das soluções encontradas

Níveis	*Centro*	*Limite.Inferior*	*Limite.Superior*	*P-valor*
Ilha (2) / Ilha (1)	0,000388389	-0,017843577	0,018620354	1
Ilha (3) / Ilha (1)	0,006076513	-0,012155452	0,024308479	0,994708027
Ilha (4) / Ilha (1)	**0,020656974**	**0,002425008**	**0,038888939**	**0,011910848**
Mestre-Escravo (1) / Ilha (1)	0,001253731	-0,016978235	0,019485696	0,999999999
Mestre-Escravo (2) / Ilha (1)	-0,001991552	-0,020223517	0,016240414	0,999999923
Mestre-Escravo (3) / Ilha (1)	0,002308819	-0,015923147	0,020540784	0,999999629

Mestre-Escravo (4) / Ilha (1)	0,002040496	-0,01619147	0,020272461	0,999999901
Vizinhança multi-indivíduos (1) / Ilha (1)	0,002192557	-0,016039408	0,020424523	0,999999786
Vizinhança multi-indivíduos (2) / Ilha (1)	0,000865337	-0,017366628	0,019097303	1
Vizinhança multi-indivíduos (3) / Ilha (1)	-0,001480789	-0,019712754	0,016751177	0,999999997
Vizinhança multi-indivíduos (4) / Ilha (1)	0,00102805	-0,017203916	0,019260015	1
Ilha (3) / Ilha (2)	0,005688124	-0,012543841	0,02392009	0,997023624
Ilha (4) / Ilha (2)	**0,020268585**	**0,00203662**	**0,038500551**	**0,015222212**
Mestre-Escravo (1) / Ilha (2)	0,000865342	-0,017366623	0,019097308	1
Mestre-Escravo (2) / Ilha (2)	-0,00237994	-0,020611906	0,015852025	0,999999488
Mestre-Escravo (3) / Ilha (2)	0,00192043	-0,016311535	0,020152396	0,999999948
Mestre-Escravo (4) / Ilha (2)	0,001652107	-0,016579858	0,019884073	0,99999999
Vizinhança multi-indivíduos (1) / Ilha (2)	0,001804169	-0,016427797	0,020036134	0,999999974
Vizinhança multi-indivíduos (2) / Ilha (2)	0,000476949	-0,017755017	0,018708914	1
Vizinhança multi-indivíduos (3) / Ilha (2)	-0,001869177	-0,020101143	0,016362788	0,999999961
Vizinhança multi-indivíduos (4) / Ilha (2)	0,000639661	-0,017592304	0,018871627	1
Ilha (4) / Ilha (3)	0,014580461	-0,003651505	0,032812426	0,26648348
Mestre-Escravo (1) / Ilha (3)	-0,004822782	-0,023054748	0,013409183	0,999342278
Mestre-Escravo (2) / Ilha (3)	-0,008068065	-0,02630003	0,010163901	0,951151911
Mestre-Escravo (3) / Ilha (3)	-0,003767694	-0,02199966	0,014464271	0,999940605
Mestre-Escravo (4) / Ilha (3)	-0,004036017	-0,022267983	0,014195948	0,99988226
Vizinhança multi-indivíduos (1) / Ilha (3)	-0,003883956	-0,022115921	0,01434801	0,999919547
Vizinhança multi-indivíduos (2) / Ilha (3)	-0,005211176	-0,023443141	0,01302079	0,998648439
Vizinhança multi-indivíduos (3) / Ilha (3)	-0,007557302	-0,025789267	0,010674664	0,969522313
Vizinhança multi-indivíduos (4) / Ilha (3)	-0,005048463	-0,023280429	0,013183502	0,998991346
Mestre-Escravo (1) / Ilha (4)	**-0,019403243**	**-0,037635208**	**-0,001171278**	**0,025750228**
Mestre-Escravo (2) / Ilha (4)	**-0,022648525**	**-0,040880491**	**-0,00441656**	**0,003106141**
Mestre-Escravo (3) / Ilha (4)	**-0,018348155**	**-0,036580121**	**-0,00011619**	**0,046934416**
Mestre-Escravo (4) / Ilha (4)	**-0,018616478**	**-0,036848443**	**-0,000384512**	**0,040465613**
Vizinhança multi-indivíduos (1) / Ilha (4)	**-0,018464416**	**-0,036696382**	**-0,000232451**	**0,04402983**
Vizinhança multi-indivíduos (2) / Ilha (4)	**-0,019791636**	**-0,038023602**	**-0,001559671**	**0,020411848**
Vizinhança multi-indivíduos (3) / Ilha (4)	**-0,022137762**	**-0,040369728**	**-0,003905797**	**0,004443415**
Vizinhança multi-indivíduos (4) / Ilha (4)	**-0,019628924**	**-0,03786089**	**-0,001396959**	**0,022514776**
Mestre-Escravo (2) / Mestre-Escravo (1)	-0,003245282	-0,021477248	0,014986683	0,999986934
Mestre-Escravo (3) / Mestre-Escravo (1)	0,001055088	-0,017176878	0,019287053	1
Mestre-Escravo (4) / Mestre-Escravo (1)	0,000786765	-0,0174452	0,019018731	1
Vizinhança multi-indivíduos (1) / Mestre-Escravo (1)	0,000938827	-0,017293139	0,019170792	1
Vizinhança multi-indivíduos (2) / Mestre-Escravo (1)	-0,000388393	-0,018620359	0,017843572	1
Vizinhança multi-indivíduos (3) / Mestre-Escravo (1)	-0,002734519	-0,020966485	0,015497446	0,999997789

Vizinhança multi-indivíduos (4) / Mestre-Escravo (1)	-0,000225681	-0,018457647	0,018006284	1
Mestre-Escravo (3) / Mestre-Escravo (2)	0,00430037	-0,013931595	0,022532336	0,9997807
Mestre-Escravo (4) / Mestre-Escravo (2)	0,004032048	-0,014199918	0,022264013	0,999883399
Vizinhança multi-indivíduos (1) / Mestre-Escravo (2)	0,004184109	-0,014047856	0,022416075	0,999832181
Vizinhança multi-indivíduos (2) / Mestre-Escravo (2)	0,002856889	-0,015375076	0,021088854	0,999996506
Vizinhança multi-indivíduos (3) / Mestre-Escravo (2)	0,000510763	-0,017721202	0,018742728	1
Vizinhança multi-indivíduos (4) / Mestre-Escravo (2)	0,003019601	-0,015212364	0,021251567	0,999993786
Mestre-Escravo (4) / Mestre-Escravo (3)	-0,000268323	-0,018500288	0,017963643	1
Vizinhança multi-indivíduos (1) / Mestre-Escravo (3)	-0,000116261	-0,018348227	0,018115704	1
Vizinhança multi-indivíduos (2) / Mestre-Escravo (3)	-0,001443481	-0,019675447	0,016788484	0,999999998
Vizinhança multi-indivíduos (3) / Mestre-Escravo (3)	-0,003789607	-0,022021573	0,014442358	0,999937055
Vizinhança multi-indivíduos (4) / Mestre-Escravo (3)	-0,001280769	-0,019512735	0,016951196	0,999999999
Vizinhança multi-indivíduos (1) / Mestre-Escravo (4)	0,000152061	-0,018079904	0,018384027	1
Vizinhança multi-indivíduos (2) / Mestre-Escravo (4)	-0,001175159	-0,019407124	0,017056807	1
Vizinhança multi-indivíduos (3) / Mestre-Escravo (4)	-0,003521285	-0,02175325	0,014710681	0,999969956
Vizinhança multi-indivíduos (4) / Mestre-Escravo (4)	-0,001012446	-0,019244412	0,017219519	1
Vizinhança multi-indivíduos (2) / Vizinhança multi-indivíduos (1)	-0,00132722	-0,019559186	0,016904745	0,999999999
Vizinhança multi-indivíduos (3) / Vizinhança multi-indivíduos (1)	-0,003673346	-0,021905312	0,014558619	0,999953954
Vizinhança multi-indivíduos (4) / Vizinhança multi-indivíduos (1)	-0,001164508	-0,019396473	0,017067458	1
Vizinhança multi-indivíduos (3) / Vizinhança multi-indivíduos (2)	-0,002346126	-0,020578091	0,015885839	0,99999956
Vizinhança multi-indivíduos (4) / Vizinhança multi-indivíduos (2)	0,000162712	-0,018069253	0,018394678	1
Vizinhança multi-indivíduos (4) / Vizinhança multi-indivíduos (3)	0,002508838	-0,015723127	0,020740804	0,999999106

Com base nos resultados obtidos com a aplicação do teste de Tukey e considerando um nível de significância de 5%, pode-se afirmar que as diferenças ocorrem entre os seguintes pares: (1) o modelo ilha com 4 *threads* e os modelos ilha com 1 e 2 *threads,* (2) o modelo ilha com 4 *threads* e todos os algoritmos mestre-escravo, (3) o modelo ilha com 4 *threads* e todos os algoritmos de vizinhança multi-indivíduos. Nesse contexto, as soluções de melhor qualidade foram encontradas, em

ordem decrescente pelos algoritmos: Mestre-Escravo com 4 *threads*, Vizinhança multi-indivíduos com 3 *threads* e Ilha com 1 *thread*.

A causa mais provável desse resultado é o fato de o modelo ilha com 4 *threads* ser o que mais divide a população e dessa forma, por trabalhar sobre populações consideravelmente menores que os demais algoritmos, sofre mais facilmente com a baixa diversidade e seus efeitos negativos. Sendo assim, embora o modelo ilha com 4 *threads* tenha sido o algoritmo avaliado como mais rápido dentre os testados, ele também é o que encontrou as soluções com menor qualidade.

6.7 CONSIDERAÇÕES FINAIS

Este capítulo tratou sobre o desenvolvimento do JPEAG, uma aplicação web que permite criar código-fonte de AEPs escritos em Java, a partir de parâmetros inseridos em sua interface gráfica. Inicialmente foi apresentada a problemática que levou ao seu desenvolvimento e os requisitos que foram levados em consideração durante esse desenvolvimento. A seguir foi apresentada a arquitetura da ferramenta e seu funcionamento, mostrando a função de suas principais classes e como ela utiliza a estratégia de geração de código baseada em templates. Na sequencia foi mostrado como foi aplicado o paralelismo e a sincronização em cada um dos modelos de paralelismo que a ferramenta permite gerar. Em seguida foram apresentados os diagramas de classe dos códigos gerados para cada modelo de paralelismo e são descritas as principais classes de cada diagrama, identificando as que são exclusivas de cada modelo e as que são comuns a mais de um modelo, explicitando nesse caso, as diferenças que existem entre as implementações da mesma classe em diferentes modelos de paralelismo. É apresentado também o leque de parâmetros dos AEPs que o JPEAG permite configurar e os possíveis valores que poder ser inseridos para cada parâmetro. No final do capítulo, são apresentados os resultados dos experimentos realizados com AEPs gerados pelo JPEAG, baseados na minimização da função de Griewank, foram analisados os dados referentes aos tempos de execução, que mostram o ganho de velocidade à

medida que se utiliza mais tarefas paralelas em cada modelo de paralelismo, e são analisadas também a qualidade das soluções encontradas por cada modelo.

7 CONCLUSÃO

Este trabalho apresentou o *Java Parallel Evolutionary Algorithm Generator* (JPEAG), uma aplicação *web* gráfica que permite aos seus usuários criar algoritmos evolutivos paralelos, escritos com Java, de uma forma amigável. Sua principal vantagem é gerar código-fonte pronto para compilar e executar, mais rapidamente do que um programador o faria, caso fizesse esse mesmo trabalho à mão.

Dessa forma, o JPEAG contribui para o aumento da produtividade do programador e reduz o custo de criação do código paralelo. Além disso, para sua correta operação, ele exige apenas conhecimento acerca da teoria dos AEPs e Java, dispensando conhecimentos profundos sobre a implementação de programas paralelos. Portanto, o JPEAG pode ser utilizado mesmo por programadores sem experiência no desenvolvimento de aplicações paralelas. Além disso, o código gerado é baseado na aplicação de padrões de projeto, o que contribui para o aumento de sua qualidade.

Em relação aos padrões de projeto utilizados, sua aplicação torna o código mais flexível e legível, facilitando sua leitura pelos usuários e permitindo sua modificação conforme a situação demande. O padrão *Strategy* se mostrou eficaz na implementação dos operadores genéticos, provendo-lhes modularização, baixo acoplamento e tornando-os independentes, permitindo a sua utilização em diferentes AEPs e facilitando a inclusão de novas implementações de operadores. Esses efeitos benéficos foram observados também na sua aplicação na implementação dos critérios de parada, políticas de seleção de migrantes e topologias de migração. Além disso, o padrão *Observer* permitiu a implementação de uma solução elegante, também com baixo acoplamento, a fim de controlar a sincronização das tarefas paralelas, por meio de uma troca de informação automática entre os processos paralelos nos diferentes modelos de paralelismo aplicados.

Finalmente, a utilização da estratégia de geração de código baseada em *templates* permite incorporar a geração de novos modelos de AE e de paralelismo, por meio da alteração dos *templates* já existentes para adicionar estruturas que prevejam as características específicas do novo algoritmo. Dessa forma, a geração

de, por exemplo, AEs paralelos Híbridos pode ser facilmente incorporada ao JPEAG. Em último caso, se o novo algoritmo a ser incorporado no gerador possuir uma quantidade muito pequena de similaridades, ou ainda que seja completamente diferente dos *templates* já existentes, podem-se adicionar novos *templates* para gerá-lo.

Em Silva (2013) é apresentado o JPEAG. Trabalho esse que foi aceito, apresentado em sessão oral e publicado nos anais do 11º Congresso Brasileiro de Inteligência Computacional (CBIC).

7.1 TRABALHOS FUTUROS

A implementação do JPEAG continua em curso. Os novos recursos a serem implementados futuramente incluem: AGP *Binary-Coded*, mais operadores genéticos (tanto para AG quanto para EE) e, a implementação de diferentes metaheurísticas tais como, enxame de partículas e evolução diferencial. Uma extensão para trabalhar com algoritmo multiobjetivo também esta em consideração. Além disso, um estudo sobre como realizar a paralelização por meio de comunicação assíncrona deve ser levado em conta.

Extensões que permitam o JPEAG gerar AEPs distribuídos utilizando objetos remotos ou *frameworks* como o ProActive e o JavaMPI também estão em consideração. Implementações de AEPs em outras linguagens como C e Python também estão em mente.

REFERÊNCIAS

ALBA, E.; TOMASSINI, M . **Parallelism and evolutionary algorithms**. Evolutionary Computation, IEEE Transactions on, v. 6, n. 5, p. 443-462, 2002.

ALBA, E.; LUQUE, G. **Evaluation of parallel metaheuristics**. Lecture Notes in Computer Science, v. 4193, p. 9-14, 2006.

ALMASI, G. S., GOTTLIEB A. **Hight Parallel Computing**, 2a ed., The Benjamin Cummings Publishing Company, Inc., 1994.

BÄCK, Thomas; HAMMEL, Ulrich; SCHWEFEL, H.-P. **Evolutionary computation**: Comments on the history and current state. Evolutionary computation, IEEE Transactions on, v. 1, n. 1, p. 3-17, 1997.

BARNEY, B. et al. **Introduction to parallel computing**. Lawrence Livermore National Laboratory, v. 6, n. 13, p. 10, 2010.

CAHON, S.; MELAB, N.; TALBI, E. **ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics**. Journal of Heuristics, v. 10, n. 3, p. 357-380, 2004.

CAMARGO, R. Y.; ANDRADE, R. **Grades Computacionais: Uma Introdução Prática**. Departamento de Ciência da Computação, Instituto de Matemática e Estatística, Universidade de São Paulo, Brasil, 2007.

CANTÚ-PAZ, E. **A Survey of Parallel Genetic Algorithms**. Department of Computer Science and Illinois Genetic Algorithms Laboratory. University of Illinois at Urbana-Champaign, 1998.

CANTÚ-PAZ, E. **Parameter setting in parallel genetic algorithms**. In: Parameter Setting in Evolutionary Algorithms. Springer Berlin Heidelberg, 2007. p. 259-276.

CORTES, O. A. C.; SAAVEDRA, O. **Estratégias evolutivas paralelas em otimização multimodal**. INFOCOMP–Revista de Ciência da Computação–Anais da III SECICOM, v. 2, n. 1, p. 63-68, 2000.

CORTES. O. A. C.; SANTANA, R. H. C.; SANTANA, M. J.; MENDEZ, O. R. S. **Análise de Operadores de Recombinação em Estratégias Evolutivas Aplicados no Refinamento de um Sistema Nebuloso**. In: Simpósio Brasileiro de Automática Inteligente, 2005.

DE JONG, K. A. **Evolutionary computation**: a unified approach. Cambridge: MIT press, 2006.

EIBEN, A. E., SMITH, J. E. **Introduction to Evolutionary Computing**, Berlim: Springer Verlag, 2003.

EIBEN, A. E.; SMIT, S. K. **Parameter tuning for configuring and analyzing evolutionary algorithms**. Swarm and Evolutionary Computation, v. 1, n. 1, p. 19-31, 2011.

FENG, H.; LI-SHAN, K.; YU-PING, C. **A generic design model for evolutionary algorithms**. Wuhan University Journal of Natural Sciences, v. 8, n. 1, p. 224-228, 2003.

FERREIRA, R. **Implementação de algoritmos genéticos paralelos em uma arquitetura MPSoC**. Dissertação (Mestrado). UERJ, Rio de Janeiro – 2009.

FREEMAN, Eric; FREEMAN, Elisabeth. **Use a cabeça! (Head First) – Padrões de Projeto (Design Patterns)**. Alta Books, 2007.

GAMMA, E. et al. **Padrões de Projetos**: Soluções Reutilizáveis de software orientado a objetos. Bookman, 2008.

GOLDBERG, D. E. **Genetic algorithms in search, optimization, and machine learning.** Reading, MA, USA: Addison-Wesley Professional, 1989.

GOMES, J. L. S. **Paralelização de algoritmo de simulação de Monte Carlo para a absorção em superfícies heterogêneas bidimensionais**. Dissertação (Mestrado) - Universidade Estadual de Maringá, Maringá, PR, Brasil, 2009.

GRADECKI, J. D.; COLE, J. **Mastering Apache Velocity**. Wiley, 2003.

HAWICK, K. A.; PLAYNE, D. P. **Automated and parallel code generation for finite-differencing stencils with arbitrary data types**. Procedia Computer Science, v. 1, n. 1, p. 1795-1803, 2010.

HE, Jun; YAO, Xin. **Analysis of scalable parallel evolutionary algorithms**. In: Proceedings of CEC. 2006. p. 120-127.

HERRERA, F.; LOZANO M.; and VERDAGAY, J. L. **Tackling Real-Coded Genetic Algorithms**: Operators and Tools for Behavioural Analisys. Artificial Intelligence Review, 4(12):265–319, 1998.

HERRINGTON, J. **Code generation in action**. Greenwich: Manning Publications, 2003.

JAQUIE, K. **Extensão da Ferramenta de Apoio à Programação Paralela (F.A.P.P.) para Ambientes Paralelos Virtuais**. Dissertação (Mestrado) – USP, São Carlos, SP, Brasil, fev 1999.

KIRK, D. B.; WEN-MEI, W. H. **Programming massively parallel processors**: a hands-on approach. Morgan Kaufmann, 2010.

LACERDA, S.; CARVALHO, A. **Introdução aos algoritmos genéticos**. In: Sistemas inteligentes: aplicações a recursos hídricos e ciências ambientais. Porto Alegre, RS, Brazil: Editora da Universidade Federal do Rio Grande do Sul 1999.

LINDEN, R. **Algorítmos Genéticos**, 3ed, Rio de Janeiro: Editora Ciência Moderna, 2012.

LUCAS, D. **QCodeGenerator: um gerador de código multilinguagem baseado em templates**. FACENSA, Gravatai, RS, Brasil, 2005.

LUCRÉDIO, D. **Uma Abordagem Orientada a Modelos para Reutilização de Software.**Tese (Doutorado)–USP, São Carlos, SP, Brasil, jul de 2009.

MALACARNE, J. **Ambiente Visual para Programação Distribuída em Java**. Dissertação (Mestrado) – UFRGS, Porto Alegre, RS, Brasil, fev 2001.

MANOLESCU, D. **Pattern Languages of Program Design 5**. Reading: Addison-Wesley Professional, 2006.

MARKOWSKA-KACZMAR, U.; KRYGOWSKI, F. **The influence of using design patterns on the process of implementing genetic algorithms**. In: Trends in Applied Intelligent Systems. Springer Berlin Heidelberg, 2010. p. 173-182.

MCCOOL, M.; REINDERS, J.; ROBISON, A. **Structured Parallel Programming:** Patterns for Efficient Computation. Elsevier, 2012.

MICHALEWICZ, Z., **Genetic Algorithms + Data Structures = Evolution Programs**, 3 ed, Berlim: Springer Verlag, 1996.

NESMACHNOW, S.; CANCELA, H.; ALBA, E. **A parallel micro evolutionary algorithm for heterogeneous computing and grid scheduling**. Applied Soft Computing, v. 12, n. 2, p. 626-639, 2012.

ORACLE Java Standard Edition 7 Documentation - **http://docs.oracle.com/javase/7/docs** - Acessado em: 12/10/2013.

PACHECO, M. **Algoritmos genéticos**: princípios e aplicações. PUC, Rio de Janeiro, RJ, Brasil, 1999.

PACHECO, P. **An introduction to parallel programming**. Elsevier, 2011.

PASINI, F.; DOTTI, L. **Code Generation for Parallel Applications Modelled with Object-Based Graph Grammars, Electronic Notes in Theoretical Computer Science**, v. 184, p. 113-131, 2007.

PEREIRA, R.; MENDEZ, O. R. S.; CORTES, O. A. C. **Parallel Distributed Evolution Strategies**. In: Simposio Brasileiro de Automacao Inteligente, 2001, Gramado - RS. V Simposio Brasileiro de Automacao Inteligente, 2001.

PINHEIRO, A. **Planejamento dinâmico e controlo de trajectórias de robôs**. Dissertação (Mestrado). Instituto Superior de Engenharia do Porto, Porto, Portugual, jun 2007.

RODRIGUES, A. et al. **Automatic Multi-GPU Code Generation applied to Simulation of Electrical Machines**. Magnetics, IEEE Transactions on, v. 48, n. 2, p. 831-834, 2012.

SILVA, A. J. M. **Implementação de um algoritmo genético utilizando o modelo de ilhas**. Dissertação (Mestrado) - UFRJ, Rio de Janeiro, RJ, Brasil, ago 2005.

SILVA, J. A.; CORTES, O. A. C.; SILVA, J. C. **A Java-Based Code Generator for Parallel Evolutionary Algorithms**. In: 11º Congresso Brasileiro de Inteligência Computacional (CBIC), Porto de Galinhas-PE. 11th Brazilian Congress on Computational Intelligence (CBIC), 2013.

Apache VELOCITY Project Site - **http://velocity.apache.org** - Acessado em 10/10/2013.

YAO, Xin; LIU, Yong. **Fast evolution strategies**. In: Evolutionary Programming VI. Springer Berlin Heidelberg, 1997. p. 149-161.

YU, Xinjie; GEN, M. **Introduction to Evolutionary Algorithms**. Springer, 2010.

ZK Framework Site - **http://www.zkoss.org** - Acessado em: 10/10/2013.