UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

EVANDO CARLOS PESSINI

ALGORITMOS GENÉTICOS PARALELOS – UMA IMPLEMENTAÇÃO DISTRIBUÍDA BASEADA EM JAVASPACES

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Prof. José Mazzucco Junior, Dr.
Orientador

Florianópolis, Março de 2003

ALGORITMOS GENÉTICOS PARALELOS – UMA IMPLEMENTAÇÃO DISTRIBUÍDA BASEADA EM JAVASPACES

EVANDO CARLOS PESSINI

Esta dissertação foi julgada adequada para a obtenção do título de mestre em Ciência da Computação Área de Concentração: Sistemas de Conhecimento e aprovada em sua forma final pelo Curso de Pós-Graduação em Ciência da Computação – CPGCC

_____

Prof. Fernando A. Ostuni Gauthier, Dr. – Coordenador.

Banca Examinadora:

_____

Prof. José Mazzucco Júnior, Dr. – Orientador.

_____

Prof. Rogério Cid Bastos, Dr.

_____

Prof. Alvaro Rojas Lesana, Dr.

Dedicatória

Dedico este trabalho a minha querida e amada esposa Marcia

que sempre esteve ao meu lado oferecendo seu amor.

Agradecimentos

Agradeço a Deus pelo dom da vida.

Agradeço a minha esposa Marcia pelo apoio e pela compreensão em relação a minha ausência neste período.

Agradeço aos meus pais, meus primeiros “orientadores” nesta vida. Em especial, a minha mãe que sempre me incentivou na continuidade dos estudos.

Agradeço ao meu orientador Mazzucco por ter sido mais que um orientador, um amigo.

Agradeço ao meu amigo Vilson pelas boas discussões, sugestões e questionamentos que foram de fundamental importância para a conclusão deste trabalho.

A todos meus amigos, em especial Anibal e Ivonei, pela ajuda e pelas discussões que muito contribuíram para a realização deste trabalho.

Agradeço a UNIVEL e ao CEFET/MD, por ter disponibilizado a infra-estrutura necessária para o desenvolvimento e realização dos testes do protótipo.

Agradeço aos funcionários do laboratório de informática dessas instituições, que sempre estiveram dispostos a ajudar e, muitas vezes, tiveram que sair após o término do horário de trabalho por minha causa.

Resumo

PESSINI, E.C. **Algoritmos Genéticos Paralelos – Uma Implementação Distribuída Baseada em JavaSpaces**. 2003. 82f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Ciência da Computação, Universidade Federal de Santa Catarina, Florianópolis, SC, 2003.

Os algoritmos genéticos têm deficiências conhecidas, principalmente no que diz respeito ao alto custo computacional e a baixa qualidade das soluções devido a convergência prematura. Um algoritmo genético clássico executado em um espaço de endereçamento simples tende a alcançar um ponto de equilíbrio onde os descendentes são muito semelhantes aos seus pais. Esta diversidade limitada induz o algoritmo genético a explorar somente uma região restrita do espaço de soluções, resultando em soluções subótimas. Uma tentativa de evitar este problema é criar um ambiente onde diversas populações independentes evoluem em paralelo e, periodicamente, efetuam a troca (migração) de indivíduos objetivando evitar a convergência prematura e manter a diversidade da população. Esta pesquisa apresenta a implementação de um algoritmo genético paralelo assíncrono de granularidade grossa (*coarse grain*) que usa a tecnologia JavaSpaces como mecanismo de distribuição das populações e dos indivíduos migrantes. A tecnologia JavaSpaces foi usada como repositório de objetos para a efetivação da comunicação entre as diversas máquinas do ambiente distribuído. Para avaliar a funcionalidade e o desempenho do algoritmo, aplicou-se o mesmo na obtenção de soluções para o Problema do Caixeiro Viajante (PCV) com o uso de soluções conhecidas disponíveis na Internet.

Palavras-chave: Algoritmo Genético Paralelo, JavaSpaces, Computação Distribuída, Problema do Caixeiro Viajante.

Abstract

PESSINI, E.C. **Algoritmos Genéticos Paralelos – Uma Implementação Distribuída Baseada em JavaSpaces**. 2003. xxxxf. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Ciência da Computação, Universidade Federal de Santa Catarina, Florianópolis, SC, 2003.

The genetic algorithms have known deficiencies in that concerns the high cost computational and the low quality of the solutions due to premature convergence. A classical genetic algorithm executed in a simple address space tends to reach a balance point where the descendants are very similar to their parents. This limited diversity induces the genetic algorithm to explore only a restricted area of the space of solutions, resulting in suboptimal solutions. An attempt of avoiding this problem is to create an environment in which several independent populations evolving in parallel and, periodically, exchange (migration) individuals aiming to avoid the premature convergence and to maintain the diversity of the population. This research presents the implementation of a asynchronous coarse grain parallel genetic algorithm that uses the JavaSpaces technology as mechanism of distribution of the populations and of the migrating individuals. The JavaSpaces technology was used as repository of objects to realize the communication among the several machines of the distributed environment. To evaluate the functionality and the performance of the algorithm, it was applied to obtain solutions for the Travelling Salesman Problem (TSP).

Key-words : Parallel Genetic Algorithm, JavaSpaces, Distributed Computing, Travelling Salesman Problem.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Gráficos

1 Introdução

Algoritmos Genéticos são eficientes métodos de busca baseados nos princípios da seleção natural e da genética. Essas características tornam os algoritmos genéticos um importante modelo computacional para a busca de soluções em problemas de otimização através da simulação do processo evolutivo natural (GOLDBERG, 1994).

No entanto, os algoritmos genéticos convencionais tem deficiências conhecidas, como velocidade e convergência prematura. Na tentativa de superar essas deficiências, foram introduzidos os algoritmos genéticos paralelos. A paralelização justifica-se também pelo paralelismo intrínseco existente no processo de evolução natural simulado pelo algoritmo genético.

A paralelização do algoritmo genético depende de fatores, como por exemplo, a distribuição das populações (simples ou múltiplas), o processo de seleção (global ou local), a aplicação dos operadores e da migração dos indivíduos (NOWOSTAWSKI & POLI, 1999). Tendo em vista as inúmeras possibilidades de combinação destes fatores, não há um consenso entre os pesquisadores sobre os modelos de algoritmos genéticos paralelos. Algumas taxonomias para categorização desses algoritmos foram propostas por (CANTÚ-PAZ, 1997) e (NOWOSTAWSKI & POLI, 1999).

De maneira geral, os algoritmos genéticos paralelos podem ser classificados em 3 categorias. No modelo AG paralelo global, todos os indivíduos pertencem a uma grande população

(*panmitic model*), sendo que o paralelismo ocorre na etapa de avaliação dos indivíduos, a qual é distribuída nos diversos processadores do ambiente.

Ao contrário do AG paralelo global, no AG de granularidade grossa (*coarse grain GA*) os indivíduos são divididos em várias subpopulações (*demes*). As subpopulações executam em paralelo nos diversos processadores e trocam informações através da migração. Pelo mecanismo de migração é possível realizar a troca de material genético entre as subpopulações e conduzir a busca na direção do ótimo global. Este modelo é conhecido também como a*lgoritmo genético distribuído.*

Outro tipo de algoritmo genético paralelo é o AG de granularidade fina (*fine grain GA*), o qual é próprio para execução em computadores maciçamente paralelos. Este modelo diferencia-se do algoritmo genético de granularidade grossa pelo aumento do número de subpopulações e pela diminuição do número de indivíduos que compõem a subpopulação.

1.1 Motivação

O presente trabalho foi motivado pelo desejo de desenvolver um algoritmo genético paralelo capaz de explorar o paralelismo existente em um ambiente distribuído. O uso de um ambiente distribuído torna possível a solução de instâncias mais complexas (maiores) do PCV e de outros problemas dessa natureza.

1.2 Objetivo Geral

Este trabalho tem como objetivo principal desenvolver um algoritmo genético paralelo assíncrono de granularidade grossa para ser executado num ambiente distribuído.

1.2.1 Objetivos Específicos

- Desenvolver um mecanismo baseado na tecnologia JavaSpaces que possibilite a distribuição da execução do algoritmo genético paralelo.
- Avaliar o impacto da migração de indivíduos entre as diversas populações.
- Avaliar o desempenho da implementação proposta através da aplicação em instâncias públicas do problema do caixeiro viajante (PCV).

1.3 Limitações da pesquisa

Este trabalho não tem o objetivo de usar o mecanismo proposto para superar atuais recordes do PCV.

O algoritmo genético paralelo desenvolvido é o AG clássico de granularidade grossa (*também conhecido como Algoritmo Genético Distribuído*), sendo que os outros modelos de algoritmo genético paralelo estão fora do escopo desta pesquisa.

A avaliação quantitativa de desempenho da tecnologia JavaSpaces também não faz parte do objetivo deste trabalho.

O algoritmo genético desenvolvido foi testado numa rede local, sendo que sua distribuição em outros ambientes (por exemplo, Internet) está fora do objetivo da pesquisa.

1.4 Estrutura do trabalho

O presente trabalho é composto de 6 capítulos, organizados da seguinte forma.

O capítulo 1 inicia com uma breve introdução aos algoritmos genéticos paralelos. A seguir são apresentados os objetivos, as limitações e a relevância da pesquisa.

O capítulo 2 discute os detalhes relacionados aos algoritmos genéticos, em especial os métodos de paralelização dos algoritmos genéticos, bem como questões decorrentes dessa paralelização, como política de migração, topologia e tamanho das populações.

O capítulo 3 fornece uma visão geral da tecnologia JavaSpaces empregada neste trabalho como mecanismo de distribuição do algoritmo genético paralelo.

No capítulo 4 é descrito o funcionamento do mecanismo de distribuição do algoritmo genético paralelo proposto e implementado, assim como as classes que compõe o protótipo usado na bateria de testes.

No capítulo 5 são apresentados os resultados gerados pelo algoritmo em forma de tabelas e gráficos. No final do capítulo, é feita uma discussão sobre os resultados obtidos nos testes realizados com as instâncias do PCV.

Por fim, no capítulo 6 são apresentadas as conclusões e as sugestões de pesquisas futuras.

2 Algoritmos Genéticos

2.1 Introdução

A solução de problemas de elevado nível de complexidade computacional tem sido um desafio constante para os pesquisadores de diversas áreas. Particularmente em Otimização, Pesquisa Operacional, Ciência da Computação, Matemática e Engenharias, os pesquisadores defrontam-se freqüentemente com problemas altamente combinatórios cuja solução ótima, em muitos casos, ainda está limitada somente a pequenas instâncias (OCHI & VIANNA, 1998).

Segundo (DE JONG, 1993), os métodos de otimização tradicionais, denominados exatos, caracterizam-se pela rigidez de seus modelos matemáticos, dificultando a representação de situações reais cada vez mais complexas e dinâmicas. Esta falta de flexibilidade foi amenizada pela associação de técnicas de otimização com ferramentas de Inteligência Artificial, mais especificamente, com as ferramentas de busca heurística. De fato, os algoritmos heurísticos, ou simplesmente heurísticas, caracterizam-se pela sua flexibilidade e buscam encontrar soluções de boa qualidade num tempo computacional aceitável.

Entretanto, as heurísticas isoladamente também possuem suas limitações, sendo que a principal delas é a deficiência histórica de não conseguirem, em muitos casos, superar as

armadilhas dos ótimos locais em problemas de otimização. Além disso, a falta de uma base teórica dos métodos heurísticos produz algoritmos muito especializados.

A união dos modelos rígidos da otimização com os métodos flexíveis da busca heurística proporcionaram o surgimento de novos métodos, os quais procuram o desenvolvimento de técnicas dotadas de uma certa rigidez matemática e com facilidades em incorporar novas situações.

Os anos 80 marcam o surgimento de novos métodos heurísticos com ferramentas adicionais para tentar superar as limitações das heurísticas convencionais. Dentre os vários métodos produzidos para tentar reduzir o risco de paradas prematuras, destacam-se: as Redes Neurais Artificiais, Simulated Annealing, Tabu Search e a Programação Evolutiva, a qual inclui os Algoritmos Genéticos inicialmente propostos por (HOLLAND, 1975), a Programação Genética proposta por (KOZA, 1992) e a Programação Evolutiva proposta por (FOGEL, 1966).

Embora com filosofias distintas, estas metaheurísticas possuem, em comum, características que as distinguem das heurísticas convencionais como, por exemplo, incluir ferramentas para tentar escapar das armadilhas dos ótimos locais e a facilidade para trabalhar em ambientes paralelos (MIKI et all, 1999).

Dentre estas técnicas, os Algoritmos Genéticos têm se destacado na solução de uma diversidade de problemas devido a sua simplicidade e sua facilidade em resolver problemas sem exigir muito conhecimento prévio dos mesmos.

Os algoritmos genéticos foram inicialmente propostos pelo professor John Holland da Universidade de Michigan (HOLLAND, 1975), mas somente a partir dos anos 80 é que realmente começaram a se popularizar. A idéia inicial de (HOLLAND, 1975) foi tentar imitar as etapas do processo de evolução natural das espécies incorporando-as em um algoritmo computacional.

2.2 Algoritmos Genéticos Convencionais

Os Algoritmos Genéticos formam a parte da área dos Sistemas Inspirados na Natureza; simulando os processos naturais e aplicando-os à solução de problemas reais. São métodos generalizados de busca e otimização que simulam os processos naturais de evolução, aplicando a idéia darwiniana de seleção. De acordo com a aptidão e a combinação com outros operadores genéticos, são produzidos métodos de grande robustez e aplicabilidade (HOLLAND, 1975).

A solução de um problema usando a abordagem de Algoritmos Genéticos é representada através de unidades denominadas *genes*. Uma coleção de *genes* é chamada de *cromossomo (indivíduo)*. Um conjunto de *cromossomos* (indivíduos) forma uma *população*. Em Algoritmos Genéticos, existem várias maneiras de codificar uma solução para um problema, incluindo representação em strings de bits e árvores. A qualidade de um *cromossomo* (indivíduo) é chamada de *aptidão*.

Na natureza os indivíduos competem entre si por recursos como comida, água e refúgio. Adicionalmente, entre os animais de uma mesma espécie, aqueles que não obtêm êxito tendem provavelmente a ter um número reduzido de descendentes, tendo portanto, menor probabilidade de seus genes serem propagados ao longo de sucessivas gerações. A combinação entre os genes dos indivíduos que perduram na espécie, podem produzir um novo indivíduo muito melhor adaptado às características de seu meio ambiente (MAULDIN, 1984).

Os algoritmos genéticos utilizam uma analogia direta deste fenômeno de evolução na natureza, onde cada indivíduo representa uma possível solução para um problema dado. A cada indivíduo se atribui uma aptidão, dependendo da resposta dada ao problema por este indivíduo. Aos mais aptos é dada a oportunidade de reproduzir-se mediante cruzamentos com outros indivíduos da população, produzindo descendentes com características de ambas as partes.

O algoritmo genético pode convergir em uma busca de azar, porém sua utilização assegura que nenhum ponto do espaço de busca tem probabilidade zero de ser examinado. Toda tarefa de busca e otimização possui vários componentes, entre eles: o espaço de busca onde são

consideradas todas as possibilidades de solução de um determinado problema, e a função de avaliação (ou função de custo), uma maneira de avaliar os indivíduos do espaço de busca.

As técnicas de busca e otimização tradicionais iniciam-se com um único candidato que, iterativamente, é manipulado utilizando algumas heurísticas (estáticas) diretamente associadas ao problema a ser solucionado. Por outro lado, as técnicas de computação evolutiva operam sobre uma população de candidatos em paralelo. Assim, elas podem fazer a busca em diferentes áreas do espaço de solução, alocando um número de membros apropriado para a busca em várias regiões (ZUBEN, 2000).

Os Algoritmos Genéticos diferem dos métodos tradicionais de busca e otimização, principalmente em quatro aspectos (GOLDBERG, 1994):

- AGs trabalham com uma codificação do conjunto de parâmetros e não com os próprios parâmetros
- AGs trabalham com uma população e não com um único ponto
- AGs utilizam informações de custo ou outro conhecimento auxiliar
- AGs utilizam regras de transição probabilísticas e não determinísticas.

2.2.1 Composição de um Algoritmo Genético

Um algoritmo genético é composto de uma população inicial de cromossomos que se mantém a margem de todo processo evolutivo. A cada cromossomo atribui-se um valor de aptidão que está relacionado com o valor da função objetivo a ser otimizada. Cada cromossomo representa um ponto do espaço de busca do problema.

Dois cromossomos são selecionados para serem pais de uma nova solução através do mecanismo de *crossover*. Nos algoritmos originais de Holland, um deles era eleito de acordo com o seu valor de aptidão medido por uma função de avaliação, enquanto o outro pai era eleito aleatoriamente. A operação de *crossover* produz as configurações binárias dos filhos. Estas soluções substituem duas soluções da população que eram eleitas ao acaso (HOLLAND, 1975).

Esta é a descrição de um modelo muito simples de evolução que trata da incorporação dos conceitos de sobrevivência e seleção do mais apto.

Desde o início, tem se concentrado grandes esforços no estudo de variações deste esquema básico dos algoritmos genéticos. Independentemente da sofisticação do algoritmo genético, existem 5 componentes que devem ser incluídos (DALLE MOLE, 2002):

1. Construção de uma população inicial de cromossomos.
2. Avaliação dos cromossomos usando a função de avaliação, para determinar a qualidade de cada cromossomo com relação ao problema em questão.
3. Aplicação da seleção para determinar os cromossomos que formarão a próxima geração.
4. Reprodução de cromossomos através da aplicação dos operadores genéticos *mutação* e *crossover*.
5. Verificação da condição de parada. Se a condição não foi atingida, volta ao passo 2.

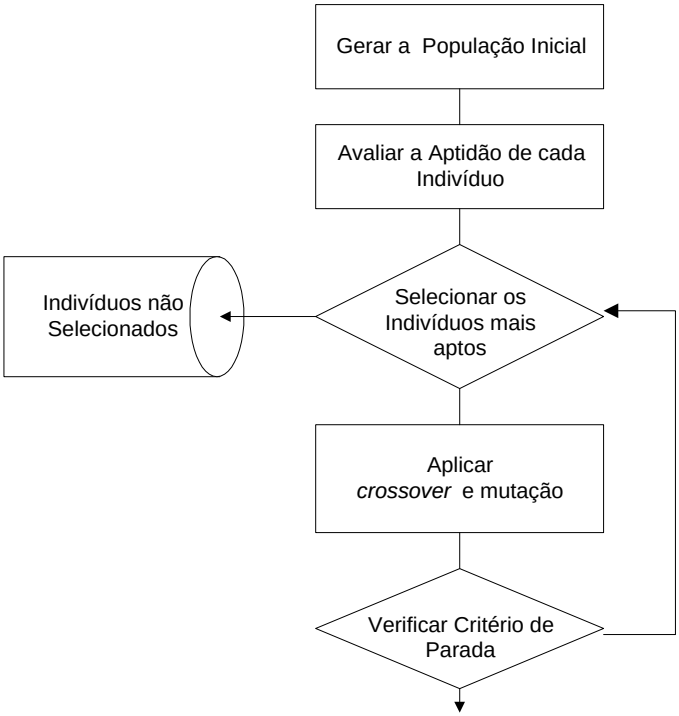


Figura 1 - Fluxograma genérico dos Algoritmos Genéticos (DALLE MOLE, 2002 – pg 41)

2.2.2 Representação da solução na estrutura de cromossomos

Segundo (MAZZUCCO, 1999), o emprego do algoritmo genético na resolução de um determinado problema depende fortemente da realização de dois importantes passos iniciais:

1. Encontrar uma representação adequada na forma de cromossomo para as possíveis soluções do problema,

2. Determinar uma função de avaliação que forneça uma medida do valor (da aptidão) de cada cromossomo gerado.

Os Algoritmos Genéticos, em sua forma original (HOLLAND, 1975), trabalham com uma codificação binária para representar uma solução do problema abordado. Embora esta representação se mostrou eficiente na solução de vários problemas, observou-se que em diversos problemas com um elevado número de restrições, esta representação pode era a mais adequada. Surgiram então alternativas como a representação por inteiros, onde um cromossomo é descrito por um vetor de números inteiros.

Independentemente do tipo de representação selecionada, deve-se sempre verificar se a representação está corretamente associada com as soluções do problema analisado. Ou seja, que toda solução tenha um cromossomo associado e reciprocamente que todo cromossomo gerado pelo AG esteja associado a uma solução válida do problema analisado.

2.2.3 Construção de uma população inicial de cromossomos

O problema de gerar uma população inicial de cromossomos num AG normalmente não é uma tarefa das mais difíceis. Pelo contrário, em muitos problemas esta tarefa é muito simples.

No caso do PCV com n cidades, gerar k soluções viáveis na estrutura de k cromossomos usando uma representação por números inteiros, significa gerar k seqüências com n números inteiros distintos de 1 até n . Neste caso, gera-se aleatoriamente k cromossomos viáveis.

2.2.4 Avaliação dos cromossomos (aptidão)

A determinação da função de avaliação da aptidão é, em geral, simples quando apenas um critério ou especificação deve ser atendido. Esta especificação é então representada por uma função objetivo e esta é utilizada como função de avaliação da aptidão. Porém, a maior parte dos problemas de interesse prático deve atender a *múltiplos objetivo*s, uma vez que diversas especificações devem ser levadas em consideração (GOLDBERG, 1989). A função de avaliação retorna um escalar como medida de desempenho do cromossomo.

A função de avaliação constitui-se no único elo de ligação entre o algoritmo genético e o problema a ser resolvido. Recebendo um cromossomo como entrada, essa função retorna um número ou uma lista de números que exprime a medida do desempenho daquele cromossomo, no contexto do problema a ser resolvido. A função de avaliação, no algoritmo genético, desempenha o mesmo papel que o ambiente no processo de evolução natural. Assim como, a interação de um indivíduo com seu meio ambiente fornece uma medida de sua aptidão, a interação de um cromossomo com uma função de avaliação também resulta em uma medida de aptidão, medida essa que o algoritmo genético utiliza na realização do processo de reprodução.

2.2.5 Mecanismo de reprodução – Mutação e Crossover

O operador genético representa o núcleo de um AG. O objetivo básico de um operador genético é produzir novos cromossomos que possuam propriedades genéticas superiores às encontradas nos pais. Nos Algoritmos Genéticos convencionais, os operadores genéticos clássicos são: **mutação** e **crossover**.

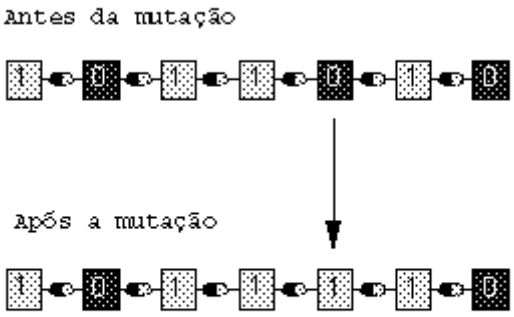


Figura 2- Aplicação do operador Mutação (STRACUZZI, 1998 – pg 2)

O operador de *mutação* implica na alteração do valor de um ou mais genes de um cromossomo buscando restaurar o material genético perdido ou não explorado em uma população, visando prevenir a convergência prematura do AG para soluções sub-ótimas, uma vez que nas imediações de um ótimo local os cromossomos se tornam muito semelhantes anulando, desta forma, as ações do *crossover*.

O operador *crossover* clássico consiste em gerar um ou dois cromossomos filhos a partir das informações dos dois cromossomos pais. No operador *crossover* determina-se inicialmente um ou mais cortes entre dois genes adjacentes de cada cromossomo pai.

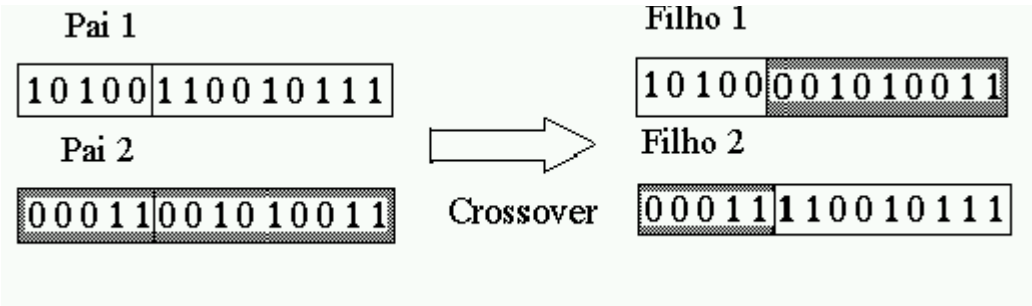


Figura 3 - Aplicação do operador Crossover (CANTÚ-PAZ, 1997 – pg 4)

O operador *crossover* é responsável pela troca de material genético entre os cromossomos com probabilidade de produzirem mais freqüentemente bons cromossomos, ou seja, cromossomos mais aptos ao ambiente.

Os dois operadores clássicos do AG, *mutação* e *crossover*, são normalmente utilizados conjuntamente. Em cerca de 80% das reproduções é utilizado o operador *crossover* e um pequeno percentual destinado ao uso do operador *mutação*. Normalmente o operador *mutação* é utilizado dentro de um AG para permitir uma diversificação no processo de busca, tentando com isso escapar de ótimos locais evitando a convergência prematura (OCHI & VIANNA, 1998).

Estes dois mecanismos de reprodução, no entanto, embora muito simples e genéricos, mostraram ser limitados para vários problemas de elevada complexidade, como por exemplo, problemas de otimização com vários pontos de ótimo local. O uso de operadores clássicos, muitas vezes, não analisam muitas regiões promissoras onde podem estar localizadas as melhores soluções do problema.

2.2.6 Parâmetros Genéticos

É importante analisar também de que maneira alguns parâmetros influenciam no comportamento dos algoritmos genéticos, para que se possa estabelecê-los conforme as necessidades do problema e dos recursos disponíveis.

- **Tamanho da População**: o tamanho da população afeta o desempenho global e a eficiência dos AGs. Uma população pequena oferece uma pequena cobertura do espaço de busca, causando uma queda no desempenho. Uma grande população fornece uma melhor cobertura do domínio do problema e previne a convergência prematura para soluções locais. Entretanto, com uma grande população tornam-se necessários recursos computacionais maiores, ou um tempo maior de processamento do problema (GOLBERG et all, 1995).

- **Taxa de Cruzamento**: quanto maior for esta taxa, mais rapidamente novas estruturas serão introduzidas na população. Entretanto, isto pode gerar um efeito indesejado pois a maior parte da população será substituída podendo ocorrer perda de estruturas de alta aptidão. Com um valor baixo, a convergência torna-se mais lenta.

- **Taxa de Mutação**: uma baixa taxa de mutação previne que uma dada população fique estagnada em um valor, além de possibilitar que se chegue em qualquer ponto do espaço de busca. Com uma taxa muito alta a busca torna-se essencialmente aleatória.
- **Intervalo de Geração**: controla a percentual da população que será substituída durante a próxima geração.

2.3 Algoritmos Genéticos Paralelos

Algoritmos genéticos demandam muitos recursos computacionais quando comparados a outros métodos determinísticos de busca e otimização. Isto se deve ao fato dos algoritmos genéticos usar uma abordagem probabilística a medida que buscam alcançar uma região maior que os algoritmos determinísticos, permitindo que vários pontos no domínio do problema sejam pesquisados simultaneamente (LIN et all, 1994).

Este fator torna os AGs ideiais para paralelização. Em alguns casos, o algoritmo genético pode ser paralelizado de forma que cada cromossomo tenha seu próprio processador para executar os cálculos necessários. Isso essencialmente reduz o tempo de execução para cada geração.

Porém, a velocidade não é a única razão para paralelizar o algoritmo genético. Algumas implementações de algoritmos genéticos paralelos tem um custo significativamente maior que os algoritmos genéticos convencionais, porém aumentam a cobertura do espaço de soluções aumentando a chance de encontrar a solução ótima (DALLE MOLE, 2002).

O argumento mais intuitivo para paralelização do algoritmo genético é simplesmente o fato do modelo natural implementado pelo algoritmo genético ser intrinsecamente paralelo.

Existem outras razões para paralelizar um algoritmo genético. Uma delas é reduzir a chance de convergência prematura, a qual ocorre quando uns poucos indivíduos com alta aptidão começam a dominar a população fazendo com que os outros indivíduos assemelhem-se a eles (MAULDIN, 1984).

Segundo (NOWOSTAWSKI & POLI, 1999), a maneira pela qual os Algoritmos Genéticos podem ser paralelizados depende de alguns fatores:

- Como são aplicados os operadores genéticos *mutação* e *crossover*.
- Como são distribuídas as populações (simples ou múltiplas)
- No caso de múltiplas populações (demes), como os indivíduos são trocados
- Como a seleção é aplicada (localmente ou globalmente)

Dependendo da implementação de cada um desses fatores, vários métodos diferentes de paralelização podem ser obtidos. Como existem várias possibilidades de combinação desses fatores, não há um consenso com relação a nomenclatura usada para classificar os algoritmos genéticos paralelos.

Houve algumas tentativas no sentido de desenvolver uma taxonomia unificada de algoritmos genéticos paralelos, por exemplo (CANTÚ-PAZ, 1997). Uma das mais abrangentes foi definida por (NOWOSTAWSKI & POLI, 1999).

2.3.1 Taxonomia de CANTÚ-PAZ

Segundo (CANTÚ-PAZ, 1997), a idéia básica dos algoritmos paralelos é dividir uma tarefa em pedaços e resolve-los simultaneamente usando múltiplos processadores. Essa abordagem de dividir para conquistar pode ser aplicada a Algoritmos Genéticos de várias maneiras. Alguns métodos de paralelização usam uma única população, enquanto outros dividem a população em diversas subpopulações relativamente isoladas. Alguns métodos podem explorar arquiteturas de computadores maciçamente paralelos enquanto outros são adequados a múltiplos computadores com maior poder de processamento.

A classificação definida por (CANTÚ-PAZ, 1997) é similar a outras (ADAMIDIS, 1994), (GORDON & WHITLEY, 1993) e (LIN et all, 1994), porém foi extendida para incluir outras categorias. Há três tipos principais de algoritmos genéticos paralelos:

- Algoritmo Genético Mestre-Escravo com população global

- Algoritmo Genético de Granularidade Fina
- Algoritmo Genético de Granularidade Grossa

2.3.1.1 Algoritmo Genético Mestre-Escravo com população global

Como em um algoritmo genético convencional, cada indivíduo pode competir e reproduzir com qualquer outro da população, isto é, a seleção e reprodução são realizadas globalmente. Algoritmos genéticos paralelos com população global são geralmente implementados como programas mestre-escravo, onde o processo mestre armazena a população e os processos escravos avaliam a aptidão dos indivíduos. Neste caso, a operação paralelizada é a avaliação dos indivíduos, pelo fato da avaliação da aptidão ser independente do restante da população e não haver necessidade de comunicação durante esta etapa.

A avaliação dos indivíduos é paralelizada pela associação de uma parte da população a cada um dos processadores disponíveis. A comunicação ocorre somente quando o processo escravo recebe seu conjunto de indivíduos para avaliar e quando deve retornar os valores de aptidão.

O algoritmo é dito *síncrono* quando deve aguardar o recebimento dos valores de toda a população para prosseguir com a próxima geração. O algoritmo genético mestre-escravo *síncrono* tem exatamente as mesmas propriedades do AG convencional, diferenciando apenas no incremento da velocidade de processamento das gerações.

O modelo de paralelização global não faz suposições sobre a arquitetura computacional a ser usada, podendo ser implementada eficientemente em computadores com memória compartilhada ou distribuída. Em um computador com memória compartilhada, a população é armazenada na memória compartilhada e cada processador lê seu comjunto de indivíduos e escreve os resultados da avaliação de volta sem qualquer conflito. Em computadores com memória distribuída, a população pode ser armazenada em um processador. Este processador "*mestre*" é responsável por enviar explicitamente os indivíduos para os outros processadores (escravos) para avaliação, coletando os resultados, e aplicando os operadores genéticos para produzir a próxima geração.

Em resumo, os algoritmos genéticos paralelos "*mestre-escravo*" são fáceis de implementar e apresentam-se como um método eficiente de paralelização quando a etapa de avaliação

consume muitos recursos computacionais. Além disso, o método tem a vantagem de não alterar o comportamento do AG, o que permite aplicar a teoria disponível para os AGs convencionais.

2.3.1.2 Algoritmo Genético Paralelo de Granularidade Fina

Os algoritmos genéticos paralelos de granularidade fina são um meio termo entre os algoritmos com população global (mestre-escravo) e os algoritmos com múltiplas populações isoladas.

Esse tipo de algoritmo pode ser visualizado de duas maneiras diferentes. Pode ser modelado como uma população simples com estrutura espacial que limita a interação entre os indivíduos, sendo que um indivíduo pode competir e reproduzir somente com seus vizinhos. Entretanto, a sobreposição da vizinhança permite que boas soluções sejam dissiminadas pela população inteira. Outra maneira é mostrar diversas subpopulações separadas e sobrepostas (Figura 4). Quando a seleção é realizada, somente cromossomos da mesma população podem reproduzir, porém, muitos cromossomos são parte de múltiplas populações permitindo que o material genético se espalhe pela população inteira.

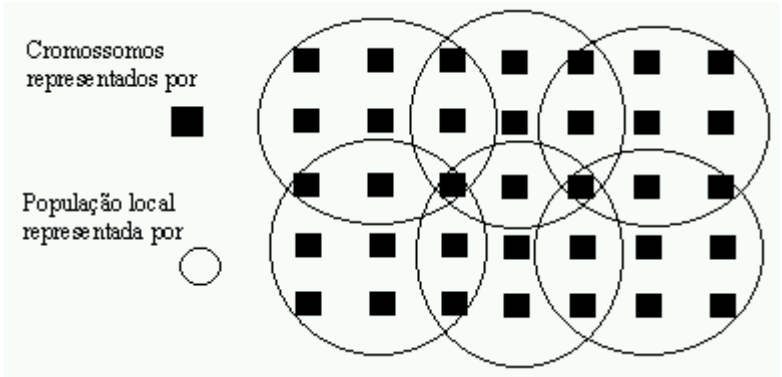


Figura 4 - Representação espacial de um AG de granularidade fina (STRACUZZI, 1998 – pg 7)

Na prática, algoritmos genéticos de granularidade fina são geralmente implementados em máquinas MIMD[1], sendo que cada cromossomo é alocado a um processador. A topologia da rede determina a vizinhança entre os cromossomos. O volume de comunicação exigido pelo modelo torna-o inviável para ambientes distribuídos.

[1] Segundo a classificação de Flynn

Esse modelo possui algumas vantagens. A representação é simples e evita alguns problemas relacionados à política de migração de cromossomos. A população local é separada suficientemente para prevenir que cromossomos mais aptos “dominem” a população. Esse modelo é adequado para execução em máquinas multiprocessadas. Toda interação é feita nas “margens” das populações, não sendo necessário o transporte de informações através da rede, o que reduz consideravelmente a sobrecarga de comunicação.

2.3.1.3 Algoritmo Genético Paralelo de Granularidade Grossa

Este modelo é conhecido também como *algoritmo genético distribuído*. Este modelo considera um conjunto de subpopulações (*demes*) que evoluem em paralelo e de forma independente (Figura 5). Periodicamente, as subpopulações trocam indivíduos entre si. Cada subpopulação evolui da mesma forma que num algoritmo genético convencional e a solução é o melhor indivíduo encontrado no conjunto das subpopulações.

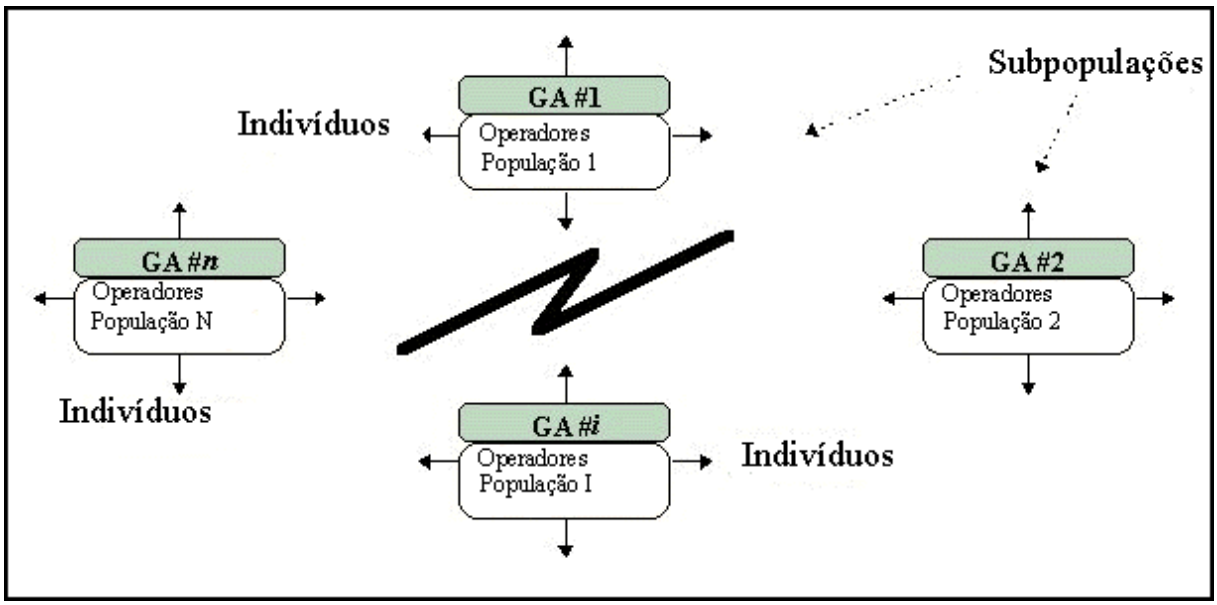


Figura 5 - Algoritmo genético de granularidade grossa (STRACUZZI, 1998 – pg 8)

A troca de indivíduos, denominada *migração*, permite a troca de material genético entre as subpopulações.

Os algoritmos genéticos distribuídos requerem a definição de alguns parâmetros adicionais. Um deles é a taxa de migração, que define a periodicidade com que os indivíduos migram. Outros parâmetros passam pela definição do número de indivíduos que migram por cada migração e qual o critério usado para decidir quais indivíduos devem migrar. O último parâmetro relaciona-se com a topologia e estabelece qual é a subpopulação de destino do indivíduo migrado.

2.3.2 Taxonomia de NOWOSTAWSKI & POLI

A taxonomia de (NOWOSTAWSKI & POLI, 1999) define oito classes:

- Paralelismo Mestre-Escravo (com avaliação de aptidão distribuída)
- Subpopulações estáticas com migração
- Subpopulações estáticas com sobreposição (sem migração)
- Algoritmos genéticos maciçamente paralelos
- Subpopulações dinâmicas (com sobreposição)
- Algoritmos genéticos paralelos Steady-State
- Algoritmos genéticos paralelos desorganizados
- Métodos híbridos

A Figura 6 ilustra o relacionamento entre estas classes de algoritmos.

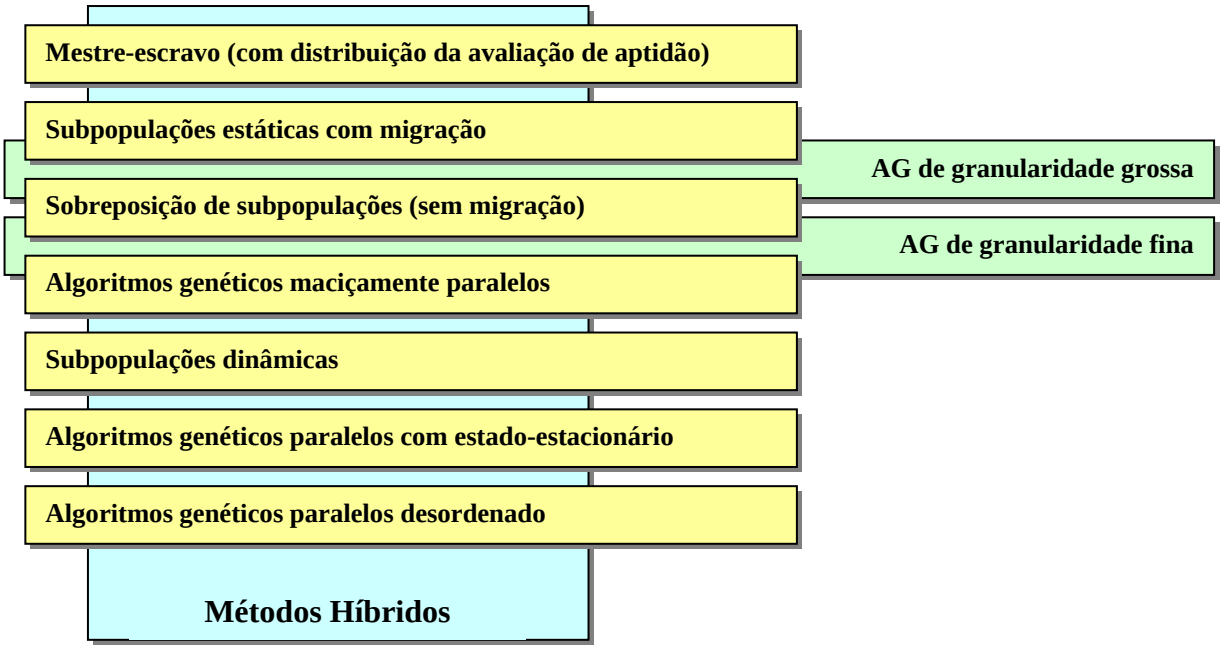


Figura 6 - Taxonomia de Algoritmos Genéticos Paralelos (NOWOSTAWSKI et. all, 1999 – pg 3)

2.3.2.1 Paralelismo Mestre-Escravo

Foi uma das primeiras aplicações de sucesso dos Algoritmos Genéticos Paralelos. Este método usa um modelo de população simples, sendo que a avaliação dos indivíduos e a aplicação dos operadores genéticos são executadas em paralelo. A seleção e a reprodução são realizadas globalmente, uma vez que cada cromossomo pode competir e reproduzir com qualquer outro da população.

A avaliação da aptidão dos indivíduos é facilmente paralelizada, pois precisa somente de informações do indivíduo a ser avaliado, não sendo necessário nenhum tipo de comunicação nesta fase. É geralmente implementado usando programas mestre-escravos, onde o mestre armazena a população e os escravos avaliam a aptidão, aplicam a mutação e as vezes realizam a operação de *crossover*.

Este algoritmo pode ser implementado de maneira síncrona ou assíncrona. O algoritmo é dito síncrono se o programa mestre interrompe seu processamento para aguardar o recebimento dos valores da aptidão de todo os indivíduos antes de proceder com a próxima geração. É basicamente um algoritmo genético convencional, exceto pelo aumento da velocidade de execução.

Na versão assíncrona deste algoritmo, o programa mestre não interrompe seu processamento para aguardar resultados produzidos por processadores mais lentos. A seleção dos indivíduos é feita com uma fração da população que já tenha sido processada, o que o diferencia de um algoritmo genético convencional.

2.3.2.2 Subpopulações estáticas com migração

Este algoritmo divide a população em subpopulações (demes) isoladas geograficamente e usa um operador de migração para efetuar a troca de indivíduos entre as subpopulações. De tempo em tempo, alguns indivíduos são movidos (copiados) de uma subpopulação para outra. Se os indivíduos podem ser movidos para qualquer subpopulação, o modelo é chamado *ilha*. No caso dos indivíduos migrarem somente para as subpopulações vizinhas, o modelo é chamado *degrau*.

A migração de indivíduos de uma subpopulação para outra é controlada por alguns parâmetros como:

a) a topologia que define as conexões entre as subpopulações. Topologias comumente usadas são: hipercubo, malha bidimensional e tri-dimensional, torus, etc.

b) a taxa de migração que controla quantos indivíduos devem migrar.

c) um esquema de migração que controla quais indivíduos de uma subpopulação (melhor, pior, randômico) migram para outra subpopulação, e quais indivíduos são trocados (pior, randômico, etc).

d) intervalo de migração que determina a freqüência das migrações.

Algoritmos de granularidade grossa é um termo geral usado para identificar um modelo com um número relativamente pequeno de subpopulações com muitos indivíduos. Estes modelos são caracterizados pelo longo período de tempo necessário para processar uma geração de indivíduos dentro de cada subpopulação e pela ocasional comunicação para troca de indivíduos. Este tipo de algoritmo é normalmente implementado em computadores MIMD com memória distribuída e também em redes heterogêneas.

Algoritmos de granularidade fina são o oposto. Requerem grande número de processadores pelo fato da população ser dividida em um grande número de demes com poucos indivíduos.

O modelo de múltiplas subpopulações, do ponto de vista da implementação, é uma simples extensão do algoritmo genético convencional rodando em máquinas paralelas, aplicando em tempos prédefinidos o operador de migração.

2.3.2.3 Sobreposição de subpopulações estáticas sem migração

Este modelo é semelhante ao anterior, sendo diferenciado pela ausência do operador de migração. A troca de indivíduos entre as populações é feita através de áreas de sobreposição, permitindo desta forma que um indivíduo pertença a mais de uma subpopulação. As subpopulações são organizadas em uma estrutura espacial que fornece a interação entre elas (exemplos de estruturas espaciais usadas, malha 2-D e malha 3-D, etc).

A implementação deste modelo pode ser feita facilmente em sistemas de memória compartilhada, uma vez que a população inteira pode ser colocada na área compartilhada. É necessário um esquema de sincronização para as áreas de sobreposição quando as operações de crossover, mutação e seleção forem aplicadas.

2.3.2.4 Algoritmos Genéticos Maciçamente Paralelos

O aumento do número de subpopulações no modelo de subpopulações estáticas com sobreposição, associado à redução do número de indivíduos em cada subpopulação resulta em uma categoria denominada Algoritmo Genético maciçamente paralelo (também chamado de AG de granularidade fina). Nesta categoria de algoritmo há uma única população, porém, existe uma estrutura espacial que limita a interação entre os indivíduos da população. Os indivíduos podem competir e casar somente com seus vizinhos.

Este modelo é adequado para execução em computadores maciçamente paralelos, podendo também ser simulado em um cluster de estações, com a desvantagem de envolver um alto custo de comunicação.

2.3.2.5 Subpopulações Dinâmicas

Este modelo é o resultado da combinação do algoritmo genético Mestre-Escravo distribuído (paralelismo global) com um AG de granularidade grossa (modelo com sobreposição de subpopulações). Neste modelo não há operador de migração, uma vez que a população inteira é tratada durante a evolução como uma única coleção de indivíduos e a troca de informações entre os indivíduos são feitas através da reorganização dinâmica das subpopulações durante os ciclos de processamento.

Do ponto de vista do processamento paralelo, este modelo ajusta-se perfeitamente na categoria MIMD como um algoritmo mestre-escravo múltiplo assíncrono.

O algoritmo pode ser executado em máquinas paralelas com memória compartilhada ou distribuída. Devido a sua escalabilidade, pode ser usado em sistemas computacionais com poucos elementos de processamento (EP) assim como em sistemas com um grande número de elementos de processamento.

2.3.2.6 Algoritmos Genéticos Paralelos Steady-State

Neste modelo, a paralelização dos operadores genéticos é direta, uma vez que o modelo usa esquemas de atualizações contínuas da população. Se os descendentes são gradativamente introduzidos em uma população simples e em constante desenvolvimento, a única atividade a fazer é aplicar a seleção e um esquema de troca na região crítica. Os outros operadores genéticos, incluindo a avaliação de aptidão, podem ser executados em paralelo.

2.3.2.7 Algoritmos Genéticos Paralelos desordenados

Este modelo de algoritmo é composto por três fases. Na fase de inicialização, a população inicial é gerada, sendo reduzida através de algum tipo de torneio de seleção na fase seguinte. Então na fase de justaposição, as soluções parciais encontradas na fase anterior são mescladas.

A fase inicial pode ser realizada paralelamente, uma vez que não há necessidade de comunicação. Já a fase de seleção e avaliação necessita da maior parte do tempo de execução, sendo necessário algum tipo de concorrência para obter-se bom desempenho.

2.3.2.8 Algoritmos Genéticos Híbridos

Na paralelização dos AG é possível também usar alguma combinação dos modelos descritos anteriormente. Este nova categoria de AG é chamada de *algoritmos híbridos*.

2.3.3 Tamanho da população

O tamanho da população é um parâmetro de grande importância para qualquer Algoritmo Genético. Sua escolha afeta a qualidade da solução final bem como o tempo de processamento (HARIK et all, 1997).

Uma população pequena apresenta uma diversidade genética limitada, em termos de diversidade dos seus elementos constituintes, proporcionado uma menor cobertura do espaço de soluções, podendo resultar em soluções subótimas.

Por outro lado, o incremento no tamanho da população proporciona um aumento da probabilidade de produzir melhores soluções, através da maior cobertura do espaço de soluções e prevenindo a convergência prematura, embora à custa de um maior esforço computacional.

2.3.4 Política de Migração

A divisão explícita da população nos algoritmos genéticos de granularidade grossa levanta diversas questões relacionadas a política de migração de indivíduos. A política de migração mais simples é não migrar indivíduos. O resultado é um conjunto de algoritmos genéticos convencionais independentes, ou ilhas isoladas. Apesar desta política ser muito simples de implementar, não há outra vantagem em relação aos algoritmos genéticos convencionais além da ampliação do espaço de busca (TANESE, 1989).

No modelo denominado algoritmo genético paralelo síncrono, as populações evoluem na mesma velocidade e realizam as trocas no mesmo tempo. Um critério global para o início da migração é estabelecido e cada vez que este critério for satisfeito, todas as populações enviam

e recebem um grupo de indivíduos migrantes. Em ambientes distribuídos, uma máquina mais rápida pode desenvolver a população e ser forçada a aguardar as máquinas mais lentas atingirem o critério estabelecido. Os ambientes distribuídos podem ser explorados com mais eficiência através do emprego de uma política de migração assíncrona, ou seja, cada população decide independentemente quando os indivíduos devem ser trocados

Os algoritmos genéticos paralelos assíncronos, por outro lado, levantam uma questão importante. Uma população com indivíduos de baixa qualidade será dominada quando indivíduos de alta qualidade são inseridos por um nó mais evoluído, ou a população será capaz de incorporar esses novos indivíduos e encontrar o ótimo global? Freqüentemente a resposta para esta questão é que a população torna-se dominada e converge prematuramente, ou os indivíduos de alta qualidade são ignorados, pois são incompatíveis com o restante da população local.

Outro aspecto importante da política de migração é a clonagem. Quando os indivíduos são selecionados para migrar, estes podem ser removidos da população de origem e copiados para a população de destino, ou uma cópia dele é enviada para a população de destino.

2.3.5 Topologia de interconexão

Um aspecto dos algoritmos genéticos paralelos, muitas vezes desconsiderado, é a topologia de interconexão entre as subpopulações. A topologia é um fator importante na eficiência do algoritmo genético paralelo, pois determina a velocidade de propagação de uma boa solução entre as subpopulações. Se a topologia for densamente conectada, boas soluções se propagarão rapidamente para todas as subpopulações e podem dominá-la. Por outro lado, se a topologia for esparsamente conectada, as soluções se propagarão mais lentamente permitindo o surgimento de diferentes soluções (STRACUZZI, 1998).

2.3.6 Estrutura dos nós

Existem alguns métodos para estruturação das populações em um AG de granularidade grossa. O método mais simples e amplamente utilizado é o de estruturação homogênea, no qual todas as populações recebem o mesmo conjunto de parâmetros para coordenar sua evolução. As únicas diferenças entre as populações são o conjunto de cromossomos e a velocidade de processamento da máquina. Todas as outras variáveis, tais como - taxa de *crossover*, taxa de *mutação*, tamanho da população, intervalo de migração, etc – permanecem constantes em todas as populações (STRACUZZI, 1998).

Outro método possível é aquele em que cada população tem seus próprios parâmetros. Essa estrutura heterogênea permite que diferentes populações explorem diferentes áreas do espaço de busca. Esse método aumenta também a chance de um conjunto de parâmetros ideal ser usado na solução do problema. A solução de muitos problemas está intimamente ligada a escolha correta dos parâmetros do AG, apesar de ser algo extremamente difícil (CANTÚ-PAZ, 1997).

3 Computação Distribuída com JavaSpaces

Este capítulo aborda as características da tecnologia JavaSpaces usada como mecanismo de de distribuição do algoritmo genético paralelo.

3.1 Introdução

Atualmente, as redes de computadores tornaram-se vitais para o mundo da computação, mudando a maneira que as pessoas usam o computador e, é claro, a maneira como os projetistas desenvolvem suas aplicações. É nesse contexto que surge a computação distribuída (ZAPATA, 2000).

A computação distribuída é um tipo de computação na qual as aplicações são compostas por um conjunto de processos que estão distribuídos através da rede, cooperando para atingir um objetivo comum. Entretanto, a aplicação pode ser vista como uma coleção de componentes e objetos que podem ser alocados em diferentes computadores conectados a uma rede (COULOURIS et all, 2001).

Segundo (VAN STEEN, M. & TANENBAUM, A, 2002), a computação distribuída possui as seguintes vantagens:

- É possível conseguir maior desempenho apenas adicionando novos computadores no sistema onde a aplicação esteja rodando, isto é, a aplicação é facilmente escalável.
- Desde que os processos rodando em computadores diferentes estão se comunicando uns com os outros, estes podem compartilhar seus recursos.
- Além disso, a tolerância a falhas é mais facilmente obtida pois se um processo falhar ou um computador ficar off-line, o resto do sistema continua executando.

Aplicações distribuídas são mais difíceis de projetar e implementar do que aplicações *standalone*. Esta complexidade adicional é, na maioria dos casos, devido a grande variedade de plataformas de hardware e software.

Entretanto, esta não é a única dificuldade que deve ser considerada quando são projetadas e implementadas aplicações distribuídas. Os processos de uma aplicação distribuída precisam se comunicar para conseguirem trabalhar em conjunto. Uma vez que a comunicação se dá sobre uma rede, os tempos de transmissão são geralmente altos quando comparados a velocidade do processador dos computadores envolvidos. O tempo de transmissão de uma origem ao seu destino é denominado *latência*, e deve ser levado em conta no projeto de aplicações distribuídas (FREEMAN et all, 1999).

Outra dificuldade existente no projeto de aplicações distribuídas é a sincronização entre os processos, pois na maioria das vezes um processo tem que esperar por alguma informação de outro processo para prosseguir sua execução (FREEMAN et all, 1999).

A tolerância a falhas é outro ponto crucial no projeto de aplicação distribuídas, pois deve-se manter um estado global consistente no caso de falhar parciais (FREEMAN et all, 1999).

No intuito de se evitar estes tipos de problemas, os desenvolvedores de aplicações distribuídas podem fazer uso do JavaSpaces. A tecnologia JavaSpaces é uma ferramenta simples e expressiva para construção de sistemas distribuídos. Ela reduz o esforço de projeto e o montante de código necessário para criar aplicações colaborativas e distribuídas, pois usa um modelo de programação que cuida dos problemas herdados da computação distribuída em benefício dos desenvolvedores. O modelo usado pela tecnologia JavaSpaces foi concebido

como uma coleção de processos comunicando-se através do envio e da retirada de objetos de um ou mais espaços (ZAPATA, 2000)

Entretanto, o desenvolvimento de aplicações baseadas em espaço passa pelo projeto de estruturas de dados distribuídas e de protocolos distribuídos que irão operar sobre elas. Uma estrutura de dados distribuída é apenas um conjunto de objetos que estão armazenados em um ou mais espaços. Os protocolos distribuídos especificam de que forma os processos da aplicação compartilham e modificam as estruturas de dados de maneira coordenada (FREEMAN et all, 1999).

Essa nova abordagem que o modelo de programação JavaSpaces proporciona ao desenvolvedor, torna a construção de aplicações distribuídas mais fácil, mesmo tratando-se de assuntos mais complexos como por exemplo, balanceamento de carga. Além disso, como JavaSpaces é construído sobre a linguagem Java, os desenvolvedores pode tirar proveito de todas as vantagens que essa linguagem de programação provê.

3.2 JavaSpaces

JavaSpaces é uma especificação desenvolvida pela Sun Microsystems para o paradigma de programação baseada em espaço. A especificação desenvolvida é baseada na linguagem Java e totalmente orientada a objetos. A implementação de referência da Sun foi desenvolvida como um serviço dentro da arquitetura de computação distribuída *Jini*[2] (JAVASPACES SPECIFICATION, 2000).

A tecnologia JavaSpaces provê um modelo de programação fundamentalmente diferente. Este modelo visualiza a aplicação como uma coleção de processos cooperando através de um fluxo de objetos para dentro e fora de um ou mais espaços (JAVASPACES SPECIFICATION, 2001).

[2] *Jini*™ é uma arquitetura de rede aberta que habilita desenvolvedores criar serviços centrados na rede – implementados em hardware ou software – que são altamente adaptáveis a mudanças.

O espaço é um repositório de objetos compartilhado. Os processos usam o repositório como um mecanismo de troca e armazenamento persistente de objetos. Ao invés dos processos comunicarem diretamente, eles se coordenam trocando objetos através dos espaços (Figura 7).

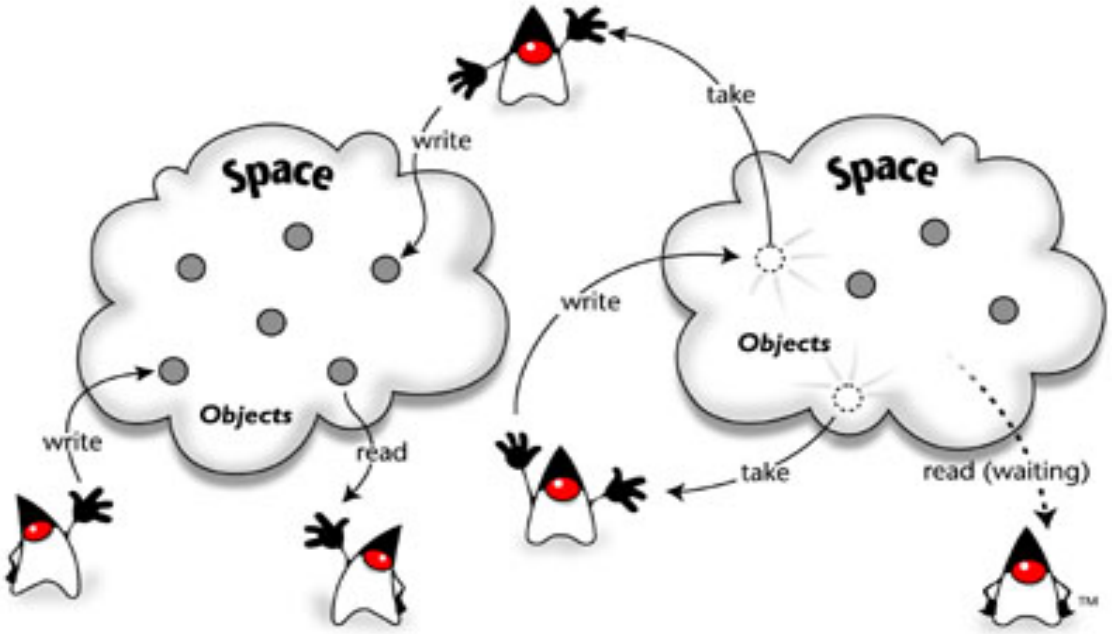


Figura 7 - JavaSpaces (http://java.sun.com/products/javaspaces – pg 1)

Os processos fazem operações simples, como escrever (write) novos objetos no espaço, retirar (take) objetos do espaço ou ler (read – fazer uma cópia) objetos do espaço.

Para ler ou retirar objetos do espaço, os processos procuram o objeto desejado a partir de um padrão (template). Se o objeto não é encontrado imediatamente, o processo pode esperar até que o objeto desejado seja colocado no espaço.

Processos não modificam os objetos que estão no espaço e nem podem chamar seus métodos diretamente. Para modificar um objeto, o processo deve removê-lo do espaço, atualizar seus atributos (estado) e inseri-lo no espaço novamente como um novo objeto.

3.2.1 Origem do JavaSpaces

JavaSpaces é um novo modelo para computação distribuída inspirada nas pesquisas desenvolvidas pelo professor David Gelenter da Universidade de Yale.

(GELENTER, 1985) desenvolveu uma ferramenta denominada *Linda* para a construção de aplicações distribuídas. *Linda* consistia de um pequeno número de operações combinadas com um armazenamento persistente denominado *tuple space*.

O resultado foi surpreendente. O uso de um meio de armazenamento de objetos juntamente com um pequeno número de operações simples, permitiu implementar facilmente uma grande classe de problemas paralelos e distribuídos.

3.2.2 Características da tecnologia JavaSpaces

A lista abaixo fornece algumas características associadas ao JavaSpaces.

- **Espaços são compartilhados** - Espaços são "memórias compartilhadas" acessíveis através da rede, onde vários processos remotos podem interagir concorrentemente.

- **Espaços são persistentes** - Espaços provêm armazenamento confiável para os objetos. Uma vez que o objeto é colocado no espaço, ele permanecerá lá até que um processo o remova explicitamente.

- **Espaços são associativos** - Os objetos de um espaço são localizados através de busca associativa, ao invés de um endereço de memória ou de um identificador. A busca associativa provê um modo simples de localizar objetos de acordo com o seu conteúdo, sem precisar saber como o objeto é chamado, quem o possui, quem o criou, ou onde ele está armazenado.

- **Transações em espaços são seguras** - A tecnologia JavaSpaces provê um modelo de transação, que assegura que uma operação no espaço é atômica.

- **Espaços permitem a troca de conteúdo** - Enquanto o objeto está no espaço, ele é apenas um dado passivo. Entretanto, quando o objeto é copiado ou retirado do espaço, uma cópia local do objeto é criada, possibilitando então o acesso aos seus campos privados e a invocação de seus métodos.

3.2.3 Vantagens da tecnologia JavaSpaces

A tecnologia JavaSpaces apresenta as seguintes vantagens:

- **É simples** - A tecnologia não requer conhecimento de uma interface de programação complexa, consistindo de um conjunto de operações simples.

- **É expressiva** - Usando um pequeno conjunto de operações, pode-se construir aplicações distribuídas sem escrever muito código.

- **Suporta protocolos de baixo acoplamento** - Uma vez que remententes e receptores são desacoplados, espaços suportam protocolos simples, flexíveis e confiáveis. O desacoplamento facilita a composição da grandes aplicações, suporta análise global e aumentam o reuso de software.

- **Facilita a tarefa de escrever sistemas cliente/servidor** - Quando se projeta o servidor, características como acesso concorrente de múltiplos clientes, persistência e transação são reinventados a cada vez. A tecnologia JavaSpaces provê estas funcionalidades.

3.3 Acesso ao espaço

O espaço pode ser acessado, basicamente, de duas maneiras diferentes. Um espaço pode ser registrado como um *Jini lookup service*[3] ou pode ser registrado com um *RMI registry*[4].

[3] *Jini lookup service* é o mecanismo central de um sistema baseado no Jini, o qual permite que serviços efetuem seu registro para que sejam acessíveis (conhecidos) pelos clientes do sistema

[4] *RMI registry* é um serviço de registro de nomes. Os servidores de objetos disponibilizam seus métodos para chamada remota através da ligação destes à um nome no *RMI Registry*.

Para tornar esta tarefa mais fácil, pode-se usar uma classe que usa *Jini* ou *RMI* para localizar espaços:

```
public class Espaco {
        public static JavaSpace buscaEspaco(String nome){
                try {
                   if (System.getSecurityManager()==null) {
                      System.setSecurityManager(new RMISecurityManager());
                   }
                   if (System.getProperty("com.sun.jini.use.registry")==null){
                         Locator locator = new com.sun.jini.outrigger.DiscoveryLocator();
                         Finder finder = new com.sun.jini.outrigger.LookupFinder();
                         return (JavaSpace)finder.find(locator, nome);
                   }
                   else {
                         RefHolder rh = (RefHolder)Naming.lookup(nome);
                         return (JavaSpace)rh.proxy();
                   }
                }
                catch (Exception e) {
                         System.err.println (e.getMessage());
                }
                return null;
        }

        public static JavaSpace buscaEspaco(){
                return buscaEspaco ("JavaSpaces");
        }
}
```

O método estático *buscaEspaco* retorna um objeto que implementa a interface **JavaSpace**. Este método pode ter como parâmetro uma String, que é o nome do espaço que será acessado.

Primeiramente instancia-se o SecurityManager, que neste caso é um RMISecurityManager. Em seguida acessa a propriedade *com.sun.jini.use.registry*. Esta propriedade determina se deve ser usado *Jini* ou *RMI* para localizar o espaço. Se a propriedade tiver valor *null*, usa-se o *Jini lookup service*, caso contrário, usa-se o *RMI registry*.

Se for usado o *Jini lookup service*, deve-se usar duas classes do pacote *com.suj.jini.outrigger*. Esse pacote encapsula o código necessário para fazer buscas. A classe *DiscoveryLocator* sabe como localizar um *Jini lookup service*. A classe *LookupFinder* tem o método *finder*, que tem como parâmetros um *Locator* e o nome do serviço, e retorna uma interface para o serviço. No caso do serviço JavaSpace, esta interface é um proxy local que implementa a interface JavaSpace e usa um protocolo privado para se comunicar como o espaço remoto.

Se for usado *RMI registry*, deve-se chamar o método estático *Naming.lookup* do pacote *java.rmi* para procurar o nome no registro. Se for encontrado, uma referência para o objeto

remoto é retornada. No caso da implementação da Sun MicroSystem, esta referência é do tipo *com.sun.jini.mahout.binder.RefHolder*, e não um proxy do espaço. O método proxy do objeto *RefHolder* retorna o proxy local do espaço.

3.4 Entradas (Entrys)

Aplicações baseadas em espaços têm como ponto comum as entradas (Entry). Através da troca de entradas, os processos podem se comunicar, sincronizar e coordenar suas atividades. Uma entrada é apenas um objeto que segue algumas convenções. Estas convenções garantem a segurança do seu trânsito através dos espaços (JAVASPACES SPECIFICATION, 2001).

Uma entrada é definida da seguinte maneira:

```
IMPORT NET.JINI.CORE.ENTRY.*;
public class IndividuoEntry implements Entry
{
   //construtor sem argumentos
   public IndividuoEntry () {}
`
```

Para que uma classe seja uma entrada, ela deve implementar a interface Entry. Também pode-se criar um entrada estendendo uma classe que já implementa a interface Entry. Uma entrada precisa ter um construtor sem argumentos.

A definição da interface Entry é (JAVASPACES SPECIFICATION, 2001):

public interface *Entry* **extends** *java.io.Serializable* { }

A interface Entry é vazia, não possui métodos a serem implementados. Interfaces vazias são usadas como marcadores. A interface Entry marca uma classe que é apropriada ao uso em espaços.

A classe que representa a entrada pode ter qualquer número de campos e métodos.

```
public class IndividuoEntry implements Entry {
        public Integer geracao;
        public String populacao;

        //construtor sem argumentos
        public IndividuoEntry () {}

        public IndividuoEntry (String populacao, int geracao){
                this.populacao = populacao;
                this.geracao    = new Integer(geracao);
        }
}
```

Observe que os campos são públicos, o que vai contra as noções de encapsulamento. Isto é necessário, porque a busca de objetos no espaço é associativa. Declarando os campos das entradas como públicos, permite-se que outros processos encontrem as entradas através dos valores dos seus campos.

Outra observação importante, é que os campos de uma entrada devem conter referências para objetos e não tipos primitivos. Para incluir tipos primitivos numa entrada, deve-se usar as classes correspondentes (Integer, Boolean, Float, Double).

Se uma entrada tem um conjunto de métodos públicos, qualquer processo, que leia ou retire essa entrada do espaço, será capaz de invocar esses métodos.

3.4.1 Adição de entradas no espaço

Entradas são escritas no espaço através da chamada ao método *write* do proxy local (seção 3.3). Se uma entrada não é escrita no espaço, ela permanece como um objeto local e não pode ser usada por outros processos.

A sintaxe do método *write* é:

- **Lease** write (**Entry** e, **Transaction** t, **long** lease) **throws** RemoteException, TransactionException;

O método *write* tem três argumentos: uma *entry*, uma *transaction* e um *lease*.

O parâmetro *Entry* é o objeto que será colocado no espaço. O parâmetro *Transaction* indica se a operação no espaço é feita sob uma transação ativa. Se o parâmetro for setado para *null*, significa que a transação tem uma única operação. O parâmetro *lease* especifica o tempo que a entrada permanecerá no espaço antes de ser removida. Isto permite que o espaço faça sua própria versão de coleta de lixo (*garbage collection*).

Uma vez que uma cópia do objeto foi colocada no espaço, não podemos invocar seus métodos ou examinar o valor de seus campos, sem primeiro obter uma cópia do objeto(entrada) do espaço.

Alguns modelos de objetos distribuídos armazenam objetos ativos remotamente, e seus métodos podem ser chamados por processos ativos. Neste modelo, os objetos do espaço são objetos passivos.

3.4.2 Obtenção de entradas do espaço

A interface JavaSpace provê duas versões para a operação de leitura de entradas com o método *read*.

- **Entry** read (**Entry** tmpl, **Transaction** t, **long** timeout) **throws** TransactionException, UnusableEntryException, RemoteException, InterruptedException;
- **Entry** readIfExists (**Entry** tmpl, **Transaction** t, **long** timeout) **throws** TransactionException,UnusableEntryException,RemoteException,InterruptedException;

Ambos os métodos têm os mesmos parâmetros do método *write*. Eles procuram e retornam uma entrada do espaço que coincida com o template, se tal entrada existir. Se houverem várias entradas que coincidam com o template, apenas uma será retornada. A escolha é arbitrária.

O método *read* é uma operação bloqueante. Se uma entrada não for encontrada, o método espera por um *timeout* até que a entrada seja encontrada. Dois valores para *timeout* tem significado especial: **Long.MAX_VALUE** indica espera indefinida, enquanto que **JavaSpaces.NO_WAIT** significa retorno imediato. Se o método retornar sem achar a entrada, o valor *null* é retornado pelo método.

O método *readIfExists* é uma operação não bloqueante. Retorna imediatamente se a entrada não é encontrada. O parâmetro *timeout* só é levado em conta quando a operação ocorre sob uma transação.

Assim como *read*, o método *take* tem duas formas:

- **Entry** take (**Entry** tmpl, **Transaction** t, **long** timeout) **throws** TransactionException, UnusableEntryException, RemoteException, InterruptedException;
- **Entry** takeIfExists (**Entry** tmpl, **Transaction** t, **long** timeout) **throws** TransactionException, UnusableEntryException, RemoteException,InterruptedException;

Ambos os métodos funcionam como os correspondentes na operação de leitura. A diferença é que estes métodos removem a entrada do espaço. Se retornarem valores não nulos, é garantido que a entrada foi retirada do espaço, e nenhuma outra operação no espaço retornará a mesma entrada.

3.4.2.1 Busca associativa

A busca de entradas no espaço é feita através do uso de *templates*. Um *template* é uma entrada que especifica o conteúdo do tipo da entrada que deve ser encontrada. Para criar um *template*, é necessário instanciar uma entrada e especificar o valor de alguns campos ou deixar seu valor como *null*.

Um *template* coincide com uma entrada se duas regras forem verdadeiras:

- O tipo do *template* é do mesmo tipo da entrada, ou é um supertipo da entrada. Todo campo do *template* coincide com o campo correspondente da entrada.
- Se o campo do *template* tiver valor *null*, então ele coincide com o campo correspondente da entrada. Se o campo do *template* tem valor diferente de *null*, então ele coincide com o campo correspondente da entrada se os dois têm o mesmo valor.

3.4.2.2 Campos primitivos

As entradas devem conter campos que sejam referências para objetos, e não tipos primitivos. A razão para isto é a busca de entradas no espaço. É necessária uma forma de indicar que um campo possa ter qualquer valor. Isto é feito atribuindo *null* ao campo correspondente do padrão (*template*).

No caso de objetos, a escolha do valor *null* é óbvia. Mas, e no caso dos tipos primitivos, qual deveria ser o valor? Assim, os projetistas da tecnologia JavaSpaces decidiram que todos os campos das entradas deveriam ser referências para objetos.

3.5 Serialização de Entradas

Cada objeto JavaSpace é um objeto local que atua como um meio de acesso à implementação remota do espaço, e não como uma referência ao objeto remoto. Como resultado, todas as operações no espaço seguem através do proxy intermediário, e em seguida no espaço remoto.

Quando uma entrada é escrita no espaço, primeiro o proxy serializa seus campos e então transmite para o espaço. Enquanto está no espaço, uma entrada é armazenada na sua forma serializada. Quando obtem-se uma entrada do espaço, ela é transmitida de volta para o proxy na forma serializada. Os campos são deserializados numa entrada antes que ela seja retornada pelos métodos *read* ou *take*.

O modo como o método *write* serializa uma entrada é um pouco diferente da serialização normal. Quando o método *write* serializa uma entrada, o proxy local primeiro escreve a informação da classe na *stream*, serializa cada campo público e escreve sua serialização na *stream*. Cada campo público é serializado independentemente dos outros. Campos privados não são serializados na stream.

Este tipo de serialização tem alguns inconvenientes: se dois campos de uma entrada são referências para o mesmo objeto, a forma serializada terá duas cópias do mesmo objeto. Quando a entrada for retornada pelos métodos *read* ou *take*, a entry estará estruturada

diferentemente de quando foi escrita. Os dois campos que referenciavam o mesmo objeto, agora referenciam objetos diferentes.

Se uma entrada tiver campos privados, eles não serão serializados e seus valores são perdidos.

3.6 Padrões de Aplicações baseadas em espaço

Esta seção introduz os principais padrões de aplicações que são usados na programação baseada em espaço.

3.6.1 O padrão Mestre-Escravo

Também conhecido como padrão "Replicated-Worker". Um processo mestre coloca entradas no espaço que representam uma ou mais tarefas. Um ou mais escravos buscam essas entradas, executam as tarefas e então coloca os resultados de volta no espaço (FREEMAN et all, 1999)..

O padrão Mestre-Escravo suporta facilmente o problema da decomposição. Por exemplo:

- Um mestre deposita a tarefa no espaço
- Um escravo/mestre troca a tarefa por duas outras subtarefas
- Um escravo completa uma subtarefa e envia o resultado para o espaço
- Outro escravo/mestre troca a segunda subtarefa por outras sub-subtarefas
- E assim por diante.

A taxa de computação/comunicação indica se este padrão é apropriado. Se uma baixa taxa não puder ser melhorada pelo ajuste do montante de computação entre as sucessivas comunicações, uma abordagem monoprocessada pode ser mais eficiente.

3.6.2 O padrão Command

Este padrão é similiar ao Mestre-Escravo, porém com maior ênfase na transmissão de comportamento, e não apenas dados.

Pelo fato dos objetos serem transmitidos com dados e código juntos, os escravos não precisam ter conhecimento do conteúdo, o que os torna simples executores de tarefas.

3.6.3 O padrão Blackboard

Este padrão de aplicação especialista difere do Mestre-Escravo pelo fato de cada escravo fornecer um único serviço. Um grupo de especialistas observa o espaço e então adiciona, apaga ou atualiza operações até que a solução do problema seja encontrada.

4 O Método Desenvolvido

Este capítulo apresenta, primeiramente, o problema do caixeiro viajante – PCV – usado como corpo de prova ao algoritmo proposto. Em seguida, apresenta o relato de alguns trabalhos relacionados e a descrição do mecanismo de distribuição desenvolvido para dar suporte à paralelização do algoritmo genético, bem como o funcionamento do protótipo implementado.

4.1 O Problema do Caixeiro Viajante

O problema do caixeiro viajante (PCV) ou TSP (*Traveling Salesman Problem*), pode ser enunciado como: *dado um número finito de cidades junto com o custo de viagem entre cada par delas, achar o caminho mais curto para visitar todas as cidades exatamente uma vez e voltando ao ponto de partida*

Em termos da teoria dos grafos, as cidades são representadas pelos vértices e os custos das viagens são pesos associados aos arcos que determinam o grafo. O grafo assim constituído é um grafo completo[5] finito. O problema passa então a ser: encontrar um ciclo de Hamilton[6]

[5] Um grafo G é dito completo se para todo vértice V de G existe um arco partindo de V para todo e qualquer outro vértice V' pertencente a G

[6]Um ciclo de Hamilton consiste em um caminho cíclico que passa por todos os vértices do grafo, passando uma única vez em cada vértice

com peso total mínimo. Assim o PCV pode ser visto como uma versão em linguagem não matemática do problema de encontrar um ciclo de Hamilton em um grafo completo, cujo peso seja mínimo.

A Figura 8 apresenta uma instância do PCV com cinco cidades ABCDE, representada pelo grafo G(*V*,*E*) no qual cada vértice *V* representa uma cidade e cada arco ***e*** representa um caminho entre um par de cidades.

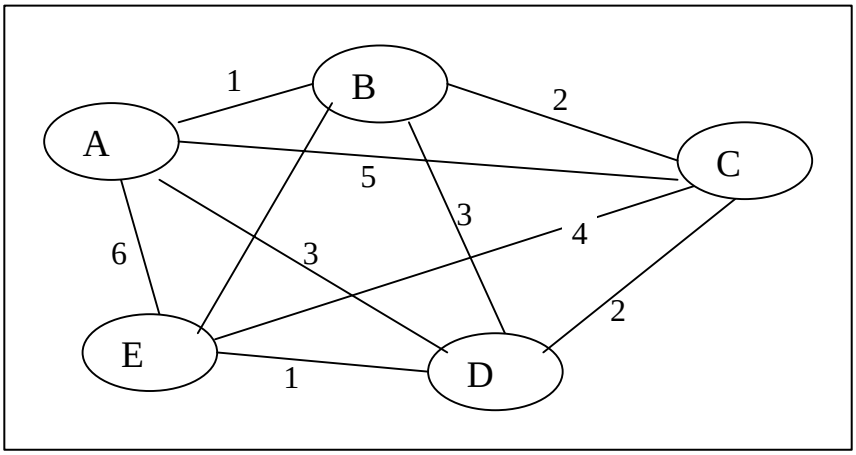


Figura 8 - Uma instância do PCV representada na forma de grafo (DALLE MOLE, 2002 –pg 23)

Encontrar a solução para o PCV consiste em encontrar o menor caminho passando por todas as cidades uma única vez e retornando ao ponto de partida. Um exemplo de viajem seria ABCDEA com custo total 12, onde o custo é a soma dos pesos dos arcos em seqüência. Outro caminho poderia ser ABCEDA com custo 11.

Neste trabalho foi assumido que os custos de viagem entre duas cidades A e B são simétricos, de tal modo que viajando da cidade A para cidade B vale o mesmo que viajando de B até A. A bateria de testes foi realizada com instâncias publicas coletadas na internet[7], tendo sido escolhidas as instancias: **BERLIN52, KROC100, TS225, PCB442, GR606** e **PR1002**. A escolha foi aleatória tomando-se apenas a precaução de escolher instancias de tamanhos crescentes.

[7] Site - http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/ disponível em 29-04-2002.

4.2 Trabalhos Relacionados

As seções a seguir (4.2.1 e 4.2.2) apresentam um relato de trabalhos relacionados a pesquisa atual.

4.2.1 Algoritmo de DALLE MOLE

O presente trabalho é uma continuação do trabalho de (DALLE MOLE, 2002), sendo que o algoritmo genético base usado aqui foi cedido pelo mesmo. Para um melhor entendimento deste, recomenda-se anteriormente uma consulta ao referido trabalho.

(DALLE MOLE, 2002) desenvolveu uma abordagem paralela para algoritmos genéticos. O foco principal estava na estrutura de paralelização, ou seja, o modelo de algoritmo genético paralelo.

O algoritmo genético paralelo foi modelado segundo o conceito de populações cooperantes, coordenadas por um nó central. Nesse modelo, cada máquina é responsável por processar uma população com N indivíduos onde a troca de informações, entre populações, se realiza através da migração de indivíduos. Os indivíduos foram modelados como unidades autônomas com a responsabilidade da aplicação dos operadores de *crossover* e mutação. Cada indivíduo é também responsável pela determinação da sua aptidão. Este enfoque torna simples a exploração do paralelismo encontrado nos sistemas multiprocessados.

A hipótese defendida por (DALLE MOLE, 2002), é a de que um conjunto de populações cooperantes, executando em paralelo, equivale a uma grande população. Para avaliar o desempenho desta nova abordagem, (DALLE MOLE, 2002) realizou testes utilizando um protótipo construído para simular o ambiente distribuído e o paralelismo dos sistemas multiprocessados, através de programação concorrente (threads). Como corpo de prova, foi utilizado um dos mais conhecidos *benchmarks* na área de otimização, o problema do caixeiro viajante. Os resultados obtidos estão descritos no referido trabalho.

4.2.2 Trabalho de ZORMAN

(ZORMAN et all, 2002) exploraram os benefícios e as desvantagens do uso da tecnologia Jini e JavaSpaces no desenvolvimento de um Algoritmo Genético Distribuído. O problema da mochila (*knapsack problem*) foi usado como corpo de prova do Algoritmo Genético Distribuído.

Como resultado, (ZORMAN et all, 2002) concluíram que as tecnologias *Jini* e *JavaSpaces* facilitam a distribuição do algoritmo genético. Com relação à performance da implementação, observaram que a medida que o tamanho da instância aumentava havia uma redução na taxa de computação/comunicação, e a máquina remota que servia como espaço ficava sobrecarregada.

A sobrecarga observada é decorrência da elevada taxa de migração usada na execução do Algoritmo Genético Distribuído, 30% da população a cada 30 gerações segundo (ZORMAN et all, 2002).

Na tentativa de atenuar esta sobrecarga, de aumentar a taxa de computação/comunicação e de melhorar a performance, (ZORMAN et all, 2002) propuseram uma implementação baseada em múltiplos espaços. Nos testes realizados, (ZORMAN et all, 2002) observaram melhorias nos resultados obtidos com a nova implementação.

4.3 Estrutura do Algoritmo Genético Paralelo

A estrutura do algoritmo implementado segue basicamente a proposta de (DALLE MOLE, 2002). Como descrito em (DALLE MOLE, 2002), um nó supervisor cria as populações com um conjunto de genes codificados e um conjunto de parâmetros iniciais (número máximo e mínimo de indivíduos, operadores de *crossover* e mutação, periodicidade de envio e aceite de indivíduos migrantes, desvio padrão a ser considerado como mínimo na detecção de estagnação). O nó supervisor serve também como intermediário na migração de indivíduos entre as populações. O supervisor armazena os indivíduos migrantes até serem requisitados

por outra população que não a de origem, ou até serem substituídos por novos migrantes mais aptos. Apenas um migrante de cada população é mantido.

Na nova implementação, o nó supervisor continua existindo, porém seu papel foi modificado para adequar-se ao modelo de distribuição adotado (seção 4.3.1). As diversas subpopulações são executadas paralelamente em cada máquina da rede e, periodicamente, enviam um percentual de seus indivíduos para o espaço. Da mesma forma, é feita uma busca no espaço para obter indivíduos de outras populações.

4.3.1 O Mecanismo de Distribuição

O mecanismo de distribuição adotado por esta implementação está baseado na tecnologia JavaSpaces da Sun MicroSystems. Dentre os fatores que motivaram a escolha do JavaSpaces como meio de distribuição do AG pode-se destacar:

1. O algoritmo genético base foi desenvolvido com a linguagem de programação Java.

2. A estrutura do algoritmo genético paralelo proposto por (DALLE MOLE, 2002) enquadra-se perfeitamente no modelo de computação distribuída Mestre/Escravo, cuja implementação torna-se natural com JavaSpaces.

A Figura 9 ilustra como é feita a distribuição das populações e a migração de indivíduos entre elas.

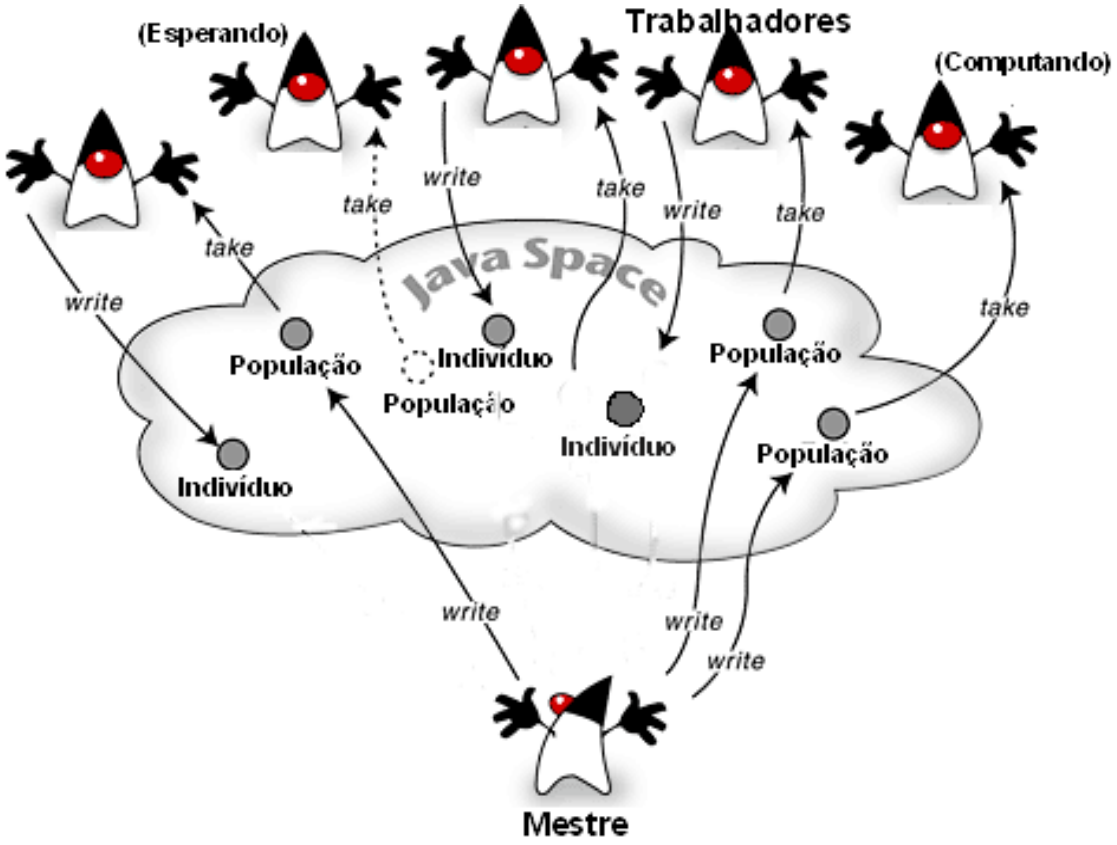


Figura 9 - Distribuição das populações e dos indivíduos migrantes

O nó supervisor (mestre) cria as populações e as envia para o espaço. A criação (randômica) dos indivíduos que compõem a população é feita no momento em que a população for executada na estação da rede. Desta maneira, o nó supervisor apenas cria uma entrada no espaço representando a população, juntamente com os parâmetros do AG usados na execução. A Figura 10 ilustra o fluxo de execução do algoritmo.

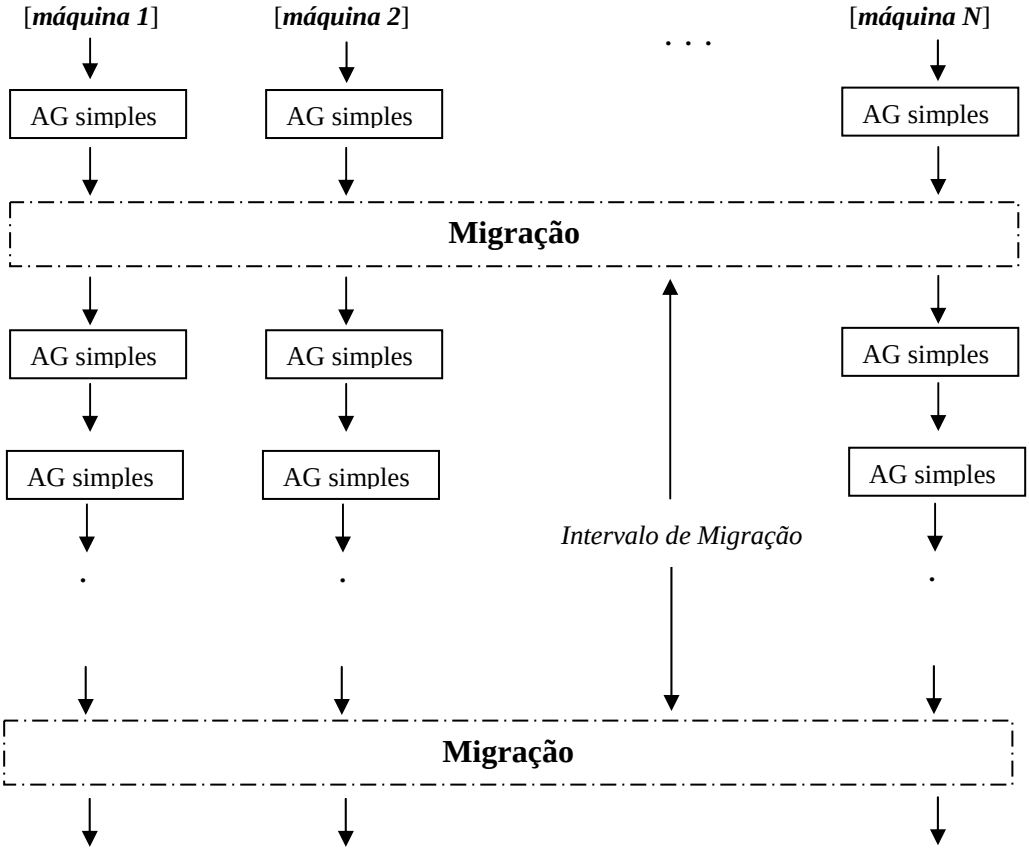


Figura 10 – Fluxo de execução do algoritmo genético paralelo

4.3.2 Política de Migração de Indivíduos

Dentre os parâmetros passados pelo nó supervisor para a população, estão a taxa (percentual) de migração e o intervalo de migração.

Quando o intervalo de migração é atingido, a população seleciona aleatoriamente o percentual de indivíduos indicado pelo parâmetro taxa de migração, e os envia para o espaço. O indivíduo é enviado para o espaço somente algumas informações, como a população de origem, sua codificação da solução (genes) e sua aptidão (fitness).

Da mesma forma, a população retira do espaço o percentual de indivíduos indicado pela taxa de migração e os insere na população. Os indivíduos provenientes da própria população são descartados.

4.3.3 Exploração do Paralelismo

A estrutura proposta por (DALLE MOLE, 2002) permite a exploração do paralelismo existente em cada máquina, no caso de máquinas multiprocessadas, e do paralelismo oferecido pelos ambientes distribuídos, no caso das redes de computadores. Na implementação realizada por (DALLE MOLE, 2002), o paralelismo do ambiente distribuído foi simulado através do uso de *threads*.

A estrutura deste Algoritmo Genético explora efetivamente o paralelismo do ambiente distribuído através do uso da tecnologia JavaSpaces.

4.4 Implementação do Protótipo

O protótipo implementado neste trabalho é uma extensão do algoritmo genético desenvolvido por (DALLE MOLE, 2002). A partir deste, desenvolveu-se o mecanismo que permite a distribuição da execução do AG numa rede de computadores. A distribuição do AG foi viabilizada através do uso do JavaSpaces. Nesta implementação, foram usadas as implementações de referência da Sun MicroSystems dos serviços necessários para o funcionamento do protótipo: serviço *HTTP*, serviço *Jini lookup* e o serviço *JavaSpaces*.

O protótipo foi submetido a uma bateria de testes com instâncias públicas do Problema do Caixeiro Viajante. Na bateria de testes foram usadas as instâncias *Berlin52, Kroc100, Ts225, Pcb442, Gr666* e *Pr1002*.

Os equipamentos utilizados nos testes foram 25 microcomputadores Pentium IV 1.4 GHZ com 128 MB RAM. A infraestrutura de comunicação utilizada foi uma rede ETHERNET 100 Mbits.

4.4.1 Estrutura do Protótipo

A estrutura do protótipo segue, basicamente, aquela proposta por (DALLE MOLE, 2002), com adição e alterações de algumas classes. As adições e alterações estão descritas nos itens a seguir.

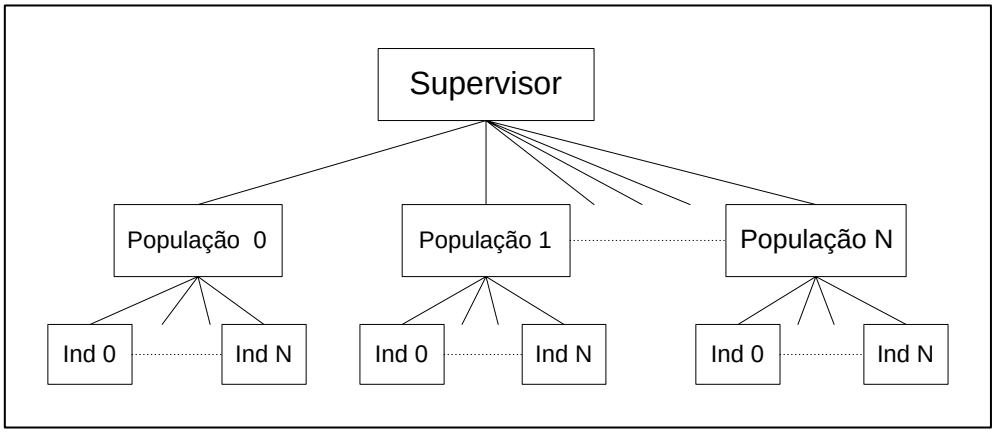


Figura 11 – Hierarquia de Comando (DALLE MOLE, 2002 – pg 64)

4.4.1.1 Classe *Supervisor*

A classe Supervisor foi alterada para atender o mecanismo de distribuição adotado. O método construtor da classe localiza o espaço (através da classe Espaço), cria as populações e as envia para o espaço.

Os métodos *insertReprodutor* e *getReprodutor* propostos na implementação de (DALLE MOLE, 2002) foram desativados, tendo em vista que as populações não terão mais uma referência ao Supervisor para efetivar a troca de indivíduos. Como descrito na seção 4.3.1, a troca de indivíduos dar-se-á através do espaço, o que evita um forte acoplamento entre as classes Supervisor e Population.

4.4.1.2 Classe *Population*

Como descrito na seção 4.4.1.1, as populações são enviadas para o espaço para que sejam recebidas e executadas pelas diversas máquinas da rede. Para que isto seja possível, a classe Population precisa implementar a interface Entry, conforme descrito em 3.3.

Os métodos *enviaReprodutor* e *buscaReprodutor* propostos na implementação de (DALLE MOLE, 2002) foram retirados, dando lugar a dois outros métodos: *migraIndivíduos* e *recebeIndivíduos*. Para que a migração de indivíduos entre as populações seja efetivada, estas devem utilizar o espaço como mecanismo de troca de material genético. Ambos os métodos colocam e retiram do espaço a quantidade de indivíduos indicada pelo parâmetro *Taxa de Migração* da população.

4.4.1.3 Classes *Indivíduo* e *IndivíduoEntry*

Na proposta de (DALLE MOLE, 2002), os indivíduos são entidades autônomas, implementadas através de *threads*. Essa implementação impossibilitou que a classe *Indivíduo* fosse usada como entrada e pudesse ser enviada ao espaço, pelo fato que a linguagem Java não permite a serialização de *threads* (FREEMAN et all, 1999).

Para superar essa limitação desenvolveu-se a classe *IndivíduoEntry*, a qual serve como entrada do espaço para efetivar a migração de indivíduos entre as populações. Essa classe possui os elementos básicos necessários para a troca de material genético entre as populações.

A classe *IndivíduoEntry* é composta por quatro atributos e um método construtor. Os atributos são:

- *cromossomos*: é um vetor de números inteiros que contêm a codificação da solução (no caso do caixeiro viajante, ordem das cidades).
- *população*: é o nome da população a qual o indivíduo pertencia. Esse atributo é usado para evitar que, durante a migração, a população receba indivíduos originados da própria população.
- *geração*: é um número inteiro que indica em que geração o indivíduo foi gerado.

- *fitness*: indica a qualidade da solução codificada no atributo *cromossomos*.

Na classe *Indivíduo* foram acrescentados dois métodos:

- *codifica()*: esse método recebe como parâmetro um vetor de referências aos genes que compõem a solução do indivíduo e, como resultado, retorna um objeto da classe *IndivíduoEntry* com a solução codificada em um vetor de inteiros.

- *decodifica()*: esse método faz o inverso, recebe um objeto da classe *IndivíduoEntry* e retorna um objeto da classe *Indivíduo* com a solução codificada novamente em um vetor de referências aos genes.

4.4.1.4 Classe *Trabalhador*

A classe *Trabalhador* foi criada para ser executada nas diversas máquinas da rede. Suas funções são:

- Obter acesso ao espaço (através da classe *Espaço*)

- Buscar uma população

- Executar a população invocando o método *Run* (inicializa a algoritmo genético)

A partir daí, a própria população é responsável pelo envio e recebimento de indivíduos do espaço.

4.4.1.5 Classe *Espaço*

A classe *Espaço* foi adicionada para facilitar o acesso ao espaço por outras classes. A classe possui dois métodos:

- *buscaEspaço (String nome)* : esse método localiza na rede o serviço JavaSpaces identificado pelo nome passado como parâmetro.

- *buscaEspaço ()* : esse método localiza na rede o serviço JavaSpaces identificado pelo nome padrão. No momento da inicialização do espaço é fornecido o nome do serviço. O nome padrão é JavaSpaces.

5 Análise dos Resultados

Este capítulo apresenta os resultados obtidos pelo algoritmo proposto, sendo que os mesmos encontram-se resumidos e tabelados por instâncias do PCV. Para cada instância é apresentada uma tabela e 4 gráficos mostrando o desempenho do algoritmo para variadas taxas de migração.

A tabela apresentada em cada instância detalha os parâmetros usados nos testes, sendo eles: o número de populações (uma para cada máquina da rede), o número de indivíduos em cada população, o intervalo de migração (freqüência de migração, em gerações), a taxa de migração (percentual de indivíduos migrantes), o desvio padrão de estagnação, a solução conhecida (dada pelo benchmark), a solução obtida e o número de gerações processadas até a estagnação completa das populações.

O algoritmo genético proposto e implementado foi testado em 6 instâncias do PCV (*Berlin52, Kroc100, Ts225, Pcb442, Gr666* e *Pr1002*) e os resultados são apresentados na seqüência do trabalho. Para avaliar a qualidade das soluções obtidas pelo algoritmo, estas foram comparadas com as soluções ótimas dada por *benchmarks* disponíveis na Internet[8].

[8] http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/ disponível em 07-08-2002

5.1 Instância BERLIN52

Para esta instância, a solução ótima foi alcançada com 140 gerações, sendo que o algoritmo foi executado com 25 populações de 500 indivíduos, como ilustrado na Tabela 1.

Berlin52 – 52 cidades							
Populações	**Indivíduos**	**Intervalo de migração (em gerações)**	**Taxa de Migração**	**Desvio Padrão Estagnação**	**Solução conhecida**	**Solução Obtida**	**Número de Gerações**
25	**500**	**50**	**5%**	**15**	**7544,3659**	**7544,3659**	**200**
25	**500**	**50**	**15%**	**15**	**7544,3659**	**7544,3659**	**140**
25	**500**	**50**	**30%**	**15**	**7544,3659**	**7544,3659**	**160**

Tabela 1 – Parâmetros e resultados da instância BERLIN52

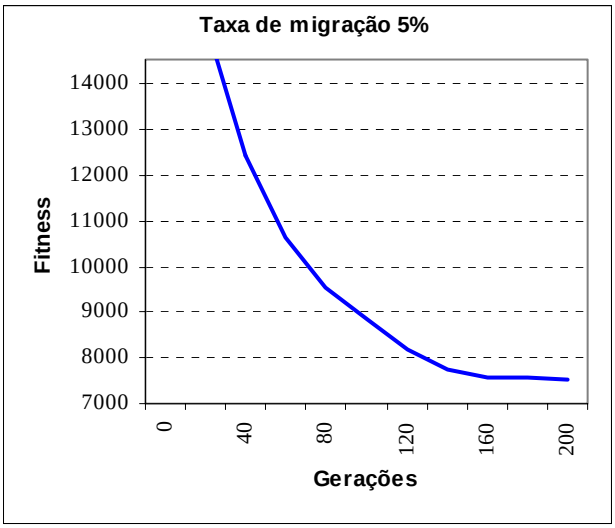


Gráfico 1 - Berlin52 - Taxa de migração 5%

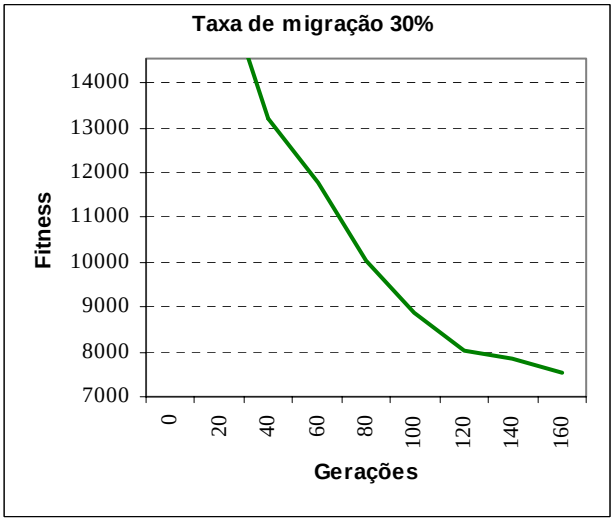
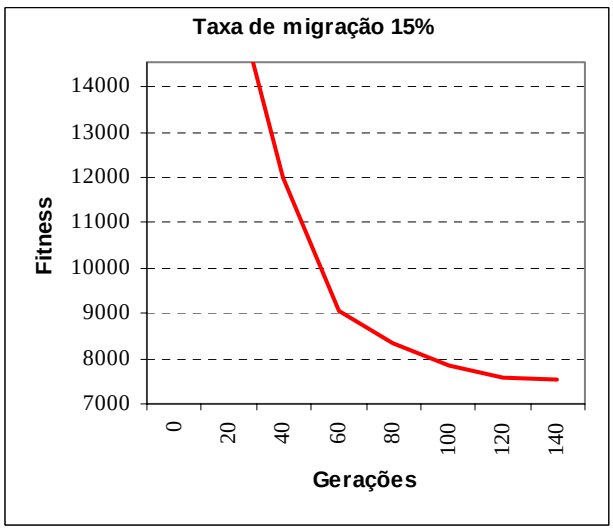


Gráfico 2 - Berlin52 - Taxa de migração 15%



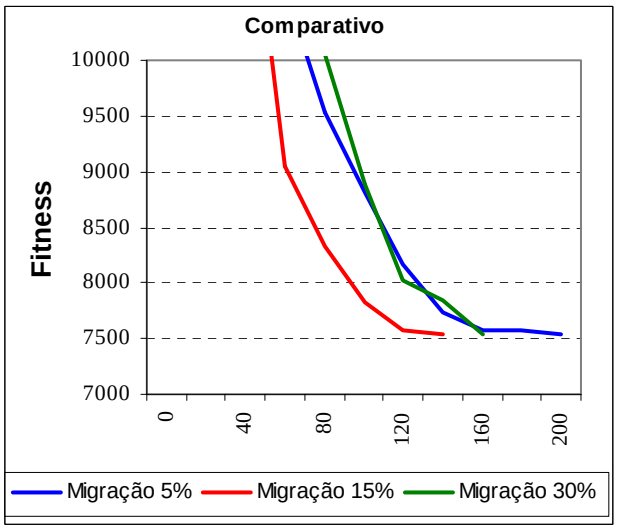


Gráfico 3 - Berlin52 - Taxa de migração 30%

Gráfico 4 - Berlin52 - Comparativo da curva de convergência

5.2 Instância KROC100

Nos testes realizados com esta instância, a solução ótima conhecida foi alcançada após 620 gerações, sendo que o algoritmo foi executado com 25 populações de 500 indivíduos, como ilustrado na Tabela 2.

Kroc100 – 100 cidades							
Populações	**Indivíduos**	**Intervalo de migração (em gerações)**	**Taxa de Migração**	**Desvio Padrão Estagnação**	**Solução conhecida**	**Solução Obtida**	**Número de Gerações**
25	**500**	**50**	**5%**	**20**	**20750,762**	**20750,762**	**620**
25	**500**	**50**	**15%**	**20**	**20750,762**	**20750,762**	**460**
25	**500**	**50**	**30%**	**20**	**20750,762**	**20852,278**	**520**

Tabela 2 - Parâmetros e resultados da instância KROC100

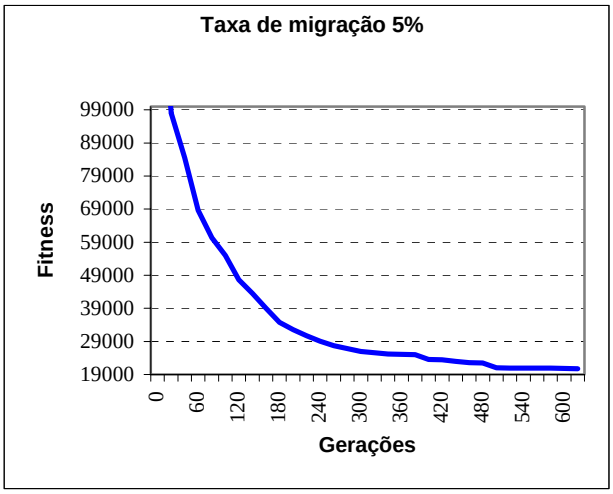


Gráfico 5 - Kroc100 - Taxa de migração 5%

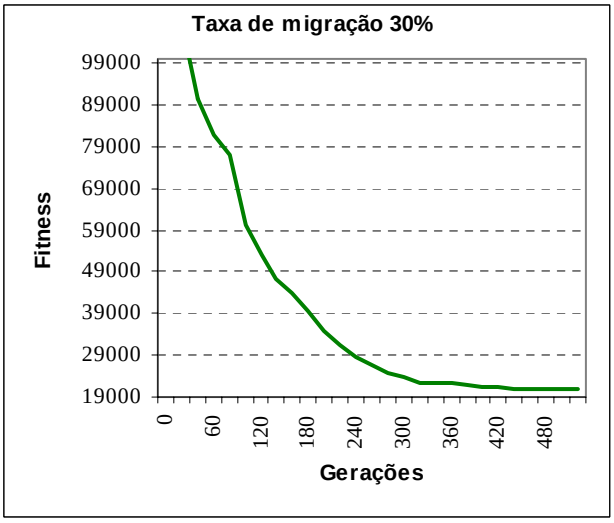


Gráfico 7 - Kroc100 - Taxa de migração 30%

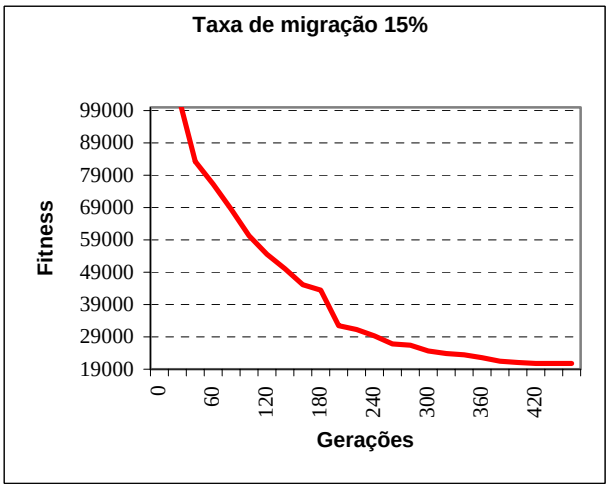


Gráfico 6 - Kroc100 - Taxa de migração 15%

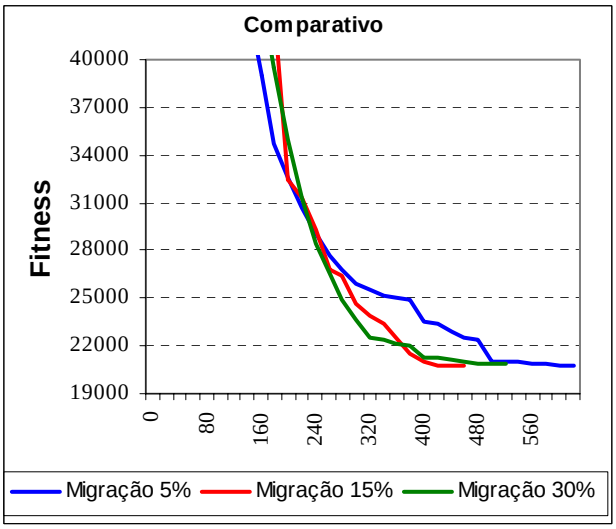


Gráfico 8 - Kroc100 - Comparativo da curva de convergência

5.3 Instância TS225

Para esta instância, a solução ótima foi alcançada após 1750 gerações. Os dados referentes a este teste estão relacionados na Tabela 3.

Ts225 – 225 cidades							
Populações	**Indivíduos**	**Intervalo de migração (em gerações)**	**Taxa de Migração**	**Desvio Padrão Estagnação**	**Solução conhecida**	**Solução Obtida**	**Número de Gerações**
25	**500**	**50**	**5%**	**50**	**126643**	**126643**	**1750**
25	**500**	**50**	**15%**	**50**	**126643**	**129580,61**	**2440**
25	**500**	**50**	**30%**	**50**	**126643**	**131750,69**	**4180**

Tabela 3 - Parâmetros e resultados da instância TS225

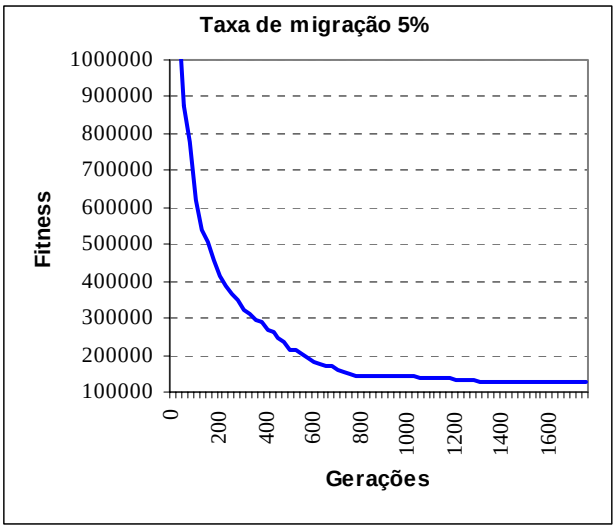


Gráfico 9 - Ts225 - Taxa de migração 5%

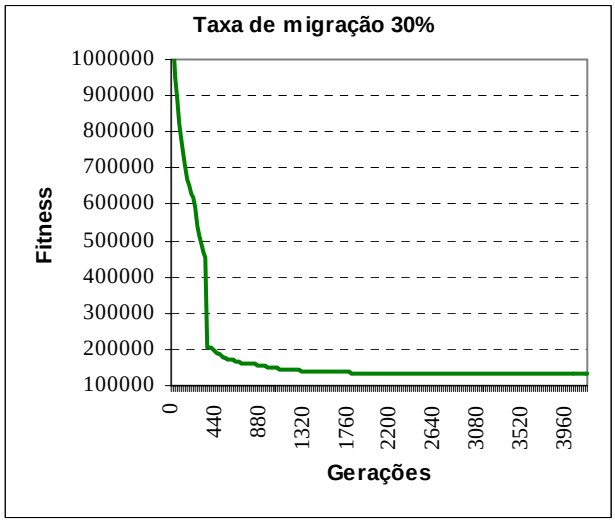


Gráfico 11 – Ts225 - Taxa de migração 30%

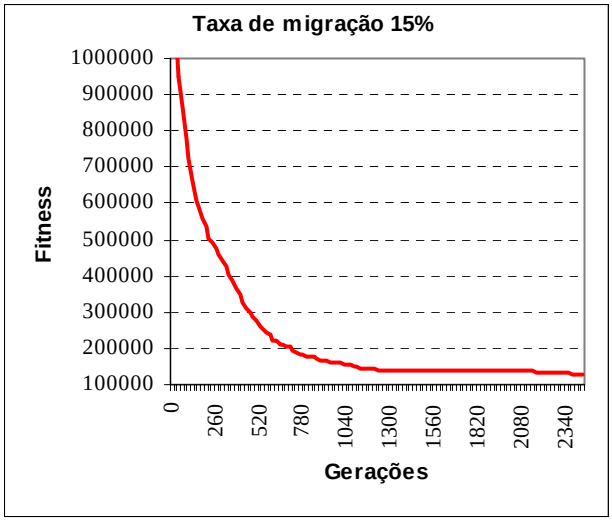


Gráfico 10 - Ts225 - Taxa de migração 15%

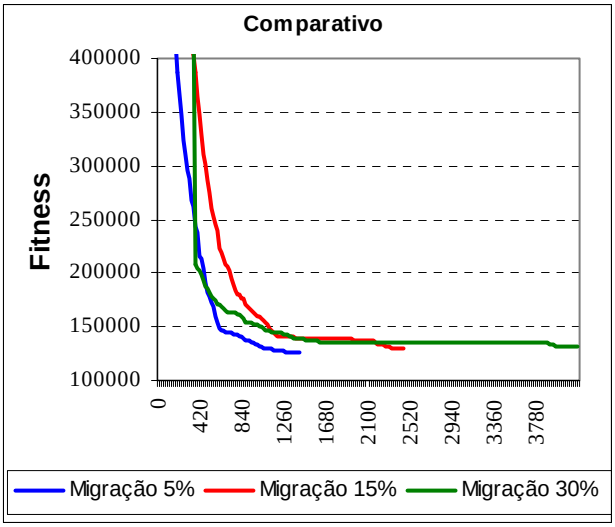


Gráfico 12 – Ts225 - Comparativo da curva de convergência

5.4 Instância PCB442

Com esta instância, obteve-se apenas uma aproximação da solução ótima conhecida. Após 7250 gerações ocorreu a estagnação das populações.

Ts225 – 225 cidades							
Populações	**Indivíduos**	**Intervalo de migração (em gerações)**	**Taxa de Migração**	**Desvio Padrão Estagnação**	**Solução conhecida**	**Solução Obtida**	**Número de Gerações**
25	**500**	**50**	**5%**	**100**	**50783,547**	**65424,88**	**4480**
25	**500**	**50**	**15%**	**100**	**50783,547**	**66429,36**	**4420**
25	**500**	**50**	**30%**	**100**	**50783,547**	**63270,37**	**4320**

Tabela 4 - Parâmetros e resultados da instância PCB442

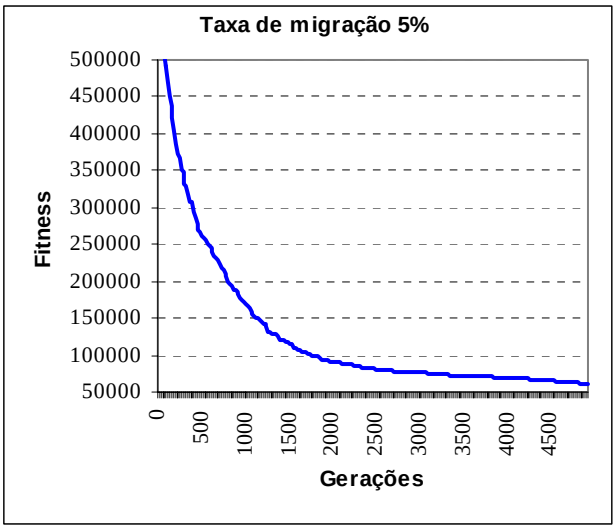


Gráfico 13 - Pcb442 - Taxa de migração 5%

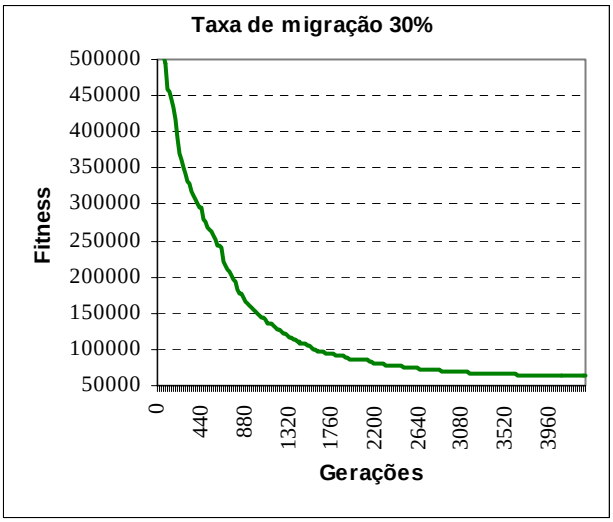


Gráfico 15 – Pcb442 - Taxa de migração 30%

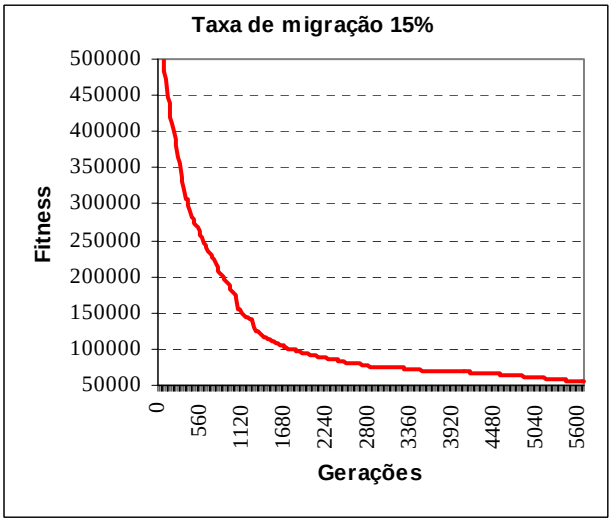


Gráfico 14 - Pcb442 - Taxa de migração 15%

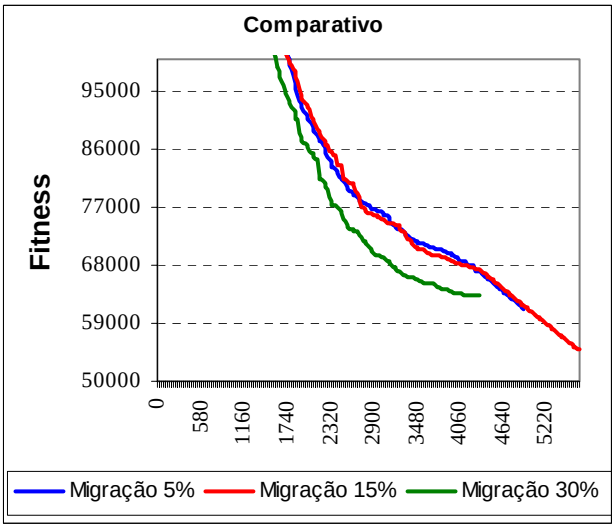


Gráfico 16 – Pcb442 - Comparativo da curva de convergência

5.5 Instância GR666

Para esta instância, a solução ótima conhecida foi superada. Observou-se que a solução conhecida não é de fato a melhor solução, pois seu percurso apresenta linhas cruzadas (indicativo que existe um caminho mais curto).

Gr666 – 666 cidades							
Populações	**Indivíduos**	**Intervalo de migração (em gerações)**	**Taxa de Migração**	**Desvio Padrão Estagnação**	**Solução conhecida**	**Solução Obtida**	**Número de Gerações**
25	**500**	**50**	**5%**	**100**	**3952,53**	**4104,48**	**20970**
25	**500**	**50**	**15%**	**100**	**3952,53**	**3699,73**	**5920**
25	**500**	**50**	**30%**	**100**	**3952,53**	**3948,98**	**5740**

Tabela 5 - Parâmetros e resultados da instância GR666

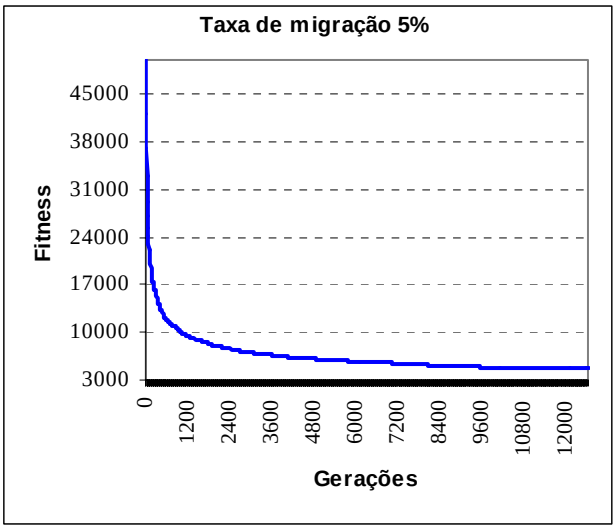


Gráfico 17 - Gr666 - Taxa de migração 5%

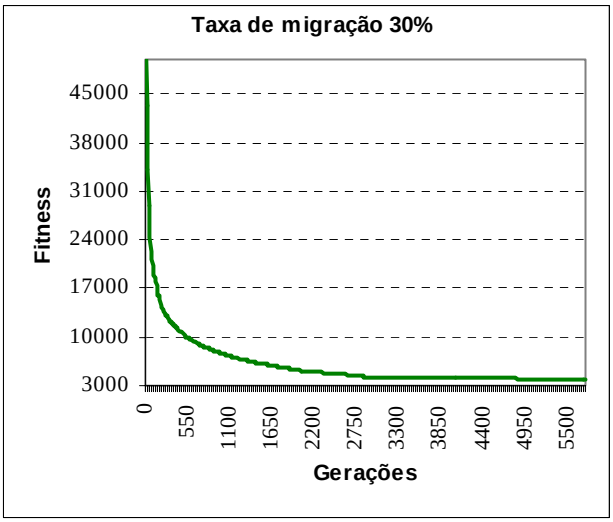


Gráfico 19 – Gr666 - Taxa de migração 30%

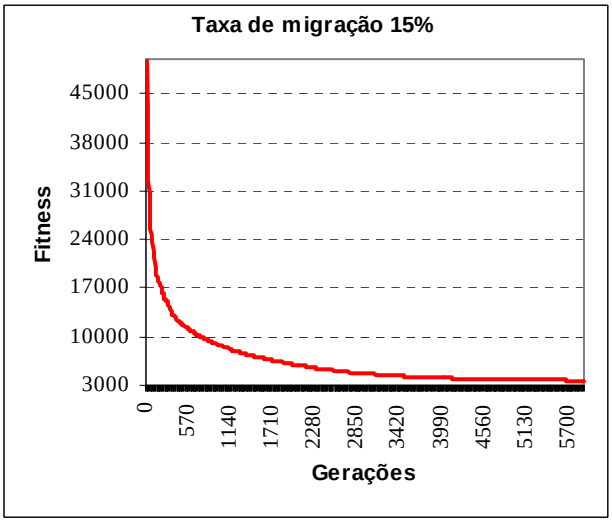


Gráfico 18 - Gr666 - Taxa de migração 15%

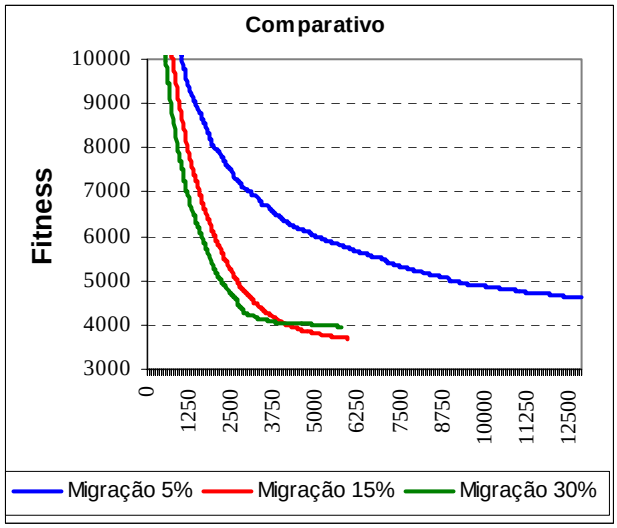


Gráfico 20 – Gr666 - Comparativo da curva de convergência

5.6 Instância PR1002

Nos testes realizados com esta instância não foi possível alcançar a solução ótima conhecida, obtendo-se apenas uma aproximação razoável. A Tabela 6 ilustra os parâmetros usados e o resultado obtido.

Pr1002 – 1002 cidades							
Populações	**Indivíduos**	**Intervalo de migração (em gerações)**	**Taxa de Migração**	**Desvio Padrão Estagnação**	**Solução conhecida**	**Solução Obtida**	**Número de Gerações**
25	**500**	**50**	**5%**	**100**	**259066,66**	**449286,09**	**8550**
25	**500**	**50**	**15%**	**100**	**259066,66**	**368405,78**	**10050**
25	**500**	**50**	**30%**	**100**	**259066,66**	**477256,66**	**6850**

Tabela 6 - Parâmetros e resultados da instância PR1002

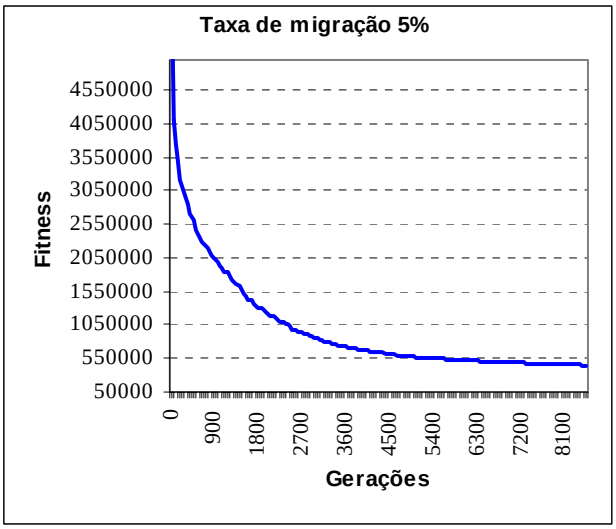


Gráfico 21 - Pr1002 - Taxa de migração 5%

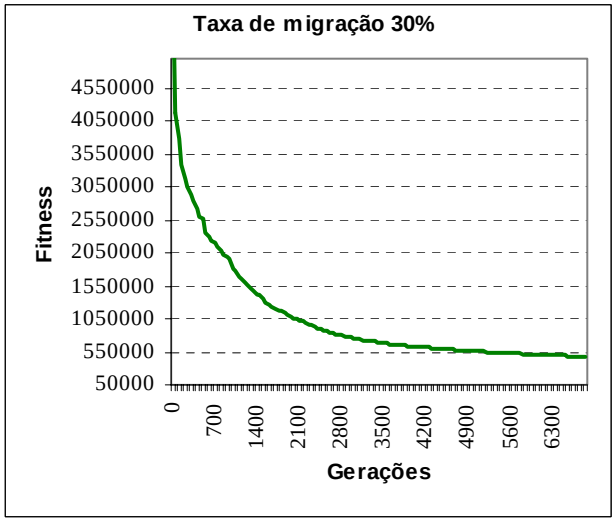


Gráfico 23 – Pr1002 - Taxa de migração30%

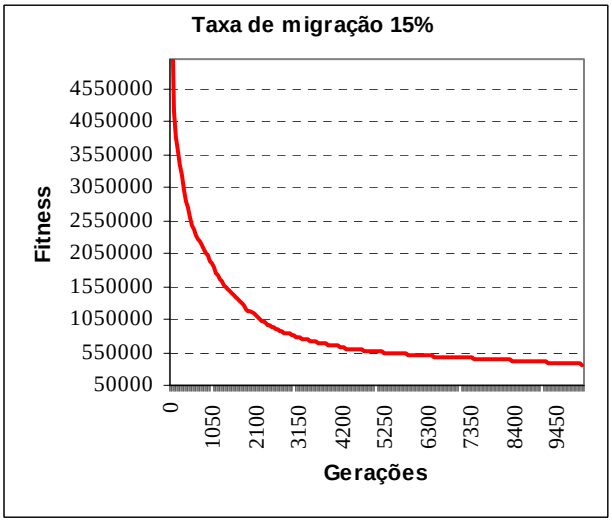


Gráfico 22 - Pr1002 - Taxa de migração 15%

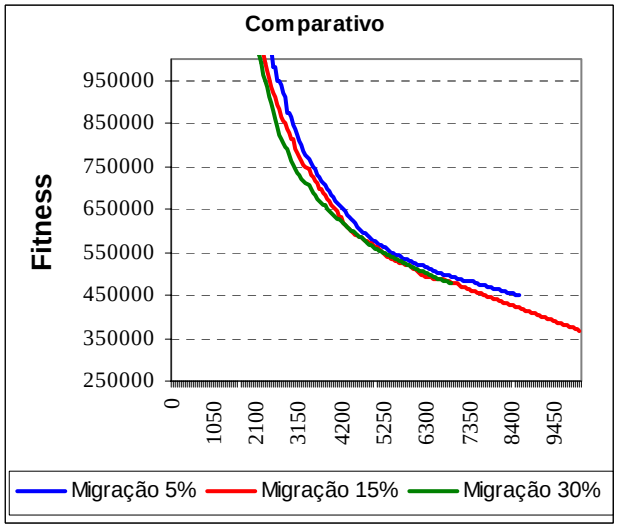


Gráfico 24 – Pr1002 - Comparativo da curva de convergência

5.7 Discussão dos Resultados

No total foram testadas 6 instâncias do PCV (*Berlin52, Kroc100, Ts225, Pcb442, Gr666 e Pr1002*). Como demonstrado nos gráficos e nas tabelas acima, a solução dada como ótima no *benchmark* foi alcançada em 4 instâncias (*Berlin52, Kroc100, Ts225, e Gr666*), sendo que na instância *Gr666* a solução ótima do *benchmark* foi superada.

No caso das instâncias *Pcb442* e *Pr1002*, obteve-se apenas uma aproximação da solução ótima.

Nos testes realizados, foram usados os mesmos parâmetros para todas as instâncias. A taxa e o intervalo de migração foram setados empiricamente em 5% e 50 gerações, respectivamente, objetivando maximizar a independência das populações e manter a diversidade genética de cada população.

Para as instâncias menores (*Berlin52 e Kroc100*), observou-se que a grande diversidade genética das populações (*500 indivíduos em cada população*) fez com que a curva de convergência atingisse, em poucas gerações, a solução ótima dada pelo *benchmark*. A medida que o tamanho das instâncias aumentou, o ciclo evolutivo das populações foi mais longo até que a convergência fosse alcançada. Em algumas instâncias, as populações convergiram para um ótimo local.

Observou-se também que as populações estagnadas conseguiam, após o recebimento de indivíduos migrantes, continuar seu ciclo evolutivo por mais algumas gerações. Essa continuidade foi possível até o momento em que todas as populações convergiram para a mesma solução.

6 Considerações Finais

6.1 Conclusões

Embora não tenham sido realizados testes exaustivos, o mecanismo de distribuição desenvolvido para suportar a paralelização do algoritmo genético de granularidade grossa mostrou ser bastante adequado, apresentando bons resultados conforme tabelas e gráficos relatados no capítulo 5.

A implementação do Algoritmo Genético Paralelo, juntamente com o mecanismo de distribuição, possibilitou a obtenção de soluções (embora aproximadas) para instâncias maiores do PCV usando um ambiente distribuído (rede local), sem a necessidade de sofisticados computadores paralelos ou da configuração de cluster de estações. Deve ser ressaltado aqui que, como o objetivo principal do trabalho foi avaliar a potencialidade do Algoritmo Genético Paralelo, o algoritmo utilizado na implementação do AG foi o mais básico possível.

Com o mecanismo de distribuição desenvolvido, o algoritmo genético paralelo demonstrou ser extremamente escalável. Essa escalabilidade é resultado, em partes, da tecnologia JavaSpaces utilizada na implementação do mecanismo de distribuição. Durante a execução do

algoritmo, novas populações podem ser inicializadas simplesmente com a adição de mais computadores na rede.

Por fim, a migração de indivíduos entre as populações mostrou ser de suma importância para que boas soluções sejam alcançadas. Apesar de não ter sido feito um estudo aprofundado, observou-se que as populações estagnadas retomavam, muitas vezes, sua evolução quando recebiam indivíduos de outras populações.

6.2 Sugestões para novas pesquisas

Este trabalho não tem o objetivo de usar o mecanismo proposto para superar atuais recordes do PCV.

O algoritmo genético paralelo desenvolvido é o AG clássico de granularidade grossa (*também conhecido como Algoritmo Genético Distribuído*), sendo que os outros modelos de algoritmo genético paralelo estão fora do escopo desta pesquisa.

A avaliação quantitativa de desempenho da tecnologia JavaSpaces também não faz parte do objetivo deste trabalho.

O algoritmo genético desenvolvido foi testado numa rede local, sendo que sua distribuição em outros ambientes (por exemplo, Internet) está fora do objetivo da pesquisa.

7 Referências Bibliográficas

(ADAMIDIS, 1994)

ADAMIDIS, P. – **Review of parallel genetic algorithms bibliography** – Technical. Report, Universidade de Thessaloniki, Grécia, 1994

(CANTÚ-PAZ, 1997)

CANTÚ-PAZ E. – **A survey of parallel genetic algorithms** – IllGAL Report 97003, Universidade de Illinois, 1997.

(CANTÚ-PAZ, 1997)

CANTÚ-PAZ, E. – **Designing efficient master-slave parallel genetic algorithms** –IlliGAL Report N. 97004, Universidade de Illinois – Illinois Genetic Algorithms Laboratory, Urbana, IL, 1997.

(COULOURIS et all, 2001)

COULOURIS, G & DOLLIMORE, J & KINDBERG, T. – **Distributed Systems Concepts and Design**. – Addison-Wesley, 3 edição, 2001.

(DALLE MOLE, 2002)

DALLE MOLE, V. L. – **Algoritmos Genéticos – Uma abordagem paralela baseada em populações cooperantes** – Tese de Mestrado defendida no Programa de Pós-Graduação em Ciência da Computação, UFSC, 2002

(DE JONG, 1993)

DE JONG, K.A – **Evolutionary Computation** – MIT Press, 1993.

(FOGEL, 1966)

FOGEL, L.J. – **Evolutionary Programming in Perspective: The Top-Down View** – Comp. Intelligence - IEEE Press, 1966.

(FREEMAN et all, 1999)

FREEMAN, E. & HUPFER, S.& ARNOLD, K. – **JavaSpaces Principles,Patterns, and Practice** – Addison-Wesley, 1999.

(GELENTER, 1985)

GELENTER, D. – **Generative Communication in Linda** –ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1, pp. 80–112, 1985.

(GOLDBERG, 1989)

GOLDBERG, D.E. – **Genetic Algorithms in Search, Optimization and Machine Learning** – Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1989.

(GOLDBERG, 1994)

GOLDBERG, D.E. – **Genetic and Evolutionary Algorithms Come of Age** – Communications of the ACM, vol. 37, nº3, pp. 113-119.

(GOLDBERG et all, 1995)

GOLBERG,D.E. & KARGUPTA, H. & CANTÚ-PAZ, E. – **Critical deme size for serial and parallel genetic algorithms**. – Technical Report IlliGAL 95002, Universidade de Illinois, 1995.

(GORDON & WHITLEY, 1993)

GORDON, V.S. & WHITLEY, D. – **Serial and Parallel Genetic Algorithms as Function Optimizers** –. In FORREST S., Ed., Proceedings of the Fifth International Conference on Genetic Algorithms, p. 177-183, Morgan Kaufmann (San Mateo, CA), 1993.

(HARIK et all, 1997)

HARIK, G. & CANTU-PAZ, E. & GOLDBERG, D.E. & MILLER, B.L. – **The gambler's ruin problem, genetic algorithms, and the sizing of populations** – In Proceedings of 1997 IEEE International Conference on Evolutionary Computation, p. 7-12, IEEE Press, 1997

(HOLLAND, 1975)

HOLLAND, J., – **Adaptation in Natural and Artificial Systems** – University of Michingan Press, Ann Arbor, EUA, 1975. MIT Press Cambridge, MA, USA 1992

(JAVASPACES SPECIFICATION, 2000)

Sun Microsystems, Inc. – **JavaSpaces Service Specification 1.1** – Outubro, 2000, `http://www.sun.com/jini/specs.`

(KOZA, 1992)

KOZA, J. R., – **Genetic Programming On the Programming of Computers by Means of Natural Selection** – MIT Press, 1992.

(LIN et all, 1994)

LIN, S.C. & PUNCH, W. & GOODMAN, E. – **Coarse-Grain Parallel Genetic Algorithms : Categorization and New Approach** –. In Sixth IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society Press, Los Alamitos - CA, 1994.

(MAZZUCCO, 1999)

MAZZUCCO, J. J. – **Uma abordagem híbrida do problema da programação da produção através dos Algoritmos Genéticos e Simulated Annealing** – Tese de Doutorado, Florianópolis : UFSC, 1999.

(MAULDIN, 1984)

MAULDIN, M.L. – **Maintaining Diversity in Genetic Search** – Proceedings of the National Conference on Artificial Intelligence (AAAI-84), 1984.

(MIKI *et all*, 1999)

MIKI, M., HIROYASU, T., KANEKO, M., IIATANAKA, K. – **A Parallel Genetic Algorithm with Distributed Enviroment Scheme** – Departamento de Engenharia do Conhecimento, Universidade de Doshiha - Kyto, Japan 1999.

(MÜHLENBEIN, 2000)

MÜHLENBEIN, H. – **Evolution in Time and Space – The Parallel Genetic Algorithm** – GMD Schloss Birlinghoven, D-5205 Sankt Augustin, 2000l.

(NOWOSTAWSKI & POLI, 1999)

NOWOSTAWSKI, M. & POLI, R. – **Parallel Genetic Algorithm Taxonomy** – Universidade de Otago, Nova Zelândia, 1999.

(OCHI & VIANNA, 1998)

OCHI, L.S. & VIANNA, D.S. – **A Parallel Evolutionary Algorithm for the Vehicle Routing Problems** – Springer-Verlag, 1998.

(STRACUZZI, 1998)

STRACUZZI, D. J. – **Some Methods for the Parallelization of Genetic Algorithms** - 1998

(TANESE, 1989)

TANESE, – **Distributed Genetic Algorithms** – Proceedings of the Third International Conference on Genetic Algorithms, ed. J.D. Schaffer, Morgan Kaufmann, pp. 434-439, 1989.

(VAN STEEN & TANENBAUM, 2002)

VAN STEEN, M. & TANENBAUM, A. – **Distributed Systems Principles and Paradigms** – Prentice Hall, 2002.

(ZORMAN et all, 2002)

ZORMAN, B & KAPFHAMMER, G. M. & ROOS, R. – **Creation and Analysis of a JavaSpace-Based Distributed Genetic Algorithm** - Technical Report CS02-18, Department of Computer Science, Allegheny College, Meadville, PA, 2002.

(ZAPATA, 2000)

ZAPATA, M.G. – **JAVASPACES: A distributed computing model** – G.Mobile Networks - Nokia Research Center, Helsinki, FINLÂNDIA, 2000

(ZEBULUN, 1999)

ZEBULUM, R. S. – **Sintese de Circuitos Eletrônicos por Computação Evolutiva** – Tese de doutorado, PUC-RIO - Departamento de Engenharia Elétrica, Rio de Janeiro, 15 de Abril de 1999.

(ZUBEN, 2000)

ZUBEN, F. J. V.. – **Computação Evolutiva: Uma Abordagem Pragmática** – Tese de Doutorado, DCA/FEEC/Unicamp, 2000.