UNIVERSIDADE DO ESTADO DA BAHIA - UNEB

BACHARELADO EM SISTEMAS DE INFORMAÇÃO

RAUL FERREIRA LERMEN

PARALELIZAÇÃO DE UM ALGORITMO GENÉTICO HÍBRIDO PARA OTIMIZAÇÃO DE ROTAS NO PLANO CARTESIANO

Salvador

2014

RAUL FERREIRA LERMEN

PARALELIZAÇÃO DE UM ALGORITMO GENÉTICO HÍBRIDO PARA OTIMIZAÇÃO DE ROTAS NO PLANO CARTESIANO

Monografia apresentada ao curso de Sistemas de Informação da Universidade do Estado da Bahia – UNEB, como pré-requisito para obtenção do grau de bacharel, sob orientação do Prof. Cláudio Alves de Amorim.

Salvador

2014

RAUL FERREIRA LERMEN

ESTUDO DE PARALELIZAÇÃO DE UM ALGORITMO GENÉTICO HÍBRIDO NA SOLUÇÃO DE PROBLEMAS DE PONTOS CARTESIANOS

Monografia apresentada ao curso de Sistemas de Informação da Universidade do Estado da Bahia – UNEB, como pré-requisito para obtenção do grau de bacharel.

COMISSÃO EXAMINADORA

_____

Prof. Cláudio Alves de Amorim

_____

Prof. Diego Frias

_____

Prof. Leandro Coelho

Salvador, 20 de Dezembro de 2014

AGRADECIMENTOS

Primeiramente a Deus por todas as pessoas e oportunidades que colocou na minha vida por todos esses anos.

À minha família e principalmente a minha mãe, que sempre foi a maior incentivadora dos meus projetos e sempre quis o meu bem.

À minha namorada Julia Versoza, sempre companheira e cúmplice, contribui para que eu seja uma pessoa melhor a cada dia.

A todos os meus amigos do Bebeja, que nunca acreditaram que esse momento ia chegar, mas que sempre me apoiaram e tornaram esses anos os melhores da minha vida.

À meus colegas e amigos de faculdade Bruno Leal, Gustavo Santana, Lucas Trindade, Luciana Campos, Luis Serra, Mateus Matos, Rick Andrade, Roberval Junior e Yuri Guimarães e tantos outros que deixaram minha passagem na UNEB muito mais prazerosa e divertida.

Ao professor Claudio Amorim, que sem saber, foi inspiração para minha formação técnica e humana, Eduardo Jorge que conseguiu levar as matérias práticas a um alto nível, Diego Frias, Jorge Farias, Leandro Coelho e a todos os outros professores sempre foram solícitos e estão dispostos a contribuir para a formação da UNEB ser o nível que é.

E por fim, à todas as pessoas que passaram pela minha vida, deixando um pouco delas e também levando um pouco de mim.

"Everything around you that you call life, was made up by people that were no smarter than you."

(STEVE JOBS)

RESUMO

Este trabalho descreve a implementação paralela de um algoritmo genético híbrido para a otimização de rotas no plano cartesiano. O desempenho do novo algoritmo é comparado com o da sua versão sequencial, em termos do tempo de execução e qualidade dos resultados. Considerando resultados similares, o *speed-up* obtido variou entre 1,12x e 2,88x, em um computador equipado com processador Core i5, de 2 núcleos. Os testes foram rodados sobre os problemas d198 e a280 da TSPLIB, que representam modelos de placas de circuito impresso.

Palavras-chave: Algoritmos Genéticos, Problema do caixeiro viajante, Paralelização, Otimização.

ABSTRACT

This paper describes the implementation of the parallelization of a hybrid genetic algorithm for route optimizations in a Cartesian plane. The performance of the new algorithm is compared with the serial version, in terms of execution time and quality of results. Considering similar results, the speed-up obtained ranged between 1.12x and 2.88x, on a computer equipped with Core i5 processor, 2 cores. The tests were run on the d198 and a280 problems of the TSPLIB, that representing models of printed circuit boards.

Keywords: Genetic Algorithms, the traveling salesman problem, Parallelization, Optimization.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1. INTRODUÇÃO

O problema do caixeiro viajante é um dos diversos problemas matemáticos que despertam o interesse de diversas áreas. No meio teórico, ele é a tradução de diversos problemas encontrados na natureza e sua complexidade está na classe dos problemas NP-Completos, considerados problemas complexos e de difícil solução.

Sendo de grande importância, esse problema sempre é alvo de estudos e de melhorias, assim como ocorre nesse projeto, que deu continuidade ao estudo feito por Bartira Sena em 2011 onde conseguiu atingir bons resultados de otimização utilizando um algoritmo genético e o aplicando junto com o problema do caixeiro viajante para resolver problemas de perfuração de placas de circuito impresso.

Esse projeto apresenta uma proposta de melhoria do algoritmo de perfuração de placas usado no estudo feito por Bartira Sena (2011) e fazer um estudo do ganho da paralelização do algoritmo.

A utilização e execução de código em paralelo possibilitam o aumento do volume de processamento e consequentemente a possibilidade de encontrar mais soluções ótimas em menos tempo. Testar possibilidades de aprimorar o código contribui para as pesquisas e o desenvolvimento dessa área, além enriquecer ainda mais o campo de algoritmos genéticos híbridos, que apesar de ser uma área bastante estudada, ainda existe muito espaço para inovações e melhorias.

Além disso, o desenvolvimento de um método eficiente para paralelização de algoritmos genéticos traz a possibilidade de aplicação nas mais diversas áreas, potencialmente ajudando outras pesquisas de adaptação de algoritmo. Com a chegada de *smartphones* e *tablets*, por exemplo, esse método pode permitir a resolução de problemas importantes de forma autônoma sem o contato com servidores, em lugares remotos, e até mesmo melhorar o desempenho de jogos.

Para analisar os resultados foram feitos testes para verificar o tempo de execução total do algoritmo paralelo em relação ao algoritmo original e outro teste para verificar se a qualidade dos resultados permaneceram os mesmos depois da paralelização. Nos dois cenários foram montados gráficos específicos para medição dos resultados.

Os resultados apresentaram a eficiência da paralelização e a possibilidade de aplicação do algoritmo em diversas áreas, além da possibilidade de aplicação em novas situações, como na computação na nuvem afim de resolver problemas mais complexos.

A monografia esta organizada em 5 capítulos. O capitulo 2 mostra a revisão bibliográfica e apresenta uma breve introdução ao problema do caixeiro viajante, além das estratégias utilizadas para paralelizar o algoritmo original.

O capítulo 3 apresenta como o algoritmo genético ficou depois da paralelização, além dos problemas e soluções encontradas durante suas modificações, os resultados obtidos e como os algoritmos se comportaram durante os testes.

O capítulo 4 apresenta as conclusões finais e as possibilidades de trabalhos futuros.

E por fim, o capítulo 5 apresenta as referencias bibliográficas utilizadas no trabalho.

2. REVISÃO BIBLIOGRÁFICA

2.1. O PROBLEMA DO CAIXEIRO VIAJANTE

O problema de perfuração de placas é uma variação do problema do caixeiro viajante (TSP). Nele, um caixeiro deve percorrer todas as cidades dentro de um determinado território de modo que cada cidade seja percorrida apenas uma vez, depois deve retornar ao seu ponto de origem no final do percurso. Essa viagem deve ser feita com a menor distância percorrida. Apesar de não existir uma solução polinomial, existe uma série de métodos para se chegar a soluções boas (porém não ótimas), denominados algoritmos de aproximação ou algoritmos heurísticos.

O TSP foi primeiramente estudado por Karl Manger nos anos de 1930 em Viena e Harvard e logo depois investigado por Hassler Whitney e Merrill Flood em Princeton (Maderia, 2010). Métodos de resolução do TSP começaram a aparecer em artigos nos anos de 1950 e depois disso despertou o interesse de muitos pesquisadores, matemáticos, engenheiros da computação, físicos e até químicos.

Apesar de ser fácil de entender, o problema do caixeiro viajante é um problema difícil de se resolver pois está na classe dos problemas NP-Completos, e são considerados insolúveis por algoritmos polinomiais. Os métodos de solução do TSP foram se aprimorando com o passar dos anos e o tamanho dos problemas resolvidos se tornam cada vez maiores, como mostrado na tabela 1.

Tabela 1 – Evolução das soluções encontradas para o TSP.

Ano	Time de Pesquisa	Tamanho da Instância
1954	G. Dantzig, R. Fulkerson, and S. Johnson	49 cidades
1971	M. Held and R.M. Karp	64 cidades
1975	P.M. Camerini, L. Fratta, and F. Maffioli	67 cidades
1977	M. Grötschel	120 cidades
1980	H. Crowder and M.W. Padberg	318 cidades
1987	M. Padberg and G. Rinaldi	532 cidades
1987	M. Grötschel and O. Holland	666 cidades
1987	M. Padberg and G. Rinaldi	2.392 cidades
1994	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	7.397 cidades
1998	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	13.509 cidades
2001	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	15.112 cidades
2004	D. Applegate, R. Bixby, V. Chvátal, W. Cook, and K . Helsgaun	24.978 cidades
2006	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	85.900 Cidades

Para provar a eficiência do método, alguns times de pesquisa utilizaram distâncias de cidades reais, e em 1973 pesquisadores utilizaram o TSP diretamente na aplicação de um problema de perfuração de uma placa de 318 pontos. A partir dai, aplicação do TSP em problemas de perfuração se tornou comum e, de acordo com Cook (2012), a maior instância do TSP foi resolvida em 2006 por ele e sua equipe em um problema de perfuração de placas em que o laser tinha que percorrer 85.900 pontos. Apesar de ótimos resultados alcançados, Cook acredita que ainda estamos a mais de uma década para resolver um problema de 100.000 pontos.

O TSP pode ser a tradução dos mais diversos problemas. Um exemplo interessante é o planejamento de telescópios espaciais feito pela NASA. Chamado de "*travelling planet-finder problem*", ou "problema do viajante procurador de planetas" em português, seu maior objetivo é enviar satélites a procura de planetas parecidos com a terra em orbitas ao redor de estrelas. Nesse caso, o TSP é usado para determinar a sequência de observações feiras pelos telescópios, de modo a minimizar o combustível utilizado e maximizar o ganho da viagem.

O TSP também é aplicado na criação de chips de computadores, cristalografia de raio-X, perfuração em bronze, limpeza de *wafers* de silício e até mesmo para melhorar a performance de software de jogos.

A busca por bons resultados levou o a área do TSP a buscar métodos de diversas outras áreas. Algoritmos matemáticos como o *Local Search* e o *Hill Climbing* têm sido

aplicados na procura por soluções cada vez melhores, bem como heurísticas estatísticas e métodos de otimização diversos, dentre eles os algoritmos genéticos.

No trabalho de Sena (2011), um algoritmo genético (GA) híbrido foi aplicado para a obtenção de soluções aproximadas, usando o problema da perfuração de placas como modelo conceitual para estudar a melhoria do algoritmo genético voltado ao TSP.

2.2. UM GA HÍBRIDO PARA A SOLUÇÃO DO TSP

Os GAs constituem uma família de algoritmos meta-heurísitcos inspirados nos princípios da seleção natural e sobrevivência dos mais aptos. Os GAs utilizam termos da genética como cromossomos e genes, onde genes são unidades que formam um cromossomo onde nesse contexto representa uma possível solução para o problema.

A execução do GA cria uma população inicial de possíveis soluções e a partir daí se inicia a evolução através de operadores genéticos. Os operadores são usados para a criação de novos indivíduos que serão reinseridos na população, ao final de cada geração a adaptação da população é avaliada através do fitness, uma medida que caracteriza a qualidade do cromossomo. Assim alguns indivíduos são selecionados para a próxima geração, onde serão recombinados ou mutados para formar uma nova população. Cada nova população é usada como entrada para a população seguinte.

Após a representação do GA para o problema do TSP, o algoritmo usado como base para o trabalho de Bartira foi construído seguindo seguintes os passos: foi escolhida aleatoriamente metade da população inicial de cromossomos e a outra metade a partir da heurística construtiva de *savings* e a partir dela é executada uma função de aptidão que julga a qualidade da solução apresentada, ou seja, se ela se aproxima de uma solução ótima ou não. Nesse ponto são executados os operadores genéticos: o crossover, um operador de cruzamento que tem por fim originar um ou mais cromossomos a partir de dois cromossomos pertencentes a população, e a Mutação, que altera um ou mais genes de um determinado cromossomo a fim de garantir a diversidade genética da população. A cada execução o algoritmo busca bons resultados na população, faz o teste de aptidão, e depois faz modificações através dessas duas técnicas para formar novas soluções.

Em continuidade Sena (2011) fez alterações no algoritmo original de Jayalakshmi et al.(2001) e executados testes, que foram comparados com resultados da benchmark da biblioteca TSPLIB. Por fim, diferentes testes mostraram ganhos entre 0,6% e 3,0%

comparado ao benchmark do problema d198 sendo ele o único que representa o modelo de placa de circuito impresso nessa biblioteca.

O GA implementado por ela é resumido no final em 5 passos:

PASSO 1:

Inicialize 50% da população através da Heurística de Savings.

Inicialize 50% da população aleatoriamente.

PASSO 2:

Aplique o algoritmo RemoveSharp em toda a população inicial.

Aplique o algoritmo LocalOpt em toda a população inicial.

PASSO 3:

Selecione randomicamente dois pais.

Aplique o operador de crossover *edge recombination* entre os pais e gere um descendente.

Aplique o algoritmo RemoveSharp no descendente.

Aplique o algoritmo LocalOpt no descendente.

Se CustoRota(descendente) < CustoRota(qualquer um dos pais):

Se o descendente possuir a melhor rota dentre todas as rotas existentes na população, substitua pelo descendente o pai que mais se assemelhe ao mesmo.

Senão substitua pelo descendente o pai com o pior custo.

PASSO 4:

Selecione randomicamente um número entre 0 e 1. Se o número selecionado for menor do que a taxa de mutação, aplique o operador de mutação a um cromossomo da população selecionado randomicamente.

PASSO 5:

Repita os passos 3 e 4 até um determinado número de iterações. A seguir, serão discutidas as escolhas efetuadas para esta configuração.

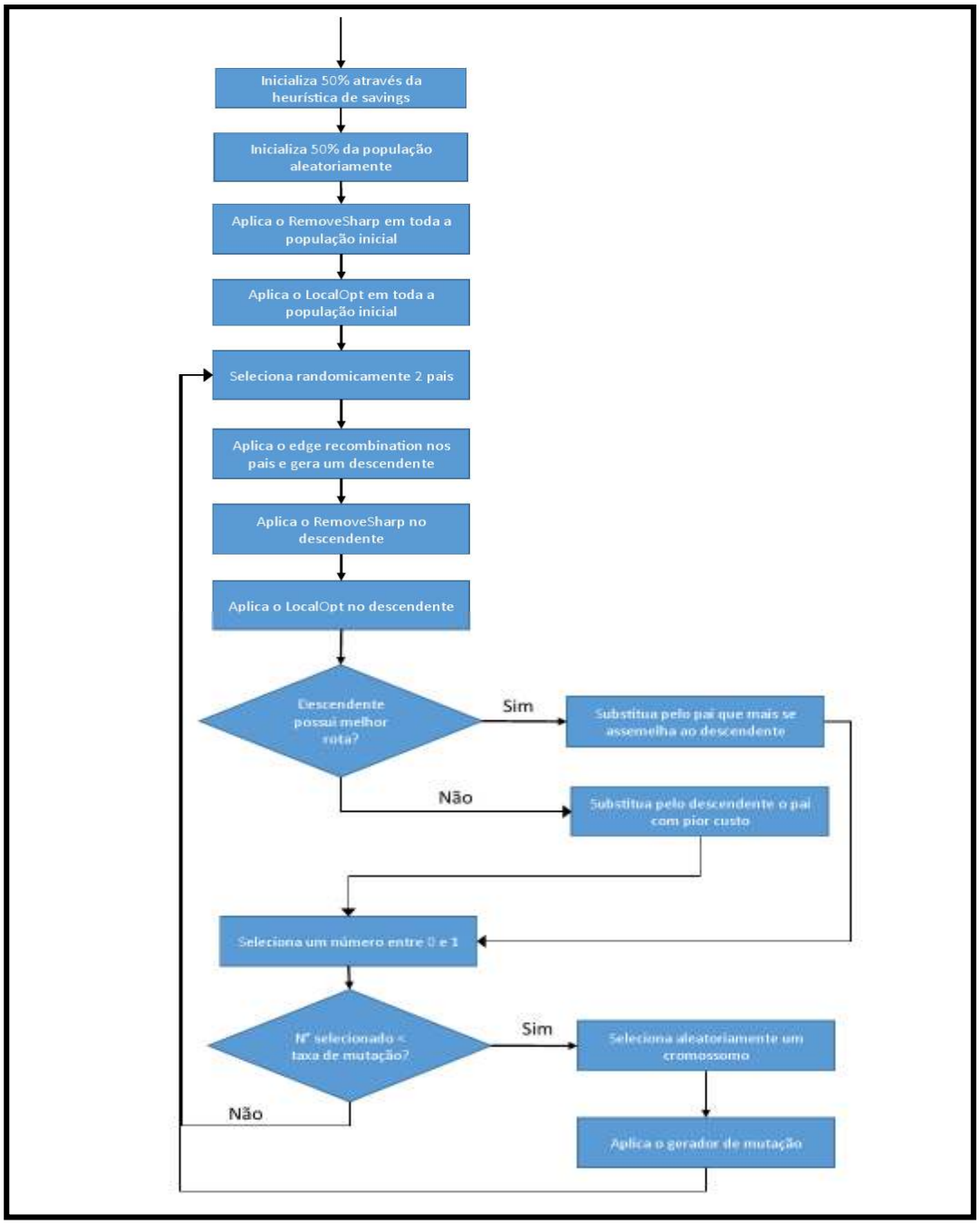

Figura 1 – Fluxograma de execução do algoritmo genético híbrido.

2.2.1. HEURÍSTICAS E ALGORITMOS DE MELHORIA UTILIZADOS

Durante a inicialização, o algoritmo criado por Bartira inicializa metade da sua população utilizando a heurística de *Savings*, um algoritmo de complexidade $O(n^2 \log(n))$ que segue os seguintes passos:

1. Seleciona um vértice k por algum critério ou aleatoriamente.

2. Para os demais vértices do grafo, crie sub rotas saindo de ki, onde i = 1, ..., n, passando pelo vértice k e retornando ao vértice ki. Ou seja, um circuito não hamiltoniano que passa n vezes pelo nó k.

3. Obtenha a lista das economias da seguinte forma: Sij = cjk – cij, onde i,j = 1, ..., n, onde S é a economia feita se o vértice i for ligado diretamente ao vértice j sem passar por k. □

4. Ordene a lista de economias em ordem decrescente (da maior para a menor economia). □

5. Percorra a lista de economias iniciando pela primeira posição. □

6. Se a inserção da aresta (i,j) e a retirada da aresta (k,j) e (j,k) resultar em uma rota□iniciando em k e passando pelos demais vértices, elimina da lista. □Caso contrário, tenta a ligação seguinte da lista. □

7. Repetir o processo até que um ciclo hamiltoniano tenha sido formado.

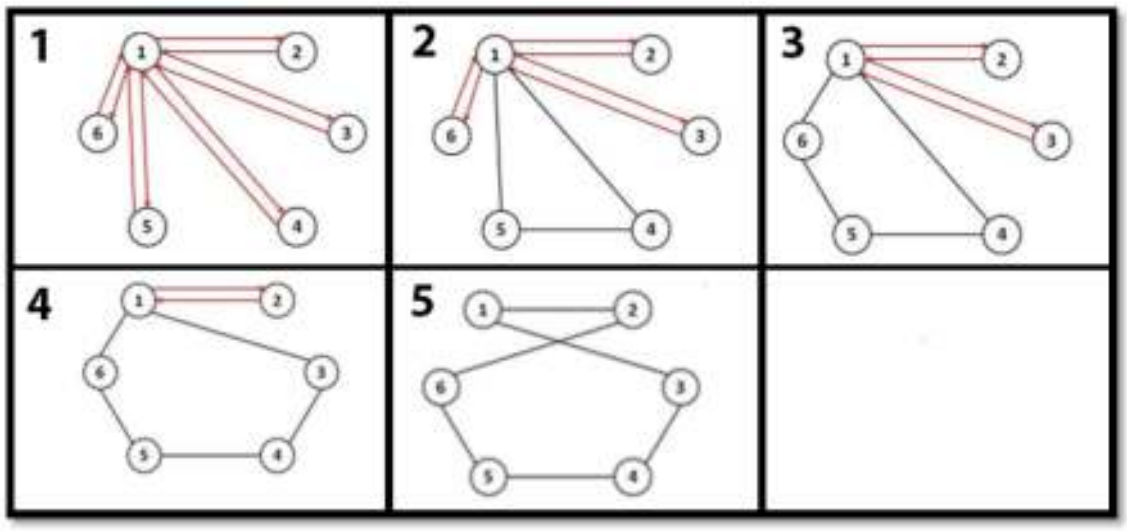

Figura 2 – Exemplo de execução da heurística de *Savings* (Adaptado de Bartira, 2011)

Por seguinte, o algoritmo usado para inicialização da outra metade da população é o *Nearest Neighbor*, ou "Vizinho mais próximo", um algoritmo de complexidade $O(n^2)$ que constrói rotas aleatórias a partir dos seguintes passos :

1. Escolhe uma cidade arbitrária.
2. Encontra a cidade mais próxima à ultima cidade adicionada, e que ainda não esteja na rota, e adiciona a rota conectando-as.
3. Quando todas as cidades tiverem sido adicionadas à rota, conecte a primeira cidade adicionada à ultima cidade adicionada.

Após a inicialização da população inicial, os algoritmos de melhoria *RemoveSharp* e *LocalOpt* são utilizados para otimizar as rotas inicializadas. A cada nova rota gerada eles são aplicados.

O *RemoveSharp* funciona da seguinte forma:

1. Uma lista (NEARLIST) contendo as m cidades mais próxima da cidade selecionada é criada. □

2. A cidade selecionada é removida da rota e uma nova rota com N-1 cidades é formada. □

3. A cidade selecionada é então reinserida na rota antes ou depois de qualquer uma das cidades presentes na NEARLIST e o custo da nova rota é calculado para cada caso. □

4. A rota de menor custo é então selecionada. □

5. Os passos são repetidos para cada cidade presente na rota.

Abaixo um exemplo de otimização utilizando o algoritmo *RemoveSharp*:

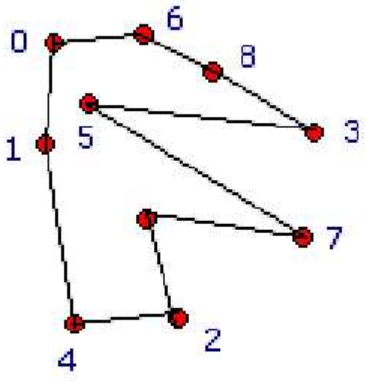

Figura 3 - Rota com cidade mal posicionada (Jayalakshmi et al., 2001).

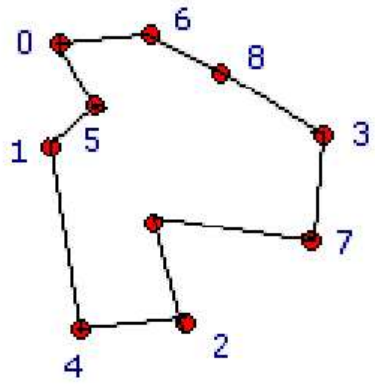

Figura 4 - Rota otimizada pelo algoritmo RemoveSharp (Jayalakshmi et al., 2001)

Por ultimo, o algoritmo *LocalOpt* seleciona cidades consecutivas da rota e as reorganiza de modo que a distância entre as cidades seja mínima através da busca por todos os arranjos possíveis. Abaixo, as figuras 4 e 5 apresentam o exemplo da aplicação do algoritmo em uma rota, cuja distância entre as cidades 1 e 6 é diminuída.

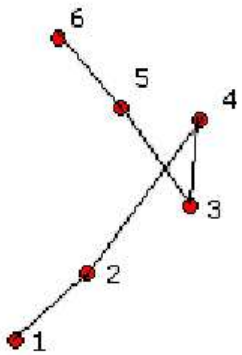

Figura 5 – Rota antes da utilização do LocalOpt.

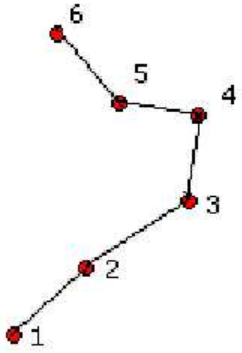

Figura 6 – Rota após a utilização do LocalOpt

2.3. ALGORITMOS GENÉTICOS PARALELOS (PGAs)

De acordo com Abtin Hassani e Jonatan Treijs (2010), algoritmos genéticos executados em paralelos podem ser otimizados nos mais diferentes usos e implementações. A paralelização permite que o algoritmo genético seja executado mais rapidamente com a ajuda de multiprocessadores. Manter subpopulações relativamente isoladas é um método de paralelização que ajuda a ganhar mais diversidade. Os PGAs podem ser divididos em três classes genéricas: baixa granulação, alta granulação e Mestre-Escravo.

2.3.1. MODELO MESTRE-ESCRAVO (MSM)

MSM é um modelo criado principalmente para ganhar velocidade, escala e poder de processamento. A distribuição dos processos de crossover e mutação pode ser executados por processadores diferentes, isso permite o uso de poder de processamento independente em cada processador ou em sistemas distribuídos.

Nele uma fração da população é distribuída para cada processador para a operação evolutiva e pode ser executada de duas maneiras: uma sincronizada, em que uma vez que a população for associada ao processador, o algoritmo espera que todos os processadores terminem suas operações para acontecer a sincronização antes de avaliar a nova população; no segundo, o algoritmo não espera a execução de todos os processadores terminar e acaba por não retornar resultados de processadores mais lentos.

2.3.2. ALGORITMO GENÉTICO PARALELO REFINADO (FGA)

Os algoritmos FGAs tem apenas uma população que possui uma estrutura especial onde os indivíduos da sua população interagem apenas com seus vizinhos. Esse tipo de algoritmo tem um problema de desempenho que se degrada de acordo com o aumento da população.

2.3.3. ALGORITMO GENÉTICO DE GRANULACAO ALTA (CGA)

Também conhecido como "ilha" PGA. CGA divide a população em subpopulações onde pode ser computado em processadores separados. As subpopulações tem uma quantidade de migrações entre elas. Os benefícios de ter populações pequenas é a rápida troca entre as populações. A desvantagem é que o rápido crescimento do fitness para no menor valor do fitness da subpopulação. Ao primeiro ver, parece um método simples de usar porém os problemas de implementação aparecem quando são analisados mais de perto: quando se coloca um valor baixo de migrações entre as populações, se obterá uma quantidade baixa de boas soluções, mas quando se aumenta muito o nível de migração, o comportamento volta a ser o de uma população grande (Hassani, 2009).

2.4. PARALELISMO – THREADS EM JAVA

A linguagem Java foi desenvolvida para suportar programação concorrida, que quando aplicada ganha o nome de thread. Thread é uma chamada de sequência que é executada de maneira independente e paralela, possibilitando o compartilhamento de recursos de uma máquina (Schütte, 2012).

Quanto se executa um código em paralelo, o programador deve se preocupar com a comunicação entre as threads para que elas trabalhem em sincronia umas com as outras. A sincronização faz com que o objeto seja modificado apenas por uma thread por vez, fazendo com que elas não acessem objetos desatualizados ou parcialmente modificados por outras threads.

De acordo com estudo feito por Smith e Bull, testes de benchmark mostram que o desempenho das aplicações que utilizam thread dependem do modelo de programação, ambiente de execução e o hardware utilizados.

2.4.1. O PROBLEMA DA SINCRONIZAÇÃO EM THREADS

Na computação paralela, sincronização de threads é algo crucial para o resultado da aplicação. A grande preocupação quando se utiliza threads é saber se os dados utilizados por cada uma estão sincronizados e consistentes com os outros. Existem diversas aplicações para a solução desse problema, e a utilização de cada uma depende da natureza de cada problema.

2.4.2. MÉTODO DE SINCRONIZAÇÃO – BARRIER

O barrier é um método de sincronização utilizado quando um grupo de threads estão sendo executados e o código principal precisa prosseguir utilizando os resultados finais de cada processo. Nesse caso é criado uma barreira que espera o termino de todas as threads para só então dar continuidade a execução do código principal.

3. PARALELIZAÇÃ DO ALGORITMO GENÉTICO

Para implementação do GA paralelo foi escolhido o modelo mestre-escravo. Nele, o algoritmo principal usa threads para criar os novos cromossomos e espera, usando um barrier, o termino de todas elas para fazer a sincronização. Após a sincronização o algoritmo principal volta a execução.

A modificação do código foi feito basicamente em duas etapas. Na primeira etapa foi feita a pesquisa e o desenvolvimento das adaptações do código original onde foram identificados os possíveis pontos de execução de Stream dentro do código Java, e criados pontos de paralelismo. Em seguinte, o código foi usado para gerar e testar populações genéticas e comparar sua através de um benchmark com o código original.

3.1. ALGORITMO PARALELO

Sena (2011) demonstrou ganhos em seu algoritmo e uma média próxima da solução global no modelo d198 que também se mostrou eficaz ainda em outras instâncias do TSPLIB como a a208 e pcb442. Dando continuidade, esse trabalho propõe melhorias em seu GA hibrido a fim de poder executá-lo mais rapidamente, além de observar sua eficiência comparando-o com o código serial.

Como algoritmo final, foi adicionado mais um passo (Passo 3) e modificado outro (Passo 4) no algoritmo original, a fim de alterar as restrições de paralelização do código original e adaptá-lo à nova lógica das threads.

O novo algoritmo ficou com 6 passos:

PASSO 1:

- Inicialize 50% da população através da Heurística de *Savings*.
- Inicialize 50% da população aleatoriamente.

PASSO 2:

- Aplique o algoritmo *RemoveSharp* em toda a população inicial.
- Aplique o algoritmo *LocalOpt* em toda a população inicial.

PASSO 3:

- Para cada filho descendente gerado, crie uma thread.

PASSO 4:

- Selecione randomicamente dois pais.
- Aplique o operador de crossover *edge recombination* entre os pais e gere um descendente.
- Aplique o algoritmo *RemoveSharp* no descendente.
- Aplique o algoritmo *LocalOpt* no descendente.
- Se CustoRota(descendente) >CustoRota(qualquer um dos pais):
    - Se algum dos pais possuir melhor roda que o descendente, o cromossomo mais apto será considerado descendente.

PASSO 5:

- Selecione randomicamente um número entre 0 e 1. Se o número selecionado for menor do que a taxa de mutação, aplique o operador de mutação a um cromossomo da população selecionado randomicamente.

PASSO 6:

- Repita os passos 3 e 4 até um determinado número de iterações. A seguir, serão discutidas as escolhas efetuadas para esta configuração.

Na figura 1 foi feita uma representação visual resumida para compreender melhor como funciona a execução em paralelo do novo algoritmo.

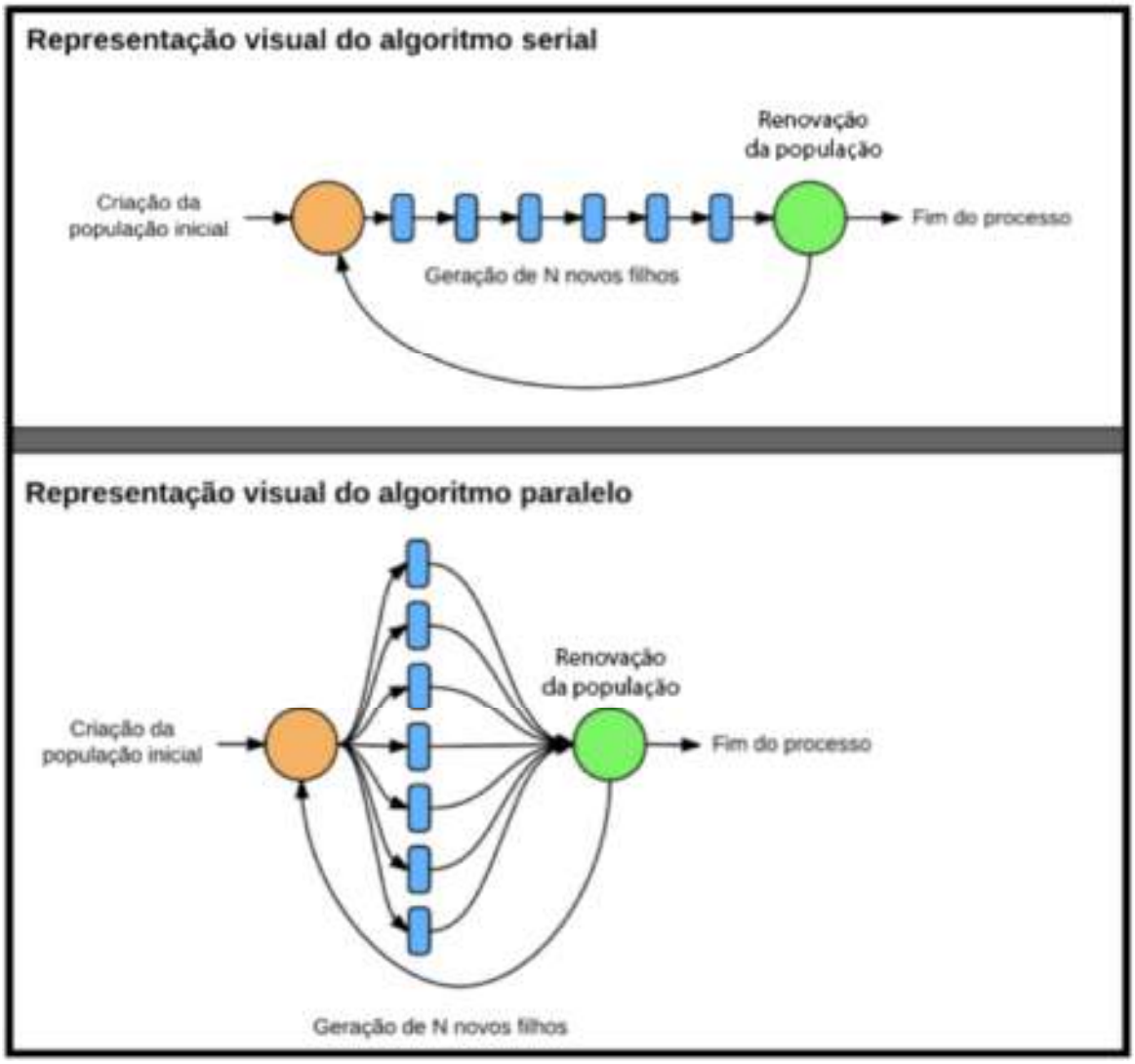

Figura 7 – Representações visuais da execução dos algoritmos.

3.2. PARALELIZAÇÃO DA GERAÇÃO DE NOVOS FILHOS

O cálculo do fitness geralmente é o responsável pela maior parte do processamento de um GA comum, o que acaba fazendo ele se tornar o maior alvo de adaptações para processamento paralelo, porém, pela natureza do problema abordado por Sena (2011), um melhor ganho de paralelização acontece nos passos de geração dos novos cromossomos filhos.

Por isso, ao invés de cada filho ser gerado de maneira serializada, cada filho agora é gerado paralelamente e a quantidade de threads é determinada pela formula:

Número de threads = ([Tamanho da população]/2)(1 – [Taxa de preservação])*

Nesse caso, o número de threads criadas sempre supera o número de processadores usados nos testes e, de acordo com Boratto (2014), quando se utiliza um número de threads maior que o número de núcleos, o ganho pode ser maior que o número de processadores se o Intel Hyper-Threading [64] estiver ativado. Como foi o caso desse experimento.

3.3. AJUSTES NO CÓDIGO ORIGINAL

Ao aplicar a paralelização na geração dos filhos, foi identificado um gargalo no código original que impedia a comparação direta com desempenho do código alterado, nele, o *array* responsável por fazer a gestão dos cromossomos da população que já tinham feito cruzamento recebia uma carga grande de acessos quando a maioria de seus espaços estavam ocupados, já que estatisticamente, quanto mais cromossomos sorteados mais chances do próximo cromossomo estar sorteado, situação não encontrada na nova lógica de threads, já que o *array* é sorteado previamente e cada thread recebe os números dos cromossomos que irão participar do cruzamento.

Para equiparar os resultados, essa situação foi alterada no código original, que acabou resultando em um ganho de quase 8% em relação a suas execuções iniciais.

3.4. MÉTRICAS

Nessa monografia é feita a avaliação da qualidade dos resultados utilizando métricas.

3.4.1. SPEEDUP

O *speedup* é uma métrica usada para medir o ganho que uma tarefa tem quando ela é executada. Geralmente é aplicada na computação paralela para mostrar o ganho de performance que uma tarefa tem quando é comparada com sua execução original. Sua fórmula é dada por:

$$S = \frac{T_{old}}{T_{new}}$$

S = Resultado do speedup;
Told = tempo de execução sem o ganho;
Tnew = tempo de execução

3.4.2. TEMPO

Apesar dos resultados apresentados na execução do algoritmo pelo compilador ser dado em milissegundos, nesse trabalho todos as representações temporais foram traduzidas para segundos na intenção de melhorar a leitura do trabalho.

3.4.3. RESULTADOS

Cada problema utilizado na execução do algoritmo tem medidas diferentes. Enquanto o problema d198 é a representação de cidades em quilômetros, o problema pcb442 é a representação dos milímetros dos furos de uma placa de circuito impresso. Por isso, os resultados finais apresentados na execução dos problemas foram traduzidos como uma unidade de medida (u) na intenção de facilitar a leitura do trabalho.

3.5. RESULTADOS E TESTES

Os testes desse projeto foram executados em uma máquina Macbook Air meados de 2013, com processador Core i5 1,7GHz operando com sistema operacional OSX 10.10. O algoritmo foi escrito em Java na versão 1.6. As instâncias do problema do caixeiro viajante e o benchmark dos experimentos foram as mesmas utilizadas por Sena (2011), retiradas da biblioteca do TSPLIB (<http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/>).

A seguir estão os dados dos resultados de cada execução do algoritmo, mostrando seu comportamento nos cenários de teste.

Tabela 2 – Descrição dos cenários de teste.

Cenário de Teste	Objetivo	Problemas utilizados	Parâmetros	Quantidade de execuções
Cenário 1	Determinar ganho no tempo de execução.	d198 / a280 / pcb442	Pop. 50 cromossomos; 20.000 gerações executadas.	2 vezes cada problema
Cenário 2	Determinar qualidade dos resultados.	d198 / a280	Pop. 50 cromossomos; 200/500/2000/5000 gerações executadas.	10 vezes cada problema

3.5.1. EFEITOS DAS MODIFICAÇÕES EM RELAÇÃO AO TEMPO DO ALGORITMO ORIGINAL

Os parâmetros escolhidos para a execução do GA foram os mesmos utilizados por Sena (2011), que baseiam-se nas experiências relatadas por Jayalakshmi et al.(2001) utilizando uma população de 50 cromossomos e uma taxa de mutação de 0,02.

Para a realização dos testes iniciais o código original foi executado nas mesmas condições do código em paralelo para uma comparação direta dos resultados. Abaixo as tabelas e um gráfico comparativo mostrando o tempo do código paralelo em relação ao código original, usando como base os três problemas utilizados em seu projeto original.

Tabela 3 – Tempo de execução de cada código para 20.000 gerações.

Problema	Código Original	Código Modificado	*Speedup*
d198	8 405s	3 100s	2,71x
a280	17 635s	6 937s	2,54x
pcb442	46 525s	28 050s	1,66x

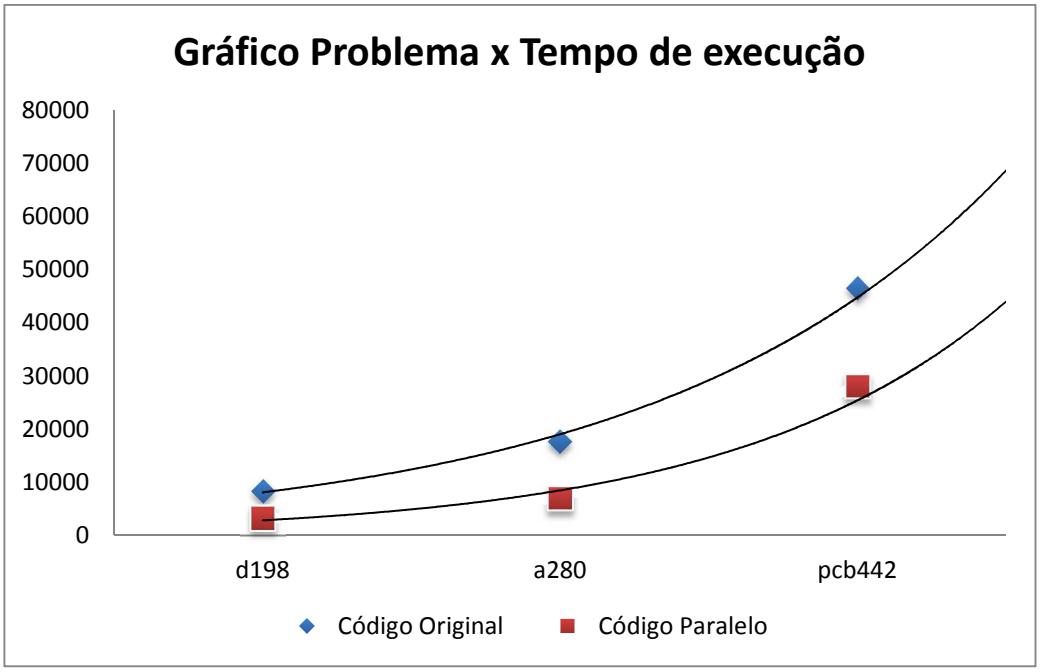

Figura 8 – Gráfico comparativo dos tempos de execução para 20.000 gerações.

Os experimentos demonstram que a paralelização do código se mostrou eficiente e consegue desempenho com 2 processadores de até 2,7 vezes melhor em relação ao tempo de execução do código original. O desempenho é maior do que o número de processadores por causa do Hyper-Threading utilizado em processadores core i5. Esse recurso permite um melhor aproveitamento do escalonamento do processador para realizar tarefas no core enquanto ele estiver esperando o término de outras execuções, emulando 4 processadores apesar de ter apenas 2 núcleos.

A execução de instâncias maiores mostra uma queda de desempenho nesse ambiente devido ao tamanho das threads ficarem cada vez maiores (por cada thread ser um cromossomo criado), sobrecarregando a memória do processador e cada vez mais tornando o escalonamento menos efetivo. Problema possivelmente contornado com o aumento do número de núcleos físicos.

3.5.2. EFEITO DAS MODIFICAÇÕES NA QUALIDADE DOS RESULTADOS

Para comprovar a qualidade dos resultados, para os dois algoritmos foram executados testes que utilizaram como critério de parada os valores de 200, 500, 2000 e 5000 gerações sem novos resultados melhorados. Com esse critério, quando o algoritmo chega a um

determinado numero de gerações sem novos resultados melhores sua execução é suspensa e, comparando a média dos resultados finais, é possível verificar se a qualidade foi mantida e o tempo de execução melhorada.

O algoritmo foi executado 10 vezes cada, usando o problema d198 e o problema a280 como referência com a população de 50 cromossomos e taxa de mutação de 0.02 como nos testes anteriores.

3.5.2.1. RESULTADOS DO PROBLEMA d198

3.5.2.1.1. RESULTADOS PARA 200 GERAÇÕES

O primeiro teste foi executado com o critério de parada de 200 gerações sem novos filhos e, de acordo com a tabela 4, a média dos resultados teve uma diferença de 1,22% e ganho de 1,43x em relação ao código original. Considerando que o GA aplicado é um algoritmo não determinístico, é possível comprovar que a qualidade dos resultados permanecem semelhantes.

Tabela 4 – Média dos resultados encontrados para 200 gerações – d198.

	Código Original	Código Modificado	Diferença
Média dos Resultados	16 517	16 318	1,22%
Média dos Tempos	137s	96s	143,5%
Desvio padrão	193,4	242,4	20,2%

No gráfico 3, foram escolhidas as duas execuções que tiveram convergências para resultados próximos. Nela pode ser observado que os comportamentos de execução permanecem semelhantes.

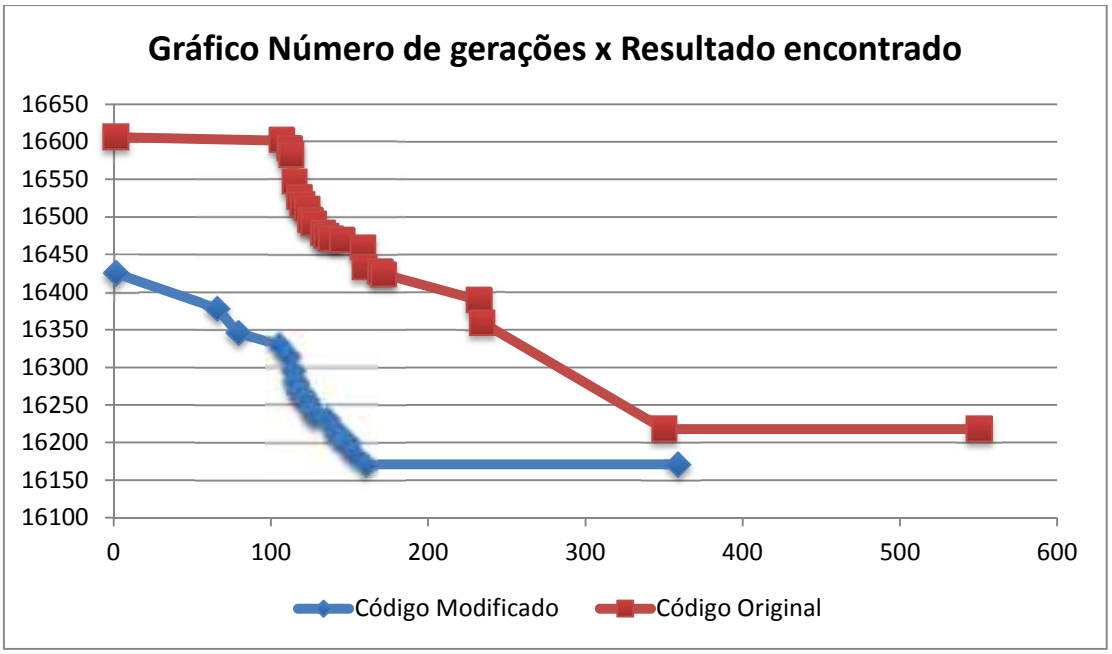

Figura 9 - Gráfico das execuções com resultados semelhantes para 200 gerações – d198.

3.5.2.1.2. RESULTADOS PARA 500 GERAÇÕES

O segundo teste foi executado com o critério de parada de 500 gerações sem novos filhos e, de acordo com a tabela, a média dos resultados teve uma diferença de 1,08% e ganho de desempenho de 1,46x, mostrando um leve ganho de semelhança e desempenho em relação ao primeiro teste.

Tabela 5 – Média dos resultados encontrados para 500 gerações – d198.

	Código Original	Código Modificado	Diferença
Média dos Resultados	16 388	16 214	1,08%
Média dos Tempos	398s	271s	146,7%
Desvio padrão	157,0	138,5	13,4%

O gráfico 4 mostra também duas execuções com convergências a resultados semelhantes para observar o comportamento.

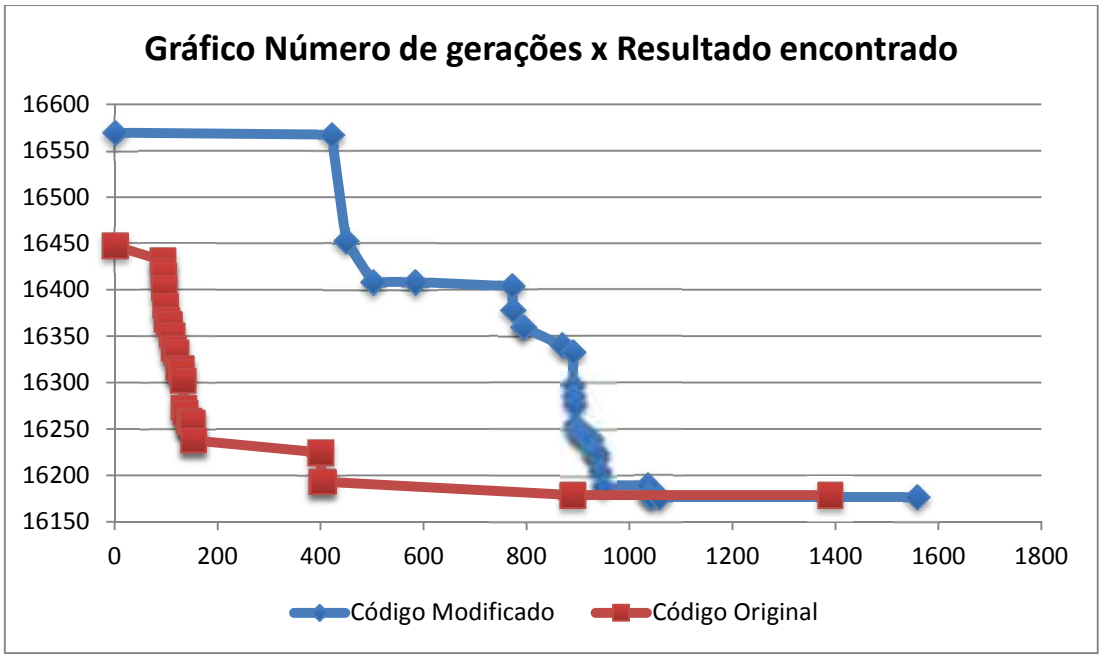

Figura 10 - Gráfico das execuções com resultados semelhantes para 500 gerações – d198.

3.5.2.1.3. RESULTADOS PARA 2000 GERAÇÕES

Em seguinte, o teste foi executado com o critério de parada de 2000 gerações sem novos resultados melhores, mais uma vez a média dos resultados se aproximou da semelhança com uma diferença de 0,75% o ganho de velocidade aumentou para 2,04x.

Tabela 6 – Média dos resultados encontrados para 2000 gerações – d198.

	Código Original	Código Modificado	Diferença
Média dos Resultados	16200	16080	0,75%
Média dos Tempos	2 306s	1 129s	204,3%
Desvio padrão	44,6	75,4	40,8%

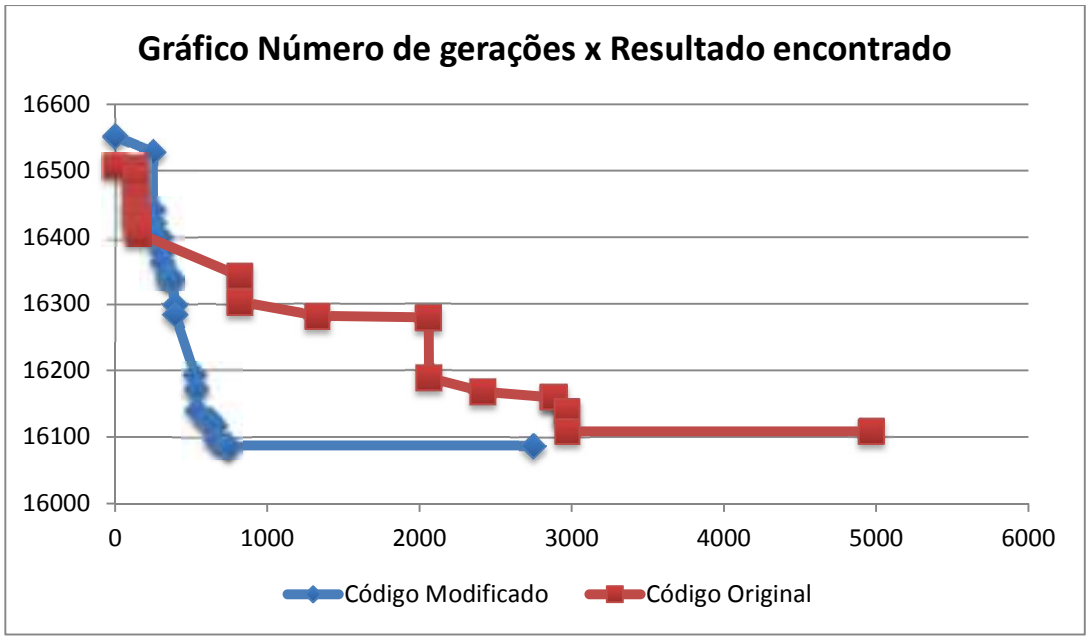

Figura 11 - Gráfico das execuções com resultados semelhantes para 2000 gerações – d198.

3.5.2.1.4. RESULTADOS PARA 5000 GERAÇÕES

Por último, o teste foi executado com o critério de parada de 5000 gerações sem novos resultados melhores, onde a média dos resultados teve uma diferença de 0,40% que mais um vez comprova a proximidade dos resultados.

Tabela 7 – Média dos resultados encontrados para 5000 gerações – d198.

	Código Original	Código Modificado	Diferença
Média dos Resultados	16 074	16 011	0,40%
Média dos Tempos	6 088s	1 512s	402,6%
Desvio Padrão	139	36	284,3%

No gráfico 3, foram escolhidas as duas execuções que tiveram resultados mais parecidos. Nela pode ser observado que os comportamentos de convergência permanecem próximos também para execuções maiores, comprovando mais uma vez os resultados.

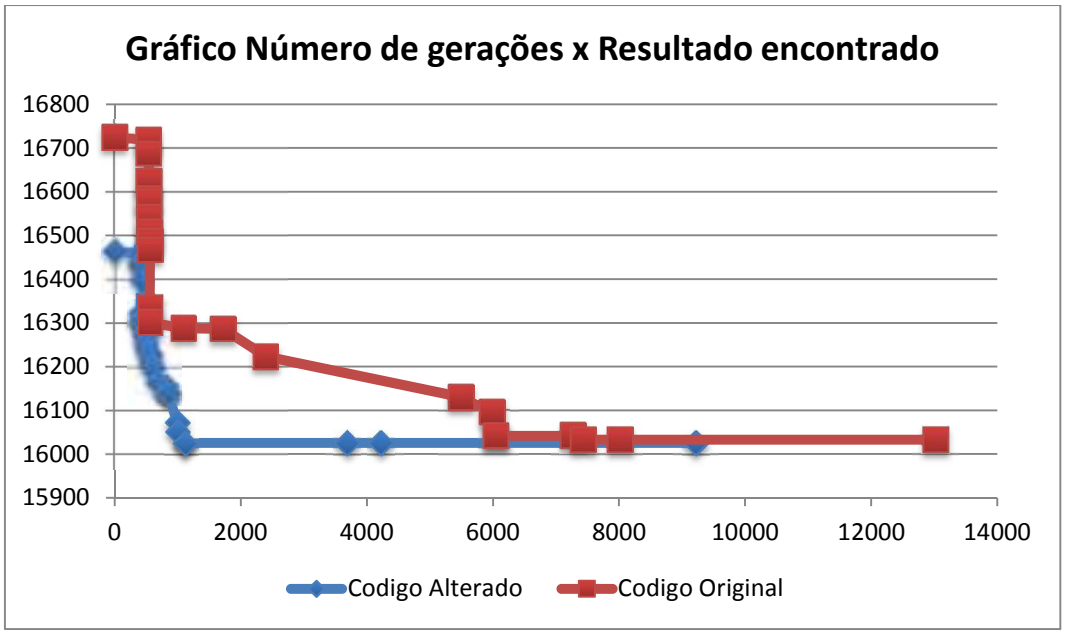

Figura 12 - Gráfico das execuções com resultados mais semelhantes para 5000 gerações – d198.

3.5.2.1.5. ANÁLISE DOS RESULTADOS DO PROBLEMA d198

Usando como critério de parada o numero de gerações sem encontrar novos resultados, a eficiência do código pode ser comprovada como demonstrado nos gráficos abaixo.

Na figura 7, se observa a aproximação da qualidade dos resultados de acordo com o número de gerações utilizadas como critério de parada. É possível perceber que quanto mais gerações esperando novos resultados, mais garantia se tem de resultados mais próximos, já que poucas gerações de espera diminuem as chances de convergências para melhores resultados.

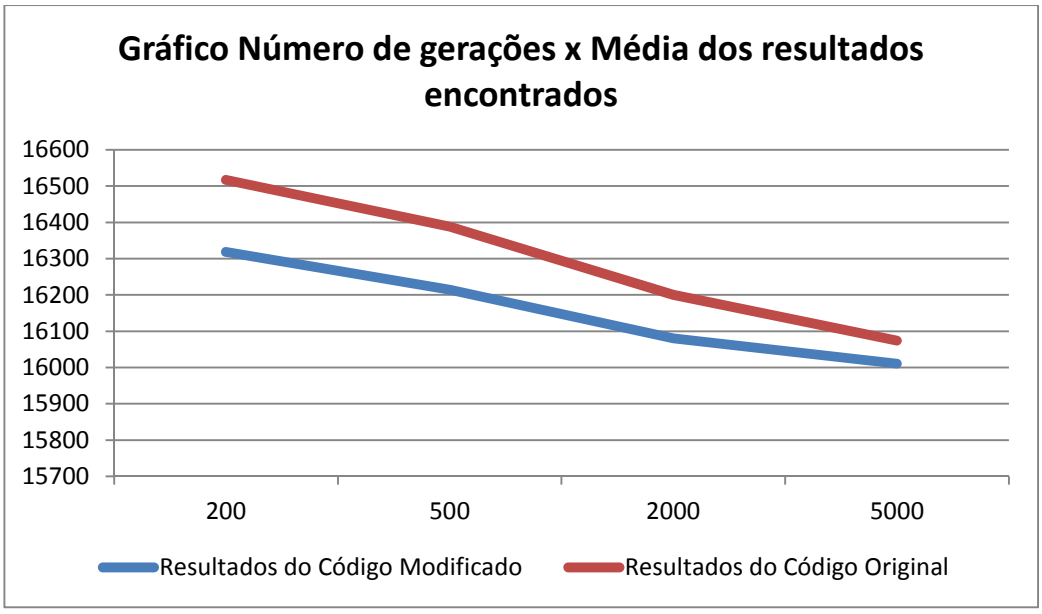

Figura 13 - Gráfico da média dos resultados encontrados de acordo com os critérios de parada – d198.

Na figura 8 é possível observar o ganho de tempo comparativo entre os códigos. Quanto mais gerações executadas, mais relevante se torna o ganho na execução paralela. Isso acontece porque o algoritmo paralelo consegue executar uma geração mais rápido que o algoritmo serial e quanto mais execuções, maior será o tempo ganho no processamento paralelo.

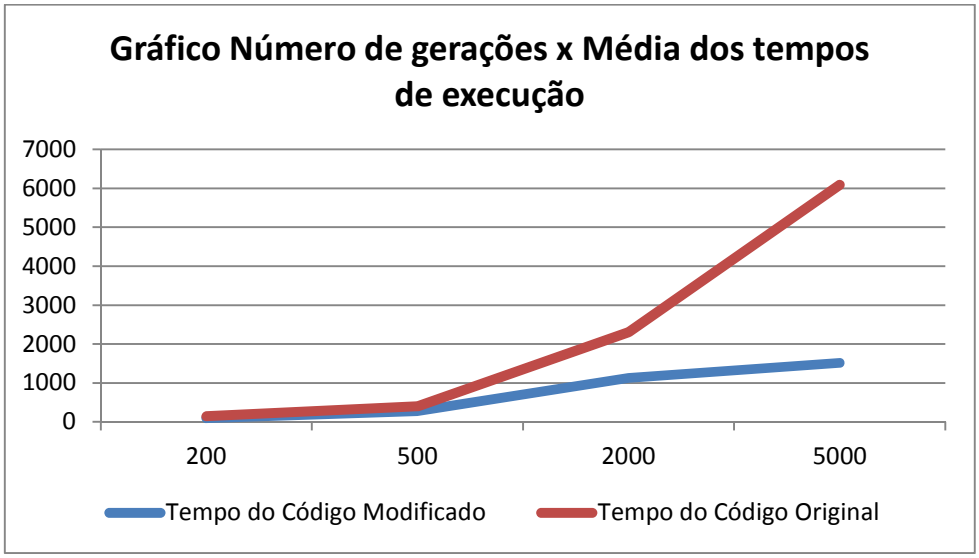

Figura 14 - Gráfico da média dos tempos de acordo com os critérios de parada – d198.

3.5.2.2. RESULTADOS DO PROBLEMA a280

3.5.2.2.1. RESULTADOS PARA 200 GERAÇÕES

O primeiro teste do problema a280, também foi executado com o critério de parada de 200 gerações sem novos filhos e, de acordo com a tabela 4, a média dos resultados teve uma diferença de 3,59% e ganho de 1,36x em relação ao código original, uma diferença levemente maior na qualidade dos resultados em relação ao problema d198. Diferença esperada considerando que o problema anterior contém uma quantidade menor de pontos cartesianos.

Tabela 8 – Média dos resultados encontrados para 200 gerações – a280.

	Código Original	Código Modificado	Diferença
Média dos Resultados	2821	2723	3.59%
Média dos Tempos	383s	281s	136.2%
Desvio padrão	7,2	50,1	85.5%

3.5.2.2.2. RESULTADOS PARA 500 GERAÇÕES

O segundo teste do problema a280 foi executado com o critério de parada de 500 gerações sem novos filhos e, de acordo com a tabela, a média dos resultados teve uma diferença de 2,22% e ganho de desempenho de 1,12x, mostrando um leve ganho de semelhança e desempenho em relação ao primeiro teste.

Tabela 9 – Média dos resultados encontrados para 500 gerações – a280.

	Código Original	Código Modificado	Diferença
Média dos Resultados	2742	2682	2,22%
Média dos Tempos	856s	759s	112.7%
Desvio padrão	51,7	39.5	30.9%

3.5.2.2.3. RESULTADOS PARA 2000 GERAÇÕES

Em seguinte, o teste foi executado com o critério de parada de 2000 gerações sem novos resultados melhores, no final, a média dos resultados se aproximou da semelhança com uma diferença de 2,325% o ganho de velocidade aumentou para 2,87x.

Tabela 10 – Média dos resultados encontrados para 2000 gerações – a280.

	Código Original	Código Modificado	Diferença
Média dos Resultados	2732	2682	1.859%
Média dos Tempos	8805s	3060s	287.7%
Desvio padrão	44.0	40.8	7.8%

3.5.2.2.4. RESULTADOS PARA 5000 GERAÇÕES

Por último, o teste foi executado com o critério de parada de 5000 gerações sem novos resultados melhores, onde a média dos resultados teve uma diferença de 0,737% que mais um vez comprova a proximidade dos resultados.

Tabela 11 – Média dos resultados encontrados para 5000 gerações – a280.

	Código Original	Código Modificado	Diferença
Média dos Resultados	2709	2689	0.737%
Média dos Tempos	11033s	4375s	252.2%
Desvio padrão	44.0	27.0	62.8%

O gráfico 14 mostra duas execuções com convergências a resultados semelhantes para observar o comportamento.

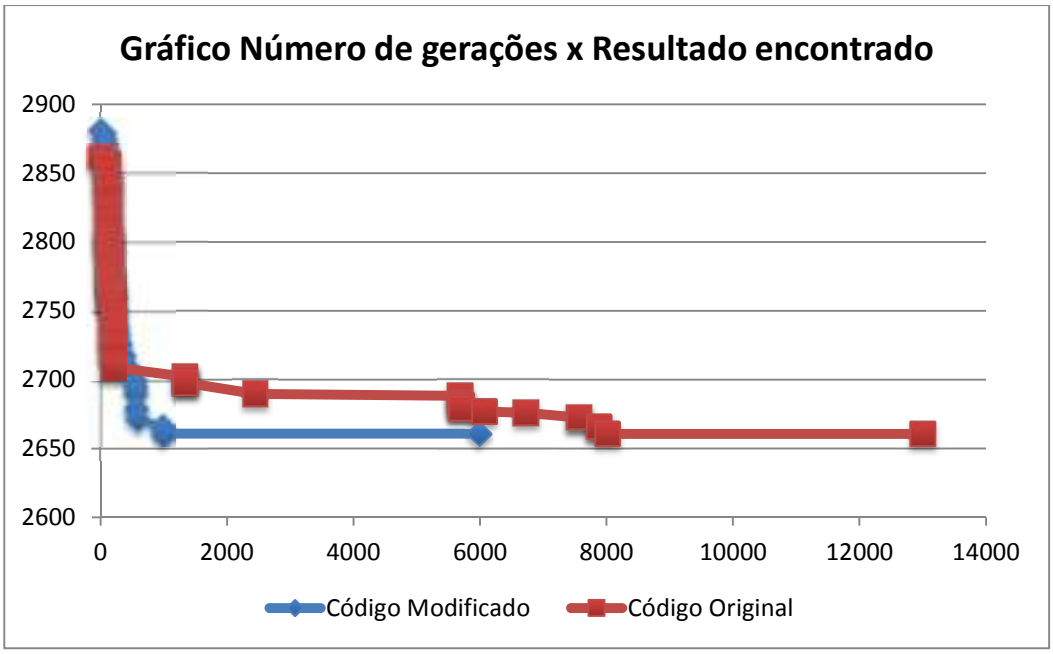

Figura 15 - Gráfico das execuções com resultados mais semelhantes para 5000 gerações – a280.

3.5.2.2.5. ANÁLISE DOS RESULTADOS DO PROBLEMA a280

Mais uma vez, usando como critério de parada o número de gerações sem encontrar novos resultados, a eficiência do código pode ser comprovada como demonstrado nos gráficos abaixo.

Na figura 7, se observa a aproximação da qualidade dos resultados de acordo com o número de gerações utilizadas como critério de parada. É possível perceber que a tendência dos resultados do problema d198 permanecem, e quanto mais gerações esperando novos resultados, mais garantia se tem de resultados mais próximos, já que poucas gerações de espera diminuem as chances de convergências para melhores resultados.

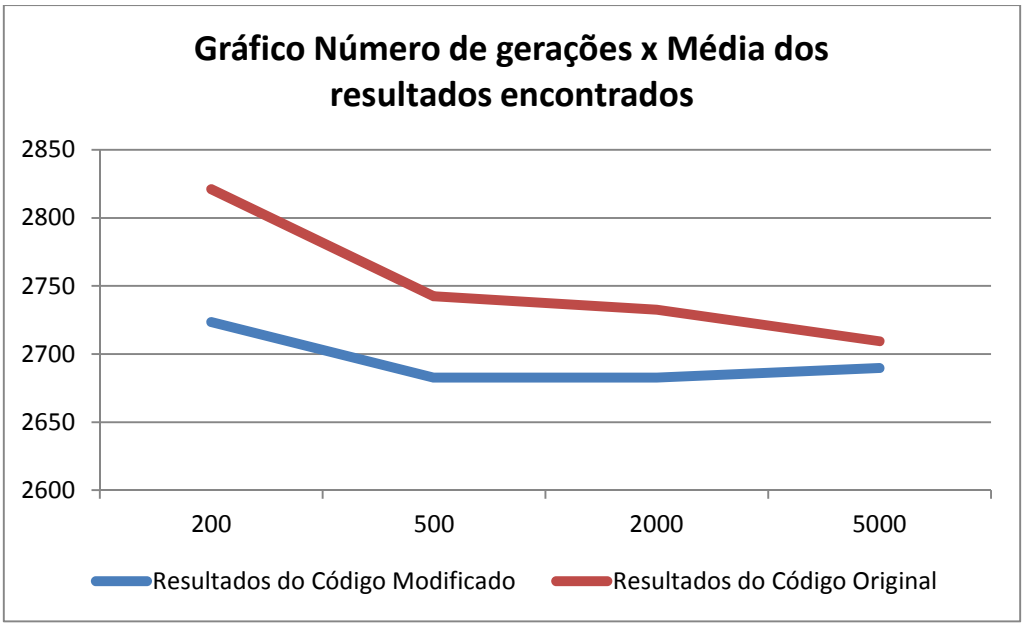

Figura 16 - Gráfico da média dos resultados encontrados de acordo com os critérios de parada – a280.

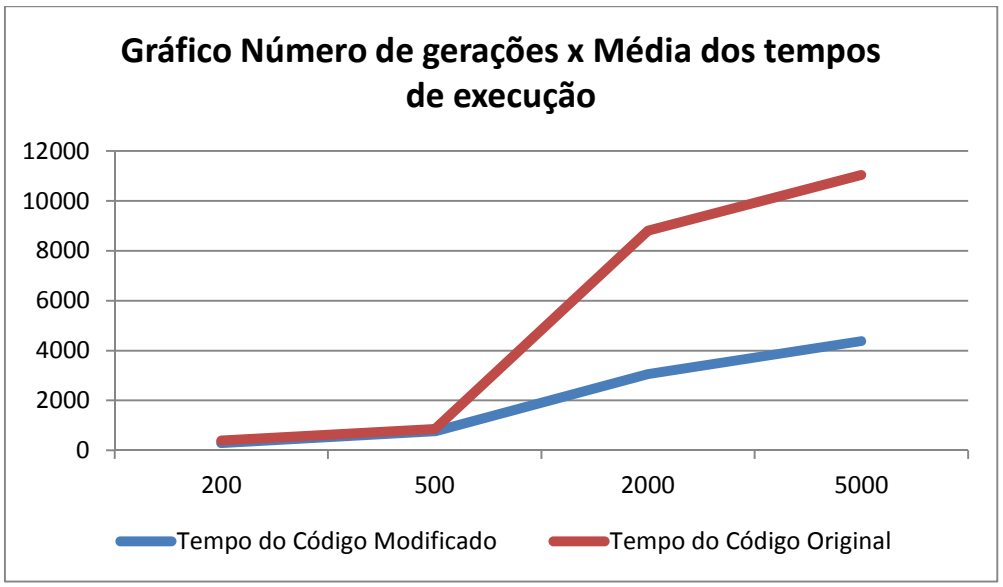

Figura 17 - Gráfico da média dos tempos de acordo com os critérios de parada – a280.

4. CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho foi implementado a paralelização de um algoritmo genético hibrido montado por Sena (2012) para a solução do problema do caixeiro viajante com o intuito de otimizar o processo de procura por melhores resultados, propondo as seguintes modificações:

- Criação de threads independentes durante a geração de cada filho;
- Adaptação da lógica de geração de novos filhos para melhorar o desempenho do algoritmo.

Com os resultados encontrados no capitulo 3, comprova-se a eficácia do método proposto quando comparado ao método original para os modelos de placa de circuito impresso d198 e a280 do TSPLIB, onde os parâmetros utilizados nos testes foram até 2,88 vezes mais rápidos em relação aos tempos de execução do algoritmo de Sena (2012) e mostram que podem ser melhores em instâncias maiores.

A paralelização do código se mostrou promissora na execução em máquinas com maior quantidade de núcleos e memória. Uma maior quantidade de processadores poderá gerenciar melhor as threads e consequentemente suportar populações maiores na inicialização e execução do problema, além disso, o desempenho na execução geral do problema mostrou a possibilidade da flexibilização do algoritmo para instâncias maiores.

Em relação ao código original, o algoritmo paralelo é mais eficaz nos primeiros minutos de execução. Como a convergência tem a tendência de acontecer mais cedo, o algoritmo paralelo ganha vantagem em relação ao serial, porém, em espaços maiores de tempo o algoritmo serial consegue diminuir a diferença por conseguir mais tempo para achar o resultado já encontrado pelo paralelo.

Como sugestão para trabalhos futuros pode ser feito um estudo do algoritmo paralelo rodando em máquinas de grande porte, como uma maquina virtual da Amazon ou Azure. Além disso, podem ser feitas melhorias dentro do próprio algoritmo genético, como a paralelização da inicialização da população, melhoria no algoritmo de cruzamento e ajustes de desempenho.

5. REFERÊNCIAS

BORATTO, Murilo. Modelos Paralelos para la Resolución de Problemas de Ingeniería Agrícola. Tesis Doctoral. Universitat Politècnica de València. Departamento de Sistemas Informáticos y Computación. 2014.

ANCĂU, M. **The optimization of printed circuit board manufacturing by improving the drilling process productivity**. Computer & Industrial Engineering. p. 279-294, 2008.

FOSTER, R. **Programação multi-threaded escalonável com pools de threads.** MSDN Magazine. Disponível em: <http://msdn.microsoft.com/pt-br/magazine/gg232758.aspx>. Acesso em março. 2014.

FRUMKIM, Michael, MSCHULTZ, Matthew. JIN, Haoqiang. YAN, Jerry. **Implementation of NAS Parallel Benchmarks in Java**. NASA Ames Research Center. EUA.

GORDON, S. WHITLEY, D. **Serial and Parallel Genetic Algorithms as Function Optimizers**. Department of Computer Science. The 5th International Conference on Genetic Algorithms. Urbana – Champaign, Illinois. Setembro de 1993.

HASSANI, A. TREIJS, J. **An Overview of Standard and Parallel Genetic Algorithms.** Mälardalen's University. Västerås, suíça. 2007.

JAYALAKSHMI, G. A.; SATHIAMOORTHY S.; RAJARAM R. A Hybrid Genetic Algorithm – A New Approach to Solve Traveling Salesman Problem. **International Journal of Computational Engineering Science.** 2001.

LIM, Wei Chen Esmonde, KANAGARAJ, G. PONNAMBALAM, S. G. **The Scientific World Journal.** PCB Drill Path Optimization by Combinatorial Cuckoo Search Algorithm. 2014. Disponível em <http://www.hindawi.com/journals/tswj/2014/264518/>. Acessado em Abril 2014.

MACHADO, B. **Um algoritmo genético para otimização do processo de perfuração de placas de circuito impresso**. 2011. Monografia (Graduação do curso de sistemas de informação) – CSI, Universidade Estadual da Bahia, Bahia. 2011.

MAREDIA, **A. History, Analysis, and Implementation of Traveling Salesman Problem (TSP) and Related Problems**. Department of Computer and Mathematical Sciences, University of Houston-Downtown. 2010.

ORACLE. **The Java Tutorials – Parallelism**. Disponível em: <http://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html>. Acesso em Abril 2014.

SILVEIRA, G. **Concorrência ou paralelismo: Threads, Processes, Fibers e Actors**. Caelum Ensino e Inovação.
Disponível em: <http://blog.caelum.com.br/concorrencia-ou-paralelismo-threads-processes-fibers-e-actors/>. Acesso em março. 2014.

SMITH, L. A. BULL, J, N. **A Parallel Java Grande Benchmark Suite**. The University of Edinburgh. Scotland, U.K. 2011.