Algoritmo Genéticos Paralelo: uma abordagem hierárquica

Derik Evangelista Rodrigues da Silva[1], **Raphael Henrique Ferreira de Andrade**[1], **Eduardo Spinosa**[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19081 – 81531-980 – Curitiba – PR – Brasil

{dersilva, rhfandrade, spinosa}@inf.ufpr.br

Abstract. *The Genetic Algorithm has been successfully applied in several applications, in different domains. However, there are some problems that may make it impratical to use: a) when the population has to be arbitrary large, becoming impossible to execute in a single machine; b) very large execution time; c) difficulty to escape of suboptimal areas in the search space and, consequently, turning hard to find the global optimal. This problemas can be solved (or, at least, minimized) using a parallel version of the algorithm. In this paper, we explore some of the most common parallelization tecniches applied to genetic algorithm: the multiple population model and the master-slave model. Further, we explore a hierarchical model, that parallelized the process in two levels: in the superior layer, it uses a multiple populations approach and in the lower layer, a master-slave approach. The results are then compared.*

Resumo. *O Algoritmo genético tem tido sucesso e dado bons resultados em muitas aplicações de diferentes domínios do conhecimento. Entretanto, existem alguns problemas que podem tornar inviáveis o uso deste algoritmo: a) a necessidade de uma população arbitrariamente grande, podendo impossibilitar a utilização eficiente; b) tempo de execução muito grande; c) dificuldade do algoritmo de escapar de regiões sub-ótimas no espaço de busca, impossibilitanto a descoberta da solução ótima. Estes problemas podem ser solucionados (ou minimizados) utilizando paralelismo. Neste trabalho, exploramos algumas das técnicas mais comuns de paralelização: utilização de multiplas populações (multi-demme) e paralelização dos cálculos de fitnesse e geração de nova população (master-slave). Além disso, exploramos um método hierárquico, que paraleliza o processo em dois níveis: no nível superior, paraleliza-se utilizando um algoritmo multi-demme, enquanto no nível inferior, utiliza-se um modelo master-slave. Os diferentes algoritmos são comparados no problema de minimização de funções.*

1. Introdução

Algoritmo Genetico (*Genetic Algorithm* – GA) são algoritmos de busca inspirados no processo de evolução e seleção natural [Goldberg 1989] e tem tido grande sucesso em problemas de busca e de otimização, principalmente quando o espaços de busca é grande, complexo ou pouco conhecido, onde métodos de buscas convencionais (enumerativos, heurísticos, ...) não são apropriados [Herrera et al. 1998].

Um GA sequencial inicia-se gerando um conjunto de indivíduos para formar uma população inicial. Cada indivíduo representa uma possível solução do problema. Usando

uma função de avaliação (chamada de função *fitness*), mede-se a qualidade de cada indivíduo desta população. O cálculo do *fitness* é, geralmente, o processo mais custoso de um GA [Nowostawski and Poli 1999]. Seleciona-se aleatoriamente, então, um subconjunto de indivíduos desta população e neste é aplicado operadores estocásticos de seleção, mutação e cruzamento. Por fim, os indivíduos menos adaptados (ou seja, com pior *fitness*) são descartados, para dar lugar a indivíduos mais bem adaptados.

Apesar do sucesso em muitas aplicações em diferentes domínios, existem, de acordo com [Nowostawski and Poli 1999], algums problemas que podem ser resolvidos com o uso de um Algoritmo Genético Paralelo (*Parallel GA* – PGA):

- Para alguns tipos de problemas, o tamanho da população precisa ser muito grande, requerendo, consequentemente, uma grande quantidade de memória, podendo impossibilitar a execução eficiente em uma única máquina.
- O cálculo do *fitness* consome muito tempo. Há registros na literatura de uma única execução consumindo mais de 1 ano de CPU.
- GA's sequencias podem ficar presos em regiões sub-ótimas, ficando impossibilitados de encontrar uma melhor solução. PGA's podem buscar em multiplos subespaços de busca em paralelo, e tem menos chance de ficar preso em regiões sub-ótimas.

O motivo mais importante para se estudar PGAs, ainda segundo [Nowostawski and Poli 1999], é que em muitos casos eles tem uma melhor performance do que os sequenciais, mesmo quando o paralelismo é simulado em uma máquina convencional.

Este trabalho tem como objetivo comparar três tipos de arquiteturas de PGAs: múltiplas populações, arquitetura mestre-escravo e um híbrido de ambas, ou seja, uma combinação de múltiplas populaçõoes com mestre-escravo, aplicadas a otimização de funções. Além disso, compararemos os resultados com um GA sequencial convencional.

2. Revisão de literatura

O Algoritmo Genético foi desenvolvido por John Holland na Universidade de Michigan, em 1970 [Holland 1975], inspirado no processo de seleção natural e evolução, e apresenta uma alternativa as técnicas clássicas de otimização, usando buscas aleatórias dirigidas para localizar soluções ótimas em espaços de buscas complexos [Srinivas and Patnaik 1994]. O objetivo original de Holland não era construir um algoritmo que resolvesse um problema específico, mas formalizar o estudo do fenômeno de adaptação da mesma forma que este acontece na natureza e desenvolver mecanismos de importar este comportamento em sistemas computacionais [Michell 1998].

Tendo sua inspiração na biologia, alguns termos desta área são usados para descrever o GA [Luke 2009]:

Indivíduo Solução candidata;
População Conjunto de indivíduos;
Filhos e Pais Um filho é uma cópia perturbada de seu pai (ambos são indivíduos);
***Fitness* (Adaptabilidade)** Medida de qualidade de dada solução;
Função de *Fitness* Função de qualidade.
Seleção Escolha de indivíduos, baseado em seu *fitness*;

Mutação Pequena perturbação na solução;
Recombinação / Cruzamento Grande perturbação na estrutura do indivíduo. Geralmente gera dois filhos recombinando a estrutura de seus pais.
Genoma / Genótipo Estrutura do indivíduo;
Geração População gerada em cada ciclo do algoritmo, que envolve as funções e transformações previamente definidas.

O algoritmo apresentado em [Holland 1975] é usualmente chamado de canônico [Yang 2002] ou Algoritmo Genético Simples (SGA) [Srinivas and Patnaik 1994] e trabalha, essencialmente, com indivíduos sendo um vetor de bits, ou seja, a solução é codificada em termos de 0 e 1. Como método de seleção, o SGA usa o esquema de *roleta*, onde um determinado indivíduo tem mais chance de ser escolhido para procriar dependendo de seu *fitness* calculado.

Algumas variações foram apresentadas, como a inclusão de elitismo [De Jong 1975], que consiste em manter um número de indivíduos com melhor *fitness* de uma geração para outra, o *Steady-State Genetic Algorithm* [Whitley et al. 1988], que atualiza a população assim que os filhos são gerados, descartando-os ou inserindo-os no lugar de alguns de indivíduos piores da população e o *Tree-Style Genetic Programming Pipeline*, que utiliza uma forma diferente de procriação: com 90% de probabilidade, dois pais serão selecionados e será efetuado o cruzamento convencional e, por outro lado, com 10% de probabilidade, será selecionado apenas um pai, que será copiado para a nova população. Existem versões, também, que se preocupam em adaptar as taxas de cruzamento e mutação em tempo de execução e abordagens híbridas, como efetuar uma busca local em cada indivíduo, usando outro algoritmo [Bersini and Renders 1994] [Katare et al. 2004].

Muitos dos algoritmos evolutivos são inerentemente paralelos [Høverstad 2010], pela natureza independente de suas operações [Alba and Troya 1999], e o paralelismo surge como uma alternativa para melhorar a eficiência dos GAs.

Os algoritmos genéticos paralelos (PGA) não são apenas versões paralelas de GA sequenciais. De fato, na maioria dos casos, o todo (PGA) tem melhor performance que a soma das sub-partes que o compões [Alba and Troya 1999].

A maneira com que os GAs são paralelizados depende dos seguintes elementos [Nowostawski and Poli 1999]:

- Como é calculado a função *fitness* e como a mutação é aplicada;
- Se multiplas subpopulações *demes* são usadas;
- Se multiplas populações são usadas e como os indivíduos interagem e;
- Como a seleção é aplicada (globalmente ou localmente);

Dos parâmetros acima, pode-se extrair quatro tipos principais de PGA [Cantú-Paz 1998]:

- Mestre-escravo *(Master Slave)*: globais de uma única população;
- Única população com paralelização *fine-grained*;
- Múltiplas populações com paralelização *coarsed-grained* e;
- Combinação dos métodos acima

Onde *fine-grained* refere-se a algoritmos paralelos com frequente comunicação entre as partes, enquanto *coarse-grained* refere-se ao contrário.

No esquema mestre-escravo, usa-se uma única população e paraleliza-se os cálculos de *fitness* nos processodores. Os *fine-grained* PGA (FGPGA) são usados em máquinas massivamente paralelas e consistem de uma única população espacialmente estruturada e os *coarsed-grained* PGA (CGPGA, também chamados de GA Distribuídos) consistem de múltiplas populações (também chamado de *demes* ou *ilhas*) que evoluem em paralelo e trocam indivíduos ocasionalmente (esta troca é chamada de *migração*). Aos esquemas que combinam multiplas populações com mestre-escravo ou FGPGA dá se o nome de *hierárquicos* (HPGA).

O primeiro PGA foi proposto em 1987 por Pettey, Leuze e Grefenstette e utilizava o esquema de multiplas populações, migrando sempre o melhor [Alba and Troya 1999]. [Tanese 1989] utilizou o esquema de populações distribuídas e obteve bons resultados com migração de 20% da população a cada 20 gerações. [Gorges-Schleuter 1989] usavam um FGPGA e aplicava *hill climbing* caso não obtivesse nenhuma melhora em um determinado número de gerações. [Adamidis and Petridis 1996] utilizam múltiplas ilhas e é particularmente interessante pois cada uma das ilhas possuem suas próprias probabilidades de mutação, cruzamento e operadores especializados. [Wilson and Banzhaf 2010] apresentam um PGA que faz uso dos diversos núcleos das placas de vídeo de consoles de video-game, obtendo bons resultados em tempo de execução reduzido.

[Lim et al. 2007] apresentam um arcabouço para desenvolvimento de HPGA com suporte à computação em grade, e sugeriram, através de um estudo empírico utilizando problemas de *benchmark* e problemas reais, que pode-se obter uma aceleração desde que os limites do custo da função *fitness*, tamanho do *cluster* e *overheads* na comunicação sejam satisfeitos.

[Wen-hua et al. 2007] sugere um novo PGA, chamado de *asynchronous heterogeneos hierarchical parallel genetic algorithm* (AHHPGA) aplicado para a resolução do *Redundancy Allocation Problem*, e usa um modelo *coarsed-grained* como camada superior e um modelo *fine-grained* na camada inferior. Este modelo proposto possui multiplas populações heterogêneas, com diferentes níveis de exploração das soluções e diferentes topologias em cada subpopulação. A migração acontece de forma assíncrona, tanto o envio quanto a recepção de novos indivíduos. Os resultados obtidos por este modelo foram ligeiramente melhores do que os obtidos por um GA tradicional, mas a faltou uma análise estatística elaborada para corroborar o modelo.

[Lee et al. 2009] apresentam um modelo hierarquico com competição justa, ou seja, a população é divida em camadas (ou classes) de acordo com o *fitness* do indivíduo e estes só competem com outros da mesma camada. Indivíduos mudam de classe quando atingem um grau de aptidão superior ao limiar de aceitação da outra classe. Desta forma, mantém-se uma maior diversidade na população, comprovada pelos testes realizados.

[Benitez and Lopes 2010] apresentam um modelo hierárquico (multiplas populações *coarsed-grained* no nível mais alto e, no nível mais baixo, mestre-escravo) para o problema de desdobramento de proteínas, obtendo melhores resultados em 7 de 10 em comparação com o *benchmark* de outro PGA não-hierárquico.

3. Experimentos

3.1. Problema

Três funções de *benchmark* da literatura foram escolhidas, duas definidas em [De Jong 1975] e a função de Rastrigin. Aplicamos estas funções ao problema de minimização. As definições das funções foram extraídas de [Temby et al. 2005].

A primeira função de [De Jong 1975], a Função Esfera (*Sphere Function*), é relativamente simples. Não possui ótimos locais e é facilmente resolvida por um GA. O valor de x varia entre -5.12 e $+5.12$.

$$f(Sphere) = \sum_{i=1}^{n} x_i^2 \tag{1}$$

A segunda função utilizada foi a Função *Rotated Hyper-ellipsoid function* é continua, convexa e unimodal. O valor de x varia entra -5.12 e $+5.12$.

$$f(Step) = \sum_{i=1}^{n} \sum_{j=1}^{i} x_j^2 \tag{2}$$

A função de Rastrigin é outro problema muito difícil já que define um espaço de busca muito grande, com $n = 20$ com valores entre -5.12 e $+5.12$. Além disso, possui muitos ótimos locais. Esta combinação faz com que muitos algoritmos possuam grande dificuldade para encontrar a solução ótima. As variáveis A e ω controlam a frequencia e modulação do espaço de busca. No estudo feito, $A = 10$ e $\omega = 2\pi$;

$$f(Rastrigin) = 10 \times n + \sum_{i=1}^{n} \left[x_i^2 - 10 \times cos(2\pi x_i) \right] \tag{3}$$

Os códigos foram desenvolvido em C, com uso da biblioteca *OpenMP* para paralelização. Todos os testes foram executados em uma máquina Linux de 64bits com 4 *cores* de processamento.

3.2. Representação e Parâmetros

Os casos de testes consistiam em achar um conjunto de elementos $(x_0, x_1, x_2, \ldots x_n)$ que minimizasse as funções definidas na seção 3.1.

Utilizamos o método definido em [Srinivas and Patnaik 1994, p. 3] para representar os números em ponto flutuante. Substituimos o valor do número pelo seu equivalente em inteiro, multiplicando por 10^p, onde p é o grau de precisão (casas decimais). Este número inteiro é então representado em uma *string de bits*, em binário. Como as funções definidas em 3.1 possuem domínio entre o intervalo -5.12 e $+5.12$, utilizamos uma *string de bits* v de 10 posições, sendo a posição v_0 referente ao sinal, para cada número que queremos representar.

O cruzamento dá-se de forma convencional, pelo método *One-point Crossover*: considere dois indivíduos v e q e m o tamanho do indivíduo; os indivíduos z e w, resultados do cruzamento de v e q serão:

$$z = [v_0, v_1, v_2, \ldots, v_n, q_{n+1}, q_{n+2}, \ldots q_m]$$

	Método	**Taxa**
Mutação	Negação de *bits*	1%
Cruzamento	*One-point crossover*	100%
Seleção	Torneio	
Elitismo	Apenas o melhor	

Tabela 1. Métodos dos GAs

	Função		
Parâmetro	**F1**	**F2**	**F3**
Tamanho do indivíduo	20	20	200
Tamanho da população	400		
Número de Gerações	3000		

Tabela 2. Parâmetros dos GAs

$$w = [q_0, q_1, q_2, \ldots, q_n, v_{n+1}, v_{n+2}, \ldots v_m]$$

Onde n é um valor escolhido de forma aleatória, $0 \leq n < m$. A taxa de cruzamento é de 100%.

A mutação também é realizada da forma clássica: seja v um indivíduo. Para todo $x_i \in v$, com probabilidade p, faça $x_i = \overline{x_i}$, onde:

$$\overline{x_i} = \begin{cases} 0 & : x_i = 1 \\ 1 & : x_i = 0 \end{cases}$$

e a propabilidade de mutação $p = 0.01$, ou seja, 1%.

O método de seleção para o cruzamento escolhido foi o *Torneio*. A cada iteração, são selecionados quatro indivíduos da população aleatoriamente e os dois indivíduos com maior *fitness* são então escolhidos para gerar os filhos. A cada geração, toda a população, salvo o melhor indivíduo, é substituida pela nova população gerada.

O tamanho da população foi fixado em 400 indivíduos. O tamanho do indivíduo depende do n das funções. Para as funções 1 e 2, $n = 5$, logo, o tamanho do indivíduo é 20 (2×10, já que cada número ocupa 10 *bits*). Já para a função 3, $n = 20$, fazendo com que o tamanho do indivíduo seja 200 (20×10). O número total de gerações foi 3000.

As tabelas 1 e 2 resume os parâmetros e métodos acima explicitados.

3.3. Algoritmos

3.3.1. GA Sequencial

O GA sequencial foi desenvolvido conforme descrito em [Holland 1975], com adição do elitismo [De Jong 1975] de apenas um (o melhor da geração) indivíduo.

3.3.2. PGA Mestre-escravo

Na nossa implementação, foi paralelizado tanto a geração de uma nova população quanto o cálculo da função de *fitness* dos indivíduos gerados. A *thread master* distribui para

as *threads slaves* uma faixa de índices, onde serão armazenados os filhos gerados por cada uma destas *threads*. Após toda a população ter sido gerada, a *thread master* novamente distribui para as *slaves* uma faixa de índices para que estas calculem o *fitness* dos indivíduos dentro desta faixa.

É importante notar que, no primeiro caso, os indivíduos gerados pelas *slaves* são inseridos diretamente na nova população, enquanto que no cálculo de *fitness* cada *slave* retorna o melhor indivíduo encontrado naquela faixa de valores, e cabe a *master* decidir qual é o melhor indivíduo entre eles.

Os demais parâmtros permaneceram inalterados.

3.3.3. PGA Multiplas Populações

Na implementação do *Multi-Demme* PGA, cada *thread* possui uma instância do GA sequencial, com sua própria população. Neste caso, para que todas as populações se beneficiem da evolução de outra, implementamos a migração: considere o conjunto de populações $\mathrm{P} = \{p_0, p_1, \ldots, p_n\}$, onde p_i é a população da *thread* i. A cada g gerações, o melhor indivíduo da população p_i migra para a população p_{i+1} (no caso de $i + 1 = n$, o indivíduo migra para p_0).

Além disso, foi diminuido o número de gerações em cada sub-população proporcionalmente ao número de *threads* disponíveis, para que todos os GAs tivessem o mesmo gasto no cálculo da função *fitness*.

3.3.4. PGA Hierárquico

Utilizando como base o PGA descritos em 3.3.3 e em 3.3.2, foi implementado uma versão híbrida de ambos. O algoritmo hierárquico é separado em camadas: na camada mais externa (*top layer*) é executado exatamente o mesmo algoritmo descrito em 3.3.3. Entretanto, cada população executa o GA descrito em 3.3.3, ou seja, cada sub-população paraleliza a geração de uma nova população e do cálculo da função *fitness*.

Desta forma, procura-se acelerar a velocidade de execução (através do *master-slave*) ao mesmo tempo que explora-se melhor o espaço de busca (através do *multi-demme*).

3.4. Resultados

Os resultados podem ser analisados olhando os gráficos das 10 execuções dos GAs. Nos gráficos do GA *Multi-demme* e *Multi-demme master-slave* é moestrado apenas o melhor *fitness* de uma das populações, pela dificuldade encontrada em inserir todas as populações no mesmo gráfico.

Para a função 1, podemos perceber que o ganho das abordagens paralelas em relação ao sequencial não foi grande, pelo fato do problema ser relativamente simples. Ao analisar o ganho de tempo dos GAs com a abordagem *multi demme*, é importante lembrar que nestes foram feitas apenas 1/4 das gerações que nos outros (para manter o mesmo número de avaliações da função de *fitness*).

Para a função 2 o mesmo se aplica: a solução encontrada pelo sequencial foi equivalente a solução encontrada pelas abordagens paralelas. Entretanto, o ganho em tempo foi alto.

Já para a função 3, as abordagens com multiplas populações tiveram um resultado muito melhor do que o sequencial ou a paralelização *master-slave* (que apenas paraleliza as operações do GA sequencial, trazendo ganhos apenas temporais). Isso se deve ao fato desta função ser muito mais complexa que as anteriores, com um espaço de busca muito maior, permitindo que cada subpopulação explore uma região deste espaço. O tempo de execução da abordagem hierárquica foi pior do que o do GA *multi-demme* pois precisamos simular o paralelismo daquele, já que não tinhamos a disposição uma máquina com a quantidade de núcleos necessários para ter ganhos reais de tempo.

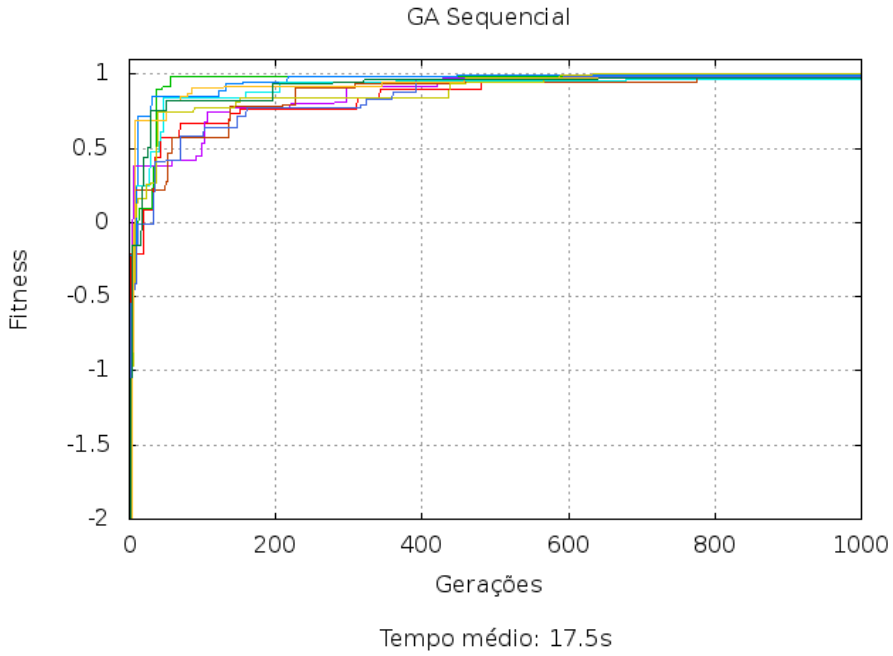

Figura 1. GA Sequencial - Função *Sphere*

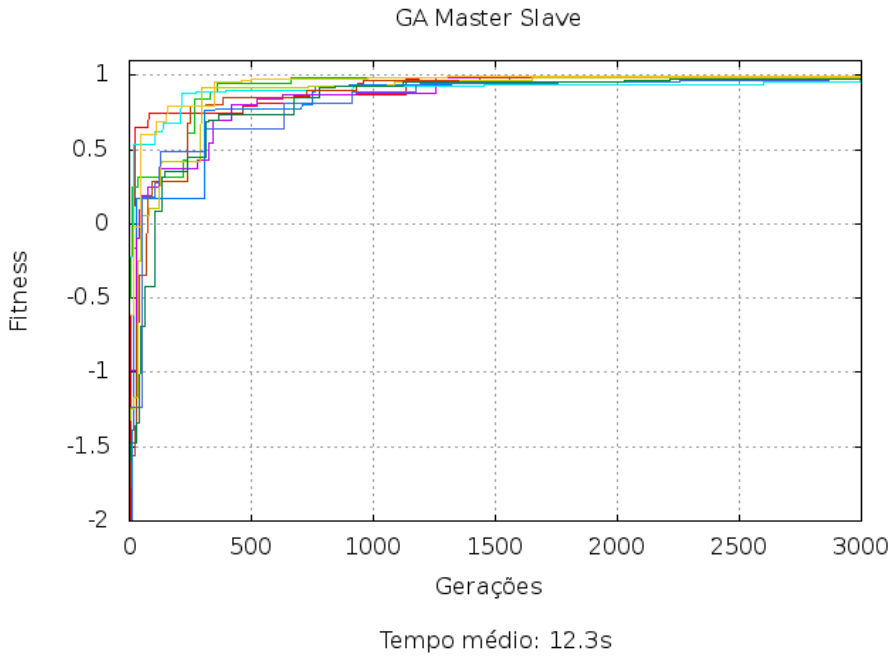

Figura 2. GA *Master-slave* - Função *Sphere*

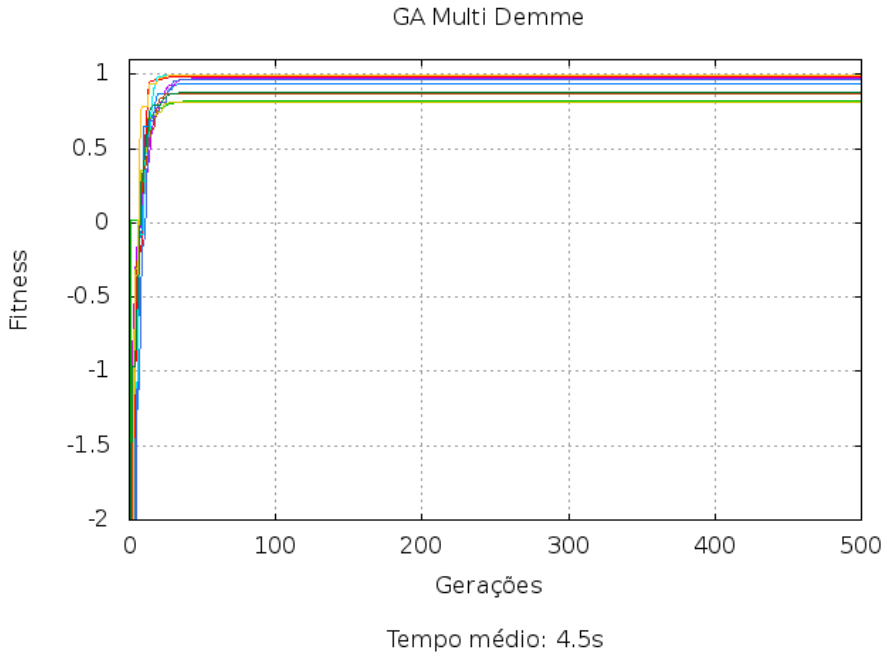

Figura 3. GA *Multi-demme* - Função *Sphere*

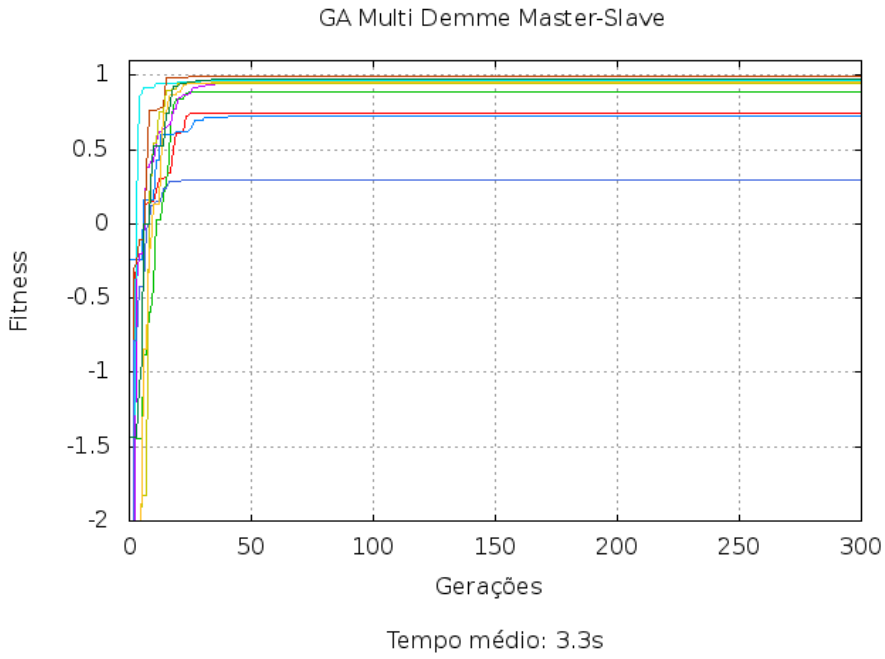

Figura 4. GA *Multi-demme Master-Slave* - Função *Sphere*

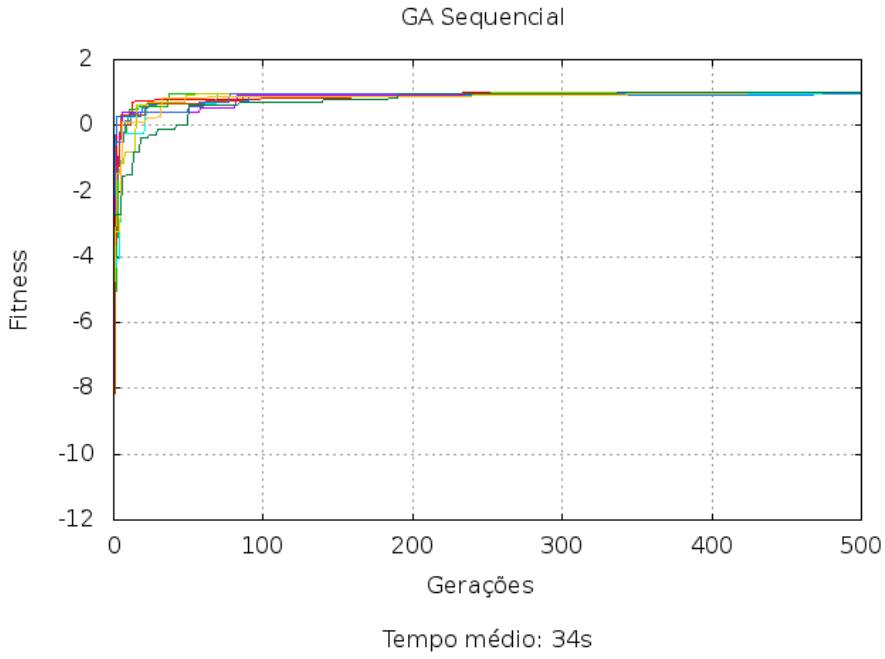

Figura 5. GA Sequencial - Função *Rotated Hyper-ellipsoid*

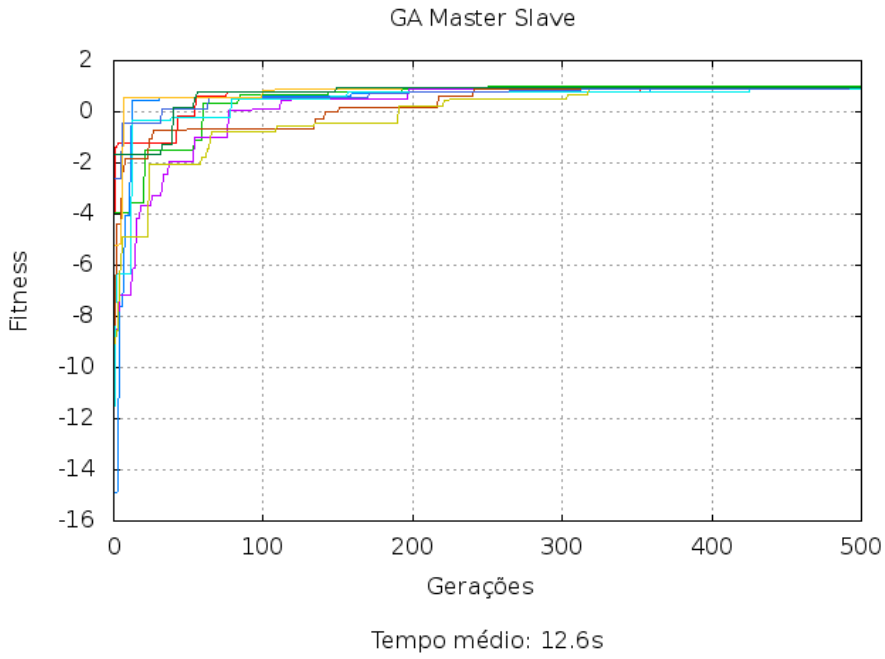

Figura 6. GA *Master-slave* - Função *Rotated Hyper-ellipsoid*

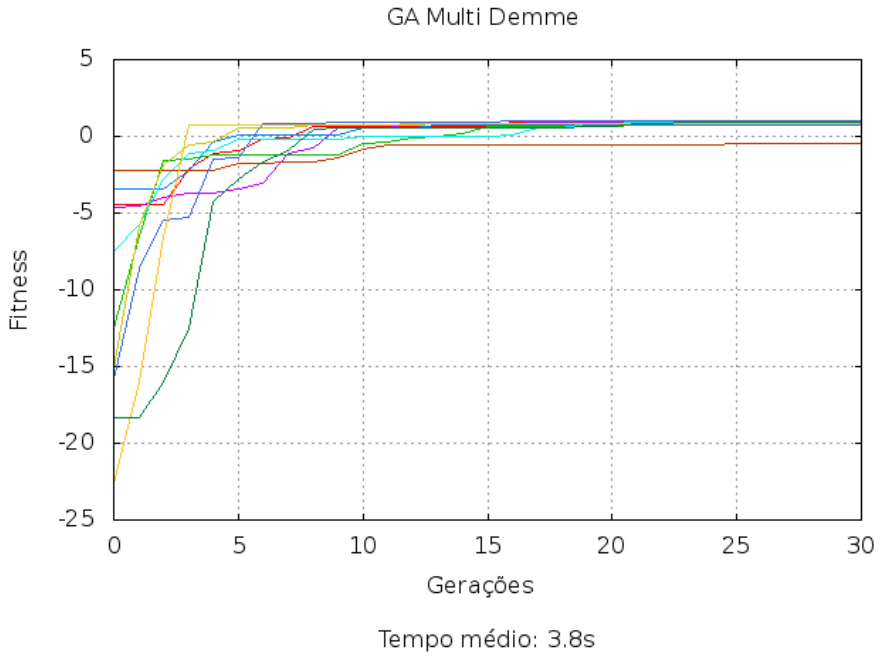

Figura 7. GA *Multi-demme* - Função *Rotated Hyper-ellipsoid*

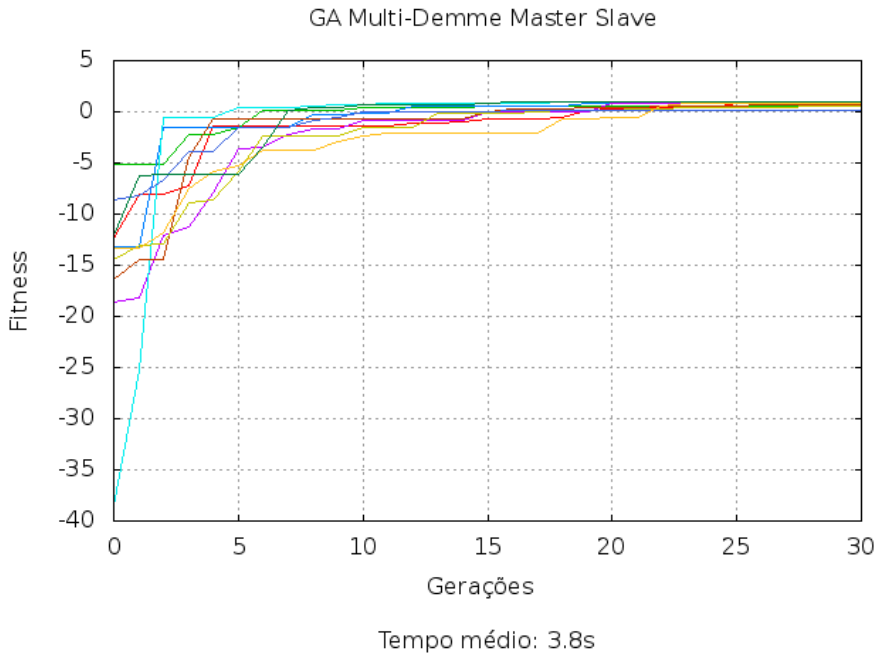

Figura 8. GA *Multi-demme Master-Slave* - Função *Rotated Hyper-ellipsoid*

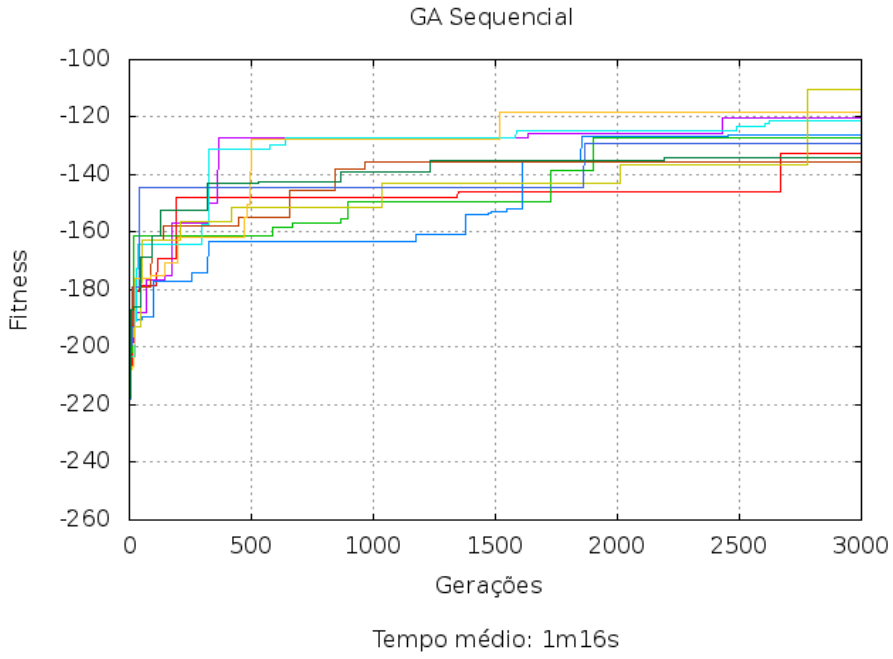

Figura 9. GA Sequencial - Função *Rastrigin*

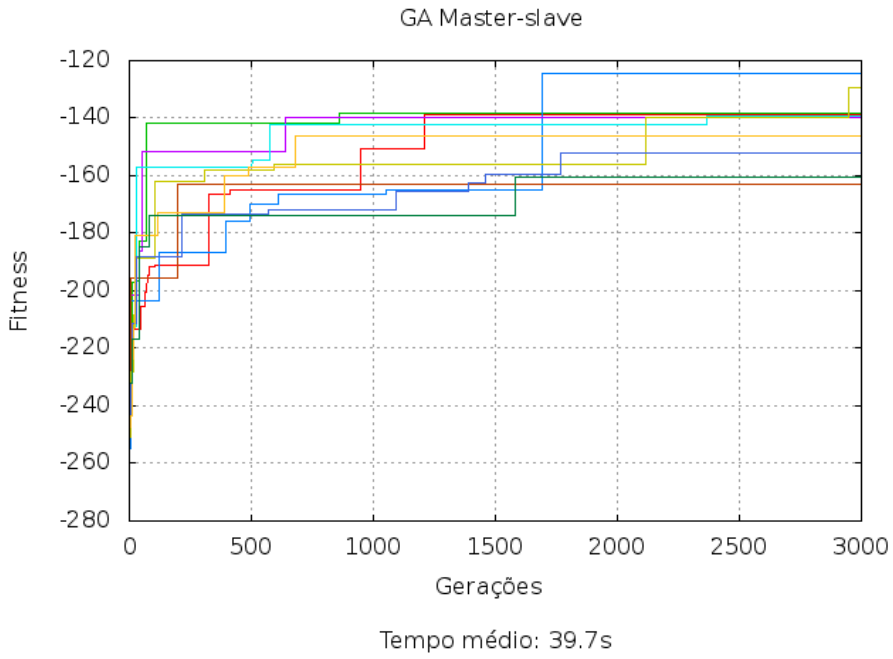

Figura 10. GA *Master-slave* - Função *Rastrigin*

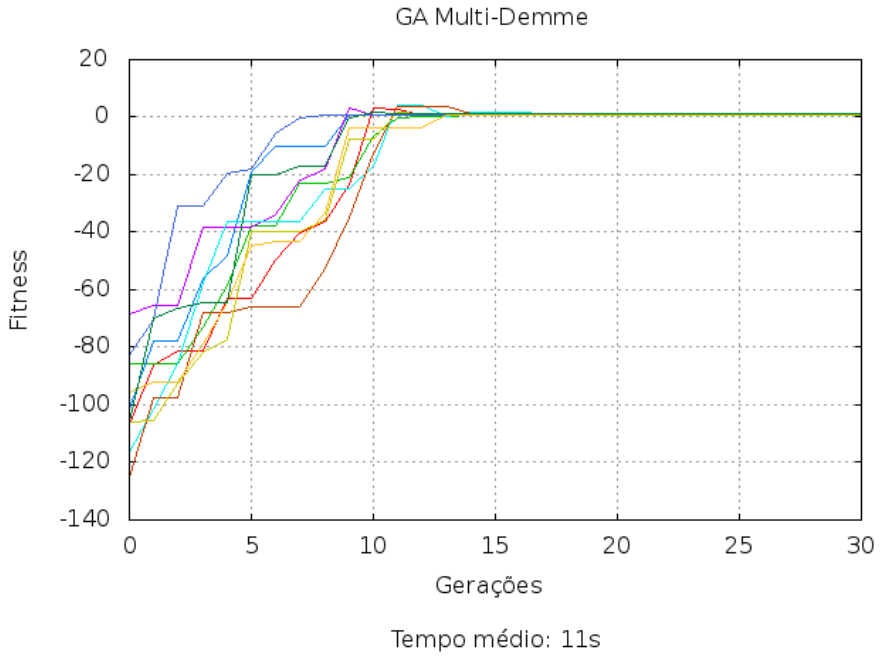

Figura 11. GA *Multi-demme* - Função *Rastrigin*

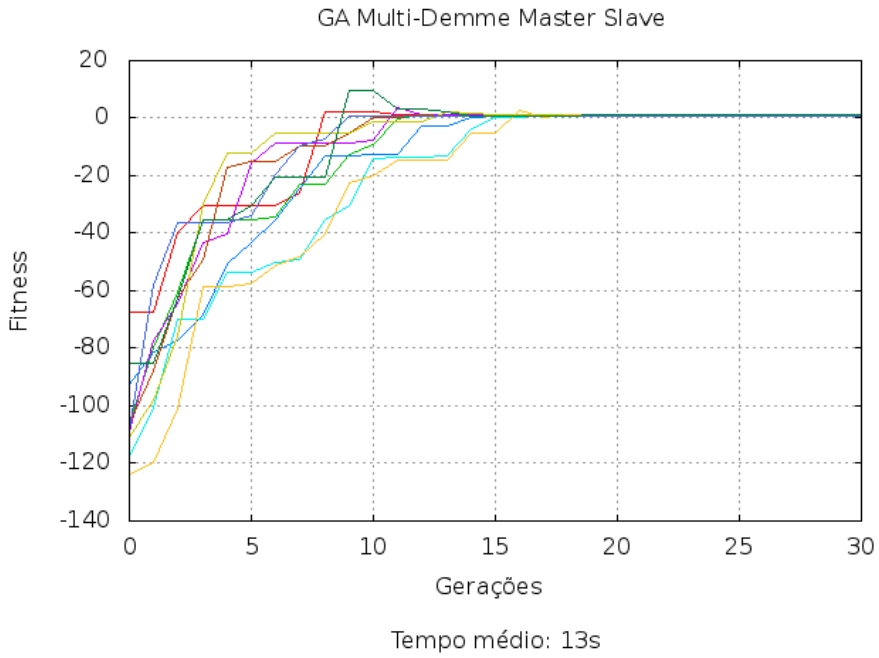

Figura 12. GA *Multi-demme Master-Slave* - Função *Rastrigin*

4. Conclusão

Com os experimento, pudemos observar que o paralelismo sempre traz ganhos em tempo de execução, pra qualquer uma das abordagens. Entretanto, para problemas simples, como a função 1 e mesmo a função 2 os ganhos não são suficientemente grandes para justificar a paralelização. Já para problemas mais complexos e com um espaço de busca grande, como a função 3, os métodos de paralelização aumentam muito a eficiência do algoritmo, apresentando soluções melhores e em menos tempo.

A falta de uma plataforma para testes maiores dificulta uma análise profunda dos ganhos quando combinados o GA *muti-demme* e *master-slave*, mas uma análise da diferença entre o GA sequencial e o *master-slave* nos permite afirmar que a abordagem hierárquica consegue encontrar melhores soluções (característica da paralelização *multi-demme*) em e em menos tempo de execução (característica do *master-slave*), quando comparado aos outros métodos aqui apresentados.

Referências

Adamidis, P. and Petridis, V. (1996). Co-operating populations with different evolution behaviours. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 188–191.

Alba, E. and Troya, J. M. (1999). A survey of parallel distributed genetic algorithms. *Complex.*, 4(4):31–52.

Benitez, C. and Lopes, H. (2010). Hierarchical parallel genetic algorithm applied to the three-dimensional hp side-chain protein folding problem. In *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on*, pages 2669–2676.

Bersini, H. and Renders, J.-M. (1994). Hybridizing genetic algorithms with hill-climbing methods for global optimization: Two possible ways. In *International Conference on Evolutionary Computation*, pages 312–317.

Cantú-Paz, E. (1998). A survey of parallel genetic algorithms. *CALCULATEURS PARALLELES*, 10.

De Jong, K. A. (1975). *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, Ann Arbor, MI, USA. AAI7609381.

Goldberg, D. (1989). *Genetic algorithms in search, optimization, and machine learning*. Artificial Intelligence. Addison-Wesley.

Gorges-Schleuter, M. (1989). Asparagos an asynchronous parallel genetic optimization strategy. In *Proceedings of the third international conference on Genetic algorithms*, pages 422–427, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Herrera, F., Lozano, M., and Verdegay, J. L. (1998). Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artif. Intell. Rev.*, 12(4):265–319.

Holland, J. (1975). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press.

Høverstad, B. A. (2010). Simdist: a distribution system for easy parallelization of evolutionary computation. *Genetic Programming and Evolvable Machines*, 11(2):185–203.

Katare, S., Bhan, A., Caruthers, J. M., Delgass, W. N., and Venkatasubramanian, V. (2004). A hybrid genetic algorithm for efficient parameter estimation of large kinetic models. *Computers & Chemical Engineering*, 28(12):2569 – 2581.

Lee, H., Hong, S., and Kim, E. (2009). Optimal classifier design method using hierarchical fair competition model based parallel genetic algorithm. In *ICCAS-SICE, 2009*, pages 2907–2910.

Lim, D., soon Ong, Y., and sung Lee A, B. (2007). Efficient hierarchical parallel genetic algorithms using grid computing. *Future Generation Computer Systems*, 23.

Luke, S. (2009). *Essentials of Metaheuristics*. Lulu. Disponível em: <http://cs.gmu.edu/∼sean/book/metaheuristics/>. Acesso em: 27/02/2013.

Michell, M. (1998). *An Introduction to Genetic Algorithms*. Complex Adaptive Systems Series. Mit Press.

Nowostawski, M. and Poli, R. (1999). Parallel genetic algorithm taxonomy. In *Proceedings of the Third International*, pages 88–92. IEEE.

Srinivas, M. and Patnaik, L. (1994). Genetic algorithms: a survey. *Computer*, 27(6):17–26.

Tanese, R. (1989). Distributed genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 434–439, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Temby, L., Vamplew, P., and Berry, A. (2005). Accelerating real-valued genetic algorithms using mutation-with-momentum. In *in: The 18th Australian Joint Conference on Artificial Intelligence*, pages 1108–1111.

Wen-hua, Z., Papadopoulos, Y., and Parker, D. (2007). Reliability optimization of series-parallel systems using asynchronous heterogeneous hierarchical parallel genetic algorithm. *China Academic Journal Eletctronic Publishing House*, 1(4):403–412.

Whitley, D., Kauth, J., and of Computer Science, C. S. U. D. (1988). *GENITOR: A Different Genetic Algorithm*. Technical report (Colorado State University. Dept. of Computer Science). Colorado State University, Department of Computer Science.

Wilson, G. and Banzhaf, W. (2010). Deployment of parallel linear genetic programming using gpus on pc and video game console platforms. *Genetic Programming and Evolvable Machines*, 11(2):147–184.

Yang, S. (2002). Genetic algorithms based on primal-dual chromosomes for royal road functions.