Técnicas de otimização em memória para algoritmos genéticos usando computadores *multicore*

Harison H. Silva[1], **Carlos A.P.S. Martins**[1], **Gustavo L. Soares**[1]

[1]Programa de Pós-Graduação em Engenharia Elétrica – PUC MG
Caixa Postal 1.686. CEP 30535-901 – Belo Horizonte – Minas Gerais – Brazil

harison.silva@sga.pucminas.br, {capsm, gsoares}@pucminas.br

Abstract. *Genetic algorithms are heuristic evolutionary computing with features that allow memory optimizations in parallel computer architectures. In this paper we show techniques that take advantage of these features in relation to a specific architecture, in this case, multicore computers with shared memory. The experiments show that the time speedup motivates perform such procedures on algorithms with similar cases.*

Resumo. *Os algoritmos genéticos são heurísticas da computação evolucionária com características que permitem otimizações de memória em arquiteturas de computadores paralelos. Neste trabalho são apresentadas técnicas que tiram proveito destas características em relação a uma arquitetura específica, neste caso, de computadores com múltiplos núcleos e memória compartilhada. Os experimentos mostram que o ganho de desempenho em relação ao tempo de implementação motiva realizar tais procedimentos em algoritmos com casos similares.*

1. Introdução

O uso de computadores com processadores de múltiplos núcleos de processamento dentro do chip (*multicore*) para computação paralela de propósito geral tem sido abordado na literatura, principalmente com o foco no *speedup* que é possível obter nestes computadores. O fato de diversas aplicações científicas serem viáveis, em tempo hábil graças a esta forma de computação [Trobec et al. 2009], também faz com que este tema seja explorado. Entretanto, a evolução da memória principal tem sido inferior à dos processadores, em média 10% e 60% ao ano, respectivamente [Patterson and Hennessy 2007]. Assim, a capacidade dos processadores constantemente é limitada por acessos à memória, que possuem frequência de operação inferior, forçando então o processador a esperar.

Os algoritmos genéticos (AGs) possuem características que podem ser exploradas para otimizar o uso da memória. Neste trabalho foram desenvolvidas e apresentadas técnicas sobre os AGs de forma a melhorar o desempenho na arquitetura de computadores (AC) paralelos de memória compartilhada. Estas podem reduzir a percepção da latência da memória e utilizar os demais recursos disponíveis de forma otimizada.

Nos trabalhos encontrados relacionados a otimizações em memória nos AGs, as otimizações são propostas isoladamente. Como em [Chang and Huang 2009] que há a proposta de otimizar o uso da memória *cache*, entretanto não foi abordada a

programação paralela, especialmente com escalonamento do fluxo de execução (*threads*) [Cruz et al. 2010], para prevenir perda de desempenho devido a memória compartilhada. Também não foi proposta uma técnica para redução do uso de memória simultaneamente.

O objetivo deste trabalho é implementar diferentes técnicas para otimizar o uso da memória em AGs, com alterações do algoritmo e uso de operações a nível binário, considerando os perfis das aplicações e da AC *multicore*. São apresentadas análises comparativas destas isoladas e em suas implementações híbridas.

A Seção 2 apresenta a revisão bibliográfica, seguida da Seção 3 que apresenta as implementações e da Seção 4 que apresenta os resultados. As conclusões do trabalho são apresentadas na Seção 5.

2. Revisão bibliográfica

2.1. Algoritmos Genéticos

Os AGs são heurísticas metaforicamente baseadas em princípios da seleção natural e genética, onde indivíduos mais bem adaptados tendem a permanecer para próximas gerações [Goldberg 1989, De Jong 2006]. Os primeiros trabalhos em AGs são atribuídos a Holland em 1975 [Holland 1975] e desde então os AGs têm sido amplamente difundidos. Diversas aplicações podem utilizar o AG, como aplicações *Non-deterministic Polynomial-time hard* (*NP-hard*) [Garey and Johnson 1990] ou com formulação da função-objetivo não-trivial.

Os AGs possuem *n* indivíduos em uma população, e cada um representa uma possível solução. Desta forma, havendo *m* variáveis em cada instância do problema, o indivíduo terá *m* cromossomos, geralmente representados por códigos binários. Cada *bit* de um cromossomo é chamado de alelo.

Como na natureza, a evolução dos indivíduos depende do cruzamento (*crossover*) e da mutação que podem torná-los mais aptos à seleção para as próximas gerações. O cruzamento é o processo no qual algumas características de dois indivíduos são combinadas para formar novos. A mutação ocorre em seguida e tem o propósito de evitar a convergência para um ponto ótimo local. A execução do AG simples segue o Algoritmo 1, baseado em [Soares 1997], onde avaliação, seleção, cruzamento e mutação ocorrem até que o ponto de convergência seja alcançado.

Algoritmo 1 - Algoritmo genético simples - adaptado de [Soares 1997]

```
Inicialize as probabilidades de cruzamento e mutação, e tamanho da população.
Gere população inicial
enquanto critério convergência não alcançado faça
    Avalie os indivíduos da população
    Execute a seleção
    Execute cruzamento
    Execute mutação
fim enquanto
```

2.2. Arquitetura de computadores

Um arquiteto de computadores deve adaptar a tecnologia existente para solucionar problemas de computação, e deve planejar o futuro deste campo de tal forma a influenciá-

lo para melhor [Foster 1972]. A performance dos sistemas de *software* é fortemente afetada pelo quão bem os seus desenvolvedores entendem a AC ao trabalhar em um sistema [Patterson and Hennessy 2007] pois todos níveis do planejamento e desenvolvimento podem ser trabalhados simultaneamente.

A otimização em memória de algoritmos baseados em população pode ser realizada pela visão da AC, com amplas possibilidades de proporcionar melhor desempenho se comparado a otimizações apenas em alto-nível, por exemplo com a inserção do paralelismo por OpenMP. Uma possível otimização é baseada na observação dos acessos à hierarquia de memórias, de forma que haja redução desse gargalo do processo que é o acesso à memória. Otimizações combinadas, principalmente com a programação paralela, podem ter um comportamento e desempenho diferentes do esperado das técnicas isoladamente sendo possível que haja *speedup*.

Como tendência apontada por [Trobec et al. 2009], as futuras otimizações poderão ser realizadas em todos os níveis da computação: algoritmos, desenvolvimento, sistema operacional, compilador e *hardware*. Ratificando assim a importância do conhecimento da AC, que deve ser enfatizada desde os cursos introdutórios de algoritmos e desenvolvimento de programas [Viana et al. 2009].

2.2.1. Melhorias em Algoritmos Genéticos

Apesar de Holland [Holland 1959] em 1959 propor um computador capaz de executar um número arbitrário de tarefas simultaneamente, alguns dos primeiros trabalhos que efetivamente utilizaram paralelismo nos AGs são do fim da década de 1980, como relatado em [Paz 1998]. O uso deste recurso foi mais difundido após a programação paralela alcançar maiores níveis de abstração, facilitando o melhor uso de processadores de múltiplos núcleos através de ferramentas como o OpenMP [Dagum and Menon 1998], que podem exigir pouco conhecimento arquitetural do desenvolvedor que deseja um paralelismo mais simples [Silva and Martins 2012].

Entretanto, alguns trabalhos encontrados com o objetivo de otimizar um AG trabalham apenas no nível do desenvolvimento, usando linguagens de programação que permitem paralelizar a execução do código, deixando de otimizar os demais níveis. Dentre as exceções estão [Chang and Huang 2009] e [Pedemonte et al. 2011].

Em [Chang and Huang 2009] foram propostas alterações no AG de forma que a memória *cache* seja melhor aproveitada no processo de evolução e avaliação dos indivíduos. Já em [Pedemonte et al. 2011] foi proposta a implementação de indivíduos com codificação binária real ao invés de usar um vetor booleano com representação da codificação binária, o que reduz em 8 vezes o tamanho de cada indivíduo na memória, em comparação ao vetor booleano[1]. Nestes trabalhos o AG obteve *speedup* com a otimização da memória.

[1]Caso fosse utilizado um vetor *int* a redução no uso da memória com o uso da técnica proposta em [Pedemonte et al. 2011] seria de 32 vezes.

3. Técnicas propostas

Neste trabalho foram utilizadas técnicas para otimização da memória nos AGs. A avaliação das propostas foi realizada medindo o tempo de execução do AG otimizado para as aplicações apresentadas a seguir.

A aplicação *Expressões* consiste em encontrar uma expressão numérica $(((e_1 \odot_1 e_2) \odot_2 \cdots) \odot_{n-1} e_n) = va$, $e_i \in \mathbb{R} \mid 0 \leq e_i < 10$, $\odot_j \in \{+, -, \times, \div\}$, $1 \leq i \leq n$, $1 \leq j < n$ e *va* um número real aleatório. Tomando um indivíduo *k*, cujo valor da expressão é exp_k, seu erro absoluto é dado por $Erro_k = \mid exp_k - va \mid$, e sua função F de aptidão a maximizar é $F(exp_k) = \frac{1}{Erro_k + \varepsilon}$, com a constante ε, $0 < \varepsilon < 1$, inserida para evitar divisão por zero.

A Aplicação *One-max* [Pedemonte et al. 2011], consiste em encontrar uma *string* $\overrightarrow{x} = \{x_1, x_2, \cdots, x_n\}$, com $x_i \in \{0, 1\}$, $1 \leq i \leq n$, que maximize $F(\overrightarrow{x}) = \sum_{i=1}^{n} x_i$.

Por possuírem diferentes características, estas aplicações podem demonstrar a eficácia dos métodos evolutivos em diferentes características de indivíduos. Além disso, estes problemas são facilmente implementados, sendo a aplicação Expressões de difícil resolução por algoritmo força bruta para grandes instâncias, enquanto a aplicação Onemax possui solução trivialmente encontrada.

As técnicas propostas foram idealizadas através da análise de trabalhos relacionados, código *assembly* e tempo de execução dos métodos. As aplicações foram inicialmente implementadas em código sequencial sem otimizações, o que é referido neste trabalho como *implementação sequencial*. Com a inserção do paralelismo esta é referida apenas como *paralela*. Sobre a implementação paralela foram desenvolvidas as técnicas propostas.

Estas técnicas podem ser aplicadas a qualquer algoritmo com características semelhantes ao AG, entretanto é possível que elas tenham melhor desempenho no AG devido às características do algoritmo, como frequente uso de operações de exponenciação na base 2 e possuir demanda por grandes cadeias binárias.

3.1. Exponenciação binária

Uma característica do sistema binário é que a multiplicação por 2 é realizada com a adição de um *bit* 0 após o *bit* menos significativo do número, que neste trabalho será assumido como o bit mais a direita. Isto ocorre devido a forma como a multiplicação é realizada em binário [Widmer and Tocci 2011], o que é representado na Figura 1. Logo, uma exponenciação de base 2 no sistema binário é uma sequência de "saltos" para a esquerda no binário, ou *left shifts*.

A primeira técnica proposta neste trabalho (*Exponenciação*) consiste em substituir operações de exponenciação na base 2 por operações de *left shifts*. Isto pode ser aplicado, essencialmente, em três funções utilizadas por AGs: a conversão de binário para decimal, a conversão de decimal para binário e a normalização das variáveis. Estes métodos, executados com muita frequência nos AGs, geralmente possuem a necessidade de operações de exponenciação na base 2 que realizada pela função *pow* da linguagem C++ [Stroustrup 2013] é mais custosa computacionalmente do que *left shifts*.

O Algoritmo 2 apresenta uma conversão de binário para decimal através de exponenciação na operação 5 e sua alternativa através de *left shift* na operação 6, assumindo 1_2

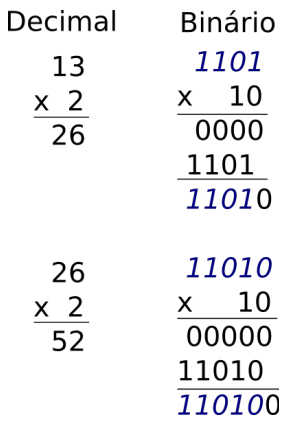

Figura 1. Multiplicação no sistema binário

o número binário 1 que será deslocado e o tamanho da cadeia binária (tcb) com o maior número de dígitos binários desejados por número.

Algoritmo 2 - Conversão de binário para decimal

requisitos O vetor inteiro bin com valores binários
1: dec1 ← 0
2: dec2 ← 0
3: **para** i ← 0 → tcb-1 **faça**
4: **se** bin[i] = 1 **então**
5: dec1 ← dec1 + $2^{tcb-1-i}$
6: dec2 ← dec2 + 1_2 *left shift* (tcb-1-i)
7: **fim se**
8: **fim para**

3.2. Otimização por *cache*

Em [Chang and Huang 2009] é proposta a alteração de alguns passos do AG para que ele utilize melhor a memória *cache*. Esta técnica foi proposta pelos autores devido a memória *cache* ser mais rápida que a memória principal, fazendo com que menos ciclos sejam perdidos pelo processador esperando uma operação de *load* ou *store* [Patterson and Hennessy 2007]. O algoritmo deve avaliar cada cromossomo enquanto este está na memória *cache*, como apresentado no Algoritmo 3.

Algoritmo 3 - Algoritmo genético simples com otimização em memória *cache*, adaptado de [Chang and Huang 2009]

Gere população inicial, indivíduo a indivíduo, e avalie cada cromossomo
enquanto critério convergência não alcançado **faça**
Realize o cruzamento e mutação, avaliando cada cromossomo imediatamente após sua criação
Selecione a nova população
fim enquanto

Entretanto, em um sistema simétrico multiprocessado (SMP), cada processador possui sua memória *cache* local e é possível que haja um falso compartilhamento, que ocorre quando uma *thread* altera uma variável compartilhada na linha da *cache* [Gabb et al. 2009]. Com isso, é necessário realizar atualização dos dados na *cache*, o que pode provocar perda de desempenho.

Para evitar este problema é possível utilizar uma solução semelhante ao proposto em [Cruz et al. 2010], onde cada trecho paralelizado é mapeado para processadores específicos. No presente trabalho que utiliza ambiente *multicore* com OpenMP deve ser utilizada uma variável de ambiente GOMP_CPU_AFFINITY que mapeia cada *thread* para um núcleo específico do processador. Então, GOMP_CPU_AFFINITY="0-n" irá fazer com que a *thread* 0 seja executada no núcleo 0, a *thread* 1 no núcleo 1, e assim por diante até a *thread n* mapeada para o núcleo *n*.

Diferente do proposto originalmente por [Chang and Huang 2009], a segunda técnica proposta (*Cache*) neste trabalho pode ser parcialmente aplicada em alguns problemas onde a avaliação de um indivíduo deve ser realizada sequencialmente, ou seja, onde a avaliação de apenas um cromossomo é inviável. Isto ocorre na aplicação Expressões, por exemplo, pois o valor do *n*-ésimo cromossomo irá alterar a aptidão do indivíduo em função do cromossomo $n-1$. Neste tipo de aplicação esta técnica deve ser alterada para realizar a avaliação de um indivíduo inteiro ao invés de avaliar cada cromossomo.

Neste trabalho, também de forma diferente de [Chang and Huang 2009], esta técnica *Cache* possibilita avaliar apenas o trecho alterado no indivíduo, podendo gerar melhor *speedup*, uma vez que a avaliação é um processo custoso computacionalmente.

3.3. Cruzamento e mutação *bitwise*

O indivíduo é um conjunto de *m* variáveis e estas são representadas em codificação binária, a fim de facilitar a implementação do cruzamento e mutação. Uma forma de implementação dessa representação é aquela em que cada indivíduo possui *m* vetores de tipo *t*, ou seja, é uma matriz de tipo *t*. Cada elemento da matriz representa um *bit* da codificação binária, $[[t_{00}t_{01}\dots t_{0n}],\dots[t_{m0}t_{m1}\dots t_{mn}]]$, $t_{xy} \in \{0,1\}$, com no mínimo 8 *bits* que é o menor tamanho de uma variável numérica, apenas para representar um único *bit*.

Utilizando operadores que a linguagem C++ possui para manipular o valor binário de tipos numéricos, como short, int e long é possível realizar o cruzamento e mutação binários em números decimais. Os operadores que a linguagem disponibiliza para manipulação destes binários são: OR, AND, NOT, *left shift* e *right shift* que operam os tipos numéricos *bit* a *bit*. Portanto, é possível utilizar valores decimais e sua codificação binária real para representar os indivíduos. Esta é a terceira técnica proposta (*Bitwise*), através da qual não há a necessidade de conversões de binário para decimal e vice-versa, bem como há menor necessidade de acessos à memória, uma vez que um *byte* possui 8 alelos. A Figura 2 apresenta um indivíduo representado por matriz e sua forma equivalente por *Bitwise*, ou seja o binário 00010101_2 corresponde ao decimal 21 e o binário 01111110_2 corresponde ao decimal 126.

Assim, ao utilizar um vetor de tipo numérico (por exemplo, booleano) cada posição do vetor terá 1 *byte* (8 *bits*) para representar um único *bit* enquanto ao utilizar a técnica *Bitwise* cada decimal fornece 8 *bits* ou mais, de acordo com a precisão da variável.

A precisão de uma variável do AG nesta implementação é representada pela quantidade de *bits* do tipo numérico, que pode ser por exemplo, 16 *bits* para *short int* ou 32 *bits* para *int*. Estes valores podem variar em algumas arquiteturas e portanto deve ser utilizada a função *sizeof(T)* da linguagem C++, para que o valor correto para a arquitetura seja utilizado. Esta função retorna o número de *bytes* de uma variável de tipo T.

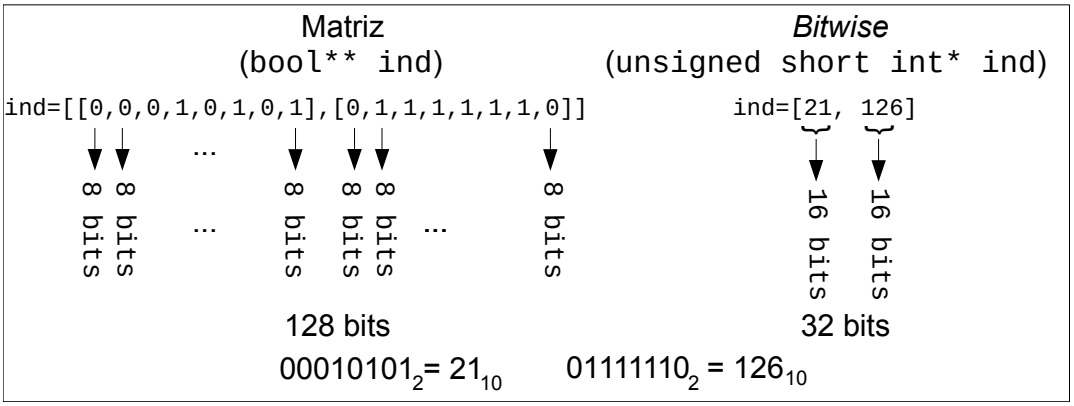

Figura 2. Representação de indivíduo por matriz e *Bitwise*

O cruzamento com esta representação decimal é detalhado no Algoritmo 4 e a mutação no Algoritmo 5. Neste cruzamento, diferente do proposto por [Pedemonte et al. 2011] que utiliza maior número de operações, o ponto de corte irá selecionar a variável a ser cruzada, em cada indivíduo, e o segundo ponto de corte será o término desta variável. Assim, o trecho binário referente a uma única variável do tipo numérico será cruzada.

Algoritmo 4 - Cruzamento com representação real

Selecione as variáveis, ind1 e ind2, em cada indivíduo
st ← tamanho da cadeia binária do tipo numérico
pc ← ponto de corte (pc) aleatório no intervalo [0..st[
mot ← maior valor possível com st *bits*
novaVar1 ← ((ind1 *right shift* pc) *left shift* pc) OR (((ind2 *left shift* (st-pc)) AND mot) *right shift* (st-pc))
novaVar2 ← ((ind2 *right shift* pc) *left shift* pc) OR (((ind1 *left shift* (st-pc)) AND mot) *right shift* (st-pc))
Substitua as variáveis selecionadas em cada indivíduo por novaVar1 e novaVar2

Algoritmo 5 - Mutação com representação real

```
ind ← indivíduo selecionado
st ← tamanho da cadeia binária do tipo numérico
varMut ← a variável a ser alterada em ind, no intervalo [0..m[
bitMut ← valor aleatório em potência de 2 no intervalo [1..st[
auxVar ← ind[varMut] OR bitMut
se auxVar = ind[varMut] então
    auxVar ← auxVar AND (NOT bitMut)
fim se
novaVar ← auxVar
Substitua varMut por auxVar no indivíduo selecionado
```

3.4. Técnicas híbridas

As técnicas híbridas são as combinações possíveis entre as técnicas propostas:

- Exponenciação + *Cache*
- Exponenciação + *Bitwise*
- *Cache* + *Bitwise*

- Exponenciação + *Cache* + *Bitwise*

A técnica de Exponenciação não é implementável em todos casos, devido a inexistência de operações de exponenciação na base 2. Nestes casos, como a aplicação One-max, a única combinação possível é de *Cache* e *Bitwise*. E em alguns problemas onde não é possível avaliar os cromossomos de cada indivíduo independentemente, deve-se utilizar a implementação original da otimização em cache, apresentada em [Chang and Huang 2009].

4. Resultados

A plataforma utilizada foi o sistema operacional Linux, a linguagem de programação C++ [Stroustrup 2013] com OpenMP [Dagum and Menon 1998], o compilador GCC [Stallman and Community 2009], em um computador com processador Intel Core i5-2400 [INTEL CORPORATION 2011], de 4 núcleos, 4 *threads* e 4 GB de memória primária DDR3 de 1600 MHz. As configurações utilizadas no AG para cada aplicação são apresentadas na Tabela 1, as quais foram executadas 10 vezes cada.

Tabela 1. Configurações

Aplicação	Expressões	One-max
Critério de parada	máximo erro absoluto = 0,0001	encontrar valor ótimo
Tamanho da instância	500	10.000, 20.000, 30.000 e 40.000
Probabilidade de cruzamento	0,6	0,9
Probabilidade de mutação	0,01	0,1
Tamanho da população	variando entre 70, 100 e 130	

Na Figura 3 são apresentados os tempos de execução, e os respectivos valores de *speedup*, para cada técnica na aplicação Expressões. Cada execução tem seu tempo iniciado juntamente com o programa executável e encerra com este. Cada execução é encerrada ao atingir o critério de parada, independente do número de gerações. O valor apresentado para cada técnica é a média geométrica de 30 execuções, 10 execuções para cada configuração apresentada na Tabela 1. A técnica *sequencial* representa a primeira implementação sequencial realizada sem otimizações e *paralela* representa a implementação inicial com paralelismo sem outras otimizações. As demais técnicas foram implementadas sobre a versão paralela.

Para estas execuções o erro absoluto, ou seja, a diferença absoluta entre a expressão do indivíduo e o valor objetivo, é apresentado na Figura 4. Como apresentado na Tabela 1, o critério de parada nesta aplicação é o máximo erro absoluto de 0,0001. Ou seja, a execução é encerrada quando houver um indivíduo cujo a expressão tem erro absoluto de no máximo 0,0001 em relação ao valor ótimo.

A Figura 5 apresenta os resultados da aplicação One-max, cujo as implementações com a técnica Exponenciação não são aplicadas ao problema, pois não há necessidade de conversão entre binário e decimal, e não há necessidade de normalização das variáveis. Portanto, apenas são utilizadas as otimizações em *Cache*, *Bitwise* e a combinação destas.

4.1. Análises dos resultados

Na aplicação Expressões as otimizações com *Bitwise* obtiveram maior *speedup*, o que pode ser devido ao menor uso de memória em comparação com as demais. Nesta aplicação as otimizações com a técnica de otimização de *Cache* não obtiveram tanto *speedup*

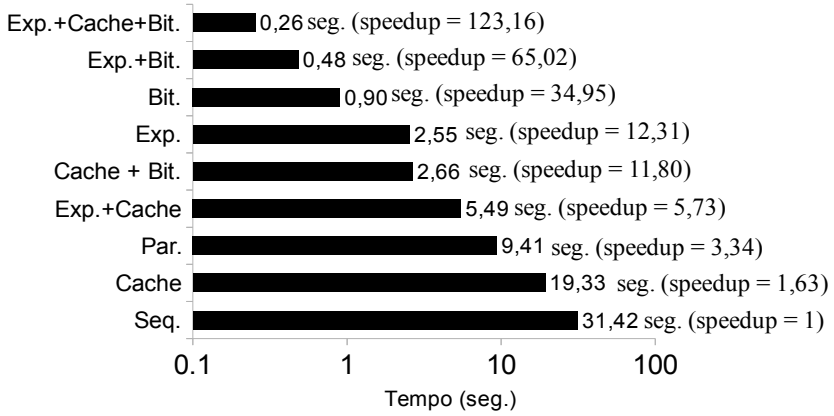

Figura 3. Resultados aplicação Expressões

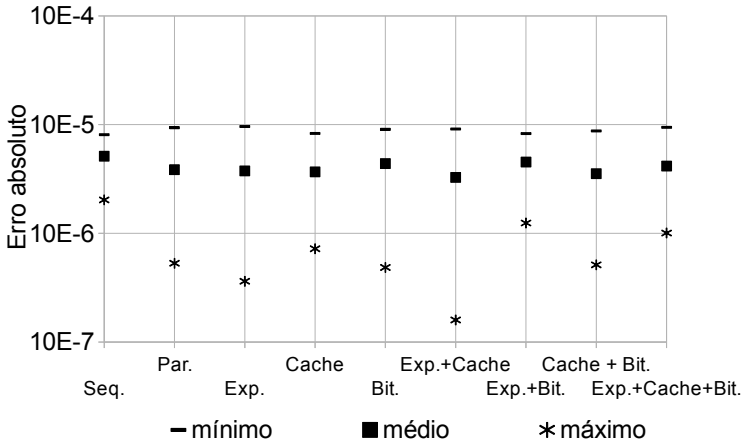

Figura 4. Erro absoluto na aplicação Expressões

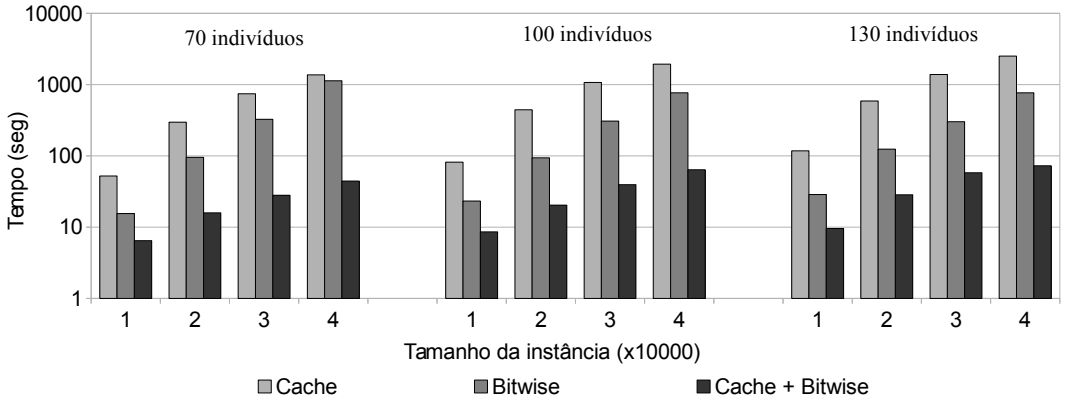

Figura 5. Média geométrica do tempo na aplicação One-max

quanto as otimizações com *Bitwise* pois não há possibilidade de avaliar apenas o trecho alterado de um indivíduo. Em ambas aplicações é possível observar que a combinação de técnicas pode levar a diferentes ganhos em relação à simples adição de ganho individual

destas. Por exemplo, na aplicação Expressões a técnica Exponenciação obteve *speedup* igual a 12,31 e a técnica *Bitwise* obteve *speedup* igual a 34,95, mas o *speedup* da implementação híbrida destas duas técnicas é igual a 65,02.

A Figura 4 apresenta os valores de erro absoluto dos melhores e piores indivíduos, bem como a média, de 30 execuções na aplicação Expressões, com 10 execuções para cada configuração. É possível observar que todas as otimizações propostas obtiveram melhor média e máximo em relação à implementação sequencial, o que garante que a semântica do algoritmo está sendo mantida. Isto também mostra que estas otimizações produzem resultados em menor tempo e com maior qualidade em relação ao sequencial.

A implementação *Cache* não se mostrou eficaz como as demais analisadas. Este pior desempenho está relacionado à grande quantidade de acessos à memória, pois cada indivíduo nesta técnica possui *n* variáveis numéricas, enquanto em *Bitwise* os indivíduos possuem $\frac{n}{m}$, onde *n* é o tamanho da instância e *m* é o número de *bits* que a variável numérica possui. Entretanto, *Cache* implementada juntamente com *Exponenciação* e *Bitwise* possui o melhor desempenho, pois devido a *Bitwise* há redução do tamanho dos indivíduos.

Para a segunda aplicação utilizada, One-max, pode-se observar através da Figura 5 que quanto menor o número de indivíduos e maior o tamanho da instancia melhor é o desempenho de *Cache* em relação a *Bitwise*. E quanto maior o número de indivíduos e maior o tamanho da instancia melhor o desempenho da implementação híbrida em relação a *Bitwise*. Estes fatos estão relacionados com o fato de que nesta aplicação a técnica *Cache* pode tirar benefício da avaliação parcial de um indivíduo, e quanto maior o tamanho da instância e da população maior será a demanda por avaliações.

5. Conclusões

Conclui-se que é importante analisar as características da aplicação para determinar qual técnica melhor se aplica. Nos casos apresentados, a implementação híbrida das 3 técnicas propostas obteve melhor desempenho tanto na aplicação Expressões quanto na aplicação One-max, mas em aplicações com instâncias pequenas e com a impossibilidade de avaliação parcial de um indivíduo a implementação híbrida da técnica Exponenciação e *Bitwise* pode obter melhor *speedup*.

Nas aplicações utilizadas, as implementações híbridas levaram a desempenho superior a soma de desempenhos individuais das técnicas, pois os recursos computacionais são melhor utilizados, como em um efeito cascata. Também é importante analisar as reais necessidades de uma aplicação. Utilizar um determinado conjunto de parâmetros pode influenciar na escolha da plataforma ideal, bem como da técnica que deve ser utilizada.

Nos resultados apresentados neste trabalho pode ser observado que otimizações em memória de uma aplicação, especialmente em AGs, podem levar a *speedup* superior a 12 com simples modificações, e é possível obter *speedup* superior a 120 com a combinação de outras técnicas. Os algoritmos que possuem grande demanda por operações de exponenciação na base 2 podem obter *speedup* com a simples substituição de métodos em alto-nível de exponenciação por operações de *left shifts*. Algoritmos com demanda de representações binárias podem utilizar a codificação binária real e reduzir em 8 vezes, ou mais de acordo com o tipo numérico, o total de memória utilizada, reduzindo tanto a

requisição de armazenamento quanto o número de acessos, o que produz *speedup* devido à redução do gargalo de memória na aplicação.

A implementação de otimizações no algoritmo de forma híbrida com otimizações no desenvolvimento podem levar a um uso mais adequado da memoria e gerar *speedup* superior a simples paralelização do algoritmo. Como apresentado, alguns trabalhos utilizam técnicas semelhantes porém em nenhum dos trabalhos encontrados é explorada a implementação híbrida destas otimizações em AGs. Como trabalho futuro propõe-se a implementação destas técnicas em GPUs que também possuem demanda por redução do uso de memória devido à latência desta. Também pode-se propor a aplicação destas técnicas com implementação em *hardware* com sistema operacional para computação de alto-desempenho, como o BareMetal [Infinity 2008].

Referências

Chang, F.-C. and Huang, H.-C. (2009). A study on the cache miss rate in a genetic algorithm implementation. In *Intelligent Information Hiding and Multimedia Signal Processing, 2009. IIH-MSP '09. Fifth International Conference on*, pages 795–798.

Cruz, E., Alves, M., and Navaux, P. (2010). Process mapping based on memory access traces. In *Computing Systems (WSCAD-SCC), 2010 11th Symposium on*, pages 72–79.

Dagum, L. and Menon, R. (1998). Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55.

De Jong, K. (2006). *Evolutionary Computation:A Unified Approach*. MIT Press.

Foster, C. (1972). Computer architecture. *Computer*, 5(2):18–19.

Gabb, H., Corden, M., Rosenquist, T., Fischer, P., Fedorova, J., Breshears, C., Zipplies, T., Tsymbal, V., Akyil, L., Pegushin, A., Kukanov, A., Petersen, P., Voss, M., Tersteeg, A., and Hoeflinger, J. (2009). *Intel Guide for Developing Multithreaded Applications*. Intel Corporation.

Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

Holland, J. (1959). A universal computer capable of executing an arbitrary number of subprograms simultaneously. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 108–113, New York, NY, USA. ACM.

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

Infinity, R. (2008). BareMetal Operating System.

INTEL CORPORATION (2011). Especificações Intel core i5-2400. online.

Patterson, D. A. and Hennessy, J. L. (2007). *Computer Organization and Design: The Hardware/Software Interface. Third Edition, Revised*. Morgan Kaufmann, 3 edition.

Paz, C. E. (1998). A Survey of Parallel Genetic Algorithms.

Pedemonte, M., Alba, E., and Luna, F. (2011). Bitwise operations for GPU implementation of genetic algorithms. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '11, pages 439–446, New York, NY, USA. ACM.

Silva, H. and Martins, C. A. (2012). Avaliação de implementações do algoritmo genético paralelo para solução do problema do caixeiro viajante usando openmp e pthreads. In *WSCAD-WIC 2012*.

Soares, G. L. (1997). Algoritmos genético: Estudo, novas técnicas e aplicações. Master's thesis, Universidade Federal de Minas Gerais, Conselho Nacional de Desenvolvimento Científico e Tecnológico.

Stallman, R. M. and Community, G. D. (2009). *Using The GNU Compiler Collection: A GNU Manual For GCC Version 4.3.3*. CreateSpace, Paramount, CA.

Stroustrup, B. (2013). *The C++ Programming Language, 4th Edition*. Addison-Wesley Professional, 4 edition.

Trobec, R., Vajtersic, M., and Zinterhof, P., editors (2009). *Parallel Computing: Numerics, Applications, and Trends*. Springer.

Viana, I. E. M., Machado-Coelho, T. M., Silva, H. H., Drumond, L., and Martins, C. A. P. S. (2009). Valorização acadêmica no processo de formação de um verdadeiro cientista da computação. In *Anais X Simpósio em Sistemas Computacionais*, volume X, pages 103–106. Workshop sobre Educação em Arquitetura de Computadores.

Widmer, N. S. and Tocci, R. J. (2011). *Sistemas digitais*. Pearson.