OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS EM ARQUITETURAS DE ALTO DESEMPENHO

Particle Swarm Optimization in High-Performance Architectures

Rogério de Moraes Calazan[1], Nadia Nedjah[2], Luiza de Macedo Mourelle[3]

Resumo: O algoritmo de Otimização por Enxame de Partículas (PSO, *Particle Swarm Otimization*) é uma técnica de otimização que vem sendo utilizada na solução de diversos problemas, em diferentes áreas do conhecimento. Porém, a maioria das implementações é realizada de modo sequencial. O processo de otimização necessita de um grande número de avaliações da função objetivo, principalmente em problemas complexos que envolvam uma grande quantidade de partículas e dimensões. Consequentemente, o algoritmo pode se tornar ineficiente em termos do desempenho, correspondente ao ganho obtido com a implementação da melhoria, tempo de resposta e até na qualidade do resultado esperado. Para superar tais dificuldades, pode-se utilizar a computação de alto desempenho e paralelizar o algoritmo, de acordo com as características da arquitetura, visando ao aumento de desempenho, a minimização do tempo de resposta e a melhoria da qualidade do resultado final. Este artigo apresenta uma versão do PSO paralelo implementado nas arquiteturas de multiprocessadores, multicomputadores e *Graphics Processing Unit* (GPU). Os diferentes testes realizados mostram que, nos casos com maiores partículas e dimensões, a arquitetura de multicomputadores obteve os melhores resultados. Todas as implementações paralelas obtiveram eficiência, em termos de otimização, igual ou superior à da implementação sequencial.

Palavras-chave: Otimização por Enxame de Partículas. Algoritmos Paralelos. Arquiteturas de Alto Desempenho.

Abstract: Particle Swarm Optimization (PSO) is an optimization technique that is used to solve many problems in different applications. However, most implementations are sequential. The optimization process requires a large number of evaluations of the objective function, especially in complex problems, involving a large amount of particles and dimensions. As a result, the algorithm may become inefficient in terms of performance, corresponding to the gain obtained with the implementation of improvement, execution time and even the quality of the expected result. To overcome these difficulties, high performance computing and parallel algorithms can be used, taking into account the characteristics of the architecture. This should increase performance, minimize response time and may even improve the quality of the final result. This paper presents a parallel version of PSO implemented in a multiprocessor, multicomputer and Graphics Processing Unit (GPU) based parallel architectures. The different performed assessments show that the multicomputer achieved the best results for problems with high number of particles and dimensions. In terms of optimization, all parallel implementations achieved equal or higher efficiency in comparison to sequential implementation.

Keywords: Particle Swarm Optimization. Parallel Algorithms. High Performance Architecture.

1. Capitão-Tenente (T); Mestre em Engenharia Eletrônica pela Universidade do Estado do Rio de Janeiro – Rio de Janeiro, RJ – Brasil. Instituto de Estudos do Mar Almirante Paulo Moreira – Arraial do Cabo, RJ – Brasil. E-mail: rgc.moraes@gmail.com

2. Doutora em Computação pela University of Manchester – Manchester – Inglaterra. Professora da Universidade do Estado do Rio de Janeiro – Rio de Janeiro, RJ – Brasil. E-mail: nadia@eng.uerj.br

3. Doutora em Computação pela University of Manchester – Manchester – Inglaterra. Professora da Universidade do Estado do Rio de Janeiro – Rio de Janeiro, RJ – Brasil. E-mail: ldmm@eng.uerj.br

1. INTRODUÇÃO

O algoritmo de Otimização por Enxame de Partículas (PSO, *Particle Swarm Otimization*) foi introduzido por Kennedy e Eberhart (1995) e é baseado no comportamento coletivo e na influência e aprendizado social. No PSO, procura-se imitar o comportamento social de grupos de animais, mais especificamente de bando de pássaros. Recentemente, o PSO tem sido utilizado como uma técnica de otimização na solução de diversos problemas (SEDIGHIZADEH; MASEHIAN, 2009). O algoritmo PSO pode ser utilizado em diferentes áreas do conhecimento, como no planejamento de rota de veículo subaquático (YANG; ZHANG, 2009) e de robôs (MA et al., 2009), biologia e bioinformática (RASMUSSEN; KRINK, 2003), entre outros.

Porém, a maioria das implementações é realizada de modo sequencial e, embora os computadores estejam com processadores cada vez mais velozes, as exigências em termos de poder computacional, em geral, crescem mais rápido do que a velocidade de operação. O processo de otimização precisa realizar um grande número de avaliações da função objetivo, a qual usualmente é feita de modo sequencial na CPU, acarretando baixo desempenho. Consequentemente, o algoritmo pode se tornar ineficiente, tendo em vista o desempenho obtido, o tempo de resposta e até a qualidade do resultado esperado. Desta forma, o PSO necessita de um longo tempo de processamento para encontrar a solução em problemas complexos e/ou que envolvam um grande número de dimensões e partículas. Para superar tais dificuldades, pode-se utilizar a computação de alto desempenho e paralelizar o algoritmo, de acordo com as características da arquitetura, de forma a aumentar o desempenho, minimizar o tempo de resposta e melhorar a qualidade do resultado. Assim, o PSO será implementado visando arquiteturas paralelas baseadas em multiprocessadores, multicomputadores e *Graphics Processing Unit* (GPU).

Alguns trabalhos (KOH et al., 2006; SHUTTE et al., 2004) consideram implementações paralelas do algoritmo PSO em *cluster* de computadores utilizando-se a biblioteca *Message Passing Interface* (MPI). Em Wang et al. (2008), os autores apresentam uma proposta de algoritmo paralelo baseado em subpopulações do PSO, implementado em *Open Multi-Processing* (OpenMP). O algoritmo realiza a execução de modo assíncrono e divide o enxame em vários subenxames que atualizam a velocidade de acordo com a melhor posição encontrada no respectivo subenxame. O paralelismo foi aplicado no laço de execução das populações, dividindo, assim, a execução dos enxames entre as *threads* do OpenMP. Os resultados obtidos demonstram uma redução de 30% no tempo de execução, quando comparado com a implementação sequencial.

Em Veronese e Krohling (2009) e Zhou e Tan (2009), os autores reportam uma implementação do PSO utilizando a plataforma de computação paralela *Compute Unified Device Architecture* (CUDA). O algoritmo se beneficia do paralelismo realizando as operações das partículas de forma independente e em paralelo. Assim, as partículas são mapeadas em *threads* e organizadas em blocos. Os experimentos são obtidos comparando-se a implementação paralela em GPU com uma implementação serial em CPU e utilizando-se um número fixo de iterações. O resultados mostram que a implementação em GPU alcança um maior *speedup* quando usada com problemas de alta dimensionalidade, utilizando-se maiores populações.

Em Calazan et al. (2013), os autores implementam o algoritmo PSO diretamente em *hardware*, utilizando um *Field-Programmable Gate Array* (FPGA). Apesar dos bons resultados obtidos em termos de tempo de execução, não foi possível escalar a arquitetura para um número maior de partículas, devido à limitação de área do FPGA. Assim, não foi possível realizar uma comparação com os resultados deste trabalho.

Este trabalho propôs paralelizar o algoritmo PSO e realizar sua implementação em *software* nas arquiteturas de multiprocessador, multicomputador e GPU. As implementações foram executadas utilizando-se diferentes arranjos de partículas e dimensões, de forma a analisar o desempenho e a eficiência do algoritmo paralelo, obtidos por cada arquitetura, na solução de problemas de otimização.

2. OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS

O PSO é um algoritmo baseado em inteligência coletiva, estocástica e interativa, o qual procura a solução de problemas de otimização em um determinado espaço de busca e é capaz de emular o comportamento social de indivíduos de acordo com objetivos definidos. Foi inicialmente desenvolvido como uma ferramenta de simulação de padrões de voo dos pássaros em busca de comida e proteção (REYNOLDS, 1987), mas Kennedy e Eberhart (1995) observaram que esse comportamento dos pássaros poderia ser adaptado e utilizado em processos de otimização, criando, assim, a primeira versão simples do PSO.

O PSO é formado por um conjunto de partículas, sendo cada uma delas uma solução em potencial do problema, possuindo coordenadas de posição em um espaço de busca multidimensional. As partículas fluem através do espaço de busca, onde as suas posições são ajustadas de acordo com a sua própria experiência e a das partículas vizinhas. No PSO, a partícula possui velocidade e direção adaptativas que determinam sua movimentação (KENNEDY; EBERHART, 1995). A partícula é dotada também de uma memória que a torna capaz de lembrar sua melhor posição anterior. Deste modo, cada partícula possui um vetor de posição, um vetor de melhor posição, um campo para aptidão e outro para melhor aptidão. Para a atualização da posição de cada partícula do algoritmo PSO, é definida uma velocidade para cada dimensão dessa posição (ENGELBRECHT, 2005).

2.1. ALGORITMO MELHOR LOCAL

O pseudocódigo do algoritmo melhor local é apresentado no Algoritmo 1. Nesse algoritmo, a vizinhança de cada partícula é formada por um subconjunto de partículas sobre as quais uma melhor partícula local, denominada *Lbest*, é selecionada. Desta forma, ele reflete uma topologia em anel. Nessa topologia, o fluxo de informações flui mais lentamente através da componente social da velocidade. Apenas a vizinhança imediata, cuja seleção é baseada nos índices das partículas, compartilha a sua melhor posição, tendo como consequência uma taxa de convergência menor. Isso implica cobrir uma maior parte do espaço de busca, provendo um melhor desempenho em termos de qualidade da solução encontrada. Essa estrutura de vizinhança também provê um melhor desempenho computacional.

O primeiro passo do algoritmo PSO é inicializar a distribuição das partículas no espaço de busca, assim como seus parâmetros de controle. Após a inicialização, o algoritmo entra no processo iterativo, onde a aptidão de cada partícula é calculada. A seguir, é realizada a atualização da melhor posição da partícula, denominada *Pbest*, em relação a resultados anteriores. Depois, é realizada a atualização da melhor posição local em relação aos valores obtidos por sua vizinhança. Nesse momento, o algoritmo está pronto para realizar a atualização da velocidade de cada partícula. A velocidade é o elemento que promove a capacidade de locomoção das partículas e pode ser calculada pela Equação 1 (KENNEDY; EBERHART, 1995; ENGELBRECHT, 2005).

$$v_{ij}(t+1)=wv_{ij}(t)+c_1r_1\left(Pbest_{ij}-x_{ij}(t)\right)+c_2r_2\left(Lbest_{ij}-x_{ij}(t)\right) \quad \textbf{(1)}$$

onde w é chamado de coeficiente de inércia, r_1 e r_2 são números aleatórios entre [0,1], c_1 e c_2 são constantes positivas, $Pbest_{ij}$ é a melhor posição atingida pela partícula i, na dimensão j, no passado e $Lbest_{ij}$ é a melhor posição encontrada na vizinhança da partícula i na dimensão j.

A posição de cada partícula é alterada conforme a Equação 2:

```
Crie e inicialize um enxame com n partículas;
repita
  para i := 1 até n faça
    calcule o fitness da partícula_i ;
    se fitness_i < f(Pbest_ij) então
      Atualize Pbest_ij com a nova posição;
    fim se
    se f(Pbest_ij) < f(Lbest_ij) então
      Atualize Lbest_ij com a nova posição;
    fim se
  fim para
  para i := 1 até n faça
      Atualize a velocidade da partícula utilizando a Equação (1);
      Atualize a posição da partícula utilizando a Equação (2);
  fim para
Até condição de parada ;
retorne melhor resultado;
```

Algoritmo 1. Melhor Local (*Lbest*).

$$x_{ij}(t+1)=v_{ij}(t+1)+x_{ij}(t) \quad \textbf{(2)}$$

onde $x_{ij}(t+1)$ é a posição atual e $x_{ij}(t)$ é a posição anterior. A velocidade guia o processo de otimização refletindo tanto a experiência da partícula quanto a troca de informações entre as partículas. O conhecimento experimental de cada partícula refere-se ao comportamento cognitivo, que é proporcional à distância entre a partícula e sua melhor posição, encontrada desde a primeira iteração.

Após atualizar a velocidade e a posição de cada partícula, é verificada a condição de parada e, então, é exibido o resultado final ou realizada mais uma iteração. A condição de parada é utilizada pelo algoritmo para terminar o processo de busca. É importante que a condição de parada não implique parar o algoritmo prematuramente, antes de se obter um resultado satisfatório, ou levar a um processamento desnecessário quando o resultado já tiver sido alcançado.

- Número máximo de iterações: utilizando-se este critério, é obvio que, para um número pequeno de iterações, o término da otimização pode acontecer antes de encontrar uma boa solução. Um número muito grande pode levar a um custo computacional desnecessário quando somente este critério é verificado na condição de parada;
- Solução aceitável: este critério irá interromper o processo de otimização quando for encontrado um erro aceitável entre o valor da função objetivo e o melhor valor encontrado pelo enxame. Porém, este critério de parada impõe o conhecimento prévio do ótimo global.

O valor de cada parâmetro do algoritmo PSO é fundamental no processo de busca e, por isso, a importância de se definirem valores adequados. O coeficiente de inércia w controla a relação entre explorar grandes espaços e uma determinada localidade. Em geral, são utilizados valores próximos mas não maiores de 1 e nem muito próximos de 0. Valores maiores que 1 tendem a deixar as partículas com uma velocidade muito alta, tendendo a haver grande divergência, enquanto valores muito menores, próximos de 0, podem desacelerar demais a busca. O coeficiente cognitivo c_1 e o coeficiente social c_2, juntamente com os fatores aleatórios r_1 e r_2, controlam a influência estocástica no cálculo da velocidade da partícula. Trabalhos recentes (ENGELBRECHT, 2005) indicam que é melhor que o coeficiente cognitivo seja maior que o social e que $c_1+c_2=4$. Essa referência indica também que bons resultados foram alcançados utilizando-se $c_1 \approx c_2 \approx 1{,}49$.

Para limitar a velocidade da partícula, de modo que a ela não saia do espaço de busca, são impostos limites, denominados velocidade máxima ou v_{max}. A velocidade máxima é definida para cada dimensão do espaço de busca e pode ser formulada como um percentual do domínio, conforme a Equação 3, onde x_{max} e x_{min} são, respectivamente, o valor máximo e o mínimo do domínio, e δ é um valor no intervalo [0,1].

$$v_{max}=\delta(x_{max}-x_{min}) \quad \textbf{(3)}$$

A eficiência do enxame é uma medida de desempenho representada pelo número de iterações realizadas para encontrar uma solução com uma determinada acurácia. A eficiência do enxame expressa o tempo relativo para alcançar uma determinada solução.

2.2. ALGORITMO DE OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS PARALELO

O algoritmo paralelo proposto decorre da ideia de que o trabalho realizado por uma partícula é independente daquele realizado pelas outras partículas do enxame. Desta forma, após a inicialização, cada partícula calcula o valor da função objetiva, atualiza a velocidade e a posição e elege *Pbest* e *Lbest* de forma independente e em paralelo com as outras partículas. Assim, cada partícula pode ser alocada em uma unidade de processamento para realizar sua execução. Os vizinhos da partícula i são as partículas $(i+1) mod\ n$ e $(i-1) mod\ n$. Assim, a melhor posição local é obtida baseada nessa definição de vizinhança. A Figura 1 exibe o fluxograma do algoritmo PSO paralelo na topologia em anel, onde $p_1,...,p_n$ denotam as n partículas do enxame e $v^{(p1)},...,v^{(pn)}$ e $x^{(p1)},...,x^{(pn)}$, as respectivas velocidades e posições.

2.3. FUNÇÃO DE APTIDÃO

Quatro funções, encontradas em Engelbrecht (2005) e Molga e Smutnicki (2005), foram implementadas com o objetivo de avaliar o desempenho das arquiteturas. A função *Esfera* é definida em 4, *Rosenbrock* em 5, *Rastrigin* em 6 e *Schwefel* em 7:

$$f_1(x_i)=\sum_{i=1}^{d} x^2_i \quad \textbf{(4)}$$

$$f_2(x_i)=\sum_{i=1}^{d} 100(x_i-x^2_{i-1})^2+(x_{i-1}-1)^2 \quad \textbf{(5)}$$

$$f_3(x_i)=\sum_{i=1}^{d} \left(x^2_i-10\cos(2\pi x_i)+10\right) \quad \textbf{(6)}$$

$$f_4(x_i)= 418{,}9829-\sum_{i=1}^{d} (x_i \sin\sqrt{|x_i|}) \quad \textbf{(7)}$$

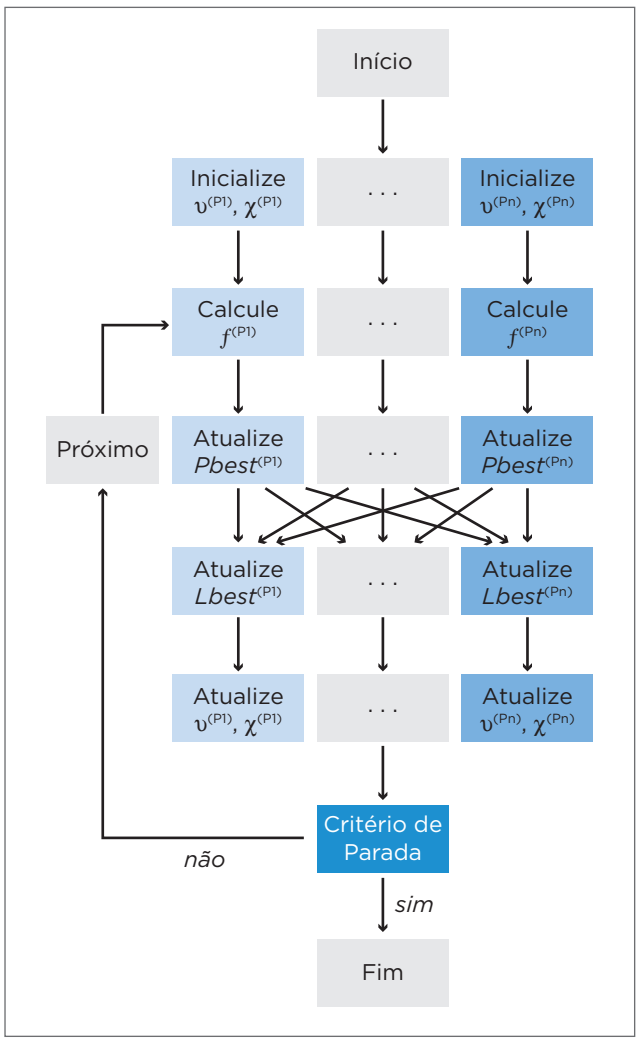


Figura 1. Computação paralela realizada por um enxame com *n* partículas na topologia em anel.

3. METODOLOGIA

Inicialmente, o PSO foi implementado na arquitetura de multicomputador utilizando-se a interface de programação de aplicativos (API – *Application Programming Interface*) OpenMP. O OpenMP foi desenvolvido para permitir e facilitar a programação no ambiente paralelo, utilizando-se multiprocessadores com uma memória compartilhada. O ambiente de multiprocessadores é um computador paralelo, no qual todos os processadores compartilham um único espaço físico de endereçamento (PATTERSON; HENNESSY, 2011). Os processadores se comunicam através de variáveis compartilhadas na memória, sendo que todos os processadores são habilitados a acessar qualquer área da memória via instruções de *load* e *store*. O OpenMP utiliza o modelo de programação *Fork-Join* (DENNIS; HORN, 1966). Nessa abordagem, o programa inicia de maneira semelhante a um processo simples, como um programa serial. Em um determinado ponto, o processo é dividido em várias subtarefas que executam em uma região paralela até o termino do trabalho. No final, os processos são reunidos e o sistema retorna ao processo simples. Na região paralela, as *threads* recebem um número de identificação denominado *tid*.

Em seguida, o PSO será implementado na arquitetura de multicomputadores utilizando-se o *Message Passing Interface Chameleon* (MPICH). O MPICH (GROPP; SMITH, 1993) é uma implementação completa do padrão MPI, projetada para ser portável e eficiente em arquiteturas de alto desempenho. O MPI é uma especificação para um padrão de biblioteca de passagem de mensagens, que foi definido no Forum MPI (MESSAGE PASSING INTERFACE FORUM, 2012). O MPICH é empregado em ambiente de multicomputadores com troca de mensagens. Os processadores são interligados por uma rede de alta velocidade e colaboram entre si na computação paralela por meio de troca de mensagens, utilizando operações de *envio* e *recebimento*. Não é possível acessar a memória do outro processador utilizando as instruções de *load* e *store*. As operações de envio e recebimento de mensagens podem ser *bloqueantes* e *não bloqueantes*. Em uma rotina *bloqueante*, os processos envolvidos na operação param a execução do programa até que a operação de envio ou recebimento da mensagem seja concluída. Em uma rotina *não bloqueante*, a execução do programa continua imediatamente após ter requisitado a comunicação.

Por fim, o PSO será implementado na arquitetura de GPU utilizando-se a plataforma CUDA. A plataforma de computação paralela CUDA (NVIDIA, 2010a) é um modelo de programação paralela escalável e uma plataforma de *software* para GPU que permite ao programador utilizar uma extensão da linguagem C como interface de programação. O modelo de programação CUDA possui um estilo de uma única instrução sobre várias *threads* (SIMT – *Single Instruction, Multiple Threads*), na qual o programador escreve um programa para uma única *thread* que será instanciada e executada por várias *threads* em paralelo nos múltiplos processadores da GPU (KIRK; HWU, 2010).

A CUDA provê três abstrações-chave: hierarquia em grupos de *threads*, memória compartilhada e barreiras de sincronização, que fornecem uma estrutura paralela para a linguagem C. Na hierarquia em grupos de *threads*, o programador organiza as *threads* em *blocos* e *grades*. Uma *grade* é um conjunto de *blocos*

que podem ser executados de forma independente e em paralelo. Um *bloco* é formado por um conjunto de *threads* concorrentes que podem cooperar via barreiras de sincronização e através de uma memória compartilhada. A memória compartilhada é uma área de memória privada do bloco, visível a todas as *threads* desse *bloco*, e a barreira de sincronização é um ponto onde as *threads* de um mesmo *bloco* esperam até que todas as *threads* desse *bloco* alcancem a barreira (KIRK; HWU, 2010).

3.1. IMPLEMENTAÇÃO EM *OPEN MULTI-PROCESSING*

O algoritmo proposto paraleliza o código dividindo a computação em partículas independentes, ou seja, cada partícula pode ser implementada como uma *thread*. Desta forma, o construtor *for* foi implementado no laço referente às partículas para distribuir a computação entre as *threads* que irão executar a região paralela, conforme exibido no Algoritmo 2. Assim, para o processamento de um enxame com 64 partículas em uma região paralela com 2 *threads*, cada *thread* executará 32 partículas. A *thread* 0 executa as partículas 0 a 31 e a *thread* 1 executará as partículas 32 a 63. O algoritmo é executado dentro de uma mesma região paralela até que a condição de parada seja atingida, evitando, assim, o *overhead* de criação de regiões paralelas a cada iteração.

Com o objetivo de paralelizar a busca pelo menor resultado do enxame, foi definido o vetor *Best*, de tamanho *nt*. Cada *thread* verifica, após a atualização de *Lbest*, o menor resultado obtido entre as suas partículas e o armazena no vetor *Best*, na posição indicada por seu *tid*. Em seguida, a *thread Mestre* realiza uma busca sequencial em *Best* e armazena o menor resultado obtido na primeira posição desse vetor. As outras *threads* consultam essa posição durante a verificação da condição de parada. De forma a reduzir o *overhead* gerado pelo sincronismo na eleição do melhor valor do enxame (linhas de 22 a 27 do Algoritmo 2), essa operação é realizada somente após certo número de iterações, escolhido empiricamente como 20. Esse intervalo de iterações é utilizado apenas na eleição da partícula com melhor aptidão do enxame. Não há perda de informação de convergência entre as partículas.

3.2. IMPLEMENTAÇÃO EM *MESSAGE PASSING INTERFACE CHAMELEON*

O Algoritmo 3 apresenta um esboço do algoritmo PSO implementado em MPICH. Cada processo MPI executa um grupo de partículas $t=n/p$, onde n é o número de partículas do enxame e p, o número de processos. Ao iniciar o ambiente MPI, os processos inicializam seu gerador de números pseudoaleatórios com uma semente somada ao valor de seu *rank*, que é um número que identifica o processo, para gerar sequências diferentes. Em seguida, são inicializadas todas as informações das partículas referentes a cada processo. Os processos realizam a computação das novas velocidades e posição, cálculo de *fitness* e atualização de *Pbest* e *Lbest* para o seu grupo de partículas em paralelo.

De forma a manter o comportamento de um único enxame, cada processo envia mensagem, referente à primeira e à última partícula do seu grupo, para o processo (*rank*-1) *mod n* e para o processo (*rank*+1) *mod n*, respectivamente. Deste modo, a comunicação via mensagem é realizada somente entre a primeira e a última partícula de cada processo (linhas 10 e 11 do Algoritmo 3). As outras partículas se comunicam apenas dentro do próprio processo. Com o objetivo de implementar uma condição de parada entre os processos paralelos, após a seleção de *Best*, que representa a partícula com a melhor aptidão do enxame, todos os processos enviam mensagem ao processo *Mestre* com o valor e a posição de *Best*. O processo *Mestre*, então, verifica se, entre os resultados obtidos, algum satisfaz a condição de parada. Caso positivo, o processo *Mestre* ativa uma *flag* de saída e a envia por mensagem a todos os processos (linha 22 do Algoritmo 3). De posse da *flag* ativada, os processos encerram o algoritmo e o processo *Mestre* retorna o valor e posição encontrada. As operações de busca pelo melhor resultado de cada processo (linhas 16 a 22 do Algoritmo 3) são realizadas a cada 20 iterações, de modo a reduzir o *overhead* de comunicação entre os processos.

3.3. IMPLEMENTAÇÃO EM *COMPUTE UNIFIED DEVICE ARCHITECTURE*

O Algoritmo 4 apresenta uma visão geral do algoritmo PSO implementado em CUDA. Cada partícula é mapeada em uma única *thread*. O algoritmo é organizado logicamente em b blocos, onde cada bloco é composto de t *threads*. Para gerar um enxame com n partículas, são necessárias $n=b\times t$ *threads*. Quatro *kernels* foram utilizados para implementar o *Parallel* PSO (PPSO). O primeiro cria e inicializa os geradores de números aleatórios. Em seguida, inicializa os valores de velocidade, posição, *Pbest* e *Lbest*. O segundo atualiza a velocidade e a posição de cada partícula. O terceiro *kernel* realiza o cálculo da função objetivo e a atualização de *Pbest*. O quarto atualiza *Lbest* comparando os resultados alcançados pelas partículas vizinhas. Em seguida, verifica o melhor resultado obtido entre as partículas do enxame. A solução *multikernel* foi adotada devido ao fato de CUDA não possuir

```
1:  seja nt = número de threads;
2:  #pragma omp parallel // inicia uma região paralela
3:  início da região paralela;
4:  tid := omp_get_thread_num();
5:  srand(seed + tid);
6:  #pragma omp for schedule(static)
7:  para i := 0 → n faça
8:    inicialize as informações da partícula i;
9:  fim para
10: repita
11:   #pragma omp for schedule(static)
12:   para i := 0 → n faça
13:     atualize v_ij e x_ij; calcule Fitness_i; atualize Pbest_ij ;
14:   fim para
15:   #pragma omp for schedule(static)
16:   para i := 0 → n faça
17:     atualize Lbest();
18:     se (f(Lbest_ij)< Best_tid) então
19:       atualize Best_tid;
20:     fim se
21:   fim para
22:   se tid = Mestre então
23:     para t := 0 → nt − 1 faça
24:       obtenha o menor valor em Best_t ;
25:     fim para
26:   fim se
27:   sincronize threads;
28: até condição de parada
29: fim da região paralela
30: retorne o resultado e a posição correspondente;
```

Algoritmo 2. ***Particle Swarm Otimization*** **paralelo implementado em** ***Open Multi-Processing*****.**

uma instrução de sincronismo entre *threads* de blocos diferentes. De forma a reduzir o *overhead* gerado pela transferência de *Best* da GPU para a CPU, essa operação é realizada somente após certo número de iterações, escolhido empiricamente como 20.

O primeiro *kernel* cria e inicializa na memória global *b*×*t* geradores de números aleatórios, um para cada *thread*, baseados na biblioteca *curand.h* (NVIDIA, 2010b). Em seguida, inicializa as informações básicas das partículas, tais como velocidade, posição, *fitness*, *Pbest* e *Lbest*. O segundo *kernel*, cujo código é exibido no Algoritmo 5, cria *b*×*t threads* que irão realizar o calculo da próxima velocidade e posição para cada partícula. Nota-se que cada *thread* executa um laço, com o número de iterações do tamanho da dimensão *d*, para atualizar os valores de cada dimensão da partícula *tid*. Em seguida, o terceiro *kernel*, cujo código é exibido no Algoritmo 6, também gera *b*×*t threads* que irão executar o cálculo de *fitness* juntamente com a atualização de *Pbest*.

Os detalhes do *kernel* atualize *Lbest* são exibidos no Algoritmo 7. Os vizinhos da partícula *tid* são as partículas (*tid*+1) *mod n* e (*tid*–1) *mod n*. Assim, a atualização da melhor posição local é realizada baseada na sua vizinhança. A atualização do melhor resultado do enxame *Best* é realizada após a eleição de *Lbest* das partículas. A *thread* com *tid* =0 realiza sequencialmente a atualização do valor e da posição referente a *Best* de acordo com o menor valor encontrado no vetor *Lbest* e o armazena na memória global. No final da iteração, *Best* é enviado para a CPU e utilizado para verificação da condição de parada.

4. RESULTADOS

As implementações foram avaliadas utilizando-se diferentes arranjos de partículas e dimensões, conforme exibidos

na Tabela 1, na otimização de quatro funções de *benchmark*. O aumento do número de dimensões implica uma queda exponencial na taxa de convergência (ENGELBRECHT, 2005). Desta forma, foi adotada a proporção de 4 vezes mais partículas que número de dimensões nos casos utilizados. A análise permite a asserção do desempenho do algoritmo, bem como suas eficiências na otimização. Esses diferentes arranjos de partículas e dimensões, que serão citados como (partículas, dimensões), fornecem casos com diferentes custos computacionais e complexidade do problema. Para a realização das avaliações, foi utilizado um coeficiente de inércia iniciando em 0,99, com um decrescimento linear até 0,2 e v_{max} sendo calculado utilizando-se um δ de 0,2. Apesar do uso comum do coeficiente de inércia de 0,9 até 0,4 (ENGELBRECHT, 2005), nos testes experimentais foram obtidos melhores resultados com os valores adotados. O critério de parada adotado é terminar a execução quando uma aceitável solução for encontrada, com um erro menor do que 10^{-4}, ou o limite de 6.000 iterações for atingido.

O *hardware* utilizado para executar os casos em OpenMP e MPICH é Cluster Octane III SGI composto de 4 nós conectados via Gigabit-Ethernet. Cada nó contém 2 processadores Intel Xeon de 2,4 GHz dotados de 4 núcleos HT cada. A implementação em GPU explorou a GPU NVIDIA GTX 460, contendo 7 SM, onde cada SM inclui 48 núcleos de 1,3 GHz. Cada caso foi executado 50 vezes com diferentes valores de sementes, para geração dos números aleatórios usados pelo algoritmo, de forma a avaliar o desempenho, a eficiência e a confiabilidade do algoritmo PSO.

A Figura 2 apresenta os resultados obtidos pela implementação do algoritmo PSO nas arquiteturas de multicomputador, multiprocessador e GPU (os valores numéricos de tempo de execução referente a esses gráficos podem ser obtidos na Tabela 2). Referente à implementação em MPICH, antes de executar os casos, foi realizada uma avaliação do desempenho, utilizando-se a função f_1, de acordo com o arranjo de partículas por processo e número de processos, de modo a executar cada caso com a sua melhor configuração. A Tabela 3 exibe os

```
1:  seja p = número de processos; seja n = número de partículas;
2:  faça t := n/p número de partículas por processo;
3:  MPI Init(); // inicia o ambiente de execução MPI
4:  srand(seed + rank);
5:  inicie as informações das partículas do processo;
6:  repita
7:    para i := 0 → t − 1 faça
8:      atualize x_ij e v_ij; calcule fitness_i; atualize Pbest_ij
9:    fim para
10:   envie mensagem com Pbest_(t−1)j para o processo (rank + 1) mod n;
11:   envie mensagem com Pbest_(0)j para o processo (rank − 1) mod n;
12:   para i := 0 → t − 1 faça
13:     atualize Lbest_tj;
14:     atualize Best_t;
15:   fim para
16:   envie mensagem com Best de rank para o processo Mestre;
17:   se rank = Mestre então
18:     se Best ≤ erro então
19:       ative a flag de saída;
20:     fim se
21:   fim se
22:   Mestre envia mensagem com a flag de saída para os processos;
23: até flag ativada
24: MPI Finalize();
25: retorne o resultado e a posição correspondente;
```

Algoritmo 3. ***Particle Swarm Otimization*** **paralelo implementado em** ***Message Passing Interface Chameleon*****.**

```
1:  seja b = número de blocos; seja t = número de threads;
2:  kernel <<< b, t >>> inicialize as informações das partículas
3:  repita
4:    kernel <<< b, t >>> calcule a velocidade e posição;
5:    kernel <<< b, t >>> calcule fitness e Pbest;
6:    kernel <<< b, t >>> atualize Lbest;
7:    transfira Best para a CPU;
8:  até condição de parada
9:  retorne o resultado e a posição correspondente;
```

Algoritmo 4. *Particle Swarm Otimization* paralelo implementado em *Compute Unified Device Architecture*.

```
1:  faça tid = threadIdx + blockIdx × blockDim;
2:  para j = 0 → d − 1 faça
3:    calcule v[tid × d + j]; calcule x[tid × d + j];
4:  fim para
```

Algoritmo 5. *Kernel* calcule a velocidade e a posição.

```
1:  Seja tid = threadIdx + blockIdx × blockDim;
2:  para j = 0 → d − 1 faça
3:    calcule fitness;
4:  fim para
5:  se (fitness[tid] < Pbest[tid]) então
6:    para j = 0 → d − 1 faça
7:      Pbestx[tid × d + j] := x[tid × d + j];
8:    fim para
9:  fim se
```

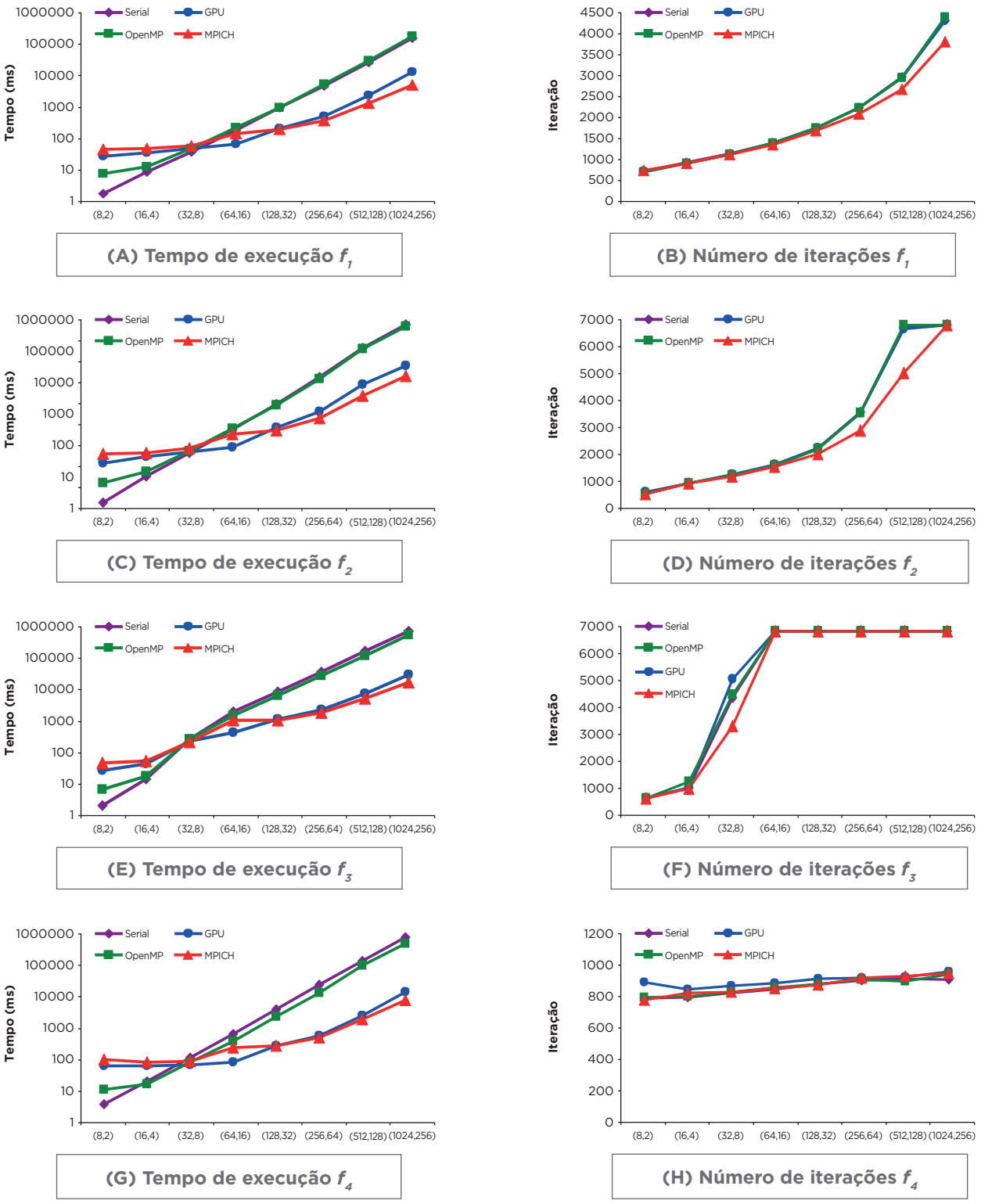
Algoritmo 6. *Kernel* calcule *fitness* e *Pbest*.

```
1:  seja tid = threadIdx + blockIdx × blockDim
2:  se (Pbest[(tid + 1)mod n] < Lbest[tid]) então
3:    para j = 0 → d − 1 faça
4:      Lbestx[tid × d + j]:= Pbestx[((tid + 1)mod n)×d + j];
5:    fim para
6:  fim se
7:  se (Pbest[(tid − 1)mod n] < Lbest[tid]) então
8:    para j = 0 → d − 1 faça
9:      Lbestx[tid × d + j]:= Pbestx[((tid − 1)mod n)×d + j];
10:   fim para
11: fim se
12: se (tid = 0) então
13:   atualize Best;
14: fim se
```

Algoritmo 7. *Kernel* atualize *Lbest*.

Tabela 1. Arranjos de partículas e dimensões utilizados nas avaliações.

Casos	1	2	3	4	5	6	7	8
Partícula	8	16	32	64	128	256	512	1.024
Dimensão	2	4	8	16	32	64	128	256



GPU: *Graphics Processing Unit*; OpenMP: *Open Multi-Processing*; MPICH: *Message Passing Interface Chameleon*.

Figura 2. Tempo de execução e número de iterações das implementações paralelas.

Tabela 2. Média e desvio padrão do tempo de execução das implementações paralelas e serial.

f(x)	Casos	Tempo (milissegundos)							
		Serial		GPU		OpenMP		MPICH	
		Média	DP	Média	DP	Média	DP	Média	DP
f_1	(8,2)	1,8	0,01	28,0	0,02	7,5	0,22	47,0	0,09
	(16,4)	8,6	0,33	36,0	0,13	12,6	0,24	50,2	0,12
	(32,8)	38,1	0,89	47,1	0,44	48,5	0,78	58,4	0,23
	(64,16)	187,7	1,65	65,0	0,51	217,4	1,43	145,6	0,87
	(128,32)	958,9	3,44	214,8	1,12	986,3	3,27	194,1	1,34
	(256,64)	4835,7	5,31	503,0	1,02	5364,2	4,99	384,3	2,96
	(512,128)	26128,3	6,98	2294,8	3,67	29878,6	8,32	1296,9	7,46
	(1024,256)	153219,2	7,76	12849,5	8,64	174389,2	10,87	5105,3	11,2
f_2	(8,2)	1,6	0,01	24,2	0,09	6,3	0,09	46,1	0,41
	(16,4)	9,7	0,54	36,9	0,12	13,6	0,11	48,1	0,42
	(32,8)	48,2	0,43	52,2	0,22	54,9	0,42	69,8	0,56
	(64,16)	254,6	1,01	74,2	0,87	260,2	1,06	181,5	1,06
	(128,32)	1397,5	2,88	279,7	1,34	1331,0	3,01	237,2	1,35
	(256,64)	9089,3	4,55	831,9	4,56	8472,0	5,17	532,9	1,99
	(512,128)	69140,1	8,23	5519,3	8,56	66044,5	13,31	2607,5	2,26
	(1024,256)	345288,4	16,09	20655,4	12,4	302410,5	14,45	10253,5	5,97
f_3	(8,2)	2,3	0,01	26,2	0,46	7,1	0,10	46,3	0,15
	(16,4)	14,4	0,26	42,0	0,51	18,5	0,12	50,9	0,13
	(32,8)	233,9	0,73	210,6	0,98	234,4	1,06	195,0	0,56
	(64,16)	1541,6	1,02	361,4	1,45	1247,9	3,74	857,2	0,75
	(128,32)	6296,7	3,91	940,5	1,90	4855,6	3,68	904,7	1,01
	(256,64)	25809,5	4,77	1797,2	4,49	19731,7	4,32	1437,1	1,65
	(512,128)	106018,9	5,34	5618,6	9,47	79127,4	9,25	4103,6	4,61
	(1024,256)	428441,5	7,46	21007,1	13,40	335269,9	22,5	12507,7	8,77
f_4	(8,2)	4,3	0,02	40,3	0,29	10,3	0,12	59,1	0,39
	(16,4)	16,5	1,21	39,5	0,57	13,7	0,12	49,8	0,34
	(32,8)	64,6	1,56	41,7	0,43	48,6	0,21	51,3	0,34
	(64,16)	258,0	2,54	50,6	0,41	166,6	3,76	117,0	0,88
	(128,32)	1062,1	1,78	129,7	1,09	699,5	4,55	131,4	1,05
	(256,64)	4320,2	4,32	234,1	1,55	2787,0	6,93	206,2	1,94
	(512,128)	17532,8	5,77	733,7	2,57	13222,3	10,74	593,4	3,01
	(1024,256)	69254,9	9,81	2948,6	3,21	48385,8	9,91	1813,4	3,71

GPU: *Graphics Processing Unit*; OpenMP: Open *Multi-Processing*; MPICH: *Message Passing Interface Chameleon;* DP: desvio-padrão.

Tabela 3. Análise de desempenho, em relação à implementação serial, de diferentes combinações de processos e partículas por processo do algoritmo *Particle Swarm Otimization* paralelo implementado em *Message Passing Interface Chameleon* para a função f_1.

Função	Processos	Part/proc	Tempo	Desempenho
(8, 2)	4	2	47,0	0,04
(16, 4)	4	4	50,2	0,17
	8	2	89,13	0,09
(32, 8)	4	8	58,4	0,65
	8	4	110,23	0,35
	16	2	110,57	0,34
(64, 16)	4	16	174,9	1,07
	8	8	145,6	1,29
	16	4	153,5	1,22
(128, 32)	4	32	268,9	3,56
	8	16	236,8	4,05
	16	8	194,1	4,94
	32	4	248,3	3,86
(256, 64)	4	64	914,4	5,28
	8	32	604,1	8,00
	16	16	447,8	10,79
	32	8	384,3	12,58
	64	4	512,0	9,44
(512, 128)	4	128	4094,5	6,38
	8	64	2363,5	10,94
	16	32	1563,2	16,71
	32	16	1296,9	20,15
	64	8	1580,0	16,53
(1024, 256)	4	256	21837,1	7,01
	8	128	11366,9	13,47
	16	64	6551,6	23,38
	32	32	5105,3	30,01
	64	16	5218,5	29,36

Part: partícula; proc: processo.

dados dessa avaliação. Os resultados demonstram que a utilização de uma configuração mais adequada permite alcançar um aumento no desempenho de até 4,2 vezes, referente ao caso (1024, 256), em comparação com as outras configurações.

De acordo com os resultados obtidos, podemos observar que, nos casos de poucas partículas e dimensões, (8,2) e (16,4), a implementação serial alcançou um melhor resultado em menor tempo de execução. A partir do caso (32,8), as implementações em CUDA e MPICH começaram a obter um menor tempo de execução, alcançando um ganho de 23 e 38 vezes na função f_4 em relação à implementação serial, respectivamente. A implementação em OpenMP alcançou um ganho de 1,55 vezes a implementação serial, também na função f_4, que possui um maior custo computacional.

Em relação à eficiência do algoritmo PSO, é possível observar que o número de iterações para atingir o resultado esperado foi semelhante ou até melhor que a implementação serial, como no caso da implementação em MPICH (Figura 2A e B). Isso demonstra que, mesmo paralelizado, o algoritmo manteve uma eficiência igual ou superior ao algoritmo serial na otimização de todos os casos para todas as funções.

5. CONCLUSÃO

Este artigo apresentou a implementação paralela do algoritmo PSO em arquiteturas de alto desempenho. Este estudo foi motivado pelo baixo desempenho do algoritmo PSO sequencial quando aplicado na solução de problemas complexos que envolvam um grande número de dimensões e partículas. O algoritmo PSO foi implementado em três diferentes arquiteturas de alto desempenho: multiprocessador, multicomputador e GPU. O *hardware* utilizado para executar as implementações em OpenMP e MPICH foi o SGI Octane III, composto de 4 nós conectados via uma rede *Gigabit-Ethernet*. Cada nó contem 2 processadores Intel Xeon de 2,4 GHz, sendo dotados de 4 núcleos HT cada. A implementação em GPU explorou a GPU NVIDIA GTX 460, que proporciona 7 SM, onde cada SM inclui 48 núcleos de 1,3 GHz. As implementações foram avaliadas utilizando-se diferentes arranjos de partículas e dimensões, proporcionando diferentes custos e complexidades, na otimização de 4 funções de *benchmark*.

A implementação em MPICH obteve o melhor resultado, seguida da implementação em CUDA, nos casos com maiores números de partículas e dimensões. No melhor caso, o MPICH chegou a obter um desempenho 2 vezes maior que a implementação em CUDA e até 34 vezes maior que a implementação em OpenMP. Nota-se que, em termos de custo, o *cluster* Octane III possui um custo 9 vezes maior que a GPU. Em relação à eficiência na otimização, as implementações paralelas apresentaram comportamento de modo semelhante ou até melhor que a implementação serial. Nos casos com poucas partículas e dimensões, a implementação serial apresentou o melhor desempenho. De um modo geral, a adoção da verificação da condição de parada a cada 20 iterações contribuiu para reduzir o sincronismo e a comunicação entre os processos ou *threads*, incrementando o desempenho das aplicações em todas as arquiteturas em aproximadamente 90%, sem perda de informação de convergência.

Como trabalhos futuros, é nossa intenção otimizar a execução de modelos numéricos oceanográficos paralelos, assim como verificar seu desempenho em diferentes arquiteturas, utilizados para o melhor conhecimento e a eficaz utilização do meio ambiente marinho, no interesse da Marinha do Brasil.

REFERÊNCIAS

CALAZAN, R.; NEDJAH, N.; MOURELLE, L.M.A hardware accelerator for Particle Swarm Optimization. *Applied Soft Computing*, 2013. In press.

DENNIS, J.B.; HORN, E.C.V. Programming semantics for multiprogrammed computations. *Communications of the ACM*, v. 9, n. 3, p. 143-155, 1966.

ENGELBRECHT, A.P. *Fundamentals of computational swarm intelligence*. New Jersey: John Wiley & Sons, 2005.

GROPP, W.; SMITH, B. *Users manual for the chameleon parallel programming tools*. Argonne, IL: Argonne National Laboratory, 1993.

KENNEDY, J.; EBERHART, R. Particle Swarm Optimization. In: *IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORK*. 1995. Perth, Australia: IEEE Press, 1995. p. 1942-1948.

KIRK, D.B.; HWU, W.W. *Programming massively parallel processors: a hands-on approach*. Burlington, MA: Morgan Kaufmann, 2010.

KOH, B. et al. Parallel asynchronous particle swarm optimization. *International Journal for Numerical Methods in Engineering*, v. 67, n. 4, p. 578-595, jul. 2006.

MA, Q.; LEI, X.; ZHANG, Q. Mobile robot path planning with complex constraints based on the second-order oscillating particle swarm optimization algorithm. In: 2009 WRI WORLD CONGRESS ON COMPUTER SCIENCE AND INFORMATION ENGINEERING, 5., 2009, Los Angeles. *Proceedings*... Los Angeles: IEEE Press, 2009. p. 244-248.

MESSAGE PASSING INTERFACE FORUM. *A Message Passing Interface Standard*. MPI Forum, 2012. Disponível em: <http://www.mpi-forum.org>. Acesso em: abr. 2002.

MOLGA, M.; SMUTNICKI, C. *Test functions for optimization needs*. 2005. Disponível em: <http://www.zsd.ict.pwr.wroc.pl/files/docs/functions.pdf>. Acesso em: 16 ago. 2013.

NVIDIA. *Manual da Nvidia sobre CUDA C Programming Guide*, 2010. Disponível em: <http://developer.nvidia.com/object/cuda 3 2 downloads>. Acesso em: 20 ago. 2010a.

_____. *Relatório da Nvidia sobre CuRAND Lybrary*, 2010. Disponível em: <http://developer.nvidia.com/cuda/curand>. Acesso em 15 set. 2010b.

PATTERSON, D.; HENNESSY, J. Computer organization and design: the hardware/software interface. 4. ed. Burlington, MA: Morgan Kaufmann, 2011.

RASMUSSEN, T.; KRINK, T. Improved Hidden Markov Model training for multiple sequence alignment by a particle swarm optimization-evolutionary algorithm hybrid. *BioSystems*, v. 72, n. 1-2, p. 5-17, 2003.

REYNOLDS, C.W. Flocks, herds and schools: a distributed behavioral model. *Computer Graphics*, v. 21, n. 4, p. 25-34, 1987.

SEDIGHIZADEH, D.; MASEHIAN, E. Particle swarm optimization methods, taxonomy and applications. *International Journal of Computer Theory and Engineering*, v. 1, n. 4, p. 486-502, oct. 2009.

SHUTTE, J.F. et al. Parallel global optimization with the particle swarm algorithm. *International Journal for Numerical Methods in Engineering*, v. 61, n. 13, p. 2296-2315, dec. 2004.

VERONESE, L.; KROHLING, R.A. Swarm's flight: accelerating the particles using C-CUDA. In: IEEE CONGRESS ON EVOLUTIONARY COMPUTATION, 11., 2009, Trondheim. *Proceedings*... Trondheim: IEEE Press, 2009. p. 3264-3270.

WANG, D. et al. Parallel multi-population particle swarm optimization algorithm for the uncapacitated facility location problem using OpenMP. In: CEC 2008. EVOLUTIONARY COMPUTATION, 2008. IEEE WORLD CONGRESS ON COMPUTATIONAL INTELLIGENCE, 2008, Hong Kong. *Proceedings*... Hong Kong: IEEE Press, 2008. p. 1214-1218.

YANG, G.; ZHANG, R. Path planning of AUV in turbulent ocean environments used adapted inertia-weight PSO. In: INTERNATIONAL CONFERENCE ON NATURAL COMPUTATION. ICNC'09, 3., 2009, Tianjin. *Proceedings*... Tianjin: IEEE Press, 2009..p. 299-302.

ZHOU, Y.; TAN, Y. GPU-based parallel particle swarm optimization. In: IEEE CONGRESS ON EVOLUTIONARY COMPUTATION, 11., 2009, Trondheim. *Proceedings*... Trondheim: IEEE Press, 2009. p. 1493-1500.