Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Faculdade de Engenharia

Rogério de Moraes Calazan

Otimização por Enxame de Partículas em Arquiteturas Paralelas de Alto Desempenho

Rio de Janeiro
2013

Rogério de Moraes Calazan

Otimização por Enxame de Partículas em Arquiteturas Paralelas de Alto Desempenho

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Sistemas Inteligentes e Automação.

Orientadora: Prof.ª Dr.ª Nadia Nedjah
Coorientadora: Prof.ª Dr.ª Luiza de Macedo Mourelle

Rio de Janeiro
2013

CATALOGAÇÃO NA FONTE
UERJ / REDE SIRIUS / BIBLIOTECA CTC / B

C143 Calazan, Rogério de Moraes
Otimização por enxame de partículas em arquiteturas paralelas de alto desempenho/Rogério de Moraes Calazan. - 2013.
138f.

Orientadora: Nadia Nedjah.
Coorientadora: Luiza de Macedo Mourelle.
Dissertação (Mestrado) – Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia.

1. Engenharia Eletrônica - Dissertações. 2. Sistemas inteligentes - Dissertações. 3. Programação paralela (Computação) - Dissertações. 4. Processamento paralelo (Computadores) - Dissertações. I. Nedjah, Nadia. II. Mourelle, Luiza de Macedo. III. Universidade do Estado do Rio de Janeiro. III. Título.

CDU 004.272.2

Assinatura

Data

Rogério de Moraes Calazan

Otimização por Enxame de Partículas em Arquiteturas Paralelas de Alto Desempenho

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Sistemas Inteligentes e Automação.

Aprovado em: 21 de fevereiro de 2013.

Banca Examinadora:

Prof.ª Dr.ª Nadia Nedjah (Orientadora)
Faculdade de Engenharia - UERJ

Prof.ª Dr.ª Luiza de Macedo Mourelle (Coorientadora)
Faculdade de Engenharia - UERJ

Prof. Dr. Valmir Carneiro Barbosa
Universidade Federal do Rio de Janeiro - UFRJ

Prof. Dr. Alberto Ferreira de Souza
Universidade Federal do Espírito Santo - UFES

Rio de Janeiro
2013

AGRADECIMENTOS

Agradeço primeiramente a Deus pelo dom da vida. A minha mãe pelo exemplo e pela educação. A minha esposa pela paciência e pela compreensão nos momentos de ausência.

Agradeço aos professores do PEL-UERJ pelas lições aprendidas nestes dois anos de curso. Em especial, agradeço às professoras Nadia Nedjah e Luiza de Macedo Mourelle, não apenas pela orientação neste trabalho e pela oportunidade de desenvolvê-lo no LSAC, mas pela disponibilidade e valiosas sugestões que enriqueceram este trabalho. Aos Professores Valmir Carneiro Barbosa e Alberto Ferreira de Souza pela participação na comissão examinadora, bem como pelas contribuições e ensinamentos.

Agradeço aos colegas de mestrado Alexandre, Fábio, Luneque, Leandro, Nicolás, Paulo e Rafael. Obrigado pelas conversas sobre as dúvidas individuais e coletivas. Os momentos de descontração no cafezinho e pela companhia na pizza durante os estudos no laboratório.

Agradeço à Marinha do Brasil, em especial a DCTIM e ao IEAPM, pela liberação e apoio durante todo o curso de mestrado, sem o qual não poderia ter desenvolvido este trabalho em tempo integral.

Reunir-se é um começo. Manter juntos é um progresso. Trabalhar juntos é sucesso.

Henry Ford

RESUMO

CALAZAN, Rogério de Moraes. *Otimização por enxame de partículas em arquiteturas paralelas de alto desempenho.* 2013. 138f. Dissertação (Mestrado em Engenharia Eletrônica) – Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2013.

A Otimização por Enxame de Partículas (PSO, *Particle Swarm Optimization*) 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 obtido, 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 o aumento de desempenho, a minimização do tempo de resposta e melhoria da qualidade do resultado final. Nesta dissertação, o algoritmo PSO é paralelizado utilizando três estratégias que abordarão diferentes granularidades do problema, assim como dividir o trabalho de otimização entre vários subenxames cooperativos. Um dos algoritmos paralelos desenvolvidos, chamado PPSO, é implementado diretamente em *hardware*, utilizando uma FPGA. Todas as estratégias propostas, PPSO (*Parallel PSO*), PDPSO (*Parallel Dimension PSO*) e CPPSO (*Cooperative Parallel PSO*), são implementadas visando às arquiteturas paralelas baseadas em multiprocessadores, multicomputadores e GPU. Os diferentes testes realizados mostram que, nos problemas com um maior número de partículas e dimensões e utilizando uma estratégia com granularidade mais fina (PDPSO e CPPSO), a GPU obteve os melhores resultados. Enquanto, utilizando uma estratégia com uma granularidade mais grossa (PPSO), a implementação em multicomputador obteve os melhores resultados.

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

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, 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 to the characteristics of the architecture. This should increase performance, minimize response time and may even improve the quality of the final result. In this dissertation, the PSO algorithm is parallelized using three different strategies that consider different granularities of the problem, and the division of the optimization work among several cooperative sub-swarms. One of the developed parallel algorithms, namely PPSO, is implemented directly in hardware, using an FPGA. All the proposed strategies, namely PPSO (Parallel PSO), PDPSO (Parallel Dimension PSO) and CPPSO (Cooperative Parallel PSO), are implemented in a multiprocessor, multicomputer and GPU based parallel architectures. The different performed assessments show that the GPU achieved the best results for problems with high number of particles and dimensions when a strategy with finer granularity is used, namely PDPSO and CPPSO. In contrast with this, when using a strategy with a coarser granularity, namely PPSO, the multi-computer based implementation achieved the best results.

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

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE SIGLAS

d	número de dimensões
n	número de partículas
Pbest	Personal Best
ACO	Ant Colony Optimization
API	Application Programming Interface
CLB	Configurable Logic Blocks
CUDA	Compute Unified Device Arquitecture
FPGA	Field Programmable Gate Arrays
Gbest	Global Best
GPU	Graphics Processing Unit
HDL	Hardware Description Language
Lbest	Local Best
LuT	Look-up Table
NUMA	NonUniform Memory Access
PSO	Particle Swarm Optimization
SFU	Special Function Units
SIMD	Single Instruction, Multiple Data
SIMT	Single-Instruction Multiple-Threads
SMP	Shared Memory Multiprocessor
UART	Universal Asynchronous Receiver/Transmitter
VHDL	Very High Speed Integrated Circuits Hardware Description Language

SUMÁRIO

INTRODUÇÃO

OTIMIZAÇÃO por Enxame de Partículas, do inglês *Particle Swarm Optimization* ou PSO, foi introduzido por Kennedy e Eberhart (KENNEDY; 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 um bando de pássaros. Se um dos elementos do grupo descobrir um caminho, onde há facilidade de encontrar o alimento, os outros componentes do grupo tendem, instantaneamente, a também seguir este caminho. No PSO, cada partícula do enxame ajusta sua posição no espaço de busca de acordo com a melhor posição encontrada por todo o enxame (REYNOLDS, 1987).

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 para solucionar diversos problemas, como no planejamento de rota de veículo subaquático (YANG; ZHANG, 2009), de aeronave (SECREST, 2001) e de robôs (MA; LEI; ZHANG, 2009), biologia e bioinformática (RASMUSSEN; KRINK, 2003), trajetória de voo de mísseis (HUGHES, 2002), 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, o que usualmente é feito de modo sequencial na CPU, acarretando um baixo desempenho. Consequentemente, o algoritmo pode se tornar ineficiente tendo em vista o desempenho obtido, 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. Outro ponto é que, normalmente, os trabalhos de paralelização do algoritmo PSO, implementados em GPU comparam o seu ganho obtido em relação a CPU com o algoritmo

sequencial, o que pode levar a uma comparação injusta, pois o *speedup* obtido é muito maior do que poderia realmente ser, caso a implementação em CPU fosse otimizada. 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.

A computação de alto desempenho ou HPC (do inglês *High-Performance Computing*) se refere ao uso de supercomputadores ou *clusters* de vários computadores e mais recentemente de GPUs em tarefas que requerem grandes recursos de computação. Computação paralela é uma forma de computação em que vários cálculos são realizados simultaneamente, operando sob o princípio de que grandes problemas geralmente podem ser divididos em problemas menores, que então são resolvidos concorrentemente. Programas de computador paralelos são mais difíceis de programar que os sequenciais, pois a concorrência introduz diversas novas classes de defeitos potenciais, como a condição de corrida (PATTERSON; HENNESSY, 2011). A comunicação e a sincronização entre diferentes subtarefas é tipicamente uma das maiores barreiras para atingir grande desempenho em programas paralelos.

O paralelismo pode ser introduzido utilizando diferentes arquiteturas. Primeiramente, pode se explorar os núcleos presentes nas CPUs atuais utilizando o mesmo endereço compartilhado e o modelo de programação *Fork-Join*. Esta arquitetura é denominada de multiprocessadores com memória compartilhada (CHAPMAN; JOST; PAS, 2008). Em uma outra arquitetura, denominada de multicomputadores, os processadores são interligados através de uma rede de alta velocidade e colaboram entre si na computação paralela por meio de troca de mensagens (GROPP; LUSK; SKJELLUM, 1999). É possível ainda a utilização de um modelo híbrido que faz uso de ambas as arquiteturas.

Outra forma é a adição de um segundo processador especializado chamado de coprocessador. Ao assumir as tarefas que consomem tempo de processamento, os coprocessadores podem acelerar o desempenho do sistema (TANENBAUM; ZUCCHI, 2009). Nos últimos anos as GPUs adquiriram capacidade de paralelismo e vários núcleos de processamento com grande poder de computação. A GPU é especialmente adequada para resolver problemas em que os dados podem ser expressos de forma paralela, ou seja, o mesmo programa com diferentes dados pode ser executado por vários processadores em paralelo (PATTERSON; HENNESSY, 2011).

O objetivo desta dissertação é investigar e desenvolver estratégias de implementação paralela do algoritmo PSO em arquiteturas paralelas de alto desempenho, além de mostrar o ganho decorrente do paralelismo, não somente em termos de redução do tempo de execução, mas também na qualidade do próprio resultado da otimização. A principal contribuição deste projeto é a análise do comportamento em termos de eficiência e desempenho das diferentes implementações paralelas do algoritmo PSO em arquiteturas de alto desempenho.

Este trabalho propõe inicialmente paralelizar o algoritmo PSO utilizando três estratégias que abordam diferentes granularidades do problema e como divisão do trabalho entre vários subenxames cooperativos. Em seguida, uma versão paralela do PSO é implementada em *hardware* dedicado utilizando uma FPGA. Após isto, as 3 estratégias são implementadas em *software* na arquitetura de multiprocessadores, multicomputadores e GPU. As implementações são executadas, utilizando diferentes arranjos de enxame e dimensões, de forma a analisar as limitações e as vantagens de cada arquitetura bem como o desempenho e a qualidade dos resultados obtidos na solução de problemas complexos de otimização. Posteriormente, as implementações paralelas são comparadas de modo a avaliar o ganho em cada arquitetura.

Esta dissertação está organizada em sete capítulos, cujos conteúdos são resumidamente descritos a seguir.

Inicialmente, o Capítulo 1 apresenta uma introdução teórica sobre a otimização por enxame de partículas. São descritos os algoritmos *Global Best PSO* e *Local Best PSO*. Valores de parâmetros são apresentados e medidas de desempenho são abordadas de forma a avaliar a otimização dos algoritmos propostos. Algumas topologias utilizadas pelo PSO como estrutura social são mostradas. É apresentada também uma breve introdução sobre projetos de algoritmos paralelos e seus principais conceitos. É feito um levantamento bibliográfico dos trabalhos mais relevantes que trataram de paralelização do algoritmo PSO.

O Capítulo 2 introduz três estratégias de implementação paralela do algoritmo PSO. A primeira estratégia considera o fato de que as computações realizadas pelas partículas são quase independentes e, portanto podem ser realizadas em paralelo. A segunda estratégia considera o fato de que as operações internas aos processos que implementam as partículas também podem ser paralelizadas, proporcionando uma decomposição das

operações realizadas em cada dimensão do problema. A terceira estratégia é baseada no algoritmo serial CPSO-S_k, desenvolvido por Van den Bergh em (BERGH et al., 2002), cuja ideia principal é subdividir o vetor de dimensões do problema original entre subenxames que irão otimizar um subproblema. Os subenxames cooperarão entre si na solução do problema original. Por fim, são abordadas vantagens e desvantagens das topologias em estrela e anel.

O Capítulo 3 mostra uma implementação em *hardware* do algoritmo PSO utilizando a aritmética de ponto flutuante. É realizada uma breve introdução dos recursos da FPGA e ao processador MicroBlaze™. Em seguida, é descrita a arquitetura do coprocessador PSO, seu controlador e suas unidades funcionais.

O Capítulo 4 descreve as implementações dos algoritmos propostos, no Capítulo 2, implementados utilizando a API OpenMP e o MPICH. Para isso, é realizada uma breve introdução a arquitetura de multiprocessador que utiliza o modelo *Fork-Join* e a arquitetura de multicomputador que utiliza a biblioteca de troca de mensagens. Em seguida, são descritas as implementações dos algoritmos nas arquiteturas de multiprocessador, multicomputador e utilizando a programação híbrida do OpenMP com o MPICH.

O Capítulo 5 apresenta uma breve introdução ao modelo de programação CUDA, a hierarquia de memória e a organização em *grade*, *blocos* e *threads*. Além disso, é mostrada a arquitetura *multithreading* da GPU. Em seguida, são descritas as implementações em CUDA dos algoritmos propostos no Capítulo 2 juntamente com o mapeamento entre *blocos* e *threads* adotado.

O Capítulo 6 apresenta a análise dos resultados obtidos pelas implementações descritas nos Capítulos 3, 4 e 5. Diferentes arranjos de partículas e dimensões são utilizados para otimizar 4 funções clássicas consideradas como *benchmarks*. A análise permite a asserção do desempenho dos algoritmos propostos, bem como da sua eficiência na otimização. São apresentados os resultados obtidos pelo coprocessador HPSO e realizada sua comparação com o processador MicroBlaze™. São apresentados também os resultados dos algoritmos propostos implementados em OpenMP, MPICH e CUDA.

Finalmente, o Capítulo 7 completa esta dissertação, apresentando as principais conclusões obtidas do desenvolvimento do presente trabalho. São apresentadas também possíveis melhorias a partir dos algoritmos propostos e as direções para futuros estudos envolvendo o algoritmo PSO.

Capítulo 1

OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS

INTELIGÊNCIA de Enxame (*Swarm Intelligence*) (BENI; WANG, 1993) é uma área da inteligência artificial baseada no comportamento coletivo e descentralizado de indivíduos que interagem uns com os outros e com o ambiente. Este campo de pesquisa foi inspirado no comportamento de grupos de indivíduos, como insetos e pássaros. Um algoritmo baseado no comportamento de enxame é composto por um grupo de agentes simples que cooperam entre si para solucionar um problema complexo. Cada membro do enxame interage com os demais indivíduos e com o ambiente, não havendo um controle central sobre o sistema. Alguns modelos foram desenvolvidos baseados neste conceito. Entre os mais difundidos estão a otimização por enxame de partículas ou PSO (*Particle Swarm Optimization*) (KENNEDY; EBERHART, 1995), inspirada no voo de um bando de pássaros, e a otimização por colônia de formigas ou ACO (*Ant Colony Optimization*) (DORIGO; MANIEZZO; COLORNI, 1996).

O algoritmo PSO foi introduzido por Kennedy e Eberhart (KENNEDY; 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. Se um dos elementos do grupo descobre um caminho onde há facilidade para encontrar o alimento, os outros componentes do grupo tendem, instantaneamente, a seguir também este caminho (REYNOLDS, 1987).

A organização deste capítulo é descrita a seguir. Na Seção 1.1 é apresentada uma introdução ao algoritmo PSO. Nas Seções 1.2 e 1.3 são apresentados os modelos *Global Best PSO* e *Local Best PSO*, respectivamente. Na Seção 1.4 são descritos possíveis valores de parâmetros utilizados neste algoritmo. Na Seção 1.5 são exibidos diferentes

medidas que quantificam o desempenho do PSO. Na Seção 1.6 são apresentados algumas topologias utilizadas. A Seção 1.7.1 apresenta uma breve introdução sobre projeto de algoritmos paralelos. Na Seção 1.7.2 é realizado um levantamento bibliográfico sobre algumas implementações paralelas do algoritmo PSO.

1.1 Introdução ao PSO

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, observaram que este 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 (KENNEDY; EBERHART, 1995), que determinam sua movimentação. 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 desta posição (ENGELBRECHT, 2006).

Inicialmente, dois algoritmos do PSO foram desenvolvidos os quais se diferem pela definição da noção de vizinhança. Estes dois algoritmos, nomeados de *Gbest* e *Lbest* PSO, são detalhados na Seção 1.2 e 1.3 respectivamente.

1.2 Global Best PSO

O pseudocódigo do *Gbest* PSO, melhor global, é apresentado no Algoritmo 1. Neste algoritmo, a vizinhança de cada partícula é formada por todas as demais partículas do enxame. Desta forma, ele reflete uma topologia em estrela, mostrada na Seção 1.6, onde os círculos representam as partículas do enxame e as linhas ligam cada partícula a suas vizinhas.

Nesta topologia, o componente social da velocidade de uma partícula é influenciado por todas as outras partículas do enxame (ENGELBRECHT, 2006). Isto resulta em uma rápida taxa de convergência porém, eventualmente o enxame pode convergir prematuramente para um mínimo local.

Algoritmo 1 Algoritmo Global Best PSO

```
Crie e inicialize um enxame com n partículas;
repita
  para i := 1 → n faça
    Calcule o fitness da particula_i
    se Fitness_i ≤ Pbest_i então
      Atualize Pbest com a nova posição
    fim se;
    se Pbest_i ≤ Gbest então
      Atualize Gbest com a nova posição
    fim se;
    Atualize a velocidade da partícula utilizando a Equação 2
    Atualize a posição da partícula utilizando a Equação 4
  fim para;
até Condição de parada
retorne Melhor resultado
```

O primeiro passo do algoritmo PSO é realizar a distribuição das partículas no espaço de busca e inicializar seus parâmetros de controle. Segundo Engelbrecht(2006), a eficiência do algoritmo PSO é influenciada pela diversidade inicial do enxame. Desta forma, a inicialização das partículas requer atenção especial. Um método proposto em (ENGELBRECHT, 2006) inicializa os valores das posições das partículas aleatoriamente em um espaço de busca definido e inicializa a velocidade, referente a cada dimensão, com o valor 0, conforme

$$\begin{aligned} x_j(0) &= x_{min,j} + r_j(x_{max,j} - x_{min,j}) \\ v_j(0) &= 0 \end{aligned} \tag{1}$$

onde j representa a dimensão, x_{max} e x_{min} os valores máximo e mínimo do espaço de busca e r_j um fator aleatório entre $[0, 1]$. O valor de $Pbest$ também é inicializado com a posição da partícula no instante $t = 0$.

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, caso a aptidão da partícula seja melhor que a melhor posição anterior. Em seguida, é realizada a atualização da melhor posição do enxame, caso a aptidão da partícula seja melhor que todas as partículas do enxame.

Depois de efetuar a atualização da melhor posição do enxame, o algoritmo irá realizar a atualização da velocidade de cada partícula. A velocidade é o elemento que promove a capacidade de locomoção e no modelo melhor global, esta é calculada como

$$v_{ij}(t+1) = wv_{ij}(t) + c_1 r_1 \left(y_{ij} - x_{ij}(t)\right) + c_2 r_2 \left(y_j - x_{ij}(t)\right) \tag{2}$$

onde w é chamado de coeficiente de inércia, r_1 e r_2 são valores aleatórios do intervalo [0,1], c_1 e c_2 são constantes positivas, y_{ij} é a melhor posição visitada pela partícula i, na dimensão j, no passado e y_j é a melhor posição na dimensão j, atingida no passado, entre todas as partículas, que representa o melhor global definido como *Gbest*. A equação da velocidade é composta por três termos, que são:

- A *velocidade anterior*, $wv_{ij}(t)$, que é como uma memória da direção do movimento no passado mais recente. Este termo pode ser visto como um momento, prevenindo a partícula de mudanças drásticas de direção. Esta componente também é chamada de componente de inércia.

- O *componente cognitivo*, $c_1 r_1 \left(y_{ij} - x_{ij}(t)\right)$, que quantifica o desempenho da partícula i em relação a desempenhos anteriores. Este termo tem o efeito de atrair a partícula para sua melhor posição no passado.

- A *componente social*, $c_2 r_2 \left(y_j - x_{ij}(t)\right)$, que quantifica o desempenho da partícula i em relação ao desempenho de sua vizinhança. Este termo tem o efeito de atrair a partícula para a melhor posição encontrada pelo grupo de partículas.

Para limitar a velocidade da partícula, de modo que ela não saia do espaço de busca, são impostos limites, denominado $v_{max,j}$, para seus valores em cada dimensão do espaço de busca de acordo com

$$v_{ij}(t+1) = \begin{cases} v'_{ij}(t+1) & \text{se } v'_{ij}(t+1) < v_{max,j} \\ \\ v_{max,j} & \text{se } v'_{ij}(t+1) \geq v_{max,j} \end{cases} \tag{3}$$

onde $v'_{ij}(t+1)$ é calculado usando a equação 2. Se a velocidade da partícula exceder a velocidade máxima especificada, ela ficará limitada à velocidade máxima.

A posição de cada partícula é alterada conforme

$$x_{ij}(t+1) = v_{ij}(t+1) + x_{ij}(t) \tag{4}$$

onde $x(t+1)$ representa a posição atual e $x(t)$ a posição anterior. A velocidade guia o processo de otimização refletindo tanto a experiência da partícula, quanto a troca de informação 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. A troca de informação entre partículas refere-se ao comportamento social da equação da velocidade.

Após atualizar a velocidade e a posição de cada partícula, é verificada a condição de parada e então exibido o resultado final ou é 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 em 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 este critério é obvio que, para um número pequeno de iterações, o termino 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 ótimo global. Porém, este critério de parada impõe o conhecimento prévio do ótimo global.

- *Nenhuma melhoria é observada sobre um número de iterações.* A melhoria pode ser medida de diferentes modos. Por exemplo, quando em um determinado ciclo de iterações não é observado nenhuma melhoria no valor da aptidão do enxame ou a média da velocidade é aproximadamente zero, então o processo de busca pode ser terminado.

1.3 Local Best PSO

O pseudocódigo do *Lbest* PSO é apresentado no Algoritmo 2. Este algoritmo é semelhante ao algoritmo 1 porém, a vizinhança de cada partícula é formada por um subconjunto de partículas sobre a qual uma *Lbest* é selecionada. Assim, ele reflete uma topologia em anel, mostrada na Seção 1.6. Nesta topologia o componente social da velocidade de uma partícula é influenciado apenas por uma vizinhança limitada (ENGELBRECHT, 2006). Desta

forma, uma vizinhança menor tenta prevenir uma prematura convergência, mantendo múltiplos atratores (KENNEDY; MENDES, 2002). A seleção da vizinhança geralmente é realizada baseada nos índices das partículas. Cada partícula tenta imitar seu melhor vizinho, movendo-se na direção da melhor posição encontrada. É importante notar que as vizinhanças se sobrepõem, o que implica em uma troca de conhecimento entre as vizinhanças, isto garante que o enxame irá convergir para um mesmo ponto, designado pela melhor partícula do enxame.

Algoritmo 2 Algoritmo Local Best PSO

```
Crie e inicialize um enxame com n partículas
repita
  para i = 1 → n faça
    Calcule o fitness da particula_i
    se Fitness_i ≤ Pbest_i então
      Atualize Pbest com a nova posição
    fim se;
    se Pbest_i ≤ Lbest_i então
      Atualize Lbest com a nova posição
    fim se;
  fim para;
  para i = 1 → n faça
    Atualize a velocidade da partícula utilizando a Equação 5
    Atualize a posição da partícula utilizando a Equação 4
  fim para;
até Condição de parada
retorne Melhor resultado
```

A velocidade é calculada como

$$v_{ij}(t+1) = wv_{ij}(t) + c_1 r_1 \left(y_{ij} - x_{ij}(t)\right) + c_2 r_2 \left(y_{ij} - x_{ij}(t)\right) \tag{5}$$

onde y_{ij} é a melhor posição encontrada na vizinhança da partícula i na dimensão j, que representa o melhor local definido como *Lbest*. O calculo da posição é realizado utilizando a mesma Equação 4 do modelo *Gbest*.

1.4 Valores de Parâmetros

O valor de cada parâmetro do algoritmo PSO é fundamental no processo de busca, e por isso a importância de se definir em valores adequados.

- O *tamanho do Enxame* define a possibilidade de abranger uma determinada porção do espaço de busca em cada iteração do algoritmo. Um número grande de partículas

permite uma maior abrangência, mas exige um maior poder computacional. De acordo com (BERGH et al., 2002) estudos empíricos mostraram que o PSO consegue chegar a soluções ótimas utilizando um pequeno número de partículas, entre 10 e 30.

- O *coeficiente de inércia* w foi introduzido por (SHI; EBERHART, 1998a) e tem por finalidade controlar a relação entre explorar grandes espaços ou uma determinada localidade. As implementações iniciais do coeficiente de inércia utilizavam um valor fixo durante todo o processo de otimização, normalmente um valor próximo de 1. Atualmente, as implementações estão utilizando atualização dinâmica de w, ou seja, durante o processamento do algoritmo, este valor é modificado, começando com um valor alto e diminuindo gradativamente.

- O *coeficiente cognitivo* (c_1) e o *coeficiente social* (c_2) levam o algoritmo a um desempenho melhor se forem balanceados, ou seja, $c_1 = c_2$. Além disso de acordo com (ENGELBRECHT, 2006), trabalhos recentes indicam que é melhor que o coeficiente cognitivo seja maior que o social, e que $c_1 + c_2 \approx 4$. Na prática, bons resultados são alcançados utilizando $c_1 \approx c_2 \approx 1,49$.

- Os *fatores* r_1 e r_2 definem o teor estocástico das contribuições cognitiva e social do algoritmo. São selecionados valores aleatórios no intervalo entre [0,1] para cada um dos fatores.

- A correta definição do valor de v_{max} contribui para a realização de uma busca balanceada em termos de exploração do espaço de busca ou realização de uma busca mais refinada de uma região promissora. A *velocidade máxima* pode ser definida como uma fração do domínio de cada dimensão do espaço de busca, $v_{max,j} = \delta(x_{max,j} - x_{min,j})$, onde $x_{max,j}$ e $x_{min,j}$ são, respectivamente, o valor máximo e mínimo de cada dimensão do domínio, e δ é um valor no intervalo $(0, 1]$, cuja definição é dependente do problema (SHI; EBERHART, 1998b).

1.5 Medidas de Desempenho

Em (ENGELBRECHT, 2006), são apresentados diferentes medidas para quantificar o desempenho do processo de otimização do algoritmo PSO. Os critérios mais usados são de

acurácia, confiabilidade e eficiência.

- A *acurácia* se refere à exatidão da solução obtida, representada pela melhor posição global. Desta forma, a exatidão pode ser expressa como o erro entre a aptidão obtida no tempo t em relação à solução ótima x^*. A acurácia da solução pode ser definida conforme

$$acuracia(t) = |f(x(t)) - f(x^*)| \tag{6}$$

  onde $f(x(t))$ representa $fitness$ no tempo t e $f(x^*)$ a $fitness$ da solução ótima.

- A *confiabilidade* se refere ao percentual de simulações realizadas que atingiram uma determinada acurácia. A confiabilidade de um enxame pode ser definida conforme

$$confiabilidade(t, \epsilon) = \frac{N_{(\epsilon)}}{N} \times 100 \tag{7}$$

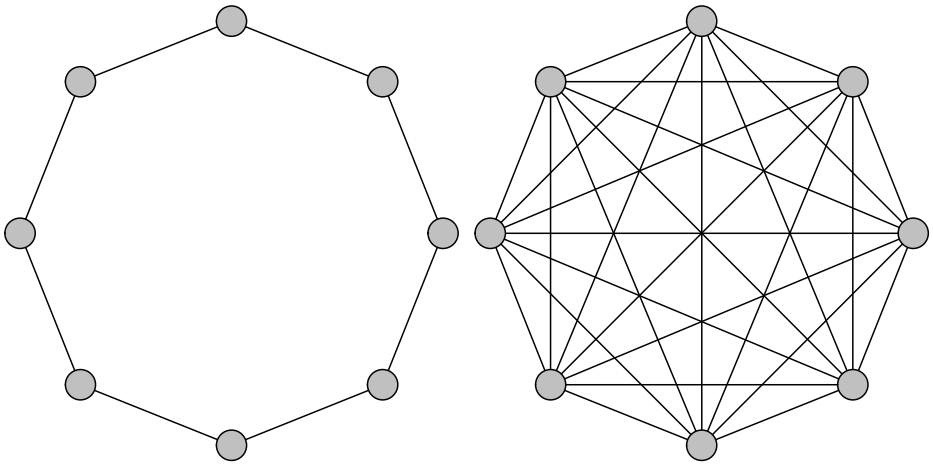
  onde ϵ representa a acurácia pretendida, $N_{(\epsilon)}$ o número de simulações que alcançaram a acurácia especificada e N o número total de simulações.

- A *eficiência* do enxame é 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.

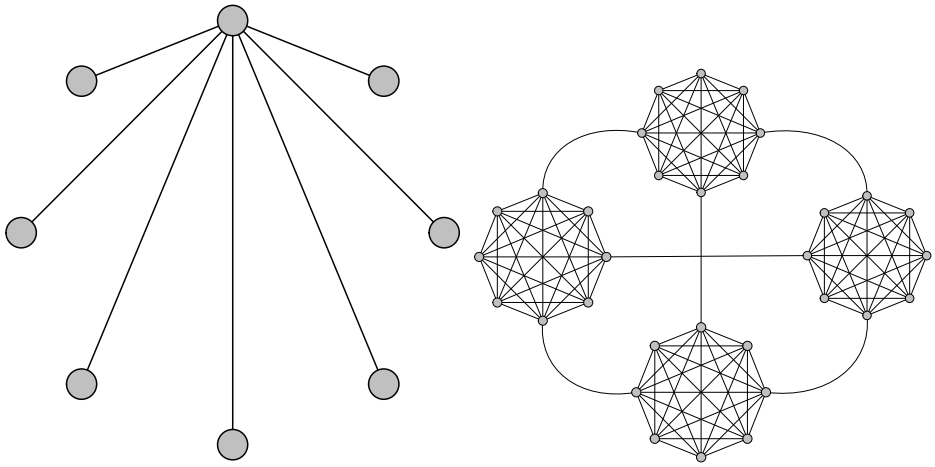
1.6 Topologias

A estrutura social do PSO é determinada pela forma com a qual as partículas trocam informação e exercem influência umas com as outras (ENGELBRECHT, 2006). Esta estrutura representa uma topologia que auxilia a compreensão da interação da partícula com o grupo. Desta forma, a topologia do PSO é definida pela maneira como é realizada a atualização da posição, ou seja, pelo cálculo da velocidade, pois é na velocidade que o conhecimento do melhor caminho é transmitido entre as partículas.

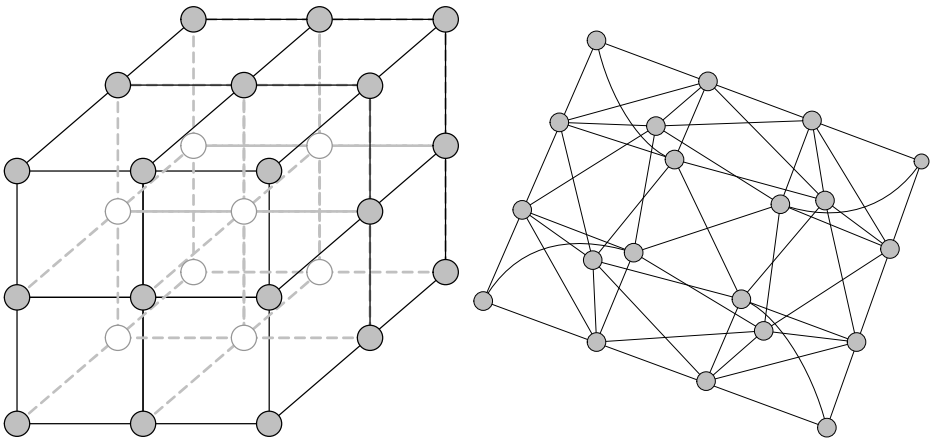
A Figura 1(a) exibe uma topologia em anel, pela qual cada partícula está conectada à sua vizinhança imediata. A informação flui mais lentamente pelo enxame fazendo com que uma área maior do espaço de busca seja coberta. A Figura 1(b) mostra a topologia em estrela, em que cada partícula está conectada a todas as partículas do enxame, ou seja, sua vizinhança é formada por todas as partículas existentes. Desta forma, cada partícula é atraída pela melhor partícula do enxame.



(a) *Anel* (b) *Estrela*



(c) *Círculo* (d) *Cluster*



(e) *Von Neumann* (f) *Pirâmide*

Figura 1: Topologias do PSO

A Figura 1(c) apresenta a topologia em círculo, onde a comunicação entre as partículas é realizada por uma partícula central. As outras partículas ficam isoladas. Esta topologia desacelera a troca de informação no enxame, permitindo que a convergência seja

realizada mais lentamente.

A Figura 1(d) ilustra a topologia em *cluster*. Nesta topologia, algumas partículas realizam a comunicação entre os *clusters* e cada partícula tem a sua vizinhança formada por todas as partículas de seu *cluster*. Na topologia *Von Neumann*, as partículas são conectadas em uma estrutura em grade, conforme Figura 1(e) (KENNEDY; MENDES, 2002). A Figura 1(f) exibe a topologia em pirâmide, a qual forma uma estrutura tridimensional.

1.7 PSO Paralelo

O algoritmo PSO na sua forma original é inspirado em comportamentos e aprendizados coletivos de indivíduos. Sendo assim, possuí uma estrutura computacional altamente paralelizável. O paralelismo pode ser obtido utilizando diferentes técnicas de particionamento, obtendo uma granularidade mais fina ou mais grossa. Pode-se ainda dividir o enxame principal em vários subenxames que cooperam entre si. Outras técnicas podem ser utilizadas como o modelo *Mestre-Escravo*, *Ilhas* e *Celular* (WAINTRAUB, 2009).

A vantagem mais imediata com o paralelismo é a redução do tempo de execução. Porém, esta redução nem sempre é alcançada pelo fato de o PSO ser um algoritmo estocástico. Desta forma, o paralelismo pode ocasionar perda da eficiência na otimização. Assim, o algoritmo paralelo precisaria de um maior número de iterações para obter um resultado que uma implementação sequencial obteria com menos iterações. Os cuidados para isso não acontecer estão relacionados à correta implementação do gerador de números aleatórios, o sincronismo entre tarefas e a comunicação de acordo com a topologia. Na Seção 1.7.1 serão levantadas as técnicas de paralelização de algoritmos e na Seção 1.7.2 serão levantados os trabalhos relacionados mais relevantes.

1.7.1 Algoritmos paralelos

O processamento paralelo é uma estratégia utilizada em computação para resolver mais rapidamente problemas computacionais complexos, dividindo-os em subtarefas que serão alocadas em vários processadores para serem executados simultaneamente. Esses processadores se comunicam para que haja sincronização ou troca de informações.

Segundo (FOSTER, 1995), a metodologia para projeto de algoritmos paralelos compreende quatro estágios distintos: particionamento, comunicação, aglomeração e mapeamento. No *particionamento* o programa é decomposto em subtarefas que serão executadas

em paralelo. A granularidade da decomposição pode ser fina ou grossa. Na granularidade *fina* o problema é decomposto em um grande número de subtarefas, onde cada uma executa um número limitado de instruções entre as trocas de informações. De outro modo, na granularidade *grossa* as tarefas executam um grande número de instruções entre as trocas de informações.

A *decomposição* pode ser funcional ou de domínio. A decomposição é dita *funcional* quando o problema é decomposto em diferentes tarefas, gerando diversos programas que serão executados em processadores diferentes. De outra forma, na decomposição de *domínio* os dados são decompostos em grupos, que serão distribuídos entre múltiplos processadores que executarão, simultaneamente, um mesmo programa.

As subtarefas geradas pelo particionamento normalmente necessitam trocar informações entre si. O fluxo de transferência dos dados entre as tarefas paralelas é especificado no estágio de *comunicação*. Esta comunicação pode ser *local*, entre um conjunto reduzido de subtarefas, ou *global*, entre todas as tarefas. A comunicação entre as tarefas podem ser síncronas ou assíncronas. Na comunicação *síncrona* as subtarefas se executam de forma coordenada, sendo que a transferência de dados é sincronizada e realizada no mesmo instante, enquanto na transferência *assíncrona*, as tarefas se executam de forma independente, não necessitando de sincronização para envio e recepção dos dados.

No estágio de *aglomeração* o algoritmo é revisado nas decisões tomadas durante as etapas de particionamento e comunicação, de modo a tornar seu desempenho mais eficiente. Neste estágio a granularidade do problema pode ser aumentada de forma a diminuir a comunicação entre as tarefas. Por último, o estágio de *mapeamento* consiste na especificação do recurso onde as tarefas serão executadas. Em geral, as tarefas aptas a serem executadas concorrentemente são alocadas em processadores diferentes e as tarefas que se comunicam frequentemente são alocadas em um mesmo processador.

1.7.2 Trabalhos relacionados

Nesta seção serão apresentados os trabalhos mais relevantes onde foram desenvolvidas implementações do algoritmo PSO, utilizando arquiteturas paralelas.

1.7.2.1 Implementação em FPGA

Uma arquitetura de *hardware/software co-design* para implementar o algoritmo PSO em uma FPGA é reportada em (LI et al., 2010). O módulo de atualização da velocidade e

posição da partícula foi implementado em *hardware*, enquanto o módulo da função objetivo foi implementado em *software*. Os módulos trabalham em conjunto de modo a acelerar o desempenho do processo de otimização. Os resultados experimentais exibem uma melhoria de até 10 vezes em comparação com o processador embutido *Nios II CPU*. Apesar da flexibilidade do projeto, obtida pela implementação da função objetivo em *software*, uma função objetivo de alto custo computacional irá implicar em menor desempenho.

Em (MUNOZ et al., 2009), o algoritmo PSO, juntamente com algumas funções de teste foram implementadas em uma FPGA, usando uma aritmética de ponto flutuante. A arquitetura explora o paralelismo atualizando a posição das partículas e a função objetivo de forma independe e em paralelo. O componente gerador de números aleatórios utiliza a técnica LFSR para prover os fatores estocásticos. A arquitetura foi validada utilizando 4 partículas na otimização de problemas de 2 dimensões. A implementação em FPGA foi comparada com uma implementação sequencial em MATLAB, obtendo um desempenho até 78 vezes melhor.

1.7.2.2 Implementação em OpenMP e MPI

Nos trabalhos de (SCHUTTE et al., 2004) e (KOH et al., 2006), os autores apresentam uma arquitetura paralela para o algoritmo PSO. O algoritmo foi implementado utilizando bibliotecas de passagem de mensagem em arquiteturas de multicomputadores. Soluções síncronas e assíncronas foram investigadas. As implementações obtiveram redução no tempo de execução para funções analíticas e um teste biomecânico. Além disso, a implementação assíncrona obteve redução do tempo de comunicação em comparação com a implementação síncrona. O algoritmo PSO foi paralelizado somente na operação de cálculo da função objetivo. As demais operações foram realizadas sequencialmente, consequentemente, em problemas com um número grande de partículas e dimensões, o desempenho será reduzido.

Em (WANG et al., 2008), os autores apresentam uma proposta de algoritmo paralelo baseado em subpopulações do PSO, implementado em 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. Cada subenxame troca a sua melhor posição com um subenxame vizinho após um certo número de gerações. 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 razoável melhoria no tempo de execução, quando comparado com a implementação sequencial.

1.7.2.3 Implementação em CUDA

Uma implementação paralela do algoritmo PSO utilizando CUDA é apresentada em (VERONESE; KROHLING, 2009). 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 foram realizados em uma GPU NVIDIA GTX 208 utilizando 100 partículas e um número fixo de iteração de 10.000 e 100.000, na otimização de algumas funções objetivo com 100 dimensões. São demonstrados resultados apenas de tempo de execução, sem avaliar a eficiência da otimização do algoritmo. Os resultados demonstram que foi obtida uma redução significativa de tempo quando comparado com implementações sequenciais em C e MATLAB.

Em (CADENAS-MONTES et al., 2011), os autores também apresentam uma implementação paralela do algoritmo PSO em GPU. A distribuição de tarefas é realizada de forma que o cálculo da função objetivo seja executado pela GPU e todo o restante do algoritmo é executado pela CPU. Ao executar o *kernel*, são geradas a quantidade de *threads* referente ao número de dimensões do problema. As *threads* são alocadas em *blocos* de acordo com o número máximo suportado pela capacidade computacional da GPU. Desta forma, o cálculo da função objetivo é realizado em paralelo para cada partícula e em cada dimensão do problema. Os resultados mostram que a implementação paralela reduziu o tempo de execução porém, aparentemente os testes demonstraram um limite no *speedup* obtido em razão do mapeamento adotado e da quantidade de núcleos da GPU.

Outra abordagem paralela do algoritmo PSO implementado em GPU é apresentado em (ZHOU; TAN, 2009). O paralelismo é aplicado somente no nível da partícula. Desta forma, é criada uma quantidade de *threads* igual ao número de partículas do enxame. As atualizações das velocidades e posições, a atualização de *Pbest* e *Gbest* e o cálculo da função objetivo são implementados em *kernels* que são executados em paralelo pela GPU. Os experimentos são obtidos comparando a implementação paralela em GPU com uma implementação serial em CPU utilizando um número fixo de iterações. Os resultados mostram que a implementação em GPU alcança um maior *speedup* quando usada com problemas de grandes dimensões utilizando maiores populações. Melhores taxas de *speedup* também são obtidas quando o custo computacional da função objetivo é maior.

No trabalho de (LEE et al., 2010), os autores realizam uma rigorosa análise de desempenho entre CPUs e GPUs, aplicando otimizações apropriadas em cada arquitetura. Um conjunto de teste é realizado utilizando uma CPU Intel Core i7-960 e uma GPU NVDIA GTX 280. Os resultados obtidos demonstram que o desempenho entre as arquiteturas é muito próximo, com a GPU obtendo um desempenho superior médio de 2,5 vezes. É abordado também, que em alguns trabalhos, onde é reportada uma grande diferença de desempenho, favorável à GPU, o código pode não ter sido corretamente otimizado, degradando o desempenho da CPU.

1.8 Considerações Finais do Capítulo

Geralmente, os trabalhos de paralelização do algoritmo PSO implementados em GPU comparam o ganho obtido em relação à CPU com o algoritmo implementado sequencialmente o que pode levar a uma comparação injusta, pois o *speedup* obtido é muito maior do que poderia realmente ser, caso a implementação em CPU fosse paralelizada.

Neste capítulo foi apresentada a otimização por enxame de partículas como um método computacional de inteligência coletiva. Os algoritmos *Gbest* e *Lbest* foram descritos. Medidas de desempenho foram mostradas como forma de quantificar a otimização do algoritmo. Uma introdução sobre o projeto de algoritmos paralelos foi mostrada. Resumidamente, trabalhos relevantes que trataram de implementações paralelas do PSO foram apresentados. No capítulo seguinte são propostas as estratégias de implementação paralela do algoritmo PSO.

Capítulo 2

ESTRATÉGIAS DE IMPLEMENTAÇÃO PARALELA DO PSO

ESTE capítulo apresenta três diferentes estratégias de implementação paralela do algoritmo PSO. Com o paralelismo do algoritmo é esperado uma melhora no desempenho sem afetar, no entanto, a acurácia e a eficiência do processo de otimização.

A organização deste capítulo é descrita a seguir: A Seção 2.1 introduz o algoritmo PPSO. Na Seção 2.2 é apresentado o algoritmo PDPSO. Na Seção 2.3, é descrito o algoritmo CPPSO, assim como a metodologia utilizada para particionamento do espaço de buscas em subespaços. Por fim, na Seção 2.4 são abordados as vantagens e as desvantagens das topologias em estrela e anel.

2.1 Algoritmo PPSO

O primeiro algoritmo proposto, chamado PPSO (*Parallel PSO*), decorre da ideia de que o trabalho realizado por uma partícula é independente daquele realizado pelas outras partículas do enxame. As Figuras 2 e 3 exibem os fluxogramas do algoritmo PPSO na topologia em estrela e anel respectivamente, onde $p_1, \ldots, p_n$ denotam as n partículas do enxame, e $v^{(p_1)}, \ldots, v^{(p_n)}$ e $x^{(p_1)}, \ldots, x^{(p_n)}$ as respectivas velocidades e posições.

Desta forma, após a inicialização, cada partícula calcula o valor da função objetivo, atualiza a velocidade e posição, e elege *Pbest* 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. Na topologia em estrela (Figura 2), a comunicação é realizada entre todas as partículas do enxame, trocando informação de *Pbest* até

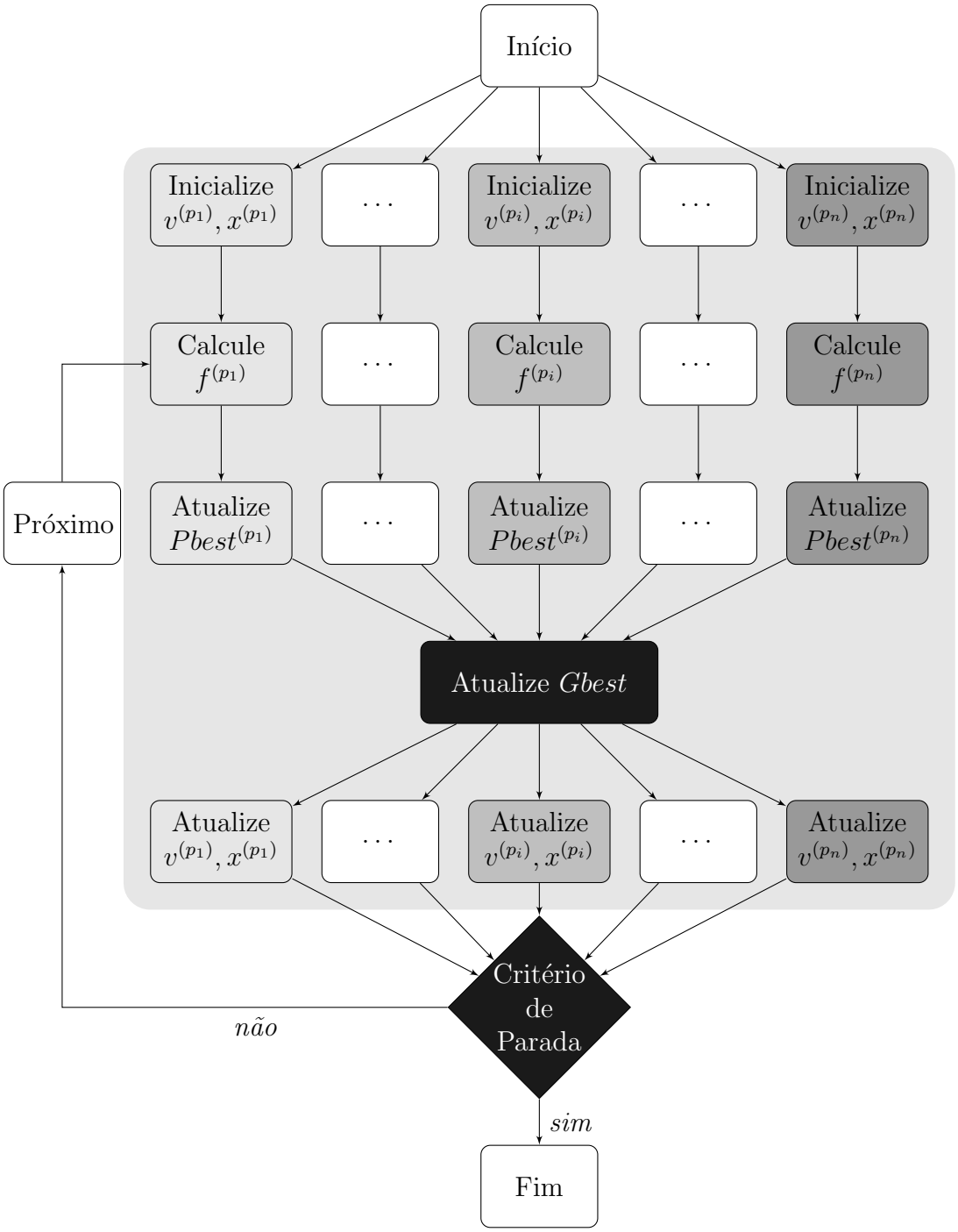


Figura 2: Computação paralela realizada por um exame com n para a topologia em estrela

a eleição de *Gbest*. Com o objetivo de sincronizar o processo e prevenir o uso de valores desatualizados de *Gbest*, a computação da nova velocidade e posição inicia somente após *Gbest* ter sido escolhida entre *Pbest* das partículas do enxame. A verificação do critério de parada pode ser realizada sem sincronizar os processos paralelos. De outro modo, na topologia em anel (Figura 3), a partícula realiza a comunicação com seus vizinhos para obter a melhor posição local, *Lbest*. Os vizinhos da partícula i são as partículas $(i+1) mod$

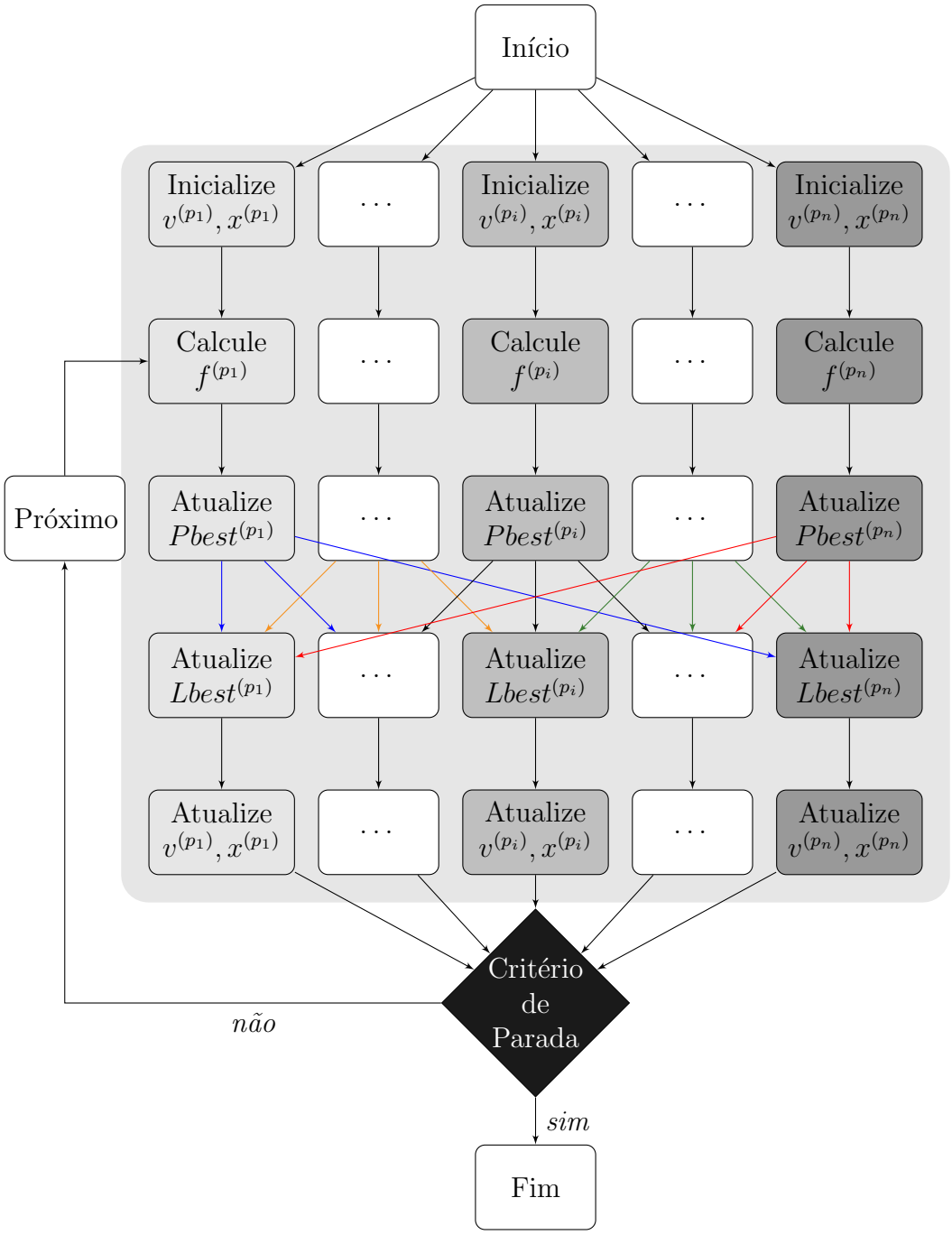


Figura 3: Computação paralela realizada por um exame com n partículas para a topologia em anel

n e $(i-1) mod\ n$. Assim, a melhor posição local é obtida baseada nesta definição de vizinhança. É fácil observar que para escolher *Lbest* é realizada menos comunicação entre as partículas do que na topologia em estrela. Contudo, para a verificação do critério de parada, é necessária uma comunicação entre todas as partículas para se obter o melhor resultado do enxame. Note que, o PPSO utiliza um único enxame de n partículas em paralelo durante todo o processo de otimização.

2.2 Algoritmo PDPSO

A segunda abordagem considera o fato de que as operações internas aos processos que implementam as partículas também podem ser paralelizadas, proporcionando uma decomposição do problema em uma granularidade mais fina. Assim, as tarefas das partículas são decompostas até o nível das dimensões.

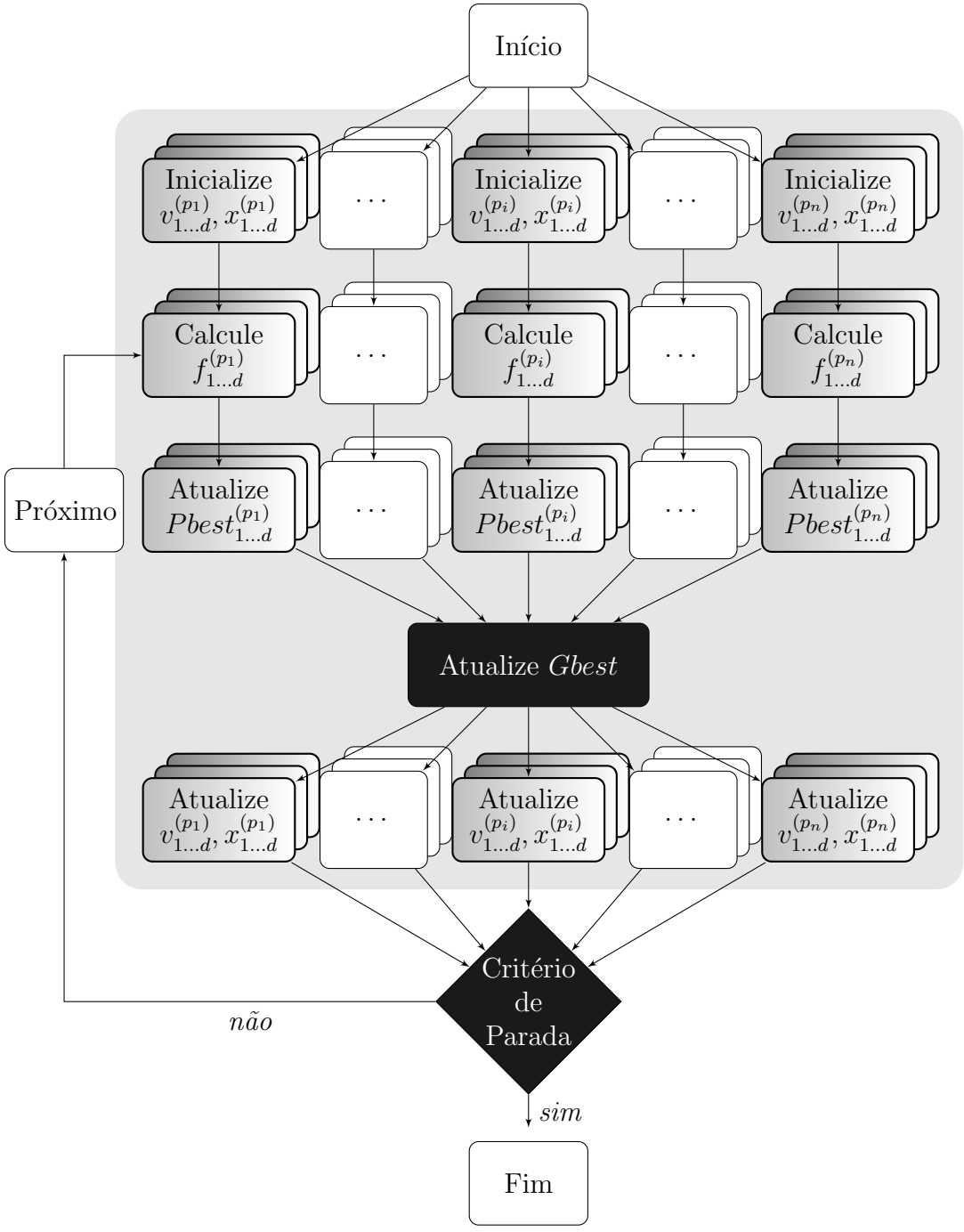


Figura 4: Computação paralela realizada por um exame com n partículas com respeito a d dimensões na topologia em estrela

O algoritmo é chamado PDPSO (*Parallel Dimension PSO*). A Figura 4 ilustra o fluxo de dados para a topologia em estrela, enquanto a Figura 5 exibe o fluxo de dados para a topologia em anel, onde $p_1, \ldots, p_n$ denota as n partículas do enxame, e d cada dimensão do problema executada em paralelo. Problemas de otimização baseados em uma função objetivo com um grande número de dimensões poderão tirar uma maior vantagem desta abordagem.

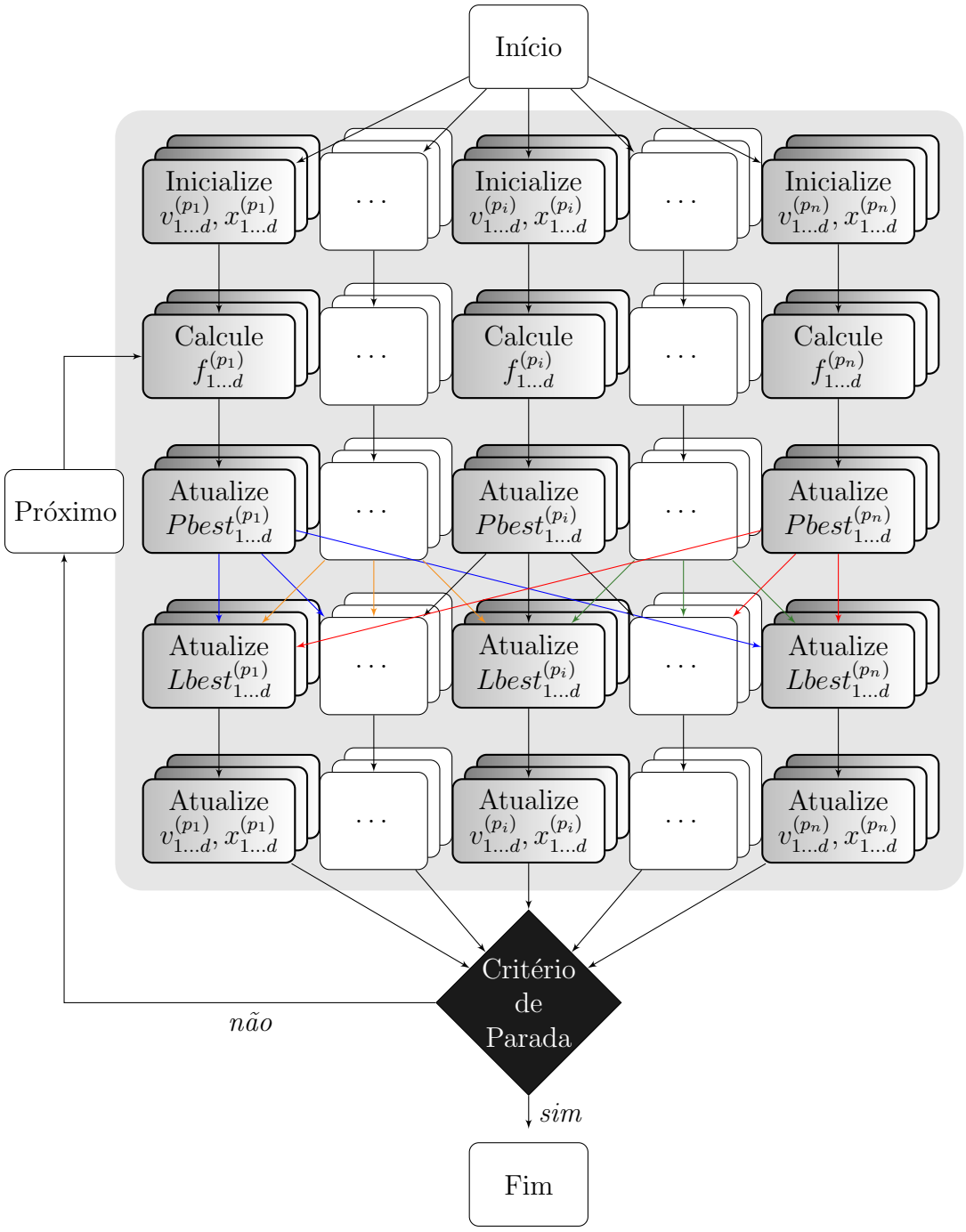


Figura 5: Computação paralela realizada por um exame com n partículas com respeito a d dimensões na topologia em anel

Por exemplo, se o número de dimensões do problema é 128, o algoritmo PPSO precisa de 128 iterações para calcular o valor de $fitness$. No PDPSO é utilizada uma única iteração para obter o valor de $fitness$ para cada dimensão e mais 7 iterações para computar os resultados intermediários com o objetivo de obter um único valor correspondente ao $fitness$ da partícula. Este processo é chamado de *redução de fitness*. Se a função objetivo é $\sum_{i=1}^{n} x_i^2$ então cada fluxo do processo retorna o valor do quadrado. A redução de $fitness$, neste caso, consiste em obter a soma de todos os quadrados.

Nas Figuras 4 e 5, as etapas de otimização, realizadas por cada partícula, são apresentadas verticalmente enquanto os blocos em profundidade representam as operações paralelas referentes a cada dimensão. Diferentemente do PPSO, o PDPSO realiza a inicialização da posição de uma partícula, atualiza a sua velocidade e a posição, assim como copia os novos valores de *Pbest* e *Gbest*/*Lbest* diretamente, mapeando cada processo em uma dimensão do problema. Assim, não é necessário um laço para executar as instruções de cada dimensão. Deste modo, é possível distribuir a carga computacional até a menor granularidade do problema. Como no caso do algoritmo PPSO, o PDPSO também utiliza um único enxame de n partículas, mas aplica o paralelismo no nível das operações independentes referente a cada dimensão, durante todo o processo de otimização.

2.3 Algoritmo CPPSO

O CPPSO (*Cooperative Parallel PSO*) é baseado no algoritmo CPSO-S_k (*Cooperative PSO Split*), desenvolvido por Van den Bergh em (BERGH et al., 2002). A ideia principal na base desta abordagem, consiste em subdividir o vetor de dimensões d, do problema original, entre k subproblemas de d/k dimensões. Cada subproblema é otimizado por um subenxame que será responsável por otimizar suas d/k dimensões, utilizando o algoritmo PSO. O número de partes k é denominado *fator de divisão*. Uma complicação desta configuração é que para calcular o valor da função objetivo é necessário um vetor com d dimensões de entrada. Se cada subenxame representa apenas uma parte do número total de dimensões não é possível realizar o cálculo da função considerando somente um subenxame isolado. Para isso, um vetor de contexto é utilizado para prover um vetor com d dimensões relativo ao problema original.

Para construir este vetor os subenxames cooperam entre si, armazenando as posições das melhores partículas obtidas por cada subenxame k, na posição k do vetor de

contexto. Para realizar o calculo de $fitness$ para todas as partículas do subenxame S_k, as outras $d-1$ entradas do vetor de contexto são mantidas constantes enquanto a entrada do vetor de contexto referente ao subenxame S_k é substituída pela coordenada de cada partícula do subenxame S_k.

As Figuras 6 e 7 ilustram o fluxo de dados usado para implementar o CPPSO utilizando as topologias em estrela e anel respectivamente, onde $S_1, \ldots, S_k$ denotam os k subenxames compostos de l partículas cada. As partes em negrito representam às etapas em que o algoritmo utiliza todas as d dimensões para realizar o cálculo de $fitness$ e gerar o vetor de contexto, mapeando os k subenxames paralelos em um único enxame com d dimensões. O enxame inicial tem seu número total de partículas n e dimensões divididos entre k subenxames. Os subenxames realizam sua otimização do algoritmo PSO de forma independente e em paralelo. Cada subenxame inicializa suas partículas $l = n/k$ e envia a posição da primeira partícula para compor o vetor de contexto. Nas Figuras 6 e 7, as partículas estão representadas pelos blocos em profundidade, realizando suas operações em paralelo.

Para a realização do cálculo de $fitness$ é utilizado o vetor de contexto com d dimensões que mapeia todo o problema original como se fosse um único enxame. Nesta etapa cada partícula substitui o valor de sua coordenada na posição do vetor de contexto referente ao seu subenxame a fim de obter o valor correspondente de $fitness$. Assumindo que o CPPSO será executado com 2 subenxames para minimizar $f(x_1, x_2)$ em x_1 e x_2 e considerando que as melhores partículas dos subenxames obtiveram as posições $P_1 = 4.5$ e $P_2 = 1.2$, o vetor de contexto gerado será $[4.5, 1.2]$. O cálculo de $fitness$ para S_1 será realizado substituindo os valores da posição de suas partículas no vetor de contexto ($x_{1\ldots l}$, 1.2) enquanto para S_2, o $fitness$ será calculado para cada partícula utilizando o vetor de contexto (4.5, $x_{1\ldots l}$). Diferentemente da geração do vetor de contexto, que é realizada de maneira síncrona, o cálculo de $fitness$ é realizado de forma independente e em paralelo.

Após o cálculo de $fitness$, cada subenxame realiza a atualização de *Pbest* e *Gbest*/*Lbest* em suas partículas e seleciona a melhor partícula do subenxame. Os enxames cooperam entre si na geração de um vetor de contexto substituindo a posição da melhor partícula encontrada por cada subenxame. Assim, os processos são sincronizados nesta etapa até a obtenção do novo vetor de contexto. O valor da melhor posição obtida entre os subenxames é compartilhado a fim de ser utilizado na verificação do critério

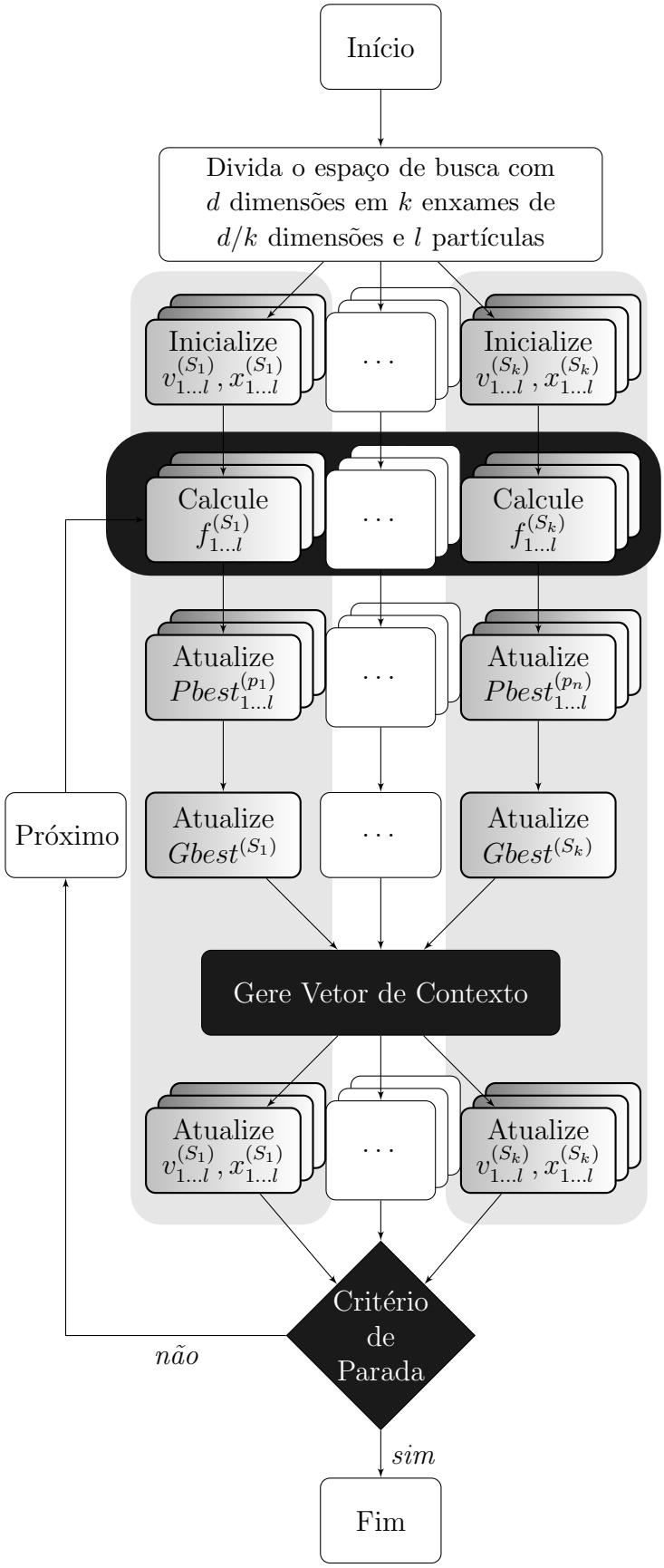


Figura 6: Computação paralela realizada por k subexames com n partículas na topologia em estrela

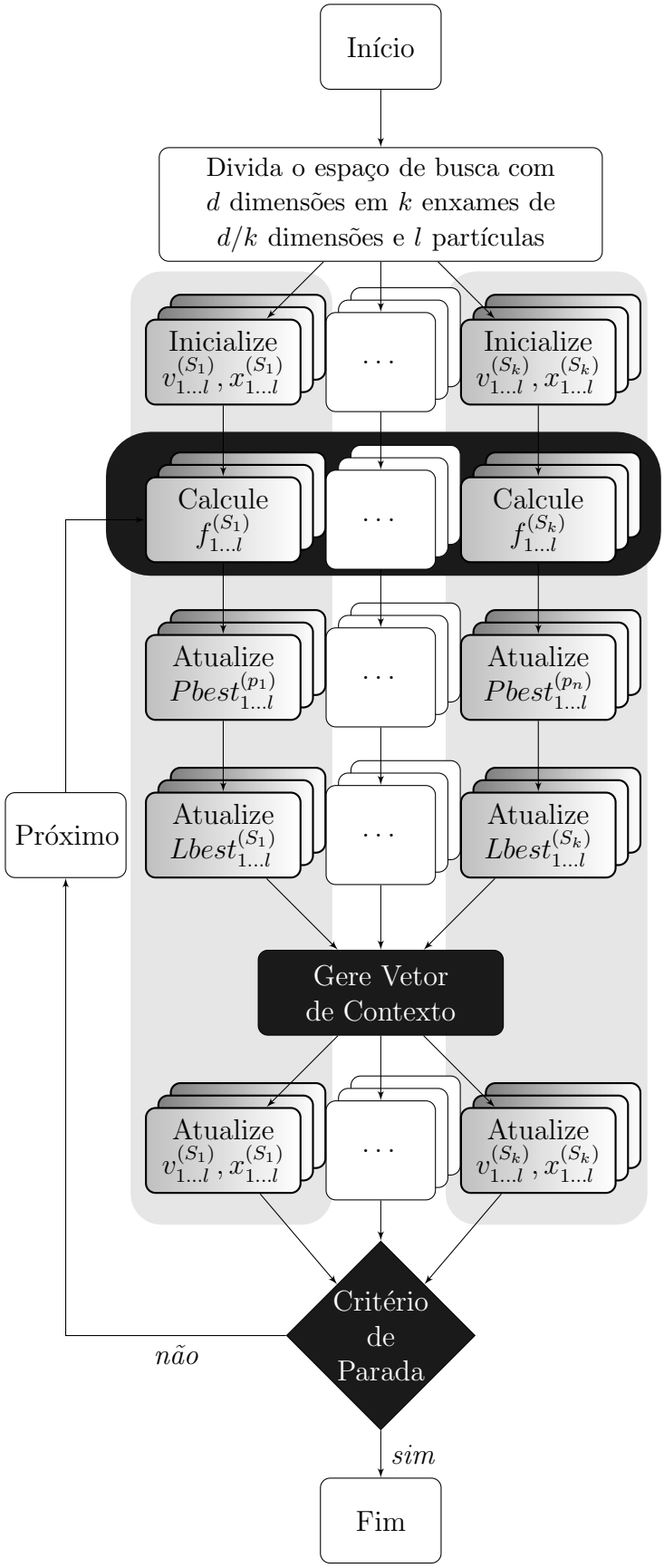


Figura 7: Computação paralela realizada por k subexames com n partículas na topologia em anel

de parada. Após esta etapa é realizado a atualização da nova velocidade e posição, e a verificação do critério de parada, para decidir quanto ao termino da otimização ou a realização de mais uma iteração. Note que, diferentemente dos algoritmos anteriores, a otimização é realizada por k subenxames que cooperam entre si na geração de um vetor de contexto, porém, sem realizar troca de informações entre os subenxames referentes a *Pbest* e *Gbest*/*Lbest*.

2.4 Vantagens e Desvantagens das Topologias

Na topologia em estrela, cada partícula se comunica com qualquer outra partícula do enxame. Assim, a partícula é atraída pela melhor solução encontrada por todo o enxame. O PSO utilizando a topologia em estrela apresenta uma convergência mais rápida, porém é mais suscetível a ficar preso em um mínimo local. Na topologia em anel, o fluxo de informações flui mais lentamente através da componente social. Apenas a vizinhança imediata da partícula compartilha a sua melhor solução, tendo como consequência uma taxa de convergência menor. Porém, uma maior parte do espaço de busca é explorada quando comparada com a topologia em estrela. Este ambiente permite que a topologia em anel provenha melhor desempenho em termos de qualidade da solução encontrada.

A Figura 8, demonstra a convergência comparando enxames com as mesmas características e parâmetros, utilizando as topologias em estrela e anel. Podemos observar que, como relatado anteriormente, quando a topologia em estrela é usada, a convergência do enxame ocorre de maneira mais precoce, porém, a qualidade do resultado obtido é inferior quando comparado a implementação utilizando a topologia em anel. De outro modo, na topologia em anel, uma vez que a população converge mais lentamente, a diversidade do enxame é mantida e maiores partes do espaço de busca são explorados. Segundo (ENGELBRECHT, 2006), algoritmos utilizando *Lbest* possuem melhor desempenho computacional devido a estrutura de vizinhança.

Assim, no restante desta dissertação, será sempre adotada a topologia em anel para as implementações dos algoritmos propostos em *software*, na minimização de funções de até 256 dimensões. A implementação em *hardware*, que devido a limitação do número de partículas, minimiza problemas de 2 dimensões, utiliza a topologia em estrela.

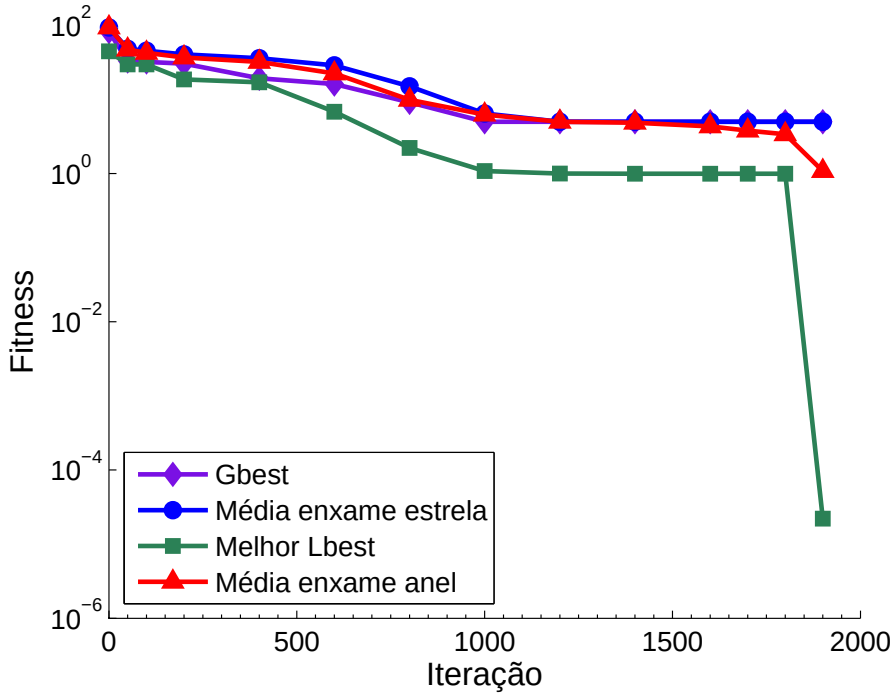


Figura 8: Convergência do PSO comparando as topologias em estrela e anel para um problema multimodal.

2.5 Considerações Finais do Capítulo

Neste capítulo foram apresentados os algoritmos propostos PPSO, PDPSO e CPPSO. Foram exibidos os fluxogramas das estratégias de paralelismo utilizadas e realizada uma breve comparação entre as topologias em estrela e anel. O capítulo seguinte apresenta a implementação do algoritmo PPSO em *hardware* utilizando uma FPGA.

Capítulo 3

IMPLEMENTAÇÃO EM HARDWARE

ESTE capítulo apresenta uma implementação em *hardware* do algoritmo PSO utilizando aritmética de ponto flutuante. O *hardware* utilizado é uma FPGA (*Field Programmable Gate Arrays*). As FPGAs apresentam diversas vantagens em relação a outras alternativas baseadas em *hardware*, como por exemplo, diminuição do *time-to-market*, utilização flexível e custo de desenvolvimento menor. A programação destes dispositivos é feita através de linguagens de descrição de *hardware* (*Hardware Description Language* - HDL).

A arquitetura será implementada como um coprocessador do processador MicroBlaze™ de forma a resolver aplicações específicas, otimizar o desempenho e liberar o processador durante a execução da otimização. A organização deste capítulo é descrita a seguir: A Seção 3.1 realiza uma breve introdução a FPGA. A Seção 3.2 apresenta o processador MicroBlaze™. Na Seção 3.3 é apresentado a descrição da arquitetura do coprocessador PSO, seu controlador e suas unidades funcionais.

3.1 FPGA

As FPGAs representam uma classe de circuitos integrados projetados para serem configurados pelo usuário após a sua fabricação (KILTS, 2007). Ela é constituída por uma matriz de blocos lógicos configuráveis (*Configurable Logic Blocks* - CLBs), conectados por canais também programáveis, que podem ser configurados para dar origem a qualquer tipo de sistema digital. Um sistema é construído a partir da configuração dos CLBs disponíveis, os quais são formados por células lógicas, agrupados em parcelas denominadas *Slices*. Dependendo do tipo da FPGA, o CLB pode ter 2 ou 4 *Slices*. Normalmente, os blocos

lógicos são compostos de uma tabela de pesquisa (*Look-up Table* - LuT) de 4 a 6 entradas, circuitos de seleção (multiplexadores) e *flip-flops*.

Após a especificação do sistema utilizando uma linguagem de descrição de *hardware*, uma ferramenta de sintetize se encarrega de compilar a especificação do *hardware* em funções lógicas que possam ser mapeadas nos blocos lógicos da FPGA. Este processo pode ser resumido na etapa de mapeamento, alocação e roteamento. Durante o mapeamento, as funções lógicas são associadas aos CLBs. Na etapa de alocação, os CLBs são implementados na FPGA. Durante o roteamento, são realizadas as conexões entre os CLBs. Dependendo da quantidade de recursos disponíveis na FPGA, é possível replicar diversos componentes do sistema e realizar operações em paralelo. Devido também a esse caráter massivamente paralelo, elas permitem alcançar um considerável desempenho de *throughput* com baixas taxas de clock, na casa dos MHz, e baixo consumo de energia.

3.2 O MicroBlaze

O MicroBlaze™ (XILINX, 2008b) é um microprocessador de propriedade intelectual da Xilinx™ que possui um conjunto de instruções reduzidas (RISC), suporte a operações de ponto flutuante e um *pipeline* de cinco estágios. Este processador pode ser sintetizado em dispositivos programáveis e é otimizado para ser implementado em FPGAs do próprio fabricante. O MicroBlaze™ é dotado de uma interface de comunicação ponto-a-ponto chamada *Fast Simplex Link* (FSL) (XILINX, 2008a), que permite a conexão de um componente externo, denominado de coprocessador, com o processador.

Estes canais de comunicação dedicados são ideais para estender a capacidade computacional do processador. O conjunto de instruções do MicroBlaze™ inclui instruções para ler dados das portas da FSL e escrever o resultado em algum registrador do processador, bem como instruções para enviar dados de um registrador do processador para a FSL. O canal FSL provê um meio de comunicação unidirecional e dedicado entre dois componentes quaisquer. Este canal é baseado em filas (FIFO) e pode conter entre 1 a 8K palavras de 32 bits. O componente atua como mestre quando escreve dados na fila, enquanto o outro atua como escravo retirando dados da fila.

3.3 Implementação do Coprocessador HPSO

O componente coprocessador, que será denominado HPSO, implementa o algoritmo PSO em FPGA (CALAZAN; NEDJAH; MOURELLE, 2012b) (CALAZAN; NEDJAH; MOURELLE, 2013). Foi todo descrito utilizando VHDL (XILINX, 2009) (*Very High Speed Integrated Circuits Hardware Description Language*). Como plataforma alvo foi utilizada a placa de desenvolvimento ML506 (XILINX, 2011), que possui uma FPGA do tipo Virtex-6 (XILINX, 2010) modelo xcvlx75t. Dois canais FSL realizam a comunicação entre o MicroBlaze™ e o coprocessador HPSO. Outros componentes podem ser conectados ao microprocessador por meio de outros barramento. Deste modo, o resultado da computação pode ser transmitido para uma estação de trabalho por meio de uma interface UART (*Universal Asynchronous Receiver/Transmitter*) e interpretado por um *software*. A Figura 9 exibe o processador conectado ao coprocessador HPSO. A arquitetura explora o paralelismo atualizando as posições das partículas e obtendo o resultado de $fitness$ de forma independente. Portanto, o HPSO implementa o algoritmo PPSO descrito no Capítulo 2.

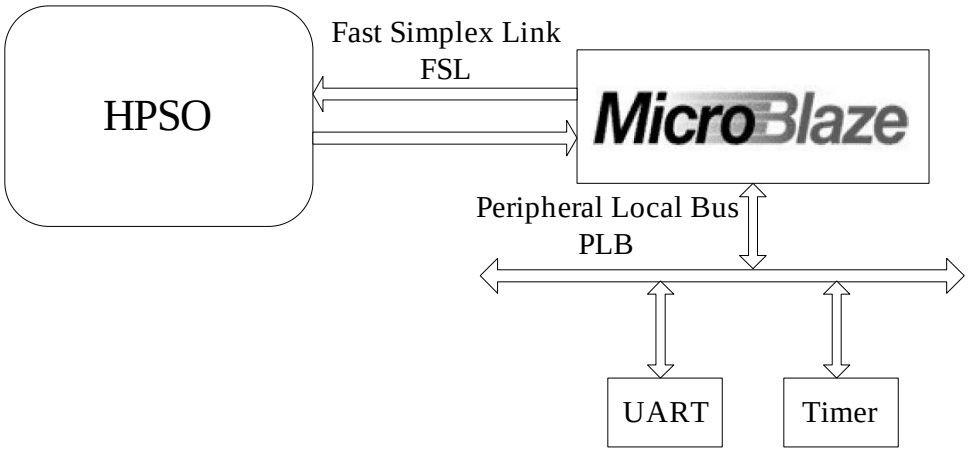


Figura 9: Coprocessador HPSO conectado via FSL ao processador MicroBlaze

3.3.1 Arquitetura do coprocessador HPSO

O principal componente da arquitetura do coprocessador é retratado na Figura 10, e denominado de unidade `SWARM`. Ela é responsável pela correta operação e sincronização do enxame. A unidade inicia habilitando a unidade `START` cuja função é gerar as posições e velocidades iniciais de cada partícula da população (CALAZAN; NEDJAH; MOURELLE, 2011).

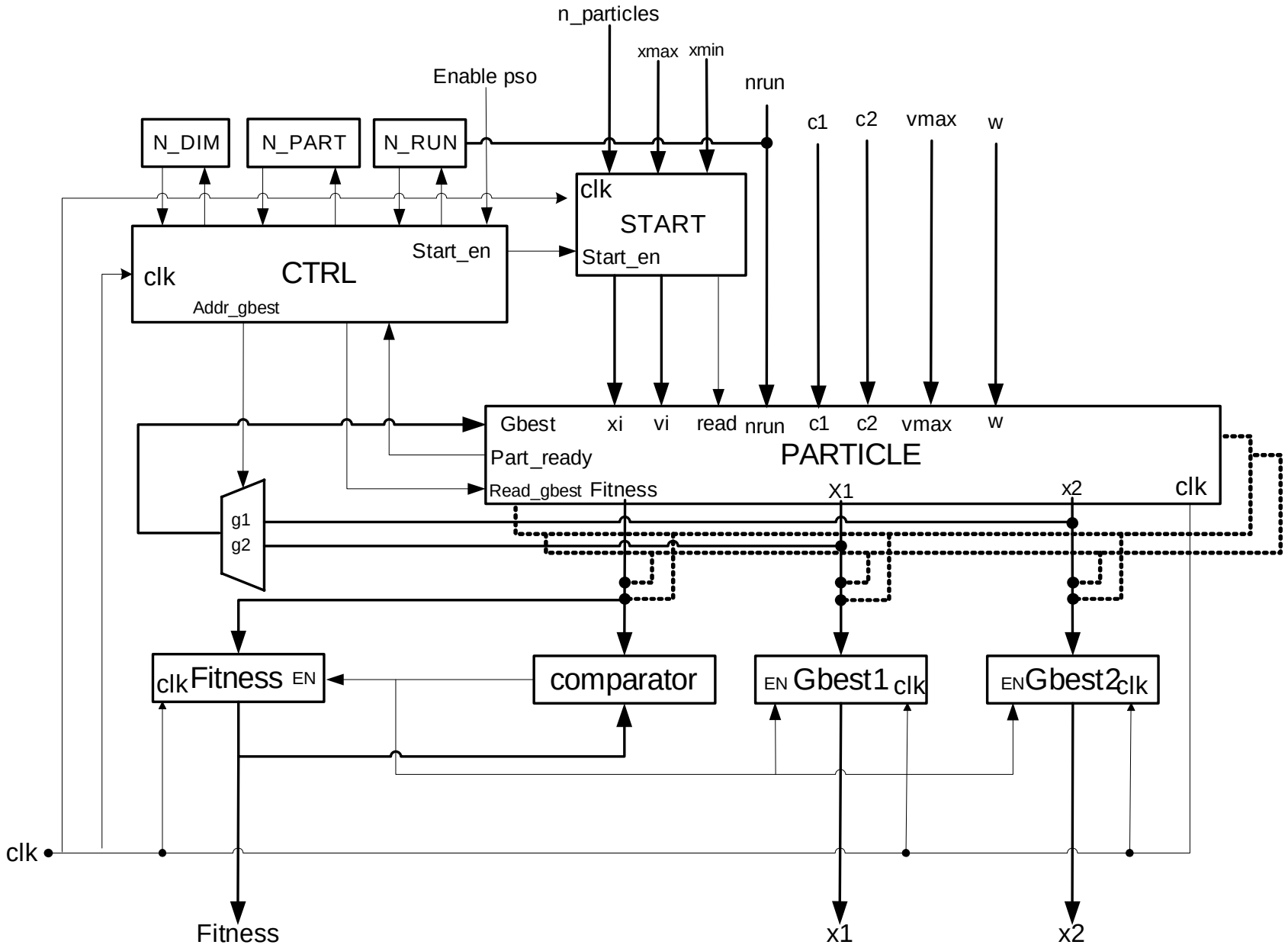


Figura 10: Arquitetura da unidade SWARM

A natureza estocástica do algoritmo PSO requer o uso de geradores de números pseudoaleatórios. Um registrador do tipo LFSR (*Linear Feedback Shift Register*) é a unidade responsável por gerar esses números aleatórios de precisão simples. Isto é feito de acordo com o valor máximo e mínimo do domínio bem como o número de partículas do enxame. Assim que esses valores são carregados, a unidade `SWARM` habilita a partícula para iniciar o cálculo de $fitness$. Sempre que a partícula estiver pronta para informar o respectivo valor de *Pbest*, o comparador verifica, entre os valores retornados pelas partículas, se o registrador *Gbest* deve ser atualizado. A unidade `SWARM` sincroniza o trabalho das partículas, permitindo que os cálculos de velocidade e posição sejam iniciados somente após a correta eleição de *Gbest*. A máquina de estados `CTRL`, entre outros controles, habilita os registradores `NDIM`, `NRUN` e `NPART` para manter o controle da dimensão, número de iterações e número de partículas respectivamente. Note que a unidade `PARTICLE` incluiu muitas partículas pelo uso do parâmetro $n\_particles$, permitindo que todas elas sejam executadas em paralelo.

3.3.2 Unidade PARTICLE

A unidade `PARTICLE`, exibida na Figura 11, é equipada com uma memória interna representada por um banco de registradores, dois para cada dimensão, um para armazenar a velocidade e outro para a posição, um módulo de execução `PSO CORE` para cálculo das novas velocidade e posição da partícula, um módulo para cálculo de $fitness$ e uma unidade de ponto flutuante de precisão simples `FPU` (padrão IEEE 754) para realizar os cálculos. Note que existirão tantas unidades `PARTICLE` quanto às partículas do enxame, permitindo assim, a execução massivamente paralela das operações relacionadas a cada partícula.

Depois de carregar os valores iniciais, a unidade `FITNESS` começa o cálculo do valor de $fitness$. O resultado desta operação é então comparado com o valor inicial e atualizado sempre que necessário. Neste ponto, o valor de *Pbest* da partícula é passado para a unidade `SWARM` que, depois de comparar os resultados de todas as partículas, elege adequadamente *Gbest*, distribui o novo valor para as unidades `PARTICLE`, desencadeando o início do respectivo modulo `PSO CORE`. Este último realiza a computação da nova velocidade e posição da partícula no espaço de busca. Isto é realizado para cada dimensão e de acordo com as equações (2) e (4) do Capítulo 1. Neste estágio, a partícula está pronta para realizar uma nova iteração ou terminar o processo de otimização de acordo com o

critério de parada adotado.

A unidade FPU, desenvolvida por (AL-ERYANI, 2006), é uma unidade de ponto flutuante que permite o processamento de dados de 32 bits (precisão simples) para realizar operações aritméticas com números de ponto flutuante padrão IEEE 754. Cada partícula possui uma unidade FPU com o objetivo de torná-la independente na realização dos cálculos necessários e garantir o paralelismo das operações.

A unidade RAND permite a geração de números pseudoaleatórios requeridos pela natureza estocástica do algoritmo PSO. Esta unidade é responsável pela geração de números aleatórios de precisão simples baseado no princípio LFSR (*Linear Feedback Shift Register*). O LFSR é um registrador de deslocamento cujo bit de entrada é uma função linear de seu estado anterior. Portanto, é um registrador de deslocamento cujo bit de entrada é impulsionado pela porta XOR de alguns bits do registrador para a mudança de valor conforme Figura 12 (MUNOZ et al., 2009).

As posições dos bits que afetam o próximo estado são chamadas de *Taps*. O ciclo do LFSR de n bits é $2^n - 1$. Realizando a operação sobre os 8 bits mais significativos da mantissa e nos 4 bits menos significativos do expoente garantimos que os números gerados permaneçam no intervalo [0,1], necessários para substituir os aleatórios r_1 e r_2 usados no algoritmo PSO (vide Capítulo 1). Uma unidade FITNESS é incluída em cada unidade PARTICLE, como mostrado na Figura 11. Isto permite que todos os cálculos sejam realizados em paralelo. A unidade FITNESS é implementada na forma de uma máquina de estados. O número de estados dessa unidade depende, então, da função objetivo que está sendo otimizada e os cálculos das operações básicas do tipo $+$, $-$ e $\times$ são realizados pela unidade FPU.

3.3.3 Controlador da unidade SWARM

A Figura 13 mostra o diagrama de transição de estados da maquina de estados finito que conduz o controlador principal da unidade SWARM, denominado CRTL na Figura 10. A descrição dos 15 estados é apresentado na sequência.

- S_{00}: Inicialize o sistema; Se *EnablePSO* estiver ativo então vá para S_{01};
- S_{01}: Habilite *startEn* da unidade START para a partícula $NPart$; Vá para S_{02};

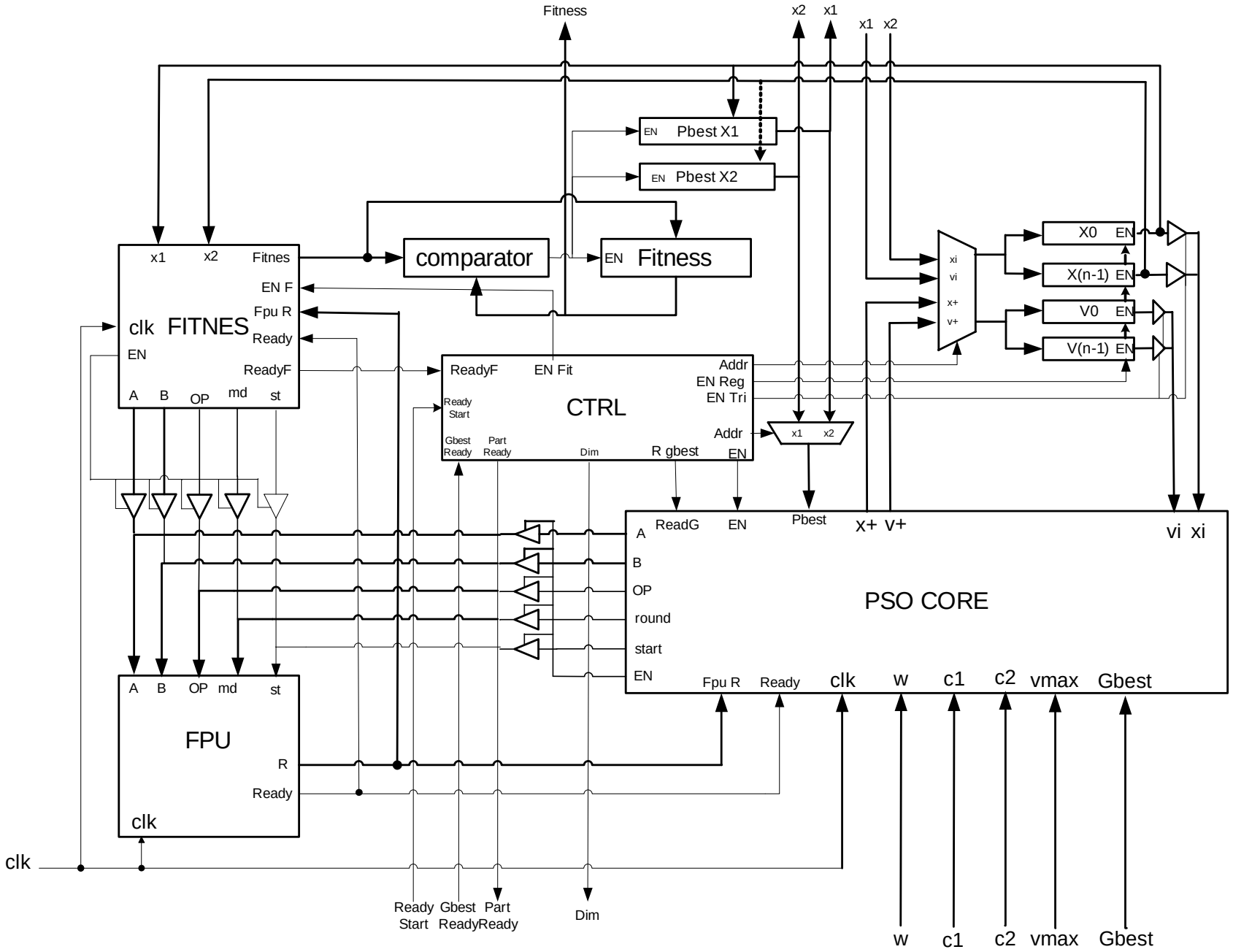


Figura 11: Arquitetura da unidade PARTICLE

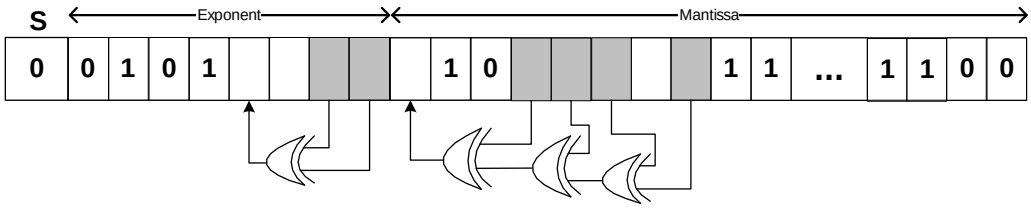


Figura 12: LFSR

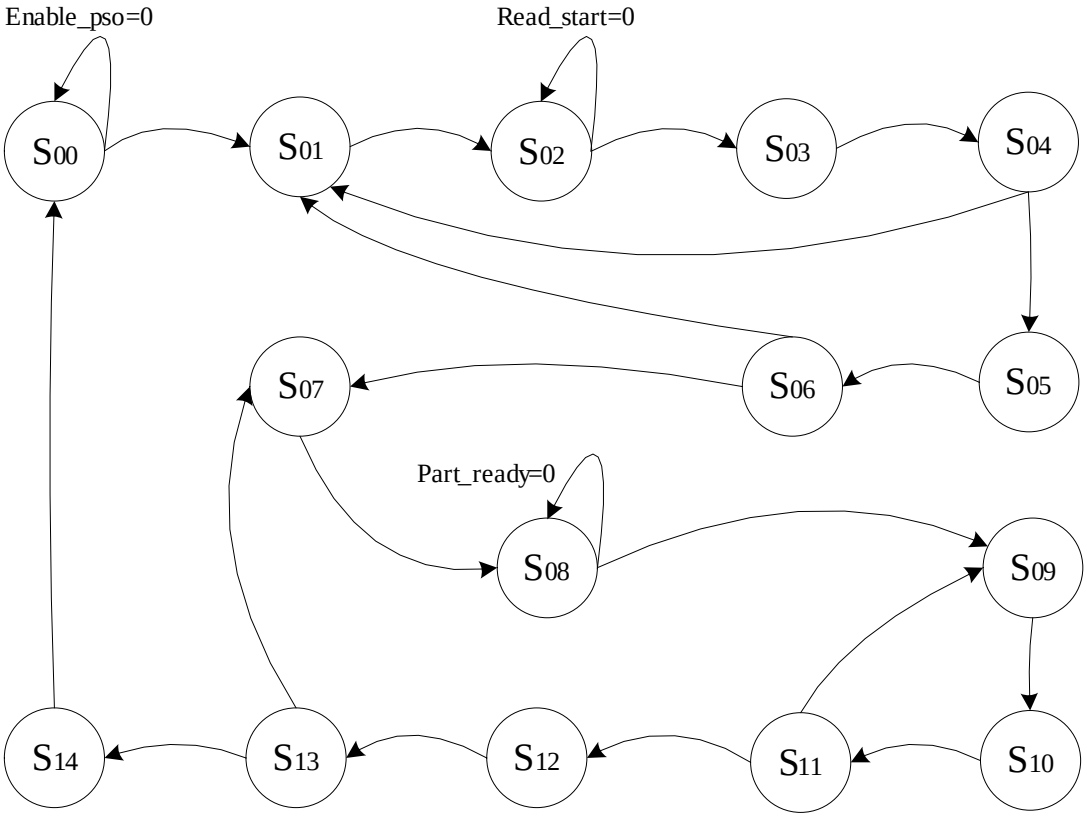


Figura 13: FSM da unidade SWARM

- S_{02}: Se *readStart* estiver ativo então habilite a unidade `PARTICLE` de número $NPart$; Inicialize a posição e a velocidade para $NDim$; Vá para S_{03};
- S_{03}: Incremente $NDim$; Vá para S_{04};
- S_{04}: Se $NDim$ atingiu o limite então vá para S_{05} Senão vá para S_{01};
- S_{05}: Incremente $NPart$; Habilite a unidade `FITNESS`; Vá para S_{06};
- S_{06}: Se $NPart$ atingiu o limite então vá para S_{07} Senão, vá para S_{01};
- S_{07}: Reinicie $NPart$; Vá para S_{08};
- S_{08}: Se *readyPart* da partícula estiver ativo então vá para S_{09};

- S_{09}: Se o valor de $fitness$ da partícula é menor que o $fitness$ global então
  Registre o novo valor de $fitness$ e sua posição; Vá para S_{10};
- S_{10}: Incremente o contador das partículas; Vá para S_{11};
- S_{11}: Se $NPart$ tiver atingido o limite então vá para S_{12} Senão vá para S_{09};
- S_{12}: Habilite *readGbest*; Atualize os registradores de velocidade e posição das partículas;
  Incremente $NRun$; Vá para S_{13};
- S_{13}: Se $NRun$ tiver atingido o limite então vá para S_{14} Senão vá para S_{07};
- S_{14}: Mostre o resultado final; Vá para S_{00}.

Note que o *hardware* é planejado para duas dimensões apenas. Entretanto, ele pode ser facilmente reconfigurado para qualquer número de dimensões, dada é claro, a quantidade necessária de recursos de *hardware*.

3.4 Considerações Finais do Capítulo

Neste capítulo foi apresentada uma arquitetura paralela do algoritmo PSO implementado como coprocessador do processador MicroBlaze™. O *hardware* utilizado foi uma $Xilinx$ Virtex 6 FPGA modelo xcvlx75t. A arquitetura explora o paralelismo atualizando as posições das partículas e obtendo o resultado de $fitness$ de forma independente. No Capítulo 6 serão apresentados os resultados do coprocessador HPSO em comparação com o processador MicroBlaze™. O capítulo seguinte apresenta a implementação dos algoritmos propostos em OpenMP e MPICH.

Capítulo 4

IMPLEMENTAÇÃO EM OPENMP E MPICH

O OPENMP (*Open Multi-Processing*) (CHAPMAN; JOST; PAS, 2008) é uma API multiplataforma para processamento paralelo baseado em arquiteturas com compartilhamento de memória para as linguagens C/C++ e Fortran. O OpenMP foi especificado por um conjunto de fabricantes de *hardware/software* visando à portabilidade, a escalabilidade e a facilidade de implementação. O MPI (*Message Passing Interface*) (GROPP; LUSK; SKJELLUM, 1999) é uma especificação para um padrão de biblioteca de troca de mensagens que foi definido no Fórum MPI (MPI_FORUM, 2012). O MPI foi concebido como um modelo de comunicação em arquiteturas de multicomputadores conectados por uma rede. O MPICH (MPI *Chameleon*) (ANL, 2012) é uma implementação do MPI desenvolvida pela *Argonne National Laboratory* (GROPP; LUSK, 1992).

Em uma implementação em OpenMP, a comunicação é realizada via a memória compartilhada, enquanto que na implementação em MPI a comunicação é realizada por troca de mensagens via rede. Além disso, uma abordagem híbrida pode ser utilizada de forma a obter proveito de ambas as arquiteturas. O OpenMP pode adicionar o conceito de *multithreading* (GROPP; LUSK; SKJELLUM, 1999) e utilizar, de forma mais eficiente, os núcleos de processamento disponíveis em cada computador que compõem o conjunto de multicomputadores ou *cluster*, executando o MPICH. A codificação híbrida também permite diminuir a latência devido à comunicação de mensagens MPICH migrando parte do código para o OpenMP.

Este capítulo apresenta a implementação dos algoritmos propostos no Capítulo 2 em OpenMP e MPICH. A Seção 4.1 descreve a implementação dos algoritmos em OpenMP. A Seção 4.2 detalha a implementação dos algoritmos em MPICH. Na Seção 4.3

são apresentados os algoritmos implementados na plataforma híbrida do OpenMP com o MPICH.

4.1 Implementação em OpenMP

Um multiprocessador com memória compartilhada (*Shared Memory Multiprocessor - SMP*) é um computador paralelo no qual todos os processadores compartilham um único espaço físico de endereçamento (PATTERSON; HENNESSY, 2011), como pode ser observado na Figura 14. 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*. Ainda assim, trabalhos independentes podem ser executados utilizando um espaço de endereçamento virtual.

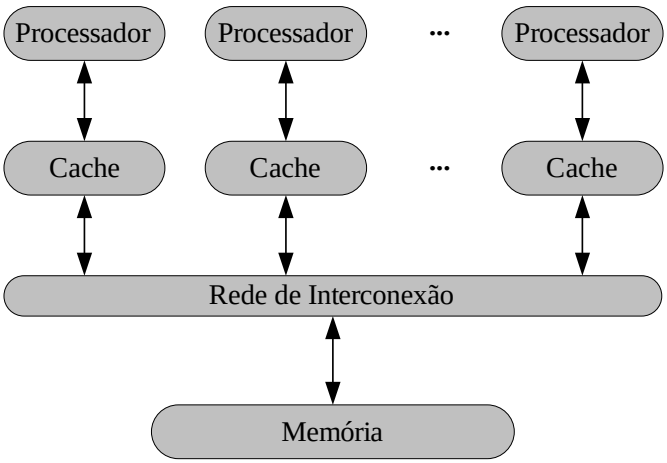


Figura 14: Arquitetura de multiprocessadores de memória compartilhada

O OpenMP (CHAPMAN; JOST; PAS, 2008) é uma API que foi desenvolvida para permitir e facilitar a programação no ambiente paralelo utilizando uma memória compartilhada. Dispõe de três componentes básicos: diretivas de compilação, bibliotecas de execução e variáveis de ambiente. Está API provê meios para o programador criar um conjunto de *threads* de execução paralela, especificar como os dados serão compartilhados entre as *threads*, declarar variáveis compartilhadas ou privadas e sincronizar as *threads*. A *thread* é uma entidade de execução que é capaz de executar um fluxo de instrução independente e é escalonada pelo Sistema Operacional (CHAPMAN; JOST; PAS, 2008). Se múltiplas *threads* colaboram para executar um programa, estas irão compartilhar o mesmo

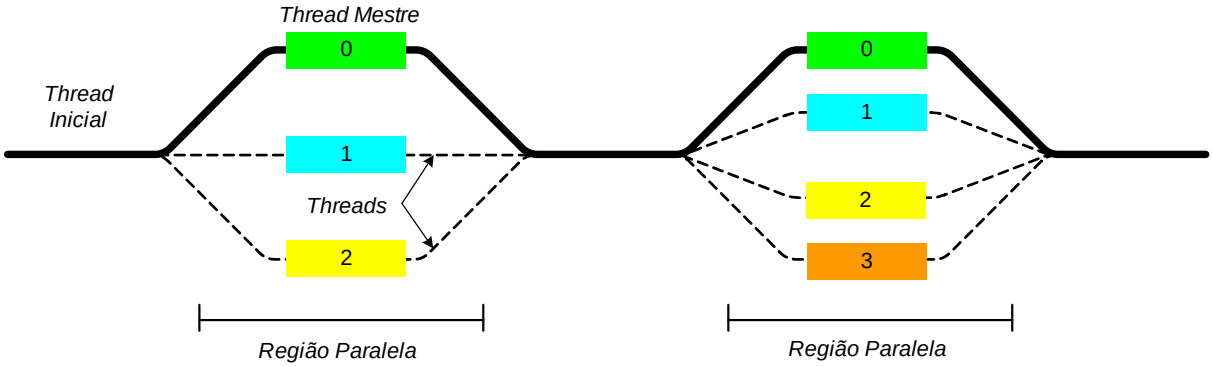


Figura 15: Modelo Fork-Join suportado pelo OpenMP.

espaço de endereçamento do processo. Porém, cada *thread* possui seu próprio conjunto de registradores, apontador de instrução e contador de programa.

O OpenMP utiliza o modelo de programação *Fork-Join* (DENNIS; HORN, 1966), ilustrado na Figura 15. Nesta abordagem, o programa começa de maneira semelhante a um processo simples, como um programa serial, denominado de *thread inicial*. Quando a *thread inicial* encontra uma construção paralela do OpenMP, então é criado um grupo de *threads* independentes e a *thread inicial* passa a ser denominada de *thread Mestre*. Este grupo de *threads* passa a colaborar entre si na execução do código até o termino da região paralela. Na região paralela, as *threads* podem compartilhar informações ou usar os dados de suas áreas privadas de memória. Quando o grupo de *threads* completar a execução da região paralela, as *threads* são sincronizadas e finalizadas, permanecendo apenas a *thread inicial*.

As construções do OpenMP proporcionam uma maneira simples de informar o compilador quais instruções devem ser executadas em paralelo e como distribuir o trabalho entre as *threads*. Cada diretiva do OpenMP inicia-se com *#pragma omp*, seguindo a convenção do C/C++ de passagem de diretivas ao compilador. Uma região paralela sempre inicia com a diretiva *parallel*. Conforme pode ser observado no Algoritmo 3, a linha 6 cria uma região paralela, as variáveis a e b são privadas e a variável c é compartilhada entre todas as *threads*.

Para definir o número de *threads* (nt) que irão executar uma região paralela é utilizado a função *omp_set_num_threads*(nt), conforme linhas 3 e 4 do Algoritmo 3. Dentro da região paralela, a função *omp_get_thread_num*() gera um identificador (tid) para cada *thread*. A *thread inicial* recebe o $tid = 0$, identificando-a como *thread Mestre*. As

demais *threads* são numeradas sequencialmente a partir de 1. O paralelismo de dados é implementado pelo uso do construtor de trabalho compartilhado *for*, conforme mostrado na linha 9 do Algoritmo 3. Este construtor especifica que as iterações serão distribuídas e executadas em paralelo por um grupo de *threads*. As iterações do laço são divididas com o emprego da clausula de escalonamento *static*. Este tipo de escalonamento divide o laço de maneira contínua e em partes iguais de acordo com o número de *threads* atribuídos à região paralela. Além disso, o escalonamento *static* apresenta baixo *overhead* quando comparado aos diferentes tipos de escalonamentos proporcionados pelo OpenMP. No final das iterações da construção *for*, as *threads* são sincronizadas antes de continuarem a execução do programa, a menos que a clausula *nowait* esteja especificada.

Algoritmo 3 Criação de uma região paralela no OpenMP

```
1: Início
2:     /* região sequencial */
3:     Seja nt número de threads
4:     omp_set_num_threads(nt)
5:     int a,b,c
6:     #pragma omp parallel private(a,b) shared(c)
7:     Início
8:         /* região paralela */
9:         #pragma omp for schedule(static)
10:        para i = 0 → p faça
11:            /* área de trabalho compartilhado */
12:        fim para
13:    Fim
14:    /* região sequencial */
15: Fim
```

4.1.1 PPSO

O algoritmo PPSO 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 as partículas para distribuir a computação entre as *threads* que irão executar a região paralela, conforme exibido no Algoritmo 4. 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 executará as partículas 0 a 31 e a *thread* 1 executará as partículas 32 a 63. O algoritmo PPSO 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.

Algoritmo 4 PPSO implementado em OpenMP

```
1: seja nt = número de threads
2: #pragma omp parallel
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_i
14:     fim para;
15:     #pragma omp for schedule(static)
16:     para i := 0 → n faça
17:         atualize Lbest()
18:         se (Lbest_i < 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
```

Para gerar números pseudoaleatórios em paralelo, necessários a execução da otimização, cada *thread* possui seu próprio gerador de números aleatórios, que é inicializado por uma semente, única para todas as *threads*, somada ao valor de *tid* para gerar sequências diferentes. Na operação de inicialização das informações das partículas, (linha 8 do Algoritmo 4) estão incluídas as inicializações da posição, velocidade, de $fitness$, do *Pbest* e *Lbest*. Os detalhes do procedimento *atualiza Lbest* (linha 17) são exibidos no Algoritmo 5. 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 nesta definição de vizinhança.

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 deste vetor. As outras *threads* consultam esta 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 4) esta operação é realizada somente após certo número de iterações, escolhido empiricamente como 20.

Algoritmo 5 Procedimento *atualize Lbest*

1: **se** $(Pbest_i < Lbest_i)$ **então**
2: **atualize** $Lbest_i$
3: **fim se**
4: **se** $(Pbest_{(i+1) \bmod n} < Lbest_i)$ **então**
5: **atualize** $Lbest_i$
6: **fim se**
7: **se** $(Pbest_{(i-1) \bmod n} < Lbest_i)$ **então**
8: **atualize** $Lbest_i$
9: **fim se**

4.1.2 PDPSO

O algoritmo PDPSO implementado em OpenMP divide o enxame em grupos de partículas a serem executadas em paralelo pelas *threads*, da mesma forma que o Algoritmo 4. Adicionalmente, uma segunda região paralela foi criada para a divisão das dimensões em grupos a serem executados por *threads* desta região. Esta nova região paralela foi implementada utilizando o paralelismo aninhado (*nested parallelism*) (CHAPMAN; JOST; PAS, 2008) do OpenMP. No paralelismo aninhado, se uma *thread*, pertencente a um grupo de *threads* executando uma região paralela, encontra outra diretiva de criação de região paralela, a *thread* cria um novo grupo e passa a ser a *thread Mestre* deste novo grupo. Os procedimentos para *cálculo de fitness* e *atualização da velocidade e posição* foram implementadas com o paralelismo aninhado. As atualizações de *Pbest* e *Lbest* foram mantidas somente com o paralelismo no laço das partículas pois a criação de regiões paralelas para a atualização das posições não melhoraram o desempenho do algoritmo.

O Algoritmo 6 apresenta a atualização da velocidade e posição para o algoritmo PDPSO. Note que, além de ser utilizado o construtor *for* no laço das partículas (linha 1 do Algoritmo 6), o mesmo construtor foi empregado no laço das dimensões junto com

a diretiva *parallel* para a criação de uma nova região paralela (linha 3 do Algoritmo 6). O Algoritmo 7 apresenta o cálculo de *fitness* para o algoritmo PDPSO. Dentro da região paralela aninhada, cada *thread* realiza a computação do cálculo de *fitness* relativa ao seu grupo de dimensões. A operação *reduction* do OpenMP realiza uma redução dos valores obtidos por cada *thread* para o seu respectivo grupo de dimensões. Os argumentos *operador* e *resultado* especificam o operador da função objetivo e o resultado da redução respectivamente. No final do laço de dimensões, as *threads* que executaram esta região irão computar o valor do seu resultado para a *thread Mestre* desta região, utilizando o *operador* da função objetivo.

Algoritmo 6 Procedimento *atualização da velocidade e posição*

```
1: #pragma omp for schedule(static)
2: para i := 0 → n faça
3:     #pragma omp parallel for schedule(static)
4:     para j := 0 → d faça
5:         atualize v_ij e x_ij
6:     fim para;
7: fim para;
```

Algoritmo 7 Procedimento *cálculo de fitness*

```
1: #pragma omp for schedule(static)
2: para i := 0 → n faça
3:     #pragma omp parallel
4:     #pragma omp for schedule(static) reduction(operador : resultado)
5:     para j := 0 → d faça
6:         calcule fitness_ij
7:     fim para;
8:     fitness_i:=resultado
9: fim para;
```

4.1.3 CPPSO

O algoritmo 8 apresenta uma visão geral do algoritmo CPPSO, implementado em OpenMP. O problema original com d dimensões é divido em k subenxames que se dedicam a um subgrupo de dimensões $e = d/k$. Do mesmo modo, o enxame original, composto de n partículas é dividido em k subenxames, cada um de $l = n/k$ partículas, onde cada subenxame otimiza um subproblema. Nesta abordagem, o subenxame é mapeado como uma *thread*. O vetor $Best$, de tamanho k, e o vetor de contexto $Bestx$, de tamanho d, são armazenados na memória compartilhada.

Algoritmo 8 CPPSO implementado em OpenMP

```
1: seja k = número de threads (subenxames)
2: #pragma omp parallel
3: início da região paralela
4: para i := 0 → l − 1 faça
5:    inicialize as informações das partículas do subenxame k
6:    se i = 0 então
7:       bestx[tid ∗ e + j] := x_ij
8:    fim se
9: fim para
10: sincronize threads
11: repita
12:    Tbestx:=bestx
13:    para i := 0 → l − 1 faça
14:       para j := 0 → e − 1 faça
15:          Tbestx[tid ∗ e + j]:=x_ij
16:       fim para;
17:       calcule fitness_i; atualize Pbest_i; atualize Lbest_i
18:       se (Lbest_i < Best[tid]) então
19:          atualize Best[tid]
20:          para j := 0 → e − 1 faça
21:             bestx[tid ∗ e + j]:=lbestx_ij
22:          fim para
23:       fim se
24:    fim para;
25:    para i := 0 → l − 1 faça
26:       atualize Lbest_i; atualize v_ij e x_ij
27:    fim para
28:    sincronize threads
29:    se tid = 0 então
30:       obtenha o menor valor em Best[tid]
31:    fim se
32: até condição de parada
33: fim da região paralela
34: retorne o resultado e a posição correspondente
```

Após o início da região paralela, as *threads* inicializam as velocidades e as coordenadas das partículas do seu respectivo subenxame. Em seguida, cada *thread* inicializa o vetor de contexto, na posição referente ao índice da *thread*, com a coordenada da partícula de índice 0 do respectivo subenxame. Depois deste procedimento, as *threads* são sincronizadas com o objetivo de impedir a cópia, do vetor de contexto não inicializado, para a memória local da *thread*.

Depois de sincronizadas, as *threads* realizam a cópia de todo o vetor de contexto para a sua memória local $Tbestx$ (linha 12 do Algoritmo 8). Antes de realizar o cálculo de

$fitness$, a *thread* substitui, na posição do vetor de contexto referente ao seu subenxame armazenado na memória local, as coordenadas relativas à partícula i (linhas 14 a 16 do Algoritmo 8). Os outros elementos do vetor de contexto são mantidos constantes. Assim, o cálculo de $fitness$ é realizado utilizando d dimensões, bem como a atualização de *Pbest* e *Lbest*. A primeira atualização de *Lbest* (linha 19 do Algoritmo 8) é referente ao valor de *Pbest* da própria partícula. Depois de realizada a atualização de *Lbest*, o vetor *Best* é atualizado com o menor valor de *Lbest* obtido pelas partículas do subenxame. Em seguida, o vetor de contexto é atualizado, na posição referente ao índice da *thread*, com as coordenadas de *Lbest* que foi selecionada.

Como antes, os vizinhos da partícula i são as partículas $(i + 1) \bmod n$ e $(i - 1) \bmod n$. Assim, *Lbest* é obtido de acordo com essa definição de vizinhança. Esta troca de informações do melhor local acontece apenas dentro de cada subenxame. Não há troca de informações entre os subenxames. Em seguida, é realizada a atualização da velocidade e posição. Neste ponto, as *threads* são sincronizadas e a *thread Mestre* realiza uma busca sequencial no vetor *Best* com a finalidade de obter o menor valor e armazená-lo na primeira posição do mesmo. As outras *threads* consultam esta posição durante a verificação da condição de parada. As operações de busca do melhor resultado do subenxame são realizadas a cada 20 iterações de modo a reduzir o *overhead* de sincronismo entre as *threads*.

4.2 Implementação em MPICH

Nos multicomputadores com troca de mensagens cada processador possui sua própria área de memória privada. Os processadores são interligados através de uma rede de alta velocidade e colaboram entre si na computação paralela por meio de troca de mensagens (PATTERSON; HENNESSY, 2011). A Figura 16 ilustra a organização de um multicomputador. É possível observar que, diferente dos multiprocessadores, a rede de interconexão não está entre a *cache* e a memória mas sim entre as memórias associadas a cada processador. Neste ambiente de troca de mensagens, os programas fazem uso de bibliotecas *message passing*, que possibilitam a comunicação entre os processadores. A interação entre os processos utiliza operações de *envio* e *recebimento* para realizar a troca de mensagens. Não é possível acessar a memória do outro processador utilizando as instruções *load* e *store*.

A interface de troca de mensagens (*Message Passing Interface* - MPI) (WALKER;

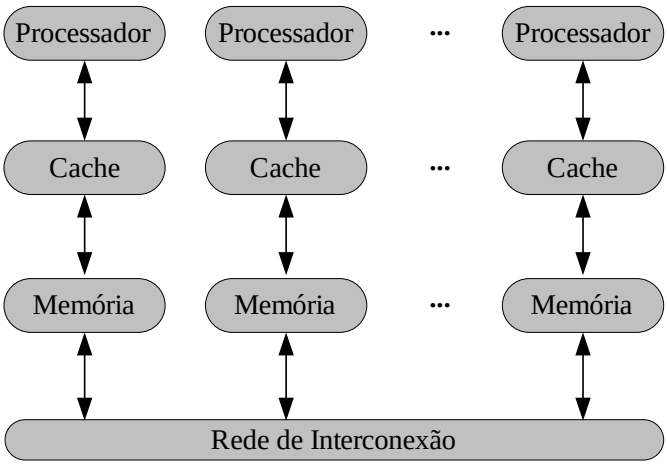


Figura 16: Modelo de multicomputadores com memória distribuída

DONGARRA, 1996) é uma especificação para um padrão de biblioteca de troca de mensagens, que foi definido no Forum MPI (MPI_FORUM, 2012). O MPICH (GROPP; SMITH, 1993) é uma implementação completa do padrão MPI, projetada para ser portável e eficiente em ambientes de alto desempenho. No MPI, os processos que se executam em paralelo têm espaços de endereços privados. A comunicação ocorre quando um processo A envia um dado do seu espaço de endereço para o *System Buffer* de um processo B, residente em outro processador, conforme ilustrado na Figura 17. O *System Buffer* é um endereço de memória reservado pelo sistema para armazenar mensagens. Esta operação é cooperativa e ocorre somente quando o processo A executa uma operação de *envio* e o processo B executa uma operação de *recebimento*.

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. O MPI relaciona os processos em grupos, e esses processos são identificados pela sua classificação dentro desse grupo. Essa classificação dentro do grupo é denominada *rank*. Assim, um processo no MPI é identificado por um número de grupo e por um *rank* dentro deste grupo. Um processo pode pertencer a mais de um grupo e, neste caso, possuir diferentes *ranks*. Um grupo utiliza um comunicador específico que descreve o universo de comunicação entre os processos. O MPI_COMM_WORLD é o comunicador pré-definido que incluem todos os processos,

definidos pelo usuário, em uma aplicação MPI.

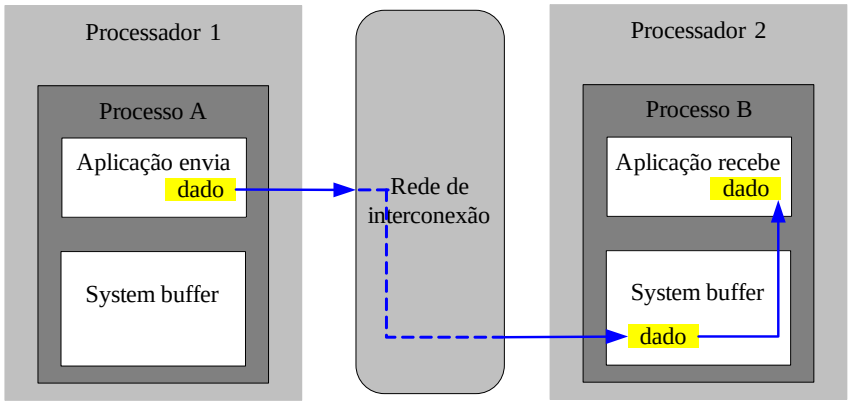


Figura 17: Comunicação por mensagem entre processos

O Algoritmo 9 exibe algumas das principais rotinas do MPI. O comando da linha 4 define e inicia o ambiente necessário para executar o MPI. A instrução da linha 5 identifica o processo dentro de um grupo de processos paralelos. A função da linha 6 retorna o número de processos dentro de um grupo de processos. A partir de então, cada processo executa em paralelo a computação especificada no bloco da região paralela e os processos cooperam entre si via troca de mensagens. A rotina da linha 9 finaliza os processos MPI.

Algoritmo 9 Exemplo de rotinas MPI implementada em MPICH

```
1:  Início
2:      /* região sequencial */
3:      int rank
4:      MPI_Init
5:      MPI_Comm_rank(MPI_COMM_WORLD,&rank)
6:      MPI_Comm_size(MPI_COMM_WORLD,&size)
7:      Início
8:          /* região paralela */
9:      MPI_Finalize()
10:     Fim
11:     /* região sequencial */
12: Fim
```

4.2.1 PPSO

O Algoritmo 10 apresenta um esboço do algoritmo PPSO 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 $rank$ para gerar sequencias diferentes. Em seguida, são inicializadas todas as informações das partículas referente a cada processo. Neste procedimento, estão incluídas as inicializações da posição, velocidade, de $fitness$, do $Pbest$ e $Lbest$. Os processos realizam a computação das novas velocidades e posição, cálculo de $fitness$ e atualização de $Pbest$ para o seu grupo de t partículas.

Algoritmo 10 PPSO implementado em MPICH

```
 1: seja p = número de processos; seja n = número de partículas
 2: faça t := n/p número de partículas por processo
 3: MPI_Init()
 4: srand(seed + rank)
 5: inicia as informações das partículas do processo
 6: repita
 7:    para i := 0 → t − 1 faça
 8:       atualize x_ij e v_ij
 9:       calcule fitness_i
10:       atualize Pbest_i
11:    fim para;
12:    envie mensagem com Pbest_{t−1} para o processo (rank + 1) mod n
13:    envie mensagem com Pbest_0 para o processo (rank − 1) mod n
14:    para i := 0 → t − 1 faça
15:       atualize Lbest
16:       atualize Best
17:    fim para;
18:    envie mensagem com Best de rank para o processo Mestre
19:    se rank = Mestre então
20:       se Best ≤ erro então
21:          ative a flag de saída
22:       fim se;
23:    fim se;
24:    Mestre envia mensagem com a flag de saída para os processos
25: até flag ativada
26: MPI_Finalize()
27: retorne o resultado e a posição correspondente
```

De forma a manter o comportamento de um único enxame, cada processo envia mensagem, referente a primeira e a última partícula do seu grupo, para o processo $(rank - 1) \bmod n$ e para o processo $(rank + 1) \bmod n$ respectivamente. Deste modo, a comunicação via mensagem é realizada somente entre a primeira e a última partícula de cada processo (linhas 12 e 13 do Algoritmo 10). As outras partículas se comunicam

apenas dentro do próprio processo. A atualização de *Lbest* é realizada de forma semelhante ao Algoritmo 5. Porém, a primeira e a última partícula de cada grupo acessam os seus respectivos *System Buffer* para proceder à atualização de *Lbest*. Durante o laço de atualização de *Lbest*, cada processo atualiza *Best* com o menor valor encontrado entre os valores de *Lbest* do grupo de partículas.

Com o objetivo de implementar uma condição de parada entre os processos paralelos, após a seleção de *Best*, 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 24 do Algoritmo 10). 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 18 a 24 do Algoritmo 10) são realizadas a cada 20 iterações de modo a reduzir o *overhead* de comunicação entre os *processos*.

4.2.2 PDPSO

O Algoritmo 11 apresenta uma visão geral do algoritmo PDPSO implementado em MPICH. De forma a implementar a decomposição de dados até o nível da dimensão, a partícula foi mapeada como um grupo de processos equivalentes ao número de dimensões do problema. Assim, são criados $p = n \times d$ processos (linha 2 do Algoritmo 11). Ao iniciar o ambiente MPI, cada processo inicializa seu identificador do *grupo local* de acordo com o seu respectivo $rank$ global. Em seguida, os processos recebem um comunicador local (*comunicador*) correspondente ao seu *grupo local* e um *rank local*, tornando-os capazes de trocarem mensagens sendo processos de um mesmo grupo.

A Figura 18 ilustra a organização em grupos e os comunicadores para um enxame de 4 partículas otimizando um problema com 3 dimensões. Cada círculo representa um processo com seu respectivo *rank local* seguido do seu $rank$ global. O processo com *rank local* igual a 0 (*rank local* $Mestre$) é utilizado para trocar mensagens entre partículas através do comunicador global, enquanto entre os processos correspondentes as dimensões de uma mesma partícula é utilizado o *rank local* e o comunicador do *grupo local*.

Cada processo executa diretamente à computação referente a sua dimensão nos procedimentos de atualização das novas velocidade e posição e cálculo de $fitness$ (linhas

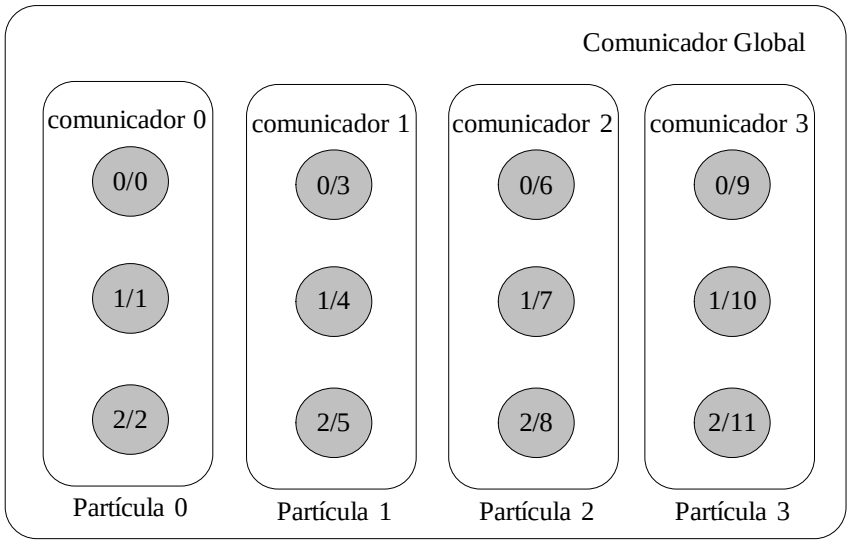


Figura 18: Geração de subgrupos de processo e comunicadores para um enxame de 4 partículas com 3 dimensões

13 e 14 do Algoritmo 11). A operação *MPI_Reduce* realiza uma redução de $fitness_j$ obtidos por cada *processo* na respectiva dimensão. Os argumentos *operador* e *comunicador* especificam o operador da função objetivo e o comunicador do *grupo local*, onde será realizada a redução respectivamente. Após esta operação, o processo $Mestre\ local$ receberá o valor de $fitness_i$ referente à partícula. Assim que a operação de redução é concluída, o processo $Mestre\ local$ envia $fitness_i$ para todos os processos que compõem a partícula. Neste momento, estes processos estão aptos a realizarem diretamente a atualização de $Pbest$. O processo $Mestre\ local$ de cada partícula envia então uma mensagem com $Pbest$ para os processos $(rank - 1)$ mod n e $(rank + 1)$ mod n (linhas de 18 a 21 do Algoritmo 11). Estes valores são difundidos aos processos do grupo local para que seja realizada a atualização de $Lbest$ (linhas 22 e 23 do Algoritmo 11).

Após a seleção de $Lbest$, cada partícula envia mensagem ao processo $Mestre\ global$ com o valor e a posição referente a seu respectivo $Lbest$. O processo $Mestre\ global$ verifica se, entre os resultados obtidos pelas partículas, algum satisfaz o critério de parada. Caso positivo, o processo $Mestre\ global$ ativa uma *flag* de saída e a envia por mensagem à todos os demais processos. De posse da *flag* ativada, os processos encerram a otimização e o processo $Mestre\ global$ retorna o valor e a posição encontrada. As operações de busca do melhor resultado (linhas 24 a 32 do Algoritmo 11) são realizadas a cada 20 iterações de modo a reduzir o *overhead* de comunicação entre os *processos*.

Algoritmo 11 PDPSO implementado em MPICH

1: **seja** n = número de partículas; d = número de dimensões
2: **faça** $p := n \times d$
3: **MPI_Init()**
4: **seja** *grupo local* como grupo de processos de uma mesma partícula
5: **se** $rank \leq d$ **então**
6: $\quad$ *grupo local* $:= 0$
7: **senão**
8: $\quad$ *grupo local* $:= |rank/d|$
9: **fim se**;
10: **inicialize** *comunicador* e *rank local*
11: **inicialize** as informações da dimensão da partícula
12: **repita**
13: $\quad$ **atualize** v_j e x_j
14: $\quad$ **calcule** $fitness_j$
15: $\quad$ **MPI_Reduce**($fitness_{ij}$, $fitness_i$, *operador*, *Mestre local*, *comunicador*)
16: $\quad$ **envie** mensagem com $fitness_i$ aos processos de mesmo *comunicador*.
17: $\quad$ **atualize** $Pbest_j$
18: $\quad$ **se** *rank local* $= Mestre$ **então**
19: $\quad\quad$ **envie** mensagem com $Pbest_i$ para o processo $(rank + 1)$ mod n
20: $\quad\quad$ **envie** mensagem com $Pbest_i$ para o processo $(rank - 1)$ mod n
21: $\quad$ **fim se**
22: $\quad$ **envie** mensagem com $Pbest_{rank+1}$ e $Pbest_{rank-1}$ aos processos de mesmo *comunicador*
23: $\quad$ **atualize** $Lbest_{ij}$
24: $\quad$ **se** *rank local* $= Mestre$ **então**
25: $\quad\quad$ **envie** mensagem com $Lbest_i$ para o processo $Mestre$
26: $\quad$ **fim se**
27: $\quad$ **se** $rank = Mestre$ **então**
28: $\quad\quad$ **se** $Lbest_i \leq erro$ **então**
29: $\quad\quad\quad$ **ative** a *flag* de saída
30: $\quad\quad$ **fim se**;
31: $\quad$ **fim se**;
32: $\quad$ **Mestre** envia mensagem com a *flag* de saída para os processos
33: **até** $flag$ ativada
34: **retorne** o resultado e a posição correspondente

4.2.3 CPPSO

O Algoritmo 12 apresenta uma visão geral do algoritmo CPPSO implementado em MPICH. O problema original com d dimensões é divido em k subenxames que se dedicam a um subgrupo de dimensões $e = d/k$. Do mesmo modo, o enxame original, composto de n partículas é dividido em k subenxames, cada um de $l = n/k$ partículas, onde cada subenxame otimiza um subproblema. Nesta abordagem o subenxame é mapeado como um *processo*. O vetor de contexto é chamado de $Bestx$, de tamanho d, e fica armazenado na

memória local de cada processo. Para a troca de informações do vetor de contexto entre os processos, foi utilizado o vetor $Kbestx$, de tamanho e, contendo apenas os componentes do vetor de contexto referente ao subenxame.

Algoritmo 12 CPPSO implementado em MPICH

1: **seja** k = número de processos (enxames)
2: **MPI_Init()**
3: **inicie** as informações das partículas e inicialize $Kbestx$
4: **envie** mensagem para todos os processos com $Kbestx$
5: **obtenha** o vetor de contexto $Bestx$ utilizando $Kbestx$ dos outros processos
6: **repita**
7: **para** $i := 0 \rightarrow l$ **faça**
8: $Bestx[rank * e + j] := x_{ij}$
9: **calcule** $fitness_i$
10: **atualize** $Pbest_i$
11: **atualize** $Lbest_i$
12: **se** $Lbest_i < Best$ **então**
13: $Best$:=$Lbest_i$
14: $Kbestx[j] := Lbest_{ij}$
15: **fim se**
16: **fim para**
17: **para** $i := 0 \rightarrow l$ **faça**
18: **atualize** $Lbest$
19: **atualize** x_{ij} e v_{ij}
20: **fim para**;
21: **envie** mensagem com $Kbestx$ para todos os processos
22: **atualize** o vetor de contexto $Bestx$ utilizando $Kbestx$
23: **envie** mensagem com $Best$ para o processo $Mestre$
24: **se** $rank = Mestre$ **então**
25: **se** $Best \leq erro$ **então**
26: **ative** a *flag* de saída
27: **fim se**;
28: **fim se**;
29: **Mestre** envia mensagem com a *flag* de saída para os processos
30: **até** $flag$ ativada
31: **MPI_Finalize()**
32: **retorne** o resultado e a posição correspondente

Ao iniciar o ambiente MPI, os processos inicializam as informações das l partículas, bem como $Kbestx$ com as coordenadas da primeira partícula do respectivo subenxame. Em seguida, cada processo envia sua mensagem para todos os outros processos com $Kbestx$ referente ao seu subenxame. Quando o processo receber os valores de $Kbestx$, referente aos outros subenxames, ele monta o vetor de contexto armazenando $Kbestx$ na posição referente ao $rank$ do processo que o enviou. Antes de realizar o cálculo de $fitness$,

cada processo substitui, na posição do vetor de contexto referente ao seu subenxame armazenado na memória local, as coordenadas relativas a partícula i (linha 8 do Algoritmo 12). As outras entradas do vetor de contexto são mantidas. Assim, o cálculo de $fitness$ é realizado utilizando d dimensões, bem como a atualização de $Pbest$ e $Lbest$. A primeira atualização de $Lbest$ (linha 11) é referente ao valor de $Pbest$ da própria partícula.

Após o procedimento *atualize Lbest*, $Kbestx$ é atualizado com as coordenadas de $Lbest$ que foi selecionada como melhor posição obtida pelo subenxame. Em seguida, é realizada a atualização de $Lbest$ (linha 18 do Algoritmo 12) com o valor obtido conforme a definição de vizinhança $(i + 1) \bmod n$ e $(i - 1) \bmod n$. Esta troca de informações do melhor local acontece apenas dentro de cada subenxame. Não há troca de informações entre os enxames. Na sequência é realizado o cálculo das novas velocidade e posição.

Cada processo envia uma mensagem com $Kbestx$ para os demais processos. Ao receberem essas mensagens, cada processo atualiza seu vetor de contexto $Bestx$. Em seguida, o processo $Mestre$ recebe o menor valor e posição encontrado por cada subenxame. O processo $Mestre$ verifica então 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. De posse da *flag* ativada, os processos encerram a otimização e o processo $Mestre$ retorna o valor e posição encontrada. As operações de busca do melhor resultado de cada subenxame (linhas 23 a 29 do Algoritmo 12) são realizadas a cada 20 iterações de modo a reduzir o *overhead* de comunicação entre os *processos*.

4.3 Implementação em OpenMP com MPICH

A Combinação do OpenMP com MPICH oferece uma abordagem hierárquica para explorar o paralelismo inerente da aplicação ou da arquitetura do processador de forma mais eficiente (CHAPMAN; JOST; PAS, 2008). Assim, adequa-se perfeitamente às arquiteturas baseadas em *cluster de multiprocessadores*, pois permite diminuir o número de comunicações em diferentes nós e aumentar o desempenho de cada nó. O estilo de programação híbrida é mais eficiente quando os processos MPI trabalham no nível de granularidade mais grossa, tirando vantagem da distribuição de dados e do escalonamento de tarefas, enquanto a paralelização baseada no OpenMP é utilizada na arquitetura de multiprocessador e proporciona à cada processo uma paralelização com granularidade mais fina. Esta combinação é retratada na Figura 19. A codificação híbrida também oferece a van-

tagem de reduzir a latência de comunicação entre mensagens MPI, migrando parte do código para o OpenMP. Adicionalmente, esta abordagem tem a capacidade de aumentar a escalabilidade da implementação. Nas implementações desta seção, as regiões paralelas OpenMP e MPI são criadas e mantidas durante toda a execução da otimização, de forma a reduzir o *overhead* de criação a cada iteração.

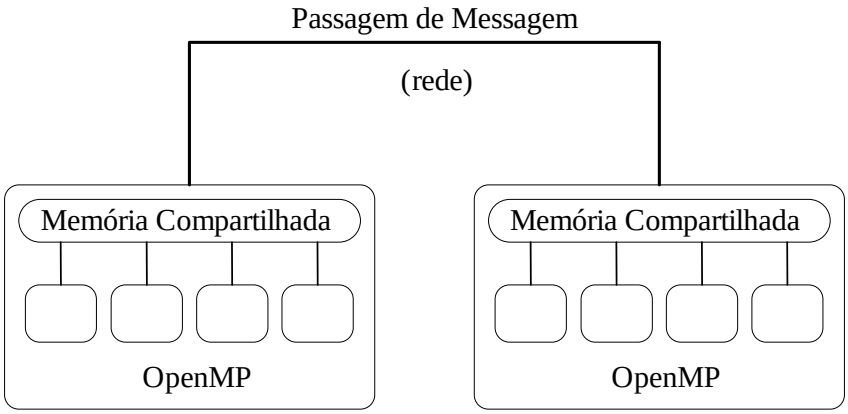


Figura 19: Arquitetura híbrida OpenMP com MPI

4.3.1 PPSO

O Algoritmo 13 apresenta uma visão geral do algoritmo PPSO implementado em OpenMP com MPICH. Esta implementação é semelhante ao Algoritmo 10. Porém, cada processo MPI, mapeando um grupo de t partículas, cria uma região paralela OpenMP. Assim, os laços dos grupos de partículas são divididos entre as *threads* para serem executados. Em cada laço, referente a um grupo de partículas, de partículas é utilizado o construtor for juntamente com a cláusula de escalonamento *static*.

No começo da região paralela, são inicializados os geradores de números pseudo-aleatórios, somando a semente (*seed*) com o identificador da *thread* e o identificador do processo multiplicado por uma constante c. O valor de c deve ser maior que o valor da maior semente utilizada para gerar sequências diferentes. São inicializadas ainda, todas as informações das partículas referente a cada processo. Neste procedimento, estão incluídas as inicializações da posição, velocidade, cálculo de $fitness$, $Pbest$ e $Lbest$. Os processos realizam a computação das novas velocidades e posição, cálculo de $fitness$ e atualização de $Pbest$ para o seu grupo de t partículas. Para a troca de informações sobre $Pbest$ entre processos diferentes, apenas a *thread Mestre* realiza o envio e o recebimento de mensagens

Algoritmo 13 PPSO implementado em OpenMP com MPICH

```
1: seja p=número de processos; seja n=número de partículas
2: faça t:= n/p
3: MPI_Init()
4: #pragma omp parallel
5: início da região paralela OpenMP
6: srand(seed + tid + rank × c)
7: inicie as informações das partículas
8: repita
9:    #pragma omp for schedule(static)
10:   para i := 0 → t − 1 faça
11:      atualize x_ij e v_ij
12:      calcule fitness_i
13:      atualize Pbest_i
14:   fim para;
15:   se tid = Mestre então
16:      envie mensagem com Pbest_{t−1} para o processo (rank + 1) mod n
17:      envie mensagem com Pbest_0 para o processo (rank − 1) mod n
18:   fim se
19:   #pragma omp for schedule(static)
20:   para i := 0 → t − 1 faça
21:      atualize Lbest; atualize Best_tid
22:   fim para;
23:   se tid = Mestre então
24:      obtenha o menor valor em Best
25:      envie mensagem com Best de rank para o processo Mestre
26:      se rank = Mestre então
27:         se Best ≤ erro então
28:            ative a flag de saída
29:         fim se;
30:      fim se;
31:      Mestre envia mensagem com a flag de saída para os processos
32:   fim se
33:   sincronize threads
34: até flag ativada
35: fim da região paralela OpenMP
36: MPI_Finalize()
37: retorne o resultado e a posição correspondente
```

MPI. De forma a manter o comportamento de um único enxame, cada processo envia uma mensagem, referente a primeira e a 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 13 a 16 do Algoritmo 13). As outras partículas se comunicam entre si apenas dentro do próprio processo.

Com o objetivo de paralelizar a busca pelo menor resultado do processo, foi definido o vetor $Best$, cujo tamanho é igual ao número de *threads* (nt). Após a atualização de $Lbest$, cada *thread* verifica o menor resultado obtido pelas suas partículas e o armazena, no vetor $Best$, na posição correspondente ao seu tid. Em seguida, a *thread Mestre* realiza uma busca sequencial no vetor $Best$ e armazena o menor resultado obtido na primeira posição do mesmo. A *thread Mestre* consulta esta posição e envia uma mensagem com seu $Best$ para o processo $Mestre$.

O processo $Mestre$ verifica então se, entre os resultados obtidos, algum satisfaz a condição de parada. As demais *threads* permanecem aguardando na barreira de sincronismo, conforme linha 31 do Algoritmo 13. Caso positivo, o processo $Mestre$ ativa uma *flag* de saída e a envia por mensagem a todos os processos, conforme linha 29 do Algoritmo 13. De posse da *flag* ativada, os processos encerram a otimização e o processo $Mestre$ retorna o valor e posição encontrada. As operações de busca do melhor resultado de cada processo (linhas 21 a 31 do Algoritmo 13) são realizadas a cada 20 iterações de modo a reduzir o *overhead* de comunicação e sincronismo entre os processos.

4.3.2 PDPSO

O Algoritmo 14 apresenta um esboço do algoritmo PDPSO implementado em OpenMP com MPICH. Cada partícula é mapeada como um processo MPI, enquanto o laço referente as dimensões é paralelizado pelo construtor *for* do OpenMP. Os geradores de números pseudoaleatórios são inicializados da mesma maneira que no caso do PPSO. Para o cálculo de $fitness$ foi utilizada a operação *reduction* do OpenMP para a redução dos valores de $fitness$, obtidos por cada *thread* nas respectivas dimensões. No final do laço de dimensões, as *threads* que executaram esta região, irão computar o valor do seu resultado para a *thread* com índice igual a 0 (*thread Mestre*), utilizando o *operador* da função objetivo. Em seguida, $Pbest$ é atualizado e seu valor e posição são enviados por mensagem pela *thread Mestre* aos processos $(rank+1)$ mod n e $(rank-1)$ mod n. Após a atualização de $Lbest$, os processos enviam mensagem com o valor e a posição de seu respectivo $Lbest$ para o processo $Mestre$.

O processo $Mestre$ verifica então se, entre os resultados obtidos, algum satisfaz a condição de parada. As demais *threads* permanecem aguardando na barreira de sincronismo, conforme linha 28 do Algoritmo 14. Caso positivo, o processo $Mestre$ ativa

Algoritmo 14 PDPSO implementado em OpenMP com MPICH

```
1: seja p=número de processos (partículas)
2: seja d=número de dimensões
3: MPI_Init()
4: #pragma omp parallel
5: início da região paralela OpenMP
6: inicia as informações da particula_rank
7: repita
8:    #pragma omp for schedule(static)
9:    para j := 0 → d faça
10:      atualize x_j e v_j
11:   fim para;
12:   #pragma omp for schedule(static) reduction(operador : resultado)
13:   para j := 0 → d faça
14:      calcule fitness_j
15:   fim para;
16:   fitness:=resultado
17:   se tid = Mestre então
18:      atualize Pbest
19:      envie mensagem com Pbest para o processo (rank + 1) mod n
20:      envie mensagem com Pbest para o processo (rank − 1) mod n
21:      atualize Lbest
22:      envie mensagem para o processo Mestre com Lbest
23:      se rank = Mestre então
24:         se Lbest_rank ≤ erro então
25:            ative a flag de saída
26:         fim se;
27:      fim se;
28:      Mestre envia mensagem com a flag de saída para os processos
29:   fim se
30:   sincronize threads
31: até flag ativada
32: fim da região paralela OpenMP
33: MPI_Finalize()
34: retorne o resultado e a posição correspondente
```

uma *flag* de saída e a envia por mensagem a todos os processos, conforme linha 26 do Algoritmo 14. De posse da *flag* ativada, os processos encerram a otimização e o processo *Mestre* retorna o valor e posição encontrada. As operações de busca do melhor resultado de cada processo (linhas 20 a 28 do Algoritmo 14) são realizadas a cada 20 iterações de modo a reduzir o *overhead* de comunicação e sincronismo entre os *processos*.

4.3.3 CPPSO

O Algoritmo 15 apresenta uma visão geral do algoritmo CPPSO implementado em OpenMP com MPICH. O problema original com d dimensões é divido em k subenxames que se dedicam a um subgrupo de dimensões $e = d/k$. Do mesmo modo, o enxame original, composto de n partículas é dividido em k subenxames, cada um de $l = n/k$ partículas, onde cada subenxame otimiza um subproblema.

A implementação é semelhante ao Algoritmo CPPSO em MPICH. Porém, cada processo MPI, mapeado como um subexame, cria uma região paralela OpenMP. Desta forma, os laços das partículas que compõem o subexame são divididos e executados pelas *threads* que compõem a região paralela. Em cada laço de partículas, é utilizado o construtor for juntamente com a cláusula de escalonamento *static*. Para a troca de informações do vetor de contexto ($Bestx$) entre os processos, foi utilizado o vetor $Kbestx$, de tamanho e, contendo apenas os componentes do vetor de contexto referente ao subenxame. Somente a *thread Mestre* realiza a atualização de $Kbestx$ (linhas 16 a 21 do Algoritmo 15).

Com o objetivo de paralelizar a busca pelo menor resultado do processo, foi definido o vetor $Best$, cujo tamanho é igual ao número de *threads* (nt). Após a atualização de $Lbest$, cada *thread* verifica o menor resultado obtido pelas suas partículas e o armazena no vetor $Best$, na posição correspondente ao seu *tid*. Em seguida, a *thread Mestre* realiza uma busca sequencial ao vetor $Best$ e armazena o menor resultado obtido na primeira posição deste vetor. A *thread Mestre* consulta esta posição e envia mensagem com seu $Best$ para o processo $Mestre$. O processo $Mestre$ verifica então 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 demais processos. De posse da *flag* ativada, os processos encerram a otimização e o processo $Mestre$ retorna o valor e posição encontrada. As operações de busca do melhor resultado de cada subenxame (linhas 30 a 40 do Algoritmo 15) são realizadas a cada 20 iterações de modo a reduzir o *overhead* de comunicação e sincronismo entre os processos.

Algoritmo 15 CPPSO implementado em OpenMP com MPICH

1: **seja** k = número de processos (enxames)
2: **MPI_Init()**
3: #pragma omp parallel
4: **início** da região paralela OpenMP
5: **inicie** as informações das partículas e inicialize $Kbestx$
6: **se** $tid = Mestre$ **então**
7: **envie** mensagem com $Kbestx$ para todos os processos
8: **monte** o vetor de contexto $Bestx$ utilizando $Kbestx$ dos outros processos
9: **fim se**
10: **repita**
11: #pragma omp for schedule(static)
12: **para** $i := 0 \rightarrow l$ **faça**
13: $Bestx[rank * e + j] := x_{ij}$
14: **calcule** $fitness_i$; **atualize** $Pbest_i$; **atualize** $Lbest_i$
15: **fim para**
16: **se** $tid = Mestre$ **então**
17: **se** $Lbest_i < Best$ **então**
18: $Best$:=$Lbest_i$
19: $Kbestx[j] := Lbest_{ij}$
20: **fim se**
21: **fim se**
22: #pragma omp for schedule(static)
23: **para** $i := 0 \rightarrow l$ **faça**
24: **atualize** $Lbest$; **atualize** x_{ij} e v_{ij}
25: **fim para**;
26: **se** $tid = Mestre$ **então**
27: **envie** mensagem com $Kbestx$ para todos os processos
28: **atualize** o vetor de contexto $Bestx$ com $Kbestx$
29: **fim se**
30: **se** $tid = Mestre$ **então**
31: **obtenha** o menor valor em $Best$
32: **envie** mensagem com $Best$ de $rank$ para o processo $Mestre$
33: **se** $rank = Mestre$ **então**
34: **se** $Best \leq erro$ **então**
35: **ative** a *flag* de saída
36: **fim se**;
37: **fim se**;
38: **Mestre** envia mensagem com a *flag* de saída para os processos
39: **fim se**
40: **sincronize** *threads*
41: **até** $flag$ ativada
42: **fim** da região paralela OpenMP
43: **MPI_Finalize()**
44: **retorne** o resultado e a posição correspondente

4.4 Considerações Finais do Capítulo

Neste capítulo foram apresentadas as implementações dos algoritmos propostos em OpenMP e MPICH. Inicialmente, foram descritas as implementações na arquitetura de multiprocessador utilizando a interface OpenMP. Em seguida, os algoritmos foram especificados na arquitetura de multicomputadores utilizando MPICH. Por fim, os algoritmos foram apresentados utilizando a programação híbrida do OpenMP com o MPICH. O capítulo a seguir apresenta a implementação dos algoritmos propostos na arquitetura de GPU utilizando CUDA.

Capítulo 5

IMPLEMENTAÇÃO EM CUDA

ESTE capítulo apresenta a implementação dos algoritmos propostos utilizando o modelo de programação CUDA como plataforma para a Unidade de Processamento Gráfico (*Graphics Processing Unit* - GPU). Inicialmente o processamento de vídeo era realizado pelo controlador gráfico de vídeo. Ao longo do tempo, estes controladores começaram a incorporar diversas funções de aceleração para tratamento de gráficos em 3D. Assim, este controlador passou a ser denominado de GPU denotando que os dispositivos gráficos haviam se tornado um processador. Recentemente, um conjunto de instruções e um *hardware* de memória foram adicionadas para dar suporte a linguagens de programação e um ambiente de programação foi criado para permitir que a GPU seja programada com linguagens de programação, incluindo C e C++. Desta forma, as GPUs passaram a ser processadores programáveis massivamente paralelos com centenas de cores permitindo a execução de milhares de *threads* (PATTERSON; HENNESSY, 2011).

A organização deste capítulo é descrita a seguir: A Seção 5.1 introduz o modelo de programação CUDA. Na Seção 5.2 é apresentada a arquitetura *multithreading* do multiprocessador da GPU. As Seções 5.5, 5.6 e 5.7 descrevem as implementações dos algoritmos propostos PPSO, PDPSO e CPPSO, respectivamente, na plataforma CUDA.

5.1 Modelo de Programação CUDA

A plataforma de computação paralela CUDA (*Compute Unified Device Arquitecture*) (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 sem a necessidade de utilizar as interfaces de programação de aplicativos (*Application Programming Interface* - API). O modelo de programação CUDA

possui um estilo de uma única instrução sobre vários dados (*Single Instruction, Multiple Data* - SIMD), 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).

CUDA provê três abstrações chaves: 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* deste *bloco* e a barreira de sincronização é um ponto onde as *threads* de um mesmo *bloco* esperam até que todas as *threads* deste *bloco* alcancem a barreira (KIRK; HWU, 2010).

Ao contrário das *threads* utilizadas na arquitetura de CPU, é desejável que as *threads* CUDA sejam extremamente leves, ou seja as tarefas individuais devem ser relativamente pequenas em termo de tamanho de código e tempo de execução, facilitando assim uma rápida troca de contexto. Isto é importante para reduzir o *overhead* de criação da *thread* assim como seu escalonamento na GPU (PATTERSON; HENNESSY, 2011).

O CUDA C estende a linguagem C permitindo que o programador defina funções denominadas *kernels*. O *kernel* é executado por um vetor de *threads* que executam o mesmo código. Ao chamar um *kernel*, o programador especifica o número de *threads* por *bloco* e o número de *blocos* que irão compor a *grade*. Cada *thread* possui um único identificador *thread ID*, dentro de seu *bloco* e cada *bloco* tem um único identificador *block ID* na sua *grade* conforme Figura 20. Uma função *kernel* é definida pela declaração __global__ e os programas CUDA chamam os *kernels* paralelos com a seguinte sintaxe: kernel$\lll$ *blocos*, *threads* $\ggg$(...parâmetros...), onde *blocos* e *threads* correspondem respectivamente a quantidade de *blocos* da *grade* e a quantidade de *threads* do *bloco*. As *threads* podem acessar dados de múltiplas áreas de memória durante sua execução. Cada *thread* tem sua própria área de memória privada, chamada de *memória local*. Cada *bloco* de *threads* possui sua área de memória privada, visível para todas as *threads* em um mesmo *bloco* e possui a mesma vida útil do *bloco*, chamada *memória compartilhada*.

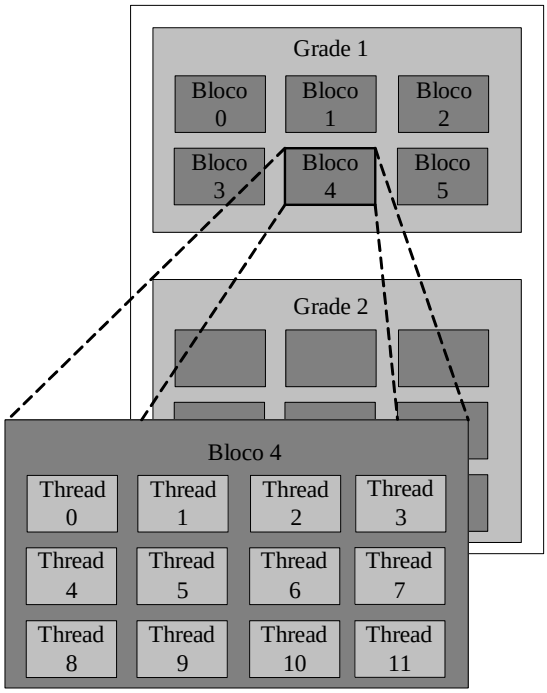


Figura 20: Decomposição resultante da *grade* em *blocos* e dos *blocos* em *threads*

Finalmente, todas as *threads*, ainda que estejam em *grades* diferentes, podem acessar a mesma memória, denominada *memória global*, conforme ilustrado na Figura 21.

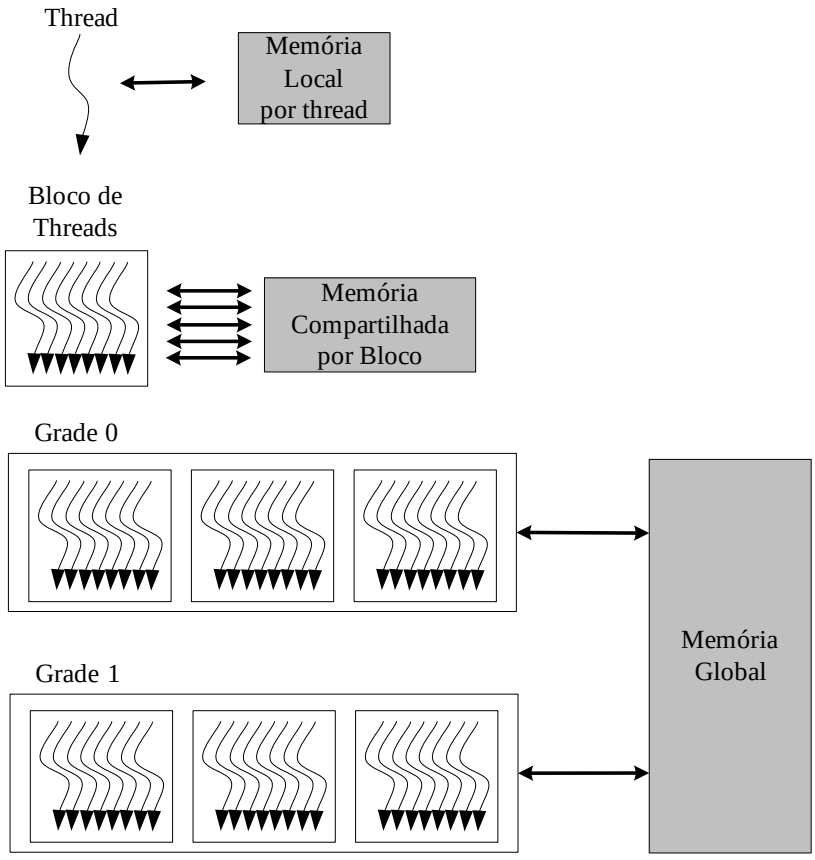


Figura 21: Hierarquia de Memória

5.2 Arquitetura *Multithreading*

A GPU implementa um número de multiprocessadores (*Streaming Multiprocessor* - SM) e cada multiprocessador é composto por vários processadores. O SM é capaz de criar, escalonar, gerenciar e executar concorrentemente *threads* em grupos de *threads* paralelas denominados *warps*. A execução de milhares de *threads* pelo SM tem como objetivo cobrir a latência de acesso a memória global, dar suporte a computação paralela com granularidade fina do modelo de programação, além de simplificar o modelo de programação paralela, desenvolvendo um programa serial para uma única *thread*.

O acesso a memória global pode requerer centenas de ciclos de *clock* do processador, ainda mais que GPUs possuem tipicamente uma memória cache menor do que a da CPU. As operações de *multhithreading* ajudam a cobrir o tempo de latência com computação. Enquanto uma *thread* está aguardando para que o acesso aos dados da memória se complete, outra está em execução no processador. O modelo de programação paralela com granularidade fina provê milhares de *threads* independentes que podem manter o processador ocupado enquanto outras *threads* estão esperando os dados da memória (FARBER, 2011) (CALAZAN; NEDJAH; MOURELLE, 2012a).

A Figura 22 mostra um SM da GPU GTX 460, arquitetura FERMI (NVIDIA, 2009), utilizada neste trabalho. Esta GPU possui 7 SM, sendo cada um deles composto de 48 núcleos de processamento SIMT (*Single Instruction, Multiple Threads*), 32K de registradores, 8 unidades de funções especiais, 4 unidades de instruções *multithread*, uma cache de instrução e uma memória compartilhada. Os blocos de *threads* são atribuídos para serem executados por SM. No SM, as *threads* são divididas por *warp*, com cada *warp* sendo composto por até 32 *threads*. Assim, o escalonador de *warp* pode selecionar um *warp* pronto para ser executado nos núcleos do SM. Cada núcleo SIMT do SM é um *hardware multithreading* que contem uma unidade lógica e aritmética e uma unidade de ponto flutuante. O núcleo SIMT pode executar concorrentemente mais *threads* que usam poucos registradores ou menos *threads* que usam mais registradores. As unidades de *LD/ST* (*Load/Store*) calculam o endereço de origem e o de destino para até 16 *threads* simultaneamente. As Unidades de Funções Especiais (*Special Function Units* - SFU) executam instruções tais como seno, cosseno e raiz quadrada. Estas instruções são executadas concorrentemente com as instruções dos núcleos SIMT.

A arquitetura SIMT é semelhante à arquitetura SIMD, a qual aplica uma instru-

ção a múltiplos dados, porém na arquitetura SIMT cada *thread* possui seu contador de programa e ainda pode executar um caminho de dados independente de outras *threads*. Embora o SM execute um *warp* com maior eficiência quando o caminho de dados das *threads* que compõem este *warp* é o mesmo, este requisito não é necessário.

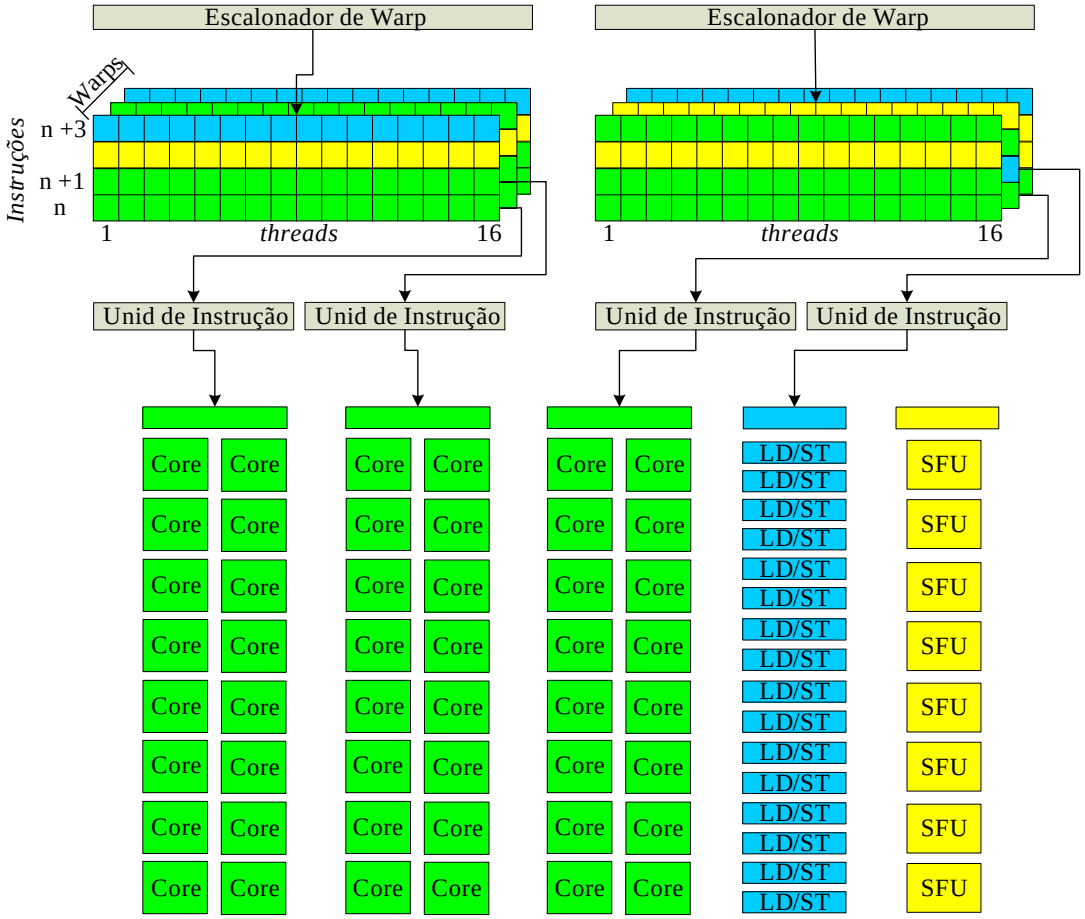


Figura 22: SM da GPU GTX 460

O SM, conforme exibido na Figura 22, escalona as *threads* por *warp* e cada SM possui dois escalonadores, permitindo que dois *warps* sejam executados ao mesmo tempo. Cada *warp* escalonado é executado por grupos de 16 núcleos SIMT, 16 *LD/ST*, ou 8 SFU, ou seja, apenas a metade das *threads* do *warp* são executadas por vez. O SM possui ainda 4 unidades de envio de instrução, 2 por escalonador. Estas unidades permitem que o SM realize operações superescalares. Deste modo, uma mesma *thread* de um *warp* escalonado pode executar duas diferentes instruções simultaneamente.

Enquanto um *warp* está aguardando o acesso a memória global, o escalonador do *warp* seleciona outro *warp* para ser executado (PATTERSON; HENNESSY, 2011). O contexto de execução (registradores, contador do programa, etc.) para cada *warp* processado pelo

SM é mantido *on-chip* durante toda execução. Assim, alterando a execução de um *warp* para outro não haverá custo adicional para carga dos registradores. Esta mudança permite que o tempo de latência no acesso a memória global fique sobreposto ao tempo de execução de outras *threads* de um *warp* disponível.

5.3 Gerador de Números Aleatórios

Durante a execução das otimizações, o algoritmo PSO precisa de uma quantidade considerável de números aleatórios para realizar o cálculo da nova velocidade da partícula. De modo a gerar números pseudoaleatórios na GPU, foi adotada a biblioteca *CURAND library* (NVIDIA, 2010b) que provê facilidades para gerar estes números com eficiência. Esta biblioteca define funções para configurar os estados dos geradores assim como gerar sequências de números aleatórios.

Isto permite que os números aleatórios sejam gerados e consumidos diretamente pelo *kernel* sem requerer que os números aleatórios sejam escritos e depois lidos da memória global. Apenas o estado do gerador precisa ser armazenado na memória global entre as chamadas do *kernel* para que a sequência seja preservada. Durante a execução do *kernel*, o estado é armazenado na memória local da *thread* para aumentar o desempenho.

5.4 Organização dos Dados

Os dados referentes a velocidade, posição e $fitness$, $Pbest$ e $Lbest$ de todas as partículas são armazenadas na memória global. Como a memória global permite apenas a alocação de vetores unidimensionais, apenas estas estruturas serão utilizadas para armazenar as informações das partículas. A geração de um identificador exclusivo para cada *thread* é realizada utilizando uma combinação de três variáveis: *threadIdx*, que contém o índice da *thread* dentro de um bloco, *blockIdx* que contém o índice do bloco dentro de uma *grade* e *blockDim* que contem a dimensão do bloco. Assim, o identificador global de cada *thread*, denominado *tid*, é calculado de acordo com a Equação 8.

$$tid = threadIdx + blockIdx \times blockDim \tag{8}$$

5.5 PPSO

O algoritmo 16 apresenta uma visão geral do algoritmo PPSO 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 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 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, esta operação é realizada somente após certo número de iterações, escolhido empiricamente como 20.

Algoritmo 16 PPSO implementado em CUDA

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

O primeiro *kernel* cria e inicializa na memória global $b \times t$ geradores de números aleatórios, um para cada *thread*, baseados na biblioteca *curand_kernel.h*, conforme explicado na Seção 5.3. 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 17, cria $b \times t$ *threads* que irão realizar o calculo da próxima velocidade e posição para cada partícula. Note 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 18, também gera $b \times t$ *threads* que irão executar o calculo de $fitness$ juntamente com a atualização de *Pbest*.

Algoritmo 17 kernel calcule a velocidade e posição

1: **faça** $tid = threadIdx + blockIdx \times blockDim$
2: **para** $j = 0 \rightarrow d - 1$ **faça**
3: **calcule** $v[tid \times d + j]$; **calcule** $x[tid \times d + j]$
4: **fim para**;

Algoritmo 18 kernel calcule $fitness$ e $Pbest$

1: **Seja** $tid = threadIdx + blockIdx \times blockDim$
2: **para** $j = 0 \rightarrow d - 1$ **faça**
3: **calcule** $fitness$
4: **fim para**;
5: **se** $(fitness[tid] < Pbest[tid])$ **então**
6: **para** $j = 0 \rightarrow d - 1$ **faça**
7: $Pbestx[tid \times d + j] := x[tid \times d + j]$
8: **fim para**;
9: **fim se**

Os detalhes do *kernel atualize Lbest* são exibidos no Algoritmo 19. 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.

Algoritmo 19 kernel atualize $Lbest$

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

5.6 PDPSO

O algoritmo PDPSO, descrito em pseudocódigo CUDA , é exibido no algoritmo 20 (CALAZAN; NEDJAH; MOURELLE, 2013). Cada partícula é mapeada em um bloco e cada dimensão em uma *thread*. Assim, o tamanho do *grade* da GPU corresponde ao número de partículas n enquanto o tamanho do *bloco* corresponde ao número de dimensões do problema d. Para um enxame com 64 partículas em um problema de 16 dimensões, serão necessárias 1024 *threads* distribuídas em 64 *blocos* de 16 *threads*. Esta abordagem leva a uma granularidade mais fina do que a usada na implementação do PPSO. Consequentemente, cada *thread* terá uma reduzida quantidade de código, registradores e tempo de execução. Assim como o PPSO, o PDPSO realiza a transferência de $Best$ da GPU para a CPU somente a cada 20 iterações de forma a reduzir o *overhead* gerado pela transferência.

Algoritmo 20 PDPSO implementado em CUDA

1: **seja** n = número de blocos (partículas)
2: **seja** d = número de *threads* por bloco (dimensões)
3: **kernel**$\lll n, d \ggg$ inicialize as informações das partículas
4: **repita**
5: **kernel**$\lll n, d \ggg$ calcule a velocidade e posição
6: **kernel**$\lll n, d \ggg$ calcule $fitness$ e $Pbest$
7: **kernel**$\lll n, d \ggg$ atualize $Lbest$
8: **transfira** $Best$ para a CPU
9: **até** condição de parada
10: **retorne** o resultado e a posição correspondente

A implementação do PDPSO também faz uso de quatro *kernels*, devido a necessidade de sincronismo entre *threads* de blocos diferentes. O primeiro gera $n \times d$ geradores de números aleatórios, ou seja, um para cada dimensão da partícula e inicializa as informações básicas desta, tais como velocidade e posição. Então, o segundo *kernel* gera $n \times d$ *threads* que serão responsáveis pelo cálculo das próximas velocidades e posições das partículas, conforme descrito no Algoritmo 21. Na sequencia, o terceiro *kernel* também gera $n \times d$ *threads*, que cuidarão do cálculo da função objetivo de acordo com a respectiva dimensão, realizando o processo de redução para obter o valor de $fitness$ da partícula representada pelo bloco. Após o processo de redução, verifica se $Pbest$ precisa ser atualizada. Se este for o caso, as *threads* deste *kernel* atualizam as coordenadas $Pbestx$ diretamente.

O processo de redução de $fitness$ não é realizado sequencialmente. Ao invés disso, a primeira metade das *threads* ($1 \leq t \leq d/2$) atuam simultaneamente, enquanto a segunda

Algoritmo 21 kernel calcule a velocidade e posição

1: **seja** $tid = threadIdx + blockIdx \times blockDim$
2: **calcule** $v[tid]$
3: **calcule** $x[tid]$

metade fica aguardando na barreira de sincronismo. A *thread* t reduz o valor de $fitness$ com respeito a dimensão correspondente e com o valor obtido pela *thread* $t + d/2$. Então, apenas um quarto das *threads* ($1 \leq t \leq d/4$) atuam simultaneamente enquanto os três quartos restantes ficam esperando na barreira. Nesta iteração, a *thread* t reduz o valor de $fitness$ obtido pelo processo de redução da *thread* k e $t+d/2$ e que foi obtido pelo processo de redução entre a *thread* $t + d/4$ e $t + 3d/4$. Este processo é repetido até que exista um único valor correspondente ao $fitness$ da partícula. O pseudocódigo CUDA para o *kernel calculo de fitness e Pbest* é apresentado no Algoritmo 22, onde os detalhes deste processo de redução estão incluídos. Note que as variáveis x, $Pbest$, $Pbestx$, $Lbest$ e $Lbestx$ são armazenadas na memória global. A memória compartilhada da GPU é simbolizada por *cache*. A utilização da memória compartilhada aumenta o desempenho do cálculo de $fitness$ e da operação de redução. Na Tabela 1 é apresentado um exemplo do processo de redução.

Algoritmo 22 kernel calcule $fitness$ e $Pbest$

1: **seja** k = blockIdx; t = threadIdx; m = blockDim
2: **seja** $tid = t + k \times m$
3: **seja** $cache[t] := x[tid]$
4: $cache[t] := fitness$ com respeito a dimensão t
5: $\ell := d/2$
6: **enquanto** $(\ell \neq 0)$ **faça**
7: **se** $(t < \ell)$ **então**
8: **reduza** ($cache[t]$, $cache[t + \ell]$) com respeito a função objetivo
9: **fim se**
10: **sincronize** *threads*
11: $\ell := \ell/2$
12: **fim enquanto**
13: **se** $(cache[0] < Pbest[k])$ **então**
14: $Pbestx[tid] := x[tid]$
15: **se** $t = 0$ **então**
16: $Pbest[k] := cache[0]$
17: **fim se**
18: **fim se**

Os detalhes do *kernel atualiza Lbest* são exibidos no algoritmo 23. Os vizinhos da partícula tid são as partículas $(tid+1)$ mod n e $(tid-1)$ mod n. Assim, a melhor posição

Tabela 1: Exemplo do conteúdo de *cache* durante o processo de redução para $f_1 = \sum_{t=1}^{8} x_t^2$

t	1	2	3	4	5	6	7	8
x_t	1	3	-1	2	5	5	-3	2
x_t^2	1	9	1	4	25	25	9	4
$\ell = 4$	26	34	10	8	25	25	9	4
$\ell = 2$	36	42	10	8	25	25	9	4
$\ell = 1$	**78**	42	10	8	25	25	9	4

Algoritmo 23 kernel atualize *Lbest*

```
1: seja b = blockIdx; t = threadIdx; d = blockDim
2: seja tid = t + bd
3: se (Pbest[(b + 1) mod n] < Lbest[b]) então
4:    Lbestx[tid] := Pbestx[(tid + d) mod nd]
5:    se t = 0 então
6:       Lbest[b] := Pbest[(b + 1) mod n]
7:    fim se
8: fim se
9: se (Pbest[(b − 1) mod n] < Lbest[b]) então
10:   Lbestx[tid] := Pbestx[(tid − d) mod nd]
11:   se t = 0 então
12:      Lbest[b] := Pbest[(b − 1) mod n]
13:   fim se
14: fim se
15: se (b = 0) então
16:   Best := Lbest
17:   calcule Best()
18: fim se
```

local é obtida baseada nesta definição de vizinhança. Em vez de realizar a atualização do menor resultado do enxame (*Best*) sequencialmente, usando uma única *thread*, a atualização é realizada em paralelo. Na primeira iteração são utilizadas d *threads*, depois $d/2$ e assim por diante, até eleger o melhor valor de *Lbest* obtido pelas n partículas do enxame. Os detalhes deste processo estão descritos no Algoritmo 23. Note que, esta operação é apenas realizada pelas *threads* de um bloco: o bloco de número 0, porque o mecanismo de sincronização funciona apenas entre as *threads* de um mesmo bloco. Um exemplo da eleição de *Best* para um enxame de 8 partículas é mostrado na Tabela 2. No final da iteração, *Best* pode ser enviado para a CPU e utilizado durante a verificação da condição de parada.

Algoritmo 24 Procedimento *calcule Best*

```
seja t = threadIdx; d = blockDim
ℓ := d;
enquanto (ℓ ≠ 0) faça
  se (t < ℓ) então
    para k = 0 → n/2d − 1 faça
      se Bbest[t + ℓk] < Bbest[t + ℓk + 2l] então
        Best[t + ℓk] := Bbest[t + ℓk]
        atualize Bestx
      senão
        Best[t + ℓk] := Bbest[t + ℓk + 2l]
        atualize Bestx
      fim se
    fim para
  fim se
  sincronize threads
  ℓ := ℓ/2;
fim enquanto
se t = 0 então
  se Bbest[1] < Bbest[0] então
    Best[0] := Bbest[1]; atualize Bestx
  fim se
fim se
```

Tabela 2: Exemplo do conteúdo de *Best* durante o processo de comparação

$b = blockIdx$	0	1	2	3	4	5	6	7
$Best[b]$	0,5	1,5	10	2,3	0,7	0,0	0,1	2,2
$\ell = 2, k = 0$	0.5	0,0	10	2,3	0,7	0,0	0,1	2,2
$\ell = 2, k = 1$	0,5	0,0	0,1	2,2	–	–	–	–
$\ell = 1, k = 0$	0,1	0,0	0,1	2,2	–	–	–	–
$\ell = 1, k = 1$	0,1	0,0	0,1	2,2	–	–	–	–
$\ell = 0$	**0,0**	0,0	0,1	2,2	–	–	–	–

5.7 CPPSO

O algoritmo CPPSO, descrito em pseudocódigo CUDA, é exibido no algoritmo 25. Conforme prevê o algoritmo, o problema original com d dimensões é dividido em k subproblemas que se dedicam a um subgrupo de dimensões $e = d/k$. Da mesma forma, o enxame original composto de n partículas é divido em k subenxames, cada um de $t = n/k$ partículas, onde cada subenxame otimiza um dos subproblemas. O subenxame é mapeado em um *bloco* e a partícula em uma *thread*. Assim, o tamanho da *grade* da GPU corresponde ao número de subenxames k enquanto o tamanho do *bloco* corresponde ao número de partículas t de cada enxame. A implementação do CPPSO também adota a solução *mul-*

tikernel devido a necessidade de sincronismo entre instruções de *blocos* diferentes. Assim como nas outras implementações em CUDA, a instrução que transfere $Best$ para a CPU (linha 8 do Algoritmo 25) é executada a cada 20 iterações de forma a diminuir o *overhead* de comunicação entre a CPU e a GPU.

Algoritmo 25 CPPSO implementado em CUDA

1: **seja** k = número de *blocos* (enxames)
2: **seja** t = número de *threads* por *bloco* (partículas por enxame)
3: **kernel**$\lll k, t \ggg$ inicialize as informações das partículas
4: **repita**
5: **kernel**$\lll k, t \ggg$ calcule a velocidade e posição
6: **kernel**$\lll k, t \ggg$ calcule $fitness$ e atualize vetor de contexto
7: **kernel**$\lll k, t \ggg$ atualize $Lbest$
8: **transfira** $Best$ para a CPU
9: **até** condição de parada
10: **retorne** o resultado e a posição correspondente

O primeiro *kernel*, cujo código é descrito no algoritmo 26, cria e inicializa na memória global, $k \times t$ geradores de números aleatórios, sendo um para cada *thread*. Os geradores são baseados na biblioteca *curand_kernel.h*, conforme explicado na seção 5.3. Em seguida, o *kernel* inicializa a velocidade e posição da partícula, bem como o vetor de contexto $Bestx$ que é de tamanho d. Assim, a *thread* com o índice $threadIdx = 0$ de cada subenxame armazena a coordenada referente a sua partícula no vetor de contexto, na posição relativa ao índice do subenxame.

Algoritmo 26 kernel inicialize informações das partículas

1: **seja** b = blockIdx; t = threadIdx; m = blockDim;
2: **faça** $tid := t + b \times m$
3: **para** $j = 0 \rightarrow e - 1$ **faça**
4: $x[tid \times e + j] := rand(max - min) + min$
5: $v[tid \times e + j] := 0.0$
6: **se** $(t = 0)$ **então**
7: $Bestx[b \times e + j] := x[b \times m \times e + j]$
8: **fim se**
9: **fim para**

O segundo *kernel*, cujo código é exibido no Algoritmo 27, cria $k \times t$ *threads*, que irão realizar o calculo das próximas velocidade e posição para cada partícula de cada subenxame em paralelo. Note que cada *thread* executa um laço, com o número de iterações e, para atualizar os valores de cada dimensão da partícula tid.

Algoritmo 27 kernel calcule a velocidade e posição

1: **faça** $tid := threadIdx + blockIdx \times blockDim$
2: **para** $j = 0 \rightarrow e - 1$ **faça**
3: **calcule** $v[tid \times e + j]$
4: **calcule** $x[tid \times e + j]$
5: **fim para**;

O terceiro *kernel*, cujo código é exibido no Algoritmo 28, também gera $k \times t$ *threads*, que irão executar o calculo de $fitness$, juntamente com a atualização de *Pbest* da partícula. Cada *thread* realiza a cópia de todo o vetor de contexto, armazenado na memória global, para uma área de memória privada. Em seguida, as *threads* substituem, na posição do vetor de contexto referente ao seu subenxame, a coordenada relativa a sua partícula. Desta forma, o calculo de $fitness$ é executado levando em conta todas as dimensões do problema, bem como a atualização de *Pbest* e *Lbest*. A atualização de *Lbest* realizada neste *kernel* é referente ao valor de *Pbest* da própria partícula. Neste *kernel* não é realizada uma comparação com a vizinhança.

Algoritmo 28 *kernel* calcule $fitness$ e atualize vetor de contexto

1: **seja** b = blockIdx; t = threadIdx; m = blockDim;
2: **seja** $Tbestx$ o vetor de contexto de d posições armazenado na memória privada
3: **faça** $tid := t + b \times m$
4: **para** $j := 0 \rightarrow d - 1$ **faça**
5: $Tbestx[j] := Bestx[j]$
6: **fim para**
7: **para** $j := 0 \rightarrow e - 1$ **faça**
8: $Tbestx[b \times e + j] := x[tid \times e + j]$
9: **fim para**
10: **para** $j := 0 \rightarrow d - 1$ **faça**
11: **calcule** $fitness(Tbestx[j])$
12: **fim para**
13: **Atualize** *Pbest*
14: **Atualize** $Lbest_{tid}$
15: **se** $(t = 0)$ **então**
16: **selecione** a menor *Lbest* do subenxame
17: $Best[b]$:=$Lbest[b \times m + t]$
18: **para** $j := 0 \rightarrow e - 1$ **faça**
19: $Bestx[b \times e] := Lbestx[(b \times m + t) \times e + j]$
20: **fim para**;
21: **fim se**

A *thread* com índice 0 de cada subenxame atualiza o vetor *Best*, na posição referente ao seu subenxame, com o menor valor de *Lbest* obtido entre as partículas. Em se-

guida, a mesma *thread* atualiza o vetor de contexto, também na posição referente ao índice do respectivo subenxame, com as coordenadas de *Lbest*. O Algoritmo 29 exibe os detalhes do *kernel atualize Lbest*. Os vizinhos da partícula *tid* são as partículas $(tid + 1) \bmod n$ e $(tid - 1) \bmod n$. Assim, a atualização da melhor posição local é realizada de acordo com a sua vizinhança. Esta troca de informações do melhor local acontece apenas dentro de cada subenxame. Não há troca de informações entre os subenxames. Como última operação do *kernel*, a *thread* com $tid = 0$ executa uma busca sequencial no vetor *Best* de forma a obter o menor valor e posição referente ao processo de otimização de cada subenxame.

Algoritmo 29 kernel atualize *Lbest*

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

5.8 Considerações Finais do Capítulo

Neste capítulo foi apresentado o modelo de programação CUDA e a arquitetura *multithreading* da GPU. Pode ser observado que a GPU é capaz de gerenciar, escalonar e executar milhares de *threads* simultaneamente. Foram descritas as implementações dos algoritmos propostos PPSO, PDPSO e CPPSO na plataforma CUDA, assim como os *kernels* que permitem a implementação paralela dos algoritmos. O capítulo a seguir apresenta a análise dos resultados obtidos pelas implementações nas arquiteturas de *hardware*, memória compartilhada, memória distribuída e GPU.

Capítulo 6

ANÁLISE DOS RESULTADOS

NESTE capítulo são analisados os resultados obtidos pelas implementações descritas nos Capítulos 3, 4 e 5. Diferentes arranjos de partículas e dimensões são utilizados para otimizar 4 funções clássicas consideradas como *benchmark*. A análise permite a asserção do desempenho dos algoritmos propostos, bem como suas eficiências na otimização.

A Seção 6.1 apresenta a descrição das funções utilizadas para avaliar a arquitetura proposta em *hardware* assim como as implementações em *software*. Na Seção 6.2 são apresentados os resultados obtidos pelo coprocessador PSO e realizada sua comparação com o processador MicroBlaze.

A Seção 6.3 apresenta a metodologia utilizada para realizar os testes dos algoritmos propostos nas plataformas de memória compartilhada, memória distribuída e GPU. Nas Seções 6.4, 6.5, 6.6 e 6.7 são apresentados os resultados dos algoritmos implementados em OpenMP, MPICH, OpenMP com MPICH e CUDA respectivamente. Na Seção 6.8 é realizada uma comparação entre as arquiteturas.

6.1 Descrição das Funções Utilizadas

Com o objetivo de verificar a eficiência e o desempenho das implementações propostas, quatro funções clássicas de *benchmark* foram otimizadas em *hardware* e *software*. Em seguida, é realizada a descrição das funções:

6.1.1 Função *Esfera*

A função *Esfera* é contínua, convexa e unimodal. Sua definição geral pode ser observada na Equação 9. O espaço de busca utilizado é restrito ao intervalo $-100 \leq x_i \leq 100$, $i = 1 \ldots d$. A função possui um mínimo global $f(x) = 0$ obtido no ponto de coordenadas

$x_i = 0$, $i = 1 \ldots d$. A curva proporcionada por essa função pode ser observada na Figura 23.

$$f_1(x) = \sum_{i=1}^{d} x_i^2 \tag{9}$$

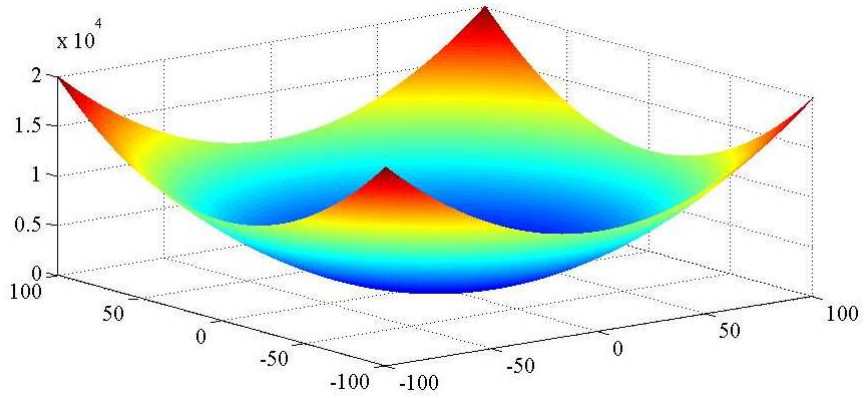


Figura 23: Curva da função *Esfera* em 2D

6.1.2 Função *Rosenbrock*

A função *Rosenbrock* é um clássico problema de otimização. O ótimo global reside em um longo e estreito vale em forma de parábola. Encontrar o vale é fácil, porém a convergência para o ótimo global é difícil (MOLGA; SMUTNICKI, 2005). A função é unimodal em até 2 dimensões e em maiores dimensões passa a ser multimodal (SHANG; QIU, 2006). Sua definição geral pode ser observada na Equação 10. O espaço de busca utilizado é restrito ao intervalo $-2,048 \leq x_i \leq 2,048$, $i = 1 \ldots d$. Essa função possui mínimo global em $f(x) = 0$ obtido no ponto de coordenadas $x_i = 0$, $i = 1 \ldots d$. A curva dessa função pode ser observada na Figura 24.

$$f_2(x) = \sum_{i=1}^{d} (100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2) \tag{10}$$

6.1.3 Função *Rastrigin*

A função *Rastrigin* é bastante complexa devido aos vários mínimos locais regularmente distribuídos. Sua definição geral pode ser observada na Equação 11. O espaço de busca utilizado é restrito ao intervalo $-5,12 \leq x_i \leq 5,12$, $i = 1 \ldots d$. Essa função possui um

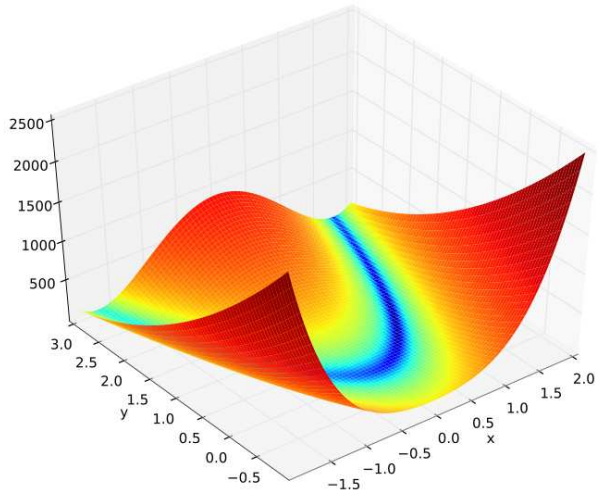


Figura 24: Curva da função *Rosenbrock* em 2D

mínimo global $f(x) = 0$ obtido no ponto de coordenadas $x_i = 0$, $i = 1 \ldots d$. A curva dessa função pode ser observada na Figura 25.

$$f_3(x) = \sum_{i=1}^{d} (x_i^2 - 10cos(2\pi x_i) + 10) \tag{11}$$

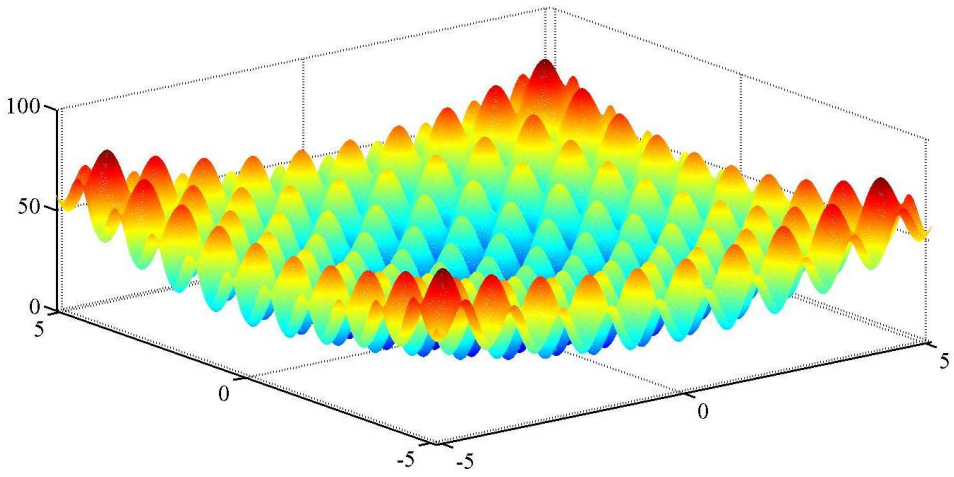


Figura 25: Curva da função *Rastrigin* em 2D

6.1.4 Função *Schwefel*

Na função *Schwefel* a solução global encontra-se geometricamente distante das soluções locais. Desta forma, os algoritmos de otimização são potencialmente propensos pela busca na direção incorreta (MOLGA; SMUTNICKI, 2005).

Sua definição geral pode ser observada na Equação 12. O espaço de busca utilizado é restrito ao intervalo $-500 \leq x_i \leq 500$, $i = 1, ...d$. Essa função possui mínimo global $f(x) = -418,9829$ obtido no ponto de coordenadas $x_i = 420,9687$, $i = 1, ..., d$. A curva dessa função pode ser observada na Figura 26.

$$f_4(x) = 418,9829 - \sum_{i=1}^{d}(x_i sin\sqrt{|x_i|}) \tag{12}$$

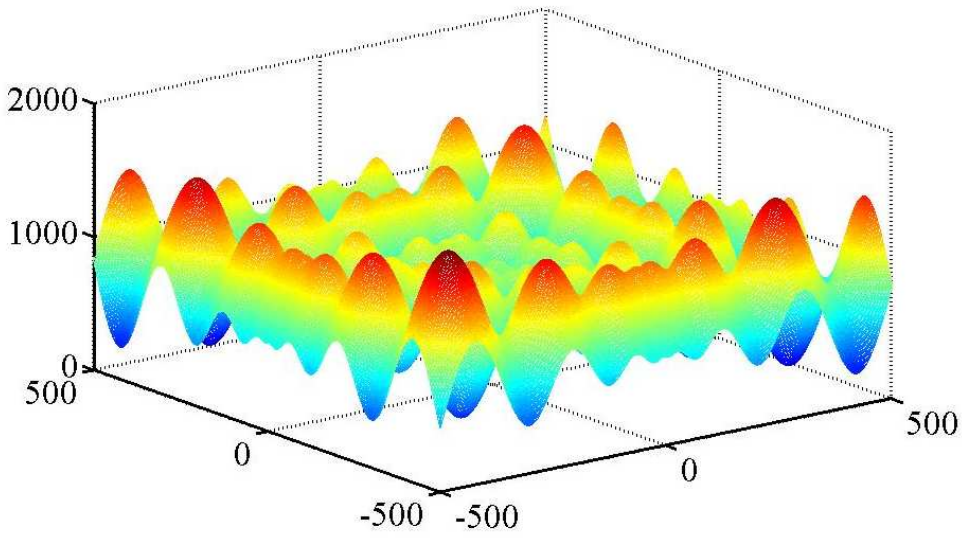
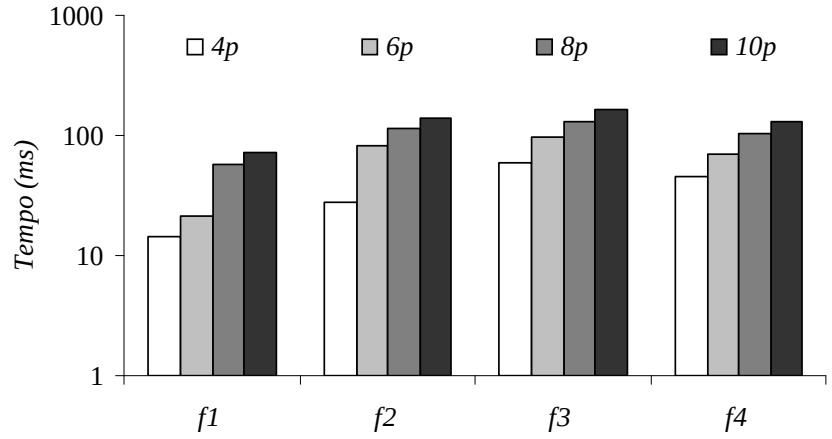


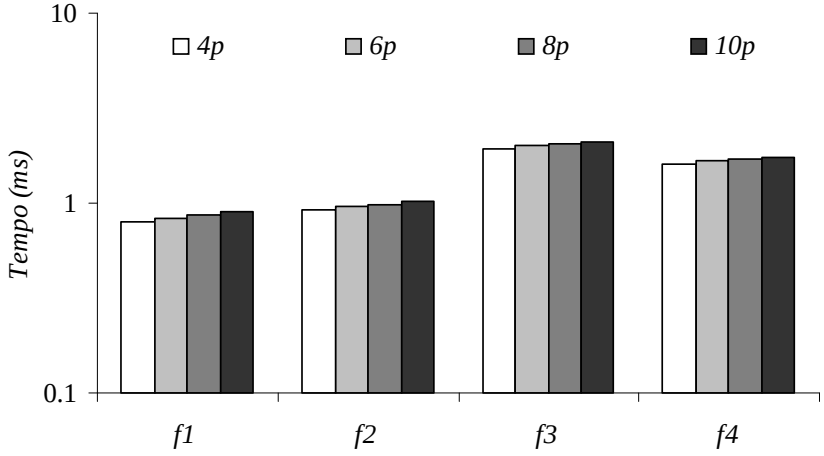
Figura 26: Curva da função *Schwefel* em 2D

6.2 Resultados do Coprocessador HPSO

Com o objetivo de avaliar o desempenho do coprocessador HPSO apresentado no Capítulo 3, o algoritmo PSO é executado pelo processador MicroBlaze™ com e sem o auxílio do coprocessador. Sem o coprocessador HPSO, o processador MicroBlaze™ realiza toda a computação. Neste caso, o algoritmo PSO juntamente com as funções objetivos é implementado em ANSI/C++. O desempenho do coprocessador HPSO é avaliado variando o número de partículas de 4 a 10 e utilizando as funções descritas na Seção 6.1 com 2 dimensões e o número de iterações fixado em 200 para todos os casos. Os resultados de tempo de execução e desempenho obtidos pelo processador MicroBlaze™, na execução do algoritmo PSO com e sem o auxilio do coprocessador, são exibidos nas Figuras 27 e 28 respectivamente. Os dados numéricos estão disponíveis na Tabela 4 do Apêndice.



(a) MicroBlaze



(b) Coprocessador

Figura 27: Tempo de execução: MicroBlaze × Coprocessador HPSO

A Figura 28 exibe o *speedup* obtido pelo coprocessador HPSO quando comparado com a implementação no processador MicroBlaze™. Pode ser observado que o desempenho do HPSO, especificamente em uma Xilinx Virtex 6 FPGA (xcvlx75t)(XILINX, 2010), operando a 200 MHz, é 18 vezes maior no pior caso e 135 vezes no melhor caso, que certamente representa uma grande melhoria. Pode ser observado ainda que o acréscimo de partículas na implementação no processador MicroBlaze™ tem um impacto no aumento da média do tempo de execução de 34%, enquanto que para o coprocessador HPSO a média fica em torno de 3%.

A Figura 29 apresenta os recursos de *hardware* necessários, obtidos pela ferramenta de síntese da Xilinx para o processador MicroBlaze™ usando um enxame de 4 a 10 partí-

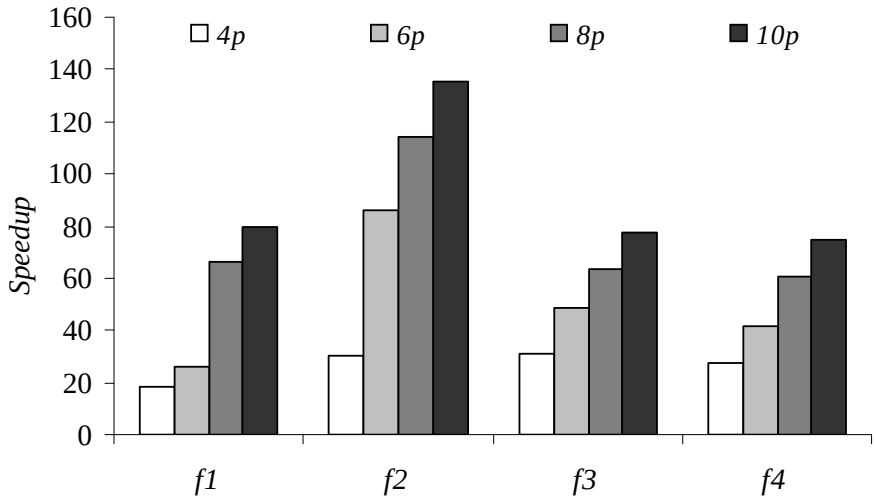
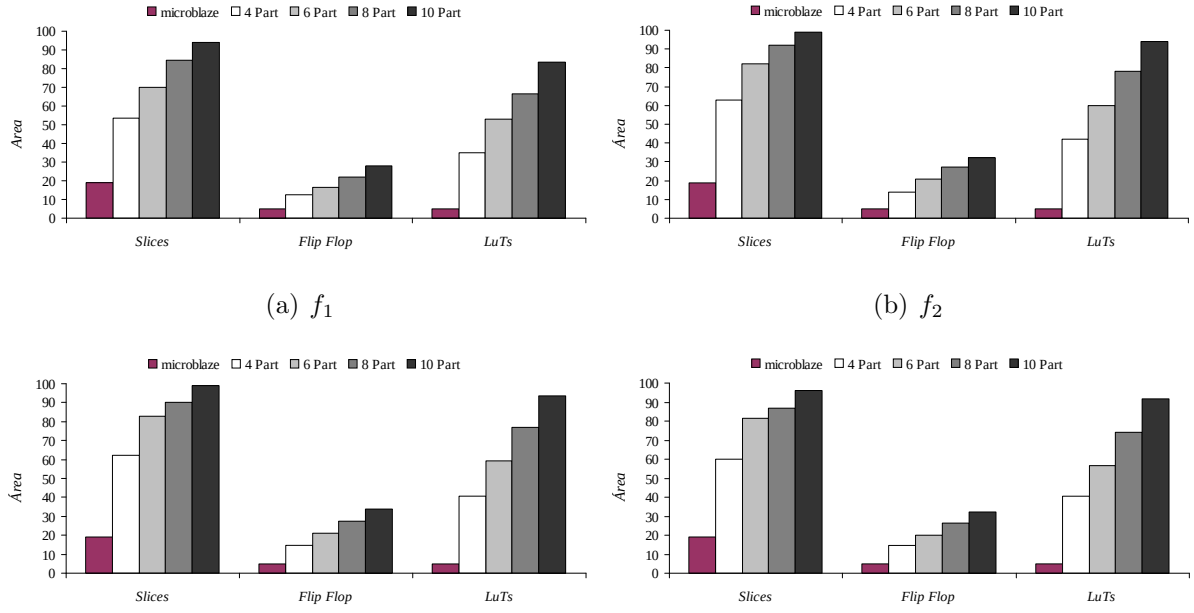


Figura 28: *Speedup*: MicroBlaze × Coprocessador HPSO

culas em uma arquitetura de 32 bits na otimização de funções. A Tabela 5 do Apêndice apresenta os valores numéricos referente a Figura 29.



(a) f_1 (b) f_2

(c) f_3 (d) f_4

Figura 29: Utilização de recursos de *hardware*: MicroBlaze × Coprocessador HPSO

Os resultados mostram os requerimentos de *hardware* para o MicroBlaze™ e para o MicroBlaze™ juntamente com o coprocessador HPSO. Nesta comparação pode ser ob-

servado que o aumenta do número de partículas resultou em um aumento considerável de utilização de área da FPGA. Desta forma, a escalabilidade da arquitetura é limitada a 10 partículas, possivelmente pela unidade de cálculo em ponto flutuante que resultou em um grande consumo de área por partícula. Ainda assim, o aumento no desempenho justifica o aumento de área devido ao número de partículas. Utilizando uma unidade FPU otimizada, possivelmente reduziria esta limitação.

6.3 Metodologia de Avaliação

As implementações em *software* foram avaliadas utilizando diferentes arranjos de número de partículas e quantidade de dimensões, conforme exibidos na Tabela 3. Estes diferentes arranjos de partículas e dimensões, que serão citados como (partículas, dimensões), fornecem casos com diferentes custos computacionais e complexidade do problema. A última linha da Tabela 3 exibe o produto entre partículas e dimensões referente a cada caso. Para a realização das avaliações foi utilizado um coeficiente de inércia iniciando em $0,99$, com um decrescimento linear até $0,2$ e $vmax$ sendo calculado utilizando um δ de $0,2$. 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 6000 iterações for atingido.

Tabela 3: Arranjos de partículas e dimensões utilizados nas avaliações

Casos	1	2	3	4	5	6	7	8
p	8	16	32	64	128	256	512	1024
d	2	4	8	16	32	64	128	256
$p \times d$	16	64	256	1024	4096	16384	65536	262144

No Capítulo 1 foi mostrado que a eficiência do algoritmo PSO é expressa pelo número de iterações necessárias para encontrar uma solução com uma determinada acurácia. Assim, o número de iterações expressa o tempo relativo para o algoritmo alcançar uma solução ótima. Desta forma, podemos calcular o tempo por iteração realizando a divisão do tempo de execução pelo tempo relativo. Utilizando a razão do tempo por iteração do algoritmo serial pelo tempo por iteração do algoritmo paralelo para calcular o *speedup*, iremos obter o desempenho referente ao ganho com a implementação paralela sem levar em conta a eficiência do algoritmo na otimização. Este *speedup* será denominado de *speedup relativo*. O *speedup* calculado pela razão do tempo de execução da implementação serial

e tempo de execução da implementação paralela, referente ao número total de iterações, será denominado de *speedup real*.

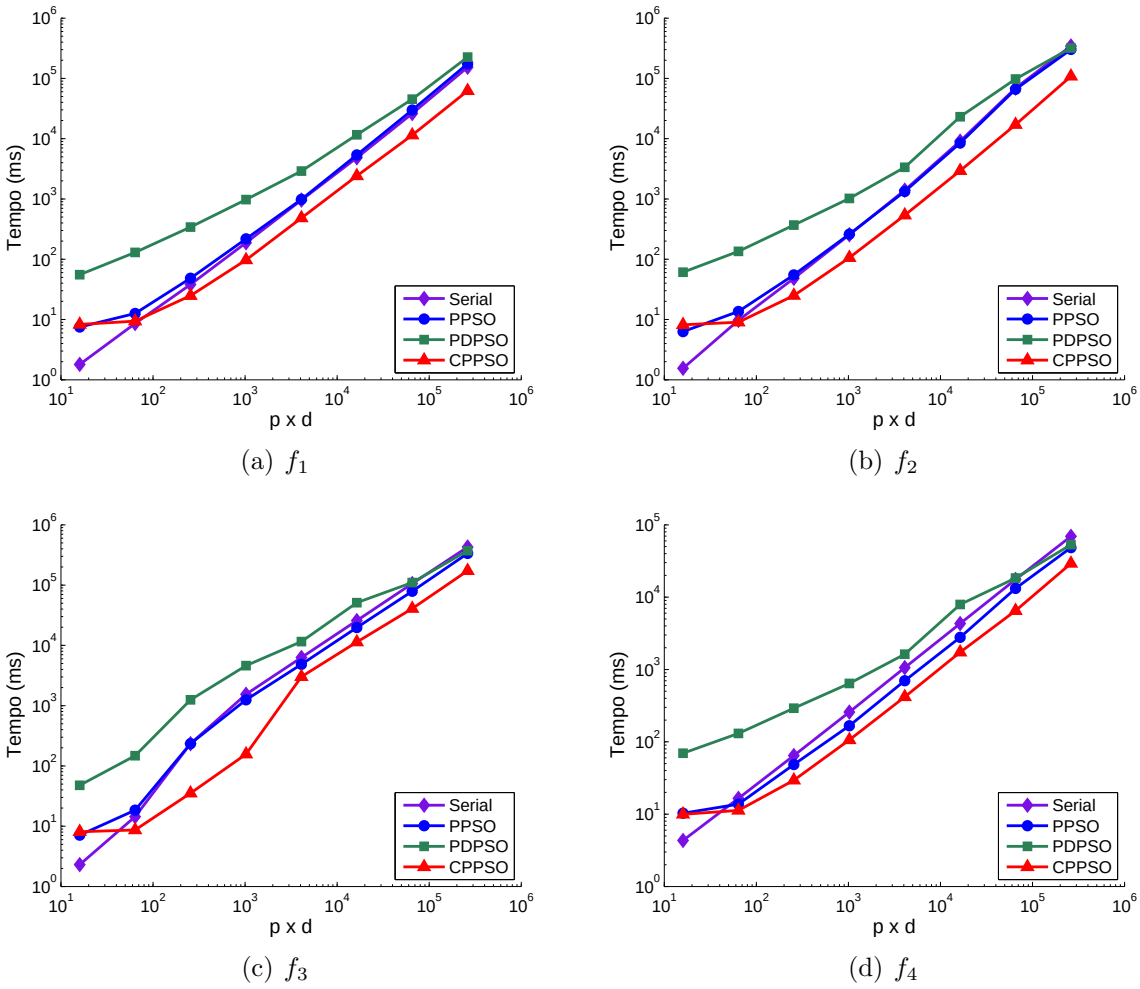
Cada avaliação foi executada 50 vezes com diferentes valores de sementes, para geração dos números aleatórios usados pelo algoritmo, de forma a avaliar o desempenho, a eficiência e a confiabilidade do algoritmo PSO. Desta forma, são apresentados no Apêndice os valores numéricos, referente a média das 50 execuções, do número de iterações, $fitness$ e tempo de execução, representado em milissegundo. Assim, são obtidos os gráficos de *tempo de execução*, *speedup relativos*, *speedup reais* e *número de iterações*, cada um deles em relação à quantidade de partículas e dimensões.

O *hardware* utilizado para executar as implementações em OpenMP e MPICH é o SGI Octane III composto de 4 nós conectados via Gigabit-Ethernet. Cada nó contêm 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, instalada em um computador com uma CPU intel core i7 de 3,07 GHz, 8 GB de RAM e SO Windows 7 de 64 bits, utilizando a linguagem CUDA versão 3.2. A implementação serial foi executada no SGI Octane III utilizando somente um núcleo de processamento. Os valores numéricos referentes à implementação serial são exibidos na Tabela 6 do Apêndice.

6.4 Resultados da Implementação em OpenMP

As implementações em OpenMP foram executadas utilizando apenas um nó do SGI Octane III, que possui 16 núcleos de processamento. Foi utilizado o compilador gcc, versão 4.3-62.198, com a opção *-fopenmp* para habilitar o suporte ao OpenMP. As Figuras 30, 31 e 32 apresentam os resultados obtidos pelos algoritmos propostos, implementados em OpenMP (Os valores numéricos referentes a estes gráficos podem ser consultados nas Tabelas 7 e 8 do Apêndice).

As avaliações das implementações dos algoritmos PPSO e CPPSO em OpenMP foram executadas com 2 *threads*. Nos testes realizados com 4 *threads* e 8 *threads* foi obtido um aumento médio no tempo de execução de 2,5 vezes em relação ao tempo realizado com apenas 2 *threads*. Adicionalmente, a implementação do algoritmo PDPSO utilizou 4 *threads* para paralelizar o laço de dimensões da região paralela aninhada, conforme Algoritmos 6 e 7 do Capítulo 4.



(a) f_1 (b) f_2

(c) f_3 (d) f_4

Figura 30: Tempo de execução das implementações paralelas em OpenMP

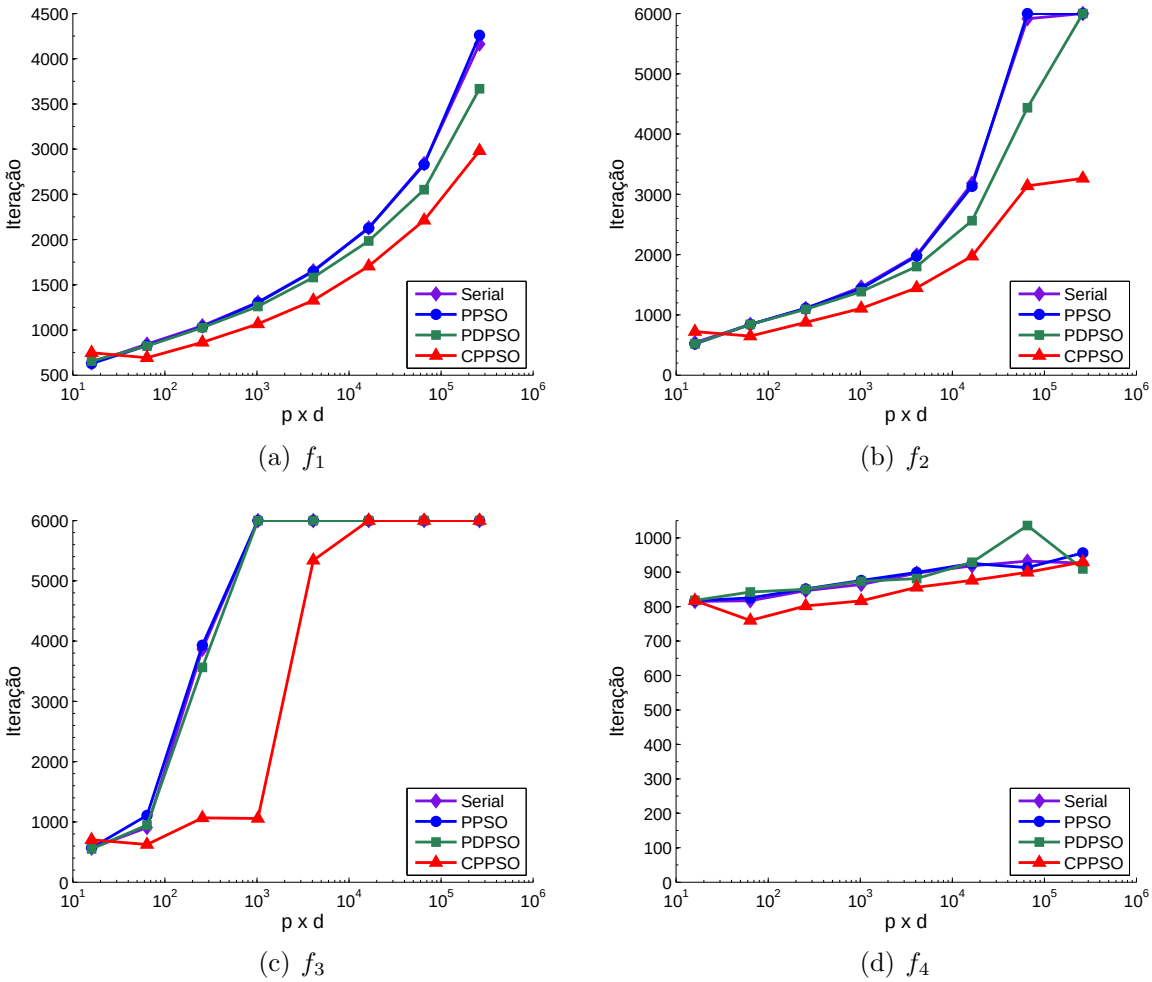
De acordo com os resultados obtidos, pode ser observado que o tempo de execução aumenta de acordo com o número de partículas e quantidade de dimensões. No caso (8, 2), a implementação serial obteve um menor tempo de execução em relação às implementações paralelas. A partir deste teste, o CPPSO começou a obter tempos de execução menores em todas as funções, seguido do PPSO. O PDPSO somente obteve um tempo de execução menor que a implementação serial no caso (1024, 256) nas funções f_2, f_3 e f_4.

Em relação à eficiência na otimização, o algoritmo CPPSO atingiu a acurácia necessária com um menor número de iterações na maioria dos casos e para todas as funções. Os algoritmos PPSO e PDPSO tiveram comportamentos semelhantes à implementação serial, exceto nas funções f_1 e f_2, onde o PDPSO realizou um número menor de iterações.

O CPPSO obteve também os melhores valores de *speedup real* e *speedup relativo* em todos os casos, chegando a atingir um *speedup real* de 9,84 vezes no caso (64, 16)

para f_3 e *speedup relativo* de 2,6 nas funções f_3 e f_4 no caso (512, 128). O super *speedup* apresentado no gráfico da Figura 32(e) nos casos (32, 8) e (64, 16) é consequência do menor número de iterações realizado pelo CPPSO para alcançar a solução ótima, em relação aos demais algoritmos. Pode ser observado ainda que o *speedup* do CPPSO aumenta conforme aumenta o tamanho do enxame e o número de dimensões até o caso (512, 256). No caso (1024, 256), o *speedup relativo* diminui de acordo com a função.

O algoritmo PPSO obteve um desempenho pouco superior ao algoritmo serial, alcançando um *speedup relativo* máximo de 1,43 obtido na função f_4 no caso (1024, 256). Foi constatado ainda que nas funções de maior custo computacional o desempenho das implementações paralelas foi ligeiramente melhor.

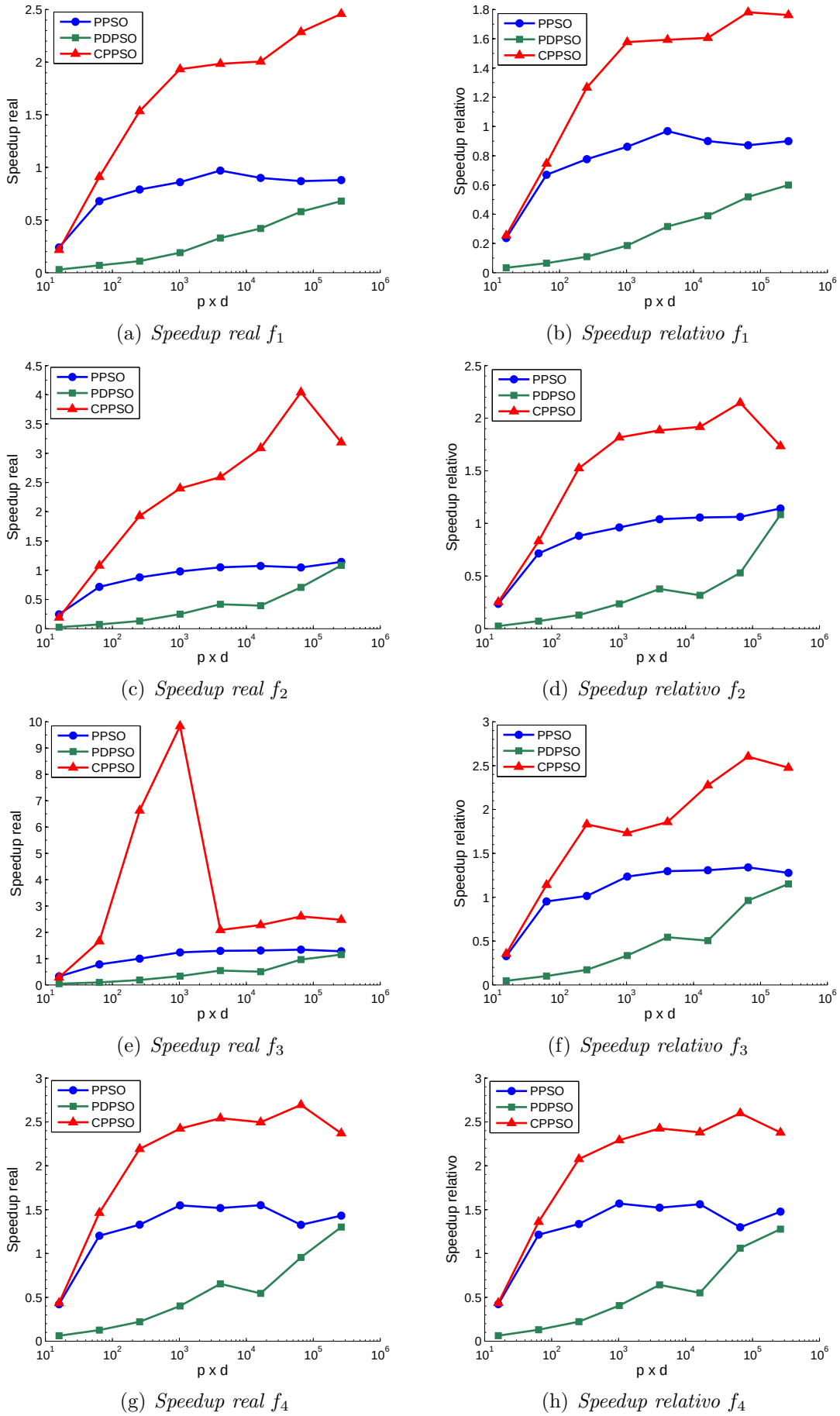


(a) f_1 (b) f_2

(c) f_3 (d) f_4

Figura 31: Número de iterações das implementações paralelas em OpenMP

O desempenho do algoritmo CPPSO pode ser explicado considerando dois aspectos. O primeiro pelo mapeamento de um subenxame por *thread* o que implica em



(a) *Speedup real* f_1

(b) *Speedup relativo* f_1

(c) *Speedup real* f_2

(d) *Speedup relativo* f_2

(e) *Speedup real* f_3

(f) *Speedup relativo* f_3

(g) *Speedup real* f_4

(h) *Speedup relativo* f_4

Figura 32: *Speedup real e relativo* das implementações paralelas em OpenMP

armazenar apenas o vetor de contexto e o melhor valor obtido por cada enxame na memória compartilhada. Todas as outras variáveis são armazenadas na memória local, que possui menor latência de acesso. O segundo motivo é devido às operações de atualização de *Pbest*, *Lbest*, velocidade e posição ocorrerem em um número menor de dimensões em relação aos outros algoritmos.

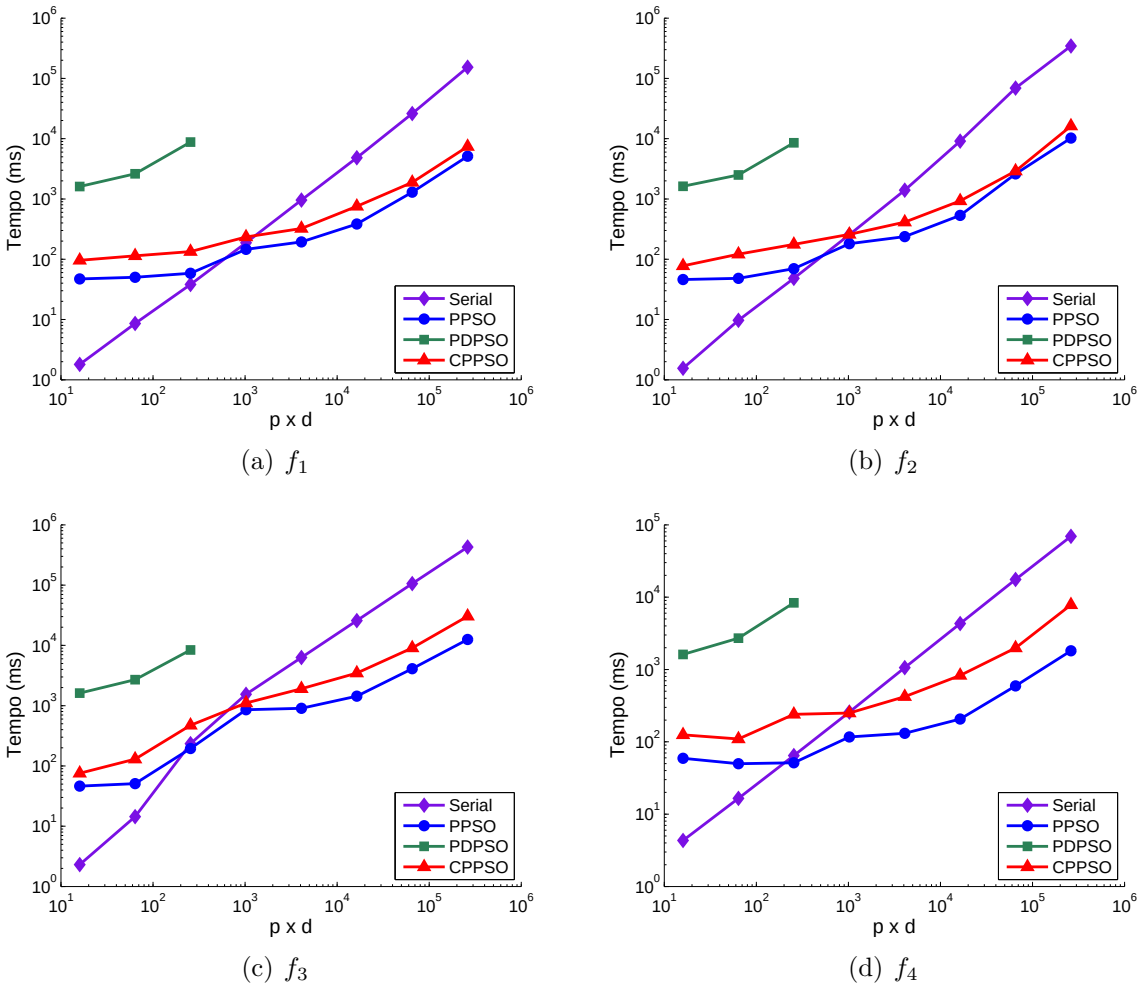
O algoritmo PDPSO obteve os piores resultados em termos de tempo de execução e desempenho. Isto pode ser explicado pela utilização do paralelismo aninhado, assim como a criação de *threads* na região paralela aninhada, gerando um *overhead* para a criação destas regiões paralelas, bem como no acesso concorrente a memória compartilhada. Ainda assim, o PDPSO manteve um aumento progressivo no *speedup* à medida que os testes aumentavam o número de partículas e a quantidade de dimensões, obtendo um *speedup relativo* de 1,3 no caso (1024, 256).

6.5 Resultados da Implementação em MPICH

As implementações em MPI foram executadas no SGI Octane III utilizando o MPICH versão 1.4.1. As Figuras 33, 34 e 35 apresentam os resultados obtidos pelos algoritmos propostos implementados em MPICH (Os valores numéricos referentes a estes gráficos podem ser consultados nas Tabelas 9 e 10 do Apêndice).

Antes de cada teste, foi realizada uma avaliação do desempenho, utilizando 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 11 do Apêndice, exibe os dados desta avaliação. Os resultados demonstram que a utilização de uma configuração mais adequada, permite alcançar um aumento no *speedup relativo* de até 4,2 vezes, referente ao caso (1024, 256), em comparação com as outras configurações. Na Tabela 11, os valores em negrito representam o melhor *speedup relativo* obtido em cada caso e então utilizado na execução das demais funções.

A mesma otimização não pôde ser aplicada ao algoritmo CPPSO, pois os resultados demonstraram que a partir de uma divisão em 8 enxames, 50% dos resultados para as sementes utilizadas não alcançavam a acurácia desejada. Dividindo em 16 enxames, 100% dos resultados não alcançaram acurácia. Assim, a divisão necessária no caso do CPPSO ficou limitada a 4 enxames, de forma a manter as características de eficiência e confiabilidade do algoritmo PSO vistas no Capítulo 1. Na implementação em MPICH o algoritmo



(a) f_1 (b) f_2

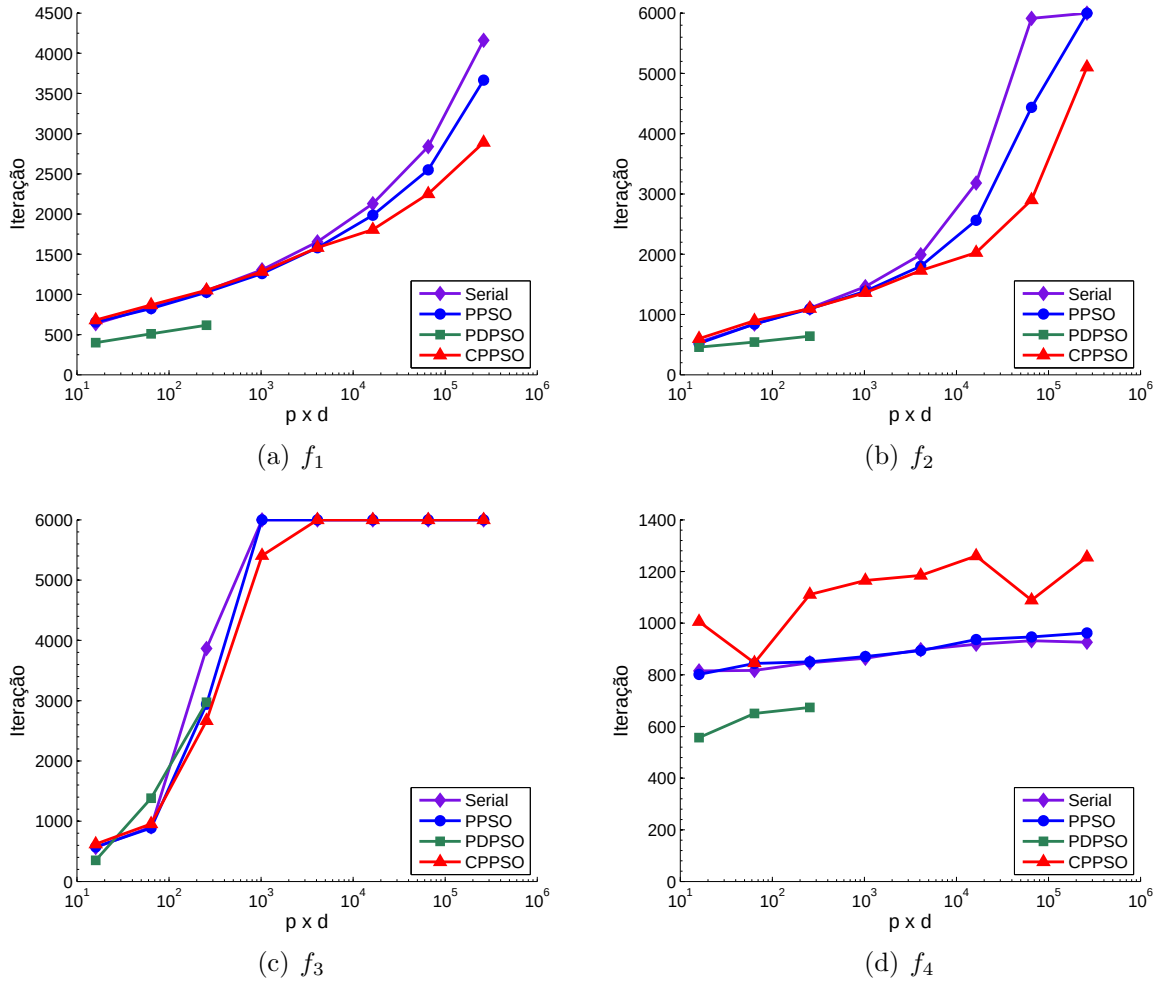
(c) f_3 (d) f_4

Figura 33: Tempo de execução das implementações paralelas em MPICH

PPSO obteve superioridade em tempo de execução, seguido do algoritmo CPPSO. Contudo, nos casos com poucas partículas e complexidade baixa, até (32, 8), o algoritmo serial foi superior as demais implementações em MPICH. Isto é devido principalmente à maior latência de comunicação do modelo de passagem de mensagem. Em relação à eficiência dos algoritmos, pode ser observado que o CPPSO apresentou melhores resultados, seguido do PPSO. Exceto no caso da função f_4, onde o algoritmo CPPSO obteve o pior resultado.

Os *speedups* do PPSO e CPPSO aumentam com o tamanho do enxame e o número de dimensões. Contudo, o algoritmo PPSO obteve os melhores resultados, atingindo um *speedup real* de 38,19 no caso (1024, 256) da função f_4, seguido do CPPSO com um valor de 21,27 no mesmo caso para a função f_2. A queda do *speedup real* no caso (1024, 256) da Figura 35 (c) é referente ao aumento do número de iterações para obter o resultado.

O desempenho do algoritmo PPSO é explicado devido à aglomeração de partículas

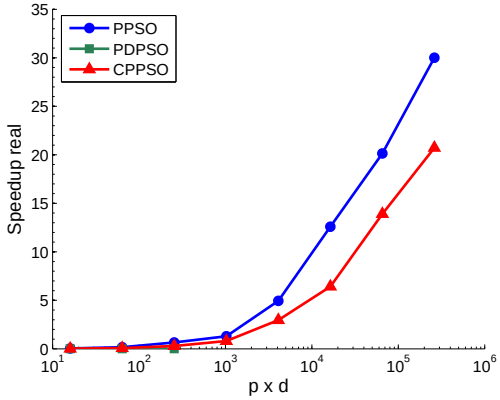


(a) f_1 (b) f_2

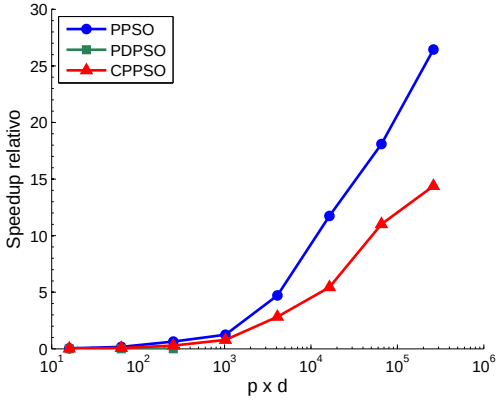
(c) f_3 (d) f_4

Figura 34: Número de iterações das implementações paralelas em MPICH

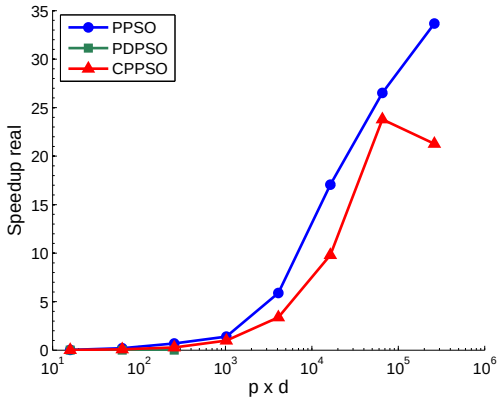
por processo, aumentando assim a granularidade do problema e diminuindo a comunicação. O algoritmo PDPSO implementado em MPICH obteve um desempenho inferior em relação aos demais algoritmos, pelo fato de que utiliza uma granularidade muito fina, mapeando cada processo a uma dimensão da partícula. Isto implica em muita comunicação e pouca computação, o que não contribui para um bom desempenho na arquitetura de memória distribuída. Além disso, a necessidade de criação de uma quantidade maior processos fez com que houvesse mais processos em execução do que processadores disponíveis, degradando o desempenho. E a partir do caso (64, 16), que cria 1024 processos paralelos, não foi possível realizar sua execução pela quantidade de processos por nó (aproximadamente 340) que atingiu o limite de escalabilidade do MPICH para versão 1.4.1 de 152 tarefas por nó.



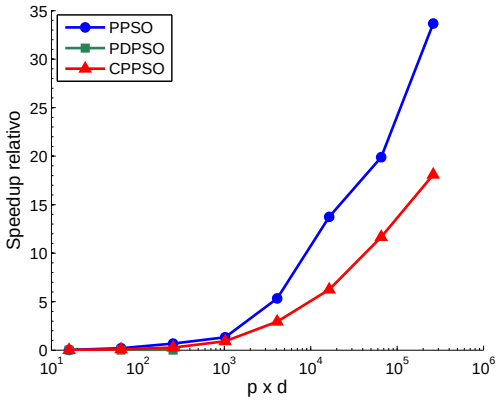
(a) *Speedup real* f_1



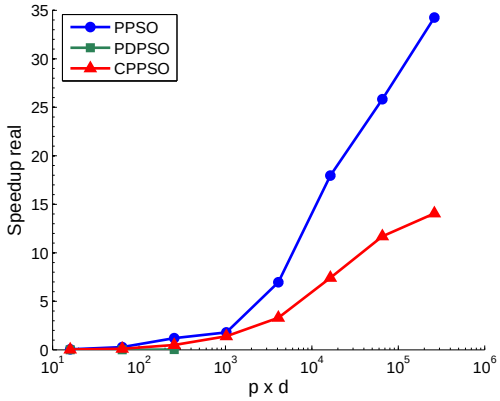
(b) *Speedup relativo* f_1



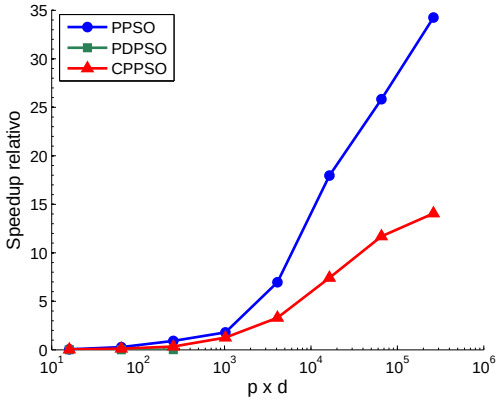
(c) *Speedup real* f_2



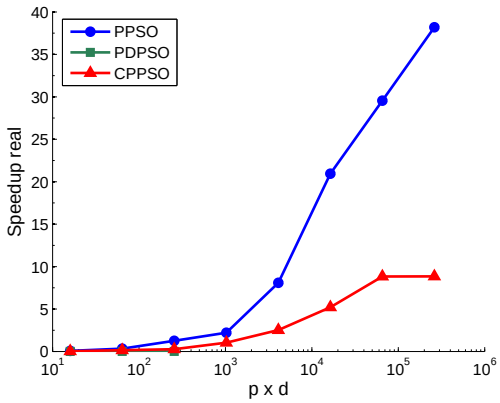
(d) *Speedup relativo* f_2



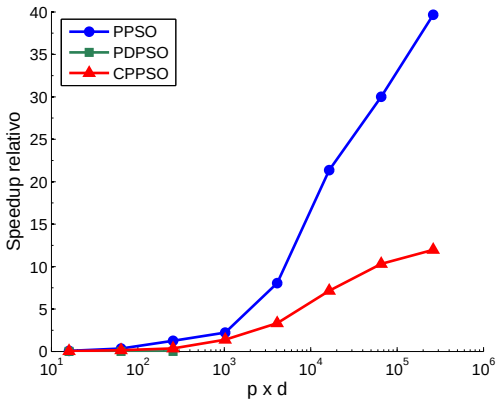
(e) *Speedup real* f_3



(f) *Speedup relativo* f_3



(g) *Speedup real* f_4

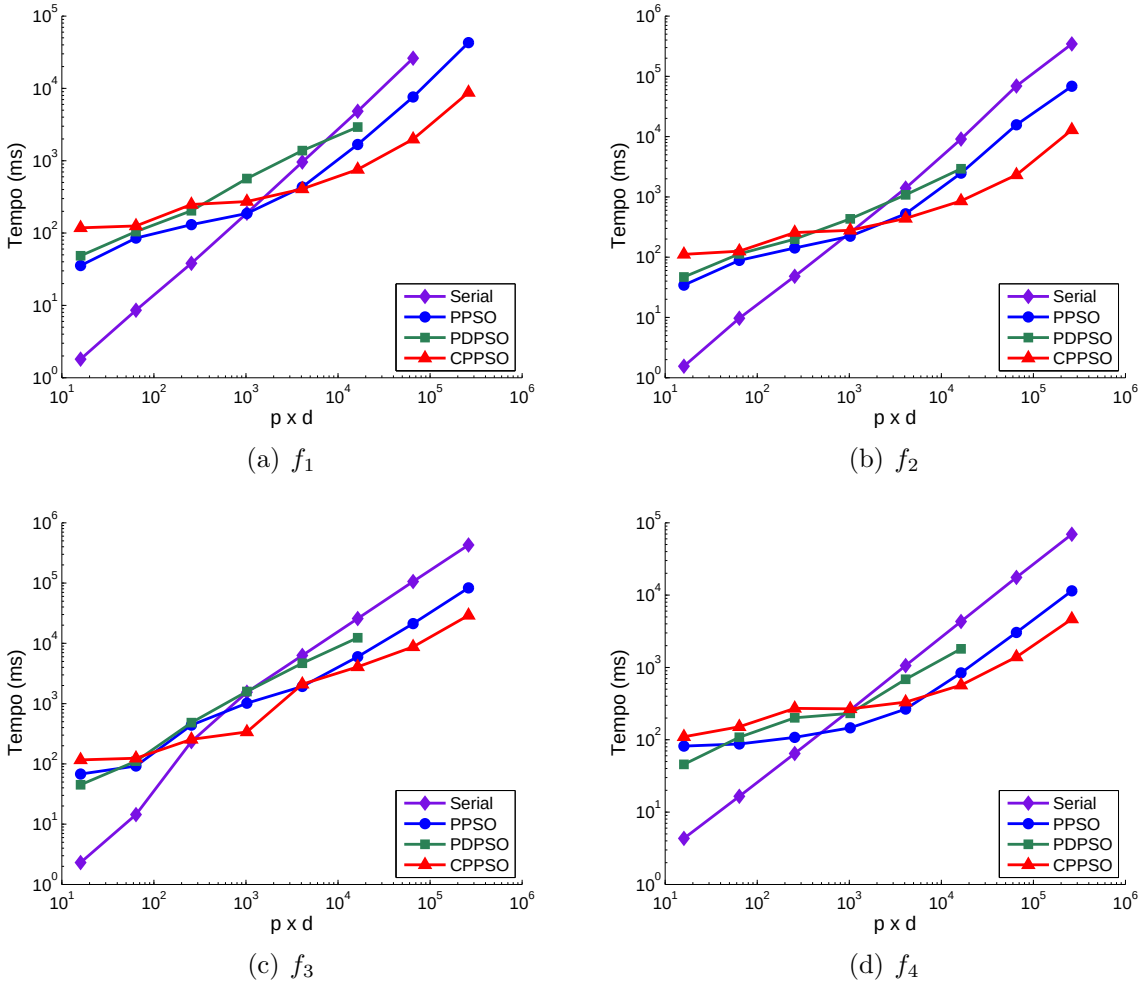


(h) *Speedup relativo* f_4

Figura 35: *Speedup real e relativo* das implementações paralelas em MPICH

6.6 Resultados da Implementação em OpenMP com MPICH

As Figuras 36, 37 e 38 apresentam os resultados obtidos pelos algoritmos propostos implementados em OpenMP com MPICH (Os valores numéricos referentes a estes gráficos podem ser consultados nas Tabelas 12 e 13 do Apêndice).



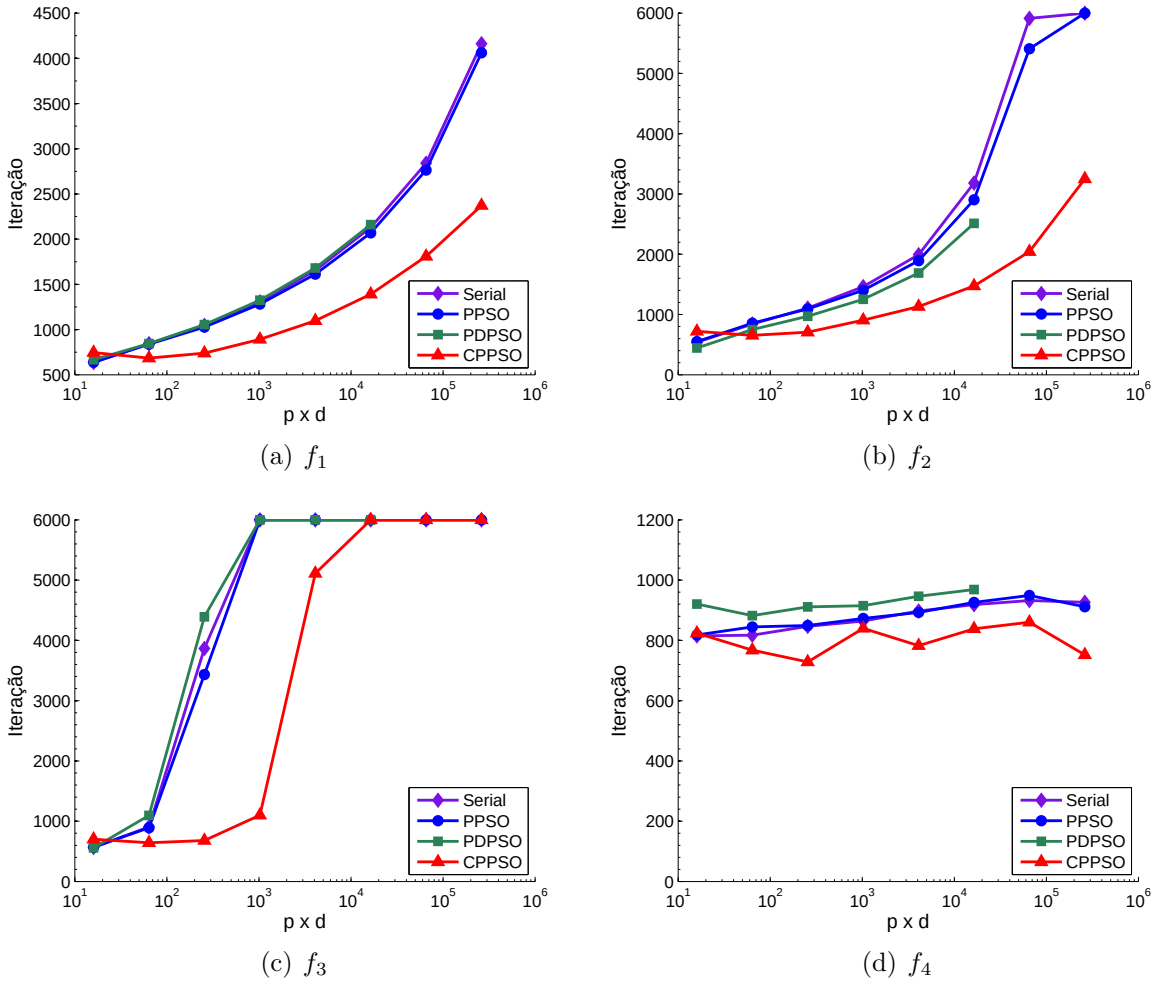
(a) f_1 (b) f_2

(c) f_3 (d) f_4

Figura 36: Tempo de execução das implementações paralelas em OpenMP com MPICH

O PPSO e o CPPSO foram executados utilizando 4 processos MPICH e 2 *threads* OpenMP, enquanto o PDPSO utilizou-se de um processo MPICH por partícula e 2 *threads* OpenMP por processo, para paralelizar o laço das dimensões.

Pode ser observado que o tempo de execução aumenta de acordo com o número de partículas e quantidade de dimensões. Nos arranjos com poucas partículas e dimensões, até o caso (32, 8), o algoritmo serial obteve tempo de execução inferior as 3 implementações em OpenMP com MPICH. Porém, a partir do caso (64, 16), o PPSO e o CPPSO começaram



(a) f_1 (b) f_2

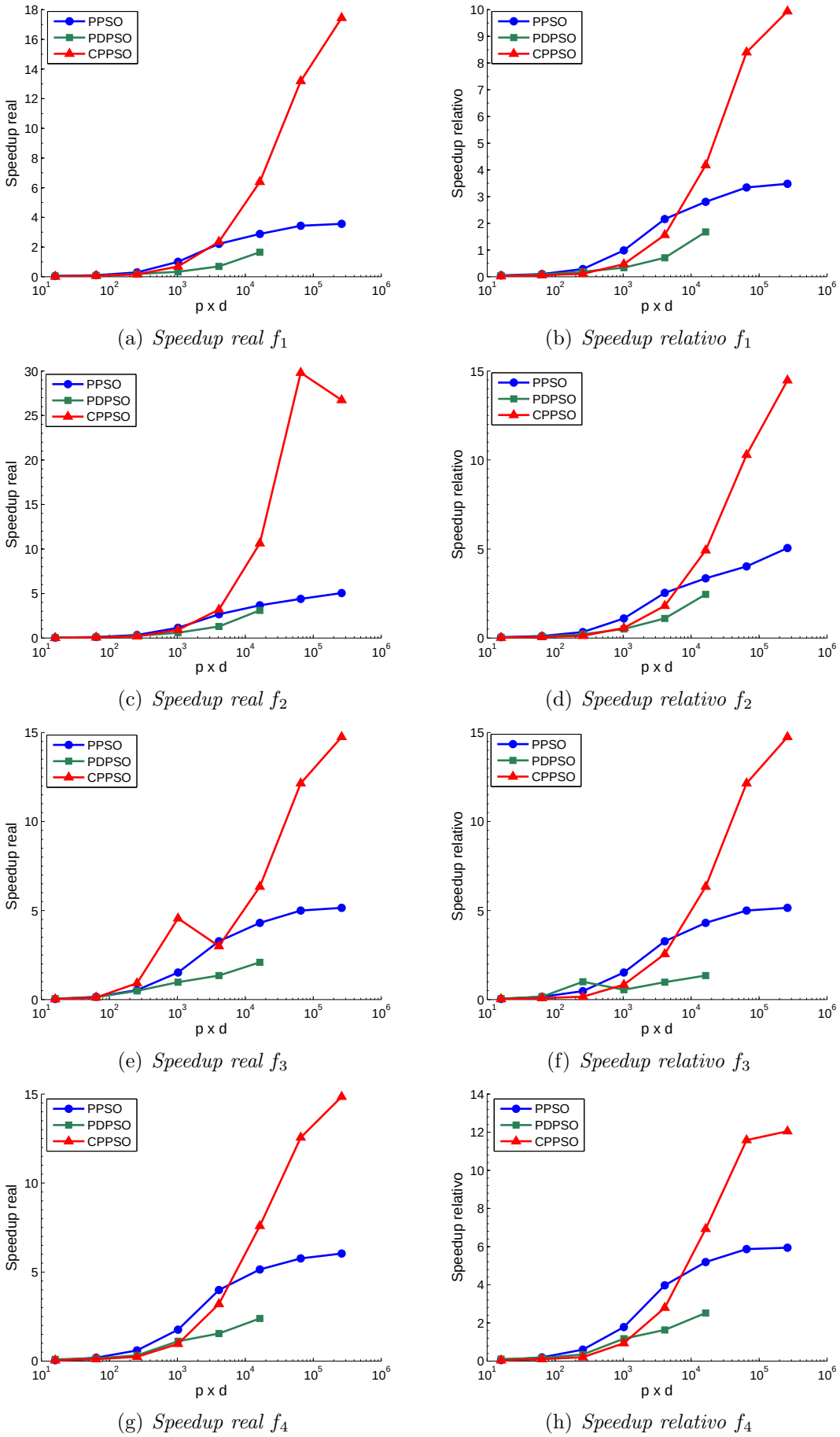
(c) f_3 (d) f_4

Figura 37: Número de iterações das implementações paralelas em OpenMP com MPICH

a obter tempos de execução menores que a implementação serial. Nota-se que, à medida que os casos aumentam o custo computacional aumenta a diferença entre o tempo de execução da implementação serial e das implementações paralelas.

Em relação à eficiência na otimização, o algoritmo CPPSO atingiu a acurácia necessária com um menor número de iterações na maioria dos casos e para todas as funções. Os Algoritmos PPSO e PDPSO obtiveram um número de iterações semelhante ao algoritmo serial.

Assim como o tempo de execução, o *speedup* aumenta com o tamanho do enxame. Pode ser observado que o algoritmo CPPSO obteve os melhores valores de *speedup real* e *speedup relativo*, atingindo um máximo de 29,8 no caso (512, 128) da função f_2 e 14,75 no caso (1024, 256) da função f_3, respectivamente. O algoritmo PPSO obteve valores máximos de *speedup real* e *speedup relativo* de 6,04 e 5,94 no caso (1024, 256) na função



(a) *Speedup real* f_1 (b) *Speedup relativo* f_1

(c) *Speedup real* f_2 (d) *Speedup relativo* f_2

(e) *Speedup real* f_3 (f) *Speedup relativo* f_3

(g) *Speedup real* f_4 (h) *Speedup relativo* f_4

Figura 38: *Speedup relativo* das implementações paralelas em OpenMP com MPICH

f_4, respectivamente.

O desempenho do algoritmo CPPSO é devido às operações de atualização de *Pbest*, *Lbest*, velocidade e posição ocorrerem com um número menor de dimensões em relação aos outros algoritmos. O algoritmo PDPSO implementado em OpenMP com MPICH obteve um desempenho inferior em relação aos demais algoritmos paralelos, devido à utilização de uma granularidade mais fina. Além disso, a necessidade de criação de uma quantidade maior processos fez com que houvesse mais processos em execução do que processadores disponíveis, degradando o desempenho. E a partir do caso (512, 128), que cria 512 processos paralelos, não foi possível realizar sua execução pela quantidade de processos por nó (aproximadamente 170) que atingiu o limite de escalabilidade do MPICH para versão 1.4.1 de 152 tarefas por nó.

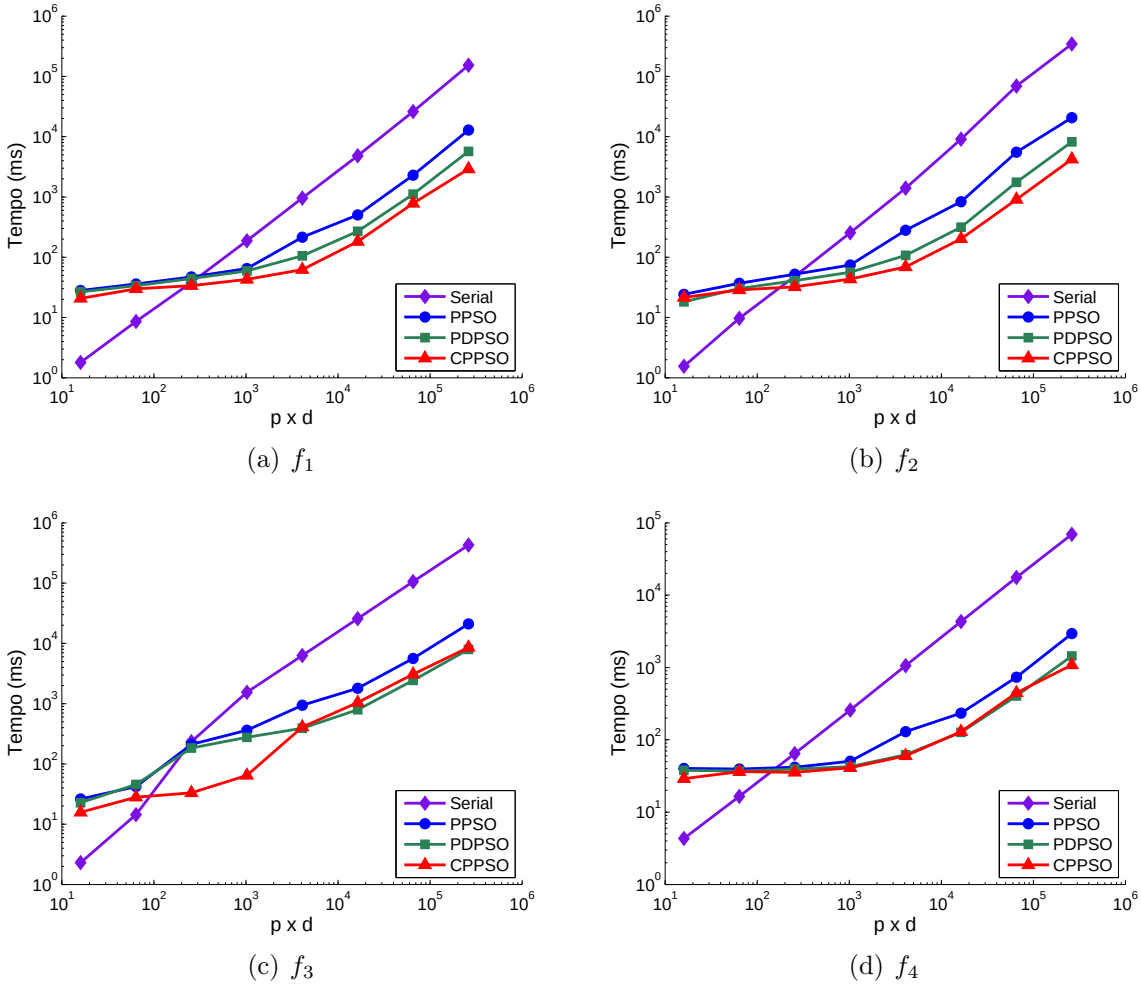
6.7 Resultados da Implementação em CUDA

As Figuras 39, 40 e 41 apresentam os resultados obtidos pelos algoritmos propostos implementados em CUDA (Os valores numéricos referentes a estes gráficos podem ser consultados nas Tabelas 14 e 15 do Apêndice).

Pode ser observado que o tempo de execução aumenta de acordo com o número de partículas e quantidade de dimensões. Nos casos (8, 2) e (16, 4), o tempo de execução das implementações em CUDA ficaram maiores que o tempo de execução do algoritmo serial. Isto é devido ao *overhead* de transferência de dados entre a CPU e a GPU. A partir destes casos, as implementações em CUDA obtêm tempos de execução menores que a implementação serial.

À medida que os testes aumentam o custo computacional, aumenta a diferença entre o tempo de execução da implementação serial para o tempo de execução das implementações paralelas. O CPPSO obteve os melhores resultados nas funções f_1, f_2 e f_3 até o caso (64, 16), enquanto que para os testes restantes da função f_3 e todos os testes da função f_4, tanto o CPPSO quanto o PDPSO obtiveram tempos de execução semelhantes.

O algoritmo CPPSO mostrou uma melhor eficiência na otimização, obtendo um menor número de iterações até atingir o critério de parada, principalmente nas funções f_1, f_2 e f_3. Os Algoritmos PPSO e PDPSO obtiveram uma eficiência semelhante ao algoritmo serial. Note que para a função f_1 as linhas que representam as iterações ficaram sobrepostas para a implementação serial, o PPSO e o PDPSO.



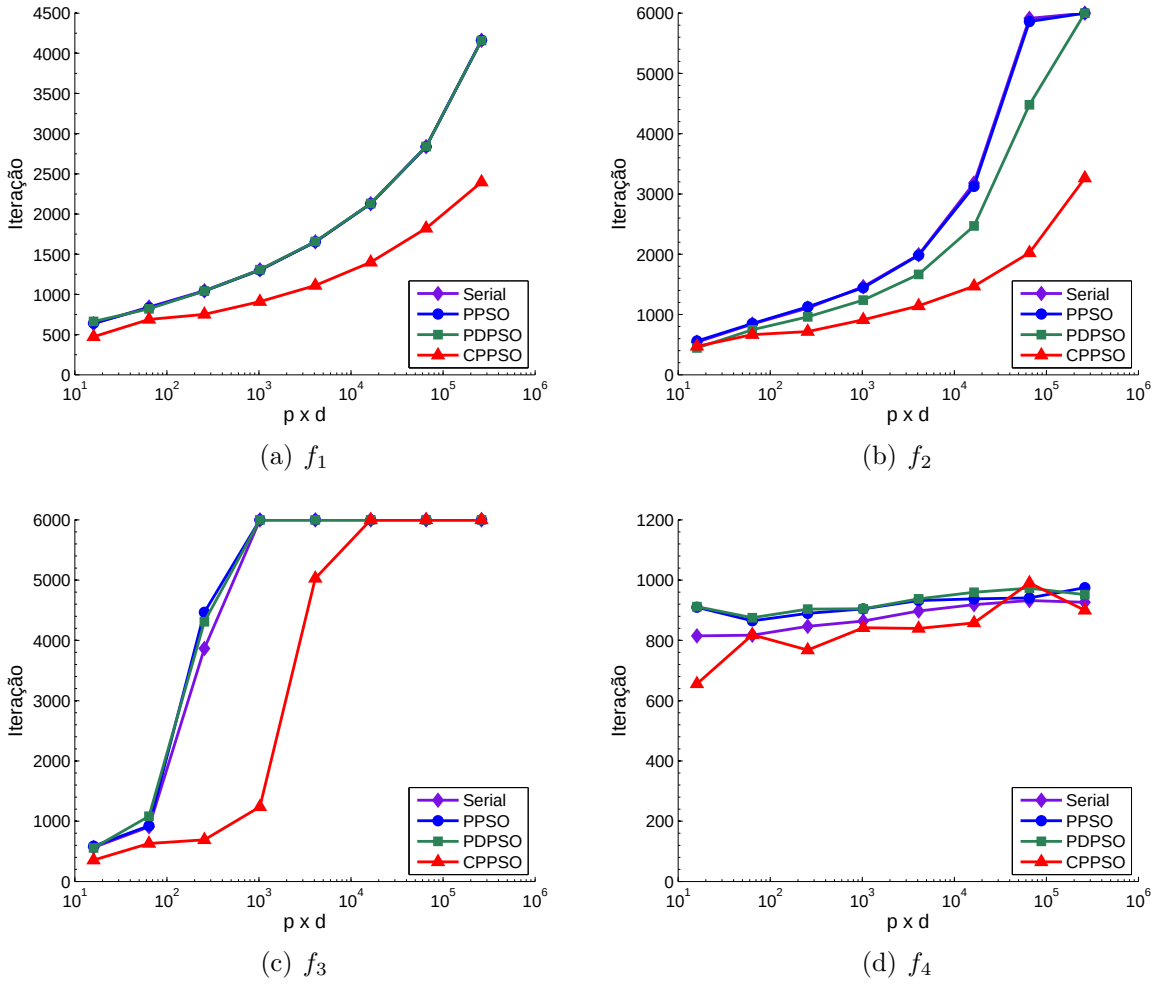
(a) f_1 (b) f_2

(c) f_3 (d) f_4

Figura 39: Tempo de execução das implementações paralelas em CUDA

Assim como o tempo de execução, o *speedup* aumenta com o tamanho do enxame. Pode ser observado que os algoritmos PDPSO e CPPSO atingiram valores próximos de *speedup relativo*, com máximos 53,81 na função f_2 e 61,97 na função f_4 respectivamente. Por outro lado, o algoritmo PPSO obteve um menor desempenho entre as implementações, alcançado um *speedup relativo* máximo de 24,72 na função f_4. Como consequência da melhor eficiência, o CPPSO apresentou também *speedup real* superior na maioria dos testes realizados, obtendo um máximo de 81,58 no caso (1024, 256) da função f_2.

O desempenho inferior do PPSO pode ser explicado devido ao mapeamento adotado de uma partícula por *thread*, levando a uma granularidade mais grossa, e um menor número de *threads*. De outro modo, o PDPSO decompõe o algoritmo com uma granularidade mais fina, gerando mais *threads* que a implementação anterior, com menos instruções por *thread*, tirando proveito da arquitetura da GPU, principalmente pelo escalonamento



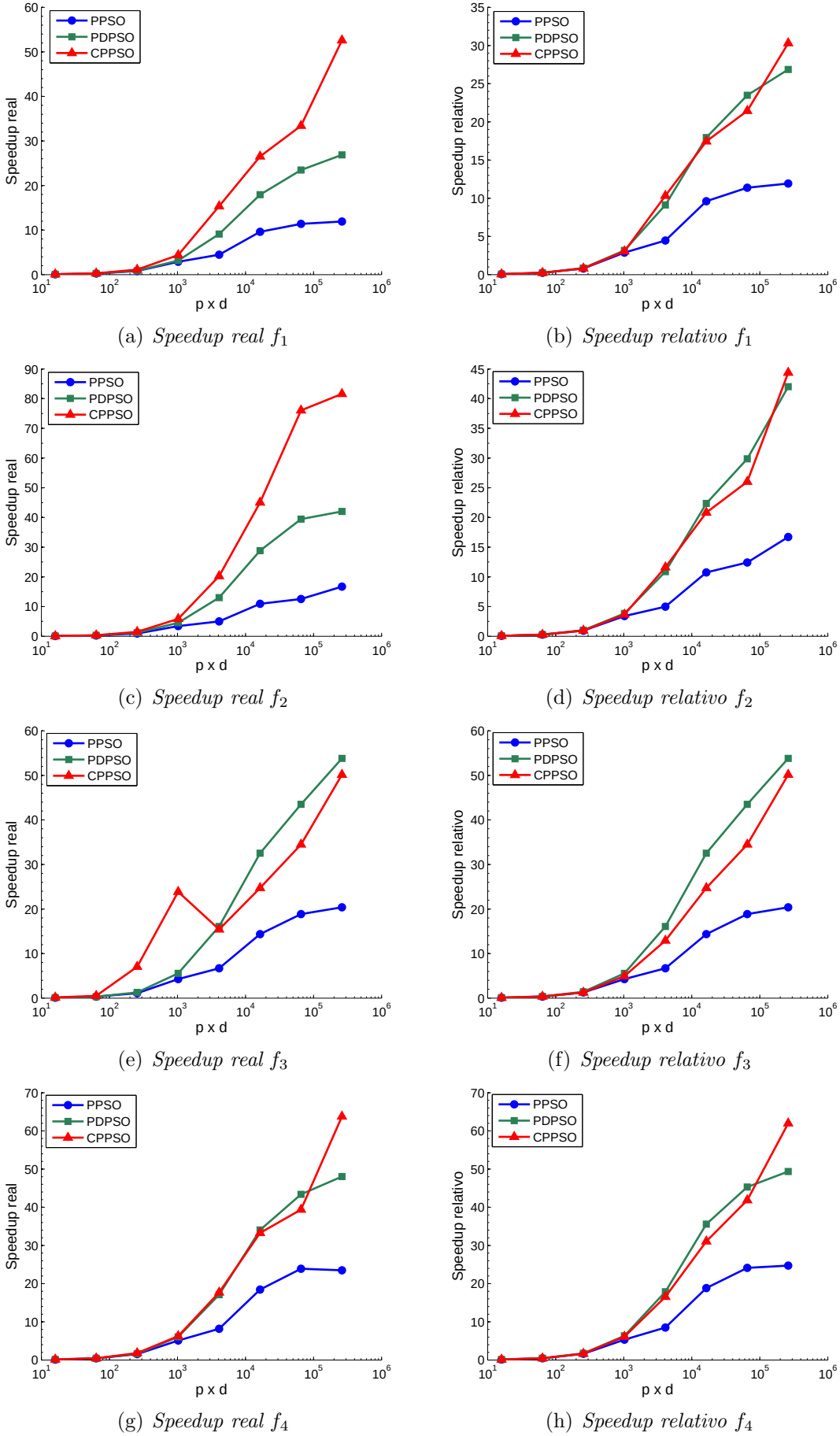
(a) f_1 (b) f_2

(c) f_3 (d) f_4

Figura 40: Número de iterações das implementações paralelas em CUDA

entre as *threads*, substituindo as que estão esperando pela latência de acesso a memória global por *threads* prontas para serem executadas.

Devido ao mapeamento adotado de enxames em blocos e partículas em *threads* e a limitação em relação à eficiência na otimização do CPPSO (um máximo de 4 enxames paralelos), este algoritmo ficou restrito a utilização de apenas 4 blocos para todos os casos abordados. Porém, as operações de atualização de *Pbest*, *Lbest*, velocidade e posição ocorrerem com um número menor de dimensões em relação aos outros algoritmos, o que certamente aumenta o desempenho. Além disso, o CPPSO não precisa compor o resultado de $fitness$ por redução conforme o algoritmo PDPSO.



(a) *Speedup real* f_1 (b) *Speedup relativo* f_1

(c) *Speedup real* f_2 (d) *Speedup relativo* f_2

(e) *Speedup real* f_3 (f) *Speedup relativo* f_3

(g) *Speedup real* f_4 (h) *Speedup relativo* f_4

Figura 41: *Speedup real e relativo* das implementações paralelas em CUDA

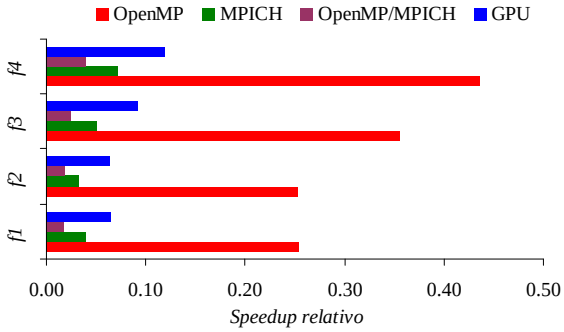
6.8 Comparação entre as Arquiteturas

Nesta seção são apresentados comparações entre as diferentes arquiteturas paralelas executando o algoritmo que obteve o melhor desempenho, conforme mostrado nas seções anteriores. Devido à limitação de área da FPGA, os resultados do coprocessador HPSO não serão comparados com as demais implementações. A Figura 42 apresenta os diagramas de barras ilustrando os *speedups relativos* obtidos pelas arquiteturas paralelas, executando o algoritmo que obteve o melhor desempenho: O PPSO implementado em MPICH; o CPPSO implementado em OpenMP, OpenMP com MPICH e GPU.

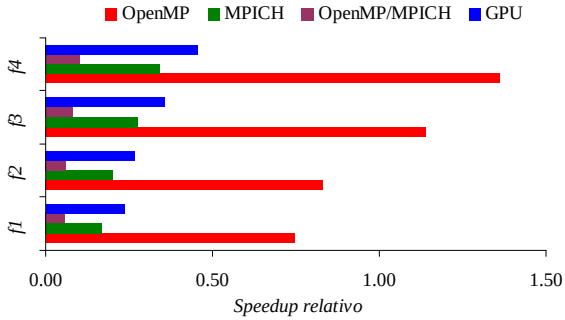
Observando cada caso separadamente, pode ser observado em todos os diagramas da Figura 42 que o desempenho obtido pelas 4 arquiteturas paralelas aumenta à medida que a função objetivo apresenta maior complexidade computacional. Pode ser observado ainda, que no caso (8, 2), de poucas partículas e dimensões, todas as implementações paralelas obtiveram desempenho inferior a implementação serial. Nos casos (16, 4) e (32, 8), a implementação em OpenMP obteve melhor desempenho que as outras arquiteturas paralelas e que a implementação serial, com exceção das funções f_1 e f_2 do caso (16, 4), conseguindo um *speedup relativo* de 2,08 na função f_4. Contudo, comparando as demais arquiteturas, a implementação em OpenMP foi a que obteve o menor *speedup relativo*, 2,60 na função f_3, e a menor proporção de crescimento do *speedup relativo* conforme aumentava o custo computacional e a complexidade dos casos.

A partir do caso (32, 8) a GPU obtém o melhor *speedup relativo* para todos os casos, aumentando o desempenho conforme aumenta o número de partículas e a quantidade de dimensões, chegando ao valor de 61,97 na função f_4. A implementação em MPICH, que possui maior latência na comunicação, obtém bons resultados a partir do caso (128, 32), em alguns casos próximos aos resultados da GPU, aumentando o desempenho conforme aumenta o número de partículas e dimensões, chegando a um *speedup relativo* de 39,67 também na função f_4.

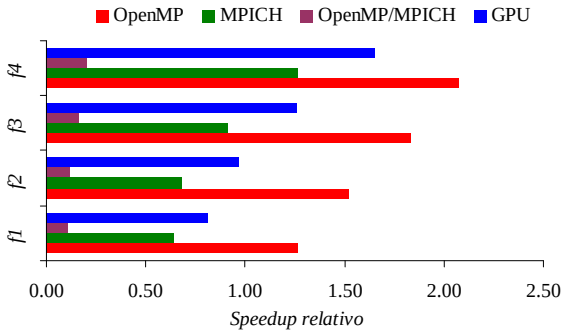
A implementação em OpenMP com MPICH obteve desempenho superior a implementação em OpenMP a partir do caso (128, 32) e alcançando um *speedup relativo* de 14,75 na função f_3. Assim, a implementação em GPU chega a ser 26 vezes mais rápida que a implementação em OpenMP, 8 vezes mais rápida que a implementação em OpenMP com MPICH e 2,7 vezes mais rápida que a implementação em MPICH.



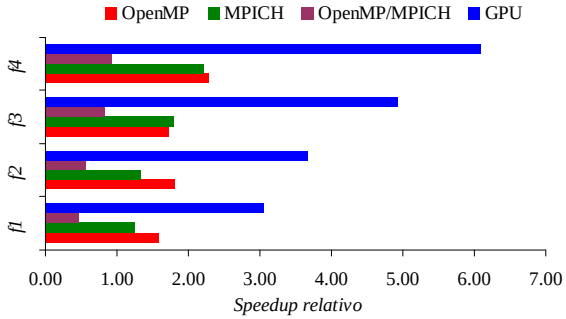
(a) (8, 2)



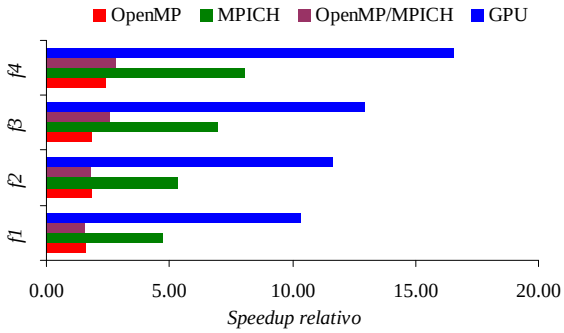
(b) (16, 4)



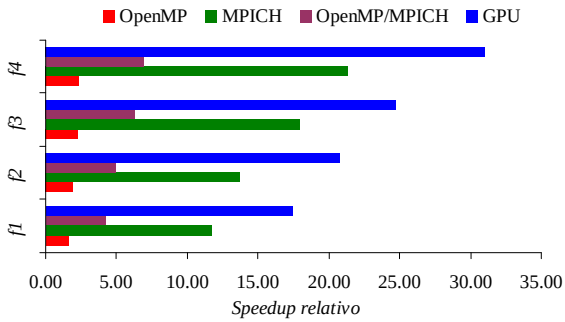
(c) (32, 8)



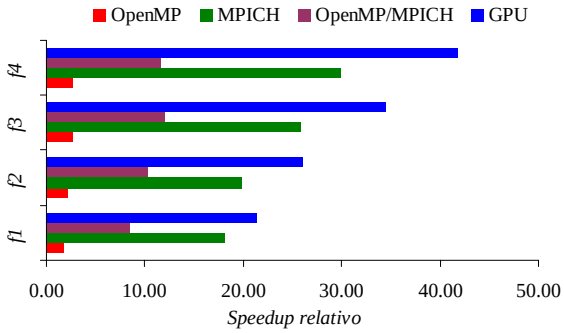
(d) (64, 16)



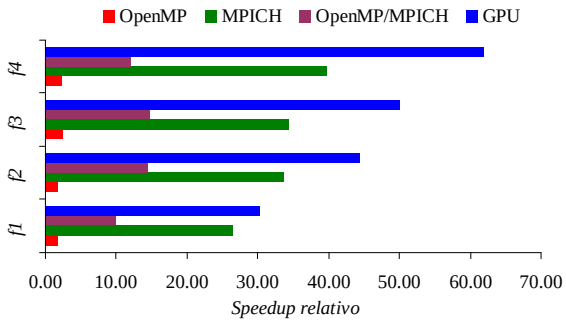
(e) (128, 32)



(f) (256, 64)



(g) (512, 128)



(h) (1024, 256)

Figura 42: *Speedup relativo* das arquiteturas paralelas

6.9 Considerações Finais do Capítulo

Neste capítulo foram apresentados os resultados obtidos pela arquitetura apresentada no Capítulo 3 bem como pelas implementações descritas nos Capítulos 4 e 5. Foi realizada a descrição das funções utilizadas e explicada a metodologia para realização das avaliações. Nos resultados obtidos, verificou-se que o coprocessador HPSO ficou limitado com relação ao número de partículas devido à falta de disponibilidade de recursos na FPGA. Pode ser observado ainda, o desempenho das implementações para os diferentes algoritmos paralelos propostos. A implementação em GPU obteve o melhor desempenho, seguida pela implementação em MPICH. O capítulo seguinte finaliza este trabalho, abordando suas principais conclusões, bem como as contribuições mais relevantes da presente dissertação. Também serão tratadas direções para possíveis trabalhos futuros.

Capítulo 7

CONCLUSÕES E TRABALHOS FUTUROS

ESTE capítulo completa esta dissertação, apresentando as principais conclusões obtidas do desenvolvimento do presente trabalho. São apresentadas também possíveis melhorias dos algoritmos propostos e as direções para futuros estudos, envolvendo o algoritmo PSO.

7.1 Conclusões

Esta dissertação abordou três estratégias de implementação paralela do algoritmo PSO em arquiteturas paralelas de alto desempenho. A primeira estratégia considera o fato de que as computações realizadas pelas partículas são quase independentes e portanto, podem ser realizadas em paralelo. A segunda estratégia considera o fato de que as operações internas aos processos que implementam as partículas também podem ser paralelizadas, proporcionando uma decomposição das operações realizadas em cada dimensão do problema que esta sendo otimizado. A terceira estratégia é baseada no algoritmo serial CPSO-S_k, desenvolvido por Van den Bergh em (BERGH et al., 2002), cuja ideia principal é subdividir o vetor de dimensões do problema original em vários subproblemas que serão otimizados por vários subenxames atuando em paralelo. Os subenxames colaboram entre si na solução do problema original.

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. Seu objetivo foi investigar e desenvolver estratégias paralelas para o PSO visando à redução do tempo de execução, assim como a manutenção ou melhoria da eficiência na otimização.

Inicialmente, o algoritmo PPSO foi implementado diretamente em *hardware* utilizando uma Xilinx Virtex 6 FPGA (xcvlx75t). A arquitetura explora o paralelismo atualizando as posições e as velocidades e calculando a função objetivo de modo independente, considerando as partículas do enxame. Os resultados de síntese demonstram que a escalabilidade do *hardware* depende do número de partículas usadas e da complexidade da função objetivo. A arquitetura do HPSO foi validada usando até 10 partículas operando em paralelo na solução de problemas de otimização de 2 dimensões. O melhor desempenho alcançado foi de 135 vezes e o menor de 20 vezes comparado ao processador MicroBlaze™. Devido à limitação de área da FPGA, os resultados obtidos pelo HPSO não puderam ser comparados com as outras arquiteturas paralelas.

Em seguida, as três estratégias paralelas do PSO foram transformadas nos algoritmos PPSO, PDPSO e CPPSO e implementadas em três diferentes arquiteturas de alto desempenho: multiprocessador, multicomputador e GPU. Os algoritmos foram avaliados utilizando diferentes arranjos de partículas e dimensões, proporcionando diferentes custos e complexidades, na otimização de 4 funções de *benchmark*. 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 contêm 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.

Implementados em OpenMP, os algoritmos PPSO e CPPSO obtiveram os melhores desempenhos nos casos que envolveram um grande número de partículas e dimensões, alcançando até 1,43 e 2,6 vezes respectivamente, em relação à implementação sequencial. Portanto, para a arquitetura de multiprocessador, os melhores resultados foram obtidos utilizando um particionamento com uma granularidade mais grossa.

Implementados em MPICH, o algoritmo PPSO obteve o melhor desempenho nos casos que apresentaram um grande número de partículas e dimensões, alcançando até 39,67 vezes em relação à implementação sequencial. A utilização de um arranjo otimizado de partículas por processo e número de processos levou a um aumento significativo no desempenho. Portanto, como a arquitetura de multiprocessadores, a arquitetura de multicomputadores obteve os melhores resultados utilizando um particionamento com uma granularidade mais grossa.

Implementados em OpenMP com MPICH, o algoritmo CPPSO obteve o melhor desempenho nos casos com grande número de partículas e dimensões, alcançando até 14,75 vezes em relação à implementação sequencial. Logo, os melhores resultados foram obtidos utilizando um particionamento com uma granularidade mais grossa.

Implementados em CUDA, os algoritmo CPPSO e PDPSO obtiveram os melhores desempenhos nos casos com grande número de partículas e dimensões, alcançando até 61,97 e 53,81 vezes respectivamente, em relação à implementação sequencial.

Conforme esperado, devido principalmente as características de cada arquitetura, a GPU obteve os melhores resultados com os algoritmos que utilizaram uma granularidade mais fina, enquanto as outras arquiteturas tiveram bons desempenhos utilizando um particionamento com uma granularidade mais grossa.

As estratégias de paralelização do algoritmo PSO utilizaram basicamente a decomposição de domínio. Esta característica foi uma das causas para que a arquitetura de GPU superasse em aproximadamente 60% o desempenho da implementação em MPICH, que obteve o segundo melhor desempenho utilizando um *hardware* que é cerca 9 vezes mais caro. De outra forma, uma aplicação que utilize decomposição funcional, ou seja, *threads* ou *processos* realizando diferentes tarefas, certamente obteria melhor desempenho implementada na arquitetura de multiprocessadores e/ou multicomputadores. 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%. Nos casos de enxames com poucas partículas e dimensões, a implementação serial apresentou o melhor desempenho.

Em relação à eficiência na otimização, os três algoritmos propostos apresentaram eficiência na maioria das otimizações de modo semelhante e até melhor que o algoritmo sequencial. Esta eficiência fica bem evidenciada nos casos em que foram obtidos *superspeedups* positivos em função da redução do número de iterações pelas implementações paralelas, especialmente pelo CPPSO. Apesar disso, a probabilidade de localizar o ótimo global decresce significantemente com o aumento do número de dimensões do problema (ENGELBRECHT, 2006), uma vez que o número de pontos do espaço de busca cresce exponencialmente. Assim, pode ser observado que para todas as implementações da função f_2 no caso (1024, 256) e da função f_3 a partir do caso (32, 8), o valor da aptidão encontrado é

cada vez maior conforme aumenta o número de dimensões. De certa forma a eficiência na otimização foi o fator que limitou o ganho no desempenho das implementações paralelas do PSO.

7.2 Trabalhos Futuros

Nesta seção, são citadas algumas possíveis modificações nos algoritmos propostos, com o intuito de melhorar seu desempenho. Também são levantadas propostas para futuros trabalhos na área de otimização por enxame de partículas.

Uma primeira investigação a ser feita é a implementação dos algoritmos propostos utilizando outras topologias. As implementações apresentadas nesta dissertação foram todas implementadas utilizando a topologia em Anel. A topologia exerce influência no componente social do algoritmo PSO e também na quantidade de comunicação entre os processos paralelos. Assim, uma topologia diferente pode modificar o desempenho e a eficiência do algoritmo PSO.

Uma segunda investigação consistiria em desenvolver outras estratégias paralelas do algoritmo PSO. O algoritmo GCPSO (*guaranteed convergence PSO*) é um algoritmo sequencial, desenvolvido para solucionar o problema de estagnação do PSO, quando este não consegue mais otimizar um problema. O GCPSO reinicializa uma ou mais partículas do enxame, sem perder a memória da melhor posição, de modo a realizar mais uma exploração no espaço de busca. Desta forma, sua aplicação em problemas complexos com grandes dimensões encontra soluções com maior precisão, porém com um tempo de execução maior.

Uma terceira investigação seria implementar as estratégias paralelas propostas de forma assíncrona ou utilizando mensagens não bloqueantes, assim como implementar as estratégias propostas na arquitetura de multiprocessadores NUMA (*NonUniform Memory Access*).

Uma quarta sugestão seria a utilização de uma FPU otimizada que poderia reduzir os requerimentos de área do HPSO e permitir a implementação com mais partículas ou a adição de mais uma FPU por partícula, aumentando o nível de paralelismo.

Por fim, as estratégias paralelas propostas podem ser utilizadas para otimizar aplicações reais. Elas podem ser aplicadas no planejamento de rota para um enxame de veículos subaquáticos autônomos. A Robótica de enxame é uma abordagem nova

que permite a coordenação de um grande número de robôs a fim de realizar uma tarefa complexa, que não pode ser realizada por um único robô, devido, principalmente pelo alto custo de fabricação. Portanto, a execução da tarefa em questão só é possível através da cooperação dos vários robôs formando um enxame, cuja operação pode ser controlada por meio do PSO paralelo.

REFERÊNCIAS

AL-ERYANI, J. *Floating point unit*. Viena, Austria, 2006. Acesso em junho 2011. Disponível em: <http://opencores.org/project/fpu100>.

ANL, A. N. L. *MPICH2 User Guide*. ., 2012. Acesso em abril 2012. Disponível em: <http://www.mpich.org/static/docs/guides/mpich21.5userguide.pdf>.

BENI, G.; WANG, J. Swarm intelligence in cellular robotic systems. *Robots and Biological Systems: Towards a New Bionics?*, Springer, p. 703–712, 1993.

BERGH, F. V. D. et al. An analysis of particle swarm optimizers. University of Pretoria, 2002.

CADENAS-MONTES, M. et al. Accelerating particle swarm algorithm with gpgpu. In: IEEE (Ed.). *Parallel, Distributed and Network-Based Processing (PDP)*. Ayia Napa, Chipre: 19th Euromicro International Conference on, 2011. p. 560–564.

CALAZAN, R.; NEDJAH, N.; MOURELLE, L. Parallel co–processor for pso. *International Journal of High Performance Systems Architecture*, Inderscience, v. 3, n. 4, p. 233–240, 2011.

CALAZAN, R.; NEDJAH, N.; MOURELLE, L. A hardware accelerator for particle swarm optimization. *Applied Soft Computing*, n. 0, p. –, 2013. ISSN 1568-4946.

CALAZAN, R.; NEDJAH, N.; MOURELLE, L. de M. Swarm grid: a proposal for high performance of parallel particle swarm optimization using gpgpu. *Computational Science and Its Applications–ICCSA 2012*, Springer, p. 148–160, 2012.

CALAZAN, R. de M.; NEDJAH, N.; MOURELLE, L. de M. A massively parallel reconfigurable co-processor for computationally demanding particle swarm optimization. In: IEEE. *Circuits and Systems (LASCAS), 2012 IEEE Third Latin American Symposium on*. Playa del Carmem, México, 2012.

CALAZAN, R. de M.; NEDJAH, N.; MOURELLE, L. de M. Parallel gpu-based implementation of high dimension particle swarm optimizations. In: IEEE. *Circuits and Systems (LASCAS), 2013 IEEE Fourth Latin American Symposium on*. Cuzco, Perú, 2013.

CHAPMAN, B.; JOST, G.; PAS, R. V. D. *Using OpenMP: portable shared memory parallel programming*. England: MIT press, 2008.

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

DORIGO, M.; MANIEZZO, V.; COLORNI, A. Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, v. 26, n. 1, p. 29 –41, fev 1996. ISSN 1083-4419.

ENGELBRECHT, A. P. *Fundamentals of Computational Swarm Intelligence*. Chichester, UK.: John Wiley & Sons, 2006. ISBN 0470091916.

FARBER, R. *CUDA application design and development*. Waltham, MA, USA: Morgan Kaufmann, 2011. ISBN 978-0-12-388426-8.

FOSTER, I. *Designing and building parallel programs*. MA, USA: Addison-Wesley Reading, 1995.

GROPP, W.; LUSK, E. A test implementation of the mpi draft message-passing standard. *Argonne Nat. Lab., ANL-92/47, Argonne, IL*, 1992.

GROPP, W.; LUSK, E.; SKJELLUM, A. *Using MPI: portable parallel programming with the message passing interface*. MA, USA: MIT press, 1999.

GROPP, W.; SMITH, B. Users manual for the chameleon parallel programming tools. *Math. and Comp. Science Div., Argonne Nat. Lab., ANL-93/23*, 1993.

HUGHES, E. Multi-objective evolutionary guidance for swarms. In: IEEE. *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*. Hawaii, US, 2002. v. 2, p. 1127–1132.

KENNEDY, J.; EBERHART, R. Particle swarm optimization. In: IEEE. *Proceedings of the 1995 IEEE International Conference on Neural Networks*. Perth, WA , Australia, 1995. v. 4, p. 1942 –1948 vol.4.

KENNEDY, J.; MENDES, R. Population structure and particle swarm performance. In: IEEE. *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*. Honolulu, HI, USA, 2002. v. 2, p. 1671–1676.

KILTS, S. *Advanced fpga design: architecture, implementation, and optimization*. New Jersey, USA: Wiley-IEEE press, 2007.

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

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

LEE, V. et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In: ACM. *ACM SIGARCH Computer Architecture News*. ., 2010. v. 38, n. 3, p. 451–460.

LI, S.-A. et al. Hardware/software co-design for particle swarm optimization algorithm. In: INFORMATION SCIENCES. *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on*. ., 2010. p. 3762 –3767. ISSN 1062-922X.

MA, Q.; LEI, X.; ZHANG, Q. Mobile robot path planning with complex constraints based on the second-order oscillating particle swarm optimization algorithm. In: IEEE. *Computer Science and Information Engineering, 2009 WRI World Congress on*. ., 2009. v. 5, p. 244–248.

MOLGA, M.; SMUTNICKI, C. Test functions for optimization needs. *Test functions for optimization needs*, 2005.

MPI_FORUM. *Message Passing Interface Forum*. ., 2012. Acesso em maio 2012. Disponível em: <http://www.mpiforum.org>.

MUNOZ, D. et al. Hardware architecture for particle swarm optimization using floating-point arithmetic. In: IEEE. *Intelligent Systems Design and Applications, 2009. ISDA'09. Ninth International Conference on*. ., 2009. p. 243–248.

NVIDIA. *Relatório da NVIDIA sobre: NVIDIA Next Generation CUDA Compute Architecture: Fermi*. ., 2009. Acesso em 11 de nov. 2010. Disponível em: <http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>.

NVIDIA. *Manual da Nvidia sobre CUDA C Programming Guide*. ., 2010. Acesso em 20 de ago. 2010. Disponível em: <http://developer.nvidia.com/object/cuda_3_2_downloads>.

NVIDIA. *Relatório da Nvidia sobre Curand Lybrary*. ., 2010. Acesso em 15 set. 2011. Disponível em: <http://developer.nvidia.com/cuda/curand>.

PATTERSON, D.; HENNESSY, J. *Computer organization and design: the hardware/software interface*. .: 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*, Elsevier, v. 72, n. 1, p. 5–17, 2003.

REYNOLDS, C. Flocks, herds and schools: A distributed behavioral model. In: ACM. *ACM SIGGRAPH Computer Graphics*. ., 1987. v. 21, n. 4, p. 25–34.

SCHUTTE, J. et al. Parallel global optimization with the particle swarm algorithm. *International Journal for Numerical Methods in Engineering*, Wiley Online Library, v. 61, n. 13, p. 2296–2315, 2004.

SECREST, B. *Traveling salesman problem for surveillance mission using particle swarm optimization*. ., 2001.

SEDIGHIZADEH, D.; MASEHIAN, E. Particle swarm optimization methods, taxonomy and applications. *International Journal of Computer Theory and Engineering*, v. 1, n. 5, p. 1793–8201, 2009.

SHANG, Y.; QIU, Y. A note on the extended rosenbrock function. *Evolutionary Computation*, MIT Press, v. 14, n. 1, p. 119–126, 2006.

SHI, Y.; EBERHART, R. A modified particle swarm optimizer. In: IEEE. *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*. ., 1998. p. 69–73.

SHI, Y.; EBERHART, R. Parameter selection in particle swarm optimization. In: SPRINGER. *Evolutionary Programming VII*. ., 1998. p. 591–600.

TANENBAUM, A. S.; ZUCCHI, W. L. *Organização estruturada de computadores*. New Jersey - USA: Pearson Prentice Hall, 2009.

VERONESE, L. de P.; KROHLING, R. Swarm's flight: accelerating the particles using c-cuda. In: IEEE. *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*. ., 2009. p. 3264–3270.

WAINTRAUB, M. *Algoritmos Paralelos de Otimização por Enxame de Partículas em Problemas Nucleares*. Tese (Doutorado) — Universidade Federal do Rio de Janeiro, 2009.

WALKER, D.; DONGARRA, J. Mpi: a standard message passing interface. *Supercomputer*, ASFRA BV, v. 12, p. 56–68, 1996.

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). IEEE Congress on*. Hong Kong, China, 2008. p. 1214 –1218.

XILINX. *Fast Simplex Link*. ., 2008. Acesso em junho 2011. Disponível em: <http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf>.

XILINX. *MicroBlaze Processor Reference Guide*. ., 2008. Acesso em junho 2011. Disponível em: <http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf>.

XILINX. *Xilinx Synthesis Technology (XST) User Guide*. ., 2009. Acesso em julho 2011. Disponível em: <http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/xst.pdf>.

XILINX. *Virtex 6 FPGA User guide*. ., 2010. Acesso em julho 2011. Disponível em: <http://www.xilinx.com/support/documentation/user_guides/ug360.pdf>.

XILINX. *ML505/ML506/ML507 Evalution Platform User Guide*. ., 2011. Acesso em julho 2011. Disponível em: <http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf>.

YANG, G.; ZHANG, R. Path planning of auv in turbulent ocean environments used adapted inertia-weight pso. In: IEEE. *Natural Computation, 2009. ICNC'09. Fifth International Conference on*. ., 2009. v. 3, p. 299–302.

ZHOU, Y.; TAN, Y. Gpu-based parallel particle swarm optimization. In: IEEE. *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*. ., 2009. p. 1493–1500.

APÊNDICE – Resultados numéricos das implementações do algoritmo PSO

Este apêndice apresenta os valores numéricos obtidos nos testes realizados para diferentes combinações de partículas e dimensões pelas implementações do algoritmo PSO exibidos no Capítulo 6.

Tabela 4: Tempo de execução e *speedup* para o processador MicroBlaze *vs.* o coprocessador HPSO para f_1, f_2, f_3 e f_4 referente as Figuras 27 e 28

Função	#partícula	Microblaze	Coprocessador			*Speedup*
		Tempo (ms)	Frequência	Ciclos	Tempo	
f_1	4	14,38	233,85	157908	0,79	18,22
	6	21,54	232,94	165236	0,83	26,07
	8	57,40	229,55	172668	0,86	66,49
	10	71,74		180116	0,90	79,66
f_2	4	27,73	233,85	183108	0,92	30,29
	6	81,75	232,94	190436	0,95	85,85
	8	113,04	229,55	197868	0,99	114,26
	10	138,69		205300	1,03	135,11
f_3	4	64,87	233,85	387568	1,93	33,47
	6	97,27	232,94	398352	1,99	48,84
	8	129,55	229,55	407639	2,04	63,67
	10	161,95		208279	2,08	77,76
f_4	4	44,79	233,85	323874	1,62	27,66
	6	69,15	232,94	331215	1,65	41,75
	8	102,76	229,55	340317	1,70	60,39
	10	129,95		348265	1,74	74,62

Tabela 5: Utilização da área para o processador MicroBlaze *vs.* o coprocessador HPSO para f_1, f_2, f_3 e f_4 referente a Figura 29

Função	#partícula	Slices		Flip-Flops		LuTs	
		#	%	#	%	#	%
		11640	100	93120	100	46560	100
MicroBlaze	4 to 10	2211	19	4656	5	2328	5
f_1	4	6232	54	11442	12	16359	35
	6	8154	70	15265	16	24741	53
	8	9854	85	20338	22	30898	66
	10	10940	94	26125	28	38977	84
f_2	4	6753	58	12179	13	17429	37
	6	8967	77	17257	18	25181	54
	8	10601	91	22464	24	32686	70
	10	11494	98	27636	29	39990	85
f_3	4	7233	62	13715	15	19004	41
	6	9630	83	19650	21	27557	59
	8	10516	90	25528	27	35795	77
	10	11530	99	31466	34	43675	94
f_4	4	6984	60	13432	14	18795	40
	6	9461	81	18831	20	26447	57
	8	10117	87	24332	26	34562	74
	10	11212	96	30169	32	42775	92

Tabela 6: Tempo de execução (ms), número de iterações e $fitness$ obtidos pela implementação serial para f_1, f_2, f_3 e f_4

Função	Teste	Tempo	#Iterações	$Fitness$
f_1	(8, 2)	1,8	636	0,000050
	(16, 4)	8,6	843	0,000069
	(32, 8)	38,1	1046	0,000081
	(64, 16)	187,7	1306	0,000090
	(128, 32)	958,9	1654	0,000094
	(256, 64)	4835,7	2129	0,000097
	(512, 128)	26128,3	2839	0,000098
	(1024, 256)	153219,2	4162	0,000099
f_2	(8, 2)	1,6	541	0,000053
	(16, 4)	9,7	842	0,000073
	(32, 8)	48,2	1106	0,000090
	(64, 16)	254,6	1462	0,000093
	(128, 32)	1397,5	1993	0,000097
	(256, 64)	9089,3	3181	0,000099
	(512, 128)	69140,1	5914	0,000100
	(1024, 256)	345288,4	6000	1,510714
f_3	(8, 2)	2,3	564	0,000047
	(16, 4)	14,4	907	0,000074
	(32, 8)	233,9	3865	0,457738
	(64, 16)	1541,6	6000	8,094686
	(128, 32)	6296,7	6000	25,606653
	(256, 64)	25809,5	6000	67,812385
	(512, 128)	106018,9	6000	177,784783
	(1024, 256)	428441,5	6000	512,303036
f_4	(8, 2)	4,3	815	0,000045
	(16, 4)	16,5	817	0,000048
	(32, 8)	64,6	847	0,000055
	(64, 16)	258,0	864	0,000054
	(128, 32)	1062,1	897	0,000049
	(256, 64)	4320,2	919	0,000050
	(512, 128)	17532,8	932	0,000047
	(1024, 256)	69254,9	926	0,000046

Tabela 7: Tempo de execução (ms), número de iterações e $fitness$ das implementações paralelas em OpenMP para f_1, f_2, f_3 e f_4 referente as Figuras 30 e 31

Função	Teste	PPSO			PDPSO			CPPSO		
		Tempo	#Iterações	$Fitness$	Tempo	#Iterações	$Fitness$	Tempo	#Iterações	$Fitness$
f_1	(8,2)	7,5	628	0,000053	55,4	656	0,000005	8,3	748	0,000061
	(16,4)	12,6	826	0,000068	129,6	823	0,000008	9,4	692	0,000065
	(32,8)	48,5	1033	0,000084	341,5	1026	0,000008	24,8	863	0,000084
	(64,16)	217,4	1303	0,000082	974,3	1260	0,000008	97,1	1066	0,000089
	(128,32)	986,3	1648	0,000093	2897,6	1580	0,000009	483,1	1327	0,000094
	(256,64)	5364,2	2126	0,000096	11591,9	1985	0,000009	2411,1	1705	0,000097
	(512,128)	29878,6	2829	0,000097	45258,0	2551	0,000010	11430,7	2213	0,000099
	(1024,256)	174389,2	4260	0,000098	225354,2	3667	0,000010	62305,7	2982	0,000099
f_2	(8,2)	6,3	518	0,000054	60,8	521	0,000006	8,2	722	0,000075
	(16,4)	13,6	843	0,000081	135,0	842	0,000007	9,0	649	0,000073
	(32,8)	54,9	1111	0,000091	368,5	1092	0,000008	25,0	874	0,000083
	(64,16)	260,2	1437	0,000090	1022,2	1386	0,000008	106,1	1107	0,000091
	(128,32)	1331,0	1974	0,000096	3351,3	1803	0,000009	539,2	1448	0,000096
	(256,64)	8472,0	3133	0,000098	23101,3	2563	0,000010	2942,5	1974	0,000098
	(512,128)	66044,5	6000	0,000996	97898,2	4436	0,000078	17106,8	3140	0,000099
	(1024,256)	302410,5	6000	5,133212	318836,9	6000	0,489492	108352,5	3265	0,000099
f_3	(8,2)	7,1	571	0,000051	47,7	553	0,000005	8,1	703	0,000061
	(16,4)	18,5	1108	0,039867	147,9	948	0,000006	8,7	624	0,000066
	(32,8)	234,4	3931	0,218954	1248,3	3563	0,016162	35,3	1068	0,000073
	(64,16)	1247,9	6000	8,433257	4604,5	6000	0,854693	156,7	1056	0,000079
	(128,32)	4855,6	6000	24,179750	11565,3	6000	2,593158	3016,6	5340	0,716623
	(256,64)	19731,7	6000	68,136515	51137,2	6000	7,003599	11339,0	6000	54,904767
	(512,128)	79127,4	6000	191,245802	110156,8	6000	18,111431	40730,7	6000	141,995778
	(1024,256)	335269,9	6000	539,847855	372173,7	6000	55,547470	173007,9	6000	367,069678
f_4	(8,2)	10,3	817	0,000051	69,5	818	0,000005	10,0	817	0,000050
	(16,4)	13,7	825	0,000052	130,8	843	0,000004	11,3	760	0,000048
	(32,8)	48,6	851	0,000043	291,7	850	0,000004	29,5	802	0,000048
	(64,16)	166,6	876	0,000041	642,5	874	0,000004	106,4	817	0,000045
	(128,32)	699,5	899	0,000043	1625,4	881	0,000004	417,9	856	0,000045
	(256,64)	2787,0	925	0,000046	7930,7	929	0,000003	1731,1	876	0,000043
	(512,128)	13222,3	913	0,000059	18356,2	1035	0,318082	6508,3	900	0,000040
	(1024,256)	48385,8	956	0,000051	53217,2	910	0,000004	29223,1	930	0,000053

Tabela 8: Valores numéricos de *speedup real* e *speedup relativo* das implementações paralelas em OpenMP para f_1, f_2, f_3 e f_4 referente a Figura 32

Função	Teste	*Speedup real*			*Speedup relativo*		
		PPSO	PDPSO	CPPSO	PPSO	PDPSO	CPPSO
f_1	(8, 2)	0,24	0,03	0,22	0,24	0,03	0,25
	(16, 4)	0,68	0,07	0,91	0,67	0,06	0,75
	(32, 8)	0,79	0,11	1,53	0,78	0,11	1,27
	(64, 16)	0,86	0,19	1,93	0,86	0,19	1,58
	(128, 32)	0,97	0,33	1,98	0,97	0,32	1,59
	(256, 64)	0,90	0,42	2,01	0,90	0,39	1,61
	(512, 128)	0,87	0,58	2,29	0,87	0,52	1,78
	(1024, 256)	0,88	0,68	2,46	0,90	0,60	1,76
f_2	(8, 2)	0,25	0,03	0,19	0,24	0,02	0,25
	(16, 4)	0,71	0,07	1,08	0,72	0,07	0,83
	(32, 8)	0,88	0,13	1,93	0,88	0,13	1,52
	(64, 16)	0,98	0,25	2,40	0,96	0,24	1,82
	(128, 32)	1,05	0,42	2,59	1,04	0,38	1,88
	(256, 64)	1,07	0,39	3,09	1,06	0,32	1,92
	(512, 128)	1,05	0,71	4,04	1,06	0,53	2,15
	(1024, 256)	1,14	1,08	3,19	1,14	1,08	1,73
f_3	(8, 2)	0,32	0,05	0,29	0,33	0,05	0,36
	(16, 4)	0,78	0,10	1,66	0,95	0,10	1,14
	(32, 8)	1,00	0,19	6,63	1,01	0,17	1,83
	(64, 16)	1,24	0,33	9,84	1,24	0,33	1,73
	(128, 32)	1,30	0,54	2,09	1,30	0,54	1,86
	(256, 64)	1,31	0,50	2,28	1,31	0,50	2,28
	(512, 128)	1,34	0,96	2,60	1,34	0,96	2,60
	(1024, 256)	1,28	1,15	2,48	1,28	1,15	2,48
f_4	(8, 2)	0,42	0,06	0,44	0,42	0,06	0,44
	(16, 4)	1,20	0,13	1,46	1,21	0,13	1,36
	(32, 8)	1,33	0,22	2,19	1,34	0,22	2,08
	(64, 16)	1,55	0,40	2,42	1,57	0,41	2,29
	(128, 32)	1,52	0,65	2,54	1,52	0,64	2,43
	(256, 64)	1,55	0,54	2,50	1,56	0,55	2,38
	(512, 128)	1,33	0,96	2,69	1,30	1,06	2,60
	(1024, 256)	1,43	1,30	2,37	1,48	1,28	2,38

Tabela 9: Tempo de execução (ms), número de iterações e *fitness* das implementações paralelas em MPICH para f_1, f_2, f_3 e f_4 referente as Figuras 33 e 34

Função	Teste	PPSO			PDPSO			CPPSO		
		Tempo	#Iterações	*Fitness*	Tempo	#Iterações	*Fitness*	Tempo	#Iterações	*Fitness*
f_1	(8,2)	47,0	656	0,000037	1608,5	399	0,000051	95,9	681	0,000057
	(16,4)	50,2	823	0,000050	2620,9	510	0,000070	113,7	869	0,000070
	(32,8)	58,4	1026	0,000058	8778,2	617	0,000083	133,7	1053	0,000076
	(64,16)	145,6	1260	0,000067	-	-	-	234,0	1284	0,000084
	(128,32)	194,1	1580	0,000067	-	-	-	323,6	1582	0,000090
	(256,64)	384,3	1985	0,000079	-	-	-	753,5	1806	0,000094
	(512,128)	1296,9	2551	0,000085	-	-	-	1879,5	2250	0,000096
	(1024,256)	5105,3	3667	0,000093	-	-	-	7393,9	2890	0,000099
f_2	(8,2)	46,1	521	0,000044	1622,7	458	0,000060	77,5	599	0,000051
	(16,4)	48,1	842	0,000055	2499,3	543	0,000070	121,4	897	0,000070
	(32,8)	69,8	1092	0,000064	8563,1	641	0,000083	176,0	1099	0,000081
	(64,16)	181,5	1386	0,000069	-	-	-	259,0	1362	0,000091
	(128,32)	237,2	1803	0,000082	-	-	-	413,4	1729	0,000095
	(256,64)	532,9	2563	0,000087	-	-	-	927,4	2027	0,000096
	(512,128)	2607,5	4436	0,000095	-	-	-	2906,1	2901	0,000099
	(1024,256)	10253,5	6000	0,122373	-	-	-	16235,1	5104	0,000100
f_3	(8,2)	46,3	574	0,000043	1615,8	352	0,000055	75,7	622	0,000050
	(16,4)	50,9	887	0,000041	2693,5	1381	0,037858	129,9	955	0,000070
	(32,8)	195,0	2941	0,199022	8390,4	2973	0,270632	473,4	2667	0,278640
	(64,16)	857,2	6000	7,768387	-	-	-	1105,8	5407	1,679379
	(128,32)	904,7	6000	24,003676	-	-	-	1907,2	6000	26,315831
	(256,64)	1437,1	6000	61,817438	-	-	-	3475,8	6000	79,065796
	(512,128)	4103,6	6000	137,709509	-	-	-	9063,1	6000	247,559837
	(1024,256)	12507,7	6000	330,217010	-	-	-	30467,7	6000	680,005892
f_4	(8,2)	59,1	801	0,000041	1613,1	557	0,000053	124,9	1006	0,000055
	(16,4)	49,8	844	0,000036	2713,3	651	0,000056	109,6	846	0,000030
	(32,8)	51,3	850	0,000037	8368,8	674	0,000054	240,1	1111	0,000064
	(64,16)	117,0	871	0,000039	-	-	-	250,0	1165	0,000045
	(128,32)	131,4	894	0,000029	-	-	-	421,1	1185	0,000048
	(256,64)	206,2	937	0,000056	-	-	-	829,4	1260	0,000044
	(512,128)	593,4	947	0,000038	-	-	-	1984,3	1089	0,000022
	(1024,256)	1813,4	962	0,000035	-	-	-	7834,6	1255	0,000079

Tabela 10: *Speedup real* e *Speedup relativo* das implementações paralelas em MPICH para f_1, f_2, f_3 e f_4 referente a Figura 35

Função	Teste	*Speedup real*			*Speedup relativo*		
		PPSO	PDPSO	CPPSO	PPSO	PDPSO	CPPSO
f_1	(8,2)	0,04	0,00	0,02	0,04	0,00	0,02
	(16, 4)	0,17	0,00	0,08	0,17	0,00	0,08
	(32, 8)	0,65	0,00	0,29	0,64	0,00	0,29
	(64, 16)	1,29	-	0,80	1,24	-	0,79
	(128, 32)	4,94	-	2,96	4,72	-	2,83
	(256, 64)	12,58	-	6,42	11,74	-	5,45
	(512, 128)	20,15	-	13,90	18,10	-	11,02
	(1024, 256)	30,01	-	20,72	26,44	-	14,39
f_2	(8,2)	0,03	0,00	0,02	0,03	0,00	0,02
	(16, 4)	0,20	0,00	0,08	0,20	0,00	0,09
	(32, 8)	0,69	0,01	0,27	0,68	0,00	0,27
	(64, 16)	1,40	-	0,98	1,33	-	0,92
	(128, 32)	5,89	-	3,38	5,33	-	2,93
	(256, 64)	17,06	-	9,80	13,74	-	6,25
	(512, 128)	26,52	-	23,79	19,89	-	11,67
	(1024, 256)	33,68	-	21,27	33,68	-	18,09
f_3	(8,2)	0,05	0,00	0,03	0,05	0,00	0,03
	(16, 4)	0,28	0,01	0,11	0,28	0,01	0,12
	(32, 8)	1,20	0,03	0,49	0,91	0,02	0,34
	(64, 16)	1,80	-	1,39	1,80	-	1,26
	(128, 32)	6,96	-	3,30	6,96	-	3,30
	(256, 64)	17,96	-	7,43	17,96	-	7,43
	(512, 128)	25,84	-	11,70	25,84	-	11,70
	(1024, 256)	34,25	-	14,06	34,25	-	14,06
f_4	(8,2)	0,07	0,00	0,03	0,07	0,00	0,04
	(16, 4)	0,33	0,01	0,15	0,34	0,00	0,16
	(32, 8)	1,26	0,01	0,27	1,26	0,01	0,35
	(64, 16)	2,21	-	1,03	2,22	-	1,39
	(128, 32)	8,08	-	2,52	8,05	-	3,33
	(256, 64)	20,95	-	5,21	21,36	-	7,14
	(512, 128)	29,55	-	8,84	30,00	-	10,32
	(1024, 256)	38,19	-	8,84	39,67	-	11,98

Tabela 11: Análise de desempenho de diferentes combinações de processos e partículas por processo do algoritmo PPSO implementado em MPICH para a função f_1

Função	Processos	Part/proc	Tempo (ms)	*Speedup relativo*
(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

Tabela 12: Tempo de execução (ms), número de iterações e *fitness* das implementações paralelas em OpenMP com MPICH para f_1, f_2, f_3 e f_4 referente as Figuras 36 e 37

Função	Teste	PPSO			PDPSO			CPPSO		
		Tempo	#Iterações	*Fitness*	Tempo	#Iterações	*Fitness*	Tempo	#Iterações	*Fitness*
f_1	(8,2)	35,6	639	0,000035	48,4	666	0,000047	118,2	746	0,000070
	(16,4)	85,3	838	0,000049	104,8	845	0,000067	125,4	685	0,000068
	(32,8)	130,6	1028	0,000054	203,8	1054	0,000085	248,5	740	0,000076
	(64,16)	186,8	1282	0,000064	566,7	1325	0,000091	273,0	892	0,000087
	(128,32)	433,2	1613	0,000072	1377,0	1681	0,000095	407,2	1098	0,000094
	(256,64)	1676,1	2069	0,000082	2926,1	2160	0,000097	756,9	1392	0,000095
	(512,128)	7610,0	2765	0,000088	-	-	-	1981,5	1810	0,000097
	(1024,256)	42966,6	4061	0,000094	-	-	-	8782,9	2371	0,000098
f_2	(8,2)	34,4	550	0,000046	46,8	444	0,000047	111,4	721	0,000070
	(16,4)	88,5	855	0,000054	113,1	751	0,000069	125,6	653	0,000071
	(32,8)	142,0	1093	0,000062	198,9	971	0,000085	256,0	708	0,000077
	(64,16)	222,5	1402	0,000068	431,3	1253	0,000095	278,4	907	0,000086
	(128,32)	523,9	1893	0,000083	1080,2	1688	0,000096	439,3	1131	0,000093
	(256,64)	2475,1	2903	0,000092	2927,5	2512	0,000099	854,8	1474	0,000097
	(512,128)	15731,4	5409	0,000096	-	-	-	2320,3	2042	0,000099
	(1024,256)	68313,8	6000	0,731999	-	-	-	12920,3	3249	0,000099
f_3	(8,2)	67,8	570	0,000051	44,9	559	0,000046	116,7	702	0,000061
	(16,4)	92,2	891	0,000051	110,7	1095	0,019968	123,8	642	0,000063
	(32,8)	440,5	3434	0,258714	483,0	4391	0,477685	255,2	679	0,000076
	(64,16)	1016,9	6000	7,396041	1583,2	6000	8,066589	338,3	1100	0,000077
	(128,32)	1922,8	6000	25,273052	4673,8	6000	26,141418	2098,9	5111	0,815935
	(256,64)	5992,7	6000	65,219309	12362,0	6000	69,860478	4072,0	6000	37,640721
	(512,128)	21201,3	6000	172,577285	-	-	-	8727,5	6000	118,219038
	(1024,256)	83212,5	6000	425,916001	-	-	-	29047,9	6000	306,605380
f_4	(8,2)	81,9	818	0,000037	45,7	920	0,000038	109,9	824	0,000050
	(16,4)	87,5	845	0,000044	108,3	882	0,000040	151,5	768	0,000044
	(32,8)	107,9	849	0,000036	201,3	911	0,000037	271,9	729	0,000043
	(64,16)	146,2	873	0,000036	232,1	915	0,000048	267,4	840	0,000048
	(128,32)	266,1	893	0,000040	686,3	946	0,000037	331,2	783	0,000044
	(256,64)	839,5	926	0,000036	1806,6	968	0,000037	569,1	838	0,000043
	(512,128)	3039,6	949	0,000034	-	-	-	1396,3	860	0,000044
	(1024,256)	11463,8	911	0,000032	-	-	-	4664,4	752	0,000039

Tabela 13: *Speedup real* e *speedup relativo* entre as implementações paralelas em OpenMP com MPICH para f_1, f_2, f_3 e f_4 referente a Figura 38

Função	Teste	*Speedup real*			*Speedup relativo*		
		PPSO	PDPSO	CPPSO	PPSO	PDPSO	PCPSO
f_1	(8,2)	0,05	0,04	0,02	0,05	0,04	0,02
	(16,4)	0,10	0,08	0,07	0,10	0,08	0,06
	(32,8)	0,29	0,19	0,15	0,29	0,19	0,11
	(64,16)	1,00	0,33	0,69	0,99	0,34	0,47
	(128,32)	2,21	0,70	2,35	2,16	0,71	1,56
	(256,64)	2,89	1,65	6,39	2,80	1,68	4,18
	(512,128)	3,43	-	13,19	3,34	-	8,41
	(1024,256)	3,57	-	17,45	3,48	-	9,94
f_2	(8,2)	0,05	0,03	0,01	0,05	0,03	0,02
	(16,4)	0,11	0,09	0,08	0,11	0,08	0,06
	(32,8)	0,34	0,24	0,19	0,34	0,21	0,12
	(64,16)	1,14	0,59	0,91	1,10	0,51	0,57
	(128,32)	2,67	1,29	3,18	2,53	1,10	1,81
	(256,64)	3,67	3,10	10,63	3,35	2,45	4,93
	(512,128)	4,40	-	29,80	4,02	-	10,29
	(1024,256)	5,05	-	26,72	5,05	-	14,47
f_3	(8,2)	0,03	0,05	0,02	0,03	0,05	0,02
	(16,4)	0,16	0,13	0,12	0,15	0,16	0,08
	(32,8)	0,53	0,48	0,92	0,47	0,55	0,16
	(64,16)	1,52	0,97	4,56	1,52	0,97	0,84
	(128,32)	3,27	1,35	3,00	3,27	1,35	2,56
	(256,64)	4,31	2,09	6,34	4,31	2,09	6,34
	(512,128)	5,00	-	12,15	5,00	-	12,15
	(1024,256)	5,15	-	14,75	5,15	-	14,75
f_4	(8,2)	0,05	0,09	0,04	0,05	0,11	0,04
	(16,4)	0,19	0,15	0,11	0,19	0,16	0,10
	(32,8)	0,60	0,32	0,24	0,60	0,35	0,20
	(64,16)	1,76	1,11	0,96	1,78	1,18	0,94
	(128,32)	3,99	1,55	3,21	3,97	1,63	2,80
	(256,64)	5,15	2,39	7,59	5,19	2,52	6,93
	(512,128)	5,77	-	12,56	5,87	-	11,59
	(1024,256)	6,04	-	14,85	5,94	-	12,05

Tabela 14: Tempo de execução (ms), número de iterações e *fitness* entre as implementações paralelas em CUDA para f_1, f_2, f_3 e f_4 referente as Figuras 39 e 40

Função	Teste	PPSO			PDPSO			CPPSO		
		Tempo	#Iterações	*Fitness*	Tempo	#Iterações	*Fitness*	Tempo	#Iterações	*Fitness*
f_1	(8, 2)	28,0	641	0,000040	26,6	664	0,000028	20,7	474	0,000041
	(16, 4)	36,0	833	0,000046	33,7	821	0,000042	29,9	688	0,000047
	(32, 8)	47,1	1042	0,000054	44,2	1042	0,000058	33,7	751	0,000042
	(64, 16)	65,0	1302	0,000064	59,3	1308	0,000067	42,8	911	0,000057
	(128, 32)	214,8	1653	0,000068	105,4	1658	0,000071	62,4	1111	0,000064
	(256, 64)	503,0	2128	0,000078	269,6	2131	0,000079	182,3	1400	0,000072
	(512, 128)	2294,8	2838	0,000088	1113,0	2840	0,000087	782,0	1823	0,000072
	(1024, 256)	12849,5	4162	0,000092	5700,8	4159	0,000093	2912,9	2398	0,000085
f_2	(8, 2)	24,2	561	0,000040	18,2	446	0,000041	21,5	473	0,000056
	(16, 4)	36,9	850	0,000055	30,3	746	0,000055	28,7	665	0,000051
	(32, 8)	52,2	1127	0,000063	40,8	961	0,000066	32,3	716	0,000054
	(64, 16)	74,2	1444	0,000072	56,3	1239	0,000072	43,5	913	0,000057
	(128, 32)	279,7	1983	0,000082	107,3	1665	0,000080	69,0	1142	0,000061
	(256, 64)	831,9	3127	0,000092	315,4	2468	0,000091	201,8	1470	0,000069
	(512, 128)	5519,3	5861	0,000097	1753,2	4479	0,000097	909,1	2022	0,000080
	(1024, 256)	20655,4	6000	1,444619	8219,4	6000	0,012294	4232,3	3263	0,000093
f_3	(8, 2)	26,2	586	0,000041	22,7	557	0,000039	15,7	355	0,000045
	(16, 4)	42,0	925	0,000046	45,7	1082	0,019939	28,1	630	0,000048
	(32, 8)	210,6	4466	0,557241	183,3	4310	0,477656	33,2	692	0,000053
	(64, 16)	361,4	6000	8,474654	276,8	6000	8,066589	64,6	1239	0,039838
	(128, 32)	940,5	6000	25,836995	391,5	6000	26,141418	408,4	5029	0,573702
	(256, 64)	1797,2	6000	67,383505	793,4	6000	69,284188	1043,6	6000	36,523101
	(512, 128)	5618,6	6000	187,207345	2437,1	6000	175,617787	3073,2	6000	116,216035
	(1024, 256)	21007,1	6000	485,598184	7962,2	6000	198,442000	8541,6	6000	311,017255
f_4	(8, 2)	40,3	910	0,000046	37,9	912	0,000034	29,1	656	0,000044
	(16, 4)	39,5	865	0,000038	37,0	875	0,000037	36,3	818	0,000042
	(32, 8)	41,7	890	0,000041	39,5	904	0,000028	35,5	768	0,000037
	(64, 16)	50,6	904	0,000044	42,6	905	0,000041	41,2	842	0,000041
	(128, 32)	129,7	932	0,000043	62,1	938	0,000030	60,0	840	0,000045
	(256, 64)	234,1	938	0,000036	126,9	960	0,000031	129,8	858	0,000036
	(512, 128)	733,7	941	0,000040	404,1	973	0,000025	445,1	990	0,000014
	(1024, 256)	2948,6	975	0,000044	1441,9	952	0,000027	1085,6	900	0,000038

Tabela 15: *Speedup real* e *speedup relativo* das implementações paralelas em CUDA para f_1, f_2, f_3 e f_4 referente a Figura 41

Função	Teste	*Speedup real*			*Speedup relativo*		
		PPSO	PDPSO	CPPSO	PPSO	PDPSO	CPPSO
f_1	(8,2)	0,06	0,07	0,09	0,06	0,07	0,06
	(16,4)	0,24	0,25	0,29	0,24	0,25	0,23
	(32,8)	0,81	0,86	1,13	0,81	0,86	0,81
	(64,16)	2,89	3,17	4,38	2,88	3,17	3,06
	(128,32)	4,46	9,10	15,37	4,46	9,12	10,33
	(256,64)	9,61	17,93	26,53	9,61	17,95	17,44
	(512,128)	11,39	23,47	33,41	11,38	23,48	21,45
	(1024,256)	11,92	26,88	52,60	11,92	26,86	30,30
f_2	(8,2)	0,06	0,09	0,07	0,07	0,07	0,06
	(16,4)	0,26	0,32	0,34	0,27	0,28	0,27
	(32,8)	0,92	1,18	1,49	0,94	1,03	0,97
	(64,16)	3,43	4,52	5,86	3,39	3,83	3,66
	(128,32)	5,00	13,02	20,26	4,97	10,88	11,61
	(256,64)	10,93	28,82	45,05	10,74	22,36	20,82
	(512,128)	12,53	39,44	76,05	12,41	29,87	26,01
	(1024,256)	16,72	42,01	81,58	16,72	42,01	44,37
f_3	(8,2)	0,09	0,10	0,15	0,09	0,10	0,09
	(16,4)	0,34	0,32	0,51	0,35	0,38	0,36
	(32,8)	1,11	1,28	7,04	1,28	1,42	1,26
	(64,16)	4,27	5,57	23,88	4,27	5,57	4,93
	(128,32)	6,70	16,08	15,42	6,70	16,08	12,92
	(256,64)	14,36	32,53	24,73	14,36	32,53	24,73
	(512,128)	18,87	43,50	34,50	18,87	43,50	34,50
	(1024,256)	20,40	53,81	50,16	20,40	53,81	50,16
f_4	(8,2)	0,11	0,11	0,15	0,12	0,13	0,12
	(16,4)	0,42	0,45	0,45	0,44	0,48	0,46
	(32,8)	1,55	1,64	1,82	1,63	1,75	1,65
	(64,16)	5,10	6,06	6,27	5,34	6,34	6,11
	(128,32)	8,19	17,11	17,70	8,50	17,87	16,56
	(256,64)	18,46	34,06	33,27	18,84	35,59	31,07
	(512,128)	23,90	43,39	39,39	24,12	45,30	41,85
	(1024,256)	23,49	48,03	63,79	24,72	49,36	61,97