UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS
FACULDADE DE COMPUTAÇÃO
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

GEOVANI OLIVEIRA CABRAL DA PAZ

O USO DE ACELERADORES GRÁFICOS APLICADOS AO MODELO XCALABLEMP PARA A PARALELIZAÇÃO DE ALGORITMOS GENÉTICOS

Belém
Março/2017

UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS
FACULDADE DE COMPUTAÇÃO
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

GEOVANI OLIVEIRA CABRAL DA PAZ

O USO DE ACELERADORES GRÁFICOS APLICADOS AO MODELO XCALABLEMP PARA A PARALELIZAÇÃO DE ALGORITMOS GENÉTICOS

Trabalho de Conclusão de Curso apresentado como parte dos requisitos necessários para obtenção do título de Bacharel em Sistemas de Informação da Faculdade de Computação da Universidade Federal do Pará.

Orientador: Prof. Dr. Josivaldo de Souza Araújo

Belém
Março/2017

Geovani Oliveira Cabral da Paz
O USO DE ACELERADORES GRÁFICOS APLICADOS AO MODELO XcalableMP PARA A PARALELIZAÇÃO DE ALGORITMOS GENÉTICOS/ Geovani Oliveira Cabral da Paz. – Belém, Março/2017.
68 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Josivaldo de Souza Araújo

Monografia – Universidade Federal do Pará
Instituto de Ciências Exatas e Naturais
Curso de Bacharelado em Sistemas de Informação, Março/2017.

1. Computação em GPU. 2. Programação Paralela. 3. XcalableMP. 4. Algoritmos Genéticos I. Título.

GEOVANI OLIVEIRA CABRAL DA PAZ

O USO DE ACELERADORES GRÁFICOS APLICADOS AO MODELO XCALABLEMP PARA A PARALELIZAÇÃO DE ALGORITMOS GENÉTICOS

Trabalho de Conclusão de Curso apresentado como parte dos requisitos necessários para obtenção do título de Bacharel em Sistemas de Informação da Faculdade de Computação da Universidade Federal do Pará.

Data da Defesa: 09 de Março de 2017
Conceito:

Banca Examinadora

Prof. Dr. Josivaldo de Souza Araújo
Faculdade de Computação - ICEN / UFPA
Orientador

Prof. Dr. Claudomiro de Souza de Sales Jr
Faculdade de Computação - ICEN / UFPA
Membro da Banca

Esdras Roberto Alves de La Roque
UFPA / FADESP
Membro da Banca

Belém
Março/2017

Dedico este trabalho à minha mãe, meu pai e a todos que me ajudaram e me incentivaram a chegar até aqui.

AGRADECIMENTOS

Primeiramente agradeço a Deus, por tudo que me deu e que ainda vai me dá. Aos meus Pais que me ensinaram o caminho certo, sempre me apoiaram e me deram força e acreditaram em mim.

Ao meu Professor Orientador Dr. Josivaldo de Souza Araújo pela ótima experiência, pela oportunidade que me deu de aprender mais sobre os temas relacioados ao trabalho, pela paciência e por todo tipo possível de ajuda que me deu.

Ao meu amigo Esdras La Roque, pela imensa ajuda que me deu durante toda fase de montagem e projeto, pela disponibilidade e pelos aprendizados que me passou.

Ao meu colega Reginaldo Santos, por me ceder os códigos das versões sequenciais dos algoritmos, e por toda ajuda na compreensão do assunto e do algoritmo.

Aos demais professores, funcionários e à diretoria da Faculdade de Computação que contribuíram para minha formação.

À Professora Marcele Pereira por toda a ajuda e paciência enquanto fui seu subordinado.

E a todos os meus colegas e amigos de turma, pela amizade e companheirismo durante todos esses anos.

“Há uma força motriz mais poderosa que o vapor, a eletricidade e a energia atômica: a vontade”

(Albert Einstein)

RESUMO

Este trabalho tem como objetivo apresentar e avaliar uma estrutura de computação de clusters de aceleradores gráficos utilizando uma proposta de modelo de programação em memória distribuída, o XcalableMP. A programação paralela em memória distribuida geralmente, tira proveito, da divisão do trabalho de computação entre as CPUs do sistema, usando mecanismos de troca de mensagens como o MPI, porém desde o surgimento e eventual crescimento da computação em GPU novas possibilidades surgiram, também, no sentido de organizar máquinas distribuídas, equipadas com GPUs em ambientes de computação paralela, visando assim obter vantagens de um sistema híbrido composto por CPUs e GPUs, principalmente no significativo ganho de desempenho computacional. Assim, o presente trabalho utiliza uma arquitetura de cluster de aceleradores gráficos com objetivo de obter ganhos computacionais na solução de funções de algoritmos genéticos. O modelo XcalableMP foi usado como o gerenciador de processos em memória distribuída e o OpenACC como modelo de programação em GPU, formando assim, toda a estrutura de habilitação de programação híbrida. Os algoritmos genéticos foram executados e testados, destancando com isso, o ganho de desempenho computacional na execução das funções *fitness*, comparando-se as execuções sequenciais em CPU e execuções utilizando uma e duas GPUs.

Palavras-chave: Computação em GPU.Programação Paralela. XcalableMP. Algoritmos Genéticos, OpenACC.

ABSTRACT

This work aims to present a computational structure of clusters of graphic calipers using a recent distributed memory programming model, XcalableMP. Distributed memory parallel programming often takes advantage of division of computing work among system CPUs, often using message exchange mechanisms such as MPI, but since the discovery and growth of GPU computing new possibilities have also emerged in the sense of To organize machines equipped with GPUs in parallel computing environments, in order to obtain the main advantages of both, mainly in the significant gain of computational performance. Thus, the work developed a cluster architecture of graphic accelerators with the objective of obtaining computational gain in the execution of genetic algorithms. The XcalableMP model was used as the distributed memory process manager, with OpenACC as the GPU programming model forming the entire framework of hybrid programming enablement. The genetic algorithms were executed and tested by untangling the computational performance gain in execution in this structure when compared to the sequential execution in CPU and execution in only one GPU.

Keywords: computation in GPU. parallel programming. XcalableMP. genetic algorithms.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AE	Algoritmo Evolucionário
AG	Algoritmos Genéticos
ALU	*Arithmetic Logic Unit*
API	*Application Programming Interface*
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
FPU	*Float Point Unit*
GPU	*Graphics Processing Unit*
GPGPU	*General Purpose Graphics Processing Unit*
HPC	*High Performance Computing*
MPI	*Message Passing Interface*
PGAS	*Partitioned Global Address Space*
SM	*Streams Multiprocessors*
SIMT	*Single Instructions Multiple Threads*
SFU	*Special Function Unit*
XMP	*XcalableMP*

SUMÁRIO

1 INTRODUÇÃO

1.1 Visão Geral.

Nas últimas décadas, muito se tem evoluído na construção de sistemas que possam executar tarefas de forma paralela. Entre os mais conhecidos estão os clusters e as grades de computadores. No entanto, para se ter um número razoável de processadores é necessário reunir alguns computadores e interligá-los em rede, o que muitas vezes, pode ser dificultado pela falta de uma estrutura própria. Porém, nos últimos anos, essa evolução na forma de se programar paralelamente, chegou aos aceleradores gráficos e com isso, puderam-se construir sistemas equipados com Unidades de Processamento Gráfico(GPU), que passaram a ser utilizados como um ótimo ambiente computacional para se obter alto desempenho para aplicações computacionais. Sua estrutura favorece a obtenção de alto desempenho computacional, pois é composta de vários núcleos de processamento e uma memória com alta largura de banda. Além disso, tais clusters se mostraram muito eficientes no consumo de energia elétrica, fato evidenciado nos dados do Green500, em junho de 2014, onde um ranking aponta os supercomputadores com maior eficiência no consumo de energia elétrica. Nessa listagem os clusters com aceleradores gráficos ocuparam as 17 primeiras posições (NAKAO et al., 2014).

Esses tipos de cluster surgiram da ideia de obter um maior desempenho a partir das estruturas de cluster tradicionais. Esse desempenho seria alcançado unindo as vantagens da computação paralela em memória distribuída com o processamento em placas gráficas, a computação em GPU. Os tipos de clusters utilizados foram os clusters do tipo Beowulf. Esses clusters são estruturas de computação paralela composta de microcomputadores comuns, usando componentes de baixo custo e amplamente disponíveis no mercado. Esses computadores são conectados por uma rede local, geralmente ethernet, de forma a montar uma estrutura com memória distribuída. Essas estruturas foram concebidas e são amplamente usadas para dar suporte às aplicações com processamento paralelo.

Processamento paralelo por sua vez, foi o paradigma de programação desenvolvido como alternativa ao tradicional modelo sequencial de processamento. Os modelos de programação sequencial segue um fluxo tradicional de execução, ou seja, todo programa é executado instrução após instrução seguindo uma ordem definida. O problema é que alguns programas, por exigirem processamento de grande fluxos de instruções e/ou dados, demandam recursos computacionais maiores. Se esses recursos não forem o suficiente, a execução de tais algoritmos se torna ineficiente, levando um tempo muito grande de processamento e podendo causar danos ao sistema.

Assim, a computação paralela trouxe a abordagem da divisão e conquista para o meio computacional. Com esse modelo um problema computacional pode ser dividido em partes menores para ser executado. Dependendo do grau de paralelismo empregado, cada parte de

um programa pode ser atribuído a uma unidade de processamento diferente, podendo estar em máquinas com memória distribuída ou compartilhada.

A computação em GPU, ou GPGPU (*General Purpose Graphics Processing Unit*), surgiu no começo dos anos 2000, quando pesquisadores analisando as estruturas das placas gráficas, notaram que elas poderiam também ser usadas para computação de propósito geral (SANDERS; KANDROT, 2010). Essas placas que antes só eram usadas exclusivamente para processamento gráfico, se mostraram capazes de realizar operações simples dentro de pequenas unidades (núcleos) que as compõe. Sendo assim, cada uma dessas unidades representam pequenos processadores, e como essas placas podem conter até centenas desses pequenos núcleos, elas se tornam uma ótima alternativa para obter grande aumento de desempenho em relação ao processamento tradicional, em CPU.

Entretanto, para o desenvolvimento de aplicações para esse tipo de estrutura, também, são necessários softwares, bibliotecas, ferramentas e tecnologias específicas. Essas tecnologias são essenciais tanto para habilitar a programação paralela, como também para obter maior eficiência no desempenho das aplicações, enquanto que outras ferramentas também contribuem para tornar a tarefa de aprendizado e programação menos custosa aos programadores.

Para programação em GPU, o modelo de programação CUDA (*Computer Unified Device Architecture*) introduzido pela NVIDIA ganhou bastante notoriedade quando ela desenvolveu placas gráficas com arquitetura mais adequada a programação. Mais adiante o modelo de programação OpenACC se destacou por fornecer boa produtividade e alto desempenho, baixos custos de aprendizado, reescrita e reajustes, além de poder ser usada para programação em diferentes modelos de placas gráficas, não sendo exclusividade de apenas uma fabricante como, por exemplo, o CUDA que é um modelo apenas voltado para as placas aceleradoras da fabricante NVIDIA.

Como cluster são estruturas de memória distribuídas conectadas em rede, mecanismos de comunicação entre as máquinas são necessárias para realizar toda a movimentação de dados, atribuição de computação, distribuição de tarefas e sincronização de processos. Por muito tempo essas tarefas eram realizadas, e ainda são na maioria das vezes, através de mecanismos de troca de mensagens, sendo que o mais utilizado é o MPI (*Message Passing Interface*). Porém a programação com MPI se mostrou muito custosa em alguns aspectos, um deles é que sua curva de aprendizado é grande, outro motivo é que as próprias funções MPI são consideradas primitivas o que leva a um descompasso com as facilidades que as linguagens mais atuais proporcionam, como o OpenACC.

Portanto, com o objetivo de simplificar, principalmente diminuir a curva de aprendizagem com o MPI, foi desenvolvido o ambiente XcalableMP (XMP), um modelo de linguagem PGAS (*Partitioned Global Address Space*) baseado em diretivas. O XMP suporta modelos de programação PGAS de visão global e visão local. No modelo de memória de visão global, que é o que será utilizado neste trabalho, um programador pode usar diretivas XMP simples para

paralelizar um código fonte sequencial entre várias máquinas. O XMP também oferece suporte para programação híbrida e pode incorporar inclusive diretivas do OpenACC, possibilitando assim também ser usada em programação em GPU. Assim, levando em consideração o alto desempenho que essas arquiteturas podem proporcionar, foram implementados e testados neste trabalho, algoritmos genéticos (funções *fitness*) que exigem grande carga computacional.

Para se obter um resultado satisfatório, a execução de todas as etapas de um algoritmo genético deve ser repetida várias vezes, fato que leva o consumo considerável de recursos computacionais, como espaço em memória e tempo de processamento em CPU. Isso faz com que muitas das vezes a execução de um algoritmo, desse tipo, leve muitos minutos e, às vezes, até horas para terminar a sua execução. Por isso, a idéia proposta foi a adaptação desses algoritmos para estruturas paralelas híbridas, na intenção de se obter um ganho significativo de desempenho, fazendo assim que o tempo de execução seja reduzido.

1.2 Objetivos.

1.2.1 Geral.

O objetivo geral do trabalho é montar uma arquitetura computacional de memória distribuída heterogênea. Essa arquitetura contém duas máquinas interligadas por rede local e são equipadas com aceleradores gráficos. Após a montagem da infraestrutura, o poder da computação heterogênea é testado visando obter um ganho de desempenho na execução de algoritmos genéticos.

1.2.2 Objetivos Específicos.

- Montar uma arquitetura de cluster com aceleradores gráficos.
- Utilizar modelos de programação que habilitem a computação em GPU nas máquinas da arquitetura montada, e utilizar ferramentas que facilitem a comunicação entre as máquinas e possibilitem a programação paralela.
- Desenvolver, a partir de uma versão sequencial (CPU), versões de algoritmos genéticos adaptados a computação em uma e duas GPUs.
- Testar e avaliar cada umas das versões dos algoritmos genéticos, destacando as melhores soluções de cada um deles e seus respectivos tempos de processamento.

1.3 Trabalhos relacionados.

Alguns trabalhos foram identificados com o objetivo de avaliar o desempenho de cluster de aceleradores gráficos com aplicações específicas e também trabalhos relacionados ao uso de

modelos de linguagem PGAS.

Em (MORAES, 2012), são apresentados conceitos gerais de programação paralela e computação em cluster Beowulf. Além disso, neste trabalho é realizado um experimento com a montagem de um cluster beowulf de GPUs com comunicação por passagem de mensagem MPI. Ao final é feita uma avaliação de uma aplicação para simulação do transporte de nêutrons, através de uma blindagem por meio do Método Monte Carlo, nesta avaliação a execução do algoritmo utilizando mais de uma GPU se mostrou mais rápida do que nas versões sequenciais e na versão com apenas uma GPU.

Em (LEE; SATO, 2010), o modelo de programação XcalableMP é apresentado e suas principais características são introduzidas. O XMP é mostrado como uma alternativa viável e mais eficiente em termos de simplificação de programação que o MPI, pois utiliza diretivas como base para sua programação assim como o OpenMP. Neste trabalho também é testada uma implementação de um algoritmo HPCC Benchmark para avaliação de desempenho. Ao final é provado que programadores podem paralelizar códigos sequenciais usando diretivas simples do XMP.

Em (NAKAO et al., 2014), é introduzido o modelo de programação XcalableACC (XACC) para clusters de GPUs. O trabalho apresenta a relação entre os modelos XcalableMP e OpenACC e como eles interagem com um código sequencial. Testes de produtividade e desempenho são realizados através da execução de algoritmos de *Benchmark* HIMENO para comparar as versões implementadas com XACC e versões implementadas com MPI e OpenACC. Ao final as versões com XACC se mostraram mais rápidas e produtivas sendo possível até mesmo serem escritas com metade de linhas que a versão com MPI.

1.4 Proposta.

A proposta desse trabalho é a montagem de um ambiente heterogêneo de computação paralela com dois computadores equipados com aceleradores gráficos. Com essa estrutura objetiva-se obter maior poder computacional através da programação paralela. E para avaliar o desempenho computacional, algumas implementações de algoritmos genéticos foram executadas e testadas, a fim de comparar o tempo de processamento das versões das implementações desses algoritmos.

O trabalho também propõe utilizar modelos de programação mais simples, em termos de programação, para GPUs, assim como modelos de mais fácil aprendizado. Além disso, é proposto a utilização de uma tecnologia alternativa de comunicação para sistemas de memória distribuída, tanto para clusters tradicionais quanto para cluster de aceleradores.

1.5 Organização do texto.

Este trabalho, além dos capítulos referentes a Introdução e Conclusão, encontra-se organizado da seguinte forma:

No capítulo 2, apresenta os conceitos relacionados a computação paralela, mostrando suas classificações e conceitos. Também é introduzido o conceito de ambientes heterogêneos, assim como os conceitos e fundamentos da computação em GPU, além da descrição do ambiente (modelo) de programação XMP.

No capítulo 3, são apresentados os principais conceitos de Algoritmos Genéticos, mostrando suas características gerais e específicas. Destaca-se também, a computação evolucionária e as possíveis classificações de algoritmos genéticos paralelos.

No capítulo 4, são apresentadas as funções *fitness* implementadas utilizando algoritmos genéticos, visando-se avaliar o ganho de desempenho de cada implementação, bem como os resultados e gráficos de tais experimentos.

2 PROGRAMAÇÃO PARALELA

2.1 Considerações Iniciais.

A computação paralela é a técnica pela qual um problema computacional é dividido em unidades menores para serem executadas de forma paralela e sincronizadas. A execução paralela requer também arquiteturas paralelas de computação com componentes adequados ao paralelismo como processadores e mais recentemente placas gráficas. Essas arquiteturas variam em termo de organização da memória que podem ser do tipo compartilhada ou distribuída. Nessas unidades, um programa maior e mais complexo é dividido em partes menores e cooperarem entre si para uma execução mais eficiente e veloz. O objetivo da divisão dessa programação é aumentar o poder de computação e, com isso, executar programas mais rapidamente, sendo muito útil à aplicações que processam grandes quantidades de dados, pois se fossem executadas sequencialmente, acabariam levando muito tempo.

A demanda por computação de alto desempenho foi o grande propulsor de pesquisas para soluções paralelas de computação. Aplicações científicas, meteorológicas, biomédicas, genéticas, mecânicas e espaciais entre outras, precisam de alto poder computacional, poder esse, muitas vezes limitados pelas arquiteturas tradicionais de computadores. Ao longo dos anos, muito se tentou melhorar as tecnologias das *Central Processing Unit* (CPU), o que a até certo ponto houve um grande incremento de suas capacidades, graças a avanços significativos no aumento na velocidade de *clock* e do advento da tecnologia multicore. Entretanto, apesar de ainda ser possível melhorar o desempenho dos processadores, a atual tecnologia dos processadores já começou a se aproximar de um limite físico para um maior incremento dessa velocidade. Algumas restrições tem freado a busca de maiores frequências de processamento, como problemas relacionados à dimensão física dos componentes, a dissipação de calor, velocidade da luz e o custo, que poderá se revelar excessivo com a aproximação desses limites (MORAES, 2012).

Com o propósito de criar computadores mais rápidos e poderosos, surge na década de 70 as primeiras inovações voltadas para supercomputação. Com objetivo de atender essa demanda para aplicações em grande escala de dados, os supercomputadores tinham grandes dimensões pois eram mais complexos e precisavam de um maior incremento de componentes para obter o desempenho esperado. Por outro lado, surgiu também, a abordagem de tentar obter alto desempenho por meio da paralelização computacional, dentre elas destaca-se as estruturas de computação com múltiplos processadores e a estrutura com multicomputadores.

2.2 Classificação de Arquiteturas.

Muito antes das arquiteturas paralelas, os computadores seguiam um modelo clássico de arquitetura, o chamado Modelo de Von Neumamm. Essa tipo de arquitetura é também conhecida

como modelo convencional, ela executa uma instrução sobre um dado de cada vez, ou seja, existe um fluxo de instruções para um fluxo de dados (MORALES, 2008). Fisicamente, ela consiste de memória principal, uma CPU, e uma interconexão entre a memoria e a CPU.

2.2.1 Taxonomia de Flynn.

Em 1966, Michael Flynn propôs um modelo de classificação para arquiteturas de computadores paralelos mais conhecidos e aceitos até hoje. A grande aceitação dessa taxonomia pode ser explicada pelo fato da mesma ser simples, de fácil entendimento e por fornecer uma compreensão próxima da realidade. O esquema classifica os computadores pela maneira como instruções e dados são organizados em um determinado tipo de processamento (MORAES, 2012). O tipos definidos na taxonomia de Flynn relaciona fluxo de dados, que significa um conjunto de dados, e "fluxo de instruções", que indica um conjunto de instruções a serem executadas (MORALES, 2008).

2.2.1.1 Single Instruction Single Data - SISD.

É o modelo mais comum e que compõe a maioria dos computadores atualmente, nesse modelo, inclusive, que se enquadra a arquitetura proposta por Von Neumann. Ele pode ser definido como composto por um único fluxo de instruções que processa um único fluxo de dados, como mostra a Figura 1. Devido essas características não pode ser considerado uma arquitetura de processamento paralelo.

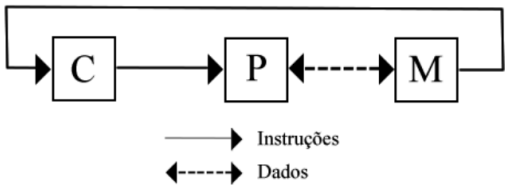

Figura 1 – Modelo SISD
(MORALES, 2008)

2.2.1.2 Multiple Instruction Single Data - MISD.

Nesse tipo de modelo, múltiplas instruções manipulam apenas um conjunto de dados. Sendo assim diferentes instruções são executadas sobre uma mesma posição de memória, a Figura 2 mostra esse modelo. Pode ser considerada uma arquitetura paralela, mesmo que seu modelo de execução seja considerado impraticável e sem sentido (MORALES, 2008).

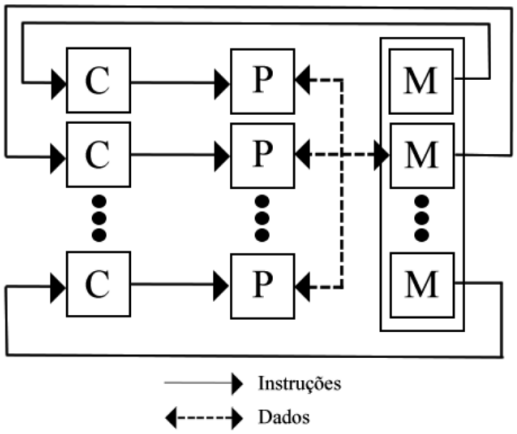

Figura 2 – Modelo MISD
(MORALES, 2008)

2.2.1.3 Single Instruction Multiple Data - SIMD.

Pode-se dizer que essa arquitetura foi o precursor dos multiprocessadores. O processador opera de modo que uma única instrução em paralelo acessa e manipula um conjunto de dados, conforme a Figura 3. Por essas características também são chamados de processadores vetoriais e matriciais.

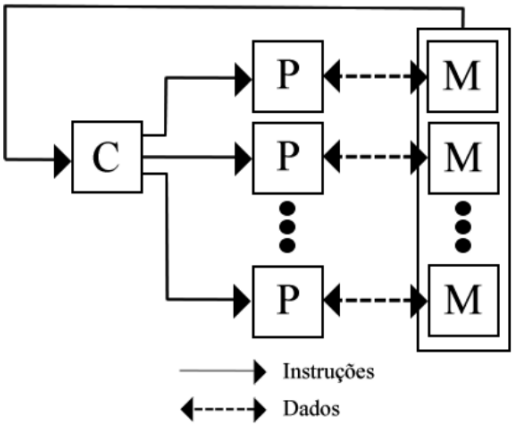

Figura 3 – Modelo SIMD
(MORALES, 2008)

2.2.1.4 Multiple Instruction Multiple Data- MIMD.

É a arquitetura mais tecnologicamente avançada, refere-se a computadores com múltiplos fluxos de instruções e múltiplos fluxos de dados. MIMD é a principal arquitetura para o propósito da computação paralela, sendo classificados pelo compartilhamento de memória, os sistemas com memória compartilhada , também conhecidos como multiprocessadores e sistemas de memória distribuída, conhecidos como multicomputadores. A Figura 4 ilustra esse modelo.

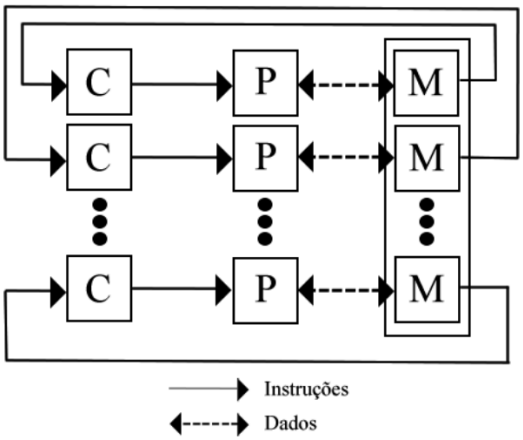

Figura 4 – Modelo MIMD
(MORALES, 2008)

2.3 Ambiente de Programação Paralela.

Segundo (ROCHA, 2003), um ambiente de programação paralela é definido pelas seguintes características: vários processadores interligados, plataforma para manipulação de processos paralelos, sistema operacional, linguagem de programação e modelos de programação paralela.

Existe, no modelo MIMD (apresentadas anteriormente), dois tipos de ambiente de programação paralela: ambientes multiprocessadores, com memória compartilhada e, ambientes multiprocessadores, com memória distribuída.

2.3.1 Multiprocessadores.

Sistemas multiprocessadores são maquinas paralelas formadas por vários processadores que compartilham um barramento comum. Nesse tipo de arquitetura, vários processadores dividem além do mesmo barramento, os sistemas de entrada e saída e especialmente a memória do sistema, ou seja, toda a memória é mapeada como uma só e fica disponível para todos os processadores de forma global (MORALES, 2008).

2.3.1.1 Memória compartilhada.

Os sistemas de memória compartilhada são máquinas com vários processadores, no qual todos podem acessar toda memória presente no sistema como um espaço de endereçamento global. A Figura 5 mostra um exemplo de configuração para um sistema de memória compartilhada onde cada processador é diretamente conectado a memória global. Esta configuração permite que múltiplos processadores operam independentemente enquanto ainda podem ter acesso a todos os recursos da memória. Um exemplo de sistema de memória compartilhada são os microcomputadores do tipo desktop.

As vantagens de usar paralelização em memória compartilhada é a maior facilidade na programação, já que é relativamente simples transformar códigos seriais em códigos paralelos

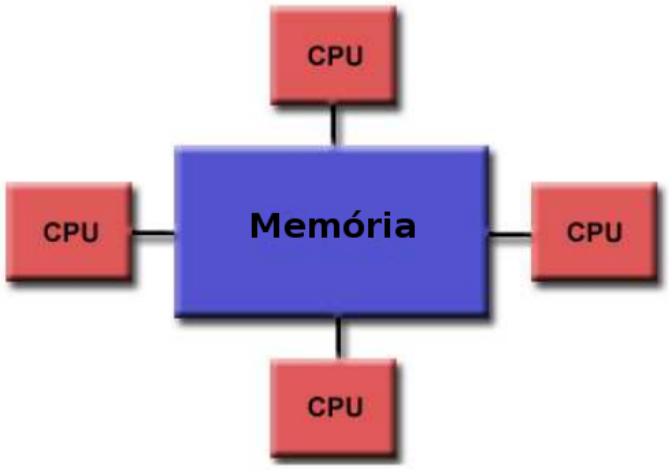

Figura 5 – Configuração de memória compartilhada
(KIESSLING, 2009)

nesses tipos de estrutura. Porém, existem algumas desvantagens nestes sistemas, uma delas é que com múltiplos núcleos e processadores acessando um única memória compartilhada simultaneamente, que pode deixar o acesso aos dados mais lento. Outro problema também está relacionado a escalabilidade, não se pode adicionar mais processadores e manter a eficiência sem aumentar a quantidade de memória disponível. Por fim, o programador é responsável por certificar que o código escrito na memória global é tratado corretamente. Este ponto é muito importante pois se múltiplos processadores estão lendo e escrevendo dados na memória simultaneamente, qualquer erro no código poderia causar resultados indesejados.

2.3.1.2 Memória Distribuída.

Em programação com memória distribuída, os núcleos podem acessar diretamente somente suas próprias memórias. Para acessar memórias remotas, movimentação de dados, distribuição de tarefas e sincronizações com outros núcleos são necessários mecanismos interconexão. Existem várias APIs usadas para esse tipo de programação. Contudo, de longe a mais amplamente usada é do tipo passagem de mensagem (PACHECO, 2011).

Nesses sistemas as mensagem são os elementos mais importantes, as mensagem são definidas como uma unidade lógica para comunicação entre processos. Cada mensagem é considerada como um conjunto de informações relacionadas enviadas como uma entidade (SILVA, 2006).

Quando uma aplicação é executada em um modelo de memória distribuída, o programa é dividido em tarefas paralelas e cada parte delas pode ser executada em processadores diferentes e são as mensagens que fazem todas atividades de comunicação entre os processadores.

Sistemas de memória distribuída tem algumas características marcantes em comum, como por exemplo comunicações por conexão de rede usados para conectar a memória dos

processadores. Essas redes de comunicação podem ser simples como uma conexão Ethernet ou mais sofisticadas. A Figura 6 mostra um exemplo de uma configuração de memória distribuída com cada processador tendo a sua própria memória local e, estão, conectados a cada um dos outros processadores através de algum tipo de rede.

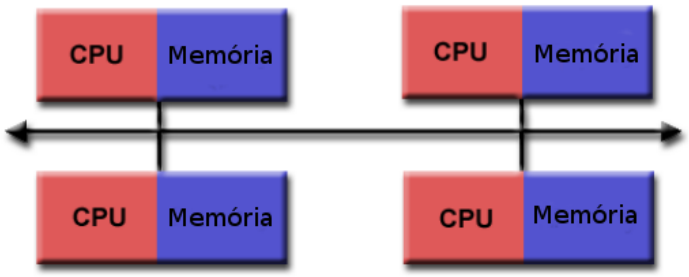

Figura 6 – Configuração de memória distribuída
(KIESSLING, 2009)

A vantagens de usar paralelização em memória distribuída é que a memória é escalável. Se houver um aumento no número de processadores, o tamanho da memória distribuída disponível também aumenta. Cada processador também acessa sua própria memória muito rapidamente do que o acesso a memórias remotas. A desvantagem é que é bem mais difícil de escrever um programa paralelo em memória distribuída e um programa escrito em série não é facilmente convertido para paralelo sem grandes modificações na reescrita no código (KIESSLING, 2009).

2.3.1.3 MPI.

O padrão MPI (*Message Passing Interface*) foi criado por uma comunidade de especialistas em computação de alto desempenho no começo da década de 90. O grupo era formado por cerca de 60 pessoas provenientes de organizações dos Estados Unidos e da Europa, unidos pelo objetivo de estabelecer um padrão de troca de mensagens que fosse largamente utilizado, portátil, eficiente e flexível.

Desde então surgiram várias versões do MPI. A primeira versão foi publicada em 1994, a MPI-1, que continha os requisitos básicos necessários para um padrão de troca de mensagens. Posteriormente, em 1997 foi lançada a versão MPI-2 que já continha melhoria em pontos mais complexos. A última atualização aconteceu em setembro de 2012, quando o grupo publicou o maior conjunto de atualizações no padrão, incluindo novas extensões, a remoção de *bindings* obsoletos para o C++ e o suporte ao FORTRAN 2008. Este último *release* foi nomeado como MPI-3.0 (COELHO, 2013).

Vale ressaltar que o MPI é uma especificação de interface, assim há no mercado várias implementações livres e pagas. Entre os modelos pagos pode-se destacar o Intel MPI, HP MPI e o MATLab MPI. Já entre as implementações *open source*, as mais utilizadas são o OpenMPI e MPICH.

O MPI é um padrão de troca de mensagens para programação paralela em memória distribuída. Este modelo assume que o hardware subjacente é um conjunto de processadores, cada um com sua própria memória local, e uma rede de interconexão de suporte à passagem de mensagens entre os processadores. A base do MPI é a comunicação entre processos em unidades geralmente com memórias separadas, como multicomputadores. Ele possui rotinas para comunicação ponto a ponto, operações coletivas, assim como suporte a grupos de tarefas, contextos de comunicação e topologias de aplicação (SILVA, 2006).

O MPI também fornece portabilidade entre diferentes plataformas. Por fornecer uma camada de abstração de alto nível, um código MPI pode ser executado em várias arquiteturas de hardware e sistemas operacionais, bastando apenas que a biblioteca MPI esteja disponível. Um código MPI também pode ser executado em sistemas de memória compartilhada, em uma rede de estações de trabalho, sistemas multicomputadores, como cluster de PCs ligados por rede local ou até mesmo em uma única máquina multiprocessada.

O MPI permite ao programador definir a forma adequada que seus algoritmos serão divididos e trocarão mensagens sem se preocupar com questões de mais baixo nível, como por exemplo, se uma mensagem foi devidamente recebida por outro processo, pois o MPI já garante essa verificação. Apesar do MPI disponibilizar as interfaces de abstração de alto nível ainda cabe ao programador a identificação de zonas paralelas no algoritmo e atividades como definição de processos, distribuição de dados, comunicação e sincronização.

2.4 Ambientes Heterogêneos.

Ambientes heterogêneos, pode-se assim dizer, pois utilizam tanto CPUs quanto GPUs para realizar o processamento das atividades.

A história das primeiras GPUs programáveis se deu início no começo dos anos 2000, quando começaram a ser projetadas placas com a finalidade de produzir uma cor por cada *pixel* na tela usando unidades aritméticas programáveis conhecidas como *pixel shaders*. Para produzir uma imagem completa, vários *pixels shader* usavam duas posições na tela (x,y), também usavam algumas informações adicionais que combinadas com outras entradas gerava assim uma cor final. As informações adicionais poderiam ser entradas de cores, coordenadas de textura, ou outro atributo que poderia ser passado para os *pixel shader*. Além disso, as operações aritméticas realizadas nas entradas de cores e texturas poderiam ser completamente controláveis pelo programador, fato que chamou a atenção de pesquisadores, que sugeriram a hipótese que essas entradas de cores poderiam ser substituídas por outro tipo de dado (SANDERS; KANDROT, 2010).

Assim se as entradas fossem dados numéricos significando qualquer coisa que não fossem apenas cores, programadores poderiam então programar os pixel shader para realizar qualquer tipo de computação com esses dados. Além do mais, por causa da alta taxa de cálculos

aritméticos que poderiam ser realizados nessas placas, os resultados inicias desses experimentos prometiam um futuro promissor para computação em GPU.

O uso de GPUs para processamento de aplicações não gráficas é conhecido como Computação em GPU de propósito geral ou simplesmente GPGPU. Tradicionalmente, as GPU sempre foram usadas para fornecer soluções para processamento gráfico, porém as tecnologias das placas evoluíram de tal forma que elas também se mostraram úteis para realizar operações matemáticas complexas, operações que também eram executadas com menores intervalos de tempo. Desde então a computação em GPU tem obtido uma enorme vantagem em velocidade de computação e desempenho sobre a computação tradicional em CPU. Graças a isso a GPGPU se tornou uma das áreas mais interessantes nos campos de pesquisa na indústria moderna.

GPUs são unidades de processamento gráfico que possibilitam a execução de gráficos de alta definição em PCs, o que corresponde a grande demanda da computação moderna. Assim como as CPUs, as GPUs tem formato de uma unidade de chip, contudo, as GPUs tem centenas de núcleos a mais que CPUs. As primeiras tarefas das GPUs eram exclusivamente as computações de funções gráficas. Devido esses tipos de cálculos serem muito pesados para CPUs, as GPU surgiram para ajudar a executar o processamento mais eficientemente, uma vez que possuíam capacidade de executar muito mais cálculos de ponto flutuante que as CPUs e tinham maior largura de banda (NVIDIA, 2012).

A razão pelo qual a capacidade de cálculos de ponto flutuante da GPU é maior que das CPU é que as GPUs foram desenvolvidas especialmente para computação intensiva e altamente paralela, como principal exemplo a renderização gráfica , e portanto as GPUs são constituídas com mais transistores dedicados ao processamento de dados em vez de elementos para cache de dados e controle de fluxo, como é esquematicamente ilustrado na Figura 7.

Figura 7 – Diferença entre CPU e GPU
(NVIDIA, 2012)

As GPUs são melhores adequadas para processar problemas que podem ser expressados através de computação de dados paralelos - o mesmo programa é executado em paralelo em vários elementos de dados com alta intensidade de cálculos aritméticos. Uma vez que o mesmo programa é executado em cada elemento de dado, existe uma menor necessidade de mecanismos sofisticados de controle de fluxo, além disso a baixa latência de acesso a memória permite que

grandes caches de dados não sejam necessárias, podendo assim ser acrescentadas mais unidades de processamento.

Um dos principais modelo de programação em GPU foi criado pela NVIDIA em 2006. Seu modelo de programação e arquitetura paralela é conhecido como CUDA e mudou a perspectiva inteira da programação GPGPU. A arquitetura CUDA tem como base as GPUs fabricadas pela própria NVIDIA. O CUDA também introduziu o modelo de programação híbrida, com execução de partes de código ora CPU ora GPU. Junto com CUDA outra grande inovação que a NVIDIA trouxe foi a fabricação de placas totalmente destinadas a computação de propósito geral em GPU, com componentes fabricados exclusivamente para esse objetivo.

Mais tarde, os modelos de programação evoluíram ainda mais, surgiam modelos menos complexos de codificação e menos dependentes de arquiteturas específicas de GPUs. O modelo de programação OpenACC é um exemplo disso. O OpenACC é uma linguagem de marcação de código baseado em diretivas de compilador. Essas diretivas são definidas para especificar loops e regiões de código nas linguagens C/C++ e Fortran. Essas diretivas são caracteres simples que são passadas do *host* (CPU) para o *device* (GPU) e tem a função de marcar as áreas que serão paralelizadas, sem necessidade dos programadores modificarem ou adaptarem o código para alguma arquitetura específica. Além disso, OpenACC fornece portabilidade entre vários sistemas operacionais e entre CPUs e GPUs.

2.4.1 CUDA.

A arquitetura de uma GPU é construída basicamente por *array* de *Streams Multiprocessors* (SM). Esses SMs por sua vez são compostos por vários núcleos CUDA, a quantidade desses núcleos variam de arquitetura para arquitetura. O paralelismo de hardware na GPU é alcançado através da replicação desta arquitetura em forma de blocos.

Cada SM em uma GPU é designado a suportar a execução paralela de centenas de *threads*. Existem geralmente múltiplos SM por GPU, assim é possível ter milhares de *threads* executando paralelamente em uma única GPU. Quando uma *grid* (conjunto de bloco de *threads*) é lançada, os blocos de *threads* da *grid* são distribuídos para execução entre os SMs disponíveis. Uma vez atribuídas a um SM, *threads* de um bloco de *threads* executam concorrentemente somente no SM atribuído a elas. Múltiplos blocos de *threads* podem ser atribuídos para os mesmos SMs de uma vez só, porém esses blocos são atribuídos baseados na disponibilidade dos recursos do SM.

CUDA emprega uma arquitetura *Single Instructions Multiple Threads* (SIMT) para gerenciar e executar grupos de 32 *threads* chamados *warp*. Todas as *threads* em um *warp* executam a mesma instrução e ao mesmo tempo. Cada *thread* tem seus próprios endereços de instrução, contadores e registradores de estado, e realizam as instruções correntes com seus próprios dados. Cada SM particiona os blocos de *threads* que lhe são atribuídos dentro warps 32 *threads*, então os programa para serem executados nos recursos de hardware disponíveis

(CHENG; GROSSMAN; MCKERCHER, 2014).

A arquitetura SIMT é similar a arquitetura SIMD (*Single Instructions Multiple Data*). Ambas implementam paralelismo transmitindo a mesma instrução para várias unidades de execução. A diferença chave é que SIMD requer que todos os elementos de um vetor em um mesmo vetor execute juntos como um grupo unificado síncrono, enquanto que SIMT permite múltiplas *threads* no mesmo *warp* execute independentemente. Mesmo que todas as threads em um warp comecem a execução juntas em mesmo endereço, é possível que *threads* tenham um comportamento individual diferente.

O modelo SIMT inclui três características chaves que SIMD não faz:

- Cada *thread* tem seu próprio contador de endereço de instrução.
- Cada *thread* tem seu próprio registrador de estado.
- Cada *thread* pode ter um caminho independente de execução.

Um bloco de *threads* é programado somente em um SM. Uma vez que um bloco de *thread* é programado em um SM, este permanece nele até sua execução terminar.

A Figura 8 ilustra uma visão lógica e visão do hardware de programação em CUDA.

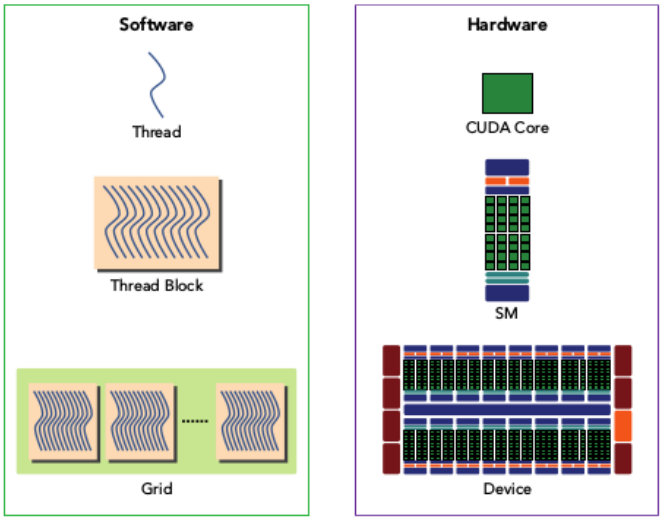

Figura 8 – Visão lógica e visão do hardware em CUDA
(CHENG; GROSSMAN; MCKERCHER, 2014)

Shared Memory e registradores são recursos preciosos em um SM. *Shared memory* é particionado entre blocos de *threads* residentes em uma SM e os registradores são particionados entre as *threads*. *Threads* em um bloco de *threads* podem cooperar e se comunicar com cada uma outras através desses recursos. Enquanto todas as *threads* em um bloco de *threads* executam logicamente em paralelo, todas as *threads* podem não executar fisicamente ao mesmo tempo.

Como consequência disso, diferentes *threads* em um bloco de *threads* podem ter progressos em tempos diferentes (CHENG; GROSSMAN; MCKERCHER, 2014).

O compartilhamento entre *threads* paralelas pode causar uma condição de corrida: múltiplas *threads* acessando os mesmos dados com uma ordenação indefinida que pode resultar em comportamento indesejado do programa. CUDA fornece um meio para sincronizar *threads* dentro de um mesmo bloco de *threads* para evitar que todas *threads* atinjam certos pontos na execução antes de fazer mais progresso. No entanto, não há primitivas fornecidas para sincronização entre blocos.

Enquanto *warps* dentro de blocos de *threads* podem ser programadas em qualquer ordem, o numero de *warps* ativas é limitado pelos recursos do SM. Quando um *warp* está vago por qualquer razão (por exemplo, esperando por valores a serem lidos da memória do *device*), o SM é livre para receber o próximo *warp* disponível de qualquer bloco de threads que é residente no mesmo SM. Alternando entre warps concorrentes não tem *overhard* porque os recursos do hardware estão particionados entre todas as *threads* e blocos em um SM, assim o estado do warp recém programado já está armazenado na SM (CHENG; GROSSMAN; MCKERCHER, 2014).

2.4.1.1 Arquitetura Fermi.

A arquitetura fermi foi a primeira arquitetura completa para computação em GPU a possuir as características necessárias para aplicações de alta performance mais exigentes. A Fermi foi amplamente adotada para acelerar programas com grandes de cargas de trabalho.

A Figura 9 ilustra um diagrama lógico de blocos da arquitetura Fermi focado na computação em GPU com seus componentes gráficos específicos. A Fermi apresenta até 512 núcleos aceleradores chamados núcleos CUDA. Cada núcleo CUDA tem uma unidade lógica aritmética inteira (ALU) e uma unidade de ponto flutuante (FPU) que executa uma instrução inteira ou de ponto flutuante por ciclo de clock. Os núcleos CUDA são organizados dentro de 16 *Streaming Multiprocessor* cada um com 32 núcleos CUDA.

Fermi tem seis interfaces de memória 384-bit GDDR5 DRAM com suporte para até um total de 6GB de memória global on-board, um recurso de computação fundamental para muitas aplicações. Uma interface *host* conecta a GPU a CPU através do barramento PCI Express. O mecanismo *Gigathread* (mostrado em laranja no lado esquerdo do diagrama) é um programador global que distribui os blocos de *thread* para os SM (CHENG; GROSSMAN; MCKERCHER, 2014).

A Fermi inclui um 768KB de cache L2, compartilhado por todos os 16 SMs. A Figura 10 mostra uma visão mais abrangente de um GPU com arquitetura Fermi, onde cada SM é representada por uma faixa retangular vertical, contendo cada um:

- Unidades de execução (núcleos Cuda)

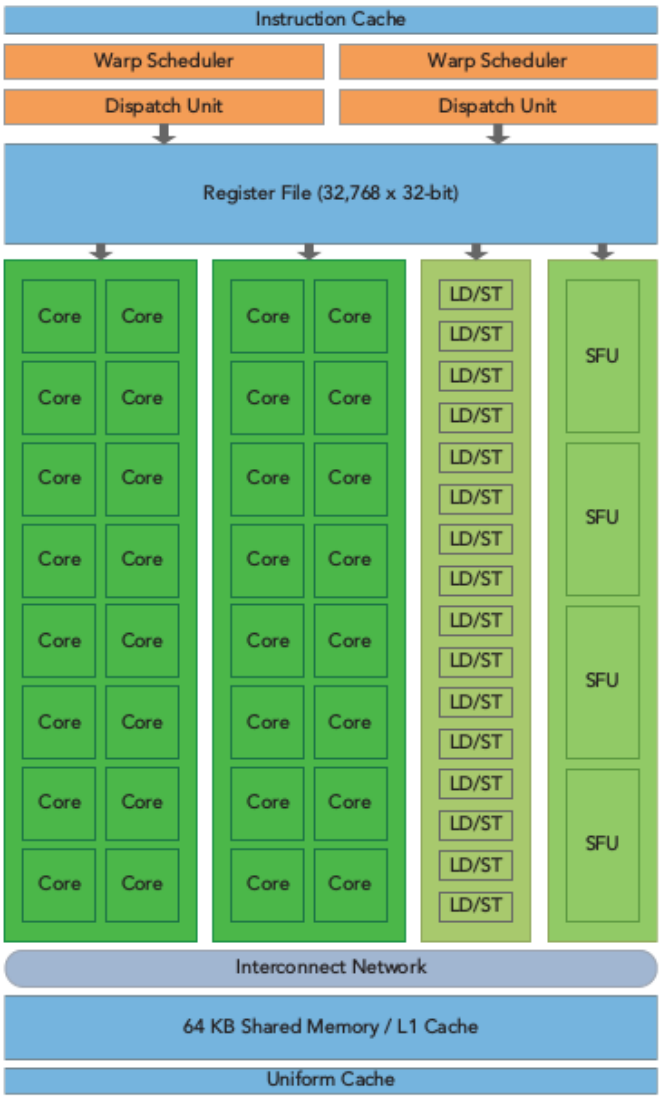

Figura 9 – Arquitetura Fermi e seus componentes
(CHENG; GROSSMAN; MCKERCHER, 2014)

- Programadores e expedidor unidade que programam um *warp*.
- *Shared Memory*, o arquivo de registro, e cache L1

Cada multiprocessador tem 16 unidades *load/store* (mostrado na Figura 9), permitindo endereços de origem e destino serem calculados por 16 *threads* (uma metade de *warp*) por ciclo de *clock*. Unidades de função especiais (SFUs) executam funções tais como seno, cosseno, raiz quadrada e interpolação. Cada SFU pode executar uma instrução por thread por ciclo de *clock*.

Cada SM apresenta dois *warps schedulers* e duas unidades *instruction dispatch*. Quando um bloco de *thread* é atribuído para um SM, todas as threads em um bloco de *threads* são divididos dentro de *warps*. Os dois *warps schedulers* selecionam duas *warps* e emite uma instrução de cada *warp* para um grupo de 16 núcleos CUDA, 16 unidades *load/store* ou 4 unidades de funções especiais. A arquitetura fermi tem capacidade de computação 2.x, pode

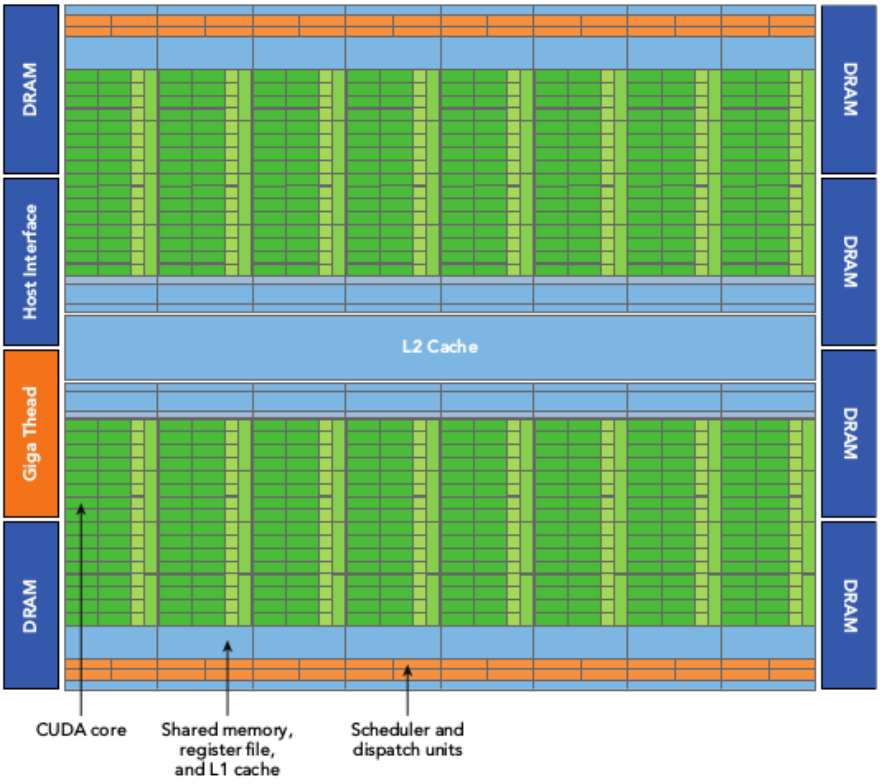

Figura 10 – Arquitetura Fermi, visão geral
(CHENG; GROSSMAN; MCKERCHER, 2014)

simultaneamente manipular 48 *warps* por SM para um total de 1,536 *threads* residentes em um único SM de uma vez (CHENG; GROSSMAN; MCKERCHER, 2014).

Uma característica fundamental de Fermi é o chip de memória de 64KB configurável, o qual é particionado entre *shared memory* e L1 cache. Para muitas aplicações de alto desempenho, *shared memory* é a chave para permitir bom desempenho. *Shared memory* permite que *threads* dentro de um bloco possam cooperar, facilita extensivo reuso dos dados *on-chip*, e reduz consideravelmente o tráfego *off-chip*. CUDA fornece uma API que pode ser usada para ajustar a quantidade de *shared memory* e de cache L1. Modificando a configuração de memória *on-chip* pode levar a melhorias de desempenho, dependendo do uso de *Shared Memory* ou cache em um determinado *kernel*.

Fermi também suporta execução de *kernel* concorrente: múltiplos *kernels* lançados a partir do mesmo contexto de aplicação executando na mesma GPU ao mesmo tempo. Execução de *kernels* concorrentes permite que programas que executem uma série de pequenos *kernels* para utilizar plenamente a GPU. Fermi permite que até 16 *kernel* possam ser executados em um *device* ao mesmo tempo. Execução de *kernel* concorrente faz a GPU parecer mais como uma arquitetura MIMD do ponto de vista do programador.

2.4.1.2 Programação CUDA.

CUDA é uma plataforma de computação paralela e um modelo de programação com um pequeno conjunto de extensões para a linguagem C/C++. Através do CUDA, pode-se implementar um algoritmo paralelo facilmente, da mesma maneira que se escreve programas em C. Com CUDA pode-se construir aplicações para uma quantidade inumerável de sistemas em GPUs NVIDIA, que vão desde dispositivos embarcados, tablets, laptops, desktops, e estação de trabalho para sistemas de cluster de HPC.

O modelo de programação CUDA possibilita a execução de aplicações em sistemas heterogêneos de computação, simplesmente anotando código com um pequeno conjunto de extensões para a linguagem de programação C. Um ambiente heterogêneo é composto por CPUs complementadas por GPUs, cada uma com sua própria memória, e separados por um barramento PCI-Express. Assim, aqui deve-se diferenciar os seguintes termos:

- *Host*: a CPU e sua memória (*host memory*)
- *Device*: a GPU e sua memória (*device memory*)

O principal componente do modelo de programação CUDA é o *Kernel* - o código que é executado na GPU. Quando um *kernel* é chamado, as computações são executada N vezes em paralelo por N diferentes *threads* CUDA, ao contrário de somente uma vez como um programa regular em C (NVIDIA, 2012) Um *kernel* é composto por várias *threads*, organizadas em uma *grid* de blocos de *threads*. Todos as *threads* em um mesmo bloco executam a mesma função do *kernel* (MORAES, 2012)

A estrutura de uma função *kernel* é definida usando a declaração "*global*"e um número de blocos de *threads* e *threads* a serem executados, que por sua vez são definidos dentro da expressão "$\langle\langle\langle\rangle\rangle\rangle$", conforme mostra a Figura 11 (NVIDIA, 2012).

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Figura 11 – Exemplo genérico de código CUDA
(NVIDIA, 2012)

Um tipico programa CUDA consiste de código serial complementado por código paralelo. Como mostra a Figura 12, o código serial (bem como as tarefas código paralelo) é executado no

host, enquanto o código paralelo é executado no *device*. O código do *host* é escrito em ANSI C, e o código *device* é escrito usando CUDA C.

Depois de compilado pelo NVCC (compilador da NVIDIA), é então gerado um código executável tanto para o *host* quanto para o *device*.

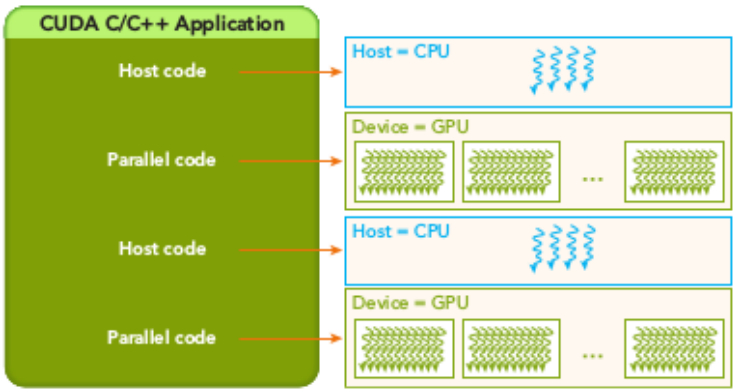

Figura 12 – Fluxo de execução *host/device*
(CHENG; GROSSMAN; MCKERCHER, 2014)

Um fluxo de processamento de um programa CUDA segue esse padrão:

1. Copia os dados da memória da CPU para memória da GPU.
2. Invoca *kernel* para operar sobre os dados armazenados na memória da GPU.
3. Copia os dados de volta da memória da GPU para memória da CPU.

2.4.2 OpenACC.

OpenACC é um padrão de programação para computação paralela desenvolvida em consórcio pelas empresas Cray, CAPS, NNIVIA e PGI. Ele fornece uma camada de abstração sobre o CUDA (CHENG; GROSSMAN; MCKERCHER, 2014), visando assim simplificar o desenvolvimento de aplicações paralelas em sistemas heterogêneos (CPU/GPU).

O OpenACC basicamente descreve um conjunto de diretivas de compilador que são usadas para especificar *loops* e trechos de código escritos nas linguagens de programação C, C++ e Fortran. Essas diretivas são repassadas de um *host* (CPU) para um *device* (GPU) a fim de fornecer portabilidade entre vários sistemas operacionais e também entre CPUs e GPUs. OpenACC permite que programadores possam fornecer caracteres simples em forma de diretivas para o compilador, identificando quais áreas de código serão paralelizadas, sem necessidade dos programadores modificarem ou adaptarem o código para alguma arquitetura específica. Essas diretivas, ao especificar paralelismo ao compilador, permitem fazer um trabalho de mapeamento da computação sequencial para a computação em GPU.

A estrutura do OpenACC é semelhante ao modelo CUDA, com apenas uma diferença, pois o OpenACC acrescenta uma dimensão de paralelismo extra. O paralelismo em OpenACC é dividido em *gangs*,*workers* e *vector elements*. Em alto nível, *gangs* são análogas a blocos de *threads* em CUDA. Uma única *gang* é atribuída a um único *Stream Multiprocessor* por vez, e as *gangs* por sua vez podem conter uma ou mais *threads*. Dentro de cada *gang*, há um ou mais *workers*. Na terminologia CUDA, um *workers* seriam análogo a uma *warp de threads*. Cada *worker* tem um *vector*, que é composto de um ou mais *vector elements*, cada *vector elements* executam uma mesma instrução ao mesmo tempo. Cada um desses *vector elements* é análogo a uma *thread* CUDA. A principal diferença entre os modelos CUDA e OpenACC é que OpenACC expõe o conceito de *workers* (isto é, *warps*) diretamente no modelo de programação, enquanto que em CUDA a criação de *warps de threads* não é explícita (CHENG; GROSSMAN; MCKERCHER, 2014).

Quando usa-se OpenACC, diretivas de compilador são inseridas pelo programador para indicar regiões de código que podem ser, ou deveriam ser, executadas em paralelo. Diretivas de compilador podem também indicar o tipo de paralelismo que será usado. Uma diretiva de compilador do OpenACC, assim como as diretivas do XcalableMP, que será visto mais adiante, começam com a palavra *#pragma*, porém com um identificador a mais a palavra-chave: acc, sendo assim todos diretivas OpenACC começam com *#pragma acc*, conforme mostra a Figura 13.

```
#pragma acc kernels
{
    for (i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
    }
    for (i = 0; i < N; i++) {
        D[i] = C[i] * A[i];
    }
}
```

Figura 13 – Exemplo de código OpenACC
(CHENG; GROSSMAN; MCKERCHER, 2014)

Diretivas de compilador são únicas em que, embora sejam uma parte do código fonte do programa, podem ou não afetar o executável gerado pelo compilador. Se um compilador não reconhecer ou suportar algum tipo de *#pragma*, ele será simplesmente ignorado e o programa irá ser compilado como se o *#pragma* não estivesse lá. Atualmente, vários compiladores como OMNI, PGI, CRAY e CAPS suportam diretivas OpenACC. Os exemplos e as aplicações no presente neste trabalho foram realizados com o compilador OMNI, mas teoricamente todos os códigos fontes podem ser compilados com os demais compiladores.

Uma das grandes vantagens que o OpenACC fornece ao programador é que ele não precisa se preocupar com detalhes específicos da arquitetura das GPUs, embora alguns parâmetros possam ser especificados para permitir otimizações em GPUs especificas (por exemplo, numero de *threads por bloco*). Comparando com CUDA, o programador somente tem de informar ao compilador sobre os laços *"for"* a serem paralelizados e sobre as variáveis envolvidas. O compilador é então responsável a ajustar a aplicação, considerando as características da arquitetura da GPU (WOLFE, 2013).

Compiladores OpenACC interpretam as diretivas OpenACC de um programa e realizam uma análise automática do código fonte para assim gerar automaticamente código fonte adequadas às GPUs. Como resultado, é possível executar automaticamente uma aplicação em GPU simplesmente adicionando poucas linhas de diretivas de compilador.

2.5 XcalableMP.

Durante muito tempo os programas voltados para sistemas de memória distribuída, tais como cluster de PC de alto desempenho eram escritos com MPI. Embora, o MPI seja de fato o padrão mais aceito para programação em sistemas de memória distribuída, os custos para sua programação são altos (NAKAO et al., 2012). Na programação com MPI os programadores são responsáveis por descrever toda distribuição de dados e tarefas, bem como a comunicação interna e sincronização, isso tudo com funções consideradas primitivas. Além disso, a curva de aprendizado que a programação com MPI exige é muito alta, o que leva os interessados a gastarem muito tempo no aprendizado desta ferramenta em vez de se dedicarem a programação em si.

A medida que os sistemas computacionais foram se tornando mais complexos, no que diz respeito à hierarquia de memória e suas topologias de interconexão, também houve a necessidade de aumentar a produtividade no desenvolvimento de aplicações paralelas, assim foi proposto o modelo de linguagem PGAS (*Partitioned Global Address Space*). Desde então o modelo PGAS tem sido avaliado como uma alternativa promissora para passagem de mensagem na computação de alto desempenho para supercomputadores e clusters (JIN et al., 2011)

O modelo de linguagem PGAS consiste de um modelo que forma um espaço de endereçamento global onde qualquer processo pode acessar os dados transparentemente, similar a um sistema de memoria compartilhada (NAKAO et al., 2012). O espaço de endereço global é particionado de forma lógica entre os processos, que abstrai grande parte da complexidade na troca de dados entre vários núcleos de processamento. Este modelo facilita loops paralelos e comunicação entre processos, consequentemente, aumenta a produtividade no processo de desenvolvimento de programas em paralelo, para ambientes de memória distribuída.

Entre as várias linguagens PGAS destaca-se o XcalableMP (XMP) que é um modelo de linguagem PGAS baseada em marcação de código para linguagens C e Fortran. O XMP per-

mite programadores paralelizar códigos sequenciais com poucas modificações usando diretivas simples assim como o OpenMP e *High Performace Fortran* (HPF) (NAKAO et al., 2012). A API do XMP é composta por uma coleção de diretivas de compilador usadas para descrever paralelismo de dados e tarefas, fornecendo um modelo de programação paralela para sistemas multiprocessadores de memória distribuída (LEE; SATO, 2010).

O XMP, assim como o OpenMP, tem como base diretivas de marcação de código, porém com uma importante diferença: o XMP é usado para paralelização em memória distribuída. O XMP abstrai toda complexidade que existe entre os núcleos de processamento e as memórias num sistema distribuído formando uma área de endereçamento global como uma só memória compartilhada. Toda tradução dessa complexidade para rotinas mais simples é feita pelo MPI, em tempo de compilação, ficando uma camada abaixo do XMP. Assim o XMP nada mais é que um *framework* que implementa a interface do MPI (NAKAO et al., 2012).A Figura 14 mostra as camadas da arquitetura do XMP.

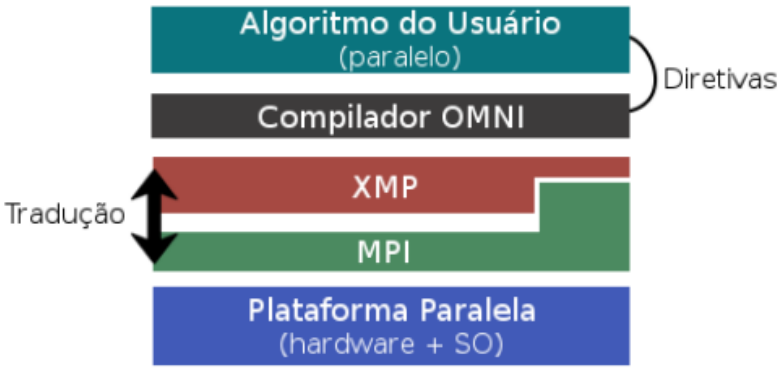

Figura 14 – Arquitetura do XMP
(Esdras La Roque)[1]

O XMP também se destaca por sua diversidade no suporte a um grande conjunto de técnicas da programação paralela, como por exemplo, a possibilidade de utilizar as marcações OpenMP, assim como diretivas OpenACC para possibilitar programação paralela em memórias de GPUs distribuídas.

2.5.1 Modelos de Programação.

O XMP possui dois modelos de programação: o modelo de programação local e o modelo de programação global. No modelo de memória local o acesso a memória remota é feito por meio de índices de cada processador, isso requer que os programadores façam seus algoritmos alocando dados e fluxos de controle explicitamente para cada processador.

[1] Figura gentilmente cedida pelo Mestrando Esdras La Roque

Já no modelo de visão global o programador descreve um espaço de endereços globais onde cada processo pode acessar dados remotos transparentemente como um sistema de memória compartilhada. Esse espaço de memória global é particionado logicamente entre os processos, essa característica permite programadores desenvolverem mais facilmente aplicações paralelas (GROUP et al., 2014).

2.5.2 Diretivas Básicas do XMP.

O foco do trabalho é a programação usando o modelo de SIMD (*Single Program Multiple Data*), usando também o modelo de programação de visão global. Por isso serão apresentadas as principais diretivas para programação em visão global dividas em três tipos: distribuição de dados, distribuição de trabalho e comunicação/sincronização.

2.5.2.1 Distribuição de dados:

- *nodes*: define os nós de processamento.
- *template*: define o modelo de particionamento.
- *distribute*: define o mapeamento do processamento.

2.5.2.2 Distribuição de trabalho:

- *align*: associação do array com o modelo.
- *loop*: executa laço de execução paralelo.

2.5.2.3 Comunicação e sincronização:

- *shadow*: cria região de sombra para comunicação.
- *reflect*: sincroniza valores da região de sombra.
- *barrier*: cria ponto de sincronização explícito.
- *gmove*: movimenta dados da área global para região local.

2.5.3 Visão geral Programação com XMP.

A Figura 15 mostra um exemplo de um fragmento de um código em XMP. Essa figura mostra a paralelização de dados, qual é executado em um cluster de PC tradicional. Primeiro, o programador declara os estados dos ***nodes***, a diretiva com número de nós que vão executar o programa. Nesse exemplo, quatro nós são usados. Então a diretiva ***template*** descreve o tamanho e modelo dos dados globais usando uma matriz virtual chamando um modelo. Nesse caso, usamos

o template **t**. A diretiva ***distribute*** é usada para mapear o template t para cada nó declarado anteriormente. Nesse caso o *template* é dividido em quatro partes e mapeado para os nós. A opção ***BLOCK*** significa que cada nó obtém uma seção de dados consecutiva. Finalmente, a diretiva ***align*** é usada para mapear cada modelo distribuído ao *array*. Aqui um *array* chamado **x** é distribuído e mapeado para os nós e os elementos de um *array* parcial são alocados em uma memória local de cada nó.

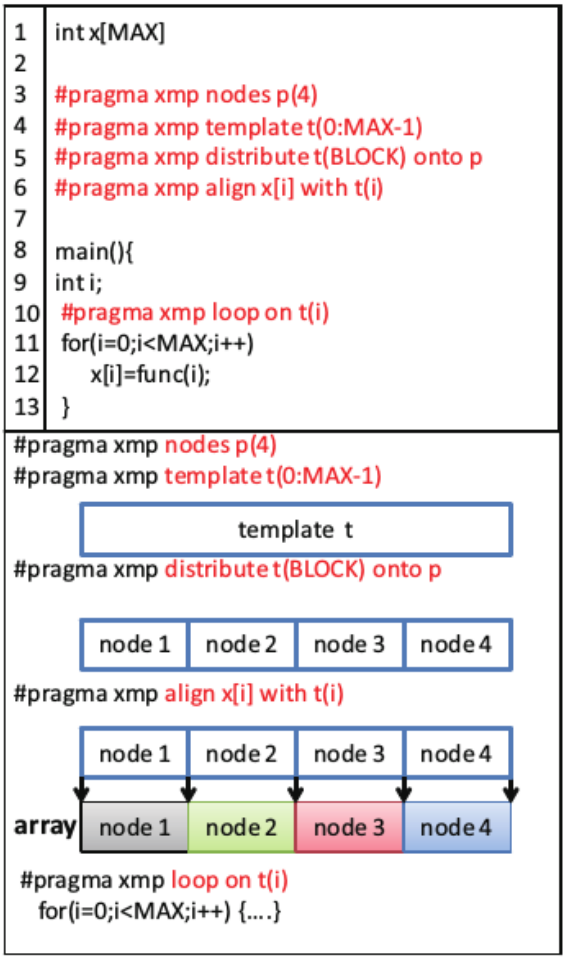

Figura 15 – Exemplo de código XMP e esquema de distribuição de dados e tarefas
(NOMIZU et al., 2012)

A diretiva ***loop*** descreve como compartilhar o trabalho para cada nó. Nesse exemplo, cada nó armazena valores calculados em seu *array* local **x**, qual corresponde a uma seção do *array* virtual **t**. Mais diretivas podem ser usadas, incluindo as diretivas de comunicação de processos através da diretiva ***shadow*** que tem a função de declarar o tamanho de regiões sincronizáveis (regiões de sombra) em um dado *array*. Já sincronização em si é realizada pela diretiva ***reflect***, a qual tem o objetivo de manter os valores da região de sombra atualizados conforme suas extremidades para cada núcleo de processamento.

2.6 XcalableACC.

Afim de desenvolver aplicações em cluster de GPUs, programadores tem usado um modelo de programação híbrida para transferir grandes quantidades de dados entre nós de um cluster, e atribuir tarefas computacionais para as GPUs. Geralmente, os programadores usam funções MPI combinadas com alguma linguagem de programação para GPUs, tais como OpenACC, OpenCL, ou CUDA. Entre essas, destaca-se o OpenACC que fornece boa produtividade e alto desempenho, junto com baixos custos de aprendizado, reescrita e reajustes. Porém, programação com MPI teria de ser substituída. O modelo de programação do MPI é considerado complexo se comparado com as vantagens que o OpenACC pode oferecer,visto que o MPI usa funções muito primitivas para tarefas importantes como distribuir, transferir, e sincronizar dados.

Assim, foi proposto a substituição do MPI pela tecnologia XMP. A partir dai foi dado outro passo importante, pois baseado nas caracterısticas do XMP e OpenACC, foi desenvolvida uma linguagem baseada em diretivas direcionada especialmente para clusters de GPUs. Esse novo modelo de programação foi chamado de XcalableACC (XACC). No XACC, programadores podem usar as diretivas do XMP e do OpenACC perfeitamente. Portanto, XACC permite à programadores desenvolver aplicações para clusters de GPUs. Além de casar as características das duas tecnologias eficientemente, XACC também pode fornecer outras diretivas como, por exemplo, para transferência de dados entre memórias das GPUs (NAKAO et al., 2014).

2.6.1 Visão Geral da programação com XcalableACC.

A Figura 16 mostra um exemplo de código XACC que exemplifica o uso de ambas diretivas, do XMP e do OpenACC. Da linha 1 a 5, diretivas XMP definem a distribuição do array a[ ]. Na linha 7, a diretiva data do OpenACC transfere o array distribuıdo a[ ]que foi definido pelo XMP para a memória da GPU. Na linha 9, a diretiva loop do XMP paraleliza o loop em cada nó. Logo em seguida, a diretiva parallel do OpenACC junto com a clausura loop, na linha 10, também paraleliza o loop já dividido pelo XMP, mas agora para as GPUs. Nesse exemplo em particular, a ordem das diretivas loop do XMP e da diretiva parallel do OpenACC não interferem na execução.

```
1  double a[N];
2  #pragma xmp template t(0:N−1)
3  #pragma xmp nodes p(4)
4  #pragma xmp distribute t(block) onto p
5  #pragma xmp align a[i] with t(i)
6  ...
7  #pragma acc data copy(a)
8  {
9  #pragma xmp loop on t(i)
10 #pragma acc parallel loop
11   for(int i=0;i<N;i++){
12     a[i] = ... ;
13   }
14 }
```

Figura 16 – Exemplo de código XACC
(NAKAO et al., 2014)

2.7 Considerações Finais do Capítulo.

Este capítulo introduziu os principais conceitos sobre programação paralela e organização e classificação das principais arquiteturas paralelas com memória compartilhada e distribuída. Os ambientes heterogêneos foram apresentados, assim como a computação em GPU e seus principais modelos de programação. O modelo de programação em memória distribuída XcalableMP foi também mostrado como alternativa para comunicação em clusters de aceleradores.

3 ALGORITMOS GENÉTICOS

3.1 Considerações Iniciais.

Os algoritmos genéticos são um tipo de algoritmos que estão compreendidos dentro de uma área da ciência da computação chamada **Computação Evolucionária** . A computação evolucionária tem como principal objeto de estudo os processos naturais de evolução. A inspiração nos processos naturais se deve ao fato de que na natureza o poder da evolução nas espécies ser bastante evidente. Durante muito tempo esses processos foram estudados, e foi notado que cada espécie tem a capacidade de se adaptar e sobreviver em um dado ambiente. Esses processos, assim, servem de base para resolução de problemas reais, mais precisamente em problemas que possam ser resolvidos pelo método de tentativa e erro.

Em um dado ambiente que é composto por uma população de indivíduos que lutam para sobreviver e se reproduzir. O *fitness* desses indivíduos são determinados pelo ambiente mostram o quanto bem eles serão bem sucedidos em alcançar seus objetivos, isto é, representa suas chances de sobrevivência e multiplicação. No contexto do método de tentativa e erro (também conhecido como gerar e testar) de processo de resolver problemas, nós temos uma coleção de soluções candidatas. Suas qualidades (isto é, o quão bem eles resolvem seus problemas) determinam a mudança que eles manterão e serão usadas como semente para construção de soluções candidatas mais adiante (EIBEN; SMITH et al., 2003)

Algoritmos evolucionários são usados para uma grande variedade de resolução de problemas. A capacidade de evoluir e adaptar indivíduos desse algoritmos se mostram muito eficaz na aprendizagem e otimização de problemas. Geralmente esses problemas são compostos de múltiplas variáveis em um universo de busca muito grande, e tais problemas precisam combinar suas características para obter um melhor resultado ou uma solução aceitável.

Os algoritmos genéticos são um tipo especial de Algoritmo evolucionário que tem um foco mais biológico. Eles realizam operações que simulam uma seleção natural, como mutações e recombinações. Os algoritmos genéticos em particular são usados para otimização de problemas complexos, onde se existe uma grande variedade de soluções (uma população), que passarão por etapas de processamentos e será avaliada por uma determinada função afim de gerar descendentes com melhorias genéticas, ou seja, outro conjunto de soluções melhores.

Aplicações de computação evolucionária são encontradas em muitas áreas como na matemática, engenharia, biologia, automação, robótica, etc. Alguns exemplos serão citados abaixo:

- Problemas que demandam por otimização complexas: problemas do caixeiro viajante, que calcula a melhor ordem para o caixeiro visitar cidades com menor custo, roteamento de veículos, otimização de Layout de circuitos, otimização de distribuição, otimização em

negócios, gerenciamento de redes.

- Otimizações de funções matemáticas.
- Robótica: onde se busca um caminho praticável, seguro e sem colisões deve ser percorrido por um robô.
- Aplicações em engenharia: redes de telecomunicações, projetos estruturais, projeto de aeronaves, projeto de estruturas espaciais, projeto de reatores químicos, teste e diagnóstico de falhas, etc.
- Simulações: simulando como um determinado sistema irá se comportar baseado em um modelo ou projeto estabelecido, como: determinação de equilíbrio em sistemas químicos, ou comportamento de uma estrutura de rede elétrica.

3.2 Inspiração na Teoria da Evolução de Darwin.

A teoria da evolução de Darwin trouxe uma explanação sobre a diversidade biológica e seus mecanismos fundamentais. Essa teoria desempenha um papel central na chamada visão macroscópica da evolução. Dado um ambiente, que pode abrigar somente um número limitado de indivíduos, esses indivíduos por sua vez possuem o instinto mais básico dos seres vivos, a reprodução. Sendo assim a seleção se torna inevitável para que a população não cresça exponencialmente e se tenha problemas com falta de recursos. A seleção natural favorece os indivíduos que competem por dados recursos mais eficientemente, em outra palavras, aqueles que são mais adaptados ou ajustados as melhores condições de ambiente. Este fenômeno também é conhecido como sobrevivência dos mais fortes.

Competição baseada em seleção é uma dos dois pilares do progresso evolucionário. As outras forças primárias identificáveis por Darwin resultam de variações fenotípicas entre os membros da população. Traços fenotípicos são as características comportamentais e físicas que são diretamente afetados por sua resposta ao ambientes (incluindo outros indivíduos), e assim determinando seu fitness. Cada indivíduo representa uma única combinação de traços fenotípicos que é calculado pelo ambiente. Se esse cálculo for favorável, então os traços são propagados para sua descendência, caso contrário esta é descartada e morrem sem descendência. Darwin tinha um pequeno conhecimento de que variações de mutações aleatórias ocorria nos traços fenotípicos durante a reprodução de geração para geração. Através dessas variações, novos traços de combinações ocorrem e são avaliadas. Os melhores sobrevivem e se reproduzem, e então a evolução progride (EIBEN; SMITH et al., 2003).

De forma resumida, a população consiste de um número de indivíduos. Esses indivíduos são as "unidades de seleção", isto é, para dizer que o sucesso reprodutivo depende do quão bem eles são adaptados a seus ambientes em relação ao resto da população. Como os mais bem

sucedidos reproduzem, mutações ocasionais dão origem a novos indivíduos para serem testados. Assim com o passar do tempo, haverá mudanças na constituição da população, isto é, sempre buscando se tornar melhor.

3.3 Como funciona um algoritmo genético.

A base de um algoritmo genético é que uma dada população é composta de vários indivíduos, cada indivíduo tem duas propriedades: sua localização (cromossomos compostos de genes) e sua qualidade (valor de fitness) (YU; GEN, 2010). O ambiente em que estão inseridos força uma seleção por causas naturais (sobrevive o mais forte), que consequentemente causa um aumento no fitness da população. Essas forças podem ser representadas por uma função de qualidade ou função de avaliação em que os indivíduos são submetidos para que sejam obtidos os melhores e/ou maiores valores de fitness.

Inicialmente, os indivíduos são escolhidos aleatoriamente para um processo de seleção, e são encaminhados para um espaço de acasalamento, nesse espaço o objetivo é que os indivíduos com melhores fitness sejam selecionados para sobreviver e se reproduzirem e assim gerar descendentes também com melhores fitness. Esse processo é chamado de seleção de pais. Operações de recombinação e/ou mutação ainda podem ser aplicados sobre os indivíduos. Recombinação é uma operação aplicada em dois ou mais candidatos selecionados (os chamados pais) e resulta em um ou mais novos candidatos (os filhos). Mutação é aplicada em um candidato e resulta em um novo candidato (descendência). Os indivíduos após essas etapas seguem para competir com outros indivíduos, também com base no seus fitness, por um lugar na próxima geração.

A ideia por trás desses passos é que os melhores indivíduos tenham mais chances de serem selecionados dentro do espaço de acasalamento, e portanto tenham mais chances de acasalamento do que indivíduos de baixo fitness. Assim a informação contida nos bons indivíduos têm mais chances de ser preservadas e passadas a próxima geração. O intercâmbio de informações entre os pais e as pequenas mudanças nos descendentes promovem a busca por melhores indivíduos. Assim, combinando esses dois fatores, a população vai se tornar mais e mais apta, prosseguindo até que a solução ótima ou quase ótima tenha sido encontrada ou o limite computacional do algoritmo previamente estabelecido seja atingido.

A relação entre os conceitos e as operações descritas acima e o princípio da teoria da seleção natural de Darwin são listadas abaixo:

- Seleção: sobrevivência do mais forte.
- Dois pais geram um ou dois filhos: crossover ou recombinação.
- Pequenas mudanças na localização dos filhos: mutação.

3.4 Características dos Algoritmos evolucionários.

Nesses processos dos algoritmos genéticos pode-se notar que existem dois procedimentos fundamentais que caracterizam bem e formam as bases dos sistemas evolucionários:

- Operações de variação (recombinação e mutação) que criam as diversidades necessárias e assim facilita o surgimento de mudanças.
- Ações de seleção como uma força de geração de qualidade.

A combinação dos operadores de variação e seleção, como citado anteriormente, geralmente conduzem a um aumento nos valores do fitness na população subsequente. Esse processo aumento é facilmente visto quando a evolução vai se otimizando, ou pelo menos, se aproximando do valor ótimo mais próximo ou mais próximo possível. Porém, do ponto de vista genético, esse elevamento de fitness é visto como um processo de adaptação a cada geração. A partir dessa perspectiva, o fitness não é apenas visto como uma função objetiva a ser otimizada, mas como uma expressão de requisitos de ambientais. Combinando esses requisitos implica em um aumento da viabilidade de adaptação, refletindo assim em número maior de descendentes. Portanto, o processo evolucionário faz a população cada vez melhor e sendo adaptada ao ambiente.

Outra característica marcante de muitos dos componentes de tais processos evolucionários é que eles são processos estocásticos. Durante os processos de seleção os melhores indivíduos têm maiores chances de serem selecionados do que os piores, mas comumente mesmo os indivíduos mais fracos tem alguma chance de tornar-se um pai ou sobreviver (EIBEN; SMITH et al., 2003). Na recombinação de indivíduos a escolha de qual peça vai ser recombinada é aleatória. Similarmente para mutação, as peças que vão ser mutadas dentro uma solução candidata, e a nova peça que vai substituir ela, também são escolhas aleatórias. O esquema global de um algoritmo evolucionário pode ser visto no algoritmo 1, em pseudo-código. A Figura 17 mostra o

algoritmo em formato de diagrama.

Algoritmo 1: Pseudocódigo de um Algoritmo Evolucionário

```
1  início
2      INICIALIZAÇÃO
3      AVALIAÇÃO
4      i = 0;
5      para i < NUMERO DE GERAÇÕES faça
6          SELEÇÃO DE PAIS
7          AVALIAÇÃO
8          MUTAÇÃO
9          RECOMBINAÇÃO
10         SELEÇÃO DE SOBREVIVENTES
11     fim
12 fim
```

Observando a imagem nota-se que o algoritmo se enquadra na categoria de algoritmos de geração e testes. A função de avaliação (fitness) representa uma estimação heurística de qualidade de solução, e o processo de busca é guiado pelas operações de variação e operações de seleção. Os algoritmos evolucionários possuem algumas características peculiares que os identificam na categoria de métodos de geração e testes, algumas já até citadas acima:

- AE são baseados em população, isto é, eles processam uma coleção completa de soluções candidatas simultaneamente.
- AE na maioria das vezes usam recombinação para misturar informação de mais soluções candidatas dentro de uma nova.
- AE são estocásticos.

3.5 Componentes de algoritmos evolucionarios.

Algoritmos evolucionários têm um número de componentes, procedimentos e operações que devem ser especificados afim de definir um determinado AE. A maioria dos mais importantes componentes de um AE são citados abaixo:

- Inicialização
- Função de avaliação (ou Função fitness)
- Mecanismo de seleção de pais.

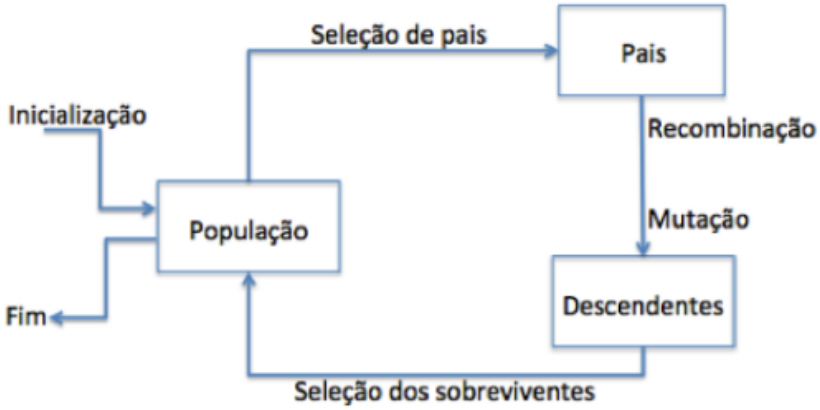

Figura 17 – Diagrama de um Algoritmo evolucionário
(TADAIESKY, 2015)

- Operações de variação (Recombinação e mutação).
- Mecanismo de seleção de sobreviventes.

Antes de começar a descrever sobre cada um dos componentes do algoritmo é importante primeiro definir a forma com que a população vai ser definida. A definição ou representação é de essencial importância para se entender o ponto de ligação entre o "mundo real" e o "mundo do AE", a ponte entre um problema real e sua soluções no espaço de evolução. Cada indivíduo da população representam as possíveis soluções para um problema, esses são referenciados como o fenótipo. Essa população é definida através de cromossomos, esses cromossomos são compostos de genes. Os genes são a representação codificada do fenótipo dentro desses cromossomos representando assim o genótipo.

Existem diversas formas de codificar os genes de um indivíduo. Os três principais tipos de representação são: a codificação binária, onde cada gene armazena um bit; codificação inteira, onde cada gene armazena um valor inteiro; e a codificação real em que os genes armazenam números reais. Dependendo do tipo do problema eles podem ser representados por qualquer um dos tipos.

3.5.1 Inicialização.

Como já citado acima AE trabalham processando informações de um determinado grupo de indivíduos. Geralmente o tamanho dessa população é definido pelos programadores através de um array de uma ou duas dimensões contendo os valores de domínio dessa população. O tamanho da população é uma das mais importantes partes dos AE, e merece bastante cuidado, pois afeta o desempenho da busca e a eficiência do algoritmo. Se o tamanho da população for muito pequena existe grande chance da população não ter diversidade necessária para convergir para uma solução ótima. Por outro lado, se a população for muito grande o algoritmo pode perder

sua eficiência por demorar a avaliar cada indivíduo na função de avaliação, além de demandar mais recursos computacionais (POZO et al., 2005).

Já a forma com que a primeira população é gerada não obedece critério definido, pois é gerada de modo aleatório. Isso é feito para que o conjunto de soluções iniciais tenha uma maior variedade possível para o processo de busca de soluções posteriores. O próximo passo após o início da geração de cromossomos é o cálculo de seus valores de *fitness* usando um decodificador de processos e a função de avaliação.

3.5.2 Função de avaliação.

A regra da função de avaliação ou função *fitness* é representar os requisitos de adaptação. Isto forma a base para seleção, e assim facilita as melhorias. Mais precisamente, define o que as melhorias significam. Da perspectiva da resolução de problemas, isto representa a tarefa a se resolver no contexto evolucionário. Tecnicamente, esta função ou procedimento que atribui uma medida de qualidade aos genótipos. Neste componente será calculado, através de uma determinada função objetivo, o valor de aptidão de cada indivíduo da população. Este é o componente mais importante de qualquer algoritmo genético. É através desta função que se mede quão próximo um indivíduo está da solução desejada ou quão boa é esta solução.

3.5.3 Mecanismo de seleção de pais.

A regra da seleção de pais ou seleção de acasalamento é diferenciar indivíduos baseado em suas qualidades (seu *fitness*), afim de permitir que os melhores indivíduos possam se tornar pais da próxima geração. Um indivíduo é um pai se foi selecionado para sofrer variações afim de criar descendentes. Juntos com o mecanismo de seleção de sobreviventes, a seleção de pais é responsável por promover melhorias de qualidade. Em computação evolucionária, seleção de pais é tipicamente probabilística. Assim, indivíduos de alta qualidade tem mais chances de torna-se pais do que os indivíduos de baixa qualidade, porém, ainda assim à indivíduos de baixa qualidade são frequentemente dadas uma pequena chance serem selecionados, podendo se tornar um pai ou até mesmo ser um candidato a ser tornar uma solução ótima.

Existem diversas formas de seleção, dentre as quais pode-se destacar um dos métodos mais usados: o método de seleção por Roleta. Nesse método cada indivíduo tem uma probabilidade de ser selecionado proporcional à sua aptidão. Para visualizar este método considere um círculo dividido em n regiões (tamanho da população), onde a área de cada região é proporcional à aptidão do indivíduo, conforme mostra a Figura 18. Assim, para indivíduos com alta aptidão é dada uma porção maior da roleta, enquanto aos indivíduos de aptidão mais baixa, é dada uma porção relativamente menor. Após um giro da roleta a posição dos cursores indica os indivíduos selecionados. Isto significa que quanto melhores são os cromossomos, mais chances de serem selecionados (POZO et al., 2005).

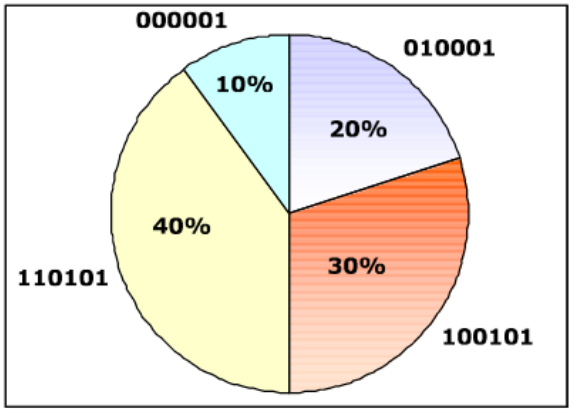

Figura 18 – Círculo da seleção por roleta
(POZO et al., 2005)

3.5.4 Operações de Variação.

A regra das operações de variação é criar novos indivíduos a partir dos mais antigos. A partir da perspectiva de geração e teste, as operações de variação realiza o passo de "geração". Existem muitas operações de variação para mudar as informações nos indivíduos no espaço de acasalamento. Se há troca de informação, isto é, troca de gene for feita entre dois ou mais indivíduos, esta operação de variação é chamada crossover ou recombinação. Se os genes de um indivíduo são mudados individualmente, esta operação de variação é chamada mutação.

3.5.4.1 Recombinação.

A primeira operação de variação é chamada "recombinação"ou *"crossover"*. Como o nome indica, tal operação funde informações de dois genótipos pais em um ou dois genótipos de filhos, afim de que esses filhos misturem as características de seus pais. Assim como a mutação, a recombinação é uma operação estocástica: os cromossomos de cada par de indivíduos a serem cruzados são particionados em um ponto, chamado ponto de corte, que é sorteado aleatoriamente, conforme mostra a Figura 19. Como resultado, um novo cromossomo é gerado permutando-se a metade inicial de um cromossomo com a metade final do outro.

O princípio por trás da recombinação é simples acasalando dois indivíduos diferentes mas com características desejáveis pode-se produzir um descendentes que combinam as melhores características dos dois indivíduos. Esse princípio sempre foi muito aplicado com sucesso por muitos anos por criadores de plantas e gado para produzir espécies com altos rendimentos ou ter outras características desejáveis. Algoritmos evolucionários criam um número de descendentes por recombinação aleatória, e por isso aceitam que alguns poderão ter combinação de características indesejáveis, e também grande maioria pode não ser o melhor ou pior do que seus pais, mas mesmo assim, espera-se que alguns possam apresentar características melhoradas (GUIMARÃES; RAMALHO, 2001).

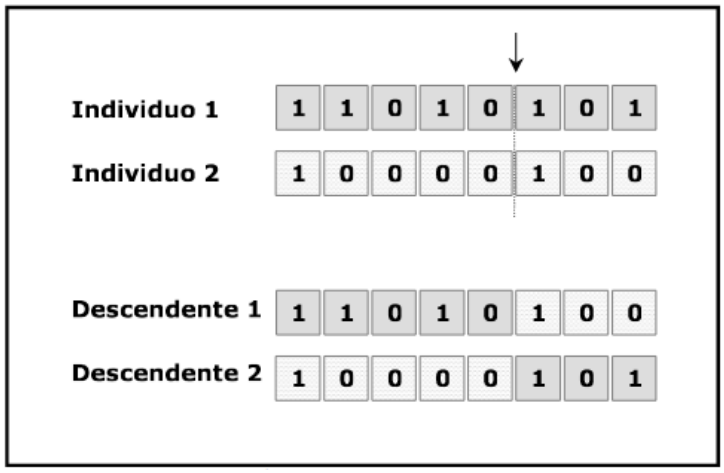

Figura 19 – Esquema da recombinação
(POZO et al., 2005)

3.5.4.2 Mutação.

A mutação é uma operação de variação unária, ou seja, tem apenas uma entrada e é aplicada em apenas um gene. Esta operação é aplicada em um genótipo e resulta em uma criança (descendente) mutante modificada (levemente), como mostra a Figura 20. Uma operação de mutação também é estocástica: sua saída, a criança, depende do resultado de uma série de escolhas aleatórias. A função da mutação é simplesmente modificar alguma característica de um indivíduo. Esta troca é importante, pois acaba por criar novos valores de características que não existiam ou apareciam em pequena quantidade na população em análise (POZO et al., 2005).

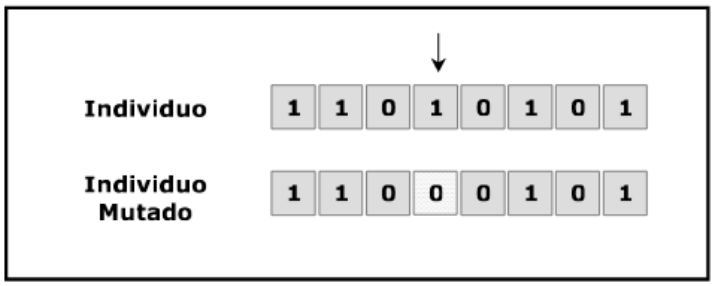

Figura 20 – Esquema da mutação
(POZO et al., 2005)

3.5.5 Mecanismo de seleção de sobreviventes.

A função da seleção de sobreviventes ou seleção de meio ambiente é selecionar os melhores indivíduos baseados na suas qualidades (os mais aptos). Esta etapa é bem similar a seleção de pais, mas é usada em um estágio diferente do ciclo evolucionário. O mecanismo de seleção de sobreviventes é chamado após já terem sido criados os descendentes pela seleção de pais. Esta etapa é importante pois nela serão feitas a escolha dos indivíduos que participarão da

próxima geração. Esta decisão é usualmente baseada no seu valor de *fitness*, favorecendo assim os indivíduos com alta qualidade.

Esta etapa também exerce a função de limitador da quantidade da população, pois a mesma deve se manter constante para não prejudicar o desempenho computacional. O critério básico para esse "corte" no número de filhos é que tenham e seus genótipos valores de função de avaliação maiores que a dos seus pais, caso contrário, a nova geração não contribuiria em nada para que a função de avaliação convergir para seu valor ótimo global (JÚNIOR; LEITE et al., 2015).

A forma de selecionar os indivíduos da seleção de sobreviventes é também diferente da seleção de pais (estocástica), já que a seleção de sobreviventes é frequentemente determinística. Através desse critério se determina e classifica os indivíduos de menores valores de fitness afim de eliminar os indivíduos necessários para manter a cardinalidade da geração anterior, ao mesmo tempo tentando promover uma melhora na população.

3.6 Algoritmos Genéticos Paralelos.

Como já citado anteriormente, Algoritmos genéticos são tipo de algoritmo que demandam muitos recursos computacionais se comparados a outros, isso se dá justamente por eles processarem uma grande quantidade de dados. Esses dados representam uma região de análise que passarão por métodos de busca e otimização que também será custoso para produzir uma saída ótima.

Por muito tempo AG sequenciais se mostraram bem sucedidos em muitos aplicações e em vários diferentes domínios e em várias funções. Porém existem alguns problemas que serão mais adequados usar alguma forma de AG paralelo, os principais motivos são listados abaixo:

- Para alguns tipos de problemas, a população precisa ser muito grande e a memória exigida para armazenar cada indivíduo deve ser considerável. Em alguns casos, se torna impossível executar uma aplicação eficientemente usando apenas uma única máquina, assim algumas formas paralelas de AG são necessárias.
- A avaliação de fitness normalmente consome muito tempo. Para algumas avaliações mais complexas de fitness o tempo de execução em uma CPU chegou a durar 1 ano (NOWOSTAWSKI; POLI, 1999). Essa é mais uma das razões para usar o poder de várias GPUs para processamento paralelo.
- AG sequenciais podem ficar preso em uma região subótima no espaço de busca, assim tornando-se incapaz de encontrar a melhor solução de qualidade. Já com AG paralelos pode-se buscar diferentes de espaço de busca em subespaços paralelos, assim tornando-os menos provável ficarem presos em espaços de baixa qualidade.

Visando superar esses problemas os AG paralelos foram propostos. Eles usam na maioria dos casos a abordagem de divisão e conquista eles procuram dividir a carga de trabalho, permitindo que vários pontos do domínio do problema sejam processados simultaneamente. Esse fator torna os AG ideais para paralelização. Alguns casos de AG paralelos podem ser paralelizados de forma tão eficaz que cada cromossomo pode até ter seu próprio processador para executar os cálculos necessários. E isso como consequência produz uma redução no tempo de execução em cada geração (PESSINI et al., 2003).

Existem outras razões além da velocidade para se paralelizar um AG. Alguns algoritmos genéticos paralelos podem abranger uma maior cobertura para o espaço de soluções, aumentando assim assim a chance de encontrar uma solução ótima. Outra razão para paralelizar um AG é reduzir a chance de uma convergência prematura, que ocorre quando poucos indivíduos com alto fitness dominam a população influenciando outros indivíduos a serem semelhantes a eles.

Segundo (NOWOSTAWSKI; POLI, 1999) a forma com que AG podem ser paralelizados dependem dos seguintes fatores:

- Como são aplicados os operadores genéticos mutação e crossover.
- Como são distribuídas às populações (simples ou múltiplas).
- No caso de múltiplas populações (demes), como os indivíduos são trocados.
- Como a seleção é aplicada (localmente ou globalmente).

Dependendo da forma como cada um desses elementos é implementado, vários métodos diferentes de paralelização de AG podem ser obtidos. Como existem diversas possibilidades de combinação desses fatores, não há um consenso com relação a uma nomenclatura a se usar para classificar os AG paralelos.

Algumas nomenclaturas foram propostas na tentativa de definir uma taxonomia para os AG paralelos. Entre elas desta-se a (CANTÚ-PAZ, 1997).

3.6.1 Taxonomia de CANTÚ-PAZ.

A taxonomia de (CANTÚ-PAZ, 1997), define a classificação dos Algoritmos Genéticos paralelos de acordo com a maneira que a população pode ser dividida. A abordagem de divisão e conquista é a base dessa taxonomia, no qual a população pode ser divida em pedaços para execução simultânea em múltiplos processadores. Alguns métodos de paralelização usam uma única população. Alguns métodos podem explorar arquitetura computacional massivamente paralela, enquanto outras são melhores adaptadas a multicomputadores com maior poder de processamento.

A classificação de (CANTÚ-PAZ, 1997) é muito similar a outras, porém mais estendida para incluir outras categorias. Há três tipos principais de algoritmos genéticos paralelos:

- Algoritmo Genético Mestre-Escravo com população global.
- Algoritmo Genético de Granularidade Fina.
- Algoritmo Genético de Granularidade Grossa.

3.6.1.1 Algoritmo Genético mestre-escravo com população global.

Esse método também é conhecido como avaliação de *fitness* distribuído, foi uma das primeiras mais bem sucedidos aplicações de algoritmos genéticos paralelos. Ele também é conhecido como modelo mestre-escravo de "paralelização global".

Nos algoritmos genéticos tradicionais, cada indivíduo compete e pode se reproduzir com qualquer outro da população, ou seja, a seleção e reprodução são realizadas globalmente. Já os AG paralelos mestre-escravo, algoritmos usam uma única população global que é armazenada no processo mestre e avaliação dos indivíduos e/ou os operadores de aplicação genética são realizados em paralelo pelos processos escravos. A seleção e cruzamento continuam sendo feitas globalmente, consequentemente, cada indivíduo pode competir e cruzar com outros.

A razão para que a operação a ser paralelizada seja a avaliação dos indivíduos se dá pois normalmente ela requer somente o conhecimento do indivíduo a ser avaliado (não a população toda), e assim não há a necessidade de comunicação durante essa fase. Por isso essa fase pode ser implementada usando processos escravos, pois o mestre só se encarrega armazenar a população e fazer as operações globais, e os escravos fazem as operações que exigem conhecimento local, como também as operações de mutação, e algumas operações de crossover (NOWOSTAWSKI; POLI, 1999).

A paralelização da avaliação de *fitness* é feita atribuindo uma fração da população a cada processador disponível(no caso ideal um indivíduo por processador). A comunicação ocorre somente quando cada escravo recebe o indivíduo (ou subconjunto de indivíduos) para avaliar e quando os escravos retornam o valor do fitness para o processo mestre.

O algoritmo é dito ser síncrono, se o mestre para e espera para receber os valores da avaliação de *fitness* de toda a população antes de prosseguir com a próxima geração. Um AG mestre-escravo síncrono tem exatamente as mesmas propriedades como um AG simples, exceto pelo incremento de velocidade no processamento das gerações. Uma versão assíncrona do algoritmo também é implementável. Nesse caso o algoritmo não pausa para esperar por qualquer processo escravo. A diferença se mostra na operação de seleção, pois em um algoritmo mestre-escravo assíncrono a seleção espera até que uma fração da população já ter sido processada para continuar a execução.

Este modelo pode ser implementado tanto em computadores com memória compartilhada tanto em computadores com memória distribuída. Em um computador com memória compartilhada, a população é armazenada na memória compartilhada e cada processador lê seu

conjunto de indivíduos e escreve os resultados da avaliação de volta no processo mestre. Em computadores com memória distribuída, a população pode ser armazenada em um processador. Este processador será o "mestre" responsável por enviar explicitamente os indivíduos para os outros processadores (escravos) para realizar a avaliação, coletando os resultados, e aplicando os operadores genéticos para produzir a próxima geração (PESSINI et al., 2003).

3.6.1.2 Algoritmo Paralelo de granularidade Fina.

Esse tipo de algoritmo exige um grande número de processadores, pois a população é dividida dentro de um grande número de pequenos demes, conforme mostra a Figura 21. Ele também pode ser definido de duas maneiras diferentes. Pode ser modelado como uma população simples, com uma estrutura espacial limitada que só permite os indivíduos competir e reproduzir somente com seus vizinhos. A outra forma é usar um esquema de sobreposição de vizinhança, com esse esquema diversas subpopulações são separadas, porém com algumas sobreposições. Isso permite que quando a operação de seleção for realizada, somente os cromossomos da mesma subpopulação possam se reproduzir, mas como muitos cromossomos estão em várias regiões de sobreposição o material genético pode se espalhar por toda população (PESSINI et al., 2003).

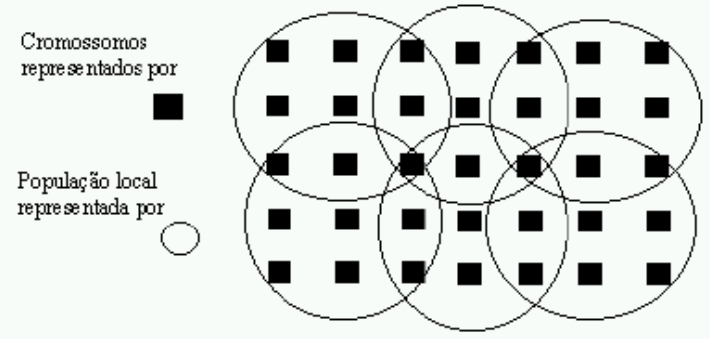

Figura 21 – Representação espacial de um AG de granularidade fina
(PESSINI et al., 2003)

As principais vantagens desse modelo é que sua implementação é simples, o que evita problemas relacionados a migração de cromossomos. A população de cada subconjunto é separada o suficiente para prevenir que os indivíduos mais aptos dominem a população. Esse modelo também é muito adequado para execução em máquinas multiprocessadas, onde se possivel, cada processador puder executar uma subpopulação.

3.6.1.3 Algoritmo Paralelo de granularidade Grossa.

Este modelo é caracterizado por conter um número pequeno conjunto de subpopulações (demes) com muitos indivíduos, conforme mostrado na Figura 22. Ele também é conhecido como algoritmo genético distribuído por sua implementação ser na maioria das vezes em computadores MIMD de memória distribuída (NOWOSTAWSKI; POLI, 1999). Os demes nesse modelo evoluem em paralelo e de forma independente. Esse modelo é conhecido também pelo tempo

relativamente longo que eles exigem para processar uma geração dentro de cada deme, isso se dá pois em cada deme os indivíduos evoluem da mesma forma que em um algoritmo sequencial.

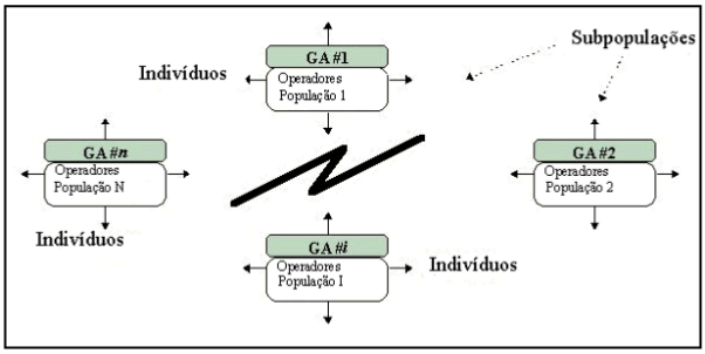

Figura 22 – Algoritmo genético de granularidade grossa
(PESSINI et al., 2003)

A comunicação entre subpopulações ocorre de maneira periódica. Essa operação é chamada migração e tem a função de permitir a troca e material genético entre as subpopulações.

3.7 Considerações Finais do Capítulo.

Neste capítulo os algoritmos genéticos e a computação evolucionária foram apresentados e detalhados, mostrando suas características e finalidades. Cada função de um algoritmo genético genérico foi descrita e explicada. Ao final foram discutidas e apresentados possibilidades de implementações paralelas de tais algoritmos genéticos.

4 APLICAÇÕES E RESULTADOS

As experiências e os resultados coletados neste trabalho foram realizados nas dependências do Laboratório de Bioinformática e Computação de Alto Desempenho (LabioCad), localizado no prédio do ICEN (Instituto de Ciências Exatas e Naturais) . A infraestrutura de hardware foi composta com duas máquinas com mesmas configurações ligadas em rede local por meio de um switch D-Link DGS-1016D. A máquina dedicada às tarefas de controle do cluster e codificação dos algoritmos (máquina mestre) chamada: Nó 01, e outra máquina como nó de processamento escravo para o processamento paralelo, a máquina Nó 02. A Figura 23 mostra o esquema geral da infraestrutura. Ambas as máquinas são equipadas com aceleradores gráficos (GPUs) NVIDIA. A configuração de cada máquina do cluster está descrita na tabela 1:

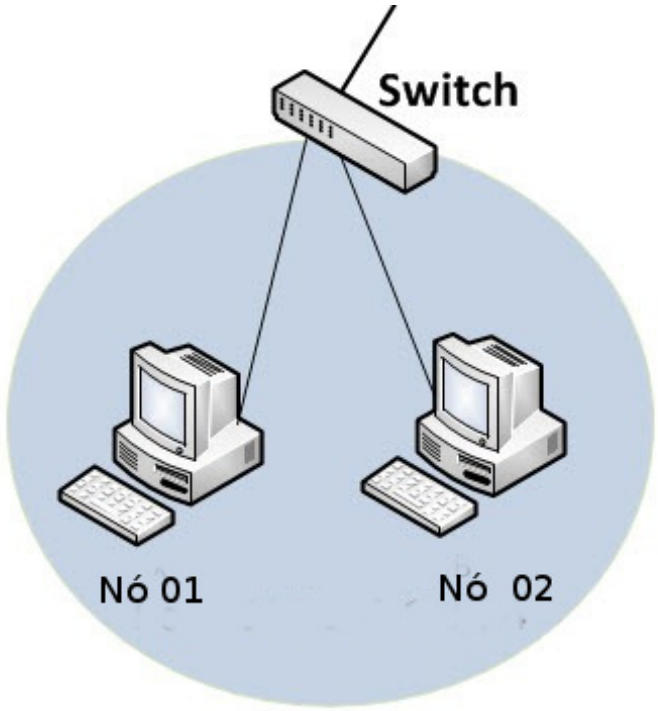

Figura 23 – Infraestrutura do Ambiente
(Autor)

Tabela 1 – Especificações de Cada Máquina

CPU	Intel(R) Core(^{TM}) I5 3.10GHz
Memória	1600 MHZ 8GB
GPU	NVIDIA GeForce GT 630/PCIe/SSE2 96 núcleos 1400MHz
Memória da GPU	4GB

O sistema operacional das duas máquinas foi o GNU/Linux Debian 8 (Jessie). Além do sistema operacional outros software e compiladores foram usados como compilador GCC da linguagem C/C++ , NVCC como compilador NVIDIA/CUDA e o OMNI compiler como compilador das tecnologias XcalableMP, OpeanACC e XcalableACC, além do editor de texto BlueFish. As configurações de software usadas no trabalho são descritas na tabela 2;

Tabela 2 – Especificações de Software

Compilador da linguagem C	GCC 4.9.2
Versão do CUDA	7.5
Compilador CUDA	NVCC 7.5.17
Compilador OpenACC/XcalableMP/XcalableACC	OMNI compiler 1.0.3
Editor de texto	BlueFish Editor 2.2.6

4.1 Aplicações.

As duas aplicações desenvolvidas no trabalho são algoritmos genéticos com características evolutivas, eles foram escritos usando como base a linguagem de programação C. Os algoritmos são compostos basicamente de sete funções: inicialização, avaliação de *fitness*, seleção de pais, mutação, *crossover*, mutação e seleção de sobreviventes, sendo que dessas sete, seis (com exceção da inicialização) são executadas várias vezes dentro de um *loop* (número de gerações). A função dessa aplicação é processar uma quantidade de indivíduos com seus respectivos valores de *fitness*, passando pelas etapas citadas acima visando ao final de um número determinado de gerações, produzir a melhor população possível e destacar também o melhor indivíduo baseado em seu valor de *fitness*.

O fitness da população será avaliado de acordo com uma determinada função de avaliação. Nos experimentos foram desenvolvidos dois algoritmos, um deles foi avaliado usando a função *De Jong's 1* (também conhecida como função esfera) e outro foi avaliado com a função *Axis parallel hyper-ellipsoid*. As duas funções foram escolhidas de pelo critério aleatório, dentre um conjunto de funções que mais são adaptadas ao processamento em placas gráficas.

Ambas as funções são multidimensionais, os seus domínios estão dentro do escopo -5,12 e 5,12 e seus valores globais mínimos (valor ótimo) de função são iguais a 0. As duas funções são descritas abaixo:

- Função *De Jong's 1*

$$f(x) = \sum_{i=1}^{n} x_i^2 \qquad \text{-5.12} \leq x_i \leq 5.12$$

Mínimo global:

$$f(x) = 0$$

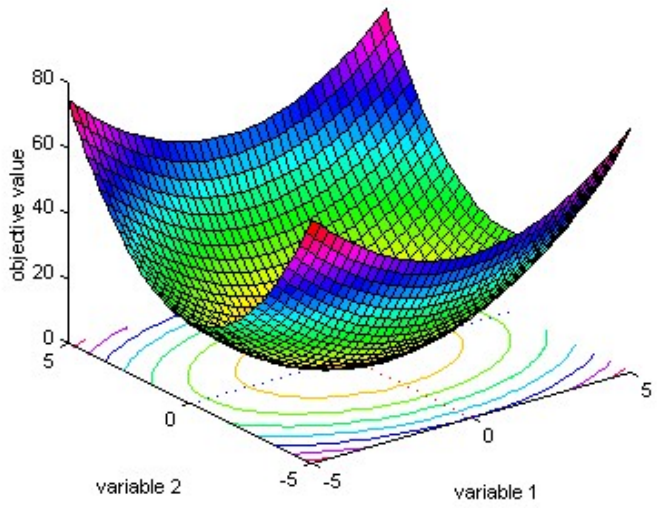

Figura 24 – Representação gráfica da função *De Jong's 1*

- Função *Axis parallel hyper-ellipsoid*

$$f(x) = \sum_{i=1}^{n} 5i * x_i^2 \qquad \text{-5.12} \leq x_i \leq 5.12$$

Mínimo global:

$$f(x) = 0$$

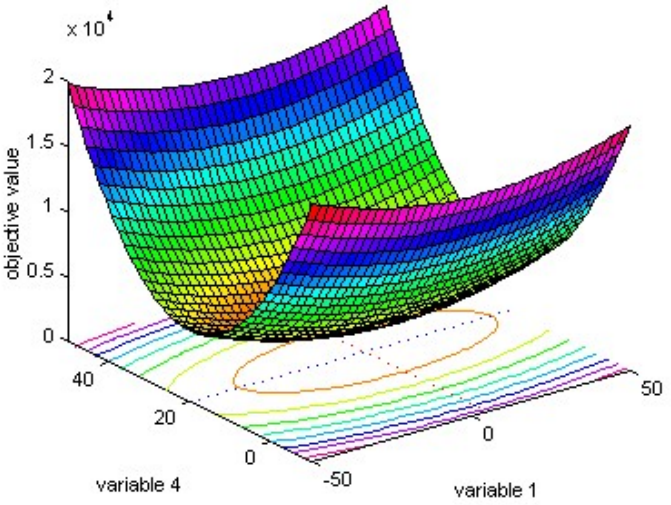

Figura 25 – Representação gráfica da função *Axis parallel hyper-ellipsoid*

Desenvolveram-se três implementações de cada um dos algoritmos com o objetivo de se avaliar o ganho de performance, com base no tempo de execução, assim como destacar o melhor indivíduo (melhor solução) e o valor de *fitness* do melhor indivíduo para cada implementação. Partindo-se de uma solução sequencial escrita apenas em linguagem C, desenvolveram-se versões

paralelas também em linguagem C, porém com adaptações para execução em uma GPU e para duas GPUs.

Os dois algoritmos são quase idênticos, variando apenas a função de avaliação. Nos dois algoritmos e em todas as suas três implementações se fizeram necessárias a definição dos seguintes parâmetros: número de indivíduos (população), quantidade de genes de cada indivíduo e número de gerações. Sendo que todas a versões dos dois algoritmos foram definidos com os mesmos parâmetros, o número de indivíduos: 500, a quantidade de genes: 400 e o número de gerações: 100000.

4.1.1 Versão Sequencial.

A versão sequencial do algoritmo foi desenvolvida puramente na linguagem C, e compilada com compilador GCC. Sua implementação foi a base para as outras implementações. O algoritmo 2 mostra sua implementação em pseudocódigo destacando as partes principais do código.

Algoritmo 2: Pseudocódigo da Versão Sequencial dos Algoritmos

```
1  NCROMOSSOMOS = 500;
2  NGENES = 400;
3  NGERAÇÕES = 100000;
4  início
5      Pop[NCROMOSSOMOS][NGENES],Sele[NCROMOSSOMOS][NGENES];
6      FitPop[NCROMOSSOMOS],FitSele[NCROMOSSOMOS];
7      INICIALIZAÇÃO(Pop)
8      AVALIAÇÃO(Pop,FitPop)
9      i = 0;
10     para i < NGERAÇÕES faça
11         SELEÇÃO DE PAIS(Pop,FitPop,Sele)
12         RECOMBINAÇÃO(Sele)
13         MUTAÇÃO(Sele)
14         AVALIAÇÃO(Sele,FitPop)
15         SELEÇÃO DE SOBREVIVENTES(Pop,FitPop,Sele,FitSele)
16     fim
17 fim
```

Nas linhas 5 e 6 do algoritmo são definidos os vetores e matrizes que irão abrigar os dados da população e seus fitness, assim como os *arrays* temporários para processamento das etapas do algoritmo. Na linhas 7 é realizada a inicialização da população, na linha 8, a primeira avaliação de *fitness* é realizada com essa primeira população. Essas duas etapas são executadas apenas uma vez.

A partir da linha 10 até a linha 16 são realizadas todas as etapas do AG pelo número de gerações pré-determinado. Essa etapa é a mais exaustiva, pois nela ocorrem as fases essenciais para a população se tornar mais apta. O número de gerações tem que ser grande justamente para que haja mais chance dos valores de *fitness* dos indivíduos se aproximar de uma solução ótima.

4.1.2 Versão Paralela com uma GPU.

A versão dos algoritmos para uma GPU foi desenvolvida com a linguagem C com o incremento de diretivas OpenACC e algumas adaptações de código para que trechos específicos do código sejam executados em GPU. Esta versão foi compilada com o compilador OMNI Compiler. Também nessa versão do algoritmo é usado o conceito de programação híbrida, pois a primeira parte do código é executado somente em CPU. O trecho de código que faz a inicialização da população e a primeira operação de cálculo de *fitness* é executada no *host* (CPU), já a parte das execuções das várias gerações é executada no *Kernel* (GPU). A representação do código pode ser visto no algoritmo 3.

Assim como na versão do algoritmo sequencial, as linhas 5 e 6 são de definições de vetores e matrizes, mas com a definição de mais dois vetores importantes nas linhas 7 e 8: rand e normal. Esses dois vetores guardam números aleatórios para serem passados para dentro do *kernel*. Essa definição é feita ainda no *host* pois o *kernel* não pode gerar números aleatórios e nem chamar funções definidas no *host*. Por isso o vetor "rand"é preenchido com valores aleatórios de ponto flutuante da biblioteca da função *rand* que pertence a biblioteca *stdlib.h* e o vetor "normal"é preenchido com valores de uma função Gaussiana, para então serem passados para dentro do *kernel*.

Nas linhas 9 e 10 a inicialização e a primeira avaliação de *fitness* continuam a serem executadas no *host*. Na linha 12 é inserida a primeira diretiva OpenACC: *#pragma acc data*. Essa diretiva tem a função de movimentação de dados, ou seja, ela transfere os dados do *host* para o *kernel* e também pode fazer o caminho inverso: transferir os dados do *kernel* para o *host*. O *loop* principal da linha 14 a linha 25 é executada totalmente dentro do *kernel*. As diretivas *#pragma acc parallel loop* acima de cada função tem o objetivo de distribuir o trabalho para

cada um dos pequenos núcleos da GPU.

Algoritmo 3: Pseudocódigo da Versão com uma GPU dos Algoritmos

```
1  NCROMOSSOMOS = 500;
2  NGENES = 400;
3  NGERAÇÕES = 100000;
4  início
5      Pop[NCROMOSSOMOS][NGENES],Sele[NCROMOSSOMOS][NGENES];
6      FitPop[NCROMOSSOMOS],FitSele[NCROMOSSOMOS];
       // vetores de numeros aleatórios
7      rand[8000000];
8      normal[8000000]
9      INICIALIZAÇÃO(Pop)
10     AVALIAÇÃO(Pop,FitPop)
11     i = 0;
12     #pragma acc data copy(Pop,Fitpop,rand,normal) create(Sele,FitSele)
13     {
14     para i < NGERAÇÕES faça
15         #pragma acc parallel loop
16         SELEÇÃO DE PAIS(Pop,FitPop,Sele)
17         #pragma acc parallel loop
18         RECOMBINAÇÃO(Sele)
19         #pragma acc parallel loop
20         MUTAÇÃO(Sele)
21         #pragma acc parallel loop
22         AVALIAÇÃO(Sele,FitPop)
23         #pragma acc parallel loop
24         SELEÇÃO DE SOBREVIVENTES(Pop,FitPop,Sele,FitSele)
25     fim
26     }
27 fim
```

4.1.3 Versão Paralela com duas GPUs.

A versão dos algoritmos para duas GPUs foi desenvolvida também predominantemente com a linguagem C, porém ela passou por maiores modificações e adaptações e também um maior incremento de diretivas, como pode-se observar no algoritmo 4. Essa versão de código foi compilada pelo compilador OMNI Compiler. Nos códigos foram incrementadas, além das diretivas do OpenACC, novas diretivas da tecnologia XcalableMP. Essas diretivas são as responsáveis por distribuir os dados e as cargas de trabalho entre os nós do cluster. Já as diretivas

do OpenACC farão o trabalho de atribuir os dados e as execuções para as GPUs.

A união dessas duas diretivas dessas tecnologias formam no código o XcalableACC, que é o principal fator para o grande desempenho que as aplicações apresentaram. Além disso, como o XMP tem a característica de formar um espaço de memória comum entre os nós como um sistema de memória compartilhada, essa versão de algoritmo se enquadra na categoria de um algoritmo genético mestre-escravo com população global. O espaço de endereçamento comum armazena toda a população e cada um dos nós realiza as operações avaliação, mutação e *crossover*.

Das linhas 5 a 8 o algoritmo é idêntico a versão com uma GPU, com as definições de vetores e matrizes. Já a partir das linhas 9 a 17 são definidas as diretivas de inicialização do XMP. Essas diretivas são as responsáveis por produzir o espaço de memória abstraído das duas máquinas, mapear esse espaço para cada máquina e alinhar cada vetor e matriz ao espaço de memória definido.

A inicialização e a primeira avaliação de *fitness* ainda são realizadas no *host*, conforme mostra as linhas 18 e 19. Na linha 21 a primeira diretiva OpenACC é introduzida com a função de movimentação de dados para dentro do *kernel*. A linha 23 a diretiva *#pragma xmp loop* tem a função de compartilhar o trabalho para cada máquina, logo abaixo na linha 24 a diretiva *#pragma acc parallel loop*, por sua vez, atribui os dados e trabalho já divididos pelo XMP, agora para

serem executadas nas GPUs de cada máquina.

Algoritmo 4: Pseudocódigo da Versão com duas GPU dos Algoritmos

```
1  NCROMOSSOMOS = 500;
2  NGENES = 400;
3  NGERAÇÕES = 100000;
4  início
5      Pop[NCROMOSSOMOS][NGENES],Sele[NCROMOSSOMOS][NGENES];
6      FitPop[NCROMOSSOMOS],FitSele[NCROMOSSOMOS];
       // vetores de numeros aleatórios
7      rand[8000000];
8      normal[8000000]
       // diretivas do XMP
9      #pragma xmp nodes p(1,*)
10     #pragma xmp template t(0:80000000-1,0:80000000-1)
11     #pragma xmp distribute t(BLOCK,BLOCK) onto p
12     #pragma xmp align Pop[i][*] with t(i,*)
13     #pragma xmp align Sele[i][*] with t(i,*)
14     #pragma xmp align FitPop[i] with t(i,*)
15     #pragma xmp align FitSele[i] with t(i,*)
16     #pragma xmp align rand[i] with t(i,*)
17     #pragma xmp align normal[i] with t(i,*)
18     INICIALIZAÇÃO(Pop)
19     AVALIAÇÃO(Pop,FitPop)
20     i = 0;
21     #pragma acc data copy(Pop,Fitpop,rand,normal) create(Sele,FitSele)
22     {
       // diretivas combinadas(XcalableACC)
23     #pragma xmp loop on t(i,*)
24     #pragma acc parallel loop
25     para i < NGERAÇÕES faça
26         SELEÇÃO DE PAIS(Pop,FitPop,Sele)
27         RECOMBINAÇÃO(Sele)
28         MUTAÇÃO(Sele)
29         AVALIAÇÃO(Sele,FitPop)
30         SELEÇÃO DE SOBREVIVENTES(Pop,FitPop,Sele,FitSele)
31     fim
32     }
33 fim
```

4.2 Resultados.

Nas implementações dos dois algoritmos e suas três versões foram obtidos como resultados saída o melhor indivíduos, sua localização dentro da população e seu respectivo valor de *fitness*. As tabelas 3 e 4 mostram esses indivíduos e seus dados para cada uma das funções de otimização usadas:

Tabela 3 – Melhores Indivíduos (Função *De Jong's 1*)

Versão do Algoritmo	Localização do indivíduo	Valor de Fitness
Sequencial(CPU)	265	0.000000
Paralelo com uma GPU	178	0.000992
Paralelo com duas GPUs	45	0.011183

Tabela 4 – Melhores Indivíduos (Função *Axis parallel hyper-ellipsoid*)

Versão do Algoritmo	Localização do indivíduo	Valor de Fitness
Sequencial(CPU)	447	0.000000
Paralelo com uma GPU	87	0.002430
Paralelo com duas GPUs	137	0.004369

Em ambas as funções, na versão sequencial (CPU) do algoritmo o melhor indivíduo teve seu *fitness* exatamente igual ao valor mínimo da função, o valor 0 (zero). Já nas implementações com uma e duas GPUs os melhores indivíduos tiveram seus valores muito próximo de 0, mostrando que as suas respectivas implementações são aptas a alcançar o objetivo do algoritmo.

Os tempos de processamento dos dois algoritmos e suas respectivas implementações são mostrados nas Figuras 26 e 27.

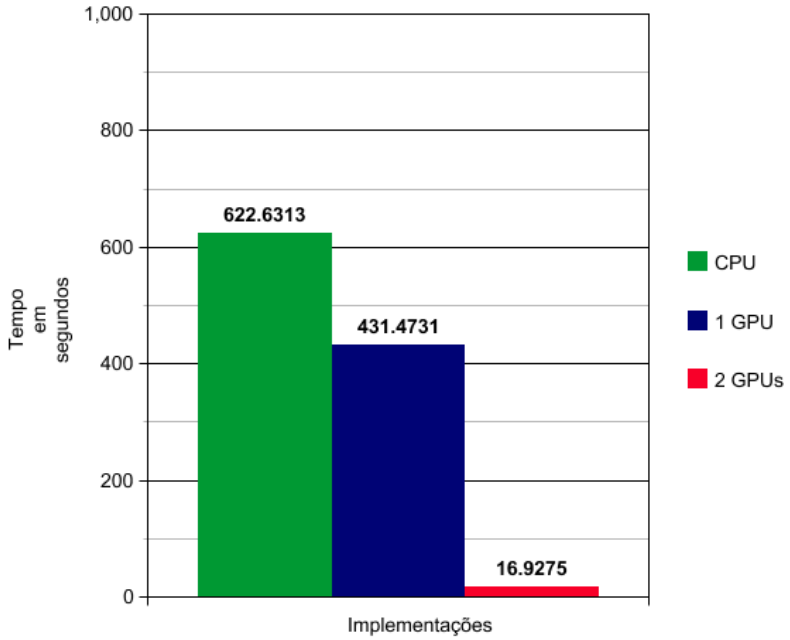

Figura 26 – Tempo de Processamento do Algoritmo usando a Função *De Jong's 1*

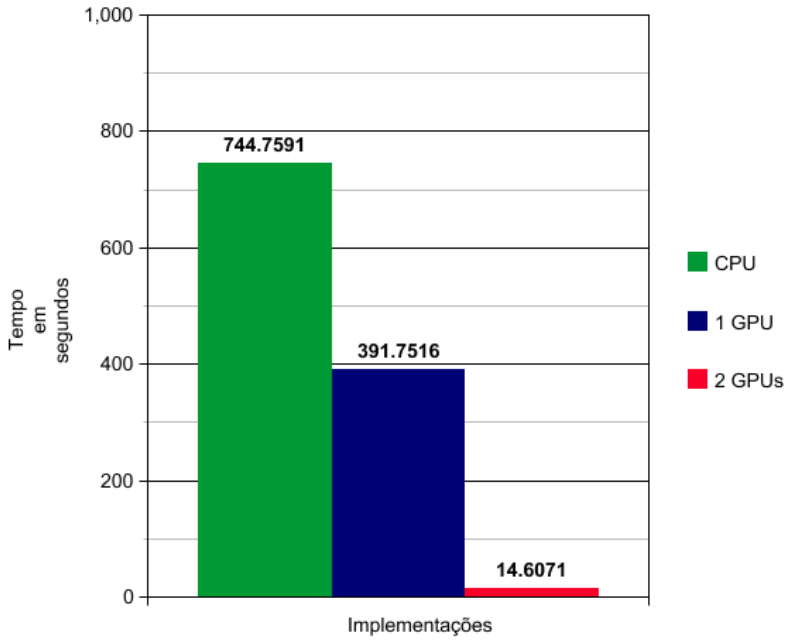

Figura 27 – Tempo de Processamento do Algoritmo usando a Função *Axis parallel hyper-ellipsoid*

Observa-se analizando os gráficos que, para ambas as versões dos algoritmos, há uma redução de tempo de processamento significativo entre as implementações em CPU e a implementação com uma GPU. Há uma redução de tempo ainda mais significativo quando se utiliza duas GPUs para o processamento da aplicação.

Teoricamente, a versão com uma GPU deveria obter uma redução de tempo de processamento maior em relação a versão em CPU. O motivo para que essa redução não ter sido alcançada se dá pois na implementação em uma GPU uma grande quantidade de número ale-

atórios (8 milhões de valores de ponto flutuante) tem que ser gerados em dois vetores (rand e normal). Esses vetores são definidos e gerados ainda no *host* da aplicação, o que eleva o tempo de processamento nesta etapa. Além disso, também se leva um tempo considerável para mover a grande quantidade de dados desses dois vetores para dentro do *kernel*, o que também eleva o tempo de processamento geral da aplicação.

A versão das implementações com duas GPUs é consideravelmente mais rápida do que a versão com uma GPUs pois utiliza os mecanismos do XMP em pontos chaves da aplicação. O XMP abstrai tanto as memórias principais quanto as memórias da GPUs para uma memória global para abrigar os dados da aplicação. A partir daí todo processamento é realizado pelas duas GPUs. Além disso, a parte da aplicação que faz a geração dos números aleatórios para os dois vetores é paralelizada nas duas máquinas, isso é feito pelo XMP ainda no *host* antes de serem envidas ao *kernel*. Portanto, só nessa etapa há significatico ganho de desempenho antes mesmo do processamento masiço nas GPUs.

5 CONCLUSÃO

A implementação sequencial do algoritmo se mostrou mais precisa em seus resultados, sendo que nas implementações das duas funções um indivíduo com valor ótimo foi encontrado (valor 0). Porém seus tempos de execução foram os mais elevados. Resultados justificados pelo uso do processamento sequencial usando uma CPU, sendo assim versão mais demorada. Já as versões com processamento com uma GPU, mesmo com resultados não tão precisos, suas execuções foram mais rápidas que as versões sequenciais com uma CPU, justificando assim seu uso para ganho de desempenho computacional.

O maior destaque se deu com o desempenho alcançado com o acréscimo de mais uma GPU, formando a estrutura paralela de aceleradores gráficos. Seus resultados também não foram tão precisos, mas o ganho de desempenho foi consideravelmente grande. A divisão do trabalho de computação realizada pelo XMP para dentro das máquinas e do OpenACC para as GPUs diminuíram o tempo de execução do algoritmo em muitos segundos, chegando a ser em média até 25 vezes mais rápida do que que a execução usando apenas uma GPU.

Com isso, conclui-se que o uso de aceleradores gráficos em sistemas de clusters computacionais se mostraram uma excelente alternativa para a execução de aplicações de algoritmos genéticos, e de outras aplicações, que possam ser adaptadas a essas estruturas. Da mesma forma, pode-se afirmar que o uso do XMP facilitou e acelerou o processo de programação com memória distribuída, reduzindo grande parte da complexidade, antes necessária para esse tipo de programação. A união das duas abordagens (XMP e GPU) foi uma abordagem interessante na busca por um melhor desempenho.

Como trabalhos futuros, pretende-se:

- Aumentar a estrutura computacional utilizada para mais placas gráficas;
- Realizar testes com outras arquiteturas, como por exemplo, o rCUDA;
- Analisar e avaliar as variações dos parâmetros dos algoritmos como, número de populações, de genes e gerações;
- Teste novos algoritmos, explorando outras funções de otimização com um nível de complexidade ainda maior.

REFERÊNCIAS

CANTÚ-PAZ, E. A survey of parallel genetic algorithms. **IlliGAL report**, v. 97003, 1997.

CHENG, J.; GROSSMAN, M.; MCKERCHER, T. **Professional Cuda C Programming**. [S.l.]: John Wiley & Sons, 2014.

COELHO, S. A. Introdução a computação paralela com o open mpi. **Simpósio Mineiro**, 2013.

EIBEN, A. E.; SMITH, J. E. et al. **Introduction to evolutionary computing**. [S.l.]: Springer, 2003. v. 53.

GROUP, X. S. W. et al. **XcalableMP Specification Version 1.2. 1**. [S.l.]: Nov, 2014.

GUIMARÃES, F. G.; RAMALHO, M. C. Implementação de um algoritmo genético. **Trabalho referente à disciplina "Otimização", junho de**, 2001.

JIN, G. et al. Implementation and performance evaluation of the hpc challenge benchmarks in coarray fortran 2.0. In: IEEE. **Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International**. [S.l.], 2011. p. 1089–1100.

JÚNIOR, S.; LEITE, E. et al. Uso de algoritmo genético no ajuste linear através de dados experimentais. Universidade Federal da Paraíba, 2015.

KIESSLING, A. An introduction to parallel programming with openmp. **The University of Ediburgh**, 2009.

LEE, J.; SATO, M. Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. In: IEEE. **Parallel Processing Workshops (ICPPW), 2010 39th International Conference on**. [S.l.], 2010. p. 413–420.

MORAES, S. R. d. S. Computação paralela em cluster de gpu aplicado a problema da engenharia nuclear. 2012.

MORALES, L. A. B. Criação de um cluster beowulf, desenvolvimento de uma aplicação utilizando processamento paralelo e análise de seu desempenho. 2008.

NAKAO, M. et al. Productivity and performance of global-view programming with xcalablemp pgas language. In: IEEE COMPUTER SOCIETY. **Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)**. [S.l.], 2012. p. 402–409.

NAKAO, M. et al. Xcalableacc: Extension of xcalablemp pgas language using openacc for accelerator clusters. In: IEEE. **Accelerator Programming using Directives (WACCPD), 2014 First Workshop on**. [S.l.], 2014. p. 27–36.

NOMIZU, T. et al. Implementation of xcalablemp device acceleration extention with opencl. In: IEEE. **Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International**. [S.l.], 2012. p. 2394–2403.

NOWOSTAWSKI, M.; POLI, R. Parallel genetic algorithm taxonomy. In: IEEE. **Knowledge-Based Intelligent Information Engineering Systems, 1999. Third International Conference**. [S.l.], 1999. p. 88–92.

NVIDIA, C. **NVIDIA CUDA C Programming Guide**. 2012.

PACHECO, P. **An introduction to parallel programming**. [S.l.]: Elsevier, 2011.

PESSINI, E. C. et al. Algoritmos genéticos paralelos: uma implementação distribuída baseada em javaspaces. Florianópolis, SC, 2003.

POZO, A. et al. Computação evolutiva. **Universidade Federal do Paraná, 61p.(Grupo de Pesquisas em Computação Evolutiva, Departamento de Informática-Universidade Federal do Paraná)**, 2005.

ROCHA, J. Cluster beowulf aspectos de projeto e implementacao. **Mestrado, Curso de Mestrado em Engenharia Elétrica, Centro Tecnológico, Universidade Federal do Pará, Belém**, 2003.

SANDERS, J.; KANDROT, E. **CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents**. [S.l.]: Addison-Wesley Professional, 2010.

SILVA, L. Modelo híbrido de programação paralela para uma aplicação de elasticidade linear baseada no método dos elementos finitos. **Brasília, UnB**, 2006.

TADAIESKY, V. W. A. Avaliação de técnicas de paralelização de algoritmos bioinspirados utilizando computação gpu: Um estudo de casos para otimização de roteamento em redes ópticas. 2015.

WOLFE, M. **The OpenACC application programming interface**. [S.l.]: Version, 2013.

YU, X.; GEN, M. **Introduction to evolutionary algorithms**. [S.l.]: Springer Science & Business Media, 2010.