Introdução a Processamento de Alto desempenho

Esbel T. Valero Orellana[1]

[1]Nucleo de Biologia Computacional e Gestão de Informações Biotecnológicas –
Universidade Estadual de Santa Cruz (UESC)
Campus Soane Nazaré de Andrade, km 16 Rodovia Ilhéus-Itabuna
CEP 45662-900. Ilhéus-Bahia

evalero@nbcgib.uesc.br

Abstract. *This article is a review of some general topics on high-performance computing. It covers the main technologies available and the techniques most used on parallel programming. The use of multithreaded programming, messaging passages and CUDA architecture are presented in summary form. The literature reviewed includes some of the texts used as a reference for advanced courses.*

Resumo. *O presente artigo pretende fazer um revisão de alguns tópicos gerais sobre processamento de alto desempenho. São abordadas as principais tecnologias disponíveis e as técnicas de programação paralela mais utilizadas. Programação utilizando multithreads, troca de mensagens e a arquitetura CUDA são apresentadas de forma resumida. A revisão bibliográfica inclui alguns dos textos utilizados como referência em cursos avançados.*

1. Introdução

A chegada das tecnologias baseadas em microprocessadores com múltiplos núcleos (*multicore*) e com muitos núcleos (*many-core*) fez de 2003 um marco importante para o Processamento de Alto Desempenho (PAD). Enquanto que os *cluster Beowulf*, introduzidos dez anos antes, permitiram levar o PAD para as instituições de pequeno e médio porte, os lançamentos daquele ano levaram o PAD diretamente aos computadores pessoais.

Os primeiros anos do novo século marcaram o fim de quase duas décadas em que os microprocessadores, baseados numa única unidade central de processamento (CPU), impulsionaram o aumento contínuo de desempenho e a redução dos custos das aplicações de computador. Durante anos, cada nova geração de microprocessadores trazia consigo maior velocidade de processamento e os programadores contavam com estes avanços tecnológicos para melhorar o desempenho dos seus aplicativos. Para os pesquisadores e usuários de computação de alto desempenho as coisas não eram muito diferentes. As arquiteturas de memória compartilhada estavam fora do alcance de uma grande parte parte das instituições, que exploravam ao máximo as potencialidades de arquiteturas de memória compartilhada como o *cluster* tipo *Beowulf*.

Entretanto por volta de 2003, os limites impostos pelo aumento do consumo de energia e da dissipação de calor frearam os avanços tecnológicos nesta área. A solução encontrada pelos fabricantes de *hardware* para as limitações, que impediam continuar aumentando a frequência do *clock*, foi passar a incluir várias unidades de processamento em cada chip. Como resultado o aumento de desempenho passou a se dar não unicamente pela frequência, como também pela quantidade de unidades de processamentos, também conhecida como núcleos, incluídos em cada chip. Desta forma os computadores equipados com os novos microprocessadores, chamados de *multicore*, passaram a ser computadores paralelos.

Por outro lado, um outro tipo de hardware, que adiciona uma grande quantidade de núcleos num dispositivo, começou a ganhar espaço no PAD. A tecnologia *many-core* chegou na forma de unidades de processamento gráficos (GPU), com uma grande quantidade de núcleos (*cores*) com capacidade de executar tarefas maciçamente paralelas. A quantidade de núcleos das GPUs, da mesma forma que na arquitetura *multicore*, se duplica a cada nova geração.

Ante o novo senário, os desenvolvedores de programas tiveram que se adaptar às novas tecnologias para conseguir aplicações cada vez mais eficientes. Aqui recomendamos algumas referências interessantes sobre o tema como (W.-mei Hwu, Keutzer, & Mattson, 2008), (Sutter & Larus, 2005). O processamento paralelo se fez uma peça imprescindível para os programadores que pretendem conseguir melhores desempenhos das novas tecnologias no mercado. Hoje, quase dez anos depois, nos encontramos com um senário totalmente novo no qual um simples laptop pode possuir um desempenho da ordem dos teraflops.

As mudanças também chegaram para o PAD. Os modernos *clusters* passaram a utilizar máquinas multiprocessadas e, em alguns casos, equipadas com poderosas GPUs. As novas arquiteturas híbridas, interligadas por redes cada vez mais rápidas, incrementaram a capacidade computacional disponível. Simultaneamente novas linhas de pesquisa foram criadas para explorar ao máximo as potencialidades das mesmas.

Neste contexto acontece a primeira Escola Regional de Alto Desempenho (ERAD) da regional Nordeste. Este texto pretende servir de base para o minicurso de Introdução a Processamento Paralelo e foi preparado com base na experiência do autor após vários semestres ministrando a disciplina CET107 – Processamento Paralelo, oferecida como optativa no curso de Ciência da Computação da Universidade Estadual de Santa Cruz (UESC).

O material deste texto está organizado da seguinte maneira, primeiramente faremos uma rápida análise na Seção 2 dos motivos para utilizar PAD. Na Seção 3 se faz uma apresentação resumida de diferentes arquiteturas paralelas. As Seção 4 apresenta alguns tópicos importantes relacionados com programação paralela assim como três dos principais paradigmas utilizados hoje em PAD: programação *multithread* com OpenMP; troca de mensagens com MPI e programação de dispositivo gráficos com CUDA. Na Seção 5 se apresenta um exemplo, baseado no problema de multiplicação de matrizes, que será tratado no minicurso pata demonstrar alguns dos conceitos introduzidos na Seção 4. Finalmente na Seção 6 se fazem algumas considerações finais sobre o tema.

Este texto se propões servir como base para estudos futuros, mais avançados, sobre os temas abordados. Com tal finalidade foi incluída uma vasta bibliografia que pode servir como referência. Pelas limitações de tempo e espaço deste minicurso, está fora do escopo deste texto ser uma referência detalhada em programação paralela.

Uma boa parte do material utilizado para preparar este texto e o minicurso em questão foi cedido pelo Prof. Dr Dany Sanchez, um dos primeiros professores e pesquisadores de Processamento Paralelo na UESC.

2. Sobre Aumento Contínuo da Demanda por PAD

A evolução do desempenho dos 500 maiores computadores do mundo nos últimos anos pode ser acompanhada através da página do projeto Top500, http://www.top500.org. De seis em seis meses é liberada uma lista elaborada com base num *benchmark* específico de desempenho, contendo os principais detalhes dos 500 maiores computadores do mundo. Para quem não está familiarizado com as magnitudes neste tipo de gráfico a Tabela 1 apresenta as unidades utilizadas para quantificar o desempenho de um destes supercomputadores.

1. Tabela: Unidade de medida de desempenho de computadores, FLOPS ou flops é acrônimo de Floating point Operations per Second

Nome	Número de Operações por segundo
kiloflops	10^3
megaflops	10^6
gigaflops	10^9
teraflops	10^{12}
petaflops	10^{15}
hexaflops	10^{18}

Olhando o gráfico de evolução de desempenho publicado no relatório mais recente do Top500, apresentado na Figura 1, fica em aberto a pergunta: qual a força motriz que impulsiona esta corrida desenfreada por um maior desempenho? Procurando uma resposta para esta pergunta podemos começar por algumas constatações simples. Para começar, a formulação e resolução de problemas cada dia mais complexos nos leva a níveis mais profundos de conhecimento. As diversas áreas das ciências e as engenharias tem contribuído, de uma forma ou de outra, com o surgimento de mais e maiores problemas a ser resolvidos.

De fato os principais paradigmas da ciência e da engenharia estão cada dia mais permeados pela computação. As simulações computacionais tem-se mostrado ferramentas confiáveis e baratas para ser utilizadas na comprovação de teorias, projeção

de sistemas, e na construção de protótipos. As ferramentas computacionais disponíveis devem então permitir a resolução de problemas complexos em um tempo razoável. O PDA passa a ser uma ferramenta importante quando nos deparamos com problemas de grande porte, ou seja, problemas que não podem ser resolvidos em um intervalo razoável de tempo com os computadores disponíveis no momento. Note que utilizamos "tempo razoável", termo bastante subjetivo e que precisa ser avaliado em cada situação de forma apropriada.

Figura 1. Evolução do desempenho dos 500 maiores computadores (Tomado de http://www.top500.org)

Para exemplificar o conceito de problema de grande porte podemos utilizar um exemplo clássico: o cálculo do movimento de corpos estelares. Basicamente o problema a ser resolvido é estimar as posições dos corpos estelares sendo que: cada corpo é atraído pelas forças gravitacionais exercidas por todos os outros corpos; o movimento de um corpo é modelado calculando o efeito de todas as forças que atuam sobre ele. Se temos N corpos teremos N-1 forças atuando sobre cada corpo e N^2 operações de ponto flutuante (flop) para determinar a posição de todos os corpos. Após determinar a posição de cada um deles o cálculo deve ser repetido de forma iterativa.

Colocando o exemplo anterior em números teremos que a galáxia tem aproximadamente 10^{11} estrelas. Se cada operação fosse feita em 1 μ**s** teríamos 10^{16} **s** ou 1 bilhão de anos. Mesmo aprimorando o algoritmo para conseguir complexidade da ordem de N*log*N somente conseguiríamos terminar cada iteração em 1 ano.

Existem outros inúmeros exemplos de problemas de grande porte em áreas como simulação de biomoléculas, simulação estocástica utilizando Monte Carlo, simulação de reservatório de hidrocarbonetos, processamento de imagens e nanotecnologia. Alguns exemplos práticos podem ser encontrados em (Kirk, W. M. W. Hwu, & W. Hwu, 2010). No entanto a demanda por supercomputadores cada vez mais potentes não se origina apenas da necessidade de reduzir o tempo de processamento. Para conseguirmos um elevado poder computacional requer-se também uma grande capacidade de

armazenamento. Recomenda-se consultar algumas referências sobre o tema como (Parhami, 1999).

Os supercomputadores modernos podem ser caracterizados como computadores paralelos. Isto significa que eles são capazes de operar com múltiplos processos que trabalham juntos na resolução de um único problema. Aqui entendemos processo como uma entidade que engloba um processador e determinada quantidade de memória. O processamento paralelo oferece, em teoria, um poder de cálculo ilimitado sempre que sejamos capazes de instruir a nossos processos sobre como trabalhar juntos. Ou seja, cada processo deve ter seu conjunto de instruções e um conjunto de dados para processar.

Entretanto, a implementação do paralelismo não é uma tarefa simples. O programador deve levar em consideração como fazer a divisão das tarefas entre os processos, deve possuir as ferramentas adequadas para organizar a execução e a comunicação entre os mesmos quando necessário. A ideia por traz dos computadores paralelos é bastante atraente: colocando **p** processadores para resolver o problema teremos **p** vezes mais velocidade de processamento. Mas a realidade nos mostra que o paralelismo tem um custo relacionado com o próprio processo de paralelização, o processo de comunicação entre os processos, o balanceamento de carga e o fato de que os algoritmos não são completamente paralelizáveis.

Uma discussão mais aprofundada sobre o porque a computação paralela chegou para ficar pode ser acompanhada em textos específicos da área como (Chapman, Jost, & Pas, 2007), (Kirk et al., 2010), (Rauber & Rünger, 2010), (Parhami, 1999)

Para entender melhor como funcionam os supercomputadores modernos será discutido, na próxima seção, alguns aspectos relevantes da arquitetura de computadores paralelos.

3. Aspectos Importantes das Arquiteturas com Múltiplos Processadores

A taxonomia de Flynn pode ser utilizada para obter uma classificação inicial que ajude no entendimento das arquiteturas paralelas. Apesar de ter se originado nos anos 70 trata-se de uma classificação ainda válida que se baseia na capacidade dos computadores de lidar com fluxos de instruções e fluxos de dados. Desta forma temos quatro grandes grupo de computadores, como mostra a Tabela 2.

Tabela 2. Classificação segundo a taxonomia de Flynn

	SD (*Single Data*)	**MD** (*Multiple Data*)
SI (Single Instruction)	**SISD**	**SIMD**
MD (Multiple Instruction)	**MISD**	**MIMD**

A categoria SISD engloba aqueles computadores com um único fluxo de instruções atuando sobre um um único fluxo de dados. A arquitetura proposta por *von Newmann*

pode ser inserida nesta categoria. O principal gargalo nesta arquitetura é a transferência de dados entre a memória principal e a CPU. Os computadores equipados com um microprocessador com uma única CPU são comumente apresentados como exemplos da arquitetura de *von Newmann*, apesar da estrutura hierárquica de memória, introduzida nos mesmos para melhorar o fluxo de dados entre a CPU e a memória principal. Já na classe MISD teríamos, na prática, múltiplos fluxos de instruções atuando sobre um único endereço de memória.

As arquiteturas paralelas podem ser encontradas tanto na categoria SIMD quanto em MIMD. No primeiro caso temos um único fluxo de instruções atuando sobre múltiplos fluxos de dados. Nesta classe se encontram as máquinas vetoriais. Já as arquiteturas que se encaixam na categoria MIMD executam múltiplos fluxos de instruções sobre múltiplos fluxos de dados. Ou seja, teremos múltiplos processadores executando seus próprios programas sobre seu próprio conjunto de dados. Mais detalhes sobre a taxonomia de Flynn aplicada a arquiteturas paralelas podem ser obtidos em (Parhami, 1999).

As máquinas MIMD, que chamaremos daqui para frente como máquinas multiprocessadas, podem ser divididas de acordo com a forma em que a memória está disponível. Podemos ter então máquinas de memória compartilhada, também conhecidas como ambiente fortemente acoplado e máquinas de memória distribuída ou de ambiente fracamente acoplado.

As máquinas de memória compartilhada são compostas por um conjunto de processadores que compartilham determinada quantidade de módulos de memória a través de um bus de dados. As novas gerações de processadores *multicore* implementam uma arquitetura de memória compartilhada. A Figura 2 mostra a estrutura de um dos nós de cálculo do CACAU (http://nbcgib.uesc.br/cacau), com dois processadores *quadcore* que compartilham 16 GB de memória RAM.

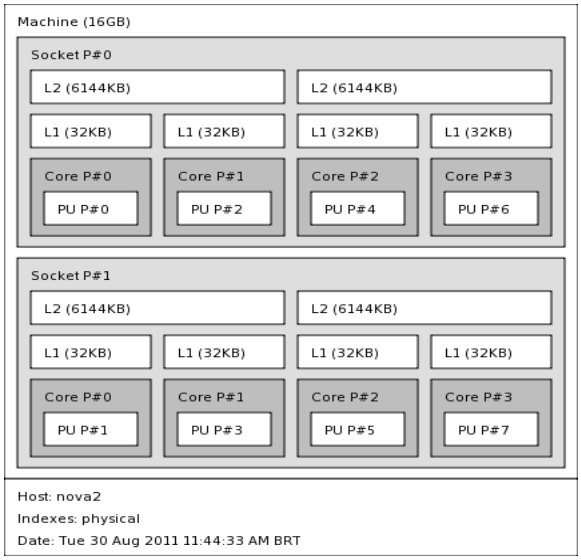

Figura 2: Informação sobre estrutura de um sistema com dois processadores quadcore

Na implementação deste tipo de arquitetura os fabricantes tem que lidar com o problema de como o controle do acesso dos processadores ao *bus* de dados. Decorrente deste problemas surgem outros como a coerência de dados na memória e a velocidade de acesso dos processadores aos diferentes módulos de memória.

Algumas implementações de arquiteturas de memória distribuída se preocupam em garantir acesso a uma velocidade uniforme de todos os processadores a todos os módulos de memória ou UMA (*Uniform Memory Access*). Quando o acesso aos módulos de memória não acontece na mesma velocidade para todos os processadores temos um sistema NUMA (Non-Uniform Memory Access). Na Seção 4 será apresentado o paradigma de programação *multithread* baseado em OpenMP, que permite implementar programação paralela em arquiteturas de MIMD de memória compartilhada.

As máquinas com memória distribuída são formadas por processadores que tem blocos de memória local aos quais apenas eles tem acesso. A comunicações entre os processos acontece através de uma rede de interconexão. O cluster de alto desempenho é um exemplo típico deste tipo de arquitetura. Na Seção 4 será apresentado o paradigma de troca de mensagens utilizando MPI, que permite implementar programação paralela em arquiteturas MIMD de memória compartilhada.

Entretanto o cluster de computadores modernos pode ser analisado como uma arquitetura híbrida, formada por um conjunto de máquinas multiprocessadas, de memória compartilhada, interconectadas numa arquitetura de memória distribuída. Estas arquiteturas introduzem uma nova linha de trabalho em PAD na busca por implementações híbridas, como aquelas baseadas em *multithreads* e troca de mensagens (OpenMPI + MPI) (Adhianto & Chapman, 2007), (Wolf, 2003), (Nakajima, 2005), (Aversa, Dimartino, Rak, Venticinque, & Villano, 2005).

Neste ponto os cursos mais atuais na área de PAD precisam fazer um aparte para comentar a arquitetura das modernos dispositivos GPU. A Figura 3 mostra um esquema que compara a arquitetura de uma CPU *multicore* com uma GPU *many-core*. Uma estrutura formada por uma grande quantidade de núcleos e uma hierarquia própria de memória de alta velocidade, como mostra a Figura 4, garantem um hardware com capacidade para executar tarefas paralelas de grande porte. Na Seção 4 se apresentam alguns aspectos da arquitetura CUDA para programação paralela baseada em dispositivos GPU do fabricante NVIDIA.

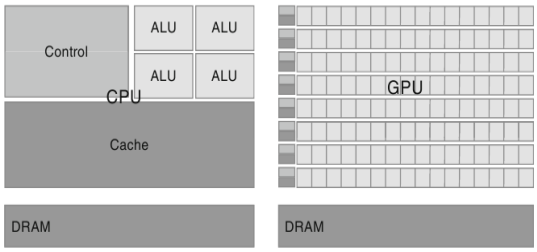

Figura 3: Arquiteturas de uma CPU *multicore* e de uma GPU *many-core*. Tomado de (Kirk et al., 2010)

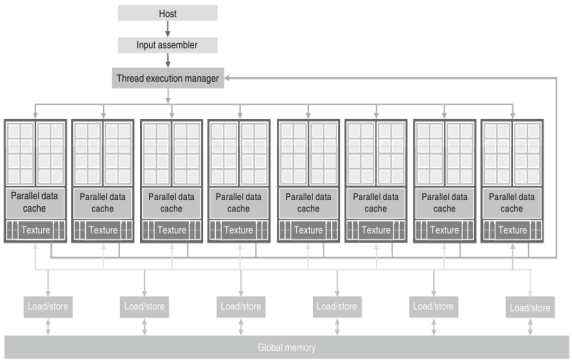

Figura 4: Esquema mais detalhado da arquitetura de uma GPU. Tomado de (Kirk et al., 2010)

Mais detalhes sobre as arquiteturas paralelas, especialmente sobre as novas arquiteturas implementadas nas modernas GPUs, podem ser consultados em (Sanders & Kandrot, 2010), (Kirk et al., 2010), (Chapman et al., 2007). Outros artigos relacionados com aplicações para arquiteturas híbridas envolvendo GPUs (Yang, Huang, & Lin, 2011), (Alonso, Cortina, Martínez-Zaldívar, & Ranilla, 2009).

Para se entender melhor a programação paralela e algumas das ferramentas disponíveis para sua implementação, serão introduzidos a seguir alguns aspectos gerais sobre o tema.

4. Visão Geral sobre Programação Paralela

Antes de começar a falar sobre programação paralela, deve-se analisar alguns aspectos relacionados com programação sequencial. O algoritmo serial ou sequencial é aquele que executa, de forma independente, em um único processador, ao contrário do algoritmo paralelo que roda simultaneamente em em dois ou mais processadores. Para cada algoritmo paralelo existe um algoritmo sequencial que realiza a mesma tarefa.

Para se implementar um algoritmo paralelo é extremadamente importante criar uma versão serial do mesmo. O algoritmo serial serve de ponto de partida, como base para o melhor e mais rápido entendimento do problema e como ferramenta para validar os resultados do programa paralelo. O desempenho do algoritmo serial auxilia também na avaliação do ganho de desempenho e da eficiência do algoritmo paralelo.

Existem diversas métricas para avaliar o desempenho de algoritmos paralelos. Como parte do ERAD será oferecido um minicurso dedicado a avaliação de desempenho em sistemas paralelos. Por este motivo não aprofundaremos muito neste tema. Apresentaremos apenas duas métricas muito importantes que são função do número de processos (p) e do tamanho do problema (n).

A primeira destas métricas é o speedup, uma das medidas mais utilizadas para avaliar o desempenho de uma algoritmo paralelo. O speedup se define como a razão entre o tempo de execução do algoritmo serial $\mathbf{T_s(n)}$, que depende apenas do tamanho **n** do problema, e o tempo de execução do algoritmo paralelo $\mathbf{T_p(n,p)}$, que depende do tamanho do problema e da quantidade **p** de processadores utilizados. De forma geral o speedup oscila entre 0 a quantidade de processadores. Na Tabela 3 são apresentados diferentes situações que podem aparecer ao determinar o speedup de um algoritmo paralelo.

Tabela 3: Caracterização de desempenho com base no speedup

Speedup $\mathbf{S(n, p)= T_s(n)/T_p(n, p)}$	Caracterização
$\mathbf{S(n, p) < 1}$	**Slowdown**, situação indesejável
$\mathbf{1< S(n, p) < p}$	**Sublinear**, comportamento geral
$\mathbf{S(n, p) = p}$	**Linear**, Ideal não, existe sobrecarga
$\mathbf{S(n, p) > p}$	**Supralinear**, situação pouco comum

O speedup pode ser utilizado também para calcular outro parâmetro importante, a eficiência. A eficiência não é mais que a medida de utilização dos processos em um programa paralelo em relação ao programa serial. Desta forma a eficiência é calculada como a razão entre o speedup e a quantidade de processadores.

Para se obter algoritmos paralelos eficientes deve-se levar em consideração alguns aspectos importantes como a divisão equitativa do trabalho entre os processos (balanceamento de carga), minimizar as necessidades de comunicação e o tempo ocioso

dos mesmos, sobrepor operações de comunicação e computação e concentrar as operações de E/S visando minimizar seu efeito.

Não podemos apresentar os aspectos teóricos relacionados com a implementação paralela sem falar das limitações previstas pela lei de Amdahl. Postulada por Gene Amdahl no final dos anos 60, desencorajou durante anos a utilização de paralelismo massivo ao estabelecer um limite superior para o speedup de um algoritmo paralelo. Quiça a melhor forma de entender a lei de Amdahl seja através da referência (Gustafson, 1988), onde, pela primeira vez, colocam-se as limitações desta abordagem e as potencialidades ainda inexploradas na utilização de paralelismo com grande quantidade de processadores. Outra discussão mais atualizada sobre o tema pode ser analisada em (Sun & Chen, 2010)

Existem basicamente dois enfoques na hora de projetar programas paralelos. São eles o paralelismo de dados e o paralelismo de controle. O paralelismo de dados dá-se através do particionamento do conjunto de dados a ser processados em subconjuntos menores que são atribuídos a cada processo. Este enfoque pode ser implementado de forma simples, não é prejudicado pela dependência entre as operações, os programas que utilizam o mesmo são facilmente escalável e geralmente utilizam pouca comunicação entre processos. As implementações paralelas utilizando CUDA em dispositivos GPU são um exemplo de uso de paralelismo de dados.

Entretanto, nem sempre é possível utilizar o paralelismo de dados. Muitas vezes precisamos dividir o problema em tarefas independentes, que podem ser atribuídas a processos diferentes e executadas em paralelo. Neste caso utilizamos o paralelismo de controle. Este enfoque deve considerar a dependência entre as operações, é mais difícil de se implementar e escalonar, e implica, geralmente, em um uso elevado de comunicação entre processos. O construtor paralelo `sections`, utilizado em OpenMP, é um exemplo de implementação de paralelismo de controle. A maior parte dos programas paralelos envolvem, de uma forma ou outra, os dois enfoques ainda que o paralelismo de dados seja mais comumente encontrado.

Finalmente, antes de começar a analisar técnicas de programação paralela, podemos desenhar um roteiro geral que pode ser utilizado para construir um programa paralelo:

1. Implementação sequencial: Analisar, implementar e validar uma solução sequencial para o problema que pretende-se solucionar;
2. Análise da divisão do trabalho: Avaliar a possibilidade divisão do conjunto de dados do problema entre os diferentes processos;
3. Avaliar a viabilidade do paralelismo de dados puro: verificar se o problema pode ser resolvido apenas executando o algoritmo serial nos distintos conjuntos de dados;
4. Análise da necessidade de comunicação: Se o paralelismo de dados não for suficiente identificar as necessidade de comunicação entre os processos;
5. Avaliar a necessidade de paralelismo de controle: Analisar a necessidade de introduzir paralelismo de controle na implementação da solução paralela;

6. Validação da implementação paralela: Verificar a solução paralela com ajuda do algoritmo serial.
7. Análise de desempenho: Avaliar diferentes métricas de desempenho para analisar as características do algoritmo paralelo implementado.

A seguir são apresentados três paradigmas utilizados em programação paralela. Estes paradigmas podem ser utilizados isoladamente ou em conjunto, em implementações híbridas, para se conseguir implementações paralelas mais eficientes.

4.1. Programação multithread com OpenMP

Os computadores multiprocessados de memória compartilhada representam uma das arquiteturas paralelas mais amplamente disponíveis nos dias de hoje. Quase todos os computadores pessoais e laptops da atualidade podem ser considerados como representantes desta classe. Uma das formas de se implementar processamento paralelo é através de programação *multithread*.

Um *thread* não é mais que a menor parte de um processo que pode ser manipulado pelo escalonador do sistema operacional. Os *threads* de um processo se originam a partir da divisão do *thread* principal em dois ou mais *threads*, que podem ser executados em paralelo são atribuídos pelo escalonador a processadores diferentes, mas que compartilham entre sim o mesmo espaço de memória.

As técnicas mais comuns para se trabalhar com *multithread* são: o *threading* explícito e as diretivas de compilação. Neste texto será abordado o uso de diretivas de compilação, através de OpenMP, que consiste em inserir diretivas no código sequencial para informar ao compilador quais regiões devem ser paralelizadas.

O Open specification for Multi Processing, ou simplesmente OpenMP, é um modelo de programação em memória compartilhada que surgiu a partir da cooperação de grandes fabricantes de hardware e software como a Sun, Intel, SGI, AMD, HP, IBM e outras. Projetada para ser utilizada com C\C++ e Fortran, as especificações são desenvolvidas e mantidas pelo grupo OpenMP ARB (Architecture Review Board). Trata-se de um padrão (não é uma linguagem de programação) que define como os compiladores devem gerar códigos paralelos através de diretivas e funções. Por este motivo o resultado depende, em grande medida, do compilador que foi utilizado para gerar o aplicativo.

A versão 1.0 destas especificações para Fortran foi liberada em Outubro de 1997 e um ano depois saíram as especificações para C/C++. Em Novembro de 1999 foi liberada a versão 1.1 para Fortran e em 2000 a versão 2.0 também para Fortran. A versão 2.0 para C/C++ demorou ainda dois anos mas, a partir 2005 com a versão 2.5, começaram a ser disponibilizadas as especificações para Fortran e C/C++ simultaneamente. Em 2008 saiu a versão 3.0 e recentemente, em julho de 2011, foi disponibilizada a versão 3.1.

A Figura 5 mostra um esquema que representa as partes que compõem a API do OpenMP. Basicamente estão disponíveis um conjunto de varáveis de ambiente, uma biblioteca de funções e um conjunto de diretivas de compilação.

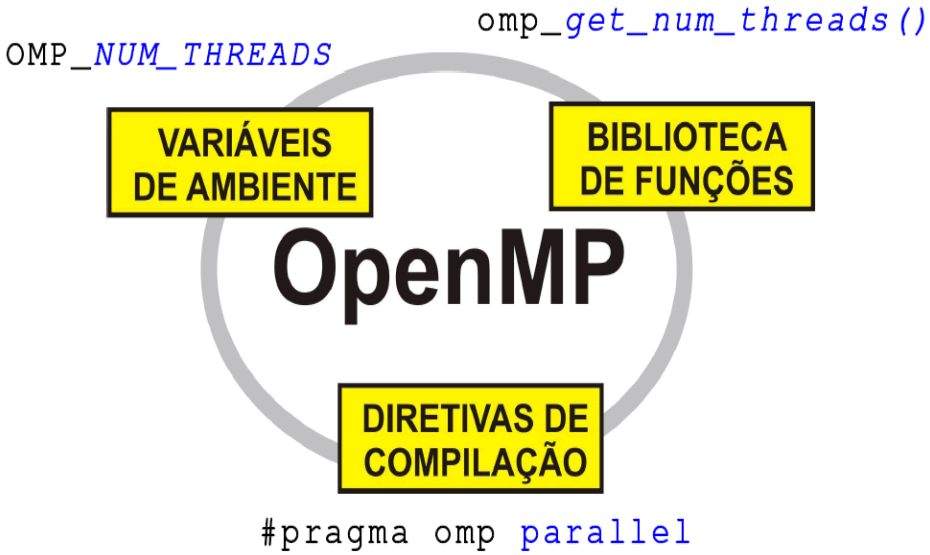

Figura 5: Componentes do OpenMP

Os programas desenvolvidos com ajuda de OpenMP utilizam um modelo de execução conhecido como Fork-Join que pode ser entendido da seguinte forma: todos os programas iniciam com a execução de uma thread principal ou master thread; a master thread executa de forma sequencial até encontrar um construtor paralelo, momento em que cria um grupo de threads; o código delimitado pelo construtor paralelo é executado pelo conjunto de threads; Ao concluírem a execução paralela o grupo de threads sincroniza numa barreira implícita com a master thread; o grupo de thread termina sua execução e a master thread continua sua execução sequencial. A Figura 6 exemplifica o modelo de execução Fork-Join.

Atualmente uma grande quantidade de compiladores implementam as especificações OpenMP, entre eles alguns dos mais populares como os compiladores da Intel e os do projeto GNU (a partir da versão 4.3.2).

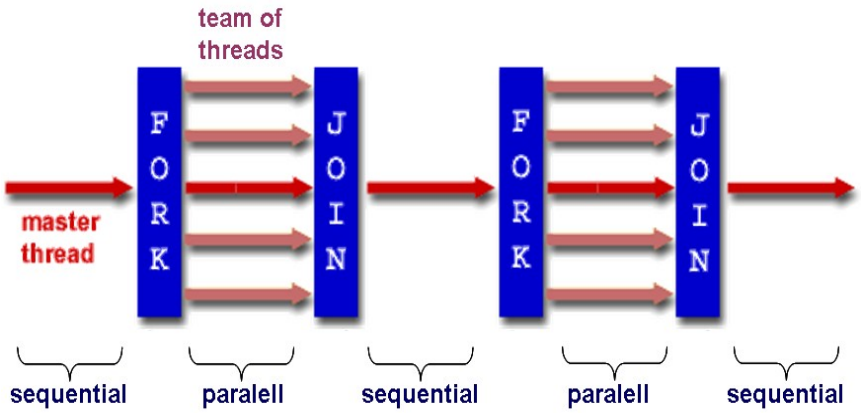

Figura 6: Modelo de execução Fork-Join

A programação multithread com OpenMP tem uma série de vantagens sobre outras formas de implementação de paralelismo em arquiteturas de memória compartilhada. Entre elas podemos citar a facilidade de conversão de programas sequenciais em paralelos, a maneira simples de se explorar o paralelismo, a facilidade de compreender o uso das diretivas entre outros.

As diretivas de compilação de OpenMP são linhas de código com um significado específico para o compilador. Estas linhas tem a seguinte sintaxe básica

```
#pragma omp diretiva [clausula, ...]
```

As diretivas se aplicam a blocos sintáticos simples, com uma única instrução, ou a blocos sintáticos compostos, delimitados por chaves. A Tabela 4 apresenta as principais diretivas e claúsulas utilizadas em OpenMP.

Um dos aspectos a serem considerados nas implementações paralelas em arquiteturas fortemente acopladas são os problemas relacionados a condições de corrida. Uma condição de corrida acontece quando duas ou mais *threads* atualizam simultaneamente uma variável compartilhada. Nestes casos o resultado final dependerá da ordem de execução das *threads*. Para garantir o resultado correto é necessário dispor de algum mecanismo para estabelecer a ordem como a variável compartilhada deve ser acessada. OpenMP define duas diretivas para tratar as possíveis condições de corrida: os construtores de sincronização `critical` e `atomic`.

Tabela 4. Diretivas de compilação de OpenMP

Diretivas	
Construtor paralelo	`#pragma omp parallel`

Construtores de divisão de trabalho	`#pragma omp for`
	`#pragma omp single`
	`#pragma omp sections`

Construtores de sincronização	`#pragma omp critical`
	`#pragma omp atomic`
	`#pragma omp barrier`
	`#pragma omp flush`
	`#pragma omp ordered`

Claúsulas: `shared, private, firstprivate, lastprivate, num_threads, schedule, default, ordered, copyprivate, if, nowait, reduction`	

As condições de corrida aparecem no tratamento de variáveis compartilhadas entre os *threads*. Entretanto, OpenMP permite que cada *thread* tenha um conjunto de variáveis privadas. De forma geral as variáveis declaradas fora da região paralela são compartilhadas por todas as threads. As variáveis declaradas dentro da região são

privadas e cada thread tem sua própria cópia da mesma. Variáveis de controle de laços paralelos e variáveis associadas a claúsulas `reduction` são privadas em cada *thread*. Este tipo de comportamento pode ser modificado através de claúsulas na declaração das diretivas de compilação. Declarar o comportamento de cada variável de forma explícita é considerado uma boa prática de programação.

Para se aprofundar na utilização de OpenMP para implementação de algoritmos paralelos em arquiteturas de memória comptilhada recomenda-se consultar referências complementares como (Chandra, 2001), (Chapman et al., 2007).

4.2. Troca de Mensagens com MPI

A implementação de programas paralelos em ambientes de memória distribuída requer um mecanismo para criação de processos que permita sua execução em máquinas diferentes e um mecanismo que viabilize a troca de mensagens entre os processos. O padrão mais utilizado para implementar troca de mensagens é MPI.

Da mesma forma que OpenMP, MPI não é uma linguagem. O padrão MPI define a sintaxe e a semântica de um conjunto de rotinas que devem ser implementadas numa biblioteca de funções para Fortran e C/C++. Este padrão surgiu como fruto da colaboração de um conjunto de pessoas e instituições representando a indústria, o meio acadêmico e importantes centros de pesquisa. Como resultado inicial desta colaboração surgiu o MPI Forum em 1992 e dois anos mais tarde o foi lançado o padrão MPI 1.0. Após um longo processo de discussão que levaram a melhoras significativas foi lançado, em 1997, o padrão 2.0 que recebeu uma atualização de menor porte (2.1) apenas em 2008.

O surgimento de implementações importantes e eficientes do padrão MPI viabilizou o desenvolvimento dos cluster tipo *Beowulf* que foram a base de muitos sistemas de alto desempenho em instituições de pequeno e médio porte. Atualmente existem inúmeras implementações do padrão MPI entre as quais destacam-se:

- OpenMPI, desenvolvida pelo OpenMPI Team, http://www.open-mpi.org
- MPICH, uma das implementações mais conhecidas e da qual se derivaram outras implementações importantes como a implementação da Intel, http://www.mcs.anl.gov/research/projects/mpich2/index.php

A programação utilizando MPI pode se resumir da seguinte forma: Todos os processadores executam o mesmo programa, entretanto cada programa executa um subconjunto específico de instruções com base numa estrutura de desvio que utiliza a identificação de cada processo chamada de `rank`. Este enfoque utilizado em sistemas MIMD é conhecido como SPMD (Single Program Multiple Data).

Outro problema importante com que tem que lidar os desenvolvedores do MPI são os tipos de comunicação. As duas principais funções de comunicação do padrão MPI são:

```
int MPI_Send(void *buffer, int count,
             MPI_Datatype datatype,
             int destination, int tag,
```

```
                    MPI_Comm comunicator);
int MPI_Recv(void *buffer, int count,
                    MPI_Datatype datatype,
                    int source, int tag,
                    MPI_Comm comunicator,
                    MPI_status * status);
```

Como foi colocado anteriormente, na busca por algoritmos paralelos mais eficientes procura-se diminuir ao máximo a comunicação entre os processos. Nesta tarefa desempenha um papel importante a localidade dos dados, ou seja o problema de atribuir um conjunto de dados a um determinado processo de forma que as operações de comunicação seja minimizadas. Na mesma linha tem que se prestar uma atenção especial ao balanceamento de carga, ou seja atribuir uma quantidade de trabalho equivalente a cada processo.

A programação paralela utilizando MPI é abordada em várias publicações na área entre as quais pode-se destacar (Pacheco, 1997), (Gropp, Lusk, & Skjellum, 1999),

4.3. Programação Paralela em GPU com CUDA

A programação paralela e, particularmente, a PAD ganharam mais uma linha importante de trabalho com o aperfeiçoamento das GPUs e das ferramentas para implementar processamento paralelo nelas. A evolução das GPUs e das suas interfaces de programação ocupariam mais tempo do que dispomos neste texto. Várias APIs já estão disponíveis no mercado, algumas delas pensadas para rodar em qualquer GPU. Destaca-se pelo avançado da sua implementação o OpenCL. Entretanto os recursos disponíveis nas GPUs da NVIDIA através da arquitetura CUDA são os que apresentam resultados mais consistentes na atualidade.

A partir dos dispositivos G80 da NVIDIA uma nova arquitetura foi desenhada para permitir computação paralela. O modelo de programação CUDA, introduzido pela NVIDIA em 2007, foi projetado para permitir a execução conjunta em CPU e GPU de um aplicativo.

CUDA tem muitos aspectos em comum com outros modelos utilizados para programação paralela, como MPI e OpenMP. Os programadores estão encarregados de construir o código paralelo através de um conjunto de extensões da linguagem C. Os conhecedores de OpenMP são unânimes em afirmar que os compiladores que utilizam esta ferramente tem um maior grau de automação na hora de criar código paralelo.

Durante o minicurso será apresentada uma implementação paralela do problema de multiplicação de matrizes utilizando cada uma das técnicas de programação paralela aqui descritos. O problema de multiplicação de matrizes é apresentado na próxima Seção. Para os interessados em se aprofundar em programação paralela com CUDA recomenda-se as seguintes referências (Kirk et al., 2010), (Sanders & Kandrot, 2010)

5. Problema de Multiplicação de Matrizes

Para exemplificar as características das diferentes técnicas de programação paralela aqui discutidas vamos utilizar um problema simples que permite um alto grau de paralelismo de dados. A multiplicação de duas matrizes implica que cada elemento da matriz resultante, ou matriz produto, **P** se forma a partir do produto escalar de uma linha da matiz de entada **M** e uma coluna da matiz de entrada **N**. Não é difícil verificar que o cálculo de cada elemento de **P** pode ser realizado de forma independente, ou seja, não depende do cálculo dos outros elementos.

Por outro lado, para matrizes de grande porte, o número de operações de ponto flutuante envolvidas é bastante elevado. Somente para exemplificar podemos verificar que o produto de duas matrizes de 1.000 ✕1.000 envolve 1.000.000 de produtos escalares, cada um com 1.000 operações aritméticas de multiplicação e 1.000 de soma.

O fragmento de código a seguir mostra uma função que implementa a multiplicação serial de matrizes. As matrizes neste exemplo são representadas por arrys unidimensionais ou vetores. Este código será utilizado como base para implementar versões paralelas utilizando multithreading com OpenMP, troca de mensagens com MPI, e paralelismo em dispositivos gráficos utilizando CUDA. Os exemplos serão compilados e testados com os recursos computacionais disponíveis no Núcleo de Biologia Computacional e Gestão de Informações Biotecnológicas (NBCGIB) da UESC.

```
void MatrixMultiplicationCPU(float* M, float* N, float*
                                        P, int Width){
  int i, j, k, linha;
  float sum, a, b;
  for(i = 0; i < Width; i++){
    linha = i * Width;
    for(j = 0; j < Width; j++){
      sum = 0;
      for(k = 0; k < Width; k++){
     a = M[linha + k];
     b = N[k * Width + j];
     sum += a*b;
      }
      P[linha + j] = sum;
    }
  }
}
```

6. Considerações Finais

A utilização de programação paralela está cada dia mais presente nas diferentes áreas da Ciências e as Engenharias. As arquiteturas multiprocessadas e as GPUs com capacidade de processamento paralelo são uma realidade que pode ser acessada por cada vez mais usuários. O minicurso introdutório desta ERAD se propõe criar as bases para que muitos dos seus participantes tenham um primeiro contato com o PAD. Estudos mais aprofundados podem ser realizados através de cursos específicos na área. A revisão bibliográfica aqui apresentada pode ser utilizada como referência para muitos desses cursos.

Bibliografia

Adhianto, L., & Chapman, B. (2007). Performance modeling of communication and computation in hybrid MPI and OpenMP applications. *Simulation Modelling Practice and Theory*, *15*(4), 481-491. doi:10.1016/j.simpat.2006.11.014

Alonso, P., Cortina, R., Martínez-Zaldívar, F. J., & Ranilla, J. (2009). Neville elimination on multi- and many-core systems: OpenMP, MPI and CUDA. *The Journal of Supercomputing*, 1–11. Springer. doi:10.1007/s11227-009-0360-z

Aversa, R., Dimartino, B., Rak, M., Venticinque, S., & Villano, U. (2005). Performance prediction through simulation of a hybrid MPI/OpenMP application. *Parallel Computing*, *31*(10-12), 1013-1033. doi:10.1016/j.parco.2005.03.009

Chandra, R. (2001). *Parallel programming in OpenMP*. Morgan Kaufmann Publishers.

Chapman, B., Jost, G., & Pas, R. (2007). *Using OpenMP: portable shared memory parallel programming*. MIT Press.

Gropp, W., Lusk, E., & Skjellum, A. (1999). *Using MPI*. MIT Press.

Gustafson, J. L. (1988). Reevaluating Amdahl's law. *Communications of the ACM*, *31*(5), 532-533. doi:10.1145/42411.42415

Hwu, W.-mei, Keutzer, K., & Mattson, T. G. (2008). The concurrency challenge. *Design & Test of Computers, IEEE*, *25*(4), 312–320. IEEE. doi:10.1109/MDT.2008.110

Kirk, D., Hwu, W. M. W., & Hwu, W. (2010). *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann Publishers.

Nakajima, K. (2005). Parallel iterative solvers for finite-element methods using an OpenMP/MPI hybrid programming model on the Earth Simulator. *Parallel Computing*, *31*(10-12), 1048-1065. doi:10.1016/j.parco.2005.03.011

Pacheco, P. S. (1997). *Parallel programming with MPI*. Morgan Kaufmann Publishers.

Parhami, B. (1999). *Introduction to parallel processing: algorithms and architectures*. Plenum Press.

Rauber, T., & Rünger, G. (2010). *Parallel Programming: For Multicore and Cluster Systems*. Springer.

Sanders, J., & Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley.

Sun, X.-H., & Chen, Y. (2010). Reevaluating Amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, *70*(2), 183-188. Elsevier Inc. doi:10.1016/j.jpdc.2009.05.002

Sutter, H., & Larus, J. (2005). Software and the concurrency revolution. *Queue*, *3*(7), 54. doi:10.1145/1095408.1095421

Wolf, F. (2003). Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, *49*(10-11), 421-439. doi:10.1016/S1383-7621(03)00102-4

Yang, C.-T., Huang, C.-L., & Lin, C.-F. (2011). Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters☆ *Computer Physics Communications*, *182*(1), 266-269. Elsevier B.V. doi:10.1016/j.cpc.2010.06.035