Capítulo

1

Introdução a Processamento de Alto Desempenho

Esbel T. Valero Orellana

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.

1.1. 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 colocaram esta importante ferramenta ao alcance dos 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 de uso geral (**GPGPU**), com uma grande quantidade de núcleos (*cores*) com capacidade de executar tarefas maciçamente paralelas. A quantidade de núcleos das **GPGPUs**, 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 [Hwu et al. 2008] e [Sutter and 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 **GPGPUs**. 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. Os cursos relacionados com HPC tiveram que see adequar ao novo contexto. Novas **APIs** chegaram ao mercado e os programadores tiverem que incorporar as mesmas no seu arenal de ferramentas computacionais. Hoje é frequente encontrar nos cursos de graduação, especializações e de pós-graduação, tópicos relacionados com **OpenMP**, **CUDA**, **OpenCL**, **OpenACC** se juntando aos já tradicionais cursos de **MPI**.

Neste contexto aconteceu, em 2011, a primeira Escola Regional de Alto Desempenho (**ERAD**) da regional Nordeste. A versão original deste texto foi utilizada como base para ministrar 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**). Dois anos depois algumas atualizações e correções se fazem necessárias.

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**. Finalmente na Seção 5 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 revisão bibliográfica 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.

1.2. 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 1.1 apresenta as unidades utilizadas para quantificar o desempenho destes supercomputadores.

Tabela 1.1. 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}
gigaflops	10^{15}
hexaflops	10^{18}

Olhando o gráfico de evolução de desempenho publicado no relatório mais recente do Top500, apresentado na Figura 1.1, 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

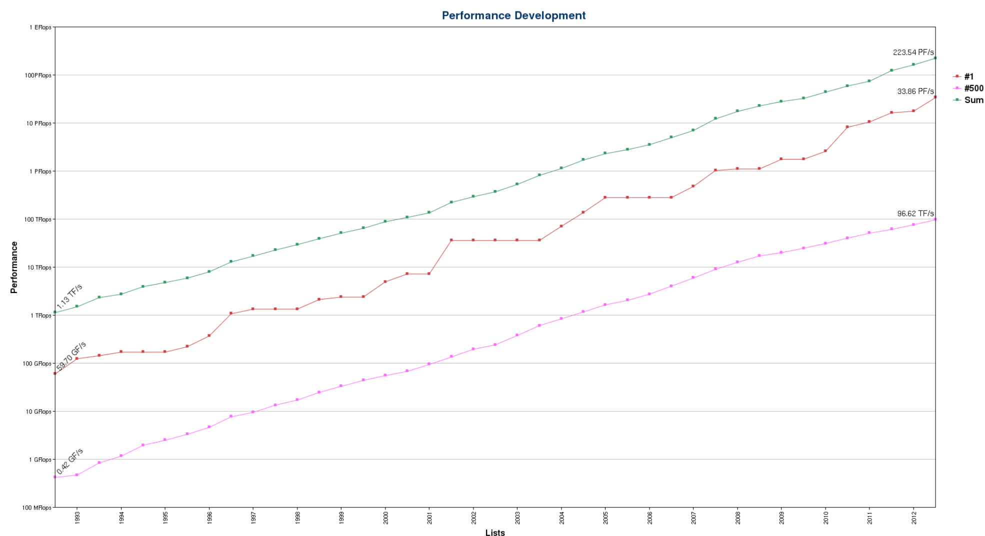

Figura 1.1. Evolução do desempenho dos 500 maiores computadores (Fonte: http://www.top500.org)

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, com os recursos computacionais disponíveis, em um intervalo razoável de tempo. Note que utilizamos ?tempo razoável?, termo bastante subjetivo e que precisa ser avaliado em cada situação de forma apropriada. Veja uma abordagem rápida a este tema em [Pacheco 1997].

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, de acordo com o período de tempo que esteja sendo simulado.

Colocando o exemplo anterior em números teremos que a galáxia tem aproximadamente 10^{11} estrelas. Se cada operação fosse feita em $1\mu\mathbf{s}$ 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 et al. 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. Veja uma abordagem mais aprofundada sobre o tema em [Parhami 1999].

Os supercomputadores pessoais 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. Discutiremos este tema melhor mas adiante, após termos introduzido as métricas de desempenho. Entretanto, se deseja adiantar a discussão recomenda-se a leitura de [Gustafson 1988].

Outros pontos de vista sobre aspectos relevantes do **Pad** podem ser acompanhados em textos específicos da área como [Chapman et al. 2007] ou [Rauber and Rünger 2010]. Para entender melhor como funcionam os supercomputadores modernos será apresentado, na próxima seção, alguns aspectos relevantes da arquitetura de computadores paralelos.

1.3. Aspectos Importantes das Arquiteturas com Múltiplos Processadores

Para entender melhor o senário atual do **PDA** devemos começar por analisar as diferentes arquiteturas paralelas. Inicialmente podemos apresentar uma forma de classificar estas arquiteturas. A taxonomia de Flynn pode ser utilizada para obter uma classificação amplamente aceita. 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 1.2.

Tabela 1.2. Classificação segundo a taxonomia de Flynn

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

A categoria **SISD** engloba aqueles computadores com um único fluxo de instruções atuando sobre um um único fluxo de dados. Trata-se de sistemas sem nenhu nível

de paralelismo. Os computadores pessoais, equipados come chips de um único núcleo, podem ser classificados como sistemas **SISD**. O principal gargalo nesta arquitetura é a transferência de dados entre a memória principal e a **CPU**. Para melhorar este fluxo de dados foi introduzido uma estrutura hierárquica de memória. Com base em computadores pessoais deste tipo e em equipamentos simples encontrados comumente nas lojas de informáticas forma construídos um tipo particular de máquina paralela: os clusters tipo *Beowulf*.

Os clusters tipo *Beowulf* é uma aglomerado de computadores interligados por uma rede utilizado para processamento paralelo. O baixo custo de implementação deste tipo de solução, junto com sua escalabilidade e disponibilidade de software para implementação dos mesmos popularizaram dos Beowulf desde o início dos anos 90. Esta estrutura de multi-computadores pode ser caracterizada como uma arquitetura **MIMU**. Se quer saber um pouco mais sobre como montar um cluster recomendamos uma leitura em [Sterling 2002]

As arquiteturas paralelas modernas podem ser caracterizadas como **SIMD** ou como **MIMD**. No primeiro caso temos um único fluxo de instruções atuando sobre múltiplos fluxos de dados. As modernas **GPGPU**s são um exemplo típico deste tipo de sistemas. Discutiremos u pouco mais os detalhes desta tecnologia mais adiante quando apresentemos a arquitetura **CUDA**. Mais detalhes sobre a taxonomia de Flynn aplicada a arquiteturas paralelas podem ser encontrada em [Parhami 1999].

As máquinas com arquitetura **MIMD**, que chamaremos daqui para frente como máquinas multiprocessadas, podem ser classificadas, por sua vez, em sistemas de mória compartilhada ou sistemas de memória distribuída.

As máquinas de memória compartilhada, ou fortemente acoplado, são constituídas por um conjunto de processadores que compartilham o espaço de memoria, formado por um conjunto de blocos ou módulos interligados a um bus de dados. As novas gerações de processadores *multicore* implementam uma arquitetura de memória compartilhada. A Figura 1.2 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**. Note que cada um dos dois chips é formado porte quatro núcleos de cálculo ou cores que compartilham uma estrutura hierárquica de memória que optimiza o acesso ao espaço de memória através do *bus* de dados.

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, o que normalmente é conhecido como **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 ... será apresentado o paradigma de programação *multithread* baseado em **OpenMP**, que permite implementar programação paralela em arquiteturas de **MIMD** de memória com-

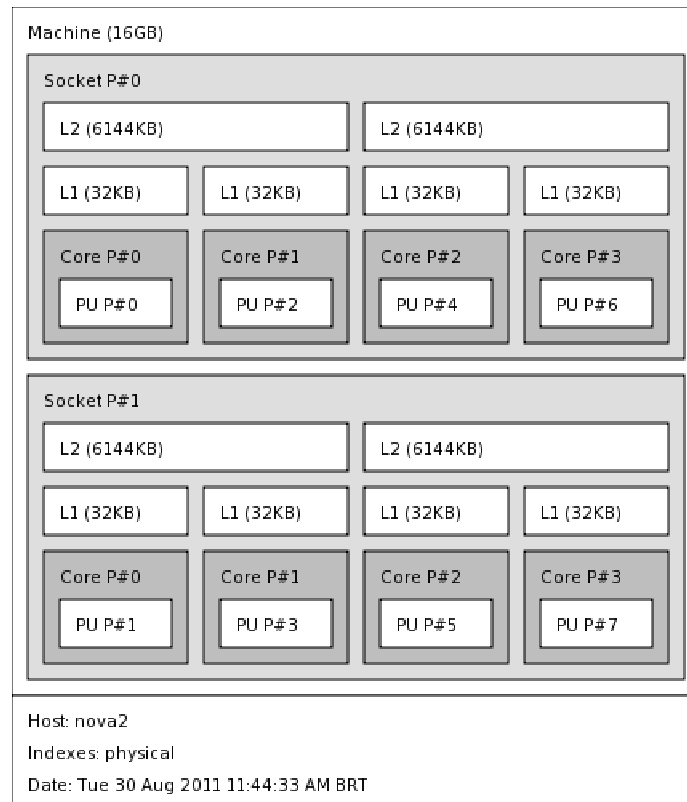

Figura 1.2. Informação sobre estrutura de um sistema com dois processadores quadcore.

partilhada.

As máquinas com memória distribuída, ou sistema fracamente acoplados, são formadas por processadores que tem espaço próprio de memória ao qual 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 ... 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 moderno deve ser encarado como uma arquitetura híbrida, formada por um conjunto de máquinas multiprocessadas, de memória compartilhada, interconectadas por uma rede formando uma 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). Veja alguns trabalhos relacionados com esta linha de pesquisa em [Adhianto and Chapman 2007], [Wolf 2003], [Nakajima 2005], [Aversa et al. 2005].

Neste ponto os cursos mais atuais na área de **PAD** precisam fazer um aparte para comentar a arquitetura das modernos dispositivos **GPGPU**. A Figura 1.3 mostra um esquema que compara a arquitetura de uma **CPU** *multicore* com uma **GPGPU** *many-core*. Uma estrutura formada por uma grande quantidade de núcleos e uma hierarquia própria de memória de alta velocidade, como mostra a Figura 1.4, garantem um hardware com capacidade para executar tarefas paralelas de grande porte. Na Seção ... se apresentam

alguns aspectos da arquitetura **CUDA** para programação paralela baseada em dispositivos **GPGPU** do fabricante **NVIDIA**.

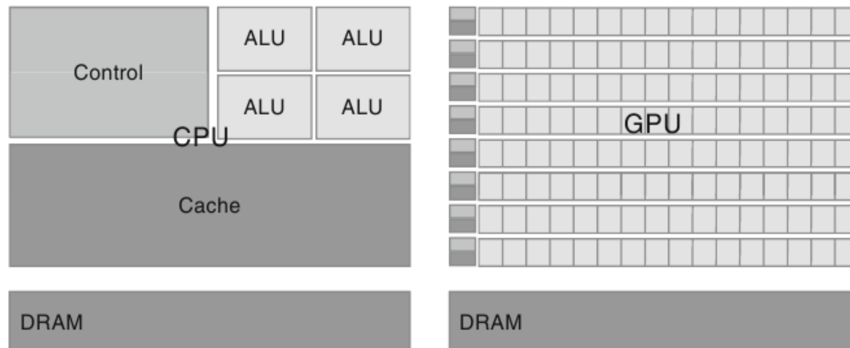

Figura 1.3. Arquiteturas de uma CPU *multicore* e de uma GPU *many-core*. Tomado de [Kirk et al. 2010]

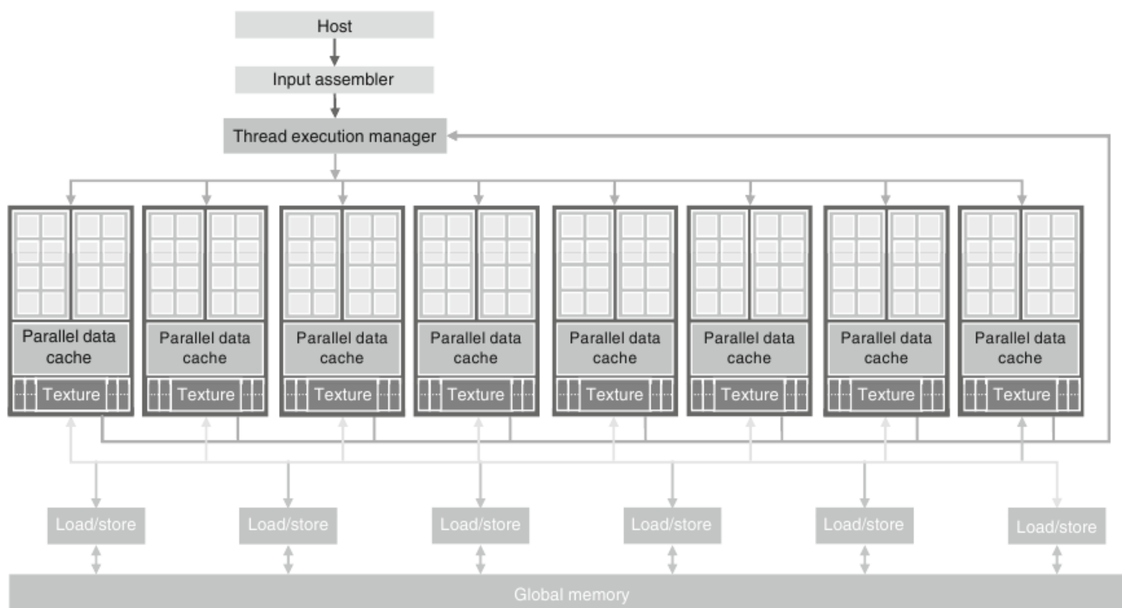

Figura 1.4. Esquema mais detalhado da arquitetura de uma GPU. Tomado de [Kirk et al. 2010]

Mais detalhes sobre as arquiteturas paralelas, especialmente sobre as novas arquiteturas implementadas nas modernas **GPGPU**s, podem ser consultados em [Sanders and Kandrot 2010], [Kirk et al. 2010] e [Chapman et al. 2007]. Outros artigos relacionados com aplicações para arquiteturas híbridas envolvendo **GPGPU**s [Yang et al. 2011], [Alonso et al. 2009].

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

1.4. 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. Os cursos específicos de programação paralela discutem diversas métricas que permitem avaliar o desempenho de aplicações paralelas. Por falta de espaço e tempo neste curso 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 $T_s(n)$, que depende apenas do tamanho n do problema, e o tempo de execução do algoritmo paralelo $T_p(n,p)$, que depende do tamanho do problema e da quantidade p de processadores utilizados. De forma geral o **speedup** é um valor entre 0 a quantidade de processadores. Na Tabela 1.3 são apresentados diferentes situações que podem aparecer ao determinar o **speedup** de um algoritmo paralelo.

Tabela 1.3. Caracterização de desempenho com base no speedup

speedup$S(n,p) = \frac{T_s(n)}{T_p(p,n)}$	**Caracterização**
$S(n,p) < 1$	**Slowdown**, situação indesejável
$1 < S(n,p) < p$	**Sublinear**, comportamento geral
$S(n,p) = p$	**Linear**, Ideal, não existe sobrecarga
$S(n,p) > p$	**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), a minimização da necessidade de comunicação entre os processos e do tempo ocioso dos mesmos, a sobreposição, na medida do possível, das operações de comunicação e computação e a concentração as operações de E/S visando minimizar seu efeito.

Não podem ser apresentados 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, a mesma 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 o senário atual do **PDA**, em relação à lei de Amdahl, seja através do trabalho de [Gustafson 1988], onde se abora, pela primeira vez, as potencialidades ainda inexploradas da utilização de paralelismo com grande quantidade de processadores. Outra discussão mais atualizada sobre o tema pode ser encontrada em [Sun and 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 **GPGPU** são um exemplo de uso intensivo 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.

1.5. Técnicas de Processamento Paralelo

Na atualidade existem diversas abordagens que podem ser utilizadas para se implementar programas paralelos. Cada uma delas depende, essencialmente, da arquitetura computacional para a qual está-se programando. O grande problema nestas abordagens hoje é a portabilidade dos códigos que, em muitos casos fica limitada por soluções específicas para determinado tipo de hardware. Presentaremos a seguir três paradigmas de programação paralela. No final voltaremos a discutir o tema de portabilidade.

1.5.1. 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* na atualidade podem ser considerados computadores multiprocessados de memória compartilhada. Uma das formas de se implementar processamento paralelo neste tipo arquitetura é através de programação *multithread*.

Uma *thread*, ou linha ou fio de execução, não é mais que a menor parte de um processo que pode ser manipulado pelo escalonador do sistema operacional. As *threads* de um processo se originam da divisão da *thread* principal em dois ou mais *threads*. Elas podem ser executadas em paralelo, são atribuídos pelo escalonador, quando possível, a processadores diferentes e 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 em julho de 2011 foi disponibilizada a versão 3.1, que está implementada

nas versões mais recentes do compilador GNU/GCC. No momento que este texto estava sendo preparado os desenvolvedores de compiladores trabalhavam na implementação do padrão 4.0 que foi liberado em Agosto de 2013.

A Figura 1.5 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.

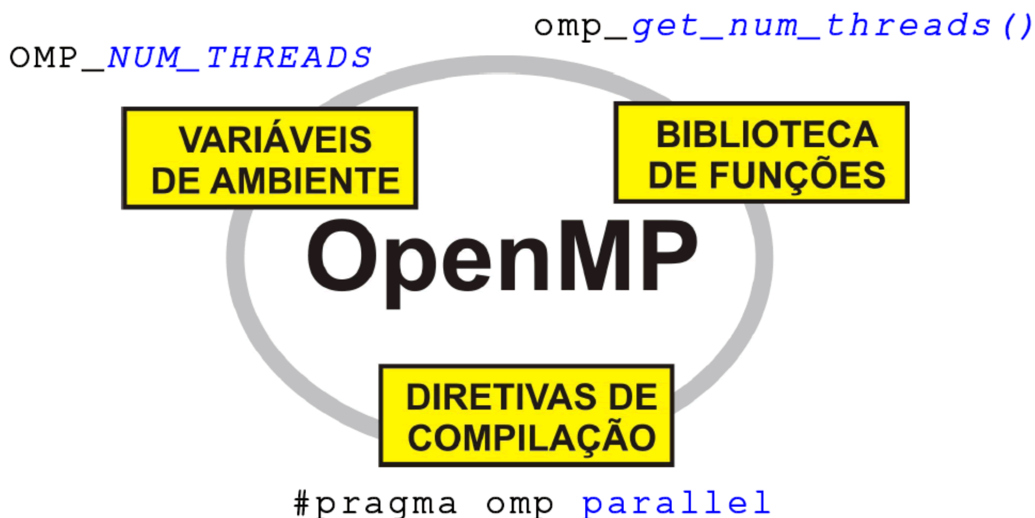

Figura 1.5. Componentes do OpenMP.

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 1.6 exemplifica o modelo de execução *Fork-Join*.

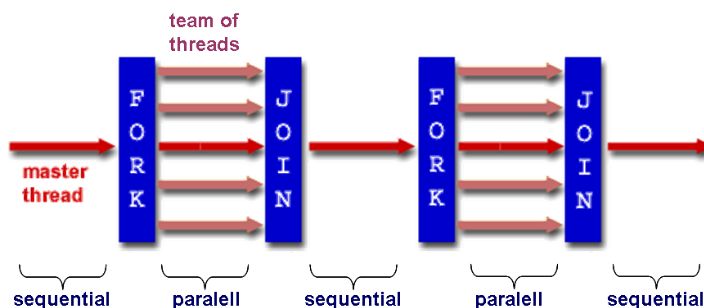

Figura 1.6. Modelo de execução *Fork-Join*.

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/GCC (a partir da versão 4.3.2).

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 1.4 apresenta as principais diretivas e cláusulas utilizadas em **OpenMP**.

Tabela 1.4. Diretivas e cláusulas de compilação de OpenMP

Tipo	Diretiva
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
Cláusulas: shared, private, firstprivate, lastprivate, num_threads, schedule, default, ordered, copyprivate, if, nowait, reduction	

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.

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 cllausulas reduction são privadas em cada *thread*. Este tipo de

comportamento pode ser modificado através de cláusulas na declaração das diretivas de compilação. Declarar o comportamento de cada variável de forma explícita é considerado uma boa prática de programação.

Neste ponto recomenda-se alguns textos adicionais para se aprofundar na utilização de **OpenMP**, visando a implementação eficiente de algoritmos paralelos em arquiteturas de memória compartilhada. Entre a bibliografía recomendada destaca-se [Chandra 2001] e [Chapman et al. 2007].

1.5.2. 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 o de **Message Passing Interface** (**MPI**).

Da mesma forma que **OpenMP**, **MPI** não é uma linguagem de programação. 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/projec

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 mecanismos de comunicação. O padrão MPI implemente um conjunto bastante amplo de mecanismos de comunicação entre os quais destacam-se funções que implementam comunicação ponto a ponto, mecanismos de comunicação bloqueada e não bloqueada e mecanismos de comunicação coletiva.

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 um tema complexo aque requer um estudo mais minucioso. Neste ponto recomenda-se a utilização de algum textos que abordam o tema, entre os quais destacam-se [Pacheco 1997] e [Gropp et al. 1999].

1.5.3. Programação Paralela em GPU com CUDA

A programação paralela e, particularmente, a **PAD** ganharam mais uma linha importante de trabalho com o aperfeiçoamento das GPGPUs e das ferramentas para implementar processamento paralelo nelas. A evolução das GPGPUs 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 GPGPU. Destaca-se pelo avançado da sua implementação o **OpenCL**. Entretanto os recursos disponíveis nas **GPGPUs** 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/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. O sucesso do modelo de paralelização implementado na arquitetura CUDA foi tanto que modelos posteriores, como o OpenCL, seguiram a mesma estratégia com excelentes resultados.

Entretanto a curva de aprendizagem destes modelos é um pouco mais lenta, devido ao conceito, totalmente novo, introduzido a partir da aplicação desta nova tecnologia. 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 método de introduzir os conceitos através de exemplos busca facilitar um primeiro contato dos programadores com estas técnicas de programação.

Para os interessados em se aprofundar em programação paralela com CUDA recomenda-se as seguintes consultar outras referências como [Kirk et al. 2010] ou [Sanders and Kandrot 2010].

1.6. 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 **GPGPU**s com capacidade de processamento paralelo são uma realidade que pode ser acessada por cada vez

mais usuários. Este texto introdutório se propõe criar as bases para que muitos dos seus leitores 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.

Referências

[Adhianto and Chapman 2007] Adhianto, L. and Chapman, B. (2007). Performance modeling of communication and computation in hybrid MPI and OpenMP applications. *Simulation Modelling Practice and Theory*, 15(4):481–491.

[Alonso et al. 2009] Alonso, P., Cortina, R., Martínez-Zaldívar, F. J., and Ranilla, J. (2009). Neville elimination on multi- and many-core systems: OpenMP, MPI and CUDA. *The Journal of Supercomputing*, pages 1–11.

[Aversa et al. 2005] Aversa, R., Dimartino, B., Rak, M., Venticinque, S., and Villano, U. (2005). Performance prediction through simulation of a hybrid MPI/OpenMP application. *Parallel Computing*, 31(10-12):1013–1033.

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

[Chapman et al. 2007] Chapman, B., Jost, G., and Pas, R. (2007). *Using OpenMP: portable shared memory parallel programming*. Number v. 10 in Scientific and Engineering Computation. MIT Press.

[Gropp et al. 1999] Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI*. Number v. 2 in Scientific and engineering computation. MIT Press.

[Gustafson 1988] Gustafson, J. L. (1988). Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533.

[Hwu et al. 2008] Hwu, W.-m., Keutzer, K., and Mattson, T. (2008). The concurrency challenge. *Design & Test of Computers, IEEE*, 25(4):312–320.

[Kirk et al. 2010] Kirk, D., Hwu, W. M. W., and Hwu, W. (2010). *Programming massively parallel processors: a hands-on approach*. Applications of GPU Computing Series. Morgan Kaufmann Publishers.

[Nakajima 2005] 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.

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

[Parhami 1999] Parhami, B. (1999). *Introduction to parallel processing: algorithms and architectures*. Plenum series in computer science. Plenum Press.

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

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

[Sterling 2002] Sterling, T. (2002). *Beowulf Cluster Computing with Linux*. Scientific and Computational Engineering Series. MIT Press.

[Sun and Chen 2010] Sun, X.-H. and Chen, Y. (2010). Reevaluating Amdahlâs law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188.

[Sutter and Larus 2005] Sutter, H. and Larus, J. (2005). Software and the concurrency revolution. *Queue*, 3(7):54.

[Wolf 2003] Wolf, F. (2003). Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10-11):421–439.

[Yang et al. 2011] Yang, C.-T., Huang, C.-L., and Lin, C.-F. (2011). Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clustersâ. *Computer Physics Communications*, 182(1):266–269.