CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA
DE MINAS GERAIS

ENGENHARIA DE COMPUTAÇÃO

Paralelização de Métodos Numéricos para Resolver Equações Diferenciais Parciais

Orientando:
Marcelo Lopes de Macedo
FERREIRA CÂNDIDO

Orientador:
Prof. Dr. Luis Alberto
D'AFONSECA

BELO HORIZONTE
4 de junho de 2019

Sumário

Capítulo 1

Introdução

Capítulo 2

Aquisições Sísmicas

introduzir

2.1 O Que São Aquisições Sísmicas e Como Modelá-las

No ramo da mineração não se pode tentar a esmo a descoberta de recursos minerais no subterrâneo de um local em que se já se suspeita sua existência. É necessário que, de alguma forma, se obtenha a forma dessa estrutura oculta para se saber os pontos onde se encontram as jazidas/poços desse recurso. A obtenção dos dados de como é essa estrutura se chama **aquisição sísmica**.

A forma de se realizar essa aquisição pode variar com o ambiente e os métodos adotados para coleta de dados e processamento dos mesmos. Os recursos minerais desejados podem se encontrar tanto em meios terrestres e/ou subaquáticos. Contudo, o meio em que a aquisição será realizada pouco importa nesse trabalho, o que será melhor explicado mais a frente.

Para a coleta dos dados a serem processados podemos citar dois exemplos de aquisição

buscar referências

1. **marítima**: um navio equipado com um canhão sonoro emite ondas sonoras cujas reflexões e refrações nas camadas terrestres submarinas são captadas por filas de hidrofones puxadas pelo mesmo navio.

2. **terrestre**: um explosivo é colocado (preferencialmente enterrado) em um terreno. Sua explosão gera uma onda sonora cujas reflexões são captadas por geofones distribuídos relativamente próximos, na superfície.

Costuma-se alterar a posição da fonte sonora na realização da aquisição para facilitar a modelagem do meio de propagação das ondas.

conferir se procede

Esse colhimento dos dados consistirá em traços, que são o gráfico das amplitudes das ondas sonoras captadas pelos hidrofones/geofones,como se pode ver na Figura **??**. A partir desses traços, detecta-se, por análise técnica, onde se encontram as interfaces entre as camadas do domínio analisado e do que são feitas essas. Nisso consiste o processamento dos dados colhidos.

Colocar a imagem dos traços

Vale lembrar que, quanto aos métodos de processamento utilizados, nesse trabalho simularemos um problema direto, ou seja, estipulando o meio (nesse caso, não-homogêneo) da aquisição, veremos como as ondas se propagam nele. Para tal, usaremos um método matemático específico. Nesse trabalho, como já foi dito no Capítulo 1, é o de Diferenças Finitas.

colocar isso

2.2 A Equação da Onda

Para que seja possível avaliar matematicamente um fenômeno ondulatório produzido por uma fonte em um domínio é necessário se ter uma fórmula matemática para o que se entende por onda. Tal fórmula, que nos servirá durante todo esse trabalho, principalmente na parte do código é a **equação da onda**, dada por

$$\frac{\partial^2 u}{\partial t^2} = \frac{1}{v^2}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + f(x,y,t) \tag{2.1}$$

onde x e y são variáveis espaciais e t, temporal. A constante (no caso desse trabalho) v representa a velocidade da frente de onda. Trata-se de uma **equação diferencial parcial hiperbólica** com solução analítica, mas que pode ser resolvida de forma fácil numericamente, utilizando, por exemplo, o **método de diferenças finitas**, a ser mais explicado na próxima seção.

Caso o leitor se interesse por estudar ou revisar mais sobre Ondulatória, pode conferir em Cândido [MD18].

2.3 O Método de Diferenças Finitas (MDF)

Uma equação diferencial parcial, que geralmente é considerada em um domínio contínuo, pode ser discretizada. Isso é feito para que a equação possa ser representada e resolvida computacionalmente.

No caso do método de diferenças finitas para esse trabalho, basta transcrever cada termo da equação para o equivalente na fórmula de diferenças finitas para derivações de segundo grau, sobre a qual podemos ver um exemplo na Tabela 2.1 para o caso da parte espacial em x da equação.

Tabela 2.1: Fórmula de Diferenças Finitas para cálculo de derivada de segunda ordem, aqui representando a derivada na dimensão espacial x

$\frac{\partial^2 u}{\partial x^2}$	$\frac{u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}}{\Delta x^2}$

Para entender o que significa o índice i visto na tabela acima é bom se seguir uma analogia: suponhamos um *array* de tamanho maior que três. A posição i seria qualquer posição intermediária, $i-1$ a antecessora e a $i+1$ sucessora. Tal estrutura é chamada de *stencil*. O mesmo vale para os índices j e k. Podemos ver uma alegoria dessa analogia na Figura **??**.

Inserir uma imagem para o stencil 1D

No caso desse trabalho, para a simulação da propagação de ondas em um meio bidimensional ao longo do tempo, teremos que usar três *stencils* (os dois restantes podem ser vistos na Tabela 2.2), um para cada dimensão espacial ou temporal. Para tal, podemos utilizar outra analogia: um *array* tridimensional, onde cada plano de posições no espaço é um instante no tempo. Tal analogia é representada na Figura **??**.

Tabela 2.2: Demais fórmulas de Diferenças Finitas para as dimensões espacial y e temporal t

$\frac{\partial^2 u}{\partial y^2}$	$\frac{u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}}{\Delta y^2}$
$\frac{\partial^2 u}{\partial t^2}$	$\frac{u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1}}{\Delta t^2}$

Contudo, até então, não falamos sobre o método de Diferenças Finitas em si. Trata-se de uma sequência de iterações que marcham em função de alguma variável. No caso desse trabalho, o avanço se dá no tempo. Essa marcha no tempo quer dizer que o valor para um ponto no espaço no próximo instante de tempo será calculado com base nos valores para pontos no espaço em instantes anteriores. Vamos ser explícitos. Temos que a equação 2.1 traduzida nas fórmulas vistas nas tabelas 2.1 e 2.2 se dá por

$$\frac{u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1}}{\Delta t^2} = \frac{1}{v^2}\left(\frac{u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}}{\Delta x^2} + \frac{u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}}{\Delta y^2}\right) + f(x,y,t) \tag{2.2}$$

onde u_{k+1} é o ponto com valor a ser calculado para o próximo instante. Logo, ele precisa ser isolado, o que é mostrado na equação 2.3

$$u_{i,j,k+1} = \frac{\Delta t^2}{v^2}\left(\frac{u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}}{\Delta x^2} + \frac{u_{i,j-1k} - 2u_{i,j,k} + u_{i,j+1,k}}{\Delta y^2}\right) + \Delta t^2 f(x,y,t) - u_{i,j,k-1} + 2u_{i,j,k} \tag{2.3}$$

buscar referências sobre o método na PIC01

Capítulo 3

A arquitetura computacional atual e a necessidade de paralelização

Para compreender a questão da paralelização envolvida nesse trabalho é necessário entender de onde e porque ela veio. Para tal, é necessário se apresentar as principais peças que constituem um computador moderno, os problemas que surgiram no processo de evolução da *arquitetura computacional* atual e como isso culminou no paralelismo [Bar18].

conferir se deve tratar como atual

Antes de iniciar esse processo, é também necessário se explicar o que se entende por arquitetura computacional. Trata-se da área do conhecimento que estuda a interface entre *software* e *hardware*, desde o mais baixo nível, no qual o processador manipula as informações (instruções e dados) entregues a ele, para toda operação realizada no computador; passando pelas políticas de manipulação de dados nas memórias cache (e seus níveis), de acesso aleatório (RAM - *random access memory*) e de armazenamento não-volátil (discos rígidos, por exemplo); chegando também à interação dos computadores com os demais periféricos que por ventura estão nele conectados, realizando *inputs* (entradas) e/ou *outputs* (saídas), também conhecidas pela abreviação "I/O", como teclado, *mouse*, monitor, etc [Cat09].

Ainda no âmbito da arquitetura de computadores, são estabelecidas análises e metas de performance. Por exemplo, tenta-se determinar se um processador A é mais rápido que um B (sendo B diferente de A) comparando-se o número de instruções que cada um processa, ou quantos ciclos (que serão explicados mais a frente) podem ser dados em um segundo [Wik19b]. Questão semelhante encontra-se na computação de alta performance: quão mais rápido um trecho de código executa ao ser paralelizado.

Essa organização focada em um processador, memórias cache, RAM e de armazenamento não-volátil vieram da **arquitetura de von Neumann**, que possui esse nome por conta de seu idealizador, John von Neumann.

3.1 A arquitetura de von Neumann

- von Neumann e o conceito de programa armazenado;
- a forma da arquitetura (componentes e ligações);
- o contraste da arquitetura idealizada com as anteriores.

3.1.1 O gargalo de von Neumann

- o gargalo existente entre o processador e a memória, que não consegue acompanhá-lo.

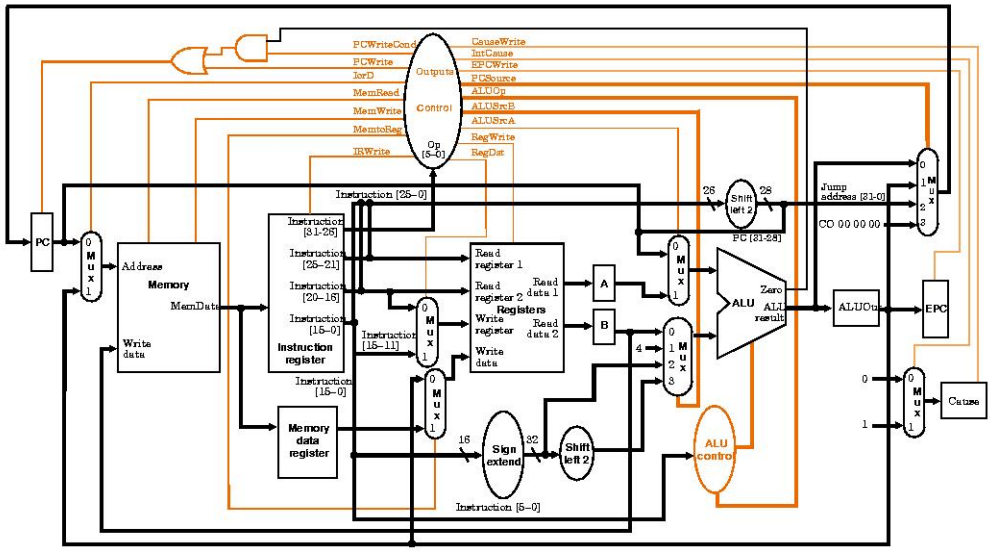
3.2 O processador

Um processador consiste em um módulo capaz de manipular instruções de máquina armazenadas em memória e produzir os resultados desejados através dessas instruções. Tais resultados podem ser de cunho ou lógico-aritmético ou manipulação de dados, no geral. Tal módulo é indispensável para o conceito de computadores como conhecemos hoje, de tal forma que, se não fosse pela necessidade de memória para a armazenagem de dados, um processador poderia ser a definição de um computador.

Essa unidade central de processamento (CPU, no inglês), como também são conhecidos os computadores são compostos por módulos menores responsáveis por interpretar as instruções por partes. A existência desses módulos define a complexidade da arquitetura de um processador.

3.2.1 Um exemplo de arquitetura: MIPS

A arquitetura MIPS (Microprocessador sem Estágios de Pipeline Intertravados, em inglês) surgiu em 1934, criada pela companhia americana MIPS *Computer Systems*, hoje chamada MIPS *Technologies*. É uma arquitetura utilizada em cursos de arquitetura de computadores como exemplo para se entender o funcionamento de instruções. Um exemplo da organização interna de um processador baseado na arquitetura MIPS pode ser visto na Figura **??** [Wik19c].



- apresentação de um processador MIPS simples;
  - entradas;
  - módulos internos;
  - saídas;

3.3 A memória principal

- introdução do que é uma memória;

- apresentação básica dos tipos de memória (enfase na RAM);
- apresentação básica das propriedades de uma memória RAM.

3.4 A cache

- O princípio da localidade no tempo e no espaço;
- A necessidade da existência de uma cache, visto o princípio da localidade;
- Explicação do que é uma cache;

3.5 Disco rígido

- O que são memórias voláteis e exemplos delas;
- A necessidade de memórias não-voláteis e o HD;
- Outras diferenças entre o HD e as outras memórias envolvidas no trabalho

3.6 Conseguir mais em menos tempo

A computação sempre foi uma das principais ferramentas humanas para avanços tecnológicos nos últimos séculos. Seja na engenharia, medicina, área militar, biologia, química, sísmica e até no cinema, os computadores tem sido utilizados em tarefas como mecânicas, estudos patológicos, ataques/defesas nacionais, enovelamento de proteínas, dinâmica molecular, monitoramento sísmico e animações tridimensionais.

Todas essas áreas costumam requerer que os resultados obtidos computacionalmente sejam gerados o mais rápido possível. Além disso, muitas (senão quase todas) as simulações numéricas envolvidas não podem ser executadas em tempo hábil com o tipo de processador visto na Seção 3.2. Visto isso, os projetistas por trás dos projetos de processadores precisaram implementar mecanismos nos modelos para explorá-los ao máximo. Nessa tarefa, encontraram barreiras [**autor:intro-par-comp**].

Ir à biblioteca

3.6.1 A barreira do paralelismo a nível de instruções - *ILP wall*

Vimos na Seção 3.2 um processador com a capacidade de executar uma instrução por ciclo de *clock*. Com isso, a duração do ciclo devia ser a mesma da execução da instrução mais demorada. Na necessidade de se adquirir mais velocidade de processamento, os projetistas perceberam que podiam particionar as instruções, de modo a executar cada estágio resultante em um ciclo de *clock*, deixando o resultado para a próxima partição a ser operada. Assim nasceu o processador **multiciclo**. Dessa forma, a duração do ciclo de *clock* reduziu para a mesma do estágio mais demorado dentre as instruções.

Contudo, a necessidade de se acelerar os processadores continuava. Em seguida, os projetistas perceberam que as unidades funcionais dos processadores ficavam ociosas quando não se tratava do estágio de uma instrução em que elas eram utilizadas. Era então possível executar uma instrução ao mesmo tempo que outra, desde que ambas se encontrassem, cada uma, em estágios A e B, sendo A o estágio da instrução mais antiga e B o da mais nova, com $B = A - 1$.

Ou seja, por exemplo, considere uma instrução *i* liberada no tempo de *clock* 1, passando por seu primeiro estágio. No tempo 2, ela estará em sua segunda etapa e as unidades funcionais responsáveis pela primeira estariam ociosas, se não fosse pela inovação apresentada acima. Com esta, uma nova instrução *j* é buscada ainda nesse tempo, tendo seu primeiro estágio executado. No tempo de ciclo de *clock* 3, a instrução *i* passará para o seu terceiro estágio, enquanto a *j* passará para o segundo e uma nova instrução poderá ser buscada. Ao "trem" de estágios foi dado o nome de ***pipeline***. À essa inovação, foi dado o nome de **paralelismo a nível de instruções**, visto que se tem mais de uma instrução sendo executada ao mesmo tempo e uma pronta a cada ciclo de *clock*.

Em seguida, os projetistas decidiram construir estágios menores, consequentemente aumentando a frequência de *clock* (que é $\frac{1}{\text{tempo de ciclo do } clock}$ e dada em hertz [Hz]), o tamanho do *pipeline* e o número de instruções operadas simultaneamente. Nasceu o **superpipeline**.

Apesar disso aumentar a performance dos processadores, ao se incrementar o número de estágios para um valor maior que oito tem-se o desempenho degradado.

buscar fonte e confirmação

Por fim, os projetistas de *hardware* ainda deram mais um passo em busca de mais performance: a **superescalaridade**, que consiste, basicamente em *pipelines* em paralelo, o que propicia a execução de múltiplas instruções ao mesmo tempo. Contudo, como é costumeiro, essa arquitetura também apresentou seus defeitos. Os *pipelines* em paralelo leva a problemas para acessar os recursos comuns, como memória, por exemplo, sendo necessário duplicá-los. Além disso, a ocorrência de dependências de Dados e dependências de Controle, fazem com que em alguns ciclos não hajam instruções para serem executadas, visto o atraso necessário para que as dependências sejam resolvidas [Sil09].

3.6.2 A barreira no gasto de energia dos processadores - *Power wall*

Em 1965, Gordon Moore publicou um artigo em que descrevia uma observação de que o número de componentes por circuitos integrados, transistores no caso, dobrava a cada dois anos. Ele então estimou que essa taxa de crescimento deveria continuar por pelo menos uma década. Essa estimativa foi batizada por **Lei de Moore** e o período da dita taxa veio então a ser alterada depois para um ano e meio e a estimativa se manteve firme por muitas décadas [Wik19d].

O crescimento da densidade desses componentes permitiu aos processadores alcançarem uma taxa muito mais alta de *clock*, passando de milhões de ciclos por segundo a bilhões [Hen12]. Tal evolução permitiu a execução de mais instruções de *hardware* em menos tempo, diminuindo o tempo de execução das aplicações em geral. Contudo, essa evolução proporcionada pela Lei de Moore não durará para sempre.

A alta taxa de *clock* acaba levando os processadores a gastarem muita energia. Esse gasto leva a um aumento na dissipação de calor pelo dispositivo, fenômeno conhecido por efeito joule. Atingirem altas temperaturas, que consequentemente leva ao *transistor leakage* [Liu+00] (vazamento nos transistores, tradução nossa), que é o gasto energético nos transistores. Logo, tem-se um ciclo.

3.6.3 A barreira da memória - *Memory wall*

Conferir se existe mesmo a memory wall

Revisar o que é a memory wall e completar o itemize abaixo

-

3.7 Paralelismo - a alternativa para se contornar as barreiras

Visto a incapacidade de se transpor o alto gasto energético das unidades de processamento, junto com as consequentes altas temperaturas, além da incapacidade de se aumentar o número de instruções prontas por ciclo de *clock*, precisamos de uma alternativa para se continuar aumentando o poder de processamento dos computadores.

Tal alternativa foi então aumentar o número de unidades de processamento, seja em um *chip* com mais de uma unidade (chamada núcleo, ou ***core*** em inglês) ou em um sistema com mais de uma máquina, que por sua vez possui uma ou mais unidades de processamento. Dessa forma, o número de tarefas concluídas por unidade de tempo aumentou, visto. A isso foi dado o nome de **paralelismo**.

Existem muitos motivos para se utilizar o paralelismo. O mais importante é que o universo possui muitos processos paralelos, ou seja, ocorrendo ao mesmo tempo, tais como mudanças climáticas, montagens de veículos e aeronaves, tráfico em um pedágio e acessos a um site. Em consequência da necessidade de se processar esses eventos por computadores, surgem outros motivos para a computação paralela:

- poupar tempo e/ou dinheiro: existem problemas computacionais que são muito demorados, senão impossíveis, para se resolver serialmente. Dessa forma, usando-se a computação serial, se paga mais, visto o uso por mais tempo do poder computacional;
- resolver problemas maiores e/ou mais complexos, visto o processamento mais rápido e a possibilidade de se dividir o trabalho mais máquinas;
- prover concorrência: realizar várias tarefas simultaneamente [Bar18].

Quando se fala em paralelismo, é importante mostrar que existem diferentes disposições de memória nesse meio, o que veremos na subseção 3.7.1. Além disso, existem diferentes paradigmas de paralelismo, sendo alguns brevemente explicados na subseção 3.7.2. Há também outros assuntos a serem tratados ao se projetar programas paralelos, sendo alguns abordados na subseção 3.7.3. Fora isso, mais informações sobre paralelismo podem ser encontradas em Barney [Bar18].

3.7.1 Arquiteturas de memória na computação paralela

Na computação paralela, existem diferentes formas pelas quais a memória é implementada em um sistema. No caso deste trabalho, são elucidadas as três seguintes:

- memória compartilhada: cada unidade de processamento de um sistema tem acesso à toda a memória, com um espaço de endereçamento global;
- memória distribuída: cada unidade de processamento possui sua própria memória, mapeada apenas para ele e que pode ser acessada pelos demais nós através de requisições feitas por meio de uma rede que os liga. Além disso, cada nó opera independentemente, visto que cada um tem sua própria memória. Dessa forma, as mudanças que um nó opera sobre sua própria memória não afetam as dos demais nós;
- híbrida: literalmente a mescla de ambas, ou seja, cada nó possui mais de uma unidade de processamento e sua própria memória, sendo também capaz de acessar as dos outros [Bar18].

Essas arquiteturas são implementadas fisicamente nos sistemas de computação paralela, contudo, o programador pode escolher interpretar a arquitetura do sistema, com o qual ele trabalha, de forma diferente ou não [Bar18]. Isso é o que veremos na próxima seção.

3.7.2 Modelos da computação paralela

Segundo Barney, os modelos de programação paralela funcionam "como uma abstração sobre o *hardware* e as arquiteturas de memória" (tradução nossa), ou seja, eles servem ao programador como formas de se mascarar (ou não) a estrutura física implementada para um dado sistema de computação paralela.

Isso quer dizer que, teoricamente, se você tem um sistema com memória distribuída que será vista pelo usuário (mediante implementação) como compartilhada. Esse foi o caso, por exemplo, do supercomputador *Kendall Square Research* (KSR). Semelhantemente, um sistema com memória compartilhada poderia ser interpretado como um de memória distribuída [Bar18].

Uma lista com modelos de computação paralela diretamente relacionados com esse trabalho é dada abaixo:

- modelo de memória compartilhada (sem *threads*): consiste em processos/tarefas compartilhando o mesmo espaço de endereçamento, onde eles podem ler e escrever assincronamente. Como vantagem, esse modelo não possui o conceito de posse de dados, o que torna desnecessária a explicitação de como os dados devem ser comunicados entre as tarefas. Como desvantagem, existe a necessidade de se controlar a localidade dos dados;
- modelo de *threads*: é também um tipo de programação com memória compartilhada que consiste em um programa principal que lança várias linhas de execução concorrentes com instruções chamadas *threads*. Essas últimas são designadas e executadas simultaneamente, sendo desse fato que vem o paralelismo desse modelo. Algumas APIs que usam esse modelo são `OpenMP` (Multiprocessamento Aberto, do inglês *Open Multi-Processing*) e `Pthreads` (POSIX Threads);
- modelo de memória distribuída / troca de mensagens: consiste em um conjunto de tarefas, que podem residir no mesmo espaço de endereçamento e/ou ao longo de um número arbitrário de máquinas, usando a memória local durante a computação. Além disso, as tarefas precisam mandar/receber dados de outras tarefas, o que é feito através do envio/recebimento de mensagens. Uma API que implementa esse modelo é a `MPI` (Message Passing Interface);
- modelo híbrido: mescla modelos acima, sendo um exemplo a aplicação da MPI para a comunicação entre as máquinas de um sistema e alguma API de *threads* para a realização das computações dentro de cada máquina, que é um subsistema de memória compartilhada.

não deixar de explicar esse conceito na seção da memória

3.7.3 Desenhando programas paralelos

Para se desenhar programas paralelos para esse trabalho, mais alguns conceitos, detalhes e diferenças devem ser explicados. Um deles é a diferença entre a paralelização manual e a automática. A manual se dá com o programador (preferencialmente usando uma API) determinando o local e o tempo ou da criação de *threads*, ou do envio/recebimento de mensagens ou de qualquer outra ação relacionada; por si só, sem automação. Já a automática ou é realizada por um compilador/pré-processador, que identifica as regiões do código que podem ser paralelizadas;

ou pelo programador, que seta flags ou diretivas para indicar ao compilador as partes do código a serem paralelizadas.

Outro detalhe importante é a necessidade de se entender o problema a se tentar paralelizar e o programa que o modela. Entender o programa é necessário para se saber se ele é paralelizável. Caso seja, o próximo passo seria construir um programa serial que o resolva. A partir daí, é necessário que se entenda esse programa a fim de que se descubra os pontos onde ele leva mais tempo e a paralelização seria mais eficiente; e os pontos de gargalo, onde a execução desacelera desproporcionalmente e a paralelização se torna ineficiente. Pode também ser necessário reestruturar o programa ou até procurar outro algoritmo que o resolva.

Por fim, é preciso elucidar dois conceitos a serem utilizados nessa iniciação:

- particionamento: consiste em quebrar um problema (a ser resolvido por uma abordagem paralela) em partes discretas para serem distribuídas a tarefas. Isso pode ser feito de duas formas:

    1. particionamento de domínio: o conjunto de dados a ser processado é decomposto em partes e cada subconjunto é processado;
    2. particionamento funcional: distribuição do trabalho a ser realizado a tarefas. Esse é o tipo de particionamento a ser utilizado nessa iniciação.

- sincronização: relacionada com o problema de se gerenciar a sequência do trabalho e das tarefas, se manifesta em diferentes tipos como trava/semáforo, operações síncronas de comunicação e "barreira". Esse último conceito, que será utilizado nesse trabalho, consiste em cada tarefa trabalhar até atingir um determinado ponto do programa (a barreira), onde elas são bloqueadas. Quando a última tarefa chega a esse ponto, o processo está sincronizado e o que acontece a partir daí fica a cargo do programador.

Capítulo 4

Paralelismo em prática

Introduzir as apis e porque elas existem

4.1 OpenMP

- Do que se trata a OpenMP e onde ela está incluída;
- um programa hello world com OpenMP.

4.2 Pthreads

POSIX *Threads*, abreviada para Pthreads, é uma API que serve como um "modelo de execução paralela" usando memória compartilhada como a OpenMP, mas fornecendo rotinas para criação e manipulação de *threads*, mecanismos de exclusão mútua e váriaveis de condição [Wik19e; Bar17]. Essa API permite mais controle sobre o código, dando mais liberdade ao programador em definir a paralelização deste.

Apresent referência para essa afirmação

4.2.1 Rotinas utilizadas

Nessa seção são listadas e brevemente explicadas as rotinas da API Pthreads usadas nesse trabalho.

pthread_create

```
1 int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr,
2     void *(*start_routine)(void*), void *restrict arg);
```

phtread_exit

```
1 void pthread_exit(void *value_ptr);
```

pthread_attr_init

```
1 int pthread_attr_init(pthread_attr_t *attr);
```

pthread_setdetachedstate

```
1 int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

pthread_join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

4.3 MPI

Message Passing Interface (MPI) é uma especificação de biblioteca para comunicação entre nós (computadores de uma rede) através do envio e recebimento de mensagens (modelo distribuído de passagem de mensagens), sendo os dados do espaço de endereçamento [**wiki:endereco**] de um processo (instanciado pelo MPI) são passados para o espaço de outro "através de operações cooperativas em cada processo" (tradução nossa) [**doc:mpi**].

É importante relatar que a MPI não é uma implementação em si, mas sim uma especificação, independente de fornecedores. Com isso, existem diversas implementações e usaremos apenas uma delas nesse trabalho: OpenMPI.

4.3.1 Rotinas Utilizadas

MPI_Init

MPI_Comm_size

MPI_Comm_rank

MPI_Send

MPI_Recv

MPI_Finalize

Capítulo 5

Do serial ao paralelo

Introduzir

5.1 A construção do código serial

5.2 Primeiro contato do código com as *threads* - OpenMP

5.2.1 Construindo o caminho para a OpenMP

- Explicar o código fake da avaliação de função;
- explicar o código fake das diferenças finitas 1D.

5.3 Um contato mais profundo do código com as *threads* - Pthreads

5.3.1 Construindo o caminho para a Pthreads

- explicar o código fake das diferenças finitas 1D.

5.4 Realizando a mescla de Pthreads e MPI

5.4.1 Contruindo o caminho para a mescla

- explicar o código fake híbrido;
- explicar como o código fake foi rodado no cluster (?).

Capítulo 6

Considerações Finais

Abreviações

API Application Programming Interface. 14

MPI Message Passing Interface. 15

POSIX Interface Portável entre Sistemas Operacionais, do inglês *Portable Operating System Interface* [Wik18a]. 12

Definições

API Interface de Programação de Aplicação, "é um conjunto de definições de subrotinas, protocolos de comunicação e ferramentas para a construção de *software*" [Wik19a]. 12

circuitos integrados . 10

computação de alta performance é um ramo da computação que usa supercomputadores e estuda técnicas de paralelismo visando alcançar velocidades de processamento maiores (do que as de computadores normais, como os pessoais) para a resolução de problemas computacionais complexos, como simulações, modelagens e análises computacionais. Esse ramo também pode, no lugar de focar em processamento mais rápido, resolver problemas de dimensões maiores, que não poderiam ser resolvidos em tempo hábil por computadores comuns [Tec] . 7

dependências de Controle semelhantemente às de dados, são situações em que uma instrução precisa que uma de suas anteriores leve a um resultado que permita a sua execução, como na estrutura `if (...) {...}` na linguagem `C`, cujo código presente dentro das chaves só poderá ser executado se a condição dentro dos parênteses for verdadeira [Wik18b]. 10

dependências de Dados eventos problemáticos que ocorrem quando duas ou mais instruções utilizam um mesmo recurso (um registrador, por exemplo) de forma que a primeira instrução não pode ler/escrever sobre esse recurso sem que a segunda (ou qualquer ordem próxima) já o tenha feito (ou vice-versa), de forma que não venha a ter interferência na coesão dos dados [Wik18b]. 10

efeito joule . 10

espaço de endereçamento "série de endereços discretos" que nomeiam as posições de memória [con19]. 11

flag (bandeira, no português), é um termo utilizado na computação para designar mecanismos que servem como indicadores para outros agentes de um mesmo sistema.. 13

nó nome dado na computação paralela para cada unidade de processamento (ou conjunto dessas) independente. 11

programa serial é aquele que seus comandos são executados em ordem, sequencialmente, sem paralelismo. 13

simulações numéricas . 9

threads de forma básica, uma ***thread*** pode ser entendida como uma tarefa (no inglês, *task*), composta por instruções e lançada por um programa, a ser executada pelo sistema operacional de forma independente (concorrente) a quem a lançou. Essa independência permite, por exemplo, que várias *threads* sejam lançadas e executem de forma independente entre si citeLLNL:pthreads . 12

transistores . 10

Bibliografia

[Liu+00] W. Liu et al. *BSIM 4.0.0 Technical Notes*. Rel. técn. UCB/ERL M00/39. EECS Department, University of California, Berkeley, 2000. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2000/3863.html.

[Cat09] John Catsoulis. *Designing Embedded Hardware*. O'REILLY, 2009.

[Sil09] Gabriel P. Silva. *Microarquiteturas Avançadas*. 2009. URL: http://www.dcc.ufrj.br/~gabriel/arqcomp2/Avancadas.pdf.

[Hen12] John Hennessy. *Computer architecture : a quantitative approach*. Amsterdam Boston: Morgan Kaufmann/Elsevier, 2012. ISBN: 9780123838735.

[Bar17] Blaise Barney. *Pthreads*. Acessado em 24/02/2019. 2017. URL: https://computing.llnl.gov/tutorials/pthreads/.

[Bar18] Blaise Barney. *Introduction to Parallel Computing*. Jun. de 2018. URL: https://computing.llnl.gov/tutorials/parallel_comp/.

[MD18] Marcelo Lopes de Macedo Ferreira Cândido e Luis Alberto D'Afonseca. *Modelagem Matemática da Propagação de Ondas em Meios Não Homogêneos*. Iniciação Científica. Centro Federal de Educação Tecnológica de Minas Gerais, 2018.

[Wik18a] Wikipédia. *POSIX — Wikipédia, a enciclopédia livre*. [Online; accessed 26-fevereiro-2018]. 2018. URL: %5Curl%7Bhttps://pt.wikipedia.org/w/index.php?title=POSIX&oldid=51373634%7D.

[Wik18b] Wikipedia contributors. *Dependence analysis — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Dependence_analysis&oldid=841756105. [Online; accessed 7-March-2019]. 2018.

[con19] Wikipedia contributors. *Address space — Wikipedia, The Free Encyclopedia*. [Online; accessed 22-February-2019]. 2019. URL: https://en.wikipedia.org/w/index.php?title=Address_space&oldid=881301824.

[Wik19a] Wikipedia contributors. *Application programming interface — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Application_programming_interface&oldid=884701073. [Online; accessed 24-February-2019]. 2019.

[Wik19b] Wikipedia contributors. *Computer architecture — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Computer_architecture&oldid=876516273. [Online; accessed 12-January-2019]. 2019.

[Wik19c] Wikipedia contributors. *MIPS architecture — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=MIPS_architecture&oldid=898291066. [Online; accessed 22-May-2019]. 2019.

[Wik19d] Wikipedia contributors. *Moore's law — Wikipedia, The Free Encyclopedia*. [Online; accessed 7-March-2019]. 2019. URL: https://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=886664781.

[Wik19e] Wikipedia contributors. *POSIX Threads — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=POSIX_Threads&oldid=891039524. [Online; accessed 22-May-2019]. 2019.

[Tec] Techopedia. *High-Performance Computing (HPC)*. URL: https://www.techopedia.com/definition/4595/high-performance-computing-hpc.