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
11 de agosto de 2019

Sumário

Capítulo 1

Introdução

Capítulo 2

Aquisições Sísmicas

Nesse capítulo, serão explicados o que são aquisições sísmicas e algumas formas pelas quais elas são realizadas. Além disso, se mostrará o instrumento matemático pelo qual se pode modelar as ondas sonoras (equação da onda) utilizadas nas aquisições sísmicas e, por fim, um método para resolver tal instrumento numericamente.

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. Do contrário, tal processo imprudente levaria a um alto custo monetário. 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.

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

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 (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 se obter mais dados de como aquele ambiente se comporta com o transporte de ondas e, baseando-se nisso, modelar sua estrutura em si.

Esse recolhimento dos dados consistirá em **traços**, como se pode ver na Figura 2.1. Esses consistem em gráficos das amplitudes das ondas sonoras, obtidas através dos hidrofones/geofones, ao longo do tempo. A partir desses traços, detecta-se, por análise técnica, onde se encontram as interfaces entre as camadas do domínio analisado e do que elas são feitas. Nisso consiste o processamento dos dados colhidos.

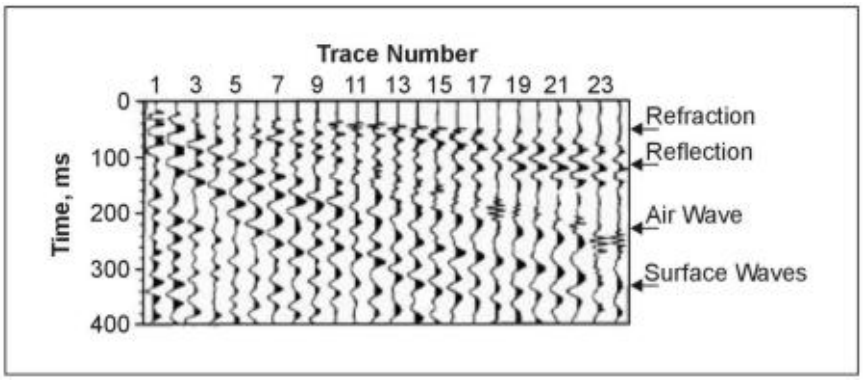

Fig. 2.1: Exemplo de traços sísmicos

Contudo, 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 **??**, é o de Método de Diferenças Finitas.

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 da implementação computacional 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 v representa (no caso desse trabalho) a velocidade da frente de onda. Trata-se de uma **equação diferencial parcial hiperbólica** com solução analítica para alguns casos, mas que pode ser resolvida numericamente, utilizando, por exemplo, o Método de Diferenças Finitas, a ser discernido 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 nas derivações de segundo grau. Podemos ver um exemplo disso na Equação 2.2, para o caso da parte espacial em x da equação.

$$\frac{\partial^2 u}{\partial x^2} \rightarrow \frac{u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}}{\Delta x^2} \tag{2.2}$$

Para entender o que significa o índice i visto na Equação acima pode-se seguir uma analogia: suponhamos um *array* com mais que três posições. A posição i seria qualquer posição

intermediária, $i-1$ a antecessora e a $i+1$ sucessora. Tal estrutura é chamada de **estêncil**. O mesmo vale para os índices j e k, mas para um *array* com três dimensões. Podemos ver uma alegoria dessa analogia na Figura **??**.

No caso desse trabalho, para a simulação da propagação de ondas em um meio bidimensional ao longo do tempo, teremos que usar três estênceis (os dois restantes podem ser vistos nas Equações 2.3 e 2.4), dois para as dimensões espaciais e um para a temporal. Para tal, podemos utilizar outra analogia: um *array* tridimensional, onde cada plano de posições no espaço é um instante no tempo.

$$\frac{\partial^2 u}{\partial y^2} \rightarrow \frac{u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}}{\Delta y^2} \tag{2.3}$$

$$\frac{\partial^2 u}{\partial t^2} \rightarrow \frac{u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1}}{\Delta t^2} \tag{2.4}$$

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 Equações 2.2, 2.3 e 2.4 se dá por

$$\begin{aligned}\frac{u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1}}{\Delta t^2} = \frac{1}{v^2}\Big(&\frac{u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}}{\Delta x^2} \\ &+ \frac{u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}}{\Delta y^2}\Big) + f(x,y,t)\end{aligned} \tag{2.5}$$

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 2.6

$$\begin{aligned}u_{i,j,k+1} = \frac{\Delta t^2}{v^2}\left(\frac{u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}}{\Delta x^2} + \frac{u_{i,j-1k} - 2u_{i,j,k} + u_{i,j+1,k}}{\Delta y^2}\right) + \\ \Delta t^2 f(x,y,t) - u_{i,j,k-1} + 2u_{i,j,k}\end{aligned} \tag{2.6}$$

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** convencional (baseando-se na arquitetura de Von Neumann) e como isso culminou no paralelismo [Bar18].

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 de máquina e dados) entregues a ele, para toda operação realizada no computador. Após esse, tem-se as políticas de manipulação de dados nas memórias cache (e seus níveis), de acesso aleatório (RAM) e de armazenamento não-volátil (discos rígidos, por exemplo). Por fim, chega-se 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].

O presente capítulo abordará alguns componentes da arquitetura computacional (processador e memória RAM) de uma forma básica, além de explicar por quais motivos a paralelização se tornou necessária e como ela se desenvolveu.

3.1 O Processador

Um processador consiste em um módulo de *hardware* 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.

3.2 A Memória Principal

3.3 Conseguir Mais Em Menos Tempo

Considerando-se que a computação iniciou com o ábaco, passando pelo uso de estruturas mecânicas (como a máquina de Charles Babbage), cartões perfurados, relés e válvulas, chegando aos atuais transistores, essa área 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 sobre patologias, ataques/defesas nacionais, enovelamento de proteínas, dinâmica molecular, monitoramento sísmico e animações tridimensionais.

Todas essas áreas costumam requerer resultados o mais rápido possível. Além disso, muitas (senão quase todas) as simulações numéricas não podem ser executadas em tempo hábil com um único processador. Visto isso, os projetistas por trás dos processadores precisaram implementar mecanismos nos processadores para explorá-los ao máximo. Contudo, encontrou-se barreiras nessa tarefa [Pac11].

3.3.1 A Barreira do Paralelismo a Nível de Instruções - *ILP Wall*

Os primeiros processadores possuíam a capacidade de executar uma instrução por ciclo de *clock*. Por isso eram chamados processadores **monociclo**. Com isso, a duração do ciclo devia ser a mesma da execução da instrução mais demorada, para que essa pudesse ser executada com segurança. Na necessidade de se adquirir mais velocidade, os projetistas perceberam que podiam particionar as instruções, de modo a executar cada estágio resultante em um ciclo de *clock*, deixando o resultado para o próximo estágio operar. Assim nasceu o processador **multiciclo**. Dessa forma, a duração do ciclo de *clock* reduziu para a mesma do estágio mais demorado dentre as instruções e a execução tornou-se mais rápida [PH17].

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, ou seja $B = A - 1$, onde A e B representam os estágios por números inteiros.

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, a próxima instrução j é buscada da memória 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. A esse encadeamento de estágios foi dado o nome de ***pipeline*** e a essa ideia, 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*, após todas as unidades funcionais estarem ocupadas.

Em seguida, para mais velocidade, 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. Com isso, nasceu o **superpipeline**.

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, 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 faz com que em alguns ciclos não haja instruções para serem executadas, visto o atraso necessário para que as dependências sejam resolvidas [Sil09].

3.3.2 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 consistente por muitas décadas [Wik19b].

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 duraria 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. O atingimento de altas temperaturas leva ao *transistor leakage* [Liu+00] (vazamento nos transistores, tradução nossa), que é o gasto energético nos transistores. Logo, tem-se um ciclo, pois o gasto energético levará novamente ao efeito joule. As altas temperaturas criam uma barreira para taxas de *clock* maiores [Fis12].

3.4 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*, necessitou-se 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) e/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 que há um número maior de unidades para realizá-las simultaneamente. 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 (ou seja, com um processador iso-

lado). 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 para mais máquinas;
- prover concorrência: realizar várias tarefas simultaneamente, o que é essencial em sistemas operacionais, por exemplo [Bar18].

Quando se fala em paralelismo, é importante mostrar que existem diferentes arquiteturas de memória nesse meio, o que veremos na subseção 3.4.1. Além disso, existem diferentes paradigmas de paralelismo, sendo alguns brevemente explicados na subseção 3.4.2. Há também outros assuntos a serem tratados ao se projetar programas paralelos, sendo alguns abordados na subseção 3.4.3. Fora isso, mais informações sobre paralelismo podem ser encontradas em Barney [Bar18].

3.4.1 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 ela 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.

3.4.2 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 pode-se ter, teoricamente, 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 capaz de lançar linhas de execução de instruções concorrentes chamadas *threads*. Essas são designadas e executadas simultaneamente, sendo desse fato que vem o paralelismo desse modelo. Algumas APIs que usam esse modelo são Open Multi-Processing (OpenMP) e POSIX Threads (Pthreads);

- 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 podem precisar mandar/receber dados de outras tarefas, o que é feito através do envio/recebimento de mensagens. Uma que implementa esse modelo é a Message Passing Interface (MPI);

- modelo híbrido: mescla modelos acima, sendo um exemplo a aplicação da MPI para a comunicação entre as máquinas de um sistema (que teria, consequentemente, memória distribuída) e alguma de *threads* para a realização das computações dentro de cada máquina, que é um subsistema de memória compartilhada.

3.4.3 Projetando Programas Paralelos

Para se projetar 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) 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 nesse iniciação:

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

    1. particionamento funcional: distribuição do trabalho a ser realizado às tarefas. Se dá bem com trabalhos cujas partes são diferentes;

2. particionamento de domínio: o conjunto de dados a ser processado é decomposto em partes. Cada um desses subconjuntos é distribuído para uma tarefa diferente e processado da mesma forma. Esse conceito será utilizado nesse trabalho;

- 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 [Bar18].

Capítulo 4

Paralelismo em prática

Para facilitar o desenvolvimento de *software* com paralelização, foram criadas interfaces com rotinas (funções) que servem de abstração para o desenvolvedor. Dessa forma, este se preocupará apenas com as estratégias de paralelização a serem adotadas e não como o paralelismo deve ocorrer em baixo nível. Tais interfaces são chamadas Application Programming Interface (API) e são indispensáveis para esse trabalho.

Nesse capítulo serão abordadas as três APIs que foram utilizadas para a realização desse trabalho: OpenMP, Pthreads e OpenMPI.

4.1 OpenMP

A API Open Multi-Processing (OpenMP) consiste em rotinas, variáveis de ambiente e diretivas de compilação reunidas em um modelo portável e escalável que serve como uma interface para desenvolvedores criarem aplicações paralelas com simplicidade e flexibilidade [**wiki:openmp**]. Essa API se encontra incluída no modelo de computação paralela de memória compartilhada, assim como a Pthreads.

4.1.1 Diretiva Utilizada

```
1 #pragma omp parallel for
```

Considerada uma diretiva de compartilhamento de trabalho, essa ferramenta permite que qualquer laço de repetição do tipo `for` (nas linguagens C/C++, `for(int i = 0; i < ALGUM_NUMERO; i++)// faz algo`, por exemplo) tenha seu número de iterações dividido entre n *thread*, sendo $0 < n <=$ número máximo de threads.

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* e outras funcionalidades fora do escopo desse trabalho [Wik19c; Bar17]. Essa API permite mais controle sobre o código, dando mais liberdade ao programador em definir a paralelização deste.

4.2.1 Rotinas Utilizadas

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

phtread_create

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

Essa rotina dá inicio a uma *thread* (primeiro argumento), podendo definí-la com um atributo previamente criado. Tal *thread* executará a rotina cuja referência é passada como argumento. O argumento `arg` é o parâmetro passado à rotina referenciada [03b].

phtread_exit

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

Rotina responsável por terminar a execução de uma *thread* [03c].

pthread_attr_init

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

Rotina responsável por inicializar um atributo a ser utilizado na rotina `pthread_create`. Tal atributo pode determinar que as *thread* a serem criadas com ele sejam sincronizáveis, por exemplo, o que será utilizado no capítulo [5] [03a].

pthread_setdetachedstate

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

Rotina que possibilita a determinação do atributo `attr`, a ser passado a uma *thread* em sua criação, como *joinable*, ou seja, sincronizável. Esse atributo também pode ser determinado como *detached*, ou seja, desanexado; contudo isso foge ao escopo desse trabalho [03e].

pthread_join

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

Essa rotina é responsável por pausar a execução da *thread* que a chamou enquanto a *thread*-alvo (passada por argumento) não terminar. Caso a *thread*-alvo já tenha terminado, a função simplesmente retorna com sucesso.

Essa função pode ser utilizada pelo desenvolvedor para sincronizar o funcionamento de um conjunto de *threads*. Ou seja, ela permite a criação de "áreas paralelas" no código, cujo início se dá na criação das *threads* e o fim quando a dita rotina for chamada para todas as *threads* existentes. Dessa forma, o programa não passará da área paralela enquanto houver alguma *thread* sem terminar [03d].

4.3 MPI

Message Passing Interface (MPI) é uma especificação de biblioteca para comunicação entre nós através do envio e recebimento de mensagens (modelo distribuído de passagem de mensagens), sendo que os dados do espaço de endereçamento de um processo (instanciado pelo MPI)

são passados para o espaço de outro "através de operações cooperativas em cada processo" (tradução nossa) [For15].

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 será utilizada apenas uma delas nesse trabalho: OpenMPI.

4.3.1 Rotinas Utilizadas

MPI_Init

```
int MPI_Init(int *argc, char ***argv)
```

Rotina responsável por inicializar o ambiente MPI, sendo que seus argumentos são referências aos argumentos do programa principal no qual o ambiente é inicializado [17d].

MPI_Comm_size

```
int MPI_Comm_size ( MPI_Comm comm, int *size )
```

A rotina acima retorna o tamanho de um grupo associado ao comunicador passado por parâmetro [17b]. No contexto do padrão MPI, um comunicador é um objeto que descreve um grupo de processos [Eij16].

MPI_Comm_rank

```
int MPI_Comm_rank ( MPI_Comm comm, int *rank )
```

Essa rotina retorna a posição do processo computacional, que a chama, no *rank* do comunicador [17a].

MPI_Send

```
int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm )
```

Rotina que tem como objetivo enviar o conteúdo do *buffer* apontado por `buf`, que contém um número (`count`) de elementos do tipo dado por `datatype` para o processo destino (`dest`), numerado conforme o *rank* do comunicador `comm`. O parâmetro `tag` é ignorado nesse trabalho.

Quando o comando dessa rotina é alcançado no código, a execução do mesmo não continua até que a mensagem que está sendo enviada seja recebida [17f].

MPI_Recv

```
int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Status *status )
```

Essa rotina trabalha semelhantemente à `MPI_Send`, mas dessa vez recebe uma mensagem de `src` [17e]. O argumento `status` é ignorado nesse trabalho.

MPI_Finalize

```
int MPI_Finalize()
```

Simplesmente finaliza o ambiente de execução MPI [17c].

Capítulo 5

Do serial ao paralelo

Como já dito anteriormente, o objetivo desse trabalho é a construção de um código paralelizado capaz de resolver a equação da onda utilizando o Método de Diferenças Finitas. O presente capítulo apresentará a construção desse código, partindo do serial equivalente, passando pelos estudos realizados com as Application Programming Interfaces (APIs) e concluindo com a construção do código paralelo final.

5.1 A construção do código serial

Como dito anteriormente na Seção 3.4.3, após se compreender o problema a ser programado em termos paralelos, deve-se então criar o código serial que resolve o problema.

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. 13, 14, 16

MPI Message Passing Interface. 11, 14, 15

OpenMP Open Multi-Processing. 11, 13

OpenMPI Open Message Passing Interface. 13

Pthreads POSIX Threads. 11, 13, 14

RAM Random Access Memory. 7

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]. 11, 13

arquitetura de Von Neumann A DEFINIR. 7

baixo nível A DEFINIR. 13

circuitos integrados . 9

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 [Wik18]. 9

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 [Wik18]. 9

efeito joule . 9

espaço de endereçamento "série de endereços discretos" que nomeiam as posições de memória [con19]. 10

flag (bandeira, no português), é um termo utilizado na computação para designar mecanismos que servem como indicadores para outros agentes de um mesmo sistema.. 11

instruções de máquina A DEFINIR. 7

localidade A DEFINIR. 11

nó nome dado na computação paralela para cada unidade de processamento (ou conjunto dessas) independente. 10, 14

processo computacional A DEFINIR. 15

programa serial é aquele que seus comandos são executados em ordem, sequencialmente, sem paralelismo. 11

thread 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. 11, 13, 14

transistores . 9

Appendices

Apêndice A

Código Serial

```
#include "stdio.h"
#include "util.h"
#include "_2DWave.h"

#define SPECS_DIR "./specs/"

using namespace std;

int main(int argc, char const *argv[]) {

    double Lx, Ly, tMax, Mx, Ny, w, A, Xp, Yp, Tp;
    int N, aux;
    char buffer[64];

    aux = scanf("%s",  buffer);
    aux = scanf("%lf", &Lx);

    aux = scanf("%s",  buffer);
    aux = scanf("%lf", &Ly);

    aux = scanf("%s",  buffer);
    aux = scanf("%lf", &tMax);

    aux = scanf("%s",  buffer);
    aux = scanf("%lf", &Mx);

    aux = scanf("%s",  buffer);
    aux = scanf("%lf", &Ny);

    aux = scanf("%s",  buffer);
    aux = scanf("%lf", &w);

    aux = scanf("%s",  buffer);
    aux = scanf("%lf", &A);

    aux = scanf("%s",  buffer);
    aux = scanf("%lf", &Xp);

    aux = scanf("%s",  buffer);
    aux = scanf("%lf", &Yp);

    aux = scanf("%s",  buffer);
    aux = scanf("%lf", &Tp);

    aux = scanf("%s",  buffer);
    aux = scanf("%d", &N);

    double vl[N + 1][3];
    double it[N][2];

    for (int i = 0; i < N; i++) {
        aux = scanf("%s", buffer);
        aux = scanf("%lf %lf %lf", &vl[i][0], &vl[i][1], &vl[i][2]);

```

```
55          aux = scanf("%s", buffer);
56          aux = scanf("%lf %lf", &it[i][0], &it[i][1]);
57      }
58
59      aux = scanf("%s",  buffer);
60      aux = scanf("%lf %lf %lf", &vl[N][0], &vl[N][1], &vl[N][2]);
61
62      int snaps;
63      aux = scanf("%s",   buffer);
64      aux = scanf("%d\n", &snaps);
65      if (snaps) {
66      aux = scanf("%s", buffer);
67          aux = scanf("%d", &snaps);
68      }
69
70      int nSrcs;
71      double offset_srcs;
72
73      aux = scanf("%s", buffer);
74      aux = scanf("%d", &nSrcs);
75
76      aux = scanf("%s",  buffer);
77      aux = scanf("%lf", &offset_srcs);
78
79      printf("\n\n");
80      printf("Lenght in x: %lf\n", Lx);
81      printf("Lenght in y: %lf\n", Ly);
82      printf("Lenght in time: %lf\n", tMax);
83      printf("Points in x: %lf\n", Mx);
84      printf("Points in y: %lf\n", Ny);
85      printf("Frequency of the wave: %lf\n", w);
86      printf("Amplitude of the wave: %lf\n", A);
87      printf("Peak's X coordinate: %lf\n", Xp);
88      printf("Peak's Y coordinate: %lf\n", Yp);
89      printf("Peak's time instant: %lf\n", Tp);
90      printf("Number of interfaces/velocities (interfaces + 1): %d\n", N);
91      for (int i = 0; i < N; i++) {
92          printf("vl0 vl1 vl2: %lf %lf %lf\n", vl[i][0], vl[i][1], vl[i][2]);
93          printf("it0 it1:     %lf %lf\n",     it[i][0], it[i][1]);
94      }
95      printf("vl0 vl1 vl2: %lf %lf %lf\n", vl[N][0], vl[N][1], vl[N][2]);
96      printf("Snaps: %d\n", snaps);
97      printf("\n\n");
98
99      ofstream wOut( SPECS_DIR "wOut.dat", ios::out | ios::binary);
100     _2DWave wv(Lx, Ly, tMax, Mx, Ny, w, A, Xp, Yp, Tp);
101     wv.serialize(&wOut);
102     wOut.close();
103
104     ofstream vOut( SPECS_DIR "vOut.dat", ios::out | ios::binary);
105     for (int i = 0; i < N + 1; i++) {
106         velocity v(vl[i][0], vl[i][1], vl[i][2]);
107         v.serialize(&vOut);
108     }
109     vOut.close();
110
111     ofstream iOut( SPECS_DIR "iOut.dat", ios::out | ios::binary);
112     for (int i = 0; i < N; i++) {
113         interface j(it[i][0], it[i][1]);
114         j.serialize(&iOut);
115     }
116     iOut.close();
117
118     ofstream interfaces_ascii( SPECS_DIR "interfaces.dat", ios::out );
119     for (int i = 0; i < N; i++) {
120       interfaces_ascii << it[i][0] << "  " << it[i][1] << endl;
121     }
122     interfaces_ascii.close();
123
124
125     ofstream nInt( SPECS_DIR "nInt.dat", ios::out);
126     nInt << N << '\n';
127     nInt << snaps << '\n';
```

```
128     nInt << nSrcs << '\n';
129     nInt << offset_srcs;
130 
131     nInt.close();
132 
133     return 0;
134 }
```

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.

[03a] *PTHREAD_ATTR_DESTROY(P) POSIX Programmer's Manual*. IEEE/The Open Group. 2003.

[03b] *PTHREAD_CREATE(P) POSIX Programmer's Manual*. IEEE/The Open Group. 2003.

[03c] *PTHREAD_EXIT(P) POSIX Programmer's Manual*. IEEE/The Open Group. 2003.

[03d] *PTHREAD_JOIN(P) POSIX Programmer's Manual*. IEEE/The Open Group. 2003.

[03e] *PTHREAD_SETDETACHEDSTATE(P) POSIX Programmer's Manual*. IEEE/The Open Group. 2003.

[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.

[Pac11] Peter S. Pacheco. *An Introduction to Parallel Computing*. Elsevier, 2011. ISBN: 978-0-12-374260-5.

[Fis12] Russell Fish. *Future of computers - Part 2: The Power Wall*. Jan. de 2012. URL: https://www.edn.com/design/systems-design/4368858/Future-of-computers--Part-2-The-Power-Wall.

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

[For15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard - Version 3.1*. Message Passing Interface Forum, 2015.

[Eij16] Victor Eijkhout. *Parallel Programming in MPI and OpenMP*. 2016. URL: %5Curl%7Bhttp://pages.tacc.utexas.edu/~eijkhout/pcse/html/index.html%7D.

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

[17a] *MPI_Comm_rank(3) Open MPI*. 2.1.1. Mai. de 2017.

[17b] *MPI_Comm_size(3) Open MPI*. 2.1.1. Mai. de 2017.

[17c] *MPI_Finalize(3) Open MPI*. 2.1.1. Mai. de 2017.

[17d] *MPI_Init(3) Open MPI*. 2.1.1. Mai. de 2017.

[17e] *MPI_Recv(3) Open MPI*. 2.1.1. Mai. de 2017.

[17f] *MPI_Send(3) Open MPI*. 2.1.1. Mai. de 2017.

[PH17] David A. Patterson e John L. Hennessy. *Organização e projeto de computadores*. Elsevier Inc., 2017.

[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.

[Wik18] 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. *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.

[Wik19c] 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.