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
31 de dezembro de 2019

Sumário

Capítulo 1

Introdução

Capítulo 2

Aquisições Sísmicas

Nesse capítulo, será explicado 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 [**notasAulas2017**].

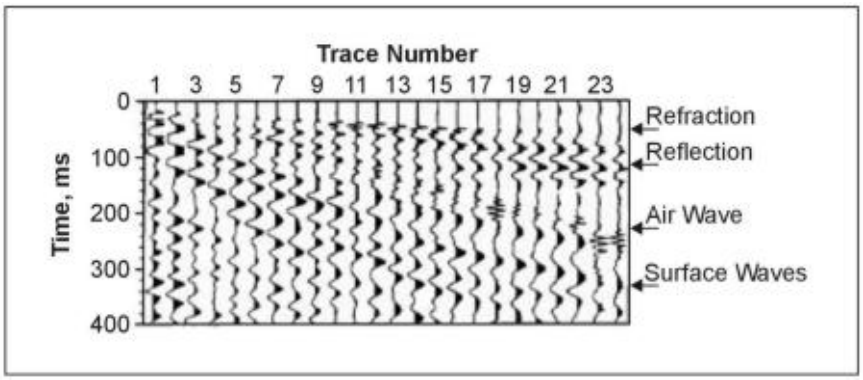


Fig. 2.1: Exemplo de traços sísmicos

Contudo, quanto aos métodos de processamento utilizados, nesse trabalho será simulado um problema direto, ou seja, estipulando o meio (nesse caso, não-homogêneo) da aquisição, averiguar-se-á como as ondas se propagam nele. Para tal, será usado um método matemático específico. Nesse trabalho, como já foi dito no Capítulo 1, 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 [**mfcandido2018**].

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 **??**.

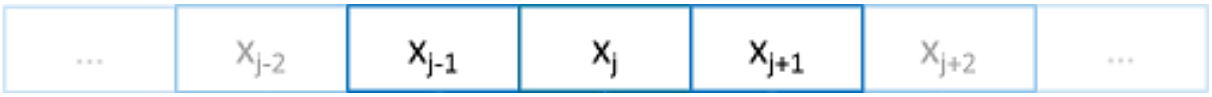


Fig. 2.2: *Stencil* unidimensional [**image:1d-stencil**]

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 [**LLNL:parcomp**].

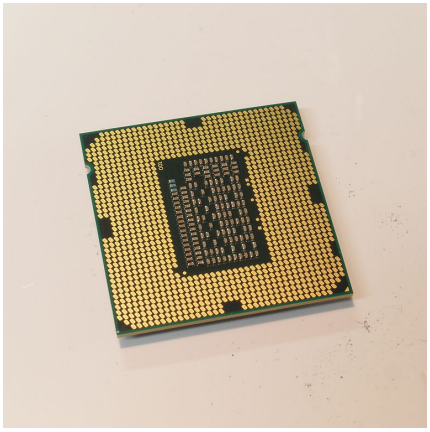
Antes de iniciar esse processo, é também necessário se explicar o que se entende por arquitetura computacional. Trata-se da área do conhecimento que estuda a interface entre *software* e *hardware*, desde o mais baixo nível, no qual o processador manipula as informações (instruções de máquina e dados) entregues a ele, para toda operação realizada no computador. Após esse nível, 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 [**Catsoulis**].

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. Um exemplo de processador pode ser visualizado nas Figuras 3.1a e 3.1b.

(a) Um processador Intel® 2500 [**wiki:i5_2500**].



(b) Os pinos de um processador Intel® 2600 [**wiki:i7_2600**]

Fig. 3.1: Exemplos de processadores

3.2 A Memória Principal

Quando se fala sobre memória computacional, pode-se estar referindo à memória cache, Read-Only Memory (ROM), Random Access Memory (RAM) ou à capacidade de armazenamento de um disco rígido. Dentre essas, a RAM é considerada a principal e um exemplo dela pode ser visto na Figura 3.2. Ela é responsável por armazenar os programas em forma de processos gerenciados pelo sistema operacional, ou seja, enquanto eles estão sendo executados. Ela também é responsável por armazenar todos os dados gerados por um programa que ainda não foram ou descartados, ou salvos em disco ou cujo acesso ainda é requisitado.

Outras características sobre a RAM são:

- o fato de que ela apenas armazena os dados até ser desligada, o que a classifica como memória volátil;
- sua capacidade de armazenamento, que, na maioria dos casos, é maior do que a de memórias cache e menor do que a de discos rígidos;
- sua velocidade, que segue o inverso da capacidade de armazenamento, ou seja, é menor do que a de memórias cache e maior do que a de discos rígidos.

Fig. 3.2: Dois módulos de RAM Swissbit® PC2-5300 DDR2 com capacidade de 2 Gigabytes, utilizada em computadores de mesa [**wiki:ddr2_ram**].

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 [**pacheco:intro-par-prog**].

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 [**Hennessy-1**].

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 [**slide:adv-microarch**].

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 [**wiki:moorelaw**].

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 [**Hennessy-2**]. 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* [**transistor-leakage**] (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 [**Fish:FoCPowerWall**].

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 isolado). 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 [**LLNL:parcomp**].

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 [**LLNL:parcomp**].

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 [**LLNL:parcomp**].

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 [**LLNL:parcomp**]. 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 [**LLNL:parcomp**].

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 [**LLNL:parcomp**].

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 [**wiki:pthreads**; **LLNL:pthreads**]. 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

```
1 int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr,
2     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 [**man:pthread_create**].

phtread_exit

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

Rotina responsável por terminar a execução de uma *thread* [**man:pthread_exit**].

pthread_attr_init

```
1 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] [**man:pthread_attr_init**].

pthread_setdetachedstate

```
1 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 [**man:pthread_attr_setdetached**

pthread_join

```
1 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 [**man:pthread_join**].

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) [**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 será utilizada apenas uma delas nesse trabalho: OpenMPI.

4.3.1 Rotinas Utilizadas

MPI_Init

```
1 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 [**man:mpi_init**].

MPI_Comm_size

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

A rotina acima retorna o tamanho de um grupo associado ao comunicador passado por parâmetro [**man:mpi_comm_size**]. No contexto do padrão MPI, um comunicador é um objeto que descreve um grupo de processos [**victor:openMP-MPI**].

MPI_Comm_rank

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

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

MPI_Send

```
1 int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest,
2                 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 [**man:mpi_send**].

MPI_Recv

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

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

MPI_Finalize

```
1 int MPI_Finalize()
```

Simplesmente finaliza o ambiente de execução MPI [**man:mpi_finalize**].

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 Código 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.

Para tal, criaram-se as classes `_2Dwave` (Código A.7), que visa representar as características da onda e também da sua fonte; `interface` (Código A.9) e `velocity` (Código A.11), que buscam representar as características do meio em que a onda propagará. Para unir essas informações e realizar os cálculos da propagação da onda utilizando o Método de Diferenças Finitas, foi criado o programa principal (Código A.19).

5.1.1 Camadas

É importante relembrar que quando se fala de camadas se refere às porções de solo subterrâneas que estão isoladas umas das outras por características diferentes e bem definidas (como o material de que são feitas e o quanto esse está comprimido, por exemplo). Duas características do meio podem ser usadas para a simulação da propagação das ondas: ângulo que as interfaces (limite de uma camada para a outra, sendo representadas por funções de primeiro nesse projeto) possuem e a velocidade (representada por uma funcão de primeiro grau com duas variáveis) das camadas.

Interfaces

Tanto no código serial quanto no final (paralelo), as interfaces das camadas são representadas pela classe `interface`. Tal classe possui dois atributos: `a` (coeficiente angular) e `b` (termo constante). Além disso, ela também possui um método, `getY(double x)`, responsável por retornar o y da reta para um dado x.

Velocidades

Já quanto às velocidades das camadas, no código serial e no final (paralelo), elas são representadas pela classe `velocity`. Essa classe possui três atributos: `a`, `b` e `c`, sendo os dois primeiros os multiplicadores das variáveis x e y da função que representa a velocidade; e o último sendo o termo constante. A classe também possui o método `getGradientVelocity(double x, double y)` que retorna a velocidade encontrada em um determinado ponto (x,y) da camada.

5.1.2 Onda

A onda bidimensional que será propagada através do meio é representada pela classe `_2Dwave`, que possui como atributos: a extensão do domínio em x (`Lx`) e y (`Ly`), o tempo máximo que a simulação da propagação deve durar (`tMax`), número de pontos em x (`Mx`) e y (`Ny`), para a discretização do domínio; a frequência da onda (`w`), sua amplitude (`A`), seu ponto (x,y) e tempo iniciais (`Xp`, `Yp` e `Tp`).

Além desses atributos, a classe também possui os métodos `evaluateFXYT(double x, double y, t)`, que retorna a amplitude da onda para determinado ponto (x,y,t); e `getVelocitiesMatrix interfaces, vector<velocity> velocities)`, que retorna uma matriz da velocidade de cada ponto (x,y) do domínio.

5.1.3 O Programa Principal

O programa principal consiste, basicamente, em recolher as informações dadas pelo usuário por meio do Código A.5; Em seguida, realizar os cálculos do Método de Diferenças Finitas e, por fim, salvar os dados resultantes a esses cálculos, que podem ser matrizes, das quais se pode obter imagens da onda propagando; ou vetores, dos quais se pode obter os traços sísmicos registrados pelos receptores simulados.

5.1.4 Os Códigos Auxiliares "Falsos"

Também foram criados códigos seriais "falsos" para que a construção do código final acontecesse de forma mais gradual e didaticamente. São chamados "falsos" porque buscam resolver problemas mais fáceis que o problema real que o projeto busca solucionar. Sobre eles foram construídos níveis de paralelização como forma de estudo para que os níveis a serem implementadas no código real fossem feitos mais facilmente.

O primeiro desses códigos (Código A.1) preenche uma matriz com os valores em z do paraboloide $z = 2x^2 + y^2$, sendo os limites de x e y delimitados pelo usuário. Já o segundo (Código A.3) calcula a propagação de uma onda unidimensional sobre uma corda ao longo do tempo, cujos limites também são definidos pelo usuário.

O primeiro código citado propõe a introdução facilitada de níveis de paralelização, não havendo nenhum problema que exija um nível maior de atenção. Já o segundo, por conta de que os cálculos do próximo instante de tempo t_1 dependem de que os cálculos de t_0 (com $t_0 < t_1$) estejam finalizados, insere o problema da sincronização ao se aplicar paralelismo com thread. Estudar esse problema é essencial antes de se implementar o código final.

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

Essa seção pretende demonstrar os esforços realizados para se construir dois códigos "falsos" que, como já dito, foram criados para facilitar a criação do código final. Esses códigos são variações dos códigos "falsos" seriais, realizando as mesmas tarefas (avaliação de função e cálculo da propagação de ondas em meio unidimensional), mas com o auxílio da API OpenMP.

5.2.1 Avaliação de uma função

O Código B.1 utiliza a estrutura

```
1 #pragma omp parallel for default(none) shared(A, x_b, y_b, x_ofst, y_ofst, x_points, y_points)
      private(i, j, x_i, y_i)
```

cuja versão mais simples foi descrita na Seção 4.1, para aplicar o paralelismo no problema.

A estrutura `default(none)` serve para explicitar que o tipo das variáveis (`shared` ou `private`) deve ser declarado pelo programador e não seguir o padrão da API, que é utilizar as variáveis como `shared` [**unp:bosco-openmp-conceitos**]. Quando declaradas como `shared`, as variáveis são compartilhadas entre as *thread* criadas pela OpenMP. Já quando declaradas com a cláusula `private`, cada *thread* possui uma cópia da variável, sendo cada cópia isolada das demais [**unp:bosco-openmp-conceitos**].

Como a API já sabe quantas iterações os laços `for` devem realizar (pela própria estrutura do laço), ela divide essa quantidade pelo número de *thread* determinadas pelo usuário, inicializando as variáveis de iteração `i` e `j` de acordo com a porção de iterações que cada *thread* recebeu para trabalhar.

- 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, 15, 17, 19

MPI Message Passing Interface. 12, 15, 16

OpenMP Open Multi-Processing. 12, 14, 19

OpenMPI Open Message Passing Interface. 14

Pthreads POSIX Threads. 12, 14, 15

RAM Random Access Memory. 7–9

ROM Read-Only Memory. 8

Definições

API Interface de Programação de Aplicação, "é um conjunto de definições de subrotinas, protocolos de comunicação e ferramentas para a construção de *software*" [**wiki:API**]. 12, 14

arquitetura de Von Neumann A DEFINIR. 7

baixo nível A DEFINIR. 14

circuitos integrados . 10

classe A DEFINIR. 17, 18

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 [**wiki:dependencies**]. 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 [**wiki:dependencies**]. 10

efeito joule . 10

espaço de endereçamento "série de endereços discretos" que nomeiam as posições de memória [**wiki:address**]. 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.. 12

instruções de máquina A DEFINIR. 7

localidade A DEFINIR. 12

nó nome dado na computação paralela para cada unidade de processamento (ou conjunto dessas) independente. 11, 15

processo computacional A DEFINIR. 16

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

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. 12, 14, 15, 18, 19

transistores . 10

Apêndice A

Códigos seriais

A.1 Códigos seriais falsos

A.1.1 Cálculo da função de um paraboloide

```
#include <iostream>
#include <stdio.h>
#include <armadillo>

using namespace std;
using namespace arma;

/** solver.cpp
 * This code aims to be an example of serial resolution of an
 * paraboloid function given the arguments below. This will be one the
 * codes for the 'fake' parallel examples.
 */

/* Args
 * x_points - number of points of the domain in the x axis
 * x_b      - beggining of the domain in x
 * x_e      - end of the domain in x
 * y_points - number of points of the domain in the y axis
 * y_b      - beggining of the domain in y
 * y_e      - end of the domain in y
**/

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

    int   x_points, y_points;
    float x_b     , y_b     ;
    float x_e     , y_e     ;

    // Creating objects for conversion of arguments
    stringstream convert0(argv[1]);
    stringstream convert1(argv[2]);
    stringstream convert2(argv[3]);
    stringstream convert3(argv[4]);
    stringstream convert4(argv[5]);
    stringstream convert5(argv[6]);

    // Putting arguments on variables
    convert0 >> x_points;
    convert1 >> x_b;
    convert2 >> x_e;
    convert3 >> y_points;
    convert4 >> y_b;
    convert5 >> y_e;

    cout << "X points: "    << x_points << "\n";
    cout << "X beggining: " << x_b      << "\n";
    cout << "X end: "       << x_e      << "\n";
```

```
    cout << "Y points: "     << y_points << "\n";
    cout << "Y beggining: " << y_b      << "\n";
    cout << "X end: "       << x_e      << "\n";

    mat A(x_points, y_points);
    rowvec parameters(6);

    // determining the space between points in x and y
    float x_ofst = (x_e - x_b) / x_points;
    float y_ofst = (y_e - y_b) / y_points;

    // Storing parameters in a vector for a file
    parameters(0) = x_points; parameters(1) = x_ofst; parameters(2) = x_b;
    parameters(3) = y_points; parameters(4) = y_ofst; parameters(5) = y_b;

    // Calculating function
    float x_i = x_b;
    float y_i = y_b;

    for (int i = 0; i < x_points; i++) {
        for (int j = 0; j < y_points; j++) {
            A(i, j) = 2. * x_i * x_i + y_i * y_i;
            y_i += y_ofst;
        }
        x_i += x_ofst;
        y_i = y_b;
    }

    parameters.save("data/outputs/pmts.dat", raw_ascii);
    A.save("data/outputs/A.dat", raw_binary);

    return 0;
}
```

Listing A.1: solver.cpp: programa que calcula um paraboloide.

```
#!/usr/bin/python2.7
#!-*- coding: utf8 -*-

import numpy as np;
import matplotlib.pyplot as plt;

'''
viewer.py
This script is responsable for generating an image that
represents the parabolloid generated by solver.cpp. For doing
this, it uses the matplotlib and numpy libraries.
'''

# Loading data
A = np.fromfile('data/outputs/A.dat', dtype=float);
[x_points, x_ofst, xi, y_points, y_ofst, yi] = np.loadtxt('data/outputs/pmts.dat');

# Preparing data for plotting
X = np.linspace(xi, xi + x_points * x_ofst, num=int(x_points));
Y = np.linspace(yi, yi + y_points * y_ofst, num=int(y_points));
A = A.reshape(int(x_points), int(y_points));

[B, C] = np.meshgrid(X, Y)

# Preparing plot
fig, ax = plt.subplots();
CS = ax.contourf(B, C, A.transpose(), 20, cmap='RdGy');
ax.clabel(CS, inline=False, fontsize=10);
ax.set_title('z = 2x^2 + y^2');

ax.plot();
plt.savefig('data/images/A.png');
```

Listing A.2: viewer.py: script que cria a imagem das projeções de camadas do paraboloide em um plano.

A.1.2 Propagação de onda em uma dimensão

```
#include <iostream>
#include <stdio.h>
#include <armadillo>

using namespace std;
using namespace arma;

/** solver.cpp
 * Aims to be an serial example of finite difference method for
 * solving 1D wave equations.
 * usage: make run xp=a xofst=b xb=c xw=d tt=e tw=f f=i
 * where
 * xp - x points
 * xb - x start
 * xt - x total
 * xw - wave's peak position
 * tt - total time
 * tw - wave's peak time
 * f - frequency
**/

#define PI 3.14159265359

// some global variables to be used in the calculations
float A, R, t_w/*ave */, x_w/*ave */, freq, freq2, pi2 = PI * PI, pi2_freq2;

float fxt(float x, float t) {
    float Dx = x - x_w;
    float Dx2 = Dx * Dx;
    float Dt = t - t_w;
    float Dt2 = Dt * Dt;
    // printf("%f %f\n", x, t);
    float result = ((1. - 2. * pi2_freq2 * Dt2) * exp(-pi2_freq2 * Dt2)) * \
                   ((1. - 2. * pi2_freq2 * Dx2) * exp(-pi2_freq2 * Dx2));
    return result;
}

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

    int   x_points   , t_points   ;
    float x_ofst     , t_ofst, x_t;
    float x_b/*egin*/, t_t/*otal*/;

    // Creating objects for conversion of arguments
    stringstream convert0(argv[1]);
    stringstream convert1(argv[2]);
    stringstream convert2(argv[3]);
    stringstream convert3(argv[4]);
    stringstream convert4(argv[5]);
    stringstream convert5(argv[6]);
    stringstream convert6(argv[7]);

    // Putting arguments on variables
    convert0 >> x_points;
    convert1 >> x_b;
    convert2 >> x_t;
    convert3 >> x_w;
    convert4 >> t_t;
    convert5 >> t_w;
    convert6 >> freq;

    x_ofst = x_t / x_points;
    t_ofst = .5 * x_ofst;
    t_points = (int) t_t / t_ofst;
    freq2 = freq * freq;
    pi2_freq2 = pi2 * freq2;

    cout << "X points    : " << x_points          << "\n";
    cout << "T points    : " << t_points          << "\n";
    cout << "X offset    : " << x_ofst            << "\n";
    cout << "T offset    : " << t_ofst            << "\n";
```

```
72      cout << "X total      : " << x_points * x_ofst << "\n";
73      cout << "T total      : " << t_t               << "\n";
74      cout << "X wave's pic: " << x_w               << "\n";
75      cout << "T wave's pic: " << t_w               << "\n";
76
77      mat A(t_points, x_points);
78      A.fill(0.);
79      rowvec parameters(5);
80
81      // Storing parameters in a vector for a file
82      parameters(0) = x_points;
83      parameters(1) = x_ofst;
84      parameters(2) = x_b;
85      parameters(3) = t_t;
86      parameters(4) = t_points;
87
88      // Calculating function
89      float x_j = x_b;
90      float t_i = 0.;
91
92      float x_ofst_2 = x_ofst * x_ofst;
93      float t_ofst_2 = t_ofst * t_ofst;
94      float termA = t_ofst_2 / x_ofst_2;
95
96      // TODO: verificar se t_i e x_j nao deveriam ser iniciados com t_ofst e \
97      // x_ofst respectivamente
98
99      for (int i = 1; i < t_points - 1; i++) {
100         for (int j = 1; j < x_points - 1; j++) {
101             A(i + 1, j) = termA * (A(i, j - 1) - 2. * A(i, j) + A(i, j + 1)) - A(i - 1, j) +
    2. * A(i, j) + t_ofst_2 * fxt(x_j, t_i);
102             x_j += x_ofst;
103         }
104         t_i += t_ofst;
105         x_j = x_b;
106     }
107
108     parameters.save("data/outputs/pmts.dat", raw_ascii);
109     A.save("data/outputs/A.dat", raw_binary);
110
111     return 0;
112 }
```

Listing A.3: solver.cpp: programa que calcula a propagação de uma onda em uma dimensão.

```
1  #!/usr/bin/python2.7
2  #!-*- coding: utf8 -*-
3
4  import numpy as np;
5  import matplotlib.pyplot as plt;
6
7  '''
8  viewer.py
9  That script can plot an image of the 1D wave propagation
10 generated by ./solver.cpp in the x and t axis.
11 '''
12
13 # Loading data
14 A = np.fromfile('data/outputs/A.dat', dtype=float);
15 [x_points, x_ofst, xi, tt, t_points] = np.loadtxt('data/outputs/pmts.dat');
16
17 # Preparing data for plotting
18 X = np.linspace(xi, xi + x_points * x_ofst, num=int(x_points));
19 T = np.linspace(0., tt, num=int(t_points));
20 A = A.reshape(int(x_points), int(t_points)).transpose();
21
22 # TODO: remove this line
23 [B, C] = np.meshgrid(X, T)
24
25 # Preparing plot
26 M = max(abs(A.min()), abs(A.max()))
27 fig, ax = plt.subplots();
28 ax.set_title('Wave in a string');
```

```
CS = ax.contourf(B, C, A, np.linspace(-M, M, 52), cmap='seismic', vmin=-M, vmax=M);
ax.clabel(CS, inline=False, fontsize=10);
plt.xlabel('X')
plt.ylabel('T')

cbar = fig.colorbar(CS)

ax.plot();
plt.savefig("data/images/A.png");
```

Listing A.4: viewer.py: script que cria a imagem da propagação da onda no plano $v \times t$.

A.2 Propagação de ondas em duas dimensões

A.2.1 Interface de linha de comando para o usuário

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

/** cli-main.cpp
 * This program is used to receive the configuration that
 * the user wants to generate data.
 */

#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];

```

```
56      for (int i = 0; i < N; i++) {
57          aux = scanf("%s", buffer);
58          aux = scanf("%lf %lf %lf", &vl[i][0], &vl[i][1], &vl[i][2]);
59
60          aux = scanf("%s", buffer);
61          aux = scanf("%lf %lf", &it[i][0], &it[i][1]);
62      }
63
64      aux = scanf("%s",  buffer);
65      aux = scanf("%lf %lf %lf", &vl[N][0], &vl[N][1], &vl[N][2]);
66
67      int snaps;
68      aux = scanf("%s",   buffer);
69      aux = scanf("%d\n", &snaps);
70      if (snaps) {
71      aux = scanf("%s", buffer);
72          aux = scanf("%d", &snaps);
73      }
74
75      int nSrcs;
76      double offset_srcs;
77
78      aux = scanf("%s", buffer);
79      aux = scanf("%d", &nSrcs);
80
81      aux = scanf("%s",  buffer);
82      aux = scanf("%lf", &offset_srcs);
83
84      printf("\n\n");
85      printf("Lenght in x: %lf\n", Lx);
86      printf("Lenght in y: %lf\n", Ly);
87      printf("Lenght in time: %lf\n", tMax);
88      printf("Points in x: %lf\n", Mx);
89      printf("Points in y: %lf\n", Ny);
90      printf("Frequency of the wave: %lf\n", w);
91      printf("Amplitude of the wave: %lf\n", A);
92      printf("Peak's X coordinate: %lf\n", Xp);
93      printf("Peak's Y coordinate: %lf\n", Yp);
94      printf("Peak's time instant: %lf\n", Tp);
95      printf("Number of interfaces/velocities (interfaces + 1): %d\n", N);
96      for (int i = 0; i < N; i++) {
97          printf("vl0 vl1 vl2: %lf %lf %lf\n", vl[i][0], vl[i][1], vl[i][2]);
98          printf("it0 it1:     %lf %lf\n",     it[i][0], it[i][1]);
99      }
100     printf("vl0 vl1 vl2: %lf %lf %lf\n", vl[N][0], vl[N][1], vl[N][2]);
101     printf("Snaps: %d\n", snaps);
102     printf("\n\n");
103
104     ofstream wOut( SPECS_DIR "wOut.dat", ios::out | ios::binary);
105     _2DWave wv(Lx, Ly, tMax, Mx, Ny, w, A, Xp, Yp, Tp);
106     wv.serialize(&wOut);
107     wOut.close();
108
109     ofstream vOut( SPECS_DIR "vOut.dat", ios::out | ios::binary);
110     for (int i = 0; i < N + 1; i++) {
111         velocity v(vl[i][0], vl[i][1], vl[i][2]);
112         v.serialize(&vOut);
113     }
114     vOut.close();
115
116     ofstream iOut( SPECS_DIR "iOut.dat", ios::out | ios::binary);
117     for (int i = 0; i < N; i++) {
118         interface j(it[i][0], it[i][1]);
119         j.serialize(&iOut);
120     }
121     iOut.close();
122
123     ofstream interfaces_ascii( SPECS_DIR "interfaces.dat", ios::out );
124     for (int i = 0; i < N; i++) {
125       interfaces_ascii << it[i][0] << "  " << it[i][1] << endl;
126     }
127     interfaces_ascii.close();
128
```

```
129
130     ofstream nInt( SPECS_DIR "nInt.dat", ios::out);
131     nInt << N << '\n';
132     nInt << snaps << '\n';
133     nInt << nSrcs << '\n';
134     nInt << offset_srcs;
135
136     nInt.close();
137
138     return 0;
139 }
```

Listing A.5: cli-main.cpp: programa para gerar as receber as entradas do usuário para o programa serial.

A.2.2 Bibliotecas criadas

```
1  #include "util.h"
2  #include "interface.h"
3  #include "velocity.h"
4
5  /** _2DWave.h
6   * 2D wave class public functions
7   */
8
9  class _2DWave {
10     /**
11      * TODO: document
12      */
13     private:
14         double Lx, Ly, tMax, Mx, Ny, w, A, Xp, Yp, Tp; // Entries by the user
15         double dx, dy, dt, Ot, R; // Method's internal variables
16
17     public:
18         // TODO: document
19         _2DWave ( double Lx,
20                   double Ly,
21                   double tMax,
22                   double Mx,
23                   double Ny,
24                   double w,
25                   double A,
26                   double Xp,
27                   double Yp,
28                   double Tp);
29
30         double getLx();
31         void setLx(double Lx);
32         double getLy();
33         void setLy(double Ly);
34         double getTMax();
35         void setTMax(double tMax);
36         double getMx();
37         void setMx(double Mx);
38         double getNy();
39         void setNy(double Ny);
40         double getW();
41         void setW(double w);
42         double getA();
43         void setA(double A);
44         double getXp();
45         void setXp(double Xp);
46         double getYp();
47         void setYp(double Yp);
48         double getTp();
49         void setTp(double Tp);
50         double getOt();
51         void setOt(double Ot);
52         double getDx();
53         void setDx(double Dx);
54         double getDy();
55         void setDy(double Dy);
```

```
56          double getDt();
57          void setDt(double Dt);
58
59          /**
60           * Function that returns a bidimensional velocities array.
61           * TODO: redocument
62           */
63          double evaluateFXYT(double X, double Y, double T);
64
65          /** Function that returns the bidimensional velocities matrix
66           *  of the medium.
67           *  For each medium's point that the wave propagates, we calculate
68           *  a velocity based on which layer of the medium this point is
69           * found.
70           *
71           *  Receives:       interfaces - a vector of interface objects
72           *                  velocities - a vector of velocity objects,
73           *                               where each object represents the
74           *                               velocity function of its
75           *                               respective layer
76           */
77          mat getVelocitiesMatrix( vector<interface> interfaces,
78                                   vector<velocity>  velocities );
79
80          void serialize(ofstream *file);
81
82          void deserialize(ifstream *file);
83  };
```

Listing A.6: _2DWave.h: arquivo-cabeçalho para a classe que representa a onda.

```
1  #include "_2DWave.h"
2
3  /** _2DWave.cpp
4   * 2D wave class definition
5   */
6
7  // TODO: document
8  _2DWave::_2DWave (double Lx,
9                    double Ly,
10                   double tMax,
11                   double Mx,
12                   double Ny,
13                   double w,
14                   double A,
15                   double Xp,
16                   double Yp,
17                   double Tp) {
18      this->Lx = Lx;                          // Extension of medium in x
19      this->Ly = Ly;                          // Depth of the medium in y
20      this->tMax = tMax;                      // Maximum simulation time
21      this->Mx = (int) Mx;                    // Number of points in the x axis
22      this->Ny = (int) Ny;                    // Number of points in the y axis
23      this->w  = w;                           // Domminant frequency omega
24      this->A  = A;                           // Wave's amplitude
25      this->Xp = Xp;                          // X coordinate of the peak of the pulse
26      this->Yp = Yp;                          // Y coordinate of the peak of the pulse
27      this->Tp = Tp;                          // Instant of the peak of the pulse
28      this->dx = Lx / (Mx - 1);               // x axis's interval
29      this->dy = Ly / (Ny - 1);               // y axis's interval
30      this->dt = this->dy / 10.;              // Time step
31      this->Ot = (int) ceil(tMax / this->dt); // Number of instants in the time
32      this->R = PI * PI + w * w;              // TODO: explain
33  }
34
35  // TODO: Esses funcoes deveriam ser inline
36
37  double _2DWave::getLx() {
38      return this->Lx;
39  }
40
41  void _2DWave::setLx(double Lx) {
42      this->Lx = Lx;
```

```
43 }
44
45 double _2DWave::getLy() {
46     return this->Ly;
47 }
48
49 void _2DWave::setLy(double Ly) {
50     this->Ly = Ly;
51 }
52
53 double _2DWave::getTMax() {
54     return this->tMax;
55 }
56
57 void _2DWave::setTMax(double tMax) {
58     this->tMax = tMax;
59 }
60
61 double _2DWave::getMx() {
62     return this->Mx;
63 }
64
65 void _2DWave::setMx(double Mx) {
66     this->Mx = Mx;
67 }
68
69 double _2DWave::getNy() {
70     return this->Ny;
71 }
72
73 void _2DWave::setNy(double Ny) {
74     this->Ny = Ny;
75 }
76
77 double _2DWave::getW() {
78     return this->w;
79 }
80
81 void _2DWave::setW(double w) {
82     this->w = w;
83 }
84
85 double _2DWave::getA() {
86     return this->A;
87 }
88
89 void _2DWave::setA(double A) {
90     this->A = A;
91 }
92
93 double _2DWave::getXp() {
94     return this->Xp;
95 }
96
97 void _2DWave::setXp(double Xp) {
98     this->Xp = Xp;
99 }
100
101 double _2DWave::getYp() {
102     return this->Yp;
103 }
104
105 void _2DWave::setYp(double Yp) {
106     this->Yp = Yp;
107 }
108
109 double _2DWave::getTp() {
110     return this->Tp;
111 }
112
113 void _2DWave::setTp(double Tp) {
114     this->Tp = Tp;
115 }
```

```
116
117 double _2DWave::getOt() {
118     return this->Ot;
119 }
120
121 void _2DWave::setOt(double Ot) {
122     this->Ot = Ot;
123 }
124
125 double _2DWave::getDx() {
126     return this->dx;
127 }
128
129 void _2DWave::setDx(double dx) {
130     this->dx = dx;
131 }
132
133 double _2DWave::getDy() {
134     return this->dy;
135 }
136
137 void _2DWave::setDy(double dy) {
138     this->dy = dy;
139 }
140
141 double _2DWave::getDt() {
142     return this->dt;
143 }
144
145 void _2DWave::setDt(double dt) {
146     this->dt = dt;
147 }
148
149 double _2DWave::evaluateFXYT(double x, double y, double t) {
150     // TODO: Refactor
151     /**
152      * Function that returns a bidimensional velocities array.
153      * TODO: redocument
154     */
155
156     // Defining Tterm
157     double tTerm = t - this->Tp;
158
159     if( tTerm > 0.1*this->tMax ) return 0;
160
161     tTerm *= tTerm;
162     tTerm *= this->R;
163
164     // Defining Xterm
165     double xTerm = x - this->Xp;
166     xTerm *= xTerm;
167
168     // Defining Yterm
169     double yTerm = y - this->Yp;
170     yTerm *= yTerm;
171
172     // Defining Dterm
173     double dTerm = xTerm + yTerm;
174
175     if( dTerm > 0.2*this->Lx ) return 0;
176
177     dTerm *= this->R;
178
179     // CAUTION: the minus in front of Tterm and Dterm
180     return this->A * exp(-tTerm) * ((1 - 2 * dTerm) * exp(-dTerm));
181
182 }
183
184 mat _2DWave::getVelocitiesMatrix( vector<interface> interfaces,
185                                   vector<velocity>  velocities ) {
186     // TODO: Refactor
187     /** Function that returns the bidimensional velocities matrix
188      *  of the medium.
```

```
189      *  For each medium's point that the wave propagates, we calculate
190      *  a velocity based on which layer of the medium this point is
191      * found.
192      *
193      *  Receives:        interfaces - a vector of interface objects
194      *                   velocities - a vector of velocity objects,
195      *                                where each object represents the
196      *                                velocity function of its
197      *                                respective layer
198      */
199
200     // Instatiating matrix for velocities
201     mat v((int) this->Mx, (int) this->Ny);
202
203     int k = 0;
204
205     // Putting the values on the velocities matrix
206     for (int i = 0; i < this->Mx; i++) {
207
208         double x;
209
210         x = i * this->dx;    // Step in the abscissas
211
212         for (int j = 0, k = 0; j < this->Ny; j++, k = 0) {
213
214             double y;
215
216             y = j * this->dy;    // Step in the ordinates
217
218             // cout << interfaces.size() - 1 << '\n';
219             while ((y > interfaces[k].getY(x)) && (k < interfaces.size()))
220             {
221                 k += 1; // Looking for on which layer the point is
222             }
223
224             // Calculating the velocity in the point
225             v(i, j) = velocities[k].getGradientVelocity(x, y);
226         }
227     }
228
229     return v;
230 }
231
232 /** Function for serialization of velocity objects
233  *  Receives:       ofstream object - file that will receive the data
234  *                                    of the this velocity object
235  */
236 void _2DWave::serialize(ofstream *file) {
237     if((*file).is_open()) {
238         double data[10]; // for writing attributes of the class
239                          // in the file
240         data[0] = this->Lx;
241         data[1] = this->Ly;
242         data[2] = this->tMax;
243         data[3] = this->Mx;
244         data[4] = this->Ny;
245         data[5] = this->w;
246         data[6] = this->A;
247         data[7] = this->Xp;
248         data[8] = this->Yp;
249         data[9] = this->Tp;
250         (*file).write((char *) &data, sizeof(data));
251     } else {
252         printf("Error: The file isn't open.\nAborting...");
253         // exit(1);
254     }
255 }
256
257 /** Function for deserialization of velocity objects
258  *  Receives:       ifstream object - file that will supply the data
259  *                                    for the this velocity object
260  */
261 void _2DWave::deserialize(ifstream *file) {
```

```
262     if((*file).is_open()) {
263         double data[10]; // for writing attributes of the class
264                          // in the file
265         (*file).read((char *) &data, sizeof(data));
266         this->Lx = data[0];
267         this->Ly = data[1];
268         this->tMax = data[2];
269         this->Mx = data[3];
270         this->Ny = data[4];
271         this->w =  data[5];
272         this->A =  data[6];
273         this->Xp = data[7];
274         this->Yp = data[8];
275         this->Tp = data[9];
276         // Redoing the atributions that depend of the data deserialize above
277         this->dx = this->Lx / (this->Mx - 1); // x axis's interval
278         this->dy = this->dx;             // y axis's interval
279         this->dt = this->dy / 2.;
280         // Number of instants in the time
281         this->Ot = (int) ceil(this->tMax / this->dt);
282         this->R = PI * PI + this->w * this->w;  // TODO: explain
283     } else {
284         printf("Error: The file isn't open.\nAborting...");
285         // exit(1);
286     }
287 }
```

Listing A.7: _2DWave.h: arquivo fonte para a classe que representa a onda.

```
1  #include <fstream>
2
3  using namespace std;
4
5  /** interface.h
6   * Inteface class' public functions
7   */
8
9  class interface {
10     /**
11      * Defines an interface as a linear function
12      */
13
14     private:
15         double a, b;
16
17     public:
18         interface(double a, double b);
19
20         // Getters
21         // (setters are not made because it's not
22         // supposed that 'a' and 'b' change after defined by
23         // the constructor)
24         double getA();
25
26         double getB();
27
28         // The interface was imaginated like a linear function:
29         //                      y = ax + b
30         // So, to get a the height of an point (the y coordinate)
31         // in the interface, we just give a point x to it
32         double getY(double x);
33
34         /** Function for serialization of interface objects
35          *  Receives:       ofstrem object - file that will receive the data
36          *                                   of the this interface object
37          */
38         void serialize(ofstream *file);
39
40         /** Function for deserialization of interface objects
41          *  Receives:       ifstrem object - file that will supply the data
42          *                                   for the this interface object
43          */
44         void deserialize(ifstream *file);
```

```
45 };
```

Listing A.8: interface.h: arquivo-cabeçalho para a classe que representa as interfaces.

```
1  #include "interface.h"
2
3  /** inteface.cpp
4   * Interface class definition
5   */
6
7  interface::interface(double a, double b) {
8      /**
9       * Constructor
10      * Receives:    a - angular coefficient
11      *              b - independent term
12      */
13     this->a = a;
14     this->b = b;
15 }
16
17 // Getters
18 // (setters are not made because it's not
19 // supposed that 'a' and 'b' change after defined by
20 // the constructor)
21 double interface::getA() {
22     return this->a;
23 };
24
25 double interface::getB() {
26     return this->b;
27 };
28
29 // The interface was imaginated like a linear function:
30 //                      y = ax + b
31 // So, to get a the height of an point (the y coordinate)
32 // in the interface, we just give a point x to it
33 double interface::getY(double x) {
34     /**
35      * Function definition
36      * Receives:    x - point's coordinate in the abscissas
37      * Returns:     y - point's coordinate in the ordinates
38      */
39     return this->a * x + this->b;
40 }
41
42 /** Function for serialization of interface objects
43  *  Receives:       ofstrem object - file that will receive the data
44  *                                   of the this interface object
45  */
46 void interface::serialize(ofstream *file) {
47     if((*file).is_open()) {
48         double data[2]; // for writing attributes of the class
49                         // in the file
50         data[0] = this->a; data[1] = this->b;
51         (*file).write((char *) &data, sizeof(data));
52     } else {
53         printf("Error: The file isn't open.\nAborting...");
54         // exit(1);
55     }
56 }
57
58 /** Function for deserialization of interface objects
59  *  Receives:       ifstrem object - file that will supply the data
60  *                                   for the this interface object
61  */
62 void interface::deserialize(ifstream *file) {
63     if((*file).is_open()) {
64         double data[2];
65         (*file).read((char *) &data, sizeof(data));
66         this->a = data[0]; this->b = data[1];
67     } else {
68         printf("The file isn't open.\nAborting...");
69         // exit(1);
```

```
    }
}
```

Listing A.9: interface.cpp: arquivo-cabeçalho para a classe que representa as interfaces.

```
#include <fstream>

using namespace std;

/** velocity.h
 *

class velocity {
    /**
     * Defines velocity as a quadractic function of the form
     *                  v(x, y) = ax + by + c
     */
    private:
        double a, b, c;

    public:
        velocity(double a, double b, double c);

        // Getters
        // (setters are not made because it's not
        // supposed that 'a' and 'b' change after defined by
        // the constructor)
        double getA();

        double getB();

        double getC();

        // TODO: document
        double getGradientVelocity(double x, double y);

        void serialize(ofstream *file);

        void deserialize(ifstream *file);

};
```

Listing A.10: velocity.h: arquivo-cabeçalho para a classe que representa as velocidades das camadas.

```
#include "velocity.h"

/** velocity.cpp
 * Layers' velocity's class
 */

velocity::velocity(double a, double b, double c) {
    /**
     * Constructor
     * Receives:    a - term 'a' of the velocity function
     *              b - term 'b' of the velocity function
     *              c - term 'c' of the velocity function
     */
    this->a = a;
    this->b = b;
    this->c = c;
}

// Getters
// (setters are not made because it's not
// supposed that 'a' and 'b' change after defined by
// the constructor)

double velocity::getA() {
    return this->a;
}

```

```
28 double velocity::getB() {
29     return this->b;
30 }
31
32 double velocity::getC() {
33     return this->c;
34 }
35
36 // TODO: document
37 double velocity::getGradientVelocity(double x, double y) {
38     return this->a * x + this->b * y + this->c;
39 }
40
41 /** Function for serialization of velocity objects
42  *  Receives:       ofstream object - file that will receive the data
43  *                                    of the this velocity object
44  */
45 void velocity::serialize(ofstream *file) {
46     if((*file).is_open()) {
47         double data[3]; // for writing attributes of the class
48                         // in the file
49         data[0] = this->a; data[1] = this->b; data[2] = this->c;
50         (*file).write((char *) &data, sizeof(data));
51     } else {
52         printf("Error: The file isn't open.\nAborting...");
53         // exit(1);
54     }
55 }
56
57 /** Function for deserialization of velocity objects
58  *  Receives:       ifstream object - file that will supply the data
59  *                                    for the this velocity object
60  */
61 void velocity::deserialize(ifstream *file) {
62     if((*file).is_open()) {
63         double data[3];
64         (*file).read((char *) &data, sizeof(data));
65         this->a = data[0]; this->b = data[1]; this->c = data[2];
66     } else {
67         printf("The file isn't open.\nAborting...");
68         // exit(1);
69     }
70 }
```

Listing A.11: velocity.cpp: arquivo fonte para a classe que representa as velocidades das camadas.

```
1 #include <cmath>
2 #include <vector>
3 #include <queue>
4 #include <iostream>
5 #include <iomanip>
6 #include <fstream>
7 #include <limits>
8 #include <armadillo>
9
10 using namespace arma;
11 using namespace std;
12
13 #define PI 3.14159265358979238463
```

Listing A.12: util.h: arquivo-cabeçalho para a definição das bibliotecas de sistema e constantes a serem utilizadas.

A.2.3 Ferramentas

```
1 #!/usr/bin/python2.7
2 #!-*- coding: utf8 -*-
3
4 import sys
5 import numpy as np
```

```
6  import matplotlib.pyplot as plt
7  import re
8  import glob
9
10 M = -1
11
12 # Set fized amplitude value
13 if( len(sys.argv) > 1 ):
14     M = float(sys.argv[1])
15
16 # Carregando arrays a partir de arquivos
17 X      = np.loadtxt('./specs/X.dat')
18 Y      = np.loadtxt('./specs/Y.dat')
19 params = np.loadtxt('./specs/output.dat')
20 inter  = np.loadtxt('./specs/interfaces.dat', ndmin=2 )
21
22 xx = X[([0,-1])]
23
24 # Criando figura
25 fig = plt.figure()
26
27 # Adicionando eixos
28 fig.add_axes()
29
30 # Criando eixo para plotagem
31 ax = fig.add_subplot(111)
32
33 # Formando base para o plot (?)
34 [Y, X] = np.meshgrid(Y, X)
35
36 # Colocando limites no plot
37 plt.xlim(0., params[4])
38 plt.ylim(0., params[5])
39
40 # Invertendo o eixo y
41 plt.gca().invert_yaxis()
42
43 # Search for all snapshot files
44 dat_files = sorted( glob.glob( "./snaps/*.dat" ) )
45
46 add_colorbar = True
47
48 # Create an image for each snapshot
49 for dat_file in dat_files:
50
51     Z = np.loadtxt( dat_file )
52
53     # Amplitude for ploting
54     if( M < 0 ):
55        M = 0.8 * max( abs(Z.min()), abs(Z.max()) )
56
57     png_file = dat_file.replace( ".dat", ".png" ).replace( "snaps/", "snaps/snap_" )
58
59     print "Creating file %s"%( png_file )
60
61     # Criando plot
62     plot = ax.contourf( X, Y, Z, np.linspace(-M,M,51), cmap=plt.cm.seismic , vmin=-M, vmax=M )
63
64     # Desenhando a barra de cores
65     if( add_colorbar ):
66         plt.colorbar(plot)
67         add_colorbar = False
68
69     # Draw interfaces
70     plt.hold(True)
71
72     for ii in range(inter.shape[0]):
73         plt.plot( xx, inter[ii,0]*xx+inter[ii,1] , '-k' )
74
75     plt.hold(False)
76
77     # Salvando a imagem
```

```
78      plt.savefig( png_file )
```

Listing A.13: plot_snapshots.py: *script* para geração dos instantâneos (*snapshots*) da propagação da onda.

```
1  #include "util.h"
2  #include "_2DWave.h"
3
4  #define SPECS_DIR "./specs/"
5  #define VEL_DIR   "./velocity/"
6
7  /**
8      This program has the purpouse of generate the velocities model.
9      That consists in a matrix with the values of velocity of each point that
10     the matrix aims to represent.
11  */
12
13 int main(int argc, char const *argv[]) {
14
15     // Getting external data
16     ifstream wf ( SPECS_DIR "wOut.dat", ios::in | ios::binary);
17     ifstream vf ( SPECS_DIR "vOut.dat", ios::in | ios::binary);
18     ifstream ifl( SPECS_DIR "iOut.dat", ios::in | ios::binary);
19
20     vector<interface> it;
21     vector<velocity> vl;
22     ifstream nInt( SPECS_DIR "nInt.dat", ios::in);
23     int N;
24     nInt >> N;
25     nInt.close();
26     for (int i = 0; i < N; i++) {
27         interface auxI(0., 0.);
28         velocity auxV(0., 0., 0.);
29         auxI.deserialize(&ifl);
30         auxV.deserialize(&vf);
31         it.push_back(auxI);
32         vl.push_back(auxV);
33     }
34     velocity auxV(0., 0., 0.);
35     auxV.deserialize(&vf);
36     vl.push_back(auxV);
37
38     _2DWave wv(0., 0., 0., 0., 0., 0., 0., 0., 0., 0.);
39
40     wv.deserialize(&wf);
41     wf.close();
42     vf.close();
43     ifl.close();
44
45     // Creating velocities matrix
46     mat velocities((int) wv.getMx(), (int) wv.getNy());
47     velocities = wv.getVelocitiesMatrix(it, vl);
48
49     // Saving velocities matrix
50     velocities.save( VEL_DIR "velocity.dat", raw_ascii);
51
52     return 0;
53 }
```

Listing A.14: gen-velocities-model.cpp: *script* para geração do modelo de velocidades.

```
1  #!/usr/bin/python2.7
2  #!-*- coding: utf8 -*-
3
4  import numpy as np
5  import matplotlib.pyplot as plt
6
7  # Carregando arrays a partir de arquivos
8  X = np.loadtxt('./specs/X.dat')
9  Y = np.loadtxt('./specs/Y.dat')
10 V = np.loadtxt('./velocity/velocity.dat')
11
```

```
12 # Criando figura
13 fig = plt.figure()
14
15 # Adicionando eixos
16 fig.add_axes()
17
18 # Criando eixo para plotagem
19 ax = fig.add_subplot(111)
20
21 # Formando base para o plot (?)
22 [Y, X] = np.meshgrid(Y,X)
23
24 # TODO: Os pontos laterais devem ser recebidos atraves do userInterface.py
25 # markers = np.array([(0., 0.), (1.5, 1.9), (3., 2.3), (4., 3.8), (6.5, 8.)], dtype=(float, 2)
       )
26
27 # Invertendo o eixo y
28 plt.gca().invert_yaxis()
29
30 # Buscando o maior valor de U para fixar o eixo em z
31 M = np.ceil( V.max() )
32
33 # Criando plot
34 plot = ax.contourf(X, Y, V, np.linspace(0,M,50), cmap=plt.cm.rainbow_r )
35
36 # Desenhando a barra de cores
37 cbar = plt.colorbar(plot)
38 cbar.ax.invert_yaxis()
39
40 # Plotando as Camadas
41 # for i in range(0, markers.size / 2):
42 #     # TODO: Trocar o 15. por uma variavel passada por parametro
43 #     ax.plot((0., 15.), (markers[i][0], markers[i][1]), '-k')
44
45 # Configurando o titulo do grafico e suas legendas
46 ax.set(title='Velocity', ylabel='Y', xlabel='X')
47
48 # Definindo caminho da plotagem
49 caminho = './velocity/velocity.png'
50
51 # Salvando a imagem
52 plt.savefig(caminho)
```

Listing A.15: plot_velocity.py: *script* para geração da imagem do modelo de velocidades criado.

```
1 #!/usr/bin/python2.7
2 #!-*- coding: utf8 -*-
3
4 import numpy as np;
5 import matplotlib.pyplot as plt;
6
7 # Loading arrays from files
8 traces = np.loadtxt('./traces/traces.dat');
9 T      = np.loadtxt('./specs/T.dat');
10 params = np.loadtxt('./output.dat');
11
12 traces = traces.transpose();
13
14 # Criando figura
15 fig = plt.figure();
16
17 # Adicionando eixos
18 fig.add_axes();
19
20 # Creating axis for ploting
21 ax = fig.add_subplot(111);
22
23 # Editing details of ploting
24 title = 'Traces';
25 ax.set(title=title, ylabel='T', xlabel='Trace');
26
27 plt.hold(1);
```

```
28
29 for i in range(int(params[6])):
30     ax.plot(T, traces[i, :]);
31     plt.savefig('./traces/' + str(i));
32     ax.clear();
```

Listing A.16: tracer.py: *script* para geração dos traços resultantes da simulação.

```
1  #!/usr/bin/python2.7
2  #!-*- coding: utf8 -*-
3
4  import numpy as np;
5
6  # Loading arrays from files
7  traces = np.loadtxt('./traces/traces.dat');
8
9  # Number of traces and samples
10 Nt = traces.shape[0];
11 Ns = traces.shape[1];
12
13 # Writing the RSF header file
14
15 header = open( "./traces/traces.rsf", "w+" )
16
17 header.write( 'tracer\n'                      )
18 header.write( 'data_format="native_float"\n' )
19 header.write( 'in="./traces/traces.rsf@"\n'  )
20 header.write( 'n1=%d\n'%( Nt )               )
21 header.write( 'n2=%d\n'%( Ns )               )
22
23 header.close()
24
25 np.transpose(traces.astype('float32')).tofile( './traces/traces.rsf@' );
```

Listing A.17: traces2rsf.py: *script* para conversão dos traços resultantes da simulação para o formato *.rsf* usado pelo Madagascar.

```
1  #!/bin/bash
2
3  # Produces wiggle plot of traces
4  sfwiggle < ./traces/traces.rsf \
5             transp=y   \
6         yreverse=y \
7         plotcol=7  \
8         zplot=.15  \
9         poly=y \
10        xmax=10 \
11        title='' \
12        n1tic=0 \
13        > ./traces/traces.vpl
14
15 # Convert vpl to png
16 vpconvert ./traces/traces.vpl \
17           format=png \
18       > ./traces/traces.png
```

Listing A.18: plot_traces.sh: *script* para geração da imagem dos traços pelo Madagascar.

A.2.4 Código principal

```
1  #include "util.h"
2  #include "_2DWave.h"
3
4  #define SPECS_DIR   "./specs/"
5  #define VEL_DIR     "./velocity/"
6  #define SNAPS_DIR   "./snaps/"
7  #define TRACES_DIR  "./traces/"
8
9  /*
10  *    This program aims to solve the bidimensional wave equation
11  *    by the finite differences method. This methods will be im-
```

```
12  *    plemented with the help of Eigen library, that provides the
13  *    necessary tools of the Linear Algebra, mainly the vectors
14  *    and matrices thats will be used to store and manipulate data,
15  *    as well the methods that are necessary to this.
16  */
17
18 int main () {
19
20     // Getting external data
21     ifstream wf ( SPECS_DIR "wOut.dat", ios::in | ios::binary);
22     ifstream vf ( SPECS_DIR "vOut.dat", ios::in | ios::binary);
23     ifstream ifl( SPECS_DIR "iOut.dat", ios::in | ios::binary);
24
25     vector<interface> it;
26     vector<velocity> vl;
27     ifstream nInt( SPECS_DIR "nInt.dat", ios::in);
28     int N;
29     nInt >> N;
30     for (int i = 0; i < N; i++) {
31         interface auxI(0., 0.);
32         velocity auxV(0., 0., 0.);
33         auxI.deserialize(&ifl);
34         auxV.deserialize(&vf);
35         it.push_back(auxI);
36         vl.push_back(auxV);
37     }
38     velocity auxV(0., 0., 0.);
39     auxV.deserialize(&vf);
40     vl.push_back(auxV);
41
42     _2DWave wv(0., 0., 0., 0., 0., 0., 0., 0., 0., 0.);
43
44     wv.deserialize(&wf);
45
46     wf.close(); vf.close(); ifl.close();
47
48     // Creating velocities matrix
49     mat v;
50     v.load( VEL_DIR "velocity.dat", raw_ascii);
51
52     // Compute 1/v^2 only once for all steps
53     for (int i = 1; i < wv.getMx() - 1; i++) {
54         for (int j = 1; j < wv.getNy() - 1; j++) {
55             v(i,j) = 1.0 / ( v(i,j) * v(i,j) );
56         }
57     }
58
59     // Creating arrays for space dimensions X e Y and for time
60     vec X = linspace<vec>(0., wv.getLx(),   wv.getMx());
61     vec Y = linspace<vec>(0., wv.getLy(),   wv.getNy());
62     vec T = linspace<vec>(0., wv.getTMax(), wv.getOt());
63
64     // Saving arrays of dimensions in space
65     X.save( SPECS_DIR "X.dat", raw_ascii);
66     Y.save( SPECS_DIR "Y.dat", raw_ascii);
67     T.save( SPECS_DIR "T.dat", raw_ascii);
68
69     nInt >> N; // reusing N
70     queue <int> numSnaps;
71     if (N > 0) { // get the frames's numbers that must be saved as
72                  // snapshots
73         int h = (int) wv.getOt() / N;
74         for (int i = 1; h * i < wv.getOt(); i++) { numSnaps.push(h * i); }
75     }
76
77     // Receiving data about receivers
78     int nRecv;
79     double offset_Recv;
80     nInt >> nRecv;
81     nInt >> offset_Recv;
82
83     nInt.close();
84
```

```
85      // Matrix to armazenate traces
86      mat traces((int) wv.getOt(), nRecv);
87      traces.fill(0.);
88
89      // the number of points between receivers
90      int offset_Recv_int = wv.getMx() / nRecv;
91
92      // Creating 3 matrices for application of the Finite Difference Method (FDM)
93      mat U1((int) wv.getMx(), (int) wv.getNy());
94      mat U2((int) wv.getMx(), (int) wv.getNy());
95      mat U3((int) wv.getMx(), (int) wv.getNy());
96
97      // Filling the matrices with zeros
98      U1.fill(0.);
99      U2.fill(0.);
100     U3.fill(0.);
101
102     // Creating a queue for administrating the arrays of FDM
103     queue <mat> U;
104     U.push(U1); U.push(U2); U.push(U3);
105
106     double dt2    =         wv.getDt() * wv.getDt();
107     double dt2dx2 = dt2 / ( wv.getDx() * wv.getDx() );
108     double dt2dy2 = dt2 / ( wv.getDy() * wv.getDy() );
109
110     for (int k = 1; k < wv.getOt() - 1; k++) {
111
112         // Getting the matrices from the queue
113         U1 = U.front(); U.pop(); // t + 1
114         U2 = U.front(); U.pop(); // t
115         U3 = U.front(); U.pop(); // t - 1
116
117         // Doing FDM calculations
118         for (     int i = 1; i < wv.getMx() - 1; i++) {
119             for ( int j = 1; j < wv.getNy() - 1; j++) {
120                 U1(i,j) =  v(i,j) * ( dt2dx2 * ( U2(i+1,j) - 2.0*U2(i,j) + U2(i-1,j) )
121                                     + dt2dy2 * ( U2(i,j+1) - 2.0*U2(i,j) + U2(i,j-1) ) )
122                         + dt2 * wv.evaluateFXYT( X(i), Y(j), T(k) )
123                         + 2.0 * U2(i,j) - U3(i,j);
124             }
125         }
126
127         // Registering traces
128         for (int ii = 1; ii * offset_Recv_int < size(U1)(0) && ii < nRecv; ii++) {
129             traces(k, ii) = U1(ii * offset_Recv_int, 1);
130         }
131
132         // if frame k is one of the required for the snaps
133         if (N > 0 && k == numSnaps.front()) {
134             ostringstream oss;
135             int save = numSnaps.front();
136             oss << SNAPS_DIR << setfill('0') << setw(5) << save << ".dat";
137             numSnaps.pop(); numSnaps.push(save);
138             U1.save(oss.str(), raw_ascii);
139         }
140
141         // Putting the matrices again in the queue
142         U.push(U3); U.push(U1); U.push(U2);
143     }
144
145     vec d(7);
146     d(0) = (int) wv.getMx();
147     d(1) = (int) wv.getNy();
148     d(2) = (int) wv.getOt();
149     d(3) = N;
150     d(4) = wv.getLx();
151     d(5) = wv.getLy();
152     d(6) = nRecv;
153
154     d.save( SPECS_DIR "output.dat", raw_ascii);
155
156     traces.save( TRACES_DIR "traces.dat", raw_ascii);
157
```

```
158     return 0;
159
160 }
```

Listing A.19: main-reflect-bound.cpp: código fonte que realiza os cálculos da propagação da onda.

Apêndice B

Códigos falsos sobre OpenMP

B.1 Cálculo da função de um paraboloide

```
#include <iostream>
#include <armadillo>
#include <stdio.h>
#include <omp.h>

using namespace std;
using namespace arma;

/* Args
 * x_points - number of points of the domain in the x axis
 * x_b      - beggining of the domain in x
 * x_e      - end of the domain in x
 * y_points - number of points of the domain in the y axis
 * y_b      - beggining of the domain in y
 * y_e      - end of the domain in y
**/

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

    int   x_points, y_points;
    float x_b     , y_b     ;
    float x_e     , y_e     ;

    // Creating objects for conversion of arguments
    stringstream convert0(argv[1]);
    stringstream convert1(argv[2]);
    stringstream convert2(argv[3]);
    stringstream convert3(argv[4]);
    stringstream convert4(argv[5]);
    stringstream convert5(argv[6]);

    // Putting arguments on variables
    convert0 >> x_points;
    convert1 >> x_b;
    convert2 >> x_e;
    convert3 >> y_points;
    convert4 >> y_b;
    convert5 >> y_e;

    cout << "X points: "    << x_points << "\n";
    cout << "X beggining: " << x_b      << "\n";
    cout << "X end: "       << x_e      << "\n";
    cout << "Y points: "    << y_points << "\n";
    cout << "Y beggining: " << y_b      << "\n";
    cout << "X end: "       << x_e      << "\n";

    mat A(x_points, y_points);
    rowvec parameters(6);

    // determining the space between points in x and y
```

```
51      float x_ofst = (x_e - x_b) / x_points;
52      float y_ofst = (y_e - y_b) / y_points;
53
54      // Storing parameters in a vector for a file
55      parameters(0) = x_points; parameters(1) = x_ofst; parameters(2) = x_b;
56      parameters(3) = y_points; parameters(4) = y_ofst; parameters(5) = y_b;
57
58      // Calculating function
59      float x_i = x_b;
60      float y_i = y_b;
61
62      int i, j;
63      #pragma omp parallel for default(none) shared(A, x_b, y_b, x_ofst, y_ofst, x_points,
        y_points) private(i, j, x_i, y_i)
64      for (i = 0; i < x_points; i++) {
65          x_i = x_b + i * x_ofst;
66          y_i = y_b;
67          for (j = 0; j < y_points; j++) {
68              A(i, j) = 2. * x_i * x_i + y_i * y_i;
69              y_i += y_ofst;
70          }
71      }
72
73      parameters.save("data/outputs/pmts.dat", raw_ascii);
74      A.save("data/outputs/A.dat", raw_binary);
75
76      return 0;
77  }
```

Listing B.1: solver.cpp: código fonte que realiza os cálculos da função de um paraboloide com o auxílio da OpenMP.

```
1  #!/usr/bin/python2.7
2  #!-*- coding: utf8 -*-
3
4  import numpy as np;
5  import matplotlib.pyplot as plt;
6
7  # Loading data
8  A = np.fromfile('data/outputs/A.dat', dtype=float)
9  [x_points, x_ofst, xi, y_points, y_ofst, yi] = np.loadtxt('data/outputs/pmts.dat')
10
11 # Preparing data for plotting
12 X = np.linspace(xi, xi + x_points * x_ofst, num=int(x_points))
13 Y = np.linspace(yi, yi + y_points * y_ofst, num=int(y_points))
14 A = A.reshape(int(x_points), int(y_points))
15
16 [B, C] = np.meshgrid(X, Y)
17
18 # Preparing plot
19 fig, ax = plt.subplots()
20 CS = ax.contourf(B, C, A.transpose(), 20, cmap='RdGy')
21 ax.clabel(CS, inline=False, fontsize=10)
22 ax.set_title('z = 2x^2 + y^2')
23
24 ax.plot()
25 plt.savefig('data/images/A.png')
```

Listing B.2: viewer.py: código fonte que gera a imagem resultante dos cálculos da função de um paraboloide.

B.2 Propagação de onda em uma dimensão

Apêndice C

Códigos falsos sobre Pthreads

C.1 Cálculo da função de um paraboloide

```
#include <iostream>
#include <stdio.h>
#include <armadillo>
#include <pthread.h>

using namespace std;
using namespace arma;

#define NUM_THREADS 4
#define PI 3.14159265359

typedef struct {
    mat *A;
    int num_p_x_sub_mtx, num_p_y_sub_mtx_start, num_p_y_sub_mtx_end;
    float x_start, y_start;
    float x_ofst, y_ofst;
} kit;

void *calculate(void *p) {

    kit *m = ((kit *) p);

    float x = m->x_start, y = m->y_start;

    printf("%d %d\n", m->num_p_x_sub_mtx, m->num_p_y_sub_mtx_end);

    for (int i = 0; i < m->num_p_x_sub_mtx; i++) {
        for (int j = m->num_p_y_sub_mtx_start; j < m->num_p_y_sub_mtx_end; j++) {
            (*(m->A))(i, j) = 2. * x * x + y * y;
            y += m->y_ofst;
            // (*(m->A))(i, j) = sin(exp(x)) * cos(log(y));
            // (*(m->A))(i, j) = sin(PI * (x * x + y * y) / (2. * 50. * 50.));
            // z = sqrt(x^2 + y^2 - 4)
        }
        x += m->x_ofst;
        y = m->y_start;
    }

    pthread_exit(NULL);

}

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

    int x_points, y_points;
    float x_ofst, y_ofst;
    float x, y;

    // Creating objects for conversion of arguments
    stringstream convert0(argv[1]);
```

```
51      stringstream convert1(argv[2]);
52      stringstream convert2(argv[3]);
53      stringstream convert3(argv[4]);
54      stringstream convert4(argv[5]);
55      stringstream convert5(argv[6]);
56
57      // Putting arguments on variables
58      convert0 >> x_points;
59      convert1 >> x_ofst;
60      convert2 >> x;
61      convert3 >> y_points;
62      convert4 >> y_ofst;
63      convert5 >> y;
64
65      cout << "X points: " << x_points << "\n";
66      cout << "X offset: " << x_ofst << "\n";
67      cout << "Y points: " << y_points << "\n";
68      cout << "Y offset: " << y_ofst << "\n";
69
70      mat A(x_points, y_points);
71      rowvec parameters(6);
72
73      // Storing parameters in a vector for a file
74      parameters(0) = x_points; parameters(1) = x_ofst; parameters(2) = x;
75      parameters(3) = y_points; parameters(4) = y_ofst; parameters(5) = y;
76
77      // Initing Pthreads tools
78      pthread_t threads[NUM_THREADS];
79      long t;
80
81      // Defining threads as joinable threads
82      pthread_attr_t attr;
83      pthread_attr_init(&attr);
84      pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
85
86      // Creating kit for calculate()
87      kit m[NUM_THREADS];
88
89      for (int i = 0; i < NUM_THREADS - 1; i++) {
90          m[i].A = &A;
91          m[i].x_ofst = x_ofst;
92          m[i].y_ofst = y_ofst;
93          m[i].x_start = x;
94          m[i].y_start = y + i * (y_points / NUM_THREADS) * y_ofst;
95          m[i].num_p_x_sub_mtx = x_points;
96          m[i].num_p_y_sub_mtx_start = i * ceil(y_points / NUM_THREADS);
97          m[i].num_p_y_sub_mtx_end = (i + 1) * ceil(y_points / NUM_THREADS);
98      }
99
100     m[NUM_THREADS - 1].A = &A;
101     m[NUM_THREADS - 1].x_ofst = x_ofst;
102     m[NUM_THREADS - 1].y_ofst = y_ofst;
103     m[NUM_THREADS - 1].x_start = x;
104     m[NUM_THREADS - 1].y_start = y + (NUM_THREADS - 1) * (y_points / NUM_THREADS) * y_ofst;
105     m[NUM_THREADS - 1].num_p_x_sub_mtx = x_points;
106     m[NUM_THREADS - 1].num_p_y_sub_mtx_start = (NUM_THREADS - 1) * ceil(y_points / NUM_THREADS
        );
107     m[NUM_THREADS - 1].num_p_y_sub_mtx_end = y_points;
108
109     // Calculating function
110     for (t = 0; t < NUM_THREADS; t++) {
111         if (pthread_create(&threads[t], &attr, calculate, (void *) &m[t])) {
112             printf("error: creation of thread %ld failed. Aborting...\n", t);
113             exit(-1);
114         }
115     }
116
117     // Joining threads
118     void *status;
119     for (t = 0; t < NUM_THREADS; t++) {
120       if (pthread_join(threads[t], &status)) {
121           printf("error: joining of thread %ld failed. Aborting...\n", t);
122           exit(-1);
```

```
123         }
124     }
125
126     // cout << size(A);
127
128     parameters.save("data/outputs/pmts.dat", raw_ascii);
129     A.save("data/outputs/A.dat", raw_binary);
130
131     pthread_exit(NULL);
132
133     return 0;
134 }
```

Listing C.1: solver.cpp: código fonte que realiza os cálculos da função de um paraboloide com o auxílio da Pthreads.

```
1  #!/usr/bin/python2.7
2  #!-*- coding: utf8 -*-
3
4  import numpy as np;
5  import matplotlib.pyplot as plt;
6
7  # Loading data
8  A = np.fromfile('data/outputs/A.dat', dtype=float);
9  [x_points, x_ofst, xi, y_points, y_ofst, yi] = np.loadtxt('data/outputs/pmts.dat');
10
11 # Preparing data for plotting
12 X = np.linspace(xi, xi + x_points * x_ofst, x_points);
13 Y = np.linspace(yi, yi + y_points * y_ofst, y_points);
14 # print X
15 # print Y
16 # print A.shape;
17 A = A.reshape(int(x_points), int(y_points));
18
19 [B, C] = np.meshgrid(X, Y)
20
21 # Preparing plot
22 fig, ax = plt.subplots();
23 CS = ax.contourf(B, C, A.transpose(), 20, cmap='RdGy');
24 ax.clabel(CS, inline=False, fontsize=10);
25 ax.set_title('z = x^2 + y^2');
26
27 # plt.xlim(-2, 2);
28 # plt.ylim(-2, 2);
29 ax.plot();
30 plt.savefig("data/images/A.png");
```

Listing C.2: viewer.py: código fonte que gera a imagem resultante dos cálculos da função de um paraboloide.

C.2 Propagação de onda em uma dimensão

```
1  #include <iostream>
2  #include <stdio.h>
3  #include <armadillo>
4  #include <queue>
5  #include <pthread.h>
6
7  using namespace std;
8  using namespace arma;
9
10 #define NUM_THREADS 6
11 #define PI 3.14159265359
12
13 typedef struct kit_t {
14     int thread_id;
15     mat *A        ;
16     int x_pt_start,
17         x_pt_end  ,
18         i         ;
```

```
19      float x_j,
20            t_i,
21            x_ofst;
22  } KIT_t;
23
24  // some global variables to be used in the calculations
25  float A,
26        R,
27        t_w/*ave */,
28        x_w/*ave */,
29        pi2_freq2,
30        x_ofst_2,
31        t_ofst_2,
32        termA;
33
34  float fxt(float x, float t) {
35      float Dx  = x - x_w;
36      float Dx2 = Dx * Dx;
37      float Dt  = t - t_w;
38      float Dt2 = Dt * Dt;
39
40      float result = ((1. - 2. * pi2_freq2 * Dt2) * exp(-pi2_freq2 * Dt2)) * \
41                     ((1. - 2. * pi2_freq2 * Dx2) * exp(-pi2_freq2 * Dx2));
42
43      return result;
44  }
45
46  void *eval(void *void_kit) {
47      KIT_t *kit = (KIT_t *) void_kit;
48
49      int thread_id  = kit->thread_id ;
50      mat *A         = kit->A         ;
51      int x_pt_start = kit->x_pt_start;
52      int x_pt_end   = kit->x_pt_end  ;
53      float x_j      = kit->x_j       ;
54      float t_i      = kit->t_i       ;
55
56      float x_ofst   = kit->x_ofst    ;
57      int i          = 1              ;
58
59      for (int j = x_pt_start; j <= x_pt_end; j++) {
60          (*A)(i + 1, j) = termA * ((*A)(i, j - 1) - 2. * (*A)(i, j) + \
61              (*A)(i, j + 1)) - (*A)(i - 1, j) + 2. * (*A)(i, j) + t_ofst_2 * \
62              fxt(x_j, t_i);
63          x_j += x_ofst;
64      }
65
66      pthread_exit(NULL);
67  }
68
69  int main(int argc, char const *argv[]) {
70
71      int   x_points   , t_points   ;
72      float x_ofst     , t_ofst, x_t;
73      float x_b/*egin*/, t_t/*otal*/;
74
75      // auxiliary variables
76      float freq,
77            freq2,
78            pi2 = PI * PI;
79
80      // Creating objects for conversion of arguments
81      stringstream convert0(argv[1]);
82      stringstream convert1(argv[2]);
83      stringstream convert2(argv[3]);
84      stringstream convert3(argv[4]);
85      stringstream convert4(argv[5]);
86      stringstream convert5(argv[6]);
87      stringstream convert6(argv[7]);
88
89      // Putting arguments on variables
90      convert0 >> x_points;
91      convert1 >> x_b;
```

```
92      convert2 >> x_t;
93      convert3 >> x_w;
94      convert4 >> t_t;
95      convert5 >> t_w;
96      convert6 >> freq;
97
98      x_ofst    = (x_t - x_b) / x_points;
99      t_ofst    = .5 * x_ofst;
100     t_points  = (int) t_t / t_ofst;
101     freq2     = freq * freq;
102     pi2_freq2 = pi2 * freq2;
103
104     cout << "X points    : " << x_points          << "\n";
105     cout << "T points    : " << t_points          << "\n";
106     cout << "X offset    : " << x_ofst            << "\n";
107     cout << "T offset    : " << t_ofst            << "\n";
108     cout << "X total     : " << x_points * x_ofst << "\n";
109     cout << "T total     : " << t_t               << "\n";
110     cout << "X wave's pic: " << x_w               << "\n";
111     cout << "T wave's pic: " << t_w               << "\n";
112
113     // creating the matrix for calculations and a row vector for saving
114     // parameters
115     mat B(t_points, x_points);
116     B.fill(0.);
117     mat A(3, x_points);
118     A.fill(0.);
119     rowvec parameters(5);
120
121     // Storing parameters in a vector for a file
122     parameters(0) = x_points;
123     parameters(1) = x_ofst;
124     parameters(2) = x_b;
125     parameters(3) = t_t;
126     parameters(4) = t_points;
127
128     // Calculating function
129     float x_j = x_b;
130     float t_i = 0.;
131
132     // setting some calculation variables
133     x_ofst_2 = x_ofst * x_ofst;
134     t_ofst_2 = t_ofst * t_ofst;
135     termA = t_ofst_2 / x_ofst_2;
136
137     // creating threads
138     pthread_t threads[NUM_THREADS];
139     pthread_attr_t attr;
140
141     // creating kits for the threads
142     KIT_t kits[NUM_THREADS];
143
144     // initializing ptheads attribute and setting threads as joinable
145     pthread_attr_init(&attr);
146     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
147
148     // determining how many points each thread will have to take care of
149     int x_pts_per_thread = (x_points - 2) / NUM_THREADS;
150
151     // distributing information for kits
152     for(int i = 0; i < NUM_THREADS - 1; i++) {
153         kits[i].thread_id  = i                                          ;
154         kits[i].A          = &A                                        ;
155         kits[i].x_pt_start = i * x_pts_per_thread + 1                  ;
156         kits[i].x_pt_end   = i * x_pts_per_thread + x_pts_per_thread;
157         kits[i].x_j        = x_ofst * x_pts_per_thread * i             ;
158         kits[i].t_i        = t_ofst                                    ;
159         kits[i].x_ofst     = x_ofst                                    ;
160     }
161     kits[NUM_THREADS - 1].thread_id  = NUM_THREADS - 1                          ;
162     kits[NUM_THREADS - 1].A          = &A                                       ;
163     kits[NUM_THREADS - 1].x_pt_start = (NUM_THREADS - 1) * x_pts_per_thread + 1;
164     kits[NUM_THREADS - 1].x_pt_end   = x_points - 2                             ;
```

```
    kits[NUM_THREADS - 1].x_j        = x_ofst * x_pts_per_thread * \
        (NUM_THREADS - 1);
    kits[NUM_THREADS - 1].t_i        = t_ofst                                    ;
    kits[NUM_THREADS - 1].x_ofst     = x_ofst                                    ;

    // Iterating in the time
    for (int i = 1; i < t_points - 1; i++) {
        // creating threads
        for (long j = 0; j < NUM_THREADS; j++) {
            kits[j].t_i = t_ofst * i;
            if (pthread_create(&threads[j], &attr, eval, (void *) &kits[j])) {
                printf("Error on creating thread %ld\n", j);
                exit(1);
            }
        }

        // joining threads
        void *status;
        for (long j = 0; j < NUM_THREADS; j++) {
            int returnCode;
            returnCode = pthread_join(threads[j], &status);
            if (returnCode) {
                printf("Error on joining thread %ld\nThread returned %d", j,
                    returnCode);
            }
        }

        // we need always to save the last line of B before 'deleting' it
        B.row(i - 1) = A.row(0);
        A.rows(0, 1) = A.rows(1, 2);
        A.row(2).zeros();
    }

    parameters.save("data/outputs/pmts.dat", raw_ascii);
    B.save("data/outputs/A.dat", raw_binary);

    return 0;
}
```

Listing C.3: solver.cpp: código fonte que realiza os cálculos da propagação de uma onda em meio unidimensional com o auxílio da Pthreads.

```
#!/usr/bin/python2.7
#!-*- coding: utf8 -*-

import numpy as np;
import matplotlib.pyplot as plt;

# Loading data
A = np.fromfile('data/outputs/A.dat', dtype=float);
[x_points, x_ofst, xi, tt, t_points] = np.loadtxt('data/outputs/pmts.dat');

# Preparing data for plotting
X = np.linspace(xi, xi + x_points * x_ofst, num=int(x_points));
T = np.linspace(0., tt, num=int(t_points));
A = A.reshape(int(x_points), int(t_points)).transpose();

# TODO: remove this line
[B, C] = np.meshgrid(X, T)

# Preparing plot
M = max(abs(A.min()), abs(A.max()))
fig, ax = plt.subplots();
ax.set_title('Wave in a string');
CS = ax.contourf(B, C, A, 52, cmap='seismic', vmin=-M, vmax=M);
ax.clabel(CS, inline=False, fontsize=10);
plt.xlabel('X')
plt.ylabel('T')

cbar = fig.colorbar(CS)

ax.plot();
# plt.show();
```

```
32 plt.savefig("data/images/A.png");
```

Listing C.4: viewer.py: código fonte que gera a imagem resultante dos cálculos da propagação da onda em meio unidimensional.

Apêndice D

Códigos falsos sobre MPI

D.1 Cálculo da função de um paraboloide

```
#include <armadillo>
#include <iostream>
#include <mpi.h>
#include <stdio.h>
#include <string>

using namespace std;
using namespace arma;

// Based on https://computing.llnl.gov/tutorials/mpi/samples/C/mpi_pi_reduce.c

#define MASTER 0

/* Args
 * x_points - number of points of the domain in the x axis
 * x_b      - beggining of the domain in x
 * x_e      - end of the domain in x
 * y_points - number of points of the domain in the y axis
 * y_b      - beggining of the domain in y
 * y_e      - end of the domain in y
**/

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

  // Variables for working with MPI
  int task_id,   // task NUM - the identification of a task
      num_tasks, // number of tasks
      rc,        // for returning code
      i;

  // initiating MPI, starting tasks and identifying them
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &task_id);

  int x_points, y_points;
  float x_b, y_b;
  float x_e, y_e;

  // Creating objects for conversion of arguments
  stringstream convert0(argv[1]);
  stringstream convert1(argv[2]);
  stringstream convert2(argv[3]);
  stringstream convert3(argv[4]);
  stringstream convert4(argv[5]);
  stringstream convert5(argv[6]);

  // Putting arguments on variables
  convert0 >> x_points;
  convert1 >> x_b;
```

```
51    convert2 >> x_e;
52    convert3 >> y_points;
53    convert4 >> y_b;
54    convert5 >> y_e;
55
56    // determining the space between points in x and y
57    float x_ofst = (x_e - x_b) / x_points;
58    float y_ofst = (y_e - y_b) / y_points;
59    // determining the beggining and the end of the domain based on task_id
60    x_b = x_b + task_id * x_ofst * x_points;
61    x_e = x_b + x_ofst * x_points;
62
63    printf("Task %d: X points = %d      \n", task_id, x_points);
64    printf("Task %d: X beggining = %.1f \n", task_id, x_b);
65    printf("Task %d: X ending = %.1f    \n", task_id, x_e);
66    printf("Task %d: Y points = %d      \n", task_id, y_points);
67    printf("Task %d: Y beggining = %.1f \n", task_id, y_b);
68    printf("Task %d: Y ending = %.1f    \n", task_id, y_e);
69
70    mat A(x_points, y_points);
71    rowvec parameters(6);
72
73    // Storing parameters in a vector for a file
74    parameters(0) = x_points;
75    parameters(1) = x_ofst;
76    parameters(2) = x_b;
77    parameters(3) = y_points;
78    parameters(4) = y_ofst;
79    parameters(5) = y_b;
80
81    // Calculating function
82    float x_i = x_b;
83    float y_i = y_b;
84
85    printf("\nTask %d: calculation is beggining!\n", task_id);
86
87    for (int i = 0; i < x_points; i++) {
88      for (int j = 0; j < y_points; j++) {
89        A(i, j) = 2. * x_i * x_i + y_i * y_i;
90        y_i += y_ofst;
91      }
92      x_i += x_ofst;
93      y_i = y_b;
94    }
95
96    printf("\nTask %d: calculation is over!\n", task_id);
97
98    if (task_id != MASTER) { // if it's not the master
99      // sending matrix's beggining pointer to master
100     printf("\nTask %d: sending!\n", task_id);
101     MPI_Send(A.begin(), A.size(), MPI_DOUBLE, MASTER, 0, MPI_COMM_WORLD);
102   } else { // if it's the master
103     mat B(num_tasks * x_points, y_points);
104     B.rows(0, x_points - 1) = A;
105     for (int i = 1; i < num_tasks; i++) {
106       printf("\nMaster: receiving from task %d!\n", i);
107       MPI_Recv(A.begin(), A.size(), MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
108                MPI_STATUS_IGNORE);
109       B.rows(i * x_points, (i + 1) * x_points - 1) = A;
110     }
111     B.save("data/outputs/A.dat", raw_binary);
112     parameters(0) *= num_tasks;
113     parameters.save("data/outputs/pmts.dat", raw_ascii);
114   }
115
116   MPI_Finalize();
117
118   return 0;
119 }
```

Listing D.1: solver.cpp: código fonte que realiza os cálculos da função de um paraboloide com o auxílio da MPI.

```
#!/usr/bin/python2.7
#!-*- coding: utf8 -*-

import numpy as np
import matplotlib.pyplot as plt

# Loading data
A = np.fromfile('data/outputs/A.dat', dtype=float)
[x_points, x_ofst, xi, y_points, y_ofst,
    yi] = np.loadtxt('data/outputs/pmts.dat')

# Preparing data for plotting
X = np.linspace(xi, xi + x_points * x_ofst, num=int(x_points))
Y = np.linspace(yi, yi + y_points * y_ofst, num=int(y_points))
A = A.reshape(int(y_points), int(x_points))

[B, C] = np.meshgrid(X, Y)

# Preparing plot
fig, ax = plt.subplots()
CS = ax.contourf(B, C, A, 20, cmap='RdGy')
ax.clabel(CS, inline=False, fontsize=10)
ax.set_title('z = 2x^2 + y^2')

ax.plot()
plt.savefig('data/images/A.png')

exit(0)
```

Listing D.2: viewer.py: código fonte que gera a imagem resultante dos cálculos da função de um paraboloide.

Apêndice E

Códigos finais

E.1 Códigos serial final falso

E.1.1 Propagação de onda em uma dimensão

```
#include <iostream>
#include <stdio.h>
#include <mpi.h>
#include <armadillo>
#include <queue>
#include <pthread.h>

using namespace std;
using namespace arma;

#define MASTER       0
#define NUM_THREADS 6
#define PI           3.14159265359

/** Args
 * @brief
 *  x_points - number of points in x axis
    x_b      - beggining of the x axis
    x_t      - end of the x axis
    x_w      - value in axis to begin the wave
    t_t      - end of the t(ime) axis
    t_w      - value of the end of the t axis
    freq     - value of the frequency of the wave
 */

typedef struct kit_t {
    int thread_id;
    mat *A        ;
    int x_pt_start,
        x_pt_end  ,
        i         ;
    float x_j,
          t_i,
          x_ofst;
} KIT_t;

// some global variables to be used in the calculations
float A,
      R,
      t_w/*ave */,
      x_w/*ave */,
      pi2_freq2,
      x_ofst_2,
      t_ofst_2,
      termA;

float fxt(float x, float t) {
```

```
48      float Dx  = x - x_w;
49      float Dx2 = Dx * Dx;
50      float Dt  = t - t_w;
51      float Dt2 = Dt * Dt;
52
53      float result = ((1. - 2. * pi2_freq2 * Dt2) * exp(-pi2_freq2 * Dt2)) * \
54                     ((1. - 2. * pi2_freq2 * Dx2) * exp(-pi2_freq2 * Dx2));
55
56      return result;
57  }
58
59  void *eval(void *void_kit) {
60      KIT_t *kit = (KIT_t *) void_kit;
61
62      int thread_id  = kit->thread_id ;
63      mat *A         = kit->A         ;
64      int x_pt_start = kit->x_pt_start;
65      int x_pt_end   = kit->x_pt_end  ;
66      float x_j      = kit->x_j       ;
67      float t_i      = kit->t_i       ;
68
69      float x_ofst   = kit->x_ofst    ;
70      int i          = 1              ;
71
72      for (int j = x_pt_start; j <= x_pt_end; j++) {
73          (*A)(i + 1, j) = termA * ((*A)(i, j - 1) - 2. * (*A)(i, j) + \
74              (*A)(i, j + 1)) - (*A)(i - 1, j) + 2. * (*A)(i, j) + t_ofst_2 * \
75              fxt(x_j, t_i);
76          x_j += x_ofst;
77      }
78
79      pthread_exit(NULL);
80  }
81
82  int main(int argc, char *argv[]) {
83
84      // Variables for working with MPI
85      int task_id,   // task NUM - the identification of a task
86          num_tasks, // number of tasks
87          rc,        // for returning code
88          i;
89
90      // initiating MPI, starting tasks and identifying them
91      MPI_Init(&argc, &argv);
92      MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);
93      MPI_Comm_rank(MPI_COMM_WORLD, &task_id);
94
95      int   x_points   , t_points   ;
96      float x_ofst     , t_ofst, x_t/*otal*/;
97      float x_b/*egin*/, t_t/*otal*/;
98
99      // auxiliary variables
100     float freq,
101           freq2,
102           pi2 = PI * PI;
103
104     // Creating objects for conversion of arguments
105     stringstream convert0(argv[1]);
106     stringstream convert1(argv[2]);
107     stringstream convert2(argv[3]);
108     stringstream convert3(argv[4]);
109     stringstream convert4(argv[5]);
110     stringstream convert5(argv[6]);
111     stringstream convert6(argv[7]);
112
113     // Putting arguments on variables
114     convert0 >> x_points;
115     convert1 >> x_b;
116     convert2 >> x_t;
117     convert3 >> x_w;
118     convert4 >> t_t;
119     convert5 >> t_w;
120     convert6 >> freq;
```

```
121
122     // finding some different place to begin the wave (if this task is not MASTER)
123     if (task_id != MASTER) {
124         if (x_w + .5 * task_id >= x_b || x_w + .5 * task_id <= x_t) {
125             x_w += .5 * task_id;
126         } else {
127             float x_w_temp;
128             do {
129                 x_w_temp = x_w + rand() / rand();
130             } while (x_w_temp < x_b || x_w_temp > x_t);
131             x_w = x_w_temp;
132         }
133     }
134
135     x_ofst    = (x_t - x_b) / x_points;
136     t_ofst    = .5 * x_ofst;    // respecting Nyquist theorem
137     t_points  = (int) t_t / t_ofst;
138     freq2     = freq * freq;
139     pi2_freq2 = pi2 * freq2;
140
141     printf("Task %d's specs report\n"
142             "X points     : \t%d   \n"
143             "T points     : \t%d   \n"
144             "X offset     : \t%f   \n"
145             "T offset     : \t%f   \n"
146             "X total      : \t%f   \n"
147             "X total      : \t%f   \n"
148             "X wave's pic: \t%f   \n"
149             "X wave's pic: \t%f   \n",
150         task_id                     ,
151         x_points                    ,
152         t_points                    ,
153         x_ofst                      ,
154         t_ofst                      ,
155         x_points * x_ofst           ,
156         t_t                         ,
157         x_w                         ,
158         t_w                         );
159
160     // creating the matrix for calculations and a row vector for saving
161     // parameters
162     mat B(t_points, x_points);
163     B.fill(0.);
164     mat A(3, x_points);
165     A.fill(0.);
166     rowvec parameters(6);
167
168     // Storing parameters in a vector for a file
169     parameters(0) = x_points;
170     parameters(1) = x_ofst;
171     parameters(2) = x_b;
172     parameters(3) = t_t;
173     parameters(4) = t_points;
174     parameters(5) = num_tasks;
175
176     // Calculating function
177     float x_j = x_b;
178     float t_i = 0.;
179
180     // setting some calculation variables
181     x_ofst_2 = x_ofst * x_ofst;
182     t_ofst_2 = t_ofst * t_ofst;
183     termA = t_ofst_2 / x_ofst_2;
184
185     // creating threads
186     pthread_t threads[NUM_THREADS];
187     pthread_attr_t attr;
188
189     // creating kits for the threads
190     KIT_t kits[NUM_THREADS];
191
192     // initializing ptheads attribute and setting threads as joinable
193     pthread_attr_init(&attr);
```

```
194     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
195
196     // determining how many points each thread will have to take care of
197     int x_pts_per_thread = (x_points - 2) / NUM_THREADS;
198
199     // distributing information for kits
200     for(int i = 0; i < NUM_THREADS - 1; i++) {
201         kits[i].thread_id  = i                                          ;
202         kits[i].A          = &A                                         ;
203         kits[i].x_pt_start = i * x_pts_per_thread + 1                   ;
204         kits[i].x_pt_end   = i * x_pts_per_thread + x_pts_per_thread;
205         kits[i].x_j        = x_ofst * x_pts_per_thread * i              ;
206         kits[i].t_i        = t_ofst                                     ;
207         kits[i].x_ofst     = x_ofst                                     ;
208     }
209     kits[NUM_THREADS - 1].thread_id  = NUM_THREADS - 1                          ;
210     kits[NUM_THREADS - 1].A          = &A                                       ;
211     kits[NUM_THREADS - 1].x_pt_start = (NUM_THREADS - 1) * x_pts_per_thread + 1;
212     kits[NUM_THREADS - 1].x_pt_end   = x_points - 2                             ;
213     kits[NUM_THREADS - 1].x_j        = x_ofst * x_pts_per_thread * \
214         (NUM_THREADS - 1);
215     kits[NUM_THREADS - 1].t_i        = t_ofst                                   ;
216     kits[NUM_THREADS - 1].x_ofst     = x_ofst                                   ;
217
218     // Iterating in the time
219     printf("Task %d: calculation is beggining!\n", task_id);
220     for (int i = 1; i < t_points - 1; i++) {
221         // creating threads
222         for (long j = 0; j < NUM_THREADS; j++) {
223             kits[j].t_i = t_ofst * i;
224             if (pthread_create(&threads[j], &attr, eval, (void *) &kits[j])) {
225                 printf("Error on creating thread %ld\n", j);
226                 exit(1);
227             }
228         }
229
230         // joining threads
231         void *status;
232         for (long j = 0; j < NUM_THREADS; j++) {
233             int returnCode;
234             returnCode = pthread_join(threads[j], &status);
235             if (returnCode) {
236                 printf("Error on joining thread %ld\nThread returned %d", j,
237                     returnCode);
238             }
239         }
240
241         // we need always to save the last line of B before 'deleting' it
242         B.row(i - 1) = A.row(0);
243         A.rows(0, 1) = A.rows(1, 2);
244         A.row(2).zeros();
245     }
246     printf("Task %d: calculation is over!\n", task_id);
247
248     if (task_id != MASTER) { // if it's not the master
249         // sending matrix's beggining pointer to master
250         printf("\nTask %d: sending!\n", task_id);
251         MPI_Send(B.begin(), B.size(), MPI_DOUBLE, MASTER, 0, MPI_COMM_WORLD);
252     } else { // if it's MASTER
253         // saving parameters
254         parameters(0) *= num_tasks;
255         parameters.save("data/outputs/pmts.dat", raw_ascii);
256         // creating auxiliary matrix for MPI_Recv()
257         mat C(size(B));
258         // resizing B for receiving data from the other tasks
259         B.resize(t_points, num_tasks * x_points);
260         // receiving data from other tasks
261         for (int i = 1; i < num_tasks; i++) {
262             printf("Master: receiving from task %d!\n", i);
263             MPI_Recv(C.begin(), C.size(), MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
264                     MPI_STATUS_IGNORE);
265             B.cols(i * x_points, (i + 1) * x_points - 1) = C;
266         }
```

```
267         B.save("data/outputs/A.dat", raw_binary);
268     }
269
270     MPI_Finalize();
271
272     return 0;
273 }
```

Listing E.1: solver.cpp: resolve a equação da onda unidimensional com o auxílio das APIs Pthreads e MPI

```
1  #!/usr/bin/python2.7
2  #!-*- coding: utf8 -*-
3
4  import numpy as np
5  import matplotlib.pyplot as plt
6
7  # Loading data
8  A = np.fromfile('data/outputs/A.dat', dtype=float)
9  [x_points, x_ofst, xi, tt, t_points, n_tasks] = \
10     np.loadtxt('data/outputs/pmts.dat')
11
12 # Preparing data for plotting
13 X = np.linspace(xi, xi + x_points / n_tasks * x_ofst, num=int(x_points / n_tasks))
14 T = np.linspace(0., tt, num=int(t_points))
15 A = A.reshape(int(x_points), int(t_points))
16
17 # TODO: remove this line
18 [B, C] = np.meshgrid(X, T)
19
20 # Preparing plot
21 for i in range(0, int(n_tasks)):
22     M = max(abs(A[int(i * (x_points / n_tasks)):int((i + 1) * (x_points / n_tasks))].min()),
23             abs(A[int(i * (x_points / n_tasks)):int((i + 1) * (x_points / n_tasks))].max()))
24
25     fig, ax = plt.subplots()
26     ax.set_title('Wave in a string')
27     CS = ax.contourf(C, B, A[int(i * (x_points / n_tasks)):int((i + 1) * (x_points / n_tasks))
       ].transpose(),
28         52, cmap='seismic', vmin=-M, vmax=M)
29     ax.clabel(CS, inline=False, fontsize=10)
30     plt.xlabel('T')
31     plt.ylabel('X')
32     cbar = fig.colorbar(CS)
33
34     ax.plot()
35     plt.savefig('data/images/A' + str(i) + '.png')
```

Listing E.2: viewer.py: script que cria a imagem da propagação da onda no plano $v \times t$.

E.2 Propagação de ondas em duas dimensões

E.2.1 Código principal

```
1  #include "util.h"
2  #include "_2DWave.h"
3
4  #define SPECS_DIR   "./specs/"
5  #define VEL_DIR     "./velocity/"
6  #define SNAPS_DIR   "./snaps/"
7  #define TRACES_DIR  "./traces/"
8
9  /*
10  *     This program aims to solve the bidimensional wave equation
11  *     by the finite differences method. This methods will be im-
12  *     plemented with the help of Eigen library, that provides the
13  *     necessary tools of the Linear Algebra, mainly the vectors
14  *     and matrices thats will be used to store and manipulate data,
15  *     as well the methods that are necessary to this.
16  */
```

```
17
18 int main () {
19
20     // Getting external data
21     ifstream wf ( SPECS_DIR "wOut.dat", ios::in | ios::binary);
22     ifstream vf ( SPECS_DIR "vOut.dat", ios::in | ios::binary);
23     ifstream ifl( SPECS_DIR "iOut.dat", ios::in | ios::binary);
24
25     vector<interface> it;
26     vector<velocity> vl;
27     ifstream nInt( SPECS_DIR "nInt.dat", ios::in);
28     int N;
29     nInt >> N;
30     for (int i = 0; i < N; i++) {
31         interface auxI(0., 0.);
32         velocity auxV(0., 0., 0.);
33         auxI.deserialize(&ifl);
34         auxV.deserialize(&vf);
35         it.push_back(auxI);
36         vl.push_back(auxV);
37     }
38     velocity auxV(0., 0., 0.);
39     auxV.deserialize(&vf);
40     vl.push_back(auxV);
41
42     _2DWave wv(0., 0., 0., 0., 0., 0., 0., 0., 0., 0.);
43
44     wv.deserialize(&wf);
45
46     wf.close(); vf.close(); ifl.close();
47
48     // Creating velocities matrix
49     mat v;
50     v.load( VEL_DIR "velocity.dat", raw_ascii);
51
52     // Compute 1/v^2 only once for all steps
53     for (int i = 1; i < wv.getMx() - 1; i++) {
54         for (int j = 1; j < wv.getNy() - 1; j++) {
55             v(i,j) = 1.0 / ( v(i,j) * v(i,j) );
56         }
57     }
58
59     // Creating arrays for space dimensions X e Y and for time
60     vec X = linspace<vec>(0., wv.getLx(),   wv.getMx());
61     vec Y = linspace<vec>(0., wv.getLy(),   wv.getNy());
62     vec T = linspace<vec>(0., wv.getTMax(), wv.getOt());
63
64     // Saving arrays of dimensions in space
65     X.save( SPECS_DIR "X.dat", raw_ascii);
66     Y.save( SPECS_DIR "Y.dat", raw_ascii);
67     T.save( SPECS_DIR "T.dat", raw_ascii);
68
69     nInt >> N; // reusing N
70     queue <int> numSnaps;
71     if (N > 0) { // get the frames's numbers that must be saved as
72                  // snapshots
73         int h = (int) wv.getOt() / N;
74         for (int i = 1; h * i < wv.getOt(); i++) { numSnaps.push(h * i); }
75     }
76
77     // Receiving data about receivers
78     int nRecv;
79     double offset_Recv;
80     nInt >> nRecv;
81     nInt >> offset_Recv;
82
83     nInt.close();
84
85     // Matrix to armazenate traces
86     mat traces((int) wv.getOt(), nRecv);
87     traces.fill(0.);
88
89     // the number of points between receivers
```

```
90      int offset_Recv_int = wv.getMx() / nRecv;
91
92      // Creating 3 matrices for application of the Finite Difference Method (FDM)
93      mat U1((int) wv.getMx(), (int) wv.getNy());
94      mat U2((int) wv.getMx(), (int) wv.getNy());
95      mat U3((int) wv.getMx(), (int) wv.getNy());
96
97      // Filling the matrices with zeros
98      U1.fill(0.);
99      U2.fill(0.);
100     U3.fill(0.);
101
102     // Creating a queue for administrating the arrays of FDM
103     queue <mat> U;
104     U.push(U1); U.push(U2); U.push(U3);
105
106     double dt2    =         wv.getDt() * wv.getDt();
107     double dt2dx2 = dt2 / ( wv.getDx() * wv.getDx() );
108     double dt2dy2 = dt2 / ( wv.getDy() * wv.getDy() );
109
110     for (int k = 1; k < wv.getOt() - 1; k++) {
111
112         // Getting the matrices from the queue
113         U1 = U.front(); U.pop(); // t + 1
114         U2 = U.front(); U.pop(); // t
115         U3 = U.front(); U.pop(); // t - 1
116
117         // Doing FDM calculations
118         for (     int i = 1; i < wv.getMx() - 1; i++) {
119             for ( int j = 1; j < wv.getNy() - 1; j++) {
120                 U1(i,j) =  v(i,j) * ( dt2dx2 * ( U2(i+1,j) - 2.0*U2(i,j) + U2(i-1,j) )
121                                     + dt2dy2 * ( U2(i,j+1) - 2.0*U2(i,j) + U2(i,j-1) ) )
122                         + dt2 * wv.evaluateFXYT( X(i), Y(j), T(k) )
123                         + 2.0 * U2(i,j) - U3(i,j);
124             }
125         }
126
127         // Registering traces
128         for (int ii = 1; ii * offset_Recv_int < size(U1)(0) && ii < nRecv; ii++) {
129             traces(k, ii) = U1(ii * offset_Recv_int, 1);
130         }
131
132         // if frame k is one of the required for the snaps
133         if (N > 0 && k == numSnaps.front()) {
134             ostringstream oss;
135             int save = numSnaps.front();
136             oss << SNAPS_DIR << setfill('0') << setw(5) << save << ".dat";
137             numSnaps.pop(); numSnaps.push(save);
138             U1.save(oss.str(), raw_ascii);
139         }
140
141         // Putting the matrices again in the queue
142         U.push(U3); U.push(U1); U.push(U2);
143     }
144
145     vec d(7);
146     d(0) = (int) wv.getMx();
147     d(1) = (int) wv.getNy();
148     d(2) = (int) wv.getOt();
149     d(3) = N;
150     d(4) = wv.getLx();
151     d(5) = wv.getLy();
152     d(6) = nRecv;
153
154     d.save( SPECS_DIR "output.dat", raw_ascii);
155
156     traces.save( TRACES_DIR "traces.dat", raw_ascii);
157
158     return 0;
159
```

```
160 }
```

Listing E.3: main-reflect-bound.cpp: código fonte que realiza os cálculos da propagação da onda bidimensional com o auxílio das APIs Pthreads e MPI.