RICE UNIVERSITY

Framework Design and Implementation of Finite Difference Based Seismic Simulations

by

Igor S. Terentyev

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Arts

Approved, Thesis Committee:

William Symes, Professor
Computational and Applied Mathematics

Matthias Heinkenschloss, Professor
Computational and Applied Mathematics

Tim Warburton, Assistant Professor
Computational and Applied Mathematics

Mark Embree, Associate Professor
Computational and Applied Mathematics

Houston, Texas

April 2008

Abstract

Framework Design and Implementation of Finite Difference Based Seismic Simulations

by

Igor S. Terentyev

Recent advances in high-performance computing have revived interest of the seismic community in large-scale partial differential equations (PDE) solvers. In this thesis, I design and implement a software framework for solving time dependent PDE in simple domains using finite difference (FD) methods. The framework is designed for parallel computations on distributed and shared memory computers, thus allowing for efficient solution of large-scale problems. The framework provides tools for description of FD schemes using stencil information. Once the stencil is supplied, the framework ensures automated data exchange between processors. This automated data exchange allows a user to add FD schemes without knowledge about underlying parallel infrastructure. The code is written in ISO C language and uses MPI and OpenMP for parallelization. I used the framework to implement a staggered second-order in time and second/fourth-order in space FD schemes for the acoustic wave equation. The acoustic solver provides perfectly matched layer and free surface boundary conditions.

Acknowledgements

I would like to thank my scientific advisor Prof. William Symes for his guidance, advice, enthusiasm and constant readiness to devote his time to answer my questions and provide all kinds of help (from code debugging to thesis text proof-reading) I needed to complete this thesis.

I would like to express my deepest gratitude to Dr. Tanya Vdovina for her invaluable help with my thesis work.

Many thanks to Dr. Janice Hewitt for teaching me how to improve my text writing and presentation skills.

Contents

Abstract ii

Acknowledgements iii

List of Figures v

List of Tables vi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Finite-Difference Methods . . . 3
- 1.3 Hardware and Software . . . 6

2 Software Framework 10
- 2.1 Introduction . . . 10
- 2.2 Object-oriented Approach . . . 12
- 2.3 Auxiliary Constructs . . . 13
- 2.4 Model Problem . . . 15
- 2.5 Base Structures: RARR, RDOM . . . 16
- 2.6 Timestep Function . . . 21
- 2.7 Automated Data Exchange . . . 22
  - 2.7.1 Stencil Datatype . . . 23
  - 2.7.2 Ghost Areas Computation . . . 25
- 2.8 Timestep Loop and Terminators . . . 27

3 Numerical Experiments 29

4 Conclusions 36

Bibliography 38

List of Figures

2.1 Two-dimensional staggered grid. Circle stands for pressure grid point, plus and cross refer to horizontal and vertical velocity grid points respectively. 17
2.2 Virtual arrays. 19
2.3 Stencil example. 23
2.4 Ghost area computation example. 26
2.5 Virtual array layout. 27

3.1 ADA speedups. Number of processors is shown on the horizontal axis, $\log_2 S_p$ is presented on the vertical axis, where S_p is a speedup on p processes; 2-2 refers to the second order in time and space FD scheme, 2-10 refers to the second order in time and tenth order in space FD scheme, Config. 1 and Config. 2 represent one process per node and one process per core configurations respectively. 33
3.2 SGI speedups. Number of processors is shown on the horizontal axis, $\log_2 S_p$ is presented on the vertical axis, where S_p is a speedup on p processes; 2-2 refers to the second order in time and space FD scheme, 2-10 refers to the second order in time and tenth order in space FD scheme, Config. 1 and Config. 2 represent one process per node and one process per core configurations respectively. 34

List of Tables

3.1 ADA timings. NP stands for number of MPI processes, 2-2 refers to the second order in time and space FD scheme, 2-10 refers to the second order in time and tenth order in space FD scheme, configuration 1 and configuration 2 represent one process per node and one process per core configurations respectively. T is a walltime (in seconds), S is a speedup. 31
3.2 SGI timings. NP stands for number of MPI processes, 2-2 refers to the second order in time and space FD scheme, 2-10 refers to the second order in time and tenth order in space FD scheme, configuration 1 and configuration 2 represent one process per node and one process per core configurations respectively. T is a walltime (in seconds), S is a speedup. 32
3.3 ADA weak speedup tests. 35

Chapter 1

Introduction

Numerical simulations are essential for understanding of seismic wave propagation in oil-and-gas exploration, prediction of tectonic events and other geoscience applications that require knowledge of the subterranean structure. The goal of this work is to design, implement, and test an efficient parallel finite difference (FD) framework for simulation of seismic wave propagation with applications in reflection seismology. The choice of finite-difference methods is determined by the fact that reflection seismology generates large-scale problems that involve terabytes of data, and FD methods achieve a reasonable balance between computational efficiency and accuracy.

1.1 Motivation

The information about interior properties of the Earth is collected by sending seismic waves into the ground and recording a portion of these waves reflected back to the surface due to the highly heterogeneous nature of the subsurface. Inversion of the recorded data allows geoscientists to reconstruct the structure of the Earth's interior. An inherent part of the inverse solver is multiple solving of the forward wave propa-

gation problem. Therefore, being able to solve the forward wave equation efficiently and accurately is crucial for inversion process.

In addition to providing a principal tool for inversion of seismic data, numerical simulation of seismic wave propagation can be used to test experimental design. For example, Regone [2] used 3D wave propagation model as an alternative to highly expensive real experiments and demonstrated the advantages of wide-azimuth towed streamer (WATS) acquisition over the traditionally employed sparse acquisition. This research led to acceptance of WATS, which was a significant change in the acquisition technology.

In this thesis I develop a parallel FD software framework. The framework provides an implementation of several numerical algorithms, tools for assessing their performance, and libraries that can be used as a basis for implementing new algorithms with a little programming effort. The software is intended to be used as

– a tool for studying seismic wave propagation in large-scale domains,

– a test-bed for different numerical methods,

– a core for an inverse solver,

– an instrument for testing experimental design.

The choice of methods and design solutions for this software is mainly determined by the applications described above and common practices in the field. One of the major requirements imposed on the framework is to ensure that it is portable, i.e. works the same on various hardware platforms. Portability is dictated by the fact that the framework is intended to be open-source and, therefore, is likely to be used on different computing systems. In addition to being portable, the code has to be reusable and easily modifiable. These requirements will allow for adding of

new methods with a minimal programming effort. Another important requirement is to produce a parallel code. The necessity to run simulations in parallel follows from the size of problems that come from reflection seismology (both memory and computational time demands exceed capabilities of a stand-alone desktop). Finally, I use FD methods, since they are the most widely accepted in reflection seismology as a relatively accurate and efficient way to solve the wave equation.

A lot if software for seismic numerical simulations exist. However, there are no known open-source codes that are (a) parallel, (b) capable of supporting variable accuracy, (c) flexible enough for adding new methods. My software is intended to fill this gap.

1.2 Finite-Difference Methods

Main challenges associated with developing an efficient and accurate wave propagation simulator come from the input data. Reflection seismology generates input data that can easily run into terabyte range and beyond resulting in an extremely computationally intensive problems. The highly heterogeneous nature of these data impacts the accuracy of the solution. FD methods have become an industry standard in reflection seismology, since they provide good computational speed for required accuracy. In addition, physical domains that result from geoscience applications are characterized by simple geometries perfectly suitable for discretization by FD grids. Finally, in most cases FD methods are relatively easy to implement (compared, for instance, to finite element/volume methods).

Computational framework described in this thesis is aimed at explicit FD methods. In explicit FD methods, the solution at the next time step can be directly computed from the solution determined at the previous times. Implicit FD methods require

solving a linear system at each time step. Although implicit schemes generally possess better stability properties and impose no constraints on the size of the time step, they are usually much more computationally intensive than explicit schemes. As a result, application of implicit methods to large-scale problems becomes prohibitive and is usually avoided. In the discussion that follows, I focus on explicit methods.

The first theoretical results for FD methods go back to 1928 and the famous paper of Courant, Friedrichs, and Lewy [5], in which the authors established a necessary condition for convergence of FD methods. They showed that the time step must be proportional to the space step in order for the FD solution to converge to the solution of the partial differential equation (PDE). General theory for FD methods for the approximation of various kinds of initial-value problems is given in [7]. The book contains material necessary to understand the concepts of stability, consistency, and convergence, provides description and analysis of the most widely used FD methods, and highlights the difficulties associated with approximation of PDEs by FD methods.

One such difficulty is a grid or numerical dispersion, a phenomenon that produces parasitic waves around the solution. If numerical grid is too coarse, waves generated by the discrete system have different velocity from the velocity of the waves produced by the continuous system. Numerical dispersion depends on the frequency of the wave and the direction of wave propagation. Waves that correspond to higher frequencies and/or propagate along the grid axes are subject to bigger errors in the velocity.

The effect of numerical dispersion can be reduced by ensuring that grid spacing is sufficient to resolve the minimum wavelength. The minimum number of grid points per wavelength is different for different FD schemes. For example, second order in space and time (2-2) scheme requires at least ten grid points per wavelength [1]. This requirement on spatial sampling imposes a serious constraint on the size of the problem that can be handled numerically and limits the application of low-order

schemes. For example, typical size of the domain of interest in reflection seismology is at least 100 wavelengths is each spatial direction. If 10 grid points per wavelength is needed to minimize dispersion, we arrive at the discrete problem of size 10^9 grid points in three dimensions. Since low-order schemes take at least 20 floating point operations (FLOP) per grid point, the number of FLOP at each time step is $2 \cdot 10^{10}$. At least 10^4 time steps is usually taken per seismic survey and as many as $5 \cdot 10^4$ surveys may be considered. Therefore, the total number of FLOP is 10^{19}, which would result in computational time of 10^{10} seconds on a 1 GFLOPS desktop, which is approximately 300 years.

Staggered-grid FD schemes usually require fewer grid points per wavelength than regular-grid schemes of the same order. The framework described in this thesis is aimed at both staggered- and regular-grid schemes. Staggered schemes turned out to be particularly advantageous when applied to the elastic wave equation for several reasons. First, staggered schemes are stable for all values of Poisson's ratio, while regular-grid schemes fail to provide a stable approximation for the materials characterized by high Poisson's ratio [9]. Futher, the dispersion relation is also independent of the Poisson's ratio. This insensibility to the Poisson's ratio makes staggered schemes ideal for mixed acoustic-elastic media typical for marine exploration problems. Finally, staggered-grid approach allows avoidance of differentiation of the material parameters making the resulting approximation more accurate and computationally efficient than a regular-grid approximation. A second order in time and space staggered scheme for the elastic wave equation was first presented by Virieux in [8] and [10] along with stability and dispersion analysis and numerical experiments that validated the theoretical results. Fourth-order extension of this approach was developed by Levander in [6]. His analysis indicates that the fourth-order approximation requires only five grid points per wavelength to minimize the effect of dispersion.

In general, increasing the order of the numerical scheme leads to decreasing dispersion in case of both regular and staggered grids. The framework allows for implementation of schemes of any order in time and space. Higher-order methods were studied by many authors; see [4], [3] and references cited therein. However, the effort is usually concentrated on higher-order in space methods, since a standard Taylor series approach to construction of higher-order in time FD approximations usually leads to unstable approximations. Stable higher-order in time schemes can be derived using the modified equation approach which consists of replacing time derivatives with spacial derivatives and discretizing the latter [4]. In one of the first papers on higher-order FD schemes [1], the authors presented a second order in time and fourth order in space (2-4) regular-grid scheme that requires half as many grid points per wavelength as correspondent 2-2 scheme. This reduction in spatial sampling leads to significant savings in FLOP and computer memory that justify the cost increase associated with application of higher-order finite differences. Nonetheless, the savings are not sufficient to ensure the solution of large scale problems that result from reflection seismology in feasible computational time. Namely, if the number of grid points per wavelength is reduced from ten to five, the computational time for a single desktop in the example above reduces from 300 to 20 years. Therefore, given the current state of computer technology, the only feasible way to model large-scale three-dimensional wave propagation is to employ high performance computers.

1.3 Hardware and Software

The code described in this thesis is designed to work on distributed and/or shared memory systems that have ISO C90 compiler with Message Passing Interface (MPI) library and, possibly, OpenMP language extension.

Distributed memory system or cluster is a collection of nodes based on commodity processors (CPU) and interconnected by fast local area networks. Each node owns CPU(s) and RAM and cannot access other nodes' memory directly. A node that consists of several cores which use the same memory is referred to as a shared memory system. A cluster with multicore/multiprocessor nodes can be viewed as a hybrid (distributed+shared memory) system.

Most of today's clusters use Unix/Linux operating systems and can be programmed with standardized computing languages, for example, FORTRAN, C, C++. At a software level, communication between processors is implemented via language extensions and/or libraries. MPI library provides convenient FORTRAN, C and C++ bindings and is now supported by most of the cluster platforms. Therefore, the software written in the listed above languages with MPI is portable. Other advantages that make MPI-based parallelization approach favorable are:

- MPI has long history (20 years including similar software tool PVM),
- MPI will be around for a long time (all new platforms and roadmaps),
- MPI is very scalable (more than 100K cores),
- MPI supports hybrid models.

There are several software technologies that allow for shared memory programming: POSIX Threads library for C/C++ in UNIX-like environments, specialized languages (e.g. Cilk), Intel Thread Building Blocks library for C++, OpenMP compiler extensions for FORTRAN, C, C++, etc. Among these, I chose OpenMP since (a) it allows for a simple parallel loop implementation essential for FD-based computations, (b) it is portable: compilers that do not support OpenMP ignore its directives.

Some of the HPC systems (or cluster nodes) can be equipped with accelerators. Accelerators are dedicated processors capable of performing specialized computations very efficiently. Although a variety of accelerating technologies (GPGPUs, FPGAs, etc.) is currently available, the uniform standard for programming these accelerators has not been developed. As a result, special programming tools are required for each type of accelerating hardware. This lack of standardization makes developing portable accelerator software currently impossible. Since portability requirement is one of the major requirements for the framework, I do not consider support of accelerators in this work.

When MPI-based program runs on a cluster, the same copy of the program is executed on each processing unit. Program copy can identify itself by acquiring a unique index called rank. Each copy performs specific actions based on its rank. With the introduction of the multicore processors, more than one copy of the program is executed on one processor. I will use the term Processing Element (PE) to identify single unit running one copy of the program. For example, PE can be a processor, a core, or an OS process (if several copies are running on a single core).

The most efficient approach to parallelization of finite difference methods is domain decomposition. In the domain decomposition, each PE assumes ownership over a portion of the physical domain and is responsible for computing the solution over this portion. PEs which contain adjacent portions of the domain are called neighbors. The solution at points located near the boundaries of the subdomain depends on the information stored on the neighboring PEs. Therefore, each PE needs to exchange data with its neighbors and to allocate memory for storage of the additional information. The amount of the data that needs to be exchanged depends on the finite difference scheme and partition of the domain. My framework allows for automatic exchange and allocation of memory based on the size of the finite-difference stencil

provided by the user.

Chapter 2

Software Framework

2.1 Introduction

This chapter describes the architecture of the software framework: concepts, datatypes and methods, and workflow. My main goal in this work is to design a forward simulator software which would allow for easy implementation of parallel FD schemes. I achieve this goal by implementing a set of tools, i.e., specific data structures and methods, that provide functionalities common for all parallel FD schemes. These tools have been carefully designed and thoroughly tested and should be used as building blocks for implementation of numerical models and methods described in Chapter 1.

The intent of the framework is to spare the user the necessity of implementing underlying and unrelated to the numerical model/method functionalities, such as data storage, data exchange between processors, parameter input and output, etc. Implementation of these functionalities is often more complicated and error-prone than the implementation of the numerical method itself. Therefore, by providing well-designed and well-tested software framework and tools targeted for parallel FD schemes, I greatly reduce user's effort and time required to architecture, implement,

and produce error-free new numerical models and schemes.

Tools provided by the framework can be subdivided into several groups. One group consists of datatypes, methods, constants and variables that play an auxiliary role and are used by different parts of the framework as well as by a user in his implementation of a particular model. This group includes error codes, primitive datatypes (such as floating point (real) type and multiindex type), global constants (such as maximum number of space dimensions), input/output (I/O) functions (such as input data parser).

Second group is a set of datatypes that naturally describe the components of the uniform-grid FD method. These are multidimensional array and domain (collection of arrays). These datatypes provide all necessary functionalities and can be used by a user without modifications or extensions.

Third group is a set of "template" (or abstract in object-oriented language) datatypes that provide some common functionalities but need to be extended for each particular model/method. These include stencil, terminator, and model. The model datatype is the encompassing datatype which contains all the information about a particular numerical method.

Finally, there is a driver. Driver contains the `main()` function. It initializes the model and runs the timestep loop. Currently provided driver is somewhat coupled to the staggered FD scheme, which I implemented, and will require some modifications for other numerical schemes. This is a drawback of the current version of the code, as, ideally, the driver should use abstract methods of the model datatype (implemented in particular models) and be independent of the model realizations.

In order to use the framework a user needs to know how to:

1. extend existing template datatypes (such as stencil, terminator, model),

2. implement model/method-specific functionalities (such as time-step functions, boundary conditions functions),

3. modify the driver.

In the following sections, I discuss each of the groups mentioned above and provide a roadmap for adding a new model/method to the framework. In Section 2.3, I describe basic structures and auxiliary constructs. In Section 2.4, I introduce a simple model and use it to illustrate the issues that a user needs to resolve when adding a new model/method to the framework. Sections 2.5 and 2.6 discuss datatypes that describe multidimensional arrays and timestep functions. Section 2.7 provides an overview of automated data exchange and related framework tools. Finally, in Section 2.8, I discuss the timestep loop and the concept of terminator.

2.2 Object-oriented Approach

The framework datatypes are implemented in the "object-like" fashion. Each datatype serves one particular functionality and is implemented as a C structure that holds corresponding data and methods (functions) that operate on this structure. The pair structure-methods can be considered as an object, i.e., an entity that encapsulates data storage and functionality. All the "object-like" datatypes have methods that perform common tasks. The most important are initialization of the datatype and its deinitialization. Since in C there is no notion of objects and automatically called methods, such as constructor and destructor, it is the responsibility of a user to initialize datatype variable after the declaration and to deinitialize the variable before it goes out of scope. Some datatypes provide functionality that can be extended by a user. Such extensions resemble class inheritance of object-oriented languages.

Datatypes that allow extension contain a `void *spec` pointer. By default (in the base datatype) `spec` pointer is `NULL`. If the user needs to extend the datatype, he uses this pointer to reference the extended datatype.

2.3 Auxiliary Constructs

In this section, I discuss basic datatypes and library functions that are used throughout the framework.

- Real datatype allows to switch between floating point modes in the entire framework based on the compiler variable `DT_REAL` defined in the `utils.h`:

  ```
  #define DT_REAL DT_FLOAT  /* Define real as float. */
  #define DT_REAL DT_DOUBLE /* Define real as double. */
  ```

  In other words, the `real` datatype is an alias for `float` or `double`. Along with the `real` datatype, other various related constants (`REAL_NAN`, `REAL_EPS`, etc.) and types (`IWAVE_MPI_REAL`) are defined in `utils.h`.

- Multi-dimensional integer and real point types represent indices in multi-dimensional arrays and real vectors:

  ```
  typedef int IPNT[RARR_MAX_NDIM];
  typedef real RPNT[RARR_MAX_NDIM];
  ```

  Here `RARR_MAX_NDIM` describes the maximum number of dimensions supported by the framework (typically 3). These point types are supplied with assignment operators.

- Input parser reads input data from a file, string or the command line and decomposes the input into a set of records represented by a datatype `PARARRAY` declared in the `parser.h`. Each record has a structure `key = value` and a user can query data values based on the key values. `PARARRAY` comes with a constructor, destructor, and query functions (e.g., "does the object contain a key?", "how many times is a key encountered?"). Once the `PARARRAY` object has been created and loaded with `key = value` records, the user can call the query functions and get values that correspond to the `key`. If the key is not found or the value cannot not be converted to the requested type, the query returns the error code. If multiple values with the same key are encountered, the last value is returned. The parser recognizes comments. The current comment symbol `"` and separator `=` can be redefined in the `utils.h` file.

- File `utils.h` contains definitions of the following global constants and functions:

  - integer error codes returned by most of the framework functions,
  - maximum number of the dimensions of multidimensional array datatype and maximum number of arrays in the domain datatypes (see Section 2.5 for detail):

    ```
    #define RARR_MAX_NDIM 3
    #define RDOM_MAX_NARR 20
    ```

  - function that returns global output stream:

    ```
    FILE* retreiveOutstream();
    ```

  - functions that return world communicator, rank, and size:

    ```
    MPI_Comm retreiveComm();
    ```

```
int retreiveRank();
int retreiveSize();
```

- other constants and methods used internally by the framework. Since the user does not interact with these constants and methods, we omit their description.

2.4 Model Problem

In this section I present a model problem which I will use to describe the main steps that a user has to make in order to implement a new model/method. Consider the following formulation of the acoustic wave equation in terms of pressure and velocity:

$$\frac{1}{\kappa(\mathbf{x})}\frac{\partial p(t,\mathbf{x})}{\partial t} = -\nabla\cdot\mathbf{v}(t,\mathbf{x}) + f(t,\mathbf{x}), \tag{2.1}$$

$$\rho(\mathbf{x})\frac{\partial \mathbf{v}(t,\mathbf{x})}{\partial t} = -\nabla p(t,\mathbf{x}), \tag{2.2}$$

where $\mathbf{x}\in\mathbb{R}^2$, $t\in[t_0,T]$, κ and ρ denote the bulk modulus and density, respectively, and f is a source of acoustic energy. The second-order in space and time numerical approximation of problem (2.1)–(2.2) is given by the following equations:

$$\begin{aligned} p_{i,j}^{n} &= p_{i,j}^{n-1} - \frac{\kappa_{i,j}\Delta t}{h_x}\left(u_{i+1/2,j}^{n-1/2} - u_{i-1/2,j}^{n-1/2}\right) \\ &\quad - \frac{\kappa_{i,j}\Delta t}{h_y}\left(v_{i,j+1/2}^{n-1/2} - v_{i,j-1/2}^{n-1/2}\right) + \Delta t f_{i,j}^{n-1/2}, \end{aligned} \tag{2.3}$$

$$u_{i+1/2,j}^{n+1/2} = u_{i+1/2,j}^{n-1/2} - \frac{\rho_{i+1/2,j}\Delta t}{h_x}\left(p_{i+1,j}^{n} - p_{i,j}^{n}\right), \tag{2.4}$$

$$v_{i,j+1/2}^{n+1/2} = v_{i,j+1/2}^{n-1/2} - \frac{\rho_{i,j+1/2}\Delta t}{h_y}\left(p_{i,j+1}^{n} - p_{i,j}^{n}\right), \tag{2.5}$$

where Δt is the time step, h_x and h_y denote spatial steps in x- and y-directions, respectively, and grid values are defined as follows (see Figure 2.1):

$$p^n_{i,j} = p(t_0 + \Delta t n, x_0 + h_x i, y_0 + h_y j), \tag{2.6}$$

$$u^{n+1/2}_{i+1/2,j} = u(t_0 + \Delta t(n + 1/2), x_0 + h_x(i + 1/2), y_0 + h_y j), \tag{2.7}$$

$$v^{n+1/2}_{i,j+1/2} = v(t_0 + \Delta t(n + 1/2), x_0 + h_x i, y_0 + h_y(j + 1/2)). \tag{2.8}$$

FD method (2.3)–(2.5) provides evolution equations that trace physical states (2.6)–(2.8) from one point in time to another. Numerical implementation of such a FD scheme is based on two major concepts: data storage and the timestep function. The timestep function is represented by `TIMESTEP_FUN` type, described in Section 2.6. As indicated by equations (2.3)–(2.5), an object of type `TIMESTEP_FUN` depends on the physical states at previous time iterations and input data. Physical variables and input data are stored in the multidimensional arrays described by `RARR` datatype. The number of physical state arrays and input data arrays depends on a particular problem and an FD method used to discretize the problem. Therefore, for a particular model a user determines the number of data arrays used in this model. These arrays constitute the domain described by `RDOM` type, which a user also needs to define. In the following section, I explain how to create objects of type `RARR` and `RDOM`, introduce their members, and describe a set of library functions that allow to conveniently manipulate such objects.

2.5 Base Structures: RARR, RDOM

The main advantage of the object of type `RARR` provided by this framework is that it allows to create subarrays or virtual arrays. The concept of virtual array is ex-

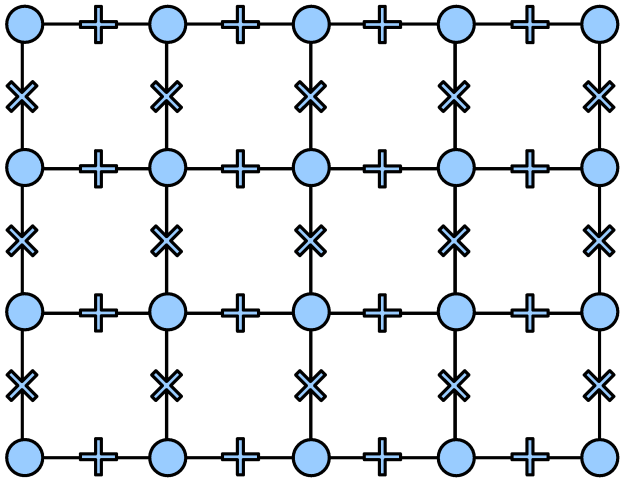

Figure 2.1: Two-dimensional staggered grid. Circle stands for pressure grid point, plus and cross refer to horizontal and vertical velocity grid points respectively.

tremely useful in the context of domain decomposition. In the domain decomposition approach, each PE allocates memory for the solution over its own subdomain. In order to store information from the neighboring PEs, each PE is required to allocate additional arrays of memory (ghost cells) along the subdomain edges. While data in the PE's subdomain area is updated differently from the data in the ghost cell areas, these areas represent parts of the same array. Efficient implementation should avoid unnecessary replication and copying of these data. We achieve this by using virtual arrays. Subarray or virtual array can be manipulated in exactly the same way as a parent array, but rather than allocating its own memory, virtual array refers to a subset of memory allocated by a parent array. Mathematical equivalent of a virtual array is a subset of values of a grid function. Figure 2.2 shows pressure arrays on two neighboring PEs and their virtual subarrays.

Each array is uniquely determined by its size and a pointer to its beginning. The members of an object of type `RARR` are

- `ndim`, which specifies the number of dimensions of a space that holds a physical

variable,

- two pointers `*_s0` and `*_s`,
- and an array of `INFODIM structures _dims[RARR_MAX_NDIM]`.

Each i-th ($i = 0, 1, 2$) entry of `_dims` provides information about i-th dimension of `RARR` array. The structure `INFODIM` has the following fields:

```
int n;
int gs;
int ge;
int n0;
int gs0;
int ge0;
```

All the array fileds ending with `0` refer to the parent or allocated array. Array fileds without `0` in the end describe the virtual subarray of the parent array. Here `n` is the size, and `gs` and `ge` are the global start and end indices along the dimension, respectively. Variables `n`, `gs`, `ge` and `n0`, `gs0`, `ge0` are related by the following equations: $n = ge - gs + 1$, $n0 = ge0 - gs0 + 1$.

The framework provides several options for allocating an object of type `RARR`. In all cases, a user has to provide the dimension of the physical space `ndim` and one of the following:

(a) the coordinates of the beginning of the array `gs0` and its length `n0`

```
int ra_create_s(RARR *arr, int ndim, IPNT gs0, IPNT n0),
```

(b) the coordinates of the end of the array `ge0` and its length `n0`

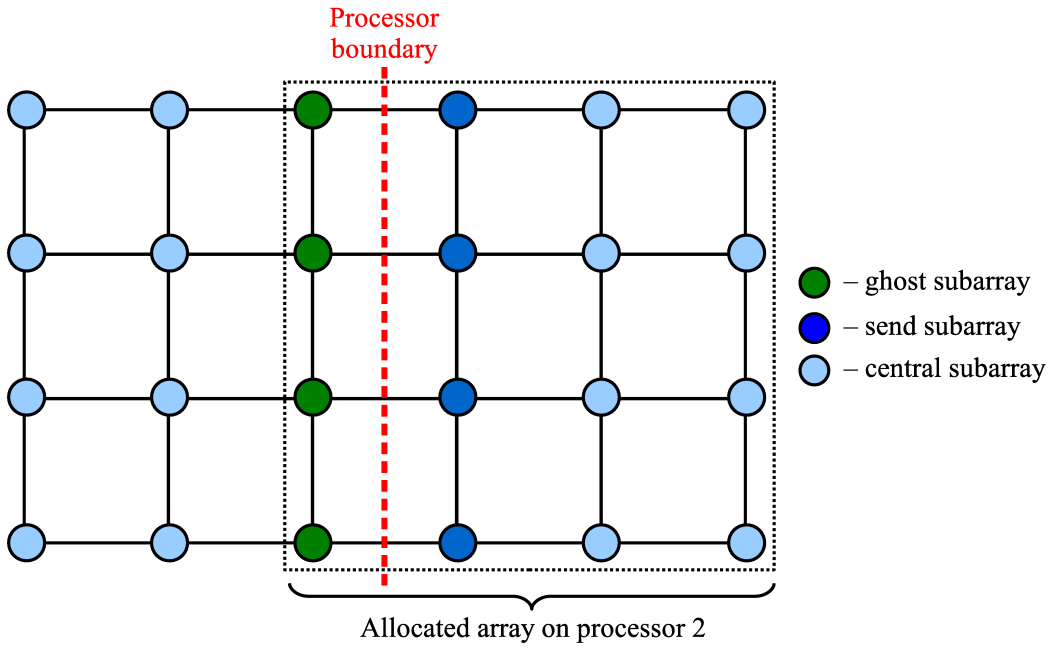

Figure 2.2: Virtual arrays.

```
int ra_create_e(RARR *arr, int ndim, IPNT ge0, IPNT n0),
```

(c) the coordinates of the beginning and end of the array `gs0` and `ge0`

```
int ra_create(RARR *arr, int ndim, IPNT gs0, IPNT ge0).
```

Memory allocation can be delayed. The following functions store array information without allocating memory:

```
int ra_declare_s(RARR *arr, int ndim, IPNT gs0, IPNT n0),
int ra_declare_e(RARR *arr, int ndim, IPNT ge0, IPNT n0),
int ra_declare(RARR *arr, int ndim, IPNT gs0, IPNT ge0),
```

and function

```
int ra_allocate(RARR *arr),
```

which can be used to allocate memory later.

When any of the functions described in the previous paragraph is called, the virtual array pointer `*_s` is set equal to the parent pointer `*_s0`. Virtual array can be created by resetting pointer `*_s` based on the information about virtual array's length `n` and its start and end coordinates `gs` and `ge`:

```
int ra_greset_s(RARR *arr, const IPNT gs, const IPNT n),
int ra_greset_e(RARR *arr, const IPNT ge, const IPNT n),
int ra_greset(RARR *arr, const IPNT gs, const IPNT ge).
```

In some cases, it is more convenient to create a virtual array by providing offsets of its coordinates relative to the start and end coordinates of the parent array:

```
int ra_offset_s(RARR *arr, const IPNT os, const IPNT n),
int ra_offset_e(RARR *arr, const IPNT oe, const IPNT n),
int ra_offset(RARR *arr, const IPNT os, const IPNT oe).
```

The framework also provides a function that performs deallocation of `RARR`:

```
int ra_destroy(RARR *arr).
```

Once an object of type `RARR` is created, the data stored in this object can be accessed directly or via library functions `ra_get` and `ra_set`. The advantage of the get/set functions is that they perform bound check on the access index. If the access index is out of array's bounds, these functions write an error message into the output stream and exit. In addition to the functions that allow access to the data stored in `RARR`, there are library functions that access `RARR`'s members `ndim`, `gs`, `ge`, and `n`:

```
int ra_ndim(RARR *arr, int *ndim),
int ra_gse(RARR *arr, IPNT gs, IPNT ge),
int ra_size(RARR *arr, IPNT n).
```

Finally, there are several functions that provide formatted and binary output of RARR's data into an output stream.

As we mentioned above, a collection of RARR objects forms a structure called RDOM which comes with its own set of library functions. RDOM's library functions fall into four distinct categories:

1. declare/allocate/destroy functions,
2. functions that allow to create virtual subdomains,
3. functions that provide access to RDOM's members,
4. functions that provide access to the data stored in RDOM arrays.

Since syntax of RDOM's library functions is very close to the syntax of RARR's functions, I will not go into any further details.

2.6 Timestep Function

Once the user has defined the number of multidimensional arrays and has created an RDOM, he needs to implement a timestep function. The framework provides the following type to describe a pointer to the timestep function

```
typedef int (*TIMESTEP_FUN)(RDOM *dom, int iarr, int it, void *pars);
```

The variable of this type is stored in IMODEL object:

```
typedef struct IMODEL {
  ...
  TIMESTEP_FUN ts;
  void *tspars;
```

```
  ...
} IMODEL;
```

The first input parameter `dom` of the timestep function is a pointer to the computational domain. The index of the array that has to be recomputed is described by the second input parameter `iarr`. The iteration number is given by `it` and `*pars` is a parameter list for a FD method. Parameter list `*pars` is used to store additional information specific to the FD method, such as CFL number, boundary computation flags, etc. Therefore, the user is responsible for defining such a parameter list.

2.7 Automated Data Exchange

In order to organize the automated data exchange the user needs to:

- break up global physical domain into subdomains according to number of PEs provided by the driver;
- provide a stencil, which is used as a basis for computing exchange arrays.

Based on the domain decomposition and stencil, the driver:

- computes the sizes of exchange areas and prepares parent and virtual arrays,
- creates datatypes that will be used in MPI exchanges.

As mentioned above, the decomposition of the physical domain into subdomains has to be defined by a user based on the number of PEs supplied as an input parameter. In my implementation, the user may specify the MPI process decomposition in one, two, or three dimensions. Ideally, domain decomposition should be defined in such a way, that the optimal load balancing is achieved. Current version of the

code provides uniform domain decomposition for staggered FD schemes of the second order in time and $2k$-th order in space ($k = 1, \ldots, 7$).

Once the driver has the domain decomposition information, it can compute the size of the local computational array on each PE. However, in order for each PE to compute the solution over its local subdomain, one must allocate additional arrays of memory known as ghost cells, to store data from neighboring PEs along the edges of the computational subdomain. In order for the driver to determine the sizes of ghost areas, the user has to provide a FD stencil. In the following section, I describe the stencil datatype and explain how to create new stencils using second-order in space scheme as an example.

2.7.1 Stencil Datatype

FD stencil is a template that shows which grid points participate in the update of the given target point. In general, a FD scheme may have more than a single stencil. For example, for 2-2 FD scheme (2.3)–(2.5) we can define three stencils, one for each variable. Figure 2.3 shows a 2-2 staggered FD stencil for horizontal velocity. Two pressure nodes around the horizontal velocity node participate in the update of this node.

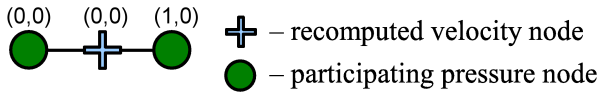

Figure 2.3: Stencil example.

In `IMODEL`, the collection of FD stencils is represented by a structure called `STENCIL`, which consists of array of `STENCIL_MASK` structures. Structure `STENCIL_MASK` describes the relative positions of the target point and participating grid points using the following data fields:

```
int ir,
int ip,
int n,
IPNT *_s,
```

where `ir` and `ip` refer to the indices of the target and participating data arrays in the `RDOM` structure, and array `*_s` of length `n` stores the position of participating grid points relative to the target point. For example, for horizontal velocity stencil described in Figure 2.3, the values of the `STENCIL_MASK` are:

`ir = 2, ip = 0, n = 2, *_s = {(0,0), (1,0)}.`

As with all of the framework structures, `STENCIL_MASK` and `STENCIL` come with the set of library functions:

- constructors and destructors,
- access methods.

The current version of the code provides stencils for 2-2k staggered FD schemes.

In order to create a new stencil, which is stored in the `IMODEL` object as `STENCIL sten;` variable, the user needs to

- define number of stencil masks `nmask` and call the `sten_create(STENCIL *sten, int nmask)` method to allocate the stencil,
- loop over stencil masks and for each mask:
  - allocate the stencil mask,
  - fill the mask with pairs of target and participating arrays' indices,
  - store stencil mask in the stencil.

2.7.2 Ghost Areas Computation

Once the user has provided a stencil, the driver calculates the sizes of the ghost cell areas and prepares exchange virtual arrays.

Figure 2.4 illustrates the calculation of the ghost cell areas based on the horizontal velocity stencil shown in Figure 2.3. Blue circles and pluses represent local pressure and horizontal velocity arrays that have to be updated on a given PE. Applying horizontal velocity stencil, we see that in order to update boundary horizontal velocities, we need pressure values from neighboring PEs. Namely, we need one additional pressure value (or half of all pressure values involved in the stencil) for each velocity that lives near the boundary. Therefore, the driver can compute the size of the ghost cells for horizontal velocity based on stencil's half-length and the size of the local domain.

Note that an array may participate in stencils for different target arrays, for example, the pressure array participates in the stencils for the horizontal and vertical velocities. Thus, the size of the pressure array that needs to be allocated on a PE is an envelope of ghost areas derived from all stencils in which pressure array participates. Once the information about the sizes of ghost cells is available, the driver allocates memory on every PE and prepares computational, frame, and central virtual arrays.

1. Computational virtual array contains all the points that have to be recomputed at each timestep.

2. Frame covers the area that has to be sent to PE's neighbours. To prepare frame virtual arrays, neighbors exchange information about the sizes of their ghost areas.

3. Central virtual array is a set difference between computational virtual array and the frame. This decomposition of the computational array allows for asyn-

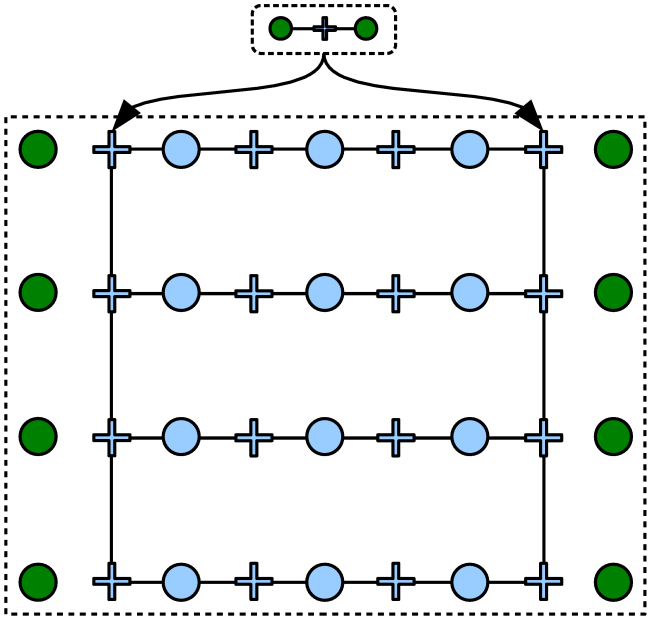

Figure 2.4: Ghost area computation example.

chronous (non-blocking) data exchange mode in which communication occurs concurrently with data update. In the non-blocking exchange mode the frame is updates first. Then MPI communications are started. While MPI communications are carried out by the system the central array is updated.

Typical 2D array layout example is presented in the Figure 2.5. Blue area corresponds to the local computational array. Green areas are ghost arrays and their envelope (allocated array) is shown using dashed line. Four diagonally hatched rectangles form the frame area that will be sent to the neighbour processes. The central virtual array is a cross hatched rectangle.

Due to the fact that exchange arrays are virtual, they may not be laid out contiguously in memory. In order to organize exchange of non-contiguous data, I create necessary MPI datatypes stored in `EXCHANGEINFO` structure. `EXCHANGEINFO` structure contains the pointer `void *buf` to the beginning of the exchange virtual array and two variables `MPI_Datatype type, type2`. The variable `type` contains the MPI

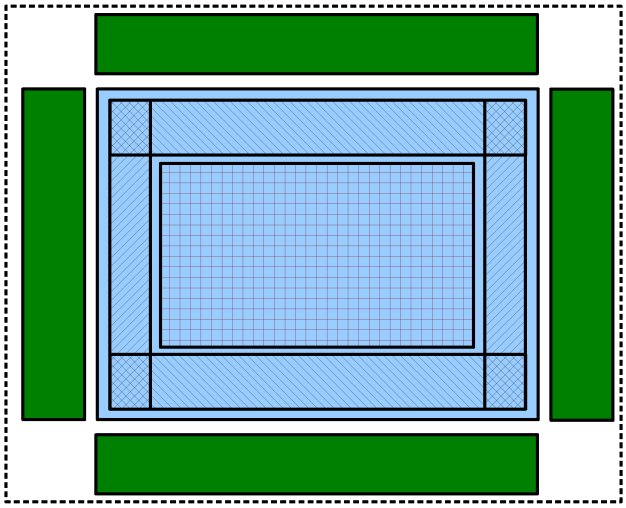

Figure 2.5: Virtual array layout.

datatype corresponding to a 3D virtual array that is exchanged during the MPI communications. It is based on the 2D slice of the array, which is stored in the `type2` variable. Function `int ra_setexchangeinfo(RARR *arr, EXCHANGEINFO *einfo)` is used to construct exchange information for virtual arrays. It takes virtual array `arr` as an input and initializes corresponding output variable `einfo`.

2.8 Timestep Loop and Terminators

The timestep loop is organized as a `while` loop, in which the terminating condition is determined by a query to a terminator object. The concept of terminator was introduced by T. Padula (REF TO THESIS) and the data stucture was implemented in my framework by Dr. W. W. Symes. The terminator data structure is another "template" data structure, which needs to be extended by a user. The main function of a terminator is `int (*query)(IMODEL * m, struct TERM * t)`, which is called before each timestep. If query function returns 0, the timeloop is terminated. Similarly to the timestep function, terminators are used to carry out actions that need

to be performed each time step. However, as opposed to the timestep function, terminator actions may be different from one time step to another. For example, source terminator is used to add source functions to the discrete equations. Since source function may have finite support in time, source terminator may become inactive after some number of iterations. Current version of the code provides the following terminators:

- trace terminator that records traces at specified space locations at every time step;
- movie terminator that records snapshots of the solution at specified time intervals;
- source terminators for the pressure-velocity formulation of the acoustic wave equation:
  - point constitutive defect,
  - point dilatational source,
  - homogeneous initial value data for radiation solution.

The time dependent part of all three source functions is calibrated to produce Ricker pulse of given amplitude at given distance from the source.

Various terminators are joined together by the `ORTERM` datatype. When the `query` function of the `ORTERM` is called, it queries all included terminators and returns 0 if all the terminators vote to stop the timestep loop.

Chapter 3

Numerical Experiments

In this chapter, we study the parallel performance of the code by analyzing its speedup. In parallel computing, speedup (sometimes called strong speedup) measures how much a parallel algorithm is faster than a corresponding sequential algorithm. Speedup is defined by the following formula:

$$S_p = \frac{T_1}{T_p},$$

where p is the number of PEs, T_1 is the execution time of the sequential algorithm, and T_p is the execution time of the parallel algorithm with p PEs. Linear or optimal speedup is obtained if $S_p = p$. An algorithm with linear speedup runs two times faster every time when the number of PEs is doubled.

Numerical experiments described in this section were performed on two different platforms:

1. Cray XD1 distributed memory research cluster (ADA, Rice University, Houston, `http://rcsg.rice.edu/ada/int`) with 316 Dual-core AMD Opteron 275 (2.2GHz) processors (632 cores total). A single node includes 2 processors (4

cores) and 8GB of local RAM. The nodes are connected by Cray "RapidArray" interconnect based on Infiniband. Linux (2.6.5 kernel) operating system and GNU C++ compiler (version 4.1) were used.

2. SGI Altix 4700 shared memory NUMA system (POPLE, Pittsburgh Supercomputing Center, `http://www.psc.edu/machines/sgi/altix/pople.php`) with 384 Dual-core Intel Itanium 2 Montvale 9130M processors (768 cores total). A single node includes 2 processors and 8GB local RAM. The nodes are connected by SGI NUMAlink interconnect. Linux (2.6.16 kernel) operating system and Intel C++ compiler (version 10.1) were used.

Tables 3.1 and 3.2 summarize observed timing results (in seconds) for a non-homogeneous medium of size $3000 \times 3000 \times 3000$ m^3 discretized into $480 \times 480 \times 390 \approx 90 \cdot 10^6$ grid blocks and 500 time steps (final simulation time is 329 ms).

Each table shows timing results for a second-order in time and space and second-order in time tenth-order in space FD scheme. For each scheme, I tested two configurations:

1. one PE per node, i.e. the number of nodes is equal to the number of PEs.

2. one PE per core, i.e. all four MPI processes are executed on each four-core node.

In all experiments I used MPI blocking data exchange. The first column of each table specifies the number of MPI processes. Next four columns (2-5) correspond to the second order in time and space FD scheme, and the last four columns (6-9) correspond to the second order in time and tenth order in space FD scheme. For each FD scheme, the first two columns show wall time of the timestep loop (excluding setup phase) and speedup coefficient for the first configuration (process per node).

The next two columns contain wall time and speedup for the second configuration (process per core).

np	2-2				2-10			
	configuration 1		configuration 2		configuration 1		configuration 2	
	time	speedup	time	speedup	time	speedup	time	speedup
1	3157	-	3157	-	8684	-	8684	-
2	1631	1.9	1690	1.9	4509	1.9	4833	1.8
4	812	3.9	961	3.3	2321	3.7	2406	3.6
8	527	6.0	554	5.7	1519	5.7	1550	5.6
16	287	11.0	289	10.9	792	11.0	804	10.8
32	188	16.8	189	16.7	534	16.3	541	16.1
64	101	31.3	101	31.3	282	30.8	280	31.0
128	63	50.1	63	50.1	176	49.3	176	49.3
256	-	-	39	81.0	-	-	112	77.5
512	-	-	22	143.5	-	-	65	133.6

Table 3.1: ADA timings. NP stands for number of MPI processes, 2-2 refers to the second order in time and space FD scheme, 2-10 refers to the second order in time and tenth order in space FD scheme, configuration 1 and configuration 2 represent one process per node and one process per core configurations respectively. T is a walltime (in seconds), S is a speedup.

Figures 3.1 and 3.2 show speedup graphs for ADA and SGI experiments respectively. The x-axis represents number of MPI processes on the logarithmic scale, the y-axis represents the speedup on the logarithmic scale.

We can see from Tables 3.1 and 3.2 that our parallel code scales well and time taken by the time step loops drops significantly every time when we increase the number of PEs. Comparing timing results for 2-2 and 2-10 schemes, we see that although 2-10 scheme runs two times longer than the 2-2 scheme, it has very similar scaling properties. We conclude that the speedup is independent of the order of the scheme.

In general, wall times in Tables 3.1 and 3.2 show that experiments on SGI architecture scale better and run approximately three times faster than experiments on

np	2-2				2-10			
	configuration 1		configuration 2		configuration 1		configuration 2	
	time	speedup	time	speedup	time	speedup	time	speedup
1	1037	-	1037	-	2389	-	2389	-
2	537	1.9	705	1.5	1224	2.0	1447	1.7
4	268	3.9	580	1.8	616	3.9	1039	2.3
8	158	6.6	313	3.3	358	6.7	459	5.2
16	82	12.6	155	6.7	191	12.5	251	9.5
32	43	24.1	79	13.1	104	23.0	118	20.2
64	26	39.9	43	24.1	65	36.8	73	32.7
128	13	79.8	22	47.1	39	61.3	39	61.3
256	-	-	12	86.4	-	-	22	108.6
512	-	-	6	172.8	-	-	14	170.6

Table 3.2: SGI timings. NP stands for number of MPI processes, 2-2 refers to the second order in time and space FD scheme, 2-10 refers to the second order in time and tenth order in space FD scheme, configuration 1 and configuration 2 represent one process per node and one process per core configurations respectively. T is a walltime (in seconds), S is a speedup.

ADA.

On the SGI architecture, we notice significant difference in wall times between the first and the second configurations, especially for the 2-2 FD scheme. This is related to the limited bus capacity of the system. Four processes running on the same node cannot receive data from memory with sufficient speed, which results in idling. At the same time, when running one process per node, the process recomputes four time less data while using the same bus capacity. For this reason speedup between one and four processor in the one process per core configuration is far from optimal. The speedup coefficients are 1.8 and 2.3 for 2-2 and 2-10 schemes compared to 4.0 optimal speedup. As the number of nodes gets doubled, speedup improves for both configurations and both schemes. The speedup coefficients are from 90 to 96, compared to optimal 128.

On the ADA cluster, bus capacity has less effect on the speedup coefficient, but in general speedup coefficients are not as good as ones for SGI, because MPI data

exchange on SGI is faster due to better interconnect between nodes.

I also performed the same tests on ADA using non-blocking MPI exchange mode. The overall timings and speedup coefficients did not differ between two exchange modes.

Staggered grid FD schemes implemented in my thesis work allow for OpenMP parallelization on shared memory architectures. In my tests I did not observe any significant difference in the computational times between MPI-based, OpenMP-based, or hybrid parallelization on ADA cluster.

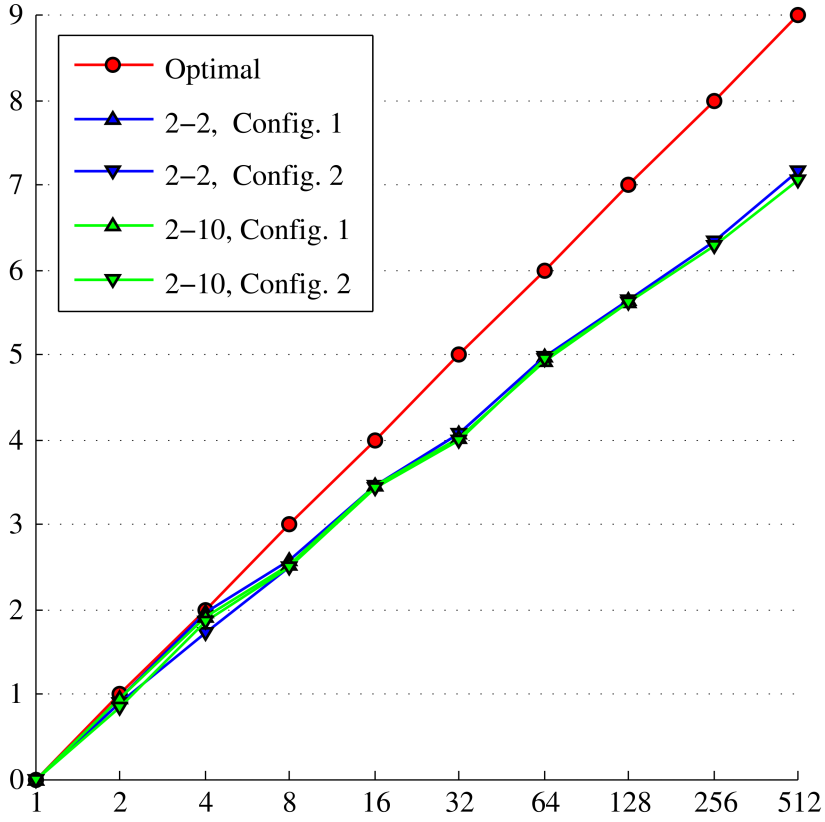

Figure 3.1: ADA speedups. Number of processors is shown on the horizontal axis, $\log_2 S_p$ is presented on the vertical axis, where S_p is a speedup on p processes; 2-2 refers to the second order in time and space FD scheme, 2-10 refers to the second order in time and tenth order in space FD scheme, Config. 1 and Config. 2 represent one process per node and one process per core configurations respectively.

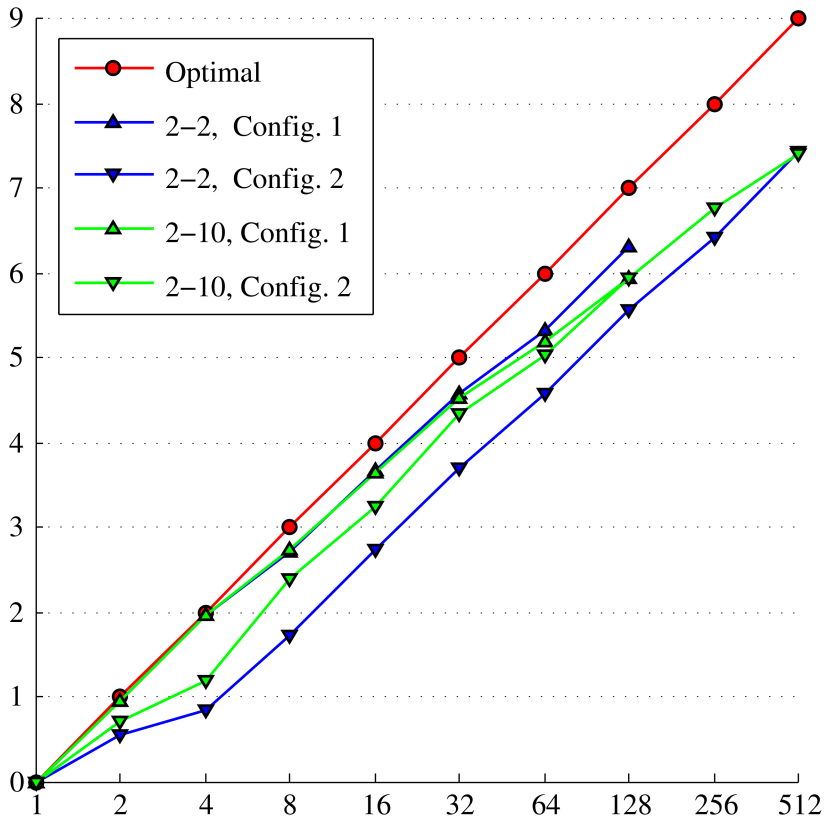

Figure 3.2: SGI speedups. Number of processors is shown on the horizontal axis, $\log_2 S_p$ is presented on the vertical axis, where S_p is a speedup on p processes; 2-2 refers to the second order in time and space FD scheme, 2-10 refers to the second order in time and tenth order in space FD scheme, Config. 1 and Config. 2 represent one process per node and one process per core configurations respectively.

In the next paragraph I describe results of the weak speedup tests on ADA. Weak speedup is assessed by doubling problem size each time the number of PEs p is doubled and comparing execution times T_p. Since the ratio between problem size and the number of PEs stays constant, the theoretical optimal weak speedup is achieved if execution times do not change: $T_1 = T_2 = \ldots = T_N$ There is an important difference between weak and strong speedup tests. For many parallel algorithms, including my implementation of FD methods, the size of the exchanged data is proportional to the size of the boundary (surface) of the local domain. At the same time, the

amount of local computations is proportional to the volume of the local domain. Thus, when measuring strong speedup, the ratio between amounts of the exchanged and recomputed data increases as the number of PE grows. For weak speedup tests, this ratio does not change. Since each PE exchanges data only with neighboring PEs and the number of neighbors does not depend on the total number of PEs, the weak speedup should normally to be closer to the optimal, than the strong speedup.

Tables 3.3 presents results of the weak speedup tests on ADA cluster. The setup is similar to the one used in the previous tests:

- 3D problem, second order in time and space staggered grid scheme,
- $300 \times 300 \times 300$ local problem size,
- 500 time steps,
- one PE per node configuration.

There is a better than optimal weak speedup between one and four processors. I do not have any reasonable explanation to this speedup and leave it to the reader.

np	total problem size	time
1	$300 \times 300 \times 300$	1631
2	$300 \times 300 \times 600$	1442
4	$300 \times 600 \times 600$	967
8	$600 \times 600 \times 600$	1111
16	$600 \times 600 \times 1200$	1098
32	$600 \times 1200 \times 1200$	1109
64	$1200 \times 1200 \times 1200$	1104
128	$1200 \times 1200 \times 2400$	1108

Table 3.3: ADA weak speedup tests.

Chapter 4

Conclusions

Numerical simulation of seismic wave propagation provides an alternative to highly expensive and time consuming real experiments. The most popular tool for simulation of seismic wave propagation is regular grid FD methods. In this thesis, I described a parallel framework for solving time-dependent PDEs in simple domains using uniform grid FD methods. To my knowledge, currently there are no open-source codes that are parallel, flexible enough for adding new methods, optimized and portable.

The main advantage of my software is its reusability. Using predefined extendable datatypes provided by the framework, a user can add new FD methods with minimal programming effort. In order to add a new FD method to the framework, the user needs to

- utilize datatypes that naturally describe the components typical for uniform-grid FD methods (such as multidimensional arrays, stencils, etc.),
- extend datatypes that provide some common functionalities typical for uniform FD methods, but need to be modified for a particular method (such as model, terminator, etc.),

- implement functionalities specific to a particular FD method (such as timestep and boundary condition functions),
- modify the driver to incorporate new model/scheme.

In addition to datatypes and methods that facilitate adding new models and numerical methods, the framework provides tools for automated data exchange between PEs in a distributed computing system, thus allowing for efficient solution of large-scale problems.

One of the advantages of my framework is its portability. It is achieved by implementing the framework in standard ISO C language with widely used MPI for data exchange between PEs in a distributed system.

Based on the framework I implemented a staggered grid solver for the acoustic equation. The solver includes:

- second order in time and 2, 4, ..., 14 order in space FD schemes for 1D, 2D, and 3D problems,
- absorbing and/or reflecting boundary conditions.

The solver was tested on several hardware architectures with up to 512 cores: RICE Opteron cluster, SGI Altix 4700 system. The main result of these tests is good performance and very good scalability of the code, regardless of particular FD schemes used. This shows that the underlying framework does not bring any noticeable performance drops to the implemented numerical schemes.

Bibliography

[1] R.M. Alford, R. Kelly, and D. M. Boore. Accuracy of the finite difference modelling of the acoustic wave equation. *Geophysics*, 39:834–842, 1974.

[2] Regone C. Using 3d finite-difference modeling to design wide-azimuth surveys for improved subsalt imaging. *Geophysics*, 72(5):SM231–SM239, 2007.

[3] G. Cohen and P. Joly. Fourth order schemes for the acoustic wave equation in heterogeneous media. Technical report, INRIA, to appear, 1990.

[4] G. C. Cohen. *Higher Order Numerical Methods for Transient Wave Equations*. Springer, New York, 2001.

[5] R. Courant, S. Friedrichs, and Lewy H. Uber die partiellen differenzengleichungen der mathematischen physik. *Mathematische Annalen*, 100(1):32–74, 1928.

[6] A. Levander. Fourth-order finite-difference P-SV seismograms. *Geophysics*, 53:1425–1436, 1988.

[7] R. Richtmyer and K. Morton. *Difference Methods for Initial-Value Problems*. Interscience Publishers, a division of John Wiley & Sons, New York, 2nd edition, 1967.

[8] J. Virieux. SH-wave propagation in heterogeneous media: Velocity stress finite-difference method. *Geophysics*, 49:1933–1957, 1984.

[9] J. Virieux. P-SV wave propagation in heterogeneous media: Velocity-stress finite-difference method. *Geophysics*, 51:889–901, 1986.

[10] J. Virieux. P-SV wave propagation in heterogeneous media: Velocity stress finite-difference method. *Geophysics*, 51:889–901, 1986.