RICE UNIVERSITY

A Software Framework for Finite Difference Simulation

by

Igor S. Terentyev

Houston, Texas

December 2008

Abstract

A Software Framework for Finite Difference Simulation

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 the ISO C language and uses MPI and OpenMP for parallelization. I used the framework to implement staggered second-order in time and various orders 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.

Special thanks to Prof. Mark Embree, Prof. Matthias Heinkenschloss, and Prof. Tim Warburton, who helped me to considerably improve the 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

List of Figures

List of Tables

Chapter 1

Introduction

Numerical simulations are essential for understanding 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

Seismologists collect information about interior properties of the earth 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 propagation problem. Therefore, being able to solve the forward wave equation efficiently and accurately is crucial for the 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 (2007) used 3D FD wave propagation modeling as an alternative to highly expensive field experiments to demonstrate the advantages of wide-azimuth towed streamer (WATS) acquisition over the traditionally employed sparse acquisition. This research led to the acceptance of WATS, which was a significant change in the acquisition technology.

In the work reported in this thesis I begin the development of 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 the addition of new methods with 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 the capabilities of serial processing). 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 of software for seismic numerical simulations exists. However, there are no open-source codes known to me 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.

The main goal of my framework is to spare the user the necessity of implementing underlying functionalities that are unrelated to the numerical model/method, 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 a well-designed and well-tested software framework and tools targeted for parallel FD schemes, I greatly reduce user's effort and time required to design, implement, and produce new error-free numerical models and schemes. In its current form the framework provides:

- core data types, structures, and methods that describe functionalities common for uniform-grid finite-difference methods. Design and implementation of these tools is completed and has been thoroughly tested.

- well-tested implementation of staggered-grid second order in time and $2k$-th, $k = 1, \ldots, 7$ order in space FD schemes for the acoustic wave equation.

At the moment, the framework provides a driver (the program entry point, general initialization, etc.) that is coupled to the implemented family of staggered schemes. Part of the future work is to separate the driver from the numerical method. Then, the only modification of the driver when implementing new numerical scheme will be adding a scheme to the list of available schemes.

1.2 Finite-Difference Methods

The 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 the terabyte range and beyond, resulting in 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 the 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).

The 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 minimal 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.

For the general theory of FD methods for the approximation of various kinds of PDEs see Godunov and Riabenkii (1962); Richtmyer and Morton (1967); Samarski (1971). These books contain material necessary to understand the concepts of stability, consistency, and convergence, provide descriptions and analysis of the most widely used FD methods, and highlight the difficulties associated with approximation of PDEs by FD methods. For recent advances and up-to-date information on FD methods for time-dependent PDEs refer, for example, to Cohen (2001); Gustafsson (2008); Leveque (2007).

The framework described in this thesis is aimed at both conventional (regular-grid) and staggered-grid schemes. In a regular-grid approach, all functions are approximated at the same grid points. In staggered-grid schemes, several computational grids are employed. The early days of application of FD methods to seismic modeling were dominated by conventional grids (Alford et al., 1974; Alterman and Karal, 1968; Boore, 1970, 1972; Dablain, 1986; Kelly et al., 1976). However, FD solutions on conventional grids are affected by instabilities in media with high velocity contrast and by grid dispersion (Alford et al., 1974; Bayliss et al., 1986; Marfurt, 1984). As a result, staggered-grid approach has become more popular in seismic applications due to the several reasons, such as smaller grid dispersion error, independence of Poisson's ratio, better efficiency.

Staggered FD schemes for hyperbolic systems have their roots in leap-frog methods discussed in detail in Richtmyer and Morton (1967). In electromagnetic applications staggered FD schemes stem from the paper by Yee (1966), who formulated second-order scheme to model Maxwell's equations in isotropic media. Three years earlier, Wilkins (1964) used staggered-grid second-order FD methods to model 1D and 2D elastic-plastic flow. Andrews (1973, 1975), uses an equivalent approach to study tectonic stress release by underground explosion. He specializes Wilkins's approach to small displacements and triangular zones, and applies it to the elastodynamic problem formulated as a first-order hyperbolic system in terms of velocity and stress. The same problem is studied by Madariaga (1976), who used a second-order staggered scheme in dynamic modeling of the earthquake fracture (see also Virieux and Madariaga, 1982). These works initiated application of staggered-grid FD schemes to numerical modeling of seismograms. Virieux (1984, 1986) considered a staggered-grid velocity-stress second-order FD scheme for the elastic wave equation. Levander (1988) formulated and analyzed a fourth-order scheme.

A numerical or grid dispersion is a phenomenon that produces parasitic waves around the solution and can lead to serious errors in interpretation of seismograms. Grid dispersion depends on both the frequency of the wave and the direction of wave propagation.

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, a second order in space and time regular-grid scheme requires at least ten grid points per wavelength (Alford et al., 1974). The requirement on spatial sampling imposes a serious constraint on the size of the problem that can be handled numerically and limits the application of regular-grid or low-order schemes. Klimes (1996) compares second- and fourth-order schemes for the elastic wave equation. He shows that the second-order staggered-grid scheme achieves a given level of accuracy twice as fast as the second-order regular-grid scheme, and fourth-order staggered-grid scheme is 1.5 times faster than fourth-order regular-grid scheme.

Staggered schemes turned out to be particularly advantageous when applied to the elastic wave equation. Virieux (1986) and Levander (1988) demonstrate that they are stable for all values of Poisson's ratio while regular-grid schemes fail to provide stable approximations for materials characterized by high Poison's ratio, see Kelly et al. (1976) for one example. The dispersion relation for P-waves is also independent of Poisson's ratio. The insensitivity to Poisson's ratio makes staggered-grid schemes well-suited for the mixed acoustic-elastic media typical for marine exploration problems.

The framework allows for implementation of schemes of any order in time and space. In general, increasing the order of the numerical scheme leads to decreasing dispersion in case of both regular and staggered grids. Higher-order methods were studied by many authors; see Cohen (2001); Cohen and Joly (1990); Dablain (1986); Gustafsson (2008) 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. Reduction in spatial sampling leads to significant savings in FLOP and computer memory that justifies the cost increase associated with application of higher-order finite differences. For example, 2-4 staggered-grid scheme for elastic wave equation described by Levander (1988) requires several times less memory and computational time than 2-2 staggered-grid scheme discussed by Vireuex. Nonetheless, the savings are not sufficient to ensure the solution of large scale problems that result from reflection seismology in feasible computational time using a single processor. For example, the typical size of the domain of interest in reflection seismology is at least 100 wavelengths in each spatial direction. If 10 grid points per wavelength are 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 are 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. Even if the number of grid points per wavelength is reduced from ten to five, the computational time for a single CPU 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 parallel 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 an ISO C99 compiler with Message Passing Interface (MPI) library and, possibly, OpenMP language extension.

A 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. The 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 above-listed languages with MPI is portable. Other advantages that make MPI-based parallelization favorable are:

- MPI has long history (20 years, including the 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, a 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 the portability requirement is one of the major requirements for the framework, I do not consider support of accelerators in this work.

When an MPI-based program runs on a cluster, the same copy of the program is executed on each processing unit. The 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 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.

Portability, optimality, and parallelization (use of MPI) requirements limit the choice of computer languages suitable for implementation of this FD framework to a small number, namely, ANSI C (C99), C++, and FORTRAN (F77). Any of these three languages could be used to create the framework described in this thesis, and advantages or disadvantages of one language over another for creating a FD simulator are rather non-critical. I chose C99 language for the following reasons:

- C99 completely satisfies the portability requirement. C standard-compliant software can be compiled on a wide range of hardware platforms with no changes to the source code. Furthermore, C is often used as a base computer language for language extensions related to specific hardware, such as CUDA for NVIDIA GPUs, and explicit SSE instructions for Intel and AMD processors. This allows for significant code reuse in case of possible extension of the framework to specific hardware, for instance, accelerators. In Appendix A, I present results of the experimental implementation of some FD schemes for GPU accelerators.

- An extendable framework assumes a certain notion of object-like design. While C99 does not support object-oriented or generic programming, it provides dynamic memory control and use of pointers, which allow for relatively simple implementation of object-like extendable data structures.
- Use of C99 allows for considerable code reuse in case of possible migration of the framework to the C++ language.

1.4 Existing Software

A number of open source software packages that can be used as a basis for a FD simulation framework exist, e. g., PETSc, Trilinos, Overture, Cactus, DUNE. Some of them, namely Overture, Cactus, and DUNE, are written in C++ language and do not satisfy the decision to implement a C99-based framework.

One of the most known open source frameworks for the numerical solution of PDEs is PETSc (Portable Extensible Toolkit for Scientific Computation). It consists of a number of modules (vectors, distributed arrays, nonlinear solvers, timesteppers, etc.), which provide building blocks for the implementation of the numerical solvers for PDEs and related problems on parallel computers. PETSc uses the MPI standard for all message passing communication.

PETSc is a large and heavyweight package and allows for various implementations of FD methods. There are two approaches to represent FD schemes of interest using PETSc provided base data structures.

First approach to implement FD methods is to represent unknown data as a vector and a timestep routine as an action of some (sparse) matrix on this vector. PETSc provides distributed vector and distributed sparse matrix data structures and various accompanying methods, such as matrix vector multiply. This approach allows for unified representation of several FD schemes (e. g. FD scheme for physical area and a different FD scheme for PML area) related to a model. This approach has two disadvantages. First, the amount of memory required to represent the operator matrix is several times greater than the amount of the input data. Despite the fact that finite-difference stencils coefficients are the same for all or almost all grid points and can be hardcoded as constants, in the matrix approach all these coefficients are stored as matrix elements requiring memory proportional to the grid size. Furthermore, additional memory is used to store the indices of non-zero matrix elements. Second, multiplication operation of a sparse matrix by a vector is slower than directly implemented FD timestep because this operation involves indirect memory access, i. e., indices of non-zero matrix entries and then vector entries at these indices are accessed.

The second approach is to implement FD scheme directly, using distributed arrays (DAs) provided by PETSc and intended for use with regular rectangular grids. To create a DA the user needs to supply the following information:

- global number of grid points in each direction (integer number),
- number of processes in each direction (integer number),
- one of two stencil types: box or star (flag),
- stencil size (integer number).

In case of the box stencil, the ghost data is exchanged with all 8 neighboring processes in 2D and 26 neighbors in 3D. In case of a star stencil, the corner neighbors in 2D and vertex and rib neighbors in 3D are ignored.

In both cases the stencil size is the same for all neighbors. The disadvantage of the fixed stencil size is that redundant exchange is performed for some FD schemes. For instance, staggered-grid schemes discussed in this thesis require exchange of each particle velocity component along single coordinate only. As a result, for 3D problem PETSc would perform six data exchanges instead of required two. Data exchange can take a noticeable fraction of total computational time and tripling the exchange, as in the above example, is not desired.

In addition to increasing total computational time, PETSc DA datatype seems to consume more memory than is needed for storage of arrays of requested size. Additional consumption of memory may be related to auxiliary data allocated by DAs, such as, for example, index maps. Appendix B contains an example of PETSc distributed array usage and related memory consumption. To my knowledge, PETSc User Manual does not provide information on whether it is possible to avoid extra memory consumption.

Due to the excessive memory consumption by PETSc data structures and non-optimal data exchange patterns, PETSc cannot provide a suitable base for efficient implementation of a framework for large scale in time and space FD simulations typical for the seismic industry.

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 project is to design and implement a set of tools (datatypes and methods) that describe functionalities common for time-dependent numerical models discretized by uniform-grid FD methods. These tools are carefully designed, thoroughly tested, and should be used as building blocks for implementation of the numerical models and methods described in Chapter 1.

A framework formed by these tools allows a user to concentrate on the implementation of the numerical scheme rather than on the implementation of supporting functionalities, such as data storage, data exchange between processors, parameter input and output, etc. Since implementation of these supporting functionalities is often time-consuming and error-prone, my framework greatly reduces time and effort required to implement a parallel FD scheme.

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).

The 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 without modifications or extensions.

The 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, which contains the `main()` function and a variety of other functions, which initialize the model and run the timestep loop. Currently, the 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 an "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 structure-methods pair 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 a 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 the class inheritance of object-oriented languages. Datatypes that allow extension contain a `void *spec` pointer. By default (in the base datatype) the `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.

- The real datatype allows one to switch between floating point modes in the entire framework based on the compiler variable `DT_REAL` defined in `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.

- The 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 `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 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.

- The 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
    ```
  - a 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 p and velocity $\mathbf{v}$:

$$\begin{aligned} \frac{1}{\kappa(\mathbf{x})}\frac{\partial p(t,\mathbf{x})}{\partial t} &= -\nabla \cdot \mathbf{v}(t,\mathbf{x}) + f(t,\mathbf{x}), && (2.1)\\ \rho(\mathbf{x})\frac{\partial \mathbf{v}(t,\mathbf{x})}{\partial t} &= -\nabla p(t,\mathbf{x}), && (2.2) \end{aligned}$$

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:

$$\begin{aligned} p^n_{i,j} &= p^{n-1}_{i,j} - \frac{\kappa_{i,j}\Delta t}{h_x}\left(u^{n-1/2}_{i+1/2,j} - u^{n-1/2}_{i-1/2,j}\right) && (2.3)\\ &\quad - \frac{\kappa_{i,j}\Delta t}{h_y}\left(v^{n-1/2}_{i,j+1/2} - v^{n-1/2}_{i,j-1/2}\right) + \Delta t f^{n-1/2}_{i,j}, \\ u^{n+1/2}_{i+1/2,j} &= u^{n-1/2}_{i+1/2,j} - \frac{\rho_{i+1/2,j}\Delta t}{h_x}\left(p^n_{i+1,j} - p^n_{i,j}\right), && (2.4)\\ v^{n+1/2}_{i,j+1/2} &= v^{n-1/2}_{i,j+1/2} - \frac{\rho_{i,j+1/2}\Delta t}{h_y}\left(p^n_{i,j+1} - p^n_{i,j}\right), && (2.5) \end{aligned}$$

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

$$\begin{aligned} p^n_{i,j} &= p(t_0 + \Delta t n, x_0 + h_x i, y_0 + h_y j), && (2.6)\\ u^{n+1/2}_{i+1/2,j} &= u(t_0 + \Delta t(n+1/2), x_0 + h_x(i+1/2), y_0 + h_y j), && (2.7)\\ v^{n+1/2}_{i,j+1/2} &= v(t_0 + \Delta t(n+1/2), x_0 + h_x i, y_0 + h_y(j+1/2)). && (2.8) \end{aligned}$$

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 an 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 the `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 the `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 one to conveniently manipulate such objects.

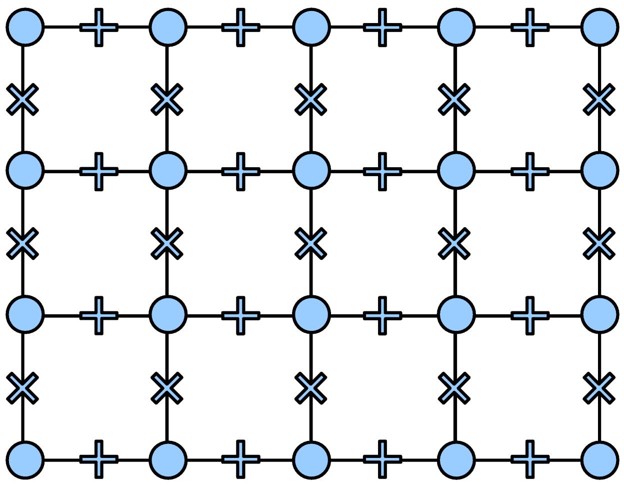

Figure 2.1: Two-dimensional staggered grid. Circles stand for pressure grid points, plusses and crosses refer to horizontal and vertical velocity grid points respectively.

2.5 Base Structures: RARR, RDOM

The main advantage of the object of type `RARR` provided by this framework is that it allows one to create subarrays or virtual arrays. The concept of virtual array is extremely 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, a virtual array refers to a subset of memory allocated by a parent array. A 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 the i-th dimension of the `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: $n = ge - gs + 1$, $n0 = ge0 - gs0 + 1$.

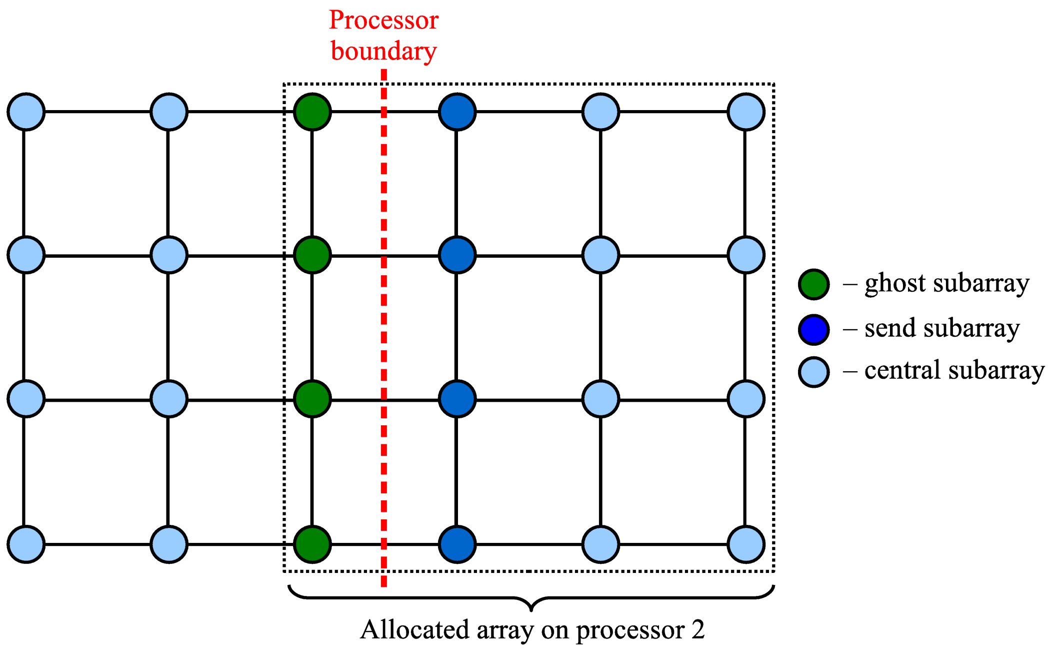

Figure 2.2: Virtual arrays.

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`

```
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),
```

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`. A virtual array can be created by resetting pointer `*_s` based on the information about the 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 a bound check on the access index. If the access index is out of the 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 an `RARR`'s data into an output stream.

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

1. declare/allocate/destroy functions,

2. functions that allow one to create virtual subdomains,
3. functions that provide access to an `RDOM`'s members,
4. functions that provide access to the data stored in `RDOM` arrays.

Since syntax of `RDOM` library functions is very close to the syntax of `RARR` 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 an `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 an FD method. The 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 the 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 optimal load balancing is achieved. The current version of the code provides uniform domain decomposition for staggered FD schemes of 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 an FD stencil. In the following section, I describe the stencil datatype and explain how to create new stencils using the second-order in space scheme as an example.

2.7.1 Stencil Datatype

An FD stencil is a template that shows which grid points participate in the update of the given target point. In general, an FD scheme may have more than a single stencil. For example, for the $2-2$ 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.

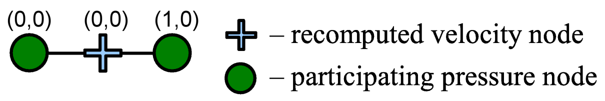

Figure 2.3: Stencil example.

In `IMODEL`, the collection of FD stencils is represented by a structure called `STENCIL`, which consists of an 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 the 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.

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 the `STENCIL sten` variable, the user needs to

- define the 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 a 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 the pressure array participates. Once the information about the sizes of ghost cells is available, the

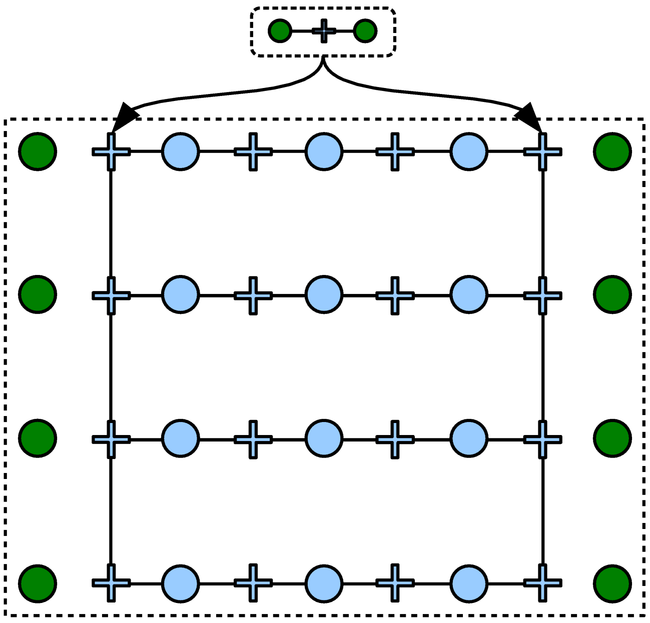

Figure 2.4: Ghost area computation example.

driver allocates memory on every PE and prepares computational, frame, and central virtual arrays.

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

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

3. The central virtual array is a set difference between the computational virtual array and the frame. This decomposition of the computational array allows for an asynchronous (non-blocking) data exchange mode in which communication occurs concurrently with a data update. In the non-blocking exchange mode the frame is updated first, then MPI communications are started. While MPI communications are carried out by the system, the central array is updated.

A typical 2D array layout example is presented in Figure 2.5. The blue area corresponds to the local computational array. Green areas are ghost arrays and their envelope (allocated array) is shown using dashed lines. 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 the exchange of non-contiguous data, I create necessary MPI datatypes stored in the `EXCHANGEINFO` structure. The `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 datatype corresponding to a 3D virtual array that is exchanged during MPI communications. It is based on a 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 the virtual array

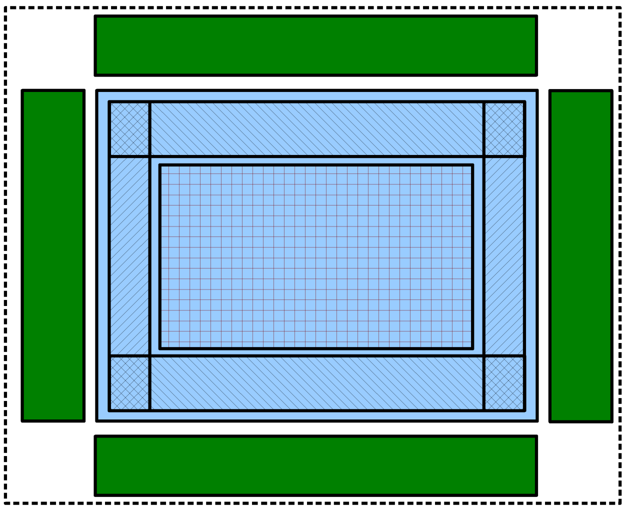

Figure 2.5: Virtual array layout.

`arr` as an input and initializes the 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 Padula et al. (2009) 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 the 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, a source terminator is used to add source functions to the discrete equations. Since the source function may have finite support in time, the source terminator may become inactive after some number of iterations.

The current version of the code provides the following terminators, adapted to staggered grid acoustic FD schemes:

- 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 a target pulse (either read from a file or the second derivative of a Gaussian) 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. A 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 the Cray "RapidArray" interconnect based on Infiniband. Linux (2.6.5 kernel) operating system and the 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 the SGI NUMAlink interconnect. Linux (2.6.16 kernel) operating system and the 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. The 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 the 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).

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.

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

Figures 3.1 and 3.2 show speedup graphs for ADA and SGI experiments respectively. The x-axis represents the number of MPI processes on a logarithmic scale, the y-axis represents the speedup on a 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 we increase the number of PEs. Comparing timing results for 2-2 and 2-10 schemes, we see that although the 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 ADA.

On the SGI architecture, we notice a significant difference in wall times between

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.

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

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 the speedup between one and four processors 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, the 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 those for the SGI, because MPI data exchange on the 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 the two exchange modes.

Staggered grid FD schemes implemented in my framework 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.

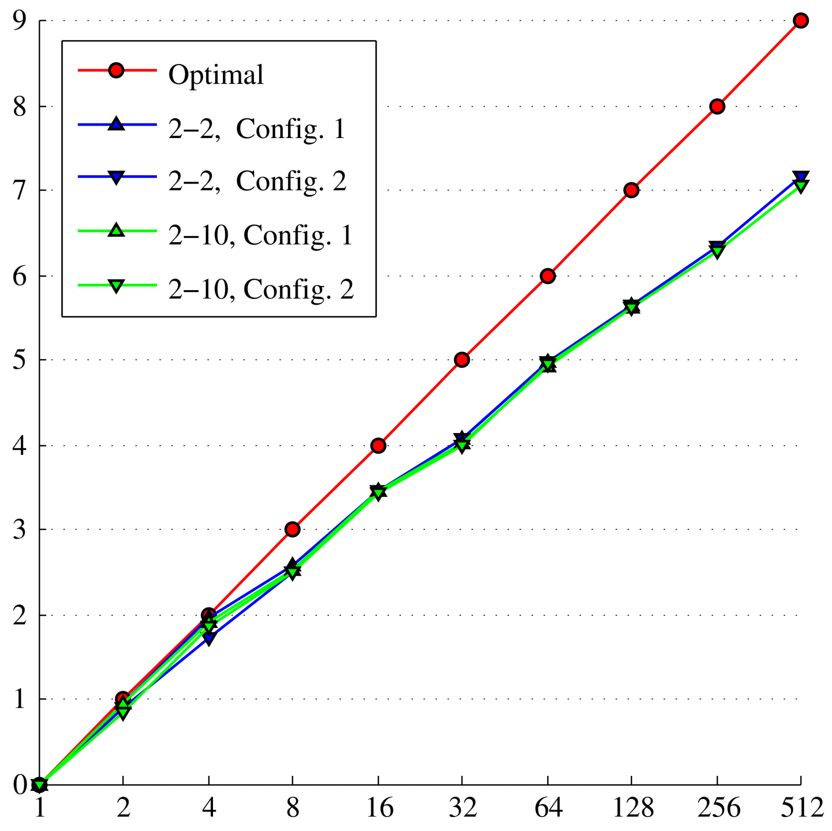

Figure 3.1: ADA speedups. Number of PEs is shown on the horizontal axis, $\log_2 S_p$ is presented on the vertical axis, where S_p is a speedup on p PEs; 2-2 refers to the second order in time and space FD scheme, 2-10 refers to the second order in time and tenth order in space FD scheme, Config. 1 and 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

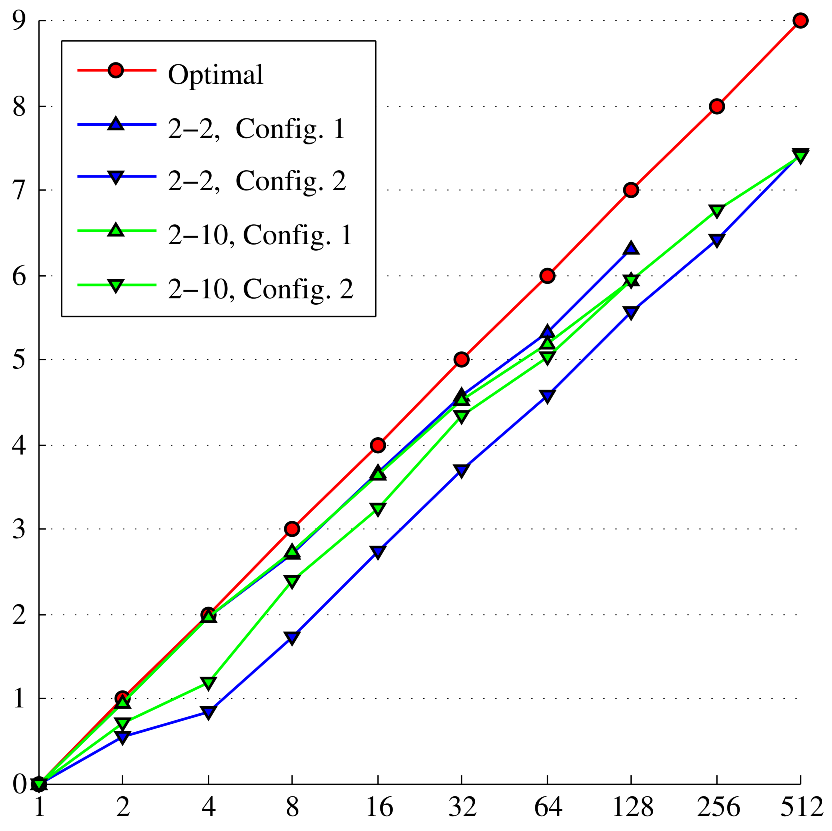

Figure 3.2: SGI speedups. Number of PEs is shown on the horizontal axis, $\log_2 S_p$ is presented on the vertical axis, where S_p is a speedup on p PEs; 2-2 refers to the second order in time and space FD scheme, 2-10 refers to the second order in time and tenth order in space FD scheme, Config. 1 and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

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 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 the 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.

Table 3.3: ADA weak speedup tests. np stands for number of MPI processes.

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

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 the 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 the 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 wave 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 Ada 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

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

Alterman, Z. and Karal, F. C. (1968). Propagation of elastic waves in layered media by finite difference methods. *Bull. Seismol. Soc. Am.*, 58:367–398.

Andrews, D. J. (1973). A numerical study of tectonic stress release by underground explosions. *Bull. Seismol. Soc. Am.*, 63:1375–1391.

Andrews, D. J. (1975). From antimoment to moment: plane-strain models of earthquakes that stop. *Bull. Seismol. Soc. Am.*, 65:163–182.

Bayliss, A., Jordan, K. E., Lemesurier, B. J., and Turkel, E. (1986). A fourth-order accurate finite-difference scheme for the computation of elastic waves. *Bull. Seismol. Soc. Am.*, 76(4):1115–1132.

Boore, D. M. (1970). Love waves in non-uniform wave guides: finite difference calculations. *Geophys. Res.*, 75:1512–1527.

Boore, D. M. (1972). Finite difference methods for seismic wave propagation in heterogeneous materials. *Chapter 1 in Methods in Computational Physics*, II.

Cohen, G. (2001). *Higher Order Numerical Methods for Transient Wave Equations.* Springer, New York.

Cohen, G. and Joly, P. (1990). Fourth order schemes for the heterogeneous acoustics equation. *Comput. Methods Appl. Mech. Eng.*, 80(1-3):397–407.

Dablain, M. A. (1986). The application of high-order differencing to the scalar wave equation. *Geophysics*, 51:54–66.

Godunov, S. K. and Riabenkii, V. S. (1962). *Vvedenie v teoriu raznostnih shem.* Fizmatgiz, Moskva.

Gustafsson, B. (2008). *High Order Difference Methods for Time Dependent PDE.* Springer, Berlin.

Kelly, K. R., Ward, R. W., Treitel, S., and Alford, R. M. (1976). Synthetic seismograms: a finite-difference approach. *Geophysics*, 41:2–27.

Klimes, L. (1996). Accuracy of elastic finite differences in smooth media. *Pure Appl. Geophys.*, 148:39–76.

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

Leveque, R. (2007). *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia.

Madariaga, R. (1976). Dynamics of an expanding circular fault. *Bull. Seismol. Soc. Am.*, 66:163–182.

Marfurt, K. (1984). Accuracy of finite-difference and finite-element modeling of the scalar and elastic wave equations. *Geophysics*, 49:553–549.

Padula, A., Scott, S., and Symes, W. (2009). The Rice Vector Library: a software framework for the abstract expression of coordinate-free linear algebra and optimization algorithms. *Association for Computing Machinery Transactions on Mathematical Software*, 36(2).

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

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

Samarski, A. A. (1971). *Teoria raznostnih shem*. Nauka, Moskva.

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

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

Virieux, J. and Madariaga, R. (1982). Dynamic faulting studied by a finite difference method,. *Bull. Seismol. Soc. Am.*, 72:345–369.

Wilkins, M. L. (1964). Calculation of elastic-plastic flow. *Methods in Computational Physics*, 3.

Yee, K. S. (1966). Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *IEEE transactions on antennas and propagation*, AP-14:302–307.

Appendix A

Computations on Graphics Processing Units

A.1 Introduction

In this appendix, I investigate the usability of graphics processing units (GPUs) for explicit FD methods. I consider the pressure update equation for staggered-grid schemes of the second order in time and $2k$-th ($k = 2$, 4, 8) order in space for a 2D acoustic wave equation. I mainly investigate the case $k = 2$ ($2 - 4$ FD scheme).

The pressure update for the $2 - 4$ staggered-grid scheme is given by the following algorithm:

```
for i = 0; i < ny
  for j = 0; j < nx
    p[i,j] += b[i,j] *
      ( LX * ( u[i,j+3]-u[i,j] + C * (u[i,j+1]-u[i,j+2]) )
        +
        LY * ( v[i+3,j]-v[i,j] + C * (v[i+1,j]-v[i+2,j]) ) )
  end
end
```

Here `C`, `LX`, and `LY` are constants. Arrays `p` (pressure), `b` (bulk modulus), `u` (horizontal velocity), and `v` (vertical velocity) have the following sizes (the second coordinate is fast):

`p`: `ny` × `nx`,

`b`: `ny` × `nx`,

`u`: `ny` × `nx+R-1`,

`v`: `ny+R-1` × `nx`.

A.2 Hardware

I used the following NVidia GPUs:

- Tesla C1060 (4 GB, compute capability 1.3),
- Tesla C870 (1.5 GB, compute capability 1.0),
- GeForce GTX 280 (1 GB, compute capability 1.3),
- Quadro FX 1700 (512 MB, compute capability 1.1).

A.3 Kernels

Arrays `p`, `b`, `u`, and `v` are stored in the global device memory. Only the pressure array `p` is updated, which allows for access of the other three arrays as read-only (using textures). This possibility increases the number of various kernels that can be implemented. I present test results for kernels that do not use texture memory and use shared memory to cache `u` and `v` arrays. I do not cache `p` and `b`, since each element of these arrays is accessed only once.

Below I describe five kernel implementations. The first two kernels represent direct translation of the CPU code (with different order of loops) and do not use shared memory. The last three kernels use shared memory and slightly differ from each other by shared memory reuse while performing loops.

A.3.1 Straight-forward Kernel

In this kernel, I implement the given above algorithm directly without using shared memory. The number of threads is `ny`, and each thread performs an inner loop along the fast coordinate for fixed slow coordinate. This kernel is extremely slow due to non-coalesced memory reads.

A.3.2 Improved Straight-forward Kernel

This kernel differs from the previous one by the loop order: number of threads is `nx`, and each thread performs an outer loop along the slow coordinate for fixed fast coordinate. I also unroll the outer loop four times and reuse three velocity `v` values instead of re-reading them. This kernel gives much better performance of up to 35 GFLOPS on fast GTX 280.

A.3.3 Block Kernel (BK)

This kernel uses shared memory. The total pressure domain is divided into rectangular blocks. Each block is executed as a single grid block on a GPU. Threads form sub-blocks inside the block. The kernel performs a loop through the sub-blocks. For instance, the total domain of 8192×8192 is divided into 16 blocks in each dimension with each block of size 512×512. Sub-block size is chosen to be 32×16 (512 threads). Assuming that `p`, `b`, `u`, and `v` are pointing at the beginning of the 512×512 block (which is done in the kernel before the loops, based on the grid block Id), the kernel is the following:

```
int tX = threadIdx.x;
int tY = threadIdx.y;
for i = 0; i < 512 / 16
  for j = 0; j < 512 / 32

    int ia = i*16+tY;                          // Shifted index.
    int ja = j*32+tX;                          // Shifted index.

    // Load u, v sub-block into shared memory su, sv.
    su[tY,tX] = u[ia,ja];                      // Main block.
    sv[tY,tX] = v[ia,ja];                      // Main block.
    if ( tx < 3 ) su[tY,tX+32] = u[ia,ja+32]; // Extra strip along y.
    if ( tx < 3 ) sv[tY+16,tX] = v[ia+16,ja]; // Extra strip along x.

    // Compute pressure.
    p[ia,ja] += b[ia,ja] *
      ( LX * ( su[tY,tX+3]-su[tY,tX] + C * (su[tY,tX+1]-su[tY,tX+2]) )
        +
        LY * ( sv[tY+3,tX]-sv[tY,tX] + C * (sv[tY+1,tX]-sv[tY+2,tX]) ) )
  end
end
```

Performance of this kernel depends on block and sub-block sizes and can reach 45 GFLOPS on GTX 280.

A.3.4 Improved Block Kernel (IBK)

In the previous kernel I re-read `su` extra strip along y axis in the inner loop. In this kernel instead of re-reading, I copy the last three columns of the `su` buffer into the first three and read the main block only. This kernel improves the performance of the Block Kernel by up to 20%.

A.3.5 Best Block Kernel (BBK)

Here I additionally copy `sv` extra strip. Since it is the outer loop, I allocate long 512×3 additional buffer to store `sv` extra strip. This kernel can give up to 10% improvement over the previous one.

A.4 Performance

Performance of the straight-forward kernel is 10 times worse than the CPU code. All other kernels give performance above 20 GFLOPS on GTX 280 and TESLA C1060 GPUs. TESLA C870 is in general two times slower. Quadro FX 1700 is much slower and is comparable to CPU.

Two tables below show performances of two GPU devices, TESLA C1060 and overclocked GTX 280 for $2-4$ scheme, 4096×4096 grid size. In both tables number of threads is 512. The first table corresponds to 32×16 thread block layout and the second table corresponds to 64×8 thread block layout. The first column represents the GPU model and the next three columns represent three shared memory kernels described above. The numbers are given in GFLOPS. It turned out to be important

Table A.1: GPU preformance, 32×16 thread layout.

GPU	BK	IBK	BBK
TESLA C1060	32.9	37.3	39.8
GTX 280	45.6	49.8	55.4

Table A.2: GPU preformance, 64×8 thread layout.

GPU	BK	IBK	BBK
TESLA C1060	34.1	36.8	43.3
GTX 280	47.1	49.9	58.2

to use aligned (by 64 Bytes) arrays. This makes Improved Block Kernel code more complicated, because extra strips have to be aligned as well. Therefore, instead of reading 3 line wide strips, I read 16 line wide strips (without any performance loss). Reading 16 line wide strips allows me to implement $2-8$ and $2-16$ schemes which run slightly slower than $2-4$. However, due to the fact that $2-8$ and $2-16$ schemes involve more FLOP per grid point (13 for $2-4$, 25 for $2-8$, and 49 for $2-16$), the FLOPS count is better for higher-order schemes. For instance, performance can increase from 45 GFLOPS ($2-4$) to 75 GFLOPS ($2-8$) and 110 GFLOPS ($2-16$) for Block Kernel and 7168×7168 grid size. The highest performance I was able to achieve is 135 GFLOPS for scheme $2-16$ using Best Block Kernel and 4096×4096 grid size.

Also, different thread block dimensions change performance by up to 20%. Namely, 64×8 is optimal (better, for instance, than 32×16). I have no explanation for this behavior, because both 32 and 64 are multiples of half-warp size, which is the determining factor for the performance of block kernels.

A.5 Error

Results of the GPU computations differ from the CPU results. The error does not depend on

- the kernel,
- the GPU model.

I.e., results of all GPU runs using various kernels and various GPU devices are identical to each other and different from the CPU results. As the pressure update is performed inside the loop, the relative error increases with the number of time steps (TS). Table below shows the error for a 1024×1024 experiment. The input arrays were randomly generated and contained values between 1 and 2.

Table A.3: GPU error.

Number of TS	Relative L^∞ error	Relative L^2 error
1	$9.886 \cdot 10^{-8}$	$9.416 \cdot 10^{-9}$
10	$7.177 \cdot 10^{-7}$	$8.621 \cdot 10^{-8}$
100	$2.772 \cdot 10^{-6}$	$4.551 \cdot 10^{-7}$
1000	$1.719 \cdot 10^{-5}$	$1.099 \cdot 10^{-6}$
10000	$1.408 \cdot 10^{-4}$	$3.681 \cdot 10^{-6}$
100000	$1.558 \cdot 10^{-3}$	$1.381 \cdot 10^{-5}$

A.6 Conclusions

GPUs can substantially increase the performance of FD methods. Below I present peak performances for FD schemes described in the introduction. They were obtained on GeForce GTX 280 GPU for large square data arrays (4096×4096 – 8000×8000) using shared memory kernels.

- 35-50 GFLOPS for 2D $2-4$ scheme,
- 55-80 GFLOPS for 2D $2-8$ scheme,
- 90-115 GFLOPS for 2D $2-16$ scheme.

A.6.1 Limiting Factors

The average performance can be lower than the numbers above due to various factors. Fine tuning of the code and parameters is required to achieve the best timings. The best parameters are hard to guess a priori and experiments are required.

- Memory (mis)alignment can change the performance by 50%.
- CUDA compilers seem to be raw. For example, CUDA 2.1 used 30 registers for some kernel, whereas CUDA 2.0 used 34 registers for the same kernel. Here 32 is the critical number of registers, because running all 512 threads with total number of 16384 registers requires at most 32 registers per thread. Using all 512 threads gave me better performance.
- In block-structured kernels, the block size distribution is important. For instance, 32×16 block gives better performance than 16×32, though both use total of 512 threads and both dimensions are multiples of half-warp size.
- Smaller matrix dimensions or narrow strips (say 10000×10) can possibly give lower performance – I have not tested them. Kernel parameters need to be adjusted for different cases of input data size.

A.6.2 CUDA Experience

My general observations of programming with CUDA are:

- CUDA is easy to learn and start using.
- On the other hand, even simple algorithm may require more time for experiments in order to find a kernel which gives best performance. A bad kernel can perform orders of magnitude worse than a good kernel. Such a behavior is not typical for CPU codes.

- Timing results may be hard to predict. Kernels which I thought would be faster were not always faster.
- CUDA might be harder to debug (no `printf`-s from kernels).

A.6.3 Extensions

Thoughts about extensions to 3D and large problems:

- More kernels can be tried (using various combinations of shared memory and textures). Potentially better performance can be achieved.
- While the performance of a 3D scheme is difficult to predict, I do not expect it to be much worse that performance of a 2D scheme. In 3D case, scarcity of registers and shared memory may become an issue.
- GPUs can significantly accelerate single time step computations, and the communication times may become an bottleneck. First, transferring data between host and device memory (via PCI-E) is not fast. Moreover, if we consider the current MPI exchange times on ADA for *SEAMX* simulator, which are about 20% of the total time, and assume that we significantly improved time step performance, the total speedup coefficient will be 5, unless we improve (get rid of) MPI exchange times.

Appendix B

PETSc Distributed Array Test

In this appendix, I provide an example of PETSc distributed array (DA) usage and observed memory usage by DA data structure. I test a 1D PETSc array distributed between multiple processors.

B.1 PETSc Example

PETSc (3.0) users manual describes DA data structure as follows:
… The PETSc DA object manages the parallel communication required while working with data stored in regular arrays. The actual data is stored in appropriately sized vector objects; the DA object only contains the parallel data layout information and communication information, however it may be used to create vectors and matrices with the proper layout.

PETSc provides three functions `DACreate1d`, `DACreate2d`, and `DACreate3d` to create 1D, 2D, and 3D distributed arrays respectively. The 1D function has the following interface:

```
DACreate1d(MPI_Comm comm, DAPeriodicType wrap, int M, int w,
           int s, int *lc, DA *inra)
```

where `M` is a global array size, `w` is a number of processors, `s` is a stencil size, and `inra` is a distributed array (output parameter).

PETSc provides a number of functions to access data in a distributed array. Functions

```
DAGetLocalVector(DA da, Vec *l)
DAGetGlobalVector(DA da, Vec *l)
```

return a pointer to a part of the distributed vector stored on the current processor (first function includes ghost points). Function

```
DAGetCorners(DA da, int *x, int *y, int *z, int *m, int *n, int *p)
```

returns size of a locally stored part of the vector; analogous `DAGetGhostCorners` function includes ghost points.

The following extract of the code creates a 1D array of size `N` with stencil size 1, initializes values of each local array to the processor rank, and performs synchronization:

```
01  int orig;   /* start index of the local array */
02  int len;    /* length of the local array      */
03  int i;
04  Vec gx, lx;       /* local vectors */
05  PetscScalar *arr; /* raw data      */
06
07  /* create distributed array and get local access */
08  DACreate1d(PETSC_COMM_WORLD, DA_NONPERIODIC, N, 1, 1,
09             PETSC_NULL, &da);
10  DAGetGlobalVector(da, &gx);
11  DAGetLocalVector(da, &lx);
12  DAVecGetArray(da, lx, &arr);
13  DAGetGhostCorners(da, &orig, PETSC_NULL, PETSC_NULL, &len,
14                    PETSC_NULL, PETSC_NULL);
15
16  /* initialize local array */
17  for ( i = orig; i < orig+len; ++i ) arr[i] = rk;
18
19  /* perform synchronization */
20  MPI_Barrier(PETSC_COMM_WORLD);
21  DALocalToGlobal(da, lx, INSERT_VALUES, gx);
```

B.2 Memory consumption

I ran the test example shown above with $N = 10^8$. A 1D array of doubles of such length occupies 763 MiB of memory.

After executing line `08`, the process occupied 765 MiB if run on a single processor, and if run on two processors, each process occupied 574 MiB Also, while executing line `08` on two processors, the memory consumption by each process temporarily jumped to almost 900 MiB.

Executing line `10` in two-processor configuration increases memory consumption by each process to 956 MiB.