PetClaw: Parallelization and Performance Optimization of a Python-Based Nonlinear Wave Propagation Solver Using PETSc

Thesis by

Amal Mohammed Alghamdi

In Partial Fulfillment of the Requirements

For the degree of

Master of Science

King Abdullah University of Science and Technology, Thuwal,

Kingdom of Saudi Arabia

April, 2012

The undersigned approve the thesis of Amal Mohammed Alghamdi

David E. Keyes Committee Member	Signature	Date
Aron J. Ahmadia Committee Member	Signature	Date
David I. Ketcheson Committee Chair	Signature	Date

King Abdullah University of Science and Technology

2012

ABSTRACT

PetClaw: Parallelization and Performance Optimization of a Python-Based Nonlinear Wave Propagation Solver Using PETSc

Amal Mohammed Alghamdi

Clawpack, a conservation laws package implemented in Fortran, and its Python-based version, PyClaw, are existing tools providing nonlinear wave propagation solvers that use state of the art finite volume methods. Simulations using those tools can have extensive computational requirements to provide accurate results. Therefore, A number of tools, such as BearClaw and MPIClaw, have been developed based on Clawpack to achieve significant speedup by exploiting parallel architectures. However, none of them has been shown to scale on a large number of cores. Furthermore, these tools, implemented in Fortran, achieve parallelization by inserting parallelization logic and MPI standard routines throughout the serial code in a non modular manner.

Our contribution in this thesis research is three-fold. First, we demonstrate an advantageous use case of Python in implementing easy-to-use modular extensible scalable scientific software tools by developing an implementation of a parallelization framework, PetClaw, for PyClaw using the well-known Portable Extensible Toolkit

for Scientific Computation, PETSc, through its Python wrapper petsc4py.

Second, we demonstrate the possibility of getting acceptable Python code performance when compared to Fortran performance after introducing a number of serial optimizations to the Python code including integrating Clawpack Fortran kernels into PyClaw for low-level computationally intensive parts of the code. As a result of those optimizations, the Python overhead in PetClaw for a shallow water application is only 12 percent when compared to the corresponding Fortran Clawpack application.

Third, we provide a demonstration of PetClaw scalability on up to the entirety of Shaheen; a 16-rack Blue Gene/P IBM supercomputer that comprises 65,536 cores and located at King Abdullah University of Science and Technology (KAUST). The PetClaw solver achieved above 0.98 weak scaling efficiency for an Euler application on the whole machine excluding the initialization overhead that is less significant in potential long runs of PetClaw applications. The solver also achieved superlinear strong scaling efficiency for an acoustics application on 1,024 cores.

Furthermore, we provide reproducibility information for all the computational experiments presented in this thesis.

ACKNOWLEDGMENTS

I sincerely would like to thank my supervisor, Prof. David Ketcheson, for this research opportunity, for his unlimited support and encouragement and for his continuous guidance. I also would like to thank Dr. Aron Ahmadi for his great support and guidance and for inviting me to work on this research at the first place. Working on this research with them has been a great opportunity for me to enrich my experience and grow academically.

I sincerely thank Prof. David Keyes for his valuable feedback and encouragement. I also would like to thank Lisandro Dalcin for his suggestions regarding serial optimizations and helping with petsc4py.

Finally, I would like to say that I'm very grateful to my parents and my siblings for their non stopping support, and to my husband for being always supportive and encouraging; and standing by my side in all times.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF CODE LISTINGS

SUMMARY

In this thesis we present our approach to parallelize an existing nonlinear wave-propagation solver based on the finite volumes Godunov-based methods for solving hyperbolic PDEs. The original serial tool is available in two implementations, the Fortran implementation, Clawpack, and the Python implementation, PyClaw. We have extended the Python-based implementation by creating a Python-based package, PetClaw, with maximal code reuse from PyClaw through class inheritance. PetClaw uses the package petsc4py, a Python interface to the Portable Extensible Toolkit for Scientific Computation, PETSc, to implement a distributed-memory parallelization based on MPI. As using an interpreted language such as Python introduces overhead leading to non-optimal performance, we have used the interface generator f2py to integrate PyClaw with a set of Clawpack routines that implements the numerically intensive parts of the solver algorithm. This combination of tools facilitates obtaining a high-level modular extensible design as a result of using Python, that also has the performance advantages of the compiled code.

To validate our optimization and parallelization approaches, we provide performance study demonstrating both the serial and parallel performance of PetClaw. This includes studying the effect of the performed serial optimizations of PyClaw and PetClaw on the execution time and presenting weak and strong scalability study in which we ran the code on up to the entirety of Shaheen, 65,536 cores.

In the introductory chapter, Chapter I, we cover the necessary mathematical and

software background. We first introduce conservation laws and hyperbolic PDEs, the class of PDEs that PetClaw solves. We also describe the finite volume methods in general and Godunov-based methods in particular, which are the type of methods PetClaw uses and introduce Clawpack as well. Then we present and assess three parallel software tools that are based on Clawpack mentioning some of their shortcomings that justify the need for PetClaw. We then discuss three approaches that can be followed to parallelize tools like Clawpack and PyClaw. After that, we discuss the advantages of using Python as a high-level language in scientific software tools followed by introducing PyClaw.

The implementation details of our approach are discussed in Chapter II; starting by giving an overview of PetClaw modular structure and elaborating on the the design of Clawpack, PyClaw and PetClaw and how f2py, petsc4py were utilized. We then present a use case scenario of using PetClaw to construct and run a selected application.

Chapter III presents a study of PetClaw serial and parallel performance. We start by defining the performance metrics we have used to assess PetClaw serial and parallel performance. Then we talk about some serial optimization techniques that we have considered in developing PetClaw along with presenting serial performance comparison of a PetClaw and Clawpack code. Finally, we present a detailed scalability study of two PetClaw representative applications on up to the entirety of Shaheen to demonstrate both the strengths and weaknesses of our tool's parallel performance.

In Chapter IV, we summarize the key conclusions of this thesis research. We also give an idea about current work being conducted for improving PetClaw and PyClaw and future plans for further extending those tools.

Finally, we provide reproducibility information for Chapter III experiments in Appendix A. We list the software and hardware dependencies and give instructions for building and running Clawpack and PyClaw applications along with the required

environment settings. We then give detailed information about the reproducibility of the serial optimization experiments, serial comparison between PetClaw and Clawpack performance, weak scalability experiment and strong scalability experiment.

The bitbucket repository `https://bitbucket.org/amal/alghamdi_thesis_experiment` contains the information, scripts and settings that are required for reproducing this thesis experiments.

Thesis Contribution

A number of people participated in the PetClaw project, lead by Professor David Ketcheson, adding new features and enhancing its design. Two papers were written and published by a number of PetClaw and PyClaw contributors to present those packages scope, design, performance and applications. One paper was published in SpringSim'11 conference proceedings and the other in SciPy 2011 conference, [3] and [1] respectively. Also, a third journal paper has been submitted to SIAM Journal on Scientific Computing (SISC) [2]. My contribution in this thesis research can be mainly summarized in the following three points:

- We demonstrate a use case of the advantageous use of Python in implementing easy-to-use modular extensible scalable scientific software tools by developing an implementation of a parallelization framework, PetClaw, for PyClaw using the well-known Portable Extensible Toolkit for Scientific Computation, PETSc, through its Python wrapper, petsc4py.

- We demonstrate the possibility of getting acceptable Python code performance when compared to Fortran performance after introducing a number of serial optimizations to PyClaw and PetClaw code including integrating Clawpack Fortran kernels into PyClaw for low-level computationally intensive parts of the code.

- We provide a demonstration of PetClaw scalability on up to the entirety of Shaheen; a 16-rack Blue Gene/P IBM supercomputer that comprises 65,536 cores, and located at KAUST.

We would like to mention that the initial design and implementation of the PyClaw framework is done by Kyle Mandli. The initial development of PetClaw including parallelization using petsc4py along with integrating Clawpack Fortran kernels into PyClaw is done as part of this thesis. Significant design, development and maintenance for PetClaw and PyClaw is done by David Ketcheson. Aron Ahmadia significantly contributed to PETSc integration, parallel I/O, programmatic testing framework, general design and setting the environment for running PetClaw on Shaheen. Other contributions and advanced features have been added by several contributors. Refer to the list in [4] for details.

Chapter I

Introduction and Background

Large-scale computational simulations were identified by Nobel laureate Ken Wilson as the third paradigm of scientific discovery joining the more traditional paradigms of theory and experiment [5]. Computational simulations enable obtaining accurate results for experiments that are difficult, very costly or time-consuming to physically experiment. They can also be used to predict situations in real-time such as weather forecasting or to perform experiments that are impossible in real life such as the evolution of the universe [6]. Experimental parameters can be controlled and changed computationally more easily than physically. In addition, experiments can be repeated hundreds of times while only paying the cost of computational execution time. The computational modeling paradigm, which emerged in the mid-20th century, has allowed for key scientific discoveries, major optimizations and solutions for many industrial, engineering and technological challenges, and enriched our understanding of natural phenomena and behavior for many physical systems.

Computational simulations inspire many challenges. Beside the mathematical and numerical complications involved in creating accurate models and suitable numerical methods for simulating those models, many technical software engineering challenges are encountered when developing high performance simulation tools.

The behavior of a phenomenon or a physical system that varies in both spatial and temporal dimensions is usually the subject of computational simulations. To conduct simulations with certain size, resolution and accuracy requirements, there might be a need for solving the system for a large spatial grid and for a large number of time steps. Moreover, the used numerical methods can be quite sophisticated to get the desirable accuracy. These requirements lead to the need for performing very expensive computations that require extensive computational resources. This introduces the need for highly optimized scientific software tools that are capable of utilizing the current advances in the computer architecture industry.

During the past few decades, significant research has been dedicated to optimizing and speeding up such scientific software tools. Exploiting distributed-memory parallel architectures, such as supercomputers and clusters, by porting these software tools to run on hundreds to thousands of cores has been one of the main strategies to achieve significant performance speed up. Nevertheless, the porting process involves several software engineering aspects. Parallelization can harm the code readability, generality and extensibility if it is not done in a careful manner.

In this thesis we first demonstrate a use case of using Python in implementing scalable scientific software tools by developing an implementation of a parallelization framework, PetClaw, for a widely used serial nonlinear wave propagation solver of hyperbolic conservation laws, Clawpack. We build the framework basically on PyClaw, a Python-based version of Clawpack, and introduce parallelization on the data level using the well known Portable Extensible Toolkit for Scientific Computation, PETSc, through its Python wrapper petsc4py. We show how Python influences the design, implementation, extensibility, scalability and usage of the tool.

Secondly, we demonstrate the possibility of getting acceptable Python code performance when compared to Fortran performance after introducing a number of serial optimizations to the Python code. These optimizations involve integrating Fortran

kernels into PyClaw for low-level computationally intensive parts and reducing redundant copy operations of the data. Thirdly, we provide a demonstration of the scalability of this Python-based scalable wave propagation solver, PetClaw, on up to the entirety of Shaheen, a 16-rack Blue Gene/P IBM supercomputer located at KAUST. Furthermore, we provide reproducibility information for all the computational experiments presented in this thesis.

In this chapter, we cover the mathematical and software background necessary for understanding the contributions of this work. In Section I.1, we introduce conservation laws and hyperbolic PDEs, the class of PDEs that PetClaw solves. We also describe the finite volume methods, which are the type of methods PetClaw uses, and introduce Clawpack as well. In Section I.2, we present and assess three parallel software tools that are based on Clawpack, mentioning some of their shortcomings that justify the need for PetClaw. We then discuss three approaches that can be followed to parallelize tools like Clawpack and PyClaw in Section I.3. After that we introduce Python and discuss the advantages of using it in scientific software tools in Section I.4; we also introduce numpy, a numerical Python package that we rely on, and f2py, a Fortran extension module generator for Python. PyClaw is introduced in Section I.5.

I.1 Hyperbolic PDEs and Finite Volume Methods

Many phenomena and physical systems can be modeled using systems of partial deferential equations, PDEs. Many fundamental PDEs can be classified into three categories: hyperbolic, parabolic and elliptic. This classification covers an important subset of of all the PDEs. Each of these classes has distinct behavior and models certain types of phenomena and physical behavior. Additionally, this classification is important for choosing suitable numerical techniques for solving the PDEs, as each

class requires different techniques.

Elliptic equations usually arise when modeling steady-state phenomena, where the physical system is in equilibrium and only the spatial dependency of the solution is considered. Parabolic and hyperbolic equations, on the other hand, are associated with time-dependent phenomena, where the solution varies in time and does not necessarily reach a steady-state. Parabolic equations model diffusion, such as the diffusion of heat, while hyperbolic equations model advective transport and wave motion phenomena [7].

In Sections I.1.1 and I.1.2, we highlight important details about hyperbolic PDEs and their relation to conservation laws. As our software tool is concerned with this particular class of PDEs, further discussion of elliptic and parabolic equations is beyond the scope of this thesis. In Section I.1.3, we introduce the Riemann problem, which forms a basis of the finite volume numerical methods discussed in Section I.1.4. Finally, the Clawpack software is introduced in Section I.1.5.

The remainder of this section is summarized from [8] and [9].

I.1.1 Conservation Laws

To have a better understanding of the hyperbolic class of PDEs, it is essential to understand the idea behind conservation laws which are strongly relevant to hyperbolic PDEs as we will see later. Assume that we have a fluid advecting in a one-dimensional pipe with a velocity $u(x,t)$, where x is the distance along the pipe and t is the time. Let us assume that u is known and we want to model the density $q(x,t)$ of a substance that have been added to the fluid in a very small amount. We measure the density of the substance by mass per unit length because we are assuming a one-dimensional case. At time t, the total mass of the substance between two points x_1 and x_2 is given

by

$$\int_{x_1}^{x_2} q(x,t)dx. \tag{I.1}$$

Now we are interested in studying the change of the mass over time in the section of the pipe bounded by x_1 and x_2. If the substance is neither being created nor destroyed in this section, the change of the total substance mass in this section will be only due to the flow of the substance particles through the boundary points x_1 and x_2, or the *flux*, and this is the concept of conservation. Assuming $F_1(t)$ and $F_2(t)$ are the fluxes at the end points x_1 and x_2 accordingly, we then have the following relation:

$$\frac{d}{dt}\int_{x_1}^{x_2} q(x,t)dx = F_1(t) - F_2(t). \tag{I.2}$$

The flux sign indicates the advection direction, we chose the convention that positive flux corresponds to advection to the right and negative flux corresponds to advection to the left. The flux magnitude, $|F_i(t)|$, represents the mass of the substance advecting through the point x_i in unit time. The Formula I.2 represents the *integral form* of a conservation law. By assuming a constant velocity where the flux depends only on the value of q and does not depend on x and t, the autonomous case of the conservation law can be written as follows:

$$\frac{d}{dt}\int_{x_1}^{x_2} q(x,t)dx = f(q(x_1,t)) - f(q(x_2,t)). \tag{I.3}$$

In fact, the flux in this case is the density q multiplied by the velocity u. Formula I.3, can be rewritten as

$$\frac{d}{dt}\int_{x_1}^{x_2} q(x,t)dx = -f(q(x,t))\Big|_{x_1}^{x_2}. \tag{I.4}$$

To be able to handle Formula I.4 by standard techniques of solving PDEs in order to solve for q, we derive the *differential form* of the conservation law from this formula assuming that $q(x,t)$ and $f(q)$ are sufficiently smooth for the derivation to be valid. By partially differentiating then integrating the right-hand side with respect to x, we get:

$$\frac{d}{dt}\int_{x_1}^{x_2} q(x,t)dx = -\int_{x_1}^{x_2} \frac{\partial}{\partial x} f(q(x,t))dx. \tag{I.5}$$

With further manipulation we get:

$$\int_{x_1}^{x_2} \big[\frac{\partial}{\partial t} q(x,t) + \frac{\partial}{\partial x} f(q(x,t))\big]dx = 0. \tag{I.6}$$

As this integral must be zero for any x_1 and x_2, then the integrated function must be zero as well which gives:

$$\frac{\partial}{\partial t} q(x,t) + \frac{\partial}{\partial x} f(q(x,t))dx = 0. \tag{I.7}$$

And this is the *differential form* of the conservation law. Using subscripts to denote partial derivatives, Formula I.7 becomes:

$$q_t(x,t) + f(q(x,t))_x = 0. \tag{I.8}$$

Note that this formula can represent a system of PDEs where q is a vector of m components and f is a vector-valued function.

I.1.2 Hyperbolic Systems of PDEs

As we mentioned earlier, hyperbolic systems of PDEs are suitable for modeling time-dependent phenomena and physical systems that involve wave motion or advective

transport of substances. In these systems we usually have the time t and one or more spatial variables x, y and z as the independent variables. Let us first consider the *differential form* of the conservation law given in I.8. This is a one dimensional, homogenous, first order system of PDEs and can be rewritten in a quasilinear form:

$$q_t + f'(q)q_x = 0, \tag{I.9}$$

where q is a vector of m components representing the unknown functions, or conserved quantities, such as velocity, pressure, mass, energy... etc. and f is the flux function. This system is classified as a hyperbolic system of PDEs if the flux Jacobian matrix $f'(q)$ is diagonalizable and has real eigenvalues.

If we consider the linear case, then the system I.9 can be written as

$$q_t(x,t) + Aq_x(x,t) = 0, \tag{I.10}$$

where A is a constant $m \times m$ real matrix that satisfies the same conditions for Jacobian matrix above. In the scalar case, where $m = 1$, A is a scaler and has to be a real number for the equation to be hyperbolic.

The three dimensional system corresponding to I.8 can be written as

$$q_t + f(q)_x + g(q)_y + h(q)_z = 0, \tag{I.11}$$

where f, g and h are the fluxes in the x, y and z directions accordingly. A hyperbolic system that is not in the conservative form can be as follows:

$$q_t + A(x)q_x = 0, \tag{I.12}$$

where the coefficient matrix A depends on x. This system has no flux function.

Note that the right-hand side of the Equations I.8, I.11 and I.12 is zero, indicating that there is no change in the total conserved quantities in the whole spatial domain as the time passes. However, *source terms* can be introduced in the right-hand side to model the cases where quantities are being added to or removed from the domain at a specific rate, ψ, as follows:

$$q_t + f(q)_x = \psi(q, x, t). \tag{I.13}$$

I.1.3 The Riemann Problem

The Riemann problem is an initial-value problem consisting of a hyperbolic equation with special initial data that is piecewise constant with a single jump discontinuity:

$$q^{\circ}(x) = \begin{cases} q_l & if \;\; x < 0 \\ \\ q_r & if \;\; x > 0. \end{cases} \tag{I.14}$$

Solving the Riemann problem results in a set of waves traveling at finite speeds. In the finite volume methods, these calculated waves and their speeds can be used to approximate the update to the solution in the next time step as we will see in Section I.1.4.

I.1.4 Finite Volume Methods

The finite volume method idea is based on dividing the spatial domain into intervals known as finite volumes or grid cells, as illustrated in Figure I.1; and keeping track of an approximation to the integral of q over each of these volumes. The value of Q_i is an approximation to the average value of q over the interval $(x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}})$ at time t_n as follows:

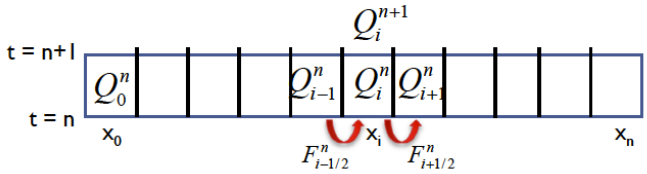


Figure I.1: Finite volume methods are based on dividing the spatial domain into intervals or volumes, each representing an approximation of q average over this interval.

$$Q_i^n \approx \frac{1}{\Delta x} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} q(x, t_n) dx. \tag{I.15}$$

In each time step, these values are updated using approximations to the flux through the endpoints of the intervals, $F_{i-\frac{1}{2}}^n$ and $F_{i+\frac{1}{2}}^n$, as shown in the following formula:

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}(F_{i+\frac{1}{2}}^n - F_{i-\frac{1}{2}}^n). \tag{I.16}$$

These approximations to the fluxes are found by approximately solving the Riemann problem between each two adjacent cells Q_{i-1} and Q_i and hence obtain a set of waves $\mathcal{W}_{i-\frac{1}{2}}^p$ and speeds $s_{i-\frac{1}{2}}^p$ for $p = 1, 2, ..., M_w$ where M_w is the number of waves in the Riemann solution. In many cases M_w equals the number of components in q, but this is not necessary. Note that the reason Riemann solutions are approximated is that solving the Riemann problem exactly might be too expensive in practice, therefore, approximate Riemann solvers can be used and they can give excellent computational results. From $\mathcal{W}_{i-\frac{1}{2}}^p$ and $s_{i-\frac{1}{2}}^p$ we can compute the left-going and right-going fluctuations, $\mathcal{A}^- \Delta Q_{i-\frac{1}{2}}$ and $\mathcal{A}^+ \Delta Q_{i-\frac{1}{2}}$, as follows:

$$\mathcal{A}^- \Delta Q_{i-\frac{1}{2}} = \sum_{p=1}^{M_w} (s_{i-\frac{1}{2}}^p)^- \mathcal{W}_{i-\frac{1}{2}}^p, \tag{I.17a}$$

$$\mathcal{A}^+ \Delta Q_{i-\frac{1}{2}} = \sum_{p=1}^{M_w} (s_{i-\frac{1}{2}}^p)^+ \mathcal{W}_{i-\frac{1}{2}}^p, \tag{I.17b}$$

where $(s)^- = min(s, 0)$ and $(s)^+ = max(s, 0)$. In Godunov-type methods, the methods that forms the basis of the algorithm in Clawpack, the values of the fluctuations $\mathcal{A}^-\Delta Q_{i-\frac{1}{2}}$ and $\mathcal{A}^+\Delta Q_{i-\frac{1}{2}}$ in I.17 are used to update the cell averages of q as follows:

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}\big[\mathcal{A}^+\Delta Q_{i-\frac{1}{2}} + \mathcal{A}^-\Delta Q_{i+\frac{1}{2}}\big], \tag{I.18}$$

which gives the first-order Godunov method. An advantage of Formula I.18 is that it can also be used to solve hyperbolic PDEs that are not in conservation form as apposed to the Formula I.16, which states that the difference between Q_i^{n+1} and Q_i^n is only due to the difference in the fluxes through the endpoints of the intervals.

To get higher order Godunov-type methods, limiters are applied to the waves $\mathcal{W}^p_{i-\frac{1}{2}}$ to get the limited waves $\tilde{\mathcal{W}}^p_{i-\frac{1}{2}}$ that are used to calculate high-resolution correction terms added to the right-hand side of I.18 as follows:

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}\big[\mathcal{A}^+\Delta Q_{i-\frac{1}{2}} + \mathcal{A}^-\Delta Q_{i+\frac{1}{2}}\big] - \frac{\Delta t}{\Delta x}\big[\tilde{F}_{i+\frac{1}{2}} - \tilde{F}_{i-\frac{1}{2}}\big], \tag{I.19a}$$

where

$$\tilde{F}_{i-\frac{1}{2}} = \frac{1}{2}\sum_{p=1}^{M_w} |s^p_{i-\frac{1}{2}}|(1 - \frac{\Delta t}{\Delta x}|s^p_{i-\frac{1}{2}}|)\tilde{\mathcal{W}}^p_{i-\frac{1}{2}}. \tag{I.19b}$$

I.1.4.1 The CFL Condition

The CFL condition is a necessary condition that must be satisfied by any finite volume method to ensure its stability. It is a constraint on the size of the time step that can be taken to ensure that information has a chance to propagate at the correct physical speed. To validate that the time step that has been taken satisfies the CFL condition,

the CFL number, denoted ν, is derived from the calculated wave speeds as follows:

$$\nu = \frac{\Delta t}{\Delta x} s_{max}, \tag{I.20}$$

where s_{max} is the largest calculated wave speed. For the method to be stable, the CFL condition $\nu \leq C$ must be satisfied, where C is an algorithm-dependent constant.

I.1.5 Clawpack

Clawpack (Conservation Laws Package) is a package that is developed to solve time-dependent linear and nonlinear systems of hyperbolic PDEs representing conservation laws in up to three space dimensions. It implements state-of-the-art high-resolution explicit Godunov-type methods that are a class of finite volume methods. These methods require Riemann solvers to resolve the jump discontinuity at the interface between two grid cells into waves propagating into the neighboring cells as described in Section I.1.4. Clawpack is flexible in the sense that it can handle all kinds of hyperbolic PDEs. It can also solve hyperbolic problems that are not in the conservative form. In addition, current extensions to cover parabolic PDEs are being developed [10].

This package has been actively developed, maintained, extended and documented since 1994 by a number of developers. Clawpack design has gone through iterations of enhancement to ensure design clarity, modularity and extensibility. The source code of the package is available free of charge for research and instructional purposes. The code availability has a considerable impact on the number of users of the package, where more than 7,000 users have registered to download it from its website since it first appeared in 1994; its availability also enables transparent reproducibility of the scientific experiments performed using Clawpack, a feature that is encouraged by a growing number of computational scientists [11].

Clawpack has been used by many users to simulate a wide variety of phenomena

and physical systems related to industrial, engineering and technological problems. To name but a few, it has been used in applications related to traffic flow, jet flow, tsunami modeling and semiconductor device simulations.

To set an application using Clawpack, the user provides the problem setup data and the required Riemann solver for the problem. The user also needs to provide any other problem-specific code for updating the boundary conditions of the solution q, initializing the solution, setting the system coefficients and/or adding any extra logic required at each time step. The user can make use of several commonly used Riemann solvers and boundary condition updating routines that are provided with the package.

I.2 Existing Parallel Codes Based on Clawpack

There are several efforts dedicated to providing parallel nonlinear wave propagation solvers such as the tools MPIClaw, ZPLClaw, BearClaw, FLASH and ChomboClaw. However, as new tools and software engineering trends and strategies are developed, scientific software tool design can be further improved. We present the tools MPIClaw, ZPLClaw and BearClaw; and some of their features in Sections I.2.1, I.2.2 and I.2.3, respectively. We also point to some drawbacks and enhancement opportunities in those tools design.

I.2.1 MPIClaw

MPIClaw provides parallelization for Clawpack by using MPI. Using this tool, one can expect to get a speed up of $0.8P$ to $0.9P$ if run using P processors [12]. This reflects the fact that the ratio of computation over communication is considerably high in this type of algorithms. However, MPIClaw has not been used on large numbers of processors [13].

Getting Clawpack application to work with MPIClaw should only require little additional work. However, the user needs to specify the number of processors in each dimension of the domain. We think this is an unnecessary complication that can be handled automatically without requiring user intervention.

MPIClaw provides alternative modified copies of some of the original Clawpack routines to handle the domain decomposition and communication among processes. These routines use the MPI standard routines to perform the communication tasks, create special temporary arrays for communicating ghost cells and boundary cells, and use common blocks for holding domain decomposition information. This parallelization approach results in using MPI basic routines such as send and receive functions throughout the code. We will see in Section II.2.2 why common blocks and temporary arrays might be harmful for the tool maintainability and writability. The idea of ghost cells and boundary cells is covered in Sections II.2.1 and II.4.2.

We think that this parallelization approach has two major drawbacks. First, new copies of the code are created introducing an overhead for maintainability and compatibility with the original code; updates to Clawpack cannot just be passed to MPIClaw, a manual update to MPIClaw will be required to keep the consistency. Second, injecting MPI specific code directly into the Fortran routines might harm the code readability and modularity. We also would like to mention that MPIClaw does not have support for solving for a source term when there is a global coupling between values [12].

I.2.2 ZPLClaw

ZPL is an array programming language designed with the goal of having optimized performance for both serial and parallel code. It simplifies parallel programming compared to using languages such as C or Fortran with MPI, yet gives a comparable performance [14]. ZPLClaw is a ZPL version of Clawpack. It has achieved

$29x$ speedup on 32-processor Cray T3E machine with shared memory communication library and $3.9x$ speedup on 4-processor Linux cluster using MPI library. One advantage of using ZPL is that it is a high-level language in the sense that it can be linked with the current machine communication library, such as MPI, while hiding the details of the protocols used for parallelization [15]. As ZPLClaw consists of a set of files that resembles Clawpack routines, ZPLClaw also has the same drawback we pointed out in MPIClaw, that it is not straight forward to keep the code compatible with Clawpack when new updates are introduced.

I.2.3 BearClaw

BearClaw, Boundary Embedded Adaptive Refinement for Conservation Laws, incorporates the best features of AMRClaw, Adaptive Mesh Refinement for Conservation Laws, and added the feature of parallelization using MPI. It is written in Fortran 90, and the designers tried to make a cleaner design than Clawpack and BearClaw to make future extensions easier [9]. BearClaw has achieved 70% speed up on 64 processors compared to the maximum theoretical speedup [16]. Actually, as BearClaw performs adaptive mesh refinement, achieving such performance can be a challenging task.

I.3 Approaches to Parallelization

Finding numerical solutions for problems of hyperbolic PDEs mainly involves stencil operations. Restricting our discussion to the structured grid case, parallelizing a serial code for this kind of problem using message-passing parallelization model requires the following generic tasks [17]:

- Define the work load partitioning among processes.
- Perform the serial computations (stencil operations).

- Prepare the data vectors that need to be communicated.
- Communicate those vectors.
- Update the local data in each process using the communicated vectors.

In this section we discuss three different approaches that can be followed to achieve the parallelization. The first one is the approach that was followed in MPIClaw and BearClaw which is to introduce parallelizing statements, including MPI standard routines, throughout the original serial code wherever it is required. Those statements include the logic for partitioning workload and declaring vectors for communication. They also include copying required slices from the local data into the communication vectors, using MPI `send` and `receive` operations to communicate those vectors and updating the local data using the communicated vectors. This approach has the disadvantage that injecting the parallel code throughout the serial code may harm its maintainability and readability; furthermore, the developer is responsible for all the mentioned parallelization tasks.

However, languages supporting modularity and class hierarchies, like Python, allow orderly separation of the serial and the parallel parts. Thus another approach suggested by X. Cai et al. [17] is to use a Python reusable parallelization class hierarchy that uses Python MPI bindings through tools like Pypar and pyMPI. We clarify X. Cai et al. approach by providing an idea about its implementation details that are discussed in [17]. Since four of the five mentioned tasks are independent of serial computations, the suggested class hierarchy, with a base-class named `BoxPartitioner`, is designed to implement these four tasks. A declared function in this class, named `prepare_communication`, is responsible for partitioning the workload and initializing the communication vectors, whereas the function `update_internal_boundaries` is for performing the last-mentioned three tasks. A first-level of subclasses of this base-class is developed to handle the cases of different number of space dimensionality,

namely, `BoxPartitioner1D`, `BoxPartitioner2D` and `BoxPartitioner3D`. Those subclasses have implementations to the function `prepare_communication`. Also, each one of those subclasses has one or more concrete subclasses to implement specific MPI operations such as those required in the function `update_internal_boundaries`. Each one of those concrete subclasses might adopt a different MPI wrapper module. For example, `BoxPartitioner2D` is subclassed to `PyMPIBoxPartitioner2D` and `PyParBoxPartitioner2D`.

One drawback of the first and second approaches is that the developer has to code the implementation of the five tasks. A third approach that avoids this drawback is to use libraries such as PETSc to achieve parallelization. PETSc is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the MPI standards for parallelism. It is intended to be used in large-scale computational applications and has proved to be scalable in several projects. PETSc parallel distributed arrays provided by the `DA` class object can be used for structured grid problems where communication of nonlocal data is needed before certain local computations can occur. For such problems, `DA` object relieves the user from worrying about most of the tasks involved in the parallelization process. Initializing the `DA` object will decide partitioning of workload, unless specified by the user. The `DA` object communication functions: `localToGlobal` and `globalToLocal`, prepare messages needed to be communicated, communicate those messages, and update the local data.

PETSc is written in the C language and can be used by C, C++ or Fortran code. However, Python-based wrappers for PETSc, petsc4py, have been developed to provide an enhanced Python interface for the PETSc library that matches Python interface style. Therefore, the approach of using PETSc to parallelize Python code can achieve both modularity and abstraction for the parallelization tasks. Furthermore, the developer can exploit the library features, such as parallel linear and nonlinear

equation solvers, to smoothly add new features that will be difficult to add otherwise. We have followed this third approach in the parallelization of PetClaw. The details of using PETSc `DA` object for parallelizing PetClaw is further discussed in Section II.4.2.

I.4 Python

Python is an object-oriented scripting programming language which we used in developing PetClaw. It has an extensive standard library and powerful open source third party packages for scientific computing such as numpy and SciPy. Python is powerful and efficient in working as a glue to easily integrate modules written in other languages such as Fortran, C and C++. It has a very clear, readable syntax that enables expressing the code structure, the control flow and the numerical computations very efficiently thus enhancing the programmer productivity. It supports full modularity of the code and hierarchical packages in addition to the powerful object oriented features. Therefore, Python enables a very clean, modular and extensible code design and implementation[18]. The advantages of using Python particularly in scientific software tools are discussed in Section I.4.1. We describe numpy and its data structure `ndarray`, which we use for holding multidimensional arrays in PetClaw in Section I.4.2. f2py, a Fortran to Python interface generator included in the numpy package, is discussed in Section I.4.3.

I.4.1 Python in Scientific Software

Scripting languages have become more popular in the scientific community. Python specifically has attracted significant attention among computational scientists for some years [17]. Python combines the advantages of scripting languages, but it is also suitable for large development projects [19]. This is due to its excellent soft-

ware engineering support. It was designed to be an object oriented language, unlike MATLAB where object oriented features were later introduced in the language [20]. Python allows for organizing the code into class hierarchies, packages and modules.

Among the scientific computing community, many reliable open source tools for simulation, analysis and visualization were developed separately over years by different research groups using different programming languages such as Fortran, C or C++. Those tools range from very general numerical solvers to specialized tools in specific applications area. However, in many contexts, researchers tend to rewrite functionalities that can be found in such tools instead of reusing them. This is due, in large part, to the difficulties involved in integrating several functionalities from different packages [21] [1]. Alternatively, they interface those tools through files at a program level as opposed to a function level. However, the latter solution does not allow the researcher to fully utilize those tools because it is not possible to directly combine functionalities from different tools to manipulate data in memory.

An approach that proved to be effective is to use a high-level language as a glue to interface those tools in a way that their functionalities can be accessed from a single script or code and data can be manipulated in memory by several tools without the need to write them to files [21]. Python has proved to be very efficient and convenient in playing this gluing role as it was designed with this goal in mind; it supports modularity and can be smoothly dynamically extended at runtime by importing extension modules and using their features [3]. Fortunately, several tools, such as f2py and SWIG, have been developed to largely automate the process of interfacing Python with other software packages written in other languages. And many well-known packages, such as PETSc and Trilinos, have provided Python interface to their libraries [3].

The practice of rewriting existing codes was further encouraged by another issue. Introducing some extensions in order to generalize, parallelize or otherwise improve

an existing package can be a very awkward process. This, sometimes, results from using languages that are limited in supporting software modularity and abstractions such as Fortran and C which results in a code that is difficult to be extended. Poor language readability results in a code that is difficult to be understood at the first place. This can, to a large extent, be avoided by using Python for implementing a well-designed scientific tools creating abstractions that are close to the problem formulation to facilitate future extensions [19]. However, although Python can be used for the high-level parts of the tool, it is not wise to use core Python in intensive computations because of performance considerations. These are addressed in Section III.2.1 together with the approaches of overcoming this challenge.

In addition, Python has very efficient built-in support for tasks usually needed in conjunction with scientific computations such as processing files, creating graphical user interface (GUI), I/O management, text processing, network utilization and interfacing with shell commands. Also, available powerful visualization libraries such as matplotlib can be used. The developer can also benefit from the self documenting capabilities, profiling, debugging features and testing frameworks. Python offers interactive shell as well, which is very handy in exploring and understanding the steps of a computational problem, and then the correct steps can be gathered in a single file and run as a program. The support for those tasks makes Python a mature development environment for computational scientists where simulation, analysis and visualization can be achieved through a single interface resulting in higher user productivity [17] [21] [19].

Python is also recommended for reproducible research, one aspect of which is clarity. The pseudocode-like syntax of Python has made it easier to share the executable Python code with publications as opposed to another approach followed in many papers and books, which is to include pseudocode to avoid listing a less readable Fortran or C code [11].

I.4.2 numpy and `ndarray`

numpy is an open-source Python library that provides a multidimensional array structure and other derived objects along with routines for fast optimized operations on these structures including layout manipulation and logical, basic linear algebraic and statistical operations. A core data structure in numpy is the `ndarray` object, which represents n-dimensional array of homogenous data types. Many of the available operations on this object are performed in a compiled code for optimized performance. numpy arrays are increasingly becoming a basic dependency in scientific computational packages [22].

In the development of PetClaw, we relay on `ndarray` for holding multidimensional arrays. An `ndarray` object has a buffer containing the raw data that can be laid out in memory in C or Fortran order, or any other strided scheme [23]. The user can specify which order is followed when creating `ndarray` object, or change the order later in execution. The number of dimensions and each dimension size is called the array shape. Indexing and slicing can be used to access the `ndarray` data [23]. By reshaping or slicing an array, views can be created to index the same array buffer in flexible different ways. If it was in a correct contiguous layout, `ndarray` data buffer can be accessed directly from a compiled code such as Fortran or C without the need for copying the data when passing the array through calls to compiled routines.

I.4.3 f2py

f2py is a Fortran to Python interface generator. It has been a part of the numpy package since 2007. It can generate Python extension modules from Fortran source code or signature files that are created either manually or using f2py itself. Using the generated extension modules, it will be possible to call Fortran 77/90/95 routines, Fortran 90/95 modules, access Fortran 77 common blocks and Fortran 90/95 module data from Python code. f2py automatically handles the differences in array layout

between Fortran arrays and numpy arrays in C layout. f2py greatly facilitates the process of interfacing Fortran to Python. The user is able to introduce changes to the generated interface and even do some preprocessing for the arguments before being passed by modifying the generated signature file. Some parameter specifications can be also added through f2py directives in the Fortran source code [24].

In this thesis research, low-level Fortran routines from the original Clawpack tool which perform the most computationally intensive parts of the code are integrated using f2py into PyClaw to achieve speedup.

I.5 PyClaw

PyClaw is a Python-based version of Clawpack. PyClaw currently provides solvers for 1D and 2D problems. The advantages of using Python particularly in scientific software tools which we discussed in Section I.4.1, have motivated the development of PyClaw.

PyClaw was carefully designed and developed with extensibility and usability goals in mind. Exploiting object oriented programming concepts and techniques, PyClaw abstracts the problem specification data using objects that resemble the numerical and physical concepts such as grid, dimension, solver objects. These design considerations help greatly in understanding and extending PyClaw by preserving the code readability and modularity as we will see in Chapter II. As Python is powerful in interfacing with other languages such as C and Fortran, the task of interfacing PyClaw with routines written in other languages is a relatively easy task. PetClaw is mainly an extension of PyClaw developed to achieve parallelization.

Chapter II

Implementation

One of the main contributions of this thesis research is to demonstrate a use case of using Python in implementing scalable scientific software tools, and how Python influences the design, implementation, extensibility, scalability and usage of the tool by developing an implementation of a parallelization framework, PetClaw, for a widely used serial nonlinear wave propagation solver of hyperbolic conservation laws, Clawpack.

In this chapter we discuss PetClaw implementation in detail. We start by giving an overview of PetClaw modular structure showing the several tools that have been integrated to develop PetClaw in Section II.1. Then we provide relevant details of Clawpack design and comment on the design complications inherited from using Fortran as Clawpack implementation language in Section II.2. After that we present PyClaw advantageous design that is made possible by using Python and discuss interfacing PyClaw with Clawpack Fortran kernels using f2py in Section II.3. In Section II.4, PetClaw implementation details are then discussed, focusing on the inheritance relationship between PyClaw and PetClaw, PetClaw data layout and the usage of PETSc package. We finally conclude this chapter by presenting a use case in Section II.5, demonstrating how PetClaw can be used to write an application script.

We would like to mention that the initial design and implementation of the PyClaw framework is done by Kyle Mandli. And the initial development of PetClaw including parallelization using petsc4py along with integrating Clawpack Fortran kernels into PyClaw is done as part of this thesis. Significant design, development and maintenance for PetClaw and PyClaw is done by David Ketcheson. Aron Ahmadia significantly contributes to PETSc integration, parallel I/O, programmatic testing framework, general design and setting the environment for running PetClaw on Shaheen. Advanced features that are out of the scope of this thesis and other contributions to PyClaw and PetClaw have been added by several contributors. Refer to the list in [4] for details.

Note that because PyClaw and PetClaw have gone through many improvements by several developers since the beginning of writing this thesis, the discussion of the implementation details holds true for the PyClaw github repository revision tagged as `alghamdi_thesis` (www.github.com/clawpack/pyclaw). This repository includes both PyClaw and PetClaw packages.

II.1 Modular Structure

PetClaw is built on top of petsc4py, Clawpack and PyClaw, relying on numpy for migrating data among those packages and using f2py to generate Fortran extension modules from Clawpack routines. Figure II.1 demonstrates an abstracted view of how those tools were combined to build PetClaw. The code that defines parallel structures and logic is contained in a thin layer of about 500 lines of code, which is the PetClaw package. The classes of the PetClaw package are inherited from the PyClaw package to add the necessary logic for parallelization. petsc4py is used in the PetClaw package to define the parallel structures and perform communication tasks. f2py is used to generate Fortran extension modules of Clawpack kernels that are used in the PyClaw

package. We provide more details about the integration of those tools throughout the remaining sections. numpy and Python were previously discussed in Section I.4.

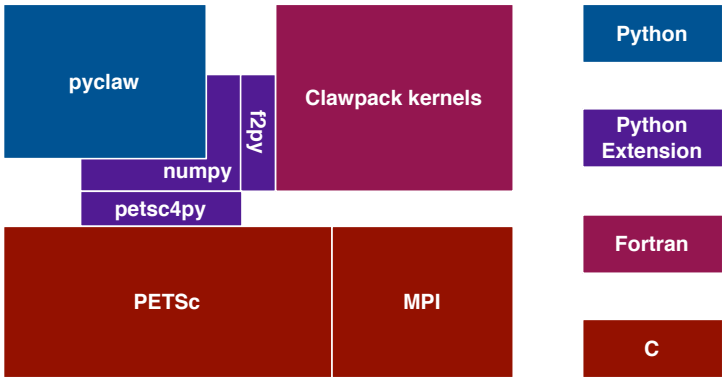


Figure II.1: Abstracted view of PetClaw modular structure. Image credit: A.J. Ahmadia [1].

II.2 Clawpack

In this section we briefly introduce the basic routines that constitute the classic Clawpack to give a high-level idea of how the Godunov-type methods are implemented in Clawpack. We focus on the 2D case. The 1D and 3D cases are quite similar. More details about Clawpack code can be obtained from the book by R. J. Leveque, "Finite Volume Methods for Hyperbolic Problems" [8], and "Clawpack 4.6 Documentation", [10].

In Section II.2.1, we focus on the control flaw and data structures of the numerical routines rather than mathematics to form a basis for further discussion of Clawpack design evaluation, interfacing PyClaw with Clawpack using f2py and parallelization. We neglect the discussion of low-level details and I/O routines as they are less relevant. We then provide our evaluation of Clawpack design focusing on the effect of the used programming language, Fortran, on its design in Section II.2.2.

II.2.1 Clawpack Design

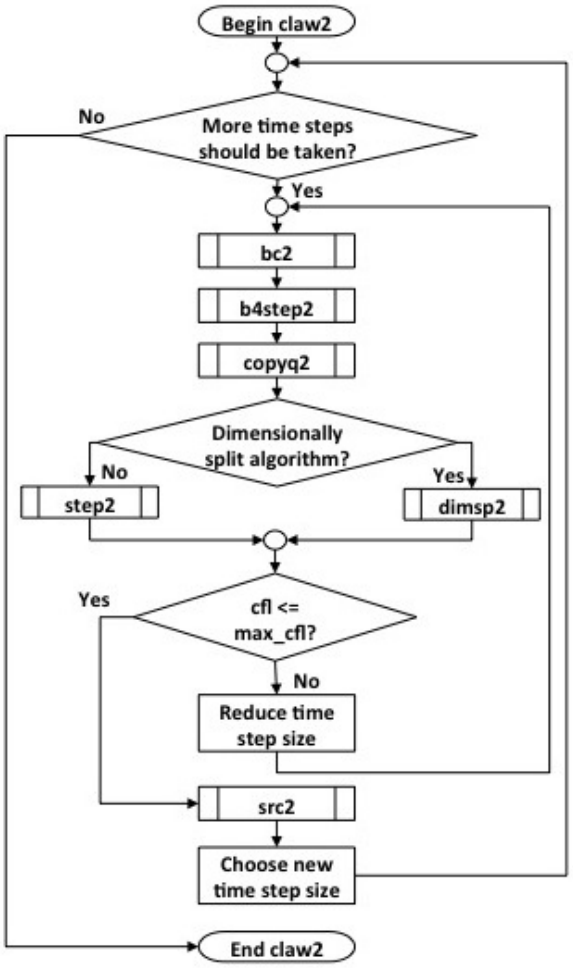


Figure II.2: High-level flowchart of `claw2` Clawpack routine.

Figures II.2 and II.3 present high-level flowcharts of Clawpack code showing its basic numerical routines. We start by describing `claw2` routine, shown in Figure II.2 , which includes the time stepping loop until the solution reachs the final time `tend` passed to `claw2` as a parameter. The time step size can be set by the user; and the user can determine whether the time step size should be fixed or adjustable by Clawpack. We will assume the second case in our discussion in this section.

To execute a single time step, a set of routines should be executed starting with `bc2` routine to update the boundary conditions using one of the predefined boundary

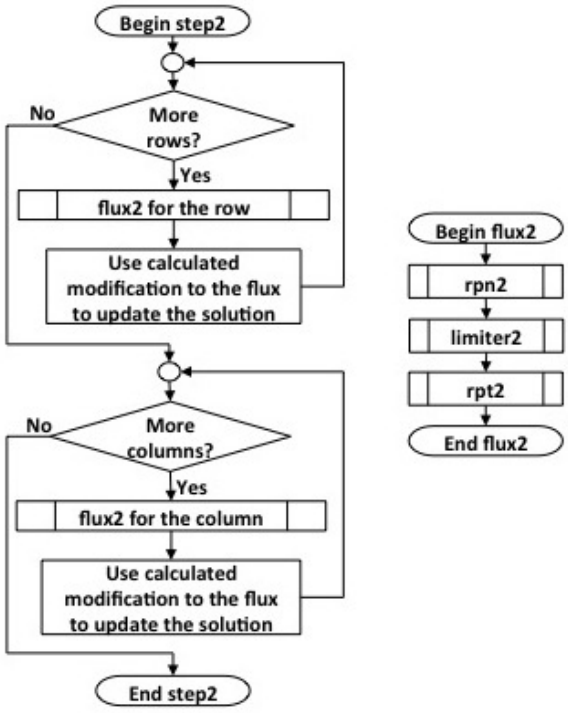


Figure II.3: High-level flowchart of `step2` and `flux2` Clawpack routines.

conditions update methods such as the periodic, the zero-extrapolation, etc. or a user-defined boundary conditions update method. Then the routine `b4step2` is called, this routine is problem dependent and should be written by the user if for example the problem has time dependent spatially varying coefficients that need to be updated each time step, or any other such tasks. The routine `copyq2` is then called to make a copy of q which is required by the non-split method and used to rollback in the case of rejecting a time step. Depending on the method used, the routine `dimsp2` or `step2` is called to evolve the solution one time step using dimensionally-split method or non-split method accordingly. After that, the largest CFL number, derived from the values of the waves speeds of the current time step, is compared to the maximum allowed CFL number which is algorithm dependent; the CFL condition was explained in I.1.4.1. If the calculated CFL exceeds the maximum allowed CFL, the step is rejected, the time step size is reduced and the step is retaken, otherwise, the step

is accepted. If the problem has source term as in Formula I.13, the routine `src2` is called. Then the time step size for the next time step is determined based on the calculated CFL number. This loop is repeated until the final time of the solution is reached. A slightly different logic happens in the case of using Strang splitting instead of Godunov splitting. However, Strang splitting is beyond our discussion here because it is not likely to be adapted in PyClaw as Godunov splitting is more efficient and gives almost as good accuracy as Strang splitting [8].

To better understand the details of evolving the solution one time step, we discuss the routine `step2`, sketched in Figure II.3. Two solution arrays, `qold` and `qnew`, are passed to this routine. When entering the routine, `qold` and `qnew` have the same values, the solution q from the previous time step. Upon exiting the routine, `qnew` will have the new solution value, and `qold` remains unchanged. Inside the routine, X-sweep is performed first by looping over the grid rows. In each iteration, `flux2` is called to generate the increments to q, `qadd`, corresponding to the second term of the right-hand side of Formula I.19a. `flux2` also generates the modifications to the flux, `fadd` and `gadd`, which are high-resolution correction terms generated at all interfaces along that row. These correction terms correspond to the third term of the right-hand side of Formula I.19a. `qadd`, `fadd` and `gadd` are then used in updating the new solution `qnew`. Then the Y-sweep is similarly performed by looping over the grid columns.

The `flux2` routine, sketched in Figure II.3 as well, is vectorized, works on a whole vector of the grid at once. The limiter, the normal Riemann Problem and the transverse Riemann Problem solvers, `limiter`, `rpn`, and `rpt` respectively, are vectorized too. Each grid cell requires information from a set of neighboring cells, referred to as stencil, in order to be updated. The algorithm stencil type can be either what is called star stencil or box stencil as shown in Figure II.4. The shaded grid cells are the ones included in the stencil. The dimensionally-split algorithm and

the non-split algorithm that does not apply transverse propagation needs star stencil, but when transverse propagation is applied, a box stencil is required.

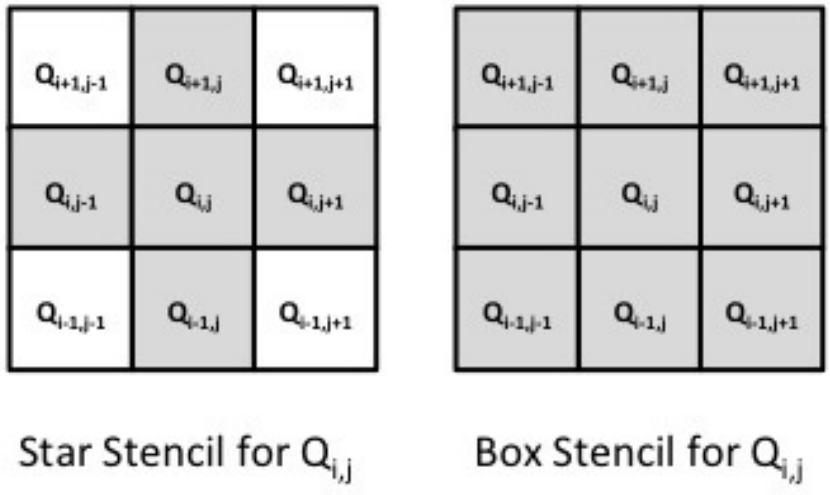


Figure II.4: Star and box stencil types for $Q_{i,j}$.

Clawpack implements boundary conditions using the so-called *ghost cells*, which are extra cells in the borders of the solution array, `q`, that are external to the solution's physical domain and hold the values of the boundary conditions. The advantage of this implementation is that the cells in the boundaries of the physical domain can be updated with the same logic used to update those cells in the interior of the physical domain. We refer to those ghost cells as boundary ghost cells to distinguish them from what we call interior ghost cells used in parallelization, which are discussed in Section II.4.2.

II.2.2 Clawpack Design Evaluation

Although Clawpack software has proved to be reliable and efficient over years of usage, it inherits some design limitations from its implementation language, Fortran 77. A major issue related to Fortran 77 code is handling the flow of data among subroutines either by using common blocks or passing data in and out as subroutine arguments.

Both approaches are used in Clawpack. However, common blocks are global data and misusing global data can lead to errors. In addition, a common block must be declared in each subroutine that uses this common block, which might introduce errors if there is inconsistency in the declarations. Also, introducing a change in a common block will require tracing all the declarations of that common block to make the same change.

The second approach results in very long, confusing subroutine calls [20]. For example, the call statement of `dimsp2` routine is as follows:

```
1       call dimsp2(maxm,maxmx,maxmy,meqn,mwaves,mbc,mx,my,
2      &                    work(i0qwrk1),q,aux,
3      &                    dx,dy,dt,method,mthlim,cfl,cflv,
4      &                    work(i0qadd),work(i0fadd),
5      &                    work(i0gadd),work(i0q1d),
6      &                    work(i0dtdx1),work(i0dtdy1),
7      &                    work(i0aux1),work(i0aux2),
8      &                    work(i0aux3),work(i0next),
9      &                    mwork1,rpn2,rpt2)
```

Note that due to the lack of ability to allocate memory dynamically in Fortran 77, even arrays such as `work`, which is used for intermediate storage, should be passed as a routine argument. It is clear that this long list of arguments can be a source of confusion and errors especially in the case of using a Fortran compiler that does not support type checking for subroutine arguments [20]. Also, as a result of static storage allocation, the code needs to be compiled with a maximum limit of the application required memory that cannot be exceeded during the run time.

The Fortran 77 language has almost no notion for data abstraction other than the common block and it has poor support for code modularity. This introduces difficulties in the code extensibility and readability and hampers the programmer productivity. These shortcomings have affected Clawpack design in many ways including the need for making three distinct libraries for the 1D, 2D and 3D cases, each needs to be modified separately for any extension or modification to be introduced.

An additional challenge is that the user needs some knowledge of Fortran when

providing customized `src`, `b4step`, `bc`, `rpn`, `rpt` routines is needed. All those challenges motivated the development of PyClaw.

II.3 PyClaw

In this section, we present the design of PyClaw and how we have interfaced it with Clawpack Fortran kernels. PyClaw includes the algorithms of Clawpack and SharpClaw. SharpClaw is based on a method of lines approach using weighted essentially non-oscillatory (WENO) reconstruction and high order Runge-Kutta methods [25][1]. However, we focus only on Clawpack solvers as SharpClaw is beyond the scope of this thesis. Designing PyClaw with the goal of extensibility in mind has later allowed PetClaw to be build smoothly on the top of PyClaw and allowed SharpClaw to be integrated with it. PyClaw design is a significant improvement over the design of its ancestor, Clawpack, in that it exploits Python object-oriented features along with packages and modules structures. The details of PyClaw design are illustrated in Section II.3.1. Integrating Clawpack Fortran kernels into PyClaw using f2py are explained in Section II.3.2.

II.3.1 PyClaw Design

PyClaw consists of several classes organized into modules and packages. A subset of these classes are shown in the UML diagram II.7. The classes `Solution`, `State`, `Grid` and `Dimension` are a realization of the PDEs system solution. The class `Solution`, represents the formulated problem's solution in a specific time. The `Solution` object owns one or more `State` class objects. Each `State` object holds solution array, `q`, and is associated with a `Grid` object which defines the grid layout. Having separate objects for defining the grid, the `Grid` object, and the `State` object, breaks the coupling between the grid layout definition on one side, and the actual solution and

variable coefficients data on the other side. As a result, a single grid layout definition can be used for more than one instance of data. Multiple `Grid` objects for a solution might be needed in contexts like adaptive mesh refinement (AMR) and multi-grid algorithms. Each `Grid` object is defined in terms of multiple `Dimension` objects as required by the problem. Figure II.5 shows a general view of PyClaw objects that compose a solution.

The `Solution` object can be passed to a `Solver` object's `evolve_to_time` method along with a final time to evolve the solution accordingly. The `Solver` class is a realization of the numerical solver, sketched in Figure II.6. It defines general functionalities that are common among solvers. Among these functionalities is the one achieved by the `setup` method that defines any logic that should take place before starting the solving process. `evolve_to_time` method is another `solver` class's method which loops over the time steps until reaching the final time resembling Clawpack `claw2` routine. The `solver` class's `qbc` method updates the boundary conditions resembling Clawpack `bc2` routine. Finally, the `solver` class's `step` method performs a single time step. The `solver` class also includes other methods.

Subclassing the `Solver` class is used to define variations of the solver. For example, we have the `ClawSolver`, inherited from the `Solver` class, which is a class for classic Clawpack solvers. We also have the `SharpClaw` class which represents other category of solvers. The `ClawSolver` class is further subclassed into `ClawSolver1D` and `ClawSolver2D` in order to define functionalities that should have different logic depending on the number of dimensions.

If Fortran kernels are not used, the `Solver` can be associated with one of the `rp` functions, the Riemann solver, contained in the riemann package and a limiter from the `limiters` package can also be associated to the `Solver`. The boundary conditions' type can be set in the `Solver` object by choosing from the `BC` class boundary conditions' enumeration. A user-defined boundary conditions can be used too.

Instead of using the `Solver` directly, the user can benefit from the `Controller` class object that controls the overall solution process including writing output to files on a frequency decided by the user. The desired format of writing can be passed as an argument to the `Controller` object which in turns uses the corresponding writing and reading routines available in the `io` package.

Note that while there was three distinct libraries in Clawpack with no code reusability, PyClaw solution's objects are generalized to handle three cases of dimensionality. The solver class is subclassed to only treat the different aspects of the 1D, 2D and 3D cases eliminating unnecessary redundancy that would cause difficulty in maintaining and extending PyClaw.

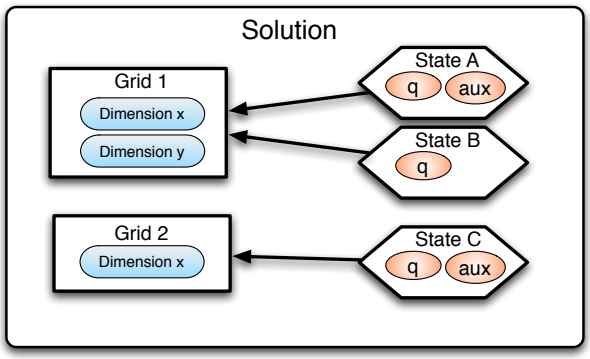


Figure II.5: PyClaw `Solution`. Image credit: D. I. Ketcheson [2].

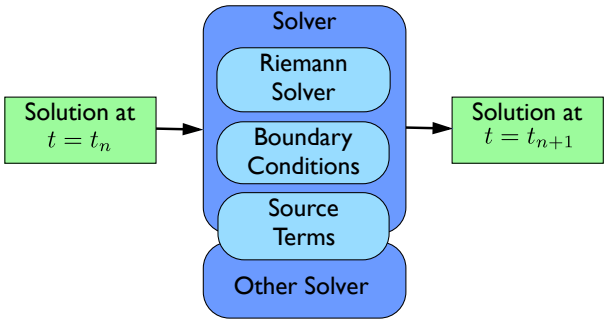


Figure II.6: PyClaw `Solver`. Image credit: K. T. Mandly [2].

II.3.2 Incorporating Clawpack Fortran Routines into PyClaw Using f2py

We have incorporated Clawpack Fortran kernels into PyClaw to execute computationally expensive segments of the solver algorithm as part of the work of this thesis. The purpose of this incorporation is to enhance the performance of PyClaw. This is because pure Python code, loops in particular, has poor performance when compared to compiled languages. We discuss Python performance in Section III.2.1. Another

advantage of this incorporation is code reusability [19]. Instead of rewriting the code in Python, we are incorporating the highly reliable Fortran routines that has been used for a long time by a large number of users.

We describe our decision of choosing the parts of the solver algorithm that will be replaced by Fortran extensions, that is the depth level of incorporating Fortran kernels, in Section II.3.2.1. Then we explain the technical details involved in generating Fortran extension modules and using them in the Python code in Section II.3.2.2.

II.3.2.1 Choice of PyClaw-Clawpack Interface Depth

We could choose interfacing PyClaw to Clawpack Fortran kernels from several possible depths of interfacing. However, we have chosen the step level to be the level of interfacing PyClaw code to Clawpack. Therefore, PyClaw code calls the wrapped Fortran routine `step2` in the case of none-split method or calls `dimsp2` in the case of dimensionally-split method; refer to Figures II.2 and II.3 for Clawpack flowchart. Those routines will perform one time step. As the Fortran routines `bc2`, `b4step2` and `copyq2` are relatively inexpensive and can be implemented in comparable efficiency using Python, they have not been integrated to PyClaw. A Python implementation of those routines is used in PyClaw instead. Thus the user can provide customized Python-based implementation for those operations. Furthermore, updating the boundary conditions requires parallel logic and communication. This is why this task should be kept in Python if we are to hide parallelization from the Fortran kernels. The routine `src2` can be either a wrapped Fortran code or implemented in Python. In fact, our tests showed that for a 2D Euler application, there was no significant performance difference between using Fortran and using Python src2 routines. Only a 2% performance loss was observed by using the Python-based source term. `src2` is local and does not require parallel logic or communication with other processes.

The routines `rpn2`, `limiter` and `rpt2` are the expensive parts of Clawpack algorithm. Another potential choice of interfacing is on the level of a pointwise version of the routines `rpn2`, `limiter` and `rpt2` which will make writing the Riemann solver routines simpler for the user and might simplifies potential improvements such as accelerating those routines using GPUs. However, this choice would require using Python loops to iterate over the solution array calling the pointwise routines `rpn2`, `limiter` and `rpt2` for each element of the array. The drawback of this choice is that Python loops can be from 10 to 100 times slower than the corresponding compiled Fortran or C code [20]. On the other hand, choosing much higher level of interfacing, `claw2` routine for example, is not adequate because there are several communication tasks that must be performed in each time step; we want to include these tasks in the Python code, as we will see in Section II.4.

There is a number of Fortran and Python Riemann solvers provided with PyClaw. However, if a customized Riemann solver is needed, the user has to write it in Fortran to benefit from the Fortran kernels. Alternatively, the user can use the provided Python kernels instead or provide a Riemann solver written in Python. Moreover, there are currently ongoing efforts to automate the generation of Riemann solvers based on a symbolic mathematical description to simplify this task [26].

II.3.2.2 Technical Details of Incorporating Clawpack Fortran Kernels into PyClaw

Python is implemented in C and designed to allow seamless C extensions. It provides what is known as Python C APIs to allow accessing and manipulating Python objects from C code and sending them back to Python. Therefore, integrating Fortran code with Python code requires writing a communication layer, called the *wrapper code*, in C using those APIs. Fortunately, tools like f2py will do the job of automatically generating this wrapping code in most of the cases. All the user needs to do is to make

adjustments to the interface description, written in Fortran 90 module interfaces format, that is also generated by f2py and can be edited to specify additional parameters' properties. Alternatively, the user can include those specifications as f2py directives in the Fortran source code [19], which is what we do in integrating Clawpack. Those specifications include, among other capabilities, optionally specifying whether each parameter is intended to be output, input or both input and output. By using the required source files, interface description, object files and wrapping code, f2py builds the Fortran extension module.

Consider the listed part of the subroutine `dimsp2` definition in Listing II.1. An f2py command that builds an extension module, named `classic2` for example, for this routine can be as follows:

```
f2py -m classic2 -c dimsp2.f  $(OTHER_SOURCES)
```

where `-m` option specifies the module name, `classic2`, and `-c` option specifies the required Fortran source and object files. We import this module in the Python code as follows:

```
import classic2
```

And from this module we access the wrapped Fortran routines and data stored in common blocks as follows:

```
classic2.cparam['u'] = 1
qnew, cfl = classic2.dimsp2(maxm,mx,my,mbc,mx,my, \
                        qold,qnew,aux,dx,dy,dt,self.method, \
                        self.mthlim,self.cfl,self.cflv, \
                        self.aux1,self.aux2,self.aux3,\
                        self.work)
```

where `cparam` is a common block declared in one of the source files and contains the variable `u`. `cparam` can be easily accessed from Python. The format of the function call of `classic2.dimsp2` does not differ from an ordinary Python function call. Even though Fortran does not support multiple return values, f2py provides a Python-based interface that looks as if it does. In this case, we specify `cfl` and `qnew` as

input/output parameters using f2py directives as shown in lines 29 and 30 of Listing II.1.

Listing II.1: `dimsp2` routine defined in `dimsp2.f`.

```
1   subroutine dimsp2(maxm,maxmx,maxmy,meqn,mwaves,
2      &                  maux,mbc,mx,my, qold,qnew,
3      &                  aux,dx,dy,dt,method,mthlim,cfl,cflv,
4      &                  qadd,fadd,gadd,q1d,dtdx1d,dtdy1d,
5      &                  aux1,aux2,aux3,work,mwork)
6
7       implicit double precision (a-h,o-z)
8       external rpn2,rpt2
9       double precision qold(meqn, 1-mbc:maxmx+mbc,
10     &                  1-mbc:maxmy+mbc)
11      double precision qnew(meqn, 1-mbc:maxmx+mbc,
12     &                  1-mbc:maxmy+mbc)
13      double precision  q1d(meqn, 1-mbc:maxm+mbc)
14      double precision cflv(4)
15      double precision qadd(meqn, 1-mbc:maxm+mbc)
16      double precision fadd(meqn, 1-mbc:maxm+mbc)
17      double precision gadd(meqn, 2, 1-mbc:maxm+mbc)
18      double precision aux(maux,1-mbc:maxmx+mbc,
19     &                  1-mbc:maxmy+mbc)
20      double precision aux1(maux,1-mbc:maxm+mbc)
21      double precision aux2(maux,1-mbc:maxm+mbc)
22      double precision aux3(maux,1-mbc:maxm+mbc)
23
24      double precision dtdx1d(1-mbc:maxm+mbc)
25      double precision dtdy1d(1-mbc:maxm+mbc)
26      integer method(7),mthlim(mwaves)
27      double precision work(mwork)
28
29  cf2py intent(in,out) cfl
30  cf2py intent(in,out) qnew
31  cf2py optional q1d, qadd, fadd, gadd, dtdx1d, dtdy1d
```

Although Fortran 77 does not support dynamic memory allocation, we can dynamically allocate memory in the Python code, as numpy `ndarray`, and pass those allocations to the incorporated Fortran routines' calls. We use the directive `cf2py optional` to make passing the temporary array storage through the function call optional. For example, line 31 of Listing II.1 specifies that passing the arrays `q1d`, `qadd`, `fadd`, `gadd`, `dtdx1d` and `dtdy1d` to `dimsp2` call is optional, which decreases the

number of the passed arguments.

Finally, we would like to mention that there are other available tools that facilitate generating extensions to Python by wrapping Fortran code, such as Fwrap and Pyfort. However, we have chosen f2py for its maturity and current support within the scientific computing community [3].

II.4 PetClaw

As shown in Figure II.7, the PetClaw package is mainly an extension of PyClaw developed to enable parallelism with maximum code reuse from PyClaw. Exploiting object oriented programming inheritance concept, we introduce the parallelization to the tool in a modular manner with minimal changes to PyClaw. We subclass PyClaw classes when necessary, overriding methods when parallel structures and logic should be introduced. Therefore, we avoid injecting parallelization statements all over the code, a practice that may harm maintainability and readability.

We start by discussing the data layout conflict issue raised when integrating petsc4py and Clawpack into PetClaw and PyClaw respectively in Section II.4.1 together with the approaches of overcoming this issue. We give a detailed explanation of how petsc4py have been used to enable the parallelization in PetClaw in Section II.4.2. After that we present PetClaw design focusing on PyClaw-PetClaw inheritance relationship in Section II.4.3.

II.4.1 Data Layout

When integrating several tools, one of the major issues that need to be resolved is how to interface the different data structures [20] and how to handle any data layout conflicts among the several tools. As we mentioned previously, in PyClaw the `ndarray` object is used for storing array data and that results in having data buffers

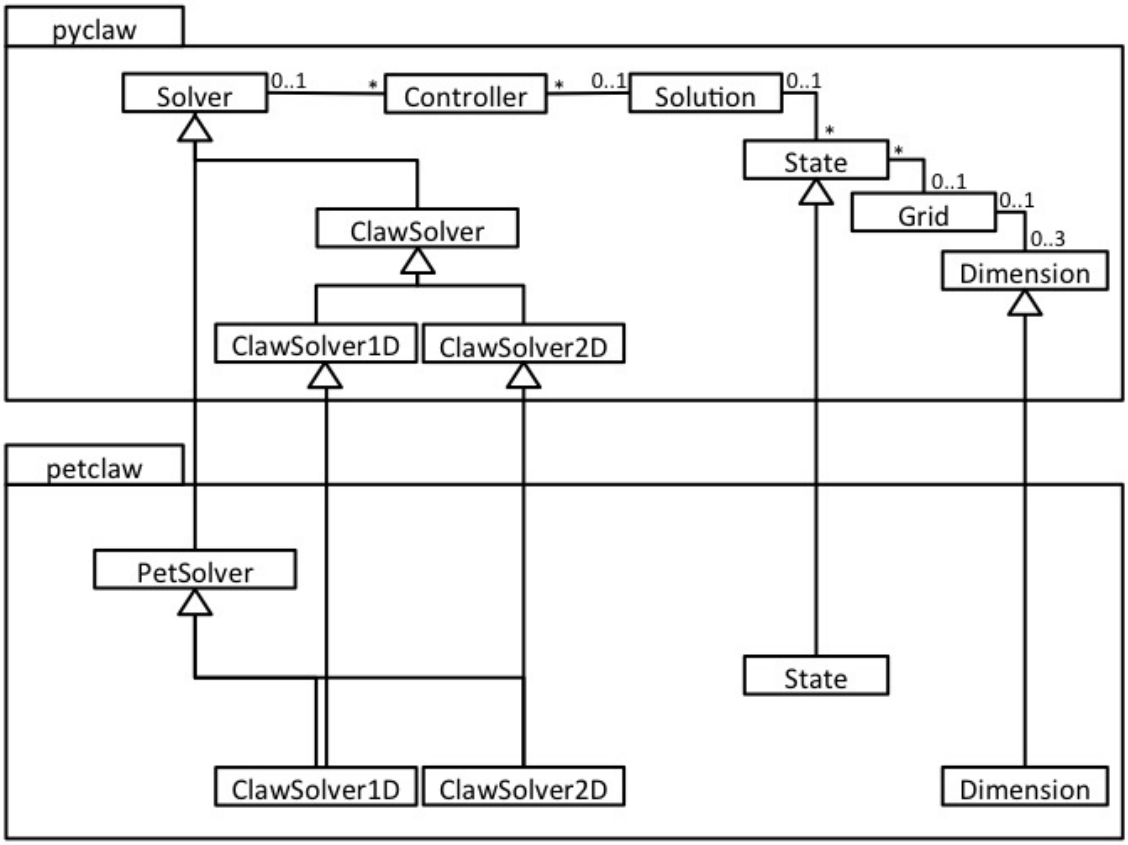


Figure II.7: UML diagram of PyClaw-PetClaw inheritance relationship.

compatible with the compiled languages. The solution array, `q`, has up to three spacial dimensions and an additional dimension for the different solution components. `PETSc.Vec` expects the data to have what we call *interleaved components*, that is for each grid cell, the different solution components of that cell are stored consecutively. Figure II.8(a) demonstrates how `PETSc.Vec` object expects a solution array `q` with two components v and p for a 2D problem to be laid out in memory. On the other hand, as Figure II.8(b) shows, Clawpack expects a *non-interleaved* arrangement of the data, that is the grid data of a single component are stored consecutively in memory followed by the data of the second component and so on.

One way to resolve this conflict is to adopt one of the two different tools' data

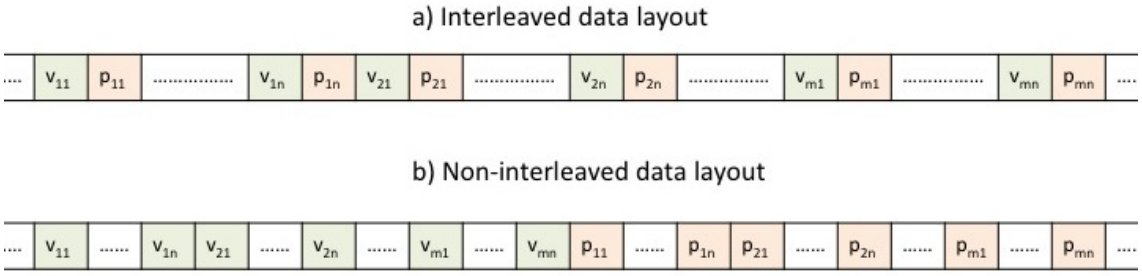


Figure II.8: Interleaved versus non-interleaved data layout for an $m \times n$ 2D solution grid with two components, v and p.

layout and make a copy of the data whenever it is passed to, or received from the other tool to match the corresponding layout. However, this solution can be inefficient if it results in large number of data required to be copied repetitively during the execution time causing noticeable performance degradation. Furthermore, if this solution was adopted, it needs to be implemented in a careful manner to ensure that the data are laid out correctly when passed to each tool.

Another solution for this issue that can work if those tools are open source, is to edit the source code of one tool and change the layout of storing data and the way they are accessed to match that of the other tool. This solution is clearly not ideal in many cases and violates important design principles: reusability, compatibility and maintainability. It might also require significant amount of code editing. In spite of all those disadvantages, we chose to modify Clawpack kernels so that the solution data, `q`, auxiliary (spatially varying coefficients) data, `aux`, waves, `wave`, and waves speed data, `s`, and other arrays used for intermediate storage are all interleaved. We think in our case this solution is justified for the following reasons:

- Clawpack basic kernels that we are interleaving are, to a large degree, stable as they have been proved reliable during years of usage. Therefore, we think maintainability might not be a big concern in this case as little chance that those kernels will be modified.

- Interleaving those kernels is reasonably feasible in terms of the time it takes to edit the code. All what it requires is changing the order of dimensions of the targeted arrays in the both cases of declaring the array and accessing it. For optimal performance, accessing those arrays through loops should be modified to respect the data spatial and temporal locality.

- In those kernels, the components of a particular grid cell are being accessed consecutively. thus we thought it might be more appropriate for those components to be consecutive in memory for better cache spatial locality. Furthermore, the interleaved layout is already adopted in AMRClaw that is based on Clawpack. As a result, the interleaved data layout is one of Clawpack planned modifications [27].Therefore, we expect there will be no compatibility issues in the future.

II.4.2 Data Partitioning and Code Parallelization with PETSc

In this section we explain how we use PETSc to implement data partitioning and parallelization in PetClaw. We use two classes from PETSc to achieve the parallelization, namely, PETSc distributed array, PETSc.`DA`, and PETSc vector, PETSc.`Vec`, classes. When running the application using x processes, the `DA` object decides the load partitioning among the x processes such that the grid is partitioned almost equally among processes and each process is assigned a part of the grid with minimal boundary to interior ratio to make the required communication minimal. In the 2D case, this ratio is minimal if the part of the grid that belongs to each process is close to a square shape as much as possible. In PetClaw, the data that need to be partitioned are the solution array, `q`, and the spatially varying coefficients, `aux`.

The `DA` object does not hold the grid data, it only stores information about the grid layout such as the grid partitioning, dimensions, number of boundary conditions, and whether those boundary conditions are periodic. We use the `DA` object to create two `Vec` objects with the proper size and partitioning, the global and the local vectors,

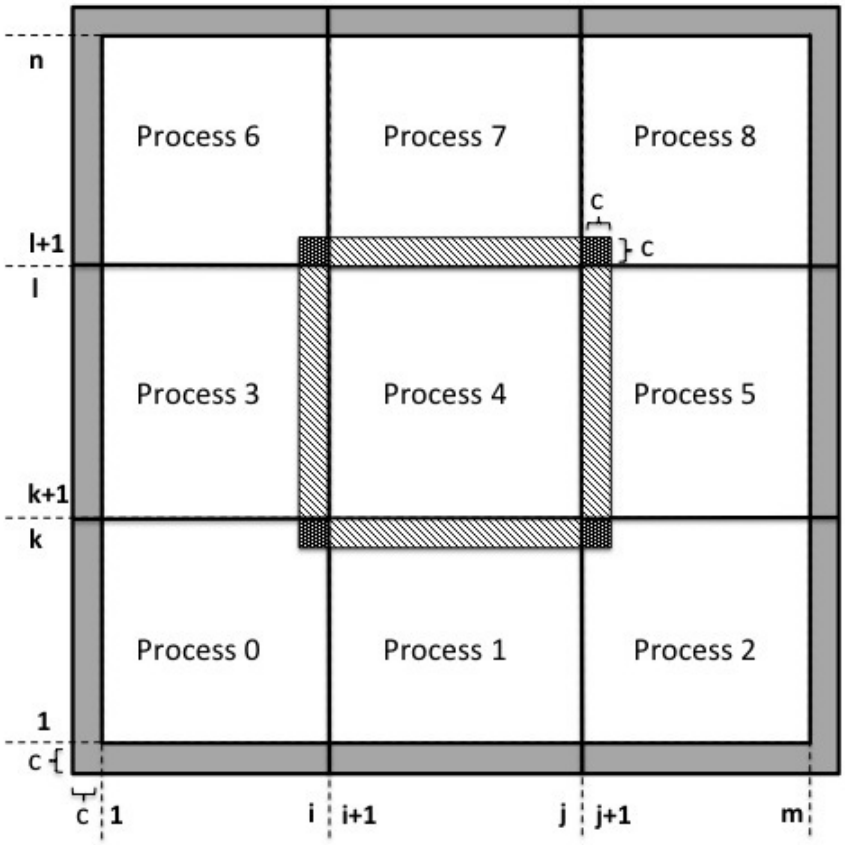


Figure II.9: Grid partitioning by the `DA` object.

which allocate memory for the grid array. In each process, the global vector allocates memory for the partition of the grid data owned by this process. However, when performing stencil operations on the grid cells that lie at the edges of the grid part associated to this process, we need information from surrounding cells that might belong to neighboring processes. To be able to handle the cells in the edges in the same way as handling interior cells, the local vector allocates storage that holds the same part of grid data held by the global vector but with additional ghost-cells bordering region. We call these cells *interior ghost cells* which hold a copy of part of the data that belong to neighboring processes but are required by the current process.

Line 2 of Listing II.2 shows how we create `DA` object to partition the data of `q`. `dim` is the number of dimensions; `dof` refers to the degree of freedom, i.e. the number

of `q` components; `sizes` is a list of the numbers of grid cells in each dimension, which in this case has only two elements as `dim` is 2; `periodic_type` defines whether the boundary conditions are periodic or not; `stencil_type` defines either box or star stencil; `stencil_width` is the number of boundary conditions (width of boundary and interior ghost cells region); and `comm` is the MPI communicator.

The creation of the `DA` object is executed by all processes since it is a collective operation. When being created, the `DA` object determines how the data should be partitioned based on the data size and the communicator size, *numprocs*. Figure II.9 shows how the `DA` defines the data partitioning for an $m \times n$ grid over nine processes. Process number 4, for example, will own a rectangular part, close to a square or possibly a square part, of the grid having $Q_{i+1,k+1}$ as the lower left corner grid cell and $Q_{j,l}$ as the upper right corner grid cell.

In line 10 of Listing II.2, we create a global vector which allocate memory for partitions of the grid in each process. For example, when process 4, shown in Figure II.9, calls that statement, the vector object will allocate an array that belongs to this process and has the size of $(j-i) \times (l-k) \times d$, where d is the number of components. We also create the local vector in line 11 which allocate storage for ghosted grid's partitions. For example, in the case of process 4 as shown in Figure II.9, the local vector array will be of size $(j-i+2\times c) \times (l-k+2\times c) \times d$. This includes the ghost cells region identified by the shaded area around process 4 box; c is the stencil width. The neighboring cells in the transverse order, shaded in dark gray, are updated only if the stencil type is box stencil. In PetClaw we use the default which is box stencil.

After taking a single time step, the data in the local vector will be updated. Ghost cells are usually read but not updated. Then we update the global vector using the new local vector data. This can be done by calling the method `localToGlobal` as in line 20 of Listing II.2. This method works entirely locally and does not communicate data with other processes. The data in the ghosted region of the local vector are

no longer valid and need to be updated from neighboring processes. In this step a communication among processes is required. We perfprm this communication by calling the `DA` method `globalToLocal` as shown in line 22.

The boundary conditions of the solution are implemented in the same way as in Clawpack using boundary ghost cells, described in section II.2.1, which is displayed in gray in Figure II.9, bounding the physical domain of the grid. The boundary ghost cells are not part of the physical domain. Hence they are allocated only if the user specifies the boundaries of the `DA` object to be periodic or ghosted. If the boundary conditions are specified to be periodic, a communication is necessary to update those boundaries. Fortunately, they will be updated automatically whenever `globalToLocal` method is called. For other boundary conditions' types, a specific code should be written to update them. This code should take into account that for a certain boundary, only some processes need to make the update. For example, updating the left boundary in Figure II.9 is the responsibility of processes 0, 3 and 6 only.

Listing II.2: `PETSc.DA` object usage.

```
1  # Create DA object for the q array
2  self.q_da = PETSc.DA().create(dim=2,
3                                dof=dof,
4                                sizes=[mx, my],
5                                periodic_type = periodic_type,
6                                stencil_type=stencil_type,
7                                stencil_width= c,
8                                comm=PETSc.COMM_WORLD)
9  # Create the local and the global vectors
10 self.gqVec = self.q_da.createGlobalVector()
11 self.lqVec = self.q_da.createLocalVector()
12 # .
13 # .
14 # Initializing q in the global vector.
15 # Make globalToLocal communication.
16 # Take a time step where the local array updated.
17 # .
18 # .
19 # Update the global vector
```

```
20 state.q_da.localToGlobal(state.lqVec,state.gqVec)
21 # Update the ghost cells of the local vector
22 state.q_da.globalToLocal(state.gqVec, state.lqVec)
23 # .
24 # .
```

After each process performs a single step on its partition of the data, a local maximum CFL number is found by each process. To find the global maximum CFL number, we create a `Vec` object holding one data item per process, which is the local maximum CFL number, refer to line 39 in Listing II.4. We then perform a maximum global reduction operation, line 40 in Listing II.4, to find the global maximum CFL number that is used to determine the validity of the taken step.

In PetClaw, the creation of `DA` and `Vec` objects takes place only once at the initialization of the application. See "PETSc Users Manual" [28] for technical information about using PETSc.

II.4.3 PetClaw Design

In this section, we discuss PetClaw design showing how parallelization of PyClaw is realized through inheriting PetClaw from PyClaw. We first talk about the inheritance of the solution classes in Section II.4.3.1. Then we talk about the inheritance of the solver classes in Section II.4.3.2.

II.4.3.1 Inheritance from PyClaw Solution Classes

Figure II.7 shows the inheritance relationship between the two packages PyClaw and PetClaw. We subclass the class `State` in PetClaw to add the methods `init_q_da` and `init_aux_da` to initialize the petsc4py `DA` objects, `q_da` and `aux_da` respectively, corresponding to the layout defined by the associated `Grid` object. As their names suggest, `q_da` is the `DA` object for the solution array and `aux_da` is the `DA` object for the variable coefficients array. Although these two `DA` objects have the same

dimensions, they might differ in the number of components, the reason why we made them distinct. These methods also create the corresponding global and local vectors for the `q`, `gqVec` and `lqVec`, and for the `aux`, `gauxVec` and `lauxVec`, respectively. Actually, it is almost unlikely that ghost cells communication will be needed for the `aux` array, however, having a `DA` object associated with `aux` array will simplify tasks such as setting the boundary conditions of the `aux` array, and in some applications, updating those boundary conditions. The `DA` object for the `aux` array is also used in writing `aux` from multiple processors to a single file using a petsc4py `Viewer` object. While the solution array `q` is merely a numpy array in the PyClaw package; in PetClaw, we override it to be a property as shown in Listing II.3. We implement this property so that each time `q` is accessed, the `ndarray` belonging to `q_da` global vector, `gqVec.array`, is retrieved and reshaped properly, line 6, as the `Vector` object will return a flat array. And whenever `q` must be set with an `ndarray`, that array is reshaped to be flat, as petsc4py expects, and used to set `gqVec.array`. We also override `aux` array to be a property in PetClaw class `State` in a similar manner.

Listing II.3: PetClaw `q` property.

```
1  def q():
2          def fget(self):
3              if self.q_da is None: return 0
4              q_dim = self.grid.ng
5              q_dim.insert(0,self.meqn)
6              q=self.gqVec.getArray().reshape(q_dim,order='F')
7              return q
8          def fset(self,q):
9              meqn = q.shape[0]
10             if self.gqVec is None: self.init_q_da(meqn)
11             self.gqVec.setArray(q.reshape([-1],order='F'))
12         return locals()
13     q = property(**q())
```

We subclass the PyClaw class `Dimension` in PetClaw to redefine the property `ng` which holds the size of the dimension. The overwriting definition of `ng` is the size of the portion of the dimension associated to the current process. The PyClaw

`Dimension` class properties `center` and `edge` are used for creating an `ndarray` of the locations of all grid cells center coordinates and edge coordinates respectively. We overwrite them in PetClaw `Dimension` class so that in each process, calling one of these properties will construct the required coordinates locations array for the portion of the `Dimension` associated to the calling process.

II.4.3.2 Inheritance from PyClaw Solver

The PyClaw `Solver` class defines the general interface of how any solver should be implemented by including dummy methods that should be overridden by subclasses. It also includes fully implemented methods that are general enough to be inherited and used by subclasses. As shown in Figure II.7, PetClaw solver class `PetSolver` is inherited from PyClaw `Solver` class to introduce the logic required for managing distributed-memory structures.

The `PetSolver` class basically contains the overriding methods listed in II.4. The method `append_ghost_cells`, as its name indicates, is used in PyClaw to append the boundary ghosted region to q. In PetClaw, this method does the corresponding parallel job by simply retrieving the array of the local vector, which contains the interior, and for some processes the boundary, ghosted region. The method `append_ghost_cells_to_aux` works in a similar manner but on the `aux` array.

The method `qbc` is responsible for updating the boundary conditions. We modified the code of PyClaw `qbc` method to make it general to work for both the serial and the parallel cases; the modified code is shown in Listing II.5. `qbc` method calls `append_ghost_cells` method, line 14, to get the ghosted `q`. The method tests whether the current process has a boundary region or not by a simple condition, lines 20 and 37. If the process partition of the grid has a boundary region, `qbc` will call one or more of the methods `qbc_lower`, `mthbc_lower`, `qbc_upper` and `mthbc_upper` to update the proper boundaries.

The PyClaw solver method `evolve_to_time` calls `qbc` to get `q` with updated ghosted region. The method `homogeneous_step` will work on this ghosted `q` to evolve it one time step updating the interior region of the ghosted `q`. Then the method `set_global_q` is used by `homogeneous_step` to update the values of the original `q`, without ghost cells, using the interior region of the updated ghosted copy. In PyClaw, this final update is done by slicing. In PetClaw, slicing can be used as well in the same manner, however, the implementation shown in line 12 of Listing II.4 is preferred for performance considerations that are discussed in Section III.2.3.

We override the method `communicateCFL` in PetClaw to perform a maximum global reduction operation to find the global maximum CFL number form the processes' local maximum CFL numbers, so that it can be used in deciding the step validity. As this communication is not needed in the serial implementation, the corresponding PyClaw implementation for `communicateCFL` simply does nothing.

Listing II.4: `PetSolver` class methods.

```
1  def append_ghost_cells(self,state):
2      """
3      Returns q with ghost cells attached.  For PetSolver,
4      this means returning the local vector.
5      """
6      state.q_da.globalToLocal(state.gqVec, state.lqVec)
7      q_dim = [n + 2*self.mbc for n in state.grid.ng]
8      q_dim.insert(0,state.meqn)
9      ghosted_q=state.lqVec.getArray().reshape(q_dim,order='F')
10     return ghosted_q
11
12 def set_global_q(self,state,ghosted_q):
13     """
14     Set the value of q using the array ghosted_q. for
15     PetSolver, this involves setting ghosted_q as the
16     local vector array then perform a local to global
17     communication.
18     """
19     state.lqVec.placeArray(ghosted_q)
20     state.q_da.localToGlobal(state.lqVec,state.gqVec)
21     state.lqVec.resetArray()
22
23 def append_ghost_cells_to_aux(self,state):
```

```
    """
    Returns aux with ghost cells attached. For PetSolver,
    this means returning the local vector.
    """
    state.aux_da.globalToLocal(state.gauxVec, state.lauxVec)
    aux_dim = [n + 2*self.mbc for n in state.grid.ng]
    aux_dim.insert(0,state.maux)
    ghosted_aux=state.lauxVec.getArray().reshape(aux_dim,\
                order='F')
    return ghosted_aux

def communicateCFL(self):
    from petsc4py import PETSc

    if self.dt_variable:
        cflVec = PETSc.Vec().createWithArray([self.cfl])
        self.cfl = cflVec.max()[1]
```

Listing II.5: PyClaw `qbc` method.

```

def qbc(self,state):
    r"""
    Appends boundary cells to q and fills them with
    appropriate values.
    .
    .
    .

    """

    import numpy as np

    qbc=self.append_ghost_cells(state)
    grid = state.grid

    for idim,dim in enumerate(grid.dimensions):
        # First check if we are actually on the boundary
        # (in case of a parallel run)
        if dim.nstart == 0:
            # If a user defined boundary condition is being
            # used, send it on, otherwise roll the axis to
            # front position and operate on it
            if self.mthbc_lower[idim] == BC.custom:
                self.qbc_lower(grid,dim,state.t,qbc,idim)
            elif self.mthbc_lower[idim] == BC.periodic:
                if dim.nend == dim.n:
                    # This process owns the whole grid
```

```
29                      self.qbc_lower(grid,dim,state.t,np.\
30                                      rollaxis(qbc,idim+1,1),idim)
31                  else:
32                      pass #Handled automatically by PETSc
33              else:
34                  self.qbc_lower(grid,dim,state.t,np.\
35                                  rollaxis(qbc,idim+1,1),idim)
36
37          if dim.nend == dim.n :
38              if self.mthbc_upper[idim] == BC.custom:
39                  self.qbc_upper(grid,dim,state.t,qbc,idim)
40              elif self.mthbc_upper[idim] == BC.periodic:
41                  if dim.nstart == 0:
42                      # This process owns the whole grid
43                      self.qbc_upper(grid,dim,state.t,np.\
44                                      rollaxis(qbc,idim+1,1),idim)
45                  else:
46                      pass #Handled automatically by PETSc
47              else:
48                  self.qbc_upper(grid,dim,state.t,np.\
49                                  rollaxis(qbc,idim+1,1),idim)
50
51      return qbc
```

As PetClaw is a parallelization of PyClaw and is based on the same numerical methods of Clawpack, `PetSolver` subclasses also inherit most of the logic from PyClaw `Solver` class. PetClaw classes `ClawSolver1D` and `ClawSolver2D` currently only override the setup method that is called in the beginning before starting evolving the solution to add insignificant modifications that we are not covering here.

II.5 Use Case: Setting Up a PetClaw Example

In this section, we present a use case scenario of setting and running an application problem using PetClaw. We chose a 2D homogenous acoustics application with a smooth radial hump as its initial data. Similar application can be found in `pyclaw/apps/acoustics/2d/homogeneous/acoustics.py`. The example directory, `pyclaw/apps/acoustics/2d/homogeneous`, includes a make file which can be used with the script illustrated in this section. By issuing the `make` command, Fortran

extension modules `classic2.so` and `sharpclaw2.so` will be generated. In this example, the module `classic2.so` is the only one used. The make file points to Fortran files of the Riemann solver routines, `rpn2_acoustics.f` and `rpt2_acoustics.f`, that are part of the riemann package and used in generating the module `classic2.so`. We start by giving a very basic script and then describe more advanced settings.

First, the numpy library is imported

```
1 import numpy as np
```

Then if the user wants to run PyClaw serial code only, PyClaw should be imported as follows:

```
import pyclaw
```

And if the user wants to use PETSc for parallelization, PetClaw should be imported instead as follows:

```
import petclaw as pyclaw
```

Note that giving `petclaw` a local name, `pyclaw`, is an effective abstraction to the fact that the user actually uses PetClaw package not PyClaw. And this is the only difference in setting PyClaw or PetClaw basic example.

```
3 # Set the solver
4 solver=pyclaw.ClawSolver2D()
5 solver.mwaves = 2
6 solver.mthbc_lower[0]=pyclaw.BC.reflecting
7 solver.mthbc_upper[0]=pyclaw.BC.outflow
8 solver.mthbc_lower[1]=pyclaw.BC.reflecting
9 solver.mthbc_upper[1]=pyclaw.BC.outflow
```

After that the user should create the solver object as in line 4. The number of waves resulting from splitting the jump in Riemann problem solver is set by the variable `solver.mwaves`. The variable `solver.limiters` used to set the wave limiter type which is the monotonized central-difference, MC, limiter in this case. Then in lines 6 through 9 the user sets the type of the boundary conditions for each dimension, index 0 for the first dimension and 1 for the second dimension. The user can determine the initial time step size by setting the variable `solver.dt_initial` with the required

step size. The default solver algorithm is the dimensionally-split algorithm, to use the non-split algorithm the user should unset the variable `solver.dim_split`.

```
10  # Initialize the grid
11 mx=100; my=100
12 x = pyclaw.Dimension('x',-1.0,1.0,mx)
13 y = pyclaw.Dimension('y',-1.0,1.0,my)
14 grid = pyclaw.Grid([x,y])
15 state = pyclaw.State(grid)
```

The lines from 11 through 15 initialize the grid. Variables `mx` and `my` are the numbers of the grid cells in each of the dimensions x and y respectively; they are used in the next two statements to create `Dimension` objects `x` and `y`. The `Dimension` object initializer's arguments are the string of the dimension name, followed by two doubles, the beginning and the end of the dimension domain, and finally an integer representing the number of grid cells. The list of dimensions, `[x,y]`, are then passed to the `Grid` class initializer to create the `Grid` object, `grid`. `grid` in turns is passed to the `State` class initializer to create an object `state`.

```
16 # Set global auxiliary parameters.
17 rho = 1.0
18 bulk = 4.0
19 cc = np.sqrt(bulk/rho)
20 zz = rho*cc
21 state.aux_global['rho']= rho
22 state.aux_global['bulk']=bulk
23 state.aux_global['zz']= zz
24 state.aux_global['cc']=cc
```

In lines 17 to 24, problem specific parameters required by the Riemann solver are set to be contained in the auxiliary global parameters dictionary, `state.aux_global`. These parameters, required by the acoustics Riemann solver, are the variables `rho`, `bulk`, `cc` and `zz` standing for density, bulk modulus, sound speed and impedance of the medium, respectively.

```
25 # Initialize the solution array q.
26 state.meqn = 3
27 Y,X = np.meshgrid(grid.y.center,grid.x.center)
28 r = np.sqrt(X**2 + Y**2)
29 width=0.2
```

```
30 state.q[0,:,:] = (np.abs(r-0.5)<=width)\
31                  *(1.+np.cos(np.pi*(r-0.5)/width))
32 state.q[1,:,:] = 0.
33 state.q[2,:,:] = 0.
```

Initializing the solution array `q` is taking place in lines 26 to 33 starting by setting the number of components, `state.meqn`, to be three, for pressure, velocity in X direction and in the Y direction. Actually, the `meqn` was overridden to be a property of the PetClaw `State` subclass so that when `meqn` is set, the property function will check whether the `q_da` was initialized or not. if `q_da` is not set, the method `init_q_da` will be called initializing `q_da`. Then two coordinate matrices `X` and `Y` will be created using the center coordinates vectors `grid.y.center` and `grid.x.center`. Recall that those coordinate vectors will be different in each process, giving only the portion of the coordinates owned by that process. Then each component of the `state.q` is initialized separately.

```
34 # Set the controller to run the solver.
35 claw = pyclaw.Controller()
36 claw.solution = pyclaw.Solution(state)
37 claw.solver = solver
38 claw.tfinal = 0.6
39 claw.output_format = "petsc"
40 status = claw.run()
```

In lines 35 to 40, we create a `Controller` object `claw` and set some of its properties. `state` is passed to the `Solution` class initializer to create a `Solution` object that is used to set `claw.solution`. `claw.solver` is set with the solver variable, `solver`, created in line 4. The final time we wish to evolve the solution to is set using the variable `claw.tfinal`. If PetClaw is used, then the variable `claw.output_format` should be set to `"petsc"` to manage writing from different procceses. Finally, to run the simulation, the method `claw.run` is called resulting in evolving the solution to the final time and writing the outputs to a directory named `_output` by default. If a specific output directory name is required, the user should set the variable `claw.outdir` with the desired name.

If HTML plots are required, the user can make the following call:

```
pyclaw.plot.plotHTML(outdir=claw.outdir,\
                                format=claw.output_format)
```

Alternatively, if the user needs an interactive plot session, the following call will do the job:

```
pyclaw.plot.plotInteractive(outdir=claw.outdir,\
                                format=claw.output_format)
```

The user can execute the script with the command:

```
python acoustics.py
```

assuming the script file name is `acoustics.py`. If the user wants to use PETSc with sixteen processes for example, a command like `mpiexec` or `mpirun` can be used as follows:

```
mpiexec -n 16 \python acoustics.py
```

If the application needs specific handling for the boundary conditions, has source term or needs extra processing required in each time step, specialized functions can be added in the same script file and associated with the solver as shown in Listing II.6. We assume that the functions `custom_bc_lower`, `custom_bc_upper`, `application_src` and `custom_start_step` are the user defined functions written to handle lower boundary conditions, upper boundary conditions, source term and any logic required in the beginning of each step, respectively.

Listing II.6: Setting user defined functions.

```
solver.user_bc_lower = custom_bc_lower
solver.user_bc_upper = custom_bc_upper
solver.src= application_src
solver.start_step = custom_start_step
```

In this section we have seen a use case of how to use PetClaw and how it is consistent with using PyClaw. All the parallelization details are hidden from the user. In addition, this use case demonstrates the power of Python as an integrated development environment. Setting the problem, running the script, writing to files

and visualization took place in the same script. Also, additional data analysis and other tasks can be easily added. The user can also run the PyClaw script interactively or in the debug mode in the same environment.

Chapter III

PetClaw Performance

In this chapter we discuss PetClaw performance. We start by defining the performance metrics we have used to assess PetClaw serial and parallel performance in Section III.1. Then we discuss some serial optimization techniques that we have considered in developing PetClaw in Section III.2 along with presenting a serial performance comparison of selected PetClaw and Clawpack applications. Finally, we present a detailed scalability study of two PetClaw representative applications on up to the entirety of Shaheen to demonstrate both the strengths and weaknesses of our tool's parallel performance in Section III.3.

III.1 Performance Metrics

In this section, we define the performance metrics that we have used to assess the serial and parallel performance of PetClaw and to inform our performance optimization process. We define speedup in Section III.1.1, then the metrics we refer to as strong and weak scalability and parallel efficiency in Section III.1.2.

Those metrics are all based on the execution time of the code. To measure the execution time of several parts of the running code, we have used profiling and timing-statements instrumentation. In the parallel case, the execution time measurements we

have considered are those of process 0, assuming frequent synchronization in PetClaw applications. We also profile process 5 to verify the results we get from process 0.

III.1.1 Speedup

In general, the speedup gained by using a program A compared to using program B can be found by the following formula:

$$\text{Speedup of } A \text{ over } B = \frac{\text{Execution time of B}}{\text{Execution time of A}}. \tag{III.1}$$

Given the total execution time of some code before and after enhancement, the overall speedup is defined as follows:

$$\text{Speedup} = \frac{\text{Execution time of the entire code before enhancement}}{\text{Execution time of the entire code after enhancement}}. \tag{III.2}$$

The performance gain obtained by a serial optimization is governed by Amdahl's Law, which states that the performance gain obtained by optimizing a portion of the code is limited by the fraction of execution time during which this portion is being executed. Therefore, the overall speedup in a program that can be achieved as a result of enhancing part or feature of the code will depend on two factors: the fraction of the total program execution time the enhanced portion consumes (prior to enhancement), denoted by $Fraction_{enhanced}$, and how much faster this portion of the program becomes (post-enhancement), $Speedup_{enhanced}$. The relation between these factors and the overall speedup can be shown as follows:

$$\text{Overall speedup} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}. \tag{III.3}$$

This formula suggests that the focus in optimizing the serial performance should be on the parts that take most of the program execution time as speeding up those parts

will have a larger impact on the overall performance of the code [29].

In the parallel case, the definition of speedup can be specified as follows:

$$\text{Speedup} = \frac{\text{Serial execution time}}{\text{Parallel execution time}}, \tag{III.4}$$

for the same problem and input data. When running the program on P cores, the speedup is expected to be less than or equal to P. However, getting a speedup equal to P, a case known as linear speedup, rarely happens because of the communication overhead, process idle time and extra computations required for the parallel run [6]. In some cases, a superlinear speedup can be achieved, speedup $> P$. This can happen, for example, when the size of the problem per process decreases as more processors are used for the run. In this case, a larger portion of the data will fit into the cache. If the performance gain resulted from fitting more data into the cache outweigh the parallelization overhead, superlinear speedup can be achieved.

III.1.2 Parallel Scalability and Efficiency

Efficiency is a measure that shows resource utilization, in the very general case the total number of consumed CPU cycles over all processors participating in a simulation, in a parallel run compared to some base serial run. One formula for parallel efficiency is:

$$\text{Parallel efficiency} = \frac{\text{Speedup}}{\text{Number of cores, } P}. \tag{III.5}$$

In the ideal case, parallel efficiency will be equal to one, a case that indicates no overhead resulting from parallelization. However, this is most of the time not the case in parallel programs due to parallelization overhead.

Weak scalability studies show how well the parallel code scales if we increase the number of cores while keeping the amount of work per core constant. In the ideal

case of 100% efficiency, the execution time is expected to be constant regardless of the number of cores being used. Assume we run a program A_n, that solves a problem of size n, on one core and run a program $A_{n\times p}$, that solves a problem of size $n \times p$, on p cores. To measure the weak scaling efficiency of the parallel run over the serial run, we use the formula:

$$\text{Weak scaling efficiency for } P \text{ cores run} = \frac{\text{Execution time of } A_n \text{ on one core}}{\text{Execution time of } A_{n\times p} \text{ on } P \text{ cores}}. \tag{III.6}$$

Strong scalability studies, on the other hand, demonstrate how well the code scales if we increase the number of cores while keeping the problem size constant. In the ideal case, a speedup equal to the number of cores is expected. We calculate the parallel efficiency in this case for a run on P_2 cores compared to a P_1 cores run as follows:

$$\text{Strong scaling efficiency for } P_2 \text{ cores run} = \frac{\text{Execution time on } P_1 \text{ cores}}{\text{Execution time on } P_2 \text{ cores}} \times \frac{P_1}{P_2}. \tag{III.7}$$

III.2 Serial Performance Optimization

The effect of serial performance on the overall performance of parallel computational software cannot be ignored; by carefully examining and optimizing serial performance, a considerable speedup of the software can be gained. In this section, we discuss serial optimization techniques that we considered in the development of PetClaw. We start by discussing integrating compiled language modules to an interpreted language, Python in our case, in Section III.2.1. Then we discuss the role of the compiler in the serial optimization in Section III.2.2. After that we point to some cases where redundant data copying can be avoided in Section III.2.3. Finally, in Section III.2.4,

we demonstrate that PetClaw serial performance is comparable to Clawpack serial performance by presenting a performance comparison experiment.

Note that all the experiments in this section were conducted on a single core of the computational building block of Shaheen, the Quad-Core PowerPC 450 processor, an 850 MhZ CPU with 4GB RAM. The IBM XLF 11.1 Fortran compiler was used to produce the binary executables.

III.2.1 Integrating Compiled Languages

We have mentioned previously the advantages of using Python in scientific software tools in Section I.4. However, In some cases, flexibility in a programing language comes with the price of lower performance. Python is an interactive scripting language that is interpreted, the code is compiled into intermediate platform-independent language, called bytecode, which is interpreted by the Python interpreter into machine code during execution [21]. This has the advantages of enabling the user to interactively use the language, and is beneficial in the stage of code development, testing and debugging because the user operates in a much more flexible and expressive computational environment. However, interpretation causes a substantial performance degradation of the language as each bytecode statement will require translation when being executed. Python is also dynamically typed and performs dynamic binding which introduce extra logic and hence overhead on the CPU during the run time. [30].

Therefore, it is expected that Python will have poor performance when compared to a compiled language, such as Fortran, where the binary code is generated during the time of compilation ahead of execution time, variables are statically typed and type checking happens during the compilation. In fact, Python loops for example can be from 10 to 100 times slower than the corresponding compiled Fortran or C code [20]. Filling a one dimensional array using Python loop was shown to be 120 times

slower when compared to the same operation implemented in Fortran 77 [17].

We would like to get a reasonable combination of the advantages of both languages. A key observation states that *"Usually, highly optimal code is needed only in small parts of a scientific application, allowing the possibility of writing clean code and also utilizing slower languages than Fortran for the large portions of the application"* [20]. Thus a focus on a small part of the code, the intensive mathematical computations, might result in a considerable performance gain as this part dominates the execution time, as Amdahl's Law indicates. For Python applications, there are two approaches that can be followed for optimizing these computationally extensive parts of the code. The first one is that instead of using Python loops on arrays, efficient array structures and vectorized operations defined on those structures provided by libraries such as numpy can be used. The underlying implementation of numpy vectorized operations is done in C which is much faster compared to Python loops. To get even better performance, from 3 to 10 times faster than the vectorized numpy code, another approach can be used which is migrating the loop entirely in Fortran or C code [17].

As we discussed in Chapter II, We have used both approaches in designing PetClaw, using numpy package and incorporating fortran routines for computationally extensive parts. We have used Python language for the high-level parts of the code to define the code modules and data structures.

III.2.2 Optimizations Introduced by Compiler

When compiling the integrated Fortran kernels, it is essential to choose a recent compiler that is known to perform well on the target platform. Another point is that one can exploit the compiler optimization capabilities by turning the optimization flags on. For example, by setting the optimization flag `-O3` of the fortran compiler when compiling fortran kernels, we got about $3.6x$ speedup on Shaheen using XLF 11.1 compiler in a PetClaw acoustics application that evolved 445 time steps on a

500×500 grid; see Section A.3 for reproducibility information. The optimizations performed by the compiler depend on the compiler itself and the chosen optimization level. Needless to say, if compiled kernels were not used, one cannot exploit the compilers capabilities to enhance the performance further.

III.2.3 Avoiding Unnecessary Data Copying

Copying data can be unnecessarily expensive if large arrays are copied redundantly and repetitively in the code. In some cases, copying data is required by the algorithm but in other cases it can be avoided. Oftentimes, the developer needs to inspect the algorithm and the code looking for any unnecessary expensive copies. f2py, for example, generates a code for copying any array data that are passed to wrapped Fortran code if the array is not Fortran-contiguous. This copying has two side effects: first, a copy will take place and secondly, any changes to that array inside the Fortran routine will have no effect on the original array [24]. Copies of this type can be avoided if the developer ensures that arrays passed to Fortran extension modules are always Fortran-contiguous.

Also, when using packages one needs to understand what is going behind the library call because some functions might make copies of the data. For example, we set the array of the global vector at the end of each step with the new solution value as follows:

```
1 | state.q=qbc[:,mbc:mx+mbc,mbc:my+mbc]
```

The slicing of `qbc` will result in making a continuous copy to be assigned to `state.q`. `q` in `state.q` is a property as defined in Listing II.3. This property involves executing the method `PETSc.Vec.setArray` that in turns takes a multidimensional numpy array as an argument and makes a copy of this array to be the `Vec` array. Therefore, the previous statement involves two copies but only one copy is necessary. An alternative to the above statement would be using the `Vec` object method, `placeArray`, that sets

a specific array temporarily to be the vector's array as follows:

```
state.lqVec.placeArray(qbc)
state.q_da.localToGlobal(state.lqVec,state.gqVec)
state.lqVec.resetArray()
```

This alternative involves one copying operation only. This modification has reduced the execution time by around 6 percent in a 2D acoustics problem experiment with a grid of size 500×500 and evolving the solution 445 time steps; the optimization flag `-O3` was set. See Section A.3 for more information about this experiment reproducibility.

Another example of redundant copying we found was a copy operation of the solution `q` occuring as a part of the CFL condition test at each time step. The algorithm makes a backup copy at each time step so that the code can rollback in case the step is rejected. Also in the 2D case, a copy of the data is required by the algorithm implemented in the Fortran kernels that will be accessed but not modified. To avoid the redundant second copy, we passed the CFL backup copy to the Fortran kernel. This reduced the execution time further by about 8 percent.

III.2.4 PetClaw Versus Clawpack Serial Performance

We conclude the discussion of the serial performance optimization by presenting this experiment in which we compare the total execution time of two different 2D problems implemented in PetClaw versus their corresponding implementations in Clawpack after introducing the serial optimization to the PetClaw code. The first problem is a system of 2D linear acoustics equations that involves a very simple Riemann solver. This problem is expected to reveal the worst-case performance difference due to the Python code overhead because the time spent in the Fortran kernels is minimal compared to other potential applications. The other problem, which we believe is more realistic, is a 2D nonlinear shallow water equations problem that involves a relatively costly Riemann solver, a Roe solver with entropy fix.

Table III.1 shows the on-core serial time comparison between the pure Fortran Clawpack code and the corresponding hybrid PetClaw code for the two systems of equations. Both Clawpack and PetClaw codes rely on similar Fortran kernels that differ only in the array layout. PetClaw Fortran kernels are interleaved whereas Clawpack kernels are not. The compiler optimization flag `-O3` was set.

Table III.1: Timing results in seconds for on-core serial experiment of acoustics and shallow water problems implemented in both Clawpack and PetClaw on the PowerPC 450 architecture.

Application	Clawpack	PetClaw	Ratio
Acoustics	192s	316s	1.6
Shallow Water	714s	800s	1.1

The difference in timing between the Clawpack and PetClaw codes is mainly due to the overhead introduced by Python, including function calls, dynamic binding, type checking, interpretation and object initialization. For the acoustics example, the PetClaw code has a longer runtime of about 64 percent of the execution time of the Clawpack code. We think this result is acceptable and the advantages gained by using Python worth this increase. However, for a more realistic application such as the solution of the shallow water equations, where a larger portion of the execution time is spent in the Fortran kernels, we get an even better comparative performance of the PetClaw code with only about a 12 percent increase in the execution time.

III.3 Parallel Scalability Study

In this section, we study PetClaw parallel performance by conducting scalability experiments for two different PetClaw applications, 2D linear acoustics system and 2D Euler equations of compressible fluid dynamics. We run these experiments on up to the entirety of Shaheen, a total of $65,536$ cores. We focus on the following aspects of the parallel performance:

- Studying the scalability of the solver. We focus on the effect of the ghost-cell communication and reduction operations taking place each time step. Weak and strong scalability studies were conducted to assess the solver performance on Shaheen for the selected 2D applications. The solver scalability is discussed in Section III.3.3.

- Studying the scalability of the output. 2D weak and strong scaling studies for writing outputs were performed to evaluate the output performance. Results of this study are presented in Section III.3.4.

- Studying the scalability of the dynamic loading of Python libraries to compute nodes, which is presented in Section III.3.5.

In Section III.3.1, we elaborate on several details of the Shaheen architecture to provide a background for understanding the scalability study results. After that, we explain set up of the experiments conducted and the execution time measurement approach we followed in Section III.3.2. Then we present and discuss the scalability results of the solver, output and Python loading in Sections III.3.3, III.3.4 and III.3.5, respectively.

III.3.1 Shaheen Architecture

In this section, we provide several relevant details about the Shaheen architecture to establish a background for understanding the parallel performance on the machine. Shaheen is a 16 rack IBM Blue Gene/P distributed-memory supercomputer. It has 16,384 compute nodes, each equipped with a quad-core PowerPC 450d processor running at 850MHz. Each node is attached to 4 GB RAM. These compute nodes are organized in 16 racks of 1,024 nodes each. The I/O nodes (not utilized for compute) also have quad-core PowerPC 450d processors running at the same CPU clock speed

with 4 GB RAM as well. 12 racks of Shaheen have 8 I/O nodes per rack, while the remaining four racks have double the ratio, 16 I/O nodes per rack.

The Shaheen compute nodes are connected to each other through a 3D point-to-point torus network, which is used for general-purpose point-to-point message passing as well as multicast operations to a selected class of nodes. Each compute node has six connections to the torus network, connecting the node to its nearest neighbors through a link of 3.4 Gbps bandwidth in each direction. Some links can be relatively long cables. The Blue Gene/P midplane size, that is the size of the smallest unit that supports the full 3D torus, is 512 compute nodes. Jobs on smaller units cannot take advantage of the machine's toroidal network topology, and are instead connected on an equivalently dimensioned 3D mesh.

Each compute node also has three connections to the global collective network at 6.8 Gbps per link. The global collective network is a high-bandwidth one-to-all network used for collective communication operations such as broadcast and reduction operations. All I/O requests on the compute nodes are directed to the I/O node through this network. I/O nodes in turns are connected through a 10 gigabit Ethernet network to one or more of the following: Service nodes which provide control of the system, Front End nodes which provide access to users and the global file system [31].

III.3.2 Experimental Setup and Timing Approach

We have considered two 2D applications for studying the scalability of PetClaw. These are 2D linear acoustics system that has a relatively simple Riemann solver and 2D Euler equations of compressible fluid dynamics as a more realistic nonlinear application problem that has a relatively expensive Riemann solver, a Roe solver with entropy fix. We have performed a weak scalability study for the two applications on up to the entirety of Shaheen and a strong scalability study for the acoustics application on up to 4 racks.

To get timing measurements, we use the Python function `time()` from the `time` package, and the Python profiler, `cProfile`. We use the `cProfile` to profile process 0, from which we extract the timing data for our scaling study assuming a frequent syncronization and equitable load balancing in PetClaw applications. We also profile process 5 to verify the results we get from process 0. To get the total job time, we use the job execution time measured by the Shaheen job scheduling software, `LoadLeveler`. To separate the time required for loading Python dynamic libraries to compute nodes from the actual execution time required by the solver to evolve the solution and write output, we have written a Python script to run each application three times in a single job, we refer to those three runs as Part 1, 2 and 3 respectively. Parts 1 and 2 are identical and evolve the solution very few time steps. Part 1 execution time will include any startup or Python loading overhead. Part 2 execution time will indicate how much time Part 1 should take excluding the startup and loading overhead. We are mostly interested in Part 3, which is the solver execution time including the time of writing the solution to files. We have profiled Part 3 using `cProfile` in only two processes to study how the time is being split among different tasks: computations, communication and i/o.

Listing III.1 shows how these three parts are being timed for an Euler application. Note that the function `shockbubble`, which executes the Euler application, is being called in lines 4 and 9 to run Part 1 and 2, and called in either line 19 with profiling or line 25 without profiling to run Part 3. We have calculated the Python loading time as follows:

$$\text{Python load time} = \text{Total job time} - \text{Part\_3 execution time} - 2 * \text{Part\_2 execution time}, \tag{III.8}$$

where the total job time is the one provided by the `loadleveler`. The reason we subtract double the execution time of Part 2 instead of subtracting the execution

time of Part 1 and Part 2 is that we assume that the execution time of Part 1 might include loading overhead due to import statements in subroutines.

Listing III.1: Experiments timing approach.

```
#.
#.
time_1=time()
shockbubble(finalt=(0.02/np.sqrt(size))/40,use_petsc=True,\
          outdir='exper6_d/_output_p1')
PETSc.COMM_WORLD.barrier()

time_2=time()
shockbubble(finalt=(0.02/np.sqrt(size))/40,use_petsc=True,\
          outdir='exper6_d/_output_p2')
PETSc.COMM_WORLD.barrier()

time_3=time()
if rank in proccessesList:
# We have chose proccessesList to contain 0 and 5.
# These processes are profiled.
    funccall = "shockbubble(finalt=0.02/np.sqrt(size),\
          use_petsc=True, outdir='exper6_d/_output_p3')"
    cProfile.run(funccall,\
          'exper6_d/profile'+str(rank)+'_'+str(size))

else:
# These processes are not profiled.
    print "process"+str(rank) +"not profiled"
    shockbubble(finalt=0.02/np.sqrt(size),use_petsc=True,\
          outdir='exper6_d/_output_p3')
PETSc.COMM_WORLD.barrier()

tend = time()
if rank == 0:
    print "Time to subtract from job time to get \
          Python load time= part2 time*2 + part3 time",\
          2*(time_3-time_2)+ (tend- time_3)
```

In the weak scalability study, we evolve the solution of the acoustics problem 178 time steps on a square grid of size 160,000 (400×400) grid cells per core, where each grid cell has three solution components. We evolve the solution of the Euler problem, on the other hand, 67 time steps on a rectangular grid of 160,000 (400×400) grid cells per core with each grid cell having five components. We started by running the

problem on one core, then increased the number of cores by a factor of 4 in each run until reaching 65,536 cores. In Section A.5, we provide reproducibility information for the weak scalability experiment.

In the strong scalability study of the acoustics problem, the grid size is fixed for all the runs, 4096^2 grid cells each has three components. We evolve the solution 114 time steps. We followed the same approach in choosing the number of cores for the runs as in the weak scalability study, but started from 16 cores until 16,384 cores only. Reproducibility information for the strong scalability experiment is found in Section A.5.

III.3.3 Solver Scalability

The Solver evolves the solution several time steps to reach the final time. To identify the factors affecting the solver parallel performance, recall that executing a time step requires the solver to:

- Update the ghost cells of the local vector of the solution array, `q`. In the 2D case, the number of communicated ghost cells depends on the stencil width, c, and the number of grid cells in each dimension of the process's partition of the grid, m_x and m_y for the x and y dimensions receptively. Each process will send $2 \times c \times m_x + 2 \times c \times m_y$ grid cells in the case of star stencil and additional $4 \times c^2$ grid cells in case of box stencil, which is the type used in PetClaw, and will receive the same number of ghost cells. This communication might be required more than once per time step. The `petsc4py.PETSc.DA` method `globalToLocal` performs this communication.

- Update the boundary conditions: In the case of periodic boundary conditions, communication is required. However, the method `globalToLocal` performs this communication while updating the ghost cells. Boundary conditions update can

happen more than once per time step.

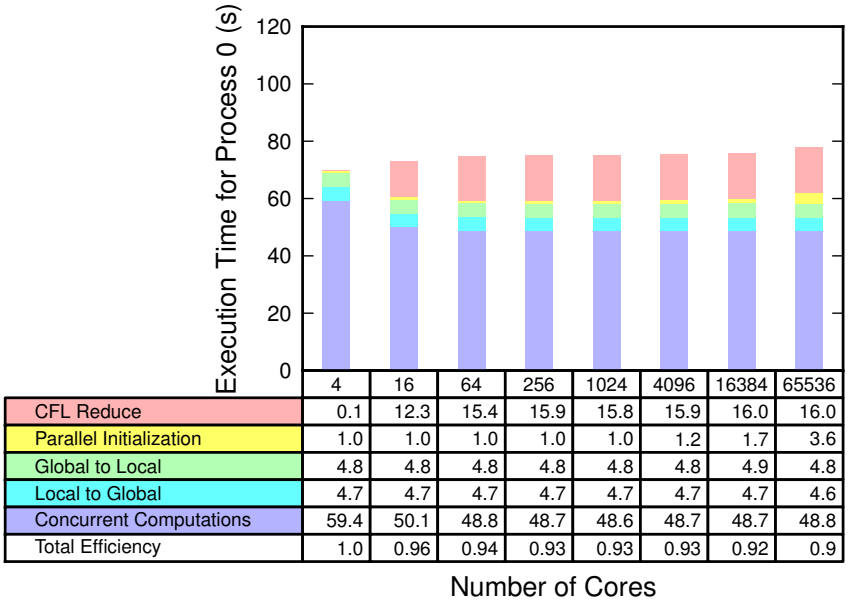
- Perform the concurrent computations required for evolving the solution in time. This includes solving the Riemann problem and might include applying limiters which happen locally in each process. Some applications might also include solving for a source term which is also a local operation that does not need communication.

- Perform reduction operation over a vector of size equal to the number of processes, one element per process, that happens once per time step to find the maximum CFL number. The `petsc4py.PETSc.Vec max` method performs this reduction.

- Update the global vector with the new solution after performing the time step. The `petsc4py.PETSc.DA` method `localToGlobal` performs this task. Note that this task is done locally and does not require communication.

We also consider the parallel initialization of the `DA` and `Vec` objects. Note that the concurrent computation time can be obtained by subtracting the execution time of the methods `localToGlobal`, `globalToLocal` and `petsc4py.PETSc.Vec max` from the execution time of the method `evolve_to_time`. We follow an empirical approach to study how this mixture of computations and communication affects the scalability of PetClaw solver in the 2D case.

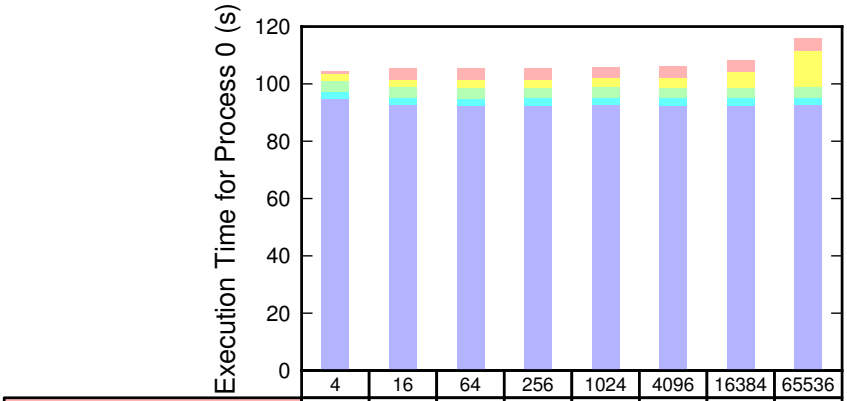
Figures III.1 and III.2 show breakdown of the solver execution time for the acoustics and the Euler applications respectively in the weak scalability study. Starting from the sixteen-core run, the concurrent computations execution time appears to scale perfectly. This is also the case for the `globalToLocal` and `localToGlobal` calls. `globalToLocal` call involves communicating the ghost cells and setting the periodic boundary conditions. For this communication, the three-dimensional point-to-point torus network is used for exchanging ghost cells. Each process will communicate with



	4	16	64	256	1024	4096	16384	65536
CFL Reduce	0.1	12.3	15.4	15.9	15.8	15.9	16.0	16.0
Parallel Initialization	1.0	1.0	1.0	1.0	1.0	1.2	1.7	3.6
Global to Local	4.8	4.8	4.8	4.8	4.8	4.8	4.9	4.8
Local to Global	4.7	4.7	4.7	4.7	4.7	4.7	4.7	4.6
Concurrent Computations	59.4	50.1	48.8	48.7	48.6	48.7	48.7	48.8
Total Efficiency	1.0	0.96	0.94	0.93	0.93	0.93	0.92	0.9

Figure III.1: Solver execution time breakdown in a weak scalability study of a 2D acoustics application (in seconds).

its neighbors at the same time through these local connections resulting in perfectly scaled communication. On the other hand, the CFL reduction operation appears to increase rapidly up to 16 cores, then shows almost constant timings from 64 to 65,536 cores. Actually, from the results of profiling process 5, we inferred that the CFL reduction operation cumulative time is a fraction of a second. However, the large execution times of the CFL reduce operation that we see sometimes is mainly due to waiting for other processes to finish executing the concurrent computations. Note that the summation of the CFL reduction operation and the concurrent computations execution times is almost constant. Other than revealing this fact, the profiling results we obtain from profiling process 5 is almost identical to the results obtained from process zero. The time for initializing PETSc objects, `DA` and `Vec`, increases in larger runs. In the Euler application, the computation time appears to be more dominant than communication when compared with the acoustics application.



	4	16	64	256	1024	4096	16384	65536
CFL Reduce	0.7	4.1	4.1	3.9	3.8	3.9	4.1	4.2
Parallel Initialization	2.7	2.7	2.7	2.9	3.0	3.5	5.5	12.9
Global to Local	3.6	3.6	3.6	3.6	3.6	3.6	3.7	3.6
Local to Global	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5
Concurrent Computations	94.8	92.7	92.5	92.6	92.8	92.6	92.5	92.8
Total Efficiency	1.0	0.99	0.99	0.99	0.99	0.98	0.96	0.9

Number of Cores

Figure III.2: Solver execution time breakdown in a weak scalability study of a 2D Euler application (in seconds).

The overall solver weak efficiency for the two problems are shown in the tables included in Figures III.1 and III.2. We have omitted the efficiency results for the one-core run and measured the efficiency relative to the four-core run. This is because `globalToLocal` and `localToGlobal` PETSc functions are taking, unexpectedly, almost double the time when running on one core compared to running with more cores. Although, conceptually, there is no inter-process communication required for the one-core run. Note that the parallel initialization overhead only takes place at the beginning of the application run. Because of the CFL condition, most of PetClaw real-world applications that are run on supercomputers require much longer runs to give meaningful results, hence, the initialization overhead almost vanishes. Therefore, we can consider the parallel efficiency excluding the parallel initialization overhead as shown in Figure III.3. In these modified efficiency results, we can see that the Euler application efficiency is much better than the acoustics application efficiency

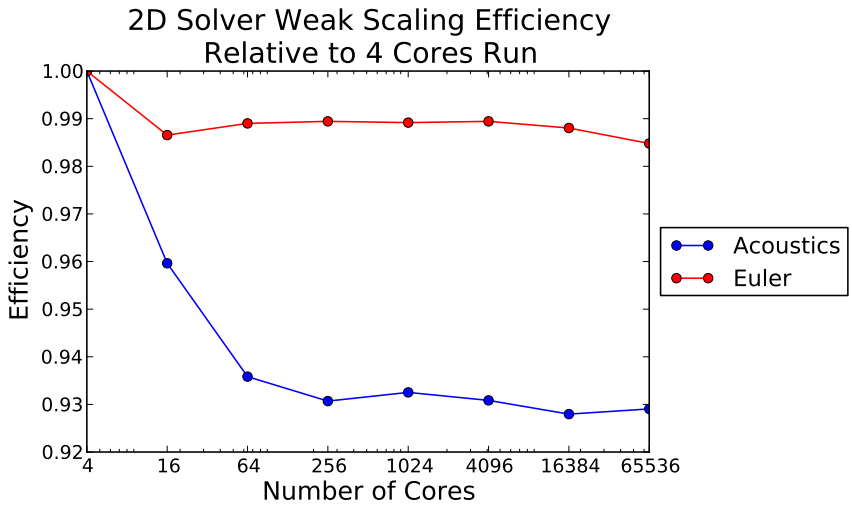


Figure III.3: 2D solver weak scalability efficiency for an acoustics and an Euler problems excluding the initialization overhead.

and does not go below 0.98.

The breakdown of the solver timing results in the strong scaling experiment is shown in Table III.2. Figure III.4 shows a scaled version of those timing results where in the ideal case the scaled time should remain one. Larger scaled time indicates poor strong scalability. This scaled timing is obtained by the following formula:

$$\text{Scaled timing for } P_2 \text{ cores run} = \frac{\text{Total execution time on } P_2 \text{ cores}}{\text{Total execution time on } P_1 \text{ cores}} \times \frac{P_2}{P_1}, \quad \text{(III.9)}$$

which is the inverse of the strong efficiency we have shown in Formula III.7. To find the scaled time for a fraction of the code f, we use the following formula:

$$\text{Scaled timing for fraction } f \text{ on } P_2 \text{ cores run} = \frac{\text{Execution time of } f \text{ on } P_2 \text{ cores}}{\text{Total execution time on } P_1 \text{ cores}} \times \frac{P_2}{P_1}. \quad \text{(III.10)}$$

The solver overall strong efficiency for the acoustics problem is shown in Figure III.5, which shows that this application exhibits excellent strong scaling efficiency up to running on 1,024 cores, where the size of the grid partition per core is 128^2.

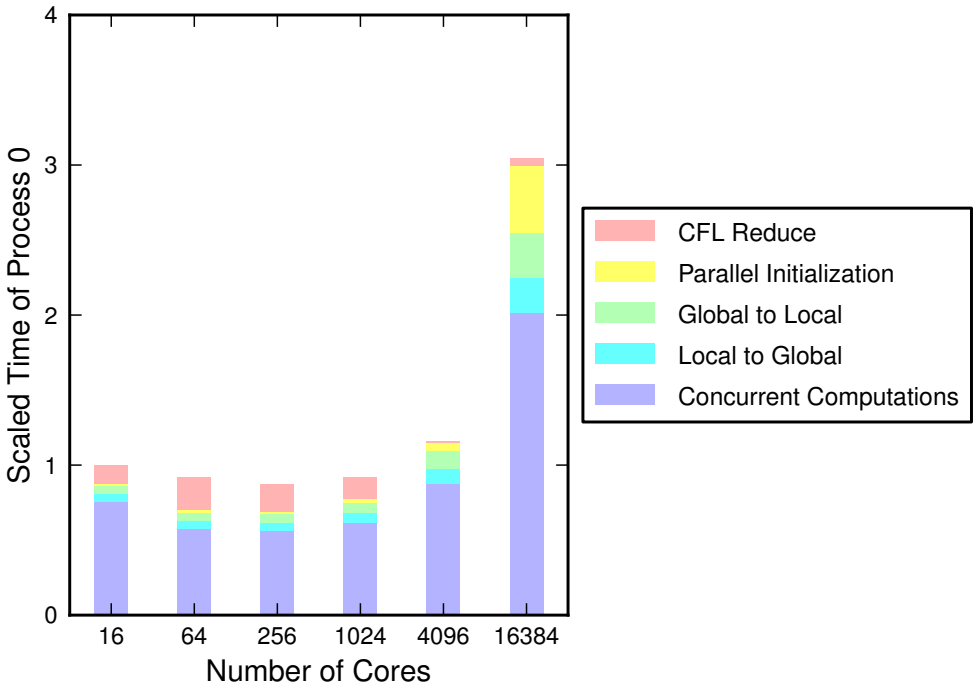


Figure III.4: Solver scaled execution time breakdown in a strong scalability study of a 2D acoustics application.

Table III.2: Solver execution time breakdown (in seconds) in a strong scalability study of a 2D acoustics application.

Number of Cores	16	64	256	1024	4096	16384
CFL Reduce	44.198	20.175	4.133	0.803	0.018	0.018
Parallel Initialization	6.284	1.584	0.406	0.152	0.079	0.152
Global to Local	19.353	4.979	1.345	0.412	0.171	0.107
Local to Global	19.108	4.837	1.266	0.369	0.140	0.082
Concurrent Computations	276.369	52.302	12.749	3.553	1.443	0.929

Actually, we see a superlinear efficiency in the 64 core, 256 core and 1,024 core runs due to the fact that the portion of the data that can fit in the cache is larger as the number of cores increases and the size of the problem per core decreases. However, for larger runs, we notice that other factors compete with the cash effect and cause efficiency degradation. The efficiency degrades to about 85% for the one-rack run, on 4,096 cores, which has a grid partition per core of size 64^2. For the four racks run, where the grid partition size is 32^2, the efficiency drops to about 32%. We performed a secondary experiment which shows that this degradation is mainly due to the reduction of the grid partition size per core as we increase the number of cores and not because of the increase in the number of cores itself. For very small partitions of the grid per core, concurrent parts of the code and overhead such as object initialization, function calls and replicated computations will form a relatively larger fraction of the code limiting the gained speedup because they do not scale. The strong efficiency breakdown for the different solver components is shown in Figure III.6. We do not include the CFL reduce operation efficiency in this figure for clarity purposes; see Figure III.7(d) for this information. Analogous to the case of weak scalability experiment, the profiling results we obtain from profiling process 5 is almost identical to the results obtained from process zero in the strong scalability experiment.

In the secondary experiment, we profiled several runs of the same application on one core while decreasing the size of the grid by a factor of four in each run. We do this in order to emulate the decrease in the size of the grid partition per core as the number of cores increases in the strong scaling study. We also profiled similar runs on 4 cores and 16 cores. In Figure III.7 we present comparison of the strong scaling efficiency to the efficiency of those constant-core runs for the different components of the solver. The Figures III.7(a) and III.7(b) show that for the methods `globalToLocal` and `localToGlobal` the behavior of losing efficiency in the four-core and sixteen-core

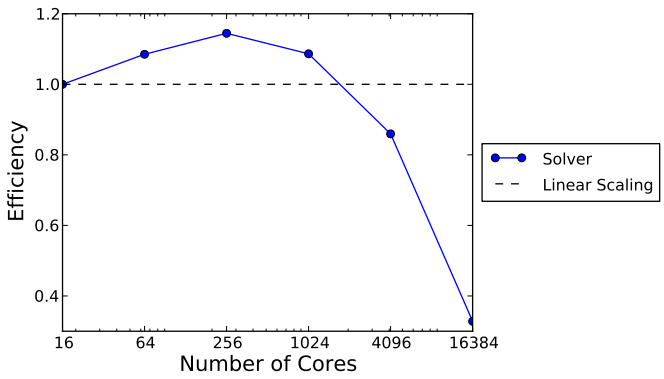


Figure III.5: Solver strong scalability efficiency for a 2D acoustics problem.

runs is similar to that of the strong scaling experiment. This indicates that those two methods mainly depend on the change of the size of the grid per core and not the number of cores. A similar conclusion can be drawn from Figure III.7(c), which shows the efficiency comparison for the concurrent computations, where the loss of efficiency in constant-core runs happens in a similar manner to the strong scaling runs.

Considering Figure III.7(d), we can see that the dependency of the performance of the reduction operation on the grid size per core is more apparent in the sixteen-core runs than in one-core and four-core runs. This indicates a dependency on the number of cores, but stronger dependency on the grid size per core as the behavior of the sixteen-core runs is comparable to that of the strong scaling runs. On the other hand, Figure III.7(e), shows a clearer dependency of the parallel initialization on the number of cores and a dependency on the grid size per core as well; however, the cost of the initialization is inexpensive and is paid only once at the beginning of the simulation. Therefore, this cost can be neglected in long simulations.

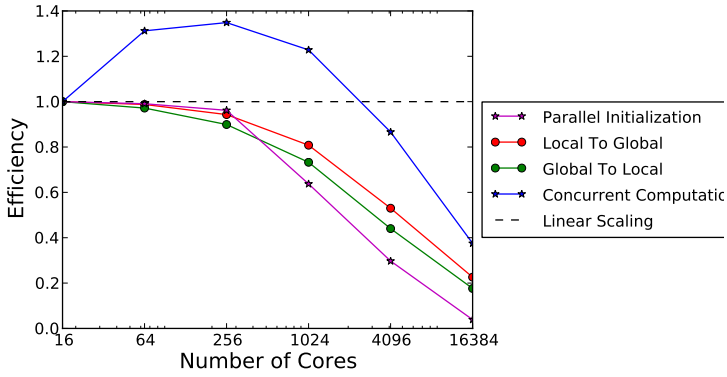
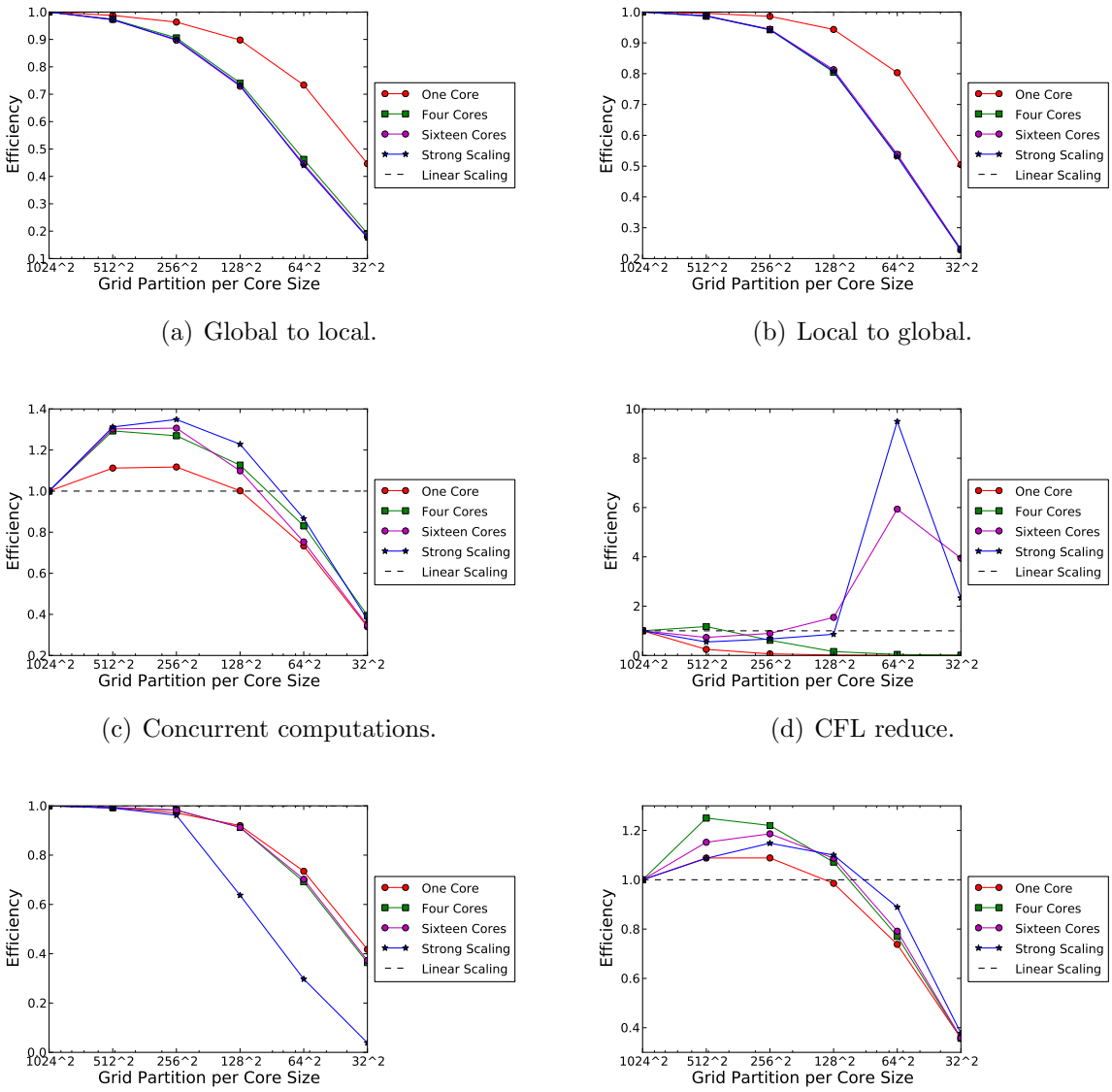


Figure III.6: Solver strong scalability efficiency breakdown for a 2D acoustics problem.

Finally, Figure III.7(f) shows the comparison results for the solver as a whole, method `evolve_to_time`, including communication and global reduction but without the cost of initialization. It shows that, in general, the loss of efficiency is mainly due to the decrease of the grid size as we stated earlier. Therefore, to get better performance, the grid partition per core and the amount of work on the grid should be large enough to sufficiently outweigh the sequential (serial) overhead. We anticipate that choosing a grid partition per core of size 128^2 will guarantee almost linear speedup for problems as simple as linear acoustics. However, smaller grid partition per core might be justified in more expensive applications.

III.3.4 Output Scalability

In this section, we present and discuss the scalability results of writing the solution array, q, to files for the experiments demonstrated in Section III.3.2. Figure III.8 shows the weak scaling efficiency and execution time for two different solution array sizes, $400 \times 400 \times 3$/core and $400 \times 400 \times 5$/core. It is clear that writing output does not scale and is executed in a serial manner. For large grid distributed on large number of cores, writing the solution to files starts occupying a significant fraction of the execution time, causing the application to significantly lose scalability. To



(a) Global to local.

(b) Local to global.

(c) Concurrent computations.

(d) CFL reduce.

(e) Parallel initialization.

(f) Solver.

Figure III.7: Analysis of parallel strong efficiency for the different components of the solver by comparison with the corresponding constant-core runs.

illustrate this point, let us consider the Euler application run on four racks. Evolving the solution 67 time steps will take little more than 100 seconds. Writing two outputs for this run (initial and final q), which has a total grid size of $102400 \times 25600 \times 5$, will take more than 2000 seconds. However, it is generally the case that the frequency of producing output will be much less than once in each 67 steps. Therefore, the effect of the poor output scalability will be less as well.

Strong scaling efficiency and execution time results for a grid of size 4096^2 with three components are shown in Figure III.9. Writing the solution in this study is also not expected to strong scale as this operation is eventually done in a serial manner. Note that the writing time for a grid of a fixed size does not vary very much for different runs on different number of cores.

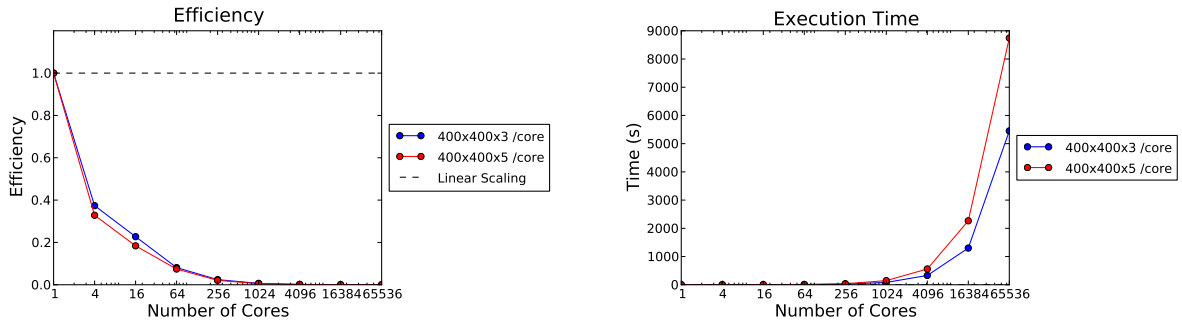


Figure III.8: Parallel weak efficiency and execution time for writing solution arrays, q, of sizes $400 \times 400 \times 3/core$ and $400 \times 400 \times 5/core$ to files.

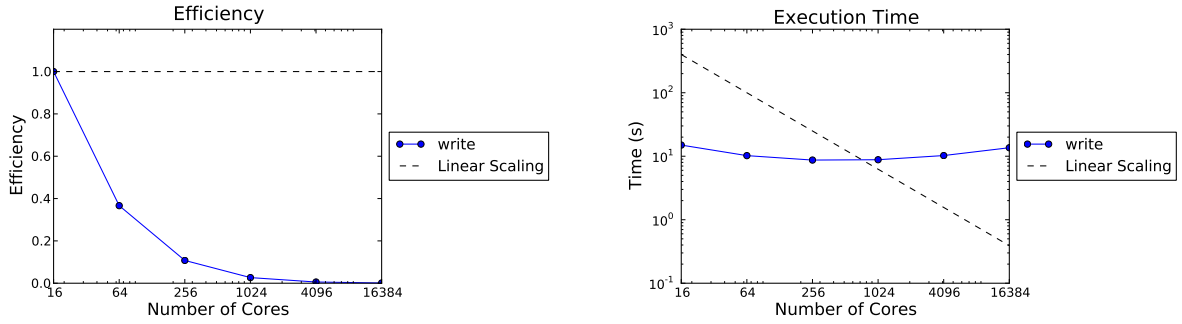


Figure III.9: Parallel strong efficiency and execution time for writing a solution array, q, of size $4096^2 \times 3$ to a file.

III.3.5 Python Dynamic Loading

There is a significant overhead of loading the required Python dynamic libraries to Shaheen compute nodes when running a Python-based application. Figure III.10 shows the loading time of the Python libraries required for running a PetClaw application. Those required libraries have a total size of about $128MB$ and should be loaded to each compute node. It is clear that this loading operation has very poor scalability. For example, it will take about an hour to load the required libraries to four racks. Although much longer simulations can to some extent justify the start up time required for dynamic loading of Python, this loading time negatively impacts the parallel scalability.

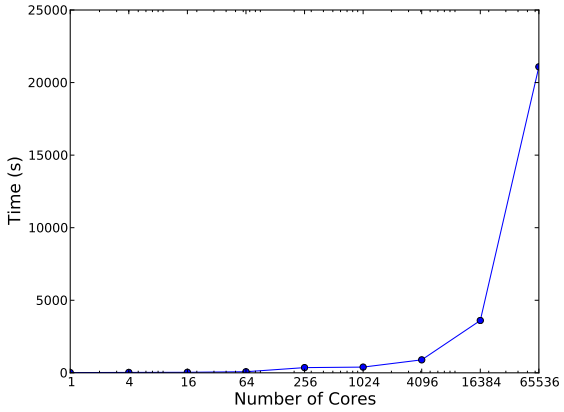


Figure III.10: Python dynamic loading time on up to 16 thousand of cores.

Chapter IV

Conclusions and Current and Future Directions

PetClaw and PyClaw are being actively developed, maintained and improved by a number of contributors [4], and the version 1.0 release of the tools has been recently issued. Two papers were published to present those packages scope, design, performance and applications; one in SpringSim'11 conference proceedings and the other in SciPy 2011 conference, [3] and [1] respectively. Also, a third journal paper has been submitted to SIAM Journal on Scientific Computing (SISC) [2].

In this chapter, we summarize the key conclusions of this thesis research in Section IV.1. Then we give an idea about current work being conducted for improving PetClaw and PyClaw and future plans for further extending those tools in Section IV.2.

IV.1 Concluding Remarks

In this thesis research we have demonstrated how using Python and petsc4py to develop an implementation of PetClaw, as we discussed in Chapter II, has several advantages including the following:

- Python support for packages, modules and class hierarchies allow for extending PyClaw and introducing parallelization through class inheritance with maximum code reuse in a modular manner. The parallelization is totally hidden from PyClaw.

- As opposed to the approach of using MPI directly to achieve parallelization, using PETSc relieves us from worrying about low-level parallelization details in PetClaw. This is because PETSc handles most of those details such as deciding the work distribution among processes, preparing outgoing messages, communication and updating local data using incoming messages.

- PETSc parallel solvers can be easily adopted in the future to solve systems of equations arises in the cases of implicit time stepping and solving for stiff source terms, with minimal effort.

- Python allows extending PyClaw with Clawpack Fortran kernels seamlessly, keeping the parallelization logic hidden from those kernels.

- By exploiting Python features, setting up a PetClaw application is identical to setting up a PyClaw application. All parallelization details are hidden from the user.

- PetClaw and PyClaw are easy-to-use and the user is able to perform analysis, visualization and other tasks that access solution data in memory using a single development environment.

- PetClaw and PyClaw code modularity enhances their readability, maintainability and extensibility.

The serial optimizations we have introduced to PyClaw, including integrating Clawpack Fortran kernels into PyClaw for low-level computationally intensive parts of the code and removing redundant copies, result in a comparable serial performance

for PetClaw code when compared to the pure Fortran Clawpack code, as we discussed in Section III.2. As a result of those optimizations, the Python overhead in PetClaw for a shallow water application is only 12 percent when compared to the corresponding Clawpack application.

By using petsc4py, we were able to scale PetClaw on up to the entirety of Shaheen, 65,536 cores, as we discussed in Section III.3. We have seen that the ratio of computation over communication is very high in PetClaw solvers which enables the code to scale. The PetClaw solver achieved above 0.98 weak scaling efficiency for an Euler application on the entirety of Shaheen excluding the parallel initialization overhead that is less significant in potential long runs of PetClaw applications. The solver also achieved a superlinear strong scaling efficiency for an acoustics application on 1,024 cores. However, more work is required to achieve better results in output and Python loading scalability.

Furthermore, we provided reproducibility information for all the computational experiments presented in this thesis.

IV.2 Current and Future Directions

PetClaw and PyClaw are actively under development introducing new algorithms and features. They have gone through numerous improvements by several developers since the initial implementation of PetClaw of this thesis took place. One can refer to [25] to see the current software status. Recently, a support for 3D applications and mapped grids and surfaces has been added. PyClaw and PetClaw now include full functional Clawpack and SharpClaw. The latter makes use of high order WENO reconstruction and Runge-Kutta methods to achieve high order accuracy.

In addition, adapting PETSc parallel solvers to solve for stiff source terms or implicit methods is being conducted. Also, GPU Riemann problem solvers are under

development with the goal to integrate them with PyClaw to further accelerate the code. In addition, automatic generator for generating Riemann problem solvers from symbolic descriptions is an undergoing project [26].

PetClaw has been used in a number of applications since it was developed. One recent work, by M. Q. de Luna [32], used PetClaw to confirm the phenomenon of the formation of solitary waves from an initial smooth wave on 2D heterogenous media. Solving this problem numerically is computationally costly because it requires a large number of grid cells in order to have a grid that is much finer than the structure of the heterogeneous medium, which is necessary to the numerical study of this phenomenon. The occurrence of this phenomenon was proved by M. Q. de Luna using a grid that has 55×55 domain units, where each unit has 120×120 grid cells.

There are several future direction for improving and extending PetClaw and PyClaw. Those include enhancing the scalability of output and loading Python. In addition, PetClaw can be extended to include support for adaptive mesh refinement based on the AMRClaw package.

APPENDICES

Appendix A

Experiments Reproducibility Information

In this appendix we provide reproducibility information for the experiments presented in Chapter III. We list the software and hardware dependencies in Section A.1. In Section A.2, we give instructions for building and running Clawpack and PyClaw applications along with the required environment settings. In Sections A.3, A.4 and A.5, we give detailed information about the reproducibility of the serial optimization experiments, comparison between PetClaw and Clawpack serial performance, weak and strong scalability experiments respectively. In Section A.6, we provide information about figures reproducibility.

The bitbucket repository (`https://bitbucket.org/amal/alghamdi_thesis_experiment`) contains information, scripts and settings required for reproducing these experiments.

A.1 Hardware and Software Dependencies

We have performed all the experiments on the Shaheen supercomputer, a 16-rack IBM Blue Gene/P with 4GB of RAM per each one of its 16,384 quadcore PowerPC

450 nodes.

The following is the list of software dependency stack for running the applications. The specific revisions of PyClaw and Clawpack will be provided with each experiment.

- Python 2.6
- numpy 1.6.1
- PETSc 3.1.8
- petsc4py 1.1.2 (dev-june2011)
- Riemann repository revision 184798b81b. (`https://github.com/clawpack/riemann`).
- IBM XLF 11.1 compiler
- IBM BlueGene/P System Driver V1R4M2_200_2010-100508P

A.2 Building and Running PetClaw and Clawpack Applications

In this section, we provide building and running instructions for Clawpack and PyClaw applications. We also provide the required environment settings on Shaheen. The PyClaw and riemann packages are available as github repositories in `https://github.com/clawpack`. Note that PetClaw package is part of the PyClaw package. For installing PyClaw on Shaheen, issue the following commands:

```
1 $mkdir clawpack
2 $cd clawpack
3 $ git clone git@github.com:clawpack/pyclaw.git
4 $ git clone git@github.com:clawpack/riemann.git
```

Refer to `http://depts.washington.edu/clawpack` for installing Clawpack 4.3 and 4.6.1.

The environment settings for running a PetClaw application on Shaheen is as follows:

```
$ export CLAW=<path_to>/clawpack
$ export RIEMANN=$CLAW/riemann
$ export PYCLAW=$CLAW/pyclaw
$ source <path_to>/prod.sh
```

A copy of `prod.sh` will be found in the main directory of the reproducibility repository. The file `prod.sh`, written by Aron Ahmadia, defines two bash functions, namely `pyclaw_build` and `pyclaw_run`. As their names implies, the first one sets environment variables for building the Fortran extension modules for the PyClaw application and the second one sets environment variables for running the PetClaw Python script.

To compile the extension modules, the following commands can be issued:

```
$ pyclaw_build
$ cd <experiment_directory>
$ make_f2py
```

We have used -O3 flag in all our PetClaw and Clawpack experiments (unless otherwise specified). If turning off optimization flags is required, -O3 should be removed from the application (and Clawpack libraries') makefile and from the environment variable `FC` in the files `prod.sh` and `prod_clawpack.sh`.

To run the PetClaw applications, (refer to `https://github.com/clawpack/pyclaw/wiki/Running-on-shaheen/2328cbd968ede04f28e07bcecf5827b13fbf99a3` for compilation and running details) issue the following commands in another terminal:

```
$ pyclaw_run
$ cd <experiment_directory>
$ kslrun -a <account_number> -n <number_of_processes> -t \
<wall_time> -p <partition_size> -m VN -r  \
"/bgsys/drivers/ppcfloor/gnu-linux/bin/python <script_name>"
```

To run a Clawpack application, the following environment settings can be used:

```
export CLAW='<path_to>/<clawpack_directory>'
export PYTHONPATH="${CLAW}/python:${PYTHONPATH}" # If Clawpack
                                                 # 4.6 is used.
source <path_to>/prod_clawpack.sh
```

The file `prod_clawpack.sh` is available in the main directory of the reproducibility repository. The following are the commands required to compile and run a Clawpack application.

```
# To build:
cd <application_directory>
python setrun.py # If Clawpack 4.6 is used.
clawpack_build
make_clawpack
# To run: (in another terminal)
cd <application_directory>
clawpack_run
kslrun -a <account_number> -n 1 -t <wall_time>  -p 1 -m SMP\
 -r  "./xclaw"
```

A.3 Serial Optimization Experiment

In this section, we present the reproducibility information of the serial optimizations discussed in Sections III.2.2 and III.2.3. This experiment finds the effect of a number of optimizations on the serial performance of acoustics problem implemented in PyClaw. The experiment files are found in the directory `serial_optimization` of the repository.

The reported results were manually gathered from the application run profiles (provided in the directory `serial_optimization/profile` of the repository). The execution time for `acoustics2D` method is considered to be the application runtime.

We have run the application three times introducing an optimization each time and measuring the performance. The reproducibility information of those runs are shown in table A.1. In the third row of the table, we have avoided a copy of `q` by using the methods `placeArray`, `resetArray` instead of slicing and using the method `setArray`. In the forth row, We have avoided another copy of the solution `q`. This copy was required by the dimensionally split algorithm. However, the solution backup copy is passed to the dimensionally split algorithm instead. All those tests have been

run on Shaheen in the SMP mode on one-core of a quadcore PowerPC 450.

Table A.1: Reproducibility information for the serial optimization results. The run in each row includes the optimization from the previous rows.

Optimization introduced	Run time	Profile file name	PyClaw revision
No optimization	1356 s	profile_0b418_shaheen_no_opt	0b41891697
Set flag -O3	374 s	profile_0b418_shaheen	0b41891697
Remove one copy of `q`	348 s	profile_beebd_shaheen	beebdc62d0
Remove another copy of `q`	322 s	profile_b4e7e_shaheen	b4e7e6efcf

A.4 PetClaw Versus Clawpack Serial Performance Experiment

In this section, we provide the reproducibility information for the comparison between PetClaw and Clawpack performance, discussed in Section III.2.4. We showed timing results (in seconds) for on-core serial run for applications solving acoustics and shallow water problems implemented in both Clawpack and PyClaw on IBM BlueGene/P. The reported results were manually gathered from the applications' output data (provided in the directory `serial_performance_comparison/profile`).

This experiment's main directory is `serial_performance_comparison`. The four sub experiments (Clawpack acoustics, PyClaw acoustics, Clawpack shallow water and PyClaw shallow water) are found in the `experiment` subdirectory and the corresponding output data are found in the `profile` subdirectory. The PyClaw revision for this experiment is ce6fe01641. Clawpack version for the acoustics application is 4.6.1 and for the shallow water application is 4.3. We ran all those experiments on Shaheen in the SMP mode.

A.5 Acoustics and Euler Applications Weak and Strong Scaling Experiments

In this section, we provide reproducibility information for the weak and strong scalability studies that we presented in Section III.3. In the weak scalability study, we ran acoustics and Euler applications on the Shaheen supercomputer over a range of 1, 4, 16, 64, 256, 1,024, 4,096, 16,384 and 65,536 cores. All the runs are in the VN mode. The experiment directory is `weak_scaling_experiments`. Under this directory, the directories `Euler` and `acoustics` will be found, for the Euler and acoustics scaling tests respectively. Under each of those directories, a directory for the experiment script, `experiment`, and another directory for the resulted profiles, `profile`, will be found. We extracted scaling data from the resulted profiles in `profile`. The application script for the acoustics test is `acoustics.py`; found in the directory `experiment`. Before running the script, a `make` command should be issued to generate the Fortran extension modules as described in the instructions in Section A.2. The application script for the Euler test is `shockbubble.py`, and will be found under the `experiment` directory. The grid size and the final time for the solution `q` will be adjusted by the script depending on the number of cores used for running the code.

In the strong scalability study, we ran acoustics application on 16 to 16,384 cores in the VN mode. The experiment directory is `strong_scaling_experiment`. The application script is `acoustics.py` that is found under the `experiment` directory. The profiles for these runs are available under the directory `profile/strong_scaling`.

The format of the name of the profile in the weak and strong scaling tests is `profile<i>_<j>`, where `<i>` is the process ID for which the profile was generated and `<j>` is the number of cores used for this run. The revision of the PyClaw github repository that we used is ec035e4f06 (`scaling_shaheen` branch).

The script for the secondary experiment that we made to analyze the strong scal-

ability results is `strong_scaling_experiment/experiment/acoustics_serial.py`. Table A.2 shows the directories for the profiles of those runs and their naming format. All those profiles' directories are found under `strong_scaling_experiment/profile`. We ran this experiment on one, four and sixteen cores. For each number of cores, we varied the size of the local partition of the grid from 32^2 to 1024^2 (increasing the grid size by a factor of four). This can be done by setting the parameter `loc_dim` by the size of one dimension of the local partition of the grid (i.e from 32 to 1024, increasing the size by a factor of two).

Table A.2: Profiles for the strong scaling secondary analysis experiment. The number `<i>` is the number of grid cells in one dimension and `<j>` is the process ID.

Experiment	Profiles directory	Profile name format
One-core	`strong_scaling`	`profile_serial_mx_<i>_<j>`
Four-core	`four_cores`	`profile_4_cores_mx_<i>_<j>`
Sixteen-core	`sixteen_cores`	`profile_16_cores_mx_<i>_<j>`

A.6 Figures Reproducibility

All the figures in Chapter III that presents PetClaw scalability results are reproducible. They extract the data from the generated performance profiles (except the Python loading figure, Figure III.10, where the data is hardcoded in the Python script). The scripts for reproducing the figures are found in each experiment directory under a subdirectory named `figures`.

Bibliography/References

[1] K. T. Mandli, A. Alghamdi, A. Ahmadia, D. I. Ketcheson, and W. Scullin, "Using Python to construct a scalable parallel nonlinear wave solver," in *Proceedings of Python for Scientific Computing Conference 2011 (SciPy 2011) (to appear)*, 2011.

[2] D. I. Ketcheson, K. T. Mandli, A. Ahmadia, A. Alghamdi, M. Quezada, M. Parsani, M. G. Knepley, and M. Emmett, "Accessible, Extensible, Scalable Tools for Wave Propagation Problems," 2011. [Online]. Available: http://arxiv.org/abs/1111.6583

[3] A. Alghamdi, A. Ahmadia, D. I. Ketcheson, M. G. Knepley, K. T. Mandli, and L. Dalcin, "Petclaw: A scalable parallel nonlinear wave propagation solver for Python," in *High Performance Computing Symposium 2011*, 2011.

[4] "About PyClaw," [accessed 21-December-2011]. [Online]. Available: http://numerics.kaust.edu.sa/pyclaw/about.html

[5] C. G. Bell, "The future of high performance computers in science and engineering," *Commun. ACM*, vol. 32, pp. 1091–1101, September 1989. [Online]. Available: http://doi.acm.org/10.1145/66451.66457

[6] P. S. Pacheco, *Parallel Programming with MPI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[7] R. J. Leveque, *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems (Classics in Applied Mathematics Classics in Applied Mathemat)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2007.

[8] R. J. LeVeque, *Finite Volume Methods for Hyperbolic Problems*. Cambridge: Cambridge University Press, 2002.

[9] R. J. Leveque and S. Mitran, "Wave-propagation methods and software for complex applications," 2002.

[10] R. J. Leveque, "Clawpack 4.6 documentation," [accessed 31-July-2011]. [Online]. Available: http://kingkong.amath.washington.edu/clawpack/users/index.html

[11] R. Leveque, "Python tools for reproducible research on hyperbolic problems," *Computing in Science & Engineering*, vol. 11, no. 1, pp. 19–27, 2009. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2009.13

[12] R. J. Leveque, "Clawpack 4.3 documentation," [accessed 31-July-2011]. [Online]. Available: http://www.amath.washington.edu/~ketch/clawdoc/index.htmll

[13] D. Ketcheson, 2011, personal communication.

[14] ZPL Team. (2005) ZPL web page. [Online]. Available: http://www.cs.washington.edu/research/zpl/home/index.html

[15] H. Wu, G. Turkiyyah, and W. Keirouz, "ZPLClaw: A parallel portable toolkit for wave propagation," in *Proceedings of ASCE Structures 2000 Congress*, 2000.

[16] S. Mitran, 2011, personal communication.

[17] X. Cai, H. Petter, and H. Moe, "On the performance of the Python programming language for serial and parallel scientific computations,"

Scientific Programming, vol. 13, no. 1, pp. 31–56, 2005. [Online]. Available: http://iospress.metapress.com/index/XAWR0DX9XG61NB7Q.pdf

[18] (2011) About Python. [Online]. Available: http://www.python.org/about/

[19] H. P. Langtangen, *Python Scripting for Computational Science (Texts in Computational Science and Engineering)*, 3rd ed. Springer, Feb. 2009. [Online]. Available: http://www.worldcat.org/isbn/3540739157

[20] A. Logg, H. P. Langtangen, and X. Cai, *Past and Future Perspectives on Scientific Software*. Heidelberg: Springer, 2009, ch. 23, pp. 321–362.

[21] S. T. Scripps and M. F. Sanner, "Python: A programming language for software integration and development," *J. Mol. Graphics Mod*, vol. 17, pp. 57–61, 1999.

[22] (2011) What is numPy? [Online]. Available: http://docs.scipy.org/doc/numpy/user/whatisnumpy.html

[23] "Numpy reference," [accessed 31-July-2011]. [Online]. Available: http://docs.scipy.org/doc/numpy/reference/

[24] P. Peterson, *F2PY Users Guide and Reference Manual*. [Online]. Available: http://cens.ioc.ee/projects/f2py2e/usersguide

[25] "PyClaw documentation," [accessed 1-August-2011]. [Online]. Available: http://numerics.kaust.edu.sa/pyclaw/index.html

[26] A. R. Terrel, "Ignition Riemann project," [accessed 1-August-2011]. [Online]. Available: http://andy.terrel.us/ignition/modules/riemann.html

[27] R. J. Leveque, D. I. Ketcheson, and K. T. Mandli, "Rewriting arrays for interleaved storage," https://github.com/clawpack/doc/wiki/Array-rewrite, [Online; accessed 18-July-2011].

[28] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc users manual," 2010. [Online]. Available: http://www.mcs.anl.gov/petsc/petsc-as/documentation

[29] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[30] R. W. Sebesta, *Concepts of Programming Languages*, 9th ed. USA: Addison-Wesley Publishing Company, 2010.

[31] C. Sosa and B. Knudson, *IBM System Blue Gene Solution: Blue Gene/P Application Development*, 4th ed., 2009.

[32] M. Q. de Luna, "Nonlinear wave propagation and solitary wave formation in two-dimensional heterogeneous media," Master's thesis, King Abdullah University of Science and Technology (KAUST), Thuwal, Makkah Province, Saudi Arabia, May 2011.