IMPROVING SCIENTIST PRODUCTIVITY, ARCHITECTURE PORTABILITY, AND APPLICATION PERFORMANCE IN PARFLOW

by

Michael Burke

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

May 2020

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Michael Burke

Thesis Title: Improving Scientist Productivity, Architecture Portability, and Application Performance in ParFlow

Date of Final Oral Examination: 11th May 2020

The following individuals read and discussed the thesis submitted by student Michael Burke, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Dr. Catherine Olschanowsky, Ph.D.	Chair, Supervisory Committee
Dr. Michael Ekstrand, Ph.D.	Member, Supervisory Committee
Dr. Alejandro Flores, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Dr. Catherine Olschanowsky, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

ACKNOWLEDGMENTS

I am sincerely grateful to my advisor, Dr. Catherine Olschanowsky, for her support, guidance, and encouragement. She has greatly helped me develop my technical, professional, and writing skills. The many opportunities she presented to me have advanced my career to a point I previously found unimaginable, and attending conferences and workshops I would have otherwise never considered. I am additionally grateful to her for providing me with a graduate assistantship and funding me over the course of my Master's degree.

I would like to thank my supervisory committee members Dr. Michael Ekstrand and Dr. Alejandro Flores for their feedback on my proposal, thesis, and oral defense. Help refining specific contributions with Dr. Ekstrand was very valuable.

I would also like to thank Dr. Reed Maxwell at Colorado School of Mines and Dr. Laura Condon at University of Arizona. Their direct participation enabled this thesis and its research to be possible, providing valuable perspectives from a computational scientist standpoint.

ABSTRACT

Legacy scientific applications represent significant investments by universities, engineers, and researchers and contain valuable implementations of key scientific computations. Over time hardware architectures have changed. Adapting existing code to new architectures is time consuming, expensive, and increases code complexity. The increase in complexity negatively affects the scientific impact of the applications. There is an immediate need to reduce complexity. We propose using abstractions to manage and reduce code complexity, improving scientific impact of applications.

This thesis presents a set of abstractions targeting boundary conditions in iterative solvers. Many scientific applications represent physical phenomena as a set of partial differential equations (PDEs). PDEs are structured around steady state and boundary condition equations, starting from initial conditions.

The proposed abstractions separate architecture specific implementation details from the primary computation. We use ParFlow to demonstrate the effectiveness of the abstractions. ParFlow is a hydrologic and geoscience application that simulates surface and subsurface water flow. The abstractions have enabled ParFlow developers to successfully add new boundary conditions for the first time in 15 years, and have enabled an experimental OpenMP version of ParFlow that is transparent to computational scientists. This is achieved without requiring expensive rewrites of key computations or major codebase changes; improving developer productivity, enabling hardware portability, and allowing transparent performance optimizations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

HPC – High Performance Computing

DSL – Domain Specific Language

eDSL – Embedded Domain Specific Language

CONUS – Continental United States

UVA – Unified Virtual Addressing

MPI – Message Passing Interface

GPU – Graphics Processing Unit

GPGPU – General Purpose Graphics Processing Unit

CHAPTER 1

INTRODUCTION

Legacy scientific applications represent significant investments by universities, engineers, and researchers and contain valuable implementations of key scientific computations. Such applications are used in a wide range of research fields including medical imaging [25], industrial manufacturing, climate modeling [24, 2], geology, hydrology, and others. Computational scientists continuously update these applications to improve the underlying mathematical models and port the code to the latest hardware. However, years of updates to mathematical formulations, porting to new hardware, and optimizing for new hardware capabilities complicate code bases, making them difficult to maintain and extend. Computational scientists need abstractions that separate key computations from hardware specific implementation details, isolating the code complexity that comes with these issues.

There is a significant need for abstractions to provide the three P's: *Productivity*, *Portability*, and *Performance*. *Productivity* is paramount, representing how well and easily a scientific programmer can implement new simulations of scientific models; productivity directly correlates to scientific impact. As new hardware emerges, implementation details in the code change. Abstractions must provide a layer between computations and the necessary underlying implementations to allow for architecture portability. This additionally contributes to scientific impact, as the longer computations take the less research can be performed in a given period of time. As new architectures develop, they offer potential increases in computational throughput. *Portability* is the ability to run an application across multiple architectures. Porting existing code to new architecture is time consuming and expensive. Abstractions allow for core computations to be separated from architecture specific implementation details. This allows for portability without the need to rewrite computations. Therefore abstractions that manage this process are highly desirable. *Performance* is a measurement of how long an application takes to solve a particular problem. Abstractions must be flexible enough to allow for architecture specific optimizations. Different architectures have different implementation requirements to achieve good *performance*, and this must be realized without significant changes to the computations themselves.

This thesis presents a series of abstractions designed to improve scientific programmer productivity by separating architecture specific code details from the primary computations. This separation provides a way to implement architecture specific optimizations without obfuscating code, and without the need to rewrite computations. The abstractions are generally applicable to many applications, and are demonstrated with ParFlow, a hydrologic modeling platform.

ParFlow is representative of a class of scientific applications that solve partial differential equations. These applications all include boundary conditions, and many frameworks exist to handle them including CHOMBO [32, 28, 3], AMReX [48], ForestClaw [10], and ClawPack [24]. Boundary conditions are critical to solving partial differential equations. There is a clear need for abstractions that maintain productivity while enabling architecture portability and improving performance.

1.1 Problem Statement

Code complexity in legacy scientific applications caused by years of updates prevents computational scientists from implementing new computations and mathematical models, degrades application performance, and impedes architecture portability.

1.2 Case Study: ParFlow

ParFlow is a hydrologic modelling application that simulates surface and subsurface water flow through porous materials. It is a highly modular platform, allowing for a wide range of simulations and research to be performed. The domains that are simulated in ParFlow may be based on real world geometry, such as the Continental United States, or may be synthetic. ParFlow first solves a steady state equation across the domain, then performs calculations to handle boundary conditions. The original developers of ParFlow saw the need for scalability, but presently this is limited to MPI only. Part of this thesis is to provide portable on-node, shared memory parallelism for multicore systems and accelerators.

Productivity. Over the course of its lifetime, the codebase in ParFlow has become highly complex. As a result, computational scientists have had difficulties in extending mathematical models. An instance of this is the addition of new boundary conditions. Computational scientists have been unable to add new boundary conditions to ParFlow models for 15 years due to code complexity issues. Boundary conditions are crucial to the development of new watershed models, and this represents a significant loss of scientific impact. This demonstrates the immediate need for productive abstractions.

Portability. Different problem models have different computational resource requirements. A small problem may have very low computational requirements and could be run on something like a laptop. Larger and more complex problems may have significantly greater computational requirements, and necessitate the use of a supercomputer. ParFlow must be able to run across these different compute resources, from a laptop to a leading class supercomputer. Available hardware changes between compute resources and over time, and different hardware architectures have specific implementation requirements. Rewriting code to fit these architecture specific requirements is prohibitively expensive. This is compounded over time as new hardware architectures are developed, requiring rewrites for each one. ParFlow has a clear need for architecture portable abstractions that do not require the rewriting of key computations.

Performance. Applications that solve complex problems take time to execute. The more complex the problem becomes, the longer it takes for the application to solve. Computational resources consume significant amounts of electricity, and the longer an application must run the most electricity is used. The world's leading supercomputer, Summit at Oak Ridge National Laboratory, consumes as much as 10 megawatts of energy [1]; as much as an entire city. This is both financially and environmentally costly. As computational scientists extend their models in ParFlow, the computational resources required increase. Performance must be improved to meet these new requirements, reducing financial and environmental costs, and improving scientific impact.

1.3 Contributions

This thesis contributes a proposed design of abstractions for boundary condition computations. The abstractions are generalizable to other scientific applications.

ParFlow is used to demonstrate the effectiveness of the abstractions. The abstractions have improved computational scientist productivity in ParFlow, and enable architecture portability.

An experimental OpenMP version of ParFlow was implemented using the abstractions. This demonstrates a realization of architecture portability in the abstractions, without significant rewriting of computations.

A performance study was conducted, comparing on-node shared memory performance of MPI, OpenMP, and CUDA versions of ParFlow. This study examines performance in different configurations of a real scientific application, and explores the complexities involved.

1.4 Organization

This work is organized into several chapters and sections. Chapter 2 provides background on ParFlow, boundary conditions, and architecture portability. Chapter 3 covers the abstractions developed, their design, and the steps necessary to perform boundary condition computations. Chapter 4 provides a performance study, comparing the different experimental versions of ParFlow. Chapter 5 contains a review of related work and other DSL abstractions. Chapter 6 concludes and summarizes this work.

CHAPTER 2

BACKGROUND

The abstractions developed in this work are applicable to a wide range of scientific applications. We demonstrate their effectiveness using the ParFlow application. This section provides a brief overview of ParFlow, with a focus on boundary conditions and domain decomposition. Challenges in architecture portability with respect to OpenMP and CUDA are also presented.

2.1 ParFlow

ParFlow is a hydrologic and geoscience application that integrates multiple watershed models [30, 5, 27, 23], and is part of the HydroFrame [38] project. The HydroFrame project aims to simulate ground water flow for the Continental United States (CONUS) and increase accessibility of integrated hydrologic simulations to a larger community of hydrologists and educators. ParFlow models the flow of water through porous material using Richards' equation integrated with overland flow using Manning's equation. ParFlow utilizes a coupled model for simulating surface and subsurface overland flows. It is a well-established research tool within the hydrology community, with on-going work to improve its mathematical models for new simulations.

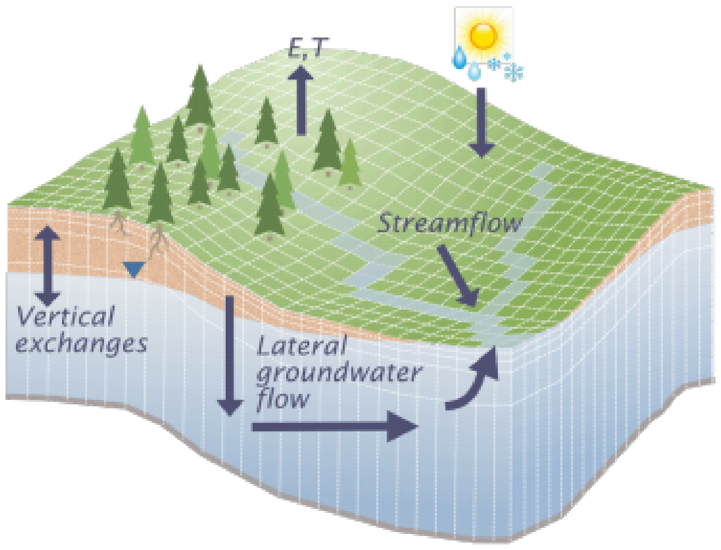


Figure 2.1: Example of a cube-like domain with an overland river on its surface (Provided by Dr. Laura Condon, University of Arizona; Dr. Reed Maxwell, Colorado School of Mines)

ParFlow simulates the flow of water through a domain. The domain may represent a real area and include historical atmospheric data, a real area with predicted or hypothetical atmospheric data, or represent a synthetic area that represents a physical area. Once configured, ParFlow discretizes the problem domain into a collection of grids. These grids are split across the X, Y, and Z axes. The Z axis represents depth of the domain, while X and Y represent the horizontal regions of the domain. Each grid is further discretized into subgrids, and each subgrid discretized into cells. Figure 2.1 is an illustration of a domain that has been divided into grids, subgrids, and cells. Cells represent the data points in the model, and are the operands to the computations. This particular discretization is called an orthogonal grid, where cells are equally distributed across the domain.

ParFlow uses Richards' equation to solve for variably saturated groundwater flow

(2.1).

$$S_S S_w \frac{\partial \psi_p}{\partial t} + \phi \frac{\partial S_w(\psi_p)}{\partial t} = \bigtriangledown \cdot q + q_s + \frac{q_e}{m'} \tag{2.1}$$

Richards' equation is well known [40], and describes the relationship between subsurface pressure heads, hydraulic conductivity, permeability of soil, saturation, and exchange rates with the surface. The discretized form of this equation forms the steady state stencil equation.

Computations are performed using stencil patterns. The use of stencil patterns in scientific computing is common [18, 13, 37]. A stencil pattern involves reading a cell and some set of neighbors to calculate a cell's value in the next iteration.

Grids, and their associated subgrids and cells, may exist on the edge of the model domain. Cells that reside in these locations are part of the boundary. Boundary cells lack at least one neighbor needed to perform the stencil computation. A boundary condition is the computations to be performed in place of the stencil, also called the steady state equation. A boundary cell will have a condition for each face along the boundary.

2.1.1 Boundary Conditions and CONUS

A cell in the very corner of a domain may have an overland flow boundary on its top face, and constant flux boundary conditions on its horizontal faces. Boundary cells represent areas with different physics and computational requirements. Examples of boundary conditions include overland flow on the surface, such as a river, atmospheric fluxes such as rainfall or snow, or constant flux values such as a river head. The cells in these regions account for water flow into or out of the domain, and are essential to scientific models. Boundary conditions can exist in surface regions, as well as subsurface regions.

ParFlow uses a free surface overland flow boundary condition to swap between solving subsurface and overland flow equations when water is ponded on the land surface of the model [27]. This is a coupled model that handles boundary condition cells on the surface that have water inflow or outflow occurring. As an example, consider a boundary condition cell on the surface of a model with rainfall. There is no immediate neighbor on the top of this cell, but the flow of rain into this cell must be included in computations. This coupled surface-subsurface model accounts for these overland boundary conditions. These are Neumann type boundary conditions, but can be switched to Dirichlet if necessary.

The HydroFrame project aims to create a more comprehensive model of the continental United States. CONUS 1.0 is a model that simulates subsurface and groundwater flows across a large section of the continental United States, but as can be seen in Figure 2.2 only a rectangular inset of the continent is modelled. CONUS 2.0 is an update to this model and will include more complex coastal regions and geological areas. Boundary condition development is critical to this new model in order to provide the necessary computations for these new regions.

Boundary conditions are computationally intensive, and the inclusion of large, complex coastal regions in CONUS 2.0 creates a need for improved performance. Additionally highly complex implementation details in the codebase prevent the development of new boundary conditions, posing a direct barrier to the development of CONUS 2.0. This highlights the immediate need for intuitive abstractions to improve for computational scientists to use in developing new boundary conditions. These abstractions must also provide architecture portability, and allow for improved

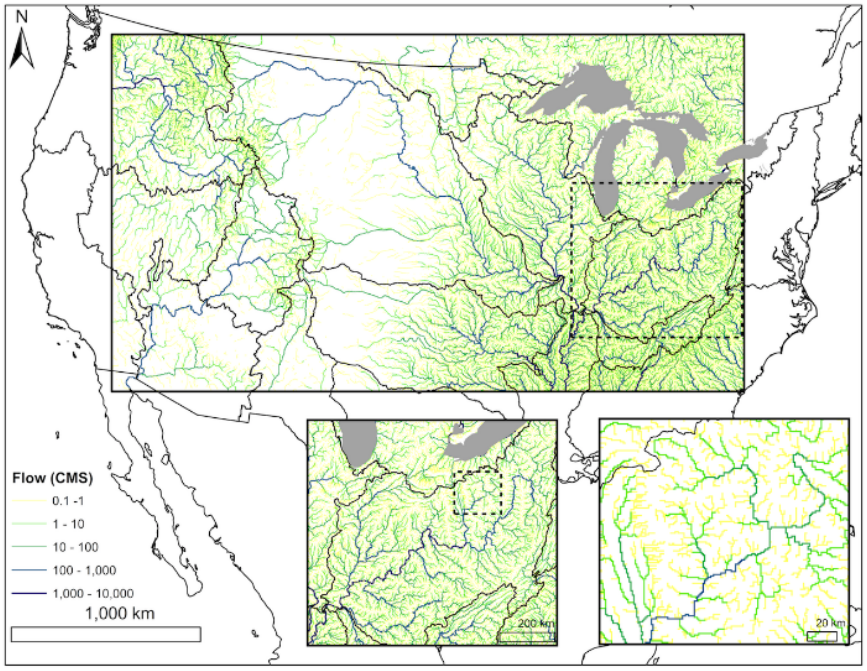


Figure 2.2: The CONUS 1.0 Domain [29]

performance in the CONUS 2.0 model.

2.2 Domain Specific Languages

A Domain Specific Language is a method of abstracting common idioms in a codebase. This is most often seen in terms of expressing complex or common looping patterns, but is also used for memory allocation and data structure manipulation. DSLs provide an interface for developers that reduces code complexity, improving productivity. Additionally, domain specific languages can help to enable architecture portability, as the back-end structure can be modified to accommodate new architectures without needing to significantly change the developer-facing interface. Similarly, domain specific languages help improve application performance by allowing performance optimizations to be done on the back-end. DSLs are often written in the form of a

```
GrGeomInLoop(i, j, k, gr_domain, r, ix, iy, iz, nx, ny, nz,
{
  ip = SubvectorEltIndex(f_sub, i, j, k);
  io = SubvectorEltIndex(x_ssl_sub, i, j, grid2d_iz);
  fp[ip] += ss[ip] * vol * z_mult_dat[ip] *
           (pp[ip] * sp[ip] * dp[ip] -
            opp[ip] * osp[ip] * odp[ip]);
});
```

Figure 2.3: An example of an iterative steady-state computation looping over the interior cells of a domain

library, but can also be embedded within a program, known as an eDSL (embedded DSL). In this context embedded means specific to the application, with no external library dependencies or additional compiler required. Other DSL efforts in atmospheric and hydrologic science are discussed in Related Work, Chapter 5.

2.2.1 Existing ParFlow eDSL

ParFlow contains an existing eDSL, primarily used for navigation of the model domain when performing computations. The eDSL abstractions encapsulate steady state computations, but leave boundary condition implementations exposed. As a result, complex control flow must be manually configured and managed by scientific programmers, hampering productivity and preventing architecture portability to GPU accelerators.

An example of the existing eDSL for steady state computations can be seen in Figure 2.3. This example, *GrGeomInLoop*, is used to navigate through the interior cells of the 3D model domain and perform computations accordingly. Iteration starts at *ix, iy, iz* up through *nx, ny, nz*. The *SubvectorEltIndex* abstraction handles mapping the iterators to memory locations for scientific programmers, providing a usable accessor index for reading and writing into data.

2.3 Architecture Portability

Legacy scientific applications have long lifespans, and over the course of those lifetimes hardware changes dramatically. ParFlow was originally developed for single core processors and modern processors are much different. Abstractions provide a single front-end view of computations, and allows complex architecture-specific code to be hidden. This section describes two target paradigms: multi-threaded execution through the OpenMP framework, and many-core GPU execution through CUDA.

2.3.1 OpenMP

OpenMP is a community driven API for directive based multi-threaded, shared-memory programming [35]. Compile-time directives are inserted by the programmer to indicate how work should be distributed and where threads must synchronize, or when work can or cannot be done in parallel. Additional directives can be specified to indicate whether a variable can be shared between all threads, or must be private to each thread.

A set of commonly used directives, also called pragmas, include *omp parallel*, *omp for*, *omp master*, and *omp single*. The *omp parallel* directive declares that a region of code should be performed in parallel, with optional clauses to declare specific variables shared or private. The *omp for* directive specifies that a loop should have its iteration

```c
/* Create a parallel region with a work-sharing for on the loop */
#pragma omp parallel for
for(int i = 0; i < N; i++)
    A[i] += B[i] * C[i];
```

Figure 2.4: Example of a streaming computation with a combined parallel-for directive

space divided among threads when inside of an *omp parallel* region. The *omp parallel* and *omp for* directives can be combined into *omp parallel for* when only a loop needs to be performed in parallel. Figure 2.4 shows an example of this, where a parallel region is created and the for loop distributed among threads.

The *omp master* and *omp single* directives are used within parallel regions, and indicate that only a single thread should execute a given block of code. The *master* directive means only the thread with id 0 will execute, and all other threads can continue without synchronization. The *single* directive means the first thread to encounter the directive will execute, but all threads must synchronize at the end of the directive. Figure 2.5 illustrates this by creating a parallel region, in which only the master thread will print the first statement and only a single thread will print the second.

OpenMP provides more directives, all with optional clauses to provide different functionality including memory management, scheduling, and synchronization. Exposed OpenMP directives introduce additional layers of complexity in managing parallelism and synchronization. As a result the directives must be abstracted away from scientific programmers in order to reduce code complexity and maintain productivity.

2.3.2 CUDA

GPU accelerators offer enormous parallelism, but existing code must be carefully adapted to make use of them. A GPU accelerator consists of thousands of threads that are grouped together in warps, with 32 threads per warp. Execution on a GPU accelerator is performed through the use of Kernels. A kernel is a function compiled by the GPU compiler for execution on the accelerator. Invoking a kernel requires the transfer of data in host (CPU) memory to device (GPU) memory, seen in Figure 2.6.

```c
/* Create a parallel region */
#pragma omp parallel
{
    /* Each thread gets its own ID */
    int tid = omp_get_thread_num();

    /* Only thread 0 executes, other threads don't sync */
    #pragma omp master
    {
        printf("Hello from master!\n");
    }

    /* First thread to reach this block executes */
    #pragma omp single
    {
        printf("Hello from thread %d!\n", tid);
    } /* All other threads implicitly sync here */
}
```

Figure 2.5: Example of the *master* and *single* directives inside an explicit *parallel* region

This transfer can be done explicitly, or though the use of a memory manager such as CUDA Unified Virtual Addressing [42] (UVA). UVA allows the CUDA runtime to map host and device memory into a single address space, and automatically transfer memory as necessary. This introduces costly overhead, but greatly simplifies development. An example of a kernel performing a streaming computation can be seen in Figure 2.7. This example assumes the transfer of data from host to device is being done through UVA, or has otherwise already occurred. Memory transfers occur through the PCIe bus, which has significant overhead costs. This transfer can become prohibitively expensive and must be managed carefully to achieve good performance on a GPU.

The layout of threads to be used in computation on the GPU must be specified. GPU accelerators allow for threads to be configuring in different dimensions, affecting memory access patterns. This configuration of threads can be seen in Figure 2.7 when

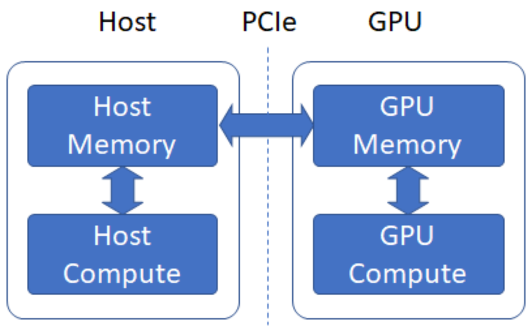


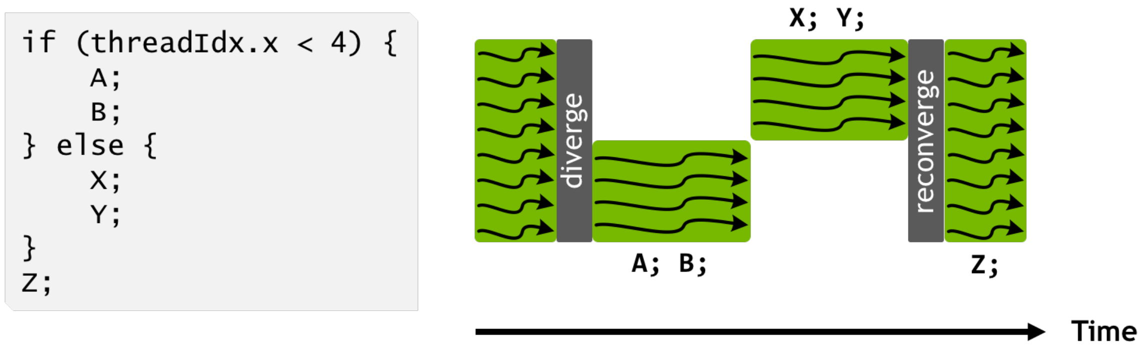
Figure 2.6: Memory transfer between CPU and GPU memory

the host invokes the kernel, lines 10 through 12. When the kernel is invoked, each thread in the warp calculates a starting index based on the configured thread layout, seen on line 3. A striding factor is calculated similarly on line 4. This makes the loop parallel, with each thread accessing data in independent indices than all other threads.

When a kernel is invoked, execution on the GPU begins. All threads in a warp operate in lockstep, only differing in the data they operate on. This produces an issue known as thread divergence, in which one thread encounters a conditional branch that others do not. When this occurs, all other threads that do not enter this branch must perform a no-op instruction [20], and do nothing until the branch is complete. This then repeats for the threads that entered the branch in turn. Figure 2.8 illustrates this, where a group of threads encounters a conditional branch. Half of the threads will enter one branch, executing statements A and B, while the other half remains idle and performs a no-op instruction. Once the first half has finished executing, the second half will execute statements X and Y, with the first half similarly remaining idle. All threads then synchronize and can continue in lockstep, executing statement Z. Thread divergence results in loss of performance and wastes compute resources, and is an area of active research [43, 19].

```
/* CUDA Kernel */
__global__ void foobar(int N, double *A, double *B, double *C) {
  int start_idx = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = start_idx; i < N; i += stride)
    A[i] += B[i] * C[i];
}
/* Invoking from host in some function */
void invoke_foobar(int N, double *A, double *B, double *C) {
  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  foobar<<<numBlocks, blockSize>>>(N, A, B, C);
}
```

Figure 2.7: Example of a CUDA kernel and the process for invoking it



```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

Figure 2.8: Thread divergence on a GPU

The implementation details of GPU accelerators become complicated quickly, increasing code complexity through memory management, synchronization, kernel invocations, and control flow. These details must be abstracted from scientific programmers to ensure productivity, while still providing performance benefits.

CHAPTER 3

BOUNDARY CONDITION ABSTRACTIONS

The proposed boundary condition abstractions are designed to improve productivity, provide portability, and allow architecture-specific optimizations to improve performance. The abstractions separate architecture-specific implementation details from primary computations. This reduces code complexity without requiring computational scientists to rewrite or reformulate computations, improving *productivity.* Several architecture-specific backends can be implemented without changing forward-facing code, providing *portability.* Different backend implementations can then have architecture-specific optimizations applied, improving *performance.* Boundary condition computations are used in several ways, and ensuring long-term productivity improvements requires the abstractions provide *full coverage.* Full coverage means that all use cases of boundary condition computations are fully encapsulated by the abstractions. This chapter will cover the design approach for the new abstractions, the abstractions themselves, and the architecture portable backends for OpenMP.

3.1 Design and Approach

Computational scientists actively participated in the design of the abstractions. Their participation ensured that productivity goals were met, resulting in a clear and intuitive design. We implemented backends iteratively during the development of the frontend abstractions to verify minimal overhead, and ensure portability goals were met. The result is a set of powerful abstractions that computational scientists are comfortable working with, and that computer scientists are able to use to produce efficient code.

Boundary conditions are interacted with in several sections of the application. During the configuration stage the different types of boundary conditions are set and the domain is defined. This involves setting the source of boundary condition values (Section 3.1.1), and setting appropriate data for each time step in the model (Section 3.1.2). The values are then used at each timestep to perform boundary condition computations (Section 3.1.3). The following sections detail this process.

3.1.1 Setting the source of Boundary Condition Values

Boundary conditions must be configured before the execution of the model begins. The first step in this configuration is to define the sources for boundary condition values. Abstractions were provided to facilitate this step in an intuitive way that is self-describing. These abstractions manage the allocation, storage, control flow, and enumeration of the necessary data for different boundary condition types.

Figure 3.1 is an example of setting up source values for a *FluxConst* boundary condition type. *Do_SetupPatchTypes* (line 1) accepts a list of boundary condition types and the details of how their source values are configured. Each boundary condition type will have a *SetupPatchType* (line 4) entry inside of the *Do_SetupPatchTypes* interface. *SetupPatchType* contains the configuration details for a specified boundary condition type and provides an interface for allocating memory, populating initial values, and storing the data. *NewTypeStruct* (line 6) allocates a data structure of the appropriate type used to store data source information. *ForEachInterval* (line 9) iterates over all time steps for the model being executed, and contains the implementation details to populate source values. *StoreTypeStruct* (line 13) stores the data structure for later retrieval.

```
 1 Do_SetupPatchTypes(public_xtra, num, i,
 2 {
 3   ...
 4   SetupPatchType(FluxConst,
 5   {
 6     NewTypeStruct(FluxConst, data);
 7     (data->values) = ctalloc(double, div);
 8     ForEachInterval(div, num)
 9     {
10       /* Populate data->values with the appropriate sources */
11     }
12     StoreTypeStruct(public_xtra, data, i);
13   });
14   ...
15 });
```

Figure 3.1: Setting the data source for a FluxConst boundary condition

3.1.2 Setting Data For Each Time Step

Once the source locations for each boundary conditions data has been set, data for each time step in the model is prepared. Figure 3.2 shows an example of setting initial timestep values for a *FluxConst* boundary condition. *ForEachInterval* (line 1) iterates over all timesteps in the model being executed. *Do_SetupPatchIntervals* (line 2) accepts a list of boundary condition types and the details of how their values are configured. Each boundary condition type will have a *SetupPatchInterval* (line 5) entry inside of the *Do_SetupPatchIntervals* interface. *SetupPatchInterval* contains the configuration details for a specified boundary condition type and provides an interface for allocating memory, retrieving source values, populating initial timestep values, and storing data.

```
 1 ForEachInterval(div, num) {
 2   Do_SetupPatchIntervals(public_xtra, num, i,
 3   {
 4     ...
 5     SetupPatchInterval(FluxConst,
 6     {
 7       NewBCPressureTypeStruct(FluxConst, i_data);
 8       BCPressureDataBCType(bc_data, i) = FluxBC;
 9       GetTypeStruct(FluxConst, data, public_xtra, i);
10       FluxConstValue(i_data) = (data->values[num]);
11       BCPressureDataInternalValue(bc_data, i, num) = (void*, i_data);
12     });
13     ...
14   });
15 }
```

Figure 3.2: Setting boundary condition data for a particular timestep

NewBCPressureTypeStruct (line 7) allocates a new data structure for the specified boundary condition. *BCPressureDataBCType* (line 8) assigns the categorical type of boundary condition, used to determine what computations to perform using the data. For example the *FluxConst* and *FluxVolumetric* types have different details for configuring initial values, but use the same mathematical computations and are of the *FluxBC* category. *GetTypeStruct* (line 9) retrieves the source values that were previously configured. *FluxConstValue* (line 10) is an acccessor into the data structure, used in this example to store the particular value for a given timestep. Each boundary condition type has its own set of accessors accordingly. *BCPressureDataInternalValue* (line 11) stores the timestep data structure for retrieval and use in computations.

3.1.3 Performing Boundary Condition Computations

Boundary condition computations are used in different ways throughout the ParFlow codebase. It is necessary to be able to apply different computations for different types of boundary conditions. There are instances in which all boundary condition types perform the same set of computations, such as to adjust coefficients when dealing with symmetric Jacobian matrices. Different computational use cases have been identified and a set of consistent abstractions developed.

Boundary condition computations can be decomposed into five primary sections. A simple example of a Flux boundary condition computation, pulled from the RichardsJacobianEval function in ParFlow, can be seen in Figure 3.3. The first section is a block that must occur before looping over the cells of the boundary condition patch, named *BeforeAllCells* (line 2). This allows for a block of code to be executed unconditionally before boundary condition computations are performed across cells. An example of this may be some uniform scalar setup from a function call, so that the function call is not repeated for every cell iteration. Another example may be a nested boundary condition loop that counts some number of values across all cells for data allocation.

The next section is *CellSetup* (line 4). This is a region that is used for setting up values to be used in the computation on the current cell, such as preparing accessor indices or calculating a scalar used in computations. Previously control flow was managed manually by computational scientists to determine which directions were valid for the current boundary condition cell, increasing code complexity. This is replaced with the next section, containing a set of *FACE* (lines 8 - 13) entries. Each *FACE* specifies the direction it represents, and contains the appropriate computations to be executed. A boundary condition cell may have multiple valid directions, but only one is valid at a given iteration. For example a cell on the corner of the domain may have two valid faces, but the appropriate face computations will happen in separate iterations.

The final two sections are *CellFinalize* (line 14) and *AfterAllCells* (line 19).

CellFinalize is the counterpart to *CellSetup*, and is where the result of computations performed in the *FACE* section are utilized. *AfterAllCells* is similarly the counterpart to *BeforeAllCells*, and will unconditionally execute after every cell in the boundary condition has been iterated over. The *DoNothing* keyword (lines 2, 19) is provided for sections that are unnecessary for the computations being executed.

```
 1 ForPatchCellsPerFace(FluxBC,
 2                      BeforeAllCells(DoNothing),
 3                      LoopVars(i, j, k, ... ),
 4                      CellSetup(
 5                        {
 6                          im = SubmatrixEltIndex(i, j, k, J_sub);
 7                        }),
 8                      FACE(Left, { op = wp; }),
 9                      FACE(Right, { op = ep; }),
10                      FACE(Down, { op = sop; }),
11                      FACE(Up, { op = np; }),
12                      FACE(Back, { op = lp; }),
13                      FACE(Front, { op = up; }),
14                      CellFinalize(
15                        {
16                          cp[im] += op[im];
17                          op[im] = 0.0;
18                        }),
19                      AfterAllCells(DoNothing)
20      );
```

Figure 3.3: Example of a Flux boundary computation in Richards Jacobian

Different regions of code in ParFlow loop over boundary condition cells at different points. Control flow for branching on boundary condition types may be hoisted far away from the loop itself, such as outside of the function. Computations may also apply to all boundary condition types, such as for handling contributions for a symmetric Jacobian matrix. For these cases, the *ALL* keyword (Figure 3.4 line 1) can be used in place of a specific boundary condition type, causing the loop to execute unconditionally. There may also be no special computations for each face direction. In this case, the *ForEachPatchCell* abstraction is provided (Figure 3.4, lines 4 and 28). Finally, the iteration space may need to reach into ghost cells. A ghost cell is used to exchange data between MPI processes. The *ForPatchCellsPerFaceAndGhost* abstraction is provided to facilitate this, and works identically to the *ForPatchCellsPerFace* abstraction.

The use of these abstractions removes the need for computational scientists to manage control flow manually, and separates implementation specific details from the computations. This is done in a way that does not require computations to be rewritten, and allowing computational scientists to continue writing code in a way they are comfortable with, maintaining and improving productivity. These abstractions are expressive and flexible, and can be nested within each other. An example of this can be seen in Figure 3.4, where the patch is first iterated over to count the number of cells for data allocation, iterated over a second time to calculate a necessary scalar, and then iterated over a third time to compute and store data.

These abstractions separate the concerns of the mathematical computations used in boundary conditions from the specific implementation details. Code complexity is reduced, with cleanly separated blocks that clearly define what is being done inside of them. Control flow is no longer dealt with directly by computational scientists. This provides architecture portability, as the implementation and computations are no longer tightly coupled.

The reduced code complexity helps to highlight potential loop fusions. An example of this can be seen in Figure 3.4. This figure shows how the original code ordered the sequences of loops. A full iteration over all cells in the boundary condition is performed to count cells for data allocation. A second iteration over all cells in the boundary condition performs an accumulation into a variable. Finally, if that variable

```
ForPatchCellsPerFace(ALL,
                     BeforeAllCells({
                         num_cells = 0;
                         ForEachPatchCell(LoopVars(i, j, k, ... ),
                         {
                           num_cells++;
                         });
                         patch_values = ctalloc(double, num_cells);
                         area = 0.0;
                       }),
                     LoopVars(i, j, k, ... ),
                     CellSetup(
                       {
                         ips = SubvectorEltIndex(z_mult_sub, i, j, k);
                       }),
                     FACE(Left, { area += dy*dz*z_mult_dat[ips]; }),
                     FACE(Right, { area += dy*dz*z_mult_dat[ips]; }),
                     FACE(Down, { area += dx*dz*z_mult_dat[ips]; }),
                     FACE(Up, { area += dx*dz*z_mult_dat[ips]; }),
                     FACE(Back, { area += dx*dy; }),
                     FACE(Front, { area += dx*dy; }),
                     CellFinalize(DoNothing),
                     AfterAllCells(
                     {
                       if (area > 0.0)
                       {
                         vol_flux = FluxVolVal(interval_data) / area;
                         ForEachPatchCell(LoopVars(i, j, k, ... ),
                         {
                           patch_values[ival] = vol_flux;
                         });
                       }
                     })
      );
```

Figure 3.4: Example of a nested abstractions for setting up Flux Volumetric boundary condition data

is not zero, a third iteration over all cells in the boundary condition populates values for use in later computations. In the new abstraction it becomes clear that this first iteration is unnecessary. The number of cells is not used until after the *area* variable has been calculated, and so the first loop can be merged into the second. This can be seen in Figure 3.5, reducing total iterations from *3N* to *2N*.

```
ForPatchCellsPerFace(ALL,
                     BeforeAllCells({
                       num_cells = 0;
                       area = 0.0;
                     }),
                     LoopVars(i, j, k, ... ),
                     CellSetup({
                         ips = SubvectorEltIndex(z_mult_sub, i, j, k);
                         num_cells++;
                     }),
                     /* Face computations ... */
                     CellFinalize(DoNothing),
                     AfterAllCells({
                       patch_values = ctalloc(double, num_cells);
                       if (area > 0.0) {
                         vol_flux = FluxVolVal(interval_data) / area;
                         ForEachPatchCell(LoopVars(i, j, k, ... ),
                         {
                           patch_values[ival] = vol_flux;
                         });
                       }
                     })
      );
```

Figure 3.5: Example of setting up Flux Volumetric boundary condition data after a loop fusion

3.2 Backend Development

The new boundary condition abstractions have enabled the development of experimental ParFlow builds for OpenMP. OpenMP was chosen for on-node memory sharing performance, providing a wide range of compile-time hints and instructions for improved threading performance. The complex control flow in the original boundary condition implementations posed barriers to parallelism with OpenMP directives and thread divergence on GPU accelerators. The proposed abstractions separate computations from architecture specific details. This permits implementation details to be hidden and prevent increased code complexity. Beyond boundary condition loops, ParFlow contains several existing looping abstractions that were modified for

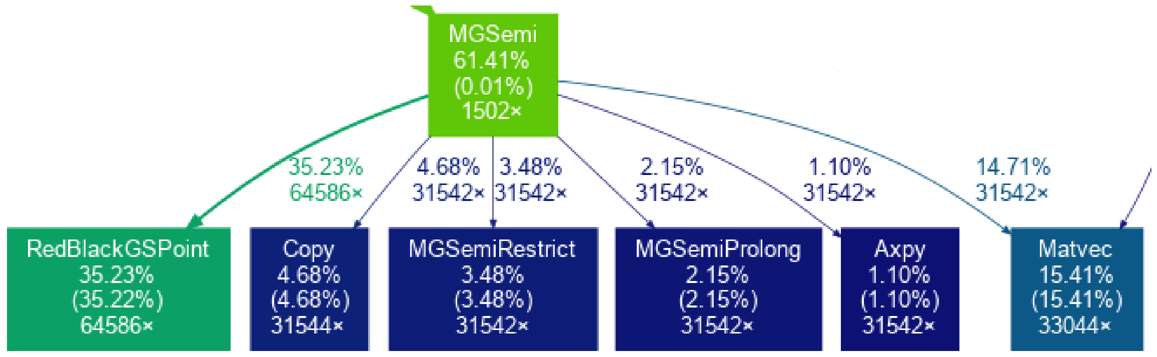


Figure 3.6: Example of a subsection of a call graph generated by gprof

portability with minimal forward-facing changes.

3.2.1 Dataflow and Profiling Analysis

Before implementing new backends, profiling of ParFlow was performed to identify time-dominating functions and their subroutines. Due to the highly configurable nature of ParFlow, this profiling focused on the test cases discussed further in chapter 4. Initial profiling was performed using the GNU gprof [17] utility. Figure 3.6 is a subsection of the call graph of one of the test cases, generated by gprof and run through the gprof2dot [15] utility for visualization. Each node represents a function and consists of the name of the function, percentages of total runtime, and number of times the function was called. The first number is the percent of cumulative time spent in the function or its subroutines against the total runtime of the application. The second number, listed in parenthesis, is the percentage of time spent directly within the function. The third number is total number of calls to the function.

Once time dominating functions were identified, dataflow analysis was performed. This was done in order to identify serial optimizations that can be applied to all versions of ParFlow, as well as to identify necessary synchronization barriers in

OpenMP. This was performed and recorded manually as a set of Macro Dataflow Graphs, which represent data dependencies, computational statements, and data points as nodes within a directed acyclic graph. Macro Dataflow Graphs and their use for automatic, compile-time transformations and optimizations is an on going area of research [14].

Graphs are read from left to right, top to bottom. Nodes indicate their meaning by their shape, and directed edges indicate data dependencies and data flow. Nodes that are shaded gray are immutable and cannot be transformed. Data nodes contain the name of the variable they represent, with a subscript to indicate when they have been assigned to in a single-static assignment view. In this context, commutative operations such as addition ($+=$) or subtraction ($-=$) into a data node is not considered a new assignment, as they can be rearranged with the same mathematical result. The full list of node types are:

- Rectangle - Data node, containing the variable name it corresponds to and a subscript to indicate its assignment count.
- Trapezoid - Statement node. These are seen as inverted triangles in other work, but due to software limitations a trapezoid was used in these graphs. These nodes represent loop patterns in this context.
- Rectangle with side-bars - Function call node.
- Directed edge - Data dependency and data flow.

A directed edge going into a node indicates a data dependency. A directed edge coming out of a node and into a data node indicates a new assignment to that data. A subsection of the dataflow graph for the function *NL_Function_Eval* can be seen in Figure 3.7. Starting from the top, a call to the *Density* function is made reading data from the *Pressure_0* node. This emits the data node *Density_0*. A call to *Saturation* is made, similarly reading data from *Pressure_0* and *Density_0*, and emitting *Saturation_0*. Two loops occur sequentially, reading data as indicated by the connected edges, and emitting nodes.

These graphs were prepared through analysis of the original codebase, and were used to identify areas that required parallelism barriers such as MPI communication. Potential loop transformations such as loop fusion [8, 7] or loop tiling [39] were also identified using these graphs.

Figure 3.7 shows that there are no direct loop-carry dependencies between the loops *GrGeomIn_0* and *GrGeomIn_1*. A loop-carry dependency is where one loop

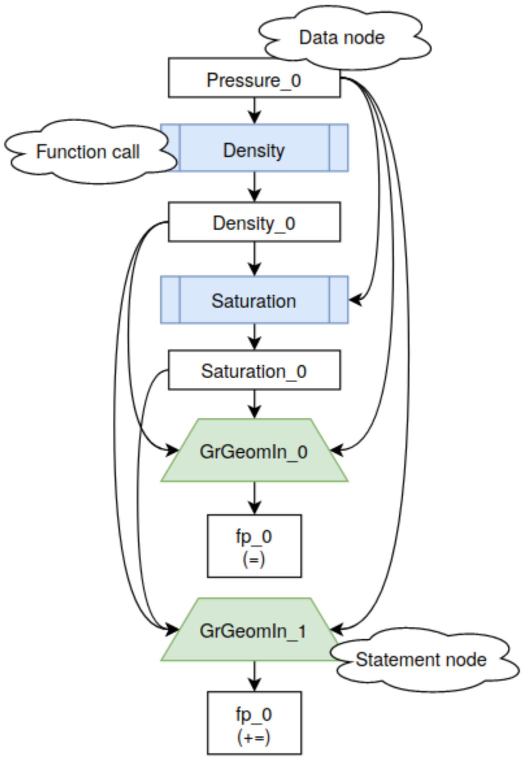


Figure 3.7: Subsection of a dataflow graph for *NL_Function_Eval*

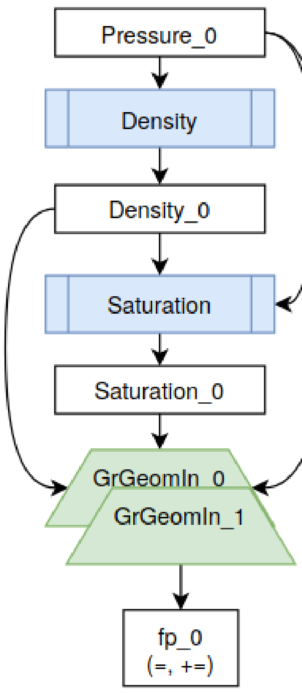


Figure 3.8: Subsection of a dataflow graph for *NL_Function_Eval* after fusion

modifies data that is used in a subsequent loop. This indicates there is a potential for loop fusion. Figure 3.8 is an example of the dataflow graph after performing a loop fusion. The full dataflow graph for *NL_Function_Eval* can be found in the appendix, showing it is possible to fuse 4 loops. This can be accomplished by reordering loops in the function and performing temporary storage transformations. These transformations were not applied in order to provide a baseline in the performance study, detailed in Chapter 4.

3.2.2 OpenMP

The design of the new boundary condition abstractions enabled rapid development of an OpenMP implementation. An additional parameter was added to abstractions, indicating one of several sets of OpenMP directives to be used across the interior looping structure. An example of this can be seen in Figure 3.9, where lines 4 and 13 contain an additional *NewParallel* clause. The *NewParallel* clause creates a new OpenMP parallel region and distributes the loop computation across multiple threads. Additional keywords exist to indicate that the loop is already in parallel region, make certain variables thread-private, perform reductions, or to skip implicit synchronizations with the *nowait* clause. This provides clear indications of what kind of parallelism is being applied without increasing code complexity. This parameterization was applied similarly to other looping patterns, such as for steady state equations.

```
 1 ForPatchCellsPerFace(ALL,
 2                    BeforeAllCells( ... ),
 3                    LoopVars(i, j, k, ... ),
 4                    NewParallel,
 5                    CellSetup( ... ),
 6                    FACE(Left, ... ),
 7                    /* Other faces ... */
 8                    CellFinalize(DoNothing),
 9                    AfterAllCells(
10                    {
11                      ...
12                      ForEachPatchCell(LoopVars(i, j, k, ... ),
13                                       NewParallel,
14                      {
15                        /* Loop body ... */
16                      });
17                    })
18      );
```

Figure 3.9: Truncated example of setting Flux Volumetric boundary condition data, with OpenMP keywords inserted

Time dominating functions were analyzed and several challenges to the implementation of OpenMP identified. These challenges include but are not limited to synchronization, data races, mixing MPI barriers with OpenMP, iteration calculations, and managing large parallel regions across multiple function calls.

Initial development was performed by analyzing each looping structure within a function to determine possible parallelism. Looping structures that were not immediately parallel had further analysis performed to identify methods that may provide parallelism. Examples of these include storage duplication and managing scatter-gather patterns. Once parallelizable, each looping structure was isolated into individual parallel regions. This means that at the beginning of the iteration space, a thread group would be requested. The body of the iteration space would be divided amongst threads and performed in parallel. Temporary scalars can be made thread-private with optional directives. At the end of the looping structure, all threads synchronize, and the program returns to serial execution.

Next, explicit parallel regions would be declared at an appropriate place in the beginning of the function. Parallel loops would be incrementally incorporated into the region to help manage and debug race conditions and synchronization issues. The use of a parallel region reduces the overhead associated with requesting thread groups and unnecessary synchronizations. Because of the abstractions, typically only a single keyword needed to be changed when moving a loop from an isolated parallel region to an incorporated one, or to add and remove barriers.

Once a subroutine was able to be run in parallel from start to end, the process was repeated in its parent function. Parallel regions were explicitly declared, and isolated parallel loops would be incorporated. This followed by incorporating the subroutines into the parallel region, with each thread entering it and maintaining the benefits of OpenMP directives. Significant data analysis was required for this to identify race conditions and eliminate unnecessary synchronization points.

As ParFlow was originally developed to be an MPI-Only application, there are frequent calls to perform updates between MPI processes. MPI calls in a hybrid

MPI-OpenMP implementation require explicit synchronizations so that data being exchanged between MPI processes is not modified by other threads. This is a common challenge when moving an application from MPI-only to a hybrid MPI-OpenMP implementation [36, 41]. The Scalasca performance tookit [16] was utilized to help identify regions of code resulting in synchronization issues, as well as general multi-threaded profiling.

3.2.3 Limitations

The use of keywords in development of OpenMP, such as *NewParallel*, poses limitations in regards to productivity. In order to safely add new mathematical formulations, computational scientists now need to understand parallelism concepts such as synchronization, race conditions, and more. A possible solution is to perform static analysis to determine where no synchronization is needed, or where a variable needs to be declared thread private. Transformations on the backend could then automatically be applied, such as removing synchronization barriers in OpenMP or CUDA. Addressing these limitations through static analysis is beyond the scope of this work.

CHAPTER 4

PERFORMANCE STUDY

This section details a performance study of experimental builds of ParFlow. A set of test cases were chosen with computational scientists. This performance study is done to examine the shared memory performance of MPI, OpenMP, and CUDA implementations of ParFlow. The OpenMP and CUDA versions are experimental. This study found that OpenMP outperforms or is competitive with MPI under certain configurations, and that CUDA outperforms MPI in some configurations.

4.1 Benchmark Suite

ParFlow is highly configurable, and different configurations result in large variations in performance characteristics. A model can be configured with different settings, including maximum number of solver iterations, tolerance values, number of time steps, domain sizes, and solver types. This creates an exponential number of combinations, not all of which may make sense. Computational scientists were consulted and a set of useful domains were chosen as test cases. These test cases were used to examine the feasibility of the experimental versions of ParFlow; analyzing performance in different domains and key sections of ParFlow. Performance was analyzed in the context of a single compute node to measure shared memory and GPU performance compared to the baseline MPI implementation. Selected test cases include: CONUS Terrain Following Grid (CONUS-TFG), CONUS Runoff (CONUS-RU), and ClayL.

CONUS-TFG is a subsection of the CONUS 1.0 model in Colorado, with multiple Z layers and real slope geometry. TFG stands for Terrain Following Grid. When using a terrain following grid, the orthogonal grid discussed in 2.1 is transformed to conform to the problem domain topography on both the surface and subsurface layers to exclude inactive areas [31]. This is beneficial when overland flow closely follow the topography.

CONUS-RU is a subsection of the CONUS 1.0 model in southern Colorado, with a single Z layer and real slope geometry. CONUS-RU is known as a ”parking lot” model where permeability is set to near 0, preventing water from entering the subsurface of the domain. This pools water across the domain, and identifies where rivers, streams, and sinks exist. Parking lot models are used frequently when developing simulations for new domains.

ClayL is a synthetic domain consisting of homogeneous soil and a large Z depth, with no complex terrain. ClayL is included in this study as it is used for acceptance testing of supercomputers in Europe.

4.2 Experimental Setup

Testing was performed on the R2 cluster at Boise State University [44], running from 1 to 28 cores on a single node. A node on the R2 cluster is configured with two Intel Xeon E5-2680 v4 14 core CPUs running at 2.4ghz, with 192GB of memory split into two NUMA nodes (96GB per CPU socket). CUDA performance was measured on a GPU node with two Intel Xeon E5-2680 v4 14 core CPUs running at 2.4ghz, with

256GB of memory split into two NUMA nodes, and an Nvidia Tesla P100 GPU. The links to the specific git commits used for each version can be found in Appendix A. ParFlow was compiled with GCC 7.2.0 using the O3 flag, and NVCC 10.0.130 was used to compile the CUDA version. MPICH 3.2.1 was used for MPI.

Results compare ParFlow running with MPI, the experimental OpenMP implementation, and an experimental CUDA version. The experimental CUDA version was provided by the Juelich Research Center in Germany. MPI communication is currently not supported in the experimental CUDA version, and is only run on one CPU core with one GPU accelerator. The CUDA version of ParFlow does not use the abstractions presented, instead being hard coded to run on GPU accelerators. However, the abstractions are able to support a unified CUDA implementation.

4.3 Results

This section presents the results of the performance study. It is important to note that ParFlow utilizes KINSOL, a nonlinear solver for algebraic systems that is part of the SUNDIALS library [21]. KINSOL is developed by Lawrence Livermore National Laboratory, and analysis for implementing OpenMP or CUDA in it is beyond the scope of this work. Analysis shows that the CUDA and OpenMP versions take a significant performance penalty due to the KINSOL library being single threaded. MPI does not see this penalty as every MPI process has its own copy of the application, allowing KINSOL to be run in parallel. MPI does not see this penalty as every MPI process has its own copy of the application, and so KINSOL can be run in parallel. The results for two time dominating functions, *NL_Function_Eval* and *MGSemi*, are included to exclude the overhead from KINSOL in OpenMP and CUDA. This is done

Table 4.1

MPI Timings with and without eDSL			
Timer	Timer	No eDSL	eDSL
CONUS-TFG	Total	41.01s	43.34s
	NL_Function_Eval	4.75s	5.31s
	MGSemi	20.95s	20.93s
CONUS-RU	Total	26.42	24.94s
	NL_Function_Eval	13.20	13.25s
	MGSemi	3.36	3.45s
ClayL	Total	98.04s	95.35s
	NL_Function_Eval	8.29s	8.34s
	MGSemi	46.07s	45.19s

to examine the performance of OpenMP, CUDA, and MPI in parallel regions of code. Configurations for each test case are detailed at the beginning of each subsection. Total runtime, and runtimes spent in *NL_Function_Eval* and *MGSemi* are examined. Full timing results for all test cases for 1 to 28 cores can be found in Appendix B.

A comparison between ParFlow without the abstractions and with the abstractions was run for each test case on 28 cores, seen in Table 4.1. The abstractions involve changes to the codebase. This can result in different compiler optimizations and execution orderings, resulting in time variations. System interrupts, file IO operations, and general system noise contribute further variations in timings. These results show that the abstractions do not have significant negative impact on runtime using 28 cores.

4.3.1 CONUS-TFG

The CONUS-TFG test was run in several MPI configurations. These are denoted as (X.1.1), (X.2.1), and (X.4.1) where X is a varying number of cores. ParFlow decomposes a domain as the product of the provided configuration. For example a

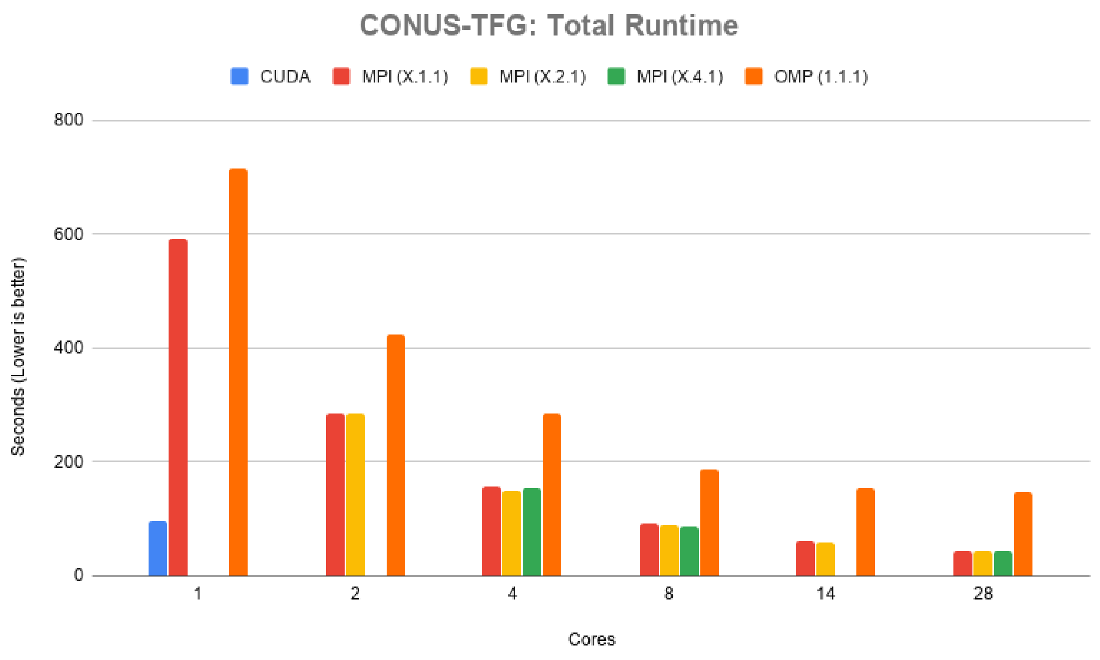


Figure 4.1: CONUS-TFG: Total runtime in seconds

decomposition of 7.4.1 will split the domain along the X axis 7 ways and the Y axis 4 ways, and distribute data to 28 cores. Domain decomposition has impact on memory layouts and domain traversal, and is under exploration. The OpenMP version was run on one MPI process, with a varying number of threads. Full timing results for 1 to 28 cores can be found in Appendix B Table B.1.

Figure 4.1 shows the total runtime for the CONUS-TFG test case in seconds. CUDA performed faster than MPI in all configurations until more than 8 cores are in use. The CUDA version solved the problem domain in 95.14 seconds. The MPI version performed equally between each of its configurations, with the fastest time of 41.87 seconds in the 14.2.1 configuration for a total of 28 cores. OpenMP was consistently slower than MPI in this test case, with 28 cores solving the problem domain in 146.08 seconds.

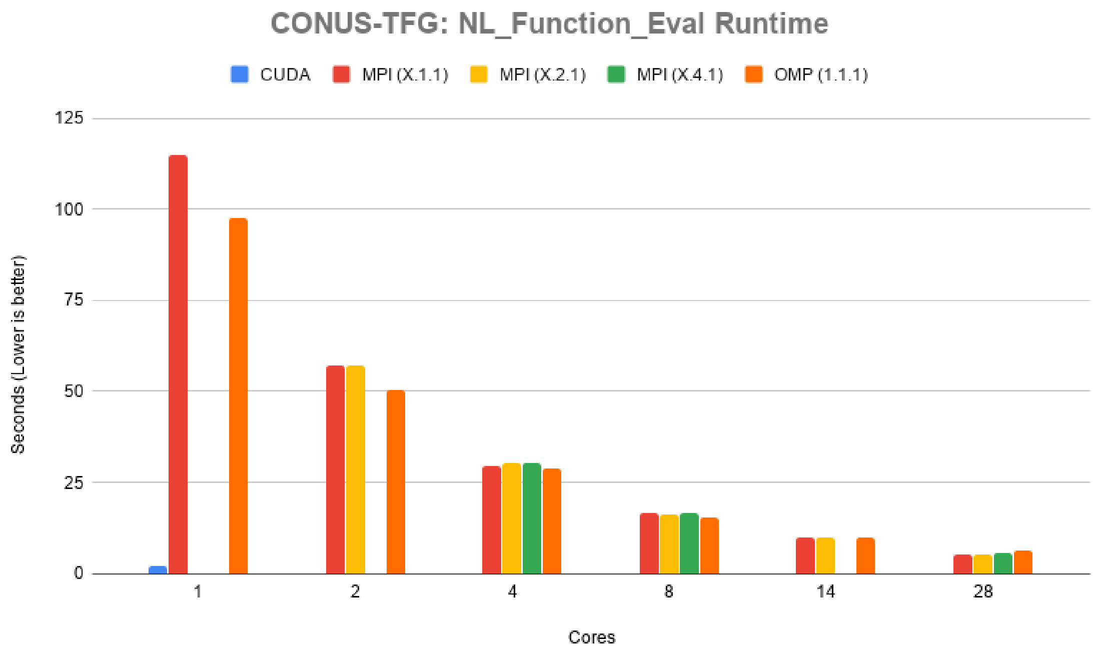


Figure 4.2: CONUS-TFG: Runtime for *NL_Function_Eval* in seconds

Figure 4.2 shows timing results for *NL_Function_Eval* in seconds. The CUDA version performed faster than all MPI configurations up to 28 cores, spending 2.14 seconds in this function. The MPI version performed equally between all configurations. The fastest time for MPI was in the 14.2.1 configuration, spending 5.30 seconds in the function. OpenMP performed faster than all MPI configurations until more than 8 cores are in use, at which point it became competitive. OpenMP spent 6.51 seconds in this function on 28 cores. Full timing results can be found in can be found in Appendix B Table B.2.

Figure 4.3 shows timing results for *MGSemi* in seconds. The CUDA version performed faster until 4 ore more cores are in use, spending 55.50 seconds in this function. MPI performed best in the X.4.1 configuration until 16 cores were in use, at which point all configurations performed equally. The fastest time for MPI was in

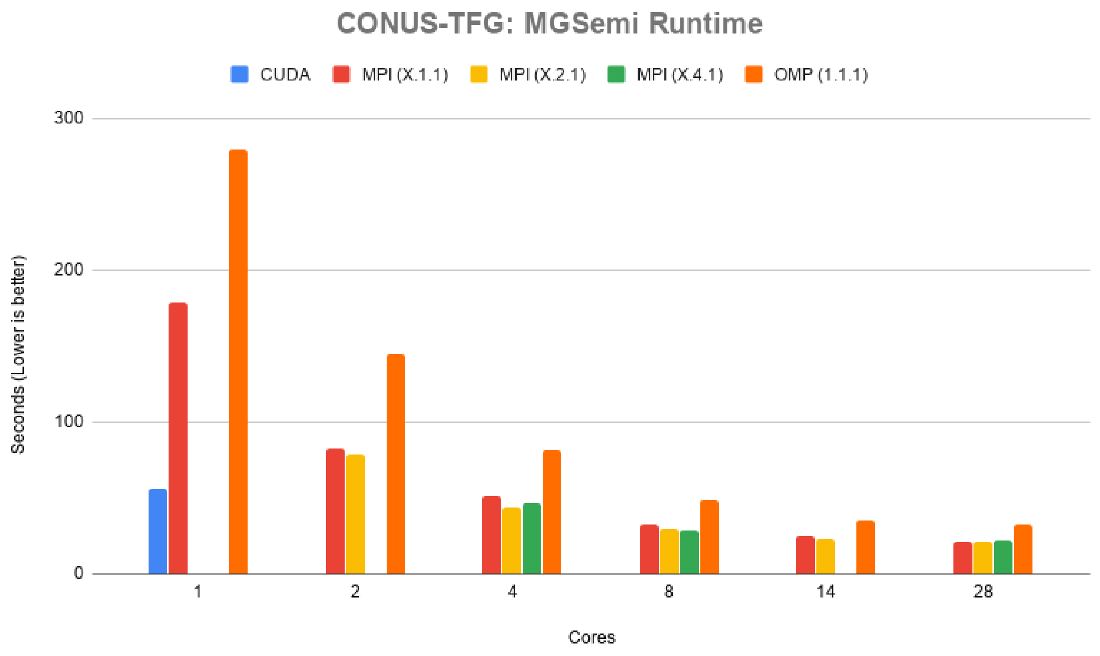


Figure 4.3: CONUS-TFG: Runtime for *MGSemi* in seconds

the 14.2.1 configuration, spending 20.75 seconds in the function. OpenMP performed slower than all MPI configurations, spending 31.78 seconds in the function on 28 cores. Both MPI and OpenMP versions saw very little performance gain once 14 cores were in use. Full timing results can be found in can be found in Appendix B Table B.3.

4.3.2 CONUS-RU

The CONUS-RU test case was run in several configurations for the MPI and OpenMP versions of ParFlow. For MPI the X in the configuration corresponds to total number of MPI processes. For OpenMP the X in the configuration corresponds to OpenMP threads per MPI process. For example an OpenMP configuration of X.4.1 on 28 cores means 4 MPI processes, each with 7 threads. Full timing results for 1 to 28 cores can

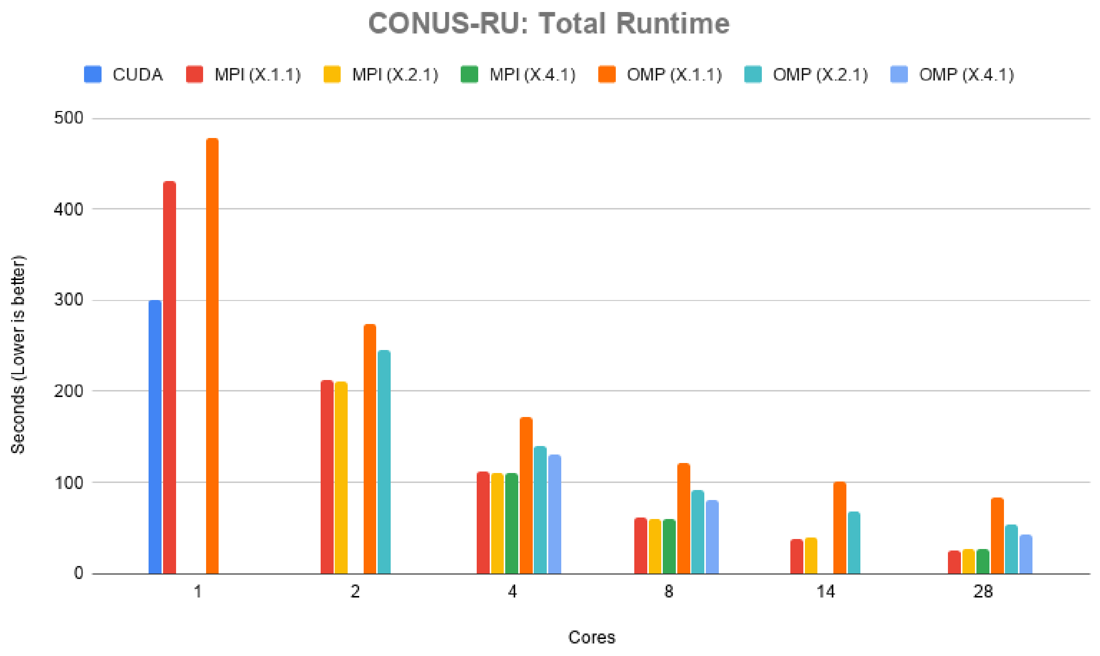


Figure 4.4: CONUS-RU: Total runtime in seconds

be found in Appendix B Table B.4.

Figure 4.4 shows the total runtime for the CONUS-RU test case in seconds. The CUDA version solved the problem domain in 300.48 seconds, and was slower after more than 2 cores were in use. The MPI version performed equally between each of its configurations with times only varying by 1 to 2 seconds, attributable to system noise and IO. The fastest time for MPI was in the 28.1.1 configuration, solving in 31.80 seconds. The OpenMP version was slower in each configuration, up to 28 cores. The fastest time for OpenMP was in the X.4.1 configuration with 28 cores in use, solving in 42.11 seconds.

Figure 4.5 shows timing results for *NL_Function_Eval* in seconds. The CUDA version was faster than the MPI version until more than one core was in use, spending in 240.40 seconds the function. The MPI version performed equally between all

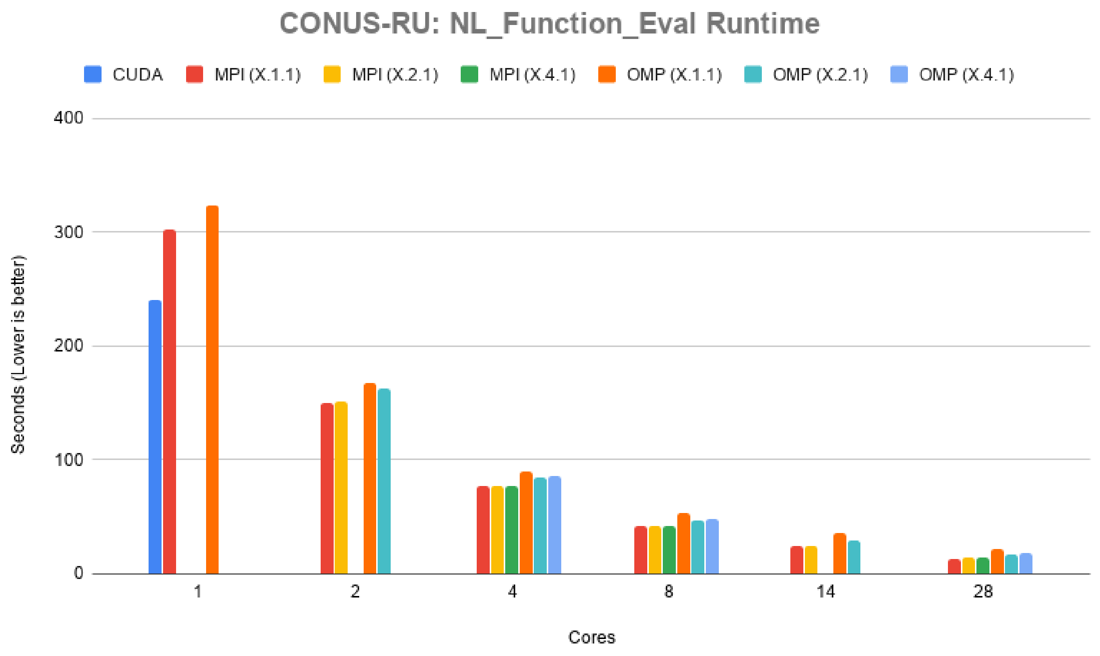


Figure 4.5: CONUS-RU: Runtime for *NL_Function_Eval* in seconds

configurations. The fastest time for MPI was in the 28.1.1 configuration, spending 17.33 seconds in the function. OpenMP was slower than all configurations up to 8 cores, at which point all configurations became competitive. The fastest time for OpenMP was 16.60 seconds in the X.2.1 configuration on 28 cores. Full timing results can be found in can be found in Appendix B Table B.5.

Figure 4.6 shows timing results for *MGSemi* in seconds. The CUDA version was faster than the MPI version until more than 5 cores were in use, spending 7.88 in the function. The MPI version performed equally between all configurations, with the fastest time of 3.45 seconds in the 28.1.1 configuration. OpenMP was slower than MPI in all configurations until 18 cores, at which point the X.2.1 configuration became competitive. The fastest time for OpenMP on 28 cores was 3.77 seconds in the X.2.1 configuration. Full timing results can be found in can be found in Appendix B

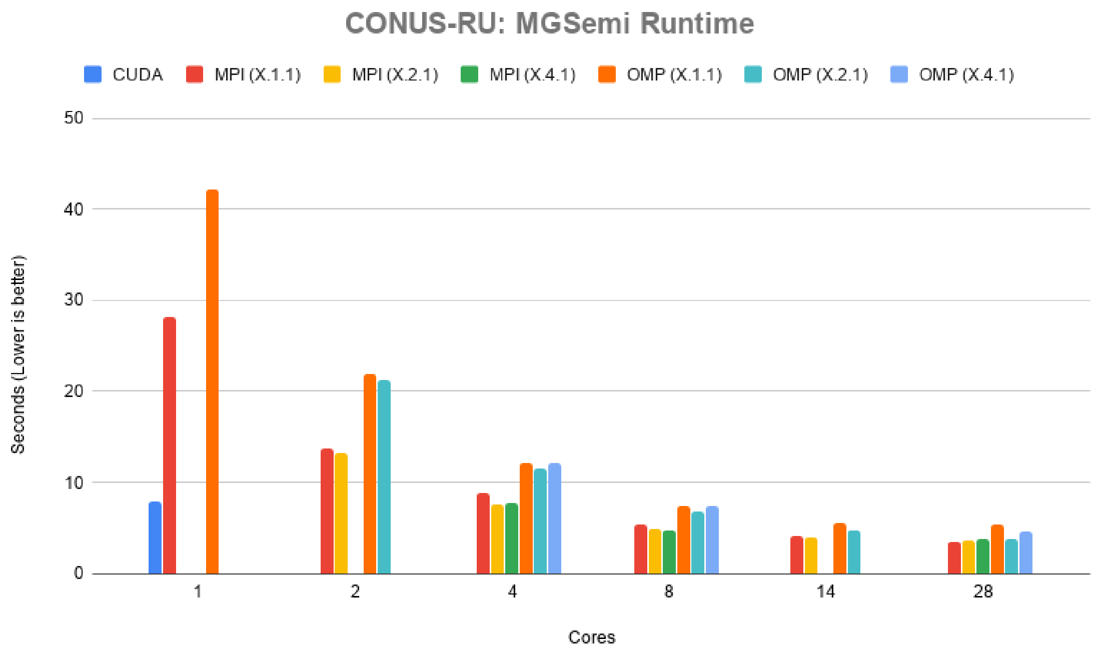


Figure 4.6: CONUS-RU: Runtime for *MGSemi* in seconds

Table B.6.

4.3.3 ClayL

The ClayL test case was run in several configurations for the MPI and OpenMP versions of ParFlow, explained in section 4.3.2. Configurations presented for the ClayL test case are grouped in terms of total core counts of 1, 2, 4, 8, 14, and 28. Full timing results for 1 to 28 cores can be found in Appendix B Table B.7.

Figure 4.7 shows the total runtime results for the ClayL benchmark. In this case, the experimental CUDA version of ParFlow performed best, solving the problem domain in 37.30 seconds. The MPI version of ParFlow performed best in the 7.4.1 configuration for 28 cores, solving the problem domain in 59.02 seconds. The experimental OpenMP version solved the problem domain in 63.59 seconds in the

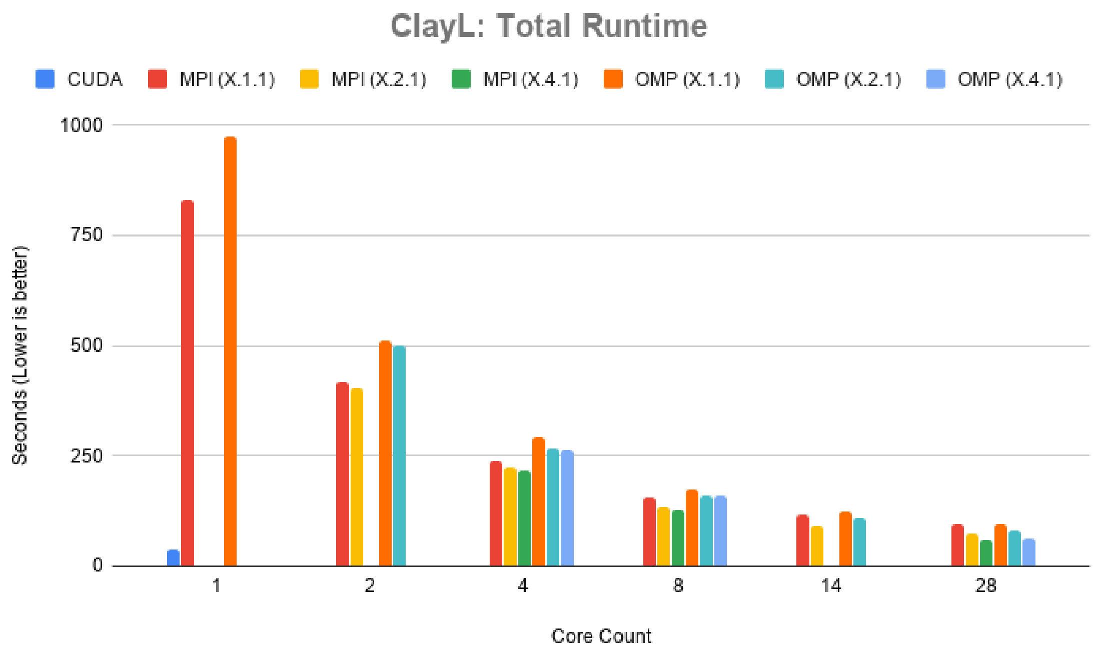


Figure 4.7: ClayL: Total runtime in seconds

X.4.1 configuration, with each MPI rank using 7 threads. This shows that the CUDA version outperforms MPI in this problem domain, and that the OpenMP version stays competitive when more than 8 cores are used. These results additionally show the impact different domain decompositions have on performance. When the domain is decomposed only along the X axis in a MPI 28.1.1 configuration, performance is 37% slower compared to a 7.4.1 configuration.

Figure 4.8 shows the runtime results for *NL_Function_Eval* in ClayL. The CUDA version spent 4.23 seconds of runtime in this function. The MPI version spent 6.66 seconds of runtime in its fastest configuration of 7.4.1. The OpenMP version spent 6.70 of runtime in its fastest configuration of X.4.1 on 28 cores. This corresponds to what was seen in the total runtime results, with CUDA being faster than MPI and OpenMP remaining competitive. Full timing results can be found in can be found in

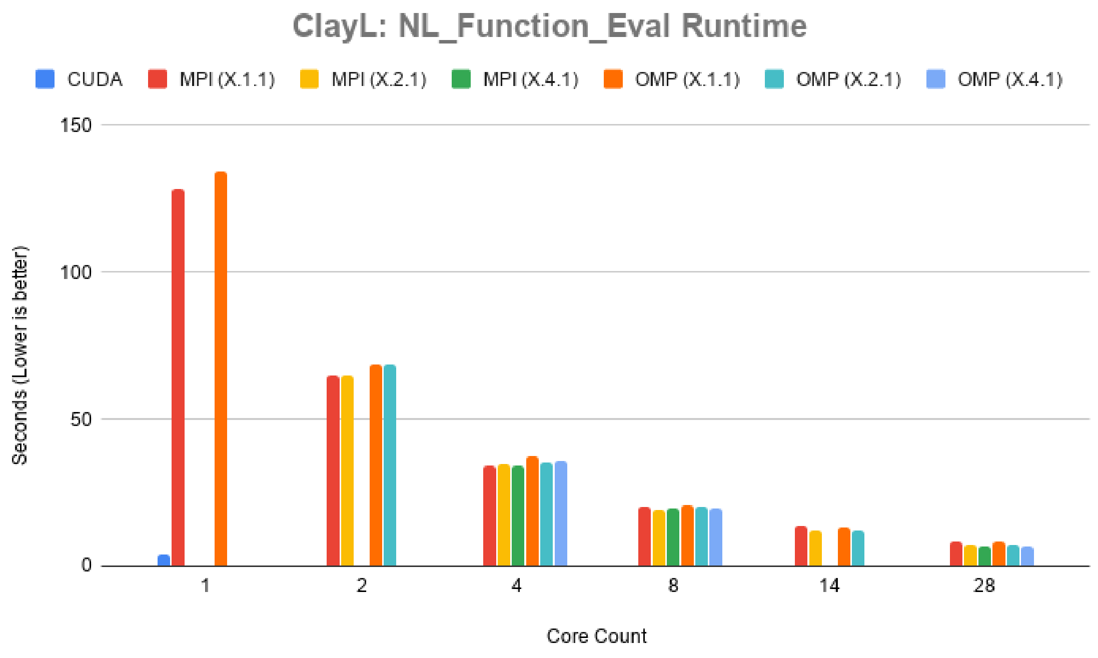


Figure 4.8: ClayL: Runtime for *NL_Function_Eval* in seconds

Appendix B Table B.8.

Figure 4.9 shows the runtime results for *MGSemi* in ClayL. The CUDA version spent 8.06 seconds of runtime in this function. The MPI version spent 20.74 seconds in its fastest configuration of 7x4x1, and OpenMP 19.17 seconds similarly. The CUDA version continues to perform better than MPI. The OpenMP version outperforms the MPI version in certain configurations. With X.1.1 configurations OpenMP performed faster than MPI more than 4 cores were in use, and faster in X.2.1 configurations when at least 14 cores were in use. The X.4.1 configuration became competitive when 20 or more cores were in use. Full timing results can be found in can be found in Appendix B Table B.9.

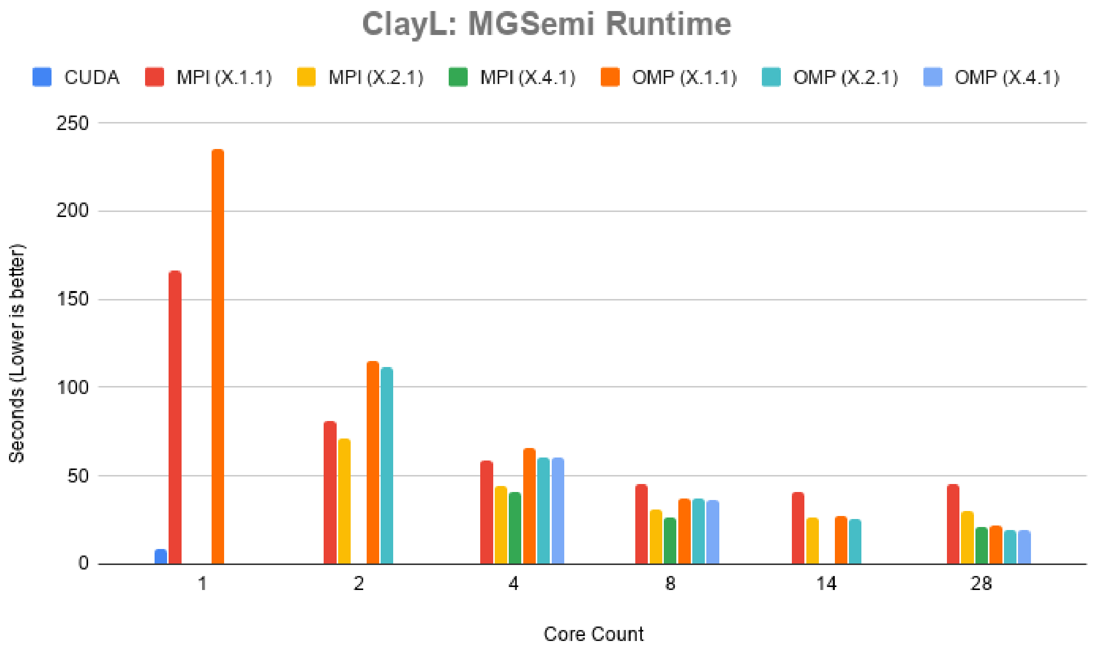


Figure 4.9: ClayL: Runtime for *MGSemi* in seconds

4.3.4 Summary

The results of this performance study indicate there is potential for improved on-node shared memory parallelism with the experimental OpenMP and CUDA versions of ParFlow. This study highlights the different performance characteristics of different problem domains. Domain decomposition affects memory layout and memory access patterns, which can have significant effect on performance results as seen in the ClayL test case. For total runtimes, we see a mix of improved performance and competitive performance for the different versions. In the ClayL test case, the CUDA version of ParFlow is faster than the MPI version in its fastest configuration on 28 cores. The OpenMP version remains competitive with the MPI version when at least 14 cores are in use. The CONUS-RU test case shows the CUDA version is slower once more than one core is in use, and that the OpenMP version is competitive up to 8 cores. In the CONUS-TFG case, both the CUDA and OpenMP versions of ParFlow are slower than the MPI version. Examining parallel regions to account for KINSOL performance shows strong potential for improved performance using CUDA and OpenMP, and indicate there is room for improvement in these experimental builds. This highlights the importance of a varied suite of test cases when performing benchmarks in a scientific application.

CHAPTER 5

RELATED WORK

Domain Specific Languages are frequently used in scientific applications. Typically, they are presented as application-independent libraries or frameworks. Different DSLs target different use cases, but have a shared interest in architecture portability and performance. DSLs are often used to capture iteration scheduling, applying parallel frameworks, and managing architecture specific details. The following sections present several related domain specific languages, grouped by categories of features they encapsulate.

The eDSL abstractions presented in this work focus on boundary condition computations. They are generalizable to other scientific applications, and are demonstrated in the ParFlow application with no external library or compiler dependencies. Particular emphasis was placed on lifting scientific computations from the existing code and inserting them into the new eDSL, with minimal rewrites. This is in contrast to most other DSL frameworks, which often require significant rewriting of computations.

5.1 Iteration Scheduling

Domain specific languages are frequently used to abstract and automatically manage iteration domain scheduling. Frameworks such as STELLA [18], Tiramisu [6], CHOMBO [28], and Intel Thread Building Blocks [26, 33] all provide interfaces for domain scientists to automatically schedule their computations.

Abstracting loop scheduling removes the need for programmers to directly manage optimizations such as loop fusion or tiling. A programmer can instead specify the domain for a given computation or set of computations and have the DSL generate applicable code. An example of this in the STELLA framework can be seen in Figure 5.1. STELLA was developed for the COSMO [2] scientific application, used in regional climate modelling. STELLA decomposes stencil computations into a set of building blocks. Each stage of the computation is defined as a C++ templated data structure. These stages are then combined in the form of a "recipe". Each *StencilStage* entry contains the appropriate stencil computation as well as its iteration domain. The iteration domain in STELLA is split into the XY axis and the Z axis, or *IJRange* and *KRange* respectively.

This separation of iteration scheduling from the computations allows for backend optimizations to be applied transparently to the programmer [4]. Loop iterations for each stencil entry can be fused, tiled, or have other optimizations applied without obfuscating key computations.

```
// Define stencil kernel stages using templated structs
template<typename TEnv> struct Lap { ... };
template<typename TEnv> struct Flx { ... };
template<typename TEnv> struct Fly { ... };
template<typename TEnv> struct Res { ... };
...
// Extracted definition of kernels and their iteration domains
StencilStage<Lap, IJRange<cIndented,-1,1,-1,1>, KRange<FullDomain,0,0>>(),
StencilStage<Flx, IJRange<cIndented,-1,0,0,0>, KRange<FullDomain,0,0>>(),
StencilStage<Fly, IJRange<cIndented,0,0,-1,0>, KRange<FullDomain,0,0>>(),
StencilStage<Res, IJRange<cComplete,0,0,0,0>, KRange<FullDomain,0,0>>()
```

Figure 5.1: Kernel stages with scheduling expressed in the STELLA DSL

5.2 GPU Parallelization

Different target architectures have different optimal scheduling patterns, especially in the context of parallel computing. For instance a GPU accelerator may see better parallel performance by ordering storage in a $k > j > i$ fashion, compared to a more typical layout of CPU iterations in a $i > j > k$ order. Storage layout directly affects memory access patterns. As processors have become faster, memory has increasingly become the bottleneck in performance. Abstracting scheduling and storage is then crucial to enabling performance portable parallel implementations.

STELLA, Tiramisu, Accelerate [12], AMReX [48], and ArrayFire [22] support GPU parallelism in their DSL implementations. STELLA offers two backends for its DSL, a CPU-based OpenMP backend and a CUDA based GPU backend. The framework allows for either backend to be selected. Storage is automatically rearranged by STELLA depending on this choice, optimizing memory access patterns for the selected architecture at compile time.

Tiramisu and AMReX also offer GPU compute support, but require more input from the programmer. Different storage choices must be explicitly stated, as opposed to STELLA's approach of specifying architecture type. Similarly iteration and loop transformations must be stated by the programmer. Accelerate and ArrayFire are both GPU-compute specific DSL extensions, with Accelerate being tailored to Haskell vector computations and ArrayFire being a general-purpose API for the C, C++, Python, and Fortran languages.

```
for (i in 0..N-2)
  for (j in 0..M-2)
    for (c in 0..3)
      bx[i][j][c] = (in[i][j][c] + in[i][j+1][c] + in[i][j+2][c]) / 3
for (i in 0..N-2)
  for (j in 0..M-2)
    for (c in 0..3)
      by[i][j][c] = (b[i][j][c] + bx[i+1][j][c] + bx[i+2][j][c]) / 3
```

Figure 5.2: 2D Blur Algorithm Pseudocode

```
// 2D Blur algorithm expressed in Tiramisu DSL
Var i(0, N-2), j(0, M-2), c(0, 3);
Computation bx(i, j, c), by(i, j, c);
bx(i, j, c) = (in(i, j, c) + in(i, j+1, c) + in(i,j+2,c))/3;
by(i, j, c) = (bx(i, j, c) + bx(i+1, j, c) + bx(i+2, j, c))/3;
```

Figure 5.3: 2D Blur Algorithm in Tiramisu

5.3 Separation of Computations

A major part of a domain specific language is the separation of computations from their implementation details. Iteration scheduling, memory and storage management, and other programming concerns must be moved away from the mathematical computations. This allows for parallelism [4] and other optimizations to be applied transparently to computational scientists.

This separation requires careful design, and often introduces a level of code complexity itself. Domain specific languages are most often presented in the form of a forward-facing API. This may result in computations needing to be reformulated to match the API.

An example of a 2D blur algorithm used in image processing can be seen in Figure 5.2. Translating this into the Tiramisu DSL requires a complete rewrite of the computations. This can be seen in Figure 5.3. Tiramisu requires computational

```
// x-flux in the STELLA DSL
template<typename TEnv>
struct Flx {
 STENCIL_STAGE(TEnv);
 STAGE_PARAMETER(FullDomain, phi);
 STAGE_PARAMETER(FullDomain, lap);
 STAGE_PARAMETER(FullDomain, flx);
 static void Do( ... ) {
  double flux = ctx[lap::Center()] - ctx[lap::At(iplus1)];
  double sign = ctx[phi::At(iplus1)] - ctx[phi::Center()];
  ctx[flx::Center()] = flux*sign > 0 ? 0 : flux;
 }
};
```

Figure 5.4: 4th Order x-Flux Smoothing in STELLA

scientists to reformulate computations into purely functional expressions. These expressions are fed into the Tiramisu polyhedral compiler, and an optimized algorithm is produced. STELLA requires similar rewriting of computations. An example of a 4th order flux smoothing in the X direction written in the STELLA DSL can be seen in Figure 5.4. STELLA discretizes stages of the computation, and computational scientists assemble the full formulation as a recipe, as described in Section 5.1.

Rewriting of computations is prohibitively expensive. As codebases become larger infrastructure becomes more complex. This causes implementation issues to rise quickly, potentially requiring additional rewriting of the program structure to fit a particular DSL. The abstractions in this work were designed with this in mind. The front-end API was designed first, specifically to minimize any rewriting of computations. Computations require little to no rewrites in the presented work, able to be lifted directly from the existing code and placed in the proposed eDSL. There is a tradeoff to this approach in that the backend becomes more complex to adapt to different hardware architectures, but benefits computational scientist productivity more directly.

5.4 Enforcing DSL Usage

DSLs provide abstractions for common computational idioms. Complex looping structures, such as Octree navigation, offer performance benefits. A DSL can encapsulate these looping structures into more easily managed interfaces, improving computational scientist productivity. Mixing non-DSL code, or ”exposed” code, such as explicit loops or conditional branching, degrades the benefits of a DSL. A DSL must encapsulate as much computation as possible, providing full coverage of the necessary sections of code. It is thus highly desirable for a DSL to be designed in such a way that it is either difficult, inconvenient, or less sensible to write exposed code intermixed with the DSL. The more computational scientists can stay within DSL abstractions, the greater impact they can produce.

Proto [34] is a meta-DSL provided as a library. The goal of Proto is to provide a way to write an eDSL utilizng a Backus-Naur style grammar. This is accomplished through C++ meta-templating features. Approaching the construction of an eDSL in this manner means that stepping outside of DSL abstractions effectively requires writing in a completely different language.

Delite [46, 9] is a framework embedded in the Scala programming language. A set of common components is provided through the framework, such as parallel loops, map functions, filtering, and reductions. Metaprogramming is leveraged to emit intermediate representation code for different languages, such as C or C++. Delite was designed with the intent of being able to write other, more specific DSL libraries, such as Forge [47] or OptiML [11]. Forge aims to provide a declarative, high-performance DSL for parallel computing. OptiML aims to provide a DSL for machine learning, and provide GPU compute portability. By leveraging Scala in this manner Delite helps to fully encapsulate computations, as well as provide a strong basis for writing other more specific DSLs.

RDL [45] is a contract-based DSL for the Ruby programming language. Contracts are implemented as a layer on top of different functions and objects. If a developer attempts to perform some computation that violates a specified contracts, compile-time checks will emit errors and prevent the program from building. This ensures that developers stay within the context of the DSL and its features, and provides a layer to ensure validity of the program.

CHAPTER 6

CONCLUSIONS

This thesis presents abstractions for boundary condition computations, demonstrated in the ParFlow application and implemented as an eDSL with no external dependencies. The abstractions are generalizable to iterative solvers that are frequently used in scientific applications, and are demonstrated in the ParFlow application. These abstractions have directly improved computational scientists productivity, enabling new boundary conditions to be successfully added. Additionally, the abstractions enable architecture portability. An experimental OpenMP version of ParFlow was implemented using these abstractions, and enable the required flexibility to unify the experimental CUDA version of ParFlow provided by the Juelich Research Center in Germany.

A performance study was conducted on the experimental builds of ParFlow. This study helps indicate what target architectures are worth exploring further. The CUDA version of ParFlow is shown to be faster than the MPI version in certain domains and configurations. The OpenMP version of ParFlow is shown to be competitive with the MPI version in certain configurations. The differing results showcase the complex nature of real world scientific applications, and highlights the need for performance metrics to be established when optimizing applications and porting to new hardware.

REFERENCES

[1] TOP500 Supercomputers List - June 2019.

[2] Cosmo: Regional climate modeling, cited August 2019.

[3] Mark Adams, Peter O Schwartz, Hans Johansen, Phillip Colella, Terry J Ligocki, Dan Martin, ND Keen, Dan Graves, D Modiano, Brian Van Straalen, et al. Chombo software package for amr applications-design document. Technical report, 2015.

[4] Todd A. Anderson, Hai Liu, Lindsey Kuper, Ehsan Totoni, Jan Vitek, and Tatiana Shpeisman. Parallelizing Julia with a Non-Invasive DSL (Artifact). *Dagstuhl Artifacts Series*, 3(2):7:1–7:2, 2017.

[5] Steven F. Ashby and Robert D. Falgout. A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations. *Nuclear Science and Engineering*, 124(1):145–159, 1996.

[6] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 193–205, Piscataway, NJ, USA, 2019. IEEE Press.

[7] I. J. Bertolacci, M. M. Strout, S. Guzik, J. Riley, and C. Olschanowsky. Identifying and scheduling loop chains using directives. In *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*, pages 57–67, Nov 2016.

[8] I. J. Bertolacci, M. M. Strout, J. Riley, S.M.J. Guzi, E. C. Davis, and C Olschanowsky. Using the loop chain abstraction to schedule across loops in existing code. In *Int. J. High Performance Computing and Networking*, volume 13, pages 86–104, 2019.

[9] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 89–100, Oct 2011.

[10] Donna A. Calhoun and Carsten Burstedde. Forestclaw: A parallel algorithm for patch-based adaptive mesh refinement on a forest of quadtrees. *CoRR*, abs/1703.03116, 2017.

[11] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. *SIGPLAN Not.*, 46(8):35–46, February 2011.

[12] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[13] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 676–687, May 2011.

[14] Eddie C. Davis, Michelle Mills Strout, and Catherine Olschanowsky. Transforming loop chains via macro dataflow graphs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 265–277, New York, NY, USA, 2018. ACM.

[15] José Fonseca. gprof2dot.

[16] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[17] Susan L Graham, Peter B Kessler, and Marshall K McKusick. Gprof: A call graph execution profiler. *ACM SIGPLAN Notices*, 39(4):49–57, 2004.

[18] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C. Schulthess. Stella: A domain-specific tool for structured grid methods in weather and climate models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 41:1–41:12, New York, NY, USA, 2015. ACM.

[19] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in GPU programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8. ACM. event-place: Newport Beach, California, USA.

[20] A. Heinecke, M. Klemm, and H. Bungartz. From GPGPU to many-core: Nvidia fermi and intel many integrated core architecture. 14(2):78–83.

[21] Alan C Hindmarsh, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005.

[22] Chris McClanahan Vishwanath Venugopalakrishnan Krunal Patel John Melonakos James Malcolm, Pavan Yalamanchili. Arrayfire: a gpu acceleration platform. 8403, 2012.

[23] Jim E. Jones and Carol S. Woodward. Newton–krylov-multigrid solvers for large-scale, highly heterogeneous, variably saturated flow problems. *Advances in Water Resources*, 24(7):763 – 774, 2001.

[24] Mandli K., Ahmadia A., Berger M., Calhoun D., George D., Hadjimichael Y., Ketcheson D., Lemoine G., and LeVeque R. Clawpack: building an open source ecosystem for solving hyperbolic pdes. *PeerJ Computer Science*, 2016.

[25] Ramgopal Kashyap and Pratima Gautam. Fast level set method for segmentation of medical images. In *Proceedings of the International Conference on Informatics and Analytics*, ICIA-16, pages 20:1–20:7, New York, NY, USA, 2016. ACM.

[26] W. Kim and M. Voss. Multicore desktop programming with intel threading building blocks. *IEEE Software*, 28(1):23–31, Jan 2011.

[27] Stefan J. Kollet and Reed M. Maxwell. Integrated surface–groundwater flow modeling: A free-surface overland flow boundary condition in a parallel groundwater flow model. *Advances in Water Resources*, 29(7):945 – 958, 2006.

[28] D. T. Graves J.N. Johnson N.D. Keen T. J. Ligocki. D. F. Martin. P.W. McCorquodale D. Modiano. P.O. Schwartz T.D. Sternberg M. Adams, P. Colella and B. Van Straalen. Chombo software package for amr applications design document. Technical report, Lawrence Berkely National Laboratory, 2019.

[29] R. M. Maxwell, L. E. Condon, and S. J. Kollet. A high-resolution simulation of groundwater and surface water over most of the continental us with the integrated hydrologic model parflow v3. *Geoscientific Model Development*, 8(3):923–937, 2015.

[30] Reed M. Maxwell. A terrain-following grid transform and preconditioner for parallel, large-scale, integrated hydrologic modeling. *Advances in Water Resources*, 53:109 – 117, 2013.

[31] Reed M Maxwell. A terrain-following grid transform and preconditioner for parallel, large-scale, integrated hydrologic modeling. *Advances in Water Resources*, 53:109–117, 2013.

[32] Zdzisław Meglicki, Stephen K. Gray, and Boyana Norris. Multigrid fdtd with chombo. *Computer Physics Communications*, 176(2):109 – 120, 2007.

[33] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 224–235, Washington, DC, USA, 2011. IEEE Computer Society.

[34] Eric Niebler. Proto: A compiler construction toolkit for dsels. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, LCSD '07, pages 42–51, New York, NY, USA, 2007. ACM.

[35] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 2015.

[36] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 427–436, Feb 2009.

[37] Shah M. Faizur Rahman, Qing Yi, and Apan Qasem. Understanding stencil code performance on multicore architectures. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, New York, NY, USA, 2011. Association for Computing Machinery.

[38] et al Reed M. Maxwell. Hydroframe: A national community hydrologic modeling framework for scientific discovery, cited August 2019.

[39] Lakshminarayanan Renganarayanan, Daegon Kim, Michelle Mills Strout, and Sanjay Rajopadhye. Parameterized loop tiling. *ACM Trans. Program. Lang. Syst.*, 34(1):3:1–3:41, May 2012.

[40] Lorenzo Adolph Richards. Capillary conduction of liquids through porous mediums. *Physics*, 1(5):318–333, 1931.

[41] Gerhard Wellein Rolf Rabenseifner. Communication and optimization aspects of parallel programming models on hybrid architectures.

[42] Nikolay Sakharnykh. Everything you need to know about unified memory, 2018.

[43] J. Sartori and R. Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. 15(2):279–290.

[44] Kyle Shannon. bsurc/r2-doi: 1.0.17, November 2018.

[45] T. Stephen Strickland, Brianna M. Ren, and Jeffrey S. Foster. Contracts for domain-specific languages in ruby. *SIGPLAN Not.*, 50(2):23–34, October 2014.

[46] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, April 2014.

[47] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: Generating a high performance dsl implementation from a declarative specification. *SIGPLAN Not.*, 49(3):145–154, October 2013.

[48] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, Max Katz, Andrew Myers, Tan Nguyen, Andrew Nonaka, Michele Rosso, Samuel Williams, and Michael Zingale. Amrex: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software*, 4(37):1370, 5 2019.

APPENDIX A

REPRODUCIBILITY

This appendix covers steps to reproduce the results in this work. See Appendix B for the full timing results of the performance study.

ParFlow was compiled using the basic instructions found in the user's manual, with the following CMAKE command

```
cd build
cmake ../$SRC -DCMAKE_BUILD_TYPE=Release -DPARFLOW_AMPS_LAYERS=mpi1 -
    DCMAKE_INSTALL_PREFIX=${PARFLOW_DIR} -DPARFLOW_HAVE_CLM=ON -
    DPARFLOW_ENABLE_TIMING=ON
```

For the CUDA version, the additional flag was enabled:

```
-DPARFLOW_ENABLE_CUDA=true
```

And similarly for the OpenMP version the additional flag was enabled:

```
-DPARFLOW_ENABLE_OMP=true
```

Libraries used were: GCC 7.2.0, MPICH 3.2.1, CMake 3.12, TCL 8.5, and CUDA 10.1. When compiling on R2, it may be necessary to explicitly tell CMake to use mpicc and mpicxx as the C and CXX compilers in order to avoid a linking error. This is an environmental issue, and can be resolved by exporting the CC and CXX variables to point directly at mpicc and mpicxx.

The MPI version can be found at the following github repo and commit hash `https://github.com/hydroframe/ParFlow_PerfTeam/tree/pf_newbc`

Hash: 983aadd45b83cce9272b971a7bbb960a7b602b92

The OpenMP version can be found at the following github repo and commit hash `https://github.com/hydroframe/ParFlow_PerfTeam/tree/pf_omp_newbc`

Hash: 237198f999f190a41cc822b0166bdd973d8f02a2

The CUDA version can be found at the following github repo and commit hash `https://github.com/hokkanen/parflow/tree/CUDA`

Hash: a7d788deaefa285c31c9ab90accbbe6dca9b412f

APPENDIX B

DATA

Appendix B contains full timing results of the performance study.

Table B.1: Total Runtime in seconds for CONUS-TFG

CONUS-TFG: Total Runtime in Seconds					
		MPI			
Cores	CUDA	(X.1.1)	(X.2.1)	(X.4.1)	OMP (1.1.1)
1	95.1409	590.2626			713.5529
2		285.476	283.4181		423.6981
3		214.0199			320.5828
4		157.0586	148.3584	153.8221	284.6438
5		137.4555			234.347
6		114.5306	108.1736		213.5326
7		104.1622			198.7523
8		91.488	87.3281	86.6709	186.0985
9		86.4653			179.3417
10		77.7692	74.1324		171.2415
11		74.2479			166.0651
12		68.2774	65.0843	64.5945	163.8548
13		66.0936			157.6794
14		60.9969	58.5117		154.3853
15		59.8671			152.0902
16		54.4879	54.1956	54.0323	149.0282
17		54.342			149.1124
18		51.1796	50.5412		145.1628
19		51.9119			144.236
20		48.7314	47.982	47.9879	145.9078
21		48.7361			145.6109
22		47.3419	45.9227		144.2522
23		47.1239			142.7701
24		45.0306	44.6332	44.4736	144.0038
25		45.2892			141.6901
26		43.5721	42.5987		144.1152
27		43.9748			142.8256
28		43.3487	41.8776	42.3887	146.0815

Table B.2: CONUS-TFG: Runtime in seconds for NL_Function_Eval

CONUS-TFG: NL_Function_Eval Runtime in Seconds					
		MPI			
Cores	CUDA	(X.1.1)	(X.2.1)	(X.4.1)	OMP (1.1.1)
1	2.1444	114.7513			97.3628
2		57.0069	57.1074		50.4961
3		39.5161			34.9301
4		29.6847	30.2003	30.3302	28.7583
5		24.9522			22.4184
6		21.1407	21.1997		19.5306
7		18.6752			17.3392
8		16.4738	16.3324	16.607	15.4116
9		14.8655			14.1376
10		13.571	13.6148		12.9955
11		12.5142			11.9384
12		11.4566	11.325	11.4796	11.2612
13		10.5095			10.4997
14		9.7908	9.8397		9.8731
15		9.3067			9.3661
16		8.7043	8.7116	8.8203	8.8203
17		8.3918			8.5589
18		7.8096	7.8118		8.1263
19		7.5098			7.8269
20		7.1457	7.1234	7.1878	7.6355
21		6.8747			7.3943
22		6.6184	6.5293		7.1815
23		6.2681			6.8459
24		6.0435	6.0936	6.1128	6.8288
25		5.8615			6.5283
26		5.6091	5.6474		6.5097
27		5.4945			6.3326
28		5.3119	5.3036	5.3556	6.5148

Table B.3: CONUS-TFG: Runtime in seconds for MGSemi

CONUS-TFG: MGSemi Runtime in Seconds					
		MPI			
Cores	CUDA	(X.1.1)	(X.2.1)	(X.4.1)	OMP (1.1.1)
1	55.5078	178.3837			279.4478
2		81.7842	78.6208		144.3533
3		68.716			101.366
4		50.9912	43.2394	46.2671	81.4114
5		46.6414			67.3962
6		38.3691	34.6733		58.0295
7		36.6272			52.589
8		31.9408	29.6093	28.2555	47.7389
9		31.6934			44.9323
10		28.638	26.3018		41.5543
11		28.1891			39.4681
12		26.2193	24.6176	24.5193	38.0664
13		26.1297			36.4633
14		24.5824	23.0906		35.4101
15		24.5963			34.2352
16		22.3855	22.5857	22.6583	33.6273
17		22.7946			33.0489
18		21.8911	21.793		32.411
19		22.5442			31.9393
20		21.3651	21.4286	21.7411	32.0211
21		21.7346			31.8331
22		21.6715	21.1169		32.2058
23		21.701			31.8775
24		20.7951	21.0244	21.5078	31.4658
25		21.2023			31.2614
26		20.7218	20.6498		31.2784
27		20.681			32.0153
28		20.9361	20.7523	21.5886	31.7805

Table B.4: Total Runtime for CONUS-RU in seconds

CONUS-RU: Total Runtime in Seconds							
		MPI			OpenMP		
Cores	CUDA	(X.1.1)	(X.2.1)	(X.4.1)	(X.1.1)	(X.2.1)	(X.4.1)
1	300.4864	431.1694			477.4971		
2		213.1075	210.9833		273.3172	245.0813	
3		153.5537			210.5096		
4		111.5284	109.9218	110.0043	171.7766	140.656	130.4315
5		95.0964			157.5567		
6		78.6069	77.4722		138.4557	109.4031	
7		70.8204			132.6268		
8		61.1082	60.3035	60.5381	121.5766	91.4095	80.1699
9		57.2208			116.2587		
10		50.1687	48.94		110.8644	80.2777	
11		47.8942			106.6913		
12		43.4997	43.4556	43.4274	100.605	72.3488	61.6485
13		41.1259			101.6875		
14		38.6181	38.7052		101.396	67.247	
15		36.3584			99.1602		
16		34.1278	34.503	34.6393	94.6351	62.8248	53.5078
17		33.2731			92.1198		
18		31.5719	31.6636		90.4766	60.6641	
19		30.6466			92.6945		
20		28.9557	29.5561	28.887	92.3335	56.7026	46.5754
21		28.5131			88.5314		
22		27.3472	27.8865		87.7378	54.969	
23		26.4875			85.5517		
24		25.8812	26.6156	26.0378	84.9141	54.5713	42.9873
25		25.302			84.1788		
26		24.624	25.4004		84.1767	53.3349	
27		24.3769			83.7941		
28		24.9497	27.0428	26.1139	84.0194	53.5785	42.1132

Table B.5: Runtime for NL_Function_Eval in CONUS-RU (in seconds)

CONUS-RU: NL_Function_Eval Runtime in Seconds							
		MPI			OpenMP		
Cores	CUDA	(X.1.1)	(X.2.1)	(X.4.1)	(X.1.1)	(X.2.1)	(X.4.1)
1	240.4099	302.5261			323.2018		
2		150.3987	150.4901		167.1557	162.1828	
3		104.1439			117.0322		
4		77.2322	77.4269	77.0043	89.8518	83.8193	85.3287
5		64.9654			77.4109		
6		53.8057	53.6859		64.5621	59.9235	
7		47.4488			58.1626		
8		41.898	41.2877	41.4103	52.8772	46.6726	47.9993
9		37.8984			48.8969		
10		33.7321	33.7181		43.8333	38.627	
11		31.1617			41.6734		
12		28.6679	28.3173	28.4395	37.937	32.7262	33.9833
13		26.3162			36.9711		
14		24.533	24.619		34.9573	28.471	
15		22.9915			33.4715		
16		21.6495	21.7547	21.908	30.9368	25.3957	26.8202
17		20.5604			30.561		
18		19.3892	19.6974		28.4455	23.253	
19		18.488			28.7874		
20		17.6434	17.9502	17.8955	26.5413	21.0852	22.6573
21		16.8863			25.8796		
22		16.1764	16.5171		24.9834	19.4564	
23		15.5512			24.2727		
24		15.136	15.4975	15.317	23.8621	18.5173	19.6114
25		14.4663			23.1727		
26		14.0604	14.3383		22.8874	17.3456	
27		13.5634			22.4859		
28		13.2584	13.5891	13.7658	22.0074	16.6084	17.9609

Table B.6: Runtime for MGSemi in CONUS-RU (in seconds)

CONUS-RU: MGSemi Runtime in Seconds							
		MPI			**OpenMP**		
Cores	CUDA	(X.1.1)	(X.2.1)	(X.4.1)	(X.1.1)	(X.2.1)	(X.4.1)
1	7.8803	28.1434			42.1077		
2		13.7523	13.1941		21.8961	21.1881	
3		11.6294			16.0321		
4		8.8021	7.5292	7.7099	12.0788	11.5428	12.1255
5		8.5262			10.7607		
6		6.4584	5.8002		9.1653	8.5101	
7		6.1075			8.5638		
8		5.3149	4.9425	4.6555	7.3846	6.8339	7.433
9		5.2372			6.9477		
10		4.7406	4.4213		6.6886	5.8802	
11		4.691			6.1321		
12		4.3843	4.212	4.0017	6.1599	5.2026	5.7377
13		4.5486			5.6864		
14		4.128	4.0093		5.5038	4.6545	
15		4.1835			5.4744		
16		3.8483	3.868	3.7993	5.5738	4.3255	5.287
17		3.8629			5.2486		
18		3.7047	3.8959		5.4743	4.1856	
19		3.6512			5.1438		
20		3.557	3.7667	3.7034	5.2924	3.8828	4.5292
21		3.5771			5.3019		
22		3.4745	3.8069		5.2502	3.7004	
23		3.5649			5.3239		
24		3.5035	3.7238	3.7205	5.3863	3.6919	4.1811
25		3.3685			5.2417		
26		3.4424	3.7663		5.2187	3.6642	
27		3.3801			5.3715		
28		3.4579	3.7058	3.8653	5.3644	3.7715	4.5466

Table B.7: Total Runtime for ClayL in seconds

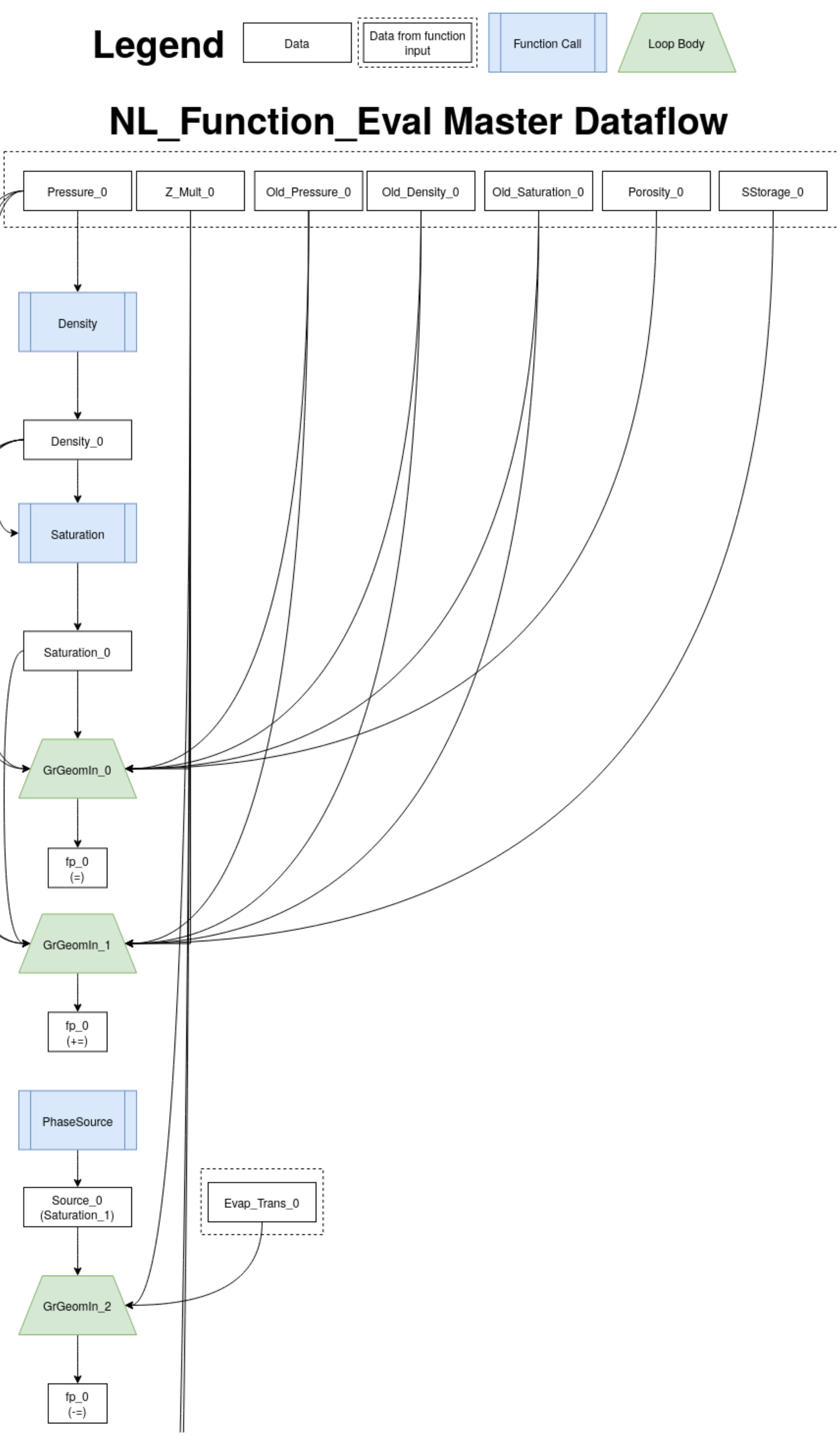
ClayL: Total Runtime in Seconds							
		MPI			**OpenMP**		
Cores	CUDA	(X.1.1)	(X.2.1)	(X.4.1)	(X.1.1)	(X.2.1)	(X.4.1)
1	37.3058	828.4127			973.4922		
2		416.6005	403.6559		509.7668	500.6202	
3		320.1411			370.3056		
4		240.0473	223.4608	218.5148	291.2624	268.3682	263.7105
5		219.6244			250.581		
6		187.8712	164.894		213.2963	207.2249	
7		174.9298			195.5182		
8		156.0162	132.5499	127.092	174.8489	160.6282	160.3136
9		152.1796			166.1589		
10		134.2828	112.9664		151.8845	143.1992	
11		135.907			145.3784		
12		123.2309	100.8743	93.4633	135.0958	119.3487	114.1549
13		122.0381			130.6556		
14		115.9421	92.6176		124.561	108.2036	
15		112.0145			121.8913		
16		108.5176	85.3744	76.8138	116.0221	108.4994	93.4524
17		105.1744			112.4914		
18		106.6106	81.8341		110.4656	98.181	
19		103.1385			107.4411		
20		98.3133	76.9272	66.6717	104.2156	95.3316	78.717
21		99.6445			105.3035		
22		97.8716	75.7887		102.9644	81.5117	
23		97.5337			102.0074		
24		98.3294	73.3117	61.317	97.4313	85.3011	72.8016
25		97.8081			97.5747		
26		98.5122	70.6949		94.5854	81.3627	
27		93.4555			93.5686		
28		95.3502	72.0447	59.0242	94.6501	78.82	63.5997

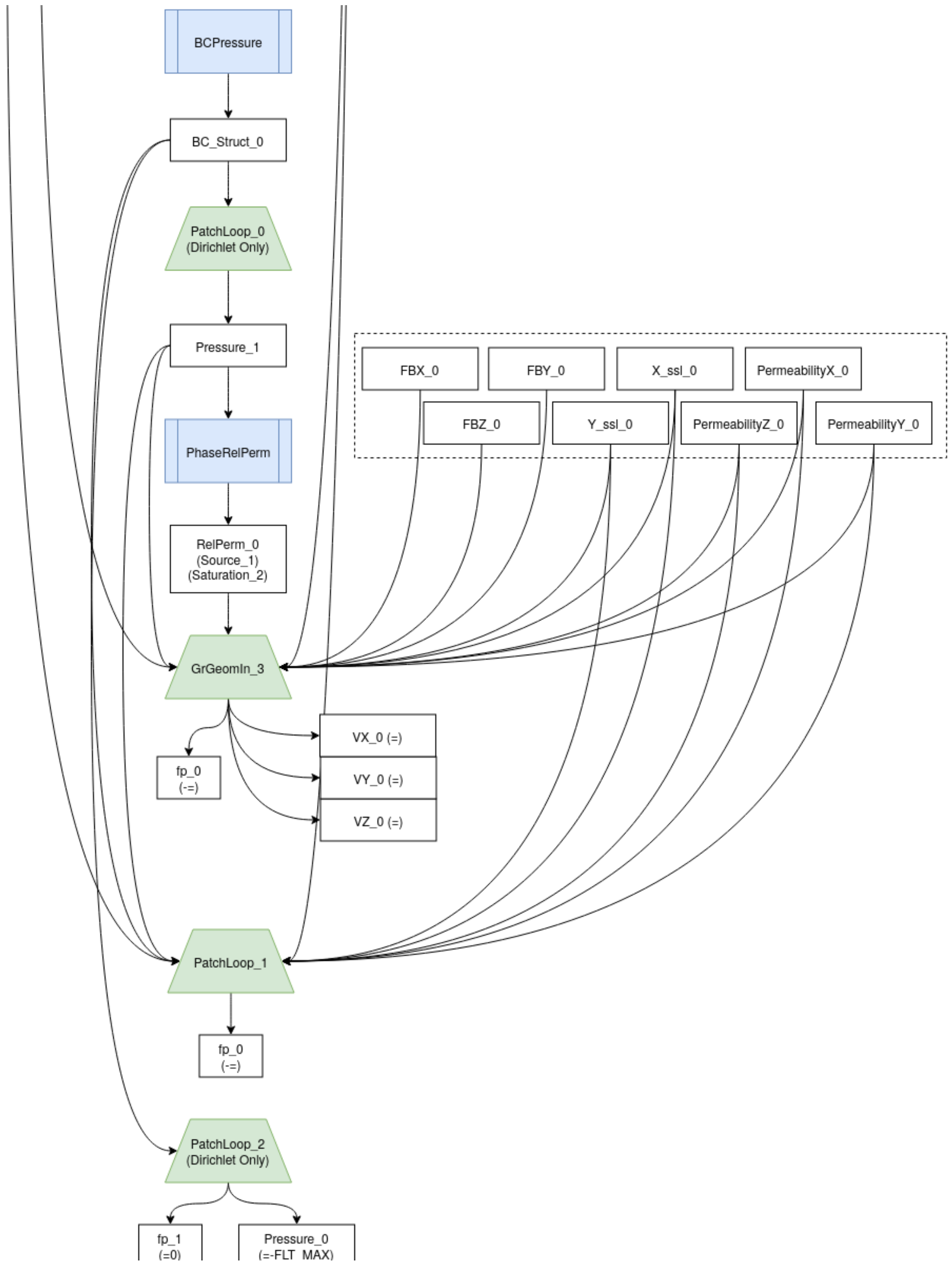
Table B.8: Runtime for NL_Function_Eval in ClayL (in seconds)

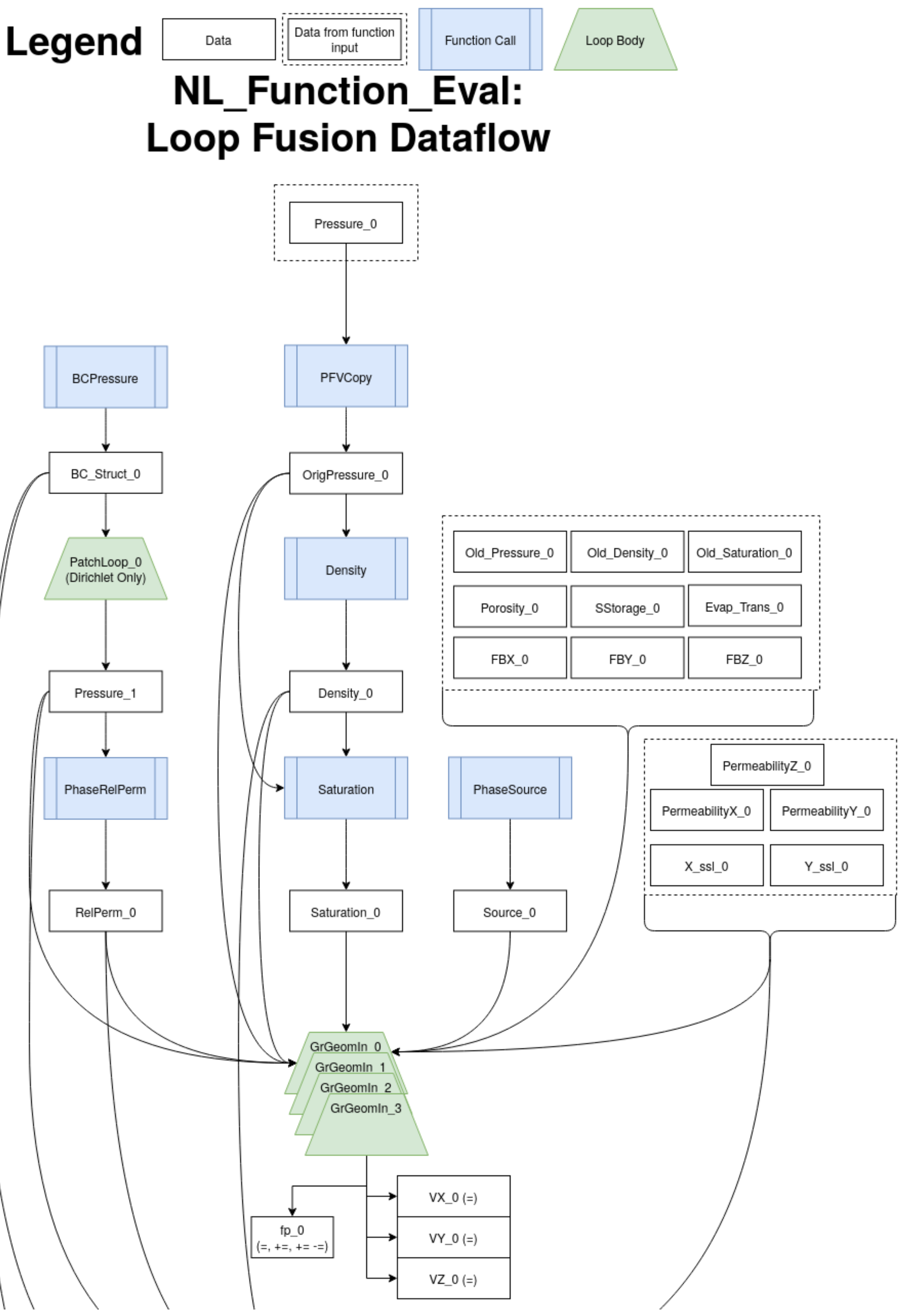
ClayL: NL_Function_Eval Runtime in Seconds							
		MPI			**OpenMP**		
Cores	CUDA	(X.1.1)	(X.2.1)	(X.4.1)	(X.1.1)	(X.2.1)	(X.4.1)
1	4.2369	128.2303			134.3273		
2		64.6123	64.5435		68.5107	68.652	
3		45.995			47.7543		
4		34.0452	34.7583	34.2437	37.5801	35.2405	35.543
5		29.3632			31.0214		
6		25.1588	24.673		25.9052	25.3705	
7		22.6781			23.2174		
8		20.3561	19.2069	19.4977	20.5535	20.007	19.5322
9		18.8421			19.2371		
10		16.623	15.884		17.2869	16.6132	
11		16.1378			16.3078		
12		14.9158	13.7248	13.5456	14.7161	14.1056	13.6969
13		13.7894			13.844		
14		13.5154	12.1569		12.9981	12.2364	
15		12.305			12.4238		
16		12.1064	10.7807	10.6727	11.6691	10.9379	10.681
17		11.0615			11.3024		
18		11.0415	9.987		11.0106	9.9144	
19		10.8817			10.2544		
20		9.7149	8.8165	8.5916	9.8387	9.1441	8.7446
21		9.7758			9.7432		
22		9.578	8.5538		9.2654	8.5761	
23		9.4235			9.1382		
24		9.4208	7.958	7.3975	8.7575	7.8241	7.5242
25		8.45			8.5231		
26		8.4589	7.2833		8.2873	7.3858	
27		8.2795			8.0791		
28		8.3447	7.2808	6.6667	8.1291	7.1221	6.707

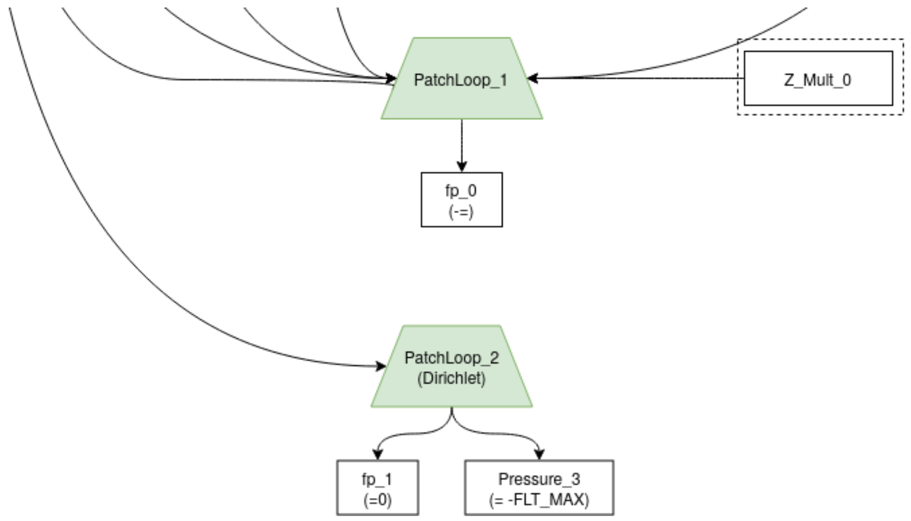
Table B.9: Runtime for MGSemi in ClayL (in seconds)

ClayL: MGSemi Runtime in Seconds							
		MPI			**OpenMP**		
Cores	CUDA	(X.1.1)	(X.2.1)	(X.4.1)	(X.1.1)	(X.2.1)	(X.4.1)
1	8.0668	166.3876			235.3667		
2		81.1277	71.0781		115.0029	111.2949	
3		78.3513			86.6416		
4		58.6169	44.5635	41.1384	65.755	60.4402	60.7304
5		61.4124			56.67		
6		51.6112	35.6064		46.5615	44.0998	
7		51.0247			43.4315		
8		45.1991	31.1486	26.0713	37.1057	37.4701	36.0915
9		48.685			35.9876		
10		43.0082	28.1439		32.1952	31.4959	
11		45.9763			31.5688		
12		40.5915	27.2615	22.0781	28.8861	27.6507	28.5602
13		43.7246			28.281		
14		40.5561	26.8293		26.9116	25.2818	
15		41.5754			26.3364		
16		39.6191	26.047	20.1237	25.3203	22.7993	24.0157
17		40.6912			24.0751		
18		42.2592	26.9332		24.2176	21.5508	
19		41.766			23.8234		
20		40.9159	27.1614	19.531	23.0907	20.1996	21.0494
21		41.8416			23.1748		
22		41.4027	27.6521		23.6919	19.8962	
23		42.461			23.7226		
24		42.8726	27.8447	19.9555	22.1985	19.4967	19.393
25		45.2082			22.4936		
26		46.2817	28.0068		21.7466	19.0071	
27		43.0156			21.5743		
28		45.1973	29.6334	20.7464	22.2664	18.9432	19.1745


Legend
Data
Data from function input
Function Call
Loop Body
NL_Function_Eval Master Dataflow
Pressure_0
Z_Mult_0
Old_Pressure_0
Old_Density_0
Old_Saturation_0
Porosity_0
SStorage_0
Density
Density_0
Saturation
Saturation_0
GrGeomIn_0
fp_0 (=)
GrGeomIn_1
fp_0 (+=)
PhaseSource
Source_0 (Saturation_1)
Evap_Trans_0
GrGeomIn_2
fp_0 (-=)


BCPressure
BC_Struct_0
PatchLoop_0 (Dirichlet Only)
Pressure_1
PhaseRelPerm
RelPerm_0 (Source_1) (Saturation_2)
GrGeomIn_3
FBX_0
FBY_0
X_ssl_0
PermeabilityX_0
FBZ_0
Y_ssl_0
PermeabilityZ_0
PermeabilityY_0
fp_0 (-=)
VX_0 (=)
VY_0 (=)
VZ_0 (=)
PatchLoop_1
fp_0 (-=)
PatchLoop_2 (Dirichlet Only)
fp_1 (=0)
Pressure_0 (=-FLT_MAX)


Legend
Data
Data from function input
Function Call
Loop Body
NL_Function_Eval:
Loop Fusion Dataflow
Pressure_0
BCPressure
PFVCopy
BC_Struct_0
OrigPressure_0
PatchLoop_0
(Dirichlet Only)
Density
Old_Pressure_0
Old_Density_0
Old_Saturation_0
Porosity_0
SStorage_0
Evap_Trans_0
FBX_0
FBY_0
FBZ_0
Pressure_1
Density_0
PermeabilityZ_0
PhaseRelPerm
Saturation
PhaseSource
PermeabilityX_0
PermeabilityY_0
X_ssl_0
Y_ssl_0
RelPerm_0
Saturation_0
Source_0
GrGeomIn_0
GrGeomIn_1
GrGeomIn_2
GrGeomIn_3
VX_0 (=)
VY_0 (=)
VZ_0 (=)
fp_0
(=, +=, += -=)


PatchLoop_1
Z_Mult_0
fp_0
(-=)
PatchLoop_2
(Dirichlet)
fp_1
(=0)
Pressure_3
(= -FLT_MAX)