HYBRID SCHEDULING FOR GRAPH-BASED ALGORITHM DECOMPOSITION IN HIGH-PERFORMANCE COMPUTING ENVIRONMENTS

by

Braden Devin Robison

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computational Engineering and Science

School of Computing

The University of Utah

May 2014

UMI Number: 1553595

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI 1553595

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Copyright © Braden Devin Robison 2014

All Rights Reserved

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Braden Devin Robison**

has been approved by the following supervisory committee members:

Name	Role	Date Approved
James Sutherland	, Chair	**4/2/2013**
Mary Hall	, Member	**3/25/2013**
Adam Bargteil	, Member	**3/20/2013**

and by **Martin Berzins**, Chair/Dean of

the Department/College/School of **Computer Science**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

At the beginning of the 21st century, it became apparent that the performance gains associated with continual die shrinks and the resulting increases in core central processing unit (CPU) speeds were beginning to flatten. This realization has gradually shifted the focus of CPU design away from single core speed increases and toward the idea of obtaining performance through increased concurrency. The resulting design paradigm has given us multi- and many-core CPUs, vector processing units, and more recently, programmable, massively parallel hardware coprocessors, such as graphics processing units from nVidia and Advanced Micro Devices, along with more recent general purpose devices such as Intel's "Knights Corner." One of the most significant resulting challenges in high-performance computing is to provide a framework in which the software development process is platform agnostic to its end users, while at the same time being capable of scaling efficiently on diverse hardware configurations. This thesis will present an improved approach for the analysis and scheduling of computational tasks within a heterogeneous hardware environment, while removing implementation details from end users. This will be presented within the context of the "Expression" framework, a component within a computational fluid dynamics solver, known as "Wasatch," developed at the University of Utah.

CONTENTS

ABSTRACT . . . . **iii**

LIST OF FIGURES . . . . **vi**

ACKNOWLEDGMENTS . . . . **viii**

CHAPTERS

1. INTRODUCTION **1**

1.1 Overview 1
1.1.1 Advantages 1
1.1.2 Challenges 2
1.2 Distributed and Heterogeneous Hardware 3
1.2.1 Grid-Based Computing 3
1.2.2 Transition from Single to Multicore Architectures 4
1.2.3 Coprocessors and Massively Parallel Hardware 4
1.3 Approaches to Computational Software 5
1.3.1 Mastery of Everything 5
1.3.2 Our Goal - Separating Form and Functionality 6
1.4 The Wasatch Framework 6

2. GRAPH THEORY AND APPLICATIONS **7**

2.1 Overview 7
2.1.1 Terminology 7
2.1.2 Algorithms 8
2.2 DAGs as Computational Models 9
2.2.1 Motivation 9
2.2.2 Related Work 9
2.2.3 The Expressions Context 11
2.2.4 The Dependency Graph 12
2.2.5 The Task Graph 12
2.3 Extracting Information 12
2.3.1 Task Granularity 12
2.3.2 Graph Introspection 12
2.3.3 Bounds on Execution Time 13
2.3.4 Parallelization 13

3. FRAMEWORK EXTENSIONS **15**

3.1 The Expressions Library 15
3.1.1 Overview 15
3.1.2 Concepts and Terminology 15

3.1.3 Implementing an Expression . . . 17
3.1.4 Constructing the Task Graph . . . 19
3.1.5 Design Goals . . . 20
3.1.6 A Flexible Scheduling Model . . . 21
3.1.7 The Priority Task Scheduler . . . 22
3.1.8 Graph-Based Execution . . . 22
3.1.9 Task Dispatch Model . . . 23
3.1.10 Improving Resource Utilization . . . 25
3.1.11 Memory Resources . . . 25
3.1.12 Threading Resources . . . 29
3.1.13 The "Hybrid" Task Scheduler . . . 31
3.1.14 Device Assignment . . . 33
3.1.15 Path Coalescing "Clustering" . . . 35
3.1.16 Reducing Edge Latency . . . 35
3.2 The Spatial Fields Library . . . 37
3.2.1 Overview . . . 37
3.2.2 Concepts and Terminology . . . 37
3.2.3 Operator Selection . . . 38
3.2.4 External Consumers . . . 38

4. RESULTS AND EVALUATION . . . 40

4.1 Test Cases . . . 40
4.1.1 Scheduler Performance - Task Threaded Operator Interaction . . . 40
4.1.2 Scheduler Performance - Task Threaded MPI-Process Interaction . . . 43
4.1.3 Scheduler Performance - Task Threaded MPI-Scaling: Multinode . . . 49
4.1.4 Hybrid Scheduler Feasibility - Stencil 2 Performance . . . 49
4.2 Conclusion . . . 54
4.2.1 Work Summary . . . 54
4.2.2 Future Work . . . 55
4.2.3 Final Thoughts . . . 56

APPENDIX: SCHEDULER CODE . . . 57

REFERENCES . . . 71

LIST OF FIGURES

3.1 Framework component diagram. 16
3.2 Example of a basic expression. 16
3.3 Constructing the task graph from individual expressions. 18
3.4 Advertise dependents implementation. 19
3.5 Bind operators implementation 19
3.6 Bind fields implementation. 19
3.7 Evaluate method implementation. 19
3.8 Visualization of a potential task graph for the heat equation. 21
3.9 Simplified communication memory model, multicore CPU. 23
3.10 Priority Task Dispatch Model. 24
3.11 Dependency graph illustrating the deallocation of a fully consumed expression. 26
3.12 Initial field manager implementation. 27
3.13 Updated field manager implementation. 28
3.14 Idealized multilevel parallelism. 30
3.15 Example task graph analysis to determine thread allocation. Note: execution times are all normalized to 1. 31
3.16 Expanded communication memory model. 32
3.17 Scheduling with mixed hardware and edge transfer cost. 34
3.18 Multiconsumer field problem. 35
3.19 Illustration of consumer prefetching. 36
3.20 Illustration of consumer prefetching. 37
4.1 Example scalability graph, eight equation, no source coupling 42
4.2 Example scalability graph, eight equation, source coupling 44
4.3 Scalability test, ember 2012, plotted as a function of operator threads - 256x128x128, 16 variables 45
4.4 Scalability test, ember 2012, plotted as a function of expression threads - 256x128x128, 16 variables 46
4.5 Single node isolated scaling for process, task, and operator parallelism, aurora 2012 - 120^3, 24 variables 47

4.6 Single node MPI scaling with 12 task and 12 operator threads, aurora 2012 - 120^3, 24 variables 48
4.7 Scalability test 2012 - 256^3, 16 variables 50
4.8 Example stencil-2 computation, 2012 51
4.9 CPU vs GPU scaling for gradient spatial operator - total graph time, 2012 . . . 52
4.10 CPU vs GPU scaling for gradient spatial operator - single operator time, 2012 53
4.11 Highly parallel task graph structure with punctuated serialization. 56

ACKNOWLEDGMENTS

I would like to acknowledge my graduate advisor, Professor James Sutherland, for all his enthusiasm, patience, and excellent advice. I learned a great deal from him and was exposed to a world of ideas that I might never have had an opportunity to experience anywhere else. I would also like to thank Dr. Tony Saad for being a wonderful friend and collegue over the last few years. He is an amazing person and will someday change the field of mass transport!

CHAPTER 1

INTRODUCTION

1.1 Overview

While the idea of predetermining the outcome or behavior of some physical system is much older than the computer itself, the ability to perform detailed simulation of large, complex problems was out of reach until the latter half of the 20th century. Even then, the tools and expertise required for such simulation were not available to anyone outside of government or large research institutions for much of that time. However, as the availability and cost associated with high-performance computing hardware has dropped, the capabilities of off-the-shelf and commodity hardware has reached a point where conducting accurate simulations has become feasible to a broad audience. Today, the average consumer smart phone is able to provide thousands of times more computing power than warehouse sized machines of the 1960s at a small fraction of the cost.

Along with the rapid improvements in hardware and its availability, there have also been significant improvements in the general accessibility of software tools for utilizing computational resources. With the standardization of large-scale message passing standards, such as the message passing interface (MPI), along with the rise in availability of time-shared super computers and clouds, the idea of using simulation to drive research and development has become more ubiquitous. This can be observed across a diverse range of applications, from the pharmaceutical industry, which attempts to model drug interactions [3], to hardware manufacturers who put new architectures through rigorous, transistor level simulations before ever taping out a physical product [15].

1.1.1 Advantages

A major component of any development process is testing and verification, to ensure that a product or process functions in the desired fashion, and poses no overt danger to an end user when used properly. These testing processes can present significant hazards; complex chemical reactions with toxic or flammable components, explosives testing, or other

high-energy interactions all have inherent risks associated with them. Utilizing the proper simulation tools, many of these systems can be examined in a safe environment, before ever being tested in the lab.

As an example, vehicle manufacturers can model thousands of arbitrary impact scenarios, while obtaining detailed information with respect to forces experienced by a passenger and vehicle components, without being forced to sacrifice valuable materials and testing equipment. This increase safety, reduces material costs, and in the event that a problem is detected during the testing process, a fix can be implemented and retested with significantly less effort than would have been possible in a more conventional testing lab.

Another issue which is important to consider is information completeness. In any real system, we are limited in the amount of data we can collect due to constraints on sensor density, and physical characteristics of the experiment itself. If we wish to examine some type of large-scale explosion or high-energy behavior, then it is entirely infeasible to experimentally capture the complete behavior of the explosive material and the resulting forces through the entire life cycle of the process. However, in simulating such a scenario, we can theoretically capture a complete data profile at all resolved scales of the simulation (so long as we have sufficient data storage capacity).

1.1.2 Challenges

The advantages listed above are not without cost; there are numerous challenges which must be overcome in terms of the science, software, and processing capability required to generate accurate and verifiable results. The process of simulating interesting "real-world" problems consists of a number of nontrivial components. The physical processes governing the problem or system must be identified and, ideally, be well understood. These processes must then be properly stated in a mathematical form, often as a series of governing equations, which can then be solved, either directly or via discretization methods, to yield the desired piece of information. Finally, the described process must be implemented in software, which is then executed on a set of test bed machinery. Once the system is properly described and implemented, then the resulting information can be validated through analysis and visualization.

The final step is of particular interest from a computational science perspective. Translating a series of mathematical expressions into a form which can be accurately solved by a computer is a complex and challenging task. For a given type of problem, there are a variety of numerical methods which can be applied, each of which has trade offs in terms of performance and accuracy. This is important, as in almost any "real-world" situation,

we will require a simulation to not only run and produce accurate results, but to do so in a "reasonable" amount of time. The problem of balancing these constraints, while still producing an effective solution, is quite challenging and will often require the collaborative efforts of engineers, physical and computer scientists, and mathematicians.

1.2 Distributed and Heterogeneous Hardware

To illustrate some of the additional challenges which have become more prominent during the 21st century, it is worthwhile to examine some of the significant changes in the philosophy of hardware and communication architectures over the last few decades, leading to grid-based computing.

1.2.1 Grid-Based Computing

As previously mentioned, the idea of high-performance computing was, for much of the last century, unavailable to the vast majority of scientists, researchers, and businesses. One of the major driving factors of this problem was that almost all initial super computers were monolithic analog processing systems, which would take up entire warehouses, vast amounts of power, and require a full technical staff to operate [10]. This began to change, to some extent, with the introduction of the digital transistor, an amazing device which allowed for electronics to shrink drastically in size and power consumption; however, the notion of a single monolithic machine persisted until the early 1990s.

Initial attempts to develop faster commodity computing infrastructures began to take advantage of the increasing availability of x86-based personal computers (PCs) and the fact that they could be networked together. The idea was fairly straightforward, a single task would be broken into smaller pieces which could each be solved independently, and then each of those task would be "pushed," or assigned, to a networked device. Each device would work on its portion of the larger problem, and then notify the grid's controlling software when it had finished. In this fashion, it became possible to construct extensible computing infrastructures, using consumer-grade PCs, with only a modest initial investment. As networking capabilities and throughput capacity began to improve, this concept of "grid-computing" began to emerge as a popular and accessible method for researchers to benefit from large-scale simulation and modeling.

Grid computing ushered in a new design paradigm, in which discrete distributed resources were connected through a generic medium, such as Ethernet or fiber optics, rather than being hard wired to a central bus of a single large piece of hardware. While making large-scale computing more accessible, this change in design significantly increased the cost of

communication and synchronization between nodes, often by an order of magnitude or more. During the 1980s, numerous message passing environ/ments were designed to deal these problems, and around 1992, the best pieces of each were coalesced into the MPI standard, which has become the primary standard for modern, grid-based, high-performance and scientific computing [3].

1.2.2 Transition from Single to Multicore Architectures

Core clock rates for central processing units (CPUs) grew drastically from 1990-2000, starting in the tens of MegaHertz and ending in the giga hertz range by the end of the century. However, as design processes continued to shrink and CPU clock rates pushed higher, it became apparent that manufacturers were rapidly approaching a performance limit with traditional designs. Problems related to current leakage and heat generation began to scale more rapidly than any associated performance gains, and CPU manufacturers such as Intel and Advanced Micro Devices began to look elsewhere for ways to improve performance.

As the speed of individual chips essentially plateaued, Moore's Law, an idea stating that the overall number of transistors on chip would double every 18 to 24 months, continued to hold [18]. Faced with ever-increasing on-chip real estate, CPU manufacturers began to fabricate chips with multiple processors or "cores" on a single die. The idea was that if it was not possible to improve the speed of serial computations, it was certainly still possible to increase the amount of concurrent work which could be performed on a single device.

1.2.3 Coprocessors and Massively Parallel Hardware

Unlike CPU cores, which have been traditionally designed to maximize serial performance, utilizing complicated circuitry for out of order execution, and doing everything possible to avoid pipeline delays, there has long been the notion of "vector-based" processors. Vector processing is the idea of having hardware which is capable of operating on many pieces of data simultaneously, often executing a single instruction in parallel across each data element, a process known as single instruction multiple data (SIMD). However, with the exception of some multimedia extensions, such as streaming SIMD extensions (SSE), found within x86 processors, general purpose vector-based processors have not traditionally been readily available to consumers. This has changed with the advent of discrete programmable graphics hardware, known as graphics processing units (GPUs). Originally designed to accelerate tasks relating to computer graphics, such as geometry translation and coloring, which are often "embarrassingly" parallel, it was shown by various researchers [24] that

GPUs had the potential for much more general computation.

Realizing the potential of generic, programmable, vector hardware, which was capable of operating on massive amounts of data concurrently, GPU vendors such as nVidia began to develop and expose application programming interfaces (APIs) allowing software developers to more easily exploit the functionality of their hardware [7]. This has, in turn, created a significant need to reexamine software design practices related to high-performance computing, as in certain cases, specific computations may run one or even two orders of magnitude faster on GPU than would be possible on CPU. With this understanding, the ability of software to properly distribute workloads across a variety of hardware, keeping task on the device which offers the best performance, has become extremely important.

1.3 Approaches to Computational Software

This trend, requiring added concurrency from software algorithms, has introduced significant complications into the process of architecting quality, high-performance, simulation software. There are now (multi/many)-core CPUs, massively parallel coprocessor like devices such as general purpose GPUs (GPGPUs), and more exotic coprocessors coming, all of which have their own memory, communication costs, and programing paradigms. Each of these devices offer additional benefits and trade-offs such as running multiple distinct processes vs multiple threads per process in the case of multicore CPUs, and in the case of GPGPUs, extremely fast vector processing in cases where data reuse is high and access patterns are regular. As a result, the choice of when and how to use available computing resources is often not straightforward.

1.3.1 Mastery of Everything

For an end user or domain scientist wishing to take advantage of the numerous computing options available today, the situation can seem quite daunting. It is no longer enough to know a general purpose programming language and then express ones computational model as a self-contained program. Care must be taken with respect to taking advantage of the features available from underlying systems hardware, and designing a program to not only run, but scale effectively on a large computing grid (often consisting of thousands if not hundreds of thousands of nodes). Additionally, as is the case with various GPGPU computing elements, more care must be taken with respect to algorithm correctness, as numeric rounding and floating point representations may not be entirely consistent between devices.

In practice, we would very much like this not to be the case. Scientists and model developers should, ideally, be insulated from what would traditionally be considered engineering or computer science problems, and instead be allowed to focus upon their domain of specialization. Of course, this cannot always be the case, particularly given the pace of modern hardware development; however, there exists significant room for improving the state of existing simulation frameworks and design philosophies.

1.3.2 Our Goal - Separating Form and Functionality

While these rapid increases in the power, data throughput, and flexibility of modern hardware are undoubtedly a step forward, they have come at a significant cost. These costs can be measured both in terms of the learning curve associated with being able to designed software effectively and the increased volume of code required to support a set of heterogeneous devices. As a result, many end users, often including engineers and domain scientists, are unnecessarily burdened by banal and pedantic implementation details, effectively wasting time on concerns which are orthogonal to their needs and goals. This significant problem provides a strong motivator for the development of computational frameworks which are able to abstract the design process of an end user, allowing them to focus on the accuracy and correctness of their work, while the framework itself is able optimize the underlying operations based on specific hardware details.

1.4 The Wasatch Framework

As stated previously, the goal of this thesis will be to extend a component, know as the "Expressions framework," of a computational fluid dynamics solver "Wasatch," to take advantage of a more diverse set of hardware targets, while obfuscating the implementation details and related considerations from end users. The Wasatch framework itself can be thought of as consisting of a variety of components, each of which exists at a different level. For the purposes of this discussion, the upper level component of this framework is responsible for providing MPI-based domain decomposition of a particular space we are simulating, an intermediate, "Expressions" layer, which is responsible for problem setup and solution within each MPI process, and a lower level "Spatial Operations" layer, which implements details related to specific mathematical operators.

CHAPTER 2

GRAPH THEORY AND APPLICATIONS

2.1 Overview

This chapter will be primarily concerned with introducing the required background material for discussion of the approach taken to algorithm decomposition and task scheduling in Chapter 3. This will begin by describing the ideas and machinery related to graph construction, and provide a generic description of the relevant algorithms used. Next, a more specific graph structure will be discussed, which will provide a basis for mapping between generic computational models and our graph construct. From there, we will discuss how the application of this structure to a problem set can provide valuable information, to include general model characteristics, such as identifying serialization points in our algorithm, and improving run-time scheduling behavior and resource management.

2.1.1 Terminology

- Graph - A graph is a set of elements, often written as $G(V, E)$, where V is a set of vertices or nodes, and E is a set of edges, each of which connects two nodes. Edges themselves can be either directed, meaning that they are defined as having specific source and destination vertices, or undirected, in which case the edge indicates only that two nodes are connected. We say that an edge, e is 'incident' on a vertex pair, v_1 and v_2 if v_1 and v_2 are connected by e. For the purpose of this discussion, all graphs will be assumed to be directed; as such, we will write a given edge as e_{v_1,v_2}, where e is said to be an "out edge" of v1 and an "in edge"j of v2.

- Path - A path through the graph is defined to be an ordered set of vertices and edges, which connect a specified source and sink, and written as $p(source, sink)$.

- Cycle - A cycle is a path within the graph such that the starting and ending vertices are equal. More specifically, if a cycle exists, then it indicates that for a given graph, $G(V, E)$, there exists a path, $p_i(source, sink)$, where $source = sink$.

- Directed Acyclic Graph (DAG) - A DAG is a graph utilizing directed edges and containing no cycles.

2.1.2 Algorithms

- Topological Sort - Given a graph, $G(V, E)$, the vertices are said to be topologically sorted if for every edge e_{v_i,v_j}, v_i comes before v_j in the final ordering; the result is that we have a set of objects which can be processed without breaking any dependency relationships. The algorithm is run as follows:
  - Let T_s be a first in first out (FIFO) queue representing our sorted list of objects, and T_u be a list of vertices with no incoming edges.
  - Push all nodes with no incoming edges onto T_u.
  - Pop a vertex, v_i, from T_u and push it onto T_s
  - For each out edge, e_{v_i,v_j}, color the edge as being processed. If v_j has no incoming edges which are unprocessed, push it to T_u.
  - Repeat until there are no more vertices in T_u.
  - T_S will now contain a topologically sorted list of vertices.
- Breadth First Search (BFS) - Given a set of root, or starting nodes, which are each colored to indicate that they have been seen, and placed into a FIFO queue, the algorithm is run as follows:
  - Pop a vertex, v_i, from the top of the queue.
  - Perform any necessary work on v_i
  - For each out edge, e_{v_i,v_j}, check to see if v_j has been colored; if not, color it and push it to the queue.
  - Repeat until the queue is empty.
- Depth First Search (DFS) - Given a set of root, or starting nodes, which are each colored to indicate that they have been seen, and placed into a queue, the algorithm is run as follows:
  - Pop a vertex, v_i, from the top of the queue.
  - Perform any necessary work on v_i

– For each out edge, e_{v_i,v_j}, check to see if v_j has been colored; if not, color it and call DFS on v_j.

– Repeat until the queue is empty.

2.2 DAGs as Computational Models

2.2.1 Motivation

DAGs have been studied extensively in their application to parallel computing problems, including: hardware task scheduling [11], [2], distributed computation [22], parallel compilers [23], [12], [20], numerical linear algebra [16], and even graphical programming tools [4]. As a result, the benefits of being able to represent a problem as a series of tasks and their dependencies within a DAG are significant. Such a representation can often provide a more intuitive understanding of the underlying structure of a computation or algorithm and is entirely open to automated inspection.

2.2.2 Related Work

The process of utilizing software to decompose a computation's form into a representative DAG, and in turn generate an ordering of the problem's component tasks, is an idea that has been utilized in a number of different environments. One approach that shares similarities with the goals and application of work described in this paper, in that it is intended to facilitate development within the context of high-performance computing (HPC), is that of the Parallel Linear Algebra for Scalable Multicore Architectures (PLASMA) project. A collaborative effort between the University of Tennessee, the University of California Berkeley, and the University of Denver Colorado. It is targeted, specifically, at producing a framework that allows programmers to efficiently generate high-performance and portable code for computational linear algebra applications.

The PLASMA library was created in response to perceived limitations within existing linear algebra solvers, such as the linear algebra package (LAPACK) and its set of base linear algebra subprograms (BLAS) [9] on multicore architectures [1]. These libraries exhibit a number of undesirable implementation characteristics on modern systems, such as overuse of “fork” and “large stride” memory access patterns [14], which lead to poor scaling performance on current hardware. As one approach to overcoming these problems, PLASMA makes use of a tiling algorithms concept [5] for QR decomposition operations. This approach provides guidelines for a variety of algorithm requirements, but of particular interest to us is its notion of utilizing a DAG to order and execute tasks.

In work done by Chan et al. [6], it was shown that one of the bottlenecks within the QR decomposition process can be reformulated to exhibit additional parallelism; this is achieved by reducing one of the intermediate matrix computations into a number of smaller block computations. The structure of the resulting computation can be well represented as a DAG and exposes a number of tasks which can be executed asynchronously and out of order. In [5], it is stated that this idea of dynamic scheduling with out-of-order execution was then applied as the basis for obtaining a fine-grained algorithm for QR factorization.

In the tiled algorithm approach developed for PLASMA, a multicore blocking algorithm is decomposed into a DAG describing the dependency structure between each block operation. Using this graph representation, it is possible to directly determine execution/scheduling dependencies of all component tasks, and identify elements forming the critical path of the computation. The result is a framework in which component tasks can be scheduled asynchronously using a simple priority scheduling policy, based on the type of task and its location along the critical path. This, in turn, results in an algorithm where idle time is almost completely eliminated and which adapts, in a basic way, to available computing resources [5].

The process of setting up a program to be able to utilize this tasking framework requires some programmer assistance. First, functions must be converted such that all arguments are removed from the function signature and redeclared as local variables; these variables are then accessed later through an unpacking macro. Next, function calls are replaced by calls to "insert_task()," which must provide a pointer to the function being called, the parameters being passed, their size, and a type identifier the specifies a usage context; usage contexts are specifically defined as "VALUE," "INPUT," "OUTPUT," and "INOUT." At run time, tasks are inserted into the scheduler in preparation for execution, and individual tasks are removed from the queue and executed by worker threads based on their priority and declared INPUT values; for a more detailed treatment, see [16].

Unlike more traditional uses for task scheduling involving DAGs, such as hardware task scheduling [11], the tile-based QR algorithm only stores a windowed subset of the original DAG at any given time. The authors assert that this is due to the extremely fine granularity of individual tasks, which results in graph structures that grow rapidly with problem size.

In contrast to this extremely fine-grained behavior, other frameworks, such as "Uintah," a multiphysics HPC framework written at the University of Utah [8], utilize DAG-based scheduling to facilitate execution of asynchronous MPI-based tasks. In this context, it is possible that each task may be an entire subroutine which must be run on a "physical"

piece of a decomposed domain; in such a framework, MPI tasks themselves may consist of finer grained DAG-based computations.

The expressions framework targets a space that is not focused on either the highly targeted use cases of a library like PLASMA, or the multinode monolithic task scheduling of a library like Uintah. Rather, the expressions library is focused more on creating a development environment in which individual tasks are run on a single node, may differ substantially in terms of computational requirements and resource usage, and which utilize graph structures that can be self-assembling, rather than explicitly directed. In this case, while graph structures can become quite large within the expressions context, they do not exhibit the exponential node growth corresponding to extremely fine task scheduling problems faced by PLASMA. The complete structure of the task graph may be kept in memory during the course of an execution; as a result, there is a great deal more flexibility with respect to analysis of the DAG and more care can be taken with respect to the scheduling of tasks.

2.2.3 The Expressions Context

While significant work exists discussing the variations of this type of process scheduling model and its application to known task sets, [13], little work has been done with respect to the creation of such task sets from an underlying algorithm or model description. In [19], the authors introduce a novel method for the application of the graph-based methodology, described above, to the decomposition of problems found in multiphysics HPC, and in turn building consistent solution algorithms. In the outlined framework, the physical models are reduced to a set of individual tasks, each of which declares its data dependencies to sister tasks; from these task declarations, a DAG is built representing the overall structure of the computation. By processing the resulting graph, it is possible to develop a variety of consistent algorithms for a given model.

To illustrate the general principal, suppose that we have a model describing some physical process. At a basic level, this abstraction will consist of some number of variables or objects, and a series of operations which will act on these objects to produce some type of result. In the simplest case, we can imagine a function which computes the sum of two objects, $F(A, B) = A + B$. This system will consist of three data objects: A, B, and their sum $S = A + B$, as well as a function F, which requires A and B and produces S. If we generalize this notion to include any number of functions and objects, then we can, for a consistent model, generate an ordering to any valid set of operations such that their data requirements are satisfied and we obtain the desired solution(s).

2.2.4 The Dependency Graph

In general, there are two distinct methods which are commonly used to represent a task set as a graph.The first, is to construct what is known as a dependency graph, which as its name suggests, describes the dependencies of each task using edges. What this means, is that the root nodes at the "top" of the dependency graph will be the last set of tasks which will be able to run. This is due to the fact that each edge, e_{v_i,v_j} in the dependency graph will indicate that v_i is a consumer of v_j, and therefore, the only root nodes in the graph will be those who have no consumers. The dependency graph represents our intuitive understanding of how a series of operations forming an algorithm are connected and will serve as an initial step in the process of translating a real model description into its graph representation.

2.2.5 The Task Graph

The second method, which is often better suited to the idea of the graph as the basis for implementing the model, is know as the "task graph." The task graph differs from the dependency graph in that each edge, e_{v_i,v_j}, of the task graph has been reversed from its equivalent edge, e_{v_j,v_i}, in the dependency graph. In this representation, the root nodes of the graph will be those which have no incoming edges, and in turn have no initial prerequisites for execution. The task graph represents the effective "flow" of an algorithm and its direction of execution; it will be used extensively when we discuss ideas related to the scheduling of individual tasks in Section 3.

2.3 Extracting Information

2.3.1 Task Granularity

Here, it is worth pointing out that the notion of what constitutes a task has purposefully been left fairly generic. Within the context of a software implementation, a task itself may range from a single assembly instruction, all the way to the execution scope of an entire model, although, in almost all cases, either extreme would not be very useful. In general, we can package as many tasks as required into a single super task, adjusting the size of the resulting graph and the amount of work done for a given vertex.

2.3.2 Graph Introspection

In addition to the direct benefit of using a graph-based approach for generating algorithms to solve a specific model, the structure of the graph itself can provide us insight into a number of the theoretic structural properties of a model. Specifically, after a bit

of relatively straightforward analysis, it will allow us to estimate properties such as the percentage of the model which can be parallelized and the minimum total execution time. These will be given without proof; for a more detailed treatment, see Sinnen [22].

2.3.3 Bounds on Execution Time

The minimum execution time for a graph, given a fixed set of hardware, can be obtained through examination of all sequential computations within a graph and then isolating the longest. To see why this is correct, we define the following:

- Schedule - A schedule is a set vertex execution ordering. We will call a schedule "consistent" if it obeys all dependency relationships of a graph

- Path Cost - For a given DAG, $G(V, E)$, and a path $p(source, sink)$, the total cost of executing p is given as:

  $$\sum_{e_i \in p} t_e(e_i) + \sum_{v_i \in p} t_e(v_i) \tag{2.1}$$

  where e_i and v_i are the edge and vertex components of the path.

- Critical Path - For all paths within the graph, the node(s) which have the maximum path cost is known as the critical path.

From our definition of a critical path, we can make two valuable observations. First, any critical path within a graph will begin at a root node and terminate at a leaf node. Intuitively, this property seems reasonable; however, it is not entirely obvious. The truth of this statement can be obtained by supposing that you have a critical path whose source node is not a root and from there proceeding to a contradiction. The second observation is that the critical path provides a lower bound on the time required to execute any feasible graph schedule. This property follows from the principal that in a task graph, edges signify a dependency relationship, and therefore, computation along a path must be sequential. If this is the case, then a critical path will represent the greatest set of sequential, or serialized tasks.

2.3.4 Parallelization

While the above discussion yields insight regarding absolute execution times of our graph, it does not tell us anything about the requirements for approaching this minimum bound. In an extreme case, we could potentially have an entirely serial graph, such as a

single chain of of n tasks, where the task $1 < k \leq n$ depends on task $k - 1$. In such a case, it would be detrimental to allocate any additional resources for task execution, or attempt to perform any detailed introspection of the graph's properties. In less extreme situations, it may be the case that we could expect to see no benefit from scheduling on more than two or three processors. It turns out that we can again utilize the structure of the graph to produce a theoretic bound on potential gains, given additional processing resources. Under the assumption of minimal communication costs, we define the following properties:

- Single Processor Time τ_1 - This is the total time required to execute a graph, given a single processor. $\tau_1 = \sum_{i=1}^{N} \tau_e(n_i)$
- Infinite Processor Time τ_∞ - This is the total time required to execute a graph, given unlimited processing resources. This value is equal to the maximum finishing time of all nodes.
- Speedup S_n - This is the reduction in the time taken to execute a graph when utilizing n processors, given as $\frac{\tau_1}{\tau_n}$. Of particular interest is the speedup value for $n = \infty$, or $S_\infty = \frac{\tau_1}{\tau_\infty}$.

The value S_∞ represents the maximum theoretic speedup we can obtain by allocating an infinite number of resources toward the execution of our graph. Using this value, we can compute an "Amdhal score," $P(G) = \frac{1-\frac{1}{S_\infty}}{1-\frac{1}{N}}$, which represents the percentage of the graph which is parallelizable. This score provides a theoretical limit to the performance improvements we can expect from a given graph and provides us with a hard limit as to the number of resources which can be legitimately allocated toward a specific problem.

CHAPTER 3

FRAMEWORK EXTENSIONS

3.1 The Expressions Library

3.1.1 Overview

The expressions library is a framework designed to elevate the development level of an end user to the point where they are able to write their code with a structure similar to that a high-level interpreted language, such as Matlab, while at the same time being able to obtain scaling performance required by traditional high-performance computing applications. This is accomplished through a high-level Domain Specific Language called NEBO (not covered in this work), and a design paradigm which allows an end user to describe processes in terms of generic operators and their data dependencies. Once an end user has used this system to described their computation, the expressions framework takes the abstraction and automatically constructs an appropriate algorithm based on available resources.

In the usage context for this work, the expressions library serves as an intermediate layer between the Uintah/Wasatch component described in (1.4), responsible for domain decomposition and MPI message passing, and the spatial operator component, responsible for the abstraction of data fields and the implementation of mathematical operators (Figure 3.1).

3.1.2 Concepts and Terminology

- Expression - An expression is a abstract representation of a mathematical operation. Expressions expose their purpose and requirements through a set of interface methods given below. The basic concept of an expression is illustrated in Figure 3.2.
  - advertise_dependents() - Returns a list of all other expressions which are required for this expression to be successfully computed; this allows us to determine ordering requirements of the model before execution.
  - bind_fields() - Retrieves objects representing base fields which are consumed and computed within the evaluate(); this connects the logical operation with

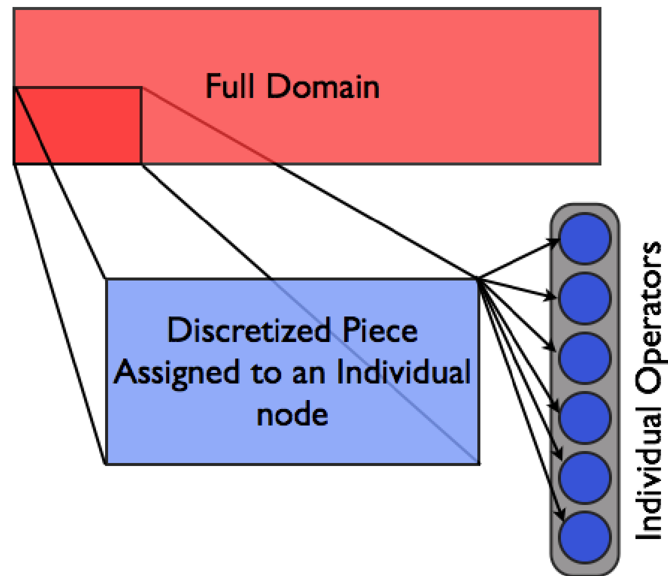

Figure 3.1. Framework component diagram.

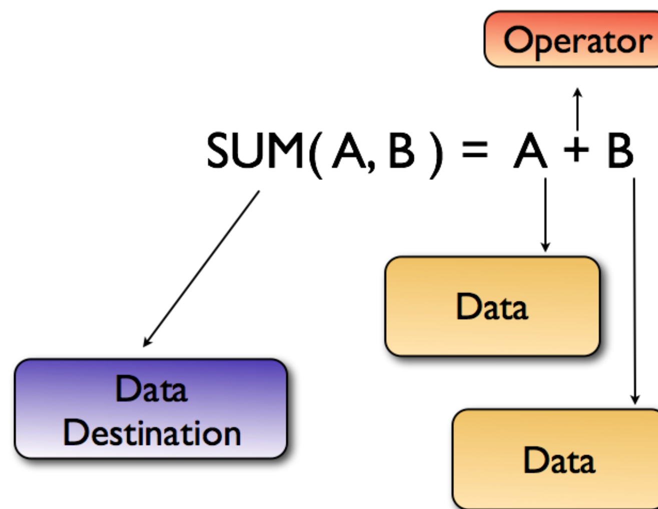

Figure 3.2. Example of a basic expression.

the physical resources it requires and allows resources for each expression to be allocated and bound independently.

– bind_operators() - Obtains objects representing various mathematical operations (ex. Div, Grad, etc.) used by the expression.

– evaluate() - Executes user-defined operations forming the core work of the ex-

pression; note that the internal contents can be as fine or coarse as desired.

- Expression Tree - A class object responsible for analyzing collections of individual expressions, exposing the overall structure of the computation, and ensuring that the structure is logically consistent. The expression tree is used to construct a dependency graph, described in Section 3.2; in this representation, graph nodes represent the expressions themselves, and directed edges express the data dependencies between expressions (Figure 3.3).

- Field Manager - A class object responsible for managing field memory for the Expression Tree. This class provides interface methods to register, allocate, migrate, and deallocate all memory resources associated with a specific field.

- Task Scheduler - A class object responsible for determining execution behavior based on a dependency graph; in general, this includes creation of either a static or dynamic schedule for each task, which preserves dependency requirements, and developing a variety of performance metrics for the overall computation.

3.1.3 Implementing an Expression

The process of defining a task, or "expression," simply requires an end user to inherit and implement the interface contract described above. As an example, suppose we are computing the Heat Equation, (Equation 3.1).

$$\partial \delta T \delta t = -\frac{1}{\rho c_p} \partial \cdot q + \frac{1}{\rho c_p} S_T \tag{3.1}$$

$$q = -\lambda \nabla T \tag{3.2}$$

One of the operations required is to compute the gradient of the temperature, an operation which requires a scalar field representing the temperature, "T," and a gradient operator. To implement the required interface methods, we can first create an expression that will compute the gradient of "T," in this case, called "GradTExpr," and set it to require another expression which represents a temperature field (Figure 3.4). This will allow the expression tree to build a list of dependencies for this operator and ensure that all required fields, in this case "tempT_," are available before attempting to evaluate this expression.

Note here that our expression for temperature, "tempT_," will not possess any dependencies or perform any computation. The element is an explicitly defined source term, which must be supplied by the caller, and will in turn have an initial condition and be

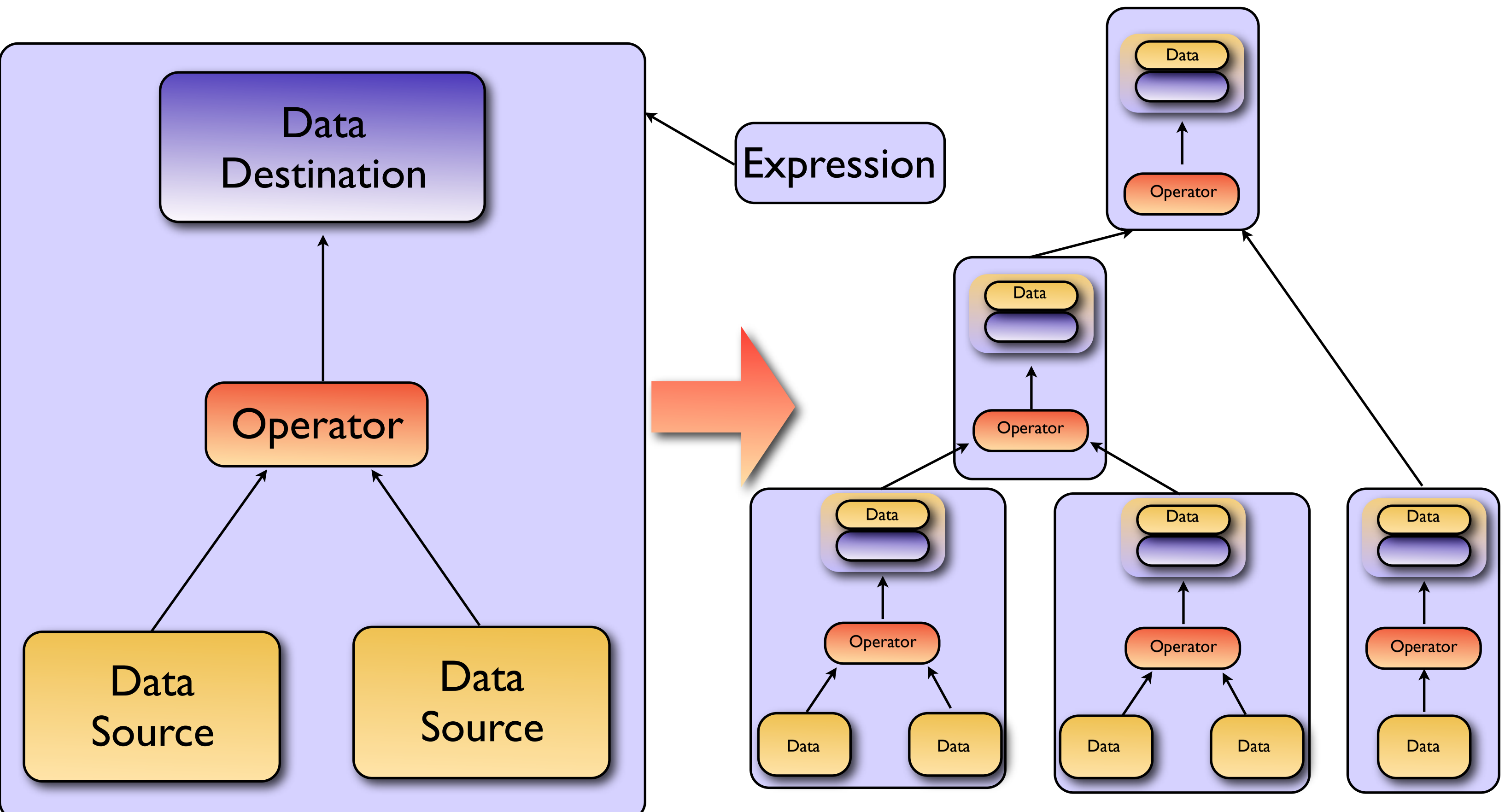

Figure 3.3. Constructing the task graph from individual expressions.

```
template < typename GradOp >
void GradTExpr<GradOp>::
advertise_dependents(Expr::ExprDeps& exprDeps){
        exprDeps.requires_expression(tempT_);
}
```

Figure 3.4. Advertise dependents implementation.

updated during successive problem iterations; these terms will become the root nodes of our execution graph

Next, we implement the bind operator method (Figure 3.5) and bind fields (Figure 3.6) functions, which will fetch the required gradient operator and obtain a reference to the source field, "tempT_," for this expression. Finally, we can implement the evaluate method, which will apply the gradient operator to our temperature field and place the result into the value field of our expression (Figure 3.7). In this fashion, we are able to completely describe the desired operations and their data dependencies, without any specialized knowledge of available hardware resources, threading model, or implementation details.

3.1.4 Constructing the Task Graph

After specifying the behavior of each expression for a given model, the expressions are passed as a collection to an expression tree. The expression tree will then parse the expression set and construct a task graph, as a "Boost Graph" structure [21], where individual

```
template < typename GradOp >
void GradTExpr<GradOp>::
bind_operators(const SpatialOps::OperatorDatabase& opDB){
        gradOp_ = opDB.retreive_operator<GradOp>());
}
```

Figure 3.5. Bind operators implementation

```
template < typename GradOp >
void GradTExpr<GradOp>::
bind_fields(const Expr::FieldManagerList& fml){
        temp_ = &fml.template field_manager<ScalarField>().field_ref(tempT\_);
}
```

Figure 3.6. Bind fields implementation.

```
template < typename GradOp >
void GradTExpr<GradOp>::
evaluate(){
.       gradOp_->apply_to_field(*temp_, this->value());
}
```

Figure 3.7. Evaluate method implementation.

vertex objects hold a reference to the base expression and related meta information, and edges represent the data dependencies between expressions. This can be done through explicit dependency specification, or by having the expressions themselves advertise their dependents and have them dynamically inserted into the expression tree. The second case, with dynamic insertion, yields a very useful method for automatic algorithm construction, in that the compute ordering is not need to be specified explicitly, but rather can be deduced by the expression tree at run time [19].

Once all expressions have been accounted for, the expression tree will perform error checking to ensure that the model does not contain circular dependencies, and register all required fields with an appropriate field manager; as a useful side-effect of this construction process, we are able to determine the problem's memory requirements before execution. At this point, the solution algorithm and computational resources required for the problem specification will have been determined, and the graph is ready for execution.

Using the example of the heat equation, one possible implementation will yield the directed graph seen in (Figure 3.8). Once assembled, the dependency graph contains all information required to represent the structure of the desired computation. After being handed to the task scheduler, this structure will form the basis for constructing an algorithm whose output is the solution to the heat equation.

3.1.5 Design Goals

The primary focus of the work described by this thesis has been to extend the functionality of the Expressions Library to utilize both existing and future acceleration technologies, without requiring an end user to have hardware specific knowledge. To support this larger aim has required significant modification to a variety of framework systems, which can be summarized a follows:

- Implementation of a flexible task scheduling system, capable of utilizing a variety of scheduling algorithms, ranging form simple to complex, depending on available hardware and software resources. To facilitate the necessary flexibility, a task scheduler should be able to exert a high-level of control over the execution process and associated resources, to include:
  - To reallocate threading resources between task and operator level execution.
  - To determine scheduling priorities for individual expressions based on available hardware and graph meta information.
  - To assign tasks to arbitrary hardware targets dynamically.

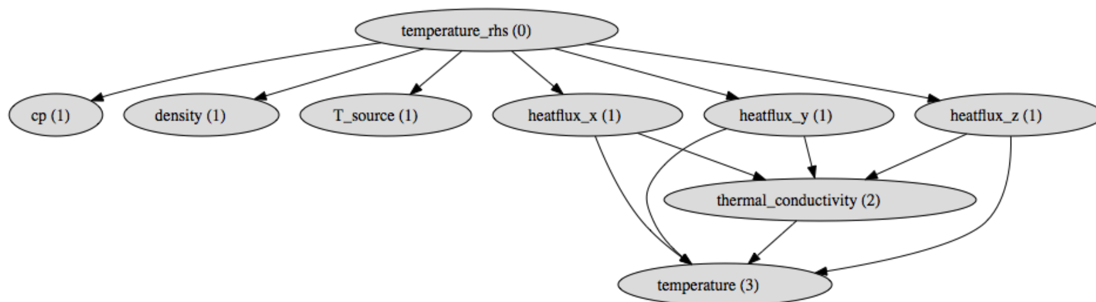

Figure 3.8. Visualization of a potential task graph for the heat equation.

  - To ensure data availability between expressions which are computed on different hardware devices.

- Modification of existing field managers to allow for delayed memory allocation, additional allocation targets based on available hardware, and support for migration of fields between devices.

- Extension of the Spatial Field library to support field consumption from a variety of devices, and transparent operator selection based on the device on which the operation will be computed (Section 4.2).

3.1.6 A Flexible Scheduling Model

To support an execution model where we may wish to utilize a variety of scheduling algorithms, each with potentially widely varying behavior, the scheduler itself is treated as a component of the execution framework, rather than a static implementation. Using a standard contract model [17], where base functionality is defined and inherited by specific scheduler implementations, we are able to test and change schedulers easily without refactoring core components. During the construction phase of an expression tree, the scheduler is built and then handed the dependency graph for the simulation model; later, when the "execute_tree" method of the expression tree is called, it will notify the scheduler to perform any requiring preprocessing, and finally to run the algorithm.

The scheduler itself is now left with all available information about the operations and constraints required to reproduce the desired model. From the most basic perspective, it could simply sort the task set topologically and execute them in order. However, in most cases of interest, this will be insufficient, and we will want to develop an implementation

that optimizes for speed, resource utilization, or some combination of both. By providing a simple interface contract to the scheduler used by the expression tree, while at the same time exposing as much control and information as possible to the scheduler's internals, we allow for new scheduling models to be rapidly prototyped as new components become available. Using this approach, we can integrate existing hardware accelerators, such as GPUs, seamlessly into the framework, while at the same time providing a foundation capable of supporting new devices as needed.

3.1.7 The Priority Task Scheduler

One of the most straightforward scheduler implementations is used when executing on a target system in which we assume a uniform, shared memory architecture with full communication interconnects. More specifically, this means that all processing devices are homogeneous, the cost of assigning a specific task to a hardware device is equal for all devices, and that communication between devices is negligible (Figure 3.9). These assumptions are reasonable in the context of our expression level process, running on a single computing node, where we leave the task of binding threads to processor cores to the operating system, and neglect latency relate to contention of the memory bus. Under this restricted model, we are left with the responsibility of ensuring that individual expressions are scheduled for execution in a near optimal fashion, based on execution times and graph layout.

3.1.8 Graph-Based Execution

As one of our requirements for the priority task scheduler, we have said that it should be able to generate task priorities based on both executing timings and the general structure of the graph. This implies that the scheduler needs to be able to examine and reason about the task graph and be able to maintain/update information stored on its vertices. To support this behavior, the scheduler assigns priorities to each node in the graph and during its setup phase, based on its depth in the graph and the total number of other tasks which consume it (Figure 3.9). Later, during execution, each task will be timed and have the result added to a moving average, providing an estimate for its likely execution time on successive iterations.

After successive iterations of the task graph, the scheduler is able to improve on its initial priority estimate, by using the measured execution times to increase priority values for heavier weight tasks. For simulations where the graph may be executed thousands, or hundreds of thousands of times, this design has the effect of heavily prioritizing tasks

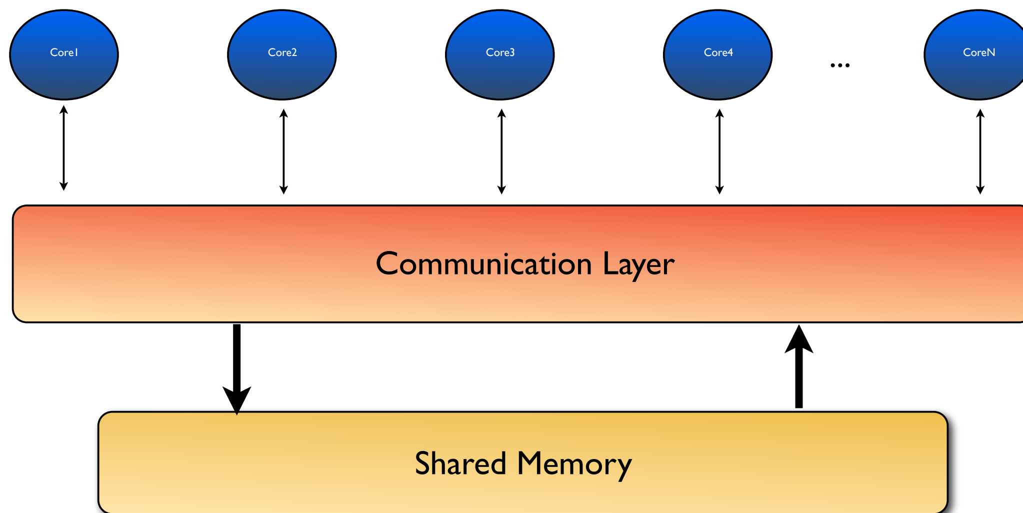

Figure 3.9. Simplified communication memory model, multicore CPU.

located near root nodes, and those which form serialization points within the graph.

3.1.9 Task Dispatch Model

Execution of a task graph is initiated in a fairly direct manner. We first push all root node vertices, defined to be those with no “out” edges (dependencies), to the task queue for execution. The perthread process of acquiring and dispatching additional tasks occurs using the following execution callback model (Figure 3.10):

- A worker thread will select a vertex from the task queue and call through the task scheduler to prepare for the vertex’s expression execution.
- The expression is executed.
- The worker thread will call back to the scheduler and inform it that the expression is done.
- The scheduler will notify all consumers of the finished expression that one of their dependencies has become available.
- If any of the dependency counters on the notified vertices reaches zero, they will indicate to the scheduler that they are in a ready state and will be added to the task queue.

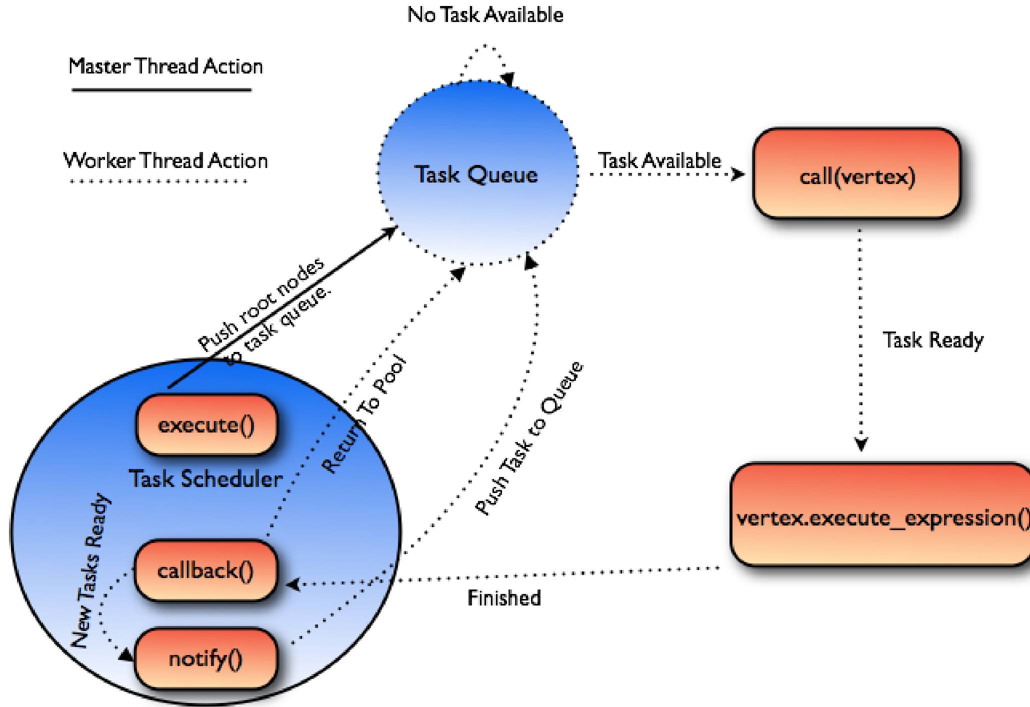

Figure 3.10. Priority Task Dispatch Model.

- The process repeats until all vertices have been exhausted.

Ignoring function call overhead, the total time required for a worker thread to execute an expression is bounded by (Equation 3.3).

$$t_{V_i} + (t_l + t_e) \cdot |V_{V_i}| \tag{3.3}$$

$$t_{V_i} \gg (t_l + t_e) \cdot |V_{V_i}| \tag{3.4}$$

where V_{V_i} is the set of all dependent vertices of the ith vertex, and t_{V_i}, t_l, and t_e are the times required to execute the expression, look up a dependency target, and update the target, respectively. For situations where task granularity is not too fine, meaning that work done by each expression is much greater than the time required to examine its incident edges (Equation 3.4), this method of scheduling has been shown to produce negligible overhead (Chapter 4).

In situations where expression granularity is extremely fine and exhibits a high degree of dependency coupling, this scheduling method may not be suitable. In such situations, where executing "on top" of the graph itself produces unacceptable overhead, a better solution may be to preallocate all necessary fields, build a topologically sorted list from the task graph, and execute from a FIFO type queue structure.

3.1.10 Improving Resource Utilization

Assuming the basic homogeneous scheduling model described above, we can begin to examine how to better exploit available information to improve performance, and look to extending the scheduler's functionality to support additional hardware targets. Within the context of the larger framework, this may include reallocation of available threading resources based on the task graph's structure, improved memory utilization, or other intelligent decision making related to consumer availability and hardware assignment.

3.1.11 Memory Resources

One immediate advantage of having execution take place on the task graph is that we are presented with a direct method for determining when the resources associated with an expression will no longer be used. As the edges of our graph represent data dependencies, then we know that for any given vertex, its resources may be released when each vertex on one of its out edges has been executed. As a result, we can refer to any vertex node V_c, for which another vertex V_d is a dependency, as a consumer of V_d; by keeping track of how many consumers a given vertex has, and decrementing that counter each time one of those consumers finishes execution, we are able to free a dependency's resources as soon as they have been fully consumed. This is very useful property, as we are only required to keep an exact working set in memory at any given time (Figure 3.11).

While the graph-based approach itself provides a direct method for determining which vertices no longer require resources, early field manager implementations did not provide a method for deallocating a field without removing its registry entry (Figure 3.12). As a core component involved in registering, allocating, and retrieving memory resources related to all fields, this inflexibility would prove to be too restrictive, not only in the context of field persistence, but also with respect to field allocations on external devices. As a result, the existing field managers were modified to store a structural interface to an underlying field, allowing for a variety of logical states, and an extensible list of allocation targets.

The rearchitected field manager system assigns a field structure to each registered field, which will exist throughout the lifetime of the field manager. The field structure itself maintains a variety of meta information related to the field, and methods for interacting with it. Meta information includes the logical state of the field, including whether or not it is currently allocated, if it contains coherent information, the type of memory management scheme associated with the field, and the type of device it resides on. Additionally, the field structures support methods for referencing, freeing, and modifying the behavior of their underlying fields. This modified behavior is exhibited in Figure 3.13.

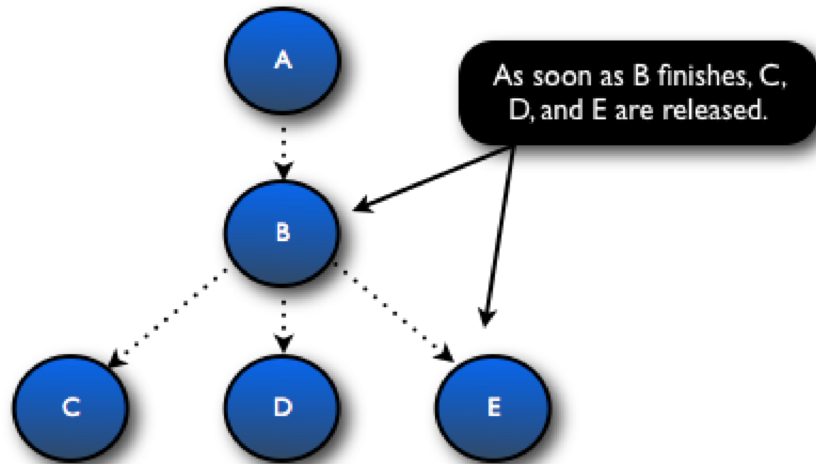

Figure 3.11. Dependency graph illustrating the deallocation of a fully consumed expression.

In the modified system, the specifics of field allocation are changed significantly. Unless a field is specifically tagged to be "static," meaning that it is required to exist and be available at all times during the life of the field manager, fields themselves are not allocated until their first access. Additionally, nonstatic fields are allocated from a central memory pool, and new memory is allocated only when the pool itself cannot provide the requested field type. Later, when a dynamic field is released back to the system, rather than being freed, it is simply returned as a resource to the memory pool. In this manner, not only is it possible to reduce memory consumption through "just in time allocation" and maintaining a minimal working set, but system call overhead associated with memory allocation is entirely removed in at most one execution of the graph.

In conjunction with the much improved usage behavior described above, the modified field manager architecture also offers the flexibility to modify management behavior automatically. This includes basic modification of the persistence flags associated with a field to full reassignment of the field to an alternate manager policy. This can be useful in cases where an end user may wish to "lock" some intermediate field after a number of iterations to sample, or spot check the model's behavior; in such a case, the user would set a persistence flag through the expression tree which would then be propagated to the task scheduler and down to the field managers. On the next model iteration the field of interest could be sampled, verified, and unlocked.

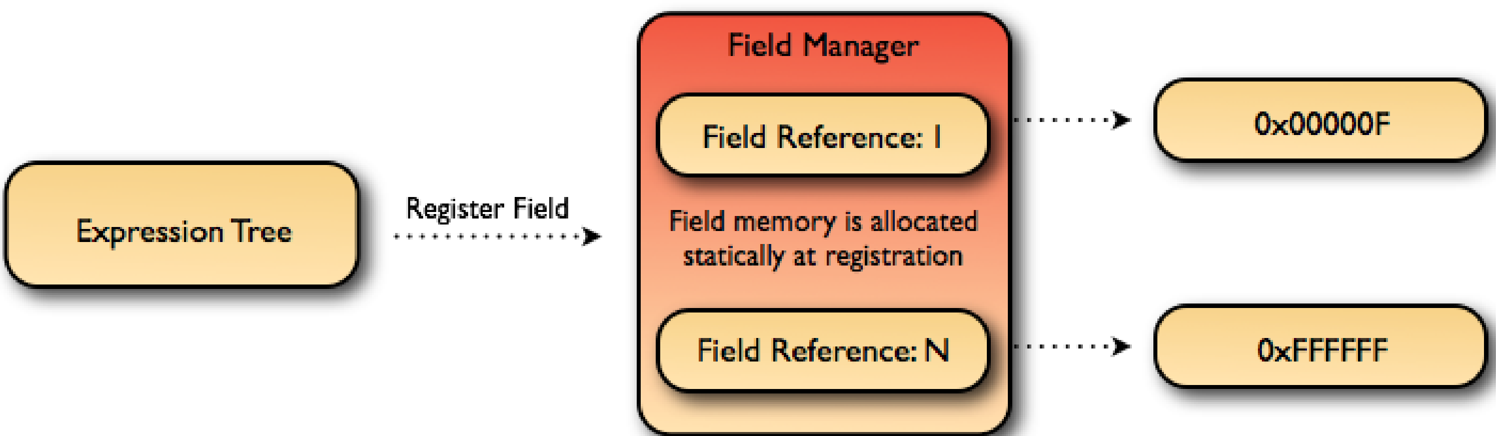

Figure 3.12. Initial field manager implementation.

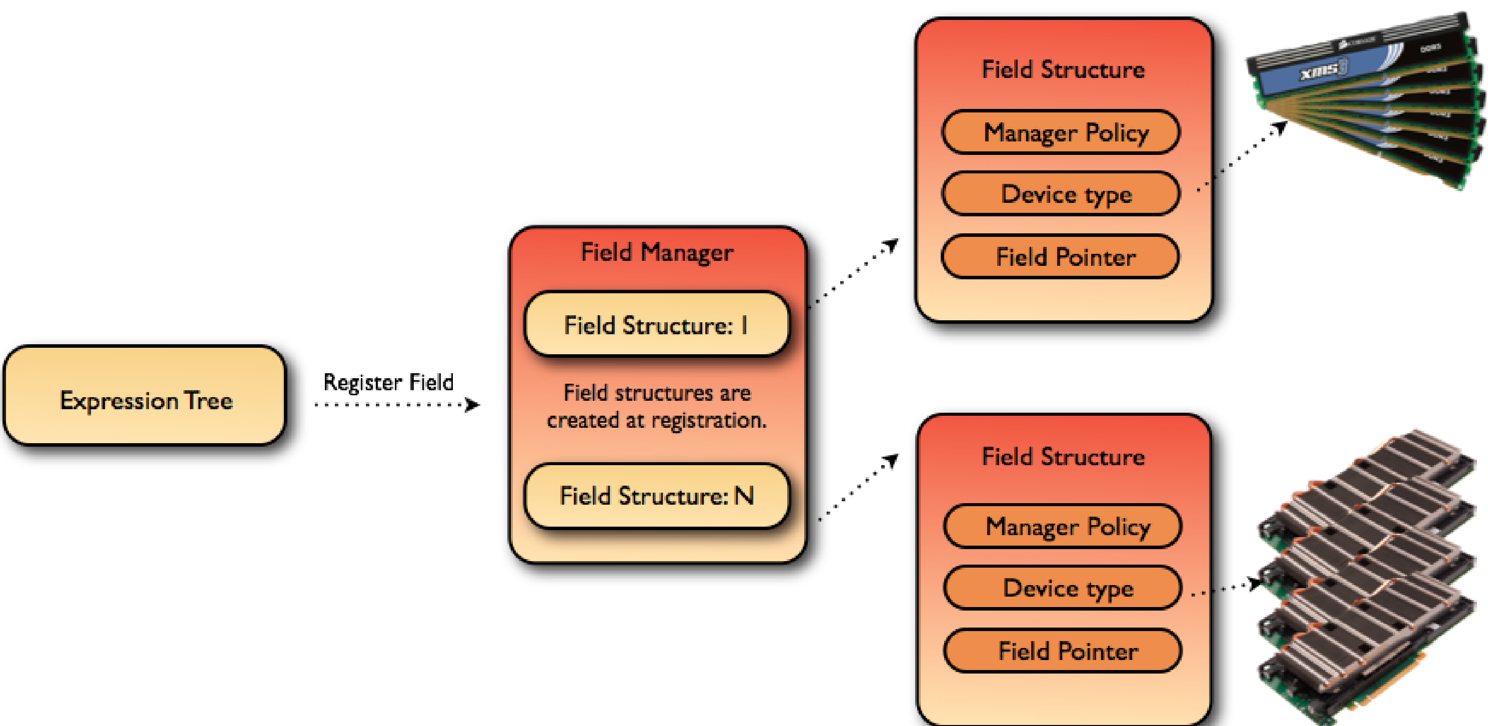

Figure 3.13. Updated field manager implementation.

Similarly, the scheduler itself may decide that an expression would be better suited to execute on some type of coprocessing device, such as a GPU, and in turn notify the field manager to update the memory policy associated with fields computed by the given expression. The field manager would then automatically ensure that any data associated with the field was automatically propagated to the proper GPU device, and the previously allocated fields would be returned to the memory controller. On the following call to execute the expression, the expression's operators will be remapped based on the updated hardware and execution will continue.

The complexity associated with introducing the described memory saving operations into our priority scheduler is equivalent in both time and space requirements as to what is required to support "on graph" execution. Each graph vertex is required to maintain a list of consumer vertices, when the vertex itself finishes, each consumer vertex will be notified, and when their consumer counts reach zero, they will have their field resources released.

3.1.12 Threading Resources

Just as the expression library is capable of exploiting task level parallelism, by decomposing a model specification and examining the related execution dependencies, it is also possible to expose inherent parallelism within mathematical operators or over their given domain. As a simple example, we can imagine an operator performing some number of direct pointwise computations on an NxN field; such an operator could, in many cases, compute many of these pointwise values for each computation simultaneously or perhaps compute the different operations in parallel.

Although not fully described here, this type of memory-based operator decomposition is a significant part of the spatial ops/NEBO framework, which has complete discretion in how an operator is implemented and as a result exploits various types of parallelism inherent to it. Given this notion, the spatial operators have their own threading resources in the form of a FIFO thread pool. As the internals of the operator thread pool are fully accessible by the task scheduler, this provides us with an additional degree of freedom in our ability to address potential computation bottlenecks.

Of course, regardless of logical differences, both libraries share the same underlying resources; therefore, at the thread level, if the total threads allocated between the operator and expression thread pools exceed the available hardware resources, workers from each thread pool will be competing for processor time. To avoid such situations, we would like the task scheduler to, at a minimum, allocate a number of total worker threads such that we force as few context switches as possible.

Accomplishing this is simple, as we can supply information regarding the total number of available processing resources to the scheduler, and it can determine a static ratio for how they should be allocated to each thread pool. If we imagine an idealized a situation, in which we have three total threads and three available tasks, each completely parallel at either the task or operator level, this trade off can be observed in Figure 3.14. However, this is rarely the case, and instead we see direct benefits to allocating threads in less extreme ratios. Once we have this notion of the task scheduler acting as a managing entity for multiple levels of parallelism, it affords us additional possibilities for performance improvement.

Suppose we wish to do better than static pool sizing, but rather, we would like the task scheduler to adjust the total number of workers assigned to each pool based on analysis of the task graph and real-time feedback during execution. In this case, our graph-based algorithm formulation will again prove to be extremely useful. Recalling the notion of parallelization scoring related to Ahmdal's Law, from (Section 3.2.2), we know that given a reasonable estimation of execution times for each vertex within our task graph, we can provide a good approximation for the maximum potential speedup and task level parallelism. This implies that as we progress through successive graph iterations, we can reasonably estimate the benefit of migrating threading resources between thread pools.

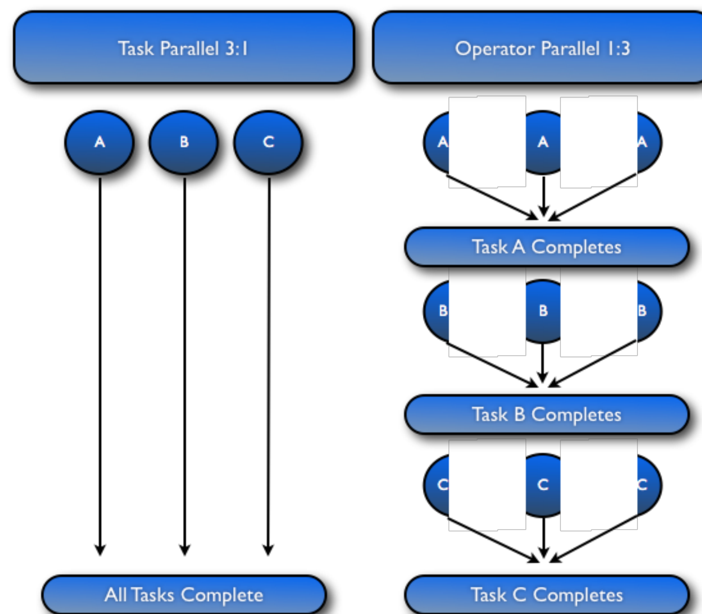

Figure 3.14. Idealized multilevel parallelism.

As an example, suppose that we are executing the task graph shown in Figure 3.15, and that each node of our system contains 8 processor cores. From section 3.2.2, we can compute the single and infinite processor case timings as $\tau_{one} = 5$, $\tau_{\text{inf}} = 4$, which yields a maximum theoretic speedup of 1.25. Therefore, in this case, it would not be worthwhile to allocate more than 2 threads toward the execution of our task graph; however, if we imagine that the underlying tasks each to be large matrix matrix multiplies, then we could potentially obtain a 6 or 7 times speedup by pushing all our threading resources into the operator thread pool.

3.1.13 The "Hybrid" Task Scheduler

Given the general priority scheduler, along with the selection of framework tools described above, our next task is to develop a scheduler that its capable of reasoning not only about task ordering, but also with respect to the assignment of tasks to devices other than the CPU. This process introduces a good deal of complexity into both the process of graph analysis and the execution time responsibilities of the task scheduler. Not only does it break our simplified memory model (Figure 3.16), used to construct the priority task scheduler, but it injects ambiguity into the process of assigning execution timings, generating parallelization scores, and the notion of a field residing in a single location. The following is a basic outline of the major complications involved in allowing for individual

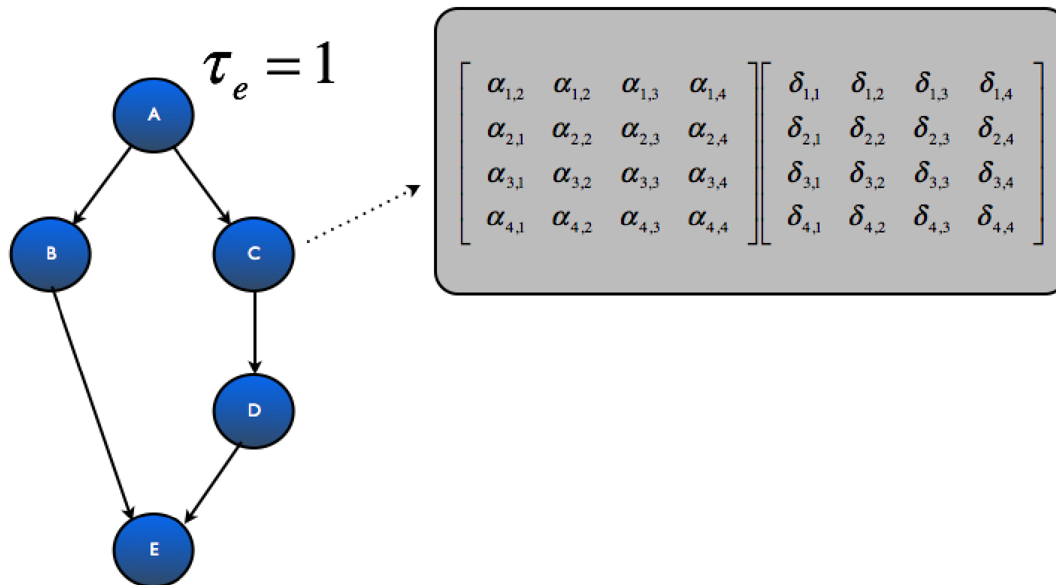

Figure 3.15. Example task graph analysis to determine thread allocation. Note: execution times are all normalized to 1.

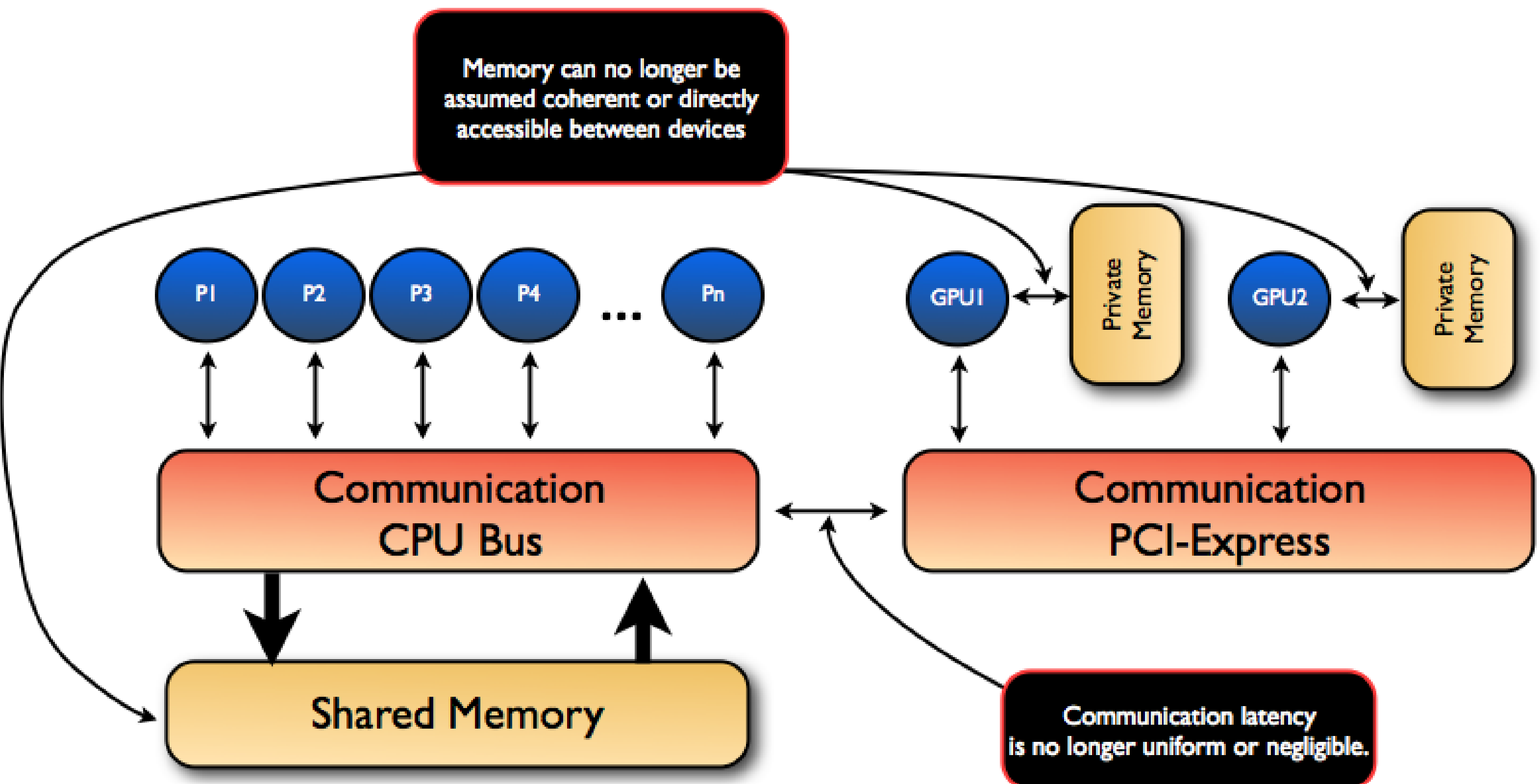

Figure 3.16. Expanded communication memory model.

tasks to be allocated across heterogeneous hardware.

Scheduling of a task to a hardware device can no longer be considered to have a uniform cost. This is because there are tangible differences between the data transfer costs associated with differing communication layers. This issue is further compounded by the fact that in many cases of interest, the cost associated with a data allocation is not necessarily unidirectional; in a general sense, an expression will consume N data fields and produce M data fields, resulting in a total data transfer to and from a nonlocal computation equal to $(N + M) * field_size/transfer_rate$. This additional timing parameter can be thought of as applying an "edge scheduling" cost, meaning that we are, in effect, faced with a problem of scheduling paths in the graph, rather than simply tasks (Figure 3.17).

Acquiring representative timings for the execution of individual tasks requires us to make a determination as to which device a given task will execute on, including the transfer times described above. This implies that given a variety of coprocessor devices, we will need to maintain device specific timings for each individual task, along with a generic transfer timing parameter. Furthermore, without prior knowledge or hints as to task execution timings, we will need to execute each task at least once on every device in order to build its timing profile.

The notion of "consuming" a field is significantly complicated; it is entirely possible to construct a situation in which a field which is computed on an external device, such as a GPU, is required as an input to a collection of other expressions, each of whom may not reside on the same device or devices. The result of a matrix multiply performed on GPU 1 could be used by GPUs 2, 3, and 4, and a variety of CPU tasks. In this case, the abstraction requiring a field to be owned by a single expression could potentially break down, if care is not taken to ensure coherency and enforce the distinction between source and destination fields (Figure 3.18).

3.1.14 Device Assignment

Device assignment occurs in two distinct stages. The first occurs during the scheduler's setup phase, where we determine which devices each expression is eligible for execution on. The reasoning behind this is that while the operator framework is quite robust, there may be situations where certain functionality is not available to a user and they wish to write their own implementation within an expression's evaluate method. In these cases, the task will only be eligible for execution on devices indicated by the end user.

Following this, several iterations of the graph must be executed in order to obtain execution timings for each unique device type. Once all necessary timings are collected,

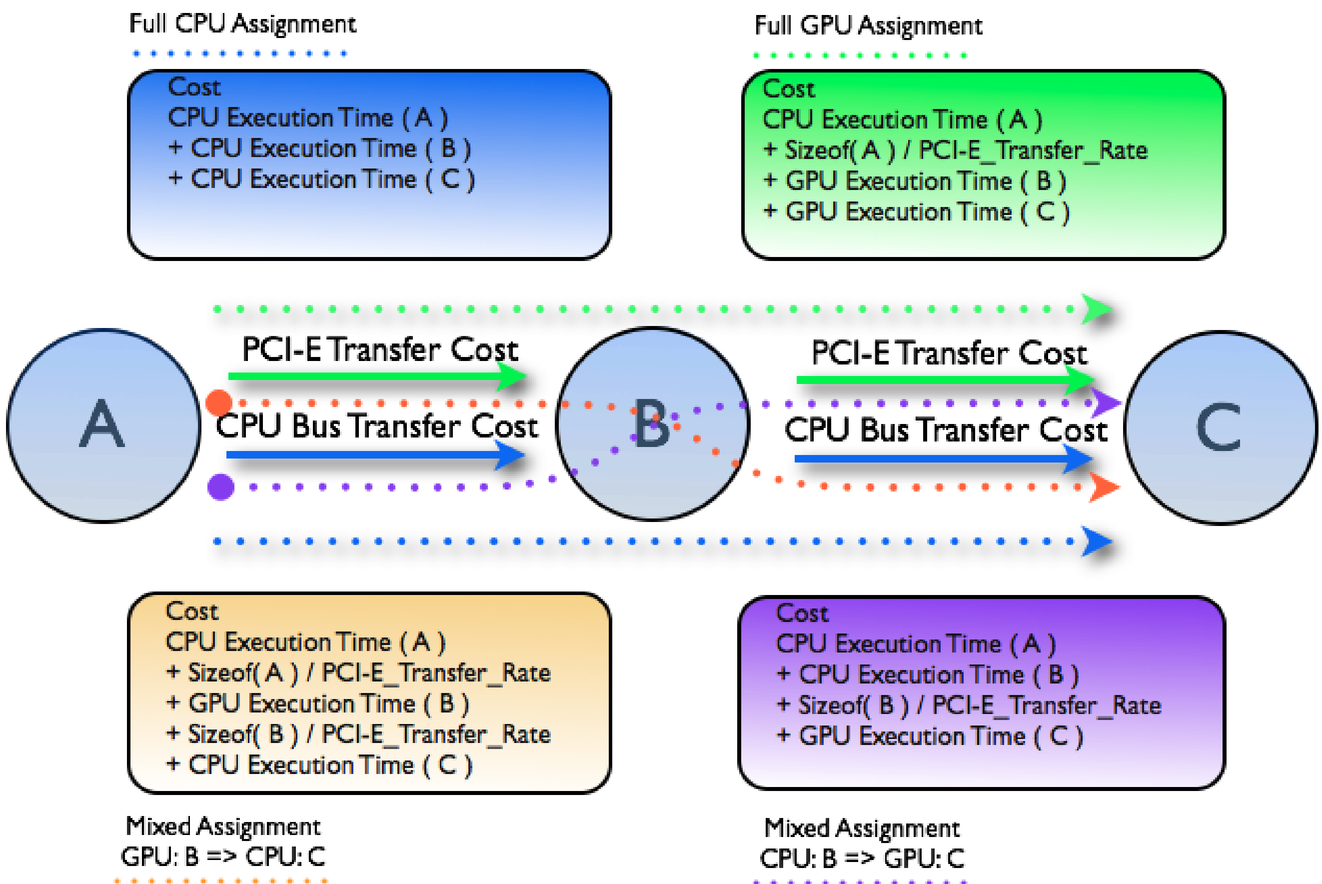

Figure 3.17. Scheduling with mixed hardware and edge transfer cost.

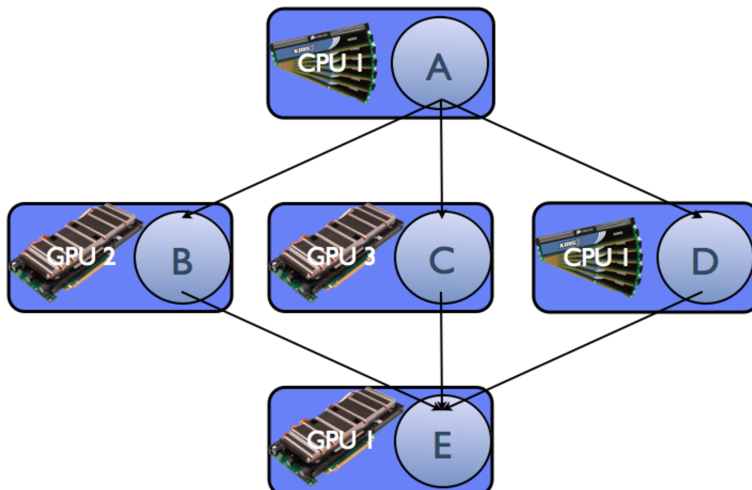

Figure 3.18. Multiconsumer field problem.

then the graph must be run through another introspection phase in which we determine the best resource to compute each expression, fix their device targets, and begin the process of "path coalescing" described below.

3.1.15 Path Coalescing "Clustering"

Path coalescing is a process designed to reduce the total number of memory copies between various devices. Currently, this process is only applied in the case of GPU-assigned expressions, because individual CPU cores are general considered to have uniform access times to shared memory. However, as each GPU maintains its own private physical memory, it is not enough to simply assign GPU nodes to devices in a random, or even evenly distributed fashion.

If we define a *path* through our graph as a series of vertices connect by a single in/out edge to another vertex, then, by construction, each path has a dependency ordering which extends from leaf to root. From this definition, it can be observed that for a given path, no parallelism exists; this must be true due to the fact that each path vertex has only a single in and out edge and our graph is acyclic. Therefore, for any individual path consisting entirely of GPU tasks, it is always optimal to schedule the entire path to a single GPU device.

3.1.16 Reducing Edge Latency

As we have noted, the problem of scheduling tasks has grown into a process more closely related to the scheduling of paths. This is the direct result of introducing nonuniform

communication layers into our model; as it takes longer to pass information from a CPU to an external GPU than it does to pass it from one CPU to another, we are now required to assign edges a cost based on the source and destination hardware targets. However, using the information available to us from the task graph, we can potentially reduce or eliminate edge overhead, in situations where an external task has multiple field dependencies.

In our construction of the generic priority scheduler, we iterate through a list of all vertex consumers, notifying each consumer that one of its resources has become available. If, in addition to notifying a vertex that its dependency has finished executing, we also compare its device target with the dependencies device target, we can identify those fields which are not ready to be consumed and begin prefetching them as necessary. In the case of Figure 3.19, this would mean that we could prefetch data from node's D and E to GPU, while F and C are computed, potentially reducing the start time of B.

Under the assumption that the target device of each node has been fixed across a given iteration, this solution is essentially without cost. In the case of Figure 3.19, where task B will execute on GPU, then we know that at some point D, E, and C must be transferred to GPU. However, since we have the flexibility to determine which of these copies will occur, we require worker threads to attempt to push resources as soon as their data become available.

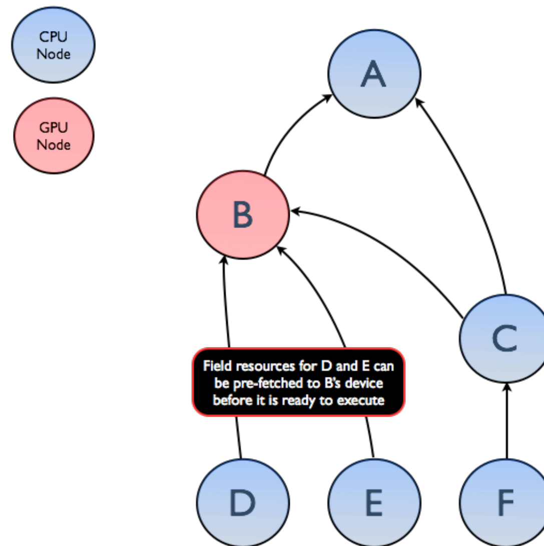

Figure 3.19. Illustration of consumer prefetching.

3.2 The Spatial Fields Library

3.2.1 Overview

The spatial fields library provides a basis for two major abstractions: the creation and storage of objects which represent data fields, and the creation of a generic interface to a set of mathematical operators. The overall goal is to provide a set of objects and operators, which can be used with the c++-based DSL described in section 4.1.1, giving an end user the ability to express operations in a manner which more closely resembles how they would write them on paper.

3.2.2 Concepts and Terminology

- Spatial Field - For our purposes, when attempting to simulate or describe any physical space, we will want to be able to discretize that space and represent each point in the region as some type of variable value; the abstract object representing such a space is called a spatial field. For most of the discussion involving spatial fields, we will assume that this value is a double; however, logically the underlying meaning of these values is not equivalent across all fields.

  For example, to represent temperature in a area, we may describe it as a scalar field of cell centered values; in other cases, where we wish to describe a quantity that is directly associated with our cell geometry, such as heat flux through a boundary, we may wish to use a "face centered" field. These differences can be observed in two dimensions in Figure 3.20.

- Spatial Operator - A method defined to perform some action on one or more spatial fields. This operation may depend not only on the type of operation being performed, but also on the device for which a given field is allocated.

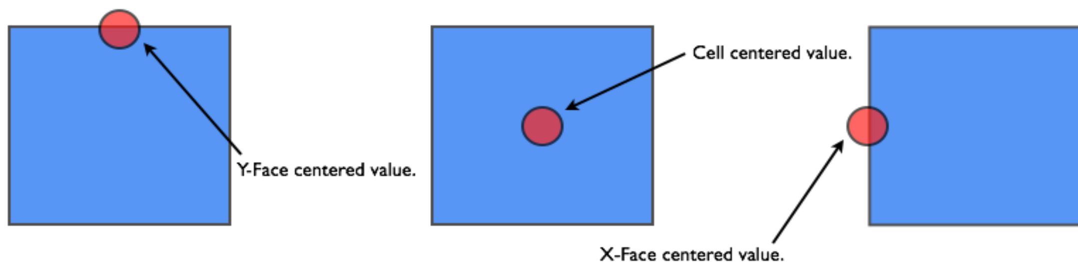

Figure 3.20. Illustration of consumer prefetching.

- Spatial Field Store - A memory pooling structure, which can be used to construct counted pointers to spatial fields. The store will allocate new memory if required, but will give preference to reusing memory which has been previously returned to it.
- Consumer Field - A read only copy of a given spatial field, which is maintained by the base field, and can be used as input to an operator.

3.2.3 Operator Selection

When an operator is selected in a user's model description, that operator will remain generic throughout the process of constructing the expression tree, and in many cases, the explicit implementation may not be known until the expression itself is run by the scheduler. How exactly a given operation is implemented will depend on the execution behavior of and problem and consequentially, on the device types available. This is due to the fact that operator implementation will vary between different devices, and the efficiency of that implementation will often depend on the size of the problem and associated costs of making its dependencies available. In this fashion, we are able to preserve flexibility in choosing the best method, for a given operation, on a specific system.

3.2.4 External Consumers

In addition to the improved functionality of the expression library field managers described in section 4.1.4, a number of extensions were required to the spatial fields themselves in order to provide the flexibility necessary to support allocation on external fields. Outside of the various structural changes involving allocation requirements for external memory and overloads for field assignment, copies, and construction, the most significant addition to the spatial fields library is related to the problem of allowing for multiple consumer fields across devices. It is not desirable to retain spatial field objects for each device on which the field may be consumed; this would introduce unnecessary complexity, create potential coherency problems, and require additional structure to determine which field was the "real" one.

Because of these considerations, we introduced the notion of "consumer fields," or perdevice copies of the primary spatial field, which are maintained by the base spatial field. Consumer fields can be added for any Spatial Field as necessary, for a specific target device, and from that point forward will be available to any computation that takes place on the device. As these fields are meant only for consumption, they are accessible to operators as read-only objects, avoiding coherency issues and more complicated bookkeeping schemes.

Whether or not a specific consumer field is created for a given spatial field is primarily controlled at the scheduler level. During the processing phase of an expressions completion

callback, the scheduler must inform all of that expression's consumers that one of their dependencies is ready for consumption. This poses a potential coherency problem, as any given expression e_c which is a consumer of the recently completed expression e_d, may not be scheduled to execute on the same device on which e_d resides. Therefore, as part of the qualification process to determine if e_c is ready to execute, the scheduler must ensure that e_d exists on e_c's computation target, and if not, allocate a consumer field.

As a simple example, we could imagine e_d being computed as a CPU target and stored in local RAM, but the computation of e_c being set to take place on GPU_1. In this case, the scheduler would note this discrepancy, and inform e_d's spatial field to construct a consumer field on GPU_1, and for the rest of e_d's lifetime, it would have a copies of itself in local RAM and on GPU_1.

CHAPTER 4

RESULTS AND EVALUATION

4.1 Test Cases

Test cases were chosen to demonstrate the core goals of this work: extension of the solver framework to support multiple hardware targets, and to provide a basis for improved flexibility in the scheduling model that is capable of interacting properly with lower level parallel components; specifically, the Spatial Operators librar . Unless otherwise specified, testing was performed on either the Ember, Updraft, or Aurora systems (Table 4.1), with the first two being part of the University of Utah's Center for High-Performance Computing (CHPC), and the third being a GPU research machine.

4.1.1 Scheduler Performance - Task Threaded Operator Interaction

Our first goal is to ensure that the updated scheduler is capable of interacting nicely with parallel components found in the Spatial Operations library. To this end, we can construct an expression set which is well suited to stressing various aspects of the computation framework and ensure that we observe desirable and consistent behavior as we vary the number of available task and operator threads. For this purpose, we will use a scalability test simulating an artificial type of scalar transport. This model allows us to increase task complexity by controlling the number of equations being solved, their interdependence, and associated convective terms. Additionally, the per-operator workload can be arbitrarily increased by enlarging the physical domain on which the problem is being solved.

$$\frac{\partial \Phi_i}{\delta t} = f(\Phi_i) \tag{4.1}$$

$$\frac{\partial \Phi_i}{\delta t} = \prod_{j=1}^{n} exp(\Phi_j) \tag{4.2}$$

An example uncoupled expression with eight equations is defined by Equation 4.1, and can be visualized as Figure 4.1. The uncoupled variant is entirely parallel and should

Table 4.1. Test System Specifications, "Updraft," "Ember," and "Aurora"

	Updraft	*Ember*	*Aurora*
Processors	256 Dual-Quad Core Nodes (2048 total cores) 2.8 GHz Intel Xeon (Harpertown) processors	262 Dual Socket-Six Core Nodes (3144 total cores) 2.8 GHz Intel Xeon (Westmere X5660) processors	Dual Socket-Six Core Intel Xeon E5-2620 2.0 GHz processors
Memory	16 Gbytes memory per node (2 Gbytes per processor core)	24 Gbytes memory per node (2 Gbytes per processor core)	15 Gbytes
Interconnects	Qlogic Infiniband DDR (InfiniPath QLE 7240) interconnect Gigabit Ethernet interconnect	Mellanox QDR Infiniband interconnect Gigabit Ethernet interconnect	N/A

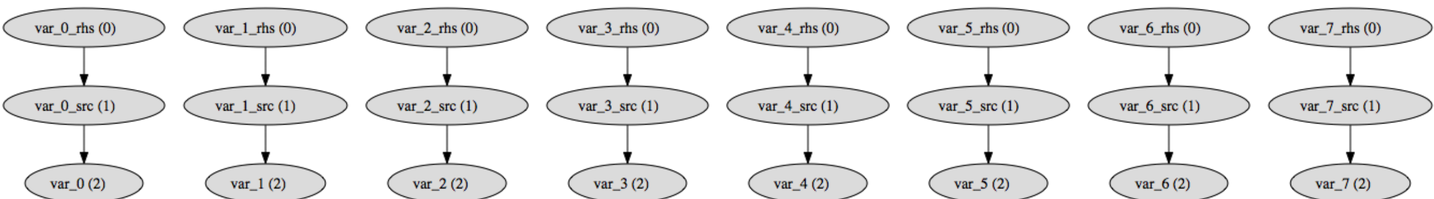

Figure 4.1. Example scalability graph, eight equation, no source coupling

exhibit maximum potential performance gains at the task level. Conversely, equation 4.2 and Figure 4.2 illustrate a fully coupled expression graph, with reduced opportunity for task parallelism.

Solving Equation 4.1 with n=16 on a single ember node, we obtain the following:

Examining the results in Figure 4.3, with each curve illustrating scaling performance for an increasing operator thread count and a fixed number of task parallel threads, we can see that after reaching a certain level of operator parallelism, it becomes beneficial to go task parallel. While these results are inherently desirable, as they prove out the scheduler's ability to introduce task parallelism without significant overhead, they are also intuitively satisfactory, in that we would not expect task parallelism to improve performance without a sufficiently sized operator thread pool. Figure 4.4 provides an alternate view of view of the same data, but with each curve representing a increasing task parallel thread count and a fixed operator thread count.

4.1.2 Scheduler Performance - Task Threaded MPI-Process Interaction

In addition to operator level parallelism, we also have another dimension of "process" parallelism, which can be explored via MPI. As previously noted, the Expressions framework is often utilized in conjunction with Wasatch, which is capable of performing domain decomposition, and assigning work to multiple processes which communicate via MPI message passing. While the utility of multiprocess coordination is obvious when utilizing multiple compute nodes, it is also worth noting that it can provide worthwhile performance benefit when running on a single node.

Figure 4.5 shows individual scaling for process (MPI), task, and operator components taken in isolation, and we see that a maximum speedup of 8.4 is attained with twelve MPI processes. At first glance, this would seem to indicate that it is better to simply go process parallel and leave it at that; however, as we have already seen in Figure 4.3, task and operator parallelism are complementary, with task parallelism improving performance only after a specific level of operator parallelism has been reached; the same is true of process level parallelism.

If we allocate twelve threads to both task and operator parallelism and run our test again, we are able to see improved behavior over any individual method (Figure 4.6). Not only do we see a performance increase over any individual scaling component, but we achieve a maximum speedup of $\tilde{1}2$, which is our effective maximum potential speedup on the 12 core (2x6) Aurora system. One explanation for this improvement over the single process

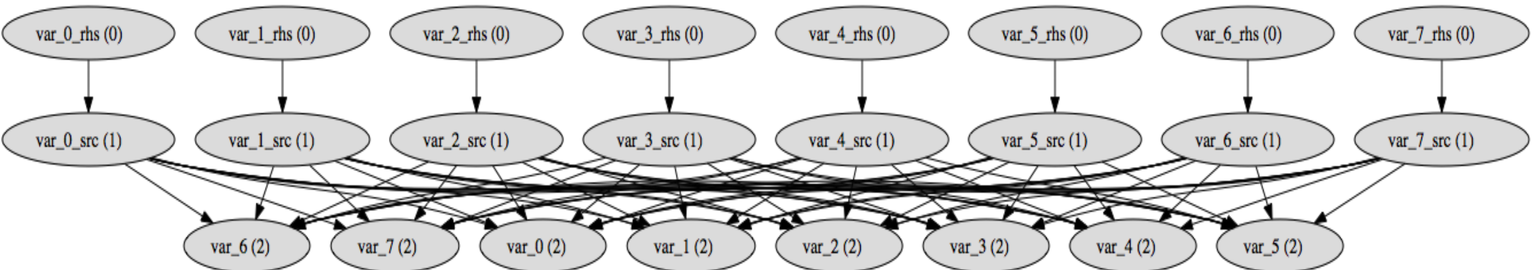

Figure 4.2. Example scalability graph, eight equation, source coupling

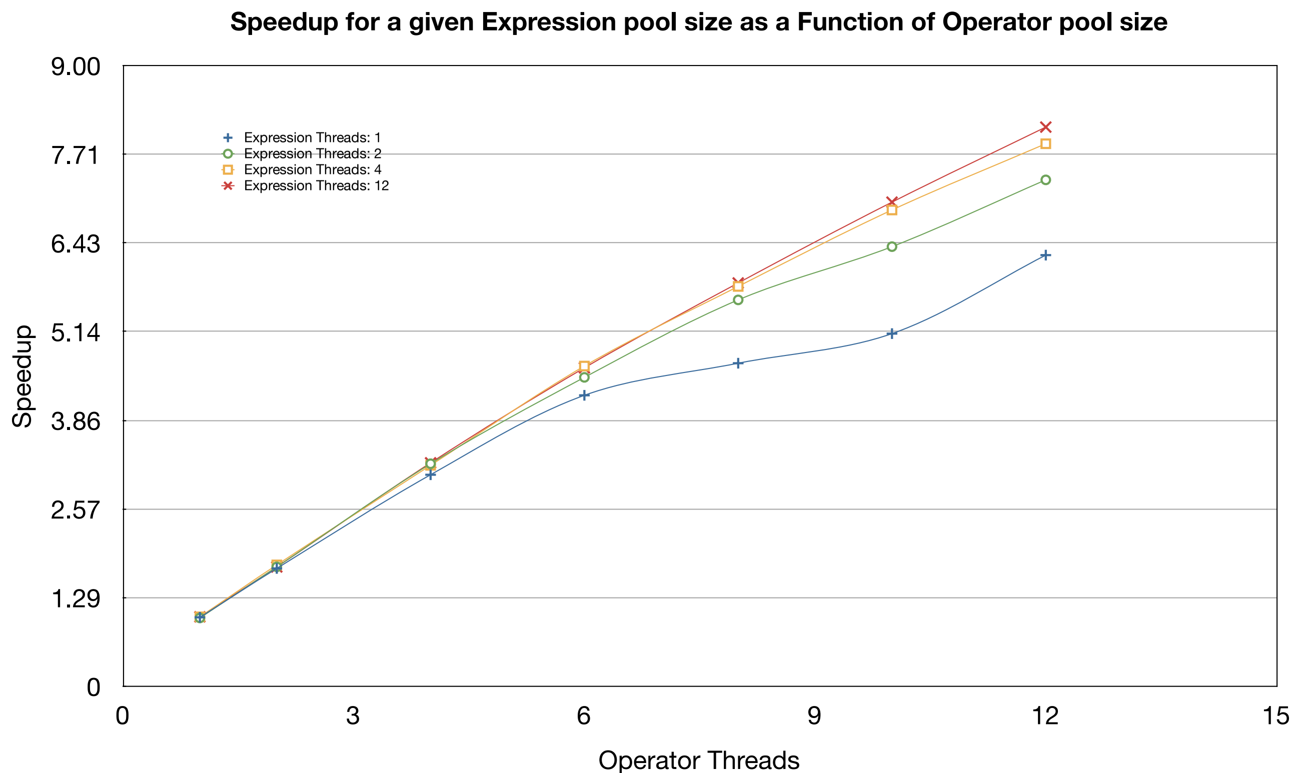

Figure 4.3. Scalability test, ember 2012, plotted as a function of operator threads - 256x128x128, 16 variables

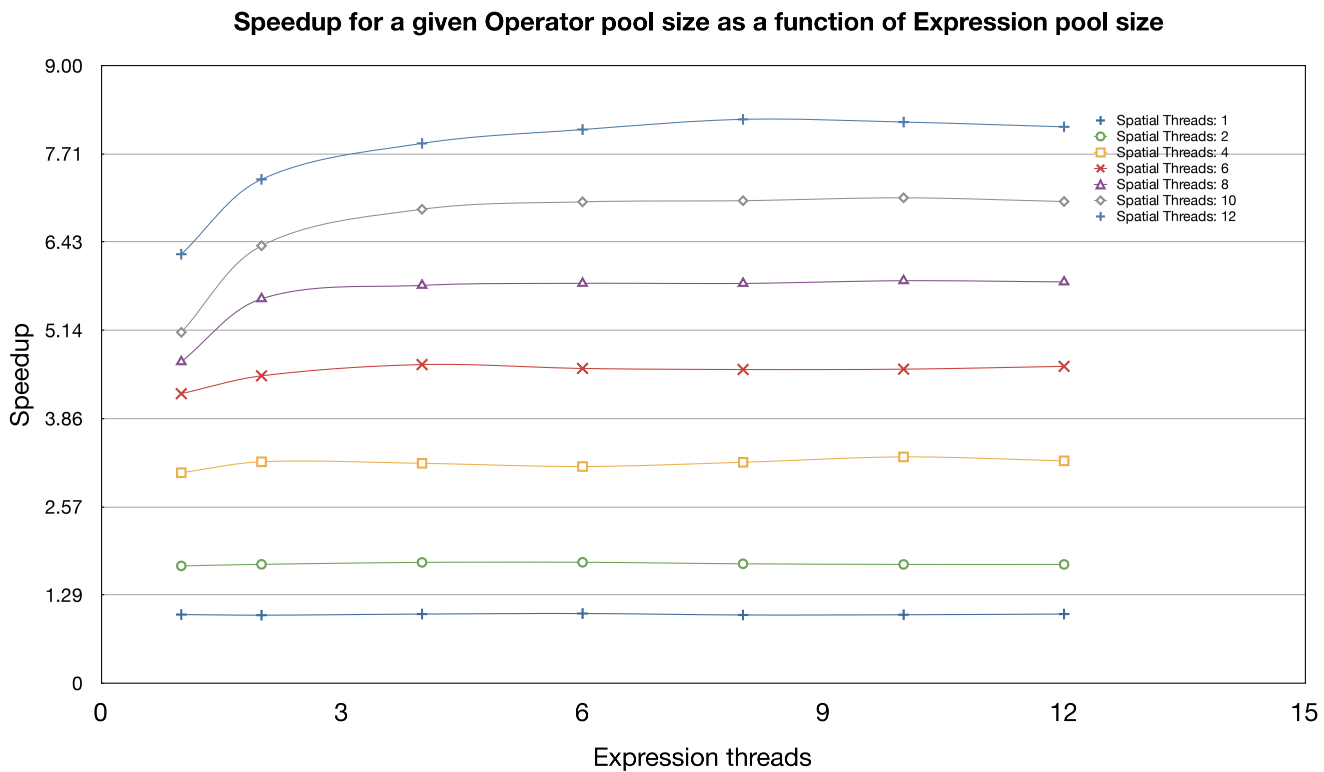

Figure 4.4. Scalability test, ember 2012, plotted as a function of expression threads - 256x128x128, 16 variables

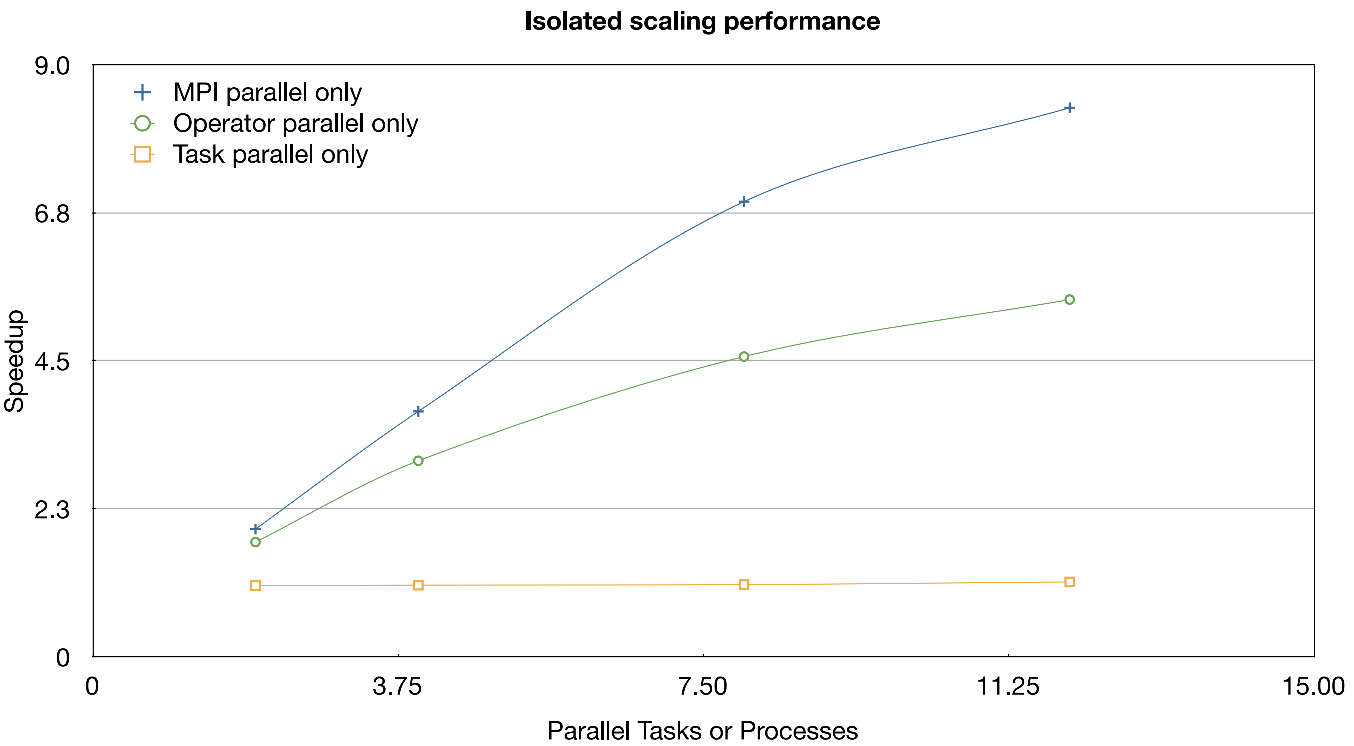

Figure 4.5. Single node isolated scaling for process, task, and operator parallelism, aurora 2012 - 120^3, 24 variables

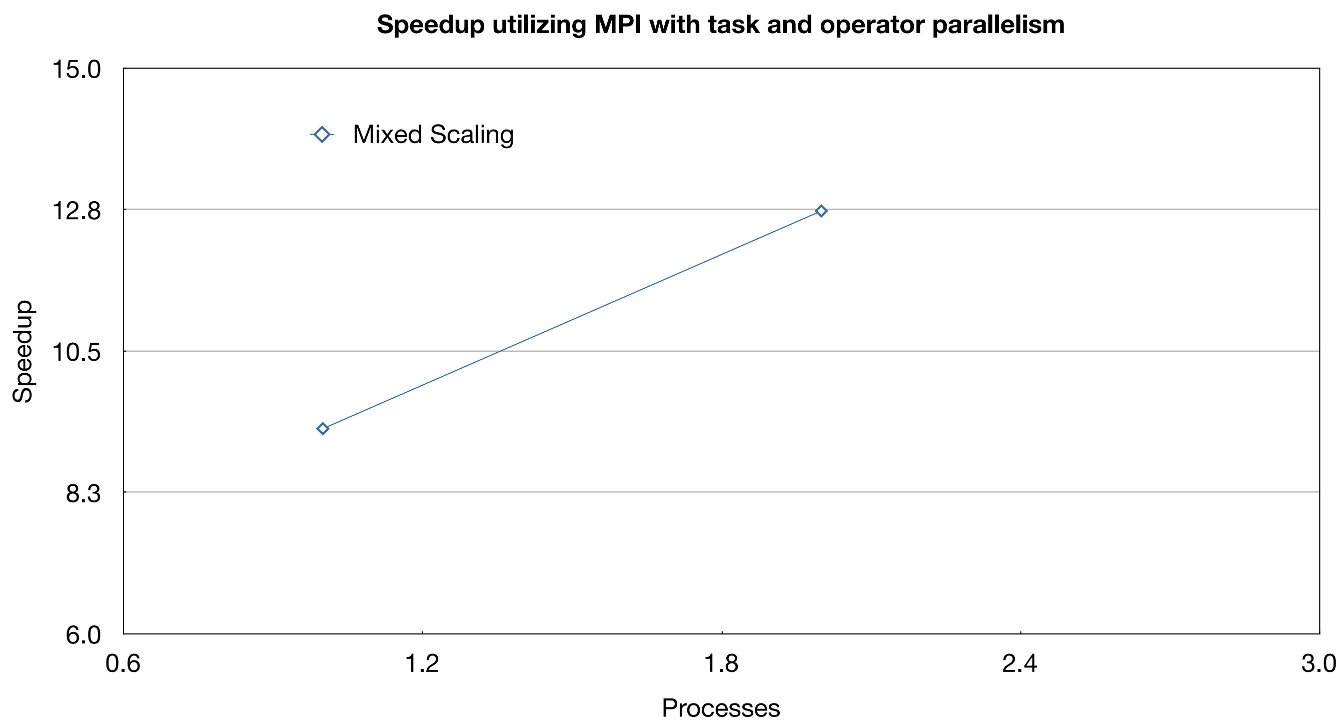

Figure 4.6. Single node MPI scaling with 12 task and 12 operator threads, aurora 2012 - 120^3, 24 variables

case is the way the MPI configuration is set up to pin process threads to a specific socket, whereas in the single process case, we may experience undesirable caching behavior with collaborative threads executing on different physical processors or potentially very different sections of the physical domain.

4.1.3 Scheduler Performance - Task Threaded MPI-Scaling: Multinode

After examining scheduler performance in the single node case with varied process, task, and operator level parallelism and finding them to conform to our expectation, it seems reasonable to look at the multinode, task parallel case. For this, we will run a scaling study on the Ember system in which each 12 core node is running a single process with 12 threads allocated at the operator level. We will then examine scaling performance from 12 to 384 cores while varying each process task scheduler to utilize 1, 2, 4, 8, and 12 threads. In doing so, we expect to see a trend similar to that observed in Figure 4.4, with efficiency increase uniformly irrespective of process count.

Running our simulation problem again, using the constraints described above, we observe the results in Figure 4.7, exactly as expected. This further supports the assertion that our modified scheduler implementation has produced a measurable gain in efficiency, without introducing performance regressions.

4.1.4 Hybrid Scheduler Feasibility - Stencil 2 Performance

At the time of this writing, only the stencil-2 operation had a GPU implementation within the Spatial Ops library, so this discussion will be restricted to direct comparison of the CPU vs GPU operator implementations and graphs utilizing them. In general, a stencil operator can be thought of as an operation that computes some value on each point of some n-dimensional space based on the value(s) of neighboring points. In the case of the stencil-2 operation we will examine and the simple heat equation using it, each new point-wise value is computed using two local points. Examples for two-point X, Y, and Z stencils can be observed in Figure 4.8.

Running a modified version of the simple heat equation, on one of ember's GPU nodes, which computes X, Y, and Z flux elements using our stencil-2 operator, we can obtain the results seen in Figure 4.9. On first inspection, it appears that there is little to be gained from scheduling tasks to GPU; however, much like our previously discussion of the complementary nature of parallel performance components, there are a number of additional pieces of information which must be taken into account.

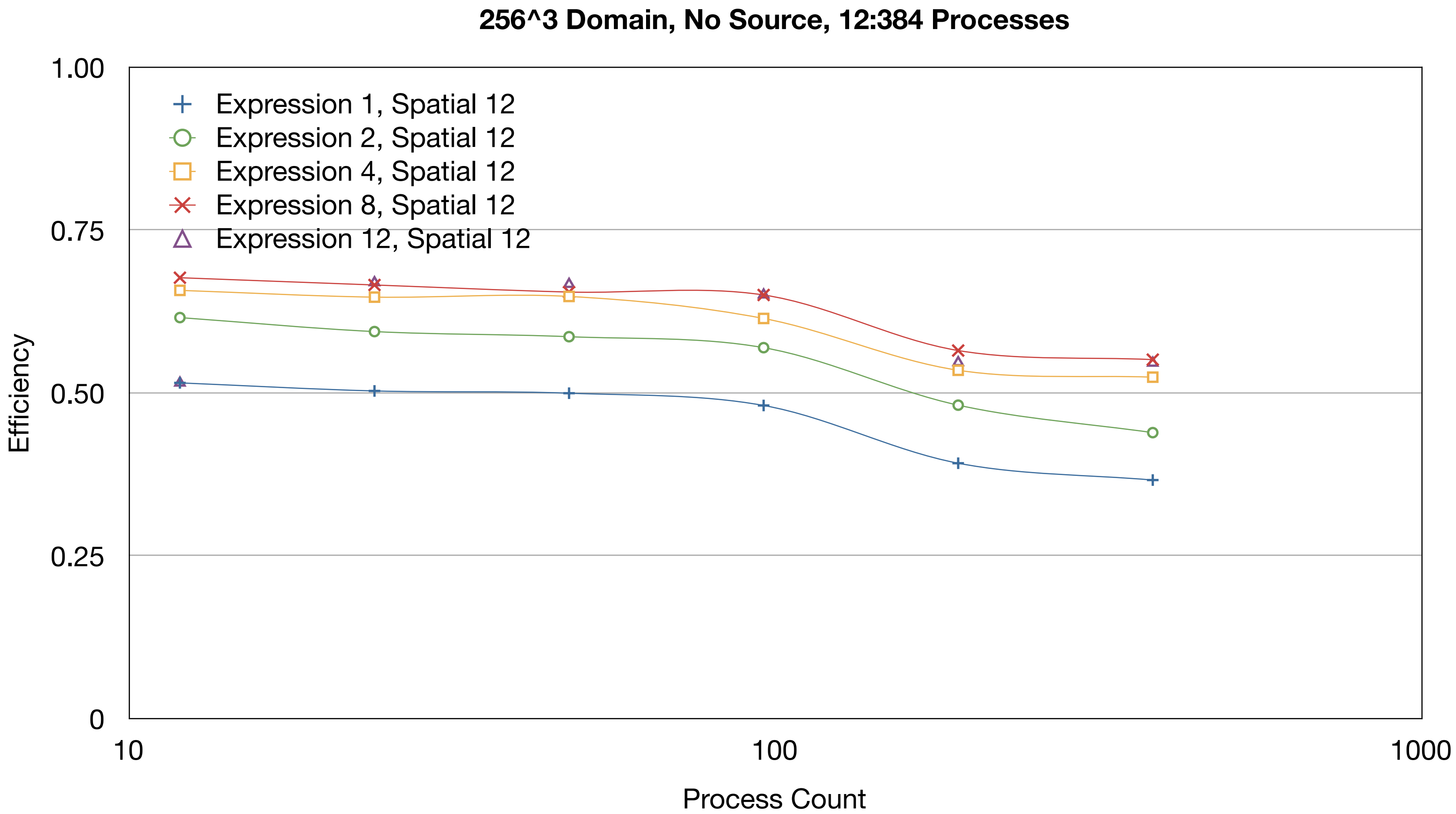

Figure 4.7. Scalability test 2012 - 256^3, 16 variables

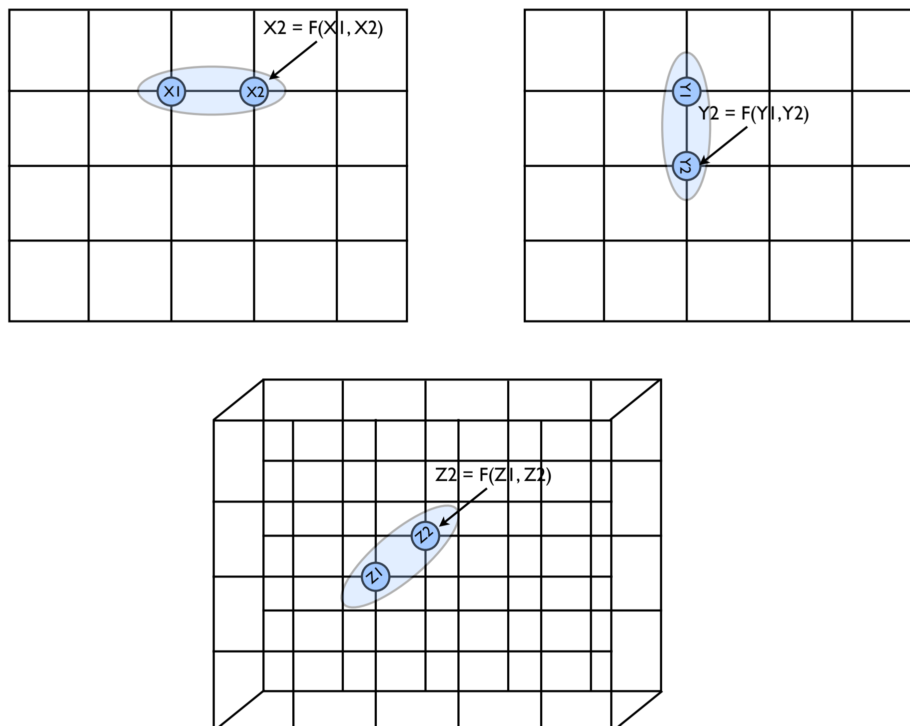

Figure 4.8. Example stencil-2 computation, 2012

Our first point of interest is that, while GPU does not drastically improve our compute time over the CPU, it does in fact solve it in an equivalent amount of time. This is important, because in any situation where we are actually attempting to compute a problem's solution, we would not restrict ourselves to a single hardware component when multiple are available. Rather, even in the simplest case, we could envision splitting a problem in half, and running one piece on the GPU and one on the CPU. Thus, we are now able to utilize an additional piece of computing hardware on an individual node; which, in the case of the stencil-2 operator, has the potential to effectively double our computing power on that node.

The next important consideration presents itself when we examine the node level performance differences between CPU and GPU components. If, instead of examining total compute time for the entire graph, we look at the time required to actually perform the stencil-2 computation, seen in Figure 4.10, the situation appears to improve greatly and it becomes apparent that we still have significant potential for performance gains. For each test case, ranging from 32,000, to over 16 million grid points, the time taken to perform the stencil-2 computation is often an order of magnitude faster on the GPU than the CPU. Looking back to Chapter 3, it is easier to see how this behavior can provide significant

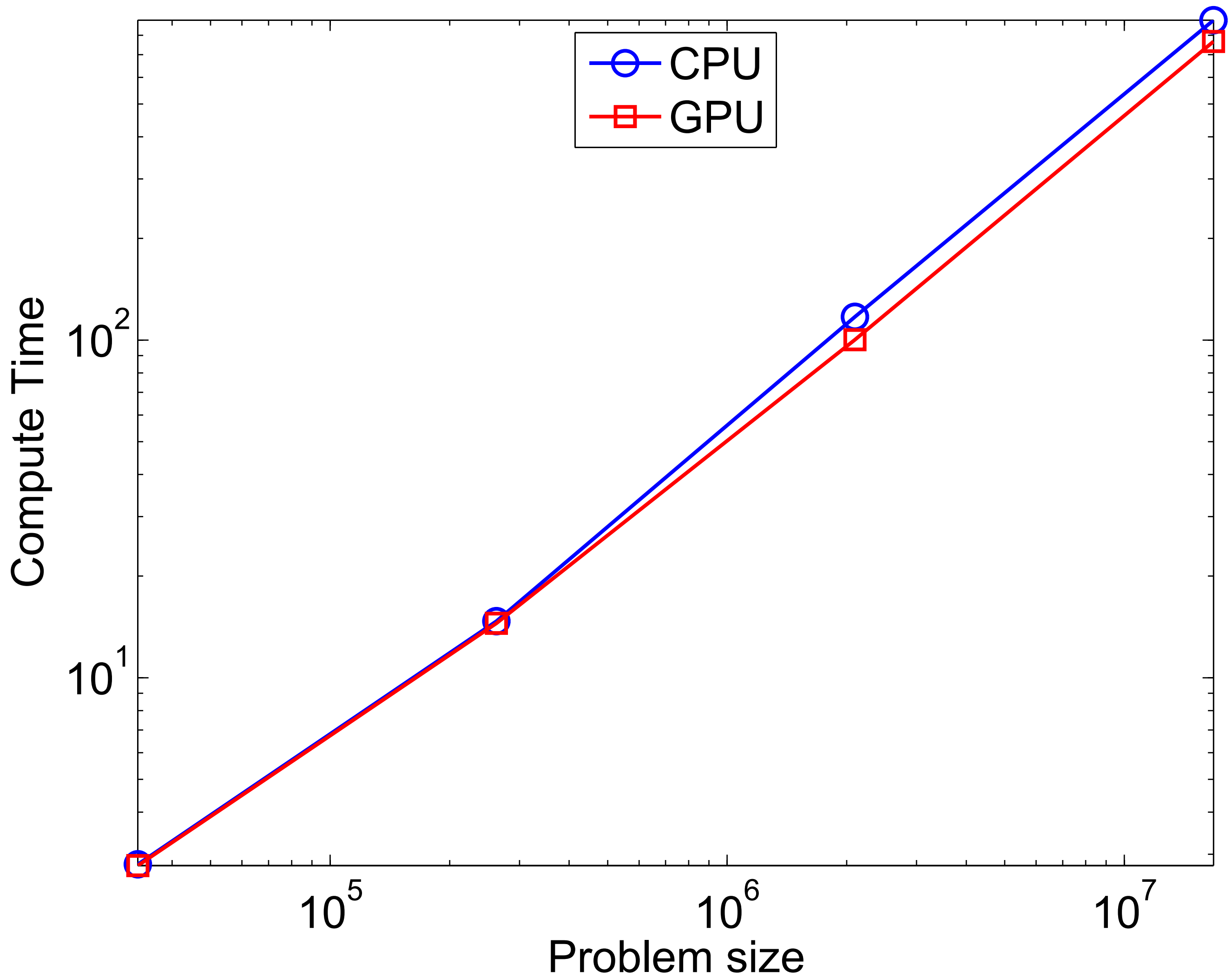

Figure 4.9. CPU vs GPU scaling for gradient spatial operator - total graph time, 2012

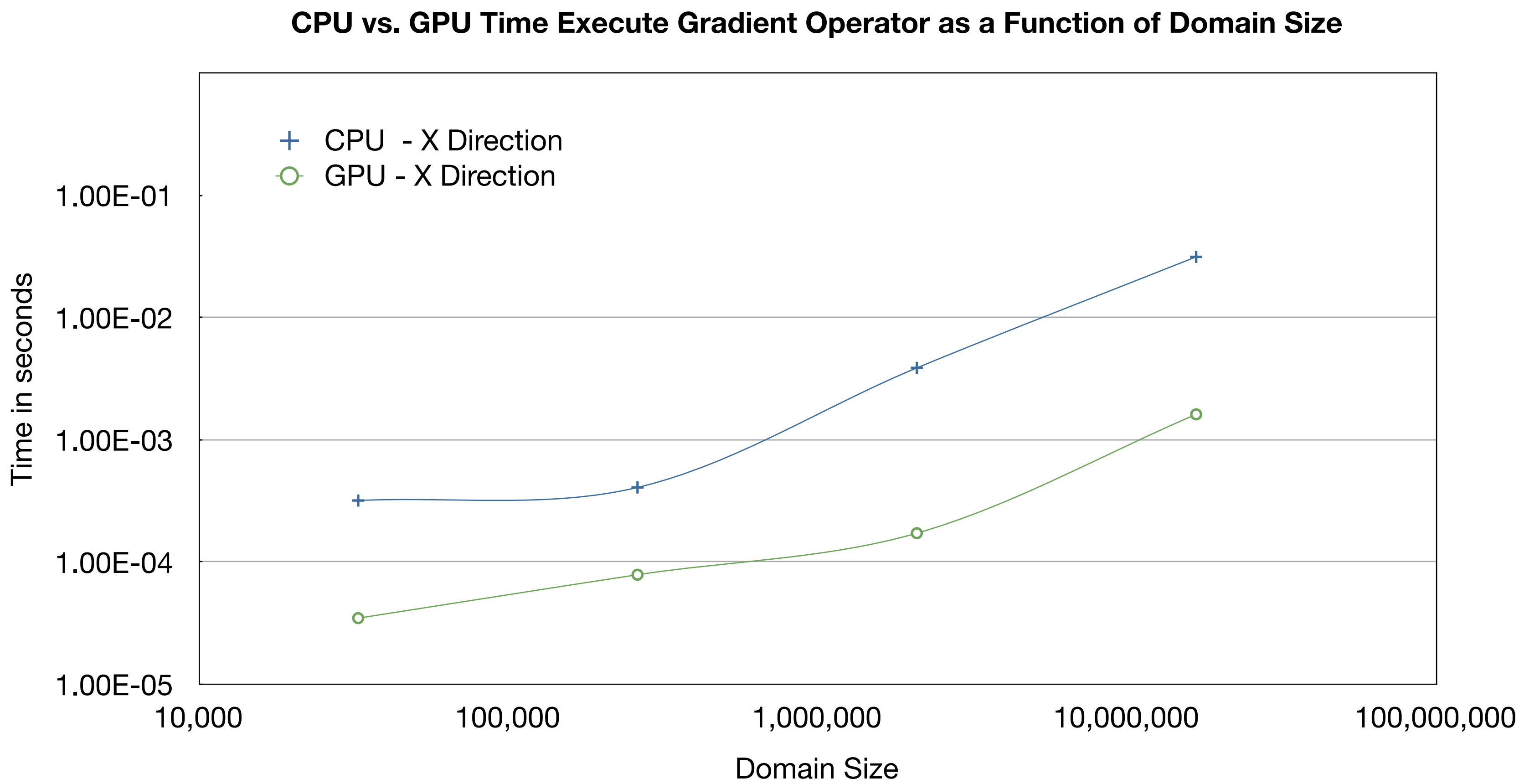

Figure 4.10. CPU vs GPU scaling for gradient spatial operator - single operator time, 2012

motivation for path coalescing and data prefetching.

4.2 Conclusion

4.2.1 Work Summary

This paper has presented a body of work supporting the extension of the Expressions framework to utilize existing and future acceleration technologies, without requiring underlying hardware knowledge from its users. This, in turn, consisted of four primary component requirements:

- Flexible task scheduling and algorithm selection
- Delayed memory allocation
- Support for coherent Spatial Field consumption on multiple target devices.
- Transparent operator selection based on field allocation source.

The contributed work required to realize these design goals can be summarized as follows:

- Spatial Operators Library
  - Modified field managers to support dynamic memory allocation/deallocation.
  - Modified spatial fields to be device aware.
  - Modified spatial fields to support read-only consumer fields for simultaneous and coherent consumption on multiple devices.
  - Modified stencil-2 operator to support GPU targets and transparent implementation selection based on field allocation source.
  - Modified of spatial field pools to support GPU pools.
  - Introduced CUDA device interface layer as a sublibrary for interacting with and managing CUDA devices.
- Expressions Library
  - Thread pool improvements.
  - Implemented new interface-based scheduling framework.
  - Implementation of a priority task scheduler, supporting:
    - Task graph introspection.

    - Thread resource management.
    - Dynamic memory resource allocation.
    - Thread safe task scheduling across multiple schedulers.
    - Dynamic node prioritization.
  - Implementation of a hybrid task scheduler, supporting:
    - Dynamic device assignment.
    - Device aware prioritization.
    - Data prefetching via consumer field assignment.
    - Multinode path coalescing for GPU targets.

4.2.2 Future Work

A number of major avenues for continued and future work exist within the context of this development effort.

- Spatial operator implementation – As noted in the discussion of the hybrid-scheduler, many of the existing operator components within the Spatial Operator library are not GPU aware and do not have existing implementations. Given the many design challenges and opportunities for performance gains when converting from a serial/CPU-parallel operator implementation to a massively parallel CUDA style implementation, there is significant room here for experimentation. Additionally, while not explored here, there are a number of potential research opportunities for intelligent GPU operator coalescing.

- Determination of device targets in a more natural way – Under the assumption that operator implementation is ubiquitous, determining how to best assign a given graph node to a hardware target is not explicitly obvious. This is because the most natural methods for deciding on a hardware target would directly compare execution timings for each operator against both CPU and GPU implementations; as we have neither at the start of a simulation run and these timings are device and system dependent, we likely must either perform an initial gathering phase to obtain timings for each expression node, or heuristically tune our device assignment as a simulation progresses. There are many potential approaches to collect these assignment metrics and in turn use them to produce more intelligent scheduling agents.

- Finer grained introspection of structural elements within the task graph – In addition to the scheduling concepts described above, it may be the case that for certain graphs

there is a high degree of task parallelism, punctuated by serializing computations that should be handled dynamically; for example, see Figure 4.11. These situations are unlikely to be reflected in any type of generalized parallelism score for the entire graph, and instead will require us to define a type of finer grained parallelism, which essentially reduces to a problem of identifying bottlenecks within a graph.

Under the assumption that we are able to identify nodes acting as bottlenecks within the graph, then it would be reasonable to assume that the task scheduler could take note when pushing such a node to the task queue. Given knowledge of a serializing task, we could then increase threading resources available to the operator pool for some duration, before returning to a more even distribution. However, this process would not be without pitfalls, and could potentially decrease performance in situations where serialization nodes did not require significant computation.

4.2.3 Final Thoughts

This paper has served to demonstrate the successful realization of our specified design goals and to provide an accurate description of the development processes required. The expressions framework is now explicitly able to support multiple hardware targets, in a manner that should serve as an extensible template capable of accomodating compute devices of the future. It is also able to actively take advantage of the graph structure assembled from a user's expression implementations and dependency specifications, while dynamically allocating and deallocating task resources. The hope is that this work will serve to facilitate improved performance an flexibility in the future, while preserving the existing transparent software development process of the expressions library.

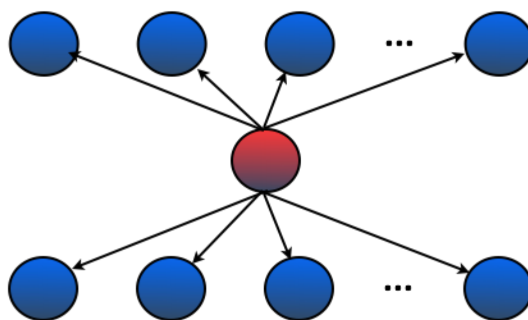

Figure 4.11. Highly parallel task graph structure with punctuated serialization.

APPENDIX

SCHEDULER CODE

A.1 Scheduler Base Class

```
//DEBUG FLAGS: DEBUG_NO_FIELD_RELEASE ( default undefined )
/*
 * Contract implementation for a task scheduler
 *
 */
#ifndef Expr_TaskSchedulerBase_h
#define Expr_TaskSchedulerBase_h

//Standard libraries
#include <map>

//Shared pointers
#include <boost/shared_ptr.hpp>

//Expressions
#include <expression/FieldDeps.h>
#include <expression/ExpressionID.h>
#include <expression/FieldManagerList.h>
#include <expression/VertexProperty.h>

#include <spatialops/structured/MemoryTypes.h>

#include <spatialops/SpatialOpsConfigure.h>  // defines thread stuff.
#ifdef ENABLE_THREADS
namespace BIP = boost::interprocess;
# include <boost/interprocess/sync/interprocess_semaphore.hpp>
# include <spatialops/ThreadPool.h>
#endif

namespace Expr {
/**
 *  \class Scheduler
 */
namespace SchedulerGraphTypes {
typedef boost::adjacency_list<boost::listS, boost::listS, boost::directedS,
    VertexProperty, boost::no_property> DefaultType;
}

class Scheduler {
  public:
    Scheduler() :
      invalid_(true)
#     ifdef ENABLE_THREADS
      , pool_ ( SpatialOps::ThreadPool::self() )
      , poolx_( SpatialOps::ThreadPoolFIFO::self() )
      , schedBarrier_(0)
#     endif
    {}

    virtual ~Scheduler() {}

    //------------------ Interface requirements ------------------

    virtual Scheduler* get_base_pointer() = 0;

    /** \brief Perform any required setup action and pre-processing */
    virtual void setup(bool hasRegisteredFields) = 0;

    /** Invalidate the current schedule **/
    virtual void invalidate(){ invalid_ = true; }

    /** \brief Execute the supplied task graph */
    virtual void run() = 0;

    /** \brief Perform any cleanup or post processing */
    virtual void finish() = 0;

```

```
68      /** \brief Process 'finished' method from a vertex element */
69      virtual void exec_callback_handler(void*) = 0;
70
71      /** \brief Return a string identifying the scheduler in use */
72
73      virtual const std::string get_identity() = 0;
74
75      /**
76       * \brief Assign a field manager list to the scheduler
77       */
78      virtual void set_fml(FieldManagerList* fml) {
79        this->fml_ = fml;
80      }
81
82      /**
83       * \brief Store a copy of the field dependencies for this grpah
84       **/
85      virtual void set_fdm(
86          std::map<ExpressionID, boost::shared_ptr<FieldDeps> >* fdm) {
87        this->fdm_ = fdm;
88      }
89
90    protected:
91      FieldManagerList* fml_;
92      std::map<ExpressionID, boost::shared_ptr<FieldDeps> >* fdm_;
93
94      bool invalid_;
95
96  #   ifdef ENABLE_THREADS
97      SpatialOps::ThreadPool& pool_;
98      SpatialOps::ThreadPoolFIFO& poolx_;
99      BIP::interprocess_semaphore schedBarrier_;
100 #   endif
101 };
102
103 } // namespace Expr
104
105 #endif // Expr_TaskScheduler_h
```

A.2 Priority Scheduler

```
1   template<class T = SchedulerGraphTypes::DefaultType>
2   class PriorityScheduler: public Scheduler {
3
4   public:
5
6       PriorityScheduler(T* graph) :
7           execGraph_(graph), taskGraph_(NULL), Scheduler() {
8       }
9
10      ~PriorityScheduler() {
11        if (taskGraph_ != NULL) {
12          delete taskGraph_;
13        }
14      }
15
16      /**
17       * \brief Return this scheduler as its base type
18       */
19      Scheduler* get_base_pointer() {
20        return dynamic_cast<Scheduler*>(this);
21      }
22
23      /**
24       * \brief after this function runs, our scheduler should be in a runnable state
25       */
26      void setup(bool hasRegisteredFields = false);
27
28      /**
29       * \brief begin executing graph nodes
30       */
31      void run();
32
33      /**
34       * \brief perform any cleanup activities
35       */
36      void finish();
37
38      /**
39       * \brief this function is called by an expression when it has finished executing
40       * we do introspection and determine which nodes are ready to run from here.
41       */
42      void exec_callback_handler(void*);
43
44      /**
45       * \brief return a string identifying which scheduler we are.
46       */
47      const std::string get_identity() {
```

```
48        return std::string("Default␣Priority␣Scheduler");
49      }
50
51      /**
52       * \brief intermediary for executing a node when it is ready, used so that we can control
53       * when we bind memory to each field
54       */
55      void call(VertexProperty& target);
56
57      /**
58       * \brief change our task graph, note this invalidates the graph and will force a full run through
59       * setup the next time it is called.
60       */
61      void set_task_graph(T* graph);
62
63    protected:
64
65      typedef typename boost::graph_traits<T>::vertex_descriptor Vertex;
66      typedef std::vector<Vertex> VertList;
67      typedef typename VertList::iterator RootIter;
68      typedef typename std::map<ExpressionID, Vertex> ID2VP;
69
70      typedef typename boost::graph_traits<T>::edge_descriptor Edge; ///< Edge in a graph
71      typedef typename boost::graph_traits<T>::edge_iterator EdgeIter; ///< Edge iterator
72      typedef typename boost::graph_traits<T>::vertex_iterator VertIter;
73      typedef typename boost::graph_traits<T>::out_edge_iterator OutEdgeIter;
74
75      T* execGraph_;
76      T* taskGraph_;
77      int nelements_;
78      int nremaining_;
79
80      VertList rootList_;
81
82      //Producer and Consumer vertex maps
83      ID2VP execVertexMap_;
84      ID2VP taskVertexMap_;
85
86      /**
87       * \brief Boost visitor structure for setting node priorities
88       */
89      struct ExecPriorityVisitor: public boost::default_bfs_visitor {
90          ExecPriorityVisitor() {
91          }
92          inline void examine_edge(Edge e, const T& g) {
93            Vertex src = boost::source(e, g);
94            Vertex dest = boost::target(e, g);
95            const int srcPriority = g[src].priority;
96            T& g2 = const_cast<T&>(g);
97            int& priority = g2[dest].priority;
98            priority = std::max(priority, srcPriority + 1);
99          }
100     };
101
102     /**
103      *  \class ExecMutex
104      *  \brief Scoped lock. An instance should be constructed within any function that touches Scheduler
105      *  member variables.
106      */
107     class ExecMutex {
108 #   ifdef ENABLE_THREADS
109         const boost::mutex::scoped_lock lock;
110         inline boost::mutex& get_mutex() const {static boost::mutex m; return m;}
111
112         public:
113         ExecMutex() : lock( get_mutex() ) {}
114         ~ExecMutex() {}
115 #   else
116       public:
117         ExecMutex() {
118         }
119         ~ExecMutex() {
120         }
121 #   endif
122     };
123
124     void dec_remaining();
125 };
126
127 /***************** Begin PriorityScheduler Implementation *****************/
128 template<class T>
129 void PriorityScheduler<T>::set_task_graph(T* graph) {
130   invalid_ = true;
131   this->execGraph_ = graph;
132 }
133
134 template<class T>
135 void PriorityScheduler<T>::call(VertexProperty& target) {
136   (target.expr)->base_bind_fields(*this->fml_);
137   target.execute_expression();
138 }
139
```

```
140 template<class T>
141 void PriorityScheduler<T>::exec_callback_handler(void* expr_vertex) {
142   T& gptr = *this->execGraph_;
143
144   Vertex v = (Vertex) expr_vertex;
145   VertexProperty& vpJustFinished = gptr[v];
146
147   // Notify the vertex that a consumer has finished, it returns true, it can be freed.
148   for( std::vector<VertexProperty*>::iterator vpit = vpJustFinished.ancestorList.begin();
149        vpit != vpJustFinished.ancestorList.end();
150        ++vpit )
151   {
152     if ((*vpit)->consumer_finished()) {
153       if ( this->fdm_ != NULL && this->fml_ != NULL ) {
154         FieldDeps& fd = *((*this->fdm_)[(*vpit)->id]);
155
156 #ifndef DEBUG_NO_FIELD_RELEASE
157         bool result = fd.release_fields(*this->fml_);
158 #endif
159       }
160     }
161   }
162
163   // Notify the vertex that an ancestor has finished, it returns true if it is ready.
164   // If it is, we either toss it to the thread pool or run it.
165   for( std::vector<VertexProperty*>::iterator vpit = vpJustFinished.consumerList.begin();
166        vpit != vpJustFinished.consumerList.end();
167        ++vpit )
168   {
169     if ((*vpit)->ancestor_finished()) {
170 #ifdef ENABLE_THREADS
171       this->pool_.schedule(
172           boost::threadpool::prio_task_func( (*vpit)->priority,
173                                              boost::bind( &PriorityScheduler<T>::call, this, **vpit ) ) );
174 #else
175       this->call(**vpit);
176 #endif
177     }
178   }
179   dec_remaining();
180 }
181
182 template<class T>
183 void PriorityScheduler<T>::dec_remaining() {
184 #   ifdef ENABLE_THREADS
185   typename PriorityScheduler<T>::ExecMutex lock;
186 #   endif
187   --nremaining_;
188
189 #   ifdef ENABLE_THREADS
190   if( nremaining_ == 0 ) {
191     this->schedBarrier_.post();
192   }
193 #   endif
194 }
195
196 /**
197  * \brief Initialize any data structures and information required for 'run()' to complete
198  */
199 template<class T>
200 void PriorityScheduler<T>::setup(bool hasRegisteredFields) {
201   //If we have been invalidated
202   // - reset all callback handles in the graph
203   // - recalculate all nparent_ counts for each vertex
204
205   //Notes on whats going on here
206   // - gptr is the execution graph
207   // - tgptr is the consumer ( dependency graph )
208   //
209   // Both are used to build up information required for determining node priority,
210   // node consumers, and node execution requirements.
211   //
212   // Step 1: reset and reconnect all variables to place the graph into state which is execute ready.
213   //
214   // Step 2: inspect the dependency graph in order to determine each nodes execution priority and
215   //   determine its consumer count. ( The number of nodes that consume an expression and its fields ).
216   //
217   // Step 3: inspect the execution graph to determine the number of parent nodes for each expression;
218   //   during execution this will allow us to know when an expression is ready to run.
219   //
220   // Step 4: determine each node's memory constraints, currently this is limited to deciding if the
221   //   expression can use dynamic memory.
222
223   //fprintf(stderr, "Scheduler->setup called\n");
224   //Quick return if we're already valid
225   if (!invalid_)
226     return;
227
228   //Variable init/clearing and setup **/
229   T& gptr = *this->execGraph_;
230
231   if (this->taskGraph_ == NULL) {
```

```
232      delete this->taskGraph_;
233    }
234    this->taskGraph_ = new T();
235    T& tgptr = *this->taskGraph_;
236
237    rootList_.clear();
238    execVertexMap_.clear();
239    taskVertexMap_.clear();
240
241    //Update element counts
242    nelements_ = boost::num_vertices(*this->execGraph_);
243    nremaining_ = nelements_;
244
245    const std::pair<VertIter, VertIter> execGraphVertices = boost::vertices(gptr);
246
247    // ------------------------------ **/
248
249    // Step 1
250    // Reconnect all signals and reset execution counts
251    VertIter iter;
252    for (iter = execGraphVertices.first; iter != execGraphVertices.second; ++iter) {
253      VertexProperty& vp = gptr[*iter];
254
255      execVertexMap_.insert(std::make_pair(gptr[*iter].id, *iter));
256
257      vp.self_ = (void*) (*iter);
258      vp.nparents = 0;
259      vp.nconsumers = 0;
260      vp.priority = 0;
261      vp.execSignalCallback.reset(new VertexProperty::Signal());
262      vp.execSignalCallback->connect(
263          boost::bind(&PriorityScheduler::exec_callback_handler, this, vp.self_));
264
265      vp.ancestorList.clear();
266      vp.consumerList.clear();
267
268      vp.set_is_edge(false);
269    }
270
271    //set node information
272    for (VertIter vit = execGraphVertices.first; vit != execGraphVertices.second; ++vit) {
273      VertexProperty& evp = (gptr)[*vit];
274      execVertexMap_.insert(std::make_pair(gptr[*vit].id, *vit));
275
276      std::pair<OutEdgeIter, OutEdgeIter> edges = boost::out_edges(*vit, gptr);
277      for (OutEdgeIter eit = edges.first; eit != edges.second; ++eit) {
278        VertexProperty& tvp = (gptr)[boost::target(*eit, gptr)];
279
280        evp.consumerList.push_back(&tvp);
281        tvp.ancestorList.push_back(&evp);
282        (evp.nconsumers)++;
283        (tvp.nparents)++;
284      }
285    }
286
287    //Step 2
288    //*** top down priority scheduling ***//
289    boost::transpose_graph(gptr, tgptr,
290        boost::vertex_index_map(boost::get(&VertexProperty::index, gptr)));
291
292    //bfs from each top down 'root'
293    //Since this is the dependence graph, root nodes are at the 'top' and will have no consumers
294    //Since no edge nodes can be 'scratch' we find root nodes in the consumer graph and use exprIDs
295    //to flag them as persistent in the execution graph.
296    const std::pair<VertIter, VertIter> taskGraphVertices = boost::vertices(tgptr);
297
298    for (VertIter vit = taskGraphVertices.first; vit != taskGraphVertices.second; ++vit) {
299      VertexProperty& vp = (tgptr)[*vit];
300
301      taskVertexMap_.insert(std::make_pair(gptr[*vit].id, *vit));
302      if (vp.nconsumers == 0) {
303        boost::breadth_first_search(
304            tgptr,
305            *vit,
306            boost::color_map(boost::get(&VertexProperty::color, tgptr)).visitor(
307                ExecPriorityVisitor()));
308
309      }
310    }
311    //Copy priorities back to the execution graph
312
313    for (VertIter vit = taskGraphVertices.first; vit != taskGraphVertices.second; ++vit) {
314      Vertex v = execVertexMap_[tgptr[*vit].id];
315      gptr[v].priority = (tgptr)[*vit].priority;
316    }
317
318    //*** end of top down scheduling ***///
319
320    //Step 3
321    //grab the root list, default remaining count to parent count
322    //assign scratch values
323    for (VertIter iter = execGraphVertices.first; iter != execGraphVertices.second; ++iter) {
```

```
324     VertexProperty& vp = gptr[*iter];
325
326     //For the execution graph nodes at the bottom of the tree are roots and have no parents.
327     //Since edge nodes cannot be 'dynamic' we flag these nodes as persistent
328     if (vp.nparents == 0) {
329       vp.set_is_edge(true);
330       rootList_.push_back(*iter);
331     }
332
333     if (vp.nconsumers == 0) {
334       vp.set_is_edge(true);
335     }
336
337     if (vp.get_is_edge()) {
338       vp.set_is_persistent(true);
339     } else { // jcs will this over-ride someone locking a field?
340       vp.set_is_persistent(false);
341     }
342
343     if( vp.get_is_persistent() ){
344       vp.mm = MEM_EXTERNAL;
345     } else {
346       vp.mm = MEM_DYNAMIC;
347     }
348
349     vp.nremaining = vp.nparents;
350     vp.ncremaining = vp.nconsumers;
351   }
352
353   invalid_ = false;
354 }
355
356 /**
357  * \brief Begin executing on the graph by loading root expressions onto the queue
358  */
359 template<class T>
360 void PriorityScheduler<T>::run() {
361   T& gptr = *this->execGraph_;
362
363   //Execute everything in the root list
364   for (RootIter rit = rootList_.begin(); rit != rootList_.end(); rit++) {
365     VertexProperty& vp = gptr[*rit];
366 #     ifdef ENABLE_THREADS
367     this->pool_.schedule( boost::threadpool::prio_task_func( vp.priority,
368             boost::bind( &PriorityScheduler<T>::call, this, vp ) ) );
369 #     else
370     this->call(vp);
371 #     endif
372   }
373
374 #   ifdef ENABLE_THREADS
375   this->schedBarrier_.wait();
376 #   endif
377   finish();
378 }
379
380 /**
381  * \brief Called when the graph is done executing, resets state variables.
382  */
383 template<class T>
384 void PriorityScheduler<T>::finish() {
385   T& gptr = *this->execGraph_;
386
387   this->nelements_ = boost::num_vertices(*this->execGraph_);
388   this->nremaining_ = this->nelements_;
389
390   const std::pair<VertIter, VertIter> execGraphVertices = boost::vertices(gptr);
391
392   //grab the root list, default remaining count to parent count
393   for (VertIter iter = execGraphVertices.first; iter != execGraphVertices.second; ++iter) {
394     VertexProperty& vp = gptr[*iter];
395     vp.nremaining = vp.nparents;
396     vp.ncremaining = vp.nconsumers;
397   }
398 }
399
400 //------------------- End Priority Scheduler ----------------------//
```

A.3 Hybrid Scheduler

```
1 /**
2  * Available debugging flags:
3  *      DEBUG_SCHED_ALL - Enable all scheduler debugging flags
4  *
5  *      DEBUG_NO_FIELD_RELEASE - Disable releasing field memory during execution
6  *
7  *
8  */
```

```
9   #ifdef DEBUG_SCHED_ALL
10  #define DEBUG_NO_FIELD_RELEASE
11  #endif
12
13  #ifndef Expr_TaskSchedulers_hxx
14  #define Expr_TaskSchedulers_hxx
15
16  //Standard libraries
17  #include <stdio.h>
18  #include <iostream>
19  #include <string>
20  #include <stdexcept>
21  #include <map>
22
23  //SpatialOps
24  #include <spatialops/structured/MemoryTypes.h>
25  #include <spatialops/structured/ExternalAllocators.h>
26
27  //Expressions
28  #include <expression/SchedulerBase.h>
29
30  //Boost includes
31  #include <boost/graph/adjacency_list.hpp>
32  #include <boost/graph/graph_traits.hpp>
33  #include <boost/graph/visitors.hpp>
34  #include <boost/graph/breadth_first_search.hpp>
35  #include <boost/graph/transpose_graph.hpp>
36  #include <boost/shared_ptr.hpp>
37  #include <boost/signal.hpp>
38  #include <boost/bind.hpp>
39
40  namespace Expr {
41
42  template<class T>
43  class HybridScheduler;
44
45  template<class T = SchedulerGraphTypes::DefaultType>
46  class HybridScheduler: public Scheduler {
47    /**
48     *  \class ExecMutex
49     *  \brief Scoped lock. An instance should be constructed within any function that touches Scheduler
50     *  member variables.
51     */
52    class ExecMutex {
53  #   ifdef ENABLE_THREADS
54        const boost::mutex::scoped_lock lock;
55        inline boost::mutex& get_mutex() const {static boost::mutex m; return m;}
56
57        public:
58        ExecMutex() : lock( get_mutex() ) {}
59        ~ExecMutex() {}
60  #   else
61      public:
62        ExecMutex(){}
63        ~ExecMutex(){}
64  #   endif
65    };
66
67    /**
68     * @class GPULoadBalancer
69     *
70     * @brief Used to maintain loading and assignment information with respect to
71     *          available hardware resources and tasks assigned to them.
72     */
73    class GPULoadBalancer {
74      typedef typename boost::graph_traits<T>::vertex_descriptor Vertex;
75
76      public:
77        enum Method { RoundRobin, MinimumLoading };
78
79        GPULoadBalancer() : gpuDeviceCount_(0), nextRR_(0), nextCID_(0), method(RoundRobin) {}
80
81        Method get_assignment_strategy() const {
82          return method;
83        }
84
85        void set_assignment_strategy( Method m ) {
86          method = m;
87        }
88
89        unsigned int get_next_cid() {
90          return nextCID_++;
91        }
92
93        unsigned int get_next_device() {
94          switch(method){
95            case RoundRobin: {
96              return ( ( ++nextRR_ ) % gpuDeviceCount_ );
97            }
98
99            case MinimumLoading: {
100             unsigned int index = 0;
```

```
101            for( unsigned int i = 0; i < deviceLoading_.size(); ++i ){
102              index = ( deviceLoading_[index] < deviceLoading_[i] ) ? index : i;
103            }
104
105            return index;
106          }
107
108          default:
109            throw("Unknown␣device␣loading␣type\n");
110        }
111      }
112    private:
113      Method method;
114
115      int gpuDeviceCount_;
116      int nextRR_;
117      int nextCID_;
118
119      std::vector<unsigned int> deviceMemorySize_;
120      std::vector<unsigned int> deviceLoading_;
121      std::map< unsigned int, std::list<Vertex> > CoalescingChains_;
122  };
123
124 public:
125
126    HybridScheduler(T* graph) :
127        execGraph_(graph), taskGraph_(NULL), __run_gpu(false), Scheduler() {
128
129 #ifdef ENABLE_CUDA
130        //Grab GPU information
131        ema::cuda::CUDADeviceInterface& CDI = ema::cuda::CUDADeviceInterface::self();
132
133        /** Determine how many GPUs we have **/
134        gpuLoadBalancer_.gpuDeviceCount_ = CDI.get_device_count();
135
136        /** Update memory information **/
137        CDI.update_memory_statistics();
138
139        ema::cuda::CUDAMemStats CMS;
140        for( int device = 0; device < gpuLoadBalancer_.gpuDeviceCount_; device++ ){
141          CDI.get_memory_statistics(CMS, device);
142          gpuLoadBalancer_.deviceMemorySize_.push_back(CMS.t);
143          gpuLoadBalancer_.deviceLoading_.push_back(0);
144        }
145 #endif
146    }
147
148    ~HybridScheduler() {
149      if (taskGraph_ != NULL) {
150        delete taskGraph_;
151      }
152    }
153
154    /**
155     * \brief after this function runs, our scheduler should be in a runnable state
156     */
157    void setup(bool hasRegisteredFields = false);
158
159    /**
160     * \brief begin executing graph nodes
161     */
162    void run();
163
164    /**
165     * \brief perform any cleanup activities
166     */
167    void finish();
168
169    /**
170     * \brief this function is called by an expression when it has finished executing
171     * we do introspection and determine which nodes are ready to run from here.
172     */
173    void exec_callback_handler(void*);
174
175    /**
176     * \brief return a string identifying which scheduler we are.
177     */
178    const std::string get_identity() {
179      return std::string("GPU␣Scheduler␣--␣Testing");
180    }
181
182    /**
183     * \brief intermediary for executing a node when it is ready, used so that we can control
184     * when we bind memory to each field
185     */
186    void call(VertexProperty& target);
187
188    /**
189     * \brief change our task graph, note this invalidates the graph and will force a full run through
190     * setup the next time it is called.
191     */
192    void set_task_graph(T* graph);
```

```
193
194     /**
195      *  Decrement the number of expression resources remaining to be computed
196      */
197     void dec_remaining();
198
199   protected:
200
201     typedef typename boost::graph_traits<T>::vertex_descriptor Vertex;
202     typedef std::vector<Vertex> VertList;
203     typedef typename VertList::iterator RootIter;
204     typedef typename std::map<ExpressionID, Vertex> ID2VP;
205
206     typedef typename boost::graph_traits<T>::edge_descriptor Edge; ///< Edge in a graph
207     typedef typename boost::graph_traits<T>::edge_iterator EdgeIter; ///< Edge iterator
208     typedef typename boost::graph_traits<T>::vertex_iterator VertIter;
209     typedef typename boost::graph_traits<T>::out_edge_iterator OutEdgeIter;
210
211     T* execGraph_;
212     T* taskGraph_;
213     int nelements_;
214     int nremaining_;
215     /** Testing **/
216     bool __run_gpu;
217
218     VertList rootList_;
219
220     /** Producer and Consumer vertex maps **/
221     ID2VP execVertexMap_;
222     ID2VP taskVertexMap_;
223
224     GPULoadBalancer gpuLoadBalancer_;
225
226     /**
227      * \brief Boost visitor structure for coalescing paths
228      *
229      * Greedy chaining algorithm. Attempts to create the longest single-path chains possible
230      */
231     struct LoadBalanceVisitor: public boost::default_bfs_visitor {
232       LoadBalanceVisitor(GPULoadBalancer* gpuLB) {
233         gpuLB_ = gpuLB;
234       }
235
236       inline void examine_edge(Edge e, const T& g) {
237         const Vertex src = boost::source(e, g);
238         const Vertex dest = boost::target(e, g);
239         T& g2 = const_cast<T&>(g);
240
241         VertexProperty& svp = g2[src];
242         VertexProperty& dvp = g2[dest];
243
244         if( svp.execTarget == GPU ){
245           //std::cout << "Source execution target is GPU\n";
246           if( svp.chainID_ == -1 ){ // Source vertex is not part of a coalescing chain. Make a new one
247             svp.chainID_ = gpuLB_->get_next_cid();
248             std::list<Vertex> temp;
249             gpuLB_->CoalescingChains_.insert( std::pair<unsigned int, std::list<Vertex> >( svp.chainID_, temp ) )
                  ;
250             std::list<Vertex>& chain = gpuLB_->CoalescingChains_[ svp.chainID_ ];
251             chain.push_back( src );
252           }
253
254           //If the destination node is already taken, look at absorbing it
255           if( dvp.execTarget == GPU ){
256             //std::cout << "Destination execution target is GPU\n";
257             if( dvp.chainID_ == -1 && svp.chainTail_ ) {
258               // Add destvp to the chain if it isn't already taken
259               svp.chainTail_ = false;
260               dvp.chainID_ = svp.chainID_;
261               //std::cout << "Destination is not part of an existing chain pushing to source chain, ID: " << dvp.
                    chainID_ << std::endl;
262               std::list<Vertex>& chain = gpuLB_->CoalescingChains_[ dvp.chainID_ ];
263               chain.push_back(dest);
264             } else {
265               std::list<Vertex>& dchain = gpuLB_->CoalescingChains_[dvp.chainID_];
266
267               if( dest == dchain.front() && svp.chainTail_ ) {
268                 svp.chainTail_ = false;
269                 unsigned int t = dvp.chainID_;
270                 //Dest is the head of another chain, attach it to our current chain
271                 std::list<Vertex>& schain = gpuLB_->CoalescingChains_[svp.chainID_];
272                 while( !dchain.empty() ){
273                   Vertex& v = dchain.front();
274                   VertexProperty& vp = g2[v];
275                   vp.chainID_ = svp.chainID_;
276                   schain.push_back(v);
277                   dchain.pop_front();
278                 }
279                 gpuLB_->CoalescingChains_.erase(t);
280               }
281             }
282           }
```

```
283         } else {
284           //If the destination node is already taken, look at absorbing it
285           if( dvp.execTarget == GPU && dvp.chainID_ == -1 ){
286             dvp.chainID_ = gpuLB_->get_next_cid();
287 
288             std::list<Vertex> temp;
289             gpuLB_->CoalescingChains_.insert( std::pair<unsigned int, std::list<Vertex> >(dvp.chainID_, temp) );
290             std::list<Vertex>& chain = gpuLB_->CoalescingChains_[dvp.chainID_];
291             chain.push_back(dest);
292           }
293         }
294       }
295 
296       GPULoadBalancer* gpuLB_;
297     };
298 };
299 
300 /***************** Begin HybridScheduler Implementation *****************/
301 /**
302  * \brief Reassign a specific task graph.
303  */
304 template<class T>
305 void HybridScheduler<T>::set_task_graph(T* graph) {
306   invalid_ = true;
307   this->execGraph_ = graph;
308 }
309 
310 /**
311  * \brief Setup and run a specific task.
312  */
313 template<class T>
314 void HybridScheduler<T>::call(VertexProperty& target) {
315   //Bind the fields for this expression
316   (target.expr)->base_bind_fields(*this->fml_);
317 
318   //Execute the expression
319   target.execute_expression();
320 }
321 
322 /**
323  * \brief Process tasks as they finish.
324  */
325 template<class T>
326 void HybridScheduler<T>::exec_callback_handler(void* expr_vertex) {
327   T& gptr = *this->execGraph_;
328 
329   Vertex v = (Vertex) expr_vertex;
330   VertexProperty& vpJustFinished = gptr[v];
331 
332   // Notify the vertex that a consumer has finished, it returns true, it can be freed.
333   for (std::vector<VertexProperty*>::iterator vpit = vpJustFinished.ancestorList.begin();
334         vpit != vpJustFinished.ancestorList.end();
335           ++vpit )
336   {
337     if ((*vpit)->consumer_finished()) {
338       if ( this->fdm_ != NULL && this->fml_ != NULL ) {
339         FieldDeps& fd = *((*this->fdm_)[(*vpit)->id]);
340 
341 #ifndef DEBUG_NO_FIELD_RELEASE
342         bool result = fd.release_fields(*this->fml_);
343 #endif
344       }
345     }
346   }
347 
348   // Notify the vertex that an ancestor has finished, it returns true if it is ready.
349   // If it is, we either toss it to the thread pool or run it.
350   FieldDeps& fd = *((*this->fdm_)[vpJustFinished.id]);
351 
352   for (std::vector<VertexProperty*>::iterator vpit = vpJustFinished.consumerList.begin();
353         vpit != vpJustFinished.consumerList.end();
354           ++vpit )
355   {
356 
357     VertexProperty& destvp = (**vpit);
358 
359     //Here, destvp will be a consumer of vpJustFinished, so vpJustFinished must be prepared to be consumed on
          whichever
360     //device destvp is set to execute on.
361 
362     SpatialOps::MemoryType smtype, dmtype;
363 
364     switch( vpJustFinished.mm ){
365     case MEM_DYNAMIC_GPU:
366       smtype = SpatialOps::EXTERNAL_CUDA_GPU;
367       break;
368     default:
369       smtype = SpatialOps::LOCAL_RAM;
370       break;
371     }
372 
373     switch( destvp.mm ){
```

```
374     case MEM_DYNAMIC_GPU:
375       dmtype = SpatialOps::EXTERNAL_CUDA_GPU;
376       break;
377     default:
378       dmtype = SpatialOps::LOCAL_RAM;
379       break;
380     }
381 
382     //Check to see if this field needs to be prepared
383     //Note: adding consumer fields is a thread safe operation
384     if( ( smtype != dmtype ) || ( vpJustFinished.deviceIndex_ != destvp.deviceIndex_ ) ){
385 #ifdef DEBUG_SCHED_ALL
386       std::cout << "Field␣requires␣preparation␣for␣consumption:␣" << srcvp.expr->name() << std::endl;
387       std::cout << "Allocating␣on␣" << ( ( destvp.mm != MEM_DYNAMIC_GPU ) ? "LOCAL␣MEMORY" : "GPU␣MEMORY" )
388           << "␣device␣index:␣" << destvp.deviceIndex_ << std::endl;
389 #endif
390 
391       fd.prep_field_for_consuption(*this->fml_, dmtype, destvp.deviceIndex_ );
392     }
393 
394     if (destvp.ancestor_finished()) {
395 #ifdef ENABLE_THREADS
396       this->pool_.schedule(
397           boost::threadpool::prio_task_func( destvp.priority,
398               boost::bind( &HybridScheduler<T>::call, this, destvp ) ) );
399 
400 #else
401       this->call(destvp);
402 #endif
403     }
404   }
405   dec_remaining();
406 }
407 
408 /**
409  * \brief decrement remaining tasks.
410  */
411 template<class T>
412 void HybridScheduler<T>::dec_remaining() {
413 # ifdef ENABLE_THREADS
414   typename HybridScheduler<T>::ExecMutex lock;
415 # endif
416   --nremaining_;
417 
418 # ifdef ENABLE_THREADS
419   if( nremaining_ == 0 ) {
420     this->schedBarrier_.post();
421   }
422 # endif
423 }
424 
425 /**
426  * \brief Initialize any data structures and information required for 'run()' to complete.
427  */
428 //TODO: To make scheduling smarter, we may want to enumerate all available devices
429 //      and memory in order to come up with a kind of 'banker's algorithm' to
430 //      avoid over scheduling.
431 template<class T>
432 void HybridScheduler<T>::setup( bool hasRegisteredFields ) {
433   /**Notes on whats going on here
434    * - gptr is the execution graph
435    * - tgptr is the consumer ( dependency graph )
436    *
437    * Step 1: Reset and reconnect all variables to place the graph into state which is execute ready.
438    *
439    * Step 2: Inspect the execution graph, flag edge nodes and set compute device.
440    *
441    * Step 3: Inspect the execution graph w/ hardware targets -- coalesce paths where possible
442    *
443    * Step 4: If our graph nodes are allocated, we update their field managers
444    *
445    * Step 5: rebuild our task graph indices.
446    **/
447 
448   // Quick return if we're already valid or if we're not fully setup yet.
449   if ( !invalid_ )
450     return;
451 
452   // Variable init/clearing and setup **/
453   T& gptr = *this->execGraph_;
454 
455   // Destroy our local task graph
456   if (this->taskGraph_ != NULL) {
457     delete this->taskGraph_;
458   }
459   this->taskGraph_ = new T();
460   T& tgptr = *this->taskGraph_;
461 
462   // Clear are scheduling lists
463   rootList_.clear();
464   execVertexMap_.clear();
465   taskVertexMap_.clear();
```

```
466
467   // Reset load balancer variables
468   gpuLoadBalancer_.CoalescingChains_.clear();
469   for( std::vector<unsigned int>::iterator vit = gpuLoadBalancer_.deviceLoading_.begin();
470       vit != gpuLoadBalancer_.deviceLoading_.end(); ++vit ){
471     (*vit) = 0;
472   }
473
474   //Update execution counters.
475   nelements_ = boost::num_vertices(*this->execGraph_);
476   nremaining_ = nelements_;
477
478   const std::pair<VertIter, VertIter> execGraphVertices = boost::vertices(gptr);
479
480   // ------------------------------ **/
481
482   /**Step - 1 Reconnect all signals and reset execution counts
483    *         Determine consumer and parent counts for all nodes in the graph
484    *
485    **/
486   VertIter iter;
487   for (iter = execGraphVertices.first; iter != execGraphVertices.second; ++iter) {
488     VertexProperty& vp = gptr[*iter];
489
490     vp.self_            = (void*) (*iter);
491     vp.nparents         = 0;
492     vp.nconsumers       = 0;
493     vp.chainID_         = -1;
494     vp.chainTail_       = true;
495     vp.vtp.eTimeCPU_    = 0.0;
496     vp.vtp.eTimeGPU_    = 0.0;
497
498     execVertexMap_.insert(std::make_pair(gptr[*iter].id, *iter));
499
500     vp.execSignalCallback.reset(new VertexProperty::Signal());
501     vp.execSignalCallback->connect(
502         boost::bind(&HybridScheduler::exec_callback_handler, this, vp.self_) );
503
504     vp.ancestorList.clear();
505     vp.consumerList.clear();
506
507     vp.set_is_edge(false);
508   }
509
510   for (iter = execGraphVertices.first; iter != execGraphVertices.second; ++iter) {
511     VertexProperty& vp = gptr[*iter];
512
513     std::pair<OutEdgeIter, OutEdgeIter> edges = boost::out_edges(*iter, gptr);
514     for( OutEdgeIter eit = edges.first; eit != edges.second; ++eit ){
515   /**
516   //                                        /‘‘‘‘> cp1 ( Add 1 to parent count )
517   // Idea: ( Add 1 to consumer count ) vp  -----> cp2 ( Add 1 to parent count )
518   //                                        \____> cp3 ( Add 1 to parent count )
519   //
520   // Doing it like this we compute all consumer and parent counts at the same time.
521   **/
522       VertexProperty& cp = gptr[boost::target(*eit, gptr)];
523
524       vp.consumerList.push_back(&cp);
525       cp.ancestorList.push_back(&vp);
526       (vp.nconsumers)++;
527       (cp.nparents)++;
528     }
529   }
530
531   /**Step 2 - Build our root list, classify persistence
532    *         - The root list is composed of nodes that do not have any parents
533    *         ( topologic edge nodes )
534    *
535    *         - At present all edge nodes, including leaf nodes are defined to be
536    *             persistent. This may change in the future.
537    *
538    *         ( Not yet implemented -- requires changes to field registration guarantees )
539    *         - Do a local sanity check on memory to make sure the device we’re assigning
540    *             can support any single field + its dependencies
541    *
542    *         - This initial pass will try and set node hardware targets based on
543    *             execution + data transfer times.
544    *
545    **/
546   for (VertIter iter = execGraphVertices.first; iter != execGraphVertices.second; ++iter) {
547     VertexProperty& vp = gptr[*iter];
548
549     //For the execution graph nodes at the bottom of the tree are roots and have no parents.
550     //Since edge nodes cannot be ’dynamic’ we flag these nodes as persistent
551
552     //Part of memory sanity check -- unimplemented
553     //vp.nodeMemoryBound_ = vp.nparents * fd.get_field_size(*this->fml_);
554
555     if (vp.nparents == 0) {
556       vp.set_is_edge(true);
557       rootList_.push_back(*iter);
```

```
558     }
559
560     if (vp.nconsumers == 0) {
561       vp.set_is_edge(true);
562     }
563
564     if (vp.get_is_edge()) {
565       vp.set_is_persistent(true);
566     } else {
567       // jcs will this over-ride someone locking a field?
568       // dvn: no, this will just inform the scheduler that there is nothing
569       //      at the graph level that forces a field to be locked.
570       vp.set_is_persistent(false);
571       vp.set_gpu_runnable(true);
572     }
573     //--------------//
574
575     // If this node can be run on a GPU, decide which is better
576     if ( vp.get_gpu_runnable() ) {
577       //Debugging stuff
578       if( __run_gpu ){
579         vp.execTarget = GPU;
580       } else {
581         vp.execTarget = CPU; //( vp.vtp.eTimeCPU_ < vp.vtp.eTimeGPU_ ) ? CPU : GPU;
582       }
583     }
584
585     switch (vp.execTarget) {
586       case CPU: {
587         if( vp.get_is_persistent() ){
588           vp.mm = MEM_EXTERNAL;
589         } else {
590           vp.mm = MEM_DYNAMIC;
591         }
592       }
593       break;
594
595       case GPU:{
596           vp.mm = MEM_DYNAMIC_GPU;
597       }
598       break;
599
600       default:
601         throw( std::runtime_error("Invalid␣execution␣target␣specified" ));
602     }
603
604     vp.nremaining = vp.nparents;
605     vp.ncremaining = vp.nconsumers;
606   }
607
608   /**Step 3 - BFS from each root node, calling our load balancer as necessary.
609    *
610    *    Example:
611    *
612    *        (A)          If we suppose that each node in this graph will be run on GPU,
613    *       /   \         then our search will construct the following chains:
614    *     (B)   (C)
615    *    /   \  /         { A->B->D, C->E }
616    *  (D)    (E)
617    *
618    **/
619   for (typename std::vector<Vertex>::iterator iter = rootList_.begin(); iter != rootList_.end(); ++iter) {
620     boost::breadth_first_search(tgptr, *iter, boost::color_map(
621         boost::get(&VertexProperty::color, tgptr)).visitor(LoadBalanceVisitor(&gpuLoadBalancer_)));
622   }
623
624   for( typename std::map<unsigned int, std::list<Vertex> >::iterator mit = gpuLoadBalancer_.CoalescingChains_.
          begin();
625       mit != gpuLoadBalancer_.CoalescingChains_.end(); ++mit ){
626     typename std::list<Vertex>& chain = mit->second;
627
628     unsigned int dIndex = gpuLoadBalancer_.get_next_device();
629     gpuLoadBalancer_.deviceLoading_[dIndex] += chain.size();
630     for( typename std::list<Vertex>::iterator lit = chain.begin(); lit != chain.end(); ++lit ){
631       VertexProperty& vp = gptr[*lit];
632       vp.deviceIndex_ = dIndex;
633       //std::cout << "Setting device index to " << dIndex << " for GPU device in chain " << mit->first << std::
            endl;
634     }
635   }
636
637   /**Step 4 - If fields are already registered, then we push any field changes to the Field managers
638    *
639    *   If we know the fields associated with this expression have already been registered
640    *   then we need to updated the field memory manager to reflect changes which may have
641    *   occurred during this setup.
642    *
643    **/
644   if( hasRegisteredFields ) {
645     for (VertIter iter = execGraphVertices.first; iter != execGraphVertices.second; ++iter) {
646       VertexProperty& vp = gptr[*iter];
647
```

```
648       FieldDeps& fd = *((*this->fdm_)[vp.id]);
649       fd.set_memory_manager( *this->fml_, vp.mm, vp.deviceIndex_ );
650     }
651   }
652
653   /**Step 5 - Build the transpose and generate vertex mappings
654   //        - We store the transpose ( task graph ), because it lets us
655   //          backtrack to find execution parent nodes during execution
656   //          and inform them consumers have finished ( remember this is a directed graph )
657   //  /\
658   // /||\
659   //  ||                        ( consumer )------>\ ( task graph directionality, stored in 'taskVertexMap_' )
660   //  ||                                    ^       \
661   //  || ( execution graph directionality )|        \
662   //  ||                                    <-------( node )<--------\
663   //  ||                                                  \          ^ ( execution graph directionality, stored in
      'execVertexMap_' )
664   //  ||                                                   \         |
665   //  ||                       ( task graph directionality )\------> ( parent )
666   //  ||
667   //  ^^
668   //  Direction of Execution
669   //==========================================================================================================*
      */
670   boost::transpose_graph(gptr, tgptr,
671       boost::vertex_index_map( boost::get(&VertexProperty::index, gptr) ) );
672   const std::pair<VertIter, VertIter> taskGraphVertices = boost::vertices(tgptr);
673
674   for (VertIter vit = taskGraphVertices.first; vit != taskGraphVertices.second; ++vit) {
675     taskVertexMap_.insert(std::make_pair(tgptr[*vit].id, *vit));
676   }
677
678   invalid_ = false;
679 }
680
681 /**
682  * \brief Begin executing on the graph by loading root expressions onto the queue.
683  */
684 template<class T>
685 void HybridScheduler<T>::run() {
686   //this->setup();
687
688   T& gptr = *this->execGraph_;
689
690   //Execute everything in the root list
691   for (RootIter rit = rootList_.begin(); rit != rootList_.end(); rit++) {
692     const VertexProperty& vp = gptr[*rit];
693 #   ifdef ENABLE_THREADS
694     this->pool_.schedule( boost::threadpool::prio_task_func( vp.priority,
695             boost::bind( &HybridScheduler<T>::call, this, vp ) ) );
696 #   else
697     this->call(vp);
698 #   endif
699   }
700
701 #   ifdef ENABLE_THREADS
702   this->schedBarrier_.wait();
703 #   endif
704   finish();
705 }
706
707 /**
708  * \brief Called when the graph is done executing, resets state variables.
709  */
710 template<class T>
711 void HybridScheduler<T>::finish() {
712   T& gptr = *this->execGraph_;
713
714   this->nelements_ = boost::num_vertices(*this->execGraph_);
715   this->nremaining_ = this->nelements_;
716
717   const std::pair<VertIter, VertIter> execGraphVertices = boost::vertices(gptr);
718
719   //grab the root list, default remaining count to parent count
720   for (VertIter iter = execGraphVertices.first; iter != execGraphVertices.second; ++iter) {
721     VertexProperty& vp = gptr[*iter];
722     vp.nremaining = vp.nparents;
723     vp.ncremaining = vp.nconsumers;
724   }
725 }
726
727 //------------------- End Hybrid Scheduler -----------------------//
728 }
```

REFERENCES

[1] Agullo, E., Hadri, B., Ltaief, H., and Dongarrra, J. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009), ACM, p. 20.

[2] Al-Mouhamed, M. A. Lower bound on the number of processors and time for scheduling precedence graphs with communication costs. *IEEE Trans. Softw. Eng. 16*, 12 (Dec. 1990), 1390–1401.

[3] Amor, D. *Internet Future Strategies: How Pervasive Computing Services Will Change the World*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[4] Andrade, H. A., and Kovner, S. Software synthesis from dataflow models for g and labview/sup tm. In *Signals, Systems & Computers, 1998. Conference Record of the Thirty-Second Asilomar Conference on* (1998), vol. 2, IEEE, pp. 1705–1709.

[5] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. Parallel tiled qr factorization for multicore architectures. *Concurrency and Computation: Practice and Experience 20*, 13 (2008), 1573–1590.

[6] Chan, E., Quintana-Orti, E. S., Quintana-Orti, G., and Van De Geijn, R. Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures* (2007), ACM, pp. 116–125.

[7] Corporation, N. *NVIDIA CUDA programming guide.*, 2011.

[8] Davison de St Germain, J., McCorquodale, J., Parker, S. G., and Johnson, C. R. Uintah: A massively parallel problem solving environment. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on* (2000), IEEE, pp. 33–41.

[9] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. S. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS) 16*, 1 (1990), 1–17.

[10] Eckert, J. P. Oral history interview with j. presper eckert, 1978.

[11] Fernandez, E. B., and Bussell, B. Bounds on the number of processors and time for multiprocessor optimal schedules. *IEEE Trans. Comput. 22*, 8 (Aug. 1973), 745–751.

[12] Frigo, M., Leiserson, C. E., and Randall, K. H. The implementation of the cilk-5 multithreaded language. *ACM Sigplan Notices 33*, 5 (1998), 212–223.

[13] Germain, D. S., J. D., M., J., P., S. G., J., and R., C. Uintah: A massively parallel problem solving environment. ninth ieee international symposium on high performance and distributed computing. In *Uintah: A Massively Parallel Problem Solving Environment. Ninth IEEE International Symposium on High Performance and Distributed Computing* (2000), pp. 33–41.

[14] Haidar, A., Ltaief, H., YarKhan, A., and Dongarra, J. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurrency and Computation: Practice and Experience 24*, 3 (2011), 305–321.

[15] Hruska, J. Nvidia offers peek into advanced design evaluation. *HotHardware* (2011).

[16] Kurzak, J., and Dongarra, J. Fully dynamic scheduler for numerical computing on multicore processors. *Univ. of Tennessee LAPACK Working Note 220* (2009).

[17] Martin, B. *The separation of interface and implementation in C++.* Hewlett-Packard Laboratories., 1990.

[18] Miller, F. P., Vandome, A. F., and McBrewster, J. *Moore's law: History of computing hardware, Integrated circuit, Accelerating change, Amdahl's law, Metcalfe's law, Mark Kryder, Jakob Nielsen (usability consultant), Wirth's law.* Alpha Press, 2009.

[19] Notz, P., Pawlowski, R., and Sutherland, J. Graph-based software design for managing complexity and enabling concurrency in multiphysics pde software. *ACM Transactions on Mathematical Software (submitted)* (2011).

[20] Randall, K. H. *Cilk: E cient Multithreaded Computing.* PhD thesis, Massachusetts Institute of Technology, 1998.

[21] Siek, J., Lee, L., and Lumsdaine, A. *Boost Graph Library: User Guide and Reference Manual, The.* Addison-Wesley Professional, 2001.

[22] Sinnen, O. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing).* Wiley-Interscience, 2007.

[23] Supercomputing Technologies Group, M. L. f. C. S. *Cilk reference manual.*, 1998.

[24] Trendall, C. "ray tracing refraction in hardware". Master's thesis, "University of Toronto", "2000".