An Exploration of Task-based Programming for Scientific Applications

A Dissertation presented

by

Eric Raut

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Applied Mathematics and Statistics

(Computational Applied Mathematics)

Stony Brook University

August 2022
Stony Brook University

The Graduate School

Eric Raut

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Barbara Chapman - Dissertation Advisor
Professor, Computer Science and Applied Mathematics & Statistics

Robert Harrison - Chairperson of Defense
Professor, Applied Mathematics & Statistics

Xiangmin Jiao - Committee Member
Associate Professor, Applied Mathematics & Statistics

Mauricio Araya-Polo - External Member
Senior R&D Manager, TotalEnergies EP R&T USA

This dissertation is accepted by the Graduate School

Celia Marshik

Interim Dean of the Graduate School
Abstract of the Dissertation

An Exploration of Task-based Programming for Scientific Applications

by

Eric Raut

Doctor of Philosophy

in

Applied Mathematics and Statistics

(Computational Applied Mathematics)

Stony Brook University

2022

This dissertation explores the state-of-the-art in task-based programming. One of the main motivating examples is Reverse Time Migration (RTM), a seismic imaging technique commonly used in energy exploration. We first give an overview of task-based programming, including a survey of current task-based programming systems and some common algorithms well suited to tasks. Then we introduce the case study for the following sections, Minimod (proprietary code provided by TotalEnergies).

The first sections focus on the physical modeling problem derived from Minimod. We present a version using OpenMP tasks, comparing the implementation across different CPU architectures. Then we extend to GPUs using OpenMP offloading, both within a node and using OpenMP offloading. Next, we move to distributed tasking using Legion. We present results with CPUs and GPUs, showing they are mostly competitive with an MPI baseline, but suffer overheads at high node counts. We also produce an MPI+OpenMP task version and compare it with an MPI+OpenMP loop-based baseline. We identify weaknesses in using OpenMP tasks in this manner and suggest improvements. Finally, we form a complete application by adding the backward pass of RTM. We find that we can effectively overlap I/O with computation to reduce time-to-solution.
Major findings of this work include a sensitivity of OpenMP tasks to cache structure, existence of an optimal task granularity for OpenMP tasks in Minimod, competitive performance between tasks and traditional MPI- and loop-based methods across the forward-pass experiments, and improvement in the full RTM application due to overlapping computation with disk I/O. These findings show a promising future for the use of tasks in scientific applications.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>xiv</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation: Potential Benefits of Tasks</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Objectives of this Dissertation</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Contributions and Major Findings</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Outline of Dissertation</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Task-based Programming</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Considerations of task-based programming</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1 Task granularity</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2 Task dependencies</td>
<td>6</td>
</tr>
<tr>
<td>2.2.3 Task priorities</td>
<td>6</td>
</tr>
<tr>
<td>2.2.4 Scheduling and load balancing</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Survey</td>
<td>7</td>
</tr>
<tr>
<td>2.3.1 Legion</td>
<td>7</td>
</tr>
<tr>
<td>2.3.2 Chapel</td>
<td>7</td>
</tr>
<tr>
<td>2.3.3 XcalableMP</td>
<td>8</td>
</tr>
<tr>
<td>2.3.4 OpenMP</td>
<td>10</td>
</tr>
<tr>
<td>2.3.5 Others</td>
<td>12</td>
</tr>
<tr>
<td>2.3.6 Comparision</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Example Task-based Algorithms</td>
<td>12</td>
</tr>
<tr>
<td>2.4.1 Conjugate Gradient Method</td>
<td>13</td>
</tr>
<tr>
<td>2.4.2 Cholesky Decomposition</td>
<td>15</td>
</tr>
</tbody>
</table>
3 Introduction to Minimod
   3.1 Related Work .............................................. 19

4 Evaluation of OpenMP CPU tasks in Minimod 21
   4.1 Introduction ........................................... 21
   4.2 Related Work ........................................... 22
   4.3 Code Structure and Taskification of Minimod .......... 22
   4.4 Evaluation ............................................... 25
      4.4.1 Experimental Setup ................................ 25
      4.4.2 Results ............................................. 26
   4.5 Discussion ............................................... 29
   4.6 Conclusions ............................................. 30

5 Porting Minimod to GPUs using OpenMP 32
   5.1 Introduction ........................................... 32
   5.2 Design of OpenMP-offload version of Minimod ....... 32
   5.3 Remote offloading plugin ................................ 33
   5.4 Performance results ................................... 34
   5.5 Conclusion ............................................... 35

6 Experiments with concurrent CPU+GPU offloading 36
   6.1 Experimental setup ...................................... 36
   6.2 Concurrent CPU+GPU experiments ....................... 36
   6.3 Conclusions ............................................... 37

7 Evaluation of Minimod Across A64FX Platforms 39
   7.1 Introduction ........................................... 39
      7.1.1 The A64FX Processor .............................. 40
      7.1.2 Chapter’s contribution and organization ....... 40
   7.2 Application and experimental setup .................... 41
      7.2.1 Application: Minimod ............................. 41
      7.2.2 Systems and Compilers ............................ 41
      7.2.3 Runtime Environment ............................... 42
      7.2.4 Compiler options ................................ 42
   7.3 Experimental results ................................... 43
      7.3.1 Ookami .............................................. 44
      7.3.2 Fugaku .............................................. 44
   7.4 Related Work ............................................ 46
   7.5 Conclusions ............................................... 47
8 Interfacing MPI with OpenMP tasks

8.1 Related work ........................................... 48
8.2 Description of code changes ............................... 49
8.3 Preliminary Results ........................................ 50
  8.3.1 Experimental Setup ................................. 50
  8.3.2 Results ........................................ 50
8.4 Perspectives on OpenMP tasks .............................. 51
8.5 Conclusion ............................................ 52

9 Minimod extension to multiple nodes using Legion 

9.1 Introduction ............................................ 53
9.2 Parallel Stencil Computation .............................. 54
9.3 Related Work: Distributed task parallelism ................. 55
9.4 Distributed MiniMod (dMiniMod) .......................... 56
9.5 ASF: Advanced Seismic Framework ......................... 57
9.6 Legion Implementation .................................... 59
  9.6.1 Parallelism in ASF-Legion ......................... 59
  9.6.2 Domain Decomposition and Mapping ................... 59
  9.6.3 Memory Management ................................ 62
9.7 Experimental Results .................................... 62
  9.7.1 Experimental Setup and Platforms .................... 62
  9.7.2 Scalability Results and Analysis ..................... 64
  9.7.3 Profiling .................................... 69
9.8 Conclusion ............................................ 69

10 Minimod extension to distributed GPUs using Legion

10.1 Introduction ............................................ 71
10.2 Related Work ........................................... 73
10.3 ASF: Advanced Seismic Framework ......................... 74
10.4 Legion Implementation .................................... 74
  10.4.1 Parallelism in ASF-Legion, CUDA Implementation .... 75
10.5 Experimental Results .................................... 75
  10.5.1 Scalability Results and Analysis—CUDA ............... 76
  10.5.2 Profiling .................................... 78
10.6 Conclusion ............................................ 78
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Dot product in Chapel using tasks</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Dot product in XcalableMP using tasks</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Dot product in OpenMP using tasks</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Relative performance comparison of task-based models for a synthetic stencil application using Task Bench (reproduced from [74]). © 2020 IEEE</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Matrix decomposition for conjugate gradient algorithm</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Block Cholesky decomposition represented as a directed acyclic graph. Nodes are shown for columns 1 and 2 only.</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>(left) x-y plane view of domain; (right) x-y blocking scheme.</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Execution time (in seconds), top panel Summit, mid panel Cori and bottom panel SeaWulf.</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Effect of block size on execution time using the LLVM compiler on Summit (top), Cori (middle), and SeaWulf (bottom).</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Simplified Minimod multi-GPU offloading and halo exchange workflow</td>
<td>33</td>
</tr>
<tr>
<td>5.2</td>
<td>Strong scaling results of Minimod (lower is better)</td>
<td>34</td>
</tr>
<tr>
<td>7.1</td>
<td>Timing for Minimod/Ookami (a) loop xy (b) tasks xy</td>
<td>44</td>
</tr>
<tr>
<td>7.2</td>
<td>Speedup for Minimod/Ookami (a) loop xy (b) tasks xy</td>
<td>45</td>
</tr>
<tr>
<td>7.3</td>
<td>Timing for Minimod/Fugaku (a) loop xy (b) tasks xy</td>
<td>45</td>
</tr>
<tr>
<td>7.4</td>
<td>Speedup for Minimod/Fugaku (a) loop xy (b) tasks xy</td>
<td>46</td>
</tr>
<tr>
<td>8.1</td>
<td>MPI+OpenMP Task Diagram</td>
<td>49</td>
</tr>
<tr>
<td>8.2</td>
<td>Minimod code snippet showing difficult-to-find race condition</td>
<td>51</td>
</tr>
</tbody>
</table>
9.1 A 9-point 2D stencil is shown, where green colored elements represent grid points within the computing domain, yellow elements are not computed but needed by the stencil (e.g. to impose boundary conditions). In the left panel, a simple threads to rows mapping is presented. In the right panel, two domains \(D_0, D_1\) shaded light blue and light orange are depicted. The hard boundary implies no memory coherency, therefore information must be exchanged between the two domains in order to solve Equation 3.3 correctly.

9.2 Traces of running ASF with 4 MPI ranks from the Traceviewer of HPCToolkit [41]: the main panel on the top left presents sequences of samples of each trace line rendered, while each of the four horizontal sub-panels represent the trace-view from one of the four MPI ranks. The different colors represent the time spent on different subroutines which are listed on the right panel.

9.3 Comparison of performance on a single Summit node between ASF (in blue) and the nearest equivalent algorithm from the MiniMod suite (in red), simulating 1000 timesteps at various grid sizes. Lower is better.

9.4 Legion levels of parallelism.

9.5 Code showing how domain is partitioned. \(nb\{x,y,z\}\) refer to the number of blocks in each dimension. Each point in the color space \(color.is\) is multiplied by the \(3 \times 3\) transform matrix to generate an offset. This offset is added to the extent.

9.6 Code showing how kernel computation tasks are launched. In this example, the logical partition for the waves \(lp.waves\) is added to the task as a region requirement, so that one subtask will be launched for each partition of the domain. The \texttt{execute_index_space} method schedules the execution of each subtask.

9.7 Throughput and speedup, grid size 1024\(^3\), on Summit and Ookami. Higher is better. Memory footprint 20GB (with a single node), data precision is 32-bit float.

9.8 Throughput and speedup, grid size 2048\(^3\), on Summit. Higher is better. Memory footprint 160GB, data precision is 32-bit float. When running the experiments in Summit, each MPI rank is mapped to socket, therefore for the largest number of nodes experiments one have 32 subdomains, which means 10GB footprint per socket.
9.9 Sample extracted from Legion profiler time trace with 16 nodes on Summit (left) and Ookami (right). Each time period is roughly 55 milliseconds. “OpenMP Proc 2” in each trace shows one of the compute processes, and each rectangle to the right of “OpenMP Proc 2” is a task computing the stencil on the subdomain of the grid assigned to that processor. Each task roughly corresponds to one timestep, with the other processors (not shown here) computing the other subdomains of the grid during the same period. The “[x] sys to [y] sys” rows show halo exchange between the indicated processes.

10.1 Aggregated bandwidth test from OSU benchmark on Cypress.

10.2 Strong scaling: Throughput (left) and speedup (right), grid size $8192 \times 1024 \times 1024$, for 1000 time steps on Cypress with CUDA. Higher is better.

10.3 Sample extracted from Legion profiler time trace with two nodes (top) and eight nodes (bottom), each with the first node shown. Total simulation grid size is $8192 \times 1024 \times 1024$. Each time period is roughly 100 milliseconds. Each “GPU Proc” shows a time trace of task execution on a GPU on the node, and each red rectangle is a task computing the stencil on the subdomain of the grid assigned to that GPU. Each task roughly corresponds to one timestep, with the other processors (not shown) computing the other subdomains of the grid during the same period.

10.4 Sample extracted from NSYS profiler time trace with one node for approximately one timestep. Total simulation grid size is $4096 \times 1024 \times 1024$. Each pair of rows represents one GPU (numbered). Blue regions indicate stencil computation, and orange regions indicate P2P data transfer between GPUs.

11.1 Throughput for RTM, saving to disk with a skip factor of 25. Grid size $8192 \times 512 \times 512$.

11.2 Profile of the execution corresponding to Fig. 11.1 with (top) and without (bottom) the I/O fence during forward pass. Green blocks are stencil computation on GPUs; red blocks are I/O.


List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Hardware and software configuration of the experimental platforms.</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>L3 miss rate [%] on each computer for each configuration.</td>
<td>29</td>
</tr>
<tr>
<td>7.1</td>
<td>Compilers of Fugaku and Ookami</td>
<td>42</td>
</tr>
<tr>
<td>7.2</td>
<td>Flags used for each compiler.</td>
<td>43</td>
</tr>
<tr>
<td>8.1</td>
<td>Preliminary strong scaling results on Ookami and Fugaku</td>
<td>50</td>
</tr>
<tr>
<td>9.1</td>
<td>Hardware and software configuration of the experimental platforms.</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>* IBM Spectrum MPI (SMPI)</td>
<td></td>
</tr>
<tr>
<td>9.2</td>
<td>Weak scaling: throughput (stable around the reference is ideal) with</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>1-32 nodes; grid size $1024^3/16$ per node; on Summit.</td>
<td></td>
</tr>
<tr>
<td>9.3</td>
<td>Weak scaling: throughput (stable around the reference is ideal) with</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>1-32 nodes; grid size $1024^3/16$ per node; on Ookami.</td>
<td></td>
</tr>
<tr>
<td>9.4</td>
<td>Sensitivity of ASF-MPI and ASF-Legion to the block partitioning scheme (number of partitions in x,y,z dimensions) of the domain. Elapsed time (seconds) is shown for each scheme. These simulations use a grid size of $1024^3$, with 16 nodes, on Summit.</td>
<td>66</td>
</tr>
<tr>
<td>9.5</td>
<td>Strong scaling: elapsed time for grid size $1024^3$ and corresponding</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>parallel efficiency; Summit</td>
<td></td>
</tr>
<tr>
<td>9.6</td>
<td>Strong scaling: elapsed time for grid size $2048^3$; Summit</td>
<td>67</td>
</tr>
<tr>
<td>9.7</td>
<td>Strong scaling: elapsed time for grid size $1024^3$ and corresponding</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>parallel efficiency; Ookami</td>
<td></td>
</tr>
<tr>
<td>10.1</td>
<td>Comparison of halo exchange techniques for MPI-CUDA implementation on Cypress, using host-side staging buffers in CUDA pinned memory, device-side staging buffers and direct transfer between wave-grid memory regions on separate nodes. Run with a grid size of $1536^3$; runtimes shown in seconds.</td>
<td>74</td>
</tr>
</tbody>
</table>
10.2 Hardware and software configuration of the experimental platform. 76
10.3 CUDA weak scaling: runtime (stable around the reference is ideal) with 1-16 nodes; grid size $1024^3$ per node. Averaged over three trials. 77
10.4 Strong scaling: elapsed time for grid size $8192 \times 1024 \times 1024$ and corresponding parallel efficiency. Averaged over three trials. Mem% shows the amount of available GPU memory utilized across all nodes. 77
Acknowledgements

First, I would like to thank my advisor, Dr. Barbara Chapman, for her invaluable support of this work and of my PhD.

This dissertation benefitted immensely from a long-running collaboration with TotalEnergies Exploration and Production Research and Technologies, who provided funding for the research. Particular thanks goes to Mauricio Araya-Polo and Jie Meng of TotalEnergies for their guidance throughout the PhD.

Furthermore, my committee members provided helpful feedback which improved this manuscript.

We thank Vivek Kale at Brookhaven National Laboratory for his help in guiding the OpenMP tasking experiments in this paper, and the Legion developers for helpful discussions and insight regarding the performance of ASF-Legion.

Finally, we used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. We also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Furthermore, we would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the SeaWulf and Ookami computing systems. Seawulf is funded by NSF grant #1531492, and Ookami by NSF grant #1927880. We thank TotalEnergies for providing access to Mesquite and Cypress clusters. Lastly, we thank the RIKEN Center for Computational Science for access to Fugaku.
Chapter 1

Introduction

Changes to typical HPC architectures, such as large heterogeneous nodes and deep memory hierarchies, require novel programming frameworks to make full use of them while increasing programmer productivity. Current mainstream parallel programming frameworks include MPI and OpenMP. These frameworks place a large burden on the user to map computations and data to resources. A more flexible approach is desired.

Recently, there have been several efforts to create new programming frameworks based on tasks. A task is a small unit of work that is executed sequentially. Multiple such tasks can be scheduled for execution by the runtime. Issues such as placement of data and scheduling of the tasks on compute resources are left to the runtime, relieving the user of these burdens.

1.1 Motivation: Potential Benefits of Tasks

Writing an application in a task-based style potentially allows the developer to focus on algorithms without worrying about the parallel aspects of the computation. In within-node task-based programming models, the runtime manages how tasks are mapped to cores and in what order the tasks execute. Only data dependencies need to be specified between tasks by the developer. In distributed task-based programming models, issues such as inter-node data transfer and mapping of work to nodes can be left to the runtime, alleviating a major burden of writing MPI-style applications.

However, task-based programming has its own set of productivity challenges. For many scientific applications, expression of the algorithm in a traditional loop-based style is more straightforward, and for these applications there is much legacy code written using loops. Furthermore, as we explore in later chapters, it can be
challenging to understand the performance of task-based applications because of their asynchronous style of execution. Current mainstream profilers often have little or no support for tasks, leaving developers to write custom tools to diagnose performance of their codes. In distributed task-based programming models, programs are written in a global data view, requiring custom facilities to represent data. This makes it cumbersome to port an application from MPI to a task-based representation. Also, specifying constraints for mapping work to distributed resources can be cumbersome.

In summary, some pros and cons of tasks are as follows.

Pros:
- Automatic task execution order / dependency analysis
- (Distributed) Automatic inter-node data transfer and mapping of work

Cons:
- Less straightforward than loops for many applications
- More difficult to diagnose performance issues
- Cumbersome to port applications from loops to tasks, esp. distributed

1.2 Objectives of this Dissertation

The goal of this dissertation is to explore the current state-of-the-art in task-based programming in the context of scientific applications. The application that we focus on, Minimod, is a mini-app developed by TotalEnergies. Minimod is derived from an industrial code performing reverse-time migration (RTM), a common method of seismic imaging used in geophysics applications. Minimod solves the wave equation—a partial differential equation—using the finite-difference method, forming what is commonly called a stencil application. Since many scientific applications across a variety of domains rely on stencil computations, the conclusions obtained from Minimod are applicable to many other scientific applications (e.g., climate simulations, computational fluid dynamics, quantum chromodynamics).

There are few large-scale scientific applications written using task-based programming. This is especially true for established applications like reverse-time migration, for which traditional methods of parallelization have long been the norm. Therefore, there is much to learn about task-based programming in this context, particularly in terms of performance and programmer productivity.
1.3 Contributions and Major Findings

By porting Minimod to a number of different task-based programming models across architectures, this work contributes valuable lessons about task-based programming for scientific applications. To this point, few works have focused on applying task-based programming methods to scientific applications in general, and especially in the context of geophysics applications. We perform experiments on a broad range of programming paradigms: within-node tasking using OpenMP, tasks on GPUs using OpenMP offloading features, a hybrid MPI+OpenMP task implementation, and truly distributed tasks on CPUs and GPUs using the Legion parallel programming model. We also use a diverse set of architectures: CPUs with IBM POWER9, Intel, and A64FX architectures; and NVIDIA V100 and A100 GPUs.

Here we mention several major findings of this work. On the performance front, we include an analysis of task performance sensitivity to granularity, finding that our application seems to have an “optimal” task size across the architectures studied. We also find that OpenMP task dependencies have a significant effect on performance. We also obtain competitive performance between a distributed task-based implementation of our application compared to a traditional MPI-based implementation. Finally, we find that in a full RTM application we can successfully overlap communication and computation, significantly reducing time-to-solution.

Regarding portability, we observe differences in the performance of tasks between different architectures, particularly between static and dynamic work distribution, due to the different cache structure between CPUs. We also evaluate performance of loops and tasks on A64FX, a new architecture rising in popularity due to the Fugaku supercomputer, and discover an issue running the task-based version of Minimod on one compiler for that platform.

Finally, looking at productivity, we remark on the limitations of the OpenMP depend clause, and difficulties identifying bugs in OpenMP task-based applications. We note difficulties in converting an application from a traditional MPI-based implementation to a task-based implementation, partly due to changing from a local view of the data to a global view.

1.4 Outline of Dissertation

The remainder of this dissertation is organized as follows. In Chapter 2 we provide some background on task-based programming, and introduce some task-based programming models and algorithms commonly targeted with task-based approaches. In Chapter 3 we introduce the motivating example for the remainder of the disser-
tation, Minimod. Chapters 4 to 6 explore using porting Minimod to use OpenMP
tasks, both on CPU and GPU. Chapters 7 and 8 explore performance of tasks on two
A64FX platforms, Ookami and Fugaku, including an attempt at creating a hybrid
MPI+OpenMP task version of Minimod.

Chapters 9 and 10 introduce distributed task-parallel versions of Minimod using
the Legion programming system, on CPUs and GPUs. Chapter 11 forms a complete
reverse time migration implementation using Legion, and demonstrates preliminary
improved performance using tasks to overlap communication and computation. Fi-
nally, Chapter 12 provides concluding remarks and explores potential next steps.
Chapter 2

Task-based Programming

In this section, we give a short introduction to task-based programming, including considerations when using them. We then describe some of the task-based frameworks that have been developed.

2.1 Introduction

Task-based programming is an approach to parallel programming in which a program is written as a collection of units of work called tasks. Each task is executed sequentially, but multiple tasks can potentially be executed simultaneously, subject to ordering constraints provided by the developer. The set of tasks and dependencies between them can be represented as a directed acyclic graph (DAG).

Compared to traditional methods of parallelization (e.g., MPI), task-based programming methods potentially offer the ability to alleviate load imbalances and thereby make better use of available parallel resources. They can also allow for overlap between computation and other overheads, like data transfer or disk I/O. Task-based programming models have been around for several years (e.g., Cilk [10]), but are recently gaining in popularity as architectures become increasingly diverse.

Task-based programming models can be grouped into shared-memory models and distributed-memory models. Shared-memory models and their runtime systems manage the problem of distributing work to CPU cores within a node, and to accelerators such as GPUs. Distributed-memory models additionally handle the challenge of scheduling tasks across nodes in a cluster connected by a network.
2.2 Considerations of task-based programming

Compared to traditional loop-based programming models, there are several performance and productivity issues to take into account when developing task-based programming models and applications that use them. Here, we list several of these considerations.

2.2.1 Task granularity

If there are too many tasks (i.e., the execution time of the tasks is too small), the runtime system may not be able to schedule tasks as quickly as they are executed. If this is the case, the task-scheduling runtime system itself may become the performance bottleneck in an application.

2.2.2 Task dependencies

Typically, tasks are not independent; rather, some tasks need, as input, the output of another task. This dependence relationship can be represented as a DAG. Most task-based frameworks provide a way for the user to express these dependence relationships to the runtime system. The runtime system then ensures the tasks are executed in an order that respects the dependencies. The runtime system can also handle making the needed input data available to a task before it runs, potentially transferring data between nodes as needed.

2.2.3 Task priorities

In some applications, certain tasks are more “important” than others. Many frameworks include the ability to assign priorities to tasks. Priorities can be used to reduce the total runtime of an algorithm by giving higher priority to tasks along the critical path.

2.2.4 Scheduling and load balancing

Scheduling refers to the placement of tasks and data on computing resources (nodes, threads, accelerators, etc.). In large-scale applications spanning several nodes, consideration of data locality becomes extremely important. Ideally, each task will always be executed on the node that contains the data it uses. In practice, this isn’t always achievable, and finding a scheduling policy that minimizes the amount of data transfer is an important problem.
Load balancing refers to allocating work evenly among the different processing units so that idle time is minimized. One of the big advantages of task-based programming models is the ability to allow the runtime system to perform dynamic load-balancing through *work stealing*, whereby a node which is idle can take work from the queue of another node which is busy.

## 2.3 Survey

### 2.3.1 Legion

Legion [7] is a parallel programming system developed at Stanford. It is a data-centric programming system: users abstractly describe the structure of the program data and the tasks that operate on the data. Legion is able to automatically extract parallelism from this description. Legion automates the process of moving data and tasks around on complex heterogeneous architectures, eliminating the need to do this manually and reducing bugs.

Legion provides a mapper interface enabling users to manually control placement of tasks and data. This allows users to take advantage of application specific knowledge to improve performance. Legion provides a default mapper as a starting point. The choice of mapping decisions is orthogonal to program correctness; that is, a Legion program will produce the same result no matter which mapper is used. Legion programs can be executed across multiple nodes using GASNet.

Legion’s low-level runtime system is written in C++. Writing a C++ program that directly targets this runtime system requires a lot of boilerplate code and is quite tedious; Appendix A gives an example of computing DAXPY in Legion. Thus, Legion also has a Lua-based language called Regent, which makes it easier to target the Legion runtime system. Regent code is compiled, using a backend called Terra, to LLVM bytecode, which is then compiled to native machine code using LLVM.

Execution begins at the “top-level task,” which can in turn spawn subtasks. The subtasks are created dynamically at runtime by the user’s code.

### 2.3.2 Chapel

Chapel [13] is a modern programming language developed at Cray. Its aim is to make parallel programming more accessible. The makers of Chapel see a disconnect between “traditional” HPC languages (C, Fortran, MPI), and new HPC developers who are more accustomed to high-level languages like Python. Chapel’s goal is to
enable productive parallel programming in a high-level language without sacrificing performance.

Chapel offers constructs for both task and data parallelism. Chapel’s task parallelism facilities consist of the `begin`, `cobegin`, `coforall` statements. Each of these statements are used to launch a new task independent of the current thread of execution.

Chapel tasks support enforcing execution order using sync variables. A sync variable has two states, `empty` and `full`. Reading from a sync variable that is `empty`, or writing to a sync variable that is `full`, will result in the thread blocking until the state changes. This can be used to implement a form of task ordering. Fig. 2.1 shows an example of computing a dot product in Chapel using tasks. An array of `single` variables is used to control the order of task execution. (The reduction could also be implemented using an atomic variable.)

### 2.3.3 XcalableMP

XcalableMP [37] is another framework attempting to improve productivity of HPC programming. It consists of a set of directives for C and Fortran. XcalableMP directives are designed to be able to be added into existing sequential code to enable parallelization of the code with minimal modification.

XcalableMP offers two programming models: a global-view model and a local-view model. The global-view model is higher-level, allowing the programmer to describe the parallelization structurally and let the implementation handle explicit communication. The global-view allows parallelization of existing sequential code easily. In the local-view model, the programmer explicitly specifies communication between nodes, allowing a finer degree of control.

The current version of the XcalableMP specification (1.4) includes the `task` construct for creating tasks. Compared to the tasking facilities in the other models, XcalableMP has a severe restriction: the programmer must specify on which node(s) a task executes, rather than letting the runtime system automatically map the task to a free execution unit. This leads to a relatively static and inflexible distribution of work. This is addressed by a new feature in the upcoming XcalableMP 2.0 spec called a tasklet.

Fig. 2.2 shows an example of computing a dot product using tasks. Since the XMP `loop` construct implicitly creates a set of tasks, we do not need to use the `task` construct explicitly.

XcalableMP can be used in conjunction with MPI and OpenMP, which further aids introducing XcalableMP into existing code.
\texttt{const D: domain(1) = \{1..(num\_tasks*task\_size)\};
var x: [D] real;
var y: [D] real;

var sumpart$: [1..num\_tasks] single real;

// Set x and y

// Partial sum tasks
for ti in 1..num\_tasks {
    begin {
        var mysum: real = 0.0;
        for i in ((ti-1)*task\_size+1)..(ti*task\_size) {
            mysum += x[i]*y[i];
        }
        sumpart$[ti] = mysum;
    }
}

// Reduction task
begin {
    var sum: real = 0.0;
    for ti in 1..num\_tasks {
        // The following statement will be executed after task ti above
        // has set sumpart$[ti].
        sum = sum + sumpart$[ti];
    }
}

Figure 2.1: Dot product in Chapel using tasks
```c
#define task_size 4
#define num_tasks 4

#pragma xmp nodes p[num_tasks]
#pragma xmp template t[task_size*num_tasks]
#pragma xmp distribute t[block] onto p

double sum = 0.0, x[num_tasks*task_size], y[num_tasks*task_size];
#pragma xmp align x[i] with t[i]
#pragma xmp align y[i] with t[i]

// Set x and y ...

#pragma xmp loop on t[i] reduction(+:sum)
for (int i = 0; i < num_tasks*task_size; i++) {
    sum += x[i]*y[i];
}
```

Figure 2.2: Dot product in XcalableMP using tasks

### 2.3.4 OpenMP

OpenMP [55] is a parallel programming framework for shared memory parallelism and accelerators (e.g., GPUs). OpenMP is the de-facto framework for shared memory parallelism.

OpenMP introduced tasks in version 3.0. One can use the `task` directive to create a new task. OpenMP 4.0 added automatic dependency analysis to tasks, such that the compiler can automatically determine the order of task execution based on user supplied data dependencies. Like OpenMP threads in general, OpenMP tasks are local to the node on which they are created.

OpenMP programs are C, C++, or Fortran programs, with added directives to enable parallelization. The `task` directive can be used within a parallel region to immediately create a new task and schedule it for execution. A task can use the `depend` clause to list data dependencies that the compiler uses to control order of execution.

Fig. 2.3 shows an example of computing a dot product using OpenMP tasks. In contrast to the Chapel example, here we must spawn `num_tasks` to perform the reduction. Each of these tasks adds the corresponding part of `sum_part` to the final sum. In this case, we could have also used an OpenMP `critical` directive to directly add to the final sum from each task.

The OpenMP `depend` clause currently has a few shortcomings that limit flexibility. First, dependency-induced ordering is only honored among sibling tasks. Sibling
double sum = 0.0, x[num_tasks*task_size], y[num_tasks*task_size];

// Set x and y ...

double sum_part[num_tasks] = {0.0};

// Partial sum tasks
for (int ti = 0; ti < num_tasks; ti++) {
    #pragma omp task depend(in: x[ti*task_size:task_size], \
        y[ti*task_size:task_size]) depend(out: sum_part[ti])
    {
        sum_part[ti] = 0.0;
        for (int i = task_size*ti; i < task_size*(ti+1); i++) {
            sum_part[ti] += x[i]*y[i];
        }
    }
}

// Reduction tasks
for (int ti = 0; ti < num_tasks; ti++) {
    #pragma omp task depend(in: sum_part[ti]) depend(inout: sum)
    {
        sum += sum_part[ti];
    }
}

Figure 2.3: Dot product in OpenMP using tasks
tasks are defined as tasks generated by the same parent task. For example, if a task generates subtasks, and if those subtasks specify dependencies, the dependencies will only influence the ordering among these subtasks, and not, e.g., the subtasks of a different task. Second, each unique item in the dependency lists of sibling tasks must refer to a memory region that is disjoint from all others. So, for example, you could not have one task depend on a portion of an array and a sibling task depend on the entire array.

2.3.5 Others

HPX [32] is a programming model for shared-memory and distributed-memory systems. PaRSEC [11] supports task parallelism through two paradigms: “parameterized task graph”, a static expression of tasks in the application, and a dynamic alternative, “dynamic task discovery” [30]. Dask [17] is a Python-based library for scaling computations with a focus on data science applications. Dask makes two APIs available: one that clones interfaces from popular data science libraries (e.g., NumPy, pandas, scikit-learn) to form a drop-in replacement in applications, and a more general futures interface allowing application to express arbitrary task-like parallelism.

Kokkos [78] is a programming model for shared-memory systems focusing on transparently handling differing data layouts between heterogeneous architectures (e.g., between CPUs and GPUs). Kokkos has preliminary support for task-based parallelism but it is still a work in progress.

2.3.6 Comparision

A recent attempt to compare performance of different task-based programming models is Task Bench [74]. Task Bench attempts to evaluate multiple task-based programming models, including most of the ones listed here, on competitive grounds using a synthetic set of benchmarks representing common communication patterns. For example, Fig. 2.4 (reproduced from [74]) shows the relative performance of several task-based models for a communication pattern representing a stencil.

2.4 Example Task-based Algorithms

This section contains a description of two commonly used scientific algorithms and how they can be implemented as tasks.
2.4.1 Conjugate Gradient Method

The conjugate gradient algorithm [29] solves a system of the form $Ax = b$, where $A$ is a given $n \times n$ symmetric positive definite (SPD) matrix, $b$ is a given $n$-vector, and $x$, an $n$-vector, is the solution to the equation.

A pseudocode outline of the algorithm is presented in Algorithm 1.

The most expensive operation is the matrix-vector multiplication, which, for dense matrices and vectors, is a $\Theta(n^2)$ operation.

A simple way to represent the algorithm as tasks is to decompose the matrix into blocks by rows and the vector into corresponding chunks. An example of this decomposition is shown in Fig. 2.5 where a matrix is decomposed into four blocks, each block containing the same number of rows. In this figure, the colors red, green, orange, and blue represent tasks.

Using this task decomposition, Algorithm 1 can be used, which each operation being performed on each block of the matrix/vector involved in the operation. For example, matrix-vector multiplication can be expressed as in Algorithm 2, using the row-based task decomposition in Fig. 2.5.

A more expressive task decomposition is to decompose the matrix into 2-D tiles, and the vectors into corresponding blocks. In general, there is a trade-off between expressing more task parallelism to the runtime system and minimizing runtime system execution overhead.
Data: $A$, an $n \times n$ SPD matrix; $b$, an $n$-vector

Result: $x$, an $n$-vector

1. $r_0 := b - Ax_0$;
2. $p_0 := r_0$;
3. $k := 0$;
4. while not converged do
5. \[ \alpha_k := \frac{r_k^T r_k}{p_k^T Ap_k}; \]
6. \[ x_{k+1} := x_k + \alpha_k p_k; \]
7. \[ r_{k+1} := r_k - \alpha_k A p_k; \]
8. if converged then
9. \[ \text{exit loop} \]
10. end
11. \[ \beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}; \]
12. \[ p_{k+1} := r_{k+1} + \beta_k p_k; \]
13. \[ k := k + 1; \]
14. end
15. Result is $x_{k+1}$;

\textbf{Algorithm 1:} Conjugate gradient algorithm

![Figure 2.5: Matrix decomposition for conjugate gradient algorithm](image)
2.4.2 Cholesky Decomposition

The Cholesky decomposition decomposes a SPD matrix into a product of a lower-
triangular matrix and its transpose:

\[ A = LL^T \]

In performance codes (e.g., LAPACK), the Cholesky decomposition is often com-
puted using a block algorithm, where the matrix \( A \) is split into 2-D tiles. The block
Cholesky decomposition algorithm given by Golub[24] is shown in Algorithm 3.
Algorithm 3 is readily expressed in a task-based framework, where each block is computed by a task. Without the help of a tool to manage parallelism, it is easy to miss fine-grain parallelism opportunities. For example, after $L_{11}$ is computed (directly using Cholesky()), $L_{21}, L_{31}, \ldots, L_{p1}$ can be computed in parallel (since they depend only on $L_{11}$). Furthermore, once $L_{21}$ is computed, $L_{22}$ can be computed, and $L_{i2}$ depend only on $L_{22}, L_{21}$, and $L_{i1}$, for $3 \leq i \leq p$. Other columns are similar. This complicated dependency structure can be represented as a DAG, as in Fig. 2.6.
Chapter 3

Introduction to Minimod

Seismic imaging refers to the process of mapping the subsurface of the Earth. It is useful for many reasons. For example, energy companies use seismic imaging to locate undiscovered oil and gas reserves or potential sites for CO2 sequestration. Reverse-time migration (RTM) is a popular method for seismic acoustic imaging. It has high accuracy, but it is relatively computationally demanding.

RTM consists of two passes: a forward pass and a reverse pass. In the forward pass, the propagation of acoustic waves is modeled as the waves are emitted from the source. In the backward pass, the recording of waves from a receiver is used to propagate waves backward from the receiver into the earth. The correlation between the forward and backward waves is used to reconstruct an image of the subsurface.

Minimod is a proxy application that simulates the propagation of waves through the Earth models, by solving a Finite Difference (FD) discretized form of the wave equation. It is designed and developed by TotalEnergies Exploration and Production Research and Technologies [42]. Minimod extracts the core stencil computations used in industrial production simulations. The proxy application purpose is to benchmark new HW platforms for technology tracking. Minimod is self-contained and designed to be portable across multiple compilers. The application suite provides both non-optimized and optimized versions of computational kernels for targeted platforms. The main purpose is benchmarking of emerging new hardware and programming technologies. Non-optimized versions are provided to allow analysis of pure compiler-based optimizations. Minimod is currently not publicly available; however, the plan is to eventually make it available to the community as open-source software.

In this work, we study one of the kernels contained in Minimod, the isotropic propagator in a constant-density domain [59]. For this propagator, the wave equation
PDE has the following form:

\[
\frac{1}{V^2} \frac{\partial^2 u}{\partial t^2} - \nabla^2 u = f, \tag{3.1}
\]

where \( u = u(x, y, z) \) is the wavefield, \( V \) is the Earth model (with velocity as rock property), and \( f \) is the source perturbation. The equation is discretized in time using a second-order centered stencil, resulting in the semi-discritized equation:

\[
u^{n+1} - Q u^n + u^{n-1} = (\Delta t^2) \ V^2 f^n, \text{ with } Q = 2 + \Delta t^2 V^2 \nabla^2. \tag{3.2}
\]

Finally, the equation is discretized in space using a 25-point stencil in 3D space, with four points in each direction as well as the centre point:

\[
\nabla^2 u(x, y, z) \approx \sum_{m=0}^{4} c_{xm} [u(i + m, j, k) + u(i - m, j, k)] + c_{ym} [u(i, j + m, k) + u(i, j - m, k)] + c_{zm} [u(i, j, k + m) + u(i, j, k - m)] \tag{3.3}
\]

where \( c_{xm}, c_{ym}, c_{zm} \) are discretization parameters.

A simulation in Minimod consists of solving the wave equation at each timestep for some number of timesteps. Pseudocode of the algorithm is shown in algorithm 4. We apply a Perfectly Matched Layer (PML) \[9\] boundary condition to the boundary regions. The resulting domain consists of an “inner” region where Equation 3.2 is applied, and the outer “boundary” region where a PML calculation is applied, as shown in Figure 3.1.

```
Data: f: source
Result: u^n: wavefield at timestep n, for n ← 1 to T
1 u^0 := 0;
2 for n ← 1 to T do
3     for each point in wavefield u^n do
4         Solve Eq. 3.2 (left hand side) for wavefield u^n;
5     end
6     u^n = u^n + f^n \text{ (Eq. 3.2 right hand side)};
7 end
```

**Algorithm 4:** Minimod high-level description
We note that the stencil does not have a uniform computational intensity across the domain: the PML regions require more calculations than the inner regions. This suggests an inherent load imbalance that may be amenable to improvement with tasks. Furthermore, a full simulation includes additional kernels, such as I/O and compression. These additional kernels are not evaluated in this study but will be added in the future.

### 3.1 Related Work

A great amount of research effort has been devoted to optimizing stencil computations to achieve higher performance. For example, Nguyen et al. [49] introduced higher dimension cache optimizations, and de la Cruz et al. proposed the semi-stencil algorithm [15] which offers an improved memory access pattern and efficiently reuses accessed data by dividing the computation into several updates. New hardware technologies (such as GPUs, FPGA, etc.) have motivated researchers to further investigate how to re-purpose stencil algorithms to take advantage of their unique characteristics, following two major approaches: programming models and pure algorithmic optimization. On the programming models front, in 2012, Ghosh et al. [23] analyzed the performance and programmability of three high-level directive-based GPU programming models (PGI, CAPS, and OpenACC) on an NVIDIA GPU on kernels for reverse time migration (RTM). In 2017, Qawasmeh et al. [59] implemented an MPI + OpenACC approach for seismic modeling and RTM. In a separate line of research,
domain-specific languages (DSLs) and domain-specific parallel programming models and compiler optimizations for stencils have been proposed (e.g., [38, 28, 65]). Performance models have been developed for this computing pattern (see [16]), and the kernel has been ported to a variety of platforms ([3, 4]). Most recently, Sai et al. [70] studied high-order stencils with a manually crafted collection of implementations of a 25-point seismic modeling stencil in CUDA for the latest GPU hardware. Along this line of hardware-oriented stencil optimization, Matsumura et al. ([40]) proposed a framework (AN5D) for GPU stencil optimization, obtaining remarkable results.

On the algorithmic optimization path, spatial and temporal blocking has been proposed ([81, 21]). A further example is the semi-stencil algorithm proposed by de la Cruz et al. [15], which offers an improved memory access pattern and data reuse. Promising results are also provided by a higher dimension cache optimization introduced by Nguyen et al. [49] which accommodates both thread-level and data-level parallelism.
Chapter 4

Evaluation of OpenMP CPU tasks in Minimod

Note: this material appears in [60] and is reproduced here in modified form.

4.1 Introduction

Many industrial and scientific applications use stencil computation for solving PDEs discretized with Finite Difference (FD) or Finite Volume (FV) methods. These can range from geophysics to weather forecasting models [76]. Improving performance is of utmost interest since this facilitates faster decision making as well as more opportunities to explore further scientific questions. Optimization of stencil computation has been addressed in the past aplenty (see Section 4.2) from many different angles, e.g. low-level optimization, parallelism at different levels, and DSLs.

In this work, we create OpenMP task-based versions of an industrial stencil-based seismic modeling code and compare performance of the task-based versions to traditional loop-parallelized versions of the code. The motivation of this work is to explore how task-based programming models and task parallelism can support the stencil computation pattern in practice.

Our main contributions are the following: (1) we introduce task parallelism to a stencil code in a proxy for an industrial application; (2) we test our task-based stencil code on several architectures and compilers; and (3) we analyze its behavior and compare results of the task-based stencil with several variants written using parallel loops.

The chapter is organized as follows: Section 4.2 describes relevant literature works and contributions. Section 4.3 details the application code structure and how it was
ported to task parallelism. In Section 4.4, the experimental environment and results are presented. Section 4.5 and 4.6 provide discussion and conclusions.

4.2 Related Work

In recent years, task-based parallel programming has been recognized as a promising approach to improve performance in scientific applications such as stencil-based algorithms. For example, in [46], Moustafa et al. illustrated the design and implementation of a FD method-based seismic wave propagation simulator using PaRSEC, although their implementations are limited by scalability and only implemented on CPUs.

Researchers have been working on exploring the advantages of tasking in OpenMP since tasks were introduced in version 3.0. Right after its release, Virouleau et al. [80] evaluated OpenMP tasks and dependencies with the KASTORS benchmark suite. Duran et al. [19] evaluated different OpenMP task scheduling strategies with several applications. Rico et al. [68] provided insights on the benefits of tasking over the work-sharing loop model by introducing tasking to an adaptive mesh refinement proxy application. Atkinson et al. [5] optimized the performance of an irregular algorithm for the fast multipole method with the use of tasks in OpenMP. Vidal et al. [79] evaluated the task features of OpenMP 4.0 extensions with the OmpSs programming model. Recently, Raut et al. [61] (the basis of this chapter) implemented OpenMP task parallelism into the industrial seismic modeling kernel used in the current chapter, and presented taskified kernels which are competitive with traditional OpenMP-augmented loops.

Several programming systems supporting tasks have been proposed, some of which (e.g., OpenMP) focus on shared-memory systems. Cilk [10] is an early programming API supporting tasks using spawn keyword. Intel Thread Building Blocks [66] and Kokkos [12] also support shared-memory task parallelism. StarSs [57] is a task-based framework for multi/many-core systems using a pragma syntax. OmpSs [20] is an attempt to extend OpenMP with tasking features using StarSs runtime.

4.3 Code Structure and Taskification of Minimod

In this section, we describe the code structure of Minimod and explain how it has been ported to a version that makes use of OpenMP tasks. The most computationally expensive component of Minimod (algorithm 4) is the computation of the wavefield
for each point. The (original) serial version of the code has the structure shown in algorithm 5.

```
Data: \( u^{n-1}, u^{n-2} \): wavefields at previous two timesteps
Result: \( u^n \): wavefield at current timestep
1 for \( i \leftarrow \text{xmin} \) to \( \text{xmax} \) do
2     if \( i \geq x3 \) and \( i \leq x4 \) then
3         for \( j \leftarrow \text{ymin} \) to \( \text{ymax} \) do
4             if \( j \geq y3 \) and \( j \leq y4 \) then
5                 // Bottom Damping (i, j, z1...z2)
6                 // Inner Computation (i, j, z3...z4)
7                 // Top Damping (i, j, z5...z6)
8             else
9                 // Back and Front Damping (i, j, zmin...zmax)
10             end
11         end
12     else
13         // Left and Right Damping (i, ymin...ymax, zmin...zmax)
14     end
15 end
```

**Algorithm 5:** Wavefield solution step

We evaluate several different configurations for the parallelization of this code, using both OpenMP parallel loops and tasks. In the \( x \)-loop versions, we simply apply an `omp parallel for` directive to the \( x \)-loop on line 1 of algorithm 5. The OpenMP schedule is selected at runtime; we test the static, dynamic, and guided OpenMP schedules in this study.

In addition to simply looping over the \( x \)-dimension, we also evaluate the effect of loop blocking in the \( x \)-\( y \) plane. See Figure 3.1. In the blocked version, we apply OpenMP loop parallelism to the 2-D loop nest over \( x \)-\( y \) blocks. Again, we evaluate the static, dynamic, and guided schedules.

In the task-based configurations, we insert an `omp parallel master` region surrounding the entire timestep loop (before line 2 in algorithm 4). Then, in the wavefield solution step we generate tasks representing parallel units of work. The OpenMP `depend` clause is used to manage dependencies between timesteps. In this stencil computation, the computation of each block depends on its neighbors from the previous timestep.
The OpenMP depend clause does not support overlapping array sections as dependencies. The most natural way to express dependencies between the regions is to list, in array section form, the specific array elements that each block depends on. However, this would result in overlapping dependency regions and is therefore not supported. Instead, in our implementation we simply choose one element of each neighboring block to include in the dependency list. This workaround, however, is limited to simple dependence patterns. For example, it is not possible to use more blocks (smaller block size) in the PML regions than in the inner region, because each inner block would depend on multiple PML blocks. OpenMP 5.0 supports using iterators in the depend clause, which provides some additional flexibility; however, iterators are not supported in any compilers we tested.

We evaluate the following configurations in this chapter:

- **Loop x static/dynamic/guided**: an OpenMP parallel for loop is applied to the x loop in line 1 of Algorithm 5. A static/dynamic/guided schedule is used.
- **Loop xy static/dynamic/guided**: Uses blocking in the x and y dimensions. An OpenMP parallel for loop is applied to the 2-D loop nest over x-y blocks. (A \texttt{collapse(2)} is used to combine the two loops). A static/dynamic/guided schedule is used. Several different block sizes are evaluated.
- **Tasks xy**: Each x-y block is a task. OpenMP’s \texttt{depend} clause is used to manage dependencies between timesteps.
- **Tasks xy nodep**: Same as above, but OpenMP dependencies are not used.

In order to prevent a race condition, an explicit task synchronization point (\texttt{taskwait}) is added at the end of the timestep (i.e., before line 7 of Algorithm 4).

An alternative approach, not evaluated here, would be to apply a \texttt{taskloop} construct to the loops, generating one task for each chunk of iterations (with configurable size). Currently, the \texttt{taskloop} construct does not support dependencies, so an explicit task synchronization would be required, as in Tasks xy nodep.

Our application is not currently NUMA-aware, which hurts performance on NUMA architectures, including the nodes used in this study. The conventional NUMA awareness for OpenMP tasks can be achieved with the \texttt{affinity} clause of OpenMP 5.0 [35]; however, to the best of our knowledge, this clause is not supported on any publicly available compilers as of the time of writing. (In [35], an LLVM runtime with preliminary support of task affinity is implemented. We are currently evaluating our application with this runtime.) In our application, all data is allocated and initialized by a single thread and so will likely reside on a single NUMA domain.
<table>
<thead>
<tr>
<th>Computer</th>
<th>Hardware and Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summit</td>
<td></td>
</tr>
<tr>
<td>CPUs</td>
<td>2x IBM Power9</td>
</tr>
<tr>
<td>CPU cores</td>
<td>44 (22 per CPU)</td>
</tr>
<tr>
<td>Memory</td>
<td>512 GB</td>
</tr>
<tr>
<td>L3</td>
<td>10 MB (per two cores)</td>
</tr>
<tr>
<td>L2</td>
<td>512 KB (per two cores)</td>
</tr>
<tr>
<td>L1</td>
<td>32+32 KB</td>
</tr>
<tr>
<td>Device fabrication</td>
<td>14nm</td>
</tr>
<tr>
<td>Software</td>
<td>LLVM 9.0</td>
</tr>
<tr>
<td>Cori</td>
<td></td>
</tr>
<tr>
<td>CPUs</td>
<td>2x Intel Xeon E5-2698v3</td>
</tr>
<tr>
<td>CPU cores</td>
<td>32 (16 per CPU)</td>
</tr>
<tr>
<td>Memory</td>
<td>128 GB</td>
</tr>
<tr>
<td>L3</td>
<td>40 MB (per socket)</td>
</tr>
<tr>
<td>L2</td>
<td>256 KB</td>
</tr>
<tr>
<td>L1</td>
<td>32+32 KB</td>
</tr>
<tr>
<td>Device fabrication</td>
<td>22nm</td>
</tr>
<tr>
<td>Software</td>
<td>LLVM 10.0</td>
</tr>
<tr>
<td>SeaWulf</td>
<td></td>
</tr>
<tr>
<td>CPUs</td>
<td>2x Intel Xeon Gold 6148</td>
</tr>
<tr>
<td>CPU cores</td>
<td>40 (20 per CPU)</td>
</tr>
<tr>
<td>Memory</td>
<td>192 GB</td>
</tr>
<tr>
<td>L3</td>
<td>28 MB (per socket)</td>
</tr>
<tr>
<td>L2</td>
<td>1024 KB</td>
</tr>
<tr>
<td>L1</td>
<td>32+32 KB</td>
</tr>
<tr>
<td>Device fabrication</td>
<td>14nm</td>
</tr>
<tr>
<td>Software</td>
<td>LLVM 11.0 (git 3cd13c4)</td>
</tr>
</tbody>
</table>

Table 4.1: Hardware and software configuration of the experimental platforms.

### 4.4 Evaluation

The different versions of Minimod are evaluated on Summit (a supercomputer with IBM POWER9 architecture) and Cori and SeaWulf (supercomputers with an Intel architecture).

#### 4.4.1 Experimental Setup

Summit [51] is a computing system at the Oak Ridge Leadership Computing Facility (see Table 4.1 top panel). Each node also has 6 NVIDIA V100 GPUs; however, we
do not use GPUs in this study. We use 42 OpenMP threads in all experiments with each thread bound to a physical core.

Cori [47] is a computing system at the National Energy Research Scientific Computing Center (NERSC) (see Table 4.1 middle panel). We perform experiments on Haswell nodes of Cori. 32 OpenMP threads on the Haswell nodes were used, each thread bound to a physical core (using OMP_places=cores and OMP_PROC_BIND=true).

SeaWulf is a computing system at Stony Brook University. Details are given in Table 4.1 (bottom panel). In each run, we use 40 OpenMP threads (one per physical core) with each thread bound to a physical core.

Each simulation is run with grid sizes between $64^3$ (64 in each of the three dimensions) and $1024^3$. Sizes $512^3$ and $1024^3$ are reported in this chapter. Results with the LLVM compiler on each computer are reported in this chapter. Cache statistics were collected using the Perf and HPCToolkit [41] profilers. Execution times are averaged over three trials on Summit and SeaWulf. We were unable to compute a three-run average on Cori due to lack of availability; however, the application shows little variation in run time on the other machines, so it likely would make little difference.

### 4.4.2 Results

Execution times for each configuration from Section 4.3 on all three platforms are shown in Figure 4.1. For each of the xy-blocked configurations, the time shown is for the block size that gives the lowest execution time for each configuration. On Cori, poor performance is seen from “Loop x static” as compared to other configurations. Performance among the xy-blocked configurations are generally quite similar.

To understand the relative performance and how it relates to the architecture used, we gathered cache use statistics for each configuration. Table 4.2 shows the L3 miss rate for each configuration on Summit, Cori, and SeaWulf, respectively. On Summit, the miss rate is significantly lower for static configurations than for the other configurations. On Cori and SeaWulf, the L3 miss rate is highest for “Loop x static”, and relatively similar among all xy-blocked configurations.

Figure 4.2 shows the effect of block size on execution time for each of the xy-blocked configurations. The given block size is the size of both the x and y dimensions of each block/task. We frequently see that at a small block size of $4^2$, “Tasks xy” does significantly worse than other configurations. Also noteworthy is that at larger block sizes, “Tasks xy” usually outperforms “Tasks xy nodep”, showing the benefit of fine-grained synchronization.

We also ran experiments with other compilers (IBM XL 16.1.1 on Summit, and
Figure 4.1: Execution time (in seconds), top panel Summit, mid panel Cori and bottom panel SeaWulf.
Figure 4.2: Effect of block size on execution time using the LLVM compiler on Summit (top), Cori (middle), and SeaWulf (bottom).
Table 4.2: L3 miss rate [%] on each computer for each configuration.

Intel 19 on Cori and SeaWulf). The general trends discussed here (for the LLVM compiler) also apply to other compilers, indicating that these conclusions are intrinsic to the code and architecture. Due to space constraints, results with the other compilers are not shown here.

## 4.5 Discussion

As shown in Table 4.2, the L3 miss rate on Summit (POWER9 architecture) is lower for static-schedule configurations than other configurations, while for Cori and SeaWulf (Intel architectures) this relationship does not hold. To understand why, we must examine the cache hierarchies of these architectures. On the POWER9 architecture (Summit), the L3 cache is shared between each pair of cores only (Table 4.1). With a static schedule, the assignment of domain regions to threads does not change between timesteps, and data resident in the L3 cache will be reused at subsequent timesteps. With non-static schedules (including tasks), the assignment of domain regions to threads is arbitrary and can change at each timestep, introducing L3 cache misses (and an expensive fallback to main memory) when a region moves to a different pair of physical cores. On Intel architectures (Cori and SeaWulf), the L3 cache is shared on the entire socket, so movement of regions between timesteps does not cause L3 cache misses unless the movement is between sockets.

A notable trend in the block size plots (Figure 4.2) is that for very small block sizes (i.e., $4^2$), there is a large overhead seen in “Tasks xy”. This sensitivity is usually not seen in the other configurations (although on SeaWulf a similar time
increase occurs in the “Tasks xy nodep” configuration). This indicates that the LLVM OpenMP runtime has a significant overhead associated with scheduling small tasks. The difference between “Tasks xy” and “Tasks xy nodep” suggests that there is also a significant overhead associated with handling the dependencies between tasks for fine-grained synchronization. The bulk synchronization of “Tasks xy nodep” (task synchronization at the end of each timestep) has less overhead.

Most of the block size experiments in Figure 4.2 show that there is a “minimum point”, usually around a square block size of 16-32, where the execution time is minimized. In general, there is a trade-off with respect to choosing a block size. Small block sizes expose more parallelism to the runtime, resulting in more opportunities for load balancing. However, as each block is a task that must be scheduled for execution, small block sizes incur increased runtime task scheduling overhead. It is interesting to see that the minimum point for block size is relatively similar across computers in Figure 4.2.

Especially at larger block sizes, we see a significant improvement of “Tasks xy” over “Tasks xy nodep”. This shows potential for improvement of the fine-grained synchronization provided by task dependencies. However, this improvement is diminished at smaller block sizes. If the overheads of task dependency resolution could be reduced, this approach might also benefit smaller block sizes.

4.6 Conclusions

In this chapter, the Minimod application was ported to use OpenMP tasks. Even for this relatively regular stencil application, task-based parallelism is competitive with traditional loop-based parallelism, and is even better in some experiments. This is a promising result for the effectiveness of OpenMP tasking.

A key finding of this chapter is that the movement of domain region computations between timesteps is more expensive on the POWER9 architecture than on Intel architectures due to the difference in L3 cache hierarchy between them (Section 4.5). This stresses the importance of locality-aware task scheduling and suggests that the optimal policies for such a scheduler may be architecture-dependent. The affinity clause introduced in OpenMP 5.0 may help improve the locality of tasks, increasing performance. The OpenMP metadirective, also introduced in version 5.0, could potentially help set scheduling parameters for different target platforms.

As discussed in Section 4.5, our results indicate the potential for decreasing the overhead associated with handling task dependencies. However, task dependencies currently also have a lack of expressivity (see Section 4.3). Increasing the expressivity without increasing overhead may prove difficult.
More research is needed to pinpoint the causes of these performance characteristics. For example, we plan to use a profiler to continue to explore OpenMP overheads and barriers for each configuration. We would also like to better understand the extent to which tasks move between threads over the simulation. We hope to see better support for tasks from performance tools.

In future work, this code will be ported to GPUs using OpenMP 4.0+ offloading features, including using tasks to coordinate the work of multiple GPUs. We would also like to extend the code to run on multiple nodes. One possibility is to use MPI to coordinate OpenMP tasks between nodes. We will also add more kernels to Minimod to form a more complete seismic imaging application; in doing so, we expect to further exploit the benefits of task-based parallelism.
Chapter 5

Porting Minimod to GPUs using OpenMP

This chapter presents results obtained from porting the version of Minimod described in the previous chapter to GPUs using OpenMP. Part of the material from this chapter is published in [39].

5.1 Introduction

OpenMP [55] is the de-facto standard programming model for shared-memory parallelism. OpenMP 4.0 added the ability to offload computation to GPUs and other accelerators using the \texttt{#pragma omp target} family of directives.

OpenMP’s accelerator offloading facilities were designed for offloading to a single accelerator in a straightforward manner. Offloading to multiple accelerators simultaneously (for example, multiple GPUs within a node) is more challenging. Some strategies for multi-GPU offloading and load balancing were explored in [34]. Here, we employ a static distribution of work onto GPUs with partitions along the \(x\) dimension (slowest varying).

5.2 Design of OpenMP-offload version of Minimod

The single-GPU offloaded version of Minimod is derived from the version in the original Fortran version of Minimod [42]. Compared to the CPU-based version described
for (int g = 0; g < nGPUs; g++) {
    #pragma omp task depend(...)
    #pragma omp target teams distribute parallel for device(g)
    for (...) {
        // Stencil computation
    }
}

// Halo exchange
for (int g = 0; g < nGPUs; g++) {
    // Left halo region: DtoH
    #pragma omp task depend(...)
    #pragma omp target update from(...) device(g)
    // Left halo region: HtoD
    #pragma omp task depend(...)
    #pragma omp target update to(...) device(g-1)
    // Repeat for the right halo regions ...
}

Figure 5.1: Simplified Minimod multi-GPU offloading and halo exchange workflow

in Chapter 3, the GPU version launches seven OpenMP-accelerated GPU kernels – one per region shown in Figure 3.1.

Minimod was adapted to support multi-device OpenMP offloading using target regions wrapped in OpenMP tasks (see Figure 5.1). The 3D grid used in Minimod is partitioned along the $X$-axis (i.e. sliced parallel to the $YZ$-plane), regardless of the number of devices it is running on. Therefore, the amount of halo data exchanged between the devices is only related to the size of the grid, since the area of the cross-section of the grid is always $dimY \times dimZ$.

5.3 Remote offloading plugin

Classically, the OpenMP offloading infrastructure is used for offloading computations to local accelerators on the device. However, nothing inherent to the specification prevents the accelerator devices from being located on other computers and accessed over the network. This was the idea behind the work by Patel and Doerfert[56]. In this work, a plugin was developed for the LLVM OpenMP runtime that exposes accelerators on different devices to one central ”client” running an OpenMP application. The remote accelerators are connected to ”servers” that accept requests from the clients for computation.

Using the plugin, applications that were originally designed to run on a single
node can be scaled to multiple nodes without any changes to the application source code. This is useful in several scenarios. One example is running an application on an unusually large domain without rewriting the application to use MPI or another distributed programming model.

5.4 Performance results

In [39] we obtained preliminary results using the remote offloading plugin with Minimod. Figure 5.2 shows strong scaling results using different combinations of GPUs. The notation is as follows: $N(x, y)$ is an experiment run using $y$ GPUs on $x$ nodes. (Each node has 4 GPUs.) As explained in [39], the remote offloading plugin with only runtime optimizations is referred to as Opt1; while plugin version Opt2 has all optimizations applied. Version Opt2L enables Client-side offloading, the Client to offload to its local GPUs directly (instead of go through the remote plugin). On the vertical axis, “Relative Run Time” is in comparison to a benchmark result in which four GPUs on a single node are used without using the remote-offloading plugin.

As the figure shows, although the optimized version of the remote offloading plugin perform better, using the plugin for this application does not result in any performance improvment, and in fact leads to a degradation in performance as more GPUs are added. Indeed, with this application it seems that the overhead of program

Figure 5.2: Strong scaling results of Minimod (lower is better)
serialization and data transfer through the plugin outweigh the benefits. This seems to be due to limitations of the plugin.

5.5 Conclusion

Although OpenMP can be used directly to accelerate computations on multiple GPUs, this approach is not widely used, and doing so is cumbersome in the current OpenMP interface. In this chapter we presented a version of Minimod which achieves multiple GPU support using target regions wrapped in tasks inside a parallel region. We also show that we can use the remote offloading functionality now included in LLVM to utilize GPUs across multiple nodes with no changes needed to the application code. However, currently the plugin fails to achieve improved performance over a single node in distributed setups.
Chapter 6

Experiments with concurrent CPU+GPU offloading

This chapter contains some results obtained from preliminary research on concurrent CPU+GPU offloading in Minimod.

6.1 Experimental setup

The experiments described below were run on several different computers:
- *Mesquite*, TotalEnergies’s cluster in Houston. Each Mesquite node has two IBM POWER9 processors and four NVIDIA GPUs.
- *SeaWulf*, SBU’s cluster. Each SeaWulf node has two Intel Xeon Gold 6148 processors with 20 cores each, for a total of 40 cores (one thread per core). No GPUs.
- *Exxact-f1*, a computer in our group. The node has two Intel Xeon Gold 5115 processors with 10 cores, for a total of 20 cores (two threads per core). It also has two NVIDIA GeForce RTX 2080 GPUs.

6.2 Concurrent CPU+GPU experiments

Several experiments were run with concurrent CPU and GPU computations using OpenMP’s tasking features. A variable, frac_gpu, was introduced to control what fraction of the domain is allocated to the GPU, with the rest being allocated to the CPU. The domain is split along the $z$ domain according to the value of frac_gpu.
(Note that Fortran uses column-major ordering, so this results in contiguous portions of memory allocated to the CPU and GPU).

On Mesquite, the execution time was evaluated with different values of GPU fraction. The results for Mesquite with the IBM XL compiler are:

<table>
<thead>
<tr>
<th>GPU Fraction</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>191.8</td>
</tr>
<tr>
<td>50%</td>
<td>102.9</td>
</tr>
<tr>
<td>60%</td>
<td>79.1</td>
</tr>
<tr>
<td>70%</td>
<td>61.0</td>
</tr>
<tr>
<td>75%</td>
<td>54.2</td>
</tr>
<tr>
<td>80%</td>
<td>47.7</td>
</tr>
<tr>
<td>85%</td>
<td>38.7</td>
</tr>
<tr>
<td>90%</td>
<td>37.3</td>
</tr>
<tr>
<td>95%</td>
<td>36.2</td>
</tr>
<tr>
<td>100%</td>
<td>21.4</td>
</tr>
<tr>
<td>(Default)</td>
<td>20.5</td>
</tr>
</tbody>
</table>

The results for SBU’s Exxact computer (GFortran) are:

<table>
<thead>
<tr>
<th>GPU Fraction</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>75%</td>
<td>172.2</td>
</tr>
<tr>
<td>80%</td>
<td>213.4</td>
</tr>
<tr>
<td>85%</td>
<td>220.4</td>
</tr>
<tr>
<td>90%</td>
<td>230.9</td>
</tr>
<tr>
<td>95%</td>
<td>237.2</td>
</tr>
<tr>
<td>100%</td>
<td>249.2</td>
</tr>
<tr>
<td>(Default)</td>
<td>317.0</td>
</tr>
</tbody>
</table>

The GFortran compiler is seen to have extremely poor GPU performance. In fact, we notice that as GPU fraction increases, the total execution time actually increases for GFortran!

### 6.3 Conclusions

The goal of the preliminary experiments of this chapter was to utilize OpenMP tasks to make simultaneous use of the GPU and CPU on our test machines. The hypothesis was that using both computational resources, with an appropriate domain
distribution, would result in better performance than using either resource individually. However, we find that, on IBM’s compiler, it is better to simply use the GPU exclusively and not use the CPU. On GFortran, however, due to poor OpenMP offloading performance, putting more of the domain on the GPU results in deteriorating performance.

Further study is needed to determine whether a hybrid CPU+GPU strategy could improve performance in Minimod. Different domain decompositions can be evaluated, as well as different compilers.
Chapter 7

Evaluation of Minimod Across A64FX Platforms

Note: this material appears in [45] and is partially reproduced here—specifically, the parts relevant to Minimod.

Abstract

The development of the A64FX processor by Fujitsu has created a massive innovation in High-Performance Computing and the birth of Fugaku: the current world’s fastest supercomputer. A variety of tools are used to analyze the run-time and performance Minimod, on the A64FX processor. We examine the performance and behavior through OpenMP scaling and how the performance differs across different compilers both on the new Ookami cluster at Stony Brook University as well as the Fugaku supercomputer at RIKEN in Japan.

7.1 Introduction

The introduction of the A64FX processor by Fujitsu, and its use in the Fugaku supercomputer (Fugaku), has sparked the re-emergence of vectorized processors/programming and the birth of the next world’s-fastest supercomputer \(^1\). This comes on top of the fact that the A64FX chip also brings an unprecedented co-design approach, impressive performance, and energy awareness that puts it at the top of all

\(^1\)https://top500.org/
5 major HPC benchmarks. In this chapter, we will be analyzing OpenMP, a well-known shared memory/parallel programming model, from its scaling abilities on the A64FX processor to how it performs across different compiler toolchains.

The full list of current compilers that support OpenMP can be found online. Although there is one OpenMP specification, compiler support varies both in terms of specific OpenMP features and general performance.

In the next two subsections we give a brief overview of the A64FX processor, followed by the chapter’s contribution and organization.

### 7.1.1 The A64FX Processor

The A64FX processor [54, 71] is the processor specifically manufactured for Fugaku, which was made possible as part of the Japanese FLAGSHIP 2020 project as a co-design between RIKEN and Fujitsu. Currently, Fugaku is ranked number 1 on both Top500 and HPCG lists. The A64FX, is a general-purpose processor based on the Armv8.2-A architecture [71] and comes with 48 compute cores + 2/4 cores dedicated to OS activities.

The A64FX processor produced by Fujitsu has 4 core memory groups (CMG). In the FX700 chip, each CMG has 12 cores, while the FX1000 chip has 2-4 extra assistant cores. Ookami currently has the FX700 chips, with each core laid out sequentially: cores 0-11 make up CMG 0, 12-23 make up CMG 1, etc. [14].

### 7.1.2 Chapter’s contribution and organization

Although OpenMP support is available in many compilers, to the best of our knowledge, there were no studies of OpenMP’s performance in various compilers (and specific versions) specifically for A64FX processors with the application considered in this chapter, and features of OpenMP used. To that end, the chapter’s contributions are as follows:

- We present and evaluate the single node performance of Minimod using all available compilers on two systems that have A64FX processors, viz. Ookami and Fugaku.
- We present, evaluate, and compare the differences in performance on two different models of A64FX.
- We discuss our findings, and based on the results obtained, we summarize the maturity level of compilers available on these two systems to fully utilize the features of A64FX processors.

[^2]: https://www.openmp.org/resources/openmp-compilers-tools/
The rest of the chapter is organized as follows. In section 7.2 we present the details of the application considered in this study, and of the systems and compilers used. In section 7.3 we present and discuss the results obtained as well as inferences obtained from running Minimod through a performance analysis tool for A64FX. In section 7.4 we list related work and discuss their contributions and the contribution of our work. Finally, in section 7.5 we summarize our findings and list some work to be performed in near future.

7.2 Application and experimental setup

7.2.1 Application: Minimod

Minimod [43, 62, 64] - a seismic modeling mini-app that solves the acoustic wave equation using finite differences with a stencil. Minimod is developed by TotalEnergies and is designed as a platform to study the performance of emerging compilers and runtimes for HPC. In this chapter we consider the OpenMP loop-based and task-based variants of the code [62].

7.2.2 Systems and Compilers

Fugaku is the world’s fastest supercomputer, located at the RIKEN Center for Computational Science in Japan [69], and runs on the FX1000 A64FX, which provides extra cores for OS-communication. Its underlying TofuD interconnect is implemented as an interconnect controller (ICC) chip to allow for low latency and offloading.[22]

Ookami is a cluster installed at Stony Brook University (SBU) in the middle of 2020. It contains 174 compute nodes, with another two set aside for quick experimentation. Ookami was funded through an NSF grant [50] as the first A64FX cluster outside of Japan. It comes with an array of software modules, including GNU, LLVM, and Cray compilers, profilers, and MVAPICH/OpenMPI packages. Ookami uses a non-blocking HDR 200 switching fabric via 9 40-port Mellanox Infiniband switches in a 2-level tree, which allows for a peak bandwidth of 100 Gb/s between nodes. In addition, each node currently has 32GB of high-bandwidth memory with a peak memory bandwidth of 1 TB/s. Both systems’ compiler toolchains are shown in Table 7.1.
Table 7.1: Compilers of Fugaku and Ookami

<table>
<thead>
<tr>
<th>Compiler Family</th>
<th>Fugaku</th>
<th>Ookami</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>-</td>
<td>20.3</td>
</tr>
<tr>
<td>Cray</td>
<td>-</td>
<td>10.0.1</td>
</tr>
<tr>
<td>Fujitsu</td>
<td>4.3.0a, 4.4.0a</td>
<td>-</td>
</tr>
<tr>
<td>GCC</td>
<td>8.3.1, 10.2.1</td>
<td>8.3.1, 10.2.1, 11.0.0</td>
</tr>
<tr>
<td>LLVM</td>
<td>11.0.0</td>
<td>11.0.0, 12.0.0</td>
</tr>
</tbody>
</table>

### 7.2.3 Runtime Environment

Each benchmark was run on 1 compute node with 1 MPI rank/process to avoid shared memory operations that occur with 2 or more processing elements and oversubscription of threads to cores, which result in degraded performance. Threads are bound to cores using the `OMP_PLACES` environment variable.

Threads are assigned to specific cores (e.g. Thread 0 is assigned to Core 0) and divided equally among specific CMGs. For example, 32 threads are divided equally among the four CMGs on a single Ookami node (cores 0-8 in CMG 0, 12-19 in CMG 1, etc.) using

\[
OMP\_PLACES=\{0\}\:8,\{12\}\:8,\{24\}\:8,\{36\}\:8\].
\]

We ran experiments using 1, 2, 4, 8, 12, 16, 24, 32, 36, and 48 OpenMP threads. For every value up to 12, we placed all threads in one CMG. The 16-thread and 24-thread experiments were run on 2 CMGs, with each group having half the total thread values. The 32-thread and 48-thread experiments were run on all 4 CMGs on the A64FX chip, with 36 threads being run on 3 CMGs.

### 7.2.4 Compiler options

For each compiler mentioned in Section 7.2.2, we turned on specific flags, maximizing thread optimization, SVE instruction generation, and execution speed while maintaining correctness of output. We also enabled fine-tuning for the A64FX processor and the ARM-8.2 architectures where possible. The flags are listed for each compiler/group are set as shown in Table 7.2. Note that GCC versions before version 9 do not support the `mcpu=a64fx` flag – for GCC 8, we compile directly on an A64FX node and use `mcpu=native`. These flags instruct the compiler to use auto-vectorization; we have not tested OpenMP’s SIMD clauses.
Table 7.2: Flags used for each compiler.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray</td>
<td>-homp -hvector3</td>
</tr>
<tr>
<td></td>
<td>-hthread3</td>
</tr>
<tr>
<td>GCC</td>
<td>-mcpu=a64fx</td>
</tr>
<tr>
<td></td>
<td>-Ofast -fopenmp</td>
</tr>
<tr>
<td>LLVM</td>
<td>-mcpu=a64fx</td>
</tr>
<tr>
<td></td>
<td>-Ofast -fopenmp</td>
</tr>
<tr>
<td>Fujitsu-Traditional</td>
<td>-Nnoclang -Nlibomp -03</td>
</tr>
<tr>
<td></td>
<td>-Kfast,</td>
</tr>
<tr>
<td></td>
<td>-Kopenmp,ARMV8_2_A</td>
</tr>
<tr>
<td></td>
<td>-KSVE,A64FX</td>
</tr>
<tr>
<td>Fujitsu-LLVM</td>
<td>-Nclang -Nlibomp</td>
</tr>
<tr>
<td></td>
<td>-Ofast -Kfast,openmp</td>
</tr>
<tr>
<td></td>
<td>-mcpu=a64fx+sve</td>
</tr>
</tbody>
</table>

7.3 Experimental results

Our experiments analyzed runtime, relative speedup through OpenMP threads, and efficiency with respect to different compiler/compiler classes – Cray, ARM, GNU, and LLVM. Subsection 7.3.1 contains all the results run on SBU’s Ookami cluster, followed by subsection 7.3.2 containing results from Fugaku. For each application, we deemed three compilers as "best in class" (best runtime) for the families mentioned above: GNU-10.2.0 (gcc/g++/gfortran), ARM/LLVM-20.1.3 (armclang/armclang+/armflang), and Cray-10.0.1 (cc/CC/ftn), with the results for the other compilers explained in the following subsections. Preliminary results are shown in [44].

Our results are drawn from running our programs 5 times per OpenMP thread value requested (1, 2, 4, 8, 12, 16, 24, 32, 36, 48) and taking the arithmetic mean values from each set of runs. These experiments are limited to a single node.

7.3.1 Ookami

We perform runs with the following two different OpenMP configurations (see Chapter 4 for details):

- Loop xy: Grid is blocked in x (largest-stride) and y dimensions. A OpenMP parallel for loop is applied to the 2-D loop nest over x-y blocks. (A collapse(2) is used to combine the two loops).
• Tasks xy: Grid is blocked in $x$ and $y$ dimensions. Each $x$-$y$ block is a task using OpenMP’s task directive. OpenMP’s depend clause is used to manage dependencies between timesteps.

A grid size of $512^3$ was used. Minimod times are shown for each configuration in Figures 7.1(a) and 7.1(b), with speedups in 7.2(a) and 7.2(b). Note that the Cray C compiler was unable to compile this code, due to an internal compiler error, so only GCC and Arm compiler results are shown.\(^3\)

![Figure 7.1: Timing for Minimod/Ookami (a) loop xy (b) tasks xy](image)

In the GCC compilers, the loop-based configuration tends to outperform the task-based configuration. In LLVM compilers, however, the performance is similar between the two configurations.

We profiled Minimod using the ARM Forge Performance Report tool with 48 threads. We find that in both configurations, the application spends almost the entire runtime within OpenMP regions, and both have a high number of stalled cycles (76.5% and 80.7% of cycles for loop-xy and tasks-xy configurations respectively), indicating that the application is memory-bound. This makes the HBM2 memory of the A64FX processor potentially advantageous for this type of application.

### 7.3.2 Fugaku

With Fugaku’s customized Linux kernel and its compute node’s processors having 2 extra cores compared to Ookami, it was a slight challenge creating experiments

\(^3\)A Fortran version of Minimod was also evaluated using the Cray Fortran compiler. While this version was successfully compiled, the final numerical result was incorrect with optimization turned on.
whose environment matched the conditions set in the Ookami-based experiments. The Fujitsu compiler’s ability to compile with either their traditional backend and an LLVM backend creates the ability to compare a compiler’s performance with itself. In this section, we will break down and explain our results on the Fugaku supercomputer comparing results between GNU Compilers, and the Fujitsu compilers.

Minimod times are shown for each configuration in Figures 7.3(a) (loop xy) and 7.3(b) (tasks xy), with speedups in 7.4(a) and 7.4(b). Because the traditional backend for the Fujitsu compiler supports OpenMP up to only version 3.0, it cannot compile the task-based version of Minimod, whereas the LLVM backend supports up through the latest OpenMP specification versions, per Figures 7.3(b) and 7.4(b).

Figure 7.2: Speedup for Minimod/Ookami (a) loop xy (b) tasks xy

Figure 7.3: Timing for Minimod/Fugaku (a) loop xy (b) tasks xy
Profiling of the Minimod application on Fugaku is currently in progress.

7.4 Related Work

In [52], a group from RIKEN reports their preliminary performance analysis of A64FX compared to the Marvell (Cavium) ThunderX2 (TX2) and Intel Xeon Skylake (SKL) processors based on 7 HPC applications and benchmarks. Some of the applications considered use only OpenMP and others use hybrid MPI + OpenMP for parallelization. The compilers used in this study are the Fujitsu Compiler 4.2.0 (under development) for A64FX, ARM-HPC Compiler 20.1 for TX2, and Intel Compiler 19.0.5.281 for SKL.

Another group [31] from EPCC at The University of Edinburgh, reports on their study of various complex scientific applications and mini-kernel benchmarks across multiple nodes, as well as on a single node on different production HPC platforms, which include Fujitsu A64FX processors, 3 Intel Xeon series – E5-2697 v2 (Ivy-Bridge), E5-2695 (Broadwell), and Platinum 8260M (Cascade Lake)– and Marvell ThunderX2. Different compiler families, including several versions of some of them, like, Fujitsu, Intel, GCC/GNU, ARM/LLVM, and Cray, were used in their study. Also, they have considered various MPI implementations and scientific libraries. Several recent works evaluated benchmark applications using multiple compilers on the A64FX processor; e.g., [58, 25]. However, these works do not focus on OpenMP.
7.5 Conclusions

In this chapter, we have studied and observed the behavior of OpenMP implementations on the A64FX processor for Minimod with several compiler toolchains. We have observed that Cray’s compilers and GNU/LLVM compilers that have support for ARM-based processors appear to scale better with OpenMP compared to the Fujitsu compilers. We have observed that, while having the most optimal performance, the Cray Compilers may fail to compile code or generate incorrect instructions, such as with the Minimod application, leading to incorrect results.
Chapter 8

Interfacing MPI with OpenMP tasks

MPI is the de-facto standard for exploiting distributed node-level parallelism, while OpenMP is the standard for parallelism within a node. These two approaches are often combined to form the hybrid MPI+OpenMP paradigm, which uses both frameworks to coordinate distributed and local parallelism, respectively.

Traditionally, the OpenMP parallelism in this hybrid paradigm is restricted to classic loop-based OpenMP parallelism. However, one can also use OpenMP tasks to coordinate MPI communication. This offers the potential benefit of alleviating load balancing within a node and overlapping communication and computation within the node.

8.1 Related work

The use of fine-grained OpenMP task parallelism with MPI was explored in 2019 by Richard et al. [67]. They found significant improvements from using the task priority feature of OpenMP to prioritize MPI tasks, and using recursive tasks to decrease the number of dependencies per task. They also found that using MPI in MPI_THREAD_MULTIPLE mode causes locking within the MPI implementation which interferes with their tasking model, recommending MPI_THREAD_SERIALIZED instead.

In a different approach, the CHAMELEON library [36] provides a framework for OpenMP tasks to migrate to different MPI ranks for load balancing.
8.2 Description of code changes

To create an MPI+OpenMP task-based version of Minimod, we began with ASF’s existing MPI framework and the “tasks xy” OpenMP-based implementation from Chapter 4. From this starting point, the remaining obstacle was handling the MPI halo exchange. As shown in Fig. 8.1, the tasks can be broken into four categories: “Compute”, where the stencil is computed for each block (see Fig. 3.1 right for the blocking scheme); “Pack”, where the halo region of the wavefield is copied to a dense region for transfer over MPI, “Exchange”, where each packed halo region is exchanged to and from each node using MPI, and “Unpack”, where each updated packed region is copied back into the wavefield array for computing.

As suggested in [67], we use the MPI_THREAD_SERIALIZED mode of MPI. In this mode, only one thread at a time is allowed to use MPI functions. In the new code, we achieve this by using a separate dependency object, which is declared as an integer in the code, on which all exchange tasks have an inout dependency.
8.3 Preliminary Results

8.3.1 Experimental Setup

We evaluate the Minimod MPI+OpenMP task hybrid scheme on two A64FX nodes, Ookami and Fugaku. These systems are described in Section 7.2.2. On both systems we use the Fujitsu compiler and Fujitsu MPI. The Fujitsu compiler is configured in “Clang mode” because the traditional mode does not support OpenMP task dependencies which are required for our implementation.

For each experiment, we use a grid size of $768^3$ and divide the domain among the processors in the $x$ and $y$ dimensions. Three configurations were used on each system: “Lx”, the traditional loop-driven implementation; “Txy-nodep”, a task-driven variant that uses bulk taskwait synchronization points instead of dependencies, and “Txy”, a task-driven variant with dependencies.

8.3.2 Results

Preliminary strong scaling results on each system are shown in Table 8.1. Some experiments fail to run to completion, particularly with smaller number of nodes, due to the simulation running out of memory; these entries are left blank in the table.

In general, results between the loop versions and task versions of the code are quite similar, suggesting that using tasks to drive MPI in this manner neither helps nor hurts performance significantly. A larger penalty is seen from using tasks on Fugaku, particularly when using dependencies (the “Txy” variant).

<table>
<thead>
<tr>
<th># Nodes</th>
<th>Ookami</th>
<th>Fugaku</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lx</td>
<td>Txy-nodep</td>
</tr>
<tr>
<td>1</td>
<td>59.8</td>
<td>51.3</td>
</tr>
<tr>
<td>2</td>
<td>31.1</td>
<td>26.7</td>
</tr>
<tr>
<td>4</td>
<td>17.0</td>
<td>18.3</td>
</tr>
<tr>
<td>8</td>
<td>11.2</td>
<td>12.5</td>
</tr>
</tbody>
</table>

Table 8.1: Preliminary strong scaling results on Ookami and Fugaku
8.4 Perspectives on OpenMP tasks

There are several observations that can be made about current versions of OpenMP from the experiments performed here. First, as discussed in previous chapters, the lack of support for overlapping dependency regions complicates the implementation in several ways. In several places, our implementation resorted to using generic dependency “objects”, declared as simple integers, to express dependencies that could not be otherwise expressed without overlapping dependency regions.

Second, when using tasks with dependencies, race conditions can sometimes be difficult to identify. For example, consider the snippet of code from Minimod in Fig. 8.2. This is part of a loop over halo regions that receives the updated regions from other ranks. For each such region, on the first line of the code snippet, the MPI Request object is initialized. Then a task is created to receive the updated region. However, because the initialization of the request object is not part of the task, it is executed in an indeterminate order with respect to the task. In initial experiments, this resulted in the application waiting incorrectly for halo exchanges to complete, a race condition which ended up changing numerical the result slightly.

These types of race conditions are easy to overlook and difficult to diagnose. A framework for automatically finding such race conditions would be useful. There has been some previous work in this direction, e.g. ROMP [26].

Finally, existing tools for profiling OpenMP applications are limited in information they provide about task-based programs. It would be helpful to have profiling tools that show, e.g., time traces of OpenMP tasks and dependency graphs.
8.5 Conclusion

In this chapter, we present a preliminary implementation of a variant of Minimod using OpenMP tasks for computation as well as communication over MPI. Dependencies are used to ensure that only one MPI task runs at a time, to avoid thread-safety issues in MPI. Performance does not improve, but also is not significantly harmed. More research is needed to determine how to achieve improved performance from this implementation.
Chapter 9

Minimod extension to multiple nodes using Legion

Note: this material appears in [64] and is reproduced here with some modifications.

9.1 Introduction

The traditional approach to reduce the time-to-solution of explicit and implicit methods for numerical simulation is to parallelize the computing. Computations can be parallelized on a single node using shared-memory parallelism, for which OpenMP is a commonly used technique. When more processing power or memory is desired than a single node can provide, distributed-memory parallelism techniques are used to split the computation among multiple nodes. MPI is a standard approach to distributed-memory parallelism.

These paradigms can be combined: a common approach is known as MPI+X, a multi-layer hybrid approach where a message-based runtime coordinates the communication of data among computing nodes, and X solve the local problem. X can be any shared-memory oriented programming model, possibly including task-based parallelism at the shared-memory level (e.g., OpenMP, Intel TBB, Kokkos), or extended to accelerators, such as CUDA or HIP. A possible alternative is to replace the MPI runtime with a PGAS (partitioned global address space)-based one, which exposes a global address space to each processor, partitioned into separate sections for the memory on each processor.

The contribution of this work is to evaluate an alternative programming model to solve the problem described above. We use the Legion model [7] to handle distributed-memory parallelism. Legion includes an OpenMP implementation, so
we retain the use of OpenMP for shared-memory parallelism. It can be summarized as Legion+OpenMP. Furthermore, we evaluate our Legion-based implementation on the newly introduced Stony Brook University’s Ookami cluster, a cluster utilizing the A64FX processor.

Legion is a global task-based programming model based around the concept of “logical regions”, a way of storing data that allows the Legion runtime to automatically extract task-level parallelism from the application. In general, task-based programming has the potential to help alleviate load-balancing issues in applications and free developers from the burden of manual communication that is used in MPI-style programming. Although the mini-application studied in this work is fairly regular, making significant performance improvements from task-based programming unlikely, we expect task-based programming to offer greater benefits over traditional programming techniques as we add different kernels to the application.

The layout of the document is the following: in Section 9.2 the challenge addressed by our method is introduced. Section 9.3 reviews related work. Sections 9.4 and 9.5 describe MiniMod, the stencil-based application studied in this chapter, and its rewrite in C. Section 9.6 presents our Legion-based implementation of the application. Section 9.7 describes our evaluation methodology, experimental results, and a discussion of our findings. Section 9.8 summarizes our conclusions and briefly discusses our plans for future work.

9.2 Parallel Stencil Computation

In Figure 9.1 left panel, it can be observed that each grid point can be computed in parallel provided each element is able to access their neighboring values before an in-place update, in order to retain the data dependencies imposed by Eq. (3.3). Thus given a sufficient number of computation units, solving this equation can be achieved in straightforward parallel fashion where all elements are computed at once. To achieve this in practice requires a coherent memory address space shared across a very large number of computation units. No current hardware platform is able to fulfill these requirements for moderately sized domains ($1024^3–2048^3$, roughly 1–8 billion elements), the closest are provided by GPGPU systems (such as the NVIDIA V100 with 5140 processing units). A more suitable alternative is to divide the domain into subdomains, which can be mapped to individual processing units and computed in parallel. A simple example of these where three threads are mapped to corresponding rows of a compute domain is depicted in Fig. 9.1 left panel. This is usually implemented using runtimes such as OpenMP on CPUs and languages such as NVIDIA’s CUDA or AMD’s HIP for GPUs.
Figure 9.1: A 9-point 2D stencil is shown, where green colored elements represent grid points within the computing domain, yellow elements are not computed but needed by the stencil (e.g. to impose boundary conditions). In the left panel, a simple threads to rows mapping is presented. In the right panel, two domains ($D_0, D_1$) shaded light blue and light orange are depicted. The hard boundary implies no memory coherency, therefore information must be exchanged between the two domains in order to solve Equation 3.3 correctly.

When scaling to large problems, a single compute node may not be sufficient in either memory capacity or processing throughput. The solution is to use a distributed parallel approach. In this case memory is distributed in multiple locations without coherent access, thus communication between the compute nodes is needed to accurately perform these stencil computations. For instance, as shown in Figure 9.1, for a stencil centered at $(x_0,y_i)$ of $D_0$, its neighbors $(x_0,y_{i+1})$ and $(x_0,y_{i+2})$ lay within $D_1$, requiring the transfer of data between these two subdomains. This approach is widely used and frequently implemented using runtimes such as MPI or PGAS, the main difference between the two being the use of explicit or implicit data motion respectively.

Finally, these two classes of approaches can be combined to form hybrid parallel approaches, for instance of the form MPI+OpenMP or MPI+CUDA. Sections 9.4 and 9.5 describe our implementation of these hybrid approaches for the research presented here.

9.3 Related Work: Distributed task parallelism

Distributed-memory task-based systems have been explored, in which the runtime automatically schedules tasks among the available nodes and implicitly handles communication and data transfer. Charm++ [2] is a C++ framework supporting dis-
tributed task parallelism. PaRSEC [11] enables an application to be expressed as a “parameterized task graph” which is problem-size-independent and therefore highly scalable. HPX [33] is a task-based framework which uses a global address space to distribute computations across nodes. XcalableMP [37] is a PGAS language with elementary support for task parallelism. YML [18, 27] allows the user to specify a computation as a graph of large-scale tasks and supports combination with XcalableMP. StarPU [6] supports OpenMP-style pragmas and provides a runtime for distributed execution. Klinkenberg et al. [36] propose a framework for distributing tasks across MPI ranks in MPI+OpenMP hybrid applications.

Legion [7] and its DSL Regent [73] are data-centric task-based programming systems developed at Stanford. Soi et al. [75] implement a task-based implicitly parallel meshfree solver using the Regent programming language. Their implementation takes advantage of the optimizations available in Regent and Legion for improving the scalability and performance of the target application, such as index launches and dynamic control replication. The performance of the Regent-based implementation is compared with an MPI-based solution as well as a Julia-based solution, evaluated using both single-node and multi-node CPU systems. Their Regent-based implementation shows a significant degradation in scalability compared to the MPI version, even at relatively small node counts.

9.4 Distributed MiniMod (dMiniMod)

dMiniMod is part of the Minimod application suite which is implemented with explicit parallel programming using MPI for running on distributed computer systems. It uses non-blocking communication in MPI to achieve higher performance by overlapping communication and computation. It carries out domain decomposition to partition the global geographical area into several sub-domains so that each process handles one sub-domain independently for fully parallelized computation.

dMiniMod provides both a sequential implementation and a hybrid-parallel implementation using the combination of MPI+OpenMP programming models. The hybrid MPI+OpenMP version allows us to further improve system performance by exploiting additional layers of parallelism and utilizing higher numbers of processors past where the scalability of the pure MPI version reaches its limits. The implementation of MPI+OpenMP follows the classical master-only style where all MPI calls are made only from the OpenMP master thread outside of any parallel regions. The hybrid implementation in dMiniMod accommodates a clear separation between MPI and OpenMP levels of parallelism.

In this chapter we focus on evaluating one of the kernels contained in Minimod:
the acoustic isotropic propagator (high-order spatial stencil) in a constant-density medium [3]. While the kernel itself has not been carefully manually optimized, the goal in this chapter is to evaluate the relative scaling of different parallel programming models.

9.5 ASF: Advanced Seismic Framework

While MiniMod and dMiniMod have been proven as useful tools for benchmarking emerging technologies, many of the details that need to be present in scientific applications are elided, such as saving results to disk for further analysis. This puts a burden on the application developer to reproduce the algorithm and performance present in the original benchmark, while making alterations to be suitable for the application’s needs. Similarly, as a benchmark suite, every combination of technologies is implemented as a separate port of the original code, so extending MiniMod to support a new technology “X” requires a substantial and cumulatively increasing effort. To prevent these from becoming serious issues as we continued development, we first spent the time to re-implement MiniMod suite as a library suitable for use by scientific applications.

The resulting library Advanced Seismic Framework (ASF) handles the myriad of technology combinations by subdividing an implementation into discrete components, responsible for computation, communication, and external I/O. Each compo-
Figure 9.3: Comparison of performance on a single Summit node between ASF (in blue) and the nearest equivalent algorithm from the MiniMod suite (in red), simulating 1000 timesteps at various grid sizes. Lower is better.

ASF also provides the option to enable distributed implementation with MPI for collective operations. The number of subdivisions in each dimension for the domain decomposition can be configured by the user. Figure 9.2 presents the trace view of running ASF with four MPI ranks on a grid sizes of 1024³. The number of domains are (1, 2, 2) in (x, y, z) dimensions respectively. The traces and statistics are collected using the HPCToolkit [41] from Rice University. The profiling results show that the majority of the runtime is consumed by kernel computations, while the MPI communication only cost 2.6% of the overall simulation time.
9.6 Legion Implementation

This section describes our implementation of ASF in the Legion programming model, which we call “ASF-Legion”. The ASF-Legion implementation performs computations on CPUs; it does not include accelerators.

Computation in Legion is done by tasks. For each task, the application developer specifies the logical regions as inputs or outputs. The Legion runtime then determines which tasks can run at the same time, and schedules them for execution on the available processors.

Algorithm 4 gives a high-level overview of the main solver in Minimod and ASF applications. In the Legion implementation, the wavefield solution step (lines 3-5) is implemented as a task, and one task is launched for each subregion of the domain. Initially, the source update step (line 6) was a separate task, but merging it into the wavefield update task improved performance due to the small execution time of the source update task.

9.6.1 Parallelism in ASF-Legion

ASF-Legion contains two levels of parallelism: Legion tasks and OpenMP threads. First, the domain is split into blocks, and the Legion runtime distributes these blocks among the Legion tasks. The second level, within each task, is OpenMP parallelism, where OpenMP pragmas are used to distribute iterations along the largest-stride dimension to threads. In order to avoid conflict between Legion’s threads and OpenMP threads, Legion implements its own OpenMP runtime. In ASF-Legion, blocks of the domain are assigned to a Legion task, which has its own OpenMP runtime and several threads assigned to it. OpenMP parallelism is then used within the task. Figure 9.4 is a diagram showing the different levels of parallelism.

9.6.2 Domain Decomposition and Mapping

The data needed by the kernel is stored as five logical regions: the wavefield at the last and current timesteps, the density field (which is constant for this kernel), and two PML wavefield arrays.

In Legion, partitions are crucial for expressing the independence of subtasks that can run in parallel. Logical regions are partitioned into subregions that are available for simultaneous read/write access. In this application, the domain is split into equally-sized pieces in each of the three dimensions. We declare two partitioning schemes for the 3-D logical regions: a disjoint set of partitions that creates equally
sized blocks that are divided among the tasks, and a second set of partitions that are the same size as the first set but also includes the ghost points of each block. This dual partition-scheme setup allows us to concisely express the inputs and outputs of each task for the wavefield solution step.

This blocked partitioning paradigm is a common use case in Legion, and can be expressed concisely using the Legion utility `create_partition_by_restriction`. Figure 9.5 shows an example. The number of partitions in each dimension is configurable.

After partitioning the domain, at each timestep, one task is launched per subregion. Figure 9.6 shows how tasks are launched in Legion. By using an index task launch, rather than launching each subtask individually, the Legion runtime is able to amortize the overhead of the task launches. Unlike the MPI version, in Legion there need not be a 1-to-1 mapping between Legion tasks and processors within a timestep. Nonetheless, in this study, one Legion task is used per process, analogous to the distribution of one block per MPI rank.

The Legion programming model allows the user to write a custom mapper to specify how tasks and data should be mapped to the processors. The mapper decides how to divide generated tasks among available compute nodes, controls work stealing/sharing for load balancing, and places data among available memories. In this application, the default Legion mapper is used. We are still investigating whether changes to the default mapping decisions could improve performance in our application.
IndexSpace is = runtime->create_index_space(ctx,  
    Rect<3>(Point<3>(0,0,0), Point<3>(nx-1,ny-1,nz-1));
IndexSpace color_is = runtime->create_index_space(ctx,  
    Rect<3>(Point<3>(0,0,0), Point<3>(nbx-1,nby-1,nbz-1)));
Transform<3,3> transform;
int bsx = nx/nbx, bsy = ny/nby, bsz = nz/bsz;
// Diagonal transform matrix
transform[0][0] = bsx;
transform[1][1] = bsy;
transform[2][2] = bsz;
Rect<3> extent(Point<3>(0,0,0),  
    Point<3>(bsx-1,bsy-1,bsz-1));
ip = runtime->create_partition_by_restriction  
    (ctx, is, color_is, transform, extent);

Figure 9.5: Code showing how domain is partitioned. nb{x,y,z} refer to the number of blocks in each dimension. Each point in the color space (color_is) is multiplied by the $3 \times 3$ transform matrix to generate an offset. This offset is added to the extent.

IndexTaskLauncher launcher(KERNEL_TASK, color_is, TaskArgument(arg,  
    sizeof(*arg)), arg_map);
launcher.add_region_requirement(RegionRequirement(lp_waves, 0,  
    READ_ONLY, EXCLUSIVE, lr_waves).add_field(fid_wave_now));
// add other regions ...
runtime->execute_index_space(ctx, launcher);

Figure 9.6: Code showing how kernel computation tasks are launched. In this example, the logical partition for the waves (lp_waves) is added to the task as a region requirement, so that one subtask will be launched for each partition of the domain. The execute_index_space method schedules the execution of each subtask.
9.6.3 Memory Management

Data from logical regions in Legion is accessed using “accessors”. When implementing the main computational kernel in ASF-Legion, we noticed that accessing data using an accessor resulted in a substantial slowdown with the Clang compiler – roughly 2-3X for a single thread. After investigation, we determined that this was due to the way the accessor addresses elements in an array. Because the accessor is written generically, each subscript in an array access is multiplied by a variable containing the stride in that dimension. Arrays are stored in row-major order, so the last dimension is stride-1 access. The lack of compile-time knowledge of stride-1 access inhibits an optimization performed by the Clang compiler. We modified the accessor to assume stride-1 access in the last dimension, and recovered the single-thread performance of the original code without Legion.

9.7 Experimental Results

In this section, the performance results of the experiments conducted with ASF-MPI and ASF-Legion are presented. First, the experimental setup is described, second weak and strong scaling performance of each parallel implementation is introduced. Finally, profiling and analysis are provided.

9.7.1 Experimental Setup and Platforms

We perform experiments on the Summit computer at Oak Ridge National Laboratory, and the Ookami cluster at Stony Brook University.

Each node of Summit [51] has two sockets containing IBM POWER9 processors, with 44 total CPU cores per node, and 512 GB memory per node. Each node also has 6 NVIDIA V100 GPUs; however, we do not use GPUs in this study.

The Ookami cluster [1] at Stony Brook University is a pre-production system containing the first deployment of the Fujitsu A64FX [53] microprocessor outside of Japan. Each node of Ookami includes the A64FX processor, which has 48 compute cores and 32 GB of HBM2 memory, organized into four NUMA “core memory groups (CMGs)” with 12 cores and 8 GB memory each. Both systems utilize an InfiniBand interconnect with a fat-tree topology.

Detailed information about each system, including the software stack used, is shown in Table 9.1.

On each Summit node, two Legion or MPI processes are used—one for each socket. Legion reserves some cores to help manage mapping and communication. We
<table>
<thead>
<tr>
<th>System</th>
<th>Hardware Specs</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summit</td>
<td>CPUs 2x IBM Power9</td>
<td>LLVM 9.0</td>
</tr>
<tr>
<td></td>
<td>CPU cores 22x2</td>
<td>SMPI* 10.3.1</td>
</tr>
<tr>
<td></td>
<td>Memory 512 GB</td>
<td>GASNet 2020.3.0</td>
</tr>
<tr>
<td></td>
<td>L3 10 MB (x2 cores)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L2 512 KB (x2 cores)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L1 32+32 KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lithography 14nm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interconnect InfiniBand EDR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TDP 190x2 W (only CPUs)</td>
<td></td>
</tr>
<tr>
<td>Ookami</td>
<td>CPU Fujitsu A64FX</td>
<td>GCC 10.2.1</td>
</tr>
<tr>
<td></td>
<td>CPU cores 12x4</td>
<td>OpenMPI 4.0.5</td>
</tr>
<tr>
<td></td>
<td>Memory 32 GB</td>
<td>GASNet 2020.3.0</td>
</tr>
<tr>
<td></td>
<td>L2 8 MB (x12 cores)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L1 64+64 KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lithography 7nm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interconnect InfiniBand HDR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TDP 160 W (full node)</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.1: Hardware and software configuration of the experimental platforms. *IBM Spectrum MPI (SMPI)
use 36 cores on each node (18 per socket) to run OpenMP threads (to perform kernel computations), and the rest are reserved for Legion. To match this configuration, we also use 36 cores per node for ASF-MPI. Although we only use one thread per physical core, we enable hyperthreading (using \textit{smt4} mode) because in experiments we found that this mode gives a dramatic performance improvement over no hyperthreading (\textit{smt1} mode) for Legion.

On Ookami, we use one MPI rank or one Legion processor per CMG (the A64FX processor has four CMGs per node). Each rank uses 10 OpenMP threads. Due to the limited amount of memory available on each node, for MPI with a grid size of $1024^3$ and larger, on a single node, we were only able to use two MPI ranks to avoid running out of memory. All the multi-node simulations on Ookami use four MPI ranks per node. Also, the $2048^3$ grid size is evaluated only on Summit, since this grid size is too large to fit within a node of Ookami.

To account for timing variability, experiments are performed three times and the median of these results is used.

### 9.7.2 Scalability Results and Analysis

We begin by evaluating the weak scaling of ASF-MPI and ASF-Legion. Weak scaling is evaluated with a simulation grid size of $1024^3/16$ grid points per node, from 1 to 32 nodes. The grid distribution for each node number is chosen as the per-rank “best” distribution for 16 nodes (e.g., Table 9.4), expanded along the x-axis for each additional rank. For example, on Summit, the best Legion distribution for 16 nodes is 16:2:1. Therefore, the Legion weak-scaling experiments on Summit use a grid size of 64:512:1024 per rank, for a total grid size of (64×nranks):512:1024, where there are two ranks per node.

The expectation with weak scaling performance is that as long as more resources are deployed and the problem size increased proportionally, the parallel efficiency should remain the same, or close to the reference. In order to have a meaningful reference, the size of the problems should be demanding enough to stress all relevant system resources. The results introduced in Table 9.2 (Summit) indeed show the expected behavior for both implementations. In Table 9.3 (Ookami), results for ASF-Legion are less promising after 8 nodes; in the extreme, the 32 node case failed. The performance of ASF-Legion on Ookami is not up to what is achieved on Summit or ASF-MPI on the same cluster, the cause of this discrepancy is under investigation.

Table 9.4 shows the sensitivity of ASF-MPI and ASF-Legion to how the domain is partitioned into blocks, for selected partition schemes. In general, ASF-Legion is much more sensitive to the partition scheme than ASF-MPI, with some configurations
<table>
<thead>
<tr>
<th>Nodes</th>
<th>ASF-MPI [s]</th>
<th>ASF-Legion [s]</th>
<th>Parallel Efficiency [%] ASF-MPI, ASF-Legion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17.9</td>
<td>20.0</td>
<td>100.0%, 100.0%</td>
</tr>
<tr>
<td>2</td>
<td>18.7</td>
<td>19.6</td>
<td>95.6%, 102.0%</td>
</tr>
<tr>
<td>4</td>
<td>18.7</td>
<td>19.7</td>
<td>95.3%, 101.7%</td>
</tr>
<tr>
<td>8</td>
<td>18.8</td>
<td>19.8</td>
<td>95.1%, 100.9%</td>
</tr>
<tr>
<td>16</td>
<td>18.8</td>
<td>19.9</td>
<td>94.9%, 100.6%</td>
</tr>
<tr>
<td>32</td>
<td>18.8</td>
<td>21.1</td>
<td>94.9%, 94.7%</td>
</tr>
</tbody>
</table>

Table 9.2: Weak scaling: throughput (stable around the reference is ideal) with 1-32 nodes; grid size 1024^3/16 per node; on Summit.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>ASF-MPI [s]</th>
<th>ASF-Legion [s]</th>
<th>Parallel Efficiency [%] ASF-MPI, ASF-Legion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20.4</td>
<td>22.3</td>
<td>100.0%, 100.0%</td>
</tr>
<tr>
<td>2</td>
<td>20.7</td>
<td>24.0</td>
<td>99.0%, 93.0%</td>
</tr>
<tr>
<td>4</td>
<td>20.6</td>
<td>24.6</td>
<td>99.2%, 90.8%</td>
</tr>
<tr>
<td>8</td>
<td>20.7</td>
<td>25.7</td>
<td>98.7%, 87.0%</td>
</tr>
<tr>
<td>16</td>
<td>20.6</td>
<td>47.9</td>
<td>99.3%, 46.6%</td>
</tr>
<tr>
<td>32</td>
<td>21.1</td>
<td>-</td>
<td>97.1%, -</td>
</tr>
</tbody>
</table>

Table 9.3: Weak scaling: throughput (stable around the reference is ideal) with 1-32 nodes; grid size 1024^3/16 per node; on Ookami.
Performing much better than others. In the strong-scaling results below, the best partition scheme is selected for each grid size and number of nodes.

In terms of strong scaling, both implementations follow on Summit the same trends for the $1024^3$ and $2048^3$ grids, as can be seen in Table 9.5 and Table 9.6. The parallel efficiency drops after the 8 nodes for the $1024^3$ case (Table 9.5). On Summit, for the 16 node case, each rank is solving a $320^3$ domain, where the halo exchange among neighbors starts to dent the overall performance. In Table 9.6 the same patterns are observed but the drop for 16 node case is less pronounced since a much larger problem is being solved ($\approx 512^3$ for 
ASF-Legion), giving more chances for computing and communication overlap.

The ASF-Legion implementation is $\approx 20$ seconds slower than ASF-MPI on Summit for 16 nodes ($2048^3$ grid, Table 9.6); this represents $\approx 16\%$ of slowdown, which is within the variation of the results when different domain decomposition are considered (see Table 9.4). The same trend is observed in Table 9.7, up to when 8 nodes are considered, in which ASF-MPI outperforms ASF-Legion by 7.6 seconds which corresponds to $\approx 19\%$. Besides the performance drop associated with the 16 node case, for which further analysis is underway, the ASF-Legion performance is comparable with the one reached by ASF-MPI, which is remarkable for such a regular computing pattern, with-in this case–a low number of opportunities to exploit task-oriented parallelism given the reduced number of computing kernels.

<table>
<thead>
<tr>
<th>Partitioning</th>
<th>ASF-MPI [s]</th>
<th>ASF-Legion [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:1:1</td>
<td>29.9</td>
<td>25.0</td>
</tr>
<tr>
<td>16:2:1</td>
<td>24.4</td>
<td>22.8</td>
</tr>
<tr>
<td>2:16:1</td>
<td>21.9</td>
<td>41.5</td>
</tr>
<tr>
<td>1:32:1</td>
<td>25.9</td>
<td>54.0</td>
</tr>
<tr>
<td>4:4:2</td>
<td>19.7</td>
<td>142.4</td>
</tr>
<tr>
<td>mean</td>
<td>24.36</td>
<td>57.14</td>
</tr>
<tr>
<td>median</td>
<td>24.39</td>
<td>41.51</td>
</tr>
<tr>
<td>std dev.</td>
<td>3.49</td>
<td>44.13</td>
</tr>
</tbody>
</table>

Table 9.4: Sensitivity of ASF-MPI and ASF-Legion to the block partitioning scheme (number of partitions in x,y,z dimensions) of the domain. Elapsed time (seconds) is shown for each scheme. These simulations use a grid size of $1024^3$, with 16 nodes, on Summit.
### Table 9.5: Strong scaling: elapsed time for grid size $1024^3$ and corresponding parallel efficiency; Summit

<table>
<thead>
<tr>
<th>Nodes</th>
<th>ASF-MPI [s]</th>
<th>ASF-Legion [s]</th>
<th>Parallel Efficiency [%] ASF-MPI, ASF-Legion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>232.7</td>
<td>241.3</td>
<td>100.0% , 100.0%</td>
</tr>
<tr>
<td>2</td>
<td>121.1</td>
<td>122.3</td>
<td>96.1% , 98.6%</td>
</tr>
<tr>
<td>4</td>
<td>64.8</td>
<td>64.5</td>
<td>89.8% , 93.5%</td>
</tr>
<tr>
<td>8</td>
<td>35.3</td>
<td>36.1</td>
<td>82.4% , 83.6%</td>
</tr>
<tr>
<td>16</td>
<td>19.7</td>
<td>22.9</td>
<td>73.8% , 65.8%</td>
</tr>
</tbody>
</table>

### Table 9.6: Strong scaling: elapsed time for grid size $2048^3$; Summit

<table>
<thead>
<tr>
<th>Nodes</th>
<th>ASF-MPI [s]</th>
<th>ASF-Legion [s]</th>
<th>Parallel Efficiency [%] ASF-MPI, ASF-Legion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1694.4</td>
<td>1694.4</td>
<td>100.0% , 100.0%</td>
</tr>
<tr>
<td>2</td>
<td>875.8</td>
<td>875.3</td>
<td>96.7% , 96.8%</td>
</tr>
<tr>
<td>4</td>
<td>450.8</td>
<td>460.4</td>
<td>94.0% , 92.0%</td>
</tr>
<tr>
<td>8</td>
<td>233.2</td>
<td>237.0</td>
<td>90.8% , 89.4%</td>
</tr>
<tr>
<td>16</td>
<td>121.5</td>
<td>141.8</td>
<td>87.1% , 74.7%</td>
</tr>
</tbody>
</table>

### Table 9.7: Strong scaling: elapsed time for grid size $1024^3$ and corresponding parallel efficiency; Ookami

<table>
<thead>
<tr>
<th>Nodes</th>
<th>ASF-MPI [s]</th>
<th>ASF-Legion [s]</th>
<th>Parallel Efficiency [%] ASF-MPI, ASF-Legion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>227.2</td>
<td>221.5</td>
<td>100.0% , 100.0%</td>
</tr>
<tr>
<td>2</td>
<td>115.6</td>
<td>148.8</td>
<td>98.3% , 74.4%</td>
</tr>
<tr>
<td>4</td>
<td>63.5</td>
<td>76.4</td>
<td>89.4% , 72.4%</td>
</tr>
<tr>
<td>8</td>
<td>38.5</td>
<td>46.0</td>
<td>73.9% , 60.2%</td>
</tr>
<tr>
<td>16</td>
<td>22.5</td>
<td>39.6</td>
<td>63.1% , 35.0%</td>
</tr>
</tbody>
</table>
Figure 9.7: Throughput and speedup, grid size $1024^3$, on Summit and Ookami. Higher is better. Memory footprint 20GB (with a single node), data precision is 32-bit float.

Figure 9.8: Throughput and speedup, grid size $2048^3$, on Summit. Higher is better. Memory footprint 160GB, data precision is 32-bit float. When running the experiments in Summit, each MPI rank is mapped to socket, therefore for the largest number of nodes experiments one have 32 subdomains, which means 10GB footprint per socket (4% of the available memory per socket).
Figure 9.9: Sample extracted from Legion profiler time trace with 16 nodes on Summit (left) and Ookami (right). Each time period is roughly 55 milliseconds. “OpenMP Proc 2” in each trace shows one of the compute processes, and each rectangle to the right of “OpenMP Proc 2” is a task computing the stencil on the subdomain of the grid assigned to that processor. Each task roughly corresponds to one timestep, with the other processors (not shown here) computing the other subdomains of the grid during the same period. The “[x] sys to [y] sys” rows show halo exchange between the indicated processes.

9.7.3 Profiling

Legion includes a built-in profiler to aid the developer in optimizing performance of applications. The profiler shows a time trace of task execution and memory copies during the simulation. During the development of this application, Legion’s built-in profiler was critical to diagnosing the performance issues and improving the code.

Figure 9.9 shows example time traces generated by the profiler. The profile shows occasional large gaps between stencil computations for successive timesteps (more so on Ookami). These gaps may indicate overhead associated with the Legion runtime on Ookami, but also suggest that there are opportunities for overlapping communication and computation for improved performance.

9.8 Conclusion

ASF-Legion achieved competitive results against ASF-MPI, in both raw performance and scalability. Even though this application is relatively regular, and therefore has less potential for improvement over MPI, it is promising that the task-based approach yields results comparable to the MPI approach in performance. In particular, results are robust on the production system, Summit. On Ookami, which is currently a pre-
production system, there are some performance issues, particularly affecting Legion, that need to be addressed.

Implementing ASF from a task-based perspective was relatively straightforward, although intrusive. Legion can’t act as a drop-in replacement for MPI, due to needing to manage the memory of the application as logical regions, as well as needing to write the application from a global view for task-based parallelism. Nonetheless, the modular structure of ASF (see Section 9.5) allowed reuse of much of the code of the ASF-MPI application code.

We will continue to investigate performance issues in ASF-Legion. Closing the gap in performance between ASF-Legion and ASF-MPI is a particular goal. We would also like to explore whether ASF-Legion can even gain an edge over ASF-MPI in performance. Over-decomposition, which would expose more task parallelism to the Legion runtime, is one possibility to achieve this.

Future work will include using GPUs to offload the most demanding computing sections, and to explore combining GPU kernels with task-based parallelism at the node-level. Also, other kernels will be implemented, such as IO and other physics-oriented processes.
Chapter 10

Minimod extension to distributed GPUs using Legion

Note: this material appears in [63] and is reproduced here with some modifications.

Abstract

In the era of exascale computing, the traditional MPI+X paradigm starts losing its strength in taking advantage of heterogeneous systems. Subsequently, research and development on finding alternative programming models and runtimes have become increasingly popular. This encourages comparison, on competitive grounds, of these emerging parallel programming approaches against the traditional MPI+X paradigm.

In this work, an implementation of distributed task-based stencil numerical simulation is compared with an MPI+X implementation of the same application. To be more specific, the Legion task-based parallel programming system is used as an alternative to MPI at out-of-node level, while the underlying CUDA-implemented kernels are kept at node level. Therefore, the comparison is as fair as possible and focused on the distributed aspects of the simulation.

Overall, the results show that the task-based approach is on par with the traditional MPI approach in terms of both performance and scalability.

10.1 Introduction

Solid earth studies are mostly constrained to remote sensing techniques, basically, recordings of the earth responses to mechanical perturbations. Once a set of record-
ings is acquired, numerical approaches can be applied to obtain useful information, for instance for earthquake analysis, shallow hazards estimation, CO₂ sequestration monitoring, and hydrocarbon exploration. The numerical approaches involved are related to the simulation of waves propagating through Earth. The propagation of waves through a variety of mediums is a well-understood phenomenon for which established numerical approaches exist. The Partial Differential Equation (PDE) 3.1 captures the phenomena when the medium is assumed to be isotropic and density constant. The above described equation and physics assumptions are implemented in our ASF application (details in Section 10.3). This PDE can be solved numerically using several methods. For simplicity, the Finite Differences (FD) method is used for this work. Equation 3.2 also sets the memory footprint for such an application, where it is needed to keep in memory data structures for \( u^n, u^{n-1}, V \) and ancillary arrays for boundary conditions. These two key elements (demanding kernel and memory footprint) set the stage for what can be accomplished when the application is parallelized and distributed in order to reduce time-to-solution.

The traditional approach to distribution is known as \( MPI+X \), a multi-layer hybrid approach where a message-based runtime (\( MPI \)) coordinates the communication of data among computing nodes, and \( X \) solves the local problem. \( X \) can be any shared-memory oriented programming model, possibly including task-based parallelism at the shared-memory level (e.g., OpenMP, Intel TBB, Kokkos), or extended to accelerators, using languages such as CUDA or HIP. Alternatively, MPI can be replaced by a task-based distributed runtime such as \( Legion \). Legion is a global task-based programming model based around the concept of “logical regions”, which are partitioned global address spaces managed by the Legion runtime. Logical regions allow the runtime to automatically extract task-level parallelism from the application. In general, task-based programming has the potential to help alleviate load-balancing issues in applications, and free developers from the MPI-style burden of implementing communication.

The contribution of this work is to evaluate a task-based programming model as an alternative to the traditional \( MPI+X \) approach described above. To that end, an existing numerical application is ported from \( MPI+CUDA \) to a task-based distributed approach. The Legion task-based programming model [7] is used to handle distributed-memory parallelism. An Legion+OpenMP version of the code was developed and evaluated in Chapter 9 (see also [64]). In this chapter, as opposed to an OpenMP approach, a CUDA implementation of the main computing kernel is ported to the Legion distributed approach, creating a \( Legion+CUDA \) implementation of the code. Results using A100 GPUs are obtained on Cypress, \( TotalEnergies \)'s HPC R&D cluster. Since the application studied in this work is highly regular, significant
performance improvements from task-based programming are not expected. Nevertheless, our results show that even under these conditions the task-based approach is competitive with the MPI+CUDA baseline. Further, it is expected that the task-based programming approach will offer greater benefits over traditional programming techniques as different kernels are added to the application.

The layout of the document is the following: Section 10.2 reviews related work. Section 10.3 describes ASF, the stencil-based application studied in this chapter. Section 10.4 presents the Legion-based implementation of the application. Section 10.5 describes our evaluation methodology, experimental results, and a discussion of our findings. Section 10.6 summarizes our conclusions and discusses plans for future work.

10.2 Related Work

GPU implementations of stencils have been studied since the late 2000s [23, 59, 65]. Recently, Sai et al. [70] studied high-order stencils with a manually crafted collection of implementations of a 25-point seismic modeling stencil in CUDA for the latest GPU hardware. An implementation from that work is used in the present study; further stencil optimizations are not considered in the present study since it is not the focus of this work.

Treichler et al. [77] implemented the S3D-Legion code to simulate two separate scientific simulations of reactivity controlled compression ignition and the temporal evolution of a nonpremixed jet mixing layer. The S3D application is more complex than the application in this chapter, including many different GPU kernels besides the stencil and potentially allowing more opportunities for latency hiding. Comparing the system used in the S3D-Legion experiments to the one used here, the former system uses single-GPU nodes (the system in this chapter uses four GPUs per node), has less-powerful GPUs (Kepler K20X vs A100; \( \sim 5X \) peak performance difference), and has a slower interconnect (\( \sim 3X \) difference). These differences make multi-node scaling more challenging for the system considered in this chapter.

It is known that Legion codes written with a single top-level task distributing work (as ASF-Legion is written) have scalability limitations. The control replication approach [72, 8] overcomes this limitation by distributing the main control logic of the application statically or dynamically over multiple nodes – as is the case in an SPMD-style application – while maintaining the improved productivity of a global-level task-based programming approach.
10.3 ASF: Advanced Seismic Framework

Advanced Seismic Framework (ASF), introduced in Chapter 9, is used for our distributed GPU experimentation. In this work, extending the capabilities of ASF, the `smem_eta_1` kernel developed by Sai et al. [70] was adapted to the MPI-based distributed processing supported in ASF. To reduce MPI-based communication between GPUs on a single node, all GPUs on a node are allocated to a single MPI rank. The domain is subdivided in the slowest-varying \((x)\) dimension within an MPI rank, allowing exchanges between GPUs on a single node to be performed without MPI with a direct GPU-GPU memory copy operation using NVLink. Operations are submitted to a single CUDA stream per GPU.

To perform the halo exchange between MPI ranks, wavegrid elements need to be copied between GPUs on separate nodes. While the elements can be copied to a host-side staging buffer in CUDA pinned memory for transfer with MPI, CUDA-aware MPI allows for the use of device-side staging buffers or a direct transfer between wavegrid memory regions on separate nodes. A comparison of the performance of these methods on Cypress is given in Table 10.1; in this study, host-side staging buffers are used since these provide the best performance for this system.

The resulting implementation of ASF, with CUDA for GPU acceleration and MPI for inter-node communication, is called ASF-MPI and is used as a baseline in our experiments.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Host</th>
<th>Device</th>
<th>Direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>18.3</td>
<td>23.1</td>
<td>21.4</td>
</tr>
<tr>
<td>4</td>
<td>15.8</td>
<td>29.7</td>
<td>18.1</td>
</tr>
<tr>
<td>8</td>
<td>12.5</td>
<td>28.5</td>
<td>32.8</td>
</tr>
</tbody>
</table>

Table 10.1: Comparison of halo exchange techniques for MPI-CUDA implementation on Cypress, using host-side staging buffers in CUDA pinned memory, device-side staging buffers and direct transfer between wavegrid memory regions on separate nodes. Run with a grid size of \(1536^3\); runtimes shown in seconds.

10.4 Legion Implementation

This section describes the implementation of ASF in the Legion programming model, which is called ASF-Legion. The basic techniques used in this implementation are
described in Chapter 9. In this chapter, the implementation is extended to include accelerators using the CUDA programming interface. The basic element of computation in Legion is the *task*. For each task, the application developer specifies the logical regions (distributed memory managed by the Legion runtime) as inputs or outputs. The Legion runtime then determines which tasks can run at the same time, and schedules them for execution on the available processors.

Algorithm 4 (in Chapter 3) gives a high-level overview of the main solver in *ASF* applications. In the Legion implementation, the wavefield solution step (lines 3-5) is implemented as a task, and one task is launched for each subregion of the domain.

### 10.4.1 Parallelism in ASF-Legion, CUDA Implementation

The CUDA implementation of ASF-Legion contains two levels of parallelism: Legion tasks and CUDA threads. First, the domain is split into subdomains, and the Legion runtime distributes these subdomains among the Legion tasks. The second level, within each task, is CUDA parallelism, where a CUDA kernel is used to compute the subdomain on a GPU. In ASF-Legion, subdomains of the domain are assigned to a Legion task, which has its own GPU assigned to it. The kernel `smem_eta_1` developed by Sai et al. [70] is used in the ASF-Legion code – the same kernel used in the MPI version described previously. The domain decomposition scheme is analogous to the OpenMP version of ASF-Legion described in Chapter 9. Legion uses GASNet (InfiniBand conduit) for communication. Legion manages the transfer of memory between the host and GPUs and communication over the network.

### 10.5 Experimental Results

In this section, the performance results of the experiments conducted with ASF-MPI and ASF-Legion are presented. First, the experimental setup is described; second, weak and strong scaling performance of each parallel implementation is presented. Finally, profiling and analysis are provided.

Experiments are performed on the Cypress cluster at TotalEnergies Houston. Detailed information about the system, including the software stack used, is given in Table 10.2.

One MPI rank / Legion process is used per node (four GPUs per rank on Cypress). To account for timing variability, experiments are performed three times and the mean of these results is used. Generally, the experiments show little variability, suggesting the results are robust.
<table>
<thead>
<tr>
<th>System</th>
<th>Hardware Specs</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cypress</td>
<td>CPU: AMD EPYC 7F52 (16 cores)</td>
<td>GCC 8.3.1 + CUDA 11.2</td>
</tr>
<tr>
<td></td>
<td>Main Memory: 256 GB</td>
<td>OpenMPI 4.1.0</td>
</tr>
<tr>
<td></td>
<td>GPU: NVIDIA A100 (x4) 40 GB</td>
<td>GASNet 2020.3.0</td>
</tr>
<tr>
<td></td>
<td>Interconnect: InfiniBand HDR (2x)</td>
<td></td>
</tr>
</tbody>
</table>

Table 10.2: Hardware and software configuration of the experimental platform.

![Figure 10.1: Aggregated bandwidth test from OSU benchmark on Cypress.](image)

One straightforward way of evaluating the network bandwidth is to use micro benchmark suite such as OMB [48] from the network-based computing laboratory at the Ohio State University. In this study, we conducted tests with the point to point tests from MVAPICH OSU Micro-Benchmarks 5.8 for measuring the network bandwidth on Cypress. It can be observed in Fig. 10.1 that the InfiniBand interconnect on Cypress delivers aggregate bandwidth close to the theoretical peak bidirectional bandwidth of the dual rail HDR-IB on Cypress (48 GB/s). In our experiments, the average message size is 15 MB, which falls into the regime where the peak bandwidth is reached.

### 10.5.1 Scalability Results and Analysis—CUDA

Evaluation begins with the weak scaling of ASF-Legion and ASF-MPI with CUDA. The experiment uses a domain size of $1024^3$ grid points per node, expanded along the $x$ dimension. (For example, with two nodes, the total simulation size is $2048 \times 1024 \times 1024$, with a $256 \times 1024 \times 1024$ grid per GPU on Cypress.) Results are shown in Table 10.3 for up to 16 nodes on Cypress. The CUDA kernel in the ASF-MPI
implementation has a couple small indexing optimizations that were not applied to the ASF-Legion kernel due to lack of time. This results in the ASF-Legion code being slightly slower with one GPU than the ASF-MPI code, but otherwise does not affect scalability. ASF-Legion exhibits a higher parallel efficiency than ASF-MPI in these results.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>ASF-MPI [s]</th>
<th>ASF-Legion [s]</th>
<th>Parallel Efficiency [%] ASF-MPI, ASF-Legion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.7</td>
<td>10.1</td>
<td>100.0% , 100.0%</td>
</tr>
<tr>
<td>2</td>
<td>10.2</td>
<td>10.6</td>
<td>85.5% , 94.7%</td>
</tr>
<tr>
<td>4</td>
<td>12.4</td>
<td>11.0</td>
<td>70.6% , 91.9%</td>
</tr>
<tr>
<td>8</td>
<td>12.4</td>
<td>11.1</td>
<td>70.4% , 91.0%</td>
</tr>
<tr>
<td>16</td>
<td>12.7</td>
<td>11.6</td>
<td>68.8% , 86.8%</td>
</tr>
</tbody>
</table>

Table 10.3: CUDA weak scaling: runtime (stable around the reference is ideal) with 1-16 nodes; grid size 1024³ per node. Averaged over three trials.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Mem%</th>
<th>ASF-MPI [s]</th>
<th>ASF-Legion [s]</th>
<th>Parallel Efficiency [%] ASF-MPI, ASF-Legion</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>52%</td>
<td>34.4</td>
<td>37.5</td>
<td>100.0% , 100.0%</td>
</tr>
<tr>
<td>4</td>
<td>27%</td>
<td>20.3</td>
<td>20.1</td>
<td>84.8% , 93.2%</td>
</tr>
<tr>
<td>8</td>
<td>14%</td>
<td>12.5</td>
<td>11.2</td>
<td>69.1% , 83.3%</td>
</tr>
<tr>
<td>16</td>
<td>8%</td>
<td>8.3</td>
<td>7.9</td>
<td>52.0% , 59.2%</td>
</tr>
</tbody>
</table>

Table 10.4: Strong scaling: elapsed time for grid size 8192 × 1024 × 1024 and corresponding parallel efficiency. Averaged over three trials. Mem% shows the amount of available GPU memory utilized across all nodes.

The strong scaling results for ASF-MPI and ASF-Legion are shown in Table 10.4 and Fig. 10.2. As discussed in Section 10.4.1, the GPU experiments only decompose in the x-dimension; therefore a wider domain size of 8192 × 1024 × 1024 is used to ensure there is enough parallel work to do. The general scaling trends are similar
to the weak-scaling results: ASF-Legion out-performs ASF-MPI at up to 16 nodes (64 A100 GPUs). There is a drop in parallel efficiency at 16 nodes observed in both the MPI and Legion implementations. The drop was also observed in the CPU experiments ([64]), but is more pronounced with GPUs, due to the extra task decomposition and communication needed between the GPUs.

10.5.2 Profiling

Legion includes a built-in profiler to aid the developer in optimizing performance of applications. The profiler shows a time trace of task execution and memory copies during the simulation. During the development of this application, Legion’s built-in profiler was critical to diagnosing performance issues and improving the code. Figure 10.3 shows screenshots from running the profiler for two- and eight-node ASF simulations, respectively. The red rectangles in each “GPU Proc” row show task executions on the GPUs over time. From these traces, the cause of reduced parallel efficiency under large node counts can be diagnosed. While the two-node profiling result shows a healthy GPU utilization, nearly 100% total between the four GPUs, the eight-node profiling result shows that the overall GPU utilization across all nodes hovers around 75%. The large gaps that are seen between kernel executions are occupied by data transfer between GPUs within the node and between nodes over the network. However, we do note that Legion is somewhat able to overlap communication and computation in the eight-node case. There are few-to-no instances where the node is completely idle.

We also used the NVIDIA NSYS profiler to collect a corresponding time trace from the MPI version of the code. Fig. 10.4 shows one timestep of execution, with GPU computation and P2P communication activities shown in blue and orange, respectively.

10.6 Conclusion

ASF-Legion achieved competitive results against ASF-MPI on up to 16 nodes of Cypress (64 A100 GPUs), in both raw performance and scalability. Even though this application is highly regular, and therefore has less potential for improvement over MPI, it is promising that the task-based approach yields results comparable to the MPI approach in performance. Implementing ASF from a task-based perspective was relatively straightforward, although intrusive. Legion is not directly interchangeable with MPI, due to the particularities of Legion regarding memory management using logical regions. Also, the ASF-Legion implementation is designed from a global view
Figure 10.2: Strong scaling: Throughput (left) and speedup (right), grid size 8192 × 1024 × 1024, for 1000 time steps on Cypress with CUDA. Higher is better.
Figure 10.3: Sample extracted from Legion profiler time trace with two nodes (top) and eight nodes (bottom), each with the first node shown. Total simulation grid size is $8192 \times 1024 \times 1024$. Each time period is roughly 100 milliseconds. Each “GPU Proc” shows a time trace of task execution on a GPU on the node, and each red rectangle is a task computing the stencil on the subdomain of the grid assigned to that GPU. Each task roughly corresponds to one timestep, with the other processors (not shown) computing the other subdomains of the grid during the same period.
Figure 10.4: Sample extracted from NSYS profiler time trace with one node for approximately one timestep. Total simulation grid size is $4096 \times 1024 \times 1024$. Each pair of rows represents one GPU (numbered). Blue regions indicate stencil computation, and orange regions indicate P2P data transfer between GPUs.

for task-based parallelism. Nonetheless, the modular structure of ASF (see Section 10.3) allowed reuse of much of the code of the ASF-MPI application code.

In future work, closing the gap in performance between ASF-Legion and ASF-MPI is a main goal. Legion’s control replication feature (discussed in Section 10.2) will be evaluated in ASF-Legion to overcome the scalability limitations seen for large number of nodes. Over-decomposition will also be explored, which would expose more task parallelism to the Legion runtime. Finally, adding other kernels, such as IO and physics-oriented post-processes, would allow further opportunities for efficient task scheduling.
Chapter 11

Reverse time migration

Up to this point, all of our experiments have focused on solving the acoustic wave equation (3.1) in the forward time direction. As expected, since stencil problems are relatively regular in computational pattern, we have not seen significant improvements from task-based programming. In this chapter, we extend the Legion+CUDA work of Chapter 10 to create a “full” reverse-time migration application and show some preliminary performance improvements.

11.1 Introduction

Reverse-time migration [82] is a technique for imaging the subsurface of the Earth. It consists of two phases: a forward pass, in which the wave equation is solved in the forward-time direction given a source term, and a backward pass, in which the equation is solved in the backward-time direction given input at the receivers. The correlation is then computed between the resulting wavefields; correlation indicates features of interest in the subsurface.

11.2 Computational considerations

A high-level description of the RTM algorithm implemented in Minimod is shown in Algorithm 6. Computationally, compared to only performing the forward pass, the main additional challenge is saving the wavefield at each timestep. This is required to compute the correlation between the wavefields at each step of the backward pass. Possible approaches are to save the wavefield snapshots to disk, and to save to memory.
Data: $f$: source; $f_s$: input from receivers

Result: $v$: seismic image

1. $u_{fw}^0 := 0$

2. **Forward pass**

3. for $n ← 1$ to $T$

   4. for each point in wavefield $u_{fw}^n$ do

   5. \[ \text{Solve Eq. 3.2 (left hand side) for wavefield } u_{fw}^n; \]

   6. end

   7. $u_{fw}^n = u_{fw}^n + f^n$ (Eq. 3.2 right hand side);

   8. Save $u_{fw}^n$ to memory or disk;

9. end

10. **Backward pass**

11. $u_{bw}^0 := 0$

12. $v := 0$

13. for $n ← T$ to 1

14. for each point in wavefield $u_{bw}^n$ do

15. \[ \text{Solve Eq. 3.2 (left hand side) for wavefield } u_{bw}^n; \]

16. end

17. $u_{bw}^n = u_{bw}^n + f_n^n$ (add receiver inputs);

18. Load $u_{fw}^n$ from memory or disk;

19. $v = v + \text{Correlation}(u_{fw}^n, u_{bw}^n)$;

20. end

Algorithm 6: Reverse-time migration high-level description
Here, we incorporate two techniques to reduce memory consumption and I/O time. The first is a simple compression scheme to reduce the size of wavefield snapshots. For each snapshot, we store the minimum and maximum values as 32-bit floats, and each wavefield value is then discretized between the two as an 8-bit integer. This decreases the size of each snapshot by 75%, at the cost of accuracy.

The second efficiency technique is to skip some timesteps. The wavefield is then saved in the forward pass, and correlated in the backward pass, every \( n \)th timestep, where \( n \) is called the “skip factor”. Among our findings is that using disk I/O with large skip factors (>20), we are able to effectively hide the cost of I/O by overlapping with computation. However, such large skip factors are found to produce erroneous results from a scientific standpoint.

### 11.3 Results

For this experiment, we use TotalEnergies’ Cypress machine described in Section 10.5.

Our first experiment saves snapshots to disk (Lustre4 parallel filesystem) and uses a skip factor of 25. Because we don’t yet have an MPI-based version of this
configuration, we instead use as a baseline a Legion-based version in which a fence is added before and after I/O operations, similar to how an MPI-based version would operate. As shown in Fig. 11.1, we are able to achieve significant improvement in throughput by omitting the fence in I/O operations.

The source of the large throughput can be seen in profiling data shown in Fig. 11.2 (with I/O fence on top and without on bottom). In the fenced version, there is no overlap of I/O and computation, resulting in low relative GPU utilization. In the version without the fence, the I/O output to disk is nearly perfectly overlapped with GPU computation.

In this first experiment, we relied on using a large skip factor of 25 in order to hide the overhead of disk I/O. However, in practice it has been found experimentally that using such large skip factors results in an erroneous output image. When using a more realistic skip factor (less than 5), we are unable to overlap communication and computation in this manner. An alternative that significantly reduces total I/O overhead is to save snapshots to main memory. However, this drastically reduces the size of the domain and simulation duration we can perform. We are in the process of obtaining results using this method.

### 11.4 Conclusion

Reverse time migration is an extension of the forward pass studied in previous chapters to form a complete application. The extra computational kernels added by RTM (especially I/O needed to store the forward wavefield) result in an imbalanced computation that can potentially be improved using task-based programming.

In a first experiment, we showed significantly improved throughput by overlapping computation and disk I/O. However, this method is contingent on using a very large skip factor that does not produce scientifically valid results. We are in the process of repeating the experiment using memory I/O, which will allow smaller skip factors but only works for smaller domains.
Figure 11.2: Profile of the execution corresponding to Fig. 11.1 with (top) and without (bottom) the I/O fence during forward pass. Green blocks are stencil computation on GPUs; red blocks are I/O.
Chapter 12

Conclusions and Discussion

In this dissertation, our overall research objective was to explore the current state-of-the-art in task-based programming, and evaluate different task-based programming models in the context of an industrial-strength application. We have provided an introduction and literature review of current work in the field. We introduced our application, Minimod, which simulates parts of reverse time migration. We described our experimental results from introducing different task-based programming models to Minimod. Finally, we evaluate an extended version of Minimod that simulates a full reverse time migration application.

Several key takeaways can be made from this work. We have seen architectural differences in the performance of tasks, tradeoffs between static work distribution and dynamic load balancing, and tradeoffs associated with fine-grained task dependency analysis as compared to bulk synchronization.

We used OpenMP to create a GPU-accelerated version of the code, where we found that utilizing the CPU and GPU simultaneously does not improve performance over using the GPU alone. We used OpenMP tasks to accelerate to multiple GPUs simultaneously — a task traditionally handled by multiple MPI ranks within a node. We also used a new plugin available in LLVM to extend this code to multiple nodes without needing MPI — however, we did not obtain good results using this method.

We perform experiments on two different A64FX platforms, highlighting how the performance of the loop- and task-based versions varies between platforms and compilers, and compatibility issues that arise with the task-based version on one compiler. On the A64FX platform, we also create a hybrid MPI+OpenMP task version of the code, as an alternative to the more standard hybrid MPI+OpenMP loop approach. Performance was very similar between loop and task versions. We highlight some productivity issues using OpenMP tasks in this manner.
We then turn our attention to a global task-based programming model, Legion. First, we create a hybrid Legion+OpenMP version of the code, and show results competitive to MPI+OpenMP. Then we create a Legion+CUDA version utilizing GPUs with a previously-written high-performance CUDA kernel. We again show a result competitive to MPI+CUDA. However, the Legion versions suffer a decrease in efficiency at high node counts; we suggest this may be due to the singular point of task generation, a common problem in task-based programming.

Finally, we form a full reverse time migration application by adding more kernels to Minimod. We find that if we skip enough timesteps for disk I/O, we can effectively overlap computation with disk I/O and obtain substantial performance improvements. However, this approach was found to produce scientifically unsatisfactory results. To get past this issue, we are attempting an approach using memory I/O instead.

Starting from this work, some future research directions would be interesting to explore. The distributed versions of Minimod described here do not attempt to perform load-balancing. Understanding the tradeoff of moving tasks to utilize idle resources, losing locality and involving potentially expensive memory transfers over the network, is an important next step. There is a potential to use task execution overlap to hide communication; this should be explored further.

The MPI+OpenMP task version of the code was intended to improve performance by potentially overlapping computation with communication using OpenMP tasks, however performance did not improve. A deeper analysis needs to be performed into this version of the code to understand the lack of improvement.

Evaluating other task-based programming models, both shared-memory and distributed, would be useful. This includes a comparison to the results presented here in terms of both performance and productivity.

In the reverse time migration code, a remaining challenge is to improve performance using tasks without sacrificing the quality of the final result. An approach is in progress to use memory I/O instead of disk I/O.
Bibliography


[47] NERSC. *Cori*. URL: https://docs.nersc.gov/systems/cori/.


Appendix A

Legion Example

The following is taken from Legion’s tutorial, and is an example of computing the daxpy operation (scalar times a vector added to another vector). It has been abbreviated and annotated for clarity. The original is available here.

```c
void top_level_task(const Task *task,
                     const std::vector<PhysicalRegion> &regions,
                     Context ctx, Runtime *runtime)
{
  int num_elements = 1024;
  int num_subregions = 4;

  // The code below sets up the data to be operated on by creating a logical region
  Rect<1> elem_rect(0, num_elements -1);
  IndexSpace is = runtime->create_index_space(ctx, elem_rect);
  FieldSpace input_fs = runtime->create_field_space(ctx);
  {
    FieldAllocator allocator =
      runtime->create_field_allocator(ctx, input_fs);
    allocator.allocate_field(sizeof(double), FID_X);
    allocator.allocate_field(sizeof(double), FID_Y);
  }
  FieldSpace output_fs = runtime->create_field_space(ctx);
  {
    FieldAllocator allocator =
      runtime->create_field_allocator(ctx, output_fs);
    allocator.allocate_field(sizeof(double), FID_Z);
  }
  LogicalRegion input_lr = runtime->create_logical_region(ctx, is, input_fs);
  LogicalRegion output_lr = runtime->create_logical_region(ctx, is,
```
output_fs);

// Divide the domain into num_subregions equal subregions
Rect<1> color_bounds(0, num_subregions - 1);
IndexSpace color_is = runtime->create_index_space(ctx, color_bounds);
IndexPartition ip = runtime->create_equal_partition(ctx, is, color_is);
LogicalPartition input_lp = runtime->get_logical_partition(ctx, input_lr, ip);
LogicalPartition output_lp = runtime->get_logical_partition(ctx, output_lr, ip);

// Task to initialize the data. One task is launched for each subregion through
// the use of of IndexLauncher.
ArgumentMap arg_map;
IndexLauncher init_launcher(INIT_FIELD_TASK_ID, color_is,
    TaskArgument(NULL, 0), arg_map);
init_launcher.add_region_requirement(
    RegionRequirement(input_lp, 0/*projection ID*/,
        WRITE_DISCARD, EXCLUSIVE, input_lr));
init_launcher.region_requirements[0].add_field(FID_X);
runtime->execute_index_space(ctx, init_launcher);

init_launcher.region_requirements[0].privilege_fields.clear();
init_launcher.region_requirements[0].instance_fields.clear();
init_launcher.region_requirements[0].add_field(FID_Y);
runtime->execute_index_space(ctx, init_launcher);

const double alpha = drand48();
// Task to perform daxpy. Again, one task for each
IndexLauncher daxpy_launcher(DAXPY_TASK_ID, color_is,
    TaskArgument(&alpha, sizeof(alpha)), arg_map);
daxpy_launcher.add_region_requirement(
    RegionRequirement(input_lp, 0/*projection ID*/,
        READ_ONLY, EXCLUSIVE, input_lr));
daxpy_launcher.region_requirements[0].add_field(FID_X);
daxpy_launcher.region_requirements[0].add_field(FID_Y);
daxpy_launcher.region_requirements[1].add_field(FID_Z);
runtime->execute_index_space(ctx, daxpy_launcher);

// Cleanup
void init_field_task(const Task *task, 
    const std::vector<PhysicalRegion> &regions, 
    Context ctx, Runtime *runtime)
{
    FieldID fid = *(task->regions[0].privilege_fields.begin());
    const int point = task->index_point.point_data[0];
    const FieldAccessor<WRITE_DISCARD, double, 1> acc(regions[0], fid);
    Rect<1> rect = runtime->get_index_space_domain(ctx,
        task->regions[0].region.get_index_space());
    for (PointInRectIterator<1> pir(rect); pir(); pir++)
        acc[*pir] = drand48();
}

void daxpy_task(const Task *task, 
    const std::vector<PhysicalRegion> &regions, 
    Context ctx, Runtime *runtime)
{
    const double alpha = *((const double*)task->args);
    const FieldAccessor<READ_ONLY, double, 1> acc_x(regions[0], FID_X);
    const FieldAccessor<READ_ONLY, double, 1> acc_y(regions[0], FID_Y);
    const FieldAccessor<WRITE_DISCARD, double, 1> acc_z(regions[1], FID_Z);
    Rect<1> rect = runtime->get_index_space_domain(ctx,
        task->regions[0].region.get_index_space());
    for (PointInRectIterator<1> pir(rect); pir(); pir++)
        acc_z[*pir] = alpha * acc_x[*pir] + acc_y[*pir];
}

int main(int argc, char **argv)
{
    Runtime::set_top_level_task_id(TOP_LEVEL_TASK_ID);
    // Here, the subtasks are registered.
    {
        TaskVariantRegistrar registrar(TOP_LEVEL_TASK_ID, "top_level");
        registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
        Runtime::preregister_task_variant<top_level_task>(registrar, "
}
TaskVariantRegistrar registrar(INIT_FIELD_TASK_ID, "init_field");
registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
registrar.set_leaf();
Runtime::preregister_task_variant<init_field_task>(registrar, "init_field");

TaskVariantRegistrar registrar(DAXPY_TASK_ID, "daxpy");
registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
registrar.set_leaf();
Runtime::preregister_task_variant<daxpy_task>(registrar, "daxpy");

return Runtime::start(argc, argv);