

RICE UNIVERSITY

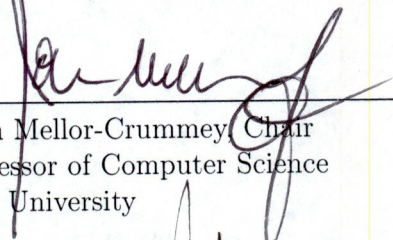
**Performance Analysis and Optimization of a  
Hybrid Distributed Reverse Time Migration  
Application**

by

**Sri Raj Paul**

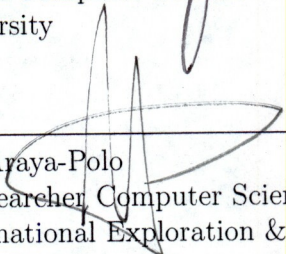
A THESIS SUBMITTED  
IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE  
**Master of Science**

APPROVED, THESIS COMMITTEE:



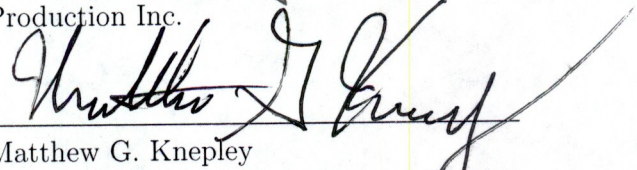
---

John Mellor-Crummey, Chair  
Professor of Computer Science  
Rice University



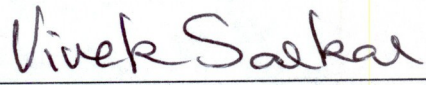
---

Mauricio Araya-Polo  
Senior Researcher, Computer Science  
Shell International Exploration &  
Production Inc.



---

Matthew G. Knepley  
Assistant Professor, Computational and  
Applied Mathematics  
Rice University



---

Vivek Sarkar  
Professor of Computer Science  
E.D. Butcher Chair in Engineering  
Rice University

Houston, Texas

December, 2015

## ABSTRACT

### Performance Analysis and Optimization of a Hybrid Distributed Reverse Time Migration Application

by

Sri Raj Paul

Applications to analyze seismic data employ scalable parallel systems to produce timely results. This thesis describes our experiences of applying performance tools to gain insight into an MPI+OpenMP code that performs Reverse Time Migration (RTM) to analyze seismic data and also assess the capabilities of available tools for analyzing the performance of a sophisticated application that employ both message-passing and threaded parallelism. The tools provided us with insights into the effectiveness of the domain decomposition strategy, the use of threaded parallelism, and functional unit utilization in individual cores. By applying insights obtained from Rice University's HPCToolkit and hardware performance counters, we were able to improve the performance of a distributed-memory RTM code by roughly 30 percent.

## Acknowledgments

I would first like to thank my advisor, Prof. John Mellor-Crummey for his mentorship and guidance. His guidance and interest were invaluable to the success of this research. I would also like to thank my co-advisor, Dr. Mauricio Araya-Polo for his guidance. Along with his work, he found time to help me with my research.

I would like to thank my committee members Prof. Matthew G. Knepley and Prof. Vivek Sarkar for their time and patience and valuable feedback on the thesis.

I would like to thank Dr. Amik St-Cyr for providing me with test cases and Dr. Detlef Hohl for supporting this research.

I would like to thank Karthik, Priyanka, Raji, Ramya and Rishi for their friendship and lot of hangouts. I am very glad to have friends like Arkabandhu, Hamim and Debarshi with whom I spend most of my time outside the college. I would like to thank all of them for supporting me when I needed it the most.

I would like to thank Deepak, Kuldeep, Prasanth, Rohan, Shailesh, Shams and Sourav for being good friends. I would also like to thank my other friends at Rice all of whom I cannot mention here.

I would like to thank my sister, Santhini for proofreading my thesis and also helping to get through my difficult times. Last but not the least, I would like to thank my parents who supported me throughout my journey.

This work was supported by Shell International Exploration & Production Inc. under research agreement PT46021 and by the DOE Office of Science under cooperative agreement DE-SC0010473.

# Contents

Abstract	ii
Acknowledgments	iii
List of Illustrations	vi
List of Tables	x
<b>1 Introduction</b>	<b>1</b>
1.1 Distributed Reverse Time Migration . . . . .	2
1.2 Performance Analysis Tools . . . . .	5
1.3 Thesis Statement . . . . .	7
1.4 Contributions . . . . .	7
<b>2 Functionality Evaluation of Performance Analysis Tools</b>	<b>9</b>
2.1 Experimental Setup . . . . .	10
2.1.1 Hardware . . . . .	10
2.1.2 Software . . . . .	12
2.1.3 Configuration . . . . .	12
2.2 HPCToolkit . . . . .	13
2.3 ITAC . . . . .	16
2.4 MAP . . . . .	21
2.5 PCM . . . . .	22
2.6 PerfExpert . . . . .	24
2.7 VTune . . . . .	25
2.8 Summary of the Tool Evaluation . . . . .	30

<b>3</b>	<b>Related Work</b>	<b>32</b>
3.1	Parallelizing RTM . . . . .	32
3.2	Domain-specific Language Frameworks . . . . .	32
3.3	Overlapping Communication with Computation . . . . .	33
<b>4</b>	<b>Analysis and Tuning of DRTM using HPCToolkit</b>	<b>35</b>
4.1	Initial Assessment . . . . .	35
4.1.1	Structure of the Application . . . . .	36
4.2	Performance Analysis and Code Optimization . . . . .	38
4.2.1	Reduce Overhead due to Abstractions . . . . .	39
4.2.2	Reduce Thread Level Load Imbalance . . . . .	43
4.2.3	Overlap Communication with Computation . . . . .	44
4.2.4	Improving Data Reuse in Cache . . . . .	49
4.2.5	Reduce Process Level Load Imbalance . . . . .	52
4.2.6	Eager Operations on Received Data . . . . .	54
4.2.7	Vectorizing Stencil Computation . . . . .	55
4.3	Assessing the Tuned Application . . . . .	61
4.3.1	Verification . . . . .	66
<b>5</b>	<b>Conclusions and Future Work</b>	<b>67</b>
	<b>Bibliography</b>	<b>69</b>

## Illustrations

1.1	A representation 3D seismic data with multiple slabs. Slabs at shallow depths have more data points compared to deep ones. . . . .	3
1.2	A domain decomposition of 3D seismic data among four processes by dividing X-Y plane. The size of the slabs decreases with depth because the number of data points at deeper depth is fewer compared to shallow ones. . . . .	4
1.3	Different phases in a time step . . . . .	5
2.1	Single node with two Xeon processors [1] . . . . .	11
2.2	<b>hpcviewer</b> : A top-down view of calling contexts. The bottom pane associates times with each level of a dynamic call chain, including procedures, loops, and source lines. The top pane shows source code associated with the highlighted item in the call path. . . . .	15
2.3	<b>hpctraceviewer</b> : This screenshot of HPCToolkit's <b>hpctraceviewer</b> user interface shows a detail of an execution trace of DRTM. Each row in the display represents an OpenMP worker thread or MPI rank. In this case, rows 2nd from the top and bottom represent MPI ranks; the rest represent OpenMP worker threads. Time flows left to right. .	16
2.4	ITAC : This screenshot of ITAC's summary page shows the distribution of time spent in user code and MPI code. It also shows the distribution of various MPI calls. . . . .	17

2.5	ITAC : This screenshot of ITAC's event timeline view shows a detail of a DRTM execution. Each row in the view represents an MPI rank. Red rectangles represent processing associated with MPI communications. Blue rectangles represent execution of user code. Black lines represent messages between the MPI ranks. . . . .	18
2.6	ITAC : This screenshot of ITAC's <i>Function Profile</i> shows load imbalance in communication among processes. . . . .	19
2.7	ITAC : This screenshot of ITAC's <i>Message Profile</i> shows interaction between different processes. The intensity of messaging is color coded from red to blue where red denotes a large number of messages and blue denotes a small number of messages. . . . .	20
2.8	ITAC : This screenshot of ITAC's <i>Function Profile</i> shows distribution of different MPI calls. . . . .	20
2.9	MAP : This screenshot of MAP shows metric (top), source (middle) and top-down (bottom) view. . . . .	22
2.10	PCM : processor usage statistics for a one MPI process, 16 thread configuration on our small data set input. . . . .	23
2.11	PerfExpert : overall performance overview of DRTM's stencil computation . . . . .	25
2.12	VTune : summary showing hotspots . . . . .	26
2.13	VTune : summary of CPU usage by different threads. . . . .	27
2.14	VTune : bottom-up view of execution of DRTM . . . . .	28
2.15	VTune : top-down view of execution of DRTM. . . . .	29
4.1	<b>hpcviewer</b> visualization of execution of the unoptimized version of DRTM. . . . .	36
4.2	<b>hpctraceviewer</b> visualization of the entire execution - rose represents idleness (32%) and green represents the stencil computation (32.5 %). . . . .	37

4.3	<code>hpctraceviewer</code> visualization of a sequence of few timesteps - magenta represents the stencil computation (36.5%) and light brown represents idleness (32.5%). . . . .	38
4.4	<code>hpctraceviewer</code> showing the division of a single time step for two processes each containing eight threads. . . . .	39
4.5	<code>hpcviewer</code> output after the introduction of <code>SHALLOW</code> copy. <code>memcpy</code> has disappeared from the list of top time consuming functions compared to Figure 4.1 . . . . .	42
4.6	A screenshot of <code>hpctraceviewer</code> showing idleness occurring during stencil computation for an MPI process with eight threads . . . . .	43
4.7	Smaller tiles at the high end of each dimension cause imbalance when OpenMP <code>static</code> scheduling is used for a collapsed 2D loop nest over the tiles. . . . .	44
4.8	Idleness within stencil computation removed using dynamic scheduling	45
4.9	A screenshot of <code>hpctraceviewer</code> shows considerable waiting (pink) occurs after stencil computation (green) implying lack of communication-computation overlap . . . . .	46
4.10	Idleness (red) occurs during the stencil computation phase (performed seperately for each halo region and each slab along the Z axis) and other operations after introducing asynchronous progress thread. . . .	47
4.11	Waiting (green) after the stencil computation is reduced by the introducing a separate communication thread. . . . .	50
4.12	Idleness (green) after the stencil computation is more for processes at the top and bottom. MPI ranks are marked on the left side. . . . .	53



4.13	4x4 configuration (4 in X and 4 in Y direction) with each process marked with its MPI rank showing the difference in the number of neighbors for each block in the 2D partition of the domain across MPI ranks. For example, top-left process (rank 12) has 2 neighbors whereas the middle four processes (5,6,9,10) have 4 neighbors. . . . .	53
4.14	4x4 configuration (4 in X and 4 in Y direction) with each process marked with its MPI rank showing partitions with reduced sizes for MPI ranks managing the domain interior to reduce load imbalance. . . . .	54
4.15	Processes towards edges wait (green) after stencil computation for halo exchange data to arrive even after <code>wait_any</code> is used. . . . .	55
4.16	<b>hpcviewer</b> visualization of profile of the final optimized version . . . . .	62
4.17	<b>hpctraceviewer</b> visualization of few timesteps before performing process level load imbalance reduction optimization. Idleness (green) after the stencil computation is more for processes at the top and bottom. . . . .	63
4.18	<b>hpctraceviewer</b> visualization of few timesteps of the final optimized version. Waiting after the stencil computation is shown in green. MPI ranks are marked on the left side. . . . .	63
4.19	The graph shows improvement when optimizations are applied. 4x4, 2x8, and 8x2 are domain decompositions of the X-Y plane on a cluster with Sandy bridge family of processors (Xeon E5-2670). The performance improvements are observed when different domain decompositons are used. . . . .	64
4.20	The graph shows improvement when optimizations are applied. 4x4, 2x8, and 8x2 are domain decompositions of the X-Y plane on a cluster with Westmere family of processors (Xeon X5660). Performance improvements due to optimizations are observed on the new cluster with a different hardware specification. . . . .	66

## Tables

2.1	Performance analysis tools used for the analysis of DRTM . . . . .	12
2.2	Performance analysis tools capability matrix . . . . .	31
4.1	Hardware performance counter values for the stencil computation loop	51
4.2	Reduction in number of cache accesses after loop interchange . . . . .	52
4.3	Profile comparison of original and optimized versions with percent of total execution time in parentheses . . . . .	62

# Chapter 1

## Introduction

Seismic imaging helps to identify subsurface structures and thus gain insight into different geological characteristics such as the type of rocks and their distribution [2]. Seismic waves reflected by subsurface structures are used to get information regarding subsurface layers. The changes in the properties of the subsurface layers, such as variations in types of rock type, cause reflections.

**Reverse Time Migration (RTM)** [3] creates an image of the subsurface layers by simulating the propagation of an acoustic wave through them [4]. During simulation, first the medium is excited by introducing a wavelet. Next, forward wave propagation is mathematically simulated using an acoustic wave equation. Then, RTM repeats the same in the backward direction; it starts from the data recorded by the receivers and propagates the wave field back in time (backward propagation). Finally, a cross-correlation between both fields (forward and backward) is performed to generate an output image.

The acoustic wave propagation equation is a Partial Differential Equation (PDE) [5] which is solved using Finite Difference (FD) [6] or Finite Element (FE) [7] methods. RTM gives more accurate results than previous methods such as Wave Equation Migration (WEM) [8] but is also computationally more expensive (at least one order of magnitude higher than WEM). Since RTM is computationally expensive, a single compute node might take very long time to compute a result. One way to resolve this issue is to accelerate RTM by distributing work among multiple compute nodes.

## 1.1 Distributed Reverse Time Migration

Distributed Reverse Time Migration is an approach for performing reverse time migration on a distributed memory computer system. Such systems use message passing to share data between compute nodes using MPI [9] and threaded parallelism to maximize utilization of functional units within a node. Within a node, threaded parallelism can be expressed using different programming models such as CUDA [10], OpenACC [11], OpenCL [12] or OpenMP [13]. One way to enable different node level programming models is to use an abstraction layer that enables one to plug in different models by implementing a predefined set of APIs.

In this thesis, we work with an implementation of **Distributed Reverse Time Migration** developed at Shell International Exploration & Production Inc., which we abbreviate as **DRTM**. DRTM is an iterative code that applies a high-order stencil on a 3D block of data. Figure 1.1 shows a representation of the 3D data. As it can be seen in Figure 1.1, DRTM uses a multiblock algorithm in which the number of data points in a block vary at different depths. As depth increases, the speed of the acoustic wave increases and, therefore, fewer data points are collected at greater depths compared to shallow ones. Thus, the number of data points in each X-Y plane is less than or equal to the one above. Instead of decreasing the number of points in each X-Y plane, an approximation is made by creating slabs/blocks containing multiple X-Y planes as shown in Figure 1.1. The resolution of X-Y planes within a single slab is uniform. The resolution of planes decreases as slab depth increases. The introduction of slabs improves computational efficiency with an acceptable reduction in accuracy.

Repeated application of stencils over large 3D slabs of data is costly. Hence, the computation needs to be partitioned among multiple compute nodes to get timely

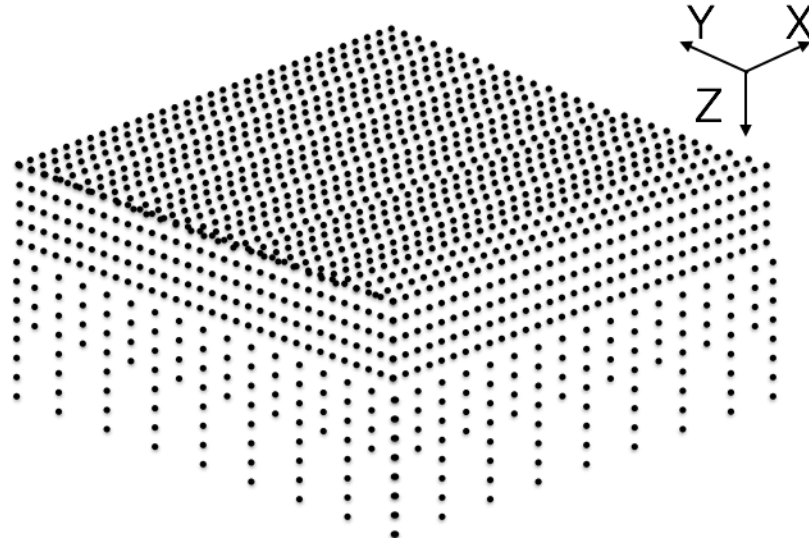


Figure 1.1 : A representation 3D seismic data with multiple slabs. Slabs at shallow depths have more data points compared to deep ones.

results. A common way of distributing data among nodes for seismic calculations is to divide the X-Y plane. It is simpler to partition the calculation along the X and Y dimensions, along which the number of data points is uniform, rather than along the Z-axis, along which the number of data points decreases with depth. For example, partitioning data across four processes is shown in Figure 1.2. Each color represents an MPI process.

Data is partitioned across processes so that each process can perform computation on data that is local to the compute node on which the process resides. Each MPI process requires a narrow slab of boundary data points from each of its neighbors during stencil computation. This boundary data slab exchanged with each neighbor is called halo region. During execution, each MPI process sends/receives halo regions to/from corresponding neighbors. Exchange of halo regions between neighbor processes is overlapped with the stencil computation of non-halo regions (internal

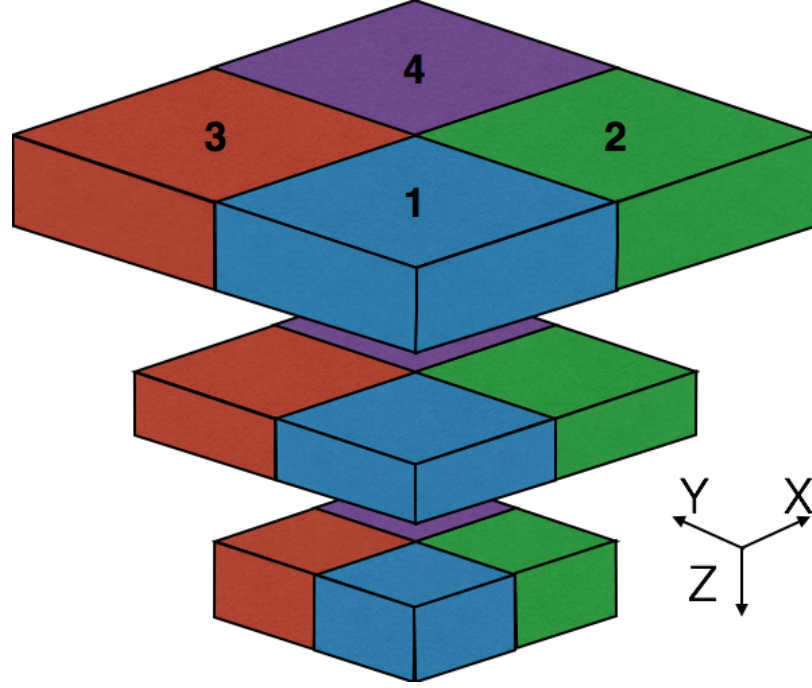


Figure 1.2 : A domain decomposition of 3D seismic data among four processes by dividing X-Y plane. The size of the slabs decreases with depth because the number of data points at deeper depth is fewer compared to shallow ones.

regions). This helps to overlap communication with computation. The number of data points are different at slab boundaries as we move along Z-axis. For a smooth transition at the slab boundary, interpolation is performed. Thus, a single time step in DRTM consists of five phases as shown in Figure 1.3.

Each time step starts with the application of the stencil computation on data in the halo regions so that they can be sent to neighbors. DRTM then initiates non-blocking MPI calls for point-to-point communication with the aim of overlapping communication of halo regions with the computation of stencil for internal regions. Next, the code performs the stencil computation for non-halo regions while communications are pending for data in the halo regions. After the stencil computation finishes, each MPI process waits for the arrival of halo data sent prior to the stencil

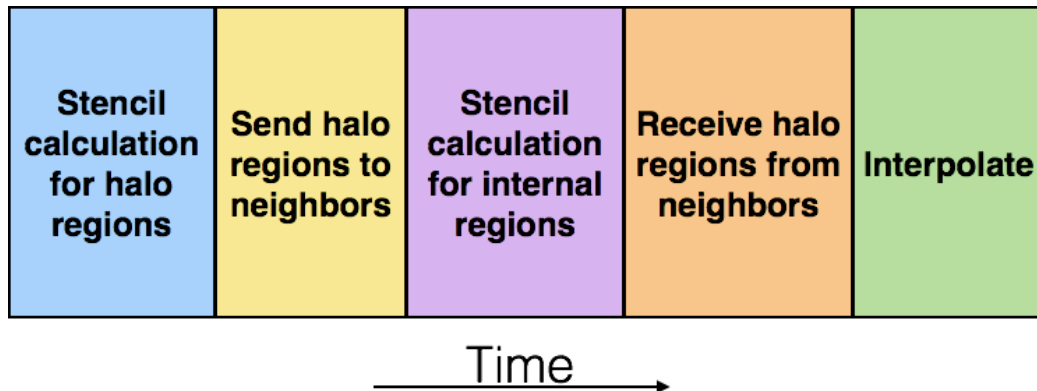


Figure 1.3 : Different phases in a time step

computation. Finally, the code receives data from the halo exchanges and performs interpolation where necessary.

In MPI+OpenMP applications such as DRTM, distribution of data and computation among different compute nodes can run into several problems such as load imbalance or inefficient communication. In a sequential program, there is no need to worry about such problems. The additional complexities make it difficult to analyze the execution of distributed hybrid parallel applications. Therefore, we need performance analysis tools to gain insight into the runtime characteristics of such applications.

## 1.2 Performance Analysis Tools

Profiling is a type of program analysis that measures runtime characteristics of the program such as memory usage and the number of function invocations. Performance analysis tools used for profiling an application should have the capability to collect performance data during execution of the application. There are mainly two ways to collect performance data - instrumentation and sampling.

**Instrumentation** is the introduction of performance monitoring instructions at points of interest to measure some particular characteristics of the program. These additional instructions are introduced into the program in two ways - source instrumentation and binary instrumentation. In source instrumentation, the programmer adds extra instructions explicitly to the source code whereas, in binary instrumentation additional instructions are injected to the binary using an external tool. Source instrumentation requires recompilation of instrumented program or usage of instrumented library. Source instrumentation is difficult when non-instrumented libraries are used (since we do not have the source code for the library). For instance, Intel ITAC [14] uses a source instrumented MPI library to collect runtime characteristics of communication. On the other hand, for binary instrumentation extra instructions are added to the program either statically before execution or dynamically during execution using an external tool such as Intel Pin [15]. Binary instrumentation has the advantage that recompilation of the source is not needed and also can be used in legacy systems whose source code is not available.

**Sampling**, on the other hand, does not involve modification of code or binary. Sampling collects performance data at regular intervals based on an event that causes an interrupt or a register to overflow. For example, a tool might sample the program counter every millisecond. Sampling is light weight compared to instrumentation, and, therefore, it maintains runtime characteristics that closely resemble the original execution without profiling. Since sampling is faithful to the original execution, it can detect issues that appear during execution that might get hidden or distorted as a side-effect of instrumentation. The number of samples collected does not depend on program characteristics (e.g. many small functions or few large functions), but only on the sampling frequency. Therefore, the amount of data collected can be easily



changed by adjusting the sampling rate. Rice University’s HPCToolkit [16] and Intel VTune [17] are examples of sampling based performance analysis tools.

### 1.3 Thesis Statement

*Efficiently mapping complex scientific applications to modern clusters using the MPI+OpenMP programming model is difficult. Without guidance from performance tools, often many opportunities for improving performance go unnoticed.*

Modern clusters include parallelism at multiple levels. They include distributed parallelism across nodes, threaded parallelism within a node and instruction level parallelism within a core. To make the matter more complex, they come with a deep memory hierarchy too. To exploit this parallelism, applications need to distribute data across nodes. To fully utilize the parallelism within a node they need to employ threading. Typically such systems use MPI+X programming model where X is a shared memory programming model such as OpenMP. Performance analysis tools are required to get insight into such complex hybrid applications. Such insights are very helpful to improve the performance of those applications.

### 1.4 Contributions

In the first part of the thesis, we evaluate the functionality of various performance analysis tools for analyzing a hybrid MPI+OpenMP application. We use DRTM as a representative hybrid application for this evaluation purpose. We mainly concentrate our study on performance analysis tools that use sampling to collect performance profile (with the exception of ITAC, which uses instrumentation). We define metrics that are important for the performance analysis of hybrid applications and summarize our findings on tool comparison based on those metrics.

In the second part of the thesis, we use Rice University’s HPCToolkit for detailed analysis of DRTM. Insights gained from HPCToolkit helped us to improve the performance of DRTM by roughly 30% and also provide insight into further optimization opportunities.

## Chapter 2

# Functionality Evaluation of Performance Analysis Tools

In this chapter, we describe our experiences using different performance analysis tools to obtain insight into the performance of DRTM. We evaluate the capabilities of various tools for analyzing the performance of a sophisticated application that employs both message-passing and threaded parallelism.

The hybrid MPI+OpenMP programming model helps to exploit both inter- and intra-node parallelism and, therefore, is a good fit for large applications running on emerging multi-core architectures. However, programming and analyzing such hybrid applications is harder than stand-alone MPI or OpenMP applications. For hybrid applications, identifying performance bottlenecks and opportunities for improvements is more difficult, and hence assistance from performance analysis tools is therefore necessary. In hybrid programming models, analyzing MPI and OpenMP together is more effective than looking at MPI or OpenMP individually, and therefore tools that provide a unified view of MPI and OpenMP are necessary. Understanding such hybrid applications requires analysis at multiple levels:

- domain decomposition and interprocess communication for a distributed-memory parallelization,
- threaded parallelism on a node, and
- functional unit and cache utilization within a core.

We evaluated several performance analysis tools with this criterion for hybrid MPI+OpenMP applications using DRTM as a representative hybrid application. We principally consider tools that use sampling to collect performance data rather than instrumentation with the exception of ITAC.

Tools used in this study are:

1. HPCToolkit from Rice University [16]
2. ITAC from Intel [14]
3. MAP from Allinea [18]
4. PCM from Intel [19]
5. PerfExpert from TACC [20]
6. VTune from Intel [17]

## 2.1 Experimental Setup

This section describes hardware, software, and configuration aspects of platforms used in our experiments.

### 2.1.1 Hardware

The experiments are run on a cluster with multiple compute nodes. Figure 2.1 shows the topology of a compute node with two sockets. Each socket has a 2.6GHz Intel Xeon E5-2670 processor. Each processor has eight cores with one thread per core (two threads per core if simultaneous multithreading, known as hyper-threading for Intel processors is enabled). Each compute node contains 128GB RAM. Compute nodes

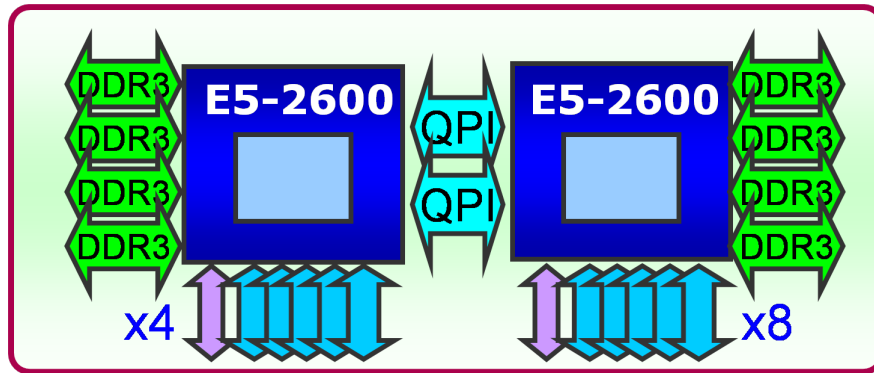


Figure 2.1 : Single node with two Xeon processors [1]

are connected with a fat-tree topology [21] using an InfiniBand interconnect [22] with a uni-directional bandwidth of 56Gb/s between nodes.

Some of the new features available in the E5-26xx [1] family processors compared to the previous generation are,

1. 32 nm process technology
2. Intel Advanced Vector Extensions (Intel AVX)
3. Intel Turbo Boost Technology 2.0
4. High Bandwidth Last Level Cache
5. High Bandwidth/Low Latency modular on-die Ring Interconnect
6. Integrated Memory Controller with 4 channel DDR3
7. CPU and PCI Express integrated on single chip

Each node in the cluster has two Xeon processors connected using QuickPath [23] interconnect.

### 2.1.2 Software

The cluster uses the Linux operating system with kernel version 2.6.32. Intel compiler suite 2014 (includes icc/icpc version 14 and MPI version 4.1) is used to build the programs. A prototype of the OMPT [24] performance tools interface for OpenMP is used to collect performance measurements for threads in an execution using OpenMP. To use the latest Intel performance analysis tools, DRTM is also built using Intel compiler suite 2015 (includes icc/icpc version 15 and MPI version 5). The performance of DRTM is better on Intel compiler suite 2014, and, therefore, we use version 14 to compile executables for our analysis and optimization work. Table 2.1 shows the list of performance analysis tools used in the exercise.

Table 2.1 : Performance analysis tools used for the analysis of DRTM

Tool	Version
ITAC	9.0 Update 3
MAP	5.0.1
PCM	2.6
PerfExpert	4.1.1
HPCToolkit	5.3.2 (revision 4692)
hpcviewer	5.3.2 (revision 1760)
hpctraceviewer	5.3.2 (revision 1833)
VTune	Amplifier XE 2015 Update 2

### 2.1.3 Configuration

We ran our experiments on a small (2.7GB) and a large (17GB) input data set. The small data set experiment uses 4 MPI processes; the large data set experiment uses

16 MPI processes. Each node in the cluster executes two MPI processes, and each process contains eight OpenMP threads.

We launch the large data set experiment on the cluster as follows,

```
mpirun -np 16 -ppn 2 -env OMP_NUM_THREADS 8 -hostfile Host_file
drtm_mpi -c acceptance_test.sim
```

`drtm_mpi` is the name of the DRTM executable and `acceptance_test.sim` is the input configuration file. The input configuration file specifies various parameters including domain decomposition, slab sizes and stencil parameters to be used. High-level specifications indicate how to partition work among the processors. For example, `partitions-xy 4x4`, specifies a two-dimensional partitioning of the input data in the X-Y plane into 16 tiles arranged in a 4x4 grid. Submission of jobs to the cluster is managed using Load Sharing Facility (LSF).

## 2.2 HPCToolkit

HPCToolkit [16] helps to measure and analyze the performance of applications executing on single-core, multi-core and distributed-memory systems. It uses periodic sampling based on timers or hardware performance counters to trigger collection of call stack profiles and call stack traces. It associates measured metrics with the full calling context in which costs are incurred. The overhead introduced during execution is low and, therefore, maintains the runtime characteristics close to original execution. It does not collect enormous amount of data and, therefore, scales well for large parallel systems. It can analyze fully optimized applications and associate performance data with source code.

HPCToolkit consists of five main components as listed below,

- **hpcrun** - collects statistical call path profiles of events of interest.
- **hpcstruct** - analyzes executables and recovers program structure.
- **hpcprof** - correlates information about the application's structure (from **hpcstruct**) with call path profiles (from **hpcrun**) and creates a performance database
- **hpcviewer** - presents performance data in a code-centric fashion with top-down, bottom-up, and flat views.
- **hpctraceviewer** - presents performance data in a time-centric view at multiple levels of abstraction.

HPCToolkit lets the user specify events used for sampling triggers during execution. Sampling multiple events is possible in a single run although the number of events that can be monitored simultaneously is limited. HPCToolkit allows usage of native hardware counter, PAPI [25] or timer events.

Figure 2.2 shows a **hpcviewer** visualization of performance data collected using an event named REALTIME. Many other events such as CPU cycles, cache misses can be used to understand how different functional units are being utilized.

The bottom left pane of Figure 2.2, is a navigation pane displaying a *calling context view*. This view represents a top-down decomposition of execution represented as a calling context tree rooted at **main**, with call paths descending from **main** representing the structure of the computation. *Callers view* (bottom up view), and *flat view* (based on static source code view from files) are also available. Clicking on any function in these views shows the source code of the function in the upper



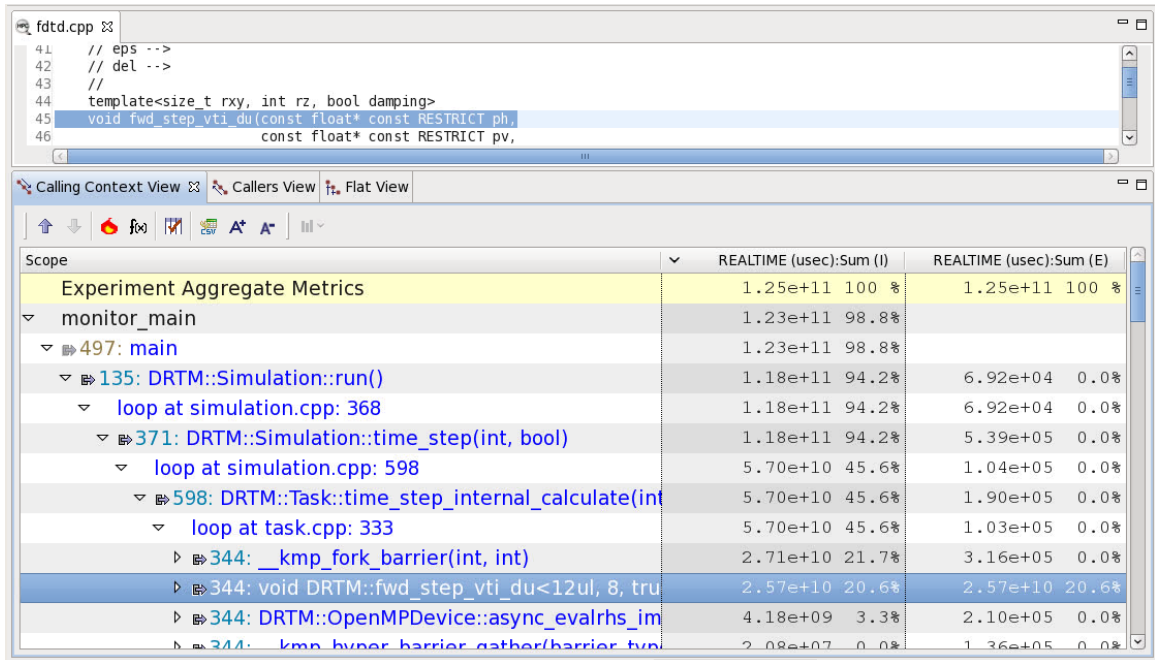


Figure 2.2 : **hpcviewer** : A top-down view of calling contexts. The bottom pane associates times with each level of a dynamic call chain, including procedures, loops, and source lines. The top pane shows source code associated with the highlighted item in the call path.

source code viewer. **hpcviewer** helps users understand where execution time is spent. **hpctraceviewer** provides insight into how a program's execution unfolds over time. A timeline view using **hpctraceviewer** is shown in Figure 2.3. In **hpctraceviewer**, the X-axis represents time flowing from left to right and groups of threads along the Y-axis represent processes.

Together, the output of **hpcviewer** and **hpctraceviewer** give a unified view of the interaction between processes, threads within a process and functional units within a core.

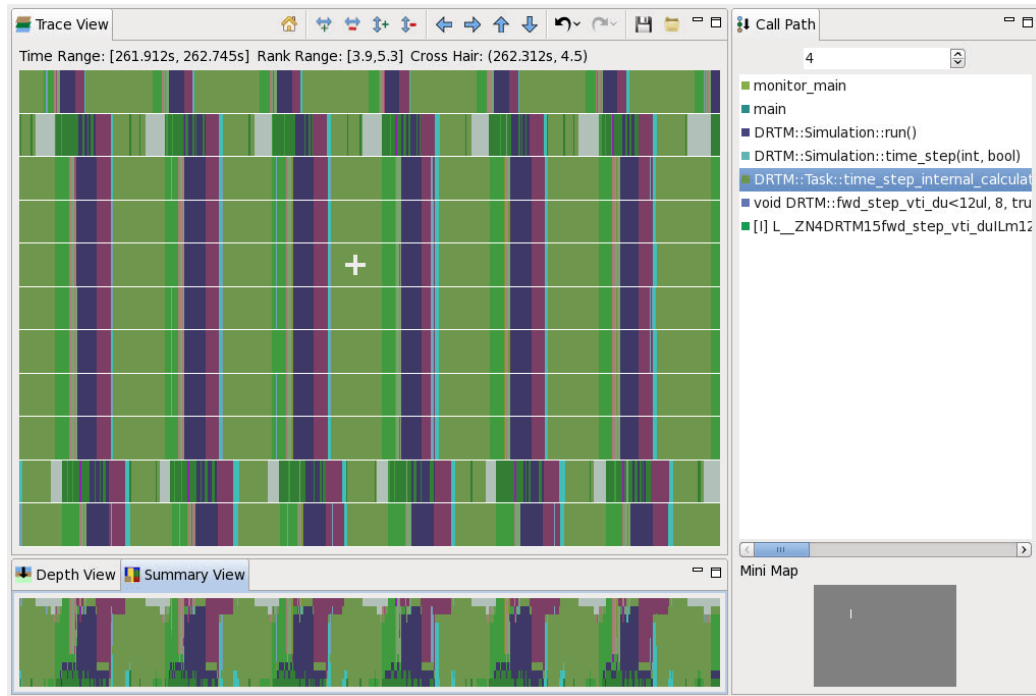


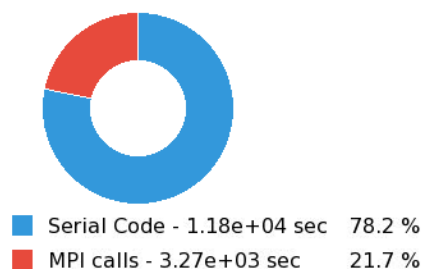
Figure 2.3 : `hpctraceviewer` : This screenshot of HPCToolkit’s `hpctraceviewer` user interface shows a detail of an execution trace of DRTM. Each row in the display represents an OpenMP worker thread or MPI rank. In this case, rows 2nd from the top and bottom represent MPI ranks; the rest represent OpenMP worker threads. Time flows left to right.

## 2.3 ITAC

Intel’s ITAC [14] is a performance analysis tool that is used to analyze MPI applications. It gives detailed statistics of MPI invocations. It provides an *event timeline* which shows MPI communication over time. It includes a *message profile* that indicates the number of messages exchanged between process pairs. Unlike sampling based tools that collect data regularly due to interrupts caused by counter overflows or interval timers, ITAC collects data whenever an MPI library function is invoked. ITAC achieves this using an instrumented MPI library, and details it gives about MPI usage are quite informative. To get more insight into the application, ITAC provides

**Summary:** drtm\_mpi.stfTotal time: **1.5e+04** sec. Resources: **16** processes, **8** nodes.[Continue >](#)**Ratio**

This section represents a ratio of all MPI calls to the rest of your code in the application.

**Top MPI functions**

This section lists the most active MPI functions from all MPI calls in the application.

MPI_Waitall	3.1e+03 sec (20.6 %)
MPI_Barrier	143 sec (0.955 %)
MPI_Isend	14.9 sec (0.0993 %)
MPI_Irecv	7.31 sec (0.0487 %)
MPI_Wtime	0.374 sec (0.00249 %)

Figure 2.4 : ITAC : This screenshot of ITAC's summary page shows the distribution of time spent in user code and MPI code. It also shows the distribution of various MPI calls.

an option to instrument either the code or binary. Binary instrumentation is done using a utility called `itcpin`. Code instrumentation requires recompilation of the application and binary instrumentation has more overhead than sampling.

ITAC's initial view shows a summary of the distribution of time spent in MPI and user code as shown in Figure 2.4. The summary also includes a distribution of different MPI calls.

Figure 2.5 shows ITAC's *event timeline* view which is used to analyze how the MPI invocations are distributed over time. Here X-axis represents time flowing from left to right and MPI processes are shown along Y-axis. ITAC's *Event timeline* only distinguishes between user code and MPI code. Red rectangles represent processing associated with MPI communications. Blue rectangles represent execution of user code. Black lines represent messages between the MPI ranks. ITAC does not support fine-grain analysis of user code. The call stack while making an MPI call is not shown

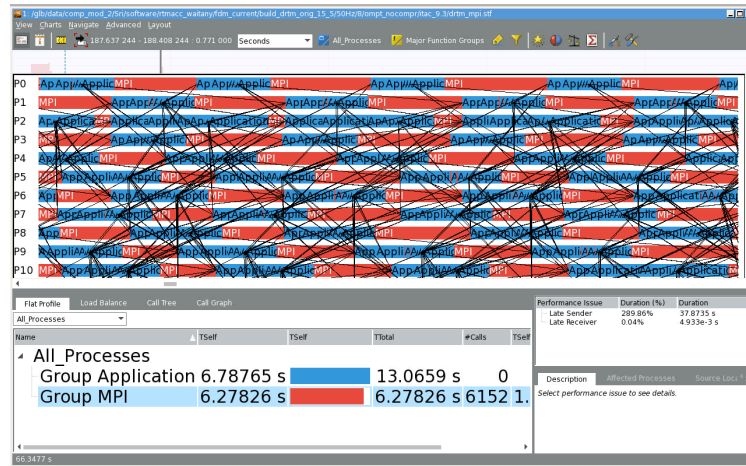


Figure 2.5 : ITAC : This screenshot of ITAC’s event timeline view shows a detail of a DRTM execution. Each row in the view represents an MPI rank. Red rectangles represent processing associated with MPI communications. Blue rectangles represent execution of user code. Black lines represent messages between the MPI ranks.

by default. Call-stack collection during execution time can be enabled at the expense of increased execution overhead.

At the bottom of Figure 2.5 is the *Function Profile* which provides a summary of the distribution of user and MPI code. The MPI summary group can be expanded to see the load imbalance in communication between processes as shown in Figure 2.6. Similarly expanding *Group Application* helps to investigate load imbalance in the user code.

Another useful view is the *Message Profile* that helps to analyze the pattern of interactions between processes as shown in Figure 2.7. The intensity of messaging is color coded from red to blue where red denotes a large number of messages and blue denotes a small number of messages. From Figure 2.7, we can see that each process interacts with only a few other processes (many boxes are white, denoting no message exchange between corresponding processes). Each box can be expanded further to see the exact amount of time spend on messaging with the respective partner.

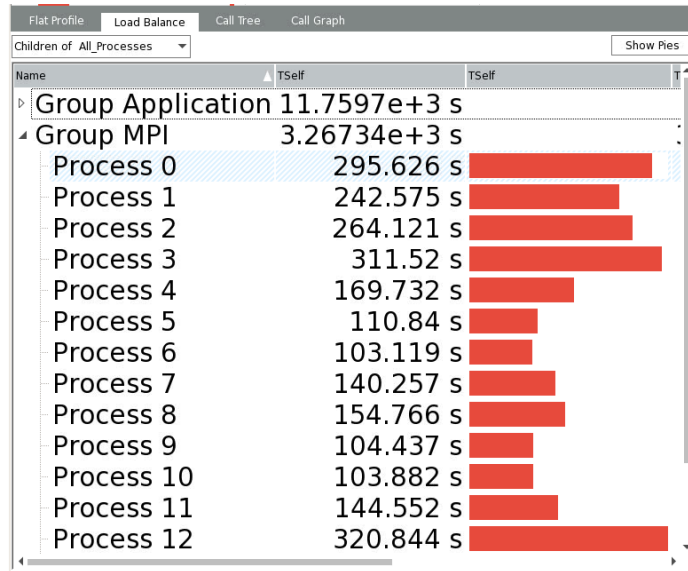


Figure 2.6 : ITAC : This screenshot of ITAC's *Function Profile* shows load imbalance in communication among processes.

ITAC's *Function Profile* (at the bottom of Figure 2.5) can be expanded to show distribution of various MPI calls. Figure 2.8 gives a distribution of different MPI calls based on various parameters such as time, the number of calls.

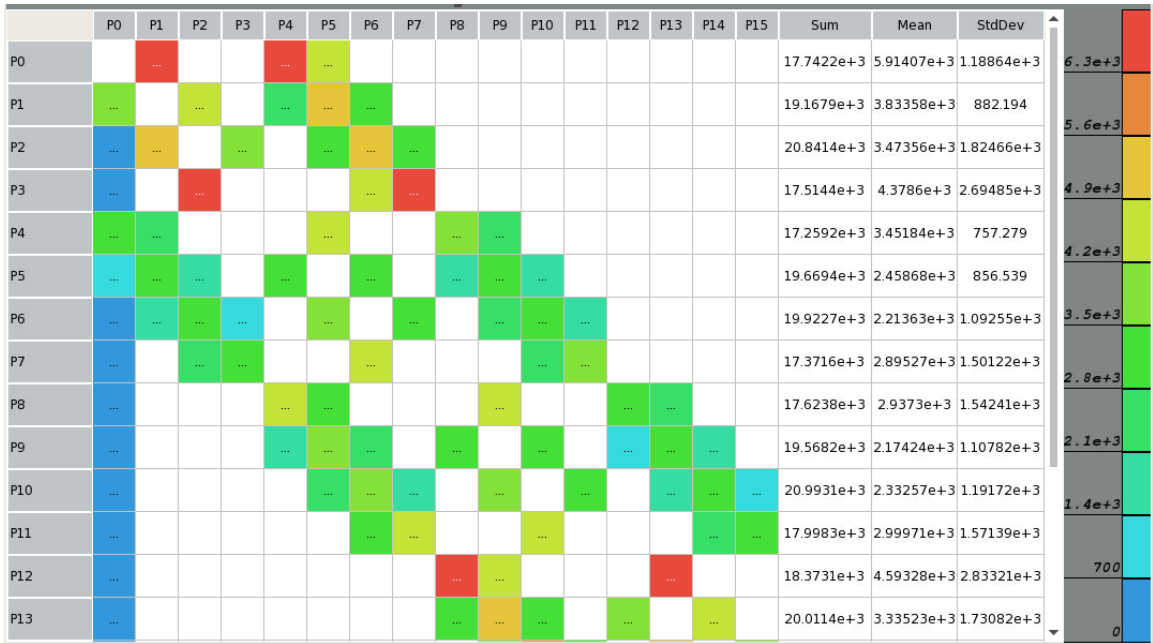


Figure 2.7 : ITAC : This screenshot of ITAC's *Message Profile* shows interaction between different processes. The intensity of messaging is color coded from red to blue where red denotes a large number of messages and blue denotes a small number of messages.

Name	Tself	Tself
All Processes		
Group Application	11.7597e+3 s	
MPI_Comm_size	15e-6 s	
MPI_Comm_free	920.998e-6 s	
MPI_Comm_rank	15e-6 s	
MPI_Comm_dup	170.564e-3 s	
MPI_Initialized	36e-6 s	
MPI_Finalize	7.09599e-3 s	
MPI_Recv	170e-6 s	
MPI_Get_proc...	2e-6 s	
MPI_Isend	14.9239 s	
MPI_Irecv	7.31188 s	
MPI_Wtime	374.29e-3 s	
MPI_Waitall	3.10105e+3 s	
MPI_Allgather	289.998e-6 s	
MPI_Allgatherv	836.998e-6 s	
MPI_Send	143e-6 s	
MPI_Barrier	143.492 s	
MPI_Allreduce	6.88799e-3 s	

Figure 2.8 : ITAC : This screenshot of ITAC's *Function Profile* shows distribution of different MPI calls.

## 2.4 MAP

Allinea MAP [18] is a profiling tool that uses sampling rather than instrumentation to keep the overhead low. It produces high-level visualizations of Memory usage, CPU usage, and MPI usage. MAP provides a flat view displaying functions that can be sorted based on various metrics and a top-down view. MAP does not offer a bottom up view of execution. It associates performance data with source code and displays measurements in a source code viewer along with the percent of execution time (and some OpenMP and MPI statistics). The top-down view starts from `main` and enables one to navigate down the call-chain to callees. MAP's top-down view shows the percent of execution time along OpenMP and MPI statistics associated with each function. MAP presents a timeline-based view of various metrics such as branch instructions, floating point instructions, memory usage, MPI calls duration, MPI bytes sent. These metrics are grouped as CPU instructions, CPU time, Memory, IO, and MPI.

MAP only displays predefined profiles. For example, users cannot see the number of cache misses since it is not defined in any profile. Also timeline visualizations cannot be zoomed-in for closer inspection.

Performance data of DRTM viewed using MAP is shown in Figure 2.9. The top pane is the metric view (timeline) which can be expanded to see a variety of performance metrics (e.g. memory usage, MPI calls duration). The middle displays the source code view that displays source code along with associated percentage of execution time. The bottom pane contains top-down and flat views. This view is used to associate performance data with different functions.

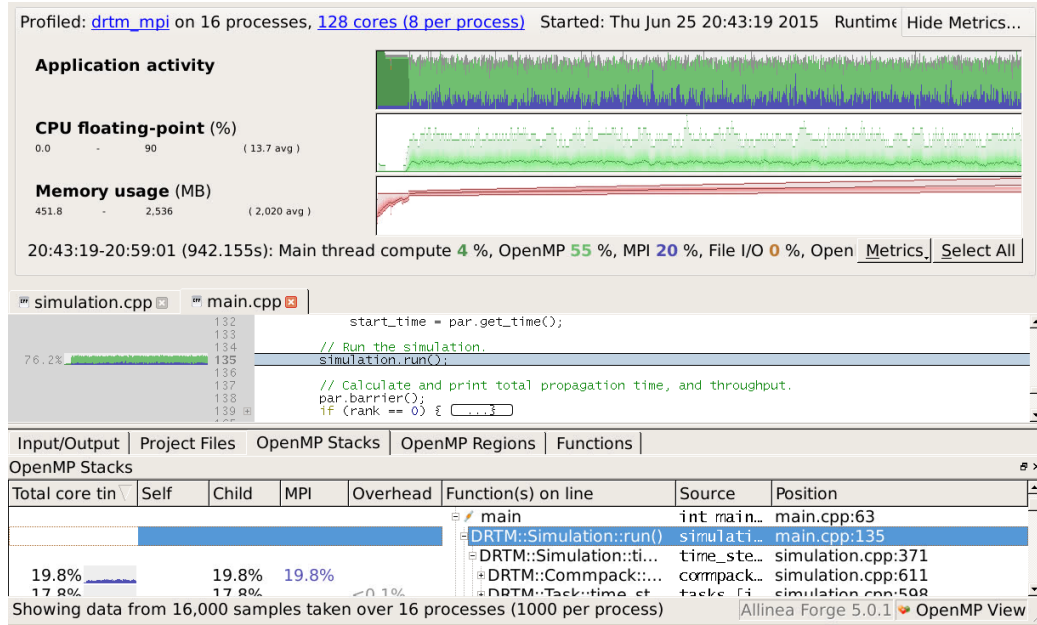


Figure 2.9 : MAP : This screenshot of MAP shows metric (top), source (middle) and top-down (bottom) view.

## 2.5 PCM

Intel PCM [19] is mainly used as a monitoring tool for online introspection. It displays various metrics that can be viewed online while the application is running or aggregated over the execution of an application. PCM includes a memory module that monitors memory bandwidth and a PCIe module that monitors PCIe bandwidth. There is also a module to monitor transactional synchronization. In addition to visualizing performance data, PCM displays power utilization information as well.

PCM provides a set of high-level interfaces that can be invoked from a C++ application to provide real-time CPU performance metrics. PCM uses the Performance Monitoring Units (PMU) [26] in Intel processors to monitor both core and uncore metrics, including instructions retired, elapsed clock cycles, data traffic in Quickpath interconnect among many others.



In addition to APIs for online introspection PCM provides a command line utility and several graphical utilities based on those APIs. We use the command line utility provided by PCM for profiling DRTM. PCM monitors and reports performance metrics for individual nodes and cannot be used for multiple node MPI applications. We evaluated PCM using the smaller data test case (not the bigger data test case used with other tools which takes very long time to run on a single node) on a single node.

Core (SKT)	EXEC	IPC	FREQ	AFREQ	L3MISS	L2MISS	L3HIT	L2HIT	L3CLK	L2CLK	READ	WRITE	TEMP	
0	0	1.29	1.14	1.13	1.15	6578 M	8252 M	0.20	0.67	0.25	0.01	N/A	N/A	42
1	0	1.29	1.14	1.13	1.15	6547 M	8232 M	0.20	0.67	0.24	0.01	N/A	N/A	40
2	0	1.31	1.16	1.13	1.15	6555 M	8241 M	0.20	0.67	0.24	0.01	N/A	N/A	41
3	0	1.29	1.14	1.13	1.15	6372 M	8070 M	0.21	0.67	0.24	0.01	N/A	N/A	42
4	0	1.01	0.90	1.13	1.15	3475 M	4270 M	0.19	0.17	0.13	0.01	N/A	N/A	40
5	0	0.97	0.85	1.13	1.15	5118 M	6083 M	0.16	0.17	0.19	0.01	N/A	N/A	43
6	0	0.96	0.85	1.13	1.15	5025 M	5955 M	0.16	0.17	0.19	0.01	N/A	N/A	44
7	0	1.01	0.89	1.13	1.15	3524 M	4343 M	0.19	0.17	0.13	0.01	N/A	N/A	44
8	1	0.98	0.86	1.13	1.15	5426 M	6314 M	0.14	0.15	0.20	0.01	N/A	N/A	25
9	1	1.37	1.21	1.13	1.15	7286 M	9020 M	0.19	0.66	0.27	0.01	N/A	N/A	27
10	1	1.34	1.18	1.13	1.15	6632 M	8369 M	0.21	0.67	0.25	0.01	N/A	N/A	26
11	1	1.01	0.89	1.13	1.15	3796 M	4647 M	0.18	0.17	0.14	0.01	N/A	N/A	29
12	1	1.34	1.18	1.13	1.15	6848 M	8602 M	0.20	0.67	0.26	0.01	N/A	N/A	28
13	1	1.31	1.16	1.13	1.15	6886 M	8621 M	0.20	0.67	0.26	0.01	N/A	N/A	31
14	1	0.97	0.86	1.13	1.15	5303 M	6273 M	0.15	0.17	0.20	0.01	N/A	N/A	27
15	1	0.97	0.86	1.13	1.15	5639 M	6614 M	0.15	0.16	0.21	0.01	N/A	N/A	29
-----														
SKT	0	1.14	1.01	1.13	1.15	43 G	53 G	0.19	0.57	0.20	0.01	0.00	0.00	39
SKT	1	1.16	1.03	1.13	1.15	47 G	58 G	0.18	0.56	0.22	0.01	0.00	0.00	25
-----														
TOTAL	*	1.15	1.02	1.13	1.15	91 G	111 G	0.19	0.56	0.21	0.01	0.00	0.00	N/A
-----														
Instructions retired: 78 T ; Active cycles: 77 T ; Time (TSC): 4265 Gticks ; C0 (active,non-halted) core residency: 98.04 %														
C1 core residency: 0.28 %; C3 core residency: 0.01 %; C6 core residency: 0.00 %; C7 core residency: 1.67 %;														
C2 package residency: 0.00 %; C3 package residency: 0.00 %; C6 package residency: 0.00 %; C7 package residency: 0.00 %;														
PHYSICAL CORE IPC : 1.02 => corresponds to 25.45 % utilization for cores in active state														
Instructions per nominal CPU cycle: 1.15 => corresponds to 28.79 % core utilization over time interval														
Intel(r) QPI data traffic estimation in bytes (data traffic coming to CPU/socket through QPI links):														

Figure 2.10 : PCM : processor usage statistics for a one MPI process, 16 thread configuration on our small data set input.

PCM shows various statistics down to the granularity of each core. For example in Figure 2.10, core-0 has 1.14 instructions per cycle (IPC) and 6578 million L3 misses. It also aggregates the result from each core and displays the total at the bottom. In our experimental platform, each node contains two sockets and therefore results are aggregated to SKT 0 and SKT 1. PCM records only a pre-configured set of metrics. It does not provide an option to specify more events.

## 2.6 PerfExpert

PerfExpert [20] is built on top of HPCToolkit. PerfExpert uses HPCToolkit to collect performance data, analyzes the data and creates a summary. PerfExpert recognizes loop nests that might cause a bottleneck and suggests recommendations to improve performance. PerfExpert also supports automatic optimization.

PerfExpert runs HPCToolkit multiple times to collect data using different performance counters. This is because HPCToolkit does not multiplex hardware counter events. PerfExpert must run a program several times to collect a large set of events since a processor's performance monitoring unit can collect only a small number of counters at once. Then PerfExpert analyzes the performance database created by HPCToolkit and identify sections of code that might cause a bottleneck. PerfExpert reports a summary performance assessment (as shown in Figure 2.11) along with a set of recommendations to improve the performance of the application. PerfExpert lacks a timeline view and hence cannot provide insight into how execution unfolds over time.

PerfExpert allows the user to set a threshold. A threshold of five percent means code fragments that run for more than 5% of the total execution time are considered for analysis and optimization. Figure 2.11 is the result of an execution of DRTM with threshold set to 5%. The PerfExpert report shown in Figure 2.11 indicates that overall execution of the stencil is good (1.01 cycles per instruction) but data accesses are taking too long since each data access takes an average of 3.84 cycles. PerfExpert also provides suggestions about how to improve performance.

```
Loop in function L_ZN4DRTM15fwd_step_vti_duILm12ELi8ELb1EEEvPKfS2_PfS3_S2_PK6float2S2
S2 S2 PKjS2 S2 77 par loop0 2 2987 in fdtd.cpp:85 (58.64% of the total runtime)
```

ratio to total instrns	%	0.....25.....50.....75.....100
- floating point	100.0	*****
- data accesses	59.4	*****
* GFLOPS (% max)	35.2	*****
- packed	34.1	*****
- scalar	1.0	*
-----		
performance assessment	LCPI	good.....okay.....fair.....poor.....bad
* overall	1.01	>>>>>>>>>>>>>>
* data accesses	3.84	>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
- L1d hits	2.00	>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- L2d hits	0.75	>>>>>>>>>>>>>>
- L3d hits	0.31	>>>>>
- LLC misses	0.78	>>>>>>>>>>>>>>
* instruction accesses	0.04	>
- L1i hits	0.00	
- L2i hits	0.00	
- L2i misses	0.04	>
* data TLB	0.00	
* instruction TLB	0.00	
* branch instructions	0.01	
- correctly predicted	0.01	
- mispredicted	0.00	
* floating-point instr	0.17	>>>
- slow FP instr	0.00	
- fast FP instr	0.17	>>>

Figure 2.11 : PerfExpert : overall performance overview of DRTM's stencil computation

## 2.7 VTune

Intel’s VTune Amplifier [17] is a profiler with a collection of useful predefined profiles. It includes some automatic analysis profiles such as **hotspot analysis** which reports where CPUs are spending most of the time, **lock and wait analysis** which reports program points where the application is spin waiting. There are few other profiles that enable data collection from a set of related hardware performance counters. Measurement data from hardware performance counters can be examined to find performance bottlenecks. If the number of events selected for collection is more than the number of performance counter registers, VTune multiplexes events to get counts

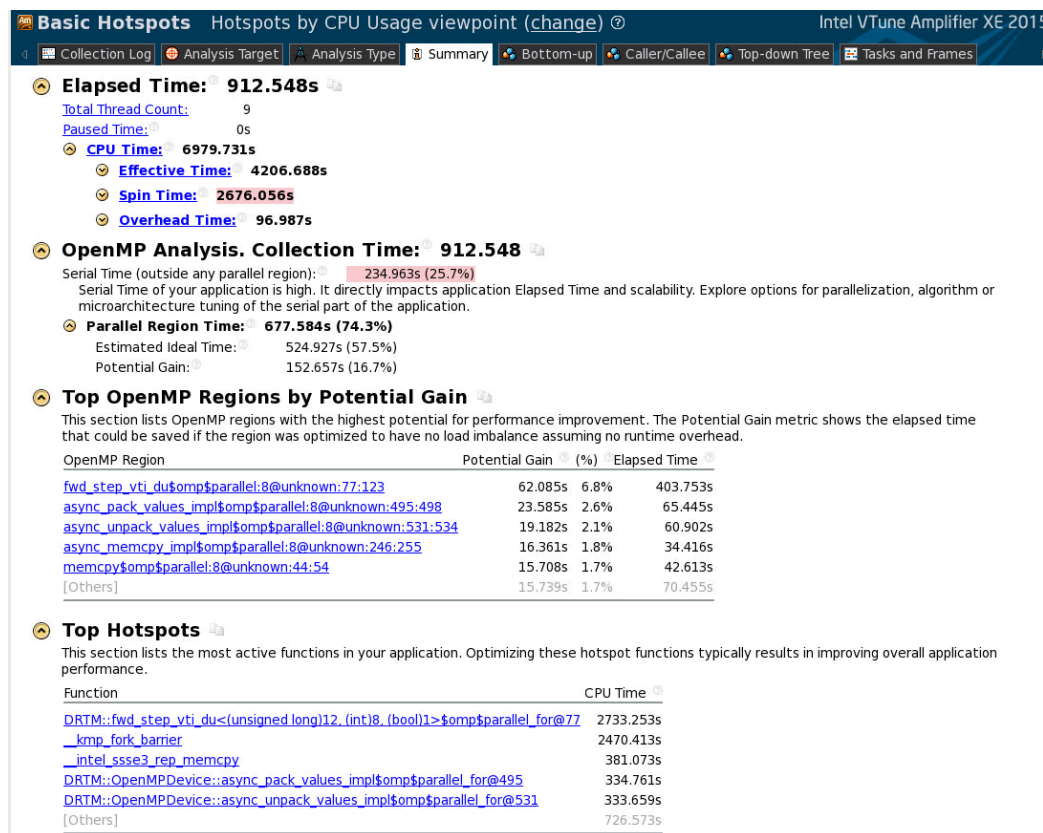


Figure 2.12 : VTune : summary showing hotspots

for all requested metrics in a single execution. VTune also gives the option to use multiple runs instead of multiplexing.

VTune is designed for collecting data about multi-threaded single process applications. VTune creates multiple output folders - one for each MPI process when multi-process applications are profiled. There is no unified interface to view the output of all processes. To use VTune for multi-process applications, first run the application using ITAC to get an overall picture of the entire execution. Then use VTune to collect performance data from few processes of interest.

Similar to ITAC, VTune starts with a summary page as shown in Figure 2.12 that

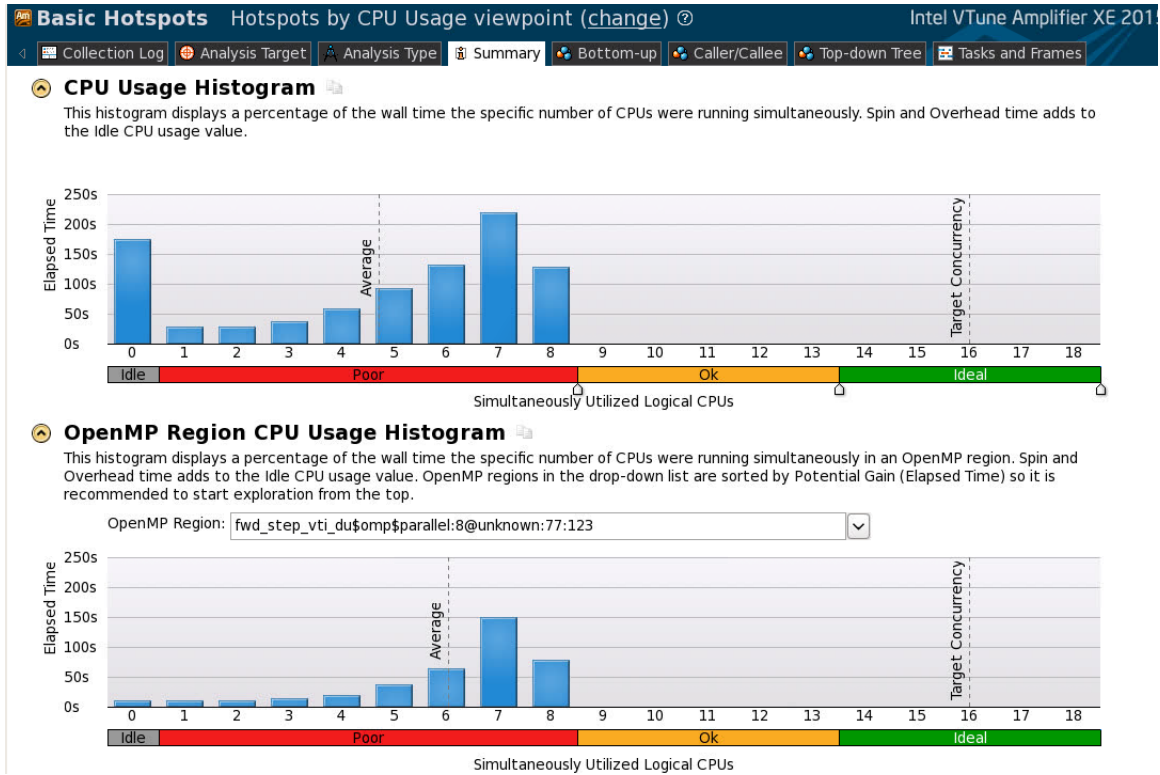


Figure 2.13 : VTune : summary of CPU usage by different threads.

displays hotspots i.e. functions that consume most of the CPU time. The functions are listed in sorted order with respect to execution time.

From Figure 2.12, it is clear that the stencil computation (`fwd_step_vti_du`) and idleness (`_kmp_fork_barrier`) take roughly equal amounts of time (2733s and 2470s respectively). This is a bad scenario since idleness wastes a lot of computational resources. For executions using other configurations, we found that the cost of idleness was commensurate with the amount of effort spent performing stencil computations.

The summary page also includes two histograms of thread utilization as shown in Figure 2.13. The first histogram shows the thread utilization of the entire execution whereas the second histogram shows the thread utilization of OpenMP regions.

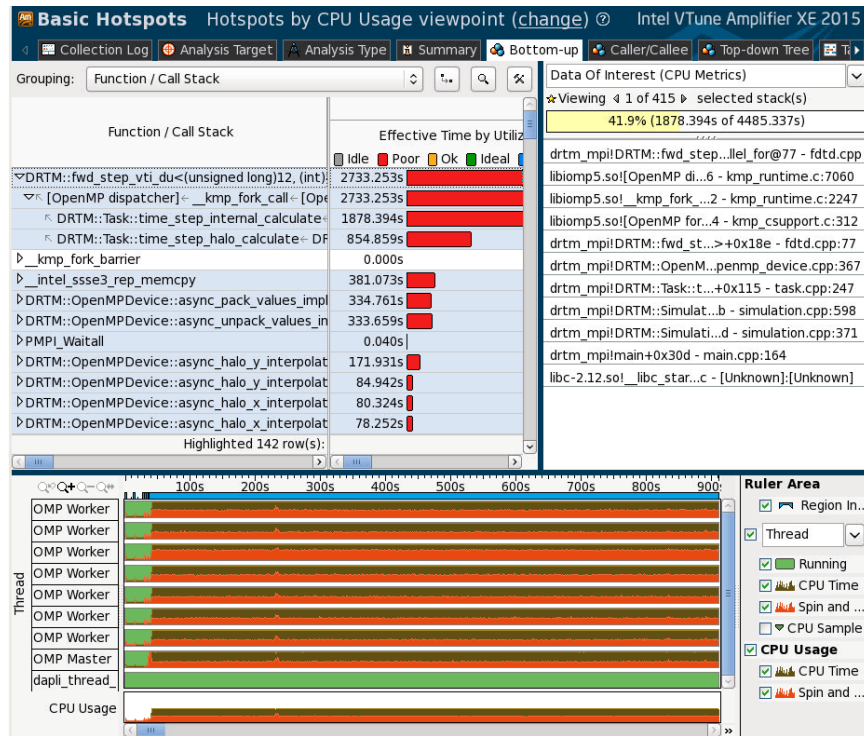


Figure 2.14 : VTune : bottom-up view of execution of DRTM

Figure 2.13 suggests that during overall execution seven threads (out of 8) are getting utilized for a longer time compared to others. But threads remain idle (number of threads running = 0) for a long time too which brings down the average thread utilization count down to 5. It also provides a similar view of OpenMP usage that shows seven threads are utilized most of the time.

VTune provides both bottom-up and top-down views of call-chains that enable detailed analysis. The bottom-up view shows each function with its execution duration and enables one to see the call chain that invoked it as shown in Figure 2.14 top-left pane. VTune provides a timeline view below the bottom-up view. The timeline view only indicates OpenMP or user code. User code cannot be further decomposed and analyzed.

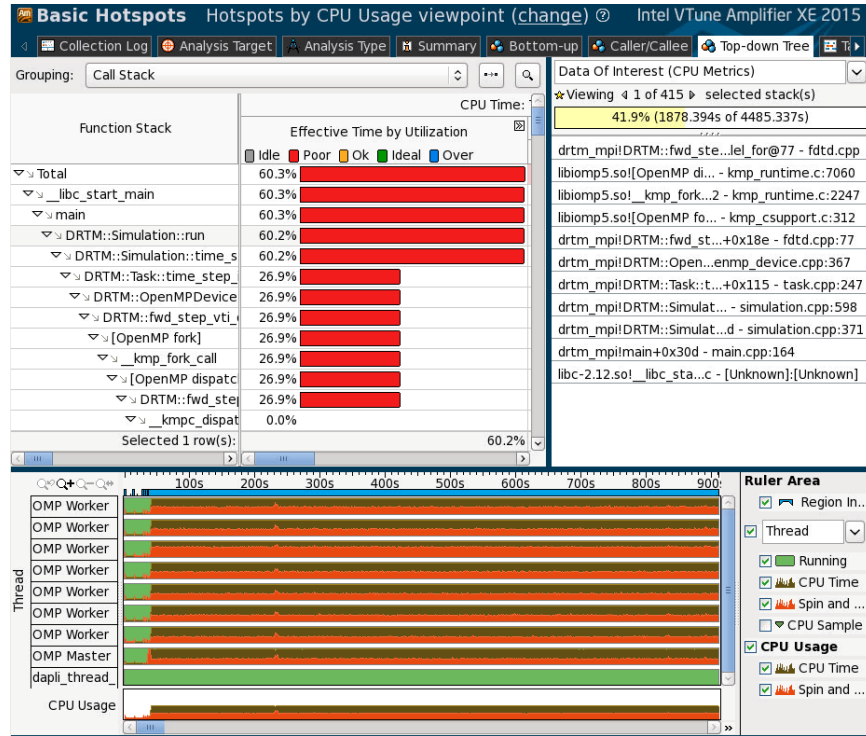


Figure 2.15 : VTune : top-down view of execution of DRTM.

Similar to the bottom-up view, there is a top-down view that starts from main and enables one to down the call chain as shown in Figure 2.15

VTune collects enormous amounts of data compared to other tools. For example, HPCToolkit collects around 2GB of data for the large data set test case when run using 16 processes i.e 125MB per process whereas VTune's hotspot analysis collects around 900MB of data per process for the same test case.

## 2.8 Summary of the Tool Evaluation

Based on our study, we decided what capabilities that would make a tool effective for analysis of hybrid MPI+OpenMP applications. The capabilities that we need from a performance analysis tool are:

- provide insight about the suitability of the decomposition of work across nodes and interprocess communication - I (Interprocess)
- provide insight regarding how threads are utilized within a node - T (Threading)
- provide insight into how different functional units (e.g. cache) are getting utilized within a core - F (Functional units)
- attribute performance data back to source code - A (Attribution)
- generate summary such as hotspots and also a detailed view for manual inspection - D (Diagnostics/Details)

A table showing capabilities of the tools we evaluated is given in Table 2.2. ITAC is used for analyzing MPI communication and, therefore, its main deficiency is the inability to look into threading and functional unit issues. MAP provides insight into MPI usage and functional unit utilization but ignores threading issues. Also, it is not possible to zoom-in to a particular region of interest for detailed analysis. PCM is not suitable for analyzing MPI applications. PerfExpert does not provide insight into MPI communication or threading issues but gives an overview of functional unit utilization. It lacks a timeline view which makes it difficult to see how execution unfolds over time. VTune lacks support for analyzing MPI communication but can be used for detailed analysis of threading and functional unit utilization.



Table 2.2 : Performance analysis tools capability matrix

Tool	I	T	F	A	D *
HPCToolkit	Yes	Yes	Yes	Yes	No/Yes
ITAC	Yes	No	No	No	Yes/Yes
MAP	Yes	No	Yes	Yes	No/No
PCM	No	Yes	Yes	No	No/Yes
PerfExpert	No	No	Yes	No	Yes/No
VTune	No	Yes	Yes	Yes	Yes/Yes

HPCToolkit provides a unified view of interprocess interactions, threading and functional unit utilization details. It also attributes measured metrics back to full calling context and enables a detailed analysis using `hpcviewer` and `hpctraceviewer`. Therefore, we decided to use HPCToolkit for in-depth analysis of DRTM. Insights we gained using HPCToolkit helped us to improve DRTM's performance.

---

\*Represents automatic diagnostics and detailed inspection. For example, No/Yes means automatic diagnostics is not present but the tool allows detailed manual inspection.

## Chapter 3

### Related Work

In this chapter we describe few different directions of prior work that relates to our analysis and optimization of DRTM.

#### 3.1 Parallelizing RTM

There have been several successful efforts to parallelize RTM computation onto multicore and distributed memory systems. Some efforts were to speed up RTM by harnessing the computation power of GPUs. Abdelkhalek *et al.* [27] and Cabezas *et al.* [28] used CUDA to leverage the computational power of GPUs. Qawasmeh *et al.* [29] employed a hybrid model, using OpenACC to program GPUs and MPI to distribute computation across nodes. Araya-Polo *et al.* [4] use OpenMP to parallelize RTM across cores. Lu and Magerlein [30] also employed a hybrid model with MPI to distribute work across nodes. Within a node, they use OpenMP in contrast with Qawasmeh’s work which uses OpenACC.

#### 3.2 Domain-specific Language Frameworks

Researchers have used domain-specific language frameworks to express stencil computations independent of the characteristics of the target machine. Some studies in this direction include SDSL [31], PATUS [32, 33] and FAST [34]. SDSL - Stencil Domain Specific Language can be embedded in C, C++ and MATLAB code. SDSL’s backend

translates the stencil specification written using SDSL to C code that can be further optimized using polyhedral frameworks such as PolyOpt/C [35] and PoCC [36]. The SDSL framework also provides locality and SIMD optimizations. PATUS is an auto-tuning framework for stencil computation targeted at multicore CPUs and GPUs. During code generation of the stencil specification, PATUS allows one to specify a **strategy**: a description of the parallelization and optimization methods to be applied. Strategies are parameterized, and auto-tuning is used to select an optimal parameter configuration with respect to the chosen stencil kernel and hardware platform. FAST is an auto-tuning framework similar to PATUS but is faster. FAST employs machine learning to predict a set of optimal solutions from the space of possible optimization solutions, thereby improving tuning speed. Halide [37, 38] is a domain specific language that allows the programmer to specify algorithm and scheduling decisions separately. This enables evaluation of various scheduling strategies which includes storage decisions, order of execution and optimizations without changing the algorithmic code. The Pochoir [39] stencil compiler allows the programmer to write the stencil specification in a domain specific language embedded in C++. The Pochoir compiler translates the stencil specification to a high-performing parallel CilkPlus [40] code.

### 3.3 Overlapping Communication with Computation

Achieving efficient and effective communication while using hybrid programming models (in general MPI+X where X is a shared memory programming model) is difficult. Most MPI implementations advance communication inside MPI library calls. The semantics of non-blocking communication does not require asynchronous progress and therefore MPI implementations might not support real communication-

computation overlap. Bamboo [41] is a source-to-source translator that translates an MPI C program into a data-driven form that overlaps communication with computation. Bamboo requires the programmer to annotate the original program with additional directives to aid the tool to infer matching sends and receives. Buettner *et al.* [42] tried to address the issue of communication-computation overlap by extending the OpenMP runtime to include communication tasks. These tasks are executed when a given condition becomes true, such as receiving an MPI message. HCMPI (Habanero-C MPI) [43], integrates Habanero-C dynamic task-parallel programming model with the MPI message-passing interface. In this model, all MPI calls are treated as asynchronous tasks which are handled by a dedicated communication worker thread. Vaidyanathan *et al.* [44] tried to address the issue of overlap with an MPI offload infrastructure using a dedicated communication thread. In this approach, MPI calls get enqueued as communication tasks which gets processed by the dedicated communication thread. This approach uses the Linux LD\_PRELOAD feature to dynamically inject their special MPI library between the application and a traditional MPI. This enables them to change the program’s communication library without any modification of the application code.

## Chapter 4

### Analysis and Tuning of DRTM using HPCToolkit

In this chapter we describe details about our analysis of DRTM using HPCToolkit and how insights from HPCToolkit helped us to improve DRTM’s performance. We describe a series of code optimization opportunities identified using HPCToolkit and the steps we took to improve performance based on the findings. We present the performance improvement associated with each optimization performed. We conclude the chapter with an assessment of the optimized version of DRTM.

#### 4.1 Initial Assessment

Figure 4.1 shows a screenshot of HPCToolkit’s `hpcviewer` code-centric user interface for the execution of DRTM. The view is sorted based on the total time used by each function. Figure 4.1 shows the top five time-consuming functions. `hpcviewer` shows that only 37.8% of total execution time is spent on the stencil computation (`fwd_step_vti_du`). Idleness in the OpenMP runtime (`omp_idle`) consumes 32.7% of the execution time. Another observation is that `memcpy` takes 7.4% of execution time. Ideally the stencil computation should fill the execution time and idleness should be minimal. We need to investigate why `memcpy` is taking so much of execution time. Our initial analysis with `hpcviewer` suggests three issues to investigate.

1. Why is around 7% of the execution time is used by `memcpy`
2. Why is around 33% of the execution time is wasted in wait-sleep

Scope	REALTIME (usec):Sum (I)	▼ REALTIME (usec):Sum (E)
Experiment Aggregate Metrics	1.20e+11 100 %	1.20e+11 100 %
▸ void DRTM::fwd_step_yti_du<12ul, 8, true>	4.54e+10 37.8%	4.53e+10 37.7%
omp_idle()	3.92e+10 32.7%	3.92e+10 32.7%
▸ __intel_sse3_rep_memcpy	8.92e+09 7.4%	8.92e+09 7.4%
▸ DRTM::OpenMPDevice::async_pack_values_	8.68e+09 7.2%	8.42e+09 7.0%
▸ DRTM::OpenMPDevice::async_unpack_value	7.11e+09 5.9%	6.79e+09 5.7%

Figure 4.1 : `hpcviewer` visualization of execution of the unoptimized version of DRTM.

### 3. Why is only around 38% of execution time is used by stencil computation

Figure 4.2 shows a screenshot of HPCToolkit’s `hpctraceviewer` time-centric user interface for the execution of DRTM’s forward solve computation. `hpctraceviewer` visualizations help provide a deeper understanding of the behavior of DRTM’s behavior. At this granularity it is not possible to make any conclusions. Hence, we zoom-in around the crosshair to get a detailed view as shown in Figure 4.3.

In Figure 4.3, the pink color represents the stencil computation and light brown represent idleness. In the figure, idleness and stencil computation occupy an almost equal amount of space (36.5% and 31.5%) implying that they consume almost equal amounts of execution time. This division of execution time is in alignment with the results reported by `hpcviewer`. Idleness is not distributed equally across processes (each horizontal bar represents an execution trace of a process over time) that imply load imbalance in domain decomposition. The next section discusses the structure of a single time step in detail.

#### 4.1.1 Structure of the Application

As mentioned in Section 1.1, DRTM applies a higher order stencil on a 3-dimensional data to solve a PDE. We use HPCToolkit’s `hpctraceviewer` to examine a time-

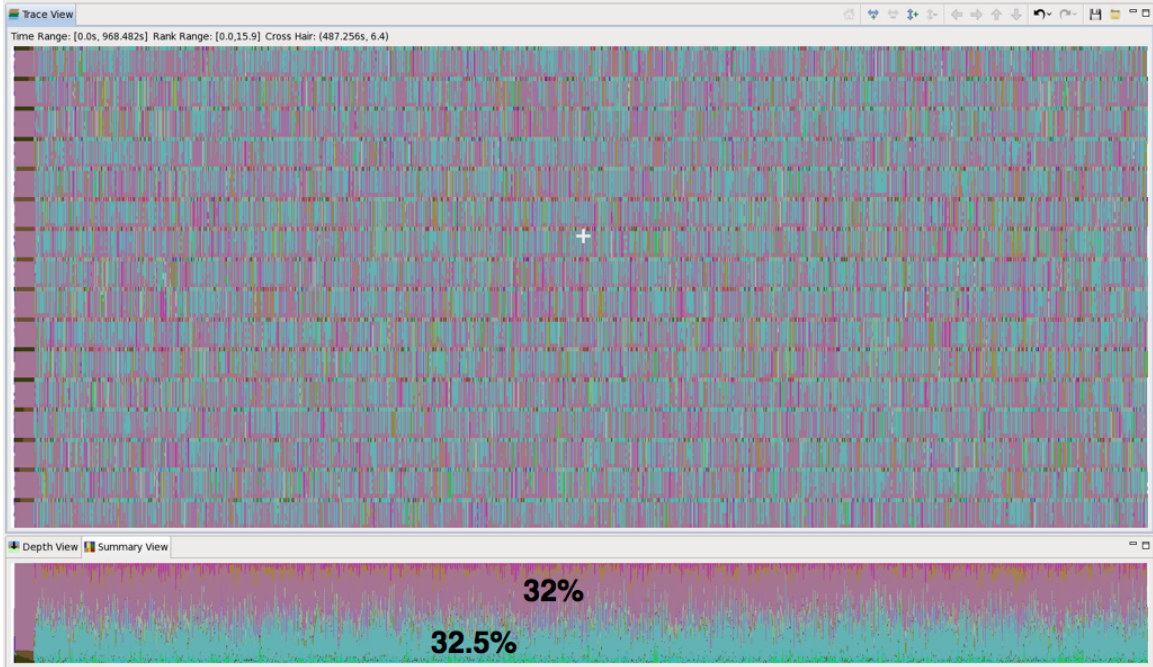


Figure 4.2 : `hpctraceviewer` visualization of the entire execution - rose represents idleness (32%) and green represents the stencil computation (32.5 %).

centric view of a computation. Figure 4.4 \* shows a view of execution on a single MPI rank of a representative time step from the forward-solve phase of DRTM when using OpenMP with `OMP_NUM_THREADS=8`.

The first step performs a stencil computation on data in the halo regions (stencil-halo). Then data points in halo regions are packed into messages (pack) and later exchanged between neighbors (send). DRTM uses nonblocking MPI calls for point-to-point communication with the aim of overlapping communication with computation. Next, the code performs stencil computation for non-halo regions (stencil-internal) while halo regions are being exchanged. After the stencil computation finishes, each MPI process waits for the arrival of halo data (wait, receive) that was initiated prior

---

\*Execution includes other activities such as reading data from input files and hence given values does not add to 100%.

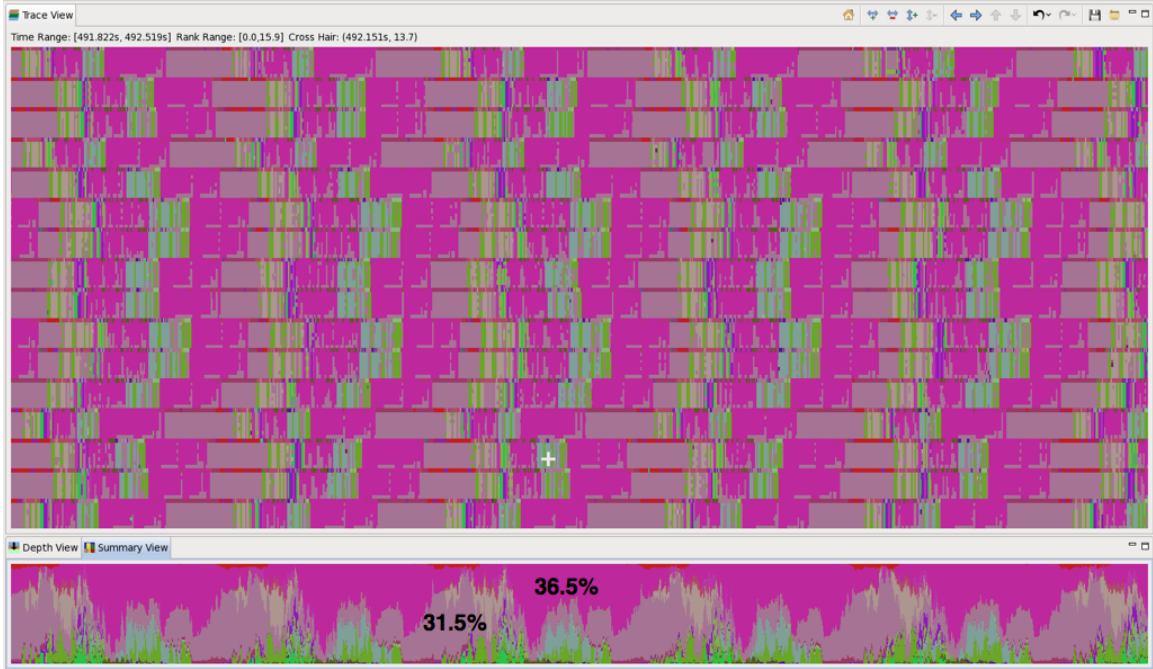


Figure 4.3 : `hpctraceviewer` visualization of a sequence of few timesteps - magenta represents the stencil computation (36.5%) and light brown represents idleness (32.5%).

to the stencil computation. Finally, the code unpacks data from the halo exchanges (unpack) and performs interpolation where necessary. The same cycle of operations is repeated for each `time_step`. The different phases observed using HPCToolkit (shown in Figure 4.4) matches with the design described in Figure 1.3.

## 4.2 Performance Analysis and Code Optimization

In this section, we describe the iterative tuning process used for optimizing DRTM. In each subsection, we identify a potential improvement opportunity and describe actions taken to exploit the opportunity along with performance gain achieved.



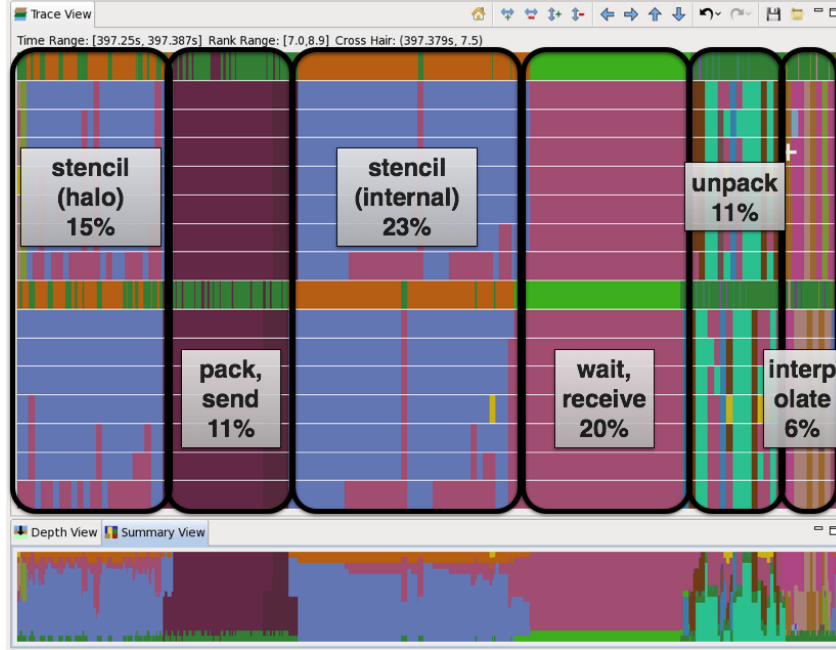


Figure 4.4 : `hpctraceviewer` showing the division of a single time step for two processes each containing eight threads.

#### 4.2.1 Reduce Overhead due to Abstractions

We first investigate the reason for the 7% overhead due to memory copies as identified in Section 4.1. DRTM employs an abstraction layer to support parallelization using different programming models including OpenMP and CUDA. To accommodate accelerator programming models, which currently require copying data into a different memory space, the abstraction layer copied data even when using OpenMP 3.1, which does not require a copy because it executes in the same memory space. Hence, the copying of data from the accelerator to host and vice-versa can be removed in case of OpenMP 3.1. While removing unnecessary copying of data is important, it is also important to keep the abstraction layer intact. That means after removing the additional memory copies, all other node-level programming models employed in DRTM should continue to work.

Listing 4.1: memcpy interfaces

```

2      // existing interface
3      void memcpy(float *dst, float *src,...);

5      //new interface
6      void memcpy(float * & dst, float *src,..., CopyType ctype);

```

One way to address this issue is to copy the pointers instead of data which we refer as **SHALLOW** copy. Doing a **SHALLOW** copy changes the location to where the destination points. Essentially both source and destination point to the same location after the invocation of **SHALLOW** copy. Any change made to destination buffer now affects the source buffer which is not the case with a **DEEP** copy. This side-effect does not affect correctness if the source buffer is not used after **SHALLOW** copy is invoked. In case of DRTM, once the copying is finished, subsequent statements use only the destination buffer. Therefore using **SHALLOW** copy does not affect the correctness of DRTM.

The abstraction layer in DRTM provides an interface named `memcpy` to handle data copying. All shared memory programming models that are plugged into DRTM should provide an implementation of this interface. As shown in Listing 4.1, we added a new interface with the same name but containing an additional parameter. The extra parameter specifies whether data needs to be copied or can be used in place. This new parameter of type `CopyType` takes two values: **DEEP**, and **SHALLOW**.

- **DEEP**: Data is copied from source to target. By default, this mode is used.
- **SHALLOW**: Pointer to data is copied from source to target.

Listing 4.2: memcpy implementation for OpenMP

```

2      //memcpy implementation with shallow and deep copy
3      void memcpy(float * & dst, float *src,..., CopyType ctype) {
4          if (ctype == SHALLOW)
5              dst = src;
6          else
7              //existing memcpy implementation without CopyType parameter
8              memcpy(dst, src,...);
9      }

```

The new interface passes a destination array by reference to reflect changes made inside function call back to its caller. The existing `memcpy` (without `CopyType` parameter) is retained for backward compatibility. Retaining the existing `memcpy` interface reduces the number of modifications required since only those invocations where a shallow copy is possible for some node-level programming models need to be changed. DEEP copy is same as the existing implementation and invokes the existing API. When a SHALLOW copy is specified only the pointer to data gets copied. By doing the pointer copy, rest of the code where the data is used does not need any modification. The OpenMP implementation of new `memcpy` interface is shown in Listing 4.2

Other programming model implementations also need to implement the new interface but might implement different handling for the SHALLOW parameter. For example, a CUDA implementation should ignore the SHALLOW parameter as shown in Listing 4.3 since today's GPUs work in a different memory space.

Replacing unnecessary DEEP with SHALLOW memory copies reduced the execution time by roughly 12% (from 870s to 760s) for the large data set test case. Although

Listing 4.3: memcpy implementation for CUDA

```

2      //memcpy implementation employs DEEP copy regardless of CopyType
3      void memcpy(float * & dst, float *src,..., CopyType ctype) {
4          memcpy(dst, src,...);
5      }

```

Scope	REALTIME (usec):Sum (I)	▼ REALTIME (usec):Sum (E)
Experiment Aggregate Metrics	1.07e+11 100 %	1.07e+11 100 %
▷ void DRTM::fwd_step_vti_du<12ul, 8, true>(float const	4.53e+10 42.2%	4.52e+10 42.1%
omp_idle()	3.62e+10 33.7%	3.62e+10 33.7%
▷ DRTM::OpenMPDevice::async_pack_values_impl(float*	8.59e+09 8.0%	8.33e+09 7.8%
▷ DRTM::OpenMPDevice::async_unpack_values_impl(floa	6.78e+09 6.3%	6.45e+09 6.0%
▷ DRTM::OpenMPDevice::async_halo_y_interpolate_impl	4.03e+09 3.8%	4.00e+09 3.7%
▷ DRTM::OpenMPDevice::async_halo_x_interpolate_impl	2.67e+09 2.5%	2.64e+09 2.5%

Figure 4.5 : `hpcviewer` output after the introduction of `SHALLOW` copy. `memcpy` has disappeared from the list of top time consuming functions compared to Figure 4.1

HPCToolkit reported that `memcpy` took just 7% of the total execution time, the performance improvement is 12%. DRTM implements multithreaded `memcpy`s using OpenMP `parallel` construct. This involves overhead for forking and joining worker threads. The additional 5% improvement comes through the reduction of overheads caused by the OpenMP runtime associated with the invocation of multithreaded `memcpy`s. The resulting `hpcviewer` output after the introduction of `SHALLOW` copy is given in Figure 4.5.

Compared to Figure 4.1, `memcpy` has disappeared from the list of functions that occupy top spots on the list. The entry for `memcpy` further down the list shows that after replacing `DEEP` copies of the halo region data to the OpenMP device with `SHALLOW` copies, the cost of `memcpy` is now less than one percent of DRTM's

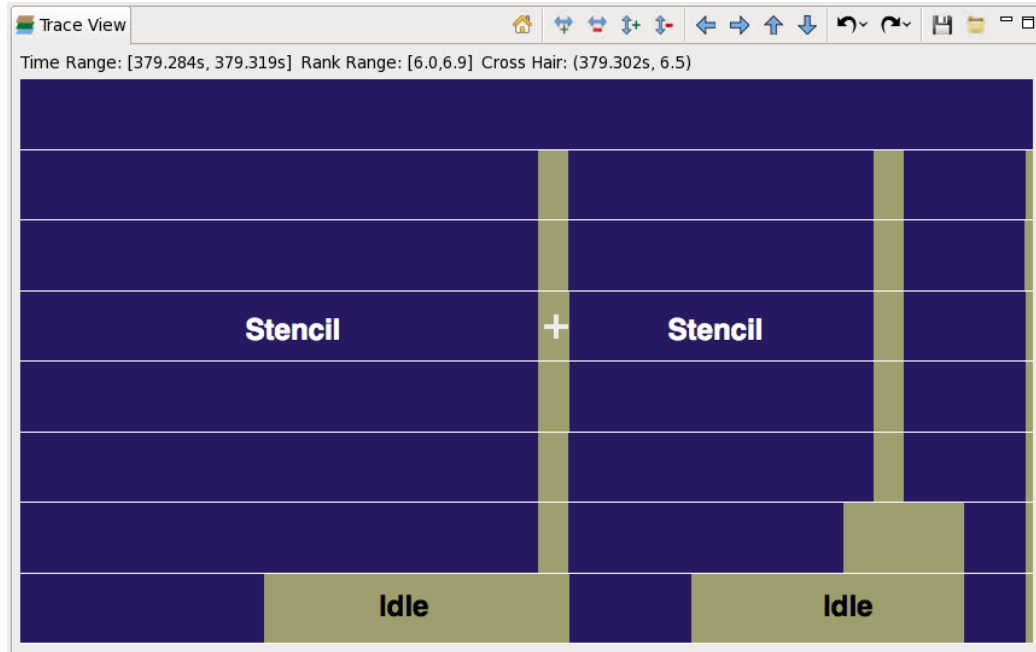


Figure 4.6 : A screenshot of `hpctraceviewer` showing idleness occurring during stencil computation for an MPI process with eight threads

#### 4.2.2 Reduce Thread Level Load Imbalance

As shown in Figure 4.6, `hpctraceviewer`'s visualization of the stencil computation by one process for an iteration of the forward solve phase of DRTM reveals unnecessary idleness. In Figure 4.6, blue portion represents stencil computation and light brown idleness. This view implies that some threads are idle during stencil calculation indicating an imbalance in the work assigned to threads.

DRTM's stencil computation loop is tiled to improve cache reuse and a pair of loops over the Y and Z dimensions are collapsed into a single loop using an OpenMP `collapse` clause. Iterations of the loop over the tiles are executed in parallel using a static schedule. In our investigation of the imbalance, we determined that the decomposition of work into tiles is as illustrated in Figure 4.7.

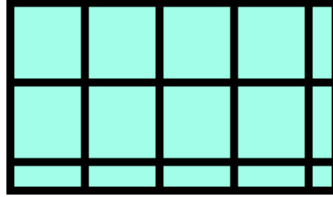


Figure 4.7 : Smaller tiles at the high end of each dimension cause imbalance when OpenMP **static** scheduling is used for a collapsed 2D loop nest over the tiles.

An OpenMP runtime partitions iterations in the collapsed loop with no knowledge of tile sizes. When static scheduling is used, each block is considered as a point by OpenMP runtime, and there is no distinction between smaller and larger tiles. Using **static** scheduling assigns an equal number of points (tiles) to each thread. The difference in tile sizes causes threads assigned a collection of small tiles to finish early and remain idle until other threads complete their stencil computation. Changing the OpenMP scheduling strategy for the stencil computation loop to **dynamic** reduced the imbalance. **hpctraceviewer** output after changing to **dynamic** scheduling is shown in Figure 4.8. Compared to Figure 4.6, the entire execution is almost filled with stencil computation (green) and idleness has nearly disappeared. Removal of waits reduces the execution time by roughly 5% (from 760s to 725s) for the large data set test case.

### 4.2.3 Overlap Communication with Computation

A **hpctraceviewer** visualization after reducing the load imbalance among threads is shown in Figure 4.9. In the figure, pink represents idleness and green stencil computation. There is a considerable amount of waiting after the stencil computation.

Even though MPI communication is performed with non-blocking sends initiated before and completed after the stencil computation, processes stall after the stencil computation waiting for messages from neighbors to arrive. This wait caused

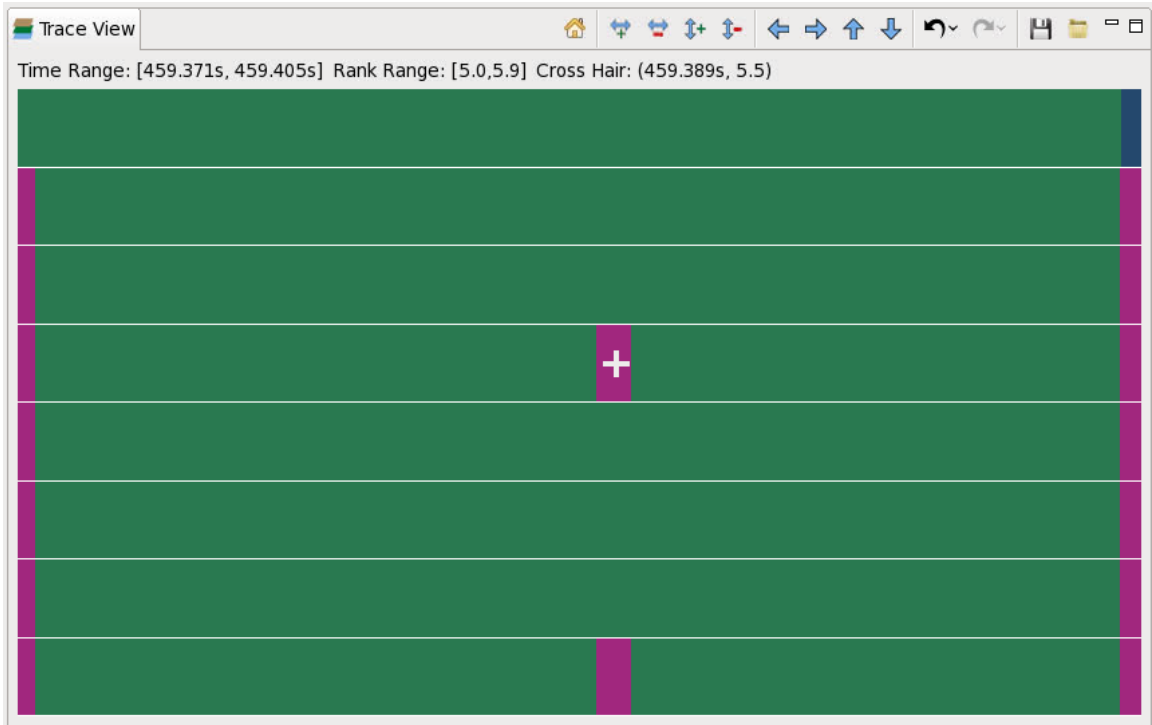


Figure 4.8 : Idleness within stencil computation removed using dynamic scheduling

us to question whether the communication was making progress during the stencil computation. To ensure true overlap we first tried using the asynchronous progress engine provided by Intel's MPI library. Since this approach gave no performance improvement, we dedicated one OpenMP thread to handle MPI communication. In the following subsections, we describe the reasons for performance degradation caused by the asynchronous mode in MPI library followed by a description of how we dedicated a thread to managing communication.

### Problems with the Asynchronous Progress Mode

Intel MPI version 5 introduced asynchronous progress mode which dedicates a separate thread for communication. The asynchronous progress thread is enabled by

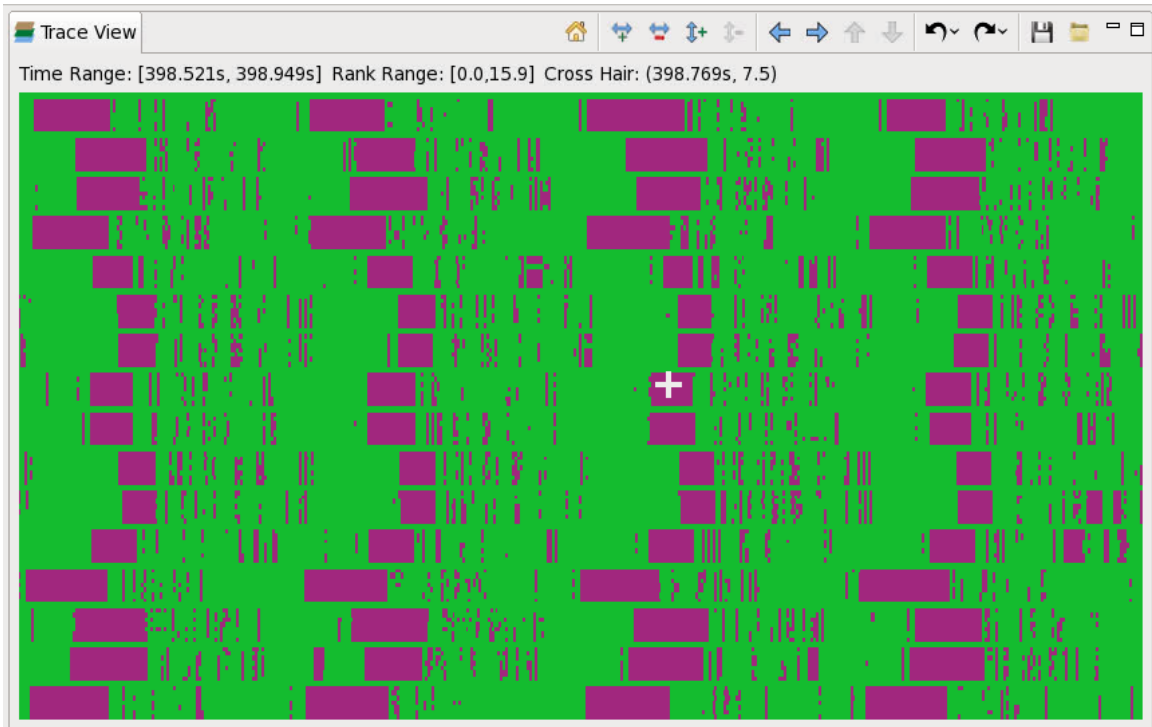


Figure 4.9 : A screenshot of `hpctraceviewer` shows considerable waiting (pink) occurs after stencil computation (green) implying lack of communication-computation overlap

setting the `MPICH_ASYNC_PROGRESS` environment variable to 1. The experiment using asynchronous progress mode was performed with two different configurations:

- Create OpenMP threads equal to number of cores
- Create one less OpenMP thread than the number of cores so that a dedicated core is available for the MPI progress thread.

Using the Intel MPI library's asynchronous progress thread in either of these configurations increased execution time by 25-30%. When asynchronous mode is enabled the thread safe MPI library is used which introduces additional overhead. Figure 4.10 gives an `hpctraceviewer` visualization of a DRTM execution using one



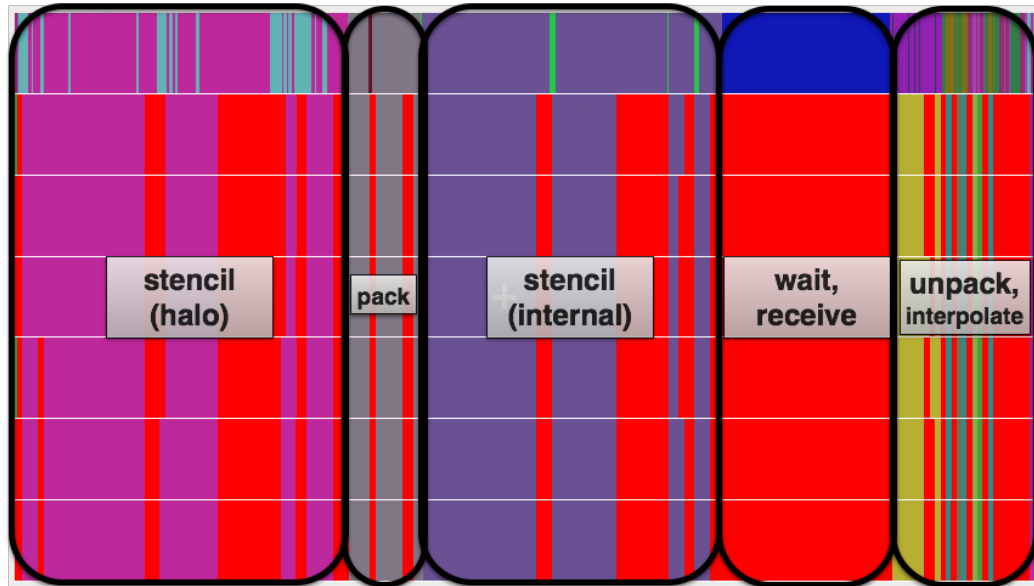


Figure 4.10 : Idleness (red) occurs during the stencil computation phase (performed separately for each halo region and each slab along the Z axis) and other operations after introducing asynchronous progress thread.

less OpenMP thread with asynchronous progress mode enabled. From Figure 4.10, it can be seen that using asynchronous progress mode introduces additional idleness within the series of stencil computations and other operations. The additional idleness introduced by the OpenMP runtime increases the total execution time.

### Enable Overlap Using a Communication Thread

Using the asynchronous progress engine provided by Intel's MPI library did not improve performance. Another method to ensure overlap is to dedicate an OpenMP thread explicitly for communication i.e., introduce a communication thread [41, 42]. Listing 4.4 shows a part of the DRTM code organization intended to overlap communication with computation using non-blocking MPI communication primitives. In this code, sends and recvs are posted asynchronously before the stencil computation

Listing 4.4: Original code intended to overlap communication with computation using non-blocking MPI primitives

```

2   MPI_Irecv(); //post recv requests
3   MPI_Isend(); //post send requests
4   stencil_computation(); //post stencil computation
5   wait_all(); //wait for sends and revcs to complete
6   unpack();

```

and awaited for completion after the computation.

In an attempt to improve communication/computation overlap, we dedicated one thread (thread 0) to perform communication using OpenMP `parallel` construct as shown Listing 4.5. The other thread (thread 1) starts with the stencil computation and later forks into multiple threads using nested threading. In the listing, the communication thread executes `wait_all` (line 8) and the computation thread proceeds with the stencil computation (line 6). This is similar to the “communication worker” model in HCMPI [43] and MPI offload thread used by Vaidyanathan *et al.* [44].

Since one thread is allocated for communication, stencil computation is performed with once less thread, i.e., number of threads used for stencil computation = number of cores - 1. Thus, the communication thread gets a core for itself, which enables better overlap of computation and communication.

Figure 4.11 shows a `hpctraceviewer` visualization of several iterations in DRTM’s execution after dedicating an OpenMP thread to progress communication. Green bars represent waiting after the stencil computation for data from the halo exchange to arrive from neighbors. Compared with Figure 4.9, processes in the middle section

Listing 4.5: Modified code that overlaps computation and communication using a explicit communication thread

```

1  MPI_Irecv(); //post recv requests
2  MPI_Isend(); //post send requests
3  #pragma omp parallel num_threads(2)
4  {
5      if(thread_id == 1)
6          stencil_computation(); //stencil computation using nested parallelism
7      else if(thread_id == 0)
8          wait_all(); //wait for sends and revcs to complete
9  }
10 unpack();

```

(along Y-axis) do not incur waiting. The difference in idleness between the edge and internal processes is an indication of load imbalance in domain decomposition. Another conclusion from the analysis of Figure 4.11 is that there is enough bandwidth for data exchange since nodes processing middle partitions (partitions with more neighbors) do not incur any waiting after the stencil computation. To further reduce idleness we need to improve load balance between nodes. Introducing a dedicated OpenMP communication thread reduces execution time by roughly 7.5% (from 725s to 670s) for the large data set test case.

#### 4.2.4 Improving Data Reuse in Cache

After adjusting the domain decomposition and threaded parallelization strategies, we next checked how functional units are being utilized. We used hardware performance counters to assess functional unit utilization. We added code to measure various characteristics of the stencil code's execution using calls to routines in the PAPI

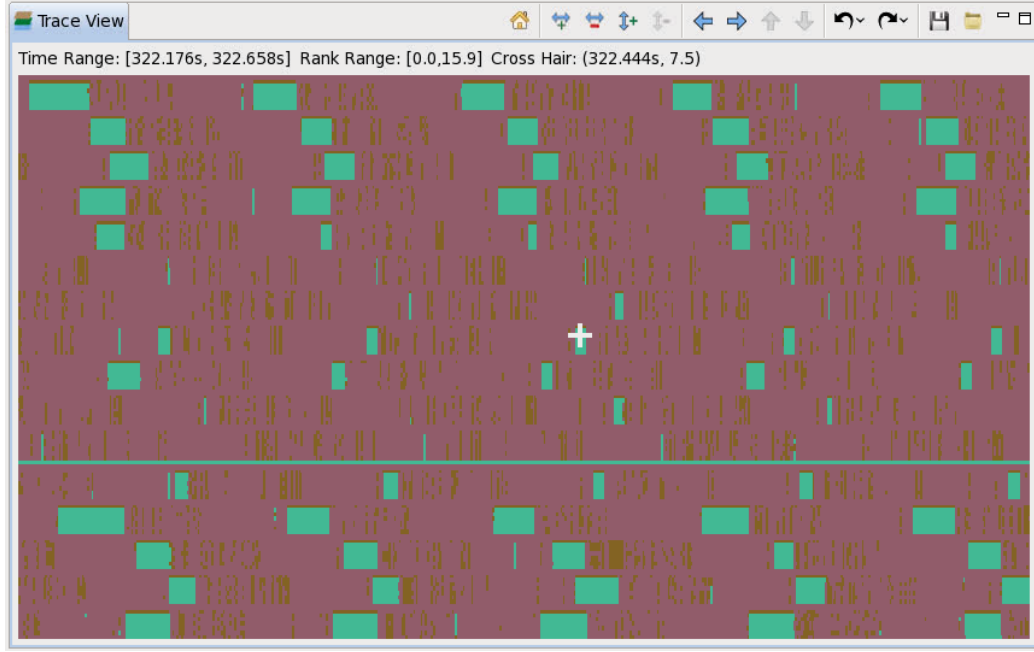


Figure 4.11 : Waiting (green) after the stencil computation is reduced by the introducing a separate communication thread.

library [25]. PAPI provides a set of consistent interfaces that can be used across different architectures to use hardware performance counters to measure program execution characteristics. A few of the counter values that we used to analyze the performance of the stencil computation in DRTM is shown in Table 4.1.

$$L1 \text{ cache miss rate} = \frac{L1 \text{ data cache miss}}{Load \text{ instructions} + store \text{ instructions}} = \frac{3.37E11}{2.62E12+7.64E10} = .125$$

$$L2 \text{ cache miss rate} = \frac{L2 \text{ data cache miss}}{L2 \text{ data cache access}} = \frac{1.17E11}{3.37E11} = .35$$

The percent of L1 and L2 cache miss is roughly 12.5% and 35% respectively. This implies one third of accesses to L2 cache are a miss, which indicates a potential optimization opportunity to reduce cache misses. We analyzed the loop nesting structure of the stencil computation to get insight into its cache performance. The stencil computation loop is tiled to improve cache reuse. Each tile iterates over the 3D data in Y-Z-X order, where X is the innermost loop. The stencil computation for one point

Table 4.1 : Hardware performance counter values for the stencil computation loop

Counter name	Value
Total cycles	4.58E+12
Floating point operations	2.64E+11
Load instructions	2.62E+12
Store instructions	7.64E+10
L1 data cache miss	3.37E+11
L2 data cache access	3.37E+11
L2 data cache miss	1.17E+11

uses 49 points (12 along each +X, -X, +Y and -Y axis) in the X-Y plane but only 41 points (12 along each +X, -X and 8 along +Z, -Z) in the X-Z plane. Assume there are 100 points along X axis. For a complete iteration of the inner most loop along X axis, roughly  $100 \cdot (24 + 16)^{\dagger}$  points need to be loaded from memory. In case, at the next outer loop level if Z axis is used, then roughly we need to load an additional  $100 \cdot 24$  points to complete iteration through X axis. But if Y axis is selected as next outer loop instead of Z axis, roughly  $100 \cdot 16$  new points needs to be loaded which implies fewer new points are loaded and therefore more points are reused. Our expectation was that interchanging Y and Z loops to access data in Z-Y-X order would increase data reuse in cache. The number of cache misses before and after loop interchange is given in Table 4.2. Interchanging the Y and Z loops reduced the cache misses by increasing reuse and improved the performance by roughly 5% (670s to 640s) for the large data set test case.

---

<sup>†</sup>Boundary points not taken into consideration for these calculations.

Table 4.2 : Reduction in number of cache accesses after loop interchange

Counter name	Old value (miss rate %)	New value (miss rate %)	Reduction
L1 data cache miss	3.367E+11 (12.5%)	3.194E+11 (11.8%)	5%
L2 data cache miss	1.171E+11 (35%)	6.671E+10 (21%)	43%

#### 4.2.5 Reduce Process Level Load Imbalance

Figure 4.12 shows a screenshot of `hpctraceviewer` visualization of the trace after performing all the above described code optimizations. In Figure 4.13, MPI ranks 0,3,12,15 are the corner processes, 1,2,4,7,8,11,13,14 are the edge process and 5,6,9,10 are the middle processes. It is clear that corner and edge processes have more idle time waiting for data to arrive from neighbors than middle nodes. The reason for the difference in idleness is the imbalance in work caused by the difference in the number of neighbors for each MPI process. As described in Figure 4.13, the middle nodes have more neighbors, and, therefore, each middle node does more work because it needs to perform halo calculation, pack, unpack and interpolate for all the neighbors. Due to the reduced amount of work, corner and edge nodes finishes early and stay idle.

DRTM already had a provision to change the size of the partitions. We realized that by changing the size of the partitions, the effect of difference in the number of neighbors can be reduced. Reducing the size of the partition for the interior nodes compensates for the higher number of neighbors as shown in Figure 4.14. Variable size partitioning improved performance by roughly 2.5% (640s to 625s) for the large data set test case with 4x4 configuration.

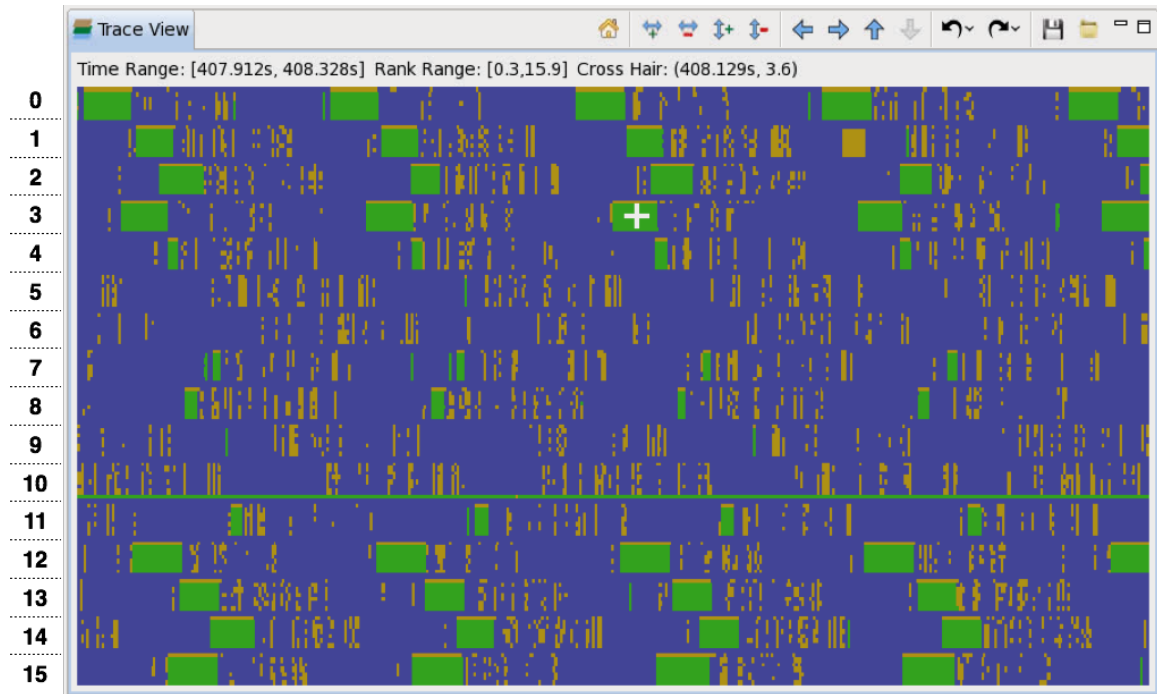


Figure 4.12 : Idleness (green) after the stencil computation is more for processes at the top and bottom. MPI ranks are marked on the left side.

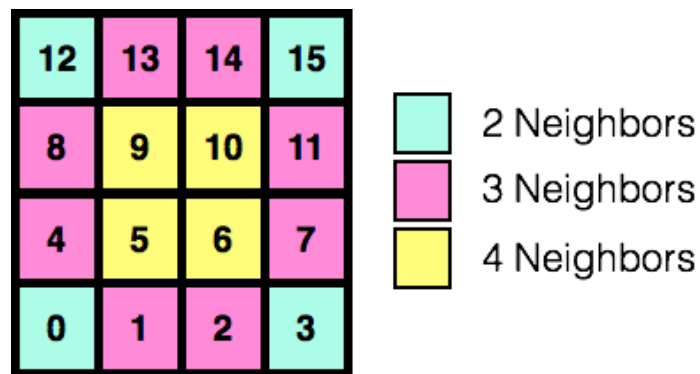


Figure 4.13 : 4x4 configuration (4 in X and 4 in Y direction) with each process marked with its MPI rank showing the difference in the number of neighbors for each block in the 2D partition of the domain across MPI ranks. For example, top-left process (rank 12) has 2 neighbors whereas the middle four processes (5,6,9,10) have 4 neighbors.

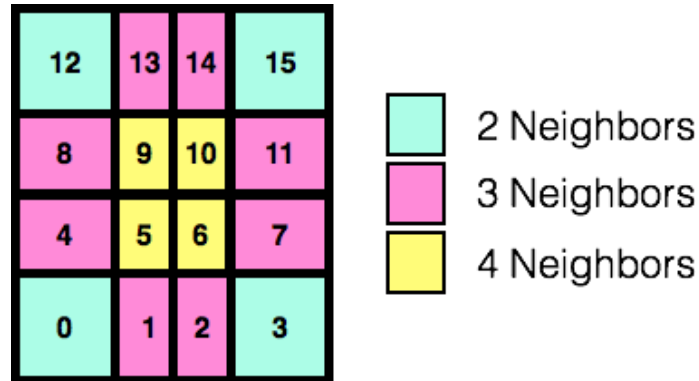


Figure 4.14 : 4x4 configuration (4 in X and 4 in Y direction) with each process marked with its MPI rank showing partitions with reduced sizes for MPI ranks managing the domain interior to reduce load imbalance.

#### 4.2.6 Eager Operations on Received Data

DRTM uses `wait_all` to wait for communication to finish so that data can be unpacked. However, unpacking can be initiated as soon as data is received; there is no need to wait for data from all neighbors to arrive. Using `wait_any` instead of `wait_all` enables eager processing of data as soon as it is received. To use `wait_any`, additional bookkeeping is needed to correlate neighbors and the positions of their messages within a `wait_any` message array.

Figure 4.15 shows a screenshot of `hpctraceviewer` visualization after the introduction of eager unpacking. Green bars represent waiting after the stencil computation for data to arrive from neighbors. Cuts in the green bars represent unpacking of data that are received early. It can be seen that even though eager processing of received data is enabled, few data exchanges finish early. Therefore, waits happen for processes towards the edges similar to the one without eager unpacking (Figure 4.11). Hence, no noticeable performance improvement is achieved by the introduction of eager processing of received data.



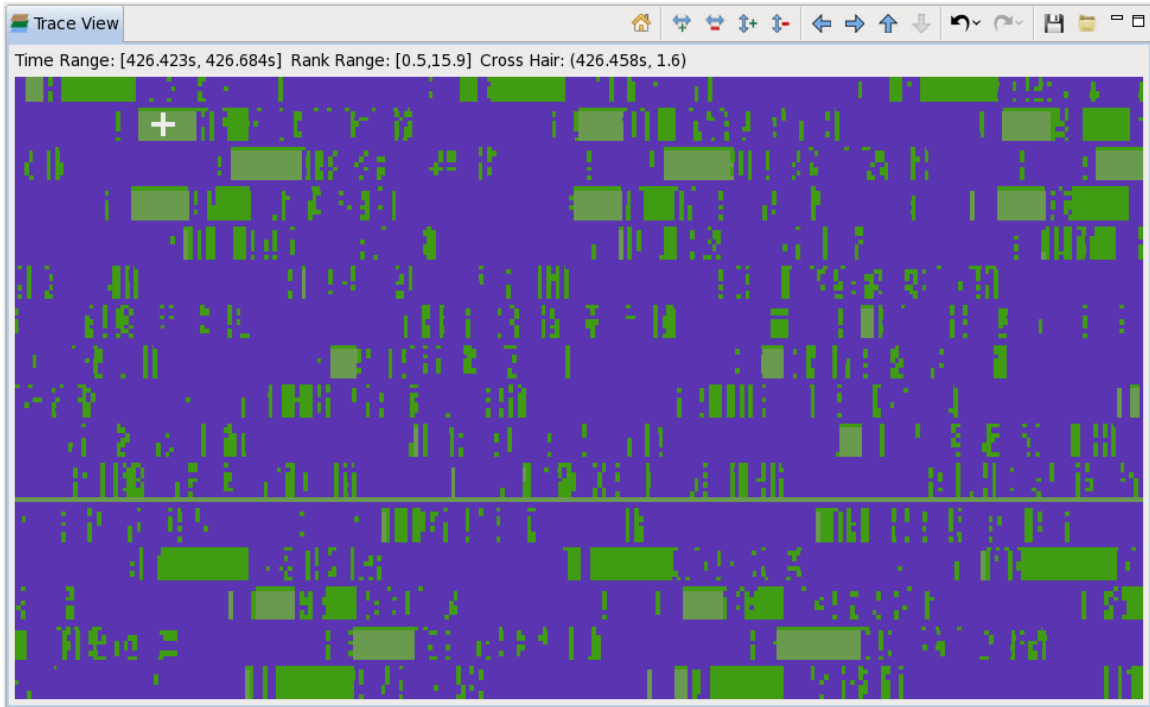


Figure 4.15 : Processes towards edges wait (green) after stencil computation for halo exchange data to arrive even after `wait_any` is used.

#### 4.2.7 Vectorizing Stencil Computation

The Intel Sandy Bridge family of processors supports Intel Advanced Vector Extensions (AVX) [45]. AVX is a set of instructions for Single Instruction Multiple Data (SIMD) operations. AVX registers are 256 bits wide and, therefore, capable of performing eight single precision floating point operations with a single instruction. One way to specify SIMD operations is to use CilkPlus's [40] SIMD extensions provided by Intel's C/C++ compiler. Listing 4.6 shows an example of a normal loop and its equivalent using CilkPlus SIMD extensions.

DRTM uses an array of structures, which prevents conversion of its stencil computations into CilkPlus notation. There have been successful efforts to solve the problem of choosing data layouts (such as an array of structures or multiple arrays) to improve

Listing 4.6: Loop to add two arrays and its CilkPlus equivalent

```

2      //Normal for loop to add two arrays
3      for(int i=0; i<N; i++)
4          a[i] = a[i] + b[i];

6      //Equivalent CilkPlus SIMD representation
7      c[0:N] = a[0:N] + b[0:N];

```

Listing 4.7: Converting array of struct to multiple arrays to aid vectorization

```

2      //Original array of struct
3      struct float2{
4          float x, y;
5      };
6      float2 arr[N];

8      //Equivalent array only representation
9      float x_arr[N], y_arr[N];

```

performance. Keasler *et al.* [46] introduced TALC, an extension of C++ to avoid the rewriting of code in different layouts. Sharma *et al.* [47] extended TALC framework with an algorithm to recommend a good layout for a given source program and target machine. To support better vectorization of DRTM's stencil computation, we want to convert the array of structures to multiple arrays - one array for each element in the structure as shown in Listing 4.7. This data transformation was necessary to enable conversion of the stencil code to CilkPlus SIMD notation.

Conversion to CilkPlus notation improved the overall performance of DRTM while using our smaller data set input by around 10%. However, the performance of the stencil using the large data set input deteriorated by around 7%. Analysis using hardware performance counters showed that the decline in performance is due to an increase in L1 cache misses. An important observation during vectorization is that starting address of some arrays used in the stencil computation are not 32-byte aligned. Using aligned arrays may improve performance with vector extensions.

To verify our claim that aligning array accesses will improve performance, we extracted the stencil kernel along with the OpenMP annotations and ran it on a representative data set, an array of size  $((200+24+\text{padding})*(200+24)*(200+16))$ . This includes halo regions of size 24 along X and Y directions and 16 along Z. Additionally, along the X axis we can add padding, which can contain space for up to thirteen floats, for use in alignment experiments. We ran experiments using three configurations.

1. Imitate the original stencil computation configurations used in DRTM. In this configuration, the code is written as a loop applying an element-wise stencil. Two out of nine arrays are not aligned to a 32 byte boundary. To understand the full impact of alignment on vectorization, we also tried starting the stencil calculation at an odd index along a row by setting `indx` variable in line 8 in Listing 4.8 to an odd number.
2. Perform the element-wise stencil calculation described above, but with all arrays aligned and padded so that rows are an integral multiple of vector length.
3. Rewrite the stencil calculation using CilkPlus array notation to express the inner loop. Perform the computation a configuration where all arrays are aligned and padded so that rows are an integral multiple of vector length.

Listing 4.8: DRTM kernel in scalar notation

```

1      #pragma omp parallel for collapse(2), schedule(dynamic)
2      for(int biz = 0; biz < iz2; biz += BZ) {
3          for(int biy = 0; biy < iy2; biy += BY) {
4              for (int iz = biz; iz < (iz2 < biz+BZ ? iz2 : biz + BZ); ++iz) {
5                  for (int iy = biy; iy < (iy2 < biy+BY?iy2 : biy + BY); ++iy) {
6                      #pragma ivdep
7                          for (int ix = 0; ix < ix2; ix++){
8                              const size_t indx = box_offset + ix + f(iy, iz);
9                              const size_t offsetpv = indx-const1, offset = f1(iz, iz1);
10                             const float tmp = vel[ indx ] *
11                                 mult_add_stride(fdzw, pv, offset, offsetpv, nxny);
12                             const float tmp2 = stencil_2D(fdxyw, ph, nx, indx);
13                             ...
14                         }
15     static inline float mult_add_stride(const float* RESTRICT a,
16                                         const float* RESTRICT b, const size_t offseta,
17                                         const size_t offsetb, const size_t strideb) {
18         return
19             a[offseta      ] * b[offsetb      ]+
20             a[offseta + 1] * b[offsetb + strideb]+
21             ...
22             a[offseta + 16] * b[offsetb + 16*strideb];
23     }
24     static inline float stencil_2D(const float* const RESTRICT fdxyw,
25                                    const float* const RESTRICT ph,
26                                    const size_t nx, const size_t indx) {
27         return
28             fdxyw[12]*(ph[indx-12]+ph[indx+12]+ph[indx-12*nx]+ph[indx+12*nx]) +
29             fdxyw[11]*(ph[indx-11]+ph[indx+11]+ph[indx-11*nx]+ph[indx+11*nx]) +
30             ...
31             fdxyw[ 1]*(ph[indx- 1]+ph[indx+ 1]+ph[indx- 1*nx]+ph[indx+ 1*nx]) +
32             fdxyw[ 0]*(ph[indx]);
33     }

```

Listing 4.9: DRTM kernel in CilkPlus vector notation

```

1      #pragma omp parallel for collapse(2), schedule(dynamic)
2      for(int biz = 0; biz < iz2; biz += BZ) {
3          for(int biy = 0; biy < iy2; biy += BY) {
4              float tmp[ix2], tmp2[ix2];
5              for (int iz = biz; iz < (iz2 < biz+BZ ? iz2 : biz + BZ); ++iz) {
6                  for (int iy = biy; iy < (iy2 < biy+BY?iy2 : biy + BY); ++iy) {
7                      const size_t indx = box_offset + f(iy, iz);
8                      const size_t offsetpv = indx-const1, offset = f1(iz, iz1);
9                      mult_add_stride_cilk(fdzw, pv, offset, offsetpv, nxny, tmp,
10                                     ix2);
11                      tmp[ 0:ix2 ] = vel[ indx:ix2 ] * tmp[ 0:ix2 ];
12                      stencil_2D_cilk(fdxyw, ph, nx, indx, tmp2, ix2);
13                      ...
14              }
15      static inline void mult_add_stride_cilk(const float* RESTRICT a,
16      const float* RESTRICT b, const size_t offseta,
17      const size_t offsetb, const size_t strideb,
18      float * const RESTRICT res, const unsigned int sz) {
19      res[0:sz] = a[offseta      ] * b[offsetb      : sz]+
20      a[offseta + 1] * b[offsetb + strideb : sz]+
21      ...
22      a[offseta + 16] * b[offsetb + 16*strideb : sz]+
23      }
24      static inline void stencil_2D_cilk(const float* const RESTRICT fdxyw,
25      const float* const RESTRICT ph,
26      const size_t nx, const size_t indx,
27      float * const RESTRICT res, const unsigned int sz) {
28      res[0:sz] = fdxyw[12]*(ph[indx-12:sz]+ph[indx+12:sz]+
29      ph[indx-12*nx:sz]+ph[indx+12*nx:sz]) +
30      fdxyw[11]*(ph[indx-11:sz]+ph[indx+11:sz]+
31      ph[indx-11*nx:sz]+ph[indx+11*nx:sz]) +
32      ...
33      fdxyw[ 1]*(ph[indx- 1:sz]+ph[indx+ 1:sz]+
34      ph[indx- 1*nx:sz]+ph[indx+ 1*nx:sz]) +
35      fdxyw[ 0]*(ph[indx:sz]);
36      }

```

Listing 4.8 shows a portion of the stencil kernel written in scalar notation and Listing 4.9 its CilkPlus equivalent. We used the Intel C++ compiler, version 2016 for this exercise and compiled using the command : `icpc -openmp -xAvx stencil.cpp`. We executed it using a single process with eight threads per process. Arrays with 32-byte alignment are allocated using `_mm_malloc`. Clauses such as `__assume_aligned(base_ptr, 32)` are used to hint the compiler that `base_ptr` is 32-byte aligned and `__assume(size%32 == 0)` to hint that `size` is divisible by 32. Aligning arrays to 32-byte boundary and rewriting DRTM’s stencil kernel using CilkPlus notation improved performance of the stencil by roughly 25%.

To get more insight into what is causing the performance improvement, we ran the three configurations of the stencil kernel benchmark using Intel Software Development Emulator (SDE) [48] and analyzed the assembly using VTune. SDE gives a histogram of the dynamic instruction mix for an execution. The counts of different instructions in the 32-byte aligned and non-aligned executions are roughly the same. Comparing executions of the aligned and non-aligned kernels using VTune revealed that the assembly code executed is the same, but the aggregate number of clock ticks attributed to various statements in the source code for the aligned execution is fewer than the unaligned one. This led us to conclude that operations on aligned data are performed more efficiently by the floating point units even when same instructions are used.

A comparison of instruction mix histograms of the compiler-vectorized scalar version of the stencil code and version of the code manually vectorized using the CilkPlus notation revealed that the instruction counts are quite different. Execution of the CilkPlus vector notation code has 25% fewer bytes read from memory. However, the number of bytes written back is doubled. Since the number of bytes loaded is an

order of magnitude higher than the number of bytes stored, performance improves due to the reduction in the number of bytes loaded. Less data movement is an indication of better reuse of registers. CilkPlus vector notation code uses more 32-byte block data movement compared to the scalar code and, therefore, data movement is more efficient. It doubled the usage of 32-byte block data movement rather than using smaller blocks. The CilkPlus vector notation generates code that reduced the dynamic frequency of AVX instructions by roughly 30% and thus make more efficient use of floating point units than compiler-vectorized scalar code. For example, the dynamic frequency of the `VINSERTF128` instruction, which is used to set values of a 256-bit register by parts if it is not possible to fill it using a quadword operation, is reduced in CilkPlus notation code. Therefore, the generated code for the kernel that uses CilkPlus vector notation executes fewer AVX instructions. Analysis of the assembly code using VTune revealed that the executable generated from the kernel that uses CilkPlus vector notation is rather different from scalar one. The dynamic frequency of BINARY instructions which includes `ADD`, `DEC`, `CMP` in the execution of the CilkPlus notation code is double that of the scalar one.

### 4.3 Assessing the Tuned Application

Figure 4.16 shows a screenshot of `hpcviewer` visualization of the profile of optimized version of DRTM. The final version includes code optimizations to

1. reduce overhead of abstractions (Section 4.2.1).
2. reduce thread level load imbalance (Section 4.2.2).
3. overlap communication with computation (Section 4.2.3).

Scope	REALTIME (usec):Sum (I)	▼ REALTIME (usec):Sum (E)
Experiment Aggregate Metrics	8.80e+10 100 %	8.80e+10 100 %
‣ void DRTM::fwd_step_vti_du<12ul, 8, true>(float const*, float*	4.38e+10 49.8%	4.34e+10 49.3%
‣ omp_idle()	1.16e+10 13.2%	1.16e+10 13.2%
‣ DRTM::OpenMPDevice::async_pack_values_impl(float**, float*	8.51e+09 9.7%	8.26e+09 9.4%
‣ DRTM::OpenMPDevice::async_unpack_values_impl(float**, float*	6.65e+09 7.6%	6.42e+09 7.3%
‣ DRTM::OpenMPDevice::async_halo_y_interpolate_impl(bool, float*	3.97e+09 4.5%	3.94e+09 4.5%
‣ __kmp_wait_sleep	5.30e+09 6.0%	3.63e+09 4.1%
‣ DRTM::OpenMPDevice::async_halo_x_interpolate_impl(bool, float*	2.68e+09 3.0%	2.64e+09 3.0%

Figure 4.16 : **hpcviewer** visualization of profile of the final optimized version

Table 4.3 : Profile comparison of original and optimized versions with percent of total execution time in parentheses

Scope	Original value in $\mu s$	Optimized value in $\mu s$
Stencil Computation	4.54E+10 (37.8%)	4.38E+10 (49.8%)
Idleness	3.92E+10 (32.7%)	1.66E+10 (19%)
memcpy	8.92E+9 (7.4%)	3.62E+8 (0.4%)

4. improve cache reuse (Section 4.2.4).
5. reduce process level load imbalance (Section 4.2.5).

A comparison of the profiles of the original version (shown in Figure 4.1) and optimized version (shown in Figure 4.16) is given in Table 4.3. The experiments are run using 16 MPI processes on the large data set with eight threads per process. Each node in the cluster executes two MPI processes. To verify consistency of the results, we ran each version of DRTM two times and the variation in their total execution times (original:870s and optimized:625s) is within a range of  $\pm 2s$ .

In the optimized version of DRTM, the stencil code takes less time than it does in the original version. Also it occupies roughly 50% of the execution time instead of



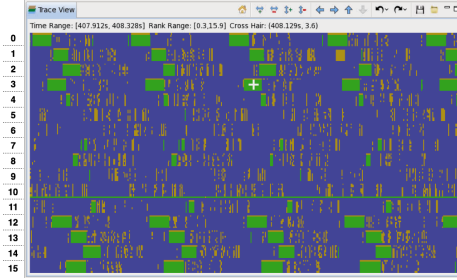


Figure 4.17 : `hpctraceviewer` visualization of few timesteps before performing process level load imbalance reduction optimization. Idleness (green) after the stencil computation is more for processes at the top and bottom.

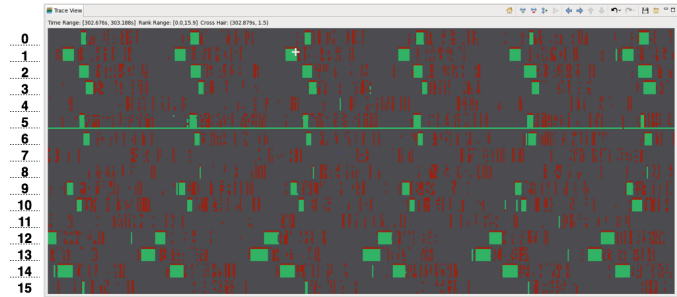


Figure 4.18 : `hpctraceviewer` visualization of few timesteps of the final optimized version. Waiting after the stencil computation is shown in green. MPI ranks are marked on the left side.

38%. Idleness is reduced by nearly half. The original version of the code had threads idle for almost one third of the time. In the optimized code, thread idleness is reduced to 19%. By using shallow copies of halo regions, memory copy overhead of 7.4% for the original code drops to 0.4% in the optimized version. Overall, the performance improved from 28%, 870s to 625s while using the 4x4 configuration run with the large data set input test case on a cluster of 8 nodes with 16 processes and eight threads per process.

Figure 4.18 shows a screenshot of `hpctraceviewer` visualization of few timesteps of the optimized version of DRTM. This includes the five code optimizations listed at the beginning of Section 4.3. Waiting for messages to arrive after the stencil computation is shown in green. By reducing process level load imbalance as described in Section 4.2.5, we are able to decrease the idle time. For example, compared with Figure 4.17 we can see that waiting has decreased by comparing the size of green bars, say for process-0 in both cases.

There are still opportunities for improvement in domain decomposition to ensure

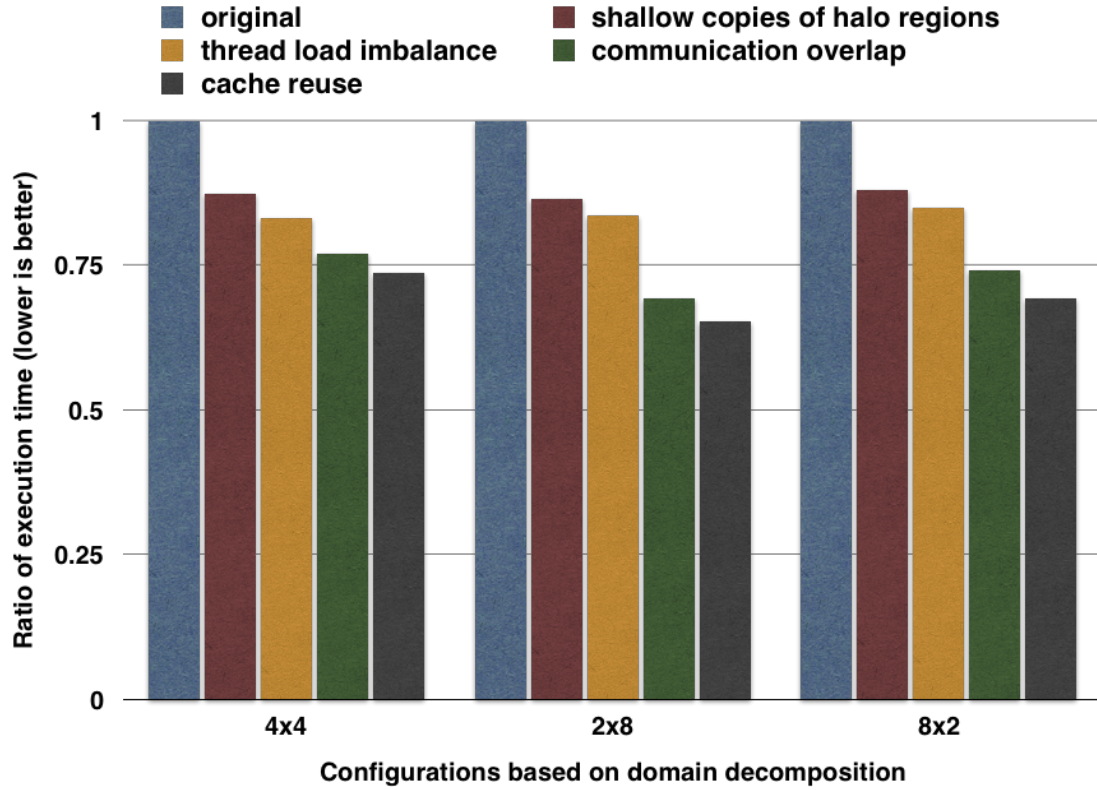


Figure 4.19 : The graph shows improvement when optimizations are applied. 4x4, 2x8, and 8x2 are domain decompositions of the X-Y plane on a cluster with Sandy bridge family of processors (Xeon E5-2670). The performance improvements are observed when different domain decompositions are used.

better load balancing. From Section 4.2.4, we can see that the hit rate of the optimized code is roughly 90%. The high hit rate implies that to improve performance the best way is to reuse registers [49, 50]. Also, working with arrays that are 32-byte aligned might help to make better use of floating point units and improve performance.

Figure 4.19 shows improvement when each code optimization is used for different domain decompositions. It takes roughly an hour to execute the large data input test case which includes a forward and backward solve phase along with writing output back to persistent storage. Since we have to run the experiment multiple times on different domain decompositions with various code optimizations, one hour per exe-

cution makes it too long to complete the full evaluation. In this thesis, we are not trying to optimize the storage efficiency, and therefore, we disabled the write back of output. Also, since backward phase contains the same operations as that of forward phase, optimizing the forward phase helps to improve the performance of backward phase too. Therefore, for our experiments, we ran only the forward solve phase which takes around 15 minutes for the large data input. Each optimization is used on top of the previous ones. For example, the vertical bar for thread local imbalance includes both shallow copies of halo regions and reduction of thread local imbalance optimizations. From Figure 4.19, we can see that performance improvement is observed even when different domain decompositions are used. Replacing deep copies of halo regions with shallow copies and reserving one of the OpenMP threads to manage communication helped to improve performance reasonably well (around 10% each). Reducing load imbalance between threads by using OpenMP `dynamic` scheduling and improving cache reuse using loop interchange improved performance by a moderate amount (around 5% each).

To ensure that the improvements are consistent across different hardware configurations we ran the large data set experiments on a different cluster. Each compute node in the cluster has two sockets. Each socket has a 2.8GHz Intel Xeon X5660 processor. Each processor has six cores with one thread per core (two threads per core in case Hyper-Threading is enabled). Each compute node contains 48GB RAM. Compute nodes are connected using InfiniBand interconnect with a bandwidth of 40Gb/s between nodes. The experiments are run with 16 MPI processes. Each node in the cluster executes two MPI processes, and each process contains six OpenMP threads. From Figure 4.20, we can see that the optimizations deliver similar performance improvements on the new cluster with a different hardware configuration.

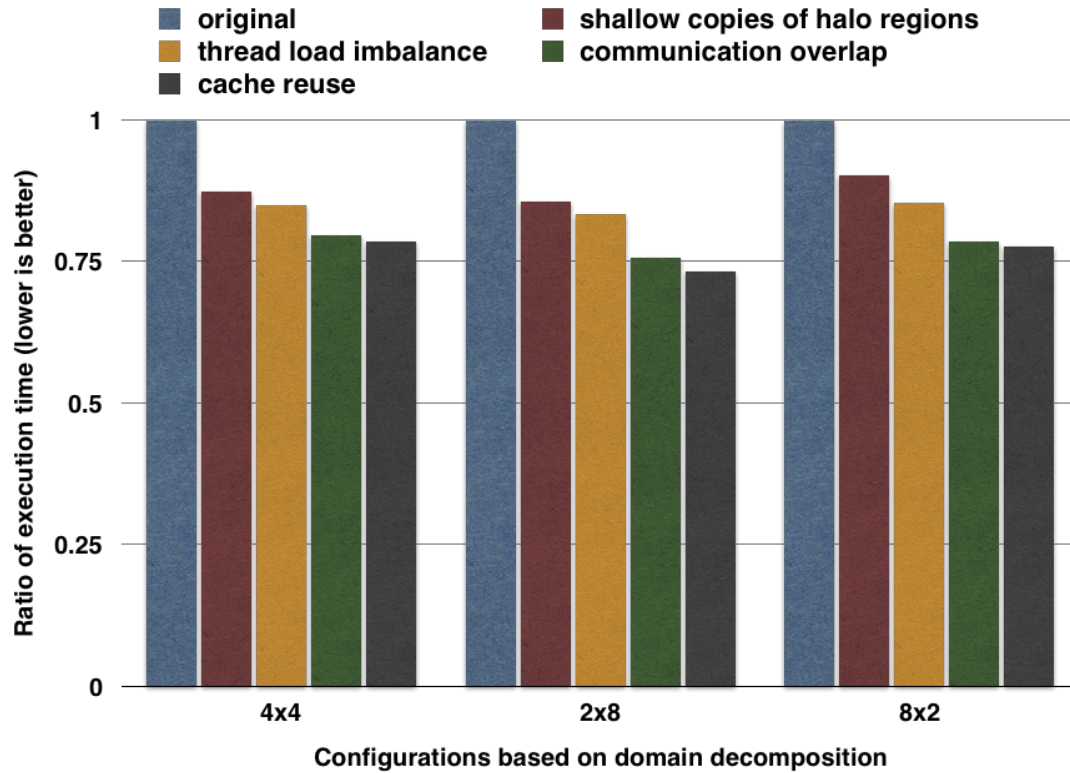


Figure 4.20 : The graph shows improvement when optimizations are applied. 4x4, 2x8, and 8x2 are domain decompositions of the X-Y plane on a cluster with Westmere family of processors (Xeon X5660). Performance improvements due to optimizations are observed on the new cluster with a different hardware specification.

#### 4.3.1 Verification

A utility that compares the output point-by-point and prints the highest ten differences was available to check whether the output of the original and optimized versions match. However, the four code optimizations (Sections 4.2.1 to 4.2.4) applied to DRTM do not change the order of calculations and therefore we expect that the output generated by the original and optimized versions of the code to be the same. By comparing the outputs of the original and optimized versions of the program with the Linux `diff` utility, we verified that the solution computed by the two versions of the code are identical.

## Chapter 5

### Conclusions and Future Work

Performance analysis and optimization of complex applications such as DRTM in a distributed memory environment is difficult. There are many factors that affect the performance such as functional unit utilization, load balance between nodes, contention for resources such as interconnect and memory bandwidth, synchronization delays, memory hierarchy and pipeline utilization. Performance analysis tools provide insight into the runtime characteristics of such complex applications. We evaluated the functionalities of various tools for analyzing the performance of DRTM - a complex application that employ both message-passing and threaded parallelism. We describe the process of tuning a complex scientific computing application to tailor it towards modern clusters with the help of performance analysis tools. This exercise showed that it is not sufficient to tune the application only for floating point units, but we have to look at the whole picture including threading and interprocess communication. Insights from the tools were critical. Without tools we could easily miss that fact that memory copies were causing a significant overhead. In some cases, fixing problems was not particularly difficult once we identified them. HPCToolkit helped to identify and analyze performance data from different levels - communication between processes, threading within a process and functional unit utilization within a core. HPCToolkit's profile view was enough to identify the problems, for example that threads spend 33% of their time as idle, but it does not provide insight into the nature of the problem. HPCToolkit's trace view helped us to pinpoint that idleness was caused by thread

load imbalance due to tiling and communication delays. Although we used the trace view to identify load imbalance, HPCToolkit also provides a “Thread-level View” [51] that could have been used to identify such issues. The optimizations we performed in response to the insights we obtained using the tools improved the performance of the application by roughly 30%. The problems that we identified such as load imbalance due to tiling, insufficient communication-computation overlap, lack of register reuse and their remedies are typical for calculations using dense arrays running on modern clusters.

Apart from helping us to improve the performance of DRTM, tools also provided us with insights regarding further opportunities for improvement. The optimizations we applied reduced idleness, but there is still room to reduce it further. Improving domain decomposition to reduce load imbalance across nodes is the first step towards this goal. Reuse of registers can help to improve performance. Register reuse in a higher order stencil is difficult. Since DRTM uses a higher order stencil, the possibility of using partial stencil computations to increase reuse of registers can be explored [49, 50]. Aligning arrays to a 32-byte boundary might help use floating point units more efficiently. Solving the communication-computation overlap problem efficiently within MPI library [44] than user code would be an ideal solution for the overlap problem.

In future, trying to perform the optimizations discussed in this thesis automatically would be a significant step. One example would be to convert automatically from the scalar notation to CilkPlus Vector notation. Another possibility is to automatically determine the ordering of loops that aligns with the placement of data, say for example whether iterating through X axis should be the outer loop or not.

## Bibliography

- [1] Intel, “Intel Xeon Processor E5-2600/4600 Product Family Technical Overview.” <https://software.intel.com/en-us/articles/intel-xeon-processor-e5-26004600-product-family-technical-overview>.
- [2] M. Fehler, “Seismic Migration Imaging,” in *Handbook of Signal Processing in Acoustics*, pp. 1585–1592, Springer, 2008.
- [3] E. Baysal, D. D. Kosloff, and J. W. Sherwood, “Reverse time migration,” *Geophysics*, vol. 48, no. 11, pp. 1514–1524, 1983.
- [4] M. Araya-Polo, F. Rubio, R. De la Cruz, M. Hanzich, J. M. Cela, and D. P. Scarpazza, “3D seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors,” *Scientific Programming*, vol. 17, no. 1-2, pp. 185–198, 2009.
- [5] A.-C. Lesage, M. Araya-Polo, and G. Houzeaux, “Wave acoustic propagation for geophysics imaging, finite difference vs finite element methods comparison and boundary condition treatment,” 2008.
- [6] B. Fornberg, “Generation of finite difference formulas on arbitrarily spaced grids,” *Mathematics of computation*, vol. 51, no. 184, pp. 699–706, 1988.
- [7] E. Süli, *Finite element methods for partial differential equations*. Oxford University Computing Laboratory, 2002.

- [8] J. F. Claerbout, “Toward a unified theory of reflector mapping,” *Geophysics*, vol. 36, no. 3, pp. 467–481, 1971.
- [9] M. P. I. Forum, “MPI: A message-passing interface standard,” tech. rep., Knoxville, TN, USA, 1994.
- [10] D. Luebke, “CUDA: Scalable parallel programming for high-performance scientific computing,” in *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pp. 836–838, IEEE, 2008.
- [11] O. W. Group *et al.*, “The openacc application programming interface,” 2011.
- [12] A. Munshi *et al.*, “The opencl specification,” *Khronos OpenCL Working Group*, vol. 1, pp. 11–15, 2009.
- [13] L. Dagum and R. Menon, “OpenMP: An industry-standard api for shared-memory programming,” *IEEE Comput. Sci. Eng.*, vol. 5, pp. 46–55, Jan. 1998.
- [14] Intel, “ITAC.” <https://software.intel.com/en-us/intel-trace-analyzer>.
- [15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’05*, (New York, NY, USA), pp. 190–200, ACM, 2005.
- [16] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, “HPCToolkit: Tools for performance analysis of optimized parallel programs,” *Concurr. Comput. : Pract. Exper.*, vol. 22, pp. 685–701, Apr. 2010.



- [17] Intel, “VTune.” <https://software.intel.com/en-us/intel-vtune-amplifier-xe>.
- [18] Allinea, “MAP.” <http://www.allinea.com/products/map>.
- [19] Intel, “PCM.” <https://software.intel.com/en-us/articles/intel-performance-counter-monitor>.
- [20] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne, “Perfexpert: An easy-to-use performance diagnosis tool for hpc applications,” in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.
- [21] C. E. Leiserson, “Fat-trees: Universal Networks for Hardware-efficient Supercomputing,” *IEEE Trans. Comput.*, vol. 34, pp. 892–901, Oct. 1985.
- [22] G. F. Pfister, “An introduction to the infiniband architecture,” *High Performance Mass Storage and Parallel I/O*, vol. 42, pp. 617–632, 2001.
- [23] B. Mutnury, F. Paglia, J. Mobley, G. K. Singh, and R. Bellomio, “QuickPath Interconnect (QPI) design and analysis in high speed servers,” in *Electrical Performance of Electronic Packaging and Systems (EPEPS), 2010 IEEE 19th Conference on*, pp. 265–268, IEEE, 2010.
- [24] A. Eichenberger, J. Mellor-Crummey, M. Schulz, N. Coptly, J. DelSignore, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, “OMPT and OMPD: OpenMP tools application programming interfaces for performance analysis and debugging,” tech. rep., 2013.

- [25] P. J. Mucci, S. Browne, C. Deane, and G. Ho, “PAPI: A portable interface to hardware performance counters,” in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pp. 7–10, 1999.
- [26] Intel, “Performance Monitoring Unit Sharing Guide.” <https://software.intel.com/sites/default/files/ea/95/30388>.
- [27] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu, “Fast seismic modeling and reverse time migration on a GPU cluster,” in *High Performance Computing & Simulation, 2009. HPCS’09. International Conference on*, pp. 36–43, IEEE, 2009.
- [28] J. Cabezas, M. Araya-Polo, I. Gelado, N. Navarro, E. Moranco, and J. M. Cela, “High-performance reverse time migration on GPU,” in *Chilean Computer Science Society (SCCC), 2009 International Conference of the*, pp. 77–86, IEEE, 2009.
- [29] A. Qawasmeh, B. Chapman, M. Hugues, and H. Calandra, “GPU technology applied to reverse time migration and seismic modeling via openacc,” in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM ’15*, (New York, NY, USA), pp. 75–85, ACM, 2015.
- [30] L. Lu and K. Magerlein, “Multi-level parallel computing of reverse time migration for seismic imaging on blue gene/q,” in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’13, (New York, NY, USA), pp. 291–292, ACM, 2013.

- [31] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, “A stencil compiler for short-vector SIMD architectures,” in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS ’13, (New York, NY, USA), pp. 13–24, ACM, 2013.
- [32] M. Christen, O. Schenk, and H. Burkhardt, “PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures,” in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS ’11, (Washington, DC, USA), pp. 676–687, IEEE Computer Society, 2011.
- [33] M. Christen, O. Schenk, and Y. Cui, “PATUS for convenient high-performance stencils: Evaluation in earthquake simulations,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC ’12, (Los Alamitos, CA, USA), pp. 11:1–11:10, IEEE Computer Society Press, 2012.
- [34] Y. Luo, G. Tan, Z. Mo, and N. Sun, “FAST: A fast stencil autotuning framework based on an optimal-solution space model,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS ’15, (New York, NY, USA), pp. 187–196, ACM, 2015.
- [35] L. Pouchet, “Polyopt/c: A polyhedral optimizer for the ROSE compiler,” 2011.
- [36] L.-N. Pouchet, C. Bastoul, and U. Bondhugula, “PoCC: the polyhedral compiler collection, 2010.”
- [37] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, “Decoupling Algorithms from Schedules for Easy Optimization of Image

- Processing Pipelines,” *ACM Trans. Graph.*, vol. 31, pp. 32:1–32:12, July 2012.
- [38] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’13, (New York, NY, USA), pp. 519–530, ACM, 2013.
- [39] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, “The Pochoir Stencil Compiler,” in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’11, (New York, NY, USA), pp. 117–128, ACM, 2011.
- [40] Intel, “Intel cilk plus.” <https://www.cilkplus.org/>.
- [41] T. Nguyen, P. Cicotti, E. Bylaska, D. Quinlan, and S. B. Baden, “Bamboo: Translating MPI applications to a latency-tolerant, data-driven form,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC ’12, (Los Alamitos, CA, USA), pp. 39:1–39:11, IEEE Computer Society Press, 2012.
- [42] D. Buettner, J. Acquaviva, and J. Weidendorfer, “Real asynchronous MPI communication in hybrid codes through OpenMP communication tasks,” in *ICPADS*, *Seoul, Korea*, 2013.
- [43] S. Chatterjee, S. Tasırlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, “Integrating asynchronous task parallelism with MPI,” in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 712–725, IEEE, 2013.

- [44] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó, “Improving Concurrency and Asynchrony in Multithreaded MPI Applications Using Software Offloading,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’15, (New York, NY, USA), pp. 30:1–30:12, ACM, 2015.
- [45] Intel, “AVX.” <https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions>.
- [46] J. Keasler, T. Jones, and D. Quinlan, “TALC: A Simple C Language Extension For Improved Performance and Code Maintainability,” 2008.
- [47] K. Sharma, I. Karlin, J. Keasler, J. R. McGraw, and V. Sarkar, “Data layout optimization for portable performance,” in *Euro-Par 2015: Parallel Processing*, pp. 250–262, Springer, 2015.
- [48] Intel, “Intel software development emulator.” <https://software.intel.com/en-us/articles/intel-software-development-emulator/>.
- [49] P. Basu, M. Hall, S. Williams, B. V. Straalen, L. Oliker, and P. Colella, “Compiler-directed transformation for higher-order stencils,” in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pp. 313–323, IEEE, 2015.
- [50] R. de la Cruz and M. Araya-Polo, “Algorithm 942: Semi-stencil,” *ACM Trans. Math. Softw.*, vol. 40, pp. 23:1–23:39, Apr. 2014.
- [51] R. University, “HPCToolkit User’s Manual.” <http://hpctoolkit.org/manual/HPCToolkit-users-manual.pdf>.