Procedia Computer Science
Volume 80, 2016, Pages 8–18
ICCS 2016. The International Conference on Computational Science

Performance Analysis and Optimization of a Hybrid Seismic Imaging Application

Sri Raj Paul[1], Mauricio Araya-Polo[2], John Mellor-Crummey[1], and Detlef Hohl[2]

[1] Rice University, Houston, TX, USA
{sriraj, johnmc}@rice.edu
[2] Shell International Exploration & Production Inc., Houston, TX, USA
{mauricio.araya, detlef.hohl}@shell.com

Abstract
Applications to process seismic data are computationally expensive and, therefore, employ scalable parallel systems to produce timely results. Here we describe our experiences of using performance analysis tools to gain insight into an MPI+OpenMP code developed by Shell that performs Reverse Time Migration on a cluster to produce models of the subsurface. Tuning MPI+OpenMP programs for modern platforms is difficult, and, therefore, assistance is required from performance analysis tools. These 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 Shell's prototype distributed-memory Reverse Time Migration code by roughly 30 percent.

Keywords: reverse time migration, performance analysis, MPI+OpenMP, hybrid programming models

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 [14]. **R**everse **T**ime **M**igration (RTM) [6] is the current preferred approach to creating subsurface images. RTM does this by simulating the propagation of an acoustic wave through subsurface layers [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. Finally, a cross-correlation between both fields is performed to generate an output image. RTM gives more accurate results than previous methods such as Wave Equation Migration [11]. RTM is computationally expensive, so we explore an implementation of RTM on a distributed memory system known as DRTM. DRTM uses message passing to share data between compute nodes using MPI [15] and threaded parallelism to maximize

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016
© The Authors. Published by Elsevier B.V.

doi:10.1016/j.procs.2016.05.293

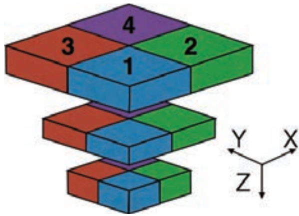

Figure 1: A domain decomposition of data among four processes by dividing the X-Y plane. The size of the slabs decreases as we go deep due to the fewer data points collected at deeper depth.

Algorithm 1: Forward phase of the DRTM

```
for all time steps do
    stencil computation for halo-regions;
    pack halo-regions;
    send halo-regions to neighbors;
    stencil computation for internal-regions;
    receive halo-regions from neighbors;
    interpolate if necessary;
end for
```

utilization of functional units within a node using CUDA [25], OpenACC [16], OpenCL [23] or OpenMP [12]. DRTM is a PDE solver that applies a high order stencil to a 3D block of data at each time step. Figure 1 shows a representation of the 3D data. The DRTM uses a multiblock algorithm in which the slab size (the number of data points) decreases as we go deeper in the direction of Z-axis.

Algorithm 1 shows a pseudo-code of the forward phase of the DRTM that gets executed in each MPI rank.[1] Each time step starts with stencil computation on data in the halo regions.[2] Then halo regions are packed into messages and exchanged between neighbors using nonblocking MPI primitives for point-to-point communication with the aim of overlapping communication with computation. Next, the code performs stencil computation for non-halo regions while communications are pending for data in the halo regions. Once stencil computation finishes, each MPI process waits for the arrival of halo data sent prior to the stencil computation. Finally, the code unpacks data from the halo exchanges and performs interpolation where necessary.

A goal of this work was to understand how to tailor the MPI+OpenMP implementation of DRTM to a cluster. Tuning such hybrid applications requires analysis at multiple levels:

- domain decomposition and interprocess communication across nodes,
- threaded parallelism on a node, and
- functional unit and cache utilization within a core.

To aid in this process, we use Rice University's HPCToolkit[2] performance tools based on a comparison study of various tools. HPCToolkit provides a unified view of interprocess interactions, threading, and functional unit utilization details and, therefore, we did an in-depth analysis of the DRTM using HPCToolkit as described in Section 2. HPCToolkit helps to measure and analyze the performance of single-core, multi-core and distributed systems. It uses the sampling of timers or hardware performance counters to gather call stack profiles and call stack traces. The overhead introduced during execution is a few percent. It can analyze fully optimized applications and attribute performance data back to the source code. To visualize performance data collected, HPCToolkit includes two presentation tools: `hpcviewer` [3] and `hpctraceviewer` [29]. `hpcviewer` presents performance data in a code-centric view and `hpctraceviewer` provides a time-centric view at multiple levels of stack depth.

[1]Backward phase is similar.

[2]Each MPI process requires a narrow slab of boundary data points from each of its neighbor during stencil computation. This boundary data slab exchanged with each neighbor is called halo region.

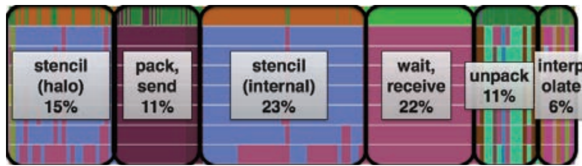

Figure 2: HPCToolkit's `hpctraceviewer` showing the division of a single time step for an MPI process containing eight thread with time along X-axis and threads along Y-axis.

Table 1: The performance profile of the original version of DRTM showing the top five time-consuming functions.

Scope	Time
fwd_step_vti_du	4.54E+10 37.8%
omp_idle	3.92E+10 32.7%
memcpy	8.92E+09 7.4%
pack_values	8.68E+09 7.2%
unpack_values	7.11E+09 5.9%

The contributions of this paper are:

- presentation of a detailed evaluation of a seismic imaging application with the help of performance analysis tools that illustrates the process of analyzing a complex hybrid application and,
- description of the insights obtained using the tools and code optimizations in response that improved the overall performance of DRTM by roughly 30%. Others can leverage on our insights to tune their hybrid scientific applications for modern clusters.

The remainder of this paper is organized as follows: Section 2 describes the iterative analysis and optimization process where we act upon the improvement opportunities identified using performance analysis tools. In Section 3, we present our overall experimental results. Section 4 includes related work. Finally, Section 5 presents our conclusions and other open issues.

2 Analysis and Optimization of DRTM

In this section, we describe details of the analysis of DRTM using HPCToolkit and how the insights got from HPCToolkit helped us to improve DRTM's performance. We present the performance improvement associated with each optimization that is performed.

2.1 Initial Assessment

To do performance analysis, we ran DRTM with 16 MPI processes where each process contained eight threads and used HPCToolkit to collect the performance profile data. Figure 2 shows the division of a time step (phases in a time step is described in Algorithm 1) for one MPI process using `hpctraceviewer` which helps to get a sense of how computation is spread over time. [3] Table 1 show a list of the most costly procedures in DRTM measured with HPCToolkit using asynchronous sampling. Stencil computation (`fwd_step_vti_du`) uses only 37.8% of the total execution time. Also, wasted resources caused due to the idle OpenMP worker threads (`omp_idle`) takes 32.7% of the execution time. Surprisingly, `memcpy` takes 7.4% of execution time. Ideally the stencil computation should fill the execution time, and idleness should be minimal. A simple place to start is to investigate why `memcpy` is taking so much of the execution time.

[3] A complete execution includes other activities such as reading data from input files and hence reported percentages do not add to 100%.

Algorithm 2: Extended memcpy for OpenMP device

```
if CopyType == SHALLOW then
    dst = src
else
    perform memcpy
end if
```

Table 2: Performance profile after the introduction of `SHALLOW` copy. `memcpy` has disappeared from the list of top time-consuming functions.

Scope	Time	
fwd_step_vti_du	4.53E+10	42.2%
omp_idle	3.62E+10	33.7%
pack_values	8.59E+09	8.0%
unpack_values	6.78E+09	6.3%
halo_y_interpolate	4.03E+09	3.8%

2.2 Improve Abstraction-layer for Node-level Threading

We first investigate the reason for the 7.4% overhead due to memory copies as identified in Section 2.1. The 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 while using OpenMP, 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 the case of OpenMP. Also, after the removal of the additional memory copies, all the programming models that used to work should continue to work.

One way to address this problem is to remove `memcpy`s while using OpenMP and directly use the source pointer for the subsequent accesses instead of destination pointer. But this would require a lot of refactoring since we need to change all references to the destination. A better way is to assign the destination with the source pointer which we refer to as `SHALLOW` copy. After `SHALLOW` copy, any change made to the destination affects the source buffer since they are pointing to the same memory location which is not the same as in the case of a normal `memcpy` because source and destination buffers are different. This side-effect does not affect correctness if source buffer is not used after the copy is invoked which is the case with DRTM. Therefore using `SHALLOW` copy does not affect the correctness of DRTM. The pseudo-code for `memcpy` interface extended with `CopyType` parameter for OpenMP device is given in Algorithm 2. This new `CopyType` parameter takes two values: `DEEP`, and `SHALLOW`.

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

The implementation of extended `memcpy` for programming models other than OpenMP ignores the `CopyType` parameter. Refactoring and replacing unnecessary memory copies with `SHALLOW` copy reduced the execution time by roughly 12%. Although memory copies took just 7% of the total time, the performance improvement achieved by its removal is 12%. The additional 5% improvement comes from the reduction of overheads caused by the OpenMP runtime associated with the invocation of multithreaded memory copies. Table 2 shows a list of the most costly procedures after the introduction of `SHALLOW` copy. Compared to Table 1, `memcpy` has disappeared from the list of top time-consuming functions; it now uses less than 1% of the total time.

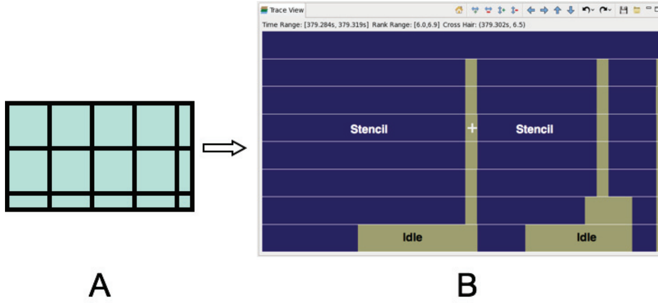

Figure 3: Time-centric view of stencil computation in a single time step for an MPI rank with eight threads. Smaller tiles at the high end of each dimension cause imbalance when OpenMP `static` scheduling is used. Threads that get assigned with smaller tiles finish their work early and remains idle. Blue/dark color represents stencil computation and brown/light color represents idleness.

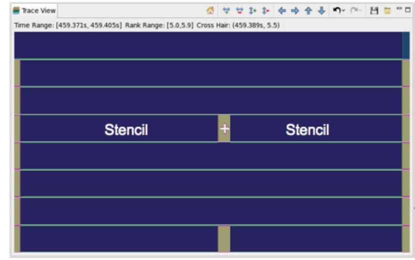

Figure 4: Time-centric view of stencil computation in a single time step for an MPI rank with eight threads after changing OpenMP scheduling from `static` to `dynamic`. Idleness has disappeared, and stencil computation(blue/dark) occupies most of the execution.

2.3 Reduce Thread-level Load Imbalance

After removing unnecessary memory copies as described in Section 2.2, we evaluated the performance of the optimized code and Figure 3:B shows a time-centric view of stencil computation in a single time step for an MPI rank as presented using `hpctraceviewer`. Here blue/dark portion represents stencil computation and brown/light for idleness. This view implies that some threads are idle during stencil calculation indicating an imbalance in the work assigned to the 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 2D region using 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 3:A. OpenMP runtime computes work partitions of the iterations 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 partial or full tiles. Using `static` scheduling assigns an equal number of points (tiles) to each thread. The difference in tile sizes causes the threads assigned to a collection of small tiles to finish early and remain idle till other threads complete their stencil computation. Changing the OpenMP scheduling strategy for the stencil computation loop to `dynamic` reduced the imbalance. A time-centric view of the stencil computation after changing to `dynamic` scheduling is shown in Figure 4. Compared to Figure 3:B, the entire execution is filled with stencil computation and idleness has nearly disappeared. Reducing the idleness helped to improve performance by roughly 5%.

2.4 Improve Overlap of Communication with Computation

After improving load balancing of threads, we evaluated the performance of the optimized code, and Figure 5 shows a time-centric view of several iterations as presented using `hpctraceviewer`. In the figure, pink color represents idleness and green for stencil computation. There is a considerable amount of idleness that occurs after the stencil computation. Even though MPI non-blocking primitives are used that are initiated before and completed after the stencil computation, processes stall after the stencil computation waiting for messages from neighbors to

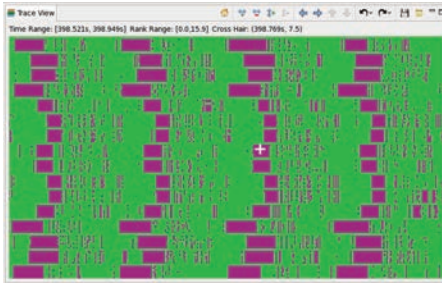

Figure 5: hpctraceviewer visualization of a few time steps for 16 MPI ranks after improving load balancing of threads shows considerable idleness (pink/dark) after stencil computation (green/light) implying a lack of communication-computation overlap.

Listing 1: Original code intended to overlap communication with computation using non-blocking MPI primitives

```
//post recv requests
MPI_Irecv();
//post send requests
MPI_Isend();
//post stencil computation
stencil_computation();
//wait for sends and revcs
//to complete
wait_all();
unpack();
```

Listing 2: Modified code that overlaps computation and communication using a dedicated communicaition thread

```
MPI_Irecv(); //post recv requests
MPI_Isend(); //post send requests
#pragma omp parallel num_threads(2)
{
  if(thread_id == 1)
    //post stencil computation
    stencil_computation();
  else if(thread_id == 0)
    //wait for send/recv completion
    wait_all();
}
unpack();
```

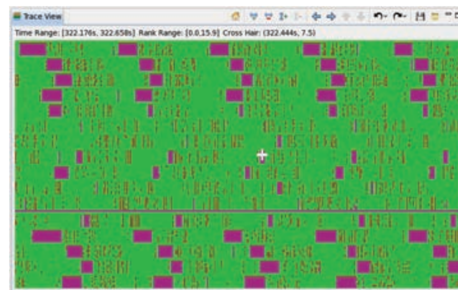

Figure 6: hpctraceviewer visualization of a few time steps for 16 MPI ranks after reserving a thread for communication. Waiting (pink/dark) after the stencil computation is reduced.

arrive. This wait caused us to question whether the communication was making progress during the stencil computation. To ensure true overlap we tried using asynchronous progress mode provided by Intel's MPI library. This approach increased the execution time by roughly 25-30%.

Another way to ensure overlap is to dedicate an OpenMP thread for communication. The original code intended to overlap communication with computation using non-blocking MPI communication primitives is given in Listing 1. In this code, sends and receives are posted asynchronously before the stencil computation and made to wait afterward for completion. To achieve real overlap, we reserved one thread (thread 0) for communication (line 10) using OpenMP `parallel` construct as shown Listing 2. The second thread (thread 1) proceeds with the stencil computation (line 7) and forks into multiple threads using nested parallelism. Also, since one thread is allocated for communication, stencil computation is performed with one less thread, so that communication thread gets a core for itself.

Figure 6 shows the `hpctraceviewer` output after the introduction of communication thread. Pink bars represent waiting after stencil computation for halo exchanged data to arrive from neighbors. Compared to Figure 5, processes in the middle section 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. Enabling overlap of communication with computation using a communication thread reduces execution time by roughly 7.5%.

Table 3: Hardware performance counter values

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

Table 4: Loop interchange reduces cache miss

Event	Old (miss rate %)	New (miss rate %)
L1 DCM	3.37E+11 (12.5%)	3.19E+11 (11.8%)
L2 DCM	1.17E+11 (35.0%)	6.67E+10 (21.0%)

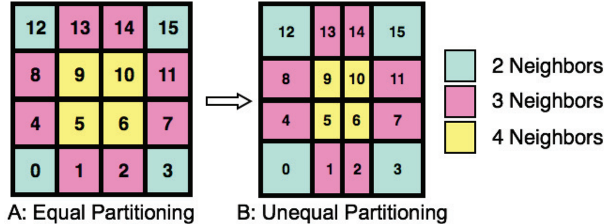

Figure 7: Partitioning data across 16 processes with MPI rank marked on each partition. 4x4 configuration (4 in X and 4 in Y direction) showing the difference in the distribution of neighbors. For example, rank-3 has two neighbors whereas rank-5 have four neighbors.

2.5 Improve Data Reuse in Cache

After adjusting the communication and threaded parallelization strategies, we next checked how functional units are being utilized. We used PAPI [22] library to measure counts of various hardware events such as cache misses, loads, stores and so on. PAPI provides a set of consistent interfaces across different architectures to use the performance counters. Few of the counter values are shown in Table 3.

From Table 4, L1 and L2 data cache miss (DCM) rate is around 12.5% and 35% respectively which seemed to be a significant number. We analyzed the loop placement of the stencil computation to get an insight into the 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 uses 49 points in the X-Y plane but only 41 points in the X-Z plane. This difference implies that if we load the X-Y plane into the cache, more values can be reused than in the case of X-Z plane. Therefore, interchanging Y and Z loops to access data in Z-Y-X order increases cache reuse. The number of cache misses before and after loop interchange is given in Table 4. Interchanging the Y and Z loops reduced the cache misses by increasing reuse and improved the performance by roughly 5%.

2.6 Reduce Process-level Load Imbalance

As discussed in Section 2.4, from Figure 6 it is clear that the processes owning edge partitions spend more time being idle, waiting for halo data to arrive from neighbors than middle ones. 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 7:A, the middle processes have more neighbors, and, therefore, they perform more work because it needs to perform halo calculation, pack, unpack and interpolate for all the neighbors. Due to the less work, edge nodes finish computation early and stay idle.

Table 5: Profile comparison of original and optimized versions with percent of execution time in parenthesis

Scope	Original value in μs	New value in μs
Stencil	4.54E10 (37.8%)	4.38E10 (49.8%)
Idleness	3.92E10 (32.7%)	1.66E10 (19.0%)
memcpy	8.92E9 (7.4%)	3.62E8 (0.4%)

DRTM had a provision to change the size of partition allocated to each MPI rank. By changing the size of the partitions, effect of the difference in the number of neighbors can be reduced. The partition size of middle nodes should be reduced to compensate for the higher number of neighbors as shown in Figure 7:B. Variable sized partitioning improved performance by roughly 2.5%.

3 Overall Results

Table 5 shows a comparison of the performance profiles of the optimized version and the original version (from Table 1). Stencil computation occupies roughly 50% of the execution time instead of 38%. Idle time is reduced to nearly half. Earlier 1/3 rd of the execution used to remain idle which has now been reduced to 19%. Another important difference is the removal of unnecessary memory copies. Overall, the performance improved by roughly 30% on 16 MPI processes with eight threads per process running on eight nodes of a cluster.

The experiments were run on a cluster connected with fat-tree topology [19] using InfiniBand interconnect. Each compute node contains 128 GB of RAM and two sockets where each socket has an Intel Xeon E5-2670 CPU. We used Intel Compiler suite version 14 for our experiments. Figure 8 presents a graphical view of improvement for each optimization with different domain decompositions. Each bar shows the cumulative result of all optimizations represented by bars to its left, say for example the third bar representing optimization to reduce thread load imbalance includes optimization to reduce memory copies using shallow copy of halo regions. From Figure 8, we can see that performance improvement is consistent across different configurations. Removal of unnecessary memory copies and the introduction of communication-thread helped in improving performance reasonably well (around 10% each). Removing thread level load imbalance using OpenMP `dynamic` scheduling and improving cache reuse using loop interchange improved performance by a moderate amount (around 5% each). Variable size domain decomposition is only used with 4X4 configuration since the difference in the number of neighbors is not so profound in the other two configurations. The insights listed here are general enough to apply to scientific applications written using the hybrid MPI+OpenMP programming model.

4 Related work

There have been several successful efforts to parallelize RTM computation onto multicore and distributed memory systems. Abdelkhalek *et al.* [1] and Cabezas *et al.* [8] used CUDA to take advantage of the computation power of GPUs. Qawasmeh *et al.* [26] employed a hybrid model which uses OpenACC to program GPU and MPI to distribute computation across nodes. Araya-Polo *et al.* [4] use OpenMP to parallelize computation across cores. Lu and Magerlein [20] employed a hybrid model with MPI to distribute work across nodes and OpenMP within a node.

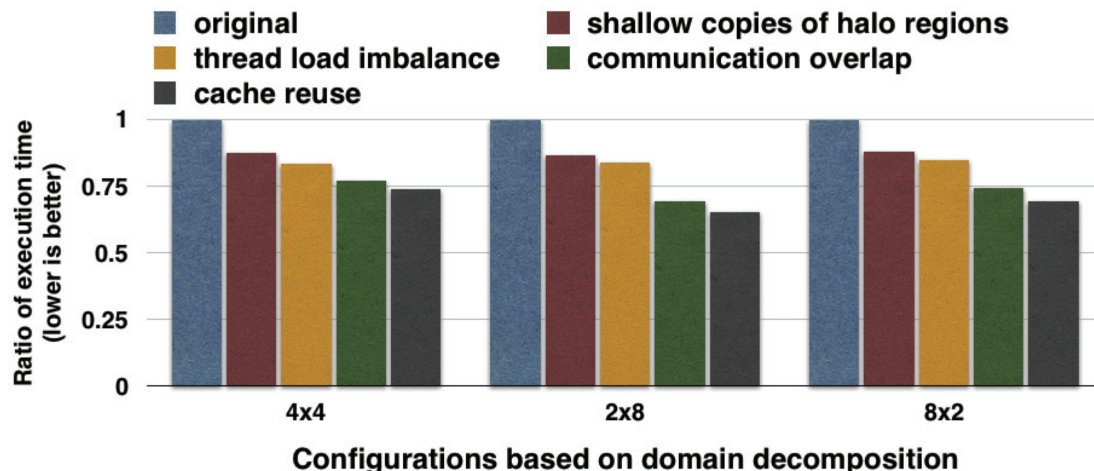

Figure 8: Bar graph showing the improvement due to each optimization for three configurations: 4x4, 2x8 and 8x2 domain decompositions of the X-Y plane. Each bar shows the cumulative result of all optimizations represented by bars towards its left.

Another direction of work is to use a framework that helps to improve productivity by expressing the stencil in a compact way. SDSL [17] is a domain specific language that can be embedded in C, C++ and MATLAB code. SDSL's backend translates the stencil specification to C code that can be further optimized using a polyhedral framework. PATUS [10] is an auto-tuning framework for stencil computation targeted at multicore CPUs and GPUs. During code generation of the stencil specification, PATUS allows to specify `strategy`: a description of the parallelization and optimization methods to be applied. Auto-tuning is used to select an optimal parameter configuration for the chosen stencil kernel and hardware platform. FAST [21] is an auto-tuning framework that employs machine learning to predict a set of optimal solutions thereby improving tuning speed. The Pochoir [30] stencil compiler allows the programmer to write the stencil specification in a domain specific language embedded in C++. Halide [27, 28] 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 compiler translates the stencil specification to a high-performing parallel CilkPlus [18] code.

Achieving efficient and effective communication while using hybrid programming models is difficult. Bamboo [24] is a source-to-source translator that translates an MPI C program into a data-driven form that overlaps communication with computation. Buettner *et al.* [7] tried to address the issue of communication-computation overlap by extending the OpenMP runtime to include communication tasks. HCMPI (Habanero-C MPI) [9], 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.* [32] tried to address the issue of overlap with an MPI offload infrastructure using a dedicated communication thread at the library level. In this approach, MPI calls get enqueued as communication tasks which get processed by the dedicated communication thread.

5 Conclusions and Open Issues

Tuning complex applications in a distributed memory environment is difficult. There are many factors that could 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. 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 floating point units, but we have to look at the whole picture including threading and interprocess communication. Insights got from the tools were critical. Without tools were could easily miss that fact that memory copies were causing a reasonable overhead. In some cases, fixing the problems was not particularly difficult once we identified them. HPCToolkit helped to view 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 give 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 the load imbalance, HPCToolkit also provides a "Thread-level View" [31] 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 insights that we listed in this paper such as load imbalance due to tiling, insufficient communication-computation overlap, lack of register reuse and their remedies are applicable to a wide range of scientific applications running on modern clusters.

Solving the communication-computation overlap problem efficiently within MPI library [32] than user code would be an ideal solution for the overlap problem. The possibility of using partial stencil computations to increase reuse of registers deserves further exploration [5, 13]. Automating the iterative optimization process is helpful when porting the application to newer hardware.

Acknowledgment

This work was partially supported by Shell International Exploration & Production Inc. under research agreement PT46021. We are grateful to Michael Thomadakis for providing feedback that helped improve this paper.

References

[1] Rached Abdelkhalek et al. Fast seismic modeling and reverse time migration on a GPU cluster. HPCS '09, pages 36–43. IEEE, 2009.

[2] L. Adhianto et al. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, April 2010.

[3] Laksono Adhianto et al. Effectively presenting call path profiles of application performance. In *39th International Conference on Parallel Processing Workshops*, pages 179–188. IEEE, 2010.

[4] Mauricio Araya-Polo et al. 3D seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors. *Scientific Programming*, 17(1-2):185–198, 2009.

[5] Protonu Basu et al. Compiler-Directed Transformation for Higher-Order Stencils. IPDPS '15, pages 313–323. IEEE, 2015.

[6] Edip Baysal et al. Reverse time migration. *Geophysics*, 48(11):1514–1524, 1983.

[7] David Buettner et al. Real Asynchronous MPI Communication in Hybrid Codes through OpenMP Communication Tasks. In *ICPADS, Seoul, Korea*, 2013.

[8] Javier Cabezas et al. High-performance reverse time migration on GPU. In *Chilean Computer Science Society (SCCC), 2009 International Conference of the*, pages 77–86. IEEE, 2009.

[9] Sanjay Chatterjee et al. Integrating asynchronous task parallelism with MPI. IPDPS '13, pages 712–725, 2013.

[10] Matthias Christen et al. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. IPDPS '11, pages 676–687. IEEE.

[11] Jon F Claerbout. Toward a unified theory of reflector mapping. *Geophysics*, 36(3):467–481, 1971.

[12] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[13] Raúl de la Cruz and Mauricio Araya-Polo. Algorithm 942: Semi-stencil. *ACM Trans. Math. Softw.*, 40(3):23:1–23:39, April 2014.

[14] Michael Fehler. Seismic Migration Imaging. In *Handbook of Signal Processing in Acoustics*, pages 1585–1592. Springer, 2008.

[15] The MPI Forum. MPI: A Message Passing Interface, 1993.

[16] OpenACC Working Group et al. The OpenACC Application Programming Interface, 2011.

[17] Tom Henretty et al. A Stencil Compiler for Short-vector SIMD Architectures. ICS '13, pages 13–24, New York, NY, USA, 2013. ACM.

[18] Intel. Intel cilk plus. https://www.cilkplus.org/.

[19] Charles E. Leiserson. Fat-trees: Universal Networks for Hardware-efficient Supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, October 1985.

[20] Ligang Lu and Karen Magerlein. Multi-level Parallel Computing of Reverse Time Migration for Seismic Imaging on Blue Gene/Q. PPoPP '13, pages 291–292, New York, NY, USA, 2013. ACM.

[21] Yulong Luo et al. FAST: A Fast Stencil Autotuning Framework Based On An Optimal-solution Space Model. ICS '15, pages 187–196, New York, NY, USA, 2015. ACM.

[22] Philip J. Mucci et al. PAPI: A Portable Interface to Hardware Performance Counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[23] Aaftab Munshi et al. The opencl specification. *Khronos OpenCL Working Group*, 1:l1–15, 2009.

[24] Tan Nguyen et al. Bamboo: Translating MPI Applications to a Latency-tolerant, Data-driven Form. SC '12, pages 39:1–39:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[25] John Nickolls et al. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[26] Ahmad Qawasmeh et al. GPU Technology Applied to Reverse Time Migration and Seismic Modeling via OpenACC. PMAM '15, pages 75–85, New York, NY, USA, 2015. ACM.

[27] Ragan-Kelley et al. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012.

[28] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. PLDI '13, pages 519–530, NY, USA, 2013. ACM.

[29] Nathan R. Tallent et al. Scalable Fine-grained Call Path Tracing. ICS '11, pages 63–74, New York, NY, USA, 2011. ACM.

[30] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The Pochoir Stencil Compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, NY, USA, 2011. ACM.

[31] Rice University. HPCToolkit User's Manual. http://hpctoolkit.org/manual/HPCToolkit-users-manual.pdf.

[32] Karthikeyan Vaidyanathan et al. Improving Concurrency and Asynchrony in Multithreaded MPI Applications Using Software Offloading. SC '15, pages 30:1–30:12, New York, NY, USA. ACM.