Quantifying Performance Benefits of Overlap using MPI-2 in a Seismic Modeling Application

S. Potluri[1], P. Lai[1], K. Tomko[2], S. Sur[1], Y. Cui[3]
M. Tatineni[3], K. Schulz[4], W. Barth[4], A. Majumdar[3] and D. K. Panda[1]
[1] Department of Computer Science and Engineering, The Ohio State University
{potluri,laipi,surs,,panda}@cse.ohio-state.edu
[2] Ohio Supercomputer Center
ktomko@osc.edu
[3] San Diego Supercomputer Center
{mahidhar,yfcui,majumdar}@sdsc.edu
[4] Texas Advanced Computing Center
{bbarth,karl}@tacc.utexas.edu

Abstract

AWM-Olsen is a widely used ground motion simulation code based on a parallel finite difference solution of the wave-propagation equations. This application runs on tens of thousands of cores and consumes several million CPU hours on the TeraGrid Clusters every year. A significant portion of its run-time (37% in a 4,096 process run), is spent in MPI communication routines. Hence, it demands an optimized communication design coupled with a low-latency, high-bandwidth network and an efficient communication subsystem for good performance. In this paper, we analyze the performance bottlenecks of the application as regards to the time spent in MPI communication calls. We find that much of this time can be overlapped with computation using MPI non-blocking calls. We use both Two-sided and MPI-2 One-sided communication semantics to re-design the communication in AWM-Olsen. We find that with our new design using MPI-2 one-sided communication semantics, the entire application can be sped up by 12% at 4K processes and by 10% at 8K processes on a state-of-the-art InfiniBand cluster, Ranger at the Texas Advanced Computing Center (TACC).

1 Introduction

Modern computing systems have enabled HEC applications to run on tens of thousands of processor cores. At such scale, the communication design of applications becomes crucial for their overall performance and scaling. In this work, we choose one such commonly used earthquake modeling application, AWM-Olsen [16, 17], and enhance its communication design to improve

its overall performance at large scale. AWM-Olsen, like most commonly used HEC codes, uses the Message Passing Interface (MPI) for managing parallel processes and the communication among them. It has a nearest neighbor communication pattern implemented using point-to-point semantics. As a first step to improve the communication design of the application we analyze the run time of the application to see how much time is spent in communication as it scales to large number of processes. Then we dissect the time spent in MPI calls and evaluate the potential for improvement by redesigning the communication. The analysis in Section 3, Figure 5(b) reveals that, on 4096 processors, the application spends 31% of its time in MPI_Waitall calls waiting on progress of communication. This shows the potential for improvement by overlapping communication with computation. So we investigate designs to hide latency in the nearest-neighbor communication pattern of this highly parallel ground motion code. We explore the following questions in the process.

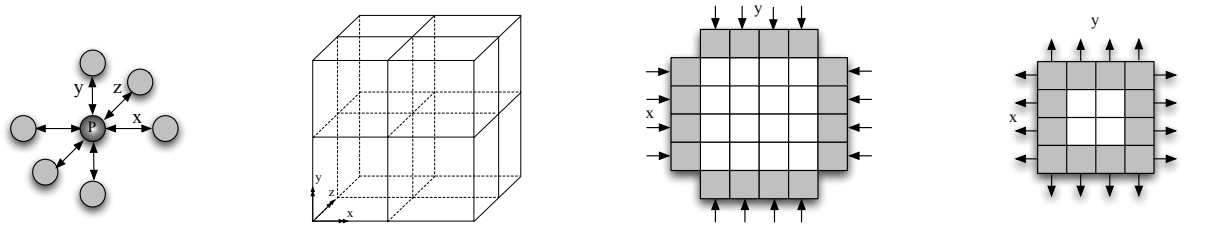
- How much overlap can we achieve in non-blocking communication designs?
- What are the design alternatives to consider when writing an efficient non-blocking communication exchange?
- Can MPI-2 one-sided communication primitives provide better performance than the frequently used two-sided primitives?

In exploring the one-sided approach, some additional questions arise as examples are scarce and the calls have not been heavily used.

- How well do MPI-2 one-sided communications work at very large scale?
- How does one write an efficient one-sided nearest-neighbor exchange? What semantics should one use?

To address these questions, we develop multiple non-blocking implementations using the asynchronous two-sided and the active synchronization-based one-sided communication mechanisms made available in MPI-1 and MPI-2, respectively. We compare our one-sided Post/Wait -Start/Complete-based version with equivalently restructured Isend/Irecv/Waitall-based versions. These designs are presented in-depth to illustrate how an MPI application can effectively hide communication latency utilizing either MPI-1 two-sided asynchronous communication semantics or MPI-2 one-sided semantics. The proposed designs have been evaluated on the TACC Ranger system using several thousands of cores. Our experiments show that the one-sided semantics provide better performance gains from overlap, 12% of the application run time, than the non-blocking two-sided semantics.

The remainder of this paper is organized as follows. Section 2 gives a brief review of the petascale AWM-Olsen, related work and provides an overview of techniques used to overlap computation and communication using MPI-2. In Section 3, we present a detailed performance analysis of the AWM-Olsen application and provide a model of the theoretical bound for performance improvement. This is followed in Section 4 by new communication designs for reducing synchronization and achieving overlap using asynchronous two-sided and MPI-2 one-sided communication mechanisms. In Section 5, we present a detailed performance evaluation. Finally, we present the conclusions and future work in section 6.



(a) Velocity and Stress Update Data Dependencies

(b) Three Dimensional Data Decomposition

(c) Receiving Data into Ghost Cells

(d) Sending Computed Data to Neighbors

Figure 1: Data Transfer Patterns in AWM-Olsen

2 Background and Related Work

2.1 Anelastic Wave Model (AWM) and its Application

AWM-Olsen is a community model [16, 17, 7] used by researchers at the Southern California Earthquake Center (SCEC) for wave propagation simulations, dynamic fault rupture studies, physics-based seismic hazard analysis, and improvement of the structural models. The AWM-Olsen code has been scaled to more than a hundred thousand processor cores, and consumed more than twenty millions allocation hours in 2009 on the NSF TeraGrid Ranger system at TACC, the Kraken XT5 at NICS, and the DOE INCITE Intrepid Blue Gene/P at ANL. The media has widely covered work using this code, including the TeraShake and SCEC ShakeOut-D exercises, and recent Wall-to-Wall simulations which are some of the most detailed simulations to date of earthquakes along the San Andreas fault in recent years. The ShakeOut-D simulations, for example, were part of the Great California ShakeOut Exercise which attracted ten millions participants in 2009 to prepare for "The Big One". The most demanding simulation of a wall-to-wall rupture scenario modeled an M8.0 earthquake with an 800x400x100-km3 domain at 100-m resolution for up to four minutes. It required 32 billion volume elements and 86,000 time steps, taking 20 hours on 32K Ranger processor cores, generating 7 terabytes of surface output and checkpoints. Recent work by Cui et al. [6, 5], enhanced the application through single-processor optimizations, optimization of I/O handling and optimization of TeraShake initialization. In this work, we improve the application performance further by optimizing communication through latency hiding techniques.

AWM-Olsen solves the 3D velocity-stress wave equation explicitly by a staggered-grid FD method, fourth-order accurate in space and 2nd-order accurate in time. The code includes a coarse-grained implementation of the memory variables for a constant-Q solid and Q relations validated against data. The code uses Perfectly Matched Layers to implement absorbing boundary conditions on the sides and bottom of the grid, and a zero-stress free surface boundary condition at the top. The 3D volume representing the ground area to be modeled is decomposed into 3D sub-grids. Each processor is responsible for performing stress and velocity calculations for its assigned sub-grid, as well as applying boundary conditions at any external edges of the volume contained within its sub-grid. Ghost cells, comprising a two-cell-thick padding layer, manage the

most recently updated wavefield parameters exchanged from the edge of the neighboring sub-grids, see Figure 1 for a 2D schematic of these cells. The Ghost cell update presents a promising scenario to use one-sided communication for accelerated performance.

The main loop of the AWM Olsen is outlined in Figure 2. The velocity values are updated for the interior and the boundary of the volume. Then velocity values are exchanged with processors containing the neighboring sub-grids in the directions of north, south, east, west, up and down. This is followed by stress calculations and updates which are done in a similar manner. As the data grids are 3-dimensional, the boundary data along the non-major axes is a large number of small non-contiguous chunks. So, intermediate staging buffers are used to accumulate and disseminate data at the source and destination, respectively.

```
MAIN LOOP IN AWM-OLSEN
do i = timestep0, timestepN
    Compute Velocities - T_v
    Swap Velocity data with neighboring sub-grids - T_cv
    Compute Stresses - T_s
    Swap Stress data with neighboring sub-grids - T_cs
enddo
SWAP VELOCITY DATA
North and South Exchange
    s2n(u1,north-mpirank, south-mpirank) ! recv from south, send to north
    n2s(u1, south-mpirank, north-mpirank) ! send to south, recv from north
    ... repeat for velocity components v1,w1
East and West Exchange
    w2e(u1,west-mpirank, east-mpirank) ! recv from west, send to east
    e2w(u1, east-mpirank, west-mpirank) ! send to west, recv from east
    ... repeat for velocity components v1,w1
Up and Down Exchange
    ...
S2N
    Copy 2 planes of data from variable to send buffer !north face excluding ghost cells
    MPI_Isend(sendbuffer, north-mpirank)
    MPI_Irecv(recvbuffer, south-mpirank)
    MPI_Waitall()
    Copy 2 planes of data from recvbuffer to variable ! south ghost cells
```

Figure 2: AWM-Olsen Application Pseudo-Code

2.2 Overlapping Computation and Communication with MPI

Improving the communication and computation overlap is an important goal in parallel applications and there are different methods to achieve this.

Many applications are written with MPI blocking send-receive semantics due to its simplicity. However, the blocking primitives require the communication must complete before the process can proceed, so it cannot provide any overlap. Traditionally, MPI non-blocking primitives have

been used to alleviate this. Particularly, there is a good chance to have the computation and communication overlap by issuing multiple non-blocking send-receive calls followed by computation on independent data. Furthermore, if receive calls are posted before the corresponding send calls are posted, more overlap can be obtained. This is an active area of research, and several researchers have explored application-level re-design, impact of overlap coupled with re-designed protocols within the MPI library and improved designs of networking architectures [4, 18, 20, 11, 14, 23].

One-sided communication was introduced in MPI-2 [12], providing new opportunities for overlapping communication with computation. In the one-sided communication model, each process defines a memory region called a *window*, in its local address space for other processes to access. Data transfer happens through communication calls: MPI_Put copies data from the caller's (origin's) memory to the target's memory; MPI_Get transfers data from the target's memory to the caller's (origin's) memory; and MPI_Accumulate atomically updates the target's memory with the result of an operation on data from origin's and target's memory. In these operations, all parameters required for the transfer are specified at the origin and thus no involvement of the target is required. However, completion of the operations requires explicit synchronization calls.

The MPI one-sided model defines two modes for synchronization: active and passive. Active synchronization involves both the origin and the target processes and offers both collective and more restrictive semantics. MPI_Win_fence is an example of active synchronization with collective semantics which requires the participation of all processes in the communicator. MPI_Win_post-MPI_Win_wait/MPI_Win_start-MPI_Win_complete provides a restrictive mode of active synchronization which allows coordination among a subset of processes in the communicator. Passive synchronization involves locking and unlocking of remote windows requiring participation of only the origin process. Both shared and exclusive lock semantics are provided. Multiple communication operations (to distinct locations in a window) can be issued between processes before they synchronize to amortize the overhead of synchronization. The communication and synchronization operations provided in the MPI one-sided model are summarized in Figure 3. The one-sided model decouples the communication and synchronization, so it is non-blocking in nature and we can utilize it to achieve communication and computation overlap. For example, if the target can post MPI_Win_post beforehand, one-sided operations issued by the origin can go simultaneously with the following computation. This scenario is shown in Figure 4. Of course, this requires the support from network hardware which can move the data without interrupting the host processors using Remote Direct Memory Access (RDMA). Our target platform, TACC Ranger [22], has InfiniBand [1] which provides RDMA features.

Recently, there has been some work by application scientists to exploit the relatively newer MPI-2 semantics. Mirin, et al. [13] explore the use of MPI one-sided semantics coupled with multi-threading to optimize the Community Atmosphere Model. Hermmans et al. investigate the performance of one-sided communication alternatives in the NAS Parallel Benchmark BT application running on 256 cores of a Blue Gene/P in [8].

AWM-Olsen is characterized by nearest-neighbor communication. Each process exchanges velocity and stress boundary values and needs to synchronize with only its neighbors in the process grid. In such a scenario, MPI_Win_fence, because of its collective semantics, will result in an unnecessary synchronization of all processes, when only localized synchronization is required by the application. The Post-Wait/Start-Complete semantics suits this application because each pro-

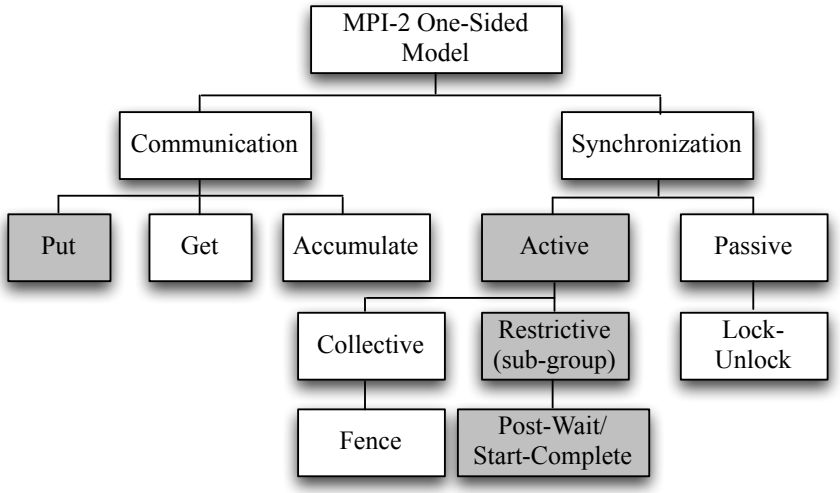


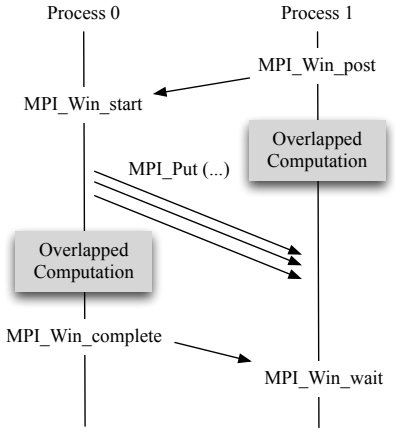
Figure 3: An Overview of MPI-2 Communication Model

cess can create a group including only its neighbors and post/access window based on this group. We note that passive lock-unlock provides point-to-point semantics, and can be potentially be used in this application scenario. In this work, we have chosen to use Post-Wait/Start-Complete semantics. Our justification is presented in Section 4.1. The MPI-2 one-sided calls utilized in our designs are highlighted in the grey boxes in Figure 3.

SOURCE - PROCESS 0
MPI_Win_start(group, 0, window)
MPI_Put(sendbuffer1, 1)
MPI_Put(sendbuffer2, 1)
. . .
Overlapped Computation
MPI_Win_complete(window)

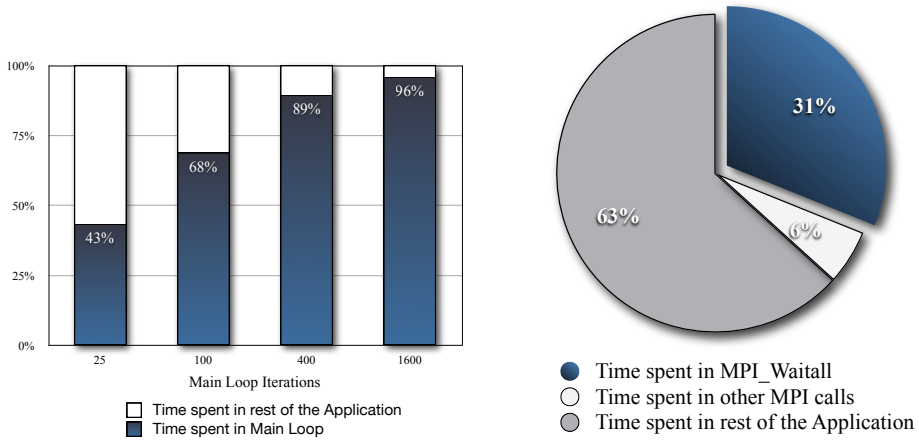
TARGET - PROCESS 1
MPI_Win_post(group, 0, window)
Overlapped Computation
MPI_Win_wait(window)

(a) MPI Pseudo-code for overlapping computation



(b) Timing Diagram

Figure 4: MPI-2 One-Sided communication using Post-Wait/Start-Complete Model



(a) Percentage of application time spent in the main loop

(b) Percentage of application time spent in MPI calls

Figure 5: Analysis of AWM-Olsen run-time

3 Performance Analysis of AWM-Olsen

In this section, we present an analysis of the AWM-Olsen application execution time and present our case that improving the communication design can considerably improve the application run-time. As discussed in Section 2, the AWM-Olsen application has a main loop where velocity and stress values are computed on each process and exchanged with its neighbors. As the complexity of the input data grid and the required accuracy increases, the number of iterations of the main loop increases. Figure 5(a) shows the time spent in the main loop as a percentage of the total application run-time on 4,096 processes. With 1,600 iterations, the main loop constitutes 96% of the total runtime. We used the mpiP profiling library [3] for this analysis. Real-world AWM-Olsen runs generally involve several thousands of main loop iterations with the largest ever involving 80,000 iterations. So the performance of the main loop code determines the overall application performance. In this paper, we aim at improving the performance of the data exchange in the main loop to improve overall application performance and scalability for large-scale runs.

3.1 Computation and Communication Breakdown

The main loop of the AWM-Olsen involves computation and exchange of velocity and stress. As shown in Figure 2, we refer to the time for velocity computation as T_v and velocity exchange/communication as T_{cv}. Similarly, the time for stress computation is referred to as T_s and the time for stress exchange is called T_{cs}. The total run-time of the main loop in the case of ideal overlap would be : max(T_v, T_{cv}) + max(T_s, T_{cs}). However, there are components in the communication times, such as the data copies from and to the staging buffers, that cannot be overlapped in reality. A detailed analysis of overlap with the actual times is presented in 5.4.

3.2 Communication Breakdown for Various MPI Calls

Figure 5(b) shows that AWM-Olsen spends 37% of its run time in MPI calls. The MPI_Waitall call, during which the communication progresses, contributes to 84% of the MPI time. This progress of communication can be overlapped with computation using RDMA capabilities of the underlying network thus improving the application performance.

4 Proposed Optimizations

4.1 Re-design Using Two-sided Non-blocking Primitives

The velocity and stress exchange routines account for most of the MPI time in the AWM-Olsen. Both velocity and stress have multiple components in them and each of these components corresponds to a data grid. During the exchange, each process sends the boundaries of these data grids to its neighbors in all directions and similarly receives boundary data from them. As each data grid is 3-dimensional, the boundaries along the non-major axes will be a large number of small non-contiguous chunks in memory. Each process accumulates these chunks into contiguous staging buffers before sending them to its neighbors. A corresponding dissemination happens at the receiving process.

In the original implementation, each of these exchanges is done in a blocking fashion i.e. one exchange is initiated only after the previous exchange has completed. Such an approach is required when there are data dependencies between the exchanges. We observe that the application does not require such strict synchronization. Both the velocity and stress computations are split in three and six *independent* components. Thus, there is an opportunity for an asynchronous design that overlaps computation of components and their communication.

Design enhancement #1: In our first design, we use non-blocking two-sided calls to initiate exchange of all components of velocity and then wait on the completion of all transfers at once using a MPI_Waitall. The high level organization of code is shown in figure 6. This allows for asynchronous progress. However enabling concurrent transfers requires additional staging buffers as one staging buffer can be tied to only one transfer. This does not cause any additional overhead in the critical execution path as these buffers are created only once and exist throughout the application run.

Design enhancement #2: The computation of the different components in velocity and stress is also independent of one another. Leveraging this, we overlap the exchange of a component with the calculation of the next thus creating computation-communication overlap. To enable this, at a higher level, we split the velocity computation and swap functions into component level granularity and interleaved them as follows: Computation and exchange of velocity are split into three parts based on its three components u, v and w. The value of u is computed and the exchange of u is overlapped with the computation of v. Similarly the exchange of v is overlapped with computation of w. For each component, we group the transfers to all neighbors to retain the advantage of concurrent progress explained above.

Though non-blocking two-sided semantics provide overlap with their underlying RDMA-based implementations, the skew created by the rendezvous handshake minimizes these benefits

VELOCITY EXCHANGE

s2n(u1,north-mpirank, south-mpirank) *! recv from south, send to north*
n2s(u1, south-mpirank, north-mpirank) *! send to south, recv from north*
. . . repeat for east-west, up-down directions and other velocity components v1,w1
wait_onedirection()
s2nfill(u1, recvbuffer, south-mpirank)
n2sfill(u1, recvbuffer, north-mpirank)
. . . repeat for east-west, up-down directions and other velocity components v1,w1

S2N

Copy 2 planes of data from variable to sendbuffer *!north face excluding ghost cells*
MPI_Isend(sendbuffer, north-mpirank)
MPI_Irecv(recvbuffer, south-mpirank)

WAIT_ONEDIRECTION

MPI_Waitall(list of receive requests)

S2NFILL

Copy 2 planes of data from recvbuffer to variable *! south ghost cells*

Figure 6: Velocity Exchange with Reduced Synchronization using Non-blocking Two-Sided Semantics

(since these are larger messages, typically of size 64K based on our problem size). To overcome this limitation, we explored the use of one-sided semantics to further improve overlap. We describe our design using one-sided semantics in the next sub-section.

4.2 Re-design Using MPI-2 One-sided Primitives

MPI-2 one-sided semantics are closely aligned with the network-level RDMA operations, thus, are expected to provide better communication-computation overlap. MPI-2 provides three different kinds of synchronization operations in its one-sided model and choosing the one that suites the target application is crucial for good performance. As described earlier, AWM-Olsen has a very localized communication, requiring each process to synchronize with only its nearest neighbors. A collective synchronization like that provided in MPI_Win_fence will lead to unnecessary overhead that increases with the number of processes. The Lock-Unlock semantics require processes to lock and unlock windows of each of its neighbors independently causing an overhead and reducing promise for concurrent communication progress.

We use the MVAPICH2 [15] MPI library available on our target platform (TACC Ranger) for the experiments. MVAPICH2 implements several different modes and mechanisms for high-performance one-sided communication as described in [10, 9, 19]. However, to the best of our knowledge, at the time of our design and implementation, MVAPICH2 did not implement Lock/Unlock semantics using RDMA primitives. The lack of direct one-sided implementation of Lock-Unlock make it less suitable for overlap. Thus, in this paper we design our overlapping

mechanisms using Post-Wait/Start-Complete semantics. The Post-Wait/Start-Complete model, with its group based semantics fits well with the nearest-neighbor communication pattern. The neighbors of each process form its Post and Start group. The MPI-2 one-sided calls utilized in our designs are highlighted in the grey boxes in Figure 3.

The receive staging buffers form the windows at each process. As window registration is an expensive collective operation, we remove this from the critical execution path by registering windows only once at the start of the application and retaining them until the end. With Post-Wait/Start-Complete semantics, the timing of posting the window for one-sided operations is crucial to achieve maximum performance and overlap. For example, if an MPI_Put is issued at a process before the window is posted by the target, the actual transfer is delayed until the first MPI call after the window is posted or even until the following synchronization where there are no following MPI_Put calls. This can reduce or totally negate the opportunity for overlap. This issue becomes more prevalent due to skew as the process count increases. To avoid this, in the AWM-Olsen code, we post the windows before the main loop for the first iteration and at the end of each iteration for the following one.

The high-level organization of code is similar to our Isend/Irecv based implementations and is given in Figure 7. The computation and communication of different components in stress and velocity are overlapped. Transfers to all neighbors are grouped into one synchronization epoch for each component thus allowing concurrent communication progress.

5 Experimental Results

5.1 Experimental Setup

We have run all of our experiments on the TACC Ranger system. Ranger is a blade-based system. Each node is a SunBlade x6420 running a 2.6.18.8 Linux kernel. Each node contains four AMD Opteron Quad-Core 64-bit processors (16 cores in all) on a single board, as an SMP unit. Each node has 32 GB of memory. The nodes are connected with InfiniBand SDR adapter from Mellanox [2]. In our experiments, we used MVAPICH2 1.4 [15] MPI library installed on the Ranger. We use a weak scaling model for our experiments to simulate the real world application use. We increase the size of the data set as the process count increases such that the data grid size per process remains at 64x64x64 elements. The message size, which is the size of the ghost cells in each direction, will be 64x64x2 real values i.e. 64K.

5.2 Reduced synchronization using two-sided non-blocking

In this section we compare the performance of the original AWM-Olsen application with the Async-2sided-basic and Async-2sided-advanced versions. Async-2sided-basic is the version described in Section 4.1 as design enhancement 1 and Async-2sided-advanced is the version redesigned for communication-computation overlap (design enhancement 2). Figure 8(L) shows the main-loop execution times of a 25 time-step application run for all the versions. Figure 8(R) shows the percentage improvements of the enhanced versions over the original version. In the performance analysis, we skip the first iteration from the timing to avoid the impact of startup

MPI_Win_post(group, 0, window) *! pre-posting the window to all neighbors*
MAIN LOOP IN AWM-OLSEN
Compute velocity component u
Start exchanging velocity component u
Compute velocity component v
Start exchanging velocity component v
Compute velocity component w
Start exchanging velocity component w
Complete Exchanges of u,v and w
MPI_Win_post(group, 0, window) *! For the next iteration*
START EXCHANGE
MPI_Win_start(group, 0, window)
s2n(u1,north-mpirank, south-mpirank) *! recv from south, send to north*
n2s(u1, south-mpirank, north-mpirank) *! send to south, recv from north*
. . . repeat for east-west and up-down
COMPLETE EXCHANGE
MPI_Win_wait(window)
MPI_Win_complete(window)
s2nfill(u1, window buffer, south-mpirank)
n2sfill(u1, window buffer, north-mpirank)
. . . repeat for east-west and up-down

S2N
Copy 2 planes of data from variable to sendbuffer *!north face excluding ghost cells*
MPI_Put(sendbuffer, north-mpirank)

S2NFILL
Copy 2 planes of data from window buffer to variable *! south ghost cells*

Figure 7: Velocity Exchange with Reduced Synchronization using Post-Wait/Start-Complete

costs. We see an 8% improvement with the Async-2sided-basic version over the original version on 4,096 processor cores. With the Async-2sided-advanced version, we see an improvement of 11% on 4,096 processors cores and 6% improvement on 8,192 cores.

5.3 Maximized overlap using MPI2 one-sided primitives

We refer to our design of AWM-Olsen based on the MPI-2 one-sided semantics as Async-1sided (Section 4.2). The high-level design of overlap is similar to the Async-2sided-advanced version but we see better performance because of the closer alignment of one-sided semantics with the underlying network RDMA operations. The advantage of overlap in Isend/Irecv model is hampered by the rendezvous handshake for large messages. With the async-1sided version we see upto 12% improvement over the original version on 4,096 processor cores and 10% improvement on 8,192 cores. The results are charted in Figure 8.

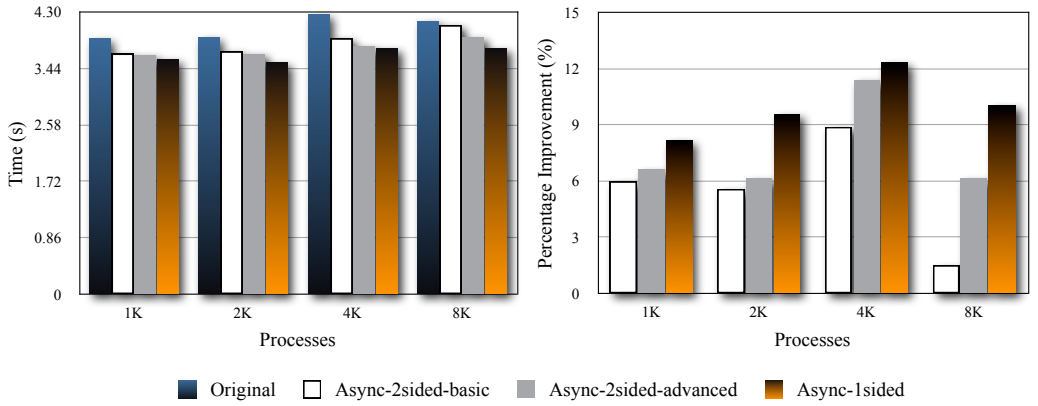


Figure 8: AWM-Olsen main loop optimizations (L) actual numbers (R) percentage improvement

5.4 Analysis of Achieved Overlap

In this section we analyze the run-time of AWM-Olsen in depth to better understand how much percentage of application time was actually overlapped using our designs. Velocity computation (T_v), velocity exchange (T_{cv}), stress computation (T_s) and stress exchange (T_{cs}) form the major part of the main loop in AWM-Olsen. In case of ideal overlap, the total cost of the main loop would be max(T_v, T_{cv}) + max(T_s, T_{cs}). As shown in the table below, this time for a 25 iteration, 4,096 process run should be 3.00s. There are parts in T_{cv} and T_{cs} that cannot be overlapped for practical reasons. First, the copy of data from and to the staging buffers involves the CPU and cannot be overlapped. This copy time was measured as 0.11s during this run. Also, due to data-dependencies in the application, the communication of the last components in velocity and stress cannot be overlapped. We need to synchronize, i.e. call MPI_Win_complete for the transfers to complete. This is due to data-dependency in the application. As velocity has 3 components and stress has 6 components, these times in velocity and stress would be $1/3T_{cv}$ and $1/6T_{cs}$, respectively. We have run our experiments on the TACC Ranger system which is the cluster used

for many of the very large real world AWM-Olsen runs. Due to the limitations of setup on the cluster, intra-node transfers cannot use IOAT [21] or similar techniques and hence do not provide an opportunity for computation-communication overlap. In the 4,096(16x16x16) process grid with 16 cores per node on Ranger, each process has its neighbors along the z-axis within the same node. This results in $(T_{cv} + T_{cs})/3$ being intra-node communication. Accounting for these components, the expected main loop time computes to 3.76s. Our actual runtime for the main loop, 3.75s, matches this expected value, getting 100% of the achievable overlap.

Timings from Original Main Loop				
T_v	T_{cv}	T_s	T_{cs}	Total Loop Time
0.40s	0.55s	2.45s	0.76s	4.16s
Theoretical Min Loop Time [100% overlap] (max(T_v, T_{cv}) + max(T_s, T_{cs})) = 3.00s				
Practically Non-overlappable Communication Components				
Last velocity component (T_{lv} = $1/3T_{cv}$)				0.18s
Last stress component (T_{ls} = $1/6T_{cs}$)				0.13s
Intra-node communication ($(T_{cv} + T_{cs})/3 - (T_{lv} + T_{ls})/3$)				0.34s
Copy costs for non-contiguous communication (measured)				0.11s
Total non-overlappable part				0.76s
Expected Main Loop Time Overlapping all Network Communication				3.00s + 0.76s = **3.76s**
Measured Main Loop Time				**3.75s**

6 Conclusion and Future work

AWM-Olsen is a widely used earthquake-induced ground wave-propagation simulation code which consumes several million CPU hours every year on the TeraGrid clusters. Efficient communication design is paramount for such large scale, heavily used applications to best utilize the available system time and resources. In this paper, we first analyzed the communication in AWM-Olsen, showing the potential for communication-computation overlap. Then we re-designed the application using MPI-2 one-sided semantics to achieve the expected overlap. We see a 12% improvement in overall application performance on 4,096 cores.

As future work, we would like to the explore the use of asynchronous progress for non-blocking two-sided operations and its impact on communication-computation overlap. We also plan to evaluate the potential for application-level overlap with use of IOAT-like technologies for intra-node communication. Based on our experience with AWM-Olsen, we aim to develop parallel programming patterns which can be used by application writers to achieve similar overlap benefits without requiring a deep understanding of the MPI-2 one-sided semantics.

References

[1] InfiniBand Trade Association. `http://www.infinibandta.org`.

[2] Mellanox Technologies. `http://www.mellanox.com`.

[3] mpiP: Lightweight, Scalable MPI Profiling. http://mpip.sourceforge.net/.

[4] Ron Brightwell and Keith D. Underwood. An Analysis of the Impact of MPI Overlap and Independent Progress. In *Proceedings of the 18th annual international conference on Supercomputing*, March 2004.

[5] Yifeng Cui, Reagan Moore, Kim Olsen, Amit Chourasia, Philip Maechling, Bernard Minster, Steven Day, Yuanfang Hu, Jing Zhu, and Thomas Jordan. Toward Petascale Earthquake Simulations. In *Acta Geotechnica*, pages 79–93, 2009.

[6] Yifeng Cui, Reagan Moore, Kim Olsen, Amit Chourasia, Philip Maechling, Bernard Minster, Steven Day, Yuanfang Hu, Jing Zhu, Amitava Majumdar, and Thomas Jordan. Enabling Very-Large Scale Earthquake Simulations on Parallel Machines. In *ICCS '07: Proceedings of the 7th international conference on Computational Science, Part I*, pages 46–53, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] Yifeng Cui, Kim Olsen, Yuanfang Hu, Steven Day, Luis Dalguer, Bernard Minster, Reagan Moore, Jing Zhu, Philip Maechling, and Thomas Jordan. Optimization and Scalability of an Large-scale Earthquake Simulation Application. In *American Geophysical Union, Fall Meeting*, 2006.

[8] Marc-André Hermanns, Markus Geimer, Bernd Mohr, and Felix Wolf. Scalable Detection of MPI-2 Remote Memory Access Inefficiency Patterns. In *16th Euro PVM/MPI, Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[9] Wei Huang, Gopalakrishnan Santhanaraman, Hyun-Wook Jin, and Dhabaleswar K. Panda. Scheduling of MPI-2 One Sided Operations over InfiniBand. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 9*, page 215.1, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, K. Dhabaleswar Panda, William Gropp, and Rajeev Thakur. High Performance MPI-2 One-sided Communication over InfiniBand. In *CCGRID '04: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*, pages 531–538, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Rahul Kumar, Amith R. Mamidala, Matthew J. Koop, Gopal Santhanaraman, and Dhabaleswar K. Panda. Lock-free Asynchronous Rendezvous Design for MPI Point-to-point communication. In *EuroPVM '08*, 2008.

[12] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, Jul 1997.

[13] Arthur A. Mirin and William B. Sawyer. A Scalable Implementation of a Finite-Volume Dynamical Core in the Community Atmosphere Model. *Intl. Journal of High Performance Computing Applications*, 19(3):203–212, 2005.

[14] Myricom. Myrinet overview. `http://www.myri.com/myrinet/overview/`.

[15] Network-Based Computing Laboratory. MVAPICH/MVAPICH2: MPI over InfiniBand and 10GigE/iWARP. `http://mvapich.cse.ohio-state.edu/`.

[16] Kim Olsen. Simulation of three-dimensional wave propagation in the Salt Lake Basin. Technical report, University of Utah, Salt Lake City, Utah, 1994.

[17] Kim Olsen, Steven Day, Bernard Minster, Yifeng Cui, Amit Chourasia, Marcio Faerman, Reagan Moore, Philip Maechling, and Thomas Jordan. Strong Shaking in Los Angeles Expected from Southern San Andreas Earthquake. In *Geophysical Research Letters, Vol 3*, pages 1–4, 2006.

[18] José Carlos Sancho, Kevin J. Barker, Darren J. Kerbyson, and Kei Davis. Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications. In *SC '06: Proceedings of ACM/IEEE Super Computing*, 2006.

[19] Gopalakrishnan Santhanaraman, Pavan Balaji, Karthik Gopalakrishnan, Rajeev Thakur, William Gropp, and Dhabaleswar K. Panda. Natively Supporting True One-Sided Communication in MPI on Multi-core Systems with InfiniBand. In *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 380–387, Washington, DC, USA, 2009. IEEE Computer Society.

[20] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K. Panda. RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. In *Symposium on PPOPP* , March 2006.

[21] Intel I/O Acceleration Technology. http://www.intel.com/technology/ioacceleration/index.

[22] Texas Advanced Computing Center. 62,976-core Ranger Cluster. `http://www.tacc.utexas.edu/resources/hpcsystems/` .

[23] Francois Trahay, Elisabeth Brunet, Alexandre Denis, and Raymond Namyst. A Multithreaded Communication Engine for Multicore Architectures. In *International Parallel and Distributed Processing (IPDPS)*, 2008.