Improving the Performance of the PSDNS Pseudo-Spectral Turbulence Application on Blue Waters using Coarray Fortran and Task Placement

Robert A. Fiedler, Nathan Wichmann, Stephen Whalen
Cray, Inc.
St. Paul, MN, USA
rfiedler@cray.com

Dmitry Pekurovsky
University of California, San Diego
San Diego Supercomputing Center
La Jolla, CA, USA

***Abstract*—The PSDNS turbulence application performs many 3D FFTs per time step, which entail frequently transposing distributed 3D arrays. These transposes are achieved via multiple concurrent All-to-All communication operations, which dominate the overall execution time at large scales. We improve the All-to-All times for benchmarks on 3072 to 12288 nodes using three main strategies: 1) eliminating off-node communication for one of the two sets of transposes by assigning one sheet of the 3D Cartesian grid to each node (35% speedup), 2) placing tasks on nodes that are distributed randomly throughout the gemini network in order to maximize the All-to-All bandwidth that can be utilized by the job's nodes (21% speedup), and 3) reducing contention and overhead by replacing calls to MPI_AlltoAll with a drop-in library written in Coarray Fortran (33% speedup). We also describe how this library is implemented and integrated efficiently in PSDNS.**

Keywords—3D FFTs, transposes, All-to-All communication optimization, CAF

I. INTRODUCTION

The National Science Foundation's request for proposals for a sustained petascale supercomputer (now known as Blue Waters) included an acceptance benchmark for direct numerical simulation (DNS) of homogeneous isotropic turbulence in which the pseudo-spectral method is to be used on a domain with 12288 grid points in each of 3 dimensions [1]. This paper describes how we improved the performance of the PSDNS turbulence application [2] that we selected for this benchmark by a factor of 2.8 from its initial value on large Cray XE6/XK7 systems with gemini interconnection networks [3].

Pseudo-spectral methods based on Fourier or orthogonal polynomial expansions are well known for accuracy and efficiency in solving partial differential equations [4, 5]. In simulating systems in which a wide range of physical scales is present, such as Direct Numerical Simulation (DNS) of Turbulence, pseudo-spectral methods are far superior to those relying on approximations based on finite differences, finite volumes, or interpolation. DNS studies aim to solve the Navier-Stokes equations without approximation, and pseudo-spectral algorithms are very popular in DNS codes. Time integration is usually done by an explicit scheme in the transformed space, where the solution variables at different modes are formally decoupled, allowing each processor to operate on its own data independently. However, pseudo-spectral methods require the evaluation of many discrete forward and inverse multidimensional Fast Fourier Transforms (FFTs) of the field variables per time step, which entails far more communication than most other methods, especially those with primarily nearest-neighbor communication patterns [6].

Multidimensional FFTs are performed efficiently as successive one-dimensional FFTs along each dimension of the computational grid. For 3D flows, the domain is often decomposed into either 1D slabs or 2D pencils (i.e., bundles of lines of grid points having rectangular cross section that span the entire domain along one dimension; see Fig. 1) so that 1D FFTs along different lines can be computed by each task concurrently using a serial algorithm [7]. After the first set of 1D FFTs (along the x direction) is completed, the 3D global field variable arrays must be transposed in xy planes so that each task owns a pencil that lies along the y dimension. Thereafter, the second set of 1D FFTs can be computed. Next, the field variable arrays are transposed in yz planes so that each task owns a pencil along the z dimension and the third set of 1D FFTs can be completed.

PSDNS is used for numerical studies of turbulence and turbulent mixing [8]. The code is written in Fortran and uses the hybrid OpenMP/MPI parallel programming model. It solves the Navier-Stokes equations using the pseudo-spectral method, and has many capabilities for scientific discovery in addition to those required for the petascale benchmark. The benchmark also specifies the use of a fourth-order Runge-Kutta explicit time-stepping scheme and double-precision computations, which are available options in PSDNS.

A. Performance Model

A simple performance model of an application can be very valuable in understanding where the time is spent, and in identifying aspects of performance that appear to fall below expectations.

For a typical time step, the PSDNS execution time can be broken down into 2 main components:

1) Computational work that involves no communication, including 1D FFTs, evaluating terms in the equations of motion and updating the field variables, and

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

packing/unpacking send/receive buffers for the transposes.

2) Communication for the transposes, not including packing/unpacking buffers.

On most existing large systems, for typical physical problem sizes the communication time is longer than the computation time. We have ignored the time taken for any IO, since little IO occurs during a typical time step. Most of the data that PSDNS writes to disk consists of checkpoint files, which are usually created only once every 50-200 typical time steps.

There are 2 distinct sets of transposes, which we refer to as xy and yz. The xy transposes are confined to xy planes in the 3D computational grid, and the yz transposes are confined to yz planes. For the xy transposes, a separate communicator is created for each row of pencils as shown in Fig. 1, and each task belongs to only one of these “row” communicators. An All-to-All communication call is invoked for each row communicator simultaneously. Thus, the communication pattern consists of multiple concurrent All-to-Alls on disjoint sets of tasks.

The yz transposes are treated in a manner analogous to the xy transposes, but the yz transposes are confined to columns of pencils instead of rows. The number of tasks in the column communicators is typically much larger than the number of tasks in the row communicators, which is often set equal to the number of tasks per node (as discussed in detail below).

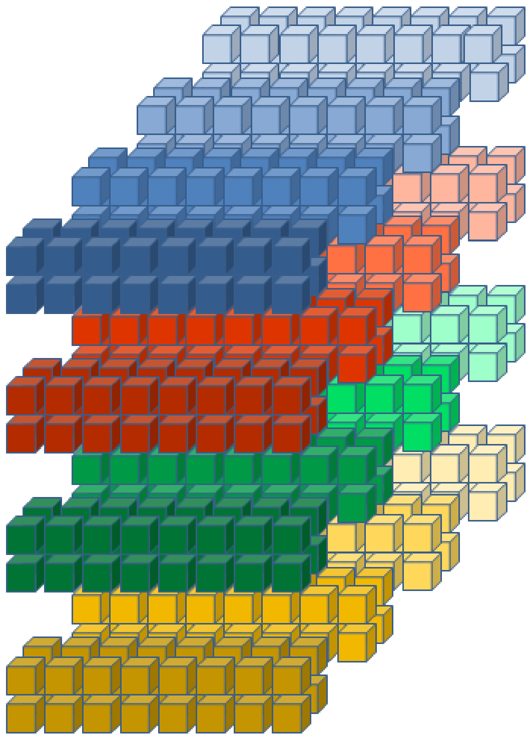
Fig. 1. Example PSDNS domain decomposition with 4 rows of pencils containing 8x2x2 grid cells. Each row communicator contains 4 pencils of similar color.

The message sizes for the All-to-Alls determines whether or not network latency is significant. Calculating message sizes for PSDNS is complicated somewhat by the fact that the implementation uses a cylindrical cutoff radius in the spectral space (equal to $2^{(1/2)}/3$ times the domain size). This creates a load imbalance in the column transpose. While message sizes are close to uniform within each column communicator, their values vary across different columns. Since the All-to-All transposes within columns happen simultaneously, columns with smaller message sizes complete their communication sooner than do those with larger messages. This load imbalance is handled by placing tasks with both small and large workloads on the same node, which balances out the differences since the tasks share the memory and interconnect bandwidths of their assigned nodes. Consequently, there is a range of message sizes. If V is the size in Bytes of a 3D global data array, N is the total number of tasks, and M is the number of members in the column communicator, the largest message size can be found from

$$V/(N*M)$$

For a problem domain with 12288^3 grid points and double-precision data running on 12288 nodes, if we use 16 tasks per node, the message sizes are:

$$12288^3 * 8 \text{ Bytes} / ((12288*16) * 12288) = 6144 \text{ Bytes.}$$

PSDNS optionally combines messages for several field variables/components (up to 5 for the petascale benchmark). If this option is used, messages can be as large as 30 kB, but 6 kB is already large enough that latency contributes little to the communication time. While the above message sizes are the maximum values, most of the other message sizes are a significant fraction thereof. Even though the smallest message sizes can be sensitive to latency, such messages are not expected to comprise a significant portion of the communication time. Thus, our model assumes that bandwidth (and not latency) determines communication performance.

Although the code optionally uses the hybrid OpenMP/MPI model, because the messages are rather large there is relatively little to be gained in terms of latency reduction when multiple OpenMP threads are used, and experiments on a dedicated system at large scales have validated this expectation.

B. *Gemini interconnect*

Applying the performance model described above requires an understanding of the nodes and interconnect in the platform on which the application runs. Here we describe the key aspects of the Blue Waters Cray XE6/XK7 system [9].

The Blue Waters interconnect is a 3D torus with 23 gemini [3] routers in the x direction, 24 in y, and 24 in z. Two nodes are attached to each gemini. There are 3072 XK7 compute nodes with 8x8x24 geminis embedded in this fabric. Each XK7 node has one 2.3 GHz AMD Interlagos processor with 8 “Bulldozer” compute units and one nVidia Kepler GPU. The rest of the ~22752 compute nodes are XE6, each with two Interlagos processors like the one in the XK7 nodes. There are also ~672 service nodes in various locations throughout the torus that perform functions such as IO, job launching, etc. The service nodes are not directly allocated to user jobs, but they do relay messages between compute nodes on behalf of user jobs.

User jobs are assigned on a per-node basis, i.e., different batch jobs do not share nodes. For optimum performance, it is best if a given job is running on both nodes attached to each gemini in the batch job reservation. If one of the nodes on a gemini is down or assigned to another job, a communication load imbalance may result. A user can avoid this situation by requesting more nodes than are needed to run the job and using our node-list randomization script (described below) or other means to avoid actually running on such nodes.

Although the two nodes attached to a given gemini use the router to exchange messages, that traffic does not traverse the links between geminis, and therefore is not considered to use the interconnect. We estimate that the application-realizable All-to-All bandwidth between same-gemini nodes ~12 GB/s, which is higher than the bandwidth of any of the individual links to neighboring nodes.

Links in the x direction of the torus consist of cables connecting different rows of cabinets. The z direction runs across the 24 boards in any given cabinet. Each cabinet has 3 cages containing 8 boards on a backplane with higher bandwidth than the cables connecting the backplanes together. These cables are the same capacity as those used for the x direction, and they determine the effective bandwidth across any set of nodes that spans more than one cage. In the y direction, the connections between different boards have only half the bandwidth of the cables used in the x and z directions. There are two geminis on each board, and the bandwidth between geminis on the same board is much higher than the bandwidth along y between boards.

Below we show how one can take advantage of the faster links in the x and z directions through careful node selection, in order to maximize the effective bisection bandwidth available to PSDNS for a given node count.

II. OPTIMIZATION STRATEGIES

A. *Optimizing the computation time*

Although we know that for PSDNS the computation time is typically less than the communication time, it makes sense to examine the performance profile to determine whether some simple changes to the code, or how the executable is built and run, might lead to substantial reductions in the computation time.

The computation time includes not only packing and unpacking send/receive buffers, but also the time spent allocating and deallocating them whenever the transpose routines are called. When the PGI or gnu compilers are used, a speedup of the computation time by up to ~20% for typical problem sizes and node counts can be obtained at the cost of using somewhat more memory by setting the environment variables [10]:

```
MALLOC_MMAP_MAX_ = 0
MALLOC_TRIM_THRESHOLD_ = 536870912
```

Setting MALLOC_MMAP_MAX_ to 0 prevents a program from using the system's mmap routine and ensures most memory is allocated from the heap. The MALLOC_TRIM_THRESHOLD_ variable specifies the amount of free space at the top of the heap (after memory is freed) that is required for malloc to return the memory to the operating system. The default value of 128 kB can result in excessive overhead through frequent system calls for memory management.

For a domain with $8k^3$ grid points on 4k XE nodes (16 tasks per node on the first socket only), the total run time (including both computation and communication) was reduced by 1.13X.

When the Cray compiler is used, the values of these environment variables are set as above by default. All of the other improvements described in this paper pertain to reducing the communication time.

B. *Improving the communication time*

1) Domain Decomposition

For typical physical problem sizes on systems having nodes with sufficiently large numbers of cores and amounts of memory, it is possible to assign one or more xy planes of the grid to a single node. Doing so eliminates all off-node communication for the xy transposes. This should be advantageous on any platform for which the communication time is dominated by effective bisection bandwidth, rather than message injection bandwidth, since it reduces the volume of data that must be transferred over the interconnect.

Although PSDNS uses a 2D-pencil decomposition rather than a slab decomposition, one can obtain a "slab-on-node" decomposition without any source code changes. In a PSDNS input file, one specifies the numbers of tasks in the row and column communicators. If the number of row communicator tasks is set to the number of cores per node, and the number of column communicator tasks is set to the number of nodes, then each node will get one complete xy plane of the grid, and each column communicator will have just one task on each node. If there are an integer multiple fewer nodes than there are xy planes in the grid, then multiple xy planes are assigned to each node.

Table 1 presents results for a 6144^3 grid running on 3072 XE nodes (using 16 tasks per node on only the first socket as though they were XK nodes) of a dedicated system (an early portion of Blue Waters with 4512 XE nodes) for different domain decompositions. For the 16x3072 decomposition, 2 complete xy planes of the grid are assigned to each node, and off-node communication is minimized. For the 24x2048 decomposition, each set of 3 xy planes are divided into 24 pencils, which means one node gets 2/3 of the pencils, and the other 1/3 are placed on another node, often one that is on another gemini. Consequently, considerable communication for the xy transposes must go off node and total run times are 1.47X longer than the 16x3072 decomposition.

For the 32x1536 decomposition, each node gets half of the pencils in a set of 4 xy planes, even more of the communication must go off node, and the total run time is 1.72X longer than 16x3072. In the (16+16)x1536 case, we ensured that the two sets of 4 pencils are placed onto the two nodes attached to the same gemini, which eliminates communication between geminis for xy planes and reduces the total run time to only 1.10X longer than that of the 16x3072 case. Randomizing the node list as described later was also applied and improved the bisection bandwidth per node, contributing to the 1.57X total run time performance improvement over the 32x1536 case.

TABLE I. EFFECT OF DOMAIN DECOMPOSITION

Decomposition (row tasks x column tasks)	*Time per step ratio*
16x3072	1.00
24x2048	1.47
32x1536	1.72
(16+16)x1536	1.10

2) *Bisection bandwidth and node selection*

The time for the off-gemini All-to-All communication required to complete yz transposes of the 3D global field variables is inversely proportional to the effective bisection bandwidth provided by the geminis directly involved in the job, as well as any additional geminis that relay messages on behalf of the job. The contribution to the effective bisection bandwidth from geminis outside of the job's allocation can be significant.

The bisection bandwidth of a system is defined as being proportional to the lowest possible value of the capacity of the links crossing any plane bisecting the interconnect. We can represent the torus as a cube as shown in Fig. 2, but note that because it is a torus rather than a mesh, any cut-plane will bisect the links along any dimension two times, not just once. For the Blue Waters system with topology 23x24x24, the cut-plane with the least bandwidth intersects the y-link cables between any 2 boards, and therefore the bisection bandwidth is proportional to 2 times the number of y-links times their per-link capacity, B_y, which works out to be equal to 23*24*2*B_y, or 23*24*B_x, where B_x is the capacity of an x-link cable.

For the petascale benchmark, it would be best to run on nearly all compute nodes to directly utilize as many geminis as possible for maximum bisection bandwidth. Blue Waters has more than 24k compute nodes, but 3072 of them are XK nodes, and PSDNS is not designed to use the GPU in such nodes. We could use only the first socket in the XK and XE nodes in order to run PSDNS on 24k nodes (without changing the source code to somehow handle nodes with different core counts in the same job). Using only the first socket would leave half of the processors on the XE nodes idle, but since the communication time dominates, and the communication time depends on the number of geminis in use, running on one socket per node could be expected to provide the best overall run times. To do so, each xy plane of the $12k^3$ domain would be assigned to one node pair, as discussed in the previous section, eliminating communication between geminis for the xy transposes.

In actuality, the petascale benchmark was run on 12k XE nodes using both sockets per node. With judicious node selection and task placement we were able to obtain overall per-step run times only 1.09X longer on 12k nodes than on 24k nodes, and we also recognized that using 2X fewer nodes should translate into 2X more time between node failures. The Blue Waters petascale benchmark was interpreted to include setbacks due to node failures (i.e., the time to restart from the last checkpoint and repeat any steps taken beyond that point), so the number of such failures had to be minimized. Our analysis and efforts described below to complete the benchmark as quickly as possible on only about half of the nodes in the system are probably more relevant to most of the user community than are full-system runs.

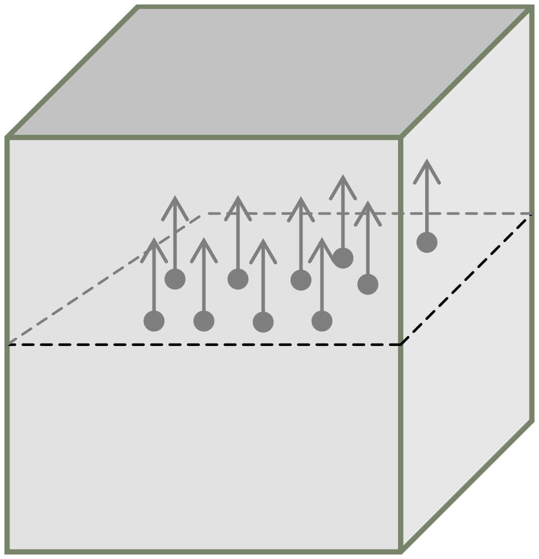
Fig. 2. Schematic of bisection bandwidth calculation for the full Blue Waters system.

To maximize the effective bisection bandwidth available to a subset of the nodes in the system, consider a section of Blue Waters containing just 6 xz planes of the torus as shown in Fig. 3. Since routing always takes the shortest path, messages never go outside the 23x6x24 section, i.e., the "topology" for this section of the system is a mesh in the y direction and a torus in x and z. Comparing the bandwidth of the links crossing cut-planes normal to x, y, and z, we find that the bisection bandwidth is proportional to 23*6*2*B_z, where $B_z = B_x$ is the capacity of the cables along z. This is one half of the bisection bandwidth of the entire system (assuming the constants of proportionality are the same), but involves only 1/4th of the compute nodes. Thus, a 23x6x24-gemini section delivers twice the bisection bandwidth per node as the full system.

If a user is charged on the basis of node-hours consumed, then the most economical way to run a job requiring a fixed number of nodes would be to obtain a node allocation that maximizes the effective bisection bandwidth per node. It turns out that 23xNx24 gives the highest bisection bandwidth per node for $1 < N < 7$. Increasing N beyond 6 adds nothing to the bisection bandwidth, and therefore the bisection bandwidth per node drops in proportion to the number of nodes until N=12, when it equals the value for the full system. Beyond N=12, some traffic wraps around the torus in the y direction, but the bisection bandwidth per node has a minimum for N = 13. For 12 < N < 24. Because only N out of 24 nodes along y are actively used in the computation, we find (by counting messages in each direction) that the bisection bandwidth is proportional to:

$$23*24*2*(N-1)/(24-1)*B_y.$$

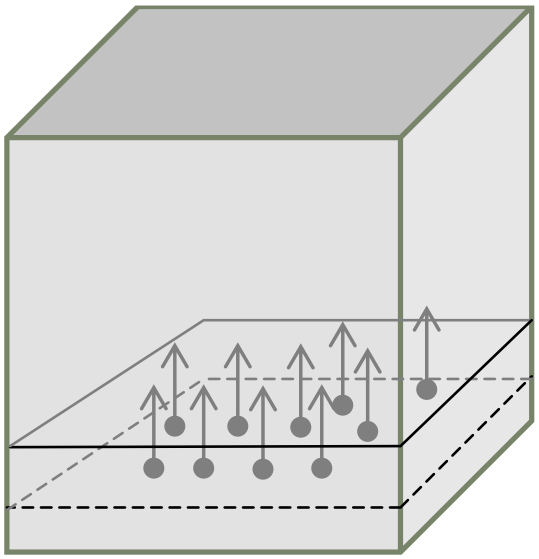

Fig. 3. Schematic of bisection bandwidth calculation for a 23x6x24-hub section of Blue Waters.

If we were to run the petascale benchmark in a 23x12x24 section, the bisection bandwidth (not per node) would be half that of the full system, and we would see 2X longer communication times than we would obtain on 24k nodes. The default placement for 12k nodes is close to 12x24x24, which gives the same bisection bandwidth per node as does 23x12x24. Due to the 8x8x24-gemini XK region on Blue Waters, default placement for 12k XE nodes gives the layout shown in Fig. 4.

Because we could choose any 12k nodes in the system for the benchmark, we were able to utilize nearly the full system bisection bandwidth by distributing the 12k nodes actively running the benchmark as much as possible throughout the torus. This was done by means of a perl script (randomizer.pl) that randomizes the list of nodes in the reservation obtained, for example, by using the command:

```
aprun -B -D0x10000 /bin/true | head -1
```

within the batch job. The randomized list is optionally grouped in pairs so that 2 adjacent sets of tasks are placed on the two nodes attached to the same gemini. This option also eliminates from the list any nodes whose partner on the same gemini is unavailable. Using nodes with an unavailable partner could introduce a load imbalance, and if xy planes of the grid are to be split between nodes on the same gemini, it would also prevent us from placing those neighboring groups of tasks as intended.

The aprun command to launch PSDNS is issued with either the "-l <node_list_file_name>" or the "-L <node_list_string>"

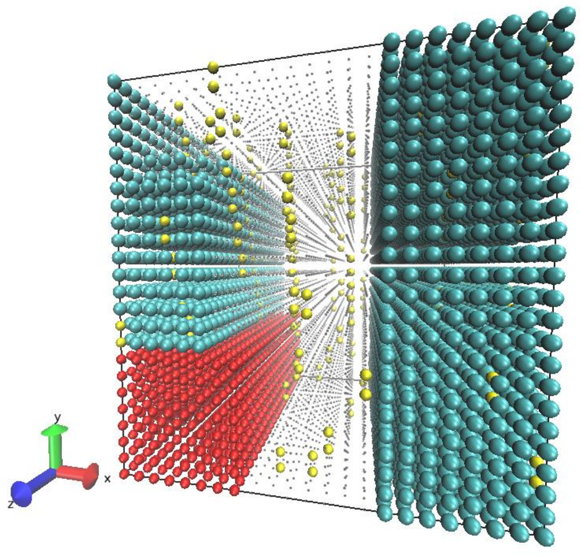

Fig. 4. Hubs used (blue spheres) for a 12k^3 grid on 12k XE nodes with default placement. Service node hubs are represented by yellow spheres and XK node hubs are marked by red spheres.

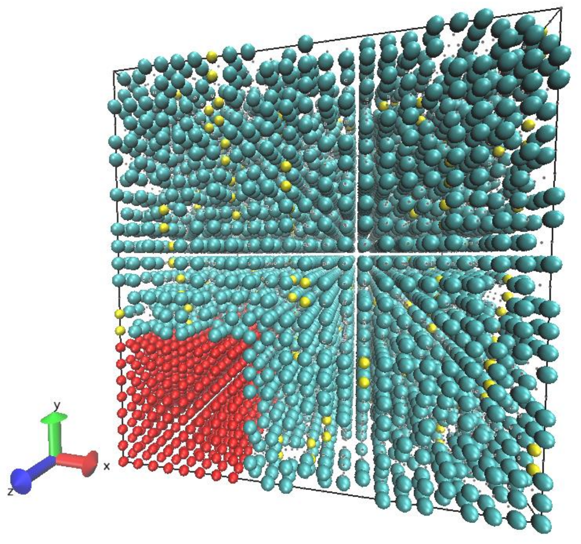

Fig. 5. Hubs used (blue spheres) for a 12k^3 grid on 12k XE nodes selected using the randomizer.pl script. Spreading the active nodes throughout the torus enables the job to utilize nearly the full bisection bandwidth of the full system.

option to use the first 12k nodes in the randomized list. Note that our batch job reservation contained nearly all compute nodes in the system, allowing us to select whichever nodes we wished to use for the benchmark. The 12k randomized XE compute nodes used by our job are depicted in Fig. 5. For the pure MPI version of PSDNS, the total time per time step for the randomized 12k node list was 1.46X shorter than the corresponding time for the default layout in Fig. 4.

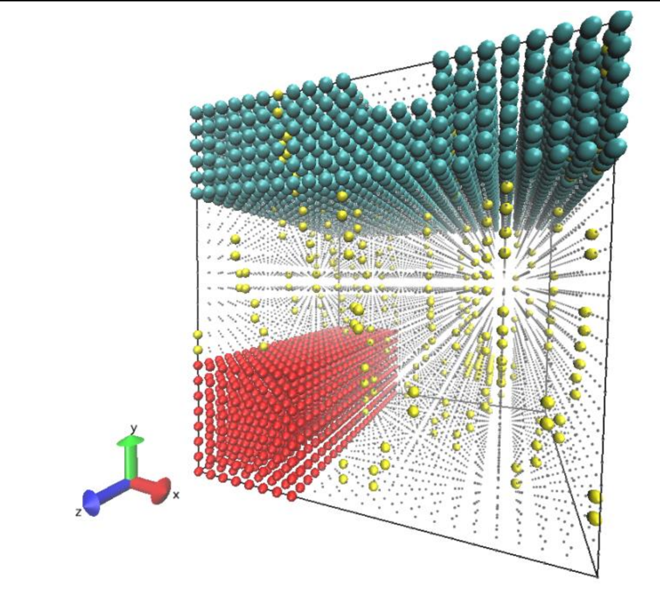

Fig. 6. Default placement of 6k XE computes (blue spheres) in a 23x6x24-hub box. This box has optimal bisection bandwidth per node.

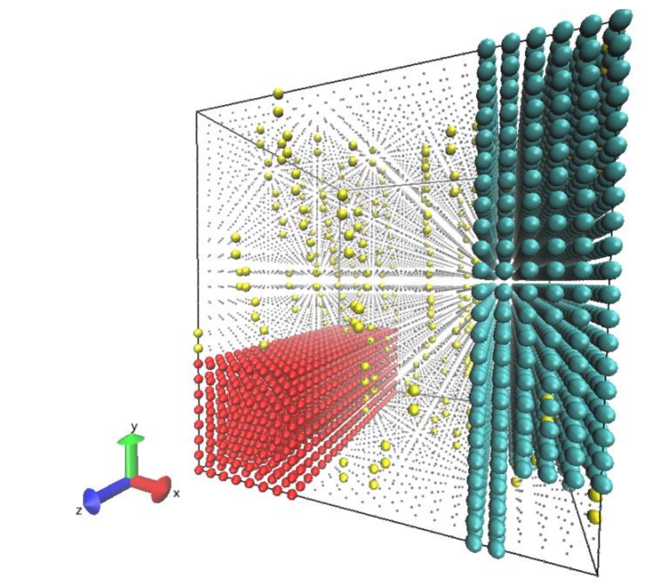

Fig. 7. Default placement for 6k XE compute nodes in a 6x24x24-hub box. This box has 1/2 of the bisection bandwidth per node of the box shown in Fig. 6 due to the slower links along the y direction.

In practice, a user who is charged by the node-hour would not benefit very much from randomizing the node list, since a much larger than desired number of nodes must be in the reservation than the minimum needed to run the job for this tactic to increase the bisection bandwidth per node significantly. However, Blue Waters users could obtain optimal bisection bandwidth per node in a section of size 23x*N*x24 (as shown in Fig. 6), with *N* between 2 and 6. Note that the bisection bandwidth per node of a section of size *N*x24x24 (Fig. 7) is only half that of a 23x*N*x24 section for this range of *N*. PSDNS runs using a domain with a $6k^3$ grid on 6k nodes took 1.64X less time when run in the 23x6x24 section than when run in the 6x24x24 section.

III. COARRAY FORTRAN ALL-TO-ALL

Coarrays [11] are included in the Fortran 2008 standard [12] and implemented in the current Cray ftn compiler. Coarray Fortran (CAF) enables parallel programming by means of a simple syntax for distributed arrays. Communication is implicit and one-sided, while synchronization is mostly invoked explicitly by the user. Coarrays of the same size and shape exist on each “image” (task), and subscripts in square brackets are used to refer to specific images.

Compared to MPI, messages passed by the lower-level communication libraries on behalf of CAF can have smaller headers (due to one-sidedness) and therefore can carry more data per packet for slightly higher bandwidth. In addition, latencies for short messages from coarray-related communication can be significantly lower than they are for the same message sizes with MPI. Finally, the user can more easily implement and hand-tune customized collective operations in CAF, such as the repeated concurrent All-to-Alls for two alternating sets of images (communicators) in PSDNS. All of these advantages lead one to anticipate better All-to-All performance from a hand-tuned CAF implementation compared to the MPI All-to-All version.

In order to ease integration of CAF with existing MPI codes that make extensive use of All-to-All collectives, we implemented and optimized a stand-alone library in CAF that enables application developers to directly substitute existing calls to MPI AlltoAll with very similar calls to a CAF version (called “compi_alltoall”). No other changes to the PSDNS source were needed, although we did introduce some additional code to verify (if desired) that all of the CAF calls produce exactly the same results as the MPI calls.

On each image (task), the compi_alltoall library statically allocates a single coarray whose size is at least 512 Bytes times the number of images in the column communicator. For the petascale benchmark on 12k nodes, the coarray size required is 6 MB. This allows the library to handle any message size without wasting too much memory for small messages, running out of memory for large messages, or allocating/deallocating the coarray on every call to compi_alltoall. Note that allocation/deallocation of a coarray can be rather time consuming, especially at large scales, since doing so also triggers implicit synchronization.

To perform the All-to-All communication, we proceed as indicated in the pseudo-code in Fig. 8. The messages are broken up into 512 Byte chunks. We begin by copying the first chunk of each message from the send buffer to the coarray called co_bucket. Next, an MPI barrier ensures that this copying step has completed on all images in the communicator before we attempt to access data in any of the remote images. After the barrier, we copy (pull) the chunk for the local image from the remote coarray images into the receive buffer. These pull operations are performed in a different fixed random order on each image, which significantly reduces the occurrence of hot spots on the interconnect. A similar concept is also used in

Cray's MPI AlltoAll library implementation, but for MPI it is not advantageous to break the messages into such small chunks. Finally, we issue a "sync memory" call and then another MPI barrier to ensure that the data in the coarrays has been copied to the receive buffers before being overwritten. These two calls entail less overhead than a "sync all" because this barrier pertains only to the images in the communicator, rather than to all images, and sync memory entails no direct synchronization effect. We then move on to the next 512 Byte chunk of the messages and repeat the copy/pull operations until all of the data in all of the messages has been transmitted.

We found that the 512 Byte chunk size gives the best performance on Blue Waters (as well as other systems with gemini interconnects) by experimenting with different chunk sizes for a wide range of message sizes and task counts.

Two other optimization details are worth mentioning here. The first time compi_alltoall is called with a communicator that it has not previously encountered, it determines and saves a mapping between the MPI ranks and CAF images in that communicator. It also computes and saves the random orderings that will be used by each image as it pulls data from the other images in that communicator.

Using the compi_alltoall library with PSDNS on 12288 nodes reduces the overall run time by 1.33X compared to the current MPI library.

```
! My image is my_im.
! The random_order array reorders the images.
Do i=1,n_chunks  ! Number of 512 Byte chunks in msg.
        i_start = 1 + (i-1)*512/8  ! 8 Bytes per word

        Do j=1,n_images  ! Number of images
                co_bucket(1:512/8, j) =
                sendbuf(i_start:i_start-1+512/8, j)
        End do  ! images

        MPI barrier (communicator, ierr)

        Do j=1,n_images
                Set k = random_order ( j )
                recvbuf(i_start:i_start-1+512/8, k) =
                co_bucket(1:512/8,my_im)[k]  ! Pull
        End do  ! images

        Sync memory  ! Ensures correct results
        MPI barrier (communicator, ierr)
End do  ! chunks
```

Fig. 8. Simplified psuedo-code for CAF All-to-All.

We also evaluated an alternative implementation that uses CAF directly integrated into the PSDNS code by declaring the send buffers as coarrays (allocated only once per run), rather than calling the compi_alltoall library, wherein the send buffers are copied to the internal coarray, which we determined consumes a non-negligible amount of time. We found in a simplified test code that the run time for the integrated CAF version was at best only about 5% faster than the library call. A plausible explanation for this is that the small coarray in compi_alltoall fits in cache on each image, and therefore the subsequent loop performing the communication can reuse that data rather than going to memory. In the more integrated implementation, the send buffer coarray is much too large to fit in any level of cache, and the data being transferred between images is not in contiguous blocks.

IV. CONCLUSIONS AND FUTURE WORK

Estimating the impact of each optimization applied to PSDNS described in this work for a fixed 12k^3 grid problem on 12k nodes, we have:

- 1.1X for memory management (environment variables or switch to Cray compiler),
- 1.4X for the slab-on-node decomposition,
- 1.4X for randomizing the node list, and
- 1.3X for the compi_alltoall library,

leading to a 2.8X overall decrease in the total wall clock time per time step.

The memory management optimization (or simply allocating arrays only once per run) pertains to many types of applications, as does the concept of minimizing the amount of off-node or off-node-pair communication [13].

Randomizing the node list is a technique that can benefit applications with All-to-All and random-pair communication patterns, especially if the user has dedicated access to at least a section of the system. Such a section should have a shape which optimizes the bisection bandwidth per node, such as 23xNx24, where $1 < N < 7$. Nodes whose partner on the same gemini is unavailable should not be used, especially if one is relying on two sets of neighboring partitions to be placed on node pairs. This would currently require requesting either a specific set of good nodes or more nodes than the job actually needs to run. In principle, some of the extra nodes could be used as spares to replace failed nodes and continue a run within the same batch job from the last checkpoint on the same number of active nodes.

The compi_alltoall library is available by request to users of Cray XE/XK systems, and should benefit any application that uses MPI_AlltoAll or MPI_AlltoAllV without any need for substantial code changes. Although All-to-All operations coded with CAF significantly outperform the equivalent MPI versions at this time on Cray XE/Xk systems with Gemini interconnects, Cray's MPI developers are continually working to close the gap to the extent that it is practical to do so.

Future work for PSDNS includes exploring overlapping communication for one component of a vector quantity undergoing a 3D FFT with computation for the next component. This has become easier to do using MPI with the recent inclusion of non-blocking collectives in Cray's MPI library. To get substantial overlap and a concomitant reduction in run time, messages need to be fairly large (at least 8kB) so that the Block Transfer Engine can be utilized effectively [14]. Getting good overlap requires 8 or fewer tasks per node. Therefore, we will need to use sufficiently large per-node problem sizes and 2 or more OpenMP threads per task.

REFERENCES

[1] National Science Foundation Solicitation NSF 06-573, “Leadership-Class System Acquisition - Creating a Petascale Computing Environment for Science and Engineering”,

http://www.nsf.gov/pubs/2006/nsf06573/nsf06573.html, 2006.

[2] D. A. Donzis, P. K. Yeung, & D. Pekurovsky, “Turbulence simulations on O(10^4) processors”. In TeraGrid 2008 Conference Proceedings, Las Vegas, NV, USA.

[3] R. Alverson, D. Roweth, and L. Kaplan, "The Gemini System Interconnect," in International Symposium on High Performance Interconnect, Aug. 2010, pp. 83 -87.

[4] C. Canuto, M. Y. Hussaini, A. Quarteroni, T. A. and Zang, Spectral Methods in Fluid Dynamics, Springer-Verlag, 1988.

[5] J. P. Boyd, Chebyshev and Fourier Spectral Methods, Second edition, Dover, New York, 2001.

[6] A. Averbuch, L. Ioffe, M. Israeli, and L. Vozovoi, “Multidimensional Parallel Spectral Solver for Navier-Stokes Equations”, in IMA Volumes in Mathematics and its Applications, vol. 120, Parallel Solution of Partial Differential Equations, P. Bjorstad and M. Luskin, eds., Springer-Verlag, New York, NY, USA, 2000.

[7] D. Pekurovsky, “P3DFFT: a framework for parallel computations of Fourier transforms in three dimensions”, SIAM Journal on Scientific Computing, Vol. 34, No. 4, pp. C192-C209, 2012.

[8] D. A. Donzis, K. R. Sreenivasan, and P. K. Yeung, “The Batchelor spectrum for mixing of passive scalars in isotropic turbulence”, Flow, Turbulence, and Combustion 85, 549-566, 2010.

[9] Blue Waters main Web page, http://www.ncsa.illinois.edu/BlueWaters/.

[10] David Whitaker, 2012, private communication.

[11] J. Reid, ”Coarrays in the Next Fortran Standard,

ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf, 2010

[12] Fortran 2008 [Final Draft International Standard (FDIS)],

ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1830.pdf, 2010

[13] R. Fiedler and S. Whalen, “Improving task placement for applications with 2D, 3D, and 4D virtual Cartesian topologies on 3D torus networks with service nodes”, CUG 2013, May 6-9, 20123 Napa, CA, USA.

[14] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella, “Leveraging the Cray Linux environment core specialization to realize MPI asynchronous progress on Cray XE systems”, CUG 2012, April 29 - May 3, 2012, Stuttgart, Germany,

https://cug.org/proceedings/attendee_program_cug2012/includes/files/pap115.pdf.