Advanced Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms

Robert Preissl John Shalf
Alice Koniges
Lawrence Berkeley
National Laboratory
{rpreissl,jshalf,aekoniges}@lbl.gov

Nathan Wichmann Bill Long
CRAY Inc.
{wichmann,longb}@cray.com

Stephane Ethier
Princeton Plasma
Physics Laboratory
ethier@pppl.gov

Abstract

In this paper we explore new parallel language constructs for the communication kernel of a real world magnetic fusion simulation code using the Partitioned Global Address Space (PGAS) model. The studied kernel is the particle shift phase of a tokamak simulation code in a toroidal geometry, which models the transit of charged particles between neighboring toroidal computational domains. We introduce Coarray implementations that send more and smaller messages than the original MPI-1 based kernel, and demonstrate that the more light-weight one-sided messaging algorithms can provide a significant performance advantage for such bandwidth-limited kernels.

Compact application versions of the new Coarray based particle movement algorithms are extracted from the fusion application and evaluated on the worlds largest HPC platform with PGAS enabled hardware (Cray XE6) and on a Cray XT4 without hardware support for one-sided messaging. Experimental evaluations show that our best Coarray algorithm improves the best MPI-1 communication kernel by 83% using 131K processors on the Cray XE6 system and by 47% using 16K processors on the Cray XT4.

Keywords: GTS, Particle-In-Cell, Fortran 2008, PGAS, Hybrid MPI & Coarray computing

1. Introduction

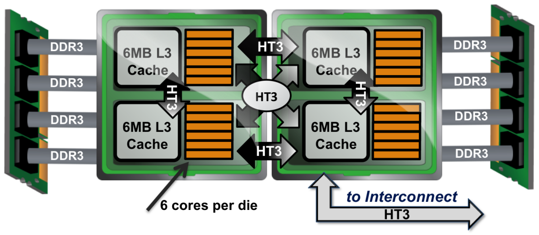
The path towards realizing next-generation petascale and exascale computing is increasingly dependent on building supercomputers with unprecedented numbers of processors. Applications and algorithms will need to change and adapt as node architectures evolve to overcome the daunting challenges posed by such massive parallelism. To prevent the communication performance from dominating the overall cost of these ultra-scale systems, there is a critical need to develop innovations in algorithms, parallel computing languages, and hardware support for advanced communication mechanisms. One such innovation in communication technology to support improved scalability is the development of one-sided messaging methods and PGAS languages such as Unified Parallel C (UPC) and Fortran 2008, which incorporates parallel features historically identified as Coarray Fortran (CAF). The biggest advantage is that remote memory is referred to directly instead of having to call a subroutine for loading and storing data. The one-sided messaging abstractions of PGAS languages also open the possibility of expressing new algorithms and communications approaches that would otherwise be impossible, or unmaintainable using the two-sided messaging semantics of communication libraries like MPI-1. The expression of the one-sided messaging semantics as language constructs (Coarrays in Fortran and shared arrays in UPC) improves the legibility of the code and allows the compiler to apply communication optimizations. Hardware support for PGAS constructs and one-sided messaging, such as that provided by the recent Cray XE6 Gemini interconnect, is essential to realize the maximal performance potential of these new approaches. However, as it will be demonstrated in this work, even without architectural support for one-sided messaging good performance improvements over tuned two-sided communication kernels can be achieved.

In certain instances one-sided communication may offer significant advantages over traditional two-sided message passing. In this paper we provide a counter-example to the common lore that performance is optimized by sending fewer and larger messages to asymptotically approach peak bandwidth. We exploit the one-sided nature of the PGAS programming model and introduce novel algorithms — that cannot be expressed in a two-sided message passing scheme — using more, but smaller messages with lower startup and completion costs. Building upon such light-weight one-sided communication techniques we will show how to efficiently spread out the communication over a longer period of time, resulting into a reduction of bandwidth requirements and a more sustained communication and computation overlap.

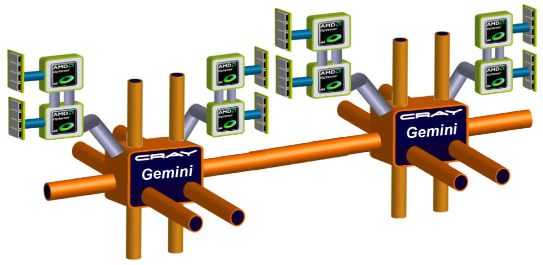
This work focuses on advanced communication optimizations for the Gyrokinetic Tokamak Simulation (GTS) [19] code, which is a global three-dimensional Particle-In-Cell (PIC) code to study the microturbulence and associated transport in magnetically confined fusion plasmas of tokamak toroidal devices. We have created a skeleton application that represents the communication requirements for the GTS application so that we could rapidly prototype and measure the performance of alternative communication strategies. The best strategies could then be easily incorporated back into the original GTS code, where we could evaluate its benefit to the overall scalability and performance of the code. We focus on Fortran's Coarray facilities because Fortran is the language used to implement the bulk of the GTS code base.

1.1 Related Work

A number of studies have investigated the simplicity and elegance of expressing parallelism using the CAF model. Barrett [2] stud-



(a) XE6 compute node with AMD “Magny Cours”



(b) 3D torus network connecting nodes in the XE6

Figure 1. The Cray XE6 incorporates AMD’s twelve-core “Magny Cours” processors and the Gemini interconnect in a three-dimensional (3D) torus network

ied different Coarray Fortran implementations of Finite Differencing Methods and Numrich et al [16] developed a Coarray enabled Multigrid solver focusing primarily on programmability aspects. Bala et al [1] demonstrated performance improvements over MPI in a molecular dynamics application on a legacy HPC platform (Cray T3E). In addition, parallel linear algebra kernels for tensors (Numrich [15]) and matrices (Reid [17]) benefit from a Coarray based one-sided communication model due to a raised level of abstraction with little or no loss of performance over MPI. Implementing the NAS parallel benchmarks in UPC (El-Ghazawi et al [11] and Cantonnet et al [7]) revealed optimization opportunities in UPC and showed no particular performance improvements. Bell et al [3] followed a similar approach to the one presented in this paper and improved performance of the NAS FT benchmark by efficiently distributing the communication operations throughout the application using UPC.

Mellor-Crummey et al. [10, 14] have proposed an alternate design for CAF, which they call CAF 2.0, that adds some capabilities not present in the standard Fortran version, but also removes some capabilities. Their CAF 2.0 compiler uses a source to source translator to convert CAF 2.0 programs into Fortran 90 (F90) programs with calls to the CAF 2.0 runtime system. The CAF 2.0 runtime uses the GASNet library [4] for one-sided communication. Applications of Coarray Fortran to the NAS parallel benchmarks (Coarfa et al. [8]) and to the Sweep3D neutron transport benchmark (Coarfa et al. [9]) show nearly equal or slightly better performance than their MPI counterparts.

Our work makes the following contributions. We use a “real world application” to demonstrate the ability to incrementally change the GTS application from MPI-1[1] to CAF. We created a compact skeleton application that enables rapid comparison of alternative communication representations. We present a novel PGAS algorithm that fully exploits hardware supported proposed CAF features to implement a new approach to communication that would not otherwise be feasible with MPI two-sided communication. We demonstrate the scalability of our new approach using up to 131K processors for the skeleton application. We also show scalability using the new kernel in the full GTS code running experiments on up to 32K processing cores. The full code result gives a 83% performance improvement at the largest concurrency over the existing MPI algorithm. Overall, our work is the first demonstration of sustainable performance improvements over MPI in a real world application and constitutes the largest CAF simulation conducted so far on a PGAS enabled hardware.

The idea is a smooth transition from MPI to Fortran 2008 (using Coarrays), i.e., we replace the existing MPI communication kernel by a new algorithm using Coarrays and leave the rest of the physics simulation code unchanged, which still has MPI function calls in it. The coexistence of different programming models is typical of what will likely be required as we move legacy codes to next-generation HPC platforms.

[1] For the rest of the paper we use the term MPI when MPI-1 is intended. If we refer to the MPI one-sided extension, we use the term MPI-2 explicitly.

1.2 Hardware Platform

The primary platform chosen for this study is a Cray XE6 supercomputer capable of scaling to over 1 million processor cores, and uses AMD’s “Magny Cours” Opteron processors and Cray’s proprietary interconnect “Gemini”. The basic compute building block of our XE6 system, shown in Figure 1(a), is a node with two AMD “Magny Cours” sockets. Each socket has two die each with six cores for a total of 24 cores on the compute node. Each die is directly attached to a low latency, high bandwidth memory such that each of the six cores on that die have equal, or flat, access to the directly attached memory, making a die, with its six cores, a natural compute unit for shared memory programming.

Figure 1(b) outlines how compute nodes on a Cray XE6 system are connected to all other compute nodes using the Gemini router via a 3D torus. Each Gemini connects to two compute nodes using unique and independent Network Interface Controllers (NICs) via HyperTransport 3 (HT3) connection. Advanced features include support for one-sided communication primitives and support for atomic memory operations. This allows any processing element on a node to access any memory location in the system, through appropriate programming models, without the “handshakes” normally required with most two-sided communication models.

The majority of experiments shown in this paper are conducted on the Hopper Cray XE6 system installed at the National Energy Research Scientific Computing Center (NERSC) comprising 6392 nodes with 153,408 processor cores with a system peak performance of 1.288 Petaflops and a reported HPL performance of 1.05 Petaflops. In addition, experiments have been carried out at NERSC’s Franklin system — a Cray XT4 supercomputer having 9572 compute nodes, connected via the “Seastar” interconnect, with each node consists of a 2.3 GHz single socket quad-core AMD Opteron processor (“Budapest”). Performance studies on Franklin (XT4) are included to demonstrate that our best CAF algorithms still outperform the best MPI algorithms even without having hardware support for one-sided communication.

The Cray Compiler Environment (CCE) version 7.3.1 on Hopper (XE6) and CCE 7.3.3 on Franklin (XT4) was used to compile all of the source code for this paper. On Hopper (XE6), CCE 7.3.1 fully supports Fortran 2008 translating all Coarray references into instruction sequences that access hardware mechanisms. On Franklin (XT4), PGAS communication such as CAF is build upon the GASNet library [4] for one-sided communication.

2. The GTS fusion simulation code

GTS is a general geometry PIC code developed to study plasma microturbulence in toroidal, magnetic confinement devices called tokamaks [19]. Microturbulence is a complex, nonlinear phenomenon that is believed to play a key role in the confinement of energy and particles in fusion plasmas [12], so understanding its characteristics is of utmost importance for the development of practical fusion energy. In plasma physics, the PIC approach amounts to following the trajectories of charged particles in both self-consistent and externally-applied electromagnetic fields. First, the charge density is computed at each point of a grid by accumulating the charge of neighboring particles. This is called the *scatter* phase. Prior to the calculation of the forces on each particle from the electric field (*gather* phase) — we solve *Poisson's equation* to determine the electrostatic potential everywhere on the grid, which only requires a two-dimensional solve on each poloidal plane (cross-section of the torus geometry) due to the quasi-two-dimensional structure of the potential[2]. This information is then used for moving the particles in time according to the equations of motion (*push* phase), which denotes the fourth step of the algorithm.

2.1 The GTS Parallel Model

The parallel model in GTS consists of three levels: **(1)** A one-dimensional domain decomposition in the toroidal direction (the long way around the torus). MPI is used for performing communication between the toroidal domains. Particles move from one domain to another while they travel around the torus — which adds another, a fifth, step to our PIC algorithm, the *shift* phase. This phase is the focus of this work. It is worth mentioning that the toroidal grid, and hence the decomposition, is limited to about 128 planes due to the long-wavelength physics being studied. A higher toroidal resolution would only introduce waves of shorter parallel wavelengths that are quickly damped by a collisionless physical process known as Landau damping, leaving the results unchanged [12]. **(2)** Within each toroidal domain we divide the particle work between several MPI processes. All the processes within a common toroidal domain of the one-dimensional domain decomposition are linked via an *intradomain MPI communicator*, while a *toroidal MPI communicator* links the MPI processes with the same intradomain rank in a ringlike fashion. **(3)** OpenMP compiler directives are added to most loop regions in the code for further acceleration and for reducing the GTS memory footprint per compute node.

Figure 2 shows the GTS grid, which follows the field lines of the externally applied magnetic field as they twist around the torus[3]. In the following we focus on the advantages of using CAF instead of MPI in a communication intensive part of GTS, the shift algorithm, and present two optimized MPI implementations as well as our new CAF algorithms.

3. Particle shift algorithms in GTS

The shift phase is the most communication intensive step of a GTS simulation. At each time step, about 10% of the particles inside of a toroidal domain move out through the "left" and "right" boundaries in approximately equal numbers. A 1-billion particle simulation

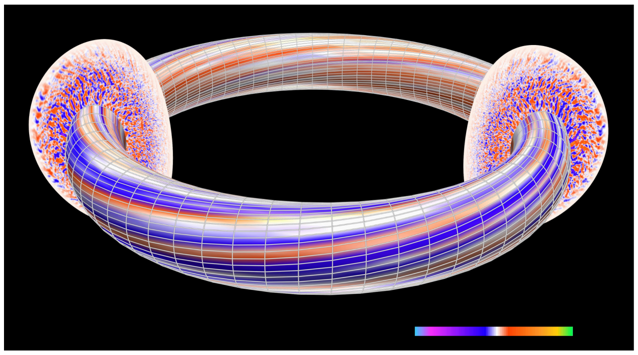


Figure 2. GTS field-line following grid & toroidal domain decomposition. Colors represent isocontours of the quasi-two-dimensional electrostatic potential

translates to about 100GB of data having to be communicated each time shift is called.

In terms of wall clock time, the particle shift contributes to approximately 20% of the overall GTS runtime and is expected to play an even more significant role at higher scales — as observed in scaling experiments on Hopper (XE6). After the push phase, i.e., once the equations of motion for the charged particles are solved, updated coordinates of a significant portion of particles are outside the local toroidal domain. Consequently affected particles have to be sent to neighboring — or in rare cases to even further — toroidal domains. The amount of shifted particles as well as the number of traversed toroidal domains depend on the toroidal domain decomposition coarsening (mzetamax), the time step (tstep), the background temperature profile influencing the particle's initial thermal velocity (umax) and the number of particles per cell (micell). The distance particles can travel along the toroidal direction in each time-step is restricted by the spatial resolution of physical dynamics in the parallel direction. For a valid simulation, particles do not travel more than 4 ranks per time-step (realized by choosing an appropriate step-size).

3.1 The MPI multi stage shifter (MPI-ms)

MPI-ms is an optimized version of the original MPI shift algorithm in GTS and models the shift of particles to adjacent or further toroidal domains of the tokamak in a ring-like fashion. It implements a nearest neighbor communication pattern, i.e., if particles need to be shifted further an additional iteration is required to move designated particles to their final destination[4]. The pseudo-code excerpt in Listing 1 highlights the major steps in the *MPI-ms* shifter routine. The most important steps are iteratively applied for every shift stage and correspond to the following:

(1) Each process iterates through its local particle array (traverses the whole array at the first shift stage and for consecutive stages only newly received particles are considered) and computes which particles have to be shifted to the left and to the right, respectively. This yields the number of right- (*shift_r*) and left-shifted (*shift_l*) particles as well as arrays (*holes_r*, *holes_l*) containing the indices of right- and left-shifted particles in *p_array*. Note, that remaining and shifted particles in *p_array* are randomly distributed and a preceded sorting step would involve too much overhead. **(2)** At every shifting stage the sum of shifted particles is communicated to all processes within the same toroidal communicator (*tor_comm*) by an allreduce call of a limited sized communicator ($\leq$ 128 processes). This denotes the break condition of the shifter, i.e., to exit

[2] Fast particle motion along the magnetic field lines leads to a quasi-two-dimensional structure in the electrostatic potential (see Figure 2).

[3] The two cross sections demonstrate contour plots of potential fluctuations driven by Ion Temperature Gradient-Driven Turbulence (ITGDT) [13], which is believed to cause the experimentally observed anomalous loss of particles and heat in the core of magnetic fusion devices such as tokamaks.

[4] Large scale experiments have shown that only a few particles with high initial thermal velocities are affected, crossing more than one toroidal domain.

```
do shift_stages=1,N
!(1) compute right- & left-shifted particles
  do i=m0,me
    dest=compute_destination(p_array(i))
    if(dest > local_tor_domain) {
      holes_r(shift_r++)=i
    }else if(dest < local_tor_domain) {
      holes_l(shift_l++)=i }
  enddo

!(2) communicate amount of shifted particles
  if(shift_stages.ne.1) {
    MPI_ALLREDUCE(shift_r+shift_l,all,tor_comm)
    if(all.eq.0) { return } }

!(3) Prepost receive requests
  MPI_IRECV(recv_left,left_rank,reqs_1(1),..)
  MPI_IRECV(recv_right,right_rank,reqs_2(1),..)

!(4) pack particle to move right and left
  do m=1,shift_r
    send_right(m)=p_array(holes_r(m))
  enddo
  do m=1,shift_l
    send_left(m)=p_array(holes_l(m))
  enddo

!(5) reorder remaining particles: fill holes
  fill_holes(p_array,holes_r,holes_l)

!(6) send particles to right and left neighbor
  MPI_ISEND(send_right,right_rank,reqs_1(2),..)
  MPI_ISEND(send_left,left_rank,reqs_2(2),..)

!(7) add received particles and reset bounds
  MPI_WAITALL(2,reqs_1,..)
  add_particles(p_array,recv_left)
  MPI_WAITALL(2,reqs_2,..)
  add_particles(p_array,recv_right)
enddo
```

Listing 1. Multi stage MPI shifter routine (MPI-ms)

the shifter if no particles from all processes within the toroidal communicator need to be shifted (*all.eq.*0). The first MPI_Allreduce call can be avoided since shifts of particles happen in every iteration of GTS. **(3)** Preposte non-blocking receive calls for particles from left and right neighbors to prevent unexpected message costs and to maximize the potential for communication overlap. Usage of pre-established receive buffers (*recv_left* and *recv_right* eliminates additional MPI function calls to communicate the number of particles being sent, which is attached to the actual message. **(4)** Pack particles, which have to be moved to their left- and right immediate toroidal neighbor into *send_right* and *send_left* buffers. **(5)** Reorder the particle array so that holes, which will be created due to the shift of particles, are filled up by remaining particles. **(6)** Send the packed shifting particles to the right and left neighboring toroidal domain. And **(7)** Wait for the receive calls to complete and incorporate particles received from left and right neighbors to *p_array*.

The shifter routine involves heavy communication due to the MPI_Allreduce call[5] and especially because of the particle exchange implemented using a ring-like send & receive functionality. In addition, intense computation is involved mostly because of the particle reordering that occurs after particles have been shifted and incorporated into the new toroidal domain respectively.

[5] Note, that the Allreduce has limited impact on the performance due the limited communicator size (128 MPI processes).

```
!(1) Prepost receive requests
do i=1,nr_dests
  MPI_IRECV(recv_buf(i),i,req(i),tor_comm,..)
enddo

!(2) compute shifted particles and fill buffer
do i=1,p_array_size
  dest=compute_destination(p_array(i))
  if(dest.ne.local_tor_domain) {
    holes(shift++)=i
    send_buf(dest,buf_cnt(dest)++)=p_array(i) }
enddo

!(3) Send of particles to destination process
do j=1,nr_dests
  MPI_ISEND(send_buf(j),j,req(j+i),tor_comm,..)
enddo
MPI_WAITALL(2*nr_dests,req,..)

!(4) fill holes with received particles
do m=1,min(recv_length,shift)
  p_array(holes(m))=recv_buf(src,cnt)
  if(cnt.eq.recv_buf(src,0)) {cnt=1; src++}
enddo

!(5) append remaining particles or fill holes
if(recv_length < shift) {
  append_particles(p_array,recv_buf) }
else { fill_remaining_holes(p_array,holes) }
```

Listing 2. Single stage MPI shifter routine (MPI-ss)

3.2 The MPI single stage shifter (MPI-ss)

In contrast to the previous MPI algorithm, this implementation employs a single stage shifting strategy, i.e., particles with coordinates outside of local toroidal domains are immediately sent to the processor holding the destination domain, rather then shifting over several stages. Consequently the allreduce call and, more importantly, additional communication and memory copying related to particles, which cross more than one toroidal section, can be saved (in *MPI-ms*, e.g., if a particle has to be moved to the second from right domain it is first sent to the immediate right toroidal neighbor and then sent from there to the final destination). The major steps in the *MPI-ss* shifter, shown in Listing 2 in pseudo code form, are:

(1) Preposte receive requests for receiving particles from each possible source process of the toroidal communicator. Since particles normally traverse no more than two entire toroidal domains in one time step in typical simulations (in practice, particles do not cross more than one toroidal section, but we allocate additional space for a few fast moving electrons), two-dimensional allocations of receive buffers for six ($= nr\_dests$) potential sources $\{rank-3, .., rank-1, rank+1, .., rank+3\}$ are performed (send buffers are similarly allocated for six potential destinations). **(2)** New coordinates of each particle in the local particle array are computed. If a particle needs to be shifted it is copied into a two-dimensional send buffer, which keeps records of shifted particles for each possible destination. In **(3)** the shift of particles occurs, where the number of shifted particles per send process is attached to the message and denotes the first element in the send buffer. A wait call ensures that newly received particles from the receive buffer can be safely read. **(4)** and **(5)** All received particles ($recv\_length$ many) are added to the particle array. First, newly received particles are used to fill the positions of shifted particles in the particle array. If there are more received particles than shifted particles we append the rest to p_array. Otherwise particle residing at the end of p_array are taken to fill remaining holes.

Additional MPI versions for the shift of particles were implemented. We studied different algorithms, other MPI communication techniques (e.g., buffered send), the usage of MPI data types (e.g., MPI_Type_create_indexed_block) to eliminate the particle packing overhead, etc.; with no — or even negative (in case of using MPI data types) — performance impacts. MPI implementations of shift have been extensively researched in the past and *MPI-ms* and *MPI-ss* represent, to the best of our knowledge, the most efficient message passing algorithms.

3.3 The CAF-lock shifter (CAF-lock)

The most significant difference of this (single stage) implementation to the *MPI-ss* algorithm from above is that it fully exploits CAF's one-sided messaging scheme. The novelty of this approach lies in the fact that a two-dimensional send buffer is successively filled as above, but messages (filling a one-dimensional receive buffer of the destination toroidal domain) are sent once the amount of particles for a specific destination image reaches a certain threshold value. Once a buffer's content has been sent, it can be reused and filled with new shifted particles with the same destination. This new algorithm sends more and smaller messages, which does not particularly result in higher overheads due to the lower software overhead of one-sided communication, whose semantics are fundamentally lighter-weight than message passing [3]. This implementation differs from the previous MPI approaches where a send buffer is filled and sent once containing all particles to be moved. This novel algorithm enables to overlap the particle work (computation of the toroidal destination plus copying of particles into send buffers) and particle communication (moving particles from neighboring domains are written into receive buffers in any order) and also results in less memory requirements for allocating send buffers. In addition, the pipelining of smaller light-weight messages, which do not require synchronization or ordering on the remote image, spreads out the communication over a longer period of time and turns out to be very efficient for such bandwidth limited problems.

Shifting particles from one source CAF image is implemented as a put operation, which adds particles to a receiving queue — implemented as a one-dimensional Coarray — on the destination image. Since access conflicts to this globally addressable receiving buffer might frequently arise (e.g., an image shifts particles to its right toroidal neighbor while at the same time the latter image also receives data from its right neighbor) we employ a locking mechanism for the particle shift to the receiving queue Coarray. That is, once the "queue-lock" on the destination image is acquired, i.e., exclusive access to the destination's image receiving queue is given, particles are appended to the end of the queue. In order to mitigate long stall cycles when waiting for an image to release the "queue lock" we partition the send buffer in equally sized chunks. Whenever a chunk is fully filled the image tries to acquire the destination's "queue-lock" to empty the whole send buffer or immediately returns to its particle work in case the lock is held by another image. However, if the last chunk in the send buffer is filled the image has to wait until it acquires the lock.

Listing 3 highlights the major steps of this algorithm in pseudo code form using CAF and F90 array notation:

(1) If a particle moves out of the current processor's domain, its location in (*p_array*) is held in *holes*. Since the send buffer is partitioned into several pieces of equal size, this moving particle with destination *dest* will be placed into the following position (*lastp*) of the send buffer (*send_buf*):

"Number of already filled chunks" (*chunkc(dest)*) $*$ "chunk-size" (*sb_size*) $+$ "the counter holding the number of stored particles (for the computed destination) in the latest non-empty chunk of the send buffer" (*buf_cnt(dest)*).

```
!(1) compute shifted particles and fill the              1
!   receiving queues on destination images
do i=1,p_array_size                                      3
  dest=compute_destination(p_array(i))
  if(dest.ne.local_tor_domain) {                         5
    holes(shift++)=i; buf_cnt(dest)++
    lastp=buf_cnt(dest)+chunkc(dest)*sb_size             7
    send_buf(dest, lastp)=p_array(i)
    if(buf_cnt(dest).eq.sb_size) {                       9
      do
        lock(q_lock[dest],acquired_lock=gotL)            11
        if(gotL) exit
        !only if sendbuffer is not totally full          13
        if(chunkc(dest).lt.(nr_chunks-1)) {
          chunkc(dest)++; buf_cnt(dest)=0                15
          goto Line3 }
      end do                                             17
      d_Qpos=Qpos[dest]
      recvQ(d_Qpos:d_Qpos+length -1)[dest]&              19
        =send_buf(dest,1:lastp)
      Qpos[dest]+=lastp                                  21
      unlock(q_lock[dest])
      chunkc(dest)=0; buf_cnt(dest)=0 }  }               23
enddo
                                                         25
!(2) shift remaining particles
empty_send_buffers(send_buf)                             27
length_recvQ=Qpos-1
                                                         29
!(3) sync with images from same toroidal domain
sync images([my_shift_neighbors])                        31

!(4) fill holes with received particles                  33
do m=1,min(length_recvQ, shift)
  p_array(holes(m))=recvQ(m)                             35
enddo
                                                         37
!(5) append remaining particles or fill holes
if(length_recvQ-min(length_recvQ, shift).gt.0) {         39
  append_particles(p_array,recvQ) }
else { fill_remaining_holes(p_array,holes) }             41
```

Listing 3. CAF-lock shifter routine

Whenever such a chunk of the send buffer for destination *dest* is full (Line 9), the image tries to acquire the lock (*q_lock*) to gain exclusive access to the receiving queue (*recvQ*) of the destination image. If the lock cannot be obtained and if there is at least one empty chunk left to fill up in the send buffer, the image resets the "last-chunk-particle-counter" (*buf_cnt*) and increments the "chunk-counter" (*chunkc*) before for returning to the particle work. Otherwise, if the lock can be acquired (immediately or after spinning in the lock-acquisition loop, from Line 10 to Line 17, in case the image has no empty send buffer chunk left) the length of the receiving particle queue (*d_Qpos*) hold by the destination image is determined through a CAF get operation. Starting from this position all particles available in the send buffer (ranging from position 1 to *lastp*) will be added to *recvQ* on image *dest* by a CAF put operation (Lines 19 and 20). After the particle put the remote image's receive queue is updated by adding the number of sent particles. Now the lock can be released and both counters ("last-chunk-particle-counter" *buf_cnt* and "chunk-counter" *chunkc*) are reset. Figure 3 illustrates the send buffer on image 6 being able to hold nine particles for each possible send destination (images 3,4,5 to the left and images 7,8,9 to the right). In this example, the send buffers' entries are split up in three chunks, for buffering three moving particles per chunk (i.e., *sb_size*=3) for each destination image. In Figure 3 we assume that the 6^{th} particle for destination image 5 has been the last inserted particle to the send buffer on image 6; Since the 2^{nd} chunk for destination 5 is in use, we conclude that image 6

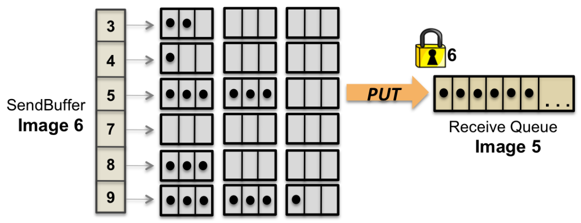


Figure 3. A two-dimensional send buffer in *CAF-lock*

could not acquire the *q_lock* on image 5 for sending its first chunk of particles and therefore *chunkc(5)*=1. After inserting the 6^{th} particle designated for destination image 5 the 2^{nd} chunk is fully filled (*buf_cnt(5)*=3) and image 6 tries again to acquire the *q_lock* on image 5. This time image 6 successfully obtains the lock and inserts six particles to the receiving buffer on image 5 (in the example in Figure 3, *recvQ* on image 5 was empty, i.e., *Qpos*[5]=1, before image 6 sends its particles). Then image 6 updates the queue size on image 5 (*Qpos*[5]=1+6), releases the *q_lock* and resets *chunkc(5)* and *buf_cnt(5)* enabling to overwrite the send buffer for image 5.

(2) Send remaining ($\leq$ $(sb\_size - 1)$) particles from the send buffer to their destination using again the CAF locking functionality for exclusive write access to the corresponding receive queue (here, images wait until the lock can be acquired). The total number of received particles (*length_recvQ*) is the local queue length (note, *Qpos.eq.Qpos*[THIS_IMAGE()][6]), which has been successively updated by neighboring images.

(3) Besides the required locks preventing access conflicts no synchronization between the images has been performed yet. Now we need to synchronize between the local image and all other images, which can theoretically add particles to a local image's receiving queue. This ensures that all images passing this synchronization statements have finished their work from (1) and (2) and all send buffers are flushed[7]. The array *my_shift_neighbors* stores the global CAF indices of images within reach of the local images, i.e., as discussed above it is an array with six entries. Adding received particles to the local particle array happens in **(4)** and **(5)**, which is analogous to (4) and (5) in *MPI-ss*.

Despite a slightly better memory efficiency, the communication and computation overlap and the more efficient network bandwidth utilization the algorithm still exhibits too many idle cycles due to the locking mechanism for preventing data races. The next presented algorithm aims to circumvent this problem by using remote atomic memory operations instead of locks.

3.4 The CAF-atomic shifter (CAF-atom)

Analyzing the *CAF-lock* algorithm reveals that it is sufficient to lock, or in other words, to atomically execute, the process of reserving an appropriate slot in the receive queue. Once such a slot from position $s + 1$ to position $s + size$, where s denotes the last inserted particle to the receiving queue by any image and $size$ stands for the number of particles to be sent to the receiving image, is securely (i.e., avoiding data races) given, the sending image can start and complete its particle shift at any time. Thus, no locking of the receiving queue is required. Besides the negligible overhead involved in the atomicity, required to safely reserve a slot in the receive queue, this new CAF algorithm enables to fully unleash the CAF images until the particles in the receiving queue

```
!(1) compute shifted particles and fill the                1
!  receiving queues on destination images
do i=1,p_array_size                                         3
  dest=compute_destination(p_array(i))
  if(dest.ne.local_tor_domain) {                            5
    holes(shift++)=i
    send_buf(dest,buf_cnt(dest)++)=p_array(i)               7
    if(buf_cnt(dest).eq.sb_size) {
      d_Qpos=Afadd(Qpos[dest],sb_size)                      9
      recvQ(d_Qpos:d_Qpos+sb_size -1)[dest]&
        =send_buf(dest,1:sb_size)                           11
      buf_cnt(dest)=0  }  }
enddo                                                       13

!(2) shift remaining particles                              15
empty_send_buffers(send_buf)
length_recvQ=Qpos-1                                         17

!(3) sync with images from same toroidal domain             19
sync images([my_shift_neighbors])
                                                            21
!(4) fill holes with received particles
do m=1,min(length_recvQ,shift)                              23
  p_array(holes(m))=recvQ(m)
enddo                                                       25

!(5) append remaining particles or fill holes               27
if(length_recvQ-min(length_recvQ,shift).gt.0) {
  append_particles(p_array,recvQ) }                         29
else { fill_remaining_holes(p_array,holes) }
```

Listing 4. CAF-atom shifter routine

are added to the local particle array and completion of any communication can be ensured. Due to the fast atomic memory operations two-dimensional receive buffers are not needed, which results into a more efficient and safe memory management. The process of claiming the required space in the receive queue on the remote destination image is performed by global atomic memory operations, which, unlike other functions, cannot be interrupted by the system and can allow multiple images or threads to safely modify the same variable under certain conditions. Global Atomic Memory Operations (global AMOs) are supported on Cray Compute Node Linux (Cray CNL) compute nodes and use the network interface to access variables in memory on Cray XE machines (when compiling with Cray CCE starting from version 7.3.0). Thus, hardware support ensuring a fast and safe execution is given for those critical operations. Listing 4 outlines the major steps of the revised CAF algorithm from section 3.3 using global AMOs instead of CAF locks in pseudo code form using CAF and F90 array notation:

(1) Similar to Listing 3, particles moving out of their toroidal domain are detected and copied into a send buffer (*send_buf*). The position of those shifting particles will be held in *holes*. Since no locking of the receiving queue is required in this algorithm, there is no benefit in partitioning the send buffer into chunks of send arrays for hiding lock acquisition costs. However, we maintain the two dimensionality of the send buffer enabling a single stage shift algorithm. Consequently we assign each moving particle to its corresponding set of particles based on its destination. If an image's send buffer for a specific destination image is full (Line 8), a global AMO — in detail, a **global atomic fetch and add operation** — is executed, which updates the remote queue size on image *dest* (*Qpos*[dest]) by adding the number of moving particles to the former queue size and returns this old value, held in *d_Qpos* (Line 9). The sending image then launches a CAF put operation to transfer *sb_size* particles to image's *dest* receiving queue starting from position *d_Qpos* to position *d_Qpos+sb_size-1*. Due to the one-sided nature of the put operation, the sending image does not need to wait for any acknowledgement from the destination

[6] The integer function THIS_IMAGE() returns the image's index between 1 and the number of images.

[7] Calling "sync images" implies a call to the sync memory intrinsic subroutine, which guarantees to other images that the image calling the function has completed all preceding accesses to Coarray data.

image. As soon as the data is sent it resets the send buffer counter (*buf_cnt*) for the image *dest* and starts adding new moving particles to the send buffer. No handshake for transmitting moving particles between the source and destination image is necessary. **(2)** If there are remaining particles in the send buffer for any destination image, we reuse the global atomic fetch and add operation described in (1) for safely inserting particles on a remote image's receiving queue. The following steps **(3)** to **(5)**, to add particles from the local receive queue to the local particle array, are similar to the steps (3) to (5) in the *CAF-lock* algorithm shown in Listing 3. Note, unlike to algorithm from Listing 3 no explicit synchronization until (3) is required.

Experiments to determine optimal buffer size threshold values for both CAF algorithms carried out for several problem sizes (varying the number of particles per CAF image) suggest $500 <$ *sb_size* < 1000. A too small setting of *sb_size* causes extra atomic or locking operations and other communication overhead (setting up the transfer, etc.) that would be difficult to amortize. For a very large buffer size threshold value, respectively, we would concentrate shifting particles into fewer, more intensive messages rather than spreading them out.

For simplicity Coarrays in Listings 3 and 4 have been accessed by a one-dimensional value (*dest*). In practice, a codimension of 2 is required to represent a poloidal and a toroidal component serving as an analogon to the poloidal and toroidal MPI communicators.

The CAF algorithm using locks described in section 3.3 and especially the CAF algorithm using global AMOs from section 3.4 fully exploit the one-sided messaging nature of CAF. That is, launching an a priori unknown number of send operations as implemented in both CAF algorithms (it is not known in advance how many particles will be moving) is beyond MPI's philosophy, which employs a two-sided messaging scheme. On the contrary, less synchronization implies a significantly higher risk for dead locks, race conditions or other similar non deterministic events as experienced in the development of our CAF algorithms. As for implementing an MPI-2 algorithm similar to the CAF versions, the complexity required to implement remote atomicity (enables the messages pipelining of the CAF algorithms) in MPI-2 one-sided, in addition to several other semantic limitations [5], is prohibitive. Further, recently conducted large scale simulations by the Cray Center of Excellence on a Cray XE6 supercomputer using a numerical fluid dynamics code showed poor performance of MPI one-sided communication compared to MPI two-sided communication [6].

3.5 Other CAF particle shift algorithms

We explored additional algorithms that extend the set of CAF particle shifters, but do not include them in the final analysis due to suboptimal performance. We briefly touch on them because we learned as much from these negative results as we did from the successful implementations.

First, in order to compare the performance of CAF and MPI communication for the specific communication requirements in GTS, we implemented a CAF shifter following exactly the *MPI-ms* algorithm but having CAF put operations instead of MPI function calls. In experiments using more than 4K processors we observed that the manually implemented CAF analogue to MPI_Allreduce did not perform as well as the MPI implementation. This clearly motivates the inclusion of optimized collective intrinsic subroutines to Fortran 2008, which are currently under development. Aside from those missing collective communication intrinsics we found that the CAF one-sided messaging performance was nearly indistinguishable from the *MPI-ms* implementation, which makes this approach not beneficial for the usage in GTS.

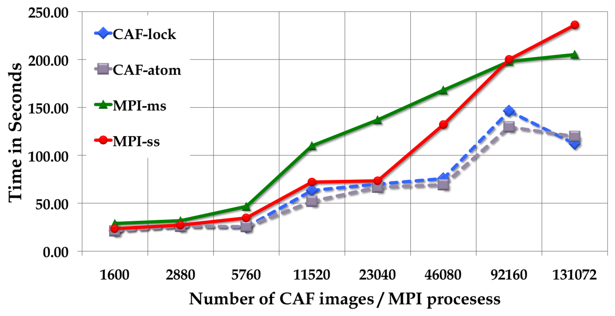
We also explored an alternative implementation that introduces a spatial sort of the particles using a fast incremental sorting technique to reduce the memory footprint and improve the spatial locality of the particle list to improve performance. In this approach we sorted the particle array in a) particles staying within the local toroidal domain followed by b) particles leaving the domain to the left and by c) particles, which will move to the right — no matter how far they have to drift. The fast particle sorting mechanism was based on the quicksort algorithm, which converges faster than a full *qsort()*, because the particle array only needs to be partially ordered. Thus, this algorithm implements a multi stage shifting strategy and breaks until the particle array only contains "non moving" particles. The remaining steps are similar to the *MPI-ms* algorithm (except for the MPI communication replaced by CAF put operations). By using the F90 array access notation the portion of particles moving to the left and to the right respectively can be directly accessed in the particle array — thus no send buffers are required, which reduces the memory footprint. Unfortunately this implementation exhibited poor performance due to the sorting preprocessing step, and because CAF put operations of the form $receive(1 : size)[dest] = p\_array(start : end)$ could not be properly vectorized for message aggregation by the current compiler implementation, which is currently being fixed. The inability of the compiler to vectorize this put operation made the performance of this approach impractically slow for integration to the GTS application.

In addition, we investigated a double send buffering strategy for the CAF implementations to allow larger overlap of particle work and communication; i.e., if a send buffer is fully filled, we started filling a second buffer while the first one being sent. This approach did not speed-up the performance at any scale and problem size.

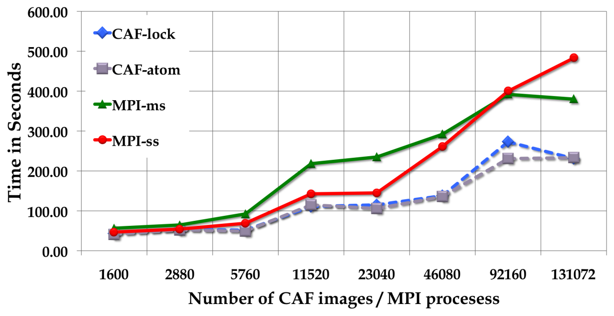
All particle shift algorithms are part of a stand alone *benchmark suite* simulating different particle shift strategies based on an artificial particle array with similar properties as the one in GTS. Having this compact application enables us to run straightforward scaling test in terms of machine and problem size on any available HPC platform.

4. Analysis

We evaluated the performance of our two advanced CAF particle shifters and compare them to the best available MPI shift algorithms on the Cray XE6 system at NERSC. Further, experiments are conducted on the Cray XT4 supercomputer for performance comparisons when no hardware support for CAF is given. First, we executed the standalone communication benchmark that we created to rapidly evaluate different implementations, and then we compare performance with the new communication method incorporated back into the GTS code to verify that the performance observed in the standalone benchmark will benefit the full application. The data was collected using GTS production run settings at increasing processor scales based on a weak scaling strategy, (i.e., keeping the number of particles constant at each processor as we scale up the parallelism). Each experiment in this section is based on a $(64/x)$ domain decomposition, i.e., using 64 toroidal domains, each having $x = N/64$ poloidal domains, where N denotes the number of processors in the actual run. It is worth mentioning that at each shift stage each processor sends data to a processor at least x ranks away due due to the toroidal communicator setup in the GTS initialization phase, which initializes MPI processes with successive ranks if they have equal toroidal communicator identifiers. A change of the instance's rank order would have a positive effect on the performance of the particle shift phase, but would cause a too high overhead for other PIC steps in GTS, which mainly operate on the poloidal communicator.



(a) Benchmark: 750K particles per processor



(b) Benchmark: 1500K particles per processor

Figure 4. Evaluation of two MPI shifter (*MPI-ms*, *MPI-ss*) and two CAF shifter (*CAF-atom*, *CAF-lock*) implementations with constant particle array sizes per processor (weak scaling) for each experiment (750K and 1500K particles) using the shifter benchmark suite

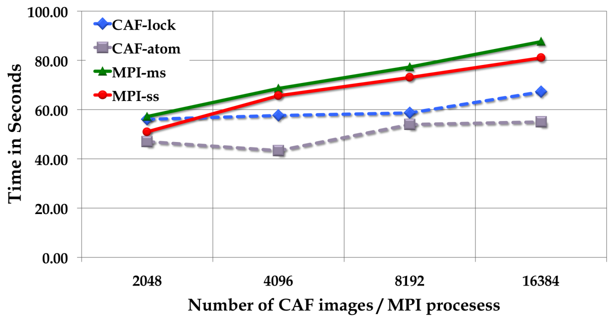


Figure 5. Execution of *MPI-ms*, *MPI-ss* and *CAF-atom*, *CAF-lock* with 1500K particles per processor in the shifter benchmark suite on up to 16K processor cores on Franklin (XT4)

The standalone communication benchmark simulates the particle shift phase using the same domain topology, same number of particles per processor and shift load size (that is, the number of particles being moved per shift iteration) observed in runs of the original GTS application. The shifter benchmark is implemented using data structures and interfaces that are identical to the full GTS code, to ensure the testing conditions (i.e., MPI process / CAF image placement and network load characteristics) are consistent across different implementations. This enables clean and consistent comparisons to be made between the presented algorithms that are also very easy to incorporate back into the original code.

Figure 4 presents the wallclock runtime on Hopper (XE6) of the MPI shift algorithms introduced in sections 3.1 and 3.2 and of the two CAF shifters (dashed lines) described in sections 3.3 and 3.4. Data was collected for concurrencies ranging from 1600 up to 131072 processor cores. Figure 4(a) shows the time measured for moving particles when each processors' particle array stores 750K particles from which 10% are moved to immediate neighbors (5% to each of the two adjacent neighbors) and 1% to the next but one direct toroidal neighbor domain. The artificially generated data accurately simulates the typical fraction and range of moving particles as occurring in GTS runs. Runtime numbers presented in Figure 4(b) are based on the same configuration, but correspond to a larger particle array containing 1500K elements. All three shift routines were executed 100 times in this experiment and simulate a constant "micell per processor" ratio[8], thus keeping the size of the particle array constant in each experiment with varying processor counts.

[8] In GTS *micell* denotes the number of particles per cell and needs to be increased for varying processor scales to insure weak scaling experiments.

Both CAF implementations substantially outperform the best MPI implementations, despite the years of work in profiling and optimization of the communication layer of the GTS MPI code. At the largest scale (131K processors) and 750K particles per process, the CAF implementations of 100 shift iterations take 112.0 seconds for *CAF-lock* and 119.7 seconds for *CAF-atom* compared to 205.2 seconds for *MPI-ms* and 236.3 seconds for *MPI-ss*. In the second case, 1500K particles per process, 100 particle shift iteration with the CAF algorithms take 231.5 seconds (*CAF-lock*) and 233.7 seconds (*CAF-atom*) as opposed to 380.2 seconds for 100 calls to *MPI-ms* and 482.9 seconds when *MPI-ss* is used on largest scale with 131K processors. For both problem sizes (750K and 1500K particles per process), we see slightly better CAF performance for low concurrencies ($\leq$ 23K processors), but the differences become much more apparent at higher concurrencies. A portion of the benefit comes from the lower overhead of Global Address Space communication since the initiator image always provides complete information describing the data transfer to be performed as opposed to transfers using MPI [3]. In addition, by exploiting the opportunities that a one-sided communication models offers, we can decouple synchronization from data transfers, and by building upon the previous observation of a more light-weight communication model, we prove that sending more frequent smaller messages enables the CAF approach to outperform the message passing implementations due to the enhanced communication and computation overlap as well as the better network bandwidth utilization.

In all performance optimization work, it is important to determine how much room there is for additional performance improvement. We conducted additional experiments using the shifter benchmark suite to measure the data transfer rate per compute node on Hopper (XE6)[9] to determine how close our implementation is to the interconnect bandwidth limit to see if there is any further opportunity for performance improvement. This brings the necessity to differentiate between three different factors that are limiting the performance of communication intense applications: a) *Injection bandwidth* reflects the rate at which data can be pushed into the network fabric, which is logically related to b) the *speed of links* between two Gemini ASICs. On the Cray XE6 the injection bandwidth is slower than the rate at which data can be exchanged using the links in X and Z dimensions between two Gemini ASICs, but higher than the bandwidth in Y direction between two Gemini ASICs. Note, one Gemini ASIC connects to its six nearest neighbor Gemini ASICs in X,Y and Z direction, where the links in X and Z direction are twice the width of the links in the Y direction.

[9] Data transfer rates per Gemini cannot be easily computed since it is not guaranteed that MPI processes with consecutive global MPI ranks residing on two different compute nodes are connected to the same Gemini.

Consequently, the Cray XE6 network is asymmetric where different links run at different speeds depending on the positions inside the network. c) In addition, the farther different messages travel on the network, the greater is the chance of resource sharing between messages. Thus, more than one message must traverse the same link reducing the available bandwidth along that link for each message, which leads to the phenomenon of *network contention*. Due to the long distances between a sending and receiving MPI process (because of the toroidal communicator setup as mentioned in the beginning of this section), especially for large scale simulations, we expect this problem to be limited by the speed of the slowest link in the network and by network contention and, only in rare cases, limited by the injection rate to push data on the Gemini ASIC. We assume the slowest link (in Y direction of the 3D torus network) runs at about 3.5 Gbytes/second under payload — this number agrees with a point-to-point bandwidth test on a Cray XE6 conducted by Vaughan et al [18]. In addition, there will likely occur contention for that slowest links due to communication from other compute nodes.

Computing the data transfer rate per node for a run with 8192 processing cores for the *MPI-ms* algorithm (replaced non-blocking MPI calls by blocking MPI functions to measure the communication time) using the shifter benchmark suite with 1500K particles per MPI process results in a data transfer rate per node ranging between 0.7 Gbytes/second and 2.5 Gbytes/second. This range of data transfer rates per node (note, two compute nodes are connected to one Gemini ASIC) agrees very well with the bandwidth of the weakest link in the torus as stated above; i.e., indicating that the communication in *MPI-ms* (and, hence from *MPI-ss* since roughly the same amount of data is being sent per MPI function call) is getting close to the network bandwidth limit. Raising the number of particles per MPI process to 3000K, and hence doubling the size of the MPI messages, yields the same number for data transfer rates per compute node, which leads to the conclusion that in both scenarios (1500K and 3000K particles per MPI process) the limits of the network bandwidth are roughly reached. In addition, if the number of used processing cores in a simulation run is increased (i.e., the difference between a sending and receiving MPI rank increases) we expect a decrease in the data transfer rate per node since as the job size increases, the average distance a message must travel on the network increases and thus the probability of network contention increases. This assumption is confirmed by simulations conducted with 3000K particles per MPI process using 16384 processor cores and using 32768 processors respectively. The data transfer rate per node for the 16384 processor cores experiment lies between 0.5 Gbytes/second and 1.9 Gbytes/second; and for the 32768 cores simulation the data transfer rate per node ranges from 0.4 Gbytes/second to 1.5 Gbytes/second.

Figure 5 shows the runtime of *MPI-ms*, *MPI-ss* and *CAF-lock*, *CAF-atom* executed 100 times with each processor holding 1500K particles on Franklin (XT4) for concurrencies ranging from 2048 up to 16384 processor cores. Both CAF implementations outperform the best available MPI implementations on both Hopper (XE6) and on Franklin (XT4), demonstrating that our implementation delivers a performance advantage across two very different Cray interconnect implementations. At the largest scale (16K processors) 100 particle shift iterations take 67.2 seconds for *CAF-lock* and 55.1 seconds for *CAF-atom* compared to 87.6 seconds for *MPI-ms* and 81.1 seconds for *MPI-ss*, which results in a 47.2% speed-up of the best CAF implementation over the best MPI particle shift implementation. The performance portability of our CAF implementations demonstrates the improved communication efficiency and network utilization that results from our algorithmic innovations and the benefits of one-sided communication models such as higher data transfer rates and better communication and computa-

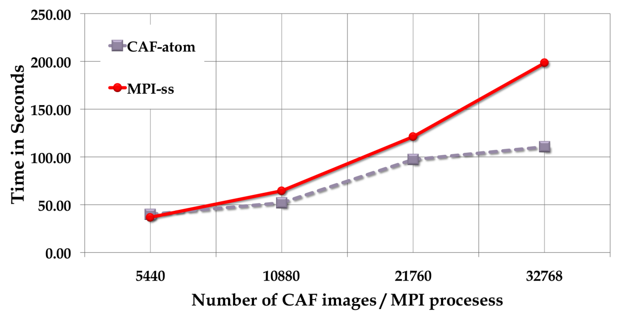


Figure 6. Weak scaling GTS experiments on Hopper (XE6) with CAF-atom & MPI-ss on up to 32K processors

tion overlap. We also observe a slightly better performance for each of the four algorithms executed on Franklin (XT4) compared to the execution on Hopper (XE6). This is related to the fact that the shift of particles in the GTS fusion application is a network bandwidth limited problem — each compute core on Franklin (XT4) has, on average, more network bandwidth available than a compute core on Hopper (XE6).

Figure 4, reflecting runs on Hopper (XE6) and also Figure 5 for experiments conducted on Franklin (XT4), demonstrate that *CAF-atom* is the most efficient CAF algorithm. The performance difference is due to the longer stall cycles in *CAF-lock*, when using locks for access restrictions to the buffer receiving moving particles. Nevertheless, costs for lock acquisition or for the remote atomics are less pronounced because both implementations are equally efficient at avoiding long stall cycles by keeping as many processors as possible active filling send buffers and communicating simultaneously. The reduction of idle time enables the sustained speed-up shown in Figure 4 and Figure 5. However, it has to be taken into account that remote atomics are Cray intrinsic operations, but this work should make the case to include them into the Fortran standard. The “lock” and “unlock” statements and the associated lock type in the *CAF-lock* algorithm are part of the base Fortran 2008 standard and ensure a portable implementation.

We then took the best-performing one- and two-sided particle shift algorithm and re-integrated it with the full GTS application code. The shifter benchmark suite suggests *MPI-ss* as best MPI shift implementation and *CAF-atom* as the most efficient particle shifter implementation representing the one-sided CAF programming model. Figure 6 highlights the performance improvements to the shifter PIC phase when running GTS with the new *CAF-atom* shifter in comparison to using *MPI-ss* as MPI implementation for a range of problem sizes. Timings from Figure 6 are for weak scaling experiments on Hopper (XE6) that use a constant particle array size of 750K per processor, which corresponds to a “micell per processor” ratio in GTS of 0.0781. Such a ratio is typical of most GTS production simulations. Various fusion device sizes are studied by varying the number of processes in a weak scaling fashion. A 32K processor simulation with this particle array size would allow the simulation of a large tokamak. Figure 6 shows the duration of the shift PIC step in four different runs of GTS with 100 time steps (i.e., 100 shifter function calls) as the number of processors per poloidal plane is varied. The number of toroidal domains is constant and set to 64, as is the case for all evaluated problem configurations. Each GTS experiment, using a total number of {5440, 10880, 21760, 32768} processors has been executed twice — using the *CAF-atom* shifter and the *MPI-ss* shift algorithm, respectively.

Figure 6 demonstrates that for the production mode of GTS using 32,768 processors the best CAF algorithm (*CAF-atom*) clearly outperforms the best MPI shifter implementation (*MPI-ss*), where 100 executions of the *CAF-atom* shifter take 110.6 seconds compared to 198.3 seconds when using *MPI-ss* – resulting in a 79% speed-up. The difference between runtime numbers of the shift phase from GTS and from the benchmark suite arise from additional computational costs to determine the toroidal position in the tokamak (function "compute_destination"), which is more computationally intense in the real application. Figure 6 confirms that the standalone shifter communication benchmark correctly predicts the performance benefits of the particle shift phase for the full application code. It remains to be seen whether other forms of communication in the GTS run (e.g., collective communication calls during the grid work in the scatter PIC phase; or MPI calls in PETSc) have an impact on the interconnect and shifter performance.

5. Conclusions

This work has demonstrated the performance potential of using a one-sided model for the communication intensive particle shift routine of the GTS magnetic fusion simulation code. The transition of charged particles between adjacent toroidal computational domains was originally implemented in MPI, and has been intensely studied and optimized over a number of years. Here, we develop novel communication algorithms using a one-sided messaging paradigm based on Coarrays, which are evaluated in a benchmark suite modeling the shift PIC phase between neighboring toroidal domains in the gyrokinetic simulation. The best performing Coarray algorithm is integrated into the GTS fusion simulation code, leaving the other PIC steps in GTS with MPI calls unchanged. The performance of the new Coarray implementations, which use more frequent smaller messages, is compared to the best MPI communication algorithms, which implement large bulk transfers to exchange moving particles. Benchmark runs on up to 131K processors and experiments with the "real world" physics application using up to 32K processors (scaling currently limited due to the PETSc numerical library in the initialization phase of the application, which is part of future research) on a Cray XE platform show that the performance of the particle shift phase can be improved by up to 83% on 131K processors in our benchmarking experiments and by up to 79% using 32K processors in the fusion application.

In future work we will discuss performance studies of multi-threaded Coarray particle shift algorithms that support the hybrid MPI and OpenMP model of the fusion application. Further, we plan a full transition from MPI to Fortran 2008 for the entire application following the stepwise approach undertaken so far.

Acknowledgments

A majority of the work in this paper was supported by the Petascale Initiative in Computational Science at NERSC. Some additional research on this paper was supported by the Cray Center of Excellence at NERSC. Additionally, we are grateful for interactions with Nicholas Wright (NERSC) and for the extended computer time as well as the valuable support from NERSC. This work was supported by the Director, Office of Science, Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] Piotr Bala, Terry Clark, and Scott L. Ridgway. Application of Pfortran and Co-Array Fortran in the parallelization of the GROMOS96 molecular dynamics module. *Scientific Programming*, 9:61–68, January 2001.

[2] Richard Barrett. Co-Array Fortran Experiences with Finite Differencing Methods, 2006. *48th Cray User Group meeting*, Lugano, Italy, May 2006.

[3] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, page 84, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Dan Bonachea. GASNet Specification, v1.1. Technical Report UCB/CSD-02-1207, University of California at Berkeley, Berkeley, CA, USA, 2002.

[5] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, 1:91–99, August 2004.

[6] K. A. Brucker, T. T. O'Shea, D. G. Dommermuth, J. Levesque, K. D. George, R. I. Walters, and M. M. Stephens. Numerical Flow Analysis. In *DoD High Performance Computing Modernization Program, 21st User Group Conference*, HPCMP, 2011.

[7] François Cantonnet, Yiyi Yao, Mohamed M. Zahran, and Tarek A. El-Ghazawi. Productivity Analysis of the UPC Language. In *Proceedings of the 18th International Conference on Parallel and Distributed Processing*, IPDPS'04, page 254, 2004.

[8] Cristian Coarfa, Yuri Dotsenko, Jason Eckhardt, and John Mellor-Crummey. Co-array Fortran performance and potential: An NPB experimental study. In *16th International Workshop on Languages and Compilers for Parallel Computing, LCPC*, pages 2–4. Springer-Verlag, Oct 2003.

[9] Cristian Coarfa, Yuri Dotsenko, and John Mellor-Crummey. Experiences with Sweep3D implementations in Co-array Fortran. *The Journal of Supercomputing*, 36:101–121, May 2006.

[10] Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A Multi-Platform Co-Array Fortran Compiler. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 29–40, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Tarek El-Ghazawi and Francois Cantonnet. UPC performance and potential: a NPB experimental study. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, SC'02, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[12] S. Ethier, W. M. Tang, R. Walkup, and L. Oliker. Large-scale gyrokinetic particle simulation of microturbulence in magnetically confined fusion plasmas. *IBM Journal of Research and Development*, 52(1/2):105–115, 2008.

[13] J. N. Leboeuf, V. E. Lynch, B. A. Carreras, J. D. Alvarez, and L. Garcia. Full torus Landau fluid calculations of ion temperature gradient-driven turbulence in cylindrical geometry. *Physics of Plasmas*, 7(12):5013–5022, 2000.

[14] John Mellor-Crummey, Laksono Adhianto, William N. Scherer III, and Guohua Jin. A new vision for Coarray Fortran. In *Proceedings of the 3rd Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 5:1–5:9, New York, NY, USA, 2009. ACM.

[15] Robert W. Numrich. Parallel numerical algorithms based on tensor notation and Co-Array Fortran syntax. *Parallel Computing*, 31:588–607, June 2005.

[16] Robert W. Numrich, John Reid, and Kim Kieun. Writing a Multigrid Solver Using Co-array Fortran. In *Proceedings of the 4th International Workshop on Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, PARA '98, pages 390–399, London, UK, 1998. Springer-Verlag.

[17] John Reid. Co-array Fortran for Full and Sparse Matrices. In *Proceedings of the 6th International Conference on Applied Parallel Computing Advanced Scientific Computing*, PARA '02, pages 61–, London, UK, 2002. Springer-Verlag.

[18] C.T. Vaughan, M. Rajan, D.W. Doerfler, R.F. Barrett, and K.T. Pedretti. Investigating the Impact of the Cielo Cray XE6 Architecture on Scientific Application Codes. In *International Workshop on Large-Scale Parallel Processing*, IPDPS, 2011.

[19] W. X. Wang, Z. Lin, W. M. Tang, W. W. Lee, S. Ethier, J. L. V. Lewandowski, G. Rewoldt, T. S. Hahm, and J. Manickam. Gyrokinetic Simulation of Global Turbulent Transport Properties in Tokamak Experiments. *Physics of Plasmas*, 13, 2006.