Multithreaded Global Address Space Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms

Robert Preissl
Lawrence Berkeley National Laboratory
Berkeley, CA, USA 94720
rpreissl@lbl.gov

Nathan Wichmann
CRAY Inc.
St. Paul, MN, USA, 55101
wichmann@cray.com

Bill Long
CRAY Inc.
St. Paul, MN, USA, 55101
longb@cray.com

John Shalf
Lawrence Berkeley National Laboratory
Berkeley, CA, USA 94720
jshalf@lbl.gov

Stephane Ethier
Princeton Plasma Physics Laboratory
Princeton, NJ, USA, 08543
ethier@pppl.gov

Alice Koniges
Lawrence Berkeley National Laboratory
Berkeley, CA, USA 94720
aekoniges@lbl.gov

ABSTRACT

We present novel parallel language constructs for the communication intensive part of a magnetic fusion simulation code. The focus of this work is the shift phase of charged particles of a tokamak simulation code in toroidal geometry. We introduce new hybrid PGAS/OpenMP implementations of highly optimized hybrid MPI/OpenMP based communication kernels. The hybrid PGAS implementations use an extension of standard hybrid programming techniques, enabling the distribution of high communication work loads of the underlying kernel among OpenMP threads. Building upon lightweight one-sided CAF (Fortran 2008) communication techniques, we also show the benefits of spreading out the communication over a longer period of time, resulting in a reduction of bandwidth requirements and a more sustained communication and computation overlap. Experiments on up to 130560 processors are conducted on the NERSC Hopper system, which is currently the largest HPC platform with hardware support for one-sided communication and show performance improvements of 52% at highest concurrency.

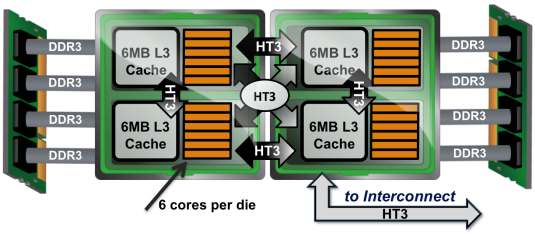
Keywords: Particle-In-Cell, Fortran 2008, Coarrays, Hybrid MPI/OpenMP & PGAS/OpenMP computing

1. INTRODUCTION

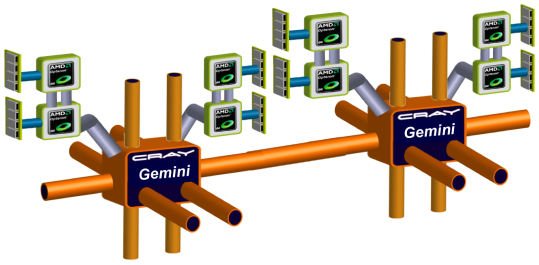
The path towards realizing next-generation petascale and exascale computing is increasingly dependent on building supercomputers with unprecedented numbers of processors and complicated memory hierarchies. Applications and algorithms need to change and adapt as node architectures evolve to overcome the 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 is the development of one-sided messaging methods and Partitioned Global Address Space (PGAS) languages such as Unified Parallel C (UPC) and Fortran 2008, which incorporates parallel features historically identified as Coarray Fortran (CAF). PGAS languages are able to directly reference remote memory as a first order construct, which reduces subroutine call overhead and enables the compiler to participate in optimization of the communication. 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 performance potential of these new approaches.

Our work focuses on advanced communication optimizations for the Gyrokinetic Tokamak Simulation (GTS) [16] 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 reduced or *skeleton* application that represents the communication requirements of the GTS application so that we could rapidly prototype and measure the performance of various communication strategies. The best strategy is then easily incorporated

[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.



(a) XE6 compute node with AMD "Magny Cours"



(b) Part of the 3D torus network in the XE6

Figure 1: The Cray XE6 supercomputer incorporates AMD's twelve-core "Magny Cours" Opteron processors and the Gemini interconnect in a three-dimensional (3D) torus network

back into the original code, where we evaluate its benefit to the overall scalability and performance of GTS. We focus on Fortran 2008's CAF extensions 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] studied different Coarray Fortran implementations of Finite Differencing Methods and Numrich et al. [13] developed a Coarray enabled Multigrid solver focusing primarily on programmability aspects. Bala et al. [1] demonstrate 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 [12]) and matrices (Reid [15]) 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.

Mellor-Crummey et al. [11, 8] have proposed an alternate design for CAF, which they call Coarray Fortran 2.0, that adds some capabilities not present in the standard Fortran version, but also removes some capabilities. Their CAF2.0 compiler uses a source to source translator to convert CAF code into Fortran 90 (F90) with calls to a low-level one-sided communication library such as GASNet [4]. Applications of Coarray Fortran to the NAS parallel benchmarks (Coarfa et al. [6]) and to the Sweep3D neutron transport benchmark (Coarfa et al. [7]) show nearly equal or slightly better performance than their MPI counterparts.

Our work makes the following contributions. We use a "real world" application (GTS) to demonstrate the performance potential of our different optimization strategies. We created a compact skeleton application that enables rapid comparison of alternative communication representations. We created highly optimized hybrid MPI/OpenMP and hybrid PGAS/OpenMP implementations to compare competing strategies for improving the scalability and performance of the GTS code. We demonstrate the scalability of our new approaches on up to 126720 processing cores for the skeleton application and on runs of the full GTS code using up to 130560 processors. The fastest overall approach is the hybrid PGAS/OpenMP model, which attains up to 52% performance improvement over competing approaches at the largest scale. Overall, our work is the first demonstration of sustainable performance improvements using a hybrid PGAS/OpenMP model and constitutes the largest production PGAS application demonstrated thus far on a PGAS-enabled hardware.

1.2 Hardware Platform

The platform chosen for this study is a Cray XE6 supercomputer capable of scaling to over 1 million processor cores with AMD's "Magny Cours" Opteron processors and Cray's proprietary "Gemini" interconnect. 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 dies that each contain six cores, for a total of 24 cores on the compute node. Each die is directly attached to a low latency, high bandwidth memory channel such that each of the six cores on that die form a NUMA-node that provides equal, effectively flat, access to the directly attached memory. This makes the die (the NUMA node) the natural compute unit for shared memory programming. GTS currently uses a hybrid MPI/OpenMP model that exploits this NUMA model on the node.

Figure 1(b) outlines how compute nodes on a Cray XE6 system are interconnected 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 by most two-sided communication models.

All experiments shown in this paper are conducted on the Hopper Cray XE6 system installed at the National Energy Research Scientific Computing Center (NERSC), which is comprised of 6392 nodes with 153408 processor cores and a system theoretical peak performance of 1.288 Petaflops. Hopper holds the 5th position on the November 2010 Top500 list of largest supercomputing sites with a reported HPL performance of 1.05 Petaflops.

The Cray Compiler Environment (CCE) version 7.3.3 is used to compile all of the source code for this paper. CCE 7.3.3 fully supports the Coarray feature of Fortran 2008 translating all Coarray references into instruction sequences that access the Gemini interconnect's hardware mechanisms for efficient one-sided messaging.

2. THE GTS FUSION SIMULATION CODE

GTS is a general geometry PIC code developed to study plasma microturbulence in toroidal, magnetic confinement

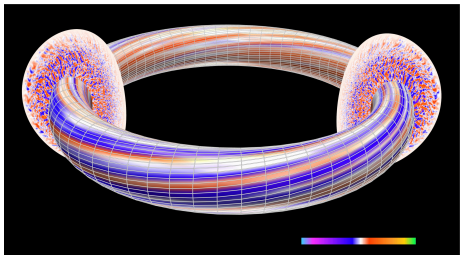


Figure 2: GTS field-line following grid & toroidal domain decomposition. Colors represent isocontours of the quasi-two-dimensional electrostatic potential

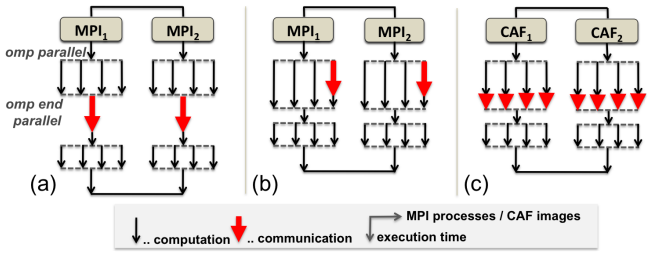


Figure 3: Hybrid parallel programming models as used in the particle shift algorithms

devices called tokamaks [16]. Microturbulence is a complex, nonlinear phenomenon that is believed to play a key role in the confinement of energy and particles in fusion plasmas [9], 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. 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 the GTS application consists of three levels: **(1)** GTS implements 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 [9]. **(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. Hence, GTS production runs will be conducted in a hybrid MPI/OpenMP mode, which motivates the design of multithreaded particle shift algorithms.

Figure 2 shows the GTS grid, which follows the field lines of the externally applied magnetic field as they twist around the torus[2]. 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 algorithm.

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 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. 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 time step-size).

In the following sections we will introduce two optimized algorithms for MPI two-sided messaging and a PGAS one-sided implementation for the particle shift phase in GTS. The first MPI implementation extends the classical hybrid MPI/OpenMP programming model (Figure 3(a)) as used in GTS where MPI processes create OpenMP thread teams for work distribution and join the team for serialized execution such as MPI communication calls and enables the main OpenMP thread to make collective MPI function calls

[2]The two cross sections demonstrate contour plots of potential fluctuations driven by Ion Temperature Gradient-Driven Turbulence (ITGDT) [10], which is believed to cause the experimentally observed anomalous loss of particles and heat in the core of magnetic fusion devices such as tokamaks.

while other threads perform computation (Figure 3(b)). The hybrid PGAS/OpenMP algorithm builds on this strategy of communicating threads, but allows *all* OpenMP threads per team to make communication calls to the thread-safe PGAS communication layer (Figure 3(c)).

3.1 The MPI multi stage shifter (MPI-ms)

MPI-ms is an optimized version of the original hybrid MPI/OpenMP shift algorithm in GTS and models the shift of particles to adjacent or further toroidal domains of the tokamak in a ring-like fashion. MPI-ms is based on the advanced MPI/OpenMP hybrid model (Figure 3(b)) and implements a novel methodology to overlap collective MPI communication with computation using OpenMP tasking. MPI-ms 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[3]. 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 MPI process spawns multiple OpenMP threads, which iterate through their segments of the local particle array (*p_array*) and compute which particles have to be shifted to the left and to the right, respectively (privatized data will be merged into shared arrays before the initial MPI process joins the OpenMP threads). This procedure is summarized in function "dest", which 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*. Each MPI process traverses the whole particle array at the first shift stage and for consecutive stages only newly received particles are considered in function "dest". Note, that remaining and shifted particles in *p_array* are randomly distributed and a prior sorting step would involve too much overhead. **(2)** Pack particles, which have to be moved to their left- and right immediate toroidal neighbor into *send_right* and *send_left* buffers. **(3)** At every shifting stage the sum of shifted particles is communicated to all processes within the same toroidal communicator (*tor_comm*, limited size of 128 MPI processes) by an allreduce call. This denotes the break condition of the shifter, i.e., to exit 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 because shifts of particles happen in every iteration of GTS. Since the packing of particles in (2) is independent of the MPI_Allreduce call, we can overlap communication (allreduce) with computation (particle packing): Only the master thread (Line 8 from Listing 1) encounters the tasking statements and creates work for the thread team for deferred execution; whereas the MPI_Allreduce call will be immediately executed, which gives us the overlap as highlighted in Figure 3(b)[4]. The particle packing of right and left shifted particles is subdivided into many tasks (each task is packing *chk_size* many particles) to guarantee a load balanced computation and to enable the master thread performing

```
do shift_stages=1,N                                              1
!(1) compute right- & left-shifted particles
  !$omp parallel                                                 3
  dest(p_array,shift_r,shift_l,holes_r,holes_l)
                                                                 5
!(2) pack particle to move right and left
  !$omp parallel                                                 7
  !$omp master
  do i=1,chunks_right                                            9
    !$omp task
    pack(i,chk_size,send_right,shift_r,holes_r)                  11
  enddo
  do j=1,chunks_left                                             13
    !$omp task
    pack(j,chk_size,send_left,shift_l,holes_l)                   15
  enddo
                                                                 17
!(3) communicate amount of shifted particles
  if(shift_stages.ne.1) {                                        19
    MPI_ALLREDUCE(shift_r+shift_l,all,tor_comm)
    if(all.eq.0) { return } }                                    21
  !$omp end master
  !$omp end parallel                                             23

!(4) Nonblocking receive requests                                25
  MPI_IRECV(recv_left,left_rank,reqs_1(1),..)
  MPI_IRECV(recv_right,right_rank,reqs_2(1),..)                  27

!(5) reorder remaining particles: fill holes                     29
  fill_holes(p_array,holes_r,holes_l)
                                                                 31
!(6) send particles to right and left neighbor
  MPI_ISEND(send_right,right_rank,reqs_1(2),..)                  33
  MPI_ISEND(send_left,left_rank,reqs_2(2),..)
                                                                 35
!(7) add received particles and reset bounds
  MPI_WAITALL(2,reqs_1,..)                                       37
  !$omp parallel
  add_particles(p_array,recv_left)                               39
  MPI_WAITALL(2,reqs_2,..)
  !$omp parallel                                                 41
  add_particles(p_array,recv_right)
enddo                                                            43
```

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

computation (in case there are remaining tasks in the task pool) after having finished the allreduce communication. It is worth mentioning that in case no particles have to be shifted no computational tasks will be created, and therefore no extra overhead is caused by simultaneously packing particles and testing the break condition. This methodology has been proven efficient to hide the costs of the collective function call [14].**(4)** 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. **(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 (using OpenMP work-sharing) particles received from left and right neighbors to *p_array*.

The shifter routine involves heavy communication especially because of the particle exchange implemented using a ring-like send & receive functionality. In addition, intense

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

[4]Note, that the underlying MPI implementation has to support at least MPI_THREAD_FUNNELED as threading level in order to allow the main thread make MPI calls.

computation is involved mostly because of the particle reordering that occurs after particles have been shifted and incorporated into the new toroidal domain respectively.

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) Receive requests are pre-posted 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 preallocated two-dimensional send buffer, which keeps records of shifted particles for each possible destination. Function "pack" summarizes this process. We use OpenMP worksharing constructs to distribute the work among the team members. 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, received particles are used to fill the positions of shifted particles (*shift* many) in the particle array. This computationally intensive step is parallelized using OpenMP. If there are more received particles then 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 hybrid MPI/OpenMP versions for the shift of particles were implemented. We studied different algorithms, other OpenMP threading or MPI communication techniques (e.g., MPI 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-atomic shifter (CAF-atom)

The design of the *CAF-atom* single stage shifter is similar to the *MPI-ss* algorithm from above, but 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 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. The efficiency of this is novel algorithm is based on three factors: (a) It enables the overlap of particle work and particle communication since particle transfers can take place without synchronization between the sending and receiving images. (b) It turns out to be very effective for such bandwidth limited problems to spread out the communication over a longer period of time. This is achieved by pipelining of smaller light-weight messages, which do not require any ordering on the remote image. (c) We distribute the computational work as well as parts of the communication loads among OpenMP threads as shown in Figure 3(c).

```
!(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
!$omp parallel
pack(p_array,shift,holes,send_buf)

!(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
!$omp parallel do
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)

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 mechanism for safely shifting particles to the receiving queue Coarray. It turns out 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,

```
!(1) compute shifted particles and fill the
!  receiving queues on destination images
!$omp parallel do schedule(dynamic,p_size/100)&
!$omp private(s_buf,buf_cnt) shared(recvQ,q_it)
do i=1,p_size
  dest=compute_destination(p_array(i))
  if(dest.ne.local_toroidal_domain) {
    holes(shift++)=i
    s_buf(dest,buf_cnt(dest)++)=p_array(i)
    if(buf_cnt(dest).eq.sb_size) {
      q_start=afadd(q_it[dest],sb_size)
      recvQ(q_start:q_start+sb_size-1)[dest] &
        =s_buf(dest,1:sb_size)
      buf_cnt(dest)=0  }  }
enddo

!(2) shift remaining particles
empty_s_buffers(s_buf)
!$omp end parallel

!(3) sync with images from same toroidal domain
sync images([my_shift_neighbors])

!(4) fill holes with received particles
length_recvQ=q_it-1
!$omp parallel do
do m=1,min(length_recvQ,shift)
  p_array(holes(m))=recvQ(m)
enddo

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

Listing 3: CAF-atom shifter routine

is securely (i.e., avoiding data races) given, the sending image can start and complete its particle shift at any time. 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 are added to the local particle array and completion of any communication can be ensured. The process of claiming the required space in the receive queue on the remote destination image is performed by a Cray intrinsic global atomic memory operation, 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 3 outlines the major steps of this multithreaded CAF algorithm with global AMOs in pseudo code form using CAF and F90 array notation:

(1) Each CAF image creates multiple threads, which operate on their designated portion of the particle array and extract particles moving out of their current toroidal domain. The OpenMP "dynamic" scheduling option is used to divide *p_size* many particles of the particle array into chunks of size *p_size*/100, which enables a dynamic assignment of iterations to OpenMP threads at runtime. Moving particles are copied into a privatized send buffer (*s_buf*) and later transmitted to a shared (one per CAF image) receive

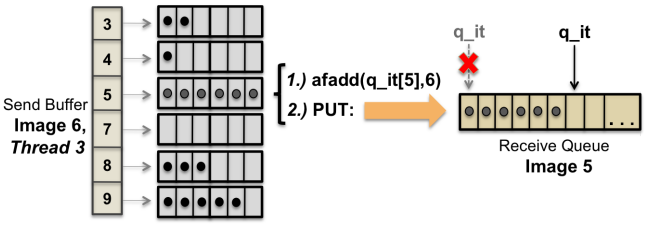


Figure 4: Shift of particles in *CAF-atom*

buffer. The position of those shifting particles will be held in a privatized buffer *holes*, which will be merged into a shared buffer before the OpenMP threads join. We maintain the two dimensionality of the send buffer, as discussed in the the *MPI-ss* approach, which enables a single stage shift algorithm. Consequently, each thread created from a CAF image assigns each moving particle to a corresponding set of particles based on its destination. If a thread's send buffer counter (*buf_cnt*) for a specific destination image *dest* indicates that the specific buffer is fully filled (i.e., a certain threshold size limit *sb_size* is reached; Line 10), a global AMO — in detail, a **global atomic fetch and add operation: "afadd"** — is executed (Line 11), which updates the remote, shared accessible queue iterator (*q_it*) on image *dest* by adding the number of moving particles to the queue iterator and returns the former value of the queue iterator (return value is held in *q_start*). The sending thread then launches a CAF put operation (Lines 12/13) to transfer *sb_size* many particles to image's *dest* shared receiving queue (*recvQ*) starting from position *q_start* to position *q_start+sb_size-1*. Due to the one-sided nature of the put operation, the sending thread does not need to wait for any acknowledgement from the destination image. As soon as the data is sent it resets the send buffer counter for the destination image *dest* and starts adding new moving particles to the send buffer. No handshake for transmitting moving particles between the sending image's thread and destination image is necessary.

Figure 4 illustrates an example scenario during the execution of *CAF-atom*, where the send buffer (shown for image 6 / OpenMP thread 3) can hold up to six particles (i.e., $sb\_size = 6$) for each possible send destination (images 3,4,5 to the left and images 7,8,9 to the right). After inserting the 6^{th} particle designated for destination image 5 the buffer for this destination is fully filled (*buf_cnt(5)*=6) and thread 3 of image 6 reserves a slot at the remote receive queue to safely transfer the 6 moving particles. Assuming no particle has been shifted yet to image 5 by any neighboring image, the global "afadd" AMO executed on the remote queue iterator (*q_it*) shifts the iterator six positions to the right (i.e., sets it to 7) and returns 1 as the old value for the position of the queue iterator. This thread safe AMO enables the OpenMP thread now to put the six moving particles to the receive queue of image 5, which execution can be deferred until a particle will be sent to the same destination image again; i.e., until it is safe to overwrite particles from the same send buffer location.

Experiments to determine optimal buffer size threshold values carried out for several problem sizes suggest $500 < sb\_size < 1000$ as optimal values for singlethreaded runs of *CAF-atom*. A too small setting of *sb_size* causes extra remote atomic operations and other communication over-

head 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. Adding OpenMP threads, however, adds an additional layer of complexity, which can cause contention for the access to the network adapter due to the high frequency of communication operations when using the *sb_size* settings from above. This reveals that the PGAS communication library is thread safe, but contains thread serialized code regions, which are a focus of current research at Cray. Hence, optimal values for *sb_size* depend on the number of OpenMP threads used and will be set at runtime. Optimal values for *sb_size* are roughly the optimal value for the singlethreaded run multiplied by the number of threads, which keeps the accesses to the network adapter per CAF image balanced, no matter how many threads are used.

In addition, experiments have shown that the more flexible OpenMP dynamic scheduling clause is effectively minimizing the load imbalance (moving particles are not evenly distributed in the particle array) among the OpenMP threads.

(2) If there are remaining particles of any thread's 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. **(3)** Besides the required remote AMOs 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 an 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[5]. The array *my_shift_neighbors* stores the global CAF indices of images within reach of the local images, i.e., as discussed before in the *MPI-ss* algorithm it is an array with six entries. The total number of received particles (*length_recvQ*) is the local queue length (note, *q_it* equals *q_it*[THIS_IMAGE()][6]), which has been successively updated by neighboring images. Adding received particles to the local particle array happens in **(4)** and **(5)**, which is analogous to (4) and (5) in *MPI-ss*.

For simplicity Coarrays in Listing 3 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.

This algorithm fully exploits 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.

It has to be taken into account that the used global AMOs are Cray intrinsic operations, but this work should make the case to include them into the Fortran standard. A similar CAF algorithm to the *CAF-atom* approach exists, which follows the same strategy of distributing computational and communication loads among OpenMP threads as presented in Listing 3, but uses "lock" and "unlock" statements to ensure safe updates of the receive queue iterator and therefore protects the receive queue from any write conflicts. Experiments have shown that the less elegant version using "locks" exhibits slightly longer stall cycles, which results in an overhead of 5% to the overall runtime compared to the *CAF-atom* approach. However, the "locks" and the associated lock types are part of the base Fortran 2008 standard and ensure a portable implementation.

3.4 Other CAF particle shift algorithms

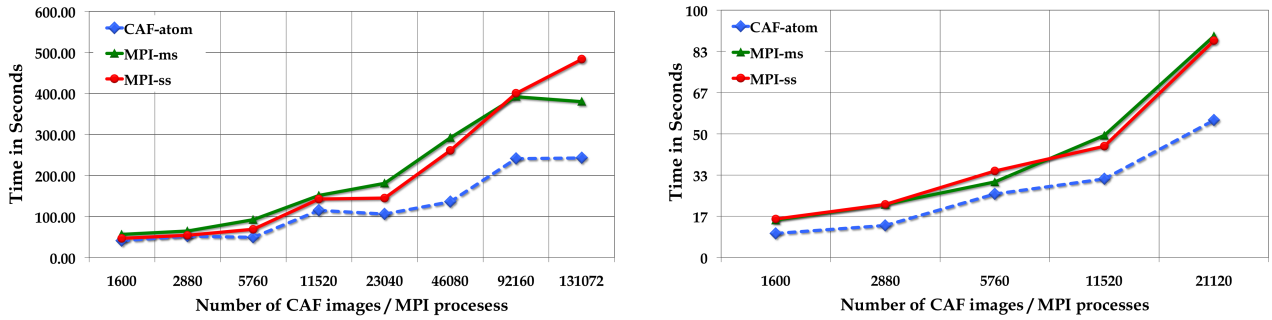
We explored additional algorithms that extend the set of CAF particle shifters, but do not include them in the final analysis due to suboptimal performance. We briefly touch on them because we learned as much from these negative results as we did from the successful implementations.

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 the next Fortran standard, 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 advantageous 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 $receive(1 : size)[dest] = p\_array(start : end)$

[5]"Executing "sync images" implies execution of "sync memory", which guarantees to other images that the image executing "sync images" has completed all preceding accesses to Coarray data.

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



(a) 1 OpenMP thread per instance

(b) 6 OpenMP threads per instance

Figure 5: Weak scaling benchmarks of the CAF shifter (*CAF-atom*) and two MPI shifter (*MPI-ms*, *MPI-ss*) implementations with no (a) and full (6 OpenMP threads per NUMA node) OpenMP support (b)

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.

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 advanced CAF particle shifter (*CAF-atom*) and compare it to the best available MPI algorithms (*MPI-ms*, *MPI-ss*) on the Cray XE6 system at NERSC. GTS production runs will be conducted in a hybrid distributed / shared memory mode with a maximum number of OpenMP threads per NUMA node because of optimal memory utilization and performance. Hence, scaling studies of our CAF algorithm and the MPI shifter implementations run with a constant setup of 6 OpenMP threads per instance[7]. In addition, we are interested in the performance characteristics in regard to varying number of OpenMP threads (i.e., keeping the number of instances constant per experiment) to analyze the quality of the OpenMP support and to gain insights for further optimization potential concerning software and hardware.

First, we executed the standalone communication benchmark that we created to rapidly evaluate different implementations, and then we compare the performance with the new communication methods incorporated back into the GTS code to verify that the performance observed in the standalone benchmark will benefit the full 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., placement of instances 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.

The artificially generated data accurately simulates the particle shift phase using the same domain topology, same number of particles per instance, shift load size (that is, the number of particles being moved per shift iteration) and range of moving particles as occurring in GTS production runs. All the data presented in this section was collected at increasing processor scales based on a weak scaling strategy, (i.e., keeping the number of particles constant at each instance as we scale up the parallelism). Each experiment is based on a $(64/x)$ domain decomposition, i.e., using 64 toroidal domains, each having $x = i/64$ poloidal domains, where i denotes the number of instances in the actual run. It is worth mentioning that at each shift stage an instance with rank i communicates with an instance with rank $i + x$ due due to toroidal communicator setup in the GTS initialization phase, which initializes MPI processes[8] with successive ranks if they have equal toroidal communicator identifiers. In case of running with several OpenMP threads per instance, the physical distance on the network between two neighboring instances increases dependent on the number of OpenMP threads used. 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.

Figure 5 presents the wallclock runtime of the *CAF-atom* shifter implementation (dashed line) described in section 3.3 and of the MPI shift algorithms introduced in sections 3.1 (*MPI-ms*) and 3.2 (*MPI-ss*), running with no OpenMP support (Figure 5(a)) and full (i.e., 6 OpenMP threads per instance, on each NUMA node) OpenMP support (Figure 5(b)). Data for the singlethreaded experiments was collected for concurrencies ranging from 1600 up to 131072 processor cores. For the multithreaded runs we run on 9600 up to 126720 processor cores on the Cray XE6 machine. All runtime numbers presented in Figure 5 are based on weak scaling experiments using the shifter benchmark suite where each instance stores an initial set of 1500K 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. All three shift routines were

[7]In the following, we will refer to the CAF images running *CAF-atom* or the MPI processes executing *MPI-ms* or *MPI-ss* as "instances".

[8]We only 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.

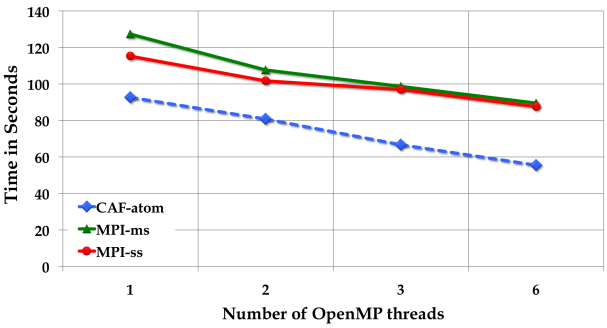
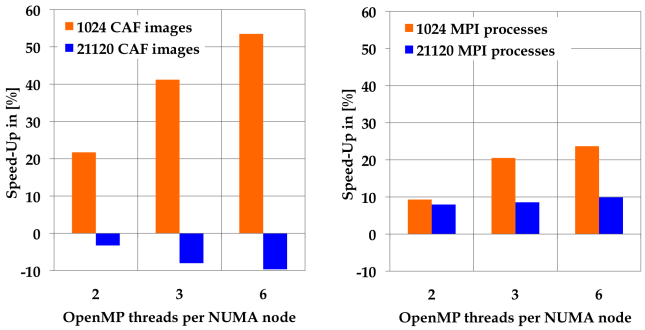


Figure 6: 21120 instances of *CAF-atom*,*MPI-ms* and *MPI-ss* with increasing number of OpenMP threads



(a) *CAF-atom* core-idling (b) *MPI-ms* core-idling

Figure 7: OpenMP scaling tests using 6144 and 126720 processor cores, respectively, with always 1 instance per NUMA node. The vertical axis denotes the speed-up compared to a singlethreaded run (thus, 5 idle cores per die)

executed 100 times in this experiment and simulate a constant "micell per processor" ratio[9], thus keeping the size of the particle array constant per instance in each experiment with varying processor counts. We can observe in both tests — for the singlethreaded and the multithreaded runs of the shifter benchmark suite — a steady increase in runtime for shifting particles in a torus with increasing concurrencies, whereas one would expect a flat line along the x-axis for weak scaling experiments. This motivates to optimize this communication intense GTS step to enable higher concurrencies as planned to model future nuclear fusion devices. Figure 5 shows that in both cases the CAF implementation substantially outperforms the best MPI implementations, despite the extensive work in profiling and optimization of the communication layer of the GTS code. At the largest scale in the singlethreaded experiment (131K processors) 100 particle shift iterations with the *CAF-atom* algorithm take 233.7 seconds as opposed to 380.2 seconds for 100 calls to *MPI-ms* and 482.9 seconds when *MPI-ss* is used. We see slightly better CAF performance for low concurrencies ($\leq$ 23K processors) within the singlethreaded runs, but the differences become much more apparent at higher concurrencies. For the multithreaded runs of the shifter benchmark suite we see performance improvements of CAF over the optimized MPI algorithms already starting from lowest concurrencies, which become more significant at largest scale (21120$*$6 processor cores), where 100 shift iterations take 55.5 seconds for *CAF-atom* compared to 89.4 seconds for *MPI-ms* and 87.6 seconds for *MPI-ss* — each running instance of the code is equipped with 6 OpenMP threads per instance. This results in a 58% speed-up of the CAF implementation over the best multithreaded MPI shifter algorithm on largest scale.

A portion of the benefit comes from the lower overhead of Global Address Space communication since the initiator instance always provides complete information describing the data transfer to be performed as opposed to transfers using MPI [3]. The Cray Fortran 2008 compiler can schedule data transfers asynchronously, which is fully exploited in the *CAF-atom* particle shift algorithm. For the AMO-controlled injection of particles in *CAF-atom*, the data transfers are always asynchronous from the perspective of the receiving instance. Hence, by exploiting the opportunities that the CAF one-sided communication models offers, we can decouple synchronization from data transfers. By building upon the previous observation of a more light-weight communication model, we also 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. Employing a similar strategy of transmitting smaller more frequent messages is not practical in MPI and would require significant efforts in MPI-2, which would obscure semantics and science due to its semantic limitations [5].

In order to analyze the quality of the OpenMP support, for the next experiments we keep the number of instances constant and vary the number of OpenMP threads each instance creates. Figure 6 shows the performance of *CAF-atom*, *MPI-ms* and *MPI-ss* running with a constant number of 21120 instances where OpenMP threads are successively added, — starting from 1 OpenMP thread going up to 6 OpenMP threads per instance — which requires increasing machine resources. For each OpenMP threading level the CAF particle shift algorithm significantly outperforms both optimized MPI implementations. For each algorithm we observe runtime improvements as we scale up the number of OpenMP threads, which is related to OpenMP worksharing constructs, but also related to increasing system resources (e.g. more memory per instance) with higher OpenMP threading levels. However, imperfect OpenMP scaling requires further investigation.

In Figure 7 we perform a similar test as the one from above, but always place just one instance on the NUMA node and increase the number of OpenMP threads. These experiments involve the idling of processor cores (if number of OpenMP threads per instance $\neq$ 6) and guarantees that the same system resources are used for each run. Figure 7(a) shows the achieved speed-up of the *CAF-atom* particle shifter with increasing OpenMP threading support for two different scenarios: running *CAF-atom* on small (1024 instances) and large scale (21120 instances). In each case, we run *CAF-atom* 100 times on a particle array of size 1500K and vary the *sb_size* threshold value (i.e., the critical value for the number of particles per send buffer entry to initiate a transfer) based on the number of OpenMP threads

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

per instance — we set $sb\_size = 512*$ "number of OpenMP threads per instance". The latter ensures a steady message injection rate per compute node over time. The height of each bar shown in Figure 7(a) denotes the speed-up using $\{2, 3, 6\}$ OpenMP threads per instance compared to the singlethreaded execution, which leaves 5 compute cores idle per NUMA node. With the maximum number of 6 OpenMP threads we observe a speed-up of 53% with 1024 instances, but a slow down of 9% with 21120 CAF instances. Note, that the shift of particles is highly communication bound, especially on large scale — analysis on high concurrencies have shown that around 80% of the runtime is due to communication. The fact that the achieved speed-up is higher of what can solely be gained through pure computation workload distribution encourages the extended multithreaded communication model implemented in the *CAF-atom* particle shifter. However the negative speed-up for runs on largest scale shows that the successfully applied message injection rate regulation is not sufficient to achieve better performance with increasing OpenMP threads. At larger concurrencies we suspect that contention in the communication fabric inhibits our ability to inject messages at a higher rate. Higher concurrencies also imply additional instances per toroidal plane since the number of toroidal planes has to be constant. This results in larger physical distances between sending and receiving instances on the network and therefore enhanced traffic. Thus, we see no advantage from running with multiple threads, and in fact slow down because the PGAS communication library is "thread safe", but not "thread optimal" due to thread critical regions for message transfers. Figure 7(b) reveals the relatively low speed-up achieved due to OpenMP threading in case of the *MPI-ms* particle shifter running with 1024 and 21120 instances, respectively. The measured speed-up with 6 OpenMP threads compared to the singlethreaded execution denotes to 23% for the small scale test and to 10% for the large scale scenario.

We then took *CAF-atom* and one of the two-sided multithreaded particle shift algorithms *MPI-ms* to be re-integrated with the full GTS application code. Figure 8 shows the runtime for the particle shift phase in GTS with the new *CAF-atom* shifter in comparison to using *MPI-ms* as MPI implementation for a range of problem sizes. Timings from Figure 8 are from weak scaling experiments using 6 OpenMP threads per instance where each instance stores 750K particles, 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 instances in a weak scaling fashion. A 32K processor simulation with this particle array size would allow the simulation of a large tokamak. Figure 8 shows the duration of the shift PIC step in five different runs of GTS with 100 time steps (i.e., 100 shifter function calls for each experiment) as the number of instances 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 uses a total number of $6*$ {1024, 2048, 5440, 10880, 21760} processing cores on the Cray XE6.

Figure 8 demonstrates that for running GTS in production mode on up to 130560 processors the *CAF-atom* particle shift algorithm clearly outperforms the *MPI-ms* shifter implementation at each level of concurrency. At largest scale 100 executions of the *CAF-atom* shifter take 36.1 seconds

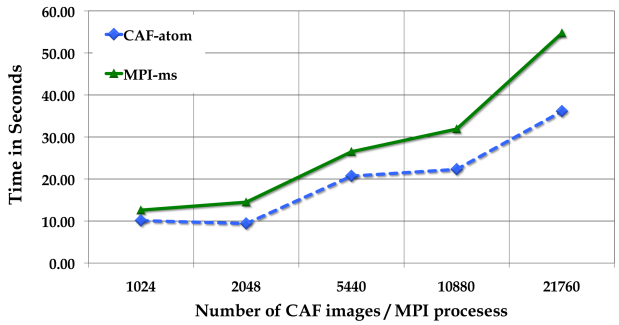


Figure 8: Weak scaling GTS experiments with CAF-atom & MPI-ms as particle shift algorithms (6 OpenMP threads per instance)

compared to 54.7 seconds when using *MPI-ms* — resulting in a 52% speed-up. Figure 8 confirms that the standalone shifter communication benchmark correctly predicts the performance benefits of the particle shift phase for the full application code.

5. CONCLUSIONS

New programming model techniques that enable effective use of highest-end computing platforms are critical for the path to exascale. In this paper we show that a hybrid PGAS/OpenMP approach can cleverly exploit hardware-supported interconnect technologies and enable the development of highly optimized algorithms. To the best of our knowledge, this work presents a first documented use of PGAS techniques to speed up a "real world" application on more than 100000 cores. We show that effective use of a first generation Cray-supported interconnect yields significant performance gains for the communication intensive particle shift routine of a hybrid MPI/OpenMP magnetic fusion simulation code. Using one-sided semantics enables the performance improvements because of the inherent advantages of a one-sided model to allow data transfers to take place without synchronization between the sending and receiving images. In our application, the shift of charged particles between neighboring toroidal computational domains in the PIC process was originally implemented in MPI/OpenMP, and has been intensely studied and optimized over a number of years. We show significant performance advantages using a novel hybrid PGAS/OpenMP communication algorithm, which distributes the computational as well as the communication work load among OpenMP threads. By sending more frequent smaller messages to adjacent toroidal domains — which is in contrast to the favored strategy of message passing algorithms of using message aggregation and large bulk transfers — we successfully reduce network contention by spreading out the intense communication over time. We also use the concept of building a *skeleton* application to quickly evaluate different programming models as an efficient means of comparing different formulations. Evaluation in a benchmark suite on up to 126720 processing cores and experiments with the "real world" simulation code using up to 130560 processors on a Cray XE6 platform show that the performance of the particle shift phase can be improved by 58% in our benchmarking experiments and by 52% in the physics application at largest scale.

6. 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] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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 Programming Models*, PGAS '09, pages 5:1–5:9, New York, NY, USA, 2009. ACM.

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

[13] 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.

[14] Robert Preissl, Alice Koniges, Stephan Ethier, Weixing Wang, and Nathan Wichmann. Overlapping communication with computation using OpenMP tasks on the GTS magnetic fusion code. *Scientific Programming*, 18:139–151, August 2010.

[15] 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.

[16] 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.