Class II Submission to the HPC Challenge Award Competition Coarray Fortran 2.0

John Mellor-Crummey, Laksono Adhianto, Guohua Jin,
Mark Krentel, Karthik Murthy, William Scherer, and Chaoran Yang
Department of Computer Science
Rice University
Houston, TX, USA

1 General Description

We implemented four HPC Challenge benchmarks: Global HPL, Global RandomAccess, EP STREAM (Triad), and Global FFT in Coarray Fortran 2.0; in addition, we implemented an extra benchmark — Unbalanced Tree Search (UTS) [9]. Coarray Fortran 2.0 is an extension of Coarray Fortran developed at Rice University to improve expressiveness and performance by adding features such as teams, collectives, event-based synchronization, and asynchronous operations (including copy, collective operations, and function shipping) [7, 15]. In this submission, we have aimed to develop high performance solutions that are also clean and elegant. In particular, we accept additional code complexity in order to improve performance. Our use of a software routing algorithm for RandomAccess (see Section 3) exemplifies this design philosophy. Table 1 shows the total source lines of code and their breakdowns for each of our four HPCC benchmark implementations and UTS.

All benchmark codes were compiled using Rice's Coarray Fortran 2.0 compiler and runtime system [13]. Our Coarray Fortran 2.0 compiler is a source-to-source compiler built using the Rose compiler infrastructure [12] developed by Lawrence Livermore National Laboratory. Our Coarray Fortran 2.0 runtime library is built on the UC Berkeley's GASNet communication library [3]. We executed our benchmark codes on a range of Cray systems available to us over the past year. On the Cray XT4 and XT5, we used GASNet's Portals and MPI conduits for communication; on the Cray XE6, we used GASNet's new Gemini conduit, which is based on Cray's GNI driver for Gemini. The version of GASNet and the conduit used for each set of experiments is specified in our table of results. For all of our experiments, we used Cray's PGI programming environment and the Portland Group's Fortran compiler for compiling the Fortran 90 codes generated by our CAF translator. Since our results were collected over time, different PGI compiler versions were used for various benchmarks; the version used for each set of experiments is specified in our table of results.

To evaluate the performance of our CAF 2.0 implementations of the FFT, and HPL benchmarks, we ran them on up to 4096 cores of Franklin, a Cray XT4 system at the National Energy Research Scientific Computing Center. Each node in Franklin contains a 2.3 GHz single socket quad-core AMD Opteron processor (Budapest) (theoretical peak performance of 9.2 GFlop/sec per core) and 2GB of memory per core. The memory speed is 800 MHz. Each node is connected to a dedicated SeaStar2 router through Hypertransport. The SeaStar2 interconnect is arranged as a 3D torus.

We evaluated our CAF 2.0 implementations of the STREAM and RandomAccess benchmarks on up to 4096 cores of Jaguar, a Cray XT4 system at Oak Ridge National Laboratory (ORNL). Jaguar contains 7,832 compute nodes. Each node contains a quad-core AMD Opteron 1354 (Budapest) processor running at 2.1 GHz. Some nodes use 8 GB of DDR2-800 memory, and others use DDR2-667 memory. Like Franklin, Jaguar's nodes are also connected with a SeaStar2 router. We also experimented with STREAM on Hopper, a Cray XE6 that includes 6,384 nodes, each with two twelve-core AMD 'MagnyCours' 2.1-GHz processors and 32 GB DDR3 1333-MHz memory.

	STREAM	RandomAccess	FFT	HPL	UTS
Computation	32	188	180	536	267
Communication & synchronization	1	12	4	46	17
Declaration	17	118	103	109	151
Comments & spaces	13	91	163	95	109
Total	63	409	450	786	544

Table 1: Source lines of code for CAF 2.0 benchmarks.

# cores	STREAM (GByte/s)		RandomAccess (GUP/s)			FFT (GFLOP/s)	HPL (GFLOP/s)	UTS (MNode/s)
	Oct 2011 Jaquar(XT5)	Oct 2010 Franklin	Oct 2011 Hopper	Oct 2011 Hopper	Oct 2010 Jaguar(XT4)	Jul 2011 Franklin	Oct 2011 Franklin	Oct 2011 Jaguar(XT5)
# cores/node	12	4	16	4	4	4	4	12
GASNet library	16.2	14.2	17.2	17.2	14.2	14.2	14.2	16.2
PGI(Gnu) compilers	10.9	10.0	11.7	11.7	10.0	10.0	10.0	10.9(4.4.3)
1								2.578
2		8.7				0.50		
4		8.5		0.0525	0.014	0.54	27.1	
8		17.0		0.0748	0.020	0.99		
16		34.0	0.119	0.117	0.031	1.77	95.9	
32		68.2	0.116	0.185	0.051	3.31		
64	82.0	137.6	0.150	0.303	0.084	6.69	363.5	163.1
128		272.5	0.212	0.434	0.14	11.90		325.8
256	327.0	544.4	0.339	0.691	0.24	22.82	1357.6	645.0
512		1089.8	0.456		0.44	38.61		1252
1024	1320	2179.2	0.746	1.84	0.64*	67.80	4990.7	2371
2048		4358.4	1.26	3.03	0.97*	97.49		4962
4096	7004	8733.4			1.67*	187.04	18333.2	7818
8192	14097				1.67*	357.80		12286

* These data for RandomAccess were measured on a larger version of the code that had many alternate implementations of verification. The routing algorithm is the same as that presented in the appendix and used to measure the other data points.

Table 2: Performance results of Coarray Fortran 2.0 implementation of the benchmarks on Cray XT.

Experiments with UTS were performed on the Cray XT5 partition of Jaguar, which consists of 18,688 compute nodes. Each compute node contains dual hex-core AMD Opteron 2435 (Istanbul) processors running at 2.6GHz, 16GB of DDR2-800 memory, and a SeaStar 2+ router.

Table 2 shows performance results of the four benchmarks running on up to 8192 cores of the aforementioned Cray systems.

2 EP STREAM Triad

The STREAM benchmark evaluates the extent to which a parallel system can deliver and sustain peak memory bandwidth by performing a simple vector operation that scales and adds two vectors:

$$a \leftarrow b + \alpha c \qquad (1)$$

Performance of the STREAM benchmark is measured in GByte/s, with the calculated performance defined as $24\frac{m}{t_{min}}10^{-9}$, where m is the size of the vectors, required to be at least a quarter of system memory; and

```
double precision, allocatable :: a(:)[*]
double precision, allocatable :: b(:)[*], c(:)[*]
! allocate with the default team
allocate(a(ndim)[], b(ndim)[], c(ndim)[])
...
do round = 1, rounds
  do j = 1, rep
    call triad(a,b,c,n,scalar)
  end do
  call team_barrier()
end do
...
subroutine triad(a, b, c, n ,scalar)
  double precision a(n), b(n), c(n), scalar
  a = b + scalar * c
end subroutine triad
```

Figure 1: Implementation of STREAM benchmark.

t_{min} is the minimum execution time over at least 10 repetitions of the benchmark kernel. The STREAM benchmark is embarrassingly parallel; the work performed on any one node is independent of that performed on others.

Since the STREAM benchmark does not require communication between processes, the Coarray Fortran version is essentially identical to the sequential Fortran implementation, with the exception that all arrays are declared and allocated as coarrays. A sketch of the essence of our implementation is shown in Figure 1.

To deliver top performance, we outlined the STREAM triad calculation from the timing loop. Since our CAF 2.0 compiler presently represents coarray data using F90 pointers, having the triad computation in a separate routine enabled us to inform the compiler that the arrays were contiguous data by using explicit shape array declarations within triad. Eventually, we will either automatically outline coarray computations for performance, or use the Fortran 2008 CONTIGUOUS attribute to inform the back-end compiler about the contiguity of coarray data and avoid the need for outlining.

We initialized b and c with identical values (generated with the random number generator from RandomAccess), and set the scalar α to -1, so that the result should be 0. We considered the calculation verified if the maximum value of the difference between the computed value and 0 was less than 10^{-9}.

3 RandomAccess

The RandomAccess benchmark evaluates the rate at which a parallel system can apply updates to randomly indexed entries in a distributed table. Performance of the RandomAccess benchmark is measured in Giga Updates Per Second (GUP/s). GUP/s is calculated by identifying the number of table entries that can be randomly updated in one second, divided by 1 billion (1e9). The term "randomly" means that there is little relationship between one table index to be updated and the next. An update is a read-modify-write operation on a 64-bit word in the table. First, a table index is generated using a random number generator. Then, the table value at that index is combined with a literal value using an xor and the resulting value is written back to memory.

On distributed-memory parallel systems that lack hardware support for shared memory, fine-grain operations on remote data are expensive. To develop a high performance implementation of RandomAccess in CAF 2.0, we exploit the "1024 element look ahead and storage" allowed by the problem specification. First, each process image generates a batch of 1024 indices of table locations to be updated. Next, the code uses a hypercube-based pattern of bulk communication to route each update to the appropriate process image co-located with the table index being updated. Finally, each process image locally applies updates to its section of the distributed table.

Figure 2 shows a sketch of the hypercube routing algorithm used in our CAF 2.0 implementation of RandomAccess. Routing occurs in $\log P$ rounds. As each round begins, each processor has a set of elements. For each element, the processor must decide whether to communicate or retain in it the current round. Each pro-

```
event,allocatable :: delivered(:)[*],received(:)[*]
integer(8),allocatable :: fwd(:,:,:)[*]

do i = world_logsize-1, 0, -1
  ...
  call split(ret(:,last), retsizes(last), ret(:,current), retsizes(current), &
             fwd(1:,out,i),fwd(0,out,i),bufsize,dist)

   if (i < world_logsize-1) then
     event_wait(delivered(i+1))
     call split(fwd(1:,in,i+1), fwd(0,in,i+1), ret(:,current), retsizes(current), &
                fwd(1:,out,i),fwd(0,out,i),bufsize,dist)
     event_notify(received(i+1)[from])
   endif

   outgoing_size = fwd(0,out,i)
   call event_wait(received(i))
   fwd(0:outgoing_size,in,i)[partner] = fwd(0:outgoing_size,out,i)
   call event_notify(delivered(i)[partner])
   ...
end do
```

Figure 2: A sketch of hypercube routing for updates in CAF 2.0 RandomAccess.

cessor splits the updates in its possession at the time into two sets: those to retain and those to communicate in the current round. After the first round, a processor uses `event_wait` to wait for communication from the prior round to complete. Once a set of updates have been received from a communication partner, they are also split for communication or retention. Following the second split, each processor first waits for its communication partner's buffer to be available by checking that its prior message to this communication partner has been already received. Each process finishes a round of routing by copying a vector of updates into its partner's coarray and notifying its partner that they have been delivered. Our CAF 2.0 compiler translates this copy/`event_notify` pair into a non-blocking, asynchronous put-with-notify operation supported by the CAF 2.0 runtime.

Similar software routing strategies have been used before, though never with CAF. Researchers at Sandia studied a different but related strategy for RandomAccess using all-to-all communication based on a hypercube communication pattern [11]. IBM also explored a software routing strategy for the RandomAccess benchmark on Blue Gene systems [4].

4 Global FFT

The HPC Challenge *FFT* (Fast Fourier Transform) benchmark measures the ability of a system to overlap computation and communication while calculating a very large Discrete Fourier Transform of size m with input vector z_i and output vector Z_i:

$$Z_k \leftarrow \sum_{j}^{m} z_j e^{-2\pi i \frac{jk}{m}}; 1 \leq k \leq m$$

Performance of the FFT benchmark is measured in GFLOP/s, with calculated performance defined as $5\frac{m\log_2 m}{t}10^{-9}$, where m is the size of the DFT and t is the execution time (in seconds). The number of processors for this benchmark may be implementation-specific; in particular, it is allowed to be an integral power of 2. Parallel FFT algorithms has been well studied in the past [16, 2, 6]. The reference FFT implementation of the HPC Challenge benchmarks uses a 1D algorithm based on [16].

Our CAF 2.0 FFT implementation uses a radix 2 binary exchange formulation that consists of five parts: permutation of data to move each source element to the position that is its binary bit reversal; local (in-core) FFT computation for levels that use data co-located with a single CAF process image; transposition of the data from block to cyclic layout; FFT computation for the remaining layers (which now requires only local data); and reverse transposition to restore data from cyclic to block layout. This is shown in Figure 3.

```
complex, allocatable :: c(:)[*]
...
! permute local data
...
! calculate twiddle table
...
! compute levels of FFT using the data co-located with a single CAF process image
...
! transpose data from block to cyclic distribution
...
! compute remaining levels
...
! transpose data back from cyclic to block distribution
```

Figure 3: Sketch of FFT computation.

5 Global HPL

In our Coarray Fortran 2.0 implementation of HPL, the main matrix is declared as a coarray and is distributed in block-cyclic fashion in both dimensions. We use a single block size for both dimensions. However, given a processor core topology, the number of matrix blocks mapped to each processor in different dimensions can be different. For simplicity, we also use the same block size of the distribution as the width of the panel in the LU panel factorization with row partial pivoting. Choosing the right block size for the block-cyclic matrix distribution is very important. The block size determines the efficiency of core computation, the load balance between processors, and the communication latency exposed. This is machine and algorithm dependent. Our implementation takes an optional block size as an input parameter. For the final runs of the experiments we used 92 double precision elements as the block size.

We use Coarray Fortran 2.0 teams to represent processor subsets. We create column teams for processors that own the same matrix column and row teams for processors that own the same matrix row. All collective operations are performed within pre-arranged teams for high efficiency. This is particularly important for achieving scalable high performance on large machines. To hide communication latency, we used a team-based asynchronous broadcast operation `team_broadcast_async` to broadcast panels. This operation is one of the asynchronous collective operations that we have designed into Coarray Fortran 2.0. We used a user-defined all-reduce MAX operation for implementing the row partial pivoting and broadcasting the pivot elements. We used Cray Scientific Libraries package, LibSci mainly for computing matrix-vector multiplication (`dger`) and matrix-matrix multiplication (`dgemm`).

Our implementation allocates a matrix with 12K $\times$ 12K double precision elements on each processor, more than half of the 2GBytes system memory. Other than the main matrix, coarrays are also used as buffers for panel broadcast and communication. A double panel buffer is used in order to perform panel factorization and update of trialing matrix in parallel. Random numbers are generated for the matrix during its initialization phases before the main computation and duing verfication. We verified and used 1.11×10^{-16} as machine precision for 64-bit floating-point values. We verified our results by checking the scaled residual formulated in [5].

6 Unbalanced Tree Search (UTS)

The UTS benchmark involves building and counting the number of nodes in an unbalanced n-ary tree. The tree is based on a geometric/binomial distribution. Each node in this tree is characterized by a 20-byte descriptor that is used to determine the number of children for the node and (indirectly) for all of its children. Descriptors are calculated from an SHA1 hash of the parent descriptor and the child's index; a more detailed explanation may be found in the UTS v1.1 benchmark [1]). Since each node's children are completely determined by its descriptor, the parent-child links need not be explicitly maintained: The UTS tree is virtual.

Our implementation of UTS is based on the T1WL UTS benchmark. We implement a UTS tree based

```
1       !while there is work to do
2   do while(queue_count .gt. 0)
3     delete_queue_end(descriptor, depth)
4     call process_work_item(descriptor, depth)
5         ...
6     !check if someone needs work
7     if ((incoming_lifeline .ne. 0) .and. (queue_count .ge. lifeline_threshold)) then
8           call push_work()
9     endif
10  enddo
11
12        ! attempting to steal work from another image
13   steal_from_img = get_random_image(my_rank, my_rank)
14   spawn steal_work_spawn(my_rank, 0)[steal_from_img]
15
16       ! set up lifelines
17   neighbor_index = 0
18   do while (neighbor_index .lt. max_neighbor_index)
19     next_neighbor = mod(my_rank+(2**neighbor_index), world_size)
20     spawn set_lifelines(my_rank, neighbor_index)[next_neighbor]
21     neighbor_index = neighbor_index + 1
22   enddo
```

Figure 4: Outline of the UTS benchmark.

on a geometric distribution with expected children size per node 4, maximum tree depth 18, and initial seed for the root descriptor 19.

The highlights of our implementation are as follows:

1. **Initial work sharing**
   Process 0 builds the initial few levels of the UTS tree and then distributes these nodes to different processes. Each process then starts working on building the rest of the tree rooted at these nodes. Process 0 does this by spawning the function *copy_item_and_activate* on the processes.

2. **Random work stealing**
   Each process, after finishing its quota of work, tries to steal work (i.e. nodes whose subtree has not been constructed) from a random process. If a process is unsuccessful in n steal attempts (set to 1 in these experiments) it then quiesces by setting up *lifelines* ([14]). Line 12 in Figure 4 depicts an attempt to steal from a remote node.

3. **Work sharing via lifelines**
   Process `A` sets up a *lifeline* on another image `B` to indicate that `A` is available for any excess work that `B` obtains in the future. In our implementation, a process sets up lifelines on each of its hypercube neighbors (the processes at offsets 2^0, 2^1, ..., $2^{log_2(num\_processes)}$). Process A uses function shipping to establish lifelines on the remote nodes. Since the spawn is executed on the process on which the lifeline has to be set up, the communication overhead for this operation is reduced to a single round trip.

6.1 Challenges posed by the benchmark

The geometric distribution of nodes in the tree causes there to be a great deal of imbalanced work in this application. Work stealing in general is a good strategy, but stealing alone cannot succeed for the following reasons:

- Amount to steal
  The stolen work might yield only a small subtree before petering out. In effect, the time spent in a network transaction to check if a process has work, lock the process's work queue, and then steal the work might not be fruitful. Few papers ([8], [10]) papers have looked at this in detail. Our steals are limited by the *ActiveMessageMediumPacket* size (a max of 9 items).

- Timing of the steal
  The steal attempt from a process A might land at process B when B is working on a few nodes. Concluding that B has a small amount of work would be erroneous if those nodes give rise to a large subtree.

References

[1] Uts 1.1 benchmark. `http://barista.cse.ohio-state.edu/wiki/index.php/UTS`.

[2] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance parallel algorithm for 1-D FFT. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 34–40, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[3] Dan Bonachea. GASNet specification, v1.8. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2006.

[4] Rahul Garg and Yogish Sabharwal. Software routing and aggregation of messages to optimize the performance of HPCC randomaccess benchmark. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2006. ACM.

[5] HPC challenge awards: Class 2 specification. `http://www.hpcchallenge.org/class2specs.pdf`, June 2005.

[6] S. Lennart Johnsson and Robert L. Krawitz. Cooley-Tukey FFT on the Connection Machine. *Parallel Computing*, 18:1201–1221, 1991.

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

[8] Min, Iancu Seung-Jai, Yelick Costin, and Katherine. Hierarchical work stealing on manycore clusters. In *Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models*, PGAS'11, 2011.

[9] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: An unbalanced tree search benchmark. *Lecture Notes in Computer Science*, 4382/2007:235–250, 2007.

[10] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. Uts: an unbalanced tree search benchmark. In *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, LCPC'06, pages 235–250, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] Steven J. Plimpton, Ron Brightwell, Courtenay Vaughan, Keith D. Underwood, and Mike Davis. A simple synchronous distributed-memory algorithm for the HPCC RandomAccess benchmark. In *CLUSTER*. IEEE, 2006.

[12] Daniel J. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[13] Rice Coarray Fortran 2.0. `http://caf.rice.edu`, 2010.

[14] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. Lifeline-based global load balancing. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 201–212, New York, NY, USA, 2011. ACM.

[15] William N. Scherer, III, Laksono Adhianto, Guohua Jin, John Mellor-Crummey, and Chaoran Yang. Hiding latency in coarray fortran 2.0. In *PGAS '10: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, New York, NY, USA, 2010.

[16] Daisuke Takahashi and Yasumasa Kanada. High-performance radix-2, 3 and 5 parallel 1-D complex FFT algorithms for distributed-memory parallel computers. *J. Supercomput.*, 15(2):207–228, 2000.

A EP STREAM Triad code

A.1 stream.caf

```
program  stream
 use module_triad
 implicit none
 integer(8), parameter :: POLY = 7, offset=0, rounds=10, rep=10
 integer(8), parameter :: n=(256 * 1024 * 1024 / 18)

 double precision, allocatable :: a(:)[*], b(:)[*], c(:)[*]
 double precision :: tmin, scalar

 integer(8):: i, j, round, ndim, me, start_time, times(rounds), rate,ran
 real(8) rnd

 ndim = n + offset
 allocate(a(ndim)[], b(ndim)[], c(ndim)[])

 ! initialize
 me = team_rank()
 call random_number(rnd) ! the range of rnd is 0 ... 1
 ran = rnd *(me+1)*n      ! the range of ran is 0 ... n(n+1)
 scalar = -1.0D0

 do i = 1, n
    ran = xor(ishft(ran,1), iand(-ishft(ran,-63),POLY))
    b(i) = ran
 end do
 c = b

 times = 0.0D0

! compute the triad
 do round = 1, rounds
    call system_clock(start_time, rate)
    do j = 1, rep
      call triad(a,b,c,n,scalar)
    end do
    call system_clock(times(round), rate)
    times(round) = times(round) - start_time
    call team_barrier()
 end do
 tmin = minval(times)

 if (me == 0) then
    print *,'min␣time=',tmin
    print *,'n=',ndim,"total␣mem␣usage:",ndim*24
    print *,'BW/PE␣(MB/s,␣WC)=',24.0D0*n*rep/tmin*1D-6*rate
 end if

! verification
 a = abs(a)
 if (maxval(a) > 1e-9) then
    print *, 'Verification␣failed'
 end if

  deallocate(a, b, c)
end program stream
```

A.2 module_triad.caf

```
module module_triad
contains
subroutine triad(a, b, c, n, scalar)
 integer n
 double precision a(n), b(n), c(n), scalar
 a = b + scalar * c
end subroutine triad
end module module_triad
```

B RandomAccess code

B.1 randomaccess.caf

```
subroutine print_problem_stats(nupdates_local)
  use module_team
  use module_table

  implicit none
  integer(8) :: nupdates_local

  if (world_rank == 0) then
    write(*,'(A)') "randomaccess␣benchmark"
    write(*,'(A,I18,A,I18,A)') "global␣table␣␣␣␣=", local_table_size * 8 * world_size, &
        "␣bytes,␣",  local_table_size * world_size, "␣elements"
    write(*,'(A,I18,A,I18,A)') "local␣table␣␣␣␣␣=", local_table_size * 8, &
        "␣bytes,␣",  local_table_size, "␣elements"
    write(*,'(/,A,I18)')      "global␣updates␣␣=", nupdates_local * world_size
    write(*,'(A,I18)'  )      "local␣updates␣␣␣=", nupdates_local
  endif
end subroutine print_table

subroutine print_gups(start_time, end_time, rate, updates, style)
  use module_team

  implicit none
  real, parameter :: billion = 1e9
  integer(8)      :: updates
  integer         :: start_time, end_time, ticks, rate
  real            :: GUPS
  character(*)    :: style

  ticks = end_time - start_time
  if (world_rank == 0) then
    GUPS = (updates * 1.0 * world_size * rate) / (ticks * billion)
    write(*,'(/,A,I18,A)')  "clock␣rate␣␣␣␣␣␣=", rate, "␣ticks␣per␣second"
    write(*,    '(A,I18)') "ticks␣␣␣␣␣␣␣␣␣␣␣=", ticks
    write(*,'(A,F18.2,A)') "elapsed␣time␣␣␣␣=", ticks/(1.0 * rate), "␣seconds"
    write(*,'(A,A,E18.4)') style, ":␣GUPS␣␣␣=", GUPS
  endif
end subroutine

subroutine perform_updates(nupdates_local, bunch_size, style)
  use module_team
  use module_route
  use module_random_seq

  implicit none
  integer            :: start_time, end_time, rate, bunch_size
  integer(8)         :: nupdates_local, start_pos, round
  character(*)       :: style

  dropped_updates = 0

  start_pos = world_rank * nupdates_local
  call random_seq_set_position(start_pos)

  call team_barrier()
  call system_clock(start_time, rate)

  do round = 1, nupdates_local - bunch_size + 1, bunch_size
     call random_seq_get_bunch(retain(1:bunch_size,0))
     call route
  enddo

  call team_barrier()
  call system_clock(end_time)

  call print_gups(start_time, end_time, rate, nupdates_local, style)
end subroutine perform_updates


program randomaccess
  use module_team
  use module_table
  use module_split
```

```
  use module_route

  implicit none
  integer, parameter :: bunch_size = 1024
  character(len=32)  :: arg
  integer            :: local_table_bits
  integer(8)         :: nupdates_local, error_count, error_bound

  call team_init()

  ! arg 1 specifies the number of elements in the local table as a power of 2
  local_table_bits = 27
  if (command_argument_count() > 0) then
     call get_command_argument(1, arg)
     if (len_trim(arg) .ne. 0) then
        read(arg, '(I2)') local_table_bits
     endif
  endif

  call table_init(local_table_bits)
  call route_init(bunch_size)

  nupdates_local = ishft(1_8, local_table_bits + 2)

  call print_problem_stats(nupdates_local)

  ! timed updates
  call perform_updates(nupdates_local, bunch_size, "routing")

  ! verification step: apply updates again (gratuitously timed)
  call perform_updates(nupdates_local, bunch_size, "routing␣verification")

  call count_update_errors(error_count, error_bound, dropped_updates)

  if (world_rank == 0) then
     write(*,'(/,A,I18)'  )     "errors␣founds␣␣␣␣␣␣␣=", error_count
     write(*,'(/,A,I18)'  )     "error␣upper␣bound␣␣␣=", error_bound
  endif

end program randomaccess
```

B.2 module_route.caf

```
module module_route
  use module_team
  use module_split
  integer(8), allocatable, dimension(:,:) :: retain
  integer(8), dimension(0:1) :: retain_sizes

  event, allocatable, dimension(:) :: delivered[*]
  event, allocatable, dimension(:) :: received[*]
  integer(8), allocatable, dimension(:,:,:) :: fwd[*]

  integer :: buffer_size, nelements_per_bunch
  integer :: last, current
contains

  subroutine route_init(nelements_per_bunch_)
    nelements_per_bunch = nelements_per_bunch_
    buffer_size = max(2000000, 2 * nelements_per_bunch)
    allocate(retain(buffer_size, 0:1))
    allocate(fwd(0:buffer_size, 0:1, 0:world_logsize-1)[])
    allocate(delivered(0:world_logsize-1)[])
    allocate(received(0:world_logsize-1)[])
    call event_init(delivered, world_logsize)
    call event_init(received, world_logsize)
    do i = world_logsize-1, 0, -1
      call event_notify(received(i))
    enddo
  end subroutine route_init

  subroutine route
    use module_bits
    integer, parameter :: out = 0, in = 1
    integer :: distance, partner, i, from
    integer :: outgoing_size, itemp, n, iii, iik, advance, next_slot
```

```
    advance(n) = 1 - n ! statement function to advance indices

    last = 0
    current = 1

    retain_sizes(last) = nelements_per_bunch

    do i = world_logsize-1, 0, -1
      distance = ishft(1, i)

      partner = mod(world_rank + distance + world_size, world_size)

      retain_sizes(current) = 0
      fwd(0,out,i) = 0  ! zero count of elements in outgoing buffer

      !-----------------------------------------------------------------
      ! partition elements retained in last routine into ones to retain
      ! or forward in the current round
      !-----------------------------------------------------------------
      call split(retain(:,last), retain_sizes(last), &
               retain(:,current), retain_sizes(current), &
               fwd(1:,out,i), fwd(0,out,i), buffer_size, distance)

      !-----------------------------------------------------------------
      ! partition elements forwarded in last routine into ones to retain
      ! or forward in the current round
      !-----------------------------------------------------------------
      if (i < world_logsize-1) then
        call event_wait(delivered(i+1))
        call split(fwd(1:,in,i+1), fwd(0,in,i+1), &
                   retain(:,current), retain_sizes(current), &
                   fwd(1:,out,i), fwd(0,out,i), buffer_size, distance)
        call event_notify(received(i+1)[from])
      endif

      call event_wait(received(i))
      outgoing_size = fwd(0,out,i)
      fwd(0:outgoing_size,in,i)[partner]= fwd(0:outgoing_size,out,i)
      call event_notify(delivered(i)[partner])

      from = mod(world_rank - distance + world_size, world_size)

      last = advance(last)
      current = advance(current)
    end do

    call apply_updates(retain(:,last), retain_sizes(last))

    call event_wait(delivered(0))
    call apply_updates(fwd(1:,in,0), fwd(0,in,0))
    from = mod(world_rank - 1 + world_size, world_size)
    call event_notify(received(0)[from])

  end subroutine route

end module route
```

B.3 module_split.caf

```
module module_split
  use module_table
  use module_team
  integer(8) :: dropped_updates
contains

  subroutine split(in, in_s, keep, keep_s, fwd, fwd_s, buffersize, distance)

     implicit none
     integer(8), dimension(:) :: in, keep, fwd
     integer(8) :: in_s, keep_s, fwd_s
     integer :: buffersize, distance, partner_rank, i, target_rank

     partner_rank = world_rank + distance

     do i = 1, in_s
```

```
            target_rank = ishft(in(i), -local_table_logsize)
            target_rank = iand(target_rank, world_size - 1)

            if (target_rank < world_rank) then
               target_rank = target_rank + world_size
            endif
            if (target_rank - world_rank < distance) then
               if (keep_s < buffersize) then
                  keep_s = keep_s + 1
                  keep(keep_s) = in(i)
               else
                  dropped_updates = dropped_updates + 1
               endif
            else
               if (fwd_s < buffersize) then
                  fwd_s = fwd_s + 1
                  fwd(fwd_s) = in(i)
               else
                  dropped_updates = dropped_updates + 1
               endif
            endif
         enddo
       end subroutine split
    end module module_split
```

B.4 module_table.caf

```
module module_table
  use module_team
  integer(8), allocatable :: table(:)[*]
  integer(8) :: local_table_size, local_table_logsize
contains

  subroutine table_init(local_n)
    implicit none
    integer    :: local_n
    integer(8) :: start_index, i

    local_table_size = ishft(1_8, local_n)
    local_table_logsize = local_n
    allocate(table(0:local_table_size-1)[])
    start_index = world_rank * local_table_size
    do i = 0, local_table_size-1
      table(i) = start_index + i
    end do
  end subroutine table_init

  subroutine apply_updates(buffer, size)
    implicit none
    integer(8) :: buffer(:)
    integer(8) :: size, index, i
    do i = 1, size
      index = iand(buffer(i), local_table_size - 1)
      table(index) = ieor(table(index), buffer(i))
    end do
  end subroutine apply_updates

  subroutine reduce_sum(in1, in2, out, nbytes)
    integer(8) :: in1(2), in2(2), out(2)
    integer :: nbytes
    out = in1 + in2
  end subroutine reduce_sum

  subroutine count_update_errors(error_count, error_bound, dropped_updates)
     integer(8) :: error_count, error_bound, dropped_updates, i, start_index
     integer(8) :: errors(2)
     errors = 0
     start_index = world_rank * local_table_size
     do i = 0, local_table_size-1
       if (table(i) .ne. start_index + i) then
         errors = errors + 1
       end if
     end do

     errors(2) = errors(2) + dropped_updates
     call team_reduce(errors, errors, 0, reduce_sum)
```

```

      error_count = errors(1)
      error_bound = errors(2)
    end subroutine count_update_errors

  end module module_table
```

B.5 module_random_seq.caf

```
  module module_random_seq
    integer(8) :: POLY
    integer(8) :: PERIOD
    integer(8) :: ran       ! next random number in the sequence
  contains

    !-------------------------------------------------------------
    ! generate the random number following val in a sequence specified
    ! by the generator
    !-------------------------------------------------------------
    integer(8) function random_seq_get_next(val)
      integer(8) :: val
      random_seq_get_next = ieor(ishft(val,1), iand(-ishft(val,-63), POLY))
    end function

    !-------------------------------------------------------------
    ! fast forward to the nth random number that would be returned by
    ! get_next_random in a sequence beginning with the value 1.
    ! this code is a Fortran version of the C function urng() found
    ! in the HPC Challenge benchmark reference implementation.
    !-------------------------------------------------------------
    subroutine random_seq_set_position(nth)
      implicit none
      integer(8) :: nth, n, temp
      integer(8) ::  m2(0:63)
      integer :: i,j

      POLY = 7
      PERIOD = 1317624576693539401_8

      n = nth
      do while (n < 0)
        n = n + PERIOD
      enddo

      do while (n > PERIOD)
        n = n - PERIOD
      enddo

      if (n == 0) then
        ran = 1
      else
        temp = 1
        do i = 0, 63
          m2(i) = temp
          temp = random_seq_get_next(temp)
          temp = random_seq_get_next(temp)
        enddo

        do i = 62, 0, -1
          if (btest(n, i)) exit
        enddo

        ran = 2
        do while (i > 0)
          temp = 0
          do j = 0, 63
            if (btest(ran, j)) temp = ieor(temp, m2(j))
          enddo
          ran = temp
          i = i - 1
          if (btest(n,i)) ran = random_seq_get_next(ran)
        enddo
      endif
    end subroutine random_seq_set_position

    subroutine random_seq_get_bunch(rand)
```

```
      integer(8), dimension(:) :: rand
      do i = 1, size(rand)
         rand(i) = ran
         ran = random_seq_get_next(ran)
      enddo
    end subroutine random_seq_get_bunch

  end module module_random_seq
```

B.6 module_team.caf

```
  module module_team
    use module_bits
    integer :: world_rank, world_size, world_logsize
  contains

    subroutine team_init()
      world_rank = team_rank()
      world_size = team_size()
      world_logsize = number_of_bits(world_size - 1)
    end subroutine team_init

  end module module_team
```

B.7 module_bits.caf

```
  module module_bits
  contains
    !-----------------------------------------------------------------------------
    ! number_of_bits: compute the index of the leftmost non-zero bit in i
    !-----------------------------------------------------------------------------
    function number_of_bits(i) result(num)
      num = 0
      itmp = i
      do while (itmp .gt. 0)
         num = num + 1
         itmp = ishft(itmp, -1)
      end do
    end function number_of_bits

  end module module_bits
```

C Global FFT

C.1 fft.caf

```
  program fft_driver
    use module_fft
    use module_reverse
    implicit none

    integer(8) :: n, two_n, world_size, world_logsize
    integer(8) :: local_size, local_logsize
    integer(8) :: tstart, tend, rate, mystart, rank
    double precision :: tdelta, gflops, tsec
    character(len=32)  :: arg

    world_size = team_size()
    world_logsize = number_of_bits(world_size)-1
    rank = team_rank()

    ! by default the local core will have 1M of elements
    local_logsize = 21

    !----------------------------------------------------
    ! get the command line
    !----------------------------------------------------
    if (0 < command_argument_count()) then
       call get_command_argument(1, arg)
       if (len_trim(arg) .ne. 0) then
          read(arg, '(I2)') local_logsize
       endif
    endif
```

```
    !---------------------------------------------------
    ! INIT: compute local size
    !---------------------------------------------------
    ! from hpcc spec: the number of elements should be at lest
    !     2^5 * m <= 0.25 * local_mem_size,
    ! since  m = 2 ^ local_logsize, then:
    !     2 ^ local_logsize <= 2^-7 * local_memsize
    !     local_logsize <= log2 (local_memsize) - 7
    local_size = ishft(1,local_logsize)

    !---------------------------------------------------
    ! INIT: compute my part of computation
    !---------------------------------------------------
    call bitinit()
    mystart = rank * local_size
    call fft_init(local_size, mystart, rank)

    !---------------------------------------------------
    ! computation
    !---------------------------------------------------
    call system_clock(tstart, rate)
    call fft_inner(local_size, 1_8)
    call system_clock(tend, rate)
    call fft_verif(world_size, local_size, mystart)

    !---------------------------------------------------
    ! timing output
    !---------------------------------------------------
    if ((0 .eq. rank) .and. (local_logsize > 12)) then
       n = local_logsize + world_logsize
       two_n = ishft(1,n)

       tsec = real(tend-tstart) / rate
       write (*,'(A,F8.4)') "Elapsed␣time:␣", tsec
       gflops = ((5 * n * two_n) / tsec) * 1D-9
       write (*, '(A,I7,A,I9,A,F8.4)') "Num␣PEs:␣", world_size, &
                                    ";␣Local␣size:", local_size, ";␣GFlops␣=␣", gflops
    endif
    call team_barrier()

  end program fft_driver
```

C.2 module_fft.caf

```
  module module_fft
      use module_permute
      use module_bits
      use module_transpose

      complex(8), allocatable, dimension (:) :: c[*]      ! data array
      complex(8), allocatable, dimension (:) :: spare[*] ! scratch space
      complex(8), allocatable, dimension (:) :: twiddles

      ! controls whether the fft calculated is a known pattern
      ! with a known result
      logical, parameter :: debug_mode = .false.
  contains

  !---------------------------------------------------------------------------
  ! Problem initialization
  !---------------------------------------------------------------------------
    subroutine initialize_data_array(n_local_size, local_start, rank, buffer)
      implicit none
      integer(8) :: n_local_size, local_start, rank, i, j
      integer :: n
      complex(8) :: buffer(0:n_local_size - 1)
      integer(4), parameter :: SEED = 314159265_4
      real(8) :: h, h2
      integer, dimension(:), allocatable :: rseeds

      ! Initialize random number generator
      call random_seed(size=n)
      allocate(rseeds(n))
      do j = 1, n
         rseeds(j) = (rank + 1) * SEED * j
```

```
     end do
     call random_seed(PUT = rseeds)
     deallocate(rseeds)

     ! Initialize data with either random numbers or an ascending sequence
     do i = 0, n_local_size - 1
       if (.not. debug_mode) then
          call random_number(h)
          call random_number(h2)
          buffer(i) = cmplx(h, h2)
       else
          buffer(i) = cmplx(i + local_start + 1)
       endif
     enddo
   end subroutine initialize_data_array


   subroutine fft_init(n_local_size, local_start, rank)
     implicit none
     integer(8) :: n_local_size, i, local_start, rank

     ! Allocate memory
     allocate(twiddles(0:n_local_size/2 - 1))
     allocate(c(0:n_local_size - 1)[])
     allocate(spare(0:n_local_size - 1)[])

     ! Initialize data (in a separate routine so we can reinitialize
     ! at verification time)
     call initialize_data_array(n_local_size, local_start, rank, c)
   end subroutine


 !---------------------------------------------------------------------------
 ! verification
 !---------------------------------------------------------------------------
   subroutine fft_verif(world_size, n_local_size, local_start)
     implicit none
     integer(8) :: i, j, rank, world_size, n_local_size, local_start, mei
     real(8)    :: norm, error, max_error, residue, logm
     real(8), parameter :: Epsilon = 1.1E-16
     real(8), parameter :: max_residue = 16.0


     ! Perform inverse fft
 !     c = c / (world_size * n_local_size)
     do i = 0, n_local_size - 1
        c(i) = c(i) / (world_size * n_local_size)
     end do
     call fft_inner(n_local_size, -1_8)

     ! Regenerate source data
     rank = team_rank()
     call initialize_data_array(n_local_size, local_start, rank, spare)


     ! Find max error for infinity norm
     max_error = -1.0; mei = -1
     do i = 0, n_local_size - 1
        error = abs(c(i) - spare(i))
        if (error .gt. max_error) then; mei = i; endif
        max_error = max(max_error, error)
     end do

     ! Calculate residue
     logm = number_of_bits(world_size) - 1 + number_of_bits(n_local_size) - 1
     residue = (max_error / Epsilon) / logm
     if (residue .lt. max_residue .and. 0 .eq. rank) then
        write (*,'(A)') "Verification␣successful"
     else if (residue .ge. max_residue) then
        write (*,'(A,ES10.4,A)') "Verification␣failed␣(residue␣=␣", residue, ")"
        write (*, '(A,ES11.4)') "␣␣␣Max␣error:␣", max_error
        write (*,'(A,2ES13.4,A,2ES11.4,A)')  "␣␣␣In:␣(", c(mei), ");␣Out:␣(", spare(mei), ")"
     endif
   end subroutine

 !---------------------------------------------------------------------------
 ! fft: compute the fft of complex c. c is of length n_local_size
```

```
! ATTN: using local view
!-------------------------------------------------------------------------
  subroutine fft_inner(n_local_size, direction)
    implicit none
    integer(8)           :: n_local_size, direction, world_size, n_world_size
    integer(8)           :: rank, lo, hi, lstride, i, j, k
    integer(8)           :: levels, l, loc_comm, m, m2
    complex(8)           :: ce, cr, cl
    double precision     :: two_pi, angle_base

    !---------------------------------------------------------------------
    ! prep for computation
    !---------------------------------------------------------------------
    rank = team_rank()
    world_size = team_size()
    n_world_size = world_size * n_local_size
    two_pi = 2.0d0 * acos(-1.0d0) * direction

    ! we assume the number of procs and the problem size are power of two
    levels = number_of_bits(n_world_size) - 1
    loc_comm = number_of_bits(n_local_size)

    ! Permute local data
    call permute(c, n_world_size, spare)

    ! Generate twiddle factor table. We only use the first half, since the
    ! second half is just -1 times the first.
    do i = 0, n_local_size/2 - 1
       spare(i) = exp(cmplx(0.0,(i) * ((-two_pi)/n_local_size)))
    enddo

    !---------------------------------------------------------------------
    ! phase 1 computation
    !---------------------------------------------------------------------
    do l = 1, loc_comm-1              ! --- for each local level in the FFT
       m = ishft(1, l)
       m2 = ishft(m, -1)
       lstride = ishft(n_local_size, -l)
       twiddles(0:m2 - 1) = spare(0:n_local_size/2-1:lstride)
       do k = 0, n_local_size-1, m    ! --- for each butterfly in a level
          do j = k, k + m2 - 1        ! --- for each point in a half bfly
             ce = twiddles(j - k)
             cr = ce * c(j + m2)
             cl = c(j)
             c(j) = cl + cr
             c(j + m2) = cl - cr
          end do
       end do
    enddo

    ! Get ready for the second phase -- transpose to cyclic layout.
    ! We can't fit in memory a twiddle table for this phase, alas.
    call transpose(c, n_world_size, world_size, spare, .true.)

    !---------------------------------------------------------------------
    ! phase 2 computation
    !---------------------------------------------------------------------
    do l = loc_comm, levels            ! --- for each level in the fft
       m  = ishft(1, l) / world_size   ! --- local elts/bfly
       m2 = ishft(m, -1)
       angle_base = (-two_pi) / real(ishft(1,l))
       do k = 0, n_local_size-1, m     ! --- for each butterfly in a level
          do j = k, k + m2 - 1         ! --- for each point in a half bfly
             ce = exp(cmplx(0.0,(((j - k) * world_size + rank) * angle_base)))
             cr = ce * c(j + m2)
             cl = c(j)
             c(j) = cl + cr
             c(j + m2) = cl - cr
          end do
       end do
    enddo

    ! restore to original layout
    call transpose(c, n_world_size, n_local_size/world_size, spare, .false.)
  end subroutine fft_inner

end module
```

C.3 module_bits.caf

```
module module_bits
contains

!-------------------------------------------------------------------------------
! number_of_bits: compute the index of the leftmost non-zero bit in i
!-------------------------------------------------------------------------------
  function number_of_bits(i) result(num)
    implicit none
    integer(8) :: i, itmp, num
    num = 0
    itmp = i
    do while (itmp .gt. 0)
       num = num + 1
       itmp = ishft(itmp, -1)
    end do
  end function number_of_bits

end module module_bits
```

C.4 module_permute.caf

```
module module_permute

   use module_bits
   use module_reverse

contains

   subroutine packf(input, output, n, p, n_b, cp_b, npadding, do_bitreverse)
      implicit none

      ! dummy arguments
      integer(8) :: n, n_b, p, npadding, p_b,cp_b
      complex(8) :: output(0:n-1 + p * npadding)
      complex(8) ::  input(0:n-1)
      logical    :: do_bitreverse

      ! local variables
      integer(8) :: pe_bufstart(0:p-1)
      integer(8) :: pe_buflen, buf, p_bits, ii, i, jj, j, ooffset, ioffset

      !----------------------------------------------------------------
      ! copy the constant from the caller into local variable
      ! without the copy, it will have runtime-crash on gfortran44
      !----------------------------------------------------------------
      p_b = cp_b

      !----------------------------------------------------------------
      ! compute a vector that will enable us to use the low log_2(p) bits
      ! of an index to find the start of the buffer for a processor based
      ! on the bitreverse of those bits
      !----------------------------------------------------------------
      p_bits = number_of_bits(p-1)
      pe_buflen = n/p + npadding
      if (do_bitreverse) then
         do i = 0, p - 1
            pe_bufstart(i) = i_bitreverse(i, p_bits) * pe_buflen
         enddo
      else
         do i = 0, p - 1
            pe_bufstart(i) = i * pe_buflen
         enddo
      endif

      !print *, "n=",n, "n_b = ", n_b, "p =", p, "p_b =", p_b, "npadding=", npadding

      ! special case handling for small numbers of processors
      if (p_b .gt. p) then; p_b = p; endif

      do jj = 0, p-1, p_b
         ooffset = 0
         do ii = 0, n-1, p * n_b
            do j = jj, jj + p_b - 1
               buf = pe_bufstart(j)
```

```
               ioffset = ooffset
               do i = ii, min(ii + p * (n_b - 1), n-1), p
                  output(buf + ioffset) = input(i+j)
                  ioffset = ioffset + 1
               enddo
            enddo
            ooffset = ooffset + n_b
         enddo
      enddo
   end subroutine packf


   subroutine permute_locally(dest, src, n, cn_b)
      implicit none

      complex(8), dimension(0:) :: src, dest
      integer(8) :: n_b, cn_b
      integer(8) :: i, j, n, n_bits
      n_b = cn_b

      n_bits = number_of_bits(n - 1)

      if (n_b .gt. n) then; n_b = n; endif

      do j = 0, n_b
         do i = j, n - 1, n_b
            dest(i_bitreverse(i, n_bits)) = src(i)
         end do
      end do
   end subroutine permute_locally

   subroutine permute(c, n, scratch)
      implicit none

      ! parameters
      integer(8), parameter :: SIZEOF_COMPLEX = 16
      complex(8), dimension(0:) :: c[*]
      complex(8), dimension(0:) :: scratch[*]
      integer(8) :: n, world_size, local_n, block_size

      world_size = team_size()
      local_n = n / world_size
      block_size = local_n / world_size

      call packf(c, scratch, local_n, world_size, 32_8, 1024_8, 0_8, .true.)
      call team_alltoall(scratch, c, block_size)
      call permute_locally(c, scratch, local_n, ishft(1_8,22))

   end subroutine permute


end module module_permute
```

C.5 module_reverse.caf

```
module module_reverse

    integer(8) :: mask64, mask32, mask16, mask8, mask4, mask2, mask1

contains

!---------------------------------------------------------------------------
! i_bitreverse: return the bitreverse of the n-bit integer i
!---------------------------------------------------------------------------
  function i_bitreverse(i, n)

    integer(8) :: i_bitreverse, i, n
    integer(8) :: ival, itmp, imask, ishift

    interchange(ival,imask,ishift) = &
        ior(ishft(iand(ival, imask), -ishift), &
            ishft(iand(ival, not(imask)), ishift))
    itmp = interchange(i,    mask32, 32_8)
    itmp = interchange(itmp, mask16, 16_8)
    itmp = interchange(itmp, mask8, 8_8)
    itmp = interchange(itmp, mask4, 4_8)
```

```
      itmp = interchange(itmp, mask2, 2_8)
      itmp = interchange(itmp, mask1, 1_8)

      i_bitreverse = ishft(itmp, n - 64_8) ! provide result as n-bit value
    end function i_bitreverse

    subroutine bitinit
      mask64 = not(0_8)
      mask32 = ishft(mask64,32_8)
      mask16 = ieor(ishft(mask32,-16_8),mask32)
      mask8 = ieor(ishft(mask16,-8_8),mask16)
      mask4 = ieor(ishft(mask8,-4_8),mask8)
      mask2 = ieor(ishft(mask4,-2_8),mask4)
      mask1 = ieor(ishft(mask2,-1_8),mask2)
    end subroutine bitinit


  end module module_reverse
```

D Global HPL

D.1 hpl.caf

```
program HPL
use support
use HPLmod

integer :: m, n, i, k
integer, allocatable :: seed(:)
double precision :: start_time, end_time, cputime, gflops, normA, normx, resid, norm_r, norm_c
double precision, allocatable :: normb(:)[*]
double precision, pointer :: a(:,:)
double precision, pointer :: b(:)
character(len=32) :: arg

! compute my location in a 2D processor grid
nprocs = team_size()
me = team_rank()
mycol = me / NPROW
myrow = me - mycol * NPROW

BLKSIZE = min(PROBLEMSIZE / NPCOL, 16)
if (0 .lt. command_argument_count()) then
  call get_command_argument(1, arg)
  if (len_trim(arg) .ne. 0) then
    read(arg, '(I10)') BLKSIZE
  end if
end if

if (nprocs .ne. NPCOL * NPROW) then
  if (me .eq. 0) print *, 'execution␣needs␣', NPCOL * NPROW, 'processors'
  stop
end if

i = NPROW
rowp2 = 1
rowlog2 = 0
do while (i .gt. 1)
   i = i / 2
   rowp2 = rowp2 * 2
   rowlog2 = rowlog2 + 1
end do

! compute my part of the matrix and allocate
m = localsize(PROBLEMSIZE, 0, BLKSIZE, NPROW, myrow)
n = localsize(PROBLEMSIZE, 0, BLKSIZE, NPCOL, mycol)
allocate(ab(m,n+1)[], x(n)[])
a => ab(1:m,1:n)
b => ab(1:m,n+1)

call team_split(team_world, myrow, mycol, rteam, myrow, ierr)
call team_split(team_world, myrow, mycol, rteam_, myrow, ierr)
call team_split(team_world, mycol, myrow, cteam, mycol, ierr)

call random_seed(size = k)
```

```
allocate(seed(k))
call random_seed(get = seed)
call init(m, n)

! collect time for the main body of work
call barrier()
call cpu_time(start_time)
call lup(m, n)
call backsolve(m, n)
call barrier()
call cpu_time(end_time)
cputime = end_time - start_time

gflops = (2.0/3.0*PROBLEMSIZE + 1.5) * (PROBLEMSIZE/1.0e9) * (PROBLEMSIZE/cputime)
if (me .eq. 0) then
  print *,'size,nprocs,blksize,time,gflops:',PROBLEMSIZE,',',nprocs,',',BLKSIZE,',',cputime,',',gflops
end if

! verify results
call random_seed(put = seed)
call init(m,n)
normA = norm_r(a(1,1),m,n,0)
normx = norm_c(x(1),n)
allocate(normb(1)[])
if (mycol .eq. mod(PROBLEMSIZE/BLKSIZE, NPCOL)) then
  normb(1) = maxval(abs(b))
  call team_reduce(normb, normb, 0, REDUCE_MAX, cteam)
  call team_broadcast(normb, 0, cteam)
  if (n .gt. 0 .and. m .gt. 0) b = b - matmul(a,x)
else
  b = 0.0
  if (n .gt. 0 .and. m .gt. 0) b = - matmul(a,x)
end if
call team_broadcast(normb, mod(PROBLEMSIZE/BLKSIZE, NPCOL), rteam)
call team_reduce(b(1:m), b(1:m), 0, REDUCE_SUM, rteam)
resid = norm_r(b(1),m,1,1)
resid = resid / (epsilon * (normA * normx + normb(1)) * PROBLEMSIZE)
if (me .eq. 0 .and. resid .lt. THRESHOLD) print *, "result␣VALID.␣scaled␣residual:␣", resid
if (me .eq. 0 .and. resid .ge. THRESHOLD) print *, "result␣INVALID.␣scaled␣residual:␣", resid

deallocate(ab,x,seed,normb)
end program HPL

function norm_r(a,s,t,is_vect) result(r)
use support
use HPLmod
integer :: s,t,i,is_vect
double precision :: r, a(s,t)
double precision, allocatable :: w(:)[*], nval(:)[*]
allocate (w(s)[], nval(1)[])
w = 0.0
do i = 1, s
  w(i) = sum(abs(a(i,:)))
end do
if (is_vect .ne. 1) call team_reduce(w(1:s), w(1:s), 0, REDUCE_SUM, rteam)
if (mycol .eq. 0) then
  nval(1) = 0.0
  if (s .gt. 0) nval(1) = maxval(w)
  call team_select(nval, nval, 0, SELECT_MAX, cteam)
end if
call team_broadcast(nval, 0)
r = nval(1)
deallocate(w, nval)
end function norm_r

function norm_c(v,n) result(r)
use support
use HPLmod
double precision :: r, v(n), w(n)
double precision, allocatable :: nval(:)[*]
allocate (nval(1)[])
nval(1) = 0.0
if (n .gt. 0) then
  w = abs(v)
  nval(1) = maxval(w)
end if
call team_select(nval, nval, 0, SELECT_MAX, rteam)
```

```
call team_broadcast(nval, 0)
r = nval(1)
deallocate(nval)
end function norm_c
```

D.2 module_hpl.caf

```
module HPLmod
integer, parameter :: PROBLEMSIZE = 64*12*1024, NUMPANELS = 2, NPCOL = 64, NPROW = 64
double precision, parameter :: EPSILON = 1.11e-16, THRESHOLD = 16.0
integer :: nprocs, rowp2, rowlog2, me, mycol, myrow, BLKSIZE
team :: rteam, rteam_, cteam, subcteam

! panelinfo: i, j, iloc, jloc, mloc, nloc, rproc, cproc, permute
integer, target, allocatable :: panelinfo_1(:)[*], panelinfo_2(:)[*]
double precision, target, allocatable :: panelbuff_1(:)[*], panelbuff_2(:)[*]
double precision, target, allocatable :: ab(:,:)[*]
double precision, allocatable :: x(:)[*]
event, allocatable, dimension(:) :: delivered[*]
integer :: isz = 0, bsz = 0

type :: panelptr
  integer, pointer :: info(:)
  double precision, pointer :: buff(:)
end type

type (panelptr) :: panels(1:NUMPANELS)

end module HPLmod
```

D.3 module_init.caf

```
subroutine init(m, n)
use HPLmod
integer :: m, n
do i = 0, NPROW * NPCOL - 1 - me
  do j = 1, n+1
    call random_number(ab(:,j))
  end do
end do
do j = 1, n+1
  ab(:,j) = ab(:,j) * 2.0 - 1.0
end do
end subroutine init
```

D.4 module_panel.caf

```
! initialize a panel starting at global index (pp, pp)
subroutine initpanel(p, pp)
  use HPLmod
  use support
  integer :: p, pp, blk, rproc, cproc, mloc, nloc, iloc, jloc    ! pp is 0-based

  blk = pp / BLKSIZE
  rproc = blk - blk / NPROW * NPROW
  cproc = blk - blk / NPCOL * NPCOL
  mloc = localsize(PROBLEMSIZE - pp, pp, BLKSIZE, NPROW, myrow)
  nloc = localsize(PROBLEMSIZE + 1 - pp, pp, BLKSIZE, NPCOL, mycol)
  isz = 8 * BLKSIZE + 2 * NPROW + 10
  bsz = (mloc + nloc + BLKSIZE + 1) * BLKSIZE
  if (p .eq. 1) then
    allocate(panelinfo_1(isz)[], panelbuff_1(bsz)[])
    panels(1)%info(1:isz) => panelinfo_1
    panels(1)%buff(1:bsz) => panelbuff_1
  else
    allocate(panelinfo_2(isz)[], panelbuff_2(bsz)[])
    panels(2)%info(1:isz) => panelinfo_2
    panels(2)%buff(1:bsz) => panelbuff_2
  end if
  iloc = localsize(pp, 0, BLKSIZE, NPROW, myrow)
  jloc = localsize(pp, 0, BLKSIZE, NPCOL, mycol)
  panels(p)%info(1:9) = (/pp+1, pp+1, iloc+1, jloc+1, mloc, nloc, rproc, cproc, 0/)  ! 0 to 1-based
end subroutine initpanel
```

D.5 module_lup.caf

```
subroutine lup(m, n)
use HPLmod
use support
integer :: m, n, ni, nn, p, pp = 0, cp = 1, pproc, cproc, ub, colstart
ni = localsize(PROBLEMSIZE+1, 0, BLKSIZE, NPCOL, mycol)
allocate(delivered(1:NUMPANELS)[@team_world])
call event_init(delivered, NUMPANELS)

nn = ni
! build panels to fill the panel buffer, factorize and update them
call initpanel(1, pp)
pp = pp + BLKSIZE
if (mycol .eq. panels(1)%info(8)) nn = nn - BLKSIZE
call initpanel(2, pp)

if (mycol .eq. panels(1)%info(8)) call fact(m, n, 1)
mloc = panels(1)%info(5)
pproc = panels(1)%info(8)
call team_broadcast_async(panelbuff_1(1:(mloc+BLKSIZE+1)*BLKSIZE),panels(1)%info(8),delivered(1),rteam)

! factorize rest panels and finish all updates
cproc = panels(2)%info(8)

do j = pp, PROBLEMSIZE - 1, BLKSIZE

  cp = j / BLKSIZE + 1
  cp = mod(cp - 1, 2) + 1
  if (cp .eq. 1) deallocate(panelinfo_1, panelbuff_1)
  if (cp .eq. 2) deallocate(panelinfo_2, panelbuff_2)
  call initpanel(cp, j)
  call event_wait(delivered(3-cp))
  numcol = 0
  colstart = 0

  if (mycol .eq. cproc) then
    numcol = localsize(min(BLKSIZE,PROBLEMSIZE-j), j, BLKSIZE, NPCOL, mycol)
    if (numcol .gt. 0) then
      if (NPCOL .eq. 1) call  update(m, n, BLKSIZE, numcol, 3 - cp)
      if (NPCOL .ne. 1) call  update(m, n, 0, numcol, 3 - cp)
    end if
    call fact(m, n, cp)
    colstart = colstart + numcol
  end if
  if (mycol .eq. pproc) colstart = colstart + BLKSIZE
  ub = (panels(cp)%info(5)+BLKSIZE+1)*BLKSIZE
  if (cp .eq. 1) call team_broadcast_async(panelbuff_1(1:ub), panelinfo_1(8), delivered(1), rteam)
  if (cp .eq. 2) call team_broadcast_async(panelbuff_2(1:ub), panelinfo_2(8), delivered(2), rteam_)

  if (nn-numcol .gt. 0) call update(m,n, colstart, nn-numcol, 3 - cp)

  if (mycol .eq. cproc) nn = nn - BLKSIZE
  pproc = cproc
  cproc = mod(cproc+1, NPCOL)

end do
numcol = localsize(1, PROBLEMSIZE, BLKSIZE, NPCOL, mycol)
colstart = ni - panels(cp)%info(4)
if (numcol .gt. 0) call update(m, n, colstart, numcol, cp)

deallocate(panelinfo_1, panelinfo_2, panelbuff_1, panelbuff_2)

end subroutine lup
```

D.6 module_fact.caf

```
! perform factorization of a panel
subroutine fact(m, n, p)
use support
use HPLmod
double precision, pointer, DIMENSION(:,:) :: a
double precision, pointer, DIMENSION(:) :: b, c, piv, l
integer :: m, n, p, clb, rlb, rub, rproc, pivl, pivg, pivproc, mloc, numcols, pivltmp(1)
double precision, allocatable :: w(:)[*]

```

```
with team cteam
allocate(w(2*BLKSIZE+4)[])

mloc = panels(p)%info(5)
a => ab(1:m,1:n)
b => ab(1:m,n+1)
l => panels(p)%buff(1:mloc*BLKSIZE)
c => panels(p)%buff(mloc*BLKSIZE+1:(mloc+BLKSIZE)*BLKSIZE)
piv => panels(p)%buff((mloc+BLKSIZE)*BLKSIZE+1:(mloc+BLKSIZE+1)*BLKSIZE)
clb = panels(p)%info(4)
rlb = panels(p)%info(3)
rub = rlb + mloc - 1
rproc = panels(p)%info(7)
numcols = min(BLKSIZE, n - clb + 1)
do i = 1, numcols
  ! compute pivots, switch rows, and update the panel
  w(1:BLKSIZE+3) = 0.0
  w(BLKSIZE+4) = nprow
  if (rub .ge. rlb) then
    pivltmp = maxloc(abs(a(rlb:rub,clb+i-1)))
    pivl = pivltmp(1)                      !  1-based
    pivg = pivl + rlb - 2                  !  0-based
    pivg = pivg + BLKSIZE * ((NPROW - 1) * (pivg / BLKSIZE) + myrow) + 1    !  1-based
    w(BLKSIZE+1) = a(rlb+pivl-1,clb+i-1)
    w(BLKSIZE+2:BLKSIZE+4) = (/pivl, pivg, myrow/)
    w(1:numcols) = a(rlb+pivl-1, clb:clb+numcols-1)
  end if
  call team_allselect(w(1:BLKSIZE+4), w(1:BLKSIZE+4), SELECT_MAX, cteam)
  piv(i) = w(BLKSIZE+3)

  pivl = w(BLKSIZE+2)
  pivproc = w(BLKSIZE+4)
  if (myrow .eq. rproc .and. myrow .ne. pivproc) then
    w(BLKSIZE+5:BLKSIZE+numcols+4)[pivproc@cteam] = a(rlb,clb:clb+numcols-1)
  end if
  call barrier(cteam)

  c(i::BLKSIZE) = w(1:numcols)
  if (myrow .eq. rproc) then
    if (myrow .ne. pivproc) then
      a(rlb, clb:clb+numcols-1) = w(1:numcols)
    else if (myrow .eq. pivproc .and. pivl .ne. 1) then
      a(pivl+rlb-1, clb:clb+numcols-1) = a(rlb,clb:clb+numcols-1)
      a(rlb, clb:clb+numcols-1) = w(1:numcols)
    end if
  else if (myrow .eq. pivproc) then
    a(pivl+rlb-1, clb:clb+numcols-1) = w(BLKSIZE+5:BLKSIZE+numcols+4)
  end if

  if (myrow .eq. rproc) rlb = rlb + 1
  a(rlb:rub,clb+i-1) = a(rlb:rub,clb+i-1) / w(BLKSIZE+1)
  if (i < BLKSIZE) call dger(rub-rlb+1,numcols-i,-1.0d0,a(rlb,clb+i-1),1,w(i+1),1,a(rlb,clb+i),m)

end do

rlb = panels(p)%info(3)
do i = clb, clb + numcols - 1
   l((i-clb)*mloc+1:(i-clb+1)*mloc) = a(rlb:rlb+mloc-1,i)
end do
deallocate(w)

end with team
end subroutine fact
```

D.7 module_update.caf

```
! use currpanel to update numcol columns of the trailing matrix of a
subroutine update(m, n, coldiff, numcol, p)
use HPLmod
use support
integer :: m, n, coldiff, numcol, lb, ub, pos, ierr, k, jlb, mykey, iu, ipiv, nn, lb_p, tsize
double precision, pointer, DIMENSION(:,:) :: a
double precision, pointer, DIMENSION(:) :: c, piv, l, u
integer, pointer, DIMENSION(:) :: pairs, aloc,uloc,len1,len2, permute
integer, pointer :: flag, numpairs
double precision, allocatable :: w(:,:)[*]
```

```
      integer, allocatable :: wptr(:)[*]
      integer :: i, j, p, ii, mloc, nloc, iloc, jloc, mloc_p, src, dst, srcloc, dstloc, numrows
      integer :: fnds, fndd, rproc, srcprow, dstprow, wrows, wlb, mydist, partner, lessip2

      with team cteam

      iloc = panels(p)%info(3)
      jloc = panels(p)%info(4)
      mloc = panels(p)%info(5)
      nloc = panels(p)%info(6)
      rproc = panels(p)%info(7)
      a => ab(1:m,1:n)
      l => panels(p)%buff(1:mloc*BLKSIZE)
      c => panels(p)%buff(mloc*BLKSIZE+1:(mloc+BLKSIZE)*BLKSIZE)
      piv => panels(p)%buff((mloc+BLKSIZE)*BLKSIZE+1:(mloc+BLKSIZE+1)*BLKSIZE)
      u => panels(p)%buff((mloc+BLKSIZE+1)*BLKSIZE+1:(mloc+nloc+BLKSIZE+1)*BLKSIZE)
      permute => panels(p)%info(9:8*BLKSIZE+2*NPROW+10)
      nn = min(numcol, PROBLEMSIZE - panels(p)%info(2))
      allocate(w(nn+1, BLKSIZE)[], wptr(1)[])
      flag => permute(1)
      numpairs => permute(2)
      pairs => permute(3:4*BLKSIZE+2)
      aloc => permute(4*BLKSIZE+3:6*BLKSIZE+2)
      uloc => permute(6*BLKSIZE+3:8*BLKSIZE+2)
      len1 => permute(8*BLKSIZE+3:8*BLKSIZE+NPROW+2)
      len2 => permute(8*BLKSIZE+NPROW+3:8*BLKSIZE+2*NPROW+2)

      ! compute row permutation from pivot information
      if (flag .eq. 0) then
        ipiv = piv(1)
        pairs(1:2) = (/ipiv, panels(p)%info(1)/)
        numpairs = 1
        if (ipiv .ne. panels(p)%info(1)) then
          pairs(3:4) = (/panels(p)%info(1), ipiv/)
          numpairs = 2
        end if
        do i = 2, min(BLKSIZE, PROBLEMSIZE-panels(p)%info(2)+1)
          src = panels(p)%info(1) + i - 1
          dst = piv(i)
          fnds = 0
          fndd = 0
          do j = 1, numpairs
            if (fnds .eq. 0 .or. src .ne. dst .and. fndd .eq. 0) then
              if (src .eq. pairs((j-1)*2+2)) fnds = j
              if (src .ne. dst .and. dst .eq. pairs((j-1)*2+2)) fndd = j
            end if
          end do

          if (fnds .eq. 0) then
            pairs(numpairs*2+1:numpairs*2+2) = (/src, dst/)
            numpairs = numpairs + 1
            pos = numpairs
          else
            pairs((fnds-1)*2+2) = dst
            pos = fnds
          end if

          if (src .ne. dst) then
            if (fndd .eq. 0) then
              pairs(numpairs*2+1:numpairs*2+2) = (/dst, src/)
              numpairs = numpairs + 1
              pos = numpairs
            else
              pairs((fndd-1)*2+2) = src
              pos = fndd
            end if
          end if
          if (i .ne. pos) then
            src = pairs((i-1)*2+1)
            dst = pairs((i-1)*2+2)
            pairs((i-1)*2+1:(i-1)*2+2) = pairs((pos-1)*2+1:(pos-1)*2+2)
            pairs((pos-1)*2+1:(pos-1)*2+2) = (/src, dst/)
          end if
        end do

        ! compute the locations in a and u where rows should be placed
        ii = 1
```

```
  len2 = 0
  do i = 1, numpairs
    src = pairs((i-1)*2+1)
    srcprow = (src - 1) / BLKSIZE
    srcprow = mod(srcprow, NPROW)
    len2(srcprow+1) = len2(srcprow+1) + 1    !   1-based, procid also 1-based
    if (myrow .eq. srcprow) then
      srcloc = localsize(src, 0, BLKSIZE, NPROW, myrow)
      aloc(ii) = srcloc - iloc
      dst = pairs((i-1)*2+2)
      if (myrow .eq. rproc) then
        dstprow = (dst - 1) / BLKSIZE
        dstprow = mod(dstprow, NPROW)
        if (dstprow .eq. rproc) then
          uloc(ii) = dst - panels(p)%info(1)
          if (dst - panels(p)%info(1) .ge. BLKSIZE) then
            dstloc = localsize(dst, 0, BLKSIZE, NPROW, myrow)
            uloc(ii) = iloc - dstloc
          end if
        else
          fndd = 0
          do j = 1, numpairs
            if (fndd .eq. 0 .and. dst .eq. pairs((j-1)*2+1)) fndd = j
          end do
          uloc(ii) = pairs((fndd-1)*2+2) - panels(p)%info(1)
        end if
      else
        uloc(ii) = dst - panels(p)%info(1)
      end if
      ii = ii + 1
    end if
  end do
  flag = 1
end if

! permute rows and broadcast u
w = 0
numrows = len2(myrow+1)
len1(:) = len2(:)
if (myrow .eq. rproc) then
  do j = 0, nn - 1
    do i = 1, numrows
      if (uloc(i) .ge. 0) u(uloc(i)+j*BLKSIZE+1) = a(aloc(i)+iloc,jloc+coldiff+j)
      if (uloc(i) .lt. 0) a((-uloc(i))+iloc,jloc+coldiff+j) = a(aloc(i)+iloc,jloc+coldiff+j)
    end do
  end do
else
  w(1,1:numrows) = uloc(1:numrows)
  do k = 0, nn - 1
    do i = 1, numrows
      w(k+2,i) = a(aloc(i)+iloc, jloc+coldiff+k)
    end do
  end do
end if
if (myrow .eq. rproc) then
   len1(myrow+1) = 0
   numrows = 0
end if

wptr(1) = numrows
mydist = myrow - rproc
if (mydist < 0) mydist = mydist + NPROW
partner = ieor(mydist, rowp2)
call barrier()
if (NPROW - rowp2 .ne. 0 .and. partner .lt. NPROW) then
  partner = mod(partner + rproc, NPROW)
  if (mydist .eq. 0) then
    lb = (mloc+BLKSIZE+1)*BLKSIZE+1
    ub = (mloc+nloc+BLKSIZE+1)*BLKSIZE
    if (p .eq. 1) then
      mloc_p = panelinfo_1(5)[partner]
      lb_p = (mloc_p+BLKSIZE+1)*BLKSIZE+1
      panelbuff_1(lb_p : lb_p+ub-lb)[partner] = panelbuff_1(lb : ub)
    else
      mloc_p = panelinfo_2(5)[partner]
      lb_p = (mloc_p+BLKSIZE+1)*BLKSIZE+1
      panelbuff_2(lb_p : lb_p+ub-lb)[partner] = panelbuff_2(lb : ub)
```

```
    end if
  else if (mydist .eq. rowp2) then
    w(:, 1 : len1(myrow+1))[partner] = w(:, 1 : len1(myrow+1))
  else if (iand(mydist, rowp2) .ne. 0) then
    wlb = wptr(1)[partner]
    w(:, wlb+1 : wlb+len1(myrow+1))[partner] = w(:, 1 : len1(myrow+1))
  else if (len1(partner+1) .gt. 0) then
    wptr(1) = wptr(1) + len1(partner+1)
  end if
end if

call barrier()
partner = ieor(mydist, rowp2)
if (NPROW - rowp2 .ne. 0 .and. partner .lt. NPROW) then
  partner = mod(partner + rproc, NPROW)
  if (mydist .eq. 0 .and. len1(partner+1) .gt. 0) then
    do i = 1, len1(partner+1)
       iu = w(1,i) + 1  ! w(1,i) is 0-based
       u(iu:iu+(nn-1)*BLKSIZE:BLKSIZE) = w(2:nn+1,i)
    end do
  end if
end if

do i = 1, NPROW-rowp2-1
  lessip2 = mod(rproc+i, NPROW)
  len1(lessip2+1) = len1(lessip2+1) + len1(mod(lessip2+rowp2, NPROW)+1)
end do

mykey = 0
wrows = wptr(1)
if (mydist .lt. rowp2) mykey = 1
if (mydist .ge. rowp2 .and. NPROW - rowp2 .ne. 0) mykey = 2
call team_split(cteam, mykey, myrow, subcteam, mykey, ierr)

with team subcteam

if (mydist .lt. rowp2) then
  do i = 1, rowlog2
    partner = ieor(mydist, ibset(0, i-1))
    partner = mod(rproc + partner, NPROW)
    if (mydist .lt. ibset(0, i) .and. mydist .lt. ibset(0, i-1)) then
      lb = (mloc+BLKSIZE+1)*BLKSIZE+1
      ub = (mloc+nloc+BLKSIZE+1)*BLKSIZE
      if (p .eq. 1) then
        mloc_p = panelinfo_1(5)[partner]
        lb_p = (mloc_p+BLKSIZE+1)*BLKSIZE+1
        panelbuff_1(lb_p : lb_p+ub-lb)[partner] = panelbuff_1(lb : ub)
      else
        mloc_p = panelinfo_2(5)[partner]
        lb_p = (mloc_p+BLKSIZE+1)*BLKSIZE+1
        panelbuff_2(lb_p : lb_p+ub-lb)[partner] = panelbuff_2(lb : ub)
      end if
      if (len1(partner+1) > 0) w(:,wrows+1:wrows+len1(partner+1)) = w(:,1:len1(partner+1))[partner]
    else if (mydist .ge. ibset(0, i)) then
      w(:,wrows+1:wrows+len1(partner+1)) = w(:,1:len1(partner+1))[partner]
    end if
    call barrier()
    if (mydist .lt. ibset(0, i) .and. mydist .lt. ibset(0, i-1)) then
       do k = 0, nn - 1
         do j = 1, len1(partner+1)
            iu = w(1,wrows+j) + 1  ! w(1,i) is 0-based
            u(iu+k*BLKSIZE) = w(k+2,wrows+j)
         end do
       end do

    end if
    if (mydist .lt. ibset(0, i) .and. mydist .ge. ibset(0, i-1)) then
       do k = 0, nn - 1
         do j = 1, numrows
            a(aloc(j)+iloc, jloc+coldiff+k) = u(uloc(j)+1+k*BLKSIZE)
            u(uloc(j)+1+k*BLKSIZE) = w(k+2, j)
         end do
       end do
       do j = numrows + 1, len1(myrow+1)
          iu = w(1,j) + 1
          u(iu:iu+(nn-1)*BLKSIZE:BLKSIZE) = w(2:, j)
       end do
```

```
    end if
    if (mydist .ge. ibset(0, i) .or. mydist .lt. ibset(0, i-1)) wrows = wrows + len1(partner+1)

    do j = 0, rowp2 - 1
       partner = ieor(j,ibset(0,i-1))
       if (partner .gt. j) then
          partner = mod(rproc + partner, NPROW)
          len1(mod(rproc+j,NPROW)+1) = len1(mod(rproc+j,NPROW)+1) + len1(partner+1)
          len1(partner+1) = len1(mod(rproc+j,NPROW)+1)
       end if
    end do
  end do
else if (NPROW - rowp2 .gt. 1) then
  call team_broadcast(u, 0)
  do i = 1, numrows
    a(aloc(i)+iloc,jloc+coldiff:) = u(uloc(i)+1::BLKSIZE)
  end do
end if
call team_free(subcteam)

end with team

if (NPROW - rowp2 .ne. 0 .and. ieor(mydist, rowp2) .lt. NPROW) then
  partner = mod(rproc + ieor(mydist, rowp2), NPROW)
  if (iand(mydist, rowp2) .eq. 0) then
    lb = (mloc+BLKSIZE+1)*BLKSIZE+1
    ub = (mloc+nloc+BLKSIZE+1)*BLKSIZE
    if (p .eq. 1) then
      mloc_p = panelinfo_1(5)[partner]
      lb_p = (mloc_p+BLKSIZE+1)*BLKSIZE+1
      panelbuff_1(lb_p : lb_p+ub-lb)[partner] = panelbuff_1(lb : ub)
    else
      mloc_p = panelinfo_2(5)[partner]
      lb_p = (mloc_p+BLKSIZE+1)*BLKSIZE+1
      panelbuff_2(lb_p : lb_p+ub-lb)[partner] = panelbuff_2(lb : ub)
    end if
  end if
end if
deallocate(wptr, w)

call dtrsm('L', 'L', 'N', 'U', BLKSIZE, nn, 1.0d0, c, BLKSIZE, u, BLKSIZE)
if (mloc .gt. 0) call module_mmult(l(1),u(1),mloc,nn,iloc,jloc,mloc,coldiff,nn,m,rproc)

end with team

end subroutine update
```

D.8 module_solve.caf

```
! solve the upper triangle matrix
subroutine backsolve(m, n)
use HPLmod
use support
integer :: i, j, m, n, numblks, pcoln, prown, pcolb, bnxt, bblk, m1, n1, blk, l, pcolp, prowp
double precision, pointer, DIMENSION(:,:) :: a
double precision, pointer, DIMENSION(:) :: b
double precision, allocatable :: work(:)[*], buf(:)[*]
a => ab(1:m, 1:n)
b => ab(1:m, n+1)
numblks = (PROBLEMSIZE - 1) / BLKSIZE
pcoln = mod(numblks, NPCOL)            ! processors own the last column block of A
prown = mod(numblks, NPROW)            ! processors own the last row block of A
pcolp = pcoln
prowp = prown
pcolb = mod(PROBLEMSIZE / BLKSIZE, NPCOL)
blk = PROBLEMSIZE - numblks * BLKSIZE
l = PROBLEMSIZE - blk

with team rteam
allocate(buf(m)[])
if (pcoln .ne. pcolb .and. mycol .eq. pcolb) buf(1:m)[pcoln] = ab(1:m,n+1)

call barrier()
if (pcoln .ne. pcolb .and. mycol .eq. pcoln) b = buf

if (mycol .ne. pcoln) b = 0
```

```

      m1 = m
      n1 = n
      if (myrow .eq. prown .and. mycol .eq. pcoln) then
        do i = m, m-blk+1, -1
          b(i) = b(i) / a(i,i-m+n)
          b(m-blk+1:i-1) = b(m-blk+1:i-1) - b(i) * a(m-blk+1:i-1,i-m+n)
        end do
        x(n-blk+1:n) = b(m-blk+1:m)
      end if

      if (myrow .eq. prown) m1 = m - blk
      if (mycol .eq. pcoln) n1 = n - blk
      allocate(work(min((NPCOL-1)*BLKSIZE,m))[])

      do j = 1, numblks
        pcolp = pcoln - 1
        prowp = prown - 1
        if (pcolp .lt. 0) pcolp = pcolp + NPCOL
        if (prowp .lt. 0) prowp = prowp + NPROW
        if (NPCOL .ne. 1) bnxt = min(l, (NPCOL-1)*BLKSIZE)
        if (NPCOL .eq. 1) bnxt = min(l, (NPROW-1)*BLKSIZE)
        bblk = localsize(bnxt, max(l-bnxt,0), BLKSIZE, NPROW, myrow)

        if (mycol .eq. pcoln) then
          call team_broadcast(x(n1+1:n1+blk), prown, cteam)
          if (bblk .gt. 0) then
            b(m1-bblk+1:m1)= b(m1-bblk+1:m1)-matmul(a(m1-bblk+1:m1,n1+1:n1+blk),x(n1+1:n1+blk))
            work(1:bblk)[pcolp] = b(m1-bblk+1:m1)
          end if
        end if

        call barrier()

        if (mycol .eq. pcolp .and. bblk .gt. 0) then
          b(m1-bblk+1:m1) = b(m1-bblk+1:m1) + work(1:bblk)
        end if

        if (myrow .eq. prowp .and. mycol .eq. pcolp) then
          do i = m1, m1-BLKSIZE+1, -1
            b(i) = b(i) / a(i,i-m1+n1)
            b(m1-BLKSIZE+1:i-1) = b(m1-BLKSIZE+1:i-1) - b(i) * a(m1-BLKSIZE+1:i-1,i-m1+n1)
          end do
          x(n1-BLKSIZE+1:n1) = b(m1-BLKSIZE+1:m1)
        end if

        if (mycol .eq. pcoln .and. m1-bblk .ge. 1) then
          b(1:m1-bblk) = b(1:m1-bblk) - matmul(a(1:m1-bblk,n1+1:n1+blk),x(n1+1:n1+blk))
        end if
        l = l - BLKSIZE
        pcoln = pcolp
        prown = prowp
        blk = BLKSIZE
        if (myrow .eq. prown) m1 = m1 - blk
        if (mycol .eq. pcoln) n1 = n1 - blk
      end do

      if (mycol .eq. pcolp) call team_broadcast(x(1:BLKSIZE), prowp, cteam)
      deallocate(work, buf)
      end with team

      end subroutine backsolve
```

D.9 module_mmult.caf

```
      subroutine module_mmult(l,u,s,t,iloc,jloc,mloc,coldiff,nn,m,rproc)
      use HPLmod
      double precision :: l(s,BLKSIZE), u(BLKSIZE,t)
      double precision, pointer, dimension(:,:) :: a
      integer s, t, iloc, jloc, mloc, nn, coldiff,rproc, blk

      a => ab(iloc:iloc+s-1,jloc+coldiff:jloc+coldiff+nn-1)
      if (myrow .ne. rproc) then
        call dgemm('N','N',s,t,BLKSIZE,-1.0d0,l(1,1),s,u(1,1),BLKSIZE,1.0d0,a(1,1),m)
      else
        blk = min(s, BLKSIZE)
```

```
  if (s .le. blk) then
    a(1:blk,:) = u(1:blk,:)
  else
    call dgemm('N','N',s-blk,t,BLKSIZE,-1.0d0,l(blk+1,1),s,u(1,1),BLKSIZE,1.0d0,a(blk+1,1),m)
    a(1:blk,:) = u(1:blk,:)
  end if
end if

end subroutine module_mmult
```

D.10 module_support.caf

```
module support
contains
subroutine reduce_max(in1, in2, out, nb)
integer :: nb
double precision :: in1, in2, out
out = max(in1, in2)
end subroutine reduce_max

subroutine reduce_sum(in1, in2, out, nb)
integer :: nb
double precision :: in1(nb/8), in2(nb/8), out(nb/8)
out = in1 + in2
end subroutine reduce_sum

! use user-defined type
subroutine select_max(in1, in2, out, nb)
use HPLmod
integer :: nb, B
double precision :: in1(nb/8), in2(nb/8), out(nb/8)
B = BLKSIZE
if ((nb .eq. 8 .and. abs(in1(1)) .lt. abs(in2(1))) .or. &
    (nb .eq. 8*(B+4) .and. (abs(in1(B+1)) .lt. abs(in2(B+1)) .or. &
      (abs(in1(B+1)) .eq. abs(in2(B+1)) .and. in1(B+3) .lt. in2(B+3))))) then
  out = in2
else
  out = in1
end if
end subroutine select_max

! compute a local portion of the array from global index i (0 based)
function localsize(n, i, blk, np, myproc) result(mypart)
integer :: n, i, blk, np, myproc, numblks, locblks, mydist, inb, srcproc, inbs, mypart
inb = i / blk
srcproc = inb - (inb / np) * np
inbs = blk - (i - inb * blk)
numblks = (n - inbs) / blk + 1
locblks = numblks / np

if (myproc .eq. srcproc) then
  mypart = n
  if (n .gt. inbs) then
    if (numblks .ne. locblks * np) mypart = inbs + locblks * blk
    if (numblks .eq. locblks * np) mypart = n + (locblks - numblks) * blk
  end if
else
  mypart = 0
  if (n .gt. inbs) then
    mydist = myproc - srcproc
    if (mydist .lt. 0) mydist = mydist + np
    if (numblks .lt. np) then
      if (mydist .lt. numblks) mypart = blk
      if (mydist .eq. numblks) mypart = n - inbs + blk * (1 - numblks)
    else
      mydist = mydist - numblks + locblks * np
      mypart = locblks * blk
      if (mydist .lt. 0) mypart = mypart + blk
      if (mydist .eq. 0) mypart = mypart + n -inbs - numblks * blk + blk
    end if
  end if
end if
end function localsize

end module support
```

E UTS

E.1 uts.caf

```

module uts
        use array_queue
        implicit none

        integer m !holds the number of children
        integer*8 total_node_count !total number of nodes processed by each process

  !lifeline threshold holds the minimum work a process has in queue
        ! before pushing work to others.

        !max_neighbor index controls the number of lifelines; as in
        !                maxlifeline = 2^(max_neighbor_index -1)

        !gen_mx is the depth

        integer world_size, my_rank, lifeline_threshold, gen_mx, max_neighbor_index

        !steal_trunk_size/push_trunk_size ,generally set to 8 elements, hold the maximum
        ! number of work items that can be obtained in a steal/push; this is because of the
        !       AMmediumpacket size limit
        integer steal_trunk_size, push_trunk_size, max_allowed_unsuccessful_steals, steal_threshold

        character (kind=1, len=20) :: root_descriptor
        integer yield_count, b_0, seed, log_world_size, last_node_index_work_pushed

        !holds the lifelines; a process toggles a bit on the incoming_lifeline
        ! to indicate the setup of a lifeline; the bit position indicates the relative position
        ! (process_rank_on_which_lifeline - 2^bit position) of the process
        integer(4) :: incoming_lifeline

        integer :: activate_running
        integer(8) tstart, tend, rate, alarm_count
        real tsec

        EXTERNAL                rng_spawn
        EXTERNAL                rng_rand
        INTEGER   rng_rand
        EXTERNAL                rng_init

contains

        !a standard initialization of the random generator
        !in fortran. attribution:world wide web.

        SUBROUTINE init_random_seed()
                INTEGER :: i, n, clock
                INTEGER, DIMENSION(:), ALLOCATABLE :: seed
                CALL RANDOM_SEED(size = n)
                ALLOCATE(seed(n))
                CALL SYSTEM_CLOCK(COUNT=clock)
                seed = clock + 37 * (my_rank + 1) * (/ (i - 1, i = 1, n) /)
                CALL RANDOM_SEED(PUT = seed)
                DEALLOCATE(seed)
        END SUBROUTINE

        !the kernel of the computation; the queue consists
        ! of the nodes of the UTS tree whose subtree has to be built.
        ! in this subroutine, a node is deleted from teh queue,
        ! the number of children and the children's descriptors
        ! are determined and added to the queue.
        ! the children's descriptors are found using the rng_spawn
        ! from the brg_sha1 library implementation.

        subroutine process_work_item(descriptor, depth)
          implicit none
                integer*4, intent(in)::depth
                integer*4 child_depth
                character (*) descriptor
                character (kind=1, len=20) :: child_descriptor
                integer num_child, child_index
                logical return_val
```

```
                num_child = get_number_children(descriptor, depth)
                total_node_count = total_node_count+1
                child_depth = depth+1
                do child_index=0,(num_child-1)
                    call rng_spawn(descriptor, child_descriptor, child_index)
                    if(insert_queue(child_descriptor, child_depth) .eq. .false.) then
                        call exit()
                    endif
                enddo
            end subroutine

            function get_lifeline(i) result(lifeline)
              implicit none
                integer i, lifeline
                lifeline = mod(my_rank - (2**(i-1)) + world_size, world_size)
            end function

function get_num_set_bits(inlifeline) result(setbitcnt)
  implicit none
  integer*4 inlifeline
  integer setbitcnt, a
  setbitcnt = 0
  a = inlifeline
  do while (a .ne. 0)
    a = IAND(a, a-1)
    setbitcnt = setbitcnt + 1
  enddo
end function

            function get_num_items_to_push(inlifeline) result(numitems)
                implicit none
                integer numitems
                integer*4 inlifeline
                numitems = CEILING((queue_count * 1.0) / (get_num_set_bits(inlifeline)+1.0))
            end function

            !push_work determines the processes registered on the incoming_lifeline
            ! and pushed work to them. It also remembers the process for which
            ! the least amount of work was pushed.

            subroutine push_work()
                implicit none
                integer :: i, num_bits_to_beshifted, num_items_to_push, push_flag
                integer*4 :: copy_inlifeline, copy_inlifeline2
                logical :: do_not_set_anymore
                integer :: new_last_node_index_work_pushed

                do_not_set_anymore = .false.
                i = last_node_index_work_pushed
                new_last_node_index_work_pushed = last_node_index_work_pushed
                num_bits_to_beshifted = (i-1)
                copy_inlifeline2 = incoming_lifeline
                incoming_lifeline = 0
                copy_inlifeline = ISHFTC(copy_inlifeline2,-num_bits_to_beshifted,log_world_size)
                num_items_to_push = get_num_items_to_push(copy_inlifeline)

                do while (copy_inlifeline .ne. 0)
                    push_flag = IAND(1, copy_inlifeline)
                    if (push_flag .eq. 1) call push_work_while_sharing(i, num_items_to_push)
                    copy_inlifeline = ISHFT(copy_inlifeline, -1)
                    i = i + 1
                    if(i .eq. (log_world_size+1)) i=1
                    if ((queue_count) .lt. ((num_items_to_push)/4)) then
                        if((copy_inlifeline .ne. 0) .and. (do_not_set_anymore .eq. .false.)) then
                            last_node_index_work_pushed = i
                            do_not_set_anymore = .true.
                        endif
                    endif
                end do

                if((last_node_index_work_pushed .ne. 1) .and. do_not_set_anymore) then
                    copy_inlifeline = 2**(log_world_size) - 1
                    copy_inlifeline = ISHFT(copy_inlifeline, last_node_index_work_pushed-1)
                    copy_inlifeline2 = IAND(copy_inlifeline2, copy_inlifeline)
                    incoming_lifeline = IOR(incoming_lifeline, copy_inlifeline2)
                endif
```

```
150             if(last_node_index_work_pushed .eq. new_last_node_index_work_pushed) then
151                 last_node_index_work_pushed = 1
152             endif
153         end subroutine
154
155         !activate; each process after finishing its quota of work performs
156         !a random steal indicated by the line 188 followed by setting up of
157         ! lifelines.
158         !yield_count controls when a process should look at pushing work/
159         !allow stealing of work.
160
161         recursive subroutine activate()
162             character (kind=1, len=20) :: descriptor
163             integer steal_from_img, processed_count, num_items_to_push
164             integer i, j, lifeline, push_flag, neighbor_index, next_neighbor
165             logical return_val
166             integer*4 copy_inlifeline, depth
167
168             activate_running = 1
169             processed_count = 0
170             do while(queue_count .gt. 0)
171                 if(delete_queue_end(descriptor, depth) .eq. .false.) then
172                     call exit()
173                 endif
174                 processed_count = processed_count +1
175                 call process_work_item(descriptor, depth)
176                 if (processed_count .eq. yield_count) then
177                     processed_count = 0
178                     call caf_async_advance
179                     if ((incoming_lifeline .ne. 0) .and. (queue_count .ge. lifeline_threshold)) then
180                             call push_work()
181                     endif
182                 endif
183             enddo
184
185             activate_running = 0
186             if(world_size .gt. 1) then
187                 steal_from_img = get_random_image(my_rank, my_rank)
188                 spawn steal_work_spawn(my_rank, 0)[steal_from_img]
189
190                 neighbor_index = 0
191                 do while (neighbor_index .lt. max_neighbor_index)
192                     next_neighbor = mod(my_rank+(2**neighbor_index), world_size)
193                     spawn set_lifelines(my_rank, neighbor_index)[next_neighbor]
194                     neighbor_index = neighbor_index + 1
195                 enddo
196             end if
197         end subroutine
198
199         !pack a set of work items to push to a process.
200
201         subroutine push_work_while_sharing(neighbor_index, num_items_to_push)
202           implicit none
203             integer size, j, neighbor_index, dest, num_items_to_push
204             logical return_val
205             type (queue_element) :: push_work_descriptor(10)
206             character (kind=1, len=20) :: descriptor
207             integer*4 :: lifeline_status, depth
208
209             size = min(queue_count-1, num_items_to_push)
210             size = min(size, push_trunk_size)
211             if(size .gt. 0) then
212         do j=1,size
213                     if(delete_queue_begin(descriptor, depth) .eq. .false.) then
214                         call exit()
215                     endif
216                     push_work_descriptor(j)%desc = descriptor
217                     push_work_descriptor(j)%depth = depth
218         enddo
219                 dest = get_lifeline(neighbor_index)
220                 spawn copy_item_and_activate(push_work_descriptor, size)[dest]
221             endif
222         end subroutine
223
224         !a successful steal/(work pushed from aprocess) arrive into a queue
225         ! via this below function.
226
```

```
  subroutine copy_item_and_activate(descriptor, num_items)
                  implicit none
                  type(queue_element) :: descriptor(10)
                  integer num_items, i
                  logical return_val

                  do i=1,num_items
                          if(insert_queue(descriptor(i)%desc, descriptor(i)%depth) .eq. .false.) then
                                  call exit()
                          endif
                  enddo

                  if (activate_running .eq. 0) then
                    activate_running = 1
                          call activate()
                  endif
  end subroutine

  !apart from setting of lifeline, a process also looks
  ! at stealing from the node

  subroutine set_lifelines(home, neighbor_index)
    implicit none
          integer :: home, neighbor_index, i, size, next_neighbor
          logical :: return_val
          integer*4 :: lifeline_status, depth
          type (queue_element) :: steal_work_descriptor(10)
          character (kind=1, len=20) :: descriptor

          incoming_lifeline = IOR(incoming_lifeline, ISHFT(1, neighbor_index))
          if (queue_count .gt. steal_threshold) then
            size = min(steal_trunk_size, (queue_count/2))
do i=1,size
                          if(delete_queue_begin(descriptor, depth) .eq. .false.) then
                                  call exit()
                          endif
                          steal_work_descriptor(i)%desc = descriptor
                          steal_work_descriptor(i)%depth = depth
enddo
                  spawn copy_item_and_activate(steal_work_descriptor, size)[home]
          end if
  end subroutine

  !this function is spawned on a process to steal work.

  subroutine steal_work_spawn(home, num_unsuccessful_steal_attempt)
    implicit none
          integer next_neighbor, home, num_unsuccessful_steal_attempt, steal_from_img
          character (kind=1, len=20) :: descriptor
          integer*4 depth
          type (queue_element) :: steal_work_descriptor(10)
          integer i, lifeline_img, work_count, size, neighbor_index
          logical return_val

          if (queue_count .gt. steal_threshold) then
            size = min(steal_trunk_size, (queue_count/2))
do i=1,size
                          if(delete_queue_begin(descriptor, depth) .eq. .false.) then
                                  call exit()
                          endif
                          steal_work_descriptor(i)%desc = descriptor
                          steal_work_descriptor(i)%depth = depth
enddo
                  spawn copy_item_and_activate(steal_work_descriptor, size)[home]
          else
                  num_unsuccessful_steal_attempt = num_unsuccessful_steal_attempt + 1
          endif
  end subroutine

  !determines the number of children; the algorithm is followed from
  ! uts 1.1. library released from ohio state university.

  function get_number_children(descriptor, depth) result(n)
          implicit none
          integer*4, intent(in)::depth
          character (*) descriptor
          integer v, n
```

```
                real p_0, d, b_i
                v = rng_rand(descriptor)
                if(v .lt. 0) then
                        v = 0
                endif
                d = v
                d = (d/HUGE(v))

                if (depth .lt. gen_mx) then
                        b_i = m
                else
                        b_i = 0
                endif
                p_0 = (1.0 / (1.0 + b_i))

                if( (d .eq. 1) .or. (p_0 .eq. 1)) then
                        n = 0
                else
                        n = (floor(alog(1-d)/alog(1-p_0)))
                endif
        end function

        !used for verification

        subroutine send_my_count(other_cnt)
                integer*8 other_cnt
                total_node_count = total_node_count + other_cnt
        end subroutine

        function get_random_image(avoid_img1, avoid_img2) result(img)
                        integer avoid_img1, avoid_img2, img
                        character (kind=1, len=20) :: descriptor
                        real harvest
                        call RANDOM_NUMBER(harvest)
                        img = mod (INT(harvest * world_size), world_size)
                        if(img .eq. avoid_img1) img = mod(img+2, world_size)
                        if(img .eq. avoid_img2) img = mod(img+2, world_size)
        end function

end module

program s
        use uts
  implicit none
        character (kind=1, len=20):: child_descriptor
        integer*4 depth, step
        integer i,j, k, desc_count, img, write_to, work_items_per_image
        logical return_val
        type(queue_element) :: push_work_descriptor(10)
        type(queue_element) :: push_desc(10)

        gen_mx = 18
        m=4
        seed = 19
        total_node_count=0
        b_0 = m

  my_rank = team_rank()
  world_size = team_size()

        call init_random_seed()
        call init_queue(40000)

        log_world_size = (alog(real(world_size))/alog(real(2)))
        max_allowed_unsuccessful_steals = 1
        incoming_lifeline = 0
        last_node_index_work_pushed = 1
        max_neighbor_index = log_world_size

        yield_count = 64
        lifeline_threshold = 1*(max_neighbor_index)
        steal_threshold = 2
        push_trunk_size = 8
        steal_trunk_size = 8

        step = 1
        call team_barrier()
```

```
        call system_clock(tstart, rate)
        call rng_init(root_descriptor, seed)

        !process 0 generates some nodes before distributing to others.

        if(my_rank .eq. 0) then
                do i=0,(b_0-1)
                                call rng_spawn(root_descriptor, child_descriptor, i)
                                return_val = insert_queue(child_descriptor, 1)
                enddo
                do while(queue_count .lt. 4*(world_size/step))
                        if(delete_queue_begin(child_descriptor, depth) .eq. .false.) then
                                call exit()
                        endif
                        call process_work_item(child_descriptor, depth)
                enddo
        endif

        work_items_per_image = 4
        !activate_running = 1
        call team_barrier()

        !use function shipping to distribute work to others;
        !after distribution the processes initially dormant start working
        ! via the call activate present in copy_item_and_activate
        finish
                if(my_rank .eq. 0) then
                        do i=1,world_size-1, step
                                j=1
                                do while((j .le. work_items_per_image) .and. (queue_count .gt. 0))
                                        return_val = delete_queue_begin(push_desc(j)%desc, push_desc(j)%depth)
                                        j = j+1
                                enddo
                                if(j .gt. 1) then
                                        spawn copy_item_and_activate(push_desc, (j-1))[i]
                                endif
                        enddo
                        call activate()
                endif
        end finish

        if (0 .eq. my_rank) then
        call system_clock(tend, rate)
    tsec = real(tend-tstart) / rate
    write (*,*) "Elapsed␣time:␣", tsec
        endif

        call team_barrier()

        !printing out the work done by each process
        do i=0,world_size-1
                if(my_rank .eq. i) then
                        write(*,*) my_rank,"'s␣total␣node␣count␣is␣:",total_node_count
                endif
                call team_barrier()
        enddo

        !accumulating the total node count
        finish
                if(my_rank .ne. 0) then
                        spawn send_my_count(total_node_count)[0]
                endif
        end  finish

        if(my_rank .eq. 0) then
                write(*,*) my_rank,"␣has␣the␣total␣node␣count:",total_node_count
        endif
        call team_barrier()
end program
```

E.2 queue.caf

```
!typical double ended queue implementation

module type_queue_element
        type queue_element
```

```fortran
                    sequence
                    !to hold the descriptor of the tree node
                    character (kind=1,len=20) :: desc
                    !to hold the depth of the tree node
                    integer*4 :: depth
            end type queue_element
    end module
    module array_queue
            use type_queue_element
            implicit none
type (queue_element), allocatable, dimension (:) :: queue
            integer :: queue_head, queue_count, queue_tail
            integer :: max_elements
    contains
            subroutine init_queue(max_elem)
                    implicit none
                    integer, intent(in) :: max_elem
                    queue_head = -1
                    queue_tail = -1
                    queue_count = 0
                    max_elements = max_elem
                    allocate (queue(0:max_elem-1))
            end subroutine
            function insert_queue(descriptor, depth) result(inserted)
                    implicit none
                    logical :: inserted
                    character (kind=1,len=20), intent(in) :: descriptor
                    integer*4, intent(in) :: depth
                    if(queue_count .eq. 0) then
                            queue_head = 0
                            queue_tail = 0
                            queue(queue_tail)%desc = descriptor
                            queue(queue_tail)%depth = depth
                            queue_count = queue_count + 1
                            inserted = .true.
                    else if (queue_count .eq. max_elements) then
                            inserted = .false.
                    else
                            queue_tail = (mod(queue_tail+1,max_elements))
                            queue(queue_tail)%desc = descriptor
                            queue(queue_tail)%depth = depth
                            queue_count = queue_count + 1
                            inserted = .true.
                    endif
            end function
            function delete_queue_begin(descriptor, depth) result(success)
                    implicit none
                    logical :: success
                    character (kind=1,len=20) :: descriptor
                    integer*4,intent(out) :: depth
                    if(queue_count .eq. 0) then
                            success = .false.
                    else if(queue_count .eq. 1) then
                            descriptor = queue(queue_head)%desc
                            depth = queue(queue_head)%depth
                            queue_head = -1
                            queue_tail = -1
                            queue_count = 0
                            success = .true.
                    else
                            descriptor = queue(queue_head)%desc
                            depth = queue(queue_head)%depth
                            queue_head = mod(queue_head+1,max_elements)
                            queue_count = queue_count - 1
                            success = .true.
                    endif
            end function
            function delete_queue_end(descriptor, depth) result(success)
                    implicit none
                    logical :: success
                    character (kind=1,len=20) :: descriptor
                    integer*4,intent(out) :: depth
                    if(queue_count .eq. 0) then
                            success = .false.
                    else if(queue_count .eq. 1) then
                            descriptor = queue(queue_tail)%desc
                            depth = queue(queue_tail)%depth
```

```
                queue_head = -1
                queue_tail = -1
                queue_count = 0
                success = .true.
            else
                descriptor = queue(queue_tail)%desc
                depth = queue(queue_tail)%depth
                queue_tail = queue_tail - 1
                if(queue_tail .lt. 0) queue_tail=max_elements-1
                queue_count = queue_count - 1
                success = .true.
            endif
        end function
    end module
```