HPC Challenge: Five Benchmarks of Interest in Chapel*

Bradford L. Chamberlain Sung-Eun Choi Vassily Litvinov
Tom Hildebrandt Greg Titus
The Chapel Team
Cray, Inc.
chapel_info@cray.com

Johnathan Claridge John G. Lewis Kristi Maschhoff
claridge@gmail.com jglewis@comcast.net kristyn@cray.com

Abstract

This report presents our best-to-date Chapel implementations of the HPC Challenge benchmarks STREAM Triad, Random Access, and HPL, the HPC Graph Analysis benchmark Scalable Synthetic Compact Applications #2 (SSCA#2), and a Adaptive Mesh Refinement (AMR) code.

The highlights of this year's submission include:

- HPCC STREAM Triad (EP and Global)
- HPCC Global Random Access
- HPCC HPL
- A graph-type independent implementation of the HPC Graph Analysis SSCA#2 benchmark version 1.2
- A rank-independent Adaptive Mesh Refinement infrastructure code

The Chapel compiler and these benchmarks are publicly available at http://sourceforge.net/projects/chapel.

1 Overview and Contents

Chapel is a new parallel programming language under development at Cray, Inc. as part of DARPA's High Productivity Computing Systems (HPCS) program. The goal of the Chapel project is to improve parallel programmability, portability, and code robustness as compared to current programming models while producing programs with performance comparable to or better than MPI. Chapel is very much a work in progress, and as such, this report should be viewed as a snapshot of Chapel's current status.

In this report, we present our best-to-date Chapel implementations of three HPC Challenge (HPCC) benchmarks—STREAM Triad, Random Access, and HPL. We also present our implementation of the (SSCA#2) graph analysis benchmark and an Adaptive Mesh Refinement infrastructure code. For each benchmark, we provide a brief overview of our Chapel implementation and source line count.

2 Benchmark Descriptions

In this section, we give a brief description and source line counts of each of the Chapel benchmarks. All source line counts exclude comments, but include any printing or validation required by the benchmark. All three of the HPCC benchmarks share a problem size generation module.

HPCC Problem Size Module Source lines of code: 56

Complete code listings are given in the appendix, with the exception of SSCA#2 and the AMR infrastructure code due to size. For SSCA#2 we include one of the graph generation algorithms and the graph-independent portions. We refer readers to the Chapel code repository for a copy of the other codes.

*This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001.

2.1 HPCC EP and Global STREAM Triad

We present both EP (embarrassingly parallel) and global versions of the STREAM Triad benchmark. The global version is far more elegant in Chapel due to its support for a global-view programming model. Chapel's multi-level design allows us to fragment execution across the locales and implement an EP version of the STREAM Triad benchmark as well.

The core computation of both versions is as follows:

```
forall (a, b, c) in (A, B, C) do
  a = b + alpha * c;
```

This pair of lines specifies parallel, element-wise iteration over the vectors A, B, and C, referring to corresponding elements as a, b, and c in the loop body. The global STREAM implementation uses Chapel's Block distribution module to distribute the arrays A, B, and C.

EP STREAM Source lines of code: 72

Global STREAM Source lines of code: 68

2.2 HPCC Random Access

The Random Access benchmark computes pseudo-random updates to a large distributed table T of 64-bit unsigned integer values. As in STREAM, our distributed memory implementation uses two *Block* distributions—one to distribute the set of N_U table updates represented using a domain named *Updates*, and a second to distribute the table T and its corresponding domain.

The core of the Chapel implementation is as follows:

```
forall ( , r) in (Updates, RAStream()) do
  on TableDist.idxToLocale(r & indexMask) do {
    const myR = r;
    local {
      T(myR & indexMask) ^= myR;
    }
 }
```

The *local*-block tells the compiler that there will not be any communication within this block.

Global RA Source lines of code: 58

RA Random Stream Source lines of code: 50

2.3 HPCC HPL

The Chapel version of HPL uses the Chapel Block-Cyclic module for performing the dense LU factorization.

The distributed matrix-multiply for the HPL benchmarks is as follows:

```
forall (row,col) in Rest by (blkSize, blkSize) {
  const RestLcl = Rest;
  local {
    for a in (RestLcl.dim(1))(row..#blkSize) do
      for w in 1..blkSize do
        for b in (RestLcl.dim(2))(col..#blkSize) do
          Ab[a,b] -= replA[a,w] * replB[w,b];
  }
}
```

The outer *forall* loop processes each block in the matrix and performs a local matrix-multiply using $replA$ and $replB$, the necessary part of the blocks that are replicated on each locale.

Global HPL Source lines of code: 162

2.4 HPC Graph Analysis SSCA#2 version 1.2

SSCA2#2 is comprised of four "kernels" that require irregular accesses to a directed, weighted graph. The first kernel builds the graph representation. We have included the code for the R-MAT (Recursive Matrix) type graph generator (Kernel 1) and its support module in the appendix. Code for three other torus graph generators are available from the Chapel source repository. Kernels 2–4 are written in a completely graph-type independent manner. That is, the same kernel code is used for all graph types. The code is written using generic programming support and iterators which are implemented by the graph support module.

Chapel's global-view programming model results in a very clean code that is also devoid of an notion of distributed data. For example, the R-MAT graph is distributed using Chapel's Block distribution module, but Kernels 2–4 are completely unaware of this.

SSCA#2 R-MAT Kernel 1 Source lines of code: 154

SSCA#2 Kernels 2–4 Source lines of code: 326

SSCA#2 R-MAT Support module Source lines of code: 56

SSCA#2 driver: 227

2.5 Adaptive Mesh Refinement

Adaptive Mesh Refinement is a methodology applied to structured grids in which the problem space can be dynamically divided up into smaller grids. Adaptive Mesh Refinement was identified as one of 13 computations that represented an important class of applications for current and future parallel computing platforms [1].

The AMR infrastructure code in Chapel is completely rank-independent. Specifically, the dimensionality of the grid is not visible in the code at all. Again, Chapel's generics support and iterators enable the code to be free of all indexing expressions. The rank of the problem is a compile-time constant that can be changed for the problem of interest.

AMR Source lines of code: 623

AMR Support Source lines of code: 663

AMR Boundary Source lines of code: 364

AMR Grid Source lines of code: 322

AMR Level Source lines of code: 301

3 Preliminary Performance Results

At the time of this writing, we are still working to get performance numbers on an Cray XE6 ™with 24-core nodes (dual 12-core AMD processors running at 2.0GHz). We are still tuning our Chapel codes, but we are also experimenting with a new (pre-release) version of the GASNet networking library that has native support for the Cray XE6 network. The release version of GASNet should be available by the date of SC11. Here we present preliminary performance results.

For EP STREAM, we ran on up to 32 nodes for both the Chapel version and the HPCC reference version 1.4.1. As the number of nodes increases, we see the Chapel version within 5% of the reference version.

For Random Access, we have been struggling with running the reference version on the Cray XE6 and have yet to collect comparison numbers. We did initial experiments on Jaguar, a Cray XT5™system which showed the Chapel version to perform about 2 times slower than the reference version (our 2009 submission was four times slower).

For HPL, we do not expect to be very competitive with the reference version because we have spent most of our time writing and tuning a general Chapel *domain map* (distribution) that is used in the distributed matrix multiplication section (shown

[1] http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

nodes	Chapel EP STREAM	HPCC EP STREAM
1	38.2	39.1
2	75.3	78.3
4	151.8	156.5
8	302.7	313.2
16	606.6	626.2
32	1205.5	1251.8

Table 1: EP STREAM Triad performance for Chapel and HPCC v1.4.1 in GB/s.

above). Due to our efforts, this version is now dominated by the scalar computation. For the final submission, we may compare these portions of the HPL benchmark rather than the entire benchmark.

For SSCA#2, though this is our initial implementation and we have no reference version to compare to, we have been able to run SSCA#2 with a 8192 node R-MAT graph with maximum out degree of 126 on 20 nodes. Kernel 4, which computes *betweenness centrality*, is typically of the most interest and thus far we have spent most of our time on it. We achieved 371735.0 TEPS (Traversed Edges Per Second) for Kernel 4.

At higher node counts (which go up by a factor of two with each "scale"), our graph generation portion (Kernel 1) consumed an unreasonable amount of time, and we opted to stop the runs before getting through to Kernel 4. We now plan to focus more attention on Kernel 1.

Acknowledgments

The authors would like to gratefully acknowledge our former team members and collaborators who assisted with our 2006, 2008, and 2009 entries in the HPCC competition: Steve Deitz, Samuel Figueroa, Mary Beth Hribar, David Iten, Andrew Stone, Adrian Tate, and Wayne Wong. In addition, we thank all of Chapel's past contributors and early users for helping us reach this stage.

A EP STREAM Triad

```
//
// Embarassingly Parallel Implementation of STREAM Triad
//
// This version of the stream benchmark is not as elegant as
// stream.chpl.  It is a per-locale code with no communication in the
// actual computation.  It highlights the ability of a Chapel
// programmer to escape the global-view programming model and write
// codes with a fragmented, per-locale model.
//
// Comments marked with '***' point out differences with the global
// version of this benchmark in stream.chpl.
//

//
// *** Since this benchmark is written on a per-locale basis, there is no
// *** need to use the Block distribution.  The following use omits
// *** BlockDist, included in stream.chpl.
//
// Use standard modules for Timing routines, Type utility functions,
// and Random numbers
//
use Time, Types, Random;

//
// Use shared user module for computing HPCC problem sizes
//
use HPCCProblemSize;

//
// The number of vectors and element type of those vectors
//
const numVectors = 3;
type elemType = real(64);

//
// *** To ensure a local problem size, we spoof the number of vectors
// *** passed to the computeProblemSize function to be the number of
// *** vectors times the number of locales.
//
// Configuration constants to set the problem size (m) and the scalar
// multiplier, alpha
//
config const m = computeProblemSize(numVectors*numLocales, elemType),
             alpha = 3.0;

//
// Configuration constants to set the number of trials to run and the
// amount of error to permit in the verification
//
config const numTrials = 10,
             epsilon = 0.0;

//
// Configuration constants to indicate whether or not to use a
// pseudo-random seed (based on the clock) or a fixed seed; and to
// specify the fixed seed explicitly
//
config const useRandomSeed = true,
             seed = if useRandomSeed then SeedGenerator.currentTime else 31415926535;

//
// *** To ensure determinism of output, there is no more printing of the
// *** arrays in initVectors and verifyResults.
//
//
// Configuration constants to control what's printed -- benchmark
// parameters and/or statistics
//
config const printParams = true,
             printStats = true;

//
// The program entry point
//
proc main() {
  printConfiguration();   // print the problem size, number of trials, etc.

  //
  // *** Aggregates for collecting per-locale results for the minimum
  // *** execution time per trial, and whether verification passed
  //
  var minTimes: [LocaleSpace] real;
  var validAnswers: [LocaleSpace] bool;

  //
  // *** Fragment control so that we have a single task running on
  // *** every locale.
  //
  coforall loc in Locales do on loc {

    //
    // *** We declare these variables outside of the local block since
    // *** we'll need to access them when we write back to the global
    // *** aggregates declared above.
    //
    var validAnswer: bool;
    var execTime: [1..numTrials] real;

    //
    // *** Indicates that all of the code in this block is local to
    // *** this locale.  There is no communication.  A violation will
    // *** result in an error, though error checking is disabled with
    // *** --fast or --no-checks.
    //
    local {

      //
      // *** A, B, and C are the three local vectors
      //
      var A, B, C: [1..m] elemType;

      initVectors(B, C);    // Initialize the input vectors, B and C

      for trial in 1..numTrials {                       // loop over the trials
        const startTime = getCurrentTime();             // capture the start time

        //
        // *** The main loop looks identical to stream.chpl.  However,
        // *** in this version we are iterating over arrays that are
        // *** not distributed.
        //
        forall (a, b, c) in (A, B, C) do
          a = b + alpha * c;

        execTime(trial) = getCurrentTime() - startTime;  // store the elapsed time
      }

      validAnswer = verifyResults(A, B, C);             // verify...
    }

    //
    // *** Write times and verification result into aggregates
    // *** declared above.  These are declared over LocaleSpace so we
    // *** can write to them in parallel.
    //
    minTimes[here.id] = min reduce execTime;
    validAnswers[here.id] = validAnswer;
  }

  //
  // *** Pass minimum, average, and maximum times to printResults
  //
  printResults(&& reduce validAnswers, minTimes);
}

//
// Print the problem size and number of trials
//
proc printConfiguration() {
  if (printParams) {
    //
    // *** Here we multiply m by the number of locales so that we can
    // *** print out the global problem size.
    //
    printProblemSize(elemType, numVectors, m * numLocales);
    writeln("Number of trials = ", numTrials, "\n");
  }
}

//
// *** Both initVectors and verifyResults are almost identical to
// *** stream.chpl even though they are called with arrays that are
// *** not distributed.  For initialization, the same random stream is
// *** used on each locale.  In the global version, a single logical
// *** stream of random numbers is used across all of the locales.
// ***
// *** In this version, we've omitted a way to print the arrays.  This
// *** ensures determinism of output.  Printing the arrays also
// *** violates the locality constraint imposed by the local block
// *** from which these functions are called.
//
// Initialize vectors B and C using a random stream of values
//
proc initVectors(B, C) {
  var randlist = new RandomStream(seed);

  randlist.fillRandom(B);
  randlist.fillRandom(C);

  delete randlist;
}

//
// Verify that the computation is correct
//
proc verifyResults(A, B, C) {

  //
  // recompute the computation, destructively storing into B to save space
  //
  forall (b, c) in (B, C) do
    b += alpha *c;

  //
  // Compute the infinity-norm by computing the maximum reduction of the
  // absolute value of A's elements minus the new result computed in B.
  // "[i in I]" represents an expression-level loop: "forall i in I"
  //
  const infNorm = max reduce [(a,b) in (A,B)] abs(a - b);

  return (infNorm <= epsilon);    // return whether the error is acceptable
}

//
// Print out success/failure, the timings, and the GB/s value.
//
// *** Here we report maximum, average, and minimum times instead of
// *** total, average, and minimum.
//
proc printResults(successful, minTimes) {
  writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
  if (printStats) {
    const maxTime = max reduce minTimes,
          avgTime = + reduce minTimes / numLocales,
```

```
215           minTime = min reduce minTimes;
216       writeln("Execution time:");
217       writeln("  max (seconds) = ", maxTime);
218       writeln("  avg (seconds) = ", avgTime);
219       writeln("  min (seconds) = ", minTime);
220
221       const maxGBPerSec = numVectors * numBytes(elemType) * (m / minTime) * 1e-9,
222             avgGBPerSec = numVectors * numBytes(elemType) * (m / avgTime) * 1e-9,
223             minGBPerSec = numVectors * numBytes(elemType) * (m / maxTime) * 1e-9;
224       writeln("Performance (GB/s):");
225       writeln("  max = ", maxGBPerSec);
226       writeln("  avg = ", avgGBPerSec);
227       writeln("  min = ", minGBPerSec);
228     }
229   }
```

B Global STREAM Triad

```
//
// Use standard modules for Block distributions, Timing routines, Type
// utility functions, and Random numbers
//
use BlockDist, Time, Types, Random;

//
// Use shared user module for computing HPCC problem sizes
//
use HPCCProblemSize;

//
// The number of vectors and element type of those vectors
//
const numVectors = 3;
type elemType = real(64);

//
// Configuration constants to set the problem size (m) and the scalar
// multiplier, alpha
//
config const m = computeProblemSize(numVectors, elemType),
             alpha = 3.0;

//
// Configuration constants to set the number of trials to run and the
// amount of error to permit in the verification
//
config const numTrials = 10,
             epsilon = 0.0;

//
// Configuration constants to indicate whether or not to use a
// pseudo-random seed (based on the clock) or a fixed seed; and to
// specify the fixed seed explicitly
//
config const useRandomSeed = true,
             seed = if useRandomSeed then SeedGenerator.currentTime else 31415926;

//
// Configuration constants to control what's printed -- benchmark
// parameters, input and output arrays, and/or statistics
//
config const printParams = true,
             printArrays = false,
             printStats = true;

//
// The program entry point
//
proc main() {
  printConfiguration();   // print the problem size, number of trials, etc.

  //
  // ProblemSpace describes the index set for the three vectors.  It
  // is a 1D domain storing 64-bit ints and is distributed according
  // to a Block distribution.  In this case, the Block distribution is 1D
  // distribution computed by blocking the bounding box 1..m across the set
  // of locales.  The ProblemSpace domain contains the indices 1..m.
  //
  const ProblemSpace:
    domain(1, int(64)) dmapped Block(boundingBox=[1..m]) = [1..m];

  //
  // A, B, and C are the three distributed vectors, declared to store
  // a variable of type elemType for each index in ProblemSpace.
  //
  var A, B, C: [ProblemSpace] elemType;

  initVectors(B, C);    // Initialize the input vectors, B and C

  var execTime: [1..numTrials] real;                // an array of timings

  for trial in 1..numTrials {                       // loop over the trials
    const startTime = getCurrentTime();             // capture the start time

    //
    // The main loop: Iterate over the vectors A, B, and C in a
    // parallel, zippered manner storing the elements as a, b, and c.
    // Compute the multiply-add on b and c, storing the result to a.
    //
    forall (a, b, c) in (A, B, C) do
      a = b + alpha * c;

    execTime(trial) = getCurrentTime() - startTime;  // store the elapsed time
  }

  const validAnswer = verifyResults(A, B, C);        // verify...
  printResults(validAnswer, execTime);               // ...and print the results
}

//
// Print the problem size and number of trials
//
proc printConfiguration() {
  if (printParams) {
    if (printStats) then printLocalesTasks();
    printProblemSize(elemType, numVectors, m);
    writeln("Number of trials = ", numTrials, "\n");
  }
}

//
// Initialize vectors B and C using a random stream of values and
// optionally print them to the console
//
proc initVectors(B, C) {
  var randlist = new RandomStream(seed);

  randlist.fillRandom(B);
  randlist.fillRandom(C);

  if (printArrays) {
    writeln("B is: ", B, "\n");
    writeln("C is: ", C, "\n");
  }

  delete randlist;
}

//
// Verify that the computation is correct
//
proc verifyResults(A, B, C) {
  if (printArrays) then writeln("A is:     ", A, "\n");  // optionally print A

  //
  // recompute the computation, destructively storing into B to save space
  //
  forall (b, c) in (B, C) do
    b += alpha *c;

  if (printArrays) then writeln("A-hat is: ", B, "\n");  // and A-hat too

  //
  // Compute the infinity-norm by computing the maximum reduction of the
  // absolute value of A's elements minus the new result computed in B.
  // "[i in I]" represents an expression-level loop: "forall i in I"
  //
  const infNorm = max reduce [(a,b) in (A,B)] abs(a - b);

  return (infNorm <= epsilon);    // return whether the error is acceptable
}

//
// Print out success/failure, the timings, and the GB/s value
//
proc printResults(successful, execTimes) {
  writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
  if (printStats) {
    const totalTime = + reduce execTimes,
          avgTime = totalTime / numTrials,
          minTime = min reduce execTimes;
    writeln("Execution time:");
    writeln("  tot = ", totalTime);
    writeln("  avg = ", avgTime);
    writeln("  min = ", minTime);

    const GBPerSec = numVectors * numBytes(elemType) * (m / minTime) * 1e-9;
    writeln("Performance (GB/s) = ", GBPerSec);
  }
}
```

C Global Random Access

C.1 Benchmark Code

```
//
// Use standard modules for Block distributions and Timing routines
//
use BlockDist, Time;

//
// Use the user modules for computing HPCC problem sizes and for
// defining RA's random stream of values
//
use HPCCProblemSize, RARandomStream;

//
// The number of tables as well as the element and index types of
// that table
//
const numTables = 1;
type elemType = randType,
     indexType = randType;

//
// Configuration constants defining log2(problem size) -- n -- and
// the number of updates -- N_U
//
config const n = computeProblemSize(numTables, elemType,
                                    returnLog2=true, retType=indexType),
             N_U = 2**(n+2);

//
// Constants defining the problem size (m) and a bit mask for table
// indexing
//
const m = 2**n,
      indexMask = m-1;

//
// Configuration constant defining the number of errors to allow (as a
// fraction of the number of updates, N_U)
//
config const errorTolerance = 1e-2;

//
// Configuration constants to control what's printed -- benchmark
// parameters, input and output arrays, and/or statistics
//
config const printParams = true,
             printArrays = false,
             printStats = true;

//
// TableDist is a 1D block distribution for domains storing indices
// of type "indexType", and it is computed by blocking the bounding
// box 0..m-1 across the set of locales.  UpdateDist is a similar
// distribution that is computed by blocking the indices 0..N_U-1
// across the locales.
//
const TableDist = new dmap(new Block(boundingBox=[0..m-1])),
      UpdateDist = new dmap(new Block(boundingBox=[0..N_U-1]));

//
// TableSpace describes the index set for the table.  It is a 1D
// domain storing indices of type indexType, it is distributed
// according to TableDist, and it contains the indices 0..m-1.
// Updates is an index set describing the set of updates to be made.
// It is distributed according to UpdateDist and contains the
// indices 0..N_U-1.
//
const TableSpace: domain(1, indexType) dmapped TableDist = [0..m-1],
      Updates: domain(1, indexType) dmapped UpdateDist = [0..N_U-1];


//
// The program entry point
//
proc main() {
  //
  // T is the distributed table itself, storing a variable of type
  // elemType for each index in TableSpace.
  //
  var T: [TableSpace] elemType;

  printConfiguration();   // print the problem size, number of trials, etc.

  //
  // In parallel, initialize the table such that each position
  // contains its index.  "[i in TableSpace]" is shorthand for "forall
  // i in TableSpace"
  //
  [i in TableSpace] T(i) = i;

  const startTime = getCurrentTime();              // capture the start time

  //
  // The main computation: Iterate over the set of updates and the
  // stream of random values in a parallel, zippered manner, dropping
  // the update index on the ground and storing the random value
  // in r.  Use an on-clause to force the table update to be executed on
  // the locale which owns the table element in question to minimize
  // communications.  Compute the update using r both to compute the
  // index and as the update value.
  //
  forall ( , r) in (Updates, RAStream()) do
    on TableDist.idxToLocale(r & indexMask) do {
      const myR = r;
      local {
        T(myR & indexMask) ^= myR;
      }
   }

  const execTime = getCurrentTime() - startTime;   // capture the elapsed time

  const validAnswer = verifyResults(T);            // verify the updates
  printResults(validAnswer, execTime);             // print the results
}

//
// Print the problem size and number of updates
//
proc printConfiguration() {
  if (printParams) {
    if (printStats) then printLocalesTasks();
    printProblemSize(elemType, numTables, m);
    writeln("Number of updates = ", N_U, "\n");
  }
}

//
// Verify that the computation is correct
//
proc verifyResults(T) {
  //
  // Print the table, if requested
  //
  if (printArrays) then writeln("After updates, T is: ", T, "\n");

  //
  // Reverse the updates by recomputing them, this time using an
  // atomic statement to ensure no conflicting updates
  //
  forall ( , r) in (Updates, RAStream()) do
    on TableDist.idxToLocale(r & indexMask) do
      atomic T(r & indexMask) ^= r;

  //
  // Print the table again after the updates have been reversed
  //
  if (printArrays) then writeln("After verification, T is: ", T, "\n");

  //
  // Compute the number of table positions that weren't reverted
  // correctly.  This is an indication of the number of conflicting
  // updates.
  //
  const numErrors = + reduce [i in TableSpace] (T(i) != i);
  if (printStats) then writeln("Number of errors is: ", numErrors, "\n");

  //
  // Return whether or not the number of errors was within the benchmark's
  // tolerance.
  //
  return numErrors <= (errorTolerance * N_U);
}

//
// Print out success/failure, the execution time, and the GUPS value
//
proc printResults(successful, execTime) {
  writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
  if (printStats) {
    writeln("Execution time = ", execTime);
    writeln("Performance (GUPS) = ", (N_U / execTime) * 1e-9);
  }
}
```

C.2 Supporting Module Code

```
//
// A helper module for the RA benchmark that defines the random stream
// of values
//
module RARandomStream {
  param randWidth = 64;              // the bit-width of the random numbers
  type randType = uint(randWidth);   // the type of the random numbers

  //
  // m2 is a table (tuple) of helper values used to fast-forward
  // through the random stream.
  //
  const m2: randWidth*randType = computeM2Vals();

  //
  // A serial iterator for the random stream that resets the stream
  // to its 0th element and yields values endlessly.
  //
```

```
  iter RAStream() {
    var val = getNthRandom(0);
    while (1) {
      getNextRandom(val);
      yield val;
    }
  }

  //
  // A "follower" iterator for the random stream that takes a range of
  // 0-based indices (follower) and yields the pseudo-random values
  // corresponding to those indices.  Follower iterators like these
  // are required for parallel zippered iteration.
  //
  iter RAStream(param tag: iterKind, followThis) where tag == iterKind.follower {
    if followThis.size != 1 then
      halt("RAStream cannot use multi-dimensional iterator");
    var val = getNthRandom(followThis(1).low);
    for followThis {
      getNextRandom(val);
      yield val;
    }
  }

  //
  // A helper function for "fast-forwarding" the random stream to
  // position n in O(log2(n)) time
  //
  proc getNthRandom(in n: uint(64)) {
    param period = 0x7fffffffffffffff/7;

    n %= period;
    if (n == 0) then return 0x1;
    var ran: randType = 0x2;
    for i in 0..log2(n)-1 by -1 {
      var val: randType = 0;
      for j in 0..#randWidth do
        if ((ran >> j) & 1) then val ^= m2(j+1);
      ran = val;
      if ((n >> i) & 1) then getNextRandom(ran);
    }
    return ran;
  }

  //
  // A helper function for advancing a value from the random stream,
  // x, to the next value
  //
  proc getNextRandom(inout x) {
    param POLY = 0x7;
    param hiRandBit = 0x1:randType << (randWidth-1);

    x = (x << 1) ^ (if (x & hiRandBit) then POLY else 0);
  }

  //
  // A helper function for computing the values of the helper tuple, m2
  //
  proc computeM2Vals() {
    var m2tmp: randWidth*randType;
    var nextVal = 0x1: randType;
    for i in 1..randWidth {
      m2tmp(i) = nextVal;
      getNextRandom(nextVal);
      getNextRandom(nextVal);
    }
    return m2tmp;
  }
}
```

D Global HPL

```
//
// Use standard modules for vector and matrix Norms, Random numbers
// and Timing routines
//
use Norm, Random, Time;

//
// Use the user module for computing HPCC problem sizes
//
use HPCCProblemSize;

//
// Use the distributions we need for this computation
//
use d, r, f;

//
// The number of matrices and the element type of those matrices
// indexType can be int or int(64), elemType can be real or complex
//
const numMatrices = 1;
type indexType = int,
     elemType = real;

//
// Configuration constants indicating the problem size (n) and the
// block size (blkSize)
//
config const n = computeProblemSize(numMatrices, elemType, rank=2,
                                    memFraction=2, retType=indexType),
             blkSize = 8;

//
// Configuration constant used for verification thresholds
//
config const epsilon = 2.0e-15;

//
// Configuration constants to indicate whether or not to use a
// pseudo-random seed (based on the clock) or a fixed seed; and to
// specify the fixed seed explicitly
//
config const useRandomSeed = true,
             seed = if useRandomSeed then SeedGenerator.currentTime else 31415;

//
// Configuration constants to control what's printed -- benchmark
// parameters, input and output arrays, and/or statistics
//
config const printParams = true,
             printArrays = false,
             printStats = true;

// These are solely to make the testing system happy given the COMPOPTS file.
// To be removed once COMPOPTS becomes a non-issue.
config var reproducible = false, verbose = false;

//
// The program entry point, currently module initialization
//
  printConfiguration();

  //
  // Compute targetLocales - required for Dimensional.
  // We hard-code 2 dimensions.
  //
  var targetIds: domain(2);
  var targetLocales: [targetIds] locale;
  setupTargetLocalesArray(targetIds, targetLocales, Locales);

  // Here are the dimensions of our grid of locales.
  const tl1 = targetIds.dim(1).length,
        tl2 = targetIds.dim(2).length;
  if printParams && printStats then
    writeln("target locales ", tl1, " x ", tl2);

  // Create the dimensional descriptors
  const
    bdim1 = new idist(lowIdx=1, blockSize=blkSize, numLocales=tl1),
    rdim1 = new vdist(tl1),

    bdim2 = new idist(lowIdx=1, blockSize=blkSize, numLocales=tl2),
    rdim2 = new vdist(tl2);

  //
  // MatVectSpace is a 2D domain of type indexType that represents the
  // n x n matrix adjacent to the column vector b.  MatrixSpace is a
  // subdomain that is created by slicing into MatVectSpace,
  // inheriting all of its rows and its low column bound.  As our
  // standard distribution library is filled out, MatVectSpace will be
  // distributed using a BlockCyclic(blkSize) distribution.
  //
  // We use 'AbD' instead of 'MatVectSpace' throughout.
  //
  const AbD: domain(2, indexType)
          dmapped DimensionalDist(targetLocales, bdim1, bdim2, "dim")
          = [1..n, 1..n+1],
        MatrixSpace = AbD[.., ..n];

  var Ab : [AbD] elemType,           // the matrix A and vector b
      piv: [1..n] indexType;         // a vector of pivot values

  //
  // Create the 1-d replicated arrays for schurComplement().
  //
  const
    replAD = [1..n, 1..blkSize]
      dmapped DimensionalDist(targetLocales, bdim1, rdim2, "distBR"),
    replBD = [1..blkSize, 1..n+1]
      dmapped DimensionalDist(targetLocales, rdim1, bdim2, "distRB");

  var replA: [replAD] elemType,
      replB: [replBD] elemType;

  initAB();

  const startTime = getCurrentTime();     // capture the start time

  LUFactorize(n, piv);                 // compute the LU factorization

  var x = backwardSub(n);  // perform the back substitution

  const execTime = getCurrentTime() - startTime;  // store the elapsed time

  //
  // Validate the answer and print the results
  const validAnswer = verifyResults(x);
  printResults(validAnswer, execTime);

//
// blocked LU factorization with pivoting for matrix augmented with
// vector of RHS values.
//
proc LUFactorize(n: indexType,
                 piv: [1..n] indexType) {

  // Initialize the pivot vector to represent the initially unpivoted matrix.
  piv = 1..n;

  /* The following diagram illustrates how we partition the matrix.
     Each iteration of the loop increments a variable blk by blkSize;
     point (blk, blk) is the upper-left location of the currently
     unfactored matrix (the dotted region represents the areas
     factored in prior iterations).  The unfactored matrix is
     partioned into four subdomains: tl, tr, bl, and br, and an
     additional domain (not shown), l, that is the union of tl and bl.

               (point blk, blk)
        +-------//------------------+
        |......//..................|
        |.....//...................|
        |....+-----+---------------|
        |....|     |               |
        |....| tl  |      tr       |
        |....|     |               |
        |....+-----+---------------|
        |....|     |               |
        |....|     |               |
        |....| bl  |      br       |
        |....|     |               |
        |....|     |               |
        +----+-----+---------------+
  */
  for blk in 1..n by blkSize {
    if printStats then writeln("processing block ", blk);

    const tl = AbD[blk..#blkSize, blk..#blkSize],
          tr = AbD[blk..#blkSize, blk+blkSize..],
          bl = AbD[blk+blkSize.., blk..#blkSize],
          br = AbD[blk+blkSize.., blk+blkSize..],
          l  = AbD[blk.., blk..#blkSize];

    //
    // Now that we've sliced and diced Ab properly, do the blocked-LU
    // computation:
    //
    panelSolve(l, piv);
    updateBlockRow(tl, tr);

    //
    // update trailing submatrix (if any)
    //
    schurComplement(bl, tr, br);
  }
}

//
// Distributed matrix-multiply for HPL. The idea behind this algorithm is that
// some point the matrix will be partioned as shown in the following diagram:
//
//    [1]----+-----+-----+-----+
//     |     |bbbbb|bbbbb|bbbbb|  Solve for the dotted region by
//     |     |bbbbb|bbbbb|bbbbb|  multiplying the 'a' and 'b' region.
//     |     |bbbbb|bbbbb|bbbbb|  The 'a' region is a block column, the
//     +----[2]----+-----+-----+  'b' region is a block row.
//     |aaaaa|.....|.....|.....|
//     |aaaaa|.....|.....|.....|  The 'a' region was 'bl' in the calling
//     |aaaaa|.....|.....|.....|  function but called AD here.  Similarly,
//     +-----+-----+-----+-----+  'b' was 'tr' in the calling code, but BD
//     |aaaaa|.....|.....|.....|  here.
//     |aaaaa|.....|.....|.....|
//     |aaaaa|.....|.....|.....|
//     +-----+-----+-----+-----+
//
// Every locale with a block of data in the dotted region updates
// itself by multiplying the neighboring a-region block to its left
// with the neighboring b-region block above it and subtracting its
// current data from the result of this multiplication. To ensure that
// all locales have local copies of the data needed to perform this
// multiplication we copy the data A and B data into the replA and
// replB arrays, which will use a dimensional (block-cyclic,
// replicated-block) distribution (or vice-versa) to ensure that every
// locale only stores one copy of each block it requires for all of
// its rows/columns.
```

```
//
proc schurComplement(AD: domain, BD: domain, Rest: domain) {

  // Prevent replication of unequal-sized slices
  if Rest.numIndices == 0 then return;

  //
  // Copy data into replicated arrays so every processor has a local copy
  // of the data it will need to perform a local matrix-multiply.
  //
  coforall dest in targetLocales[targetIds.dim(1).high, targetIds.dim(2)] do
    on dest do
      // replA on tgLocales[d1,i] gets a copy of Ab from tgLocales[d1,..]
      replA = Ab[1..n, AD.dim(2)];

  coforall dest in targetLocales[targetIds.dim(1), targetIds.dim(2).high] do
    on dest do
      // replB on tgLocales[i,d2] gets a copy of Ab from tgLocales[..,d2]
      replB = Ab[BD.dim(1), 1..n+1];

  // do local matrix-multiply on a block-by-block basis
  forall (row,col) in Rest by (blkSize, blkSize) {
    // workaround: localize Rest explicitly
    const RestLcl = Rest;

    local {
      for a in (RestLcl.dim(1))(row..#blkSize) do
        for w in 1..blkSize do
          for b in (RestLcl.dim(2))(col..#blkSize) do
            Ab[a,b] -= replA[a,w] * replB[w,b];
    }
  }
}

//
// do unblocked-LU decomposition within the specified panel, update the
// pivot vector accordingly
//
proc panelSolve(
               panel: domain,
               piv: [] indexType) {

  for k in panel.dim(2) {             // iterate through the columns
    const col = panel[k.., k..k];

    // If there are no rows below the current column return
    if col.numIndices == 0 then return;

    // Find the pivot, the element with the largest absolute value.
    const ( , (pivotRow, )) = maxloc reduce(abs(Ab(col)), col);

    // Capture the pivot value explicitly (note that result of maxloc
    // is absolute value, so it can't be used directly).
    //
    const pivotVal = Ab[pivotRow, k];

    // Swap the current row with the pivot row and update the pivot vector
    // to reflect that
    Ab[k..k, ..] <=> Ab[pivotRow..pivotRow, ..];
    piv[k] <=> piv[pivotRow];

    if (pivotVal == 0) then
      halt("Matrix cannot be factorized");

    // divide all values below and in the same col as the pivot by
    // the pivot value
    Ab[k+1.., k..k] /= pivotVal;

    // update all other values below the pivot
    forall (i,j) in panel[k+1.., k+1..] do
      Ab[i,j] -= Ab[i,k] * Ab[k,j];
  }
}

//
// Update the block row (tr for top-right) portion of the matrix in a
// blocked LU decomposition.  Each step of the LU decomposition will
// solve a block (tl for top-left) portion of a matrix. This function
// solves the rows to the right of the block.
//
proc updateBlockRow(
                   tl: domain,
                   tr: domain) {

  for row in tr.dim(1) {
    const activeRow = tr[row..row, ..],
          prevRows = tr.dim(1).low..row-1;

    forall (i,j) in activeRow do
      for k in prevRows do
        Ab[i, j] -= Ab[i, k] * Ab[k,j];
  }
}


//
// compute the backwards substitution
//
proc backwardSub(n: indexType) {
  const bd = Ab.domain.dim(1);  // or simply 1..n
  var x: [bd] elemType;

  for i in bd by -1 do
    x[i] = (Ab[i,n+1] - (+ reduce [j in i+1..bd.high] (Ab[i,j] * x[j])))
           / Ab[i,i];

  return x;
}


//
// print out the problem size and block size if requested
//
proc printConfiguration() {
  if (printParams) {
    if (printStats) then printLocalesTasks();
    printProblemSize(elemType, numMatrices, n, rank=2);
    writeln("block size = ", blkSize, "\n");
  }
}

//
// construct an n by n+1 matrix filled with random values and scale
// it to be in the range -1.0..1.0
//
proc initAB() {
  fillRandom(Ab, seed);
  Ab = Ab * 2.0 - 1.0;
}

//
// calculate norms and residuals to verify the results
//
proc verifyResults(x) {
  initAB();

  const axmbNorm = norm(gaxpyMinus(Ab[.., 1..n], x, Ab[.., n+1..n+1]), normType.normInf);

  const aInorm   = norm(Ab[.., 1..n], normType.norm1),
        aInfNorm = norm(Ab[.., 1..n], normType.normInf),
        x1Norm   = norm(Ab[.., n+1..n+1], normType.norm1),
        xInfNorm = norm(Ab[.., n+1..n+1], normType.normInf);

  const resid1 = axmbNorm / (epsilon * aInorm * n),
        resid2 = axmbNorm / (epsilon * aInorm * x1Norm),
        resid3 = axmbNorm / (epsilon * aInfNorm * xInfNorm);

  if (printStats) {
    writeln("resid1: ", resid1);
    writeln("resid2: ", resid2);
    writeln("resid3: ", resid3);
  }

  return max(resid1, resid2, resid3) < 16.0;
}

//
// print success/failure, the execution time and the Gflop/s value
//
proc printResults(successful, execTime) {
  writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
  if (printStats) {
    writeln("Execution time = ", execTime);
    const GflopPerSec = ((2.0/3.0) * n**3 + (3.0/2.0) * n**2) / execTime * 10e-9;
    writeln("Performance (Gflop/s) = ", GflopPerSec);
  }
}

//
// simple matrix-vector multiplication, solve equation A*x-y
//
proc gaxpyMinus(A: [],
                x: [?xD],
                y: [?yD]) {
  var res: [1..n] elemType;

  forall i in 1..n do
    res[i] = (+ reduce [j in xD] (A[i,j] * x[j])) - y[i,n+1];

  return res;
}
```

E Shared Problem Size Module Code for HPCC Benchmarks

```
//
// A shared module for computing the appropriate problem size for the
// HPCC benchmarks
//
module HPCCProblemSize {
  //
  // Use the standard modules for reasoning about Memory and Types
  //
  use Memory, Types;

  //
  // The main routine for computing the problem size
  //
  proc computeProblemSize(numArrays: int,    // #arrays in the benchmark
                          type elemType,     // the element type of those arrays
                          rank=1,            // rank of the arrays
                          returnLog2=false,  // whether to return log2(probSize)
                          memFraction=4,     // fraction of mem to use (eg, 1/4)
                          type retType = int(64)): retType { // type to return
    //
    // Compute the total memory available to the benchmark using a sum
    // reduction over the amount of physical memory (in bytes) owned
    // by the set of locales on which we're running.  Then compute the
    // number of bytes we want to use as defined by memFraction and the
    // number that will be required by each index in the problem size.
    //
    const totalMem = + reduce Locales.physicalMemory(unit = MemUnits.Bytes),
          memoryTarget = totalMem / memFraction,
          bytesPerIndex = numArrays * numBytes(elemType);

    //
    // Use these values to compute a base number of indices
    //
    var numIndices = memoryTarget / bytesPerIndex;

    //
    // If the user requested a 2**n problem size, compute appropriate
    // values for numIndices and lgProblemSize
    //
    var lgProblemSize = log2(numIndices);
    if (returnLog2) {
      if rank != 1 then
        halt("computeProblemSize() can't compute 2D 2**n problem sizes yet");
      numIndices = 2**lgProblemSize;
      if (numIndices * bytesPerIndex <= memoryTarget) {
        numIndices *= 2;
        lgProblemSize += 1;
      }
    }

    //
    // Compute the smallest amount of memory that any locale owns
    // using a min reduction and ensure that it is sufficient to hold
    // an even portion of the problem size.
    //
    const smallestMem = min reduce Locales.physicalMemory(unit = MemUnits.Bytes);
    if ((numIndices * bytesPerIndex)/numLocales > smallestMem) then
      halt("System is too heterogeneous: blocked data won't fit into memory");

    //
    // return the problem size as requested by the callee
    //
    if returnLog2 then
      return lgProblemSize: retType;
    else
      select rank {
        when 1 do return numIndices: retType;
        when 2 do return ceil(sqrt(numIndices)): retType;
        otherwise halt("Unexpected rank in computeProblemSize");
      }
  }

  //
  // Print out the machine configuration used to run the job
  //
  proc printLocalesTasks() {
    writeln("Number of Locales = ", numLocales);
    writeln("Tasks per locale = ", dataParTasksPerLocale);
  }

  //
  // Print out the problem size, #bytes per array, and total memory
  // required by the arrays
  //
  proc printProblemSize(type elemType, numArrays, problemSize: ?psType,
                        param rank=1, problemSize2=problemSize) {
    const bytesPerArray = problemSize**rank * numBytes(elemType),
          totalMemInGB = (numArrays * bytesPerArray:real) / (1024**3),
          lgProbSize = log2(problemSize):psType,
          lgProbSize2 = log2(problemSize2):psType;

    write("Problem size = ", problemSize);
    for i in 2..rank do write(" x ", problemSize2);
    if (2**lgProbSize == problemSize && 2**lgProbSize2 == problemSize2) {
      write(" (2**", lgProbSize);
      for i in 2..rank do write(" x 2**", lgProbSize2);
      write(")");
    }
    writeln();
    writeln("Bytes per array = ", bytesPerArray);
    writeln("Total memory required (GB) = ", totalMemInGB);
  }
}
```

F SSCA#2 R-MAT graph creation (Kernel 1)

```
module SSCA2_RMAT_graph_generator

{
  // +=========================================================================+
  // |             RMAT approximate power law graph generator                  |
  // |                                 and                                     |
  // |                           SSCA #2 Kernel 1                              |
  // |                                                                         |
  // |  Generate approximate power law graph                                   |
  // |                                                                         |
  // |  This procedure accepts a generic graph representation which must       |
  // |  provide only the capabilities to:                                      |
  // |    1. add a vertex to a neighbor list for some other vertex             |
  // |    2. assign an integer weight to an entry in an associated weight list |
  // |                                                                         |
  // |  The RMAT procedure implicitly assumes that vertices are integers       |
  // |  in the range [0, 2^SCALE].  So vertices are integers in this           |
  // |  procedure and edges are pairs of integers.                             |
  // |                                                                         |
  // |  This implementation first generates the RMAT graph as a list of        |
  // |  triples, using an array of edges and an associated array of weights.   |
  // |  These are transformed into the Chapel graph representation using       |
  // |  native Chapel syntax for associative and sparse domains and arrays.    |
  // |  That transformation is Kernel 1 of the SSCA #2 benchmark.              |
  // |                                                                         |
  // |  The code uses two auxiliary procedures for clarity and to help the     |
  // |  compiler use more abstract, higher level, code.                        |
  // +=========================================================================+

  use SSCA2_compilation_config_params;

  record directed_vertex_pair {
    var start = 1: int;
    var end   = 1: int;
  }

  proc +(l: directed_vertex_pair, r: directed_vertex_pair)
      return new directed_vertex_pair (l.start + r.start, l.end + r.end);


  // ============================
  // Quadrant selection algorithm
  // ============================

  proc assign_quadrant ( u: real, a: real, b: real, c: real, d : real,
                         bit : int ) : directed_vertex_pair
    {
      // ----------------------------------------------------------------------
      // The heart of the RMAT random graph generator is a procedure that
      // assigns a single bit of the edge starting and ending vertex by
      // assigning the edge with specified probability to one of the four
      // quadrants of a 2 x 2 grid.
      //
      // Chapel is able to promote this conditional-based scalar procedure to
      // array operations where it is not able to promote conditional code
      // directly.
      //
      // Determine randomly in which quadrant of the grid a point lies,
      // based on the following picture:
      //
      //     +---------------------------+
      //     | u < a         | u < a+b   |
      //     |-------------+-------------|
      //     | u < a + b + c | otherwise |
      //     +---------------------------+
      //
      // The probability of the edge falling in the upper left quadrant is  a,
      // in the upper right quadrant, b, the lower left quadrant  c
      // and the lower right quadrant  d, where the probabilities are
      // normalized to sum to one.
      // ----------------------------------------------------------------------

      var start_inc = 0;
      var end_inc   = 0;
      var edge : directed_vertex_pair;

      if u <= a then
        {}
      else if u <= a + b then
        { end_inc = 1;}
      else if u <= a + b + c then
        { start_inc = 1;}
      else
        { start_inc = 1; end_inc = 1;};

      edge.start = bit * start_inc;
      edge.end   = bit * end_inc;
      return ( edge );
    }


  // ====================================
  // Main RMAT Graph Generation Procedure
  // ====================================

  proc Gen_RMAT_graph ( a : real,
                        b : real,
                        c : real,
                        d : real,
                        SCALE :int,
                        N_VERTICES : int,
                        n_raw_edges : int,
                        MAX_EDGE_WEIGHT :int,
                        G )

    { use Random;

      const vertex_range = 1..N_VERTICES,
            edge_range   = 1..n_raw_edges,
            rand_range   = 1..n_raw_edges + 1;

      // Random Numbers return in the range [0.0, 1.0)

      var Rand_Gen = if REPRODUCIBLE_PROBLEMS then
                       new RandomStream (seed = 0556707007)
                     else
                       new RandomStream ();

      var   Noisy_a     : [edge_range] real,
            Noisy_b     : [edge_range] real,
            Noisy_c     : [edge_range] real,
            Noisy_d     : [edge_range] real,
            norm        : [edge_range] real,
            Unif_Random : [edge_range] real,
            Edges       = [edge_range] new directed_vertex_pair ();

       // ------------------------------------------------------------
      // The RMAT algorithm is based on recursively sub-dividing a grid.
      // In this case, the grid is square of order 2^SCALE by 2^SCALE.
      // So exactly "SCALE" steps of recursion will be required and the
      // recursion can be implemented directly by iteration.
      // ------------------------------------------------------------

      // ------------------------------------------------------------
      // Note on random number generators -- the RMAT generator creates
      // 5*SCALE vectors of length 2**SCALE.  The dependence on powers
      // of two provides an opportunity to expose statistical correlations
      // in the pseudo-random numbers.  This certainly occurs with the
      // current Chapel Random module.  The "skips" in the random number
      // sequence dramatically change the results.  Without them, the
      // Chapel RMAT matrices are inconsistent with what is seen in
      // other implementations.
      // ------------------------------------------------------------

      writeln ("Random graph generated by stride of 1 in one random stream",
               " with skips" );

      var bit = 1 << SCALE;
      var   skip : real;

      for depth in 1..SCALE do {
        bit >>= 1;

        // randomize the coefficients, tweaking them by numbers in [-.05, .05)

        skip = Rand_Gen.getNext ();
        Rand_Gen.fillRandom (Unif_Random);
        Noisy_a = a * (0.95 + 0.1 * Unif_Random);

        skip = Rand_Gen.getNext ();
        Rand_Gen.fillRandom (Unif_Random);
        Noisy_b = b * (0.95 + 0.1 * Unif_Random);

        skip = Rand_Gen.getNext ();
        Rand_Gen.fillRandom (Unif_Random);
        Noisy_c = c * (0.95 + 0.1 * Unif_Random);

        skip = Rand_Gen.getNext ();
        Rand_Gen.fillRandom (Unif_Random);
        Noisy_d = d * (0.95 + 0.1 * Unif_Random);

        norm     = 1.0 / (Noisy_a + Noisy_b + Noisy_c + Noisy_d);
        Noisy_a *= norm;
        Noisy_b *= norm;
        Noisy_c *= norm;
        Noisy_d *= norm;

        skip = Rand_Gen.getNext ();
        Rand_Gen.fillRandom (Unif_Random);

        Edges += assign_quadrant ( Unif_Random, Noisy_a, Noisy_b,
                                   Noisy_c, Noisy_d, bit );

      };

      // ------------------------------------------------------------------
      // Assign weights to edges randomly, then randomly relabel the vertices
      // to avoid locality from the obvious imbalance that will arise when
      // one of the coefficients is clearly larger than the others
      // ------------------------------------------------------------------

      var permutation : [vertex_range] int = vertex_range;
      var Edge_Weight : [edge_range] int;

      Rand_Gen.fillRandom ( Unif_Random  );
      Edge_Weight = floor (1 + Unif_Random * MAX_EDGE_WEIGHT) : int;

      Rand_Gen.fillRandom ( Unif_Random (vertex_range) );

      for v in vertex_range do
        { var new_id : int;
          new_id = floor (1 + Unif_Random (v) * N_VERTICES) : int;
          permutation (v) <=> permutation (new_id);
        };

      var node_count : [vertex_range] int = 0;

      //      for e in edge_range do {
      //        Edges(e).start = permutation (Edges(e).start);
      //        Edges(e).end   = permutation (Edges(e).end  );
      //      }
      Edges.start = permutation (Edges.start);
      Edges.end   = permutation (Edges.end  );
```

```
215       if DEBUG_GRAPH_GENERATOR || DEBUG_WEIGHT_GENERATOR then {
216         writeln ();
217         for e in edge_range do
218           writeln ("edge (", e, ") : (", Edges(e), ", ",
219                    Edge_Weight (e), ")" );
220       }

222       // -------------------------
223       // Graph consistency checking
224       // -------------------------

226       assert ( Edges.start > 0,
227                "edge start vertices out of low end of range");

229       assert ( Edges.end > 0,
230                "edge end vertices out of low end of range");

232       assert ( Edges.start <= 2**SCALE,
233                "edge start vertices out of high end of range");

235       assert ( Edges.end <= 2**SCALE,
236                "edge end vertices out of high end of range");

238       assert ( Edge_Weight > 0,
239                "edge weightd out of low end of range");

241       assert ( Edge_Weight <= MAX_EDGE_WEIGHT,
242                "edge weightd out of high end of range");

244       writeln (); writeln ("Vertex Set in G:", G.vertices);

246       // -----------------------------------------------------------
247       // Kernel 1: assemble graph from list of triples.  Include
248       // only non-self incident edges.  in case of duplicates, last
249       // in wins (+= instead of = works to take sum of weights)
250       // -----------------------------------------------------------

252       var collisions = 0, self_edges = 0;
253       for e in edge_range do {
254         var u = Edges (e).start;
255         var v = Edges (e).end  ;

257         if ( v != u ) then {
258           if G.Neighbors (u).member(v) then {
259             collisions += 1;
260           }
261           else {
262             G.Neighbors (u).add (v);
263             G.Row(u).Weight (v) = Edge_Weight (e);
264           }
265         }
266         else
267           self_edges += 1;
268       }

270       writeln ( "# of raw edges generated  ", n_raw_edges );
271       writeln ( "# of duplicate edges      ", collisions );
272       writeln ( "# of self edges           ", self_edges );
273       writeln ( "# of edges in final graph ",
274                 + reduce [v in G.vertices] G.n_Neighbors (v) );

276       if DEBUG_GRAPH_GENERATOR || DEBUG_WEIGHT_GENERATOR then {
277         writeln ();
278         writeln ("tuples denote (edge, weight)");
279         writeln ();
280         for u in G.vertices do {
281           write ( "row ", u, ": [", G.n_Neighbors (u), "] " );
282           for (v,w) in (G .Neighbors (u), G.edge_weight (u) ) do
283             write ( (v, w) );
284           writeln ();
285         }
286       }

288       var max_edges = max reduce [v in vertex_range] G.n_Neighbors (v);

290       var edge_count : [0..max_edges] int = 0;

292       for v in G.vertices do
293         edge_count (G.n_Neighbors (v)) += 1;

295       writeln ("histograph of node distributions by number of outgoing edges");
296       writeln ( "# edges  # nodes");
297       for count in 0..max_edges do
298         writeln ( count, "  ", edge_count (count) );

300       writeln ();

302     }
303   }
```

G SSCA#2 Kernels 2–4

```
module SSCA2_kernels

//  +=========================================================================+
//  |  Polymorphic Implementation of SSCA #2, Kernels 2-4                     |
//  |                                                                         |
//  |  Each kernel takes a graph argument which provides for each vertex      |
//  |  1.  an iterator for its set of neighbors                               |
//  |  2.  a parallel integer array of edge weights, which can be zipper      |
//  |      iterated with the set of neighbors                                 |
//  |  3.  the number of neighbors                                            |
//  |                                                                         |
//  |  These are the only requirements on the representation of the graph.    |
//  |                                                                         |
//  |  Filtering in Kernel 4 is turned on or off by a compilation time param. |
//  +=========================================================================+

{
  use SSCA2_compilation_config_params, Time;

  var stopwatch : Timer;

  // ========================================================
  //                           KERNEL 2:
  // ========================================================
  // Find the edges with the largest edges.  Return a list of
  // edges, all of which have the largest weight.
  // ========================================================

  proc largest_edges ( G, heavy_edge_list :domain )

    // edge_weights can be either an array over an associative
    // domain or over a sparse domain.  the output  heavy_edge_list
    // can either kind of domain or something else purpose-built
    // for this task.
    {
      if PRINT_TIMING_STATISTICS then stopwatch.start ();
      var heaviest_edge_weight$ : sync int = 0;

      // --------------------------------------------------------
      // find heaviest edge weight in a single pass over all edges
      // --------------------------------------------------------

      //       heaviest_edge_weight = max reduce [ s in G.vertices ]
      //                                         [ w in G.edge_weight (s) ] w;

      forall s in G.vertices do
        forall w in G.edge_weight (s) do
          heaviest_edge_weight$ = max ( w, heaviest_edge_weight$ );

      // ------------------------------------------------
      // in a second pass over all edges, extract list
      // of all edges  matching the heaviest weight
      // ------------------------------------------------

      forall s in G.vertices do
        forall (t, w) in ( G.Neighbors (s), G.edge_weight (s) ) do

          // should be forall, requires a custom parallel iterator in the
          // random graph case and zippering for associative domains may
          // also present a problem

          if w == heaviest_edge_weight$.readXX () then {
            heavy_edge_list.add ( (s,t) );
          };

      if PRINT_TIMING_STATISTICS then {
        stopwatch.stop ();
        writeln ( "Elapsed time for Kernel 2: ", stopwatch.elapsed (),
                  " seconds");
        stopwatch.clear ();
      }

      // ------------------------------------------------
      // should be able to write a user-defined "maxlocs"
      // reduction more efficiently than this scheme
      // ------------------------------------------------

      if DEBUG_KERNEL2 then {
        writeln ();
        writeln ( "Heaviest weight      : ",
                  heaviest_edge_weight$ . readFF ());
        writeln ( "Number of heavy edges:", heavy_edge_list.numIndices );
        writeln ();
        writeln ( "Edges with largest weight and other neighbors:" );
        for (s,t) in heavy_edge_list do {
          writeln ("edge   ", (s,t));
          for (v,w) in (G.Neighbors (s), G.edge_weight (s) ) do
            writeln ("      ", v, " ", w);}
      }
    };



  // ========================================================================
  //                              KERNEL 3:
  // ========================================================================
  // For each root (heavy) edge, find the subgraph (vertices and edges)
  // defined by directed paths of length no greater than max_path_length
  // in which the first edge traversed is the root edge
  // ========================================================================

  proc rooted_heavy_subgraphs ( G,
                                Heavy_Edge_List     : domain,
                                Heavy_Edge_Subgraph : [],
                                in max_path_length  : int )

    // ------------------------------------------------------------------------
    // there is a classic space versus time tradeoff.  if the subgraphs expanded
    // by breadth first search are small, it would make sense to use a hash
    // table or an associative domain to represent the search.  If the subgraphs
    // are large, using a full length vector to represent the search is more
    // appropriate.  We expect small diameters for power law graphs, so we
    // expect large subgraphs.
    // ------------------------------------------------------------------------

    {
      if PRINT_TIMING_STATISTICS then stopwatch.start ();

      const vertex_domain = G.vertices;

      forall ( x, y ) in Heavy_Edge_List do {
        var Active_Level, Next_Level : domain ( index (vertex_domain) );
        var min_distance$            : [vertex_domain] sync int = -1;

        if DEBUG_KERNEL3 then
          writeln ( " Building heavy edge subgraph from pair:", (x,y) );
        Active_Level.add ( y );
        Next_Level.clear ();
        Heavy_Edge_Subgraph ( (x, y) ).nodes.clear ();
        Heavy_Edge_Subgraph ( (x, y) ).edges.clear ();
        min_distance$ ( y ).writeFF (0);

        Heavy_Edge_Subgraph ( (x, y) ).edges.add ( (x, y) );
        Heavy_Edge_Subgraph ( (x, y) ).nodes.add ( x );
        Heavy_Edge_Subgraph ( (x, y) ).nodes.add ( y );

        for path_length in 1 .. max_path_length do {

          forall v in Active_Level do {

            forall w in G.Neighbors (v) do { // eventually, will be forall

              if min_distance$ (w).readXX () < 0 then {

                if min_distance$ (w).readFE () < 0 then {
                  Next_Level.add (w);
                  Heavy_Edge_Subgraph ( (x, y) ).nodes.add (w);
                  min_distance$ (w).writeEF (path_length);
                }
                else
                  min_distance$ (w).writeEF (path_length);
              }

              // min_distance$ must have been set by some thread by now

              if min_distance$ (w).readFF () == path_length then {
                Heavy_Edge_Subgraph ( (x, y) ).edges.add ( (v, w) );
              }
            }
          }

          if path_length < max_path_length then {
            Active_Level = Next_Level;
            Next_Level.clear ();
          }
        }
      }

      if PRINT_TIMING_STATISTICS then {
        stopwatch.stop ();
        writeln ( "Elapsed time for Kernel 3: ", stopwatch.elapsed (),
                  " seconds");
        stopwatch.clear ();
      }
    } // end of rooted_heavy_subgraphs


  // ========================================================
  // generic class structure must be defined outside of
  // generic procedure.  used by Betweenness Centrality kernel 4.
  // --------------------------------------------------------
  // The set of vertices at a particular distance from s form a
  // level set.  The class allows the full set of vertices to be
  // partitioned into a linked list of level sets.  Each instance
  // of the outer loop in kernel 4 creates such a partitioning.
  // ========================================================

  class Level_Set {
    type Sparse_Vertex_List;
    var Members  : Sparse_Vertex_List;
    var previous : Level_Set (Sparse_Vertex_List);
  }

  // sungeun: 8/2011
  // Added replicated level sets
  //
  // Each locale will have its own level sets.  A locale's level set
  // will only contain nodes that are physically allocated on that
  // particular locale.  We implement this using the replicated
  // distribution.
  //
  use ReplicatedDist;

  // ================================================================
  //                                KERNEL 4
  // ================================================================
  // Calculate Betweenness Centrality for simple unweighted directed or
  // undirected graphs, using Madduri, et.al.'s modification of
  // Brandes's 2001 algorithm
  // ================================================================

  proc approximate_betweenness_centrality ( G, starting_vertices,
                                            Between_Cent : [] real,
                                            out Sum_Min_Dist : real )
```

```
// ------------------------------------------------------------------------
// The betweenness centrality metric for a given node  v  is defined
// as the double sum over s not equal to v and  t not equal to
// either s or v of the ratio of the number of shortest paths from s to t
// passing through v to the number of shortest paths from s to t.
//
// Brandes's algorithm decomposes the computation of this metric into,
// first, separate sums for each vertex s, which can be computed
// independently in parallel, and
// two, a recursive, tree-based, calculation of the path counts for
// any particular s.
// The complexity of this algorithm is O ( |V||E| ) time for an unweighted
// graph.  The algorithm requires O ( |V| ) temporary space for each
// process that executes instances of the outermost loop.
// ------------------------------------------------------------------------
{
  const vertex_domain = G.vertices;

  // Had to change declaration below
  //    type Sparse_Vertex_List = sparse subdomain ( G.vertices );
  // to accommodate block distribution of G.vertices

  type Sparse_Vertex_List = domain(index(vertex_domain));

  var Between_Cent$ : [vertex_domain] sync real = 0.0;
  var Sum_Min_Dist$ : sync real = 0.0;

  // ------------------------------------------------------
  // Each iteration of the outer loop of Brandes's algorithm
  // computes the contribution (the "dependency" metric) for
  // one particular vertex  (s)  independently.
  // ------------------------------------------------------

  if PRINT_TIMING_STATISTICS then stopwatch.start ();

  forall s in starting_vertices do {

    if DEBUG_KERNEL4 then writeln ( "expanding from starting node ", s );

    // sungeun: 8/2011
    // Privatization of the following distributed arrays may
    // be of concern.

    // ------------------------------------------------------
    // all locally declared variables become private data
    // for each instance of the parallel for loop
    // ------------------------------------------------------

    var min_distance$  : [vertex_domain] sync int       = -1;
    var path_count$    : [vertex_domain] sync real (64) = 0.0;
    var depend         : [vertex_domain] real           = 0.0;
    var Lcl_Sum_Min_Dist: sync real                     = 0.0;

    // The structure of the algorithm depends on a breadth-first
    // traversal. Each vertex will be marked by the length of
    // the shortest path (min_distance$) from s to it. The array
    // path_count$ will hold a count of the number of shortest
    // paths from s to this node.  The number of paths in moderate
    // sized tori exceeds 2**64.

    // Used to check termination of the forward pass
    //
    // sungeun: 8/2011
    // Can possibly use a replicated bool with rcCollect(), but
    // not sure of the performance implications for large numbers
    // of locales.
    var Active_Remaining: [LocaleSpace] bool = true;
    var remaining = true;

    // Replicated level sets
    var Active_Level    : [rcDomain] Level_Set (Sparse_Vertex_List);
    var Next_Level      : [rcDomain] Level_Set (Sparse_Vertex_List);
    coforall loc in Locales do on loc {
      rcLocal(Active_Level) = new Level_Set (Sparse_Vertex_List);
      rcLocal(Active_Level).previous = nil;
      rcLocal(Next_Level) = new Level_Set (Sparse_Vertex_List);
      rcLocal(Next_Level).previous = rcLocal(Active_Level);
    }

    var current_distance : int = 0;

    // establish the initial level sets for the
    // breadth-first traversal from s

    on s {
      rcLocal(Active_Level).Members.add ( s );
      rcLocal(Next_Level).Members.clear ();
      min_distance$ (s) . writeFF (0);
      path_count$ (s) . writeFF (1);
    }

    while remaining do {

        // ------------------------------------------------
        // expand the neighbor sets for all vertices at the
        // current distance from the starting vertex  s
        // ------------------------------------------------

        current_distance += 1;

        // sungeun: 8/2011
        // basic single use barrier
        var count: sync int = numLocales;
        var barrier: single bool;

        // sungeun: 8/2011
        // Copy this value to a constant to enable remote value
        // forwarding optimization.
        const current_distance_c = current_distance;
        coforall loc in Locales do on loc {
         forall u in rcLocal(Active_Level).Members do { // sparse
          forall (v, w) in ( G.Neighbors (u), G.edge_weight (u) ) do on v {

            // should be forall, requires a custom parallel iterator in the
            // random graph case and zippering for associative domains

            // ------------------------------------------
            // add any unmarked neighbors to the next level
            // ------------------------------------------

            if  ( FILTERING &&  w % 8 != 0 ) || !FILTERING  then
              if  min_distance$ (v) . readXX () < 0  then
                {
                  if  min_distance$ (v) . readFE () < 0  then
                    {
                      min_distance$ (v).writeEF (current_distance_c);
                      rcLocal(Next_Level).Members.add (v);
                      if VALIDATE_BC then
                        Lcl_Sum_Min_Dist += current_distance_c;
                    }
                  else
                    // could min_distance$(v) be < current_distance?
                    min_distance$ (v) . writeEF (current_distance_c);
                }

            // ------------------------------------------------
            // only neighbors of  u  that are in the next level
            // are on shortest paths from s through v.  Some
            // thread will have set  min_distance$ (v) by the
            // time this code is reached, whether  v  lies in
            // the previous, the current or the next level.
            // ------------------------------------------------

            if  min_distance$ (v).readFF () == current_distance_c
              then
                path_count$ (v) += path_count$ (u).readFF ();
            }
          };

          // sungeun: 8/2011
          // This (split-phase) barrier is needed to insure all updates
          // to Next_Level are completed before creating the next
          // Next_Level.
          var myc = count;
          if myc!=1 {
            count = myc-1; // release the lock
            // do some work while we wait
            rcLocal(Active_Level) = rcLocal(Next_Level);

            barrier;       // wait for everyone
          } else {         // last one here
            barrier=true;  // release everyone first
            rcLocal(Active_Level) = rcLocal(Next_Level);
          }

          rcLocal(Next_Level)   = new Level_Set (Sparse_Vertex_List);
          rcLocal(Next_Level).previous = rcLocal(Active_Level);
          Active_Remaining[here.id] =
            rcLocal(Active_Level).Members.numIndices:bool;

        }

        remaining = || reduce Active_Remaining;

      };  // end forward pass

      if VALIDATE_BC then
        Sum_Min_Dist$ += Lcl_Sum_Min_Dist;

      // ------------------------------------------------------------
      // compute the dependencies recursively, traversing the vertices
      // of the graph in non-increasing order of distance (reverse
      // ordering from the initial traversal)
      // ------------------------------------------------------------

      var graph_diameter = current_distance - 1;

      if DEBUG_KERNEL4 then
        writeln ( " graph diameter from starting node ", s,
                  "  is ", graph_diameter );

      // sungeun: 8/2011
      // basic single use barrier
      var count: sync int = numLocales;
      // to simplify synchronization between multiple barriers
      var barrier: [2..graph_diameter] single bool;

      coforall loc in Locales do on loc {
        delete rcLocal(Next_Level);                  // it's empty
        rcLocal(Next_Level)   = rcLocal(Active_Level).previous;  // back up to last level
        delete rcLocal(Active_Level);
        rcLocal(Active_Level) = rcLocal(Next_Level);

        for current_distance in 2 .. graph_diameter by -1 do {

          rcLocal(Next_Level)   = rcLocal(Active_Level).previous;
          delete rcLocal(Active_Level);
          rcLocal(Active_Level) = rcLocal(Next_Level);

          // inner reduction should parallelize eventually; compiler
          // serializes it today (and warns us that it did)

          forall u in rcLocal(Active_Level).Members do
            {
              depend (u) = + reduce
                [ (v, w)  in ( G.Neighbors (u), G.edge_weight (u) ) ]
                if ( min_distance$ (v).readFF () == current_distance ) &&
                ( ( FILTERING && w % 8 != 0 || !FILTERING ) )
                then
                  ( path_count$ (u) . readFF () /
                    path_count$ (v) . readFF () )      *
                    ( 1.0 + depend (v) );
```

```
439                 // do not need conditional u != s
441                 Between_Cent$ (u) += depend (u);
442               }
443             // sungeun: 8/2011
444             // This barrier is needed to insure all updates to depend are
445             // complete before the next pass.
446             var myc = count;
447             if myc==1 {
448               count = numLocales;
449               barrier[current_distance] = true;
450             } else {
451               count = myc-1;
452               barrier[current_distance];
453             }
454           };
455           delete rcLocal(Active_Level);
456         }
458       }; // closure of outer embarassingly parallel forall
461       if PRINT_TIMING_STATISTICS then {
462         stopwatch.stop ();
463         var K4_time = stopwatch.elapsed ();
464         stopwatch.clear ();
465         writeln ( "Elapsed time for Kernel 4: ", K4_time, " seconds");
467         var n0            = + reduce [v in vertex_domain] (G.n_Neighbors (v)== 0);
468         var n_edges       = + reduce [v in vertex_domain] G.n_Neighbors (v);
469         var N_VERTICES    = vertex_domain.numIndices;
470         var TEPS          = 7.0 * N_VERTICES * (N_VERTICES - n0) / K4_time;
471         var Adjusted_TEPS = n_edges * (N_VERTICES - n0) / K4_time;
473         writeln ( "                    TEPS: ", TEPS );
474         writeln ( " edge count adjusted TEPS: ", Adjusted_TEPS );
475       }
477       if VALIDATE_BC then
478         Sum_Min_Dist = Sum_Min_Dist$;
480       Between_Cent = Between_Cent$;
482     } // end of Brandes' betweenness centrality calculation
484 }
```

H SSCA#2 R-MAT graph support code

```
module analyze_RMAT_graph_associative_array {

  // +=====================================================================+
  // |  Define associative array-based representations for general sparse  |
  // |  graphs. Provide execution template to generate a random RMAT graph |
  // |  of a specified size and execute and verify SSCA2 kernels 2 through 4. |
  // +=====================================================================+

  proc  generate_and_analyze_associative_array_RMAT_graph_representation {

    // ------------------------------------------------------------------
    // compute a random power law graph with 2^SCALE vertices, using
    // the RMAT generator. Initially generate a list of triples.
    // Then convert it to a Chapel representation of a sparse graph,
    // timing this step (Kernel 1).  Finally, execute Kernels 2, 3 and 4
    // of SSCA #2, using identically the same code as in the various
    // torus cases.
    // ------------------------------------------------------------------

    use SSCA2_compilation_config_params, SSCA2_execution_config_consts;

    use SSCA2_driver, SSCA2_RMAT_graph_generator;

    use BlockDist;

    var n_raw_edges = 8 * N_VERTICES;

    assert ( SCALE > 1, "SCALE must be greater than 1");

    select SCALE {
      when 2 do n_raw_edges = N_VERTICES / 2;
      when 3 do n_raw_edges = N_VERTICES;
      when 4 do n_raw_edges = 2 * N_VERTICES;
      when 5 do n_raw_edges = 4 * N_VERTICES;
      }

    writeln ('-------------------------------------');
    writeln ('Order of RMAT generated graph:', N_VERTICES);
    writeln ('          number of raw edges:', n_raw_edges);
    writeln ('-------------------------------------');
    writeln ();

    // ------------------------------------------------------------------------
    // The data structures below are chosen to implement an irregular (sparse)
    // graph using associative domains and arrays.
    // Each node in the graph has a list of neighbors and a corresponding list
    // of (integer) weights for the implicit edges.
    // The list of neighbors is really just a set; the only properties we need
    // are that we be able to build it (add vertices to it) and that we be
    // able to iterate over it.  Those properties are satisfied by Chapel's
    // associative domains, so each neighbor set is represented by an
    // associative domain.  The weights are an integer array over the
    // neighbor domain.
    //
    // We would have liked to have defined the global set of neighbors as
    // an array of associative domains, but that is not supported in Chapel.
    // Consequently we build an array of records, where each record provides
    // the neighbor set and the weights for a particular node.  The name
    // "row_struct" anticipates the planned use of sparse matrices for this same
    // kind of graph structure.
    // ------------------------------------------------------------------------

    const vertex_domain =
      if DISTRIBUTION_TYPE == "BLOCK" then
        [1..N_VERTICES] dmapped Block ( [1..N_VERTICES] )
      else
        [1..N_VERTICES] ;

    record row_struct {
      type vertex;
      var Row_Neighbors : domain (vertex);
      var Weight        : [Row_Neighbors] int;
    }

    class Associative_Graph {
      const vertices;
      var   Row      : [vertices] row_struct (index (vertices));

      proc   Neighbors  ( v : index (vertices) ) {return Row (v).Row_Neighbors;}

      iter   edge_weight (v : index (vertices) ) var {
        for w in Row (v).Weight do
          yield w;}  // var iterator to avoid a copy

      // Simply forward the domain's parallel iterator
      // FYI: no fast follower opt
      iter   edge_weight(v : index (vertices), param tag: iterKind)
      where tag == iterKind.leader {
        for block in Row(v).Weight._value.these(tag) do
          yield block;
      }

      iter   edge_weight(v : index (vertices), param tag: iterKind, followThis)
      where tag == iterKind.follower {
        for elem in Row(v).Weight._value.these(tag, followThis) do
          yield elem;
      }

      proc   n_Neighbors (v : index (vertices) )
      {return Row (v).Row_Neighbors.numIndices;}
    }

    var G = new Associative_Graph (vertex_domain);

    // ------------------------------------------------------------------
    // generate RMAT graph of the specified size, based on input config
    // values for quadrant assignment.
    // ------------------------------------------------------------------

    Gen_RMAT_graph ( RMAT_a, RMAT_b, RMAT_c, RMAT_d,
                     SCALE, N_VERTICES, n_raw_edges, MAX_EDGE_WEIGHT, G );

    execute_SSCA2 ( G );
    writeln (); writeln ();
    delete G;
  }
}
```