DIS STRESSMARK SUITE

SPECIFICATIONS FOR THE STRESSMARKS OF THE DIS BENCHMARK PROJECT

VERSION 1.0
AUGUST 24, 2000

ATLANTIC AEROSPACE DIVISION, TITAN SYSTEMS CORPORATION

1. Introduction _________________ 2
1.1 Purpose ______________________ 2
1.2 Background _________________ 3
1.3 Scope _______________________ 4

2. Procedures __________________ 5
2.1 Which stressmarks? ____________ 5
2.2 Code organization _____________ 5
2.3 Code optimization _____________ 5
2.4 Data Sets ____________________ 6
2.5 Acceptance Tests ______________ 6
2.6 Metrics ______________________ 6
2.7 Submission of Results __________ 8

3. Pointer Stressmark __________ 10
3.1 Input _______________________ 10
3.2 Algorithm ___________________ 11
3.3 Output ______________________ 11
3.4 Sample Code ________________ 11
3.5 Notes _______________________ 14

4. Update Stressmark __________ 16
4.1 Input _______________________ 16
4.2 Algorithm ___________________ 16
4.3 Output ______________________ 17
4.4 Sample Code ________________ 17
4.5 Notes _______________________ 20

5. Matrix Stressmark __________ 21
5.1 Input _______________________ 21
5.2 Algorithmic Specification _____ 21
5.3 Output ______________________ 23
5.4 Pseudo-code _________________ 24
5.5 Notes _______________________ 25

6. Neighborhood Stressmark ____ 26
6.1 Input _______________________ 26
6.2 Algorithmic Specification ______ 27
6.3 Output ______________________ 29
6.4 Sample Code _________________ 30
6.5 Notes _______________________ 38

7. Field Stressmark _____________ 39
7.1 Input _______________________ 39
7.2 Algorithm ___________________ 39
7.3 Output ______________________ 40
7.4 Sample Code _________________ 41
7.5 Notes _______________________ 43

8. Corner-Turn Stressmark _____ 45
8.1 Input _______________________ 45
8.2 Algorithm ___________________ 45
8.3 Notes _______________________ 46

9. Transitive Closure Stressmark_ 47
9.1 Input _______________________ 47
9.2 Algorithm ___________________ 47
9.3 Output ______________________ 48
9.4 Sample Code _________________ 48
9.5 Notes _______________________ 50

10. Support Functions ___________ 51
10.1 Random Number Generators _51

11. References__________________ 54

1. Introduction

Following the release of the Data-Intensive Systems (DIS) Benchmark Suite [1], it became apparent that additional benchmarks would be required to assist in satisfaction of the needs of some program participants. A set of smaller, more specific procedures were desired. These stressmarks would more directly illustrate particular elements of the DIS problem, and require less energy to implement, perhaps at the expense of reduced realism in certain areas. This document supplies the response to that need by defining the DIS Stressmarks, which are provided as part of the DIS Benchmark Project.

1.1 Purpose

The development of new architectures and approaches for data-intensive computing could be beneficial to many problems of interest to DARPA. Evaluation of the approaches in the context of those problems is essential in order to realize those benefits.

Equally important, the existence of simplified—but meaningful—programs derived from defense applications can provide valuable input to the development process.

On these premises were the benchmarks developed. While the purpose of the Benchmark Suite included the need to realistically represent certain applications, this sometimes lead to difficulty in their implementation. Though the benchmarks represented the essence of the data-intensive processing stripped from the source application, they still required a significant amount of code in order to be complete. The target for the benchmarks was about 1000 lines of code, derived from applications with hundreds of thousands or millions of lines. The target for these stressmarks is on the order of dozens of lines.

To reduce the code without eliminating the data-intensive nature of the problem required a focus on elementary segments of the problem. This implies that results should be worse for general-purpose machines, and architectures developed for the “data-intensive” problem should show an even greater improvement. These results, however, must necessarily be less indicative of normal machine operations, and should be interpreted accordingly.

The benchmarks were aimed specifically toward focussed code that retained the context of the enveloping application. This approach attempted to avoid the pitfalls associated with ‘kernel’-oriented benchmarks, which often indicate performance figures that are unattainable in typical operation.

Stressmarks are necessarily less realistic than benchmarks. Benchmarks are necessarily less realistic than true applications. Participants are cautioned not to allow concentration on stressmarks to interfere with development of machines and software that performs well on real applications.

The critical point for the development of the stressmarks, therefore, is to support the benchmarks, rather than to replace them.

1.2 Background

Data-intensive applications are generally characterized by large data sets, lots of data movement between processor and memory, and irregular memory access points. The stressmarks described here are designed to exhibit these characteristics.

1.2.1 Basic Requirements

One of the major goals of the benchmark suite was to retain a certain degree of application-level orientation, since some elements of data-intensive problems cannot be reliably measured using isolated kernels. The motivation behind the stressmarks, therefore, is to supply problems that are derivative of those offered in the benchmarks, but in the form of isolated kernels. Consequently, the original considerations for stressmarks were:

- Operations should be essentially representative of the data-intensive kernels within the benchmarks.
- Input and output should be limited, and only occur at the very beginning and end of processing tasks.
- Total memory required should be varied, much as the "problem size" was varied for the benchmarks.
- Acceptance and verification tests become difficult to specify, since I/O is limited.
- A random number generator and initial seeds should be used to generate input.
- Ideally, stressmarks can be run in a variety of modes corresponding to data types and precisions.
- If possible, parallel (MPI or threaded) versions could be supplied.
- Total size of source code should be extremely small.
- Represent operations on sparse, dense, regular, and irregular data. Attempt to allow parallelism coarse, fine, or both levels.

1.2.2 Development

Considering the above requirements, a series of stressmarks were proposed, and development began. During this period, it was observed that the bulk of the operations did not modify their fields of data. A new one was created to address this specifically, and others were modified to require memory writes, as well.

It had been observed that the benchmark suite minimized the effects of data starvation by capitalizing on the work of clever programmers. A goal during this development was to avoid the possibility of simple reinterpretation of the problem resulting in reduction of data starvation. To our surprise, this was not an easy task. Repeatedly, our software engineers were able to find ways to change the program steps in such a way that the bulk of the intended problem was reorganized to be more memory-friendly. While this may merely be the consequence of attempting to specify problems that only require a dozen lines of code to solve, it could indicate that the problem space is more limited than initially thought. In either case, several of the stressmarks required modification due to this issue.

1.2.3 Stressmark Suite

The final suite includes seven individual stressmarks, though their results are intended for collective interpretation. This table summarizes them; each is individually described in later sections.

Stressmark	*Problem*	*Memory Access*
Pointer	Pointer following.	Small blocks at unpredictable locations. Can be threaded.
Update	Pointer following with memory updates.	Small blocks at unpredictable locations.
Matrix	Conjugate gradient simultaneous equation solver.	Dependent on matrix representation. Likely to be irregular or mixed, with mixed levels of reuse.
Neighborhood	Calculate image texture measures by finding sum- and difference-histograms.	Regular access to pairs of words at arbitrary distances.
Field	Collect statistics on large field of words.	Regular, with little re-use.
Corner-Turn	Matrix transposition.	Block movement between processing nodes with practically nil computation.
Transitive Closure	Find all-pairs-shortest-path solution for a directed graph.	Dependent on matrix representation, but requires reads and writes to different matrices concurrently.

1.3 Scope

This document provides specifications for the stressmarks only. While it includes some motivation for the stressmarks, it does not supply the full motivation, analysis, or background for the stressmarks or their encompassing DIS Benchmark Suite. Consult [1] for more detailed information on the suite, analyses, and complete usage procedures.

The remainder of this document describes the stressmarks in detail. Section 2 covers the procedures to be employed when utilizing the stressmarks for the DIS program. Sections 3 through 9 provide the specifications for the individual stressmarks, and Section 10 specifies supporting functions used throughout the series of stressmarks.

2. Procedures

This section relates operational procedures to be observed during use of the benchmarks. These procedures apply to all of the stressmarks in the suite, and are intended to maximize the utility of the both the results and the overall exercise.

2.1 Which stressmarks?

The seven stressmarks are largely complimentary and should be considered as a set. Metrics collected for any one stressmark are not valuable when held independently from the rest. Ideally, participants should present results for all seven. At the very least, results for a majority of the seven can be useful, if caution is exercised when interpreting the results.

2.2 Code organization

Most examples provided herein demonstrate the function of both the data generation and the data processing segments. Typically, only the latter of these is of interest, and often, the former may be executed on a separate host platform or on an offline basis. Thus, examples should be used with caution. Similarly, when developing code, be wary of which actions are subject to metric collection and which are not.

2.3 Code optimization

Any code found within this document is for purposes of example only. There is no 'baseline' code, as for the Benchmark Suite. Hence, there is no restriction on code optimizations, nor is there a requirement that example code operate successfully without modification. Participants are expected to generate the best implementations they are able, using methods they deem appropriate.

If results are to be compared with 'baseline' performance figures, the code used to generate those figures must similarly be the best implementations that can be supplied.

Generating multiple implementations of stressmarks, each one optimizing a different performance characteristic, is encouraged. For example, one might optimize for maximum throughput in one case, minimum memory storage in another, and minimum power consumption in a third. When doing this, participants are reminded to collect results using as much of the supplied data sets as possible, considering each instance of each stressmark separately. The performance trade-offs of design decisions should be visible in the results, as this is an important element for evaluation of the candidate DIS approach.

As the stressmarks are minimalistic, the likelihood that problem-domain expertise will bias results is small. To further minimize that possibility, participants may be asked to share their algorithmic methods with other members of the DIS community. In some cases, source code supplied by a participant may be distributed to other teams.

The intent of each stressmark should be clear from its description. One should resist the temptation to optimize the results specifically for the given data sets, at the cost of performance on arbitrary sets. New data sets may be supplied at any time.

2.4 Data Sets

As with the Benchmark Suite, the primary control over problem size comes from the input data. A wide range of problem sizes is specified by the supplied input data sets. Participants are expected to measure performance of their implementations for each input file in the set, except when the memory requirements of the file exceed the capacity of the test system.

Additionally, participants may be expected to measure performance using additional input files that may be released from time-to-time as deemed necessary.

Participants are invited to create additional input files at their discretion; the provided files are to be considered a minimal set.

The data types of the values in the input and output files conform to IEEE 754 specifications. See Section 3.6 of [1] for more information.

2.5 Acceptance Tests

Expected output data sets are provided for verification and debugging purposes. The implementation of a stressmark will be considered successful for a given input data set when it produces results that match with the corresponding output data set.

The mathematics in some of the stressmark algorithms requires the manipulation of values that cannot be stored in finite-precision memory representations. Since the successful completion of a stressmark is determine by a comparison between the output of the stressmark implementation and provided results, numerical accuracy and precision must be addressed.

The data types used for the input and examples conform to the IEEE 754 specification, which also specifies the manipulation of these types to ensure the mathematically expected results and expected properties for finite arithmetic. The output files conform to that same specification. Users are not required to implement this specification, but the output of their instances of the stressmarks must have the same level of final precision, and perform to at least the same level of accuracy for numeric calculations.

2.6 Metrics

Of primary interest for all the stressmarks in this set is the trade-off between ‘performance’ and ‘cost’, where performance is focused mainly on maximizing throughput, and cost is focused mainly on programmer labor costs. Of course, there are many other important considerations relating to performance and cost; some important contributing factors are listed here:

Performance	**Cost**
Maximize throughput (primary)	Minimize programmer labor (primary)
Maximize scalability	Minimize development labor
Minimize power consumption	Maximize use of OTS parts
Maximize robustness	Minimize part count

Maximize implementation flexibility	Maximize ability to retrofit
	Minimize volume and weight

Unfortunately, the solutions offered by DIS participants vary widely enough that collection of many specific metrics is impractical. The metrics discussed here are therefore general. Participants are invited to collect and present more metrics that are pertinent to evaluation of their approaches. For example, increasing the global updates per second is important, while for others, reducing a cache-miss-rate is more indicative of project success.

In all cases, the time required to complete processing is of interest. While "wall-clock" time is essential, participants should additionally present an analysis of other timing measures, as appropriate. How improvements in stressmark performance can be expected to relate to performance improvements in the field should be analyzed. Given the fixed nature of the problem specification for each trial, time to complete is directly related to throughput. How throughput varies over the spectrum of input sets gives some notion of scalability with respect to problem size, for a given specific configuration. How the configuration itself can be scaled is another issue to be addressed in the *Architectural Description*. Multiple configurations of hardware and software are invited, but a complete series of input data sets should be run for each configuration, without intra-series modification.

The energy spent by implementers laboring in the development of each stressmark implementation is of special interest. As this is ultimately difficult to measure accurately, reviewers will rely on participant's candid reporting on this subject. A frank summary of the required skills, labor expended and problems encountered during the process would be beneficial to those establishing the utility of a given design.

Power consumption is an important consideration. Again, the early stages of development under the DIS program might make accurate quantification of power consumption impossible. However, participants are expected to include their best estimates of the power required for each stressmark in the suite. Measurement methods employed should be detailed in the report, as it is anticipated that no specific methodology may reasonably be imposed.

When measuring performance of a stressmark implementation, the following considerations must be made:

- Actual platform measurements are preferred over simulated results. It is understood that early iterations through the benchmarking process will necessarily be based on simulation, but these must give way to measurements of actual systems for reliable determinations to be achieved.
- If simulations are used, a description of the model and tools used, and the bases for the timing values, should be provided.
- All data sets should be used. They have been provided in a range of sizes, to test fixed-system-scaling effects resulting from limited-resource optimizations. Should particular data sets be unusable for some reason (e.g., the set requires more memory than that which is available), the reason should be reported.

- There may be no recompilation or manipulation of the software or hardware between runs producing final measurements. Recall that quantifying the effects of system design decisions is one of the goals of this effort. Therefore, the environment must be consistent throughout the tests to ensure validity of measurements relative to one another.
- Tests should be repeated enough times to ensure reproducibility.
- As the DIS effort is primarily concerned with memory issues, measurement of time to perform I/O operations shall ideally be factored out. However, because the relative need for–and speed of–I/O is determined by the architecture, these times should be measured and included in the report. If possible, the time for these operations should be noted, so they can be excluded when appropriate.

2.7 Submission of Results

Participants are expected to supply the following items relating to their tests. Note that this is essentially the same information as described for the Benchmark Suite.

Item	Description
Architecture description	A detailed description of the hardware and software environments utilized during testing should be supplied. The description should be sufficient that strengths and weaknesses of the architecture pertinent to the benchmarks can be understood. Known performance measures such as bisection bandwidth and feature size should be included. Limits of the architecture (e.g., maximum of 32 processors, or maximum clock rate of 100Mhz) should be identified, and if predicted performance is to be considered, it must be justified in the *Comments* section of the report. As it is unwise to compare raw timings, even for similar architectures, without considering the differences in technology between the systems, this description is critical to the process, and should be organized, detailed, and complete.
Source code	The source code used during testing should be supplied, along with corresponding documentation of the changes from the example code, and detailed documentation of the code compilation, assembly, and execution.
Implementation documentation	A detailed record of the implementation, including rationale and approach to optimizations, is expected. This is particularly important when deviations from the example code are employed, or when problems in implementation are encountered. An accurate account of the labor required to implement each benchmark is required.
Output data	Output data sets should be made available. Any deviations from the output data specification should be explained.
Measurements	Performance figures for each applicable benchmark should be supplied, along with a description of how they were obtained. Any missing measurements should be explained. Metrics in addition to those required by this specification are encouraged, but they must be accompanied by documentation of how they were gathered, and how they are pertinent to

	the analysis.
Comments	Participants are encouraged to include any other information pertinent to the benchmarking process, including explanations of special circumstances, or recommendations for improving the stressmark. To be considered, theoretical performance of an unbuilt architecture should be given and justified. Particular attention should be given to the scalability of the architecture with respect to each of the stressmarks. Results from implementations of other benchmarks are welcomed, also, though these should be sufficiently delineated so as not to obscure the data directly relevant to these stressmarks.

3. Pointer Stressmark

The *Pointer Stressmark* requires a system to repeatedly follow pointers (‘hop’) to randomized locations in memory. The memory access pattern is therefore defined by the need to collect small blocks of memory from unpredictable locations. This pattern is commonly encountered in applications, however it is normally associated with a small number of consecutive accesses. Still, it tests a capability orthogonal to the other stressmarks of this set.

The *Pointer Stressmark* consists of fetching a small number of words at a given address or offset, finding the median of the values, and using the result and an additional offset to determine the address for the next fetch. The process is repeated until a ‘magic number’ is found, or until a fixed number of fetches have been done. The purpose of the median operation is to require the access of multiple words at each location, and the additional offset reduces the likelihood of self-referential loops.

The entire process is performed multiple times for each test. This allows the test to be performed in parallel. The discussion here refers to ‘threads’ in this context, though there is no requirement that they be implemented using that method.

The values to be fetched can be array indices, or they can be memory addresses computed during initialization, at the discretion of the user. Consult section 3.5 for more information about this option.

3.1 Input

Input consists of a single ASCII file containing all the parameters required for a single run. The table below gives the format and description of each item. Items are given in the order provided, and are delimited by white space. Formats are given using C standard library *scanf()* control notation. See Section 3.2 for more details on item usage.

Item Number	Description	Format	Limits
1	Size of field of values, *f*, in words.	%lu	$2^4 \le val \le 2^{24}$
2	Size of sample window, *w*, in words.	%h	$2^0 \le val < 2^4$, *val* modulo 2 = 1
3	Maximum number of hops to be allowed for each starting value.	%lu	$2^0 \le val \le 2^{32}-1$
4	Seed for random number generator.	%ld	$1-2^{31} \le val \le -1$
5	Number of threads, *n*.	%d	$2^0 \le val \le 2^8$
$3i + 6$, $0 \le i < n$	The starting index for the *i*th thread.	%lu	$0 \le val < f$
$3i + 7$, $0 \le i < n$	The minimum ending index for the *i*th thread.	%lu	$0 \le val < f$
$3i + 8$, $0 \le i < n$	The maximum ending index for the *i*th thread.	%lu	$0 \le val < f$

3.2 Algorithm

The general procedure for this stressmark is as follows:

Step	Action
1	Read the input file.
2	Initialize the random number generator and fill the field with random numbers (see Section 10.1 for more information about the random number generators to be used) of the range [0…*f*-*w*-1]. These values represent indices into the field. If equivalent addresses are to be used, they may be computed now (see section 3.5 for more information).
3	Start the timer.
4	Perform these steps once for each thread: (a) Clear the hop count. (b) Set *index* to the starting index for this thread. (c) Fetch values at location given by *index*. This value may be an address or an address offset. See section 3.5 for more information. Get values at *index*, *index*+1, *index*+2, … *index*+*w*-1. (d) Set *index* to the sum of the median of the values obtained in step 4(c) and the hop count, modulo (*f*-*w*). (e) Increment hop count by one. If hop count equals the maximum number of hops allowed, store the hop count and exit this thread. (f) If *index* is greater than or equal to the minimum ending index for this thread, and *index* is less than the maximum ending index for this thread, store the hop count and exit this thread. Otherwise, go to step 4(c).
5	Stop the timer.
6	Write the output and metrics files.

3.3 Output

The program should return an ASCII file containing one white-space-delimited integer for each thread representing the number of 'hops' completed for that thread.

3.4 Sample Code

Sample code for the *Pointer Stressmark* is provided here. This source code, like all source code found within this document, is provided for example purposes only.

```
/*
 * Sample code for the DIS Pointer Stressmark
 *
 * Provided by Atlantic Aerospace Division, Titan Systems Corporation, 2000.
 *
 * This code is intended to serve as an example only.  It is not expected
 * to be particularly clean, efficient, user-friendly, or portable.  This
 * code is untested.  Please report discrepancies between this sample code
 * and the rest of the specifications to <dis@aaec.com>.
 *
 * Note in particular that most error checking is done using macros
 * that are only compiled into debugging versions.  This saves a small
 * amount of execution time and memory, but caution is required.  At
 * the very least, only run the final version using input files that
 * have previously and successfully been processed by the debug version.
 *
 * For this example, indices are used, rather than addresses.
 */


#include <stdio.h>                 /* needed for file I/O */
#include <time.h>                  /* needed for metrics collection */
#include <assert.h>                /* needed for debugging trap/macro */
#include <DISstressRNG.h>          /* needed for random numbers */

/*
 * Limits on input values as defined in the specifications.
 */
#define MIN_FIELD_SIZE 16          /* Minimum size of field of indices,
                                      in words. */
#define MAX_FIELD_SIZE 16777216    /* Maximum size of field of indices,
                                      in words. */
#define MIN_WINDOW_SIZE 1          /* Minimum size of median filter
                                      window, in words. */
#define MAX_WINDOW_SIZE 15         /* Maximum size of median filter
                                      window, in words. */
#define MIN_HOP_LIMIT 1            /* Minimum value of the maximal
                                      limit of the number of hops
                                      per thread. */
#define MAX_HOP_LIMIT 4294967295U  /* Minimum value of the maximal
                                     limit of the number of hops
                                      per thread. */
#define MIN_SEED -2147483647       /* Minimum value for RNG seed. */
#define MAX_SEED -1                /* Maximum value for RNG seed. */
#define MIN_THREADS 1              /* Minimum number of threads. */
#define MAX_THREADS 256            /* Maximum number of threads. */


/*
 * main()
 */
int main() {

  unsigned int *field;             /* The field of indices. */
  unsigned int f;                  /* Size of the field. */
  unsigned short int w;            /* Size of the filter window. */
  unsigned int maxhops;            /* Hop limit per thread. */
  int seed;                        /* Seed for random number generator. */
  unsigned int n;                  /* Number of threads.  Since this
                                      example code does not utilize
                                      system ’threads’, the term in this
                                      context is the number of starting
                                      index values. */
  time_t startTime;                /* Clock at start of hopping. */
  struct threadS {
    unsigned int initial;          /* Starting index. */
    unsigned int minStop;          /* Lower limit of stop range. */
    unsigned int maxStop;          /* Upper limit of stop range. */
    unsigned int hops;             /* Count of number of completed hops */
  } *thread;                       /* Array (eventually of size n) of
                                      values pertinent to each thread. */
  unsigned int l;                  /* Loop index. */

  /*
   * Read the input file.  First read the fundamental information, and
   * check it against the specifications.  Then, set up the thread array
   * and read in values for that.
   */
  fscanf(stdin,"%lu %u %lu %ld %u",
     &f, &l, &maxhops, &seed, &n); /* read the fundamentals */
  assert((f >= MIN_FIELD_SIZE) && (f <= MAX_FIELD_SIZE));
  w = (unsigned int) l;
  assert((w >= MIN_WINDOW_SIZE) && (w <= MAX_WINDOW_SIZE));
  assert(w % 2 == 1);
  assert(f > w);
  assert((maxhops >= MIN_HOP_LIMIT) && (maxhops <= MAX_HOP_LIMIT));
  assert((seed >= MIN_SEED) && (seed <= MAX_SEED));
  assert((n >= MIN_THREADS) && (n <= MAX_THREADS));
  if ((thread = (struct threadS *)malloc(n*sizeof(struct threadS))) == NULL)
      return(-1);
  for (l=0; l<n; l++) {
      fscanf(stdin,"%lu %lu %lu",
         &(thread[l].initial), &(thread[l].minStop), &(thread[l].maxStop));
      assert((thread[l].initial >= 0) && (thread[l].initial < f));
      assert((thread[l].minStop >= 0) && (thread[l].minStop < f));
      assert((thread[l].maxStop >= 0) && (thread[l].maxStop < f));
  } /* end of loop for l */

  /*
   * Now create the index field, using the provided random number generator.
   * The top end of the range of the random numbers is limited so that
   * filter windows do not need to 'wrap around' or otherwise be dealt with.
   */
  if ((field = (unsigned int *)malloc(f*sizeof(int))) == NULL)
      return(-1);
  initRand(seed);
  for (l=0; l<f; l++) {
      field[l] = randInt(0,f-w);
  } /* end of loop for l */

  /*
   * Start the timer.
   */
  startTime = time(NULL);

  /*
   * Loop through each thread.  For each one, initialize and start hopping.
   *
   */
  for (l=0; l<n; l++) {
      unsigned int index;          /* Index into field. */
      unsigned int minStop, maxStop;  /* local copies of struct values */
      unsigned int hops;           /* local copies of struct values */

      hops = 0;
      minStop = thread[l].minStop;
      maxStop = thread[l].maxStop;
      index = thread[l].initial;
      while ((hops < maxhops) &&   /* not yet done maximum # of hops, and */
         (!((index >= minStop) &&  /* index is not within key range */
            (index < maxStop)))) {
          /*
           * For each hop, the median value within the window must be found.
           * If w==1, this is a no-brainer.  Otherwise, we find it by
           * partitioning the values within the window, and counting the
           * number of elements greater than the partition.  If this number
           * is greater than the number of remaining elements between the
           * known min and max values, then the max is set to the partition;
           * otherwise, the min is set to the partition.  The partitioning is
           * repeated until there are no elements between the known min and
           * max.  Hi and lo keep track of how many are partitioned out.
           */
          unsigned int ll,lll;       /* loop indices */
          unsigned int max, min;     /* temp values during median */
          unsigned int partition;    /* temp values during median */
          unsigned int high;          /* temp values during median */
          partition = field[index];
          max = MAX_FIELD_SIZE;       /* larger than any actual index */
          min = 0;                    /* as small as any actual index */
          high = 0;
          for (ll=0; ll<w; ll++) {
              unsigned int balance;   /* how many values > partition */
              unsigned int x;         /* shorthand for field[index+ll] */
              x = field[index+ll];
              if (x > max) high++;
              else if (x > min) {
                  partition = x;
                  balance = 0;
                  for (lll=ll+1; lll<w; lll++) {
                      if (field[index+lll] > partition) balance++;
                  } /* end of loop for lll */
                  if (balance+high == w/2) break;
                  else if (balance+high > w/2) {
                      min = partition;
                  } /* end of if balance... */
                  else {
                      max = partition;
                      high++;
                  } /* end of else */
              } /* end of else x is in range */
              if (min == max) break;
          } /* end of loop for ll */
          index = (partition+hops)%(f-w); /* by now, partition is median */
          hops++;
      } /* end of while loop */
      thread[l].hops = hops;
  } /* end of loop for l */

  /*
   * Stop the timer.  Find the difference between the current time and
   * the starting time, to be reported as the total time.
   */
  startTime = time(NULL) - startTime;

  /*
   * Write out the results and metrics files.  The output consists of
   * the number of hops for each thread.  The metric reported here
   * is total time for completion.
   */
  for (l=0; l<n; l++) {
      fprintf(stdout,"%lu hops on thread %d\n",thread[l].hops,l);
  } /* end of loop for l */
  fprintf(stderr,"Total time = %u seconds.\n",(unsigned int)startTime);
  free(field);
  free(thread);

} /* end of main() */
```

3.5 Notes

Note the following items relating to the *Pointer Stressmark*:

- Offsets vs. addresses. The values stored in the field may be array indices, memory offset values, or memory addresses. The selection is left to the implementer's assessment of the costs and rewards associated with each. Note that offsets or addresses must be computed during the initialization phase. The time required to do this need not be included in the time recorded for performance assessment.

- Word size. For this stressmark, a *word* must represent a memory unit large enough to contain discrete array indices, offsets, or addresses.
- Relocatable code. Be aware of code relocation issues if memory addresses are utilized. While there is no requirement that final executables be relocatable, it should be reported when they are not.
- Parallel execution. As should be clear from the algorithm, there are no dependencies between threads other than the static field of indices. So, threads may be executed in arbitrary order, or simultaneously. While the term ‘thread’ has been used here to describe this attribute, there is no requirement that the programming construct of the same name must be used.
- Input limits. Implementers should strive to develop code that works properly for all legal input values. With regard to the size of the field, *f*, it is understood that large values may not be supported by some systems. Please report which tests could not be performed as a result of this limitation.

4. Update Stressmark

The *Update Stressmark* is extremely similar to the *Pointer Stressmark*. It should be considered a companion stressmark, as their memory access patterns are nearly identical to one another. The major difference is that, for this stressmark, elements in the memory field are updated at each access. This consequently makes parallelism at the 'thread' level, as discussed in section 3, an impossibility, because results become nondetermnistic. However, it should be noted that parallel implementations could be made practical at a fine level.

The *Update Stressmark* consists of:

- fetching a small number of words at a given address or offset,
- updating the word at the given address,
- finding the median of the values, and
- using the result to determine the address for the next fetch.

The process is repeated until a 'magic number' is found, or until a fixed number of fetches have been done. The purpose of the median operation is to require the access of multiple words at each location. The update requires a memory write operation, and additionally reduces the likelihood of self-referential loops.

The values to be fetched can be array indices, or they can be memory addresses computed during initialization, at the discretion of the user. Consult section 4.5 for more information about this option.

4.1 Input

Input consists of a single ASCII file containing all the parameters required for a single run. The table below gives the format and description of each item. Items are given in the order provided, and are delimited by white space. Formats are given using C standard library *scanf()* control notation. See Section 3.2 for more details on item usage.

Item Number	Description	Format	Limits
1	Size of field of values, *f*, in words.	%ld	$2^4 \le val \le 2^{24}$
2	Size of sample window, *w*, in words.	%h	$2^0 \le val < 2^4$, *val* modulo 2 = 1
3	Maximum number of hops to be allowed.	%ld	$2^0 \le val \le 2^{32}-1$
4	Seed for random number generator.	%ld	$1-2^{31} \le val \le -1$
5	The starting index.	%ld	$0 \le val < f$
6	The minimum ending index.	%ld	$0 \le val < f$
7	The maximum ending index.	%ld	$0 \le val < f$

4.2 Algorithm

The general procedure for this stressmark is as follows:

Step	Action
1	Read the input file.
2	Initialize the random number generator and fill the field with random numbers (see Section 10.1 for more information about the random number generators to be used) of the range [0…f-w-1]. These values represent indices into the field. If equivalent addresses are to be used, they may be computed now (see section 3.5 for more information).
3	Start the timer.
4	Perform these steps: (a) Clear the hop count. (b) Set *index* to the starting index. (c) Fetch values at location given by *index*. This value may be an address or an address offset. See section 3.5 for more information. Get values at *index*, *index*+1, *index*+2, … *index*+*w*-1. (d) Set the value referenced by *index* to the sum of its previous value and the hop count, modulo (*f*-*w*). E.g., `x[I]=(x[I]+c)%(f-w);` (e) Set *index* to the median of the values obtained in step 4(c). (f) Increment the hop count by one. If the hop count equals the maximum number of hops allowed, store the hop count and proceed to step 5. (g) If *index* is greater than or equal to the minimum ending index, and *index* is less than the maximum ending index, store the hop count and proceed to step 5. Otherwise, go to step 4(c).
5	Stop the timer.
6	Write the output and metrics files.

4.3 Output

The program should return an ASCII file containing one white-space-delimited integer for each thread representing the number of ‘hops’ completed for that thread.

4.4 Sample Code

Sample code for the *Update Stressmark* is provided here. This source code, like all source code found within this document, is provided for example purposes only.

```
/*
 * Sample code for the DIS Update Stressmark
 *
 * Provided by Atlantic Aerospace Division, Titan Systems Corporation, 2000.
 *
 * This code is intended to serve as an example only.  It is not expected
 * to be particularly clean, efficient, user-friendly, or portable.  This
 * code is untested.  Please report discrepancies between this sample code
 * and the rest of the specifications to <dis@aaec.com>.
 *
 * Note in particular that most error checking is done using macros
 * that are only compiled into debugging versions.  This saves a small
 * amount of execution time and memory, but caution is required.  At
 * the very least, only run the final version using input files that
 * have previously and successfully been processed by the debug version.
 *
 * For this example, indices are used, rather than addresses.
 */


#include <stdio.h>                  /* needed for file I/O */
#include <time.h>                   /* needed for metrics collection */
#include <assert.h>                 /* needed for debugging trap/macro */
#include <DISstressRNG.h>           /* needed for random numbers */

/*
 * Limits on input values as defined in the specifications.
 */
#define MIN_FIELD_SIZE 16           /* Minimum size of field of indices,
                                       in words. */
#define MAX_FIELD_SIZE 16777216     /* Maximum size of field of indices,
                                       in words. */
#define MIN_WINDOW_SIZE 1           /* Minimum size of median filter
                                       window, in words. */
#define MAX_WINDOW_SIZE 15          /* Maximum size of median filter
                                       window, in words. */
#define MIN_HOP_LIMIT 1             /* Minimum value of the maximal
                                       limit of the number of hops
                                       per thread. */
#define MAX_HOP_LIMIT 4294967295U  /* Minimum value of the maximal
                                       limit of the number of hops
                                       per thread. */
#define MIN_SEED -2147483647        /* Minimum value for RNG seed. */
#define MAX_SEED -1                 /* Maximum value for RNG seed. */


/*
 * main()
 */
int main() {

  unsigned int *field;              /* The field of indices. */
  unsigned int f;                   /* Size of the field. */
  unsigned int index;               /* Index into the field. */
  unsigned short int w;             /* Size of the filter window. */
  unsigned int maxhops;             /* Hop limit. */
  int seed;                         /* Seed for random number generator. */
  time_t startTime;                 /* Clock at start of hopping. */
  unsigned int initial;             /* Starting index. */
  unsigned int minStop;             /* Lower limit of stop range. */
  unsigned int maxStop;             /* Upper limit of stop range. */
  unsigned int hops;                /* Count of number of completed hops */
  unsigned int l;                   /* Loop index. */

  /*
   * Read the input file.
   */
  fscanf(stdin,"%u %u %u %d %u %u %u",
     &f, &l, &maxhops, &seed, &initial, &minStop, &maxStop);
  assert((f >= MIN_FIELD_SIZE) && (f <= MAX_FIELD_SIZE));
  w = (unsigned int) l;
  assert((w >= MIN_WINDOW_SIZE) && (w <= MAX_WINDOW_SIZE));
  assert(w % 2 == 1);
  assert(f > w);
  assert((maxhops >= MIN_HOP_LIMIT) && (maxhops <= MAX_HOP_LIMIT));
  assert((seed >= MIN_SEED) && (seed <= MAX_SEED));
  assert((initial >= 0) && (initial < f));
  assert((minStop >= 0) && (minStop < f));
  assert((maxStop >= 0) && (maxStop < f));
```

```
    /*
     * Now create the index field, using the provided random number generator.
     * The top end of the range of the random numbers is limited so that
     * filter windows do not need to 'wrap around' or otherwise be dealt with.
     */
    if ((field = (unsigned int *)malloc(f*sizeof(int))) == NULL)
        return(-1);
    initRand(seed);
    for (l=0; l<f; l++) {
        field[l] = randInt(0,f-w);
    } /* end of loop for l */

    /*
     * Start the timer.
     */
    startTime = time(NULL);

    /*
     * Initialize and start hopping.
     */
    hops = 0;
    index = initial;
    while ((hops < maxhops) && /* have not yet done maximum # of hops, and */
       (!((index >= minStop) && /* index is not within special stopping range */
          (index < maxStop)))) {
        /*
         * For each hop, the median value within the window must be found.
         * If w==1, this is a no-brainer.  Otherwise, we find it by
         * partitioning the values within the window, and counting the
         * number of elements greater than the partition.  If this number
         * is greater than the number of remaining elements between the
         * known min and max values, then the max is set to the partition;
         * otherwise, the min is set to the partition.  The partitioning is
         * repeated until there are no elements between the known min and
         * max.  Hi and lo keep track of how many are partitioned out.
         */
        unsigned int ll,lll;         /* loop indices */
        unsigned int max, min;       /* temp values during median */
        unsigned int partition;      /* temp values during median */
        unsigned int high;           /* temp values during median */
        max = MAX_FIELD_SIZE;        /* larger than any actual index */
        min = 0;                     /* as small as any actual index */
        high = 0;
        for (ll=0; ll<w; ll++) {     /* start at second element */
            unsigned int balance;    /* how many values > partition */
            unsigned int x;          /* shorthand for field[index+ll] */
            x = field[index+ll];
            if (x > max) high++;
            else if (x > min) {
                partition = x;
                balance = 0;
                for (lll=ll+1; lll<w; lll++) {
                    if (field[index+lll] > partition) balance++;
                } /* end of loop for lll */
                if (balance+high == w/2) break;
                else if (balance+high > w/2) {
                    min = partition;
                } /* end of if balance... */
                else {
                    max = partition;
                    high++;
                } /* end of else */
            } /* end of else x is in range */
            if (min == max) break;
        } /* end of loop for ll */
        index=(partition+hops)%(f-w);   /* by now, partition is the median */
        hops++;
```

```
} /* end of while loop */

/*
 * Stop the timer.  Find the difference between the current time and
 * the starting time, to be reported as the total time.
 */
startTime = time(NULL) - startTime;

/*
 * Write out the results and metrics files.  The output consists of
 * the number of hops.  The metric reported here is total time for
 * completion.
 */
fprintf(stdout,"%u hops\n",hops);
fprintf(stderr,"Total time = %u seconds.\n",(unsigned int)startTime);
free(field);
return(1);
} /* end of main() */
```

4.5 Notes

Note the following items relating to the *Update Stressmark*:

- Offsets vs. addresses. The values stored in the field may be array indices, memory offset values, or memory addresses. The selection is left to the implementer's assessment of the costs and rewards associated with each. Note that offsets or addresses must be computed during the initialization phase. The time required to do this need not be included in the time recorded for performance assessment.
- Word size. For this stressmark, a *word* must represent a memory unit large enough to contain integer array indices, offsets, or addresses.
- Relocatable code. Be aware of code relocation issues if memory addresses are utilized. While there is no requirement that final executables be relocatable, it should be reported when they are not.
- Input limits. Implementers should strive to develop code that works properly for all legal input values. With regard to the size of the field, *f*, it is understood that large values may not be supported by some systems. Please report which tests could not be performed as a result of this limitation.

5. Matrix Stressmark

The *Matrix Stressmark* characterizes operations dealing with data that is stored in a compacted form, and accessed in mixed patterns. In this stressmark, the iterative conjugate gradient method is used to solve a linear system represented by the equation $\boldsymbol{A} \bullet \boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{A}$ is a sparse *nxn* matrix, and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors with n elements each. For simplicity of initial data generation, matrix $\boldsymbol{A}$ is positive-definite and symmetric.

The stressmark requires solving the equation $\boldsymbol{A} \bullet \boldsymbol{x} = \boldsymbol{b}$ for vector $\boldsymbol{x}$. As the required method is iterative, the steps are performed until $\boldsymbol{x}$ is found to be within a specified error tolerance, or for a specified maximum number of iterations, whichever occurs first. A random number generator, supplied as part of the stressmark specification, is used to populate the matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$ initially. In this way, the input required to specify the trials has been reduced to a small set of parameters, which are sufficient to define the linear system. Output consists of a value dependent on the solution vector $\boldsymbol{x}$.

5.1 Input

Input for the *Matrix Stressmark* is provided in a single ASCII file, as a list of parameters defining the linear system to be solved. All values within the input file are integer or floating-point numbers, and are white-space-delimited. The table below describes each field. Formats are given using C standard library *scanf*() control notation.

Item Number	Description	Format	Limits
1	A seed value for the random number generator.	%ld	$1-2^{31} < val < -1$
2	The dimension, *n*, of matrix *A* and vectors *x* and *b*.	%d	$1 < val \le 2^{15}$
3	The number of nonzero elements to be inserted within matrix *A*.	%d	$n < val \le n^2$
4	The maximum number of iterations to be performed. The actual number of iterations required may be less if the calculated error is lower than the tolerance specified by the next field.	%d	$0 < val \le 2^{16}$
5	The tolerance of error for the solution vector.	%e	$1.0e-7 < val < 0.5$

5.2 Algorithmic Specification

The input to the *Matrix Stressmark* gives parameters needed for generation of a matrix *A* and a vector *b*, which are subsequently solved by way of an iterative conjugate gradient method. For clarity, as these processes are likely to be executed on different hosts, they are presented separately here.

5.2.1 Data Generation

Rather than provide the full matrix *A* and vector *b*, a procedure to generate them is specified. The matrix *A* is constructed to be a symmetric positive definite matrix to ensure that there is a unique solution *x* to the linear system. The matrix *A* is populated such that its non-zero values yield a strictly diagonally dominant matrix [6], which can be defined as:

$$|\boldsymbol{a}_{ii}| > \sum_{\substack{j=1 \\ j\neq i}}^{N} |\boldsymbol{a}_{ij}| \qquad \boldsymbol{i = 1, n} \qquad \text{Equation 5-a}$$

With the additional criterion that each diagonal element is positive to ensure matrix *A* is positive definite, this becomes

$$\boldsymbol{a}_{ii} > \sum_{\substack{j=1 \\ j\neq i}}^{N} |\boldsymbol{a}_{ij}| \qquad \boldsymbol{i = 1, n} \qquad \text{Equation 5-b}$$

And since *A* is symmetric, *A* equals A^T and

$$a_{ij} = a_{ji} \qquad \text{Equation 5-c}$$

implying that both matrix *A* and matrix A^T are strictly diagonally dominant. So, for *A* symmetric and positive definite,

$$\boldsymbol{a}_{ii} > \sum_{\substack{j=1 \\ j\neq i}}^{N} |\boldsymbol{a}_{ij}| = \sum_{\substack{j=1 \\ j\neq i}}^{N} |\boldsymbol{a}_{ji}| \qquad \boldsymbol{i = 1, n} \qquad \text{Equation 5-d}$$

Therefore, the needed data can be generated using the following procedure:

Step	Action
1	Read the input file.
2	Initialize the random number generator.
3	Create matrix *A* of dimension *n* and fill with zeros. (Sparse matrix storage schemes generally assume unspecified values are zero.)
4	Select a location in the upper triangular region of matrix *A* by generating random integers *i* in the range [1…*n*-1] and *j* in the range [0…*i*-1]. Use *randomUInt*() for this. These ranges assume ordinal indexing, and should be adjusted for cardinal index systems.
5	If $A_{i,j}$ is nonzero, increment *i* by one. If *i* equals *n*, increment *j* and set *i* to *j*+1. If *j* equals *n*-1, set *j* to 0, and set *i* to 1. Repeat this step until $A_{i,j}$ is zero.
6	Using *randomNonZeroFloat*(), generate a random real number *x* in the range [-3.4e10/n, 3.4e10/n], excluding the range [-1.0e-10…1.0e-10]. Let $A_{i,j} = A_{j,i} = x$.
7	For each diagonal element $A_{j,j}$, generate a random real number *y* in the

	range [1.0e-10…3.4e10] using *randomNonZeroFloat*(). If y is greater than $\Sigma_{i\neq j}A_{i,j}$, set $A_{j,j}$ to y; otherwise, set $A_{j,j}$ to $\Sigma_{i\neq j}A_{i,j}+y$.
8	For each element b_i, set b_i to a random real number in the range [-3.4e10 …3.4e10] generated using *randomFloat*().

5.2.2 Conjugate Gradient Method

The *Matrix Stressmark* consists of the conjugate gradient iterative method for solving a linear system [4]. Each input file specifies an *n*x*n* linear system represented by

$$\boldsymbol{A} \bullet \boldsymbol{x} = \boldsymbol{b}$$

Equation 5-e

Here, '•' denotes matrix multiply. The conjugate gradient algorithm is used to determine the vector x to within the specified error tolerance, or for the specified maximum number of iterations, if that occurs first.

Define two vectors, r_k and p_k, for $k = 1,2,...$ (denoting the iteration count). Choose

$$\boldsymbol{r}_1 = \boldsymbol{b} - \boldsymbol{A} \bullet \boldsymbol{x}_1$$

Equation 5-f

and form the sequence of improved estimates

$$\boldsymbol{x}_{\boldsymbol{k}+1} = \boldsymbol{x}_{\boldsymbol{k}} + \alpha_{\boldsymbol{k}} \boldsymbol{p}_{\boldsymbol{k}}$$

Equation 5-g

to obtain our solution x. For initial vectors r_1, set $p_1=r_1=b$, and choose $x_1=(0…0)$. For each subsequent iteration, use the following equations to update the vectors:

$$\alpha_k = \frac{\boldsymbol{r}_k^T \bullet \boldsymbol{r}_k}{\boldsymbol{p}_k^T \bullet \left(\boldsymbol{A} \bullet \boldsymbol{p}_k\right)}$$

Equation 5-h

$$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k \boldsymbol{A} \bullet \boldsymbol{p}_k$$

Equation 5-i

$$\beta_k = \frac{\boldsymbol{r}_{k+1}^T \bullet \boldsymbol{r}_{k+1}}{\boldsymbol{r}_k^T \bullet \boldsymbol{r}_k}$$

Equation 5-j

$$\boldsymbol{p}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k$$

Equation 5-k

As long as the recurrence does not break down because one of the denominators is zero (which theoretically cannot happen for symmetric positive definite matrix A), the iteration will complete in n steps or less. Perform the iteration steps outlined above until:

- the maximum specified number of iterations have been performed, or
- the error is within specified tolerance, which is defined as when the solution x satisfies the equation:

$$\frac{|\boldsymbol{A} \bullet \boldsymbol{x} - \boldsymbol{b}|}{|\boldsymbol{b}|} \le errorTolerance$$

Equation 5-l

5.3 Output

The output for the *Matrix Stressmark* should be an ASCII text file containing three values separated by white space:

Field	Description	Type	Format
1	Sum of the elements of vector *x*: $\sum_{i=1}^{N} x_i$	*float*	*%9.4e*
2	Actual number of iterations performed	*integer*	*%d*
3	Actual error: $\frac{\lvert \mathbf{A} \bullet \mathbf{x} - \mathbf{b} \rvert}{\lvert \mathbf{b} \rvert}$	*float*	*%9.4e*

5.4 Pseudo-code

Pseudo-code for the top-level of the stressmark is provided here. All code and pseudo-code within this document is provided for example only. Implementers are encouraged to make modifications, as long as the functionality remains unchanged. Note, however, that functions relating to input data generation must match exactly for correct results. The random number generators supplied in Section 10 should be used without modification.

```
MAIN()
/*
 * Sample code for the DIS Matrix Stressmark
 *
 * Provided by Atlantic Aerospace Division,
 * Titan Systems Corporation, 2000.
 */

open input file
read in seed, dim, numberNonzero, maxIterations, and errorTolerance.
call randInit(seed)

matrixA = initMatrix(dim, numberNonzero)
vectorB = initVector(dim)
vectorX = zeroVector(dim)

start timer
call biConjugateGradient(A, B, X, errorTolerance, maxIterations,
                         &actualError, &actualIteration)
stop timer

sum = 0;
loop for each k = 1 to n
    sum = sum + X[k]
end loop

write sum, actualError, actualIteration to output file
write time to metric file

return
```

```
biConjugateGradient(matrixA, vectorB, vectorX
                    errorTolerance, maxIterations,
                    *actualError, *actualIteration)

create scratch vectors vectorR, vectorP, matrixAvectorP, nextVectorR

vectorR = vectorB – matrixA * vectorX
```

```
vectorP = vectorR

error = |matrixA * vectorX – vectorB| / |vectorB|

iteration = 0
while (iteration < maxIterations AND error < errorTolerance) do

   alpha = (transpose(vectorR) * vectorR) /
           (transpose(vectorP) * (matrixA * vectorP))
   nextVectorR = vectorR – alpha * (matrixA * vectorP)
   beta = (transpose(nextVectorR) * nextVectorR) /
          (transpose(vectorR) * vectorR)

   /* update values */
   vectorX = vectorX + alpha * vectorP
   vectorP = nextVectorR + beta * vectorP
   vectorR = nextVectorR
   error = |matrixA * vectorX – vectorB| / |vectorB|

   increment iteration by 1
end while

*actualError = error
*actualIteration = iteration

return
```

5.5 Notes

Note the following items relating to the *Matrix Stressmark*:

- Efficient storage of the matrix can have a substantial impact on performance. The Yale sparse storage scheme is one example suggested for consideration; it can be found in [7]. Participants are encouraged to tailor storage to their architectures.
- Much of the pseudo-code presented in section 5.4 relates to matrix data generation, rather than the processing of interest to this stressmark. It is presumed that for many participants, a more expedient method is to generate the matrices offline, and download them to code which manipulates them appropriately.

6. Neighborhood Stressmark

The *Neighborhood Stressmark* deals with data that is organized in multiple dimensions, and operated upon by neighborhood operators. Memory access is therefore somewhat localized along one dimension, and spread substantially along all others. Occasionally, processing can be organized such that memory is accessed by multiple synchronous threads with unit strides.

Image processing applications commonly include this type of operator. Clever programmers will organize code to maximize order of address access, but this is not always practical.

For this stressmark, the relationships of pairs of pixels within a randomly generated image are measured. These features quantify the texture of the image, and require memory access to pairs of pixels with specific spatial relationships.

Imagery to be measured is constructed by populating a blank square image with a multitude of line segments, where the line endpoints, width, and brightness values are randomly generated. The width and bit-depth of the image are input at run-time.

The texture measurements are obtained by estimating a gray-level co-occurrence matrix (GLCM) [9]. The matrix contains information about the spatial relationships between pixels within an image. Statistical descriptors of the co-occurrence matrix have been used as a practical method for utilizing these spatial relationships. The descriptors can be estimated by using sum- and difference-histograms [10].

Two statistical descriptors, *GLCM entropy* and *GLCM energy*, are calculated within the stressmark. Each is calculated for multiple distances and directions, as defined in Section 6.2.

6.1 Input

Input consists of a single ASCII file containing all the parameters required for a single run. The table below gives the format and description of each item. Items are given in the order provided, and are delimited by white space. Formats are given using C standard library *scanf*() control notation.

Item Number	Description	Format	Limits
1	A seed for the random number generator.	%ld	$1-2^{31} < val < -1$
2	The bit-depth of the image. Pixel values range from 0 to $2^{val}-1$.	%d	$7 \leq val \leq 15$
3	The dimension, *dim*, of the input image. Images are always square, so *dim* is both the width and the height.	%d	$1 < val \leq 2^{15}$
4	The number of line segments to be inserted into the image.	%d	$0 < val \leq 2^{16}$

5	The minimum thickness, in pixels, of the line segments, *minThickness*.	%d	$0 < val < dim$
6	The maximum thickness, in pixels, of the line segments.	%d	$minThickness \leq val$, $val < dim$
7	The shorter of the distances between pixels to be measured.	%d	$0 < val < dim$
8	The longer of the distances between pixels to be measured.	%d	$0 < val < dim$

6.2 Algorithmic Specification

The input to the *Neighborhood Stressmark* includes parameters needed for generation of an image, which is subsequently used as 'input' for the GLCM estimation algorithm. These processes are presented separately here, for clarity, though they are likely to be executed on different hosts.

6.2.1 Data Generation

Rather than provide the full input image, a procedure to generate it is specified. Essentially, a blank square image is populated with randomly generated line segments. The specific procedure appears in the following table and in the example code below:

Step	Action
1	Read the input file.
2	Initialize the random number generator by calling *randInit*(*seed*) (see Section 10 for more information about the random number generators).
3	For each line segment to be inserted, perform the procedure described in the table below once.

To insert a single line segment in the image, use the procedure described in the following table. The algorithm follows the *draw_line()* routine found in [8], but is enhanced to include a line thickness, and to interpolate the pixel values along the line.

Step	Action
1	Select the starting point of the line segment within the image. It is found by selecting an unsigned random integer, *k*, in the range $[0 \ldots dim^2\text{-}1]$, using *randomUInt*(). Row and column indices can be found using the following relations: $X_{Start} = k / dim$ (integer divide) $Y_{Start} = k \% dim$ (remainder after integer divide)
2	Using the same methods as those of step 1, select the ending point (X_{End}, Y_{End}) of the line segment.
3	Using *randomUInt*(), and the ranges specified by the input parameters, find a value for line thickness, *T*.

```
4     Using randomUInt(), and the ranges specified by the input parameters,
      find the starting pixel intensity value, $Z_{Start}$.
5     Using randomUInt(), and the ranges specified by the input parameters,
      find the ending pixel intensity value, $Z_{End}$.
6     Set $x$ to $X_{Start}$, $y$ to $Y_{Start}$, and $z$ to $Z_{Start}$.
7     Determine slope of the line segment: if $|Y_{End} – Y_{Start}| < |X_{End} – X_{Start}|$ then per-
      form step 8 and skip step 9.  Otherwise, skip to step 9.
8     Set $d$ to $2\,|Y_{End} – Y_{Start}| - |X_{End} – X_{Start}|$.  For each $x$ sequentially in the range
      $[X_{Start}\ldots X_{End}]$, perform the following three steps:
8a        For each $y'$ in the range $[y\text{-}floor(T/2)\ldots\ y+floor(T/2)] \cap [0\ldots dim]$,
          set $Image_{x,y'}$ to $floor(z)$.
8b        If $d \geq 0$:
              add $signum(Y_{End} – Y_{Start})$ to $y$, and
              deduct $2\,|X_{End} – X_{Start}|$ from $d$.
8c        Add $signum(X_{End} – X_{Start})$ to $x$.
          Add $2\,|Y_{End} – Y_{Start}|$ to $d$.
          Add $(Z_{End} – Z_{Start}) / |X_{End} – X_{Start}|$ to $z$.
9     Set $d$ to $2\,|X_{End} – X_{Start}| - |Y_{End} – Y_{Start}|$.  For each $y$ sequentially in the range
      $[Y_{Start}\ldots Y_{End}]$, perform the following three steps:
9a        For each $x'$ in the range $[x\text{-}floor(T/2)\ldots\ x+floor(T/2)] \cap [0\ldots dim]$,
          set $Image_{x',y}$ to $floor(z)$.
9b        If $d \geq 0$:
              add $signum(X_{End} – X_{Start})$ to $x$, and
              deduct $2\,|Y_{End} – Y_{Start}|$ from $d$.
9c        Add $signum(Y_{End} – Y_{Start})$ to $y$.
          Add $2\,|X_{End} – X_{Start}|$ to $d$.
          Add $(Z_{End} – Z_{Start}) / |Y_{End} – Y_{Start}|$ to $z$.
```

6.2.2 GLCM Estimation

The *Neighborhood Stressmark* entails estimation of two texture measurements obtained through the gray-level co-occurrence matrix (GLCM). These descriptors are *GLCM entropy* and *GLCM energy* [9]. Two input parameters–*distanceShort* and *distanceLong*–define the spacing to use when calculating the descriptors, thus determining the scale of the textures being measured.

GLCM entropy and *GLCM energy* can be found, without calculation of a gray-level co-occurrence matrix, by finding a sum histogram, *sumHist*, and a difference histogram, *diffHist* [10]. Like the GLCM, these histograms are dependent on a specific offset distance and direction. The sum histogram is simply the normalized histogram of the sums of all pairs of pixels a given distance and direction from one another. Likewise, the difference histogram is the normalized histogram of the differences of all pairs of pixels a given distance and direction from one another. For our purposes, each histogram requires one bin for each possible result, or $2^{bit\ depth\ +\ 1}$ bins.

Based on the histograms, the GLCM descriptors are defined as:

$$GLCM\ entropy\ =\ -\sum_i sumHist(i) * log[sumHist(i)] - \sum_j diffHist(j) * log[diffHist(j)]$$

Equation 6-a

$$GLCM\ energy\ =\ \sum_i sumHist(i)^2 * \sum_j diffHist(j)^2$$

where *sumHist(i)* is the *i*th bin of the normalized sum histogram, and *diffHist(j)* is the *j*th bin of the normalized difference histogram for the distance and direction of interest.

For this stressmark, the *GLCM energy* and *GLCM entropy* are found for each of four directions at two distances. The distances are given as the input parameters *distanceShort* and *distanceLong*. The directions are constant for all tests: 0° (horizontal), 45° (right diagonal), 90° (vertical), and 135° (left diagonal). Note that since images are typically thought of as growing rightward and downward from their origin, these directions represent angles swept clockwise. Note also that with square pixels, these angles never result in quantization issues.

6.3 Output

The output for the *Neighborhood Stressmark* is an ASCII file containing a set of descriptor values separated by white space. The contents are described in this table.

Field	Description			Type	Format
	Distance	Direction	Measure		
1	DistanceShort	0°	GLCM entropy	*float*	*m.dddd* E±*xx*
2			GLCM energy	*float*	*m.dddd* E±*xx*
3		45°	GLCM entropy	*float*	*m.dddd* E±*xx*
4			GLCM energy	*float*	*m.dddd* E±*xx*
5		90°	GLCM entropy	*float*	*m.dddd* E±*xx*
6			GLCM energy	*float*	*m.dddd* E±*xx*
7		135°	GLCM entropy	*float*	*m.dddd* E±*xx*
8			GLCM energy	*float*	*m.dddd* E±*xx*
9		0°	GLCM entropy	*float*	*m.dddd* E±*xx*

10	DistanceLong		GLCM energy	*float*	*m.dddd* E±*xx*
11		45°	GLCM entropy	*float*	*m.dddd* E±*xx*
12			GLCM energy	*float*	*m.dddd* E±*xx*
13		90°	GLCM entropy	*float*	*m.dddd* E±*xx*
14			GLCM energy	*float*	*m.dddd* E±*xx*
15		135°	GLCM entropy	*float*	*m.dddd* E±*xx*
16			GLCM energy	*float*	*m.dddd* E±*xx*

6.4 Sample Code

Sample code for a correct – but inefficient – implementation of the descriptor calculation follows. This code, like all code found within this document, is provided for example purposes only. Indices for the image range in row and column dimension by [0,*dim*-1] inclusively. Let the *x* denote the column and *y* denote the row of the image, so that *image*(*x*,*y*) represents the pixel value at column *x* and row *y*.

```
/*
 * Sample code for the DIS Neighborhood Stressmark
 *
 * Provided by Atlantic Aerospace Division, Titan Systems Corporation, 2000.
 *
 * This code is intended to serve as an example only. It is not expected
 * to be particularly clean, efficient, user-friendly, or portable. This
 * code is untested. Please report discrepancies between this sample
 * code and the rest of the specifications to <dis@aaec.com>.
 *
 * Note in particular that most error checking is done using macros
 * that are only compiled into debugging versions.  This saves a small
 * amount of execution time and memory, but caution is required.  At
 * the very least, only run the final version using input files that
 * have previously and successfully been processed by the debug version.
 *
 * main() This is the mainline for the Neighborhood Stressmark. Input
 *        is taken from stdin, output is sent to stdout, and metric
 *        output is sent to stderr.
 */

#include <stdio.h>           /* needed for file I/O */
#include <stdlib.h>          /* define malloc() and free() */
#include <time.h>            /* needed for metrics collection */
#include <assert.h>          /* needed for debugging trap/macro */
#include <DISstressRNG.h>    /* needed for random number functions */

#define TRUE                1
#define FALSE               !TRUE
#define SUCCESS             TRUE
#define ERROR               FALSE

#define MIN_PIXEL                   0 /* minimum legal image value */
#define MAX_DIMENSION           32768 /* maximum image dimension */
#define MIN_SEED          -2147483647 /* minimum seed value */
#define MAX_SEED                   -1 /* maximum seed value */
#define MAX_NUMBER_LINES        65536 /* max num image line segments */
#define MIN_BIT_DEPTH               7 /* min bit depth of image values */
#define MAX_BIT_DEPTH              15 /* max bit depth of image values */

typedef struct {
  int           column;     /* value of column in image */
  int           row;        /* value of row in image */
} Coord;

/*
 * Neighborhood structure consists of the GLCM descriptors entropy and
 * energy for each of 2 distances and 4 angles.
 */
typedef struct {
  float entropy;        /* GLCM entropy */
  float energy;         /* GLCM energy */
} Descriptors;

typedef struct {
  Descriptors   deg0;   /* features for   0 degrees */
  Descriptors   deg45;  /* features for  45 degrees */
  Descriptors   deg90;  /* features for  90 degrees */
  Descriptors   deg135; /* features for 135 degrees */
} Angles;

typedef struct {
  Angles        distShort;      /* features for short distance value */
  Angles        distLong;       /* features for long distance value */
} Neighborhood;

typedef Pixel short int;     /* needs to hold MAX_BIT_DEPTH-bit value */

extern Pixel *createImage ();
extern void drawLineSegment ();
extern void neighborhoodCalculation ();
extern void calcEntropyEnergy ();

/*
 * main()
 */
int main()
{ /* beginning of main() */
  long int     seed;            /* random seed to initialize rng */
  int          dimension;       /* dimension of image to create */
  int          numberLines;     /* # line segments for image */
  int          minThickness;    /* min thickness of line segments */
  int          maxThickness;    /* max thickness of line segments */
  int          distanceShort;   /* short distance for GLCM descriptors*/
  int          distanceLong;    /* long distance for GLCM descriptors */
  int          bitDepth;         /* bit depth for legal image range */
  int          maxPixel;         /* maximum legal image value */
  Pixel        *image;          /* handle to pointer to input image */
  Neighborhood values;          /* descriptor values */

  time_t beginTime;              /* beginning time stamp */
  time_t endTime;                /* ending time stamp */

  /*
   * Read in the input parameters and check for validity.
   */
  fscanf(stdin,"%ld %d %d %d %d %d %d %d\n",
     &seed, &bitDepth, &dimension, &numberLines,
     &minThickness, &maxThickness,
     &distanceShort, &distanceLong);
  assert((seed >= MIN_SEED) && (seed <= MAX_SEED));
  assert((dimension > 0) && (dimension <= MAX_DIMENSION));
  assert((numberLines > 0) && (numberLines <= MAX_NUMBER_LINES));
  assert((minThickness > 0) && (minThickness < dimension));
  assert((maxThickness >= minThickness) && (maxThickness < dimension));
  assert((distanceShort > 0) && (distanceShort < dimension));
  assert((distanceLong > 0) && (distanceLong < dimension));
  assert((bitDepth >= MIN_BIT_DEPTH) && (bitDepth <= MAX_BIT_DEPTH));
  /*
   * Initialization
   */
  randInit(seed);       /* initialize random number generator */
  maxPixel = (1 << bitDepth) - 1; /* 2 ^ bitDepth */
  image = createImage(dimension, maxPixel, numberLines,
                      minThickness, maxThickness);
  assert(image != NULL);

  /*
   * Perform neighborhood calculation.
   */
  beginTime = time(NULL);   /* set beginning time */
  neighborhoodCalculation(image,dimension,
                          distanceShort,distanceLong,&values);
  endTime = time(NULL);     /* set ending time */

  /*
   * Write out the descriptors and metrics.  Clean up and exit.
   */
  fprintf(stdout,"%9.4e %9.4e %9.4e %9.4e %9.4e %9.4e %9.4e %9.4e ",
      values.distShort.deg0.entropy,
      values.distShort.deg0.energy,
      values.distShort.deg45.entropy,
      values.distShort.deg45.energy,
      values.distShort.deg90.entropy,
      values.distShort.deg90.energy,
      values.distShort.deg135.entropy,
      values.distShort.deg135.energy);
  fprintf(stdout,"%9.4e %9.4e %9.4e %9.4e %9.4e %9.4e %9.4e %9.4e\n",
      values.distLong.deg0.entropy,
      values.distLong.deg0.energy,
      values.distLong.deg45.entropy,
      values.distLong.deg45.energy,
      values.distLong.deg90.entropy,
      values.distLong.deg90.energy,
      values.distLong.deg135.entropy,
      values.distLong.deg135.energy);
  fprintf(stderr,"Time for neighborhood stressmark = %f\n",
          difftime(endTime, beginTime);

  free((Pixel *) image);
  return(SUCCESS);
} /* end main() */
```

```
/*
 * createImage() This routine allocates memory for a square image
 *                     for the given input parameters and fills in
 *                     random line segments as discussed in the
 *                     Neighborhood Stressmark specification.
 */
Image *createImage(int dimension,      /* dimension of image */
                   Pixel maxPixel,     /* max legal image value */
                   int numberLines,    /*num line segments in image*/
                   int minThickness,   /*min line segment thickness*/
                   int maxThickness)   /*max line segment thickness*/
{ /* beginning of createImage() */

  Pixel    *image;       /* pointer to image structure to allocate */
  int      i;            /* loop index variable */

  Coord    startPoint;   /* starting point for a line segment */
  Coord    endPoint;     /* ending point for a line segment */
  int      thickness;    /* thickness for a line segment */
  int      startValue;   /* starting value for a line segment */
  int      endValue;     /* ending value for a line segment */

  /*
   * Allocate and clear the memory space for the image.
   */
  image = (Pixel *)malloc(sizeof(Pixel) * dimension * dimension);
  assert(image != NULL);
  for (i = 0; i < dimension*dimension; i++) {
      image[i] = 0;
  } /* end for loop */

  /*
   * Populate the image with line segments where the endpoints, values,
   * and thickness of each line segment are randomly generated.
   */
  for (i = 0; i < numberLines; i++) {
      unsigned int temp;

      temp = randomUInt(0, dimension * dimension - 1);
      startPoint->row = (int) temp / dimension;
      startPoint->col = (int) temp % dimension;
      temp = randomUInt(0, dimension * dimension - 1);
      endPoint->row = (int) temp / dimension;
      endPoint->col = (int) temp % dimension;
      thickness = randomUInt(minThickness,maxThickness);
      startValue = randomUInt(MIN_PIXEL, maxPixel);
      endValue = randomUInt(MIN_PIXEL, maxPixel);
      drawLineSegment(image, startPoint, endPoint,
                      startValue, endValue, thickness);
  } /* end loop for line segments */

  return(image);
} /* end createImage() */
```

```
/*
 * drawLineSegment()This routine draws a line segment into the given
 * image starting at the coordinate startPoint and ending at the
 * coordinate endPoint. The values along this line segment
 * vary linearly from the given starting value startValue to
 * the end value endValue and have a thickness associated to it.
*
 * This routine is modeled after the routine draw_line() from
 * "Practical Computer Vision using C", by J.R. Parker, Wiley, 1994,
 * pp 114-116.
 * It has been enhanced to have values on the line segment vary and to
 * have a thickness to the line segment. The value along the line
 * segment is kept in a float variable so that fractional increments
 * along the line segment can accumulate to integer values.
 */
void drawLineSegment(Image image,    /* image to be modified */
                 Coord startPoint,  /* line segment start point*/
                 Coord endPoint,    /* line segment end point */
                 int   startValue,   /* line segment start value */
                 int   endValue,     /* line segment end value */
                 int   thickness)    /* line segment thickness */
{ /* beginning of drawLineSegment() */
  int   changeColumn, changeRow;/* change along line seg in col/row */
  int   delta;              /* increment alone line segment */
  int   column, row;        /* column&row of coord along line seg */
  float value, valueDelta;   /* value & value change along line seg */

  changeColumn = endPoint.column - startPoint.column;
  changeRow = endPoint.row - startPoint.row;
  assert((changeRow != 0) || (changeColumn != 0));
  column = startPoint.column;
  row    = startPoint.row;
  value  = startValue;

  if (ABS(changeColumn) > ABS(changeRow)) { /* col dominant */
    valueDelta = ((float) endValue - startValue) /
                 ((float) ABS(changeColumn));
    delta = 2*absRow - absColumn;
    for (column = startPoint.column;
         column == endPoint.column+sign(changeColumn);
         column += sign(changeColumn)) {
        for (t = MAX(0, row - thickness/2);
             t < MIN(dimension, row + thickness - thickness/2);
             t++)
             image[t*dimension + column] = (int)value;
        value += valueDelta;
        if (delta >= 0) {
           row += sign(changeRow);
           delta -= 2*ABS(changeColumn);
        } /* end if delta >= 0 */
        column += sign(changeColumn);
        delta += 2*ABS(changeRow);
    } /* end for loop */
  } /* end col dominant case */

  else {                  /* row dominant case */
    valueDelta = ((float) endValue - startValue) /
                 ((float) ABS(changeRow));
    delta = 2 * absColumn - absRow;
    for (row = startPoint.row;
         row == endPoint.row + sign(changeRow);
         row += sign(changeRow)) {
        for (t = MAX(0, column - thickness/2);
             t < MIN(dimension, row + thickness - thickness/2);
             t++)
             image[row*dimension + t] = (int)value;
        if (delta >= 0) {
           column += sign(changeColumn);
           delta -= 2*ABS(changeRow);
        } /* end if delta >= 0 */
        row += sign(changeRow);
        delta += 2*ABS(changeColumn);
    } /* end for loop */
  } /* end column dominant case */

  return;
} /* end of drawLineSegment() */
```

```
/*
 * neighborhoodCalculation() This routine performs the neighborhood
 * stressmark calculating the GLCM entropy and energy for various
 * distance and directions (16 in total).
 */
void neighborhoodCalculation
           (Pixel        *image,        /* input image */
            int          dimension,     /* image dimension */
            int          distanceShort, /* short distance */
            int          distanceLong,  /* long distance */
            Neighborhood *neighborhood) /* descriptor struct*/
{ /* beginning of neighborhoodCalculation() */
  int *sumHist,*diffHist;/* Histograms */

  /*
   * Set up required for histogram calculations.
```

```
    */
   numBins = (2 * (maxPixel- MIN_PIXEL + 1) - 1);
   sumHist = (int *)malloc(numBins * sizeof(int));
   assert(sumHist != NULL);
   diffHist = (int *)malloc(numBins * sizeof(int));
   assert(diffHist != NULL);

   /*
    * Set up each distance, distanceShort and distanceLong, at each
    * each direction, 0, 45, 90, 135 degrees.  CalculateEnergyEntropy()
    * does histogram and descriptor calculation.
    */

   /* for short, 0 degree direction (horizontal) */
   calcEntropyEnergy(sumHist, diffHist, image, numBins,
                     distanceShort, 0,
                     neighborhood->distShort.deg0.entropy,
                     neighborhood->distShort.deg0.energy);

   /* for short, 45 degree direction (right diagonal) */
   calcEntropyEnergy(sumHist, diffHist, image, numBins,
                     distanceShort, distanceShort,
                     neighborhood->distShort.deg45.entropy,
                     neighborhood->distShort.deg45.energy);

   /* for short, 90 degree direction (vertical) */
   calcEntropyEnergy(sumHist, diffHist, image, numBins,
                     0, distanceShort,
                     neighborhood->distShort.deg90.entropy,
                     neighborhood->distShort.deg90.energy);

   /* for short, 135 degree direction (left diagonal) */
   calcEntropyEnergy(sumHist, diffHist, image, numBins,
                     -distanceShort, distanceShort,
                     neighborhood->distShort.deg135.entropy,
                     neighborhood->distShort.deg135.energy);

   /* for long, 0 degree direction (horizontal) */
   calcEntropyEnergy(sumHist, diffHist, image, numBins,
                     distanceLong, 0,
                     neighborhood->distLong.deg0.entropy,
                     neighborhood->distLong.deg0.energy);

   /* for long, 45 degree direction (right diagonal) */
   calcEntropyEnergy(sumHist, diffHist, image, numBins,
                     distanceLong, distanceLong,
                     neighborhood->distLong.deg45.entropy,
                     neighborhood->distLong.deg45.energy);

   /* for long, 90 degree direction (vertical) */
   calcEntropyEnergy(sumHist, diffHist, image, numBins,
                     0, distanceLong,
                     neighborhood->distLong.deg90.entropy,
                     neighborhood->distLong.deg90.energy);

   /* for long, 135 degree direction (left diagonal) */
   calcEntropyEnergy(sumHist, diffHist, image, numBins,
                     -distanceLong, distanceLong,
                     neighborhood->distLong.deg135.entropy,
                     neighborhood->distLong.deg135.energy);


   /* clean up */
   free(sumHist);
   free(diffHist);

   return;
```

```
} /* end neighborhoodCalculation() */
```

```
/*
 * calcEntropyEnergy() This routine calculates the GLCM entropy and
 * energy for a given vector (defined through dx
 * and dy) of the given image.
 */
#include <math.h>       /* define log() */

void calcEntropyEnergy(
     int *sumHist,             /* sum histogram; assumed uninitialized */
     int *diffHist,            /* diff histogram; assumed uninitialized */
     Image     *image,         /* input image */
     Int numBins,              /* number of bins in histograms */
     int       dx,             /* column vector-define dist&direction*/
     int       dy,             /* row vector-define dist&direction */
     float     *entropy,       /* GLCM entropy for dist&direction */
     float     *energy)        /* GLCM energy for dist&direction */

{ /* beginning of calcEntropyEnergy() */
  int    index;                  /* counter used to init histogram */
  int    totalNumPixels;          /* count of number of pixels in hist*/
  int    rowIndex;                /* row loop index */
  int    rowLow, rowHigh;        /* row loop boundaries */
  int    columnIndex;             /* column loop index */
  int    columnLow, columnHigh;   /* column loop boundaries */
  int    columnForPixelAtDistance;/* column of pixel at dist&direction*/
  int    rowForPixelAtDistance;   /* row of pixel at dist&direction */
  int    value0RowOffset;         /* value0 row offset */
  int    value1RowOffset;         /* value1 row offset */

  /*
   * Initialize output values
   */
  *entropy = 0.0;
  *energy  = 0.0;

  /*
   * Initialize the histogram arrays to zero.
   */
  for (index = 0; index < numBins; index++) {
    sumHist[index] = 0;
    diffHist[index] = 0;
  } /* end for loop */

  /*
   * Calculate the sum and difference histograms for an image, where
   * the histograms are incremented for each pixel that is part of the
   * image with a corresponding pixel (dx, dy) away that is also part
   * of the image. For all pixels satisfying these conditions, calculate
   * the sum and difference and add to the histograms. Note that the sum
   * and difference values need to be offset to have a positive index
   * into the histograms.
   */
  /* narrow down loops to cover area with both pixel locations */
  if (dy < 0) {
    rowLow  = - dy;
    rowHigh = dimension;
  } /* end if dy < 0 */
  else { /* dy >= 0 */
    rowLow = 0;
    rowHigh = dimension - dy;
  } /* end dy >= 0 */
  if (dx < 0) {
    columnLow  = - dx;
    columnHigh = dimension;
  } /* end if dx < 0 */
  else { /* dx >= 0 */
    columnLow = 0;
    columnHigh = dimension - dx;
  } /* end dx >= 0 */

  totalNumPixels  = 0;
  value0RowOffset = rowLow * dimension;
  value1RowOffset = (rowLow+dy) * dimension;
  for (rowIndex = rowLow; rowIndex < rowHigh; rowIndex++) {
    for (columnIndex = columnLow; columnIndex < columnHigh;
         columnIndex++) {
      int value0;   /* pixel value at one endpoint of distance ray */
      int value1;   /* pixel value at other endpoint of distance ray */
      int binIndex; /* index of bin in histogram */

      rowForPixelAtDistance    = rowIndex + dy;
      columnForPixelAtDistance = columnIndex + dx;

      /* use a row offset to avoid the multiplication */
      value0 = *(image->data + value0RowOffset + columnIndex);
      value1 = *(image->data + value1RowOffset +
               columnForPixelAtDistance);
      binIndex = value0 + value1 - 2 * MIN_PIXEL;
      assert((binIndex >= 0) && (binIndex < histParms->numBins));
      sumHist[binIndex] += 1;
      binIndex = value0 - value1 + maxPixel – MIN_PIXEL;
      assert((binIndex >= 0) && (binIndex < histParms->numBins));
      diffHist[binIndex] += 1;
      totalNumPixels += 1;
    } /* end for column loop */
    value0RowOffset += dimension; /* advance to next row */
    value1RowOffset += dimension;
  } /* end for row loop */

  /*
   * Normalize sumHist and diffHist (for non-zero bins)
   * and calculate descriptors from sumHist and diffHist
   */
  if (totalNumPixels > 0) {   /* there are values in histograms */
    int    index;             /* loop index */
    double energySum;         /* energy contribution from sumHist */
    double energyDifference;  /* energy from differenceHist*/
    double entropyValue;      /* double version of entropy value */
    double sumNormalized;     /* normalized sum */
    double diffNormalized;    /* normalized diff */
    double scale;             /* normalization scale */

    energySum        = (double) 0;        /* initialize values */
    energyDifference = (double) 0;
    entropyValue     = (double) 0;
    scale            = 1.e0 / totalNumPixels;
    for (index = 0; index < numBins; index++) {
      /*
       * If sumHist = 0 or diffHist = 0, then there is
       * no contribution for that element in the histogram, only
       * positive values have a contribution. Then never take log(zero)
       * so no need to check for this case. Normalize non-zero histogram
       * bins before calculations.
       */
      if (sumHist[index] > 0) {
        sumNormalized = (double) sumHist[index] * scale;
        entropyValue  = entropyValue - sumNormalized *
                        log((double) sumNormalized);
        energySum     = energySum + sumNormalized * sumNormalized;
      } /* end sumHist bin positive */
      if (diffHist[index] > 0) {
        diffNormalized   = (double)diffHist[index] * scale;
        entropyValue     = entropyValue -
                           diffNormalized * log(diffNormalized);
        energyDifference = energyDifference +
                           diffNormalized * diffNormalized;
      } /* end diffHist bin positive */
    } /* end for loop for index */
    *energy  = energySum * energyDifference;
    *entropy = entropyValue;
  } /* end if there are values in histograms */

  return;
} /* end calcEntropyEnergy() */
```

6.5 Notes

Note the following items relating to the *Neighborhood Stressmark*:

- When generating the histograms, both pixels of a pair must be within the defined boundaries of the image. Pixels without correspondent neighbors at the appropriate distance and direction offset should be ignored. Note that a given pixel may have certain neighbors within bounds, and others beyond; thus, slightly different sets of source pixels will be utilized for each combination of direction and distance offsets.
- Much of the code presented in section 6.4 relates to image data generation, rather than the processing of interest to this stressmark. It is presumed that for many participants, a more expedient method is to generate the input imagery offline, and download it to the candidate system, which manipulates it appropriately.

7. Field Stressmark

The *Field Stressmark* emphasizes regular access to large quantities of data. It involves scanning for strings, possibly with fine-grain parallelism. In this way, it tests a system's ability to perform on searches when indices are unavailable or inadequate.

The *Field Stressmark* consists of searching an array (field) of random words for token strings, which are used as delimiters. All words between instances of the delimiter form a sample set, from which simple statistics are collected. The delimiters themselves are updated in memory. When all instances of a token are found, the process is repeated with a new one. The statistics for each sample set are reported.

7.1 Input

Input consists of a single ASCII file containing all the parameters required for a single run. The table below gives the format and description of each item. Items are given in the order provided, and are delimited by white space. Formats are given using C standard library *scanf()* control notation. See Section 8.2 for more details on item usage.

Item Number	Description	Format	Limits
1	Size of field, *f*, in words.	%ld	$2^4 \leq val \leq 2^{24}$
2	Seed for random number generator.	%ld	$1-2^{31} \leq val \leq -1$
3	Offset value for token modifier, *mod_offset*. This value is the number of words between a found token word and the word that should be used to modify it. See Section 8.2 for more information.	%ld	$2^0 \leq val \leq 2^{16}$
4	Number of tokens, *n*.	%ld	$2^0 \leq val \leq 2^8$
$5 + i$, $0 \leq i < n$	The *i*th token, given as a zero-terminated string of hexadecimal values.	%x […%x] 0	Each element of string: $0 \leq val < 2^8$ Length of string: $1 \leq val < 2^3$

7.2 Algorithm

The general procedure for this stressmark is as follows:

Step	Action
1	Read the input file.
2	Initialize the random number generator and fill the field with random integers of the range $[0…2^8-1]$.
3	Start the timer.

4 Get the next token from the input list.

5 Search through the field, looking for an instance of the token. Note that the first subfield *ends* at the first instance of the token.

<u>At the beginning of the field:</u>

(a) Set the count of the number of subfields to one.
(b) Initialize the *count* and *sum* accumulators to zero. Initialize the *minimum* accumulator to a value greater than or equal to 2^8-1.

<u>At each token instance:</u>

(a) Store the values from the *count*, *sum*, and *minimum* accumulators for subsequent output.
(b) Modify the delimiter in the field by adding the value of one new word to each word in the string, discarding the overflow of each word. The words to be added are located by finding the sum of the token modifier offset and the index of the delimiter words, using modulo arithmetic. I.e., F[*x*]+=F[(*x*+*y*)%f]. Thus, each time an instance of the token is found and used as a delimeter, it is modified by summing with another string within the field.
(c) Increment the count of the number of subfields by one.
(d) Initialize the *count*, and *sum* accumulators to zero. Initialize the *minimum* accumulator to a value greater than or equal to 2^8-1.
(e) Proceed with the search, beginning at the next word that is not part of the delimiter.

<u>For each word encountered that is not a part of a token instance:</u>

(a) Increment the *count* accumulator by one.
(b) Increment the *sum* accumulator by the value of the current word, discarding the overflow.
(c) If the value of the current word is less than that of the *minimum* accumulator, set the value of the *minimum* accumulator to that of the current word.

<u>At the end of the field:</u>

(a) Store the values from the *count*, *sum*, and *minimum* accumulators for subsequent output.
(b) Proceed to step 6.

6 Repeat steps 4 and 5 until the input list of tokens is exhausted (total of *n* times).

7 Stop the timer.

8 Write the output and metrics files.

7.3 Output

The program should return an ASCII file containing three white-space-delimited integers for each subfield (i.e., the number of tokens found, plus one), represented in the order found. The three integers should represent the values from the *count*, *sum*, and *minimum* accumulators for the corresponding subfield.

7.4 Sample Code

Sample code for the *Field Stressmark* is provided here. This source code, like all source code found within this document, is provided for example purposes only.

```
/*
 * Sample code for the DIS Field Stressmark
 *
 * Provided by Atlantic Aerospace Division, Titan Systems Corporation, 2000.
 *
 * This code is intended to serve as an example only.  It is not expected
 * to be particularly clean, efficient, user-friendly, or portable.  This
 * code is untested.  Please report discrepancies between this sample code
 * and the rest of the specifications to <dis@aaec.com>.
 *
 * Note in particular that most error checking is done using macros
 * that are only compiled into debugging versions.  This saves a small
 * amount of execution time and memory, but caution is required.  At
 * the very least, only run the final version using input files that
 * have previously and successfully been processed by the debug version.
 */


#include <stdio.h>                  /* needed for file I/O */
#include <time.h>                   /* needed for metrics collection */
#include <assert.h>                 /* needed for debugging trap/macro */
#include <DISstressRNG.h>           /* needed for random numbers */

/*
 * Limits on input values as defined in the specifications.
 */
#define MIN_FIELD_SIZE 16           /* Minimum size of field, in words. */
#define MAX_FIELD_SIZE 16777216     /* Maximum size of field, in words. */
#define MIN_SEED -2147483647        /* Minimum value for RNG seed. */
#define MAX_SEED -1                 /* Maximum value for RNG seed. */
#define MIN_MOD_OFFSET 0            /* Minimum value for modifier offset */
#define MAX_MOD_OFFSET 65535        /* Maximum value for modifier offset */
#define MIN_TOKENS 1                /* Minimum number of threads. */
#define MAX_TOKENS 256              /* Maximum number of threads. */
#define MIN_TOKEN_LENGTH 1          /* Minimum length of token string. */
#define MAX_TOKEN_LENGTH 8          /* Maximum length of token string. */
#define MIN_TOKEN_VALUE 0           /* Minimum value of word in token. */
#define MAX_TOKEN_VALUE 255         /* Maximum value of word in token. */
#define MAX_SUBFIELDS 256           /* Maximum number of subfields that
                                       need to be searched and reported. */

/*
 * main()
 */
int main() {

  unsigned char *field;             /* The field of values. */
  unsigned int f;                   /* Size of the field. */
  int seed;                         /* Seed for random number generator. */
  int mod_offset;                   /* Offset value for modifying tokens
                                       when they are found. */
  unsigned int n;                   /* Number of tokens. */
  time_t startTime;                 /* Clock at start of hopping. */
  struct tokenS {
    unsigned char delimiter[MAX_TOKEN_LENGTH];     /* Token string. */
    unsigned char length;           /* Length of token string. */
    struct statisticS {
      unsigned int count;           /* Number of words in subfield. */
```

```
        unsigned char min;           /* Minimum value of words in subfield*/
        unsigned char sum;           /* Lowest 8-bits of sum of words in
                                        subfield. */
      } stat[MAX_SUBFIELDS];         /* Statistics for each subfield. */
      unsigned char subfields;       /* Count of number of subfields. */
    } token[MAX_TOKENS];             /* Array of tokens and their results */
    unsigned int l;                  /* Loop index. */

    /*
     * Read the input file.  First read the fundamental information, and
     * check it against the specifications.  Then, fill the token array.
     */
    fscanf(stdin,"%d %d %d %d",
       &f, &seed, &mod_offset, &n);  /* read the fundamentals */
    assert((f >= MIN_FIELD_SIZE) && (f <= MAX_FIELD_SIZE));
    assert((seed >= MIN_SEED) && (seed <= MAX_SEED));
    assert((mod_offset >= MIN_MOD_OFFSET) && (mod_offset <= MAX_MOD_OFFSET));
    assert((n >= MIN_TOKENS) && (n <= MAX_TOKENS));
    for (l=0; l<n; l++) {            /* For each delimiting token string */
        int x;                       /* Gets input words for conversion.*/
        int index;                   /* Index for element in string. */
        index = 0;
        fscanf(stdin,"%x",&x);
        while (x != 0) {
           assert((x >= MIN_TOKEN_VALUE) && (x <= MAX_TOKEN_VALUE));
           token[l].delimiter[index] = (unsigned char)x;
           index++;
           fscanf(stdin,"%x",&x);
        } /* end of while */
        assert((index >= MIN_TOKEN_LENGTH) && (index <= MAX_TOKEN_LENGTH));
        token[l].length = index;
    } /* end of loop for l */

    /*
     * Now create the field, using the provided random number generator.
     */
    if ((field = (unsigned char *)malloc(f*sizeof(unsigned char))) == NULL)
        return(-1);
    initRand(seed);
    for (l=0; l<f; l++) {
        field[l] = randInt(MIN_TOKEN_VALUE,MAX_TOKEN_VALUE);
    } /* end of loop for l */

    /*
     * Start the timer.
     */
    startTime = time(NULL);

    /*
     * Loop through each token.  For each one, search through the field
     * for the token, updating the statistics as you go.  When a token
     * is found, store the statistics, modify the token, and keep going
     * until the end of the field.
     */
    for (l=0; l<n; l++) {
        unsigned int index;          /* Index into field. */

        token[l].subfields = 0;
        token[l].stat[0].count = 0;
        token[l].stat[0].sum = 0;
        token[l].stat[0].min = MAX_TOKEN_VALUE;
        index = 0;
        while ((index < f) && (token[l].subfields < MAX_SUBFIELDS)) {
        unsigned char offset;        /* So we can look ahead at string. */
        offset = 0;
        while ((field[index+offset] == token[l].delimiter[offset]) &&
           (offset < token[l].length)) {
```

```
          offset++;
          } /* end of while matching */
      if (offset == token[l].length) {      /* token found! */
          for (offset=0; offset<token[l].length; offset++) {
              field[index+offset] = (field[index+offset] +
                                     field[(index+offset+mod_offset)
                                      % f])
                                    % (MAX_TOKEN_VALUE+1);
          } /* end of loop for offset */
          index += token[l].length-1;
          token[l].subfields++;
          token[l].stat[token[l].subfields].count = 0;
          token[l].stat[token[l].subfields].sum = 0;
          token[l].stat[token[l].subfields].min = MAX_TOKEN_VALUE;
      } /* end of if token has been found */
      else {                  /* no token found */
          token[l].stat[token[l].subfields].count++;
          token[l].stat[token[l].subfields].sum += field[index];
          if (token[l].stat[token[l].subfields].min > field[index])
          token[l].stat[token[l].subfields].min = field[index];
      } /* end of else (no token found) */
      index++;
    } /* end of while index less than f */
      token[l].subfields++;
  } /* end of loop for l */

  /*
   * Stop the timer.  Find the difference between the current time and
   * the starting time, to be reported as the total time.
   */
  startTime = time(NULL) - startTime;

  /*
   * Write out the results and metrics files.  The output consists of
   * the number of hops for each thread.  The metric reported here
   * is total time for completion.
   */
  for (l=0; l<n; l++) {
      unsigned int ll;
      fprintf(stdout,"%d subfields for token %d\n",token[l].subfields,l);
      for (ll=0; ll<token[l].subfields; ll++)
      fprintf(stdout,"subfield %d:\tcount= %d\tmin= %x\tsum= %x\n",
          ll,token[l].stat[ll].count,
          token[l].stat[ll].min,token[l].stat[ll].sum);
  } /* end of loop for l */
  fprintf(stdout,"Total time = %d seconds.\n",(int)startTime);
  free(field);
  return(0);

  } /* end of main() */
```

7.5 Notes

Note the following items relating to the *Field Stressmark*:

- Instance limits. Only statistics on words up to the 256th instance of any one delimiter need be searched and reported.
- Word size. For this stressmark, a *word* must represent a memory unit large enough to contain integer values of the range $[0\ldots2^8)$ (i.e., a byte).
- Input limits. Implementers should strive to develop code that works properly for all legal input values. With regard to the size of the field, *f*, it is understood that large

values may not be supported by some systems. Please report which tests could not be performed as a result of this limitation.

- Statistical sample. All words that are not part of any delimiter should be included in the sample set.
- Subfields. The beginning and end of the field also delimit subfields. So, the first subfield ends at the first instance of the delimiter, and the last subfield ends at the terminus of the field. Therefore, *d* delimiters define *d*+1 subfields.

8. Corner-Turn Stressmark

The *Corner-Turn Stressmark* emphasizes effective memory bandwidth without stressing functional units. It involves the matrix transposition ("corner-turn") operation useful in signal processing applications. Although matrix transposition is a required element in other stressmarks and benchmarks within this suite, this stressmark involves practically no computation, so memory bandwidth issues are not readily masked behind processing latency.

The *Corner-Turn Stressmark* consists of transposing a matrix of random words repeatedly. As there is no specific computation, there is no required output.

If the candidate architecture employs multiple computation nodes, the stressmark should be executed using a variety of combinations of these nodes, so the relationships between configuration, problem size, and performance may be studied.

The *Corner-Turn Stressmark* has both *in-place* and *out-of-place* modes, referring to whether the transposed matrix must overwrite the original, or exist as a copy (on different computational nodes if multiple nodes are utilized).

8.1 Input

Input consists of a single ASCII file containing all the parameters required for a single run. The table below gives the format and description of each item. Items are given in the order provided, and are delimited by white space. Formats are given using C standard library *scanf()* control notation. See Section 8.2 for more details on item usage.

Item Number	Description	Format	Limits
1	Row dimension, *x*, of matrix, in words.	%ld	$2^4 \le val \le 2^{15}$
2	Column dimension, *y*, of matrix, in words.	%ld	$2^4 \le val \le 2^{15}$
3	Seed for random number generator.	%ld	$1-2^{31} \le val \le -1$
4	Number of times to transpose matrix, *n*.	%ld	$2^0 \le val \le 2^{16}$
5	Flag indicating operating mode. 0=*in-place*; 1=*out-of-place*.	%h	$0 \le val \le 1$

8.2 Algorithm

The general procedure for this stressmark is as follows:

Step	Action
1	Read the input file.
2	Create a matrix of y rows and x columns, stored in row-major order and evenly distributed across computation nodes. Each element of the matrix should consist of at least one word large enough to contain integer values of the range $[0 \ldots 2^{32}-1]$.

3	Initialize the random number generator and fill the matrix in row-major order with random integers of the range [0…2^{32}-1].
4	If in *out-of-place* mode, create a second matrix of *x* rows and *y* columns, to be used as a destination matrix.
5	Start the timer.
6	Transpose the input matrix completely.
7	Stop the timer and store results.
8	Repeat steps 5-7 until the matrix has been transposed n times.
9	Write the metrics file. At a minimum, best, worst, and average cases must be reported. A histogram of all results is recommended.

8.3 Notes

Note the following items relating to the *Corner-Turn Stressmark*:

- Time measurement. Be aware that this stressmark measures time of each transpose, rather than total time for all transposes. Thus, it is imperative that the timer used be accurate and precise enough for measuring short-duration events.
- Input limits. Implementers should strive to develop code that works properly for all legal input values. With regard to the size of the matrix, *x*•*y*, it is understood that large values may not be supported by some systems. Please report which tests could not be performed as a result of this limitation.
- Multiple computation nodes. It is desirable to study the relationships between configuration, problem size, and performance. So, for candidate architectures that employ multiple computation nodes, the stressmark should be executed for each of a variety of configurations (e.g., using 1 node, 2 nodes, 4 nodes, etc.).

9. Transitive Closure Stressmark

The *Transitive Closure Stressmark* emphasizes semi-regular access to elements in multiple matrices concurrently. It requires solution of the *all-pairs shortest path* problem, which is fundamental to a variety of computational problems.

The *Transitive Closure Stressmark* utilizes the Floyd-Warshall all-pairs shortest path algorithm[11]. It accepts an adjacency matrix of a directed graph as input, and uses a recurrent relationship to produce the adjacency matrix of the shortest-path transitive closure. It runs in $O(n^3)$ time, which asymptotically is no better than n calls to Dijkstra's single-source shortest-paths algorithm[11]. However, this approach is generally considered to operate better in practice than Dijkstra's, especially when adjacency matrices (as opposed to lists) are employed.

9.1 Input

Input consists of a single ASCII file containing all the parameters required for a single run. The table below gives the format and description of each item. Items are given in the order provided, and are delimited by white space. Formats are given using C standard library *scanf()* control notation. See Section 9.2 for more details on item usage.

Item Number	Description	Format	Limits
1	Number of vertices, n, in words.	%ld	$2^3 \leq val \leq 2^{14}$
2	Number of edges, m, in words.	%ld	$0 \leq val \leq n^2$
3	Seed for random number generator.	%ld	$1-2^{31} \leq val \leq -1$

9.2 Algorithm

The general procedure for this stressmark is as follows:

Step	Action
1	Read the input file.
2	Create an *nxn* adjacency matrix, D. Each element must be able to represent discrete values of the range $[0\ldots2^{31}-1]$. Initialize all matrix elements to $2^{31}-1$.
3	Initialize the random number generator. Generate m random integer triples $\{x_i, y_i, z_i\}$, where x_i and y_i are in the range $[1\ldots n]$, and z_I is in the range $[0\ldots2^8-1]$. Use the triples to initialize the adjacency matrix D of the directed graph, by using each x_I as a starting vertex, each y_I as an ending vertex, and each z_i as the length of the edge. (I.e., $D_{x,y}=z$.)
4	Start the timer.
5	For each $k=[1\ldots n]$, let $D^k_{i,j}=\min(D^{k-1}_{i,j}, D^{k-1}_{i,k}+D^{k-1}_{k,j})\ \forall\ i, j\in[1\ldots n]$. Note: This step requires additional temporary storage for D; if desired, storage space may be allocated and initialized to contain the value $2^{31}-1$

	during step 2.
6	Stop the timer.
7	Calculate the sum of each row and column of D^n, ignoring any matrix elements containing the value 2^{31}-1.
8	Write the output and metrics files.

9.3 Output

The program should return an ASCII file containing 2*n* white-space-delimited integers. The values should represent the sums of rows 1…*n* of D^n, and the sums of columns 1…*n* of D^n, respectively.

9.4 Sample Code

Sample code for the *Transitive Closure Stressmark* is provided here. This source code, like all source code found within this document, is provided for example purposes only.

```
/*
 * Sample code for the DIS Transitive Closure Stressmark
 *
 * Provided by Atlantic Aerospace Division, Titan Systems Corporation, 2000.
 *
 * This code is intended to serve as an example only.  It is not expected
 * to be particularly clean, efficient, user-friendly, or portable.  This
 * code is untested.  Please report discrepancies between this sample code
 * and the rest of the specifications to <dis@aaec.com>.
 *
 * Note in particular that most error checking is done using macros
 * that are only compiled into debugging versions.  This saves a small
 * amount of execution time and memory, but caution is required.  At
 * the very least, only run the final version using input files that
 * have previously and successfully been processed by the debug version.
 */


#include <stdio.h>                  /* needed for file I/O */
#include <time.h>                   /* needed for metrics collection */
#include <assert.h>                 /* needed for debugging trap/macro */
#include <DISstressRNG.h>           /* needed for random numbers */


/*
 * Limits on input values as defined in the specifications.
 */
#define MIN_VERTICES 8              /* Minimum # of vertices, in words. */
#define MAX_VERTICES 16384          /* Maximum # of vertices, in words. */
#define MIN_EDGES 0                 /* Minimum # of edges, in words. */
#define MAX_EDGES 268435456         /* Maximum # of edges, in words. */
#define MIN_SEED -2147483647        /* Minimum value for RNG seed. */
#define MAX_SEED -1                 /* Maximum value for RNG seed. */
#define NO_PATH 2147483647          /* This value in the adjacency
                                       matrix indicates that no path is
                                       present.  It is larger than the
                                       longest possible path so that
                                       in-loop comparisons are minimized.*/
#define MIN_EDGE 0                  /* Minimum edge length. */
#define MAX_EDGE 255                /* Maximum edge length. */

/*
```

```
 * main()
 */
int main() {

  unsigned int *din, *dout;        /* The adjacency matrices; one for
                                      reading and one for writing, they
                                      exchange places on each iteration */
  unsigned int n;                  /* The number of vertices. */
  unsigned int m;                  /* The number of edges. */
  unsigned int i, j, k;            /* Loop variables */
  int seed;                        /* Seed for random number generator. */
  time_t startTime;                /* Clock at start of hopping. */
  unsigned int sum;                /* Accumulates sums for output check */

  /*
   * Read the input file.  Read the fundamental information, and
   * check it against the specifications.
   */
  fscanf(stdin,"%d %d %d",&n,&m,&seed);     /* read the fundamentals */
  assert((n >= MIN_VERTICES) && (n <= MAX_VERTICES));
  assert((m >= MIN_EDGES) && (m <= MAX_EDGES));
  assert(m <= n*n);
  assert((seed >= MIN_SEED) && (seed <= MAX_SEED));

  /*
   * Create the adjacency matrices.  During each iteration of the main
   * loop, din will be used as source information (ie, D(k-1)), and
   * dout will be used for a destination (ie, D(k)).
   *
   * Initialize all matrix elements to 2^31-1 (larger than largest possible
   * path), then go back and set the values representing the graph edges.
   */
  if ((din = (unsigned int *)malloc(n*n*sizeof(unsigned int))) == NULL)
      return(-1);
  if ((dout = (unsigned int *)malloc(n*n*sizeof(unsigned int))) == NULL)
      return(-1);
  for (i=0; i<n*n; i++) {
      *(din + i) = NO_PATH;
      *(dout + i) = NO_PATH;
  } /* end of loop for i */
  initRand(seed);
  for (k=0; k<m; k++) {
      i = randInt(0,n);            /* pick a column */
      j = randInt(0,n);            /* pick a row */
      *(din + j*n + i) = randInt(MIN_EDGE, MAX_EDGE);
  } /* end of loop for k */

  /*
   * Start the timer.
   */
  startTime = time(NULL);

  /*
   * Perform the recurrence.  Note that for each iteration, Din and Dout
   * switch places so that we can reuse the memory.
   */
  for (k=0; k<n; k++) {
      unsigned int old, new;       /* temp for old and new path lengths*/
      unsigned int *dtemp;         /* temp for exchanging din and dout */
      for (i=0; i<n; i++) {
          for (j=0; j<n; j++) {
             old = *(din + j*n + i);
             new = *(din + j*n + k) + *(din + k*n + i);
             *(dout + j*n + i) = (new < old ? new : old);
             assert(*(dout + j*n + i) <= NO_PATH);
             assert(*(dout + j*n + i) <= *(din + j*n + i));
          } /* end of loop for j */
      } /* end of loop for i */
      dtemp = dout;                  /* swap the matrix pointers */
      dout = din;
      din = dtemp;
  } /* end of loop for k */

  /*
   * Stop the timer.  Find the difference between the current time and
   * the starting time, to be reported as the total time.
   */
  startTime = time(NULL) - startTime;

  /*
   * Write out the results and metrics files.  The output consists of
   * the sum of each row and column.  The metric reported here
   * is total time for completion.
   */
  for (j=0; j<n; j++) {             /* loop over rows */
      sum = 0;
      for (i=0; i<n; i++) {
          if (*(din + j*n + i) != NO_PATH)
              sum += *(din + j*n + i);
      } /* end of loop for i */
      fprintf(stdout,"%u\n",sum);
  } /* end of loop for j */
  for (i=0; i<n; i++) {             /* loop over columns */
      sum = 0;
      for (j=0; j<n; j++) {
          if (*(din + j*n + i) != NO_PATH)
              sum += *(din + j*n + i);
      } /* end of loop for j */
      fprintf(stdout,"%u\n",sum);
  } /* end of loop for i */

  fprintf(stdout,"Total time = %u seconds.\n",(unsigned int)startTime);
  free(din);
  free(dout);
  return(0);

  } /* end of main() */
```

9.5 Notes

Note the following items relating to the *Transitive Closure Stressmark*:

- Timer. Timing metrics should be collected with awareness that a significant range of operating times may be encountered.
- Input limits. Implementers should strive to develop code that works properly for all legal input values. With regard to the size of the matrix, *d*, it is understood that large matrices may not be supported by some systems. Please report which tests could not be performed as a result of this limitation.

10. Support Functions

This section gives code for functions that are needed by more than one of the stressmarks in the suite.

10.1 Random Number Generators

The random number generators supplied here are required by all of the stressmarks. In order to minimize the need for transmitted data and large input files, much of the data to be processed by stressmark code is generated randomly. By utilizing a common random number generator, with supplied seeds, consistency of data between participants is guaranteed.

The first two functions, *randInit*()and *randNum*(), comprise the basic random number generator. The former is called to initialize or reset the random number generator, using its one argument as a seed. The latter returns a single uniform random deviate in the range [0…1}. Both are derived from the “minimal” generator of Park and Miller with Bays-Durham shuffle [4].

The remaining functions scale, offset, and truncate the deviate as required by the various stressmarks.

```
/*
 * The following routines is an adaptation of the routine ran1()from
 * Numerical Recipes in C, 2nd Edition, by Press, W.H., Teukolsky, S.A.,
 * Vetterling, W.T., and Flannery, B.P., Cambridge University Press,
 * 1992, page 280
 *
 * “Minimal” random number generator of Park and Miller with Bays-Durham
 * shuffle and added safeguards.  Returns a uniform random deviate
 * between 0.0 and 1.0 (exclusive of the endpoint values).  Call with
 * idum a negative integer to initialize; thereafter, do not alter idum
 * between successive deviates in a sequence.  RNMX should approximate
 * the largest floating value that is less than 1.
 *
 * Pull out the initialization part of ran1() into its own routine
 * randInit() with the rest of ran1() in the routine randNum(). Then
 * rather than ran1() called with a negative idum to initialize, use a
 * call to randInit() with a negative idum to initialize, followed by
 * calls to randNum() thereby eliminating the initialization check
 * within ran1().  Change the input parameter to be a static variable
 * so that the seed be a read-only input required in randInit().
 */

#define IA   16807
#define IM   2147483647
#define AM   (1.0/IM)
#define IQ   127773
#define IR   2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS  1.2e-7
#define RNMX (1.0-EPS)

static long iy=0;
static long iv[NTAB];
static long iseed;

float randInit(long idum)
```

```
{ /* beginning of randInit() - call once before calls to randNum() */
   long        j;
   long        k;

   assert(idum <= 0)
   assert(iy == 0)

   /* Initialize routine randNum() */
   iseed = idum;
   if (-(iseed) < 1) {             /* Be sure to prevent idum=0. */
    iseed=1;
   } /* end if */
   else {
    iseed = - (iseed);
   } /* end else */
   for (j=NTAB+7; j>=0; j--) {     /* Load the shuffle table */
    k = (iseed)/IQ;                /*   (after 8 warm-ups).  */
    iseed = IA*(iseed-k*IQ)-IR*k;
    if (iseed < 0) {
       iseed += IM;
    } /* end if */
    if (j < NTAB) {
       iv[j] = iseed;
    } /* end if */
   } /* end for */
   iy = iv[0];
} /* end randInit() */

float randNum()
{ /* beginning of randNum() - precede with a call to randInit() */
   long        j;
   long        k;
   float       temp;

   assert(iy != 0)

   k = (iseed)/IQ;
   iseed = IA*(iseed-k*IQ)-IR*k; /* Compute iseed = (IA*iseed)%IM     */
   if (iseed < 0) {              /*   without overflows by Schrage's  */
    iseed += IM;                 /*   method.                         */
   } /* end if */
   j = iy/NDIV;                  /* Will be in the range 0..NTAB-1.   */
   iy = iv[j];                   /* Output previously stored value and*/
   iv[j] = iseed;                /*    refill the shuffle table.      */
   if ((temp = AM*iy) > RNMX) { /* Because users don't expect        */
      return RNMX;              /*    endpoint values.               */
   } /* end if */
   else {
    return temp;
   } /* end else */
} /* end randNum() */
```

```
randomFloat(lowest_float, highest_float)
/* Generate a "random" nonzero float in the range [lowest_float,
 * highest_float] exclusive of the endpoints.  (Note that randNum() also
 * returns a value (0,1) exclusive of endpoints.)  If the range contains
 * zero, then zero is a valid value.
 */
Float value

assert(lowest_float < highest_float)

range = highest_float - lowest_float
value = randNum() * (highest_float - lowest_float) + lowest_float
```

```
assert(value > lowest_float)
assert(value < highest_float)

return value
```

```
randomNonZeroFloat(lowest_float, highest_float, epsilon)
/* Generate a "random" nonzero float in the range [lowest_float,
 * highest_float] exclusive of the endpoints.  (Note that randNum() also
 * returns a value (0,1) exclusive of endpoints.)  The value of
 * lowest_float is negative and highest_float is positive.  Eventhough
 * zero is included in the range, all values within the range
 * [-epsilon, epsilon] are not valid return values.
 */
Double range
Float value

assert(lowest_float < 0)
assert(highest_float > 0)
assert(epsilon > 0)
assert((epsilon < -lowest_float) && (epsilon < highest_float))

range = highest_float - lowest_float
value = (randNum() * range) + lowest_float

/* make sure the magnitude is not too close to zero */
if (ABS(value) < epsilon) then
   if (value > 0) then value = value + epsilon
   else if (value < 0) then value = value - epsilon
endif

assert(value > lowest_float)
assert(value < highest_float)

return value
```

```
randomUInt(lowest_uint, highest_uint)
/* Generate a random unsigned integer in the range
 * [lowest_int, highest_int] inclusive of the endpoints.  The "+1" is
 * included in the range calculation to allow for the inclusion of the
 * endpoint integer values (since randNum() returns a value (0,1)
 * exclusive of endpoints).
 */
Float range
Uint value

range = highest_uint - lowest_uint + 1
value = floor(randNum() * range) + lowest_uint

assert(value >= lowest_uint)
assert(value <= highest_uint)

return value
```

11. References

[1] Atlantic Aerospace Electronics Corporation, *Data-Intensive Systems Benchmark Suite: Analysis and Specifications*, Version 1.0, June 30, 1999.

[2] Atlantic Aerospace Electronics Corporation, *DIS Benchmark C Style Guide*, December 28, 1998.

[3] The Boeing Company, *Data-Intensive Systems Benchmark Suite: Design of Method of Moments Benchmark Demonstration Code*, MDA972-97-C 0025, September 1999.

[4] Press, W. H., Teukolsky, S. A., Vetterling, W.T., and Flannery, B. P., *Numerical Recipes in C,* Second Edition, Cambridge University Press, 1992.

[5] Park, S. K., and Miller, K.W. 1988, *Communications of the ACM*, vol. 31, pp. 1192-1201.

[6] Golub, G. H., and Van Loan, C. F., *Matrix Computations,* Second Edition, The John Hopkins University Press, 1989.

[7] Eisenstat, S. C., Gursky, M. C., Schultz, M. H., and Sherman, A. H. 1977, “Yale Sparse Matrix Package”, *Yale University Department of Computer Science Technical Report*, volumes 112 and 114.

[8] Parker, J., *Practical Computer Vision Using C*, Wiley, 1994.

[9] Parker, J., *Algorithms For Image Processing And Computer Vision*, Wiley Computer Publishing, 1997.

[10] Unser, M., “Sum and Difference Histograms for Texture Classification,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-8, 1:118-125, 1986.

[11] Thomas Cormen, Charles Leiserson, and Ronald Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.