SANDIA REPORT

SAND2013- 8752
Unlimited Release
Printed October 2013

HPCG Technical Specification

Michael A. Heroux, Sandia National Laboratories[1]
Jack Dongarra and Piotr Luszczek, University of Tennessee

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

[1] Corresponding Author, maherou@sandia.gov

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2013- 8752
Unlimited Release
Printed October 2013

HPCG Benchmark Technical Specification

Michael A. Heroux
Scalable Algorithm Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-MS 1320

Jack Dongarra
Piotr Luszczek
Electrical Engineering and Computer Science Department
1122 Volunteer Blvd University of Tennessee
Knoxville, TN 37996-3450

Abstract

.
The High Performance Conjugate Gradient (HPCG) benchmark [2] is a tool for ranking computer systems based on a simple additive Schwarz, symmetric Gauss-Seidel preconditioned conjugate gradient solver. HPCG is similar in its purpose to High Performance Linpack (HPL) currently used to rank systems as part of the Top 500 benchmark [1], but HPCG is intended to better represent how today's applications perform.

In this paper we describe the technical details of HPCG: how it is designed and implemented, what code transformations are permitted and how to interpret and report results.

ACKNOWLEDGMENTS

The authors thank the Department of Energy National Nuclear Security Agency for funding provided for this work. We also thank Simon Hammond, Mahesh Rajan, Doug Doerfler and Christian Trott for their efforts to test early versions of HPCG and give valuable feedback.

CONTENTS

1. Introduction 7
2. HPCG Model Problem Description 7
3. HPCG Design 9
4. HPCG Implementation 12
5. HPCG Testing 12
6. Permitted Transformations and Optimizations 12
7. How To Report HPCG Benchmark Results 12
8. FAQs 13
9. Related Work and Future Adaptations 14
10. Summary and Conclusions 15
11. References 16
Distribution 19

This page is intentionally left blank.

1. INTRODUCTION

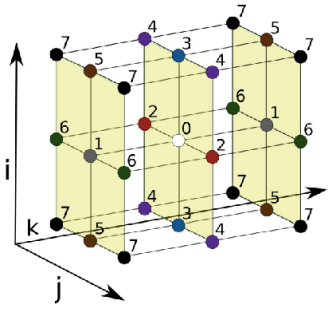
The High Performance Conjugate Gradient (HPCG) benchmark is a simple program that generates a synthetic sparse linear system that is mathematically similar to a finite element, finite volume or finite difference discretizations of a three-dimensional heat diffusion problem on a semi-regular grid. The problem is solved using domain decomposition with an additive Schwarz preconditioned conjugate gradient method where each subdomain is preconditioned using a symmetric Gauss-Seidel sweep. This document provides a technical description of the benchmark and is a companion to *Toward a New Metric for Ranking High Performance Computing Systems* [2].

2. HPCG MODEL PROBLEM DESCRIPTION

The HPCG benchmark generates a synthetic discretized three-dimensional partial differential equation model problem, and computes preconditioned conjugate gradient iterations for the resulting sparse linear system. The model problem can be interpreted as a single degree of freedom heat diffusion model with zero Dirichlet boundary conditions. The global domain dimensions are $(n_x * np_x) \times (n_y * np_y) \times (nx_z * np_z)$ where $(n_x \times n_y \times n_z)$ are the local subgrid dimensions in the *x, y,* and *z* dimensions, respectively, assigned to each MPI process. These values are read from the data file hpcg.dat, or are passed in as command line arguments. The dimensions $(np_x \times np_y \times np_z)$, are a factoring of the MPI process space that is computed automatically in the HPCG setup phase. We impose ratio restrictions on both the local and global *x, y* and *z* dimensions, which are enforced in the setup phase of HPCG.

The setup phase constructs a logically global, physically distributed sparse linear system using a 27-point stencil at each grid point in the 3D domain such that the equation at point *(i, j, k)* depends the values at its location and its 26 surrounding neighbors. The matrix is constructed to be weakly diagonally dominant for interior points of the global domain, and strongly diagonally dominant for boundary points, reflecting a synthetic conservation principle for the interior points and the impact of zero Dirichlet boundary values on the boundary equations. The resulting sparse linear system has the following properties:

- A sparse matrix with 27 nonzero entries per row for interior equations and 7 to 18 nonzero terms for boundary equations.
- A symmetric, positive definite, nonsingular linear operator.
- A generated known exact solution vector with all values equal to 1.0.
- A matching right-hand-side vector.
- An initial guess of all zeros.



27-point stencil operator

The central purpose of defining this sparse linear system is to provide a rich vehicle for executing a collection of important computational kernels embodied in the preconditioned conjugate gradient method shown in Figure 1. However, the benchmark is *not* about computing a high fidelity solution to this problem. In fact iteration counts are fixed in the benchmark code and we do not expect

convergence to the solution, regardless of problem size. We do use the spectral properties of both the problem and the preconditioned conjugate gradient algorithm as part of software testing. See Section 5 for details.

Example of parameter settings: Suppose that we have compiled HPCG with the default MPI and OpenMP modes enabled. Then, assuming a bash Linux terminal window, the following commands from the build/bin directory:

```
export OMP_NUM_THREADS=1
mpiexec –n 96 ./xhpcg 70 80 90
```

will result in:

- $n_x = 70,\ n_y = 80,\ n_z = 90$
- $n_{px} = 4,\ n_{py} = 4,\ n_{pz} = 6$
- Global domain dimensions: 280-by-320-by-540
- Number of equations per MPI process: 504,000
- Global number of equations: 48,384,000
- Global number of nonzeros: 1,298,936,872

Note that changing the value of OMP_NUM_THREADS does not change the problem size or dimensions. It only changes how many threads are active within each MPI process.

CG ALGORITHM

- $p_0 := x_0,\ r_0 := b\text{-}Ap_0$
- Loop $i = 1, 2, \ldots$
  - $z_i := M^{-1} r_{i-1}$
  - if $i = 1$
    - $p_i := z_i$
    - $\alpha_i :=$ `dot_product`(r_{i-1}, z)
  - else
    - $\alpha_i :=$ `dot_product`(r_{i-1}, z)
    - $\beta_i := \alpha_i / \alpha_{i-1}$
    - $p_i := \beta_i * p_{i-1} + z_i$
  - end if
  - $\alpha_i :=$ `dot_product`(r_{i-1}, z_i) / `dot_product`$(p_i, A*p_i)$
  - $x_{i+1} := x_i + \alpha_i * p_i$
  - $r_i := r_{i-1} - \alpha_i * A * p_i$
  - if $\|r_i\|_2 <$ tolerance then Stop
- end Loop

Figure 1: Basic Preconditioned Conjugate Gradient Algorithm

3. HPCG DESIGN

HPCG has a single main program **hpcg/testing/main.cpp.** The flow of execution is shown in Figure 2.

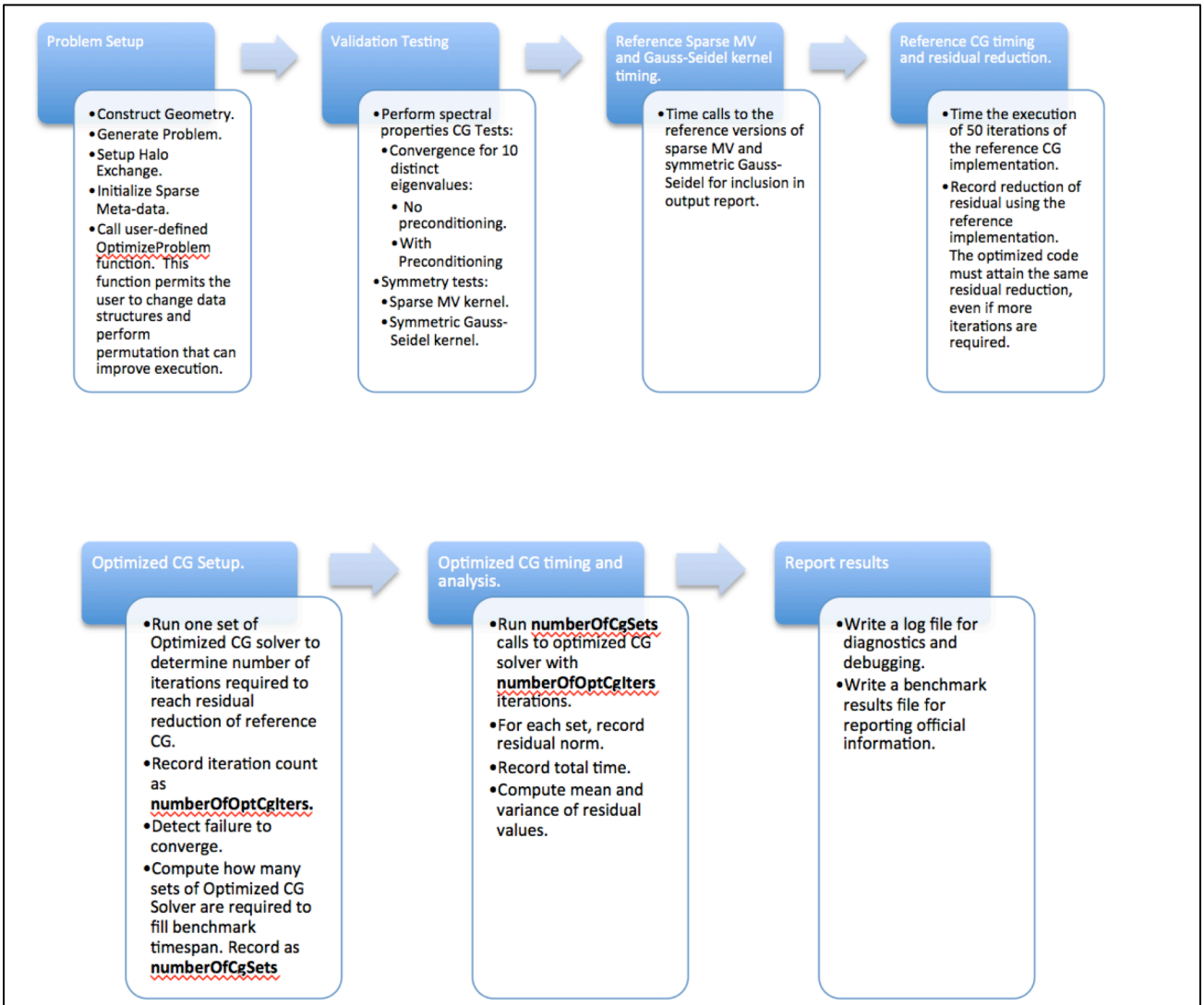


Figure 2: HPCG Execution Phases.

The HPCG code executes the following:

1. **Problem setup:**
   a. Call **GenerateGeometry** to construct the geometry based on the input parameters for the local subdomain size and number of MPI processes as described in Section 2.
   b. Call **GenerateProblem** to generate a synthetic symmetric positive definite (SPD) matrix A using an array-of-pointers-style compressed sparse row format, an exact

solution vector of all 1.0 values, a corresponding right-hand-side vector *b,* and initial guess for *x* of all 0.0 values.

c. Call **SetupHalo** to setup the halo region needed for efficient exchange of off-processor elements prior to computing the sparse matrix-vector product (called in **ComputeSpMV**).
d. Call **InitializeSparseCGData** to set up data structures for sparse matrix multiplication and the local symmetric Gauss-Seidel preconditioner.
e. Call **OptimizeProblem** to execute user-defined optimizations.
    i. Permitted optimizations are limited to:
        1. Changes in sparse matrix data structures that enable better memory access patterns. Such changes expressly **do not** permit elimination of indirect addressing of the input vector for either the **ComputeSPMV** or **ComputeSYMGS** kernels or simplified floating point representations of data elements.
        2. Permutations of the linear system to improve data parallelism.
    ii. Prohibited optimizations:
        1. Any other modifications must first be proposed to the HPCG development team. **Generally speaking, optimizations that circumvent the intention of the benchmark as a driver for common computational kernels are not permitted.**
        2. Although the matrix pattern may be regular, or nearly so, and value-symmetric, matrix storage is to be treated as unstructured and all matrix values are to be retained and used to full precision. The benchmarker is prohibited from exploiting regularity by using, for example, a sparse diagonal format and is prohibited from exploiting value symmetry to reduce storage requirements.
    iii. The time taken for this phase (any optimization in data structure performed) is counted in the final performance measurement. The cost if this phase is added to the cost of executing a single CG iteration set (which is the equivalent residual drop of 50 iterations of the reference CG implementation).

2. Verification and validation testing: In order to assure correct implementation and execution of the optimized version of HPCG, we use the properties of conjugate gradients and the symmetry of the linear operator and preconditioner as validation tests.

a. Spectral tests (**CGtest**):
    i. In this test, we modify the matrix diagonal temporarily such that the first nine diagonals are defined numerically to be (2×10^6, 3×10^6, 4×10^6, …, 10×10^6). All remaining diagonal values are set to 1×10^6. The off-diagonal values are unmodified but are so small that the matrix looks spectrally like a diagonal matrix with 10 distinct diagonal values and therefore 10 distinct eigenvalues.
    ii. By construction, regardless of problem size, the unpreconditioned conjugate gradient algorithm should converge in 11 or 12 iterations.
    iii. Similarly, preconditioned CG using symmetric Gauss-Seidel should converge in one iteration, since the preconditioner has the effect of scaling the diagonal terms to be of the same magnitude.

b. Symmetry tests (**Symtest**):
    i. In this test we confirm the symmetry of the matrix and preconditioner functions by computing two scalar products that are mathematically identical for symmetric operators.
    ii. Using two pseudo-random vectors *x* and *y,* and the user implementations of **ComputeSPMV** to apply the matrix *A* and **ComputeSYMGS** to apply the preconditioner *M* we compute two scalar values:
        1. Departure from symmetry for SPMV: $(x^T Ay - y^T Ax)$.
        2. Departure from symmetry for SYMGS: $(x^T M^{-1} y - y^T M^{-1} x)$.

c. SPMV testing: Using the exact solution vector, we compare the result generated by **ComputeSPMV** with the known RHS vector.

3. **Reference Sparse MV and Gauss-Seidel timing:** We run the reference kernels for use in our output report.
4. **Reference CG timing and residual reduction:** We will report the reference CG timing results in the output. We run the reference CG solver for a fixed number of iterations (50) and record the reduction in the residual. The optimized CG solver must also achieve the same residual reduction even if it requires more iterations.
5. **Optimized CG Setup:** We run the optimized CG solver until it reaches the same residual reduction as the reference CG solver.
    a. The time required to execute this run and the number of iterations required to achieve the residual drop are both recorded.
    b. Using the execution time of a single call to the optimized CG solver (a single *set*), we compute how many sets of runs are required.
    c. The number of iterations required to achieve the required residual drop is called **numberOfOptCgIters.** If the optimized CG does not differ from the reference CG convergence behavior, this value will be 50.
    d. The number of CG sets required to fill the benchmark time requirement is called **numberOfCgSets.**
6. **Optimized CG timing and analysis (Benchmark phase):** We now finally run the benchmark phase of HPCG. Here we run the optimized CG solver **numberOfCgSets** times, and each time run the solver for **numberOfOptCgIters** iterations.
    a. The residual value of each set is recorded as a unique value. At the end of the benchmark phase we compute, analyze and report the mean value of all recorded residuals and the variance.
    b. Small perturbations of the residual are permitted. These can occur because of variations in the order of floating point computations. For example, OpenMP execution of a dot-product typically changes the order of summation and leads to minor (round-off error) perturbations in the final dot-product result.
7. **Post-processing and reporting:** We will report a single timing result, and other metrics.
    a. Computational verification and validation metrics are reported.
    b. Timing and execution rate results are reported.
    c. Pass/fail information is reported, as is an email address where results can be submitted.

4. HPCG REFERENCE IMPLEMENTATION

The reference HPCG code is implemented in C++ using MPI and OpenMP and makes some use of the C++ standard libraries and container classes. While it is certainly possible to write the same functionality in C, many applications rely on high-level C++ features for improved developer productivity. We want HPCG to reflect the language needs of users as part of this benchmark.

Even though standard libraries and containers are used in HPCG, all computations in the benchmark phase are performed using simple for-loops and arrays, which are easily converted to C or Fortran equivalents.

5. HPCG TESTING

HPCG uses basic spectral properties of the conjugate gradient algorithm in order to confirm that the implementation used in the benchmark has expected behavior. In particular, for a matrix with *k* distinct eigenvalues, CG should take *k* iteration to reach convergence, in exact arithmetic. We temporarily make one run where we have a modify the diagonal of our matrix so that there are only 10 distinct values (2×10^6, 3×10^6, 4×10^6, 5×10^6, 6×10^6, 7×10^6, 8×10^6, 9×10^6, 10×10^6). The remaining diagonal values are set to 1×10^6. Although the off-diagonal values remain nonzero and unchanged, they are of magnitude 1.0 and have little influence on the spectral behavior of the linear operator. After performing the spectral tests, we restore the original matrix diagonal values.

For unpreconditioned CG (which is selected by a bool argument to **CG**), we should expect a bit more than 10 iterations to converge. The symmetric Gauss-Seidel preconditioned CG should converge in about 1 iteration, since this preconditioner behaves like Jacobi scaling (only the large diagonal values really matter) and the scaling makes all diagonals appear to be 1.0, so CG should require about 1 iteration.

6. PERMITTED TRANSFORMATIONS AND OPTIMIZATIONS

What can and cannot be changed:

- User is not allowed to change the basic CG algorithm or preconditioner algorithm.
- User can change coding for the preconditioner but must use the same mathematical preconditioner.
- User is allowed to change the coding for **ComputeDOT**, **ComputeWAXPBY**, **ComputeSPMV** and **ComputeSYMGS**.
- User is not allowed to change the matrix data (numerical entries).
- User can change the storage format, but the time to change is recorded and used in the computations of the performance rate.

7. HOW TO REPORT HPCG BENCHMARK RESULTS

All results are recorded in the **HPCG-Benchmark-<identifier>.yaml** file. Instructions for reporting these results are at the end of this file. YAML is a human-readable format that is also convertible to XML and other hierarchical formats, making it easier to manage output as database records.

The Output Results: See generated results after running the benchmark.

8. FAQS

Most of the information in this section can be found in other parts of this document, but we repeat it here for convenience.

1. **The sparsity pattern of the synthetic matrix is really a regular 27-point 3-dimensional stencil pattern. Can I take advantage of this and eliminate the indirect access in the ComputeSPMV and ComputeSYMGS kernels?**

   No. In order to make the HPCG benchmark simple to design, implement and understand, we have used a simple synthetic problem generator. However, you may not explicitly take advantage of this latent structure. Specifically, you *must access vector data indirectly in the ComputeSPMV and ComputeSYMGS kernels. You must also use standard double precision arithmetic for every phase of the computation.*

2. **The matrix in this problem is symmetric. Can I take advantage of symmetry to reduce storage and data access costs?**

   No. Although the problem is symmetric, all computational kernels must use the matrix as though it were non-symmetric.

3. **A mathematically equivalent operator that uses much less storage easily represents the linear operator associated with this matrix. Can I substitute the ComputeSPMV and ComputeSYMGS kernels with mathematically equivalent operators?**

   No. The only permitted optimizations for ComputeSPMV and ComputeSYMGS are permutations that expose greater potential for concurrent execution of the required arithmetic for these kernels. You may also introduce nonzero entries (that have zero values) in order to increase the efficiency of using the sparsity pattern. However, you may not in any way eliminate terms in the matrix.

4. **I have permuted the matrix structure so that the ComputeSYMGS kernel runs faster (better vectorization and more thread parallelism). However, I am now performing more iterations in order to reach the residual drop prescribed by the reference CG solver. Can I count the extra operations as part of my total operation count when computing the final GFLOP/s rating?**

No. The loss of convergence rate due to the transformation reflects the trade-off between parallelism and robustness and is considered part of the overhead cost.

5. **The analysis I perform in OptimizeProblem is fairly expensive. Must it be counted as part of the total execution time in my GFLOP/s rating?**

   Yes. However, this setup cost is added to the total cost of running one set of optimized CG iterations, so the setup cost is amortized.

9. RELATED WORK AND FUTURE ADAPTATIONS

Evolution of HPCG Benchmark

Regardless of which specific benchmark we propose, we expect it to evolve. HPL[3] started as a simple 100-by-100 dense factorization, then a 1000-by-1000, and now places no restrictions on problem size. Furthermore, the algorithms used to compute the factorization have changed dramatically; modified to take advantage of distributed memory, changes in network architecture and multicore CPUs and GPUs. We expect that our new benchmark will adapt to take into account emerging trends in a similar fashion.

Possible Future Extensions

1. Coarse Grid Solve: Presently HPCG has a simple additive Schwarz preconditioner. Realistic preconditioned iterative solvers would have some kind of coarse grid or multilevel solver in order to retain scalability of the solver by keeping iteration counts from inflating too quickly. Furthermore, the presence of a coarse grid puts much more strain on latency-impacting elements of the computer system.

   Although HPCG includes a sparse implicit solver, it is meant as a representative benchmark for a broader class of applications. The computational and communication patterns represented in HPCG cover many types of applications, adding a coarse grid solve makes it much more specialized. Even so, we will monitor the potential value of adding a coarse grid solve to a future version of HPCG.

2. Multicoloring and other reorderings for improved ComputeSYMGS performance: We use a natural ordering of equations for each local subdomain symmetric Gauss-Seidel sweep. This natural ordering tends to have a very restricted resource of thread and vector parallelism, since parallelism is restricted to wave fronts through the domain. Multicoloring orderings can dramatically increase the resource of parallelism, but typically do so at the expense of increasing iteration counts.

   If users evolved to a single approach to improve ComputeSYMGS performance, we may standardize on that ordering.

10. SUMMARY AND CONCLUSIONS

We believe that the HPCG benchmark is an attractive approach to measuring and ranking high-performance computing systems because it contains a small collection of the key computation and communication patterns present in many applications. HPCG is large enough to be mathematically meaningful, yet small enough to easily understand and use.

The HPCG reference implementation provides a starting point and framework for optimizations. Furthermore, it provides mathematical and computational validation and verification testing. Unlike HPL, HPCG will typically not approach the peak performance as measured in GFLOP/s of most computer systems. However, it will stress the most important performance impacting features of a computer system: memory and network bandwidth and latency, system balance and gather/scatter features. These features have a much greater impact on general application performance than the compute-rich dense matrix computations exhibited in HPL.

11. REFERENCES

1. Dongarra, J., et al. *Top 500 Supercomputer Sites*. 1999; Available from: http://www.top500.org.
2. Dongarra, J and Heroux M., *Toward a New Metric for Ranking High Performance Computing Systems*, in *Sandia Report* 2013, Sandia National Laboratories.
3. Dongarra, J., Luszczek, P.,and Petitet, A., *The LINPACK Benchmark: Past, Present, and Future*, Concurrency and Computation: Practice and Experience 15(9):803–820, August 2003, ISSN 1532-0634.

DISTRIBUTION

1	MS0899	Technical Library	9536 (electronic copy)

Sandia National Laboratories