THE UNIVERSITY of TENNESSEE KNOXVILLE

Toward a New Metric for Ranking High Performance Computing Systems[1]

June 10, 2013

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

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

Abstract

The High Performance Linpack (HPL), or Top 500, benchmark [1] is the most widely recognized and discussed metric for ranking high performance computing systems. However, HPL is increasingly unreliable as a true measure of system performance for a growing collection of important science and engineering applications.

In this paper we describe a new high performance conjugate gradient (HPCG) benchmark. HPCG is composed of computations and data access patterns more commonly found in applications. Using HPCG we strive for a better correlation to real scientific application performance and expect to drive computer system design and implementation in directions that will better impact performance improvement.

[1] Also released as: Sandia National Lab; SAND2013-4744

ACKNOWLEDGMENTS

The authors thank the Department of Energy National Nuclear Security Agency for funding provided for this work.

1. INTRODUCTION

The High Performance Linpack (HPL) benchmark is the most widely recognized and discussed metric for ranking high performance computing systems. When HPL gained prominence as a performance metric in the early 1990s there was a strong correlation between its predictions of system rankings and the ranking that full-scale applications would realize. Computer system vendors pursued designs that would increase HPL performance, which would in turn improve overall application performance.

Presently HPL remains tremendously valuable as a measure of historical trends, and as a stress test, especially for leadership class systems that are pushing the boundaries of current technology. Furthermore, HPL provides the HPC community with a valuable outreach tool, understandable to the outside world. Anyone with an appreciation for computing is impressed by the tremendous increases in performance that HPC systems have attained over the past several decades.

At the same time HPL rankings of computer systems are no longer so strongly correlated to real application performance, especially for the broad set of HPC applications governed by differential equations, which tend to have much stronger needs for high bandwidth and low latency, and tend to access data using irregular patterns. In fact, we have reached a point where designing a system for good HPL performance can actually lead to design choices that are wrong for the real application mix, or add unnecessary components or complexity to the system.

We expect the gap between HPL predictions and real application performance to increase in the future. In fact, the fast track to a computer system with the potential to run HPL at 1 Exaflop is a design that may be very unattractive for our real applications. Without some intervention, future architectures targeted toward good HPL performance will not be a good match for our applications. As a result, we seek a new metric that will have a stronger correlation to our application base and will therefore drive system designers in directions that will enhance application performance for a broader set of HPC applications.

2. WHY HPL HAS LOST RELEVANCE

HPL is a simple program that factors and solves a large dense system of linear equations using Gaussian Elimination with partial pivoting. The dominant calculations in this algorithm are dense matrix-matrix multiplication and related kernels, which we call Type 1 patterns. With proper organization of the computation, data access is predominantly unit stride and is mostly hidden by concurrently performing computation on previously retrieved data. This kind of algorithm strongly favors computers with very high floating-point computation rates and adequate streaming memory systems.

While Type 1 patterns are commonly found in real applications, additional computations and access patterns are also very common. In particular, many important calculations, which we call Type 2 patterns, have much lower computation-to-data-access ratios, access memory irregularly, and have fine-grain recursive computations.

A system that is designed to execute both Type 1 and 2 patterns efficiently will generally run a broad mix of applications well. However, HPL only stresses Type 1 patterns and, as a metric, is incapable of measuring Type 2 patterns. With the emergence of accelerators, which are extremely effective with Type 1 patterns relative to CPUs, but much less so with Type 2 patterns, HPL results show a skewed picture relative to real application performance.

For example, the Titan system at Oak Ridge National Laboratory has 18,688 nodes, each with a 16-core, 32 GB AMD Opteron processor and a 6GB Nvidia K20 GPU[2]. Titan was the top-ranked system in November 2012 using HPL. However, in obtaining the HPL result on Titan, the Opteron processors played only a supporting role in the result. All floating-point computation and all data were resident on the GPUs. In contrast, real applications, when initially ported to Titan, will typically run solely on the CPUs and selectively off-load computations to the GPU for acceleration.

Of course, one of the important research efforts in HPC today is to design applications such that more computations are Type 1 patterns, and we will see progress in the coming years. At the same time, most applications will always have some Type 2 patterns and our benchmarks must reflect this reality. In fact, a system's ability to effectively address Type 2 patterns is an important indicator of system balance.

3. REQUIREMENTS

Any new metric we introduce must satisfy a number of requirements. Two overarching requirements are:

1. Accurately predict system rankings for target suite of applications: The ranking of computer systems using the new metric must correlate strongly to how our real applications would rank these same systems.
2. Drive improvements to computer systems to benefit our applications: The metric should be designed so that, as we try to optimize metric results for a particular platform, the changes will also lead to better performance in our real applications. Furthermore, computation of the metric should drive system reliability in ways that help our applications.

We will perform thorough validation testing of any proposed benchmark against a suite of applications on current high-end systems using techniques similar to those identified in the Mantevo project [3]. We will furthermore specify restrictions on changes to the reference version of the code to ensure that only changes that have relevance to our application base are permitted.

4. A PRECONDITIONED CONJUGATE GRADIENT BENCHMARK

As the candidate for a new HPC metric, we consider the preconditioned conjugate gradient (PCG) method with a local symmetric Gauss-Seidel preconditioner (see the primer in the Appendix for more details about PCG).

The reference code will be implemented in C++[2] using MPI and OpenMP. It will do the following:

1. **Problem setup:** Generate a synthetic symmetric positive definite (SPD) matrix *A* (perhaps using several sparsity patterns to match the broad interests of our community) using the compressed sparse row format, and a corresponding right-hand-side vector *b,* and initial guess for *x.*
    a. Linear system size is a parameter that can be chosen via a prescribed process that assures realistic use of the machine resources.
    b. The benchmarker can use a different matrix format and the setup cost in building the new data structure is not counted in the benchmark timing, although the cost will be reported, normalized by the cost of a matrix-vector multiplication operation using the original data structures.
    c. Although the matrix pattern may be regular, or nearly so, and value-symmetric, matrix storage will be unstructured and keep a copy of all matrix values. 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.
2. **Preconditioner setup:** Set up data structures for the local symmetric Gauss-Seidel preconditioner. The reference version will use simple compressed sparse row representation for the lower and upper triangular matrices, each as a separate matrix.
    a. The benchmarker is free to make the same transformations on these matrix objects as in Step 1, again without counting this cost in the benchmark timing, but again the setup time will be reported, normalized by the cost of one symmetric Gauss-Seidel sweep using the original matrix format.
    b. We may need to introduce a simple coarse grid solve as part of the preconditioner, if the performance of a local triangular solve is not sufficiently representative of our real codes.
3. **Verification and validation setup:** We will compute preconditions, post-conditions and invariants that will aid in the detection of anomalies during the iteration phases.
    a. We can compute spectral approximates that bound the error, and use other properties of PCG and SPD matrices to verify and validate results.
    b. We can compute comparison results with reference kernels to assure accurate computation.
4. **Iteration:** We will perform *m* iterations*, n* times, using the same initial guess each time, where *m* and *n* are sufficiently large to test system uptime. By doing this we can compare the numerical results for "correctness" at the end of each *m*-iteration phase.
    a. If the result is not bit-wise identical across successive *m*-iteration phases, we can report the deviation. Acceptable deviations (as determined in the V&V setup) will not invalidate the benchmark results. Instead they will alert the benchmarker that bit-wise reproducibility has been lost.

[2] Since many large-scale applications use C++ for its compile-time polymorphism and object-oriented features, we believe it is important to have HPCG be a C++ code. Historically C++ compilers have not received sufficient attention in the early phases of new system development. HPCG will provide incentive to re-prioritize efforts.

   b. Cache will be flushed between each of the k times the *m* iterations are performed to report fair timing data for averaging.
5. **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. Also reported will be the number of nodes, total storage, processors, accelerators, precision used, compiler version, optimization level, compiler directives used, flop count, power used, cache effects, loads and stores, etc.
   d. Checkpoint/Restart capabilities may be appropriate as well.

5. JUSTIFICATION FOR HPCG BENCHMARK

The HPCG Benchmark has merit as a new metric for high performance computing for the following reasons:

1. **Provides coverage of the major communication and computational patterns:** The major communication (global and neighborhood collectives) and computational patterns (vector updates, dot products, sparse matrix-vector multiplications and local triangular solves) in our production differential equation codes, both implicit and explicit, are present in this benchmark. Emerging asynchronous collectives and other latency-hiding techniques can be explored in the context of HPCG and aid in their adoption and optimization on future systems.
2. **Represents a minimal collection of the major patterns:** HPCG is the smallest benchmark code containing these major patterns, while at the same time representing a real mathematical computation (which aids in V&V efforts).
3. **Rewards investment in high-performance of collectives:** Neighborhood and all-reduce collectives represent essential performance bottlenecks for our applications that can benefit from high-quality system design. Improving the performance of HPCG will improve the performance of our real applications.
4. **Rewards investment in local memory system performance:** The local processor performance of HPCG is largely determined by effective use of the local memory system. Improvements in the implementation of HPCG data structures, compilation of HPCG code and the performance of the underlying system will improve HPCG benchmark results and real application performance, and will inform application developers on new approaches to optimizing their own implementations.
5. **Detects and measures variances from bitwise identical computations:** It is widely believed that future computer systems will not be able to provide deterministic execution paths for floating-point computations. Because floating-point addition is not associative, this means we will generally not have bitwise reproducible results, even when running the same exact computation twice on the same number of processors of the same system. This is in contrast with many of our MPI-only applications today, and presents a big challenge to applications that must certify their computational results and debug in the presence of bitwise variability. HPCG will make the deviation from bitwise reproducibility apparent.

6. RELATED WORK AND FUTURE ADAPTATIONS

Our proposed benchmark is not entirely new, nor do we expect what we propose to remain static. At the same time, previous efforts are not appropriate to leverage, nor do expected trends in algorithms suggest a better approach at this time.

Survey of Related Efforts

Similar benchmarks have been proposed and used before:

1. **NPB CG:** The NAS Parallel Benchmarks (NPB) [3] include a CG benchmark. It shares many attributes with what is propose here. Despite the wide use of this benchmark, it has the critical flaw that the matrix is chosen to have a random sparsity pattern with a uniform distribution of entries per row. This choice has led to the unfortunate result that a two-dimensional distribution of the matrix is optimal. Therefore, computation and communication patterns are non-physical. Furthermore, no preconditioning is present, so the important features of local sparse triangular solve is not represented and is not easily introduced, again because of the choice of a non-physical sparsity pattern. Although NPB CG has been extensively used for HPC analysis, it is not appropriate as a broad metric for our effort.
2. **Iterative Solver Benchmark:** A lesser-known but more relevant benchmark, the Iterative Solver Benchmark [4] specifies the execution of a preconditioned CG and GMRES iteration using physically meaningful sparsity patterns and several preconditioners. As such, its scope is broader than what we propose here, but this benchmark does not address scalable distributed memory parallelism or nested parallelism.

Evolution of HPCG Benchmark

Regardless of which specific benchmark we propose, we expect it to evolve. HPL 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.

7. SUMMARY AND CONCLUSIONS

The High Performance Linpack (HPL) Benchmark is an incredibly successful metric for the high performance computing community. The trends it exposes, the focused optimization efforts it inspires and the publicity it brings to our community are very important. At the same time, the relevance of HPL as a proxy for real application performance has become very low and we must seek an alternative.

We do not propose elimination of HPL as a metric. We believe the historical importance and community outreach value of HPL is far too important to be abandoned. Instead, HPCG will serve as an alternative ranking of the TOP500 list, in a similar way to how the Green 500 [5] re-ranks the items on this list.

HPCG is an attractive option 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.

As we develop HPCG we will incorporate thorough verification processes and perform extensive validation against real applications on existing and emerging platforms. Thorough verification and validation will improve the quality of HPCG and instill confidence in HPCG as a valid metric.

8. REFERENCES

1. Dongarra, J., et al. *Top 500 Supercomputer Sites*. 1999; Available from: http://www.top500.org.
2. OLCF. *Introducing Titan | The World's #1 Open Science Supercomputer*. 2013 [cited 2013 May 29, 2013]; Available from: http://www.olcf.ornl.gov/titan.
3. D. Bailey, E.B., J. Barton, D. Browning. R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, *The NAS Parallel Benchmarks*, 1994, NASA Ames Research Center: Moffett Field, CA.
4. Jack Dongarra, Victor Eijkhout, Henk van der Vorst, *Iterative Solver Benchmark.* Scientific Programming, 2001. **9**(4): p. 223-231.
5. CompuGreen. *The Green500 List News and Submitted Items | The Green500*. 2013 [cited 2013 May 29, 2013]; Available from: http://www.green500.org.

Appendix: Primer on the Preconditioned Conjugate Gradient Method

The linear conjugate gradient (CG) method is a widely used iterative method for solving linear systems of equations of the form $Ax = b$, where A is a large, usually sparse, matrix (or more generally a linear operator since only the action of the operator is necessary), b is a dense vector with known values and x is the vector whose values are sought as the result of the iterations. CG is effective for symmetric and near-symmetric systems of equations. It is one member of a broad collection of Krylov methods. CG is the simplest of these methods.

The preconditioned conjugate gradient (PCG) method performs an additional operation at each iteration (after a setup phase, which we do not discuss here): Applying the operator M^{-1}, which is an easy-to-apply approximation of A^{-1}. Preconditioning takes on many forms, and the most effective preconditioners are often those that use specific knowledge about properties of the linear operator. Even so, a very common computation used in preconditioning is a sparse triangular solve. This kernel is part of the Gauss-Seidel, incomplete Cholesky and incomplete LU preconditioners, which in turn are often used as smoothers in multi-level preconditioners. Even physics-based preconditioners will often invoke a similar sweep or sparse triangular solver as part of their execution. For this benchmark we use a local symmetric Gauss-Seidel preconditioner, which computes a forward and back triangular solve for a block of rows of the matrix A as determined by an additive Schwarz decomposition for parallel distributed memory computation.

Each PCG iteration requires these basic operations:

1. One matrix-vector product ($w := Ay$), where A is the original matrix, y is a known vector and w is computed.
2. One preconditioner application ($w := M^{-1}y$), where M^{-1} is an easy-to-apply approximation of A^{-1}, y is a known vector and w is computed. Note that M^{-1} is typically not directly formed.
3. Three vector updates ($w := \alpha y + \beta z$), where y and z are known vectors, α and β are known scalar values and w is computed.
4. Two vector inner products ($\alpha := y^T z$), where y and z are known vectors and the scalar α is computed.

A typical distributed memory parallel implementation of PCG uses the single-program-multiple-data (SPMD) pattern, usually implemented on top of MPI. Data is distributed by giving a block of rows of the matrix A, and the corresponding entries of the vectors x and b to each MPI process.

The key communication patterns in PCG are:

1. **Neighborhood collective:** Each MPI process contributes to computing its portion of the vector $w := Ay$ by using its own portion of y and by collecting the remote values of y needed from other processors (it will need a y value for each column in its rows that has at least one nonzero entry). Because many of our problems are discrete differential

equations, the remote y values form a halo or ghost region that surrounds the boundary of the portion of the physical domain that is assigned to the given MPI process. In order to compute w each MPI process must exchange values with its neighbors.

This is a collective operation since for any meaningful problem each MPI process will exchange values with at least one other process. However, each process only exchanges values within its neighborhood.

2. **All-reduce collective:** Vector inner products $\alpha := y^T z$ are a collective operation. Typically each MPI process computes the local portion of the dot product or norm. Then the values are exchanged (using one of many possible collective algorithms) so that each processor eventually receives the entire result.

   In a straightforward PCG implementation this computation is synchronous, meaning that an MPI process must wait for the final value α to arrive before proceeding to the computational step. On large systems, or systems with high variability in communication or computation costs, synchronous all-reduce steps can become the single biggest impediment to scalability. Good performance of synchronous all-reduce operations is often the most important feature that comes from purchasing a high-end parallel system vs. building a comparably sized commodity cluster.