Introduction to the HPC Challenge Benchmark Suite*

Jack J. Dongarra† Piotr Luszczek‡

December 13, 2004

Abstract

The HPC Challenge suite of benchmarks will examine the performance of HPC architectures using kernels with memory access patterns more challenging than those of the High Performance Linpack (HPL) benchmark used in the Top500 list. The HPC Challenge suite is being designed to augment the Top500 list, provide benchmarks that bound the performance of many real applications as a function of memory access characteristics e.g., spatial and temporal locality, and provide a framework for including additional benchmarks. The HPC Challenge benchmarks are scalable with the size of data sets being a function of the largest HPL matrix for a system. The HPC Challenge benchmark suite has been released by the DARPA HPCS program to help define the performance boundaries of future Petascale computing systems. The suite is composed of several well known computational kernels (STREAM, High Performance Linpack, matrix multiply – DGEMM, matrix transpose, FFT, RandomAccess, and bandwidth/latency tests) that attempt to span high and low spatial and temporal locality space.

1 High Productivity Computing Systems

The DARPA High Productivity Computing Systems (HPCS) [1] is focused on providing a new generation of economically viable high productivity computing systems for national security and for the industrial user community. HPCS program researchers have initiated a fundamental reassessment of how we define and measure performance, programmability, portability, robustness and ultimately, productivity in the HPC domain.

The HPCS program seeks to create trans-Petaflop systems of significant value to the Government HPC community. Such value will be determined by assessing many additional factors beyond just theoretical peak flops (floating-point operations). Ultimately, the goal is to decrease the time-to-solution, which means decreasing both the execution time and development time of an application on a particular system. Evaluating the capabilities of a system with respect to these goals requires a different assessment process. The goal of the HPCS assessment activity is to prototype and baseline a process that can be transitioned to the acquisition community for 2010 procurements.

The most novel part of the assessment activity will be the effort to measure/predict the ease or difficulty of developing HPC applications. Currently, there is no quantitative methodology for comparing the development time impact of various HPC programming technologies. To achieve this goal, the HPCS program is using a variety of tools including

- Application of code metrics on existing HPC codes,
- Several prototype analytic models of development time,
- Interface characterization (e.g. programming language, parallel model, memory model, communication model),
- Scalable benchmarks designed for testing both performance and programmability,

*This work was supported in part by the DARPA, NSF, and DOE though the DAPRA HPCS program under grant FA8750-04-1-0219.

†University of Tennessee Knoxville and Oak Ridge National Laboratory

‡University of Tennessee Knoxville

- Classroom software engineering experiments,
- Human validated demonstrations.

These tools will provide the baseline data necessary for modeling development time and allow the new technologies developed under HPCS to be assessed quantitatively.

As part of this effort we are developing a scalable benchmark for the HPCS systems.

The basic goal of performance modeling is to measure, predict, and understand the performance of a computer program or set of programs on a computer system. The applications of performance modeling are numerous, including evaluation of algorithms, optimization of code implementations, parallel library development, and comparison of system architectures, parallel system design, and procurement of new systems.

2 Motivation

The DARPA High Productivity Computing Systems (HPCS) program has initiated a fundamental reassessment of how we define and measure performance, programmability, portability, robustness and, ultimately, productivity in the HPC domain. With this in mind, a set of kernels was needed to test and rate a system. The HPC Challenge suite of benchmarks consists of four local (matrix-matrix multiply, STREAM, RandomAccess and FFT) and four global (High Performance Linpack – HPL, parallel matrix transpose – PTRANS, RandomAccess and FFT) kernel benchmarks. HPC Challenge is designed to approximately bound computations of high and low spatial and temporal locality (see Figure 1). In addition, because HPC Challenge kernels consist of simple mathematical operations, this provides a unique opportunity to look at language and parallel programming model issues. In the end, the benchmark is to serve bothe the system user and designer communities [2].

3 The Benchmark Tests

This first phase of the project have developed, hardened, and reported on a number of benchmarks. The collection of tests includes tests on a single processor (local) and tests over the complete system (global). In particular, to characterize the architecture of the system we consider three testing scenarios:

1. Local – only a single processor is performing computations.
2. Embarrassingly Parallel – each processor in the entire system is performing computations but they do no communicate with each other explicitly.
3. Global – all processors in the system are performing computations and they explicitly communicate with each other.

The HPC Challenge benchmark consists at this time of 7 performance tests: HPL [3], STREAM [4], RandomAccess, PTRANS, FFT (implemented using FFTE [5]), DGEMM [6, 7] and b_eff Latency/Bandwidth [8, 9, 10]. HPL is the Linpack TPP (toward peak performance) benchmark. The test stresses the floating point performance of a system. STREAM is a benchmark that measures sustainable memory bandwidth (in GB/s), RandomAccess measures the rate of random updates of memory. PTRANS measures the rate of transfer for larges arrays of data from multiprocessor's memory. Latency/Bandwidth measures (as the name suggests) latency and bandwidth of communication patterns of increasing complexity between as many nodes as is time-wise feasible.

Many of the aforementioned tests were widely used before HPC Challenge was created. At first, this may seemingly make our benchmark merely a packaging effort. However, almost all components of HPC Challenge were augmented from their original form to provide consistent verification and reporting scheme. We should also stress the importance of running these very tests on a single machine and have the results available at once. The tests were useful separately for the HPC community before and with the unified HPC Challenge framework they create an unprecendented view of performance characterization of a system – a comprehensive view that captures the data under the same conditions and allows for variety of analysis depending on end user needs.

Each of the included tests examines system performance for various points of the conceptual spatial and temporal locality space shown in Figure 1. The rationale for such selection of tests is to measures per-

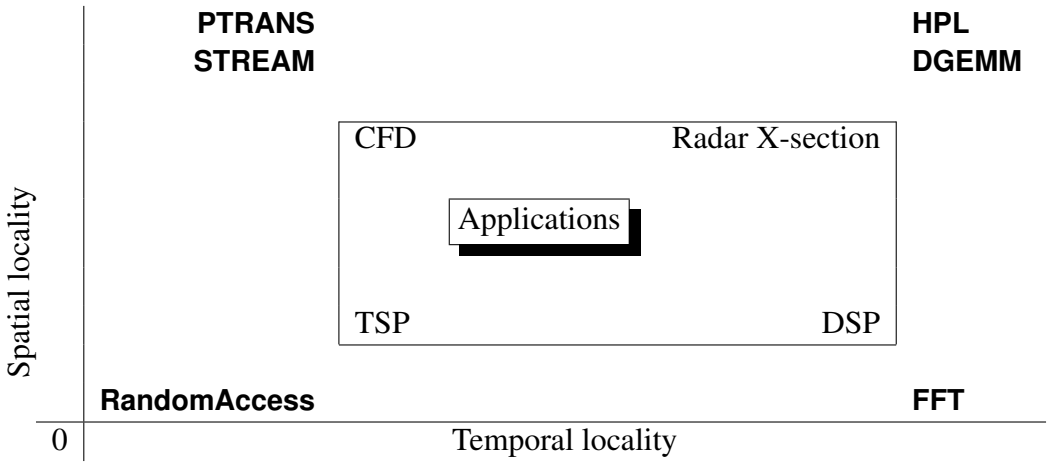

Figure 1: Targeted application areas in the memory access locality space.

formance bounds on metrics important to HPC applications. The expected behavior of the applications is to go through various locality space points during runtime. Consequently, an application may be represented as a point in the locality space being an average (possibly time-weighed) of its various locality behaviors. Alternatively, a decomposition can be made into time-disjoint periods in which the application exhibits a single locality characteristic. The application's performance is then obtained by combining the partial results from each period.

Another aspect of performance assesment addressed by HPC Challenge is ability to optimize benchmark code. For that we allow two different runs to be reported:

- Base run done with with provided reference implementation.
- Optimized run that uses architecture specific optimizations.

The base run, in a sense, represents behavior of legacy code because it is conservatively written using only widely available programming languages and libraries. It reflects a commonly used approach to prallel processing sometimes referred to as hierachical parallelism that combines Message Passing Interface (MPI) with threading from OpenMP. At the same time we recognize the limitations of the base run and hence we allow (or even encourage) optimized runs to be made. The optimizations may include alternative implementations in different programming languages using parallel environments available specifically on the tested system. To stress the productivity aspect of the HPC Challange benchmark, we require that the information about the changes made to the orignial code be submitted together with the benchmark results. While we understand that full disclosure of optimization techniques may sometimes be impossible to obtain (due to for example trade secrets) we ask at least for some guidence for the users that would like to use similar optimizations in their applications.

4 Benchmark Details

Almost all tests included in our suite operate on either matrices or vectors. The size of the former we will denote below as n and the latter as m. The following holds throughout the tests:

$$n^2 \simeq m \simeq \text{Available Memory}$$

Or in other words, the data for each test is scaled so that the matrices or vectors are large enough to fill almost all available memory.

HPL is the Linpack TPP (Toward Peak Performance) variant of the original Linpack benchmark which measures the floating point rate of execution for solving a linear system of equations. HPL solves a linear system of equations of order n:

$$Ax = b; \quad A \in \mathbf{R}^{n \times n}; \; x, b \in \mathbf{R}^n$$

by first computing LU factorization with row partial pivoting of the n by $n+1$ coefficient matrix:

$$P[A,b] = [[L,U],y].$$

Since the row pivoting (represented by the permutation matrix P) and the lower triangular factor L are applied to b as the factorization progresses, the solution x is obtained in one step by solving the upper triangular system:

$$Ux = y.$$

The lower triangular matrix L is left unpivoted and the array of pivots is not returned. The operation count for the factorization phase is $\frac{2}{3}n^3 - \frac{1}{2}n^2$ and $2n^2$ for the solve phase. Correctness of the solution is accertained by calculating scaled residuals:

$$\frac{\|Ax-b\|_\infty}{\varepsilon\|A\|_1 n}, \quad \frac{\|Ax-b\|_\infty}{\varepsilon\|A\|_1\|x\|_1}, \quad \text{and} \quad \frac{\|Ax-b\|_\infty}{\varepsilon\|A\|_\infty\|x\|_\infty},$$

where ε is machine precision for 64-bit floating-point values.

DGEMM measures the floating point rate of execution of double precision real matrix-matrix multiplication. The exact operation performed is:

$$C \leftarrow \beta C + \alpha AB$$

where:

$$A,B,C \in \mathbf{R}^{n\times n}; \quad \alpha,\beta \in \mathbf{R}^n.$$

The operation count for the multiply is $2n^3$ and correctness of the operation is accertained by calculating scaled residual: $\frac{\|C-\hat{C}\|}{\varepsilon n\|C\|_F}$ ($\hat{C}$ is the result of reference implementation of the multiplication).

STREAM a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s) and the corresponding computation rate for four simple vector kernels:

$$\begin{aligned} \text{COPY:} \quad & c \leftarrow a \\ \text{SCALE:} \quad & b \leftarrow \alpha c \\ \text{ADD:} \quad & c \leftarrow a+b \\ \text{TRIAD:} \quad & a \leftarrow b+\alpha c \end{aligned}$$

where:

$$a,b,c \in \mathbf{R}^m; \quad \alpha \in \mathbf{R}.$$

As mentioned earlier, we try to operate on large data objects. The size of these objects is determined at runtime which contrasts wit the original version of the STREAM benchmark which uses static storage (determined at compile time) an size. The original benchmark gives the compiler more information (and control) over data alignment, loop trip counts, etc. The benchmark measure GB/s and the number of items transferred is either $2m$ or $3m$ depending on the operation. The norm of differnce between reference and computed vectors is used to verify the result: $\|x-\hat{x}\|$.

PTRANS (parallel matrix transpose) exercises the communications where pairs of processors communicate with each other simultaneously. It is a useful test of the total communications capacity of the network. The performed operation sets a random an n by n matrix to a sum of its transpose with another random matrix:

$$A \leftarrow A^T + B$$

where:

$$A,B \in \mathbf{R}^{n\times n}.$$

The data transfer rate (in GB/s) is calculated by dividing the size of n^2 matrix entries by the time it took to perform the transpose. The scaled residual of the form $\frac{\|A-\hat{A}\|}{\varepsilon n}$ verifies the calculation.

RandomAccess measures the rate of integer random updates of memory (GUPS). The operation being performed on an integer array of size m is:

$$x \leftarrow f(x)$$

$$f: x \mapsto (x \oplus a_i); \quad a_i\text{– pseudo-random sequence}$$

where:

$$f: \mathbf{Z}^m \to \mathbf{Z}^m; \quad x \in \mathbf{Z}^m.$$

The operation count is m and since all the operations are in integral values using Galois field they can be checked exactly with a reference implementation. The verification procedure allows 1% of the operations to be incorrect (either skipped or done in the wrong order) which allows loosening concurrent memory update semantics on shared memory architectures.

FFT measures the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT) of size m:

$$Z_k \leftarrow \sum_j^m z_j e^{-2\pi i \frac{jk}{m}}; \quad 1 \le k \le m$$

where:

$$z, Z \in \mathbf{C}^m.$$

The operation count is taken to be $5m\log_2 m$ for the calculation of the computational rate (in GFlop/s). Verification is done with a residual $\frac{\|x-\hat{x}\|}{\varepsilon \log(m)}$ where $\hat{x}$ is the result of applying a refernce implementation of inverse transform to the outcome of the benchmarked code (in infinite-precision arithmetic the residual should be zero).

Communication bandwidth and latency is a set of tests to measure latency and bandwidth of a number of simultaneous communication patterns. The patterns are based on b_eff (effective bandwidth benchmark) – they are slightly different from the original b_eff. The operation count is linearly dependant on the number of processors in the tested system and the time the tests take depends on the parameters of the tested network. The checks are built into the benchmark code by checking data after it has been received.

5 Rules for Running the Benchmark

There must be one baseline run submitted for each computer system entered in the archive. There may also exist an optimized run for each computer system.

1. Baseline Runs
   Optimizations as described below are allowed.

   (a) Compile and load options
   Compiler or loader flags which are supported and documented by the supplier are allowed. These include porting, optimization, and preprocessor invocation.

   (b) Libraries
   Linking to optimized versions of the following libraries is allowed:

   - BLAS
   - MPI

   Acceptable use of such libraries is subject to the following rules:

   - All libraries used shall be disclosed with the results submission. Each library shall be identified by library name, revision, and source (supplier). Libraries which are not generally available are not permitted unless they are made available by the reporting organization within 6 months.
   - Calls to library subroutines should have equivalent functionality to that in the released benchmark code. Code modifications to accommodate various library call formats are not allowed.
   - Only complete benchmark output may be submitted – partial results will not be accepted.

2. Optimized Runs

   (a) Code modification
   Provided that the input and output specification is preserved, the following routines may be substituted:

   - In HPL: `HPL_pdgesv()`, `HPL_pdtrsv()` (factorization and substitution functions)
   - no changes are allowed in the DGEMM component
   - In PTRANS: `pdtrans()`
   - In STREAM: `tuned_STREAM_Copy()`, `tuned_STREAM_Scale()`, `tuned_STREAM_Add()`, `tuned_STREAM_Triad()`
   - In RandomAccess: `MPIRandomAccessUpdate()` and `RandomAccessUpdate()`
   - In FFT: `fftw_malloc()`, `fftw_free()`, `fftw_create_plan()`, `fftw_one()`, `fftw_destroy_plan()`, `fftw_mpi_create_plan()`, `fftw_mpi_local_sizes()`, `fftw_mpi()`, `fftw_mpi_destroy_plan()` (all these functions are compatible with FFTW 2.1.5 [11, 12] so the benchmark code can be directly linked against FFTW 2.1.5 by only adding proper compiler and linker flags, e.g. `-DUSING_FFTW`)
   - In Latency/Bandwidth component alternative MPI routines might be used for

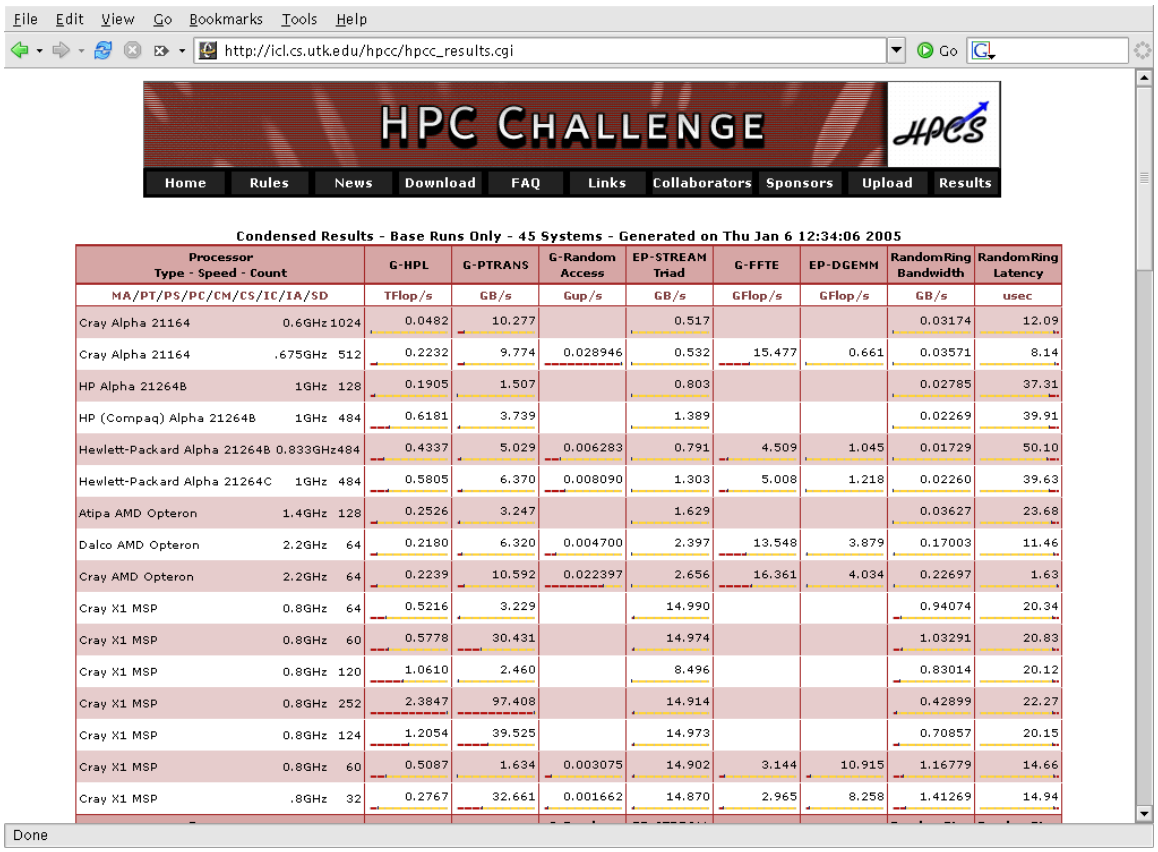

Figure 2: Sample results page.

communication. But only standard MPI calls are to be preformed and only to the MPI library that is widely available on the tested system.

(b) Limitations of Optimization

i. Code with limited calculation accuracy
The calculation should be carried out in full precision (64-bit or the equivalent). However the substitution of algorithms is allowed (see Exchange of the used mathematical algorithm).

ii. Exchange of the used mathematical algorithm
Any change of algorithms must be fully disclosed and is subject to review by the HPC Challenge Committee. Passing the verification test is a necessary condition for such an approval. The substituted algorithm must be as robust as the baseline algorithm. For the matrix multiply in the HPL benchmark, Strassen Algorithm may not be used as it changes the operation count of the algorithm.

iii. Using the knowledge of the solution
Any modification of the code or input data sets, which uses the knowledge of the solution or of the verification test, is not permitted.

iv. Code to circumvent the actual computation
Any modification of the code to circumvent the actual computation is not permitted.

6 Software Download, Installation, and Usage

The reference implementation of the benchmark may be obtained free of charge at the benchmark's web site: `http://icl.cs.utk.edu/hpcc/`. The reference implementation should be used for the base run. The in-

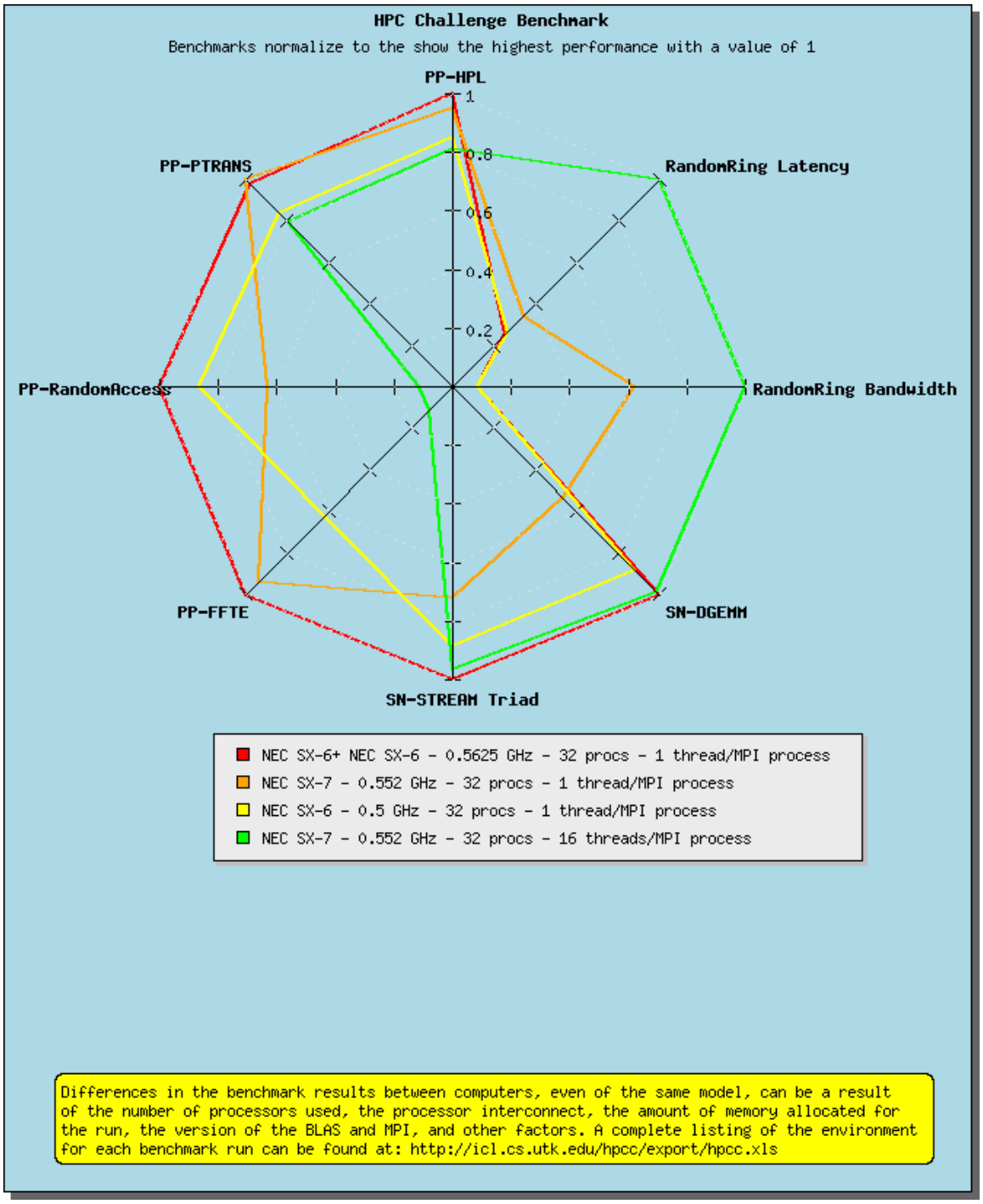

Figure 3: Sample kiviat diagram of results for two generations of hardware the same vendor with different number of threads per MPI node.

stallation of the software requires creating a script file for Unix's `make(1)` utility. The distribution archive comes with script files for many common computer architectures. Usually, few changes to one of these files will produce the script file for a given platform.

After, a succesful compilation the benchmark is ready to run. However, it is recommended that a changes be made to the benchmark's input file that describes the sizes of data to use during run. The sizes should reflect the available memory on the system and number of processors available for computations.

We have collected a comprehensive set of notes on

the HPC Challenge benchmark. They can be found at http://icl.cs.utk.edu/hpcc/faq/.

7 Example Results

Figure 2 show a sample rendering of the results web page: http://icl.cs.utk.edu/hpcc/hpcc_results.cgi. Figure 3 show a sample kiviat diagram generated using the benchmark results.

8 Conclusions

No single test can accurately compare the performance of HPC systems. The HPC Challenge benchmark test suite stresses not only the processors, but the memory system and the interconnect. It is a better indicator of how an HPC system will perform across a spectrum of real-world applications. Now that the more comprehensive, informative HPC Challenge benchmark suite is available, it can be used in preference to comparisons and rankings based on single tests. The real utility of the HPC Challenge benchmarks are that architectures can be described with a wider range of metrics than just Flop/s from HPL. When looking only at HPL performance and the Top500 List, inexpensive build-your-own clusters appear to be much more cost effective than more sophisticated HPC architectures. Even a small percentage of random memory accesses in real applications can significantly affect the overall performance of that application on architectures not designed to minimize or hide memory latency. HPC Challenge benchmarks provide users with additional information to justify policy and purchasing decisions. We expect to expand and perhaps remove some existing benchmark components as we learn more about the collection.

References

[1] High Productivity Computer Systems. (http://www.highproductivity.org/).

[2] William Kahan. The baleful effect of computer benchmarks upon applied mathematics, physics and chemistry. The John von Neumann Lecture at the 45th Annual Meeting of SIAM, Stanford University, 1997.

[3] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.

[4] John McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. (http://www.cs.virginia.edu/stream/).

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

[6] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and S. Hammarling. Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, March 1990.

[7] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:18–28, March 1990.

[8] Alice E. Koniges, Rolf Rabenseifner, and Karl Solchenbach. Benchmark design for characterization of balanced high-performance architectures. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01), Workshop on Massively Parallel Processing (WMPP)*, volume 3, San Francisco, CA, April 23-27 2001. In IEEE Computer Society Press.

[9] Rolf Rabenseifner and Alice E. Koniges. Effective communication and file-i/o bandwidth benchmarks. In *J. Dongarra and Yiannis Cotronis (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 8th European PVM/MPI Users' Group Meeting, EuroPVM/MPI 2001*, pages 24–35, Santorini, Greece, September 23-26 2001. LNCS 2131.

[10] Rolf Rabenseifner. Hybrid parallel programming on HPC platforms. In *Proceedings of the Fifth*

European Workshop on OpenMP, EWOMP '03, pages 185–194, Aachen, Germany, September 22-26 2003.

[11] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[12] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

```
import numarray, time
import numarray.random_array as naRA
import numarray.linear_algebra as naLA
n = 1000
a = naRA.random([n, n])
b = naRA.random([n, 1])
t = -time.time()
x = naLA.solve_linear_equations(a, b)
t += time.time()
r = numarray.dot(a, x) - b
r_n = numarray.maximum.reduce(abs(r))
print t, 2.0e-9 / 3.0 * n**3 / t
print r_n, r_n / (n * 1e-16)
```

Figure 4: Python code implementing Linpack benchmark.

Appendices

A Collaborators

- David Bailey — NERSC/LBL
- Jack Dongarra — UTK/ORNL
- Jeremy Kepner — MIT Lincoln Lab
- David Koester — MITRE
- Bob Lucas — ISI/USC
- John McCalpin — IBM Austin
- Rolf Rabenseifner — HLRS Stuttgart
- Daisuke Takahashi — Tsukuba

B Reference Sequential Implementation

Figures 4, 5, 6, 7, 8, and 9 show reference implementations of the tests from the HPC Challenge suite. Python was chosen (as opposed to, say, Matlab) to show that the tests can be easily implemented in a popular general purpose language.

```
import numarray, time
import numarray.random_array as naRA
import numarray.linear_algebra as naLA
m = 1000
a = naRA.random([m, 1])
alpha = naRA.random([1, 1])[0]
Copy, Scale = "Copy", "Scale"
Add, Triad = "Add", "Triad"
td = {}

td[Copy] = -time.time()
c = a[:]
td[Copy] += time.time()
td[Scale] = -time.time()
b = alpha * c
td[Scale] += time.time()
td[Add] = -time.time()
c = a * b
td[Add] += time.time()
td[Triad] = -time.time()
a = b + alpha * c
td[Triad] += time.time()
for op in (Copy, Scale, Add, Triad):
    t = td[op]
    s = op[0] in ("C", "S") and 2 or 3
    print op, t, 8.0e-9 * s * m / t
```

Figure 5: Python code implementing STREAM benchmark.

```
from time import time
from numarray import *
m = 1024
table = zeros([m], UInt64)
ran = zeros([128], UInt64)
mupdate = 4 * m
POLY, PERIOD = 7, 1317624576693539401L

def ridx(arr, i, tmp):
    tmp[0:1] = arr[i:i+1]
    if tmp.astype(Int64)[0] < 0:
        tmp <<= 1
        tmp ^= POLY
    else:
        tmp <<= 1
def starts(n):
    n = array([n], Int64)
    m2 = zeros([64], UInt64)

    while n[0] < 0:      n += PERIOD
    while n[0] > PERIOD: n -= PERIOD
    if n[0] == 0: return 1

    temp = array([1], UInt64)
    ival = array([0], UInt64)
    for i in range(64):
        m2[i] = temp[0]
        ridx(temp, 0, ival)
        ridx(temp, 0, ival)
    for i in range(62, -1, -1):
        if ((n>>i) & 1)[0]: break

    ran = array([2], UInt64)
    while (i > 0):
        temp[0] = 0
        for j in range(64):
            if ((ran>>j) & 1)[0]:
                temp ^= m2[j:j+1]
        ran[0] = temp[0]
        i -= 1
        if ((n>>i) & 1)[0]:
            ridx(ran, 0, ival)
    return ran[0]

ival = array([0], UInt64)

t = -time()
for i in range(m): table[i] = i
for j in range(128):
    ran[j] = starts(mupdate / 128 * j)
for i in range(mupdate / 128):
    for j in range(128):
        ridx(ran, j, ival)
        table[ran[j] & (m - 1)] ^= ran[j]
t += time()

temp = array([1], UInt64)
for i in range(mupdate):
    ridx(temp, 0, ival)
    table[temp & (m - 1)] ^= temp

temp = 0
for i in range(m):
    if table[i] != i: temp += 1

print t, 1e-9 * mupdate / t, 100.0*temp/m
```

Figure 6: Python code implementing RandomAccess benchmark.

```
import numarray, time
import numarray.random_array as naRA
import numarray.linear_algebra as naLA
n = 1000
a = naRA.random([n, n])
b = naRA.random([n, n])
t = -time.time()
a = numarray.transpose(a)+b
t += time.time()
print t, 8e-9 * n**2 / t
```

Figure 7: Python code implementing PTRANS benchmark.

```
import numarray, numarray.fft, time, math
import numarray.random_array as naRA
m = 1024
re = naRA.random([m, 1])
im = naRA.random([m, 1])
a = re + 1.0j * im

t = -time.time()
b = numarray.fft.fft(a)
t += time.time()

r = a - numarray.fft.inverse_fft(b)
r_n = numarray.maximum.reduce(abs(r))

Gflop = 5e-9 * m * math.log(m) / math.log(2)

print t, Gflop / t, r_n
```

Figure 8: Python code implementing FFT benchmark.

```
import numarray, time
import numarray.random_array as naRA
n = 1000
a = naRA.random([n, n])
b = naRA.random([n, n])
c = naRA.random([n, n])
alpha = a[n/2, 0]
beta = b[n/2, 0]
t = -time.time()
c = beta * c + alpha * numarray.dot(a, b)
t += time.time()
print t, 2e-9 * n**3 / t
```

Figure 9: Python code implementing DGEMM benchmark.