Special Topic: High Performance Computing

A new metric for ranking high-performance computing systems

Jack Dongarra[1],*, Michael A. Heroux[2] and Piotr Luszczek[1]

INTRODUCTION

Performance benchmarks play an important role in various stages of hardware development and use. During development, hardware designers use benchmarks as proxies for full-fledged applications because prototype systems have a limited set of software tools that the applications require for compilation and execution. At procurement time, benchmarks serve as an assurance test that establishes the new system's viability and fulfillment of contractual obligations between the system integrator and the customer. And finally, benchmarking during the system's daily use can ensure proper operation of a computer installation, exposing any potential problems to the system's administrators, and gives the users an estimate of what their applications of interest can achieve without requiring the effort of building those applications, their software dependences, and loading the necessary input data that may initially reside offsite.

The LINPACK benchmark [1] has been in continuous use since the 1980s. It was born out of necessity in the 1970s, when it was used to quickly test the performance of vector subroutines—which could serve as a good approximation of performance for the rest of the LINPACK library. Because of the nature of the implementation, the benchmark could also be used as a first-order approximation of other codes, partially due to the well-balanced hardware that offered plentiful bandwidth for every floating-point operation. Over the years, Moore's law eroded the compute-to-bandwidth balance, resulting in a memory wall. Today, this wall can be exhibited by the Intel Haswell (Xeon E7 v3 8900) processors, which feature 18 cores—each equipped with dual floating-point units (FPUs) with AVX vectors capable of fused multiply-add (FMA) instructions and clocked at 2.5 GHz (before TurboBoost)—bringing to bear, altogether, over 700 Gflop/s worth of peak performance—over 90% of which may be realized in a well-tuned matrix–matrix multiplication from a vendor library (MKL). At the same time, the memory controller for that processor has a theoretical maximum of 25 GB/s. The exact number of floating operations per every byte of bandwidth is 28.8—a far different ratio than the 1 flop per byte, which was the design point in the 1990s when the TOP500 list started ranking the supercomputers using the scalable version of the LINPACK benchmark.

To reassess the application needs in this new and drastically different hardware regime, it is worthwhile to look at the computational simulations that drive national interest. Many of these simulations involve heat diffusion, electromagnetics, and fluid dynamics. Unlike LINPACK, which tests raw floating-point performance and its delivery through the BLAS API, these real-world applications rely on partial differential equations (PDEs) that govern the continuous representations of the physical quantities such as particle speed, momentum, etc. These PDEs involve sparse (not dense) matrices that represent the 3Dl embedding of the discretization mesh. While the size of the sparse data fills the available memory to accommodate the simulation models of interest, most of the optimization techniques that help achieve close to peak performance in dense matrix calculations are only marginally useful in the context of sparse matrices originating from PDEs.

Our new benchmark, called high-performance conjugate gradients (HPCG) (further information is available at www.hpcg-benchmark.org), is based on Mantevo collection's HPCCG code base [2], but aims to go beyond its originator and represent the calculations that commonly occur during the numerical solution of PDEs in modern state-of-the-art solvers. To that end, HPCG is dominated by sparse operations such as sparse matrix-vector product and sparse matrix triangular solve. When properly implemented, these operations stress the memory subsystem, including the higher level cache memories' ability to coalesce irregularly strided memory loads, bandwidth and latency of the main memory, as well as the processing units' ability to schedule around the inherent delays of the main memory. In terms of communication, the benchmark tests global and neighborhood collectives for dot products and halo (domain boundary) exchanges, respectively. As an added irregularity, HPCG features elements of multigrid with a local smoother. This counteracts the initial regularity of the domain and increases the complexity of memory access patterns that become much less bandwidth bound and more latency bound at the coarse levels of the multigrid. As the figure of merit, HPCG uses the Gflop/s rate based on the apparent (rather than actual) number of floating-point operations executed during execution (apparent operations are those that have to be performed based on the structure of the matrix, actual operations depend on the implementation—they could be larger if the implementation chooses to recomputed some of the intermediate results). Such a metric is unconstrained and benefits from improvements applied to any of the hardware components relevant to the simulation applications based on PDEs. At the same time, the low value of the achieved Gflop/s serves as a stark reminder of the imbalance between the

floating-point capability and the speed of various data storage and transmission pathways in modern supercomputing systems.

GOALS OF BENCHMARK

The HPCG Benchmark can help alleviate many of the problems described above using the following principles:

- *Provide comprehensive coverage of the code types that test major communication and computational patterns.* Both global and neighborhood collectives are essential in terms of communication, and the important computational patterns include vector updates, dot products, sparse matrix-vector multiplication, and local triangular solves. These are inspired by our production differential equation codes, both implicit and explicit, and therefore made their way into this benchmark. In addition, 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.
- *Represent a minimal collection of the major patterns.* Even though it is a benchmark code rather than a full application, HPCG represents major computational patterns well, and is—to our knowledge—the smallest code containing them. The HPCG methods and algorithms approximate real mathematical computation very well, which aids in our validation and verification efforts described below.
- *Reward investment in high performance of collective communication primitives.* The neighborhood and all-reduce collectives represent essential performance bottlenecks for our applications. Implementations of these primitives can benefit from high-quality system design. As a consequence, improving the performance of HPCG will improve the performance of real applications.
- *Reward investment in local memory system performance.* As the local shared-memory and multicore performance of

HPCG is largely determined by the effective use of the local memory system, it becomes important to stimulate improvements at the hardware level by rewarding good design decisions with appropriate improvements in the benchmark results. A good correlation already exists between improvements in the implementation of HPCG data structures, compilation of HPCG code, the performance of the underlying system, improvements in HPCG benchmark results, and real application performance. These results will inform application developers of new approaches to optimizing their own implementations.

RELATED WORK

For a long time, the standard for measuring parallel sparse solvers was the NAS parallel benchmarks (NPB) collection [3,4] and its recent updates [5]. Since HPCG and NPB both aim to extract the most common features of scientific computational kernels, there are similarities in design and algorithmic implementation between both. There are, however, three important differences: (i) the data distribution and generation, (ii) splitting of code components, and (iii) algorithmic goals and choices of the code. While NPB uses 2D data distribution, HPCG uses 1D distribution—a more common distribution for modern solvers. NPB uses the structure of the matrix chosen from a uniform random distribution based on a multiplicative random number generator [6]. This can hardly be correlated with the actual matrices of common PDE discretizations that admit a 3D embedding or similar low-dimensional space representation. Thus, HPCG uses a regular 3D grid discretization (see below), but prohibits exploiting this structure (this may be enforced by stochastically testing the implementation with slightly perturbed structure). NPB splits the multigrid (NPB MG) and conjugate gradient (NPB CG) components, while HPCG treats them together. Additionally, HPCG combines domain decomposition with additive Schwarz coupling as that is

the preferred method for a parallel sparse solver.

Another sparse benchmark, which in our mind is very closely related to our efforts, was the iterative solver benchmark [7]. The algorithmic scope of this benchmark is broader than HPCG: it includes both CG and GMRES methods, it tests a number of meaningful sparsity patterns, and uses several preconditioners. With such a broad spectrum of tested codes it still lacks any tests of scalability that target distributed memory parallelism, multicore processors, or hardware accelerators—testing of each of these aspects of modern supercomputers is an essential prerequisite for a comprehensive benchmark code.

MODEL PROBLEM AND ITS DISCRETIZATION

The model problem that HPCG benchmark solves is a discretized Poisson PDE in three dimensions [8]. The iterative method of choice is preconditioned CG applied to the resulting symmetric positive definite sparse linear system. The original PDE models a single degree of freedom heat diffusion equation with zero Dirichlet boundary conditions. The PDE is discretized with a finite difference scheme on a 3D rectangular grid domain with regular spacing of the nodes, thus producing a sparse matrix with a simple and predictable structure.

The parallelism of the solver may be scaled arbitrarily through the domain decomposition scheme with additive Schwarz coupling. The partitioning of the global discretization grid among the distributed memory processes is regular and three dimensional: across the x-, y-, and z-axes. The factoring of the distributed processes (the ranks) into a 3D regular grid can deteriorate into a 1D or 2D structure if the total rank count does not have good integer factors, e.g. it's a prime number. In such a case, the communication patterns between the processes will be local for the most part, and will not reflect the challenges imposed by scientific applications on the large-scale HPC networks. For this reason, one

of the checks in HPCG keeps the ratio min(x,y,z)/max(x,y,z) high enough: it is not supposed to drop below 0.125. Choosing this ratio to be high would keep the shape closer to a cube, but due to the distribution of factors in process counts up to hundreds of millions, keeping the ratio high would preclude a large number of configurations from being eligible.

During the setup phase, a logically global, but physically distributed, sparse linear system is constructed using a 27-point stencil at each grid point in the 3D domain, such that the equation—at any interior point—depends on the values at that point and its 26 surrounding neighbors. The matrix is constructed to be weakly diagonally dominant for the interior points of the global domain, and strongly diagonally dominant for the 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 non-zero entries per row for interior equations and 7–18 non-zero terms for boundary equations.
- A symmetric, positive definite linear operator.
- The boundary condition is reflected by subtracting 1 from the diagonal.
- A generated known exact solution vector with all values equal to 1.
- A matching right-hand-side vector.
- An initial guess of all zeros.

PRECONDITIONED ITERATIVE SOLVER

The preconditioned CG method, shown in Fig. 1, allows the code to maintain the orthogonality relationship with a short three-term recurrence formula. This in turn allows the linear system data to be scaled arbitrarily without worrying about the excessive growth of storage requirements for the orthogonal basis, unlike GMRES, which was used in the iterative solver benchmark. GMRES in-

ternal storage requirements grow with the system size and it requires to balance the scale with the restriction inherent to the solution method. CG uses short-term recurrence relation to keep track of its progress and hence internal storage does not grow with the problem size. As mentioned above, the central purpose of defining this sparse linear system is to provide a rich vehicle for executing the collection of important computational kernels. The pseudo code shown in the figure features these kernels prominently. At the same time, the benchmark's primary function is not to compute a numerically exact solution to the model problem. In fact, the iteration counts are fixed in the benchmark code and we do not expect convergence to the solution, regardless of the problem size. We do use the spectral properties of both the problem and the preconditioned CG algorithm as part of software verification.

HPCG performs multiple sets of 20 iterations, using the same initial guess each time. These parameters were chosen to be sufficiently large to test the system's re-

silience and ability to remain operational. By doing this, we can compare the numerical results for 'correctness' at the end of each of the iteration sets.

COMPONENTS OF AN MG SOLVER

The MG method has been a subject of extensive research and may be considered ideal for elliptic PDEs. However, by varying the discretization, it is possible to apply it successfully to a much larger class of linear and non-linear PDEs [9]. As described so far, HPCG directly characterizes all of the computational and communication patterns exhibited by MG solvers. Specifically, the unaddressed issues include smoothing through hard-to-parallelize methods, such as the Gauss–Seidel iteration, and the dominant performance bottleneck at coarse grid levels in the form of latency rather than bandwidth. What dominates the overheads are the message exchanges at the fine grid levels and dot products of the preconditioned CG method. For these reasons,

```
A, b, x = hpcg_setup()

max_iter = 20
tolerance = 0.0

rtz = 0.0

p = x.copy()

Ap = SparseMatVecProduct(A, p)
r = WAXPBY(1.0, b, -1.0, Ap) # r = b - A*p
normr = sqrt(DotProduct(r, r))

normr_old = normr

for k in range(1, max_iter+1):
  if normr/normr_old > tolerance:
    break

  z = MultigridPreconditioner(A, r)

  if k == 1:
    p = z.copy()
    rtz = DotProduct(r, z)
  else:
    rtz_old = rtz
    rtz = DotProduct(r, z)
    beta = rtz / rtz_old
    p = WAXPBY(1.0, z, beta, p) # p = beta*p + z

  Ap = SparseMatVecProduct(A, p)
  alpha = rtz / DotProduct(p, Ap)
  x = WAXPBY(1.0, x,  alpha, p) # x = x + alpha*p
  r = WAXPBY(1.0, r, -alpha, Ap) # r = r - alpha*Ap
  normr = sqrt(DotProduct(r, r))
  niters = k
```

Figure 1. Preconditioned CG algorithm used by HPCG.

starting with version 2.0 of HPCG, an MG component was introduced in the reference code to model the behavior of multilevel methods. In particular, the MG method is used for preconditioning as shown in Fig. 1 with the multiple levels of coursing and refinement as well as restriction and prolongation contained therein. The problem often faced when introducing this kind of new non-trivial functionality was a potential for a substantial increase in the code complexity and growth in tangential and supporting components, such as the validation and verification modules. Hence, to minimize the impact of the change, we reused the existing components and recast them in terms of commonly used parts of a typical MG solver. The smoother/solver for all of the levels of our simulated geometric MG is a local Gauss–Seidel iteration. The mesh coarsening and refinement (together with the restriction and prolongation operators) is accomplished by simple halving of the number of points in every dimension, and thus each coarse grid level has $8(=2 \times 2 \times 2)$ times as few points as the neighboring fine grid level.

Just as was the case for the preconditioned CG from Fig. 1, our goal is only to provide basic components rather than a complete MG solver, let alone a comprehensive implementation of a full MG solver. Consequently, we include neither the full V nor W cycles, which are named after the shape of the grid mesh hierarchy. In particular, we do not perform an accurate solve at the coarsest grid level to remove all error modes and provide a good solution vector for prolongation. Instead, we limit the number of grid levels to 3, which results in a 256-fold reduction in the number of grid points, which is sufficient to address most of the bandwidth and/or latency bottlenecks and expose the performance of common algorithmic tradeoffs in broadly used solvers. At fine grain level, bandwidth is the main concern as the number of grid points is large enough for pipelining to hide the latency. When the coarse grid is used, the number of points is reduced 256 times and latency is this many times harder to hide. Adding more grid levels could potentially expose these bottlenecks even more but as the grid becomes coarser,

the latency overhead becomes one of many other bottlenecks such as loop overheads due to lack of benefits from unrolling. In this limited implementation, we also captured the prevalent recursive patterns of code execution and the integer arithmetic required to capture some of the overheads inherent in grid mesh manipulation.

IMPLEMENTATION DETAILS

As was indicated throughout this document, the aim of any benchmark—and HPCG in particular—is to strike a balance between complexity, software dependences, and requirements imposed by real scientific applications. The implementation details and low-level software engineering artifacts play a role in this, and we made design choices along these specific issues. For the programming language we chose a subset of modern C++, which is a rapidly evolving language comparatively speaking, and the compiler writers attempt to follow the progress of the standard and even include provisional features that are not guaranteed to make it to the upcoming official document in the particular form and/or semantics, if at all. As of this writing, the majority of compilers have implemented a substantial portion of the C++ 2011 and 2014 standards. But to an extent, using some of these more modern features could jeopardize the portability of the code. At the same time, the code base is fairly simple by design and the implementation remains compact and readable without relying on potentially nonportable constructs while stressing the use of C++ in scientific codes. As a point of interest, a similar reasoning applies to any other language of choice or the associated standard library of routines.

Along these lines, we chose OpenMP for multithreading and Message Passing Interface (MPI) for distributed memory communication. The former is currently in version 4, while the latter is in version 3. Both of these are backed up by decades of continuous development and widespread community support. It is safe to assume that these standards will and should be available for future supercomputers.

Finally, hardware accelerators are now an important component of modern supercomputing installations and clusters. To this end, the code available for download has provisions for optimizations that take advantage of such accelerators, namely GPUs and coprocessors. In addition, there are vendor-optimized implementations available from either the HPCG web page or from the respective vendors directly.

VALIDATION, VERIFICATION, AND BASELINE RULES

The regularity of the model PDE's discretization grid provides ample opportunity to optimize the sparse data structure for efficient computation. Results show how to optimally partition and reorder the mesh points to achieve good load balance, small communication volume, and good local performance. However, we feel that allowing such optimizations would violate the spirit of the benchmark and trivialize its results. Instead, we insist that the knowledge of the problem's regularity should not be taken into consideration when porting and optimizing the code for the user's machine. To that extent, the discretization should be treated as a generic mesh without any properties known a priori. In exchange, the users may take advantage of the simplicity of the mesh to find problems with their optimizations, since many aspects of the optimal solution are known in closed form and can serve as a useful debugging tool. Similarly, we prohibit the use of knowledge of the problem when performing the CG iteration. We do, however, recognize that users may wish to use the knowledge of the spectrum of the discretization matrix to estimate the accuracy of their optimized solver.

HPCG includes a code that detects and measures variances from bitwise identical computations because it is widely believed that future computer systems will not be able to provide deterministic execution paths for floating-point computations. In fact, bit fluctuations may already be present on some accelerators, and can even be exacerbated by disabling error correcting

code on some hardware. This may happen in particular, because floating-point addition is not associative, thus we may 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. Going forward, it presents a big challenge to applications that must ascertain their computational results and perform numerical debugging in the presence of bitwise variability. The setup and execution of the tests in HPCG makes the deviation from bitwise reproducibility easier to observe.

To detect anomalies during the iteration phases, HPCG code uses standard software engineering practices such as computing preconditions, post-conditions, and invariants. These are likely to eliminate a vast majority of errors that might creep in when implementing an optimized version of the benchmark.

In order to take full advantage of the tested hardware, the end user is encouraged—but not required—to optimize the computational kernels in HPCG. The reference code that we provide is focused on portability and readability, which may often have negative effects on performance for a specific system. In practice, we have already observed successful attempts to optimize the HPCG code by benchmarking teams and vendor engineering groups with successful outcomes and good results reported. The resulting code is available on the download page or directly from the respective vendor site. In order to validate the user-provided kernels, HPCG includes a symmetry test for the sparse matrix multiply with a discretization matrix and for the symmetric Gauss–Seidel smoother.

Finally, a spectral test is also included in HPCG, whose purpose is to test for fast convergence of the CG algorithm on a modified matrix that is close to being diagonal. From the theoretical standpoint, and according to the convergence framework underlying the CG solver [10], we know that a fixed number of iterations is required for such matrices, and the invalid optimizations attempted by the user

should easily violate this property. The spectral test is meant to detect potential anomalies in the optimized implementation related to inaccurate calculations and changes to convergence rate due to user-defined matrix ordering.

PERFORMANCE RESULTS

The main performance result reported by HPCG is the number of floating-point operations per second. It is a metric that is bound (in theory at least) by the hardware's peak performance rate (in practice, HPCG achieves a single digit fraction of the peak), which is mostly related to the number of FPU, their throughput, cycle delay, and the available instruction mix such as pipelined add, multiply, and FMA. This upper bound is unrealistic for iterative solvers for elliptic PDEs, such as HPCG, or most other scientific applications. Only the LINPACK benchmark can claim sufficient data reuse and locality in its computational patterns to be able to keep the FPUs busy at almost every cycle, and thus extract a substantial percentage of the theoretical peak performance of the hardware. For an example of such an implementation of HPL, refer to the TOP500 listing of the K computer. HPCG's performance result is largely dependent on the memory hierarchy and the quality of the interconnect. The bandwidth, latency, and parallelism in data transfers all contribute to an overall all good score reported by HPCG. It is interesting then to look at how these factors affect the largest supercomputing installations in the world. In fact, this is

the purpose of collecting and publishing the HPCG results obtained on large supercomputing installations around the world.

Figure 2 shows the HPCG results, from supercomputing installations from all over the world, as reported in June and November 2014 and in June 2015. Since HPCG is still a relatively new benchmark, not all lists of results are of the same length and the number of entries does not yet rival the TOP500. However, to put this growth rate and the benchmark adoption in perspective, it is worth noting the nearly 25-year lapse between the first publication of LINPACK results and the release of the first TOP500 list. There are currently 40 large computing systems whose results are reported in the HPCG list, and a number of interesting trends may be observed on the results figure. The HPCG results show orders of magnitude difference across the systems on a single list—a trend similar to what has been happening on the TOP500 list. Another anomaly worth noting is that a number of systems on the November 2014 HPCG list are faster than their counterparts from June 2015. This is due to the results being scaled across larger parts of the respective systems, and some of the results being withdrawn due to code and rules changes. In addition, the recent stagnation at the top of the TOP500 list, which saw almost no change in the past several editions, can also be observed on the HPCG results lists.

Another interesting metric to help compare systems is to count the number of inversions between the TOP500 and
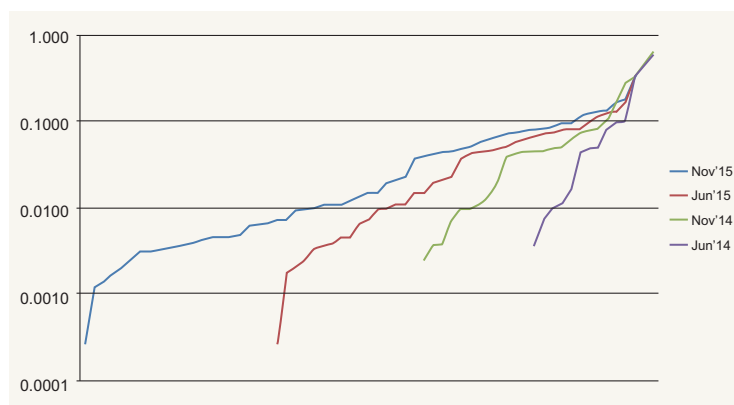

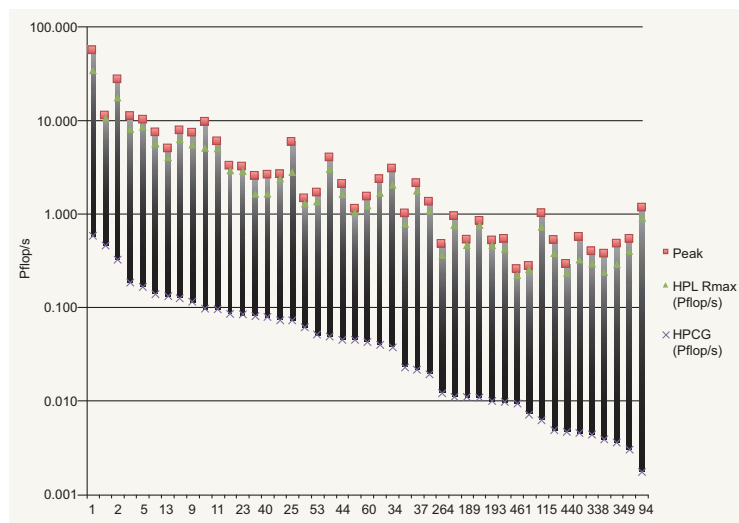**Figure 2.** HPCG results announced biannually since June 2014.

Figure 3. Comparison between Peak, HPL, and HPCG performance numbers for the TOP500 machines from November 2015.

at 34 Pflop/s. At the other end of the spectrum, with PDE solvers, HPCG achieves only 0.6 Pflop/s, yet Tianhe-2 still remains the fastest result in the world—even by this metric.

Jack Dongarra[1,*], Michael A. Heroux[2] and Piotr Luszczek[1]

[1]Innovative Computing Laboratory, EECS Department, University of Tennessee, Tennessee, USA

[2]Scalable Algorithms Department, Sandia National Laboratories, Minnesota, USA

***Corresponding author.**
E-mail: dongarra@icl.utk.edu

HPCG lists. One prominent example is the inversion that occurs between the Titan supercomputer and the K computer. While the former is ahead on the TOP500 list, the latter is faster on the HPCG list. It is worth noting that both systems use an expert-optimized version of the HPL and HPCG codes, which guarantees nearly maximized performance results. Titan's peak performance (27 Pflop/s) is superior to K computer (11 Pflop/s) due to the use of GPU accelerators from NVIDIA and HPL closely tracks the peak performance mark. However, Titan achieves only 65% of the peak while K computer achieves over 93%. The reason for this drastic difference lies mainly in the interconnect of the two machines. K uses Tofu interconnect while Titan uses Cray Gemini. The former is highly overprovisioned with respect to the local node performance while the latter, while still superior to many commercial offerings, delivers much smaller bytes/flop ratio. When running HPCG, the importance of fast network increases even further as the contribution of execution time form network communication (collectives and neighborhood exchanges) is substantial. As a result, K computer achieves 0.46 Pflop/s and Titan 0.32 Pflop/s. The former result translates to over 4% of the peak performance while the latter is at about 1%—another indication of Tofu network's superiority. Currently, it is important not

to read too much into the total inversion count (163 for the November 2015 HPCG list), until the HPCG benchmark more firmly establishes its presence in the field and more comparisons can be made across more comprehensive set of results.

Looking at the fraction of the peak performance that is achieved by various categories of systems on the HPCG list, we see three main results. Vector machines with very high memory bandwidth achieve about 10% of the peak performance. Highly specialized machines with very good interconnects achieve about 5% while more commonplace systems have the ratio stuck at around 1%. This is somewhat similar to the general trend on the TOP500 list where systems with InfiniBand interconnect achieve about 70% of the peak performance and Ethernet-based machines only attain 30%.

Figure 3 shows a comparison between performance numbers for the peak, HPL, and HPCG. The figure is ordered according to the HPCG score which allows us to see inversions with respect to the peak and HPL numbers. Another noticeable trend is a universal drop (by two orders of magnitude) in performance from HPL to HPCG.

Finally, it is also worth noting the main performance numbers for Tianhe-2, the world's largest supercomputing system. Its peak performance rate is 55 Pflop/s, and the HPL benchmark achieves about 62% of that number

REFERENCES

1. Dongarra, JJ, Luszczek, P and Petitet, A. The LINPACK benchmark: past, present and future. *Concurr Comput Pract Exp* 2003; **15**: 803–20.
2. Heroux, MA, Doerfler, DW and Crozier, PS *et al.* Improving performance via mini-applications. [Techreport] *SANDIE REPORT*, SAND2009-5574, Sandia National Laboratories, 2009.
3. Bailey, D, Barszcz, E and Barton, JT *et al.* The NAS parallel benchmarks. [Techreport] *NAS Technical Report RNR-94-007*. NASA Ames Research Center, Moffett Filed, CA, 1994.
4. Bailey, D, Harris, T and Saphir, WC *et al.* The NAS parallel benchmarks 2.0. [Techreport] *NAS-95-020*. NASA Ames Research Center, Moffett Filed, CA, 1995.
5. Wijngaart, RV. NAS parallel benchmarks version 2.4. [Techreport] *NAS Technical Report NAS-02-007*. Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division, NASA Ames Research Center, Moffett Field, CA, 2002.
6. Knuth, DE. *The Art of Computer Programming*. *Addisona-Wesley Professional*, 2nd edn. Boston: Addison-Wesley, 1998.
7. Dongarra, JJ, Eijkhout, V and van der Vorst, H An iterative solver benchmark. *Sci Progr* 2001; **9**: 223–31.
8. Mattheij, RMM, Rienstra, SW and ten Thije, JHM. *PartialDifferential Equations, Modeling, Analysis, Computation.* Philadelphia: SIAM, 2005.
9. Trottenberg, U, Oosterlee, CW and Schuller, A. *Multigrid*. London, UK: Academic Press, 2001.
10. Saad, Y. *Iterative Methods for Sparse Linear Systems*, 2nd edn. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2003.