Metrics for Ranking the Performance of Supercomputers

Tzu-Yi Chen
Dept. of Computer Science
Pomona College
Claremont, CA 91711
tzuyi@cs.pomona.edu

Meghan Gunn
Dept. of Math & CS
University of San Diego
San Diego, CA
meghan.gunn@gmail.com

Beth Simon
Dept. of CS and Engr.
UC San Diego
La Jolla, CA 92093
esimon@cs.ucsd.edu

Laura Carrington, Allan Snavely
San Diego Supercomputer Center
9500 Gilman Dr.
La Jolla, CA 92093
{lcarring,allans}@sdsc.edu

Abstract

We introduce a metric for evaluating the quality of any predictive ranking and use this metric to investigate methods for answering the question: How can we best rank a set of supercomputers based on their expected performance on a set of applications? On modern supercomputers, with their deep memory hierarchies, we find that rankings based on benchmarks measuring the latency of accesses to L1 cache and the bandwidth of accesses to main memory are significantly better than rankings based on peak flops. We show how to use a combination of application characteristics and machine attributes to compute improved workload-independent rankings.

1 Introduction

Low-level performance metrics such as processor speed and peak floating-point issue rate (flops) are commonly reported, appearing even in mass-market computer advertisements. The implication is that these numbers can be used to predict how fast applications will run on different machines, so faster is better. More sophisticated users realize that manufacturer specifications such as theoretical peak floating-point issue rates are rarely achieved in practice, and so may instead use simple benchmarks to predict relative application performance on different machines. For understanding parallel performance, benchmarks range from scaled down versions of real applications to simpler metrics (eg, the NAS parallel benchmark suites [2], the SPEC benchmark [15], the HINT benchmark [6], the HPC Challenge benchmark [10], STREAM [17], and the ratio of flops to memory bandwidth [12]).

A particularly well-known parallel benchmark is Linpack [5], which has been used since 1993 to rank supercomputers for inclusion on the Top 500 list [18] (more recently the IDC Balanced Rating [7] has also been used to rank machines). The Top 500 list is popular partly because it is easy to read, is based on a simple metric that is easy to measure (essentially peak flops), and is easy to update. Unfortunately, such benchmarks have also been found insufficient for accurately predicting runtimes of real applications [3].

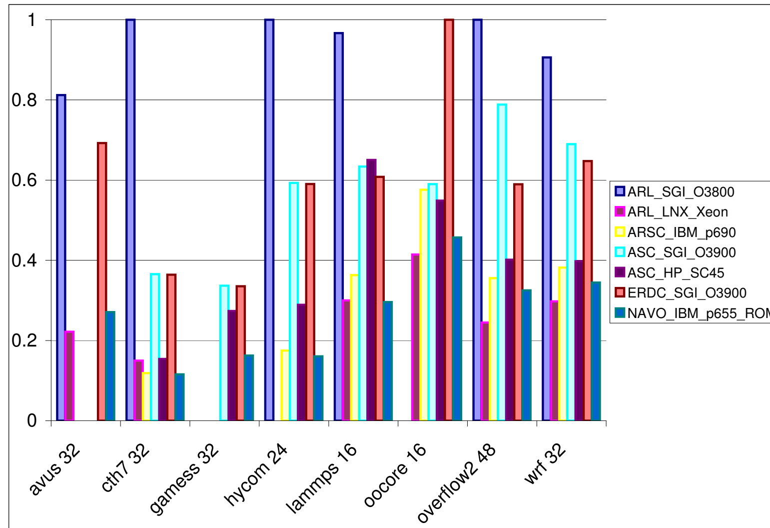
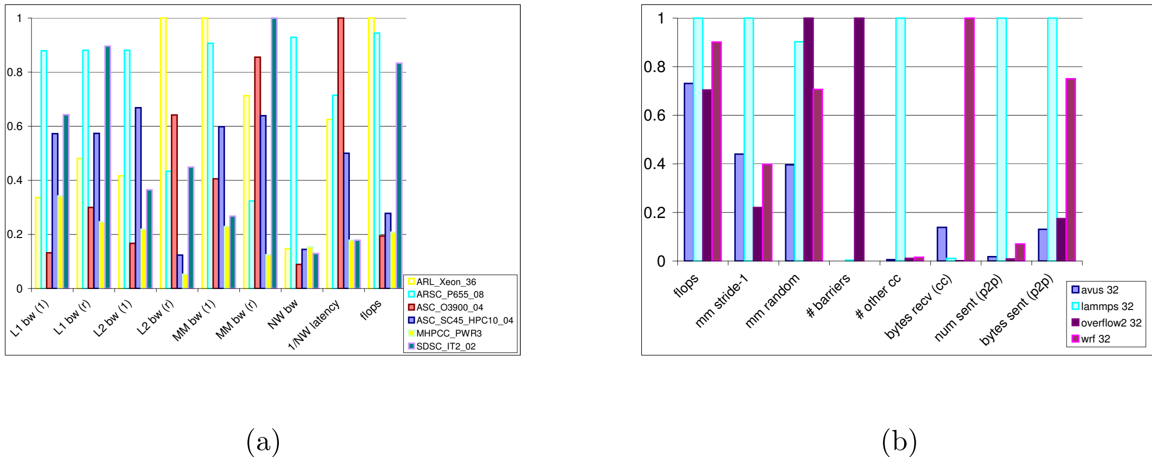


Figure 1: This graph shows the measured runtimes of 8 applications on 7 supercomputers. The runtime for each application is divided by the maximum time taken by any of the entire set of 14 machines to run that application.

The reason is clear. Consider Figure 1, which plots the performance of 8 different High Performance Computing (HPC) codes on 7 different supercomputers. The codes are a subset of those in Table 4; the machines are a subset of those in Table 5 (both in Appendix A). For each application all the running times are normalized by the slowest time over all the machines. Since the machines shown are only a subset of those on which we collected runtimes, the highest bar is not at 1 for every application. While some machines are generally faster than others, no machine is fastest (or slowest) on all the applications. This suggests performance is not a function of any single metric.

Still, in theory, the time needed to run an application should be a function of just machine and application characteristics, both plotted in Figure 2. Our goal is to determine whether there is a straightforward way to use those characteristics to rank supercomputers based on their performance on real applications. While existing methods for predicting parallel performance (eg, [1, 8, 14]) could be used to rank machines, they are designed primarily to help users optimize their code and not to predict performance on different machines. As a result, they are application-specific and typically complex to build [16].

In this paper we describe a metric for evaluating the quality of a proposed ranking. We show that if the machines are ranked using only results from simple benchmarks, those that measure the bandwidth to main memory and the latency to L1 cache are significantly better predictors of relative performance than peak flops. We next show how application traces can be used to improve rankings, without resorting to detailed performance prediction.



(a) (b)

Figure 2: (a) A plot of machine characteristics for a subset of the systems in Table 5. (b) A plot of application characteristics for a subset of the applications in Table 4. For both the highest count for each characteristic is normalized to 1.

2 Ranking using machine metrics

Ideally we would be able to perfectly rank all machines in an application-independent manner, using only machine characteristics. Although this may be impossible, we can still measure how close any ranking comes to perfect. In this section we describe a metric for ranking defined by *thresholded inversions*. We describe the optimal ranking for that metric on our combination of machines and applications, and explore how close we can come to optimal using machine characteristics both individually and in combination.

2.1 Measuring the quality of a ranking

We evaluate the quality of a ranking by the number of thresholded inversions that it contains. For example, consider a list of n systems $m_1, m_2, \ldots m_n$, sorted from highest to lowest interprocessor network bandwidth so that $i < j$ means that the bandwidth achievable on system m_i is at least as good as the bandwidth achievable on system m_j. Given an application A and times $t(1), t(2), \ldots, t(n)$, where $t(k)$ is the measured running time of A on machine m_k, we would normally say that machines m_i and m_j are inverted if $i < j$, but $t(i) > t(j)$. Intuitively, this says that all other things being equal, application A should run faster on the machine with the higher network bandwidth. If it does not, we call that an inversion. The number of inversions in the ranking, then, is the number of pairs of machines that are inverted[1]; in our example this is the number of pairs of machines for which the inter-processor network bandwidth incorrectly predicts which machine should run faster on application A. Clearly this number is at least 0 and is no larger than $n(n-1)/2$.

[1] List inversions are also used in other fields, for example to compare the results returned by different queries to a database, and are related to the statistical measure Kendall's tau.

In practice, to allow for variance in the measured runtimes, we count only *thresholded* inversions. Given a threshold α satisfying $0 \leq \alpha \leq 1$, machines m_i and m_j are considered inverted only if m_i is predicted to be faster than m_j, and $t(i) > (1+\alpha)t(j)$. If $\alpha = 0$, this is equivalent to the number of inversions as described in the previous paragraph. However, choosing a small, nonzero α allows us to handle variations in timing measurements that can be caused for reasons including those discussed in [9].

Furthermore, to allow for variance in the benchmark timings that give the machine metrics, we use a second threshold parameter β, where $0 \leq \beta \leq 1$. Even if two machines m_i and m_j would be considered inverted by the above measures because $t(i) > (1+\alpha)t(j)$, we only count the inversion as such if the results of the benchmark (e.g., network bandwidth) also differ by a relative measure β. In general β should be chosen to be smaller than α because one expects less variance in benchmark times than in full application runtimes. In this paper we use $\alpha = .01$, which means a difference of up to 1% in the application runtimes is considered insignificant, and $\beta = .001$.

Note that a metric based on thresholded inversions is practical because it is monotonic in the sense that adding another machine m_k and its associated runtime $t(k)$ cannot decrease the number of inversions in a ranking. Within our context of large parallel applications, this is useful because we have only partial runtime data: not all of the applications in Table 4, with all possible processor counts, were run on all the machines in Table 5. In some cases these gaps are necessary because some systems do not have enough compute processors to run every application with all processor counts. Furthermore, in practice, it is not unusual for timings that have been collected at different times on different machines by different people to be incomplete in this way.

2.2 Rankings using single machine metrics

We now consider the quality of the rankings produced by using most of the machine characteristics summarized in Table 6: peak flops, interprocessor network bandwidth, interprocessor network latency, bandwidth of strided and of random accesses to L1 cache, bandwidth of strided and of random accesses to L2 cache, and bandwidth of strided and of random accesses to main memory. We did not generate rankings based on the bandwidth of strided and random accesses to L3 cache because not all of the machines have L3 caches. All these characteristics are measured by simple benchmark probes, also summarized in Table 6. The fundamental difference between strided and random memory references is that the former are predictable, and thus prefetchable. Because random memory references are not predictable, the bandwidth of random accesses actually reflects the latency of an access to the specified level of the memory hierarchy.

Table 1 sums the number of thresholded inversions over all the application and processor counts. Because each application is run on a different set of processor counts and not every case has been run on every machine, the numbers in Table 1 should not be compared across applications, but only for a single application across the rankings by different machine characteristics.

The last row of Table 1 shows that the bandwidth of strided accesses to main memory provides the single best overall ranking, with 309 total thresholded inversions. The ranking generated by the bandwidth of random accesses to L1 cache is a close second. However, the data also show that no single ranking is optimal for all applications: whereas the bandwidth of strided accesses to main memory is nearly perfect for avus, and does very well on wrf and cth7, it is outperformed by the bandwidth of both strided and random accesses to L1 for ranking performance on gamess. It is also worth noting that flops is only the best predictor of rank on lammps. One interpretation of the data is that these applications fall into three categories:

Metric	l1(1)	l1(r)	l2(1)	l2(r)	mm(1)	mm(r)	1/lat	nw bw	flops
avus	51	26	44	42	**1**	61	19	30	22
cth7	32	**18**	30	82	21	117	63	37	35
gamess	25	**16**	40	55	48	76	65	35	25
hycom	26	**10**	26	83	17	126	65	28	35
lammps	136	107	133	93	80	157	95	116	**68**
oocore	44	**31**	56	71	61	91	75	50	52
overflow2	71	**39**	79	91	47	104	108	81	44
wrf	99	63	92	134	**34**	203	103	83	60
overall sum	484	310	500	651	**309**	935	593	460	341

Table 1: Sum of the number of thresholded inversions for all processor counts for each application, with $\alpha = .01$ and $\beta = .001$. The smallest number (representing the best metric) for each application is in **bold**. The last row is a sum of each column and gives a single number representing the overall quality of the ranking produced using that machine characteristic.

1. codes dominated by time to perform floating-point operations,
2. codes dominated by time to access main memory,
3. and codes dominated by time to access L1 cache.

2.3 Using combined machine metrics

If we ask how well *any* fixed ranking can do for these particular applications, an exhaustive search through the viable best ranking space reveals a ranking where the sum of the number of inversions is 195, with $\alpha = .01$ and $\beta = 0$. [2] Although this choice of β is stricter than using $\beta = .001$, the choice is unavoidable since it is unclear what the metric numbers should be. While the optimal ranking is significantly better than any of the ones in Table 1 generated using single machine characteristics, exhaustive search is an unrealistic methodology. In addition to the cost, adding a new machine or application to the ranking would require re-evaluating the ranking, no intuition can be gained from the ranking, and the result does not track to any easy-to-observe characteristic of machines or applications.

3 Rankings using application metrics

Incorporating application characteristics into our rankings can be done by computing a weighted sum of the machine characteristics. Intuitively the weights are a measure of how much an application uses that particular machine resource.

Although this allows for different rankings of machines for different applications (since each application has different characteristics), we focus on trying to find a single representative application whose characteristics can be used to generate a single "reasonable" ranking for all 8 of the applications in our test suite. Later, in Section 3.2, we discuss application-specific rankings.

[2] In comparison, the worst ranking we saw had 2008 inversions. A random sample of 100 rankings had an average of 1000 inversions with a standard deviation just over 200.

3.1 Ranking using simple metric-products

We first try using only information gathered on the applications' memory behavior (gathered using the Metasim Tracer [4]), ignoring interprocessor communication. While we test the ranking generated by data collected for each application, we are looking for a fixed ranking that can be used across all applications. Hence, we only report on the best ranking. This represents the result of taking the characteristics of a single application and using it to generate a ranking of the machines that can be used across a range of applications.

3.1.1 Strided/Random memory accesses and flops

After determining that simply counting the number of floating point operations and total memory accesses did not generate reasonable rankings, we tried partitioning the memory accesses. We partition m into $m = m_1 + m_r$, where m_1 is the number of strided accesses and m_r is the number of random accesses to memory. This is appealing because it may be possible to do this division in an architecture independent way.

Since there is no established standard for what it means to partition memory accesses between strided and random, we use a method based on the Metasim tracer [4]. This method partitions the code for an application into non-overlapping basic blocks and then categorizes each basic block as exhibiting primarily either strided or random behavior. For the results in this paper we classify each basic block using two methods — if either decides the block contains at least 10% random accesses, we categorize all memory accesses in that block as random. The number 10% is somewhat arbitrary, but is based on the observation that on many machines the sustainable bandwidth of random accesses is less than the sustainable bandwidth of strided accesses by an order of magnitude.

We determine if memory accesses are random using both a dynamic analysis method and a static analysis method. The first uses a trace of the memory accesses in each basic block and considers each access to be strided if there has been an access to a sufficiently nearby memory location within a window consisting of some small number of immediately preceding memory accesses. The advantage of a dynamic approach is that every memory access is evaluated, so nothing is overlooked. The disadvantage is that the size of the window must be chosen carefully. If the size is too small, we may misclassify some strided accesses as random. If the size is too large, the process becomes too expensive computationally. In contrast, the static analysis method searches for strided references based on an analysis of dependencies in the assembly code. Static analysis is less expensive than dynamic analysis and also avoids potentially misclassifying accesses due to a window size that is too small. On the other hand, static analysis may miss some strided accesses because of the difficulty of analyzing some types of indirect accesses. Since the two types of analysis are predisposed to misclassify different types of strided accesses as random, we apply both methods and consider an access to be strided if either method predicts it to be so.

Having partitioned the memory accesses into random (m_r) and strided (m_1), we use Equation 1 to compute the numbers $r_1, r_2, \ldots, r_n$ from which the ranking is generated:

$$r_i = \frac{8m_1}{\text{bw_mem}_1(i)} + \frac{8m_r}{\text{bw_mem}_r(i)} + \frac{f}{\text{flops}(i)}. \tag{1}$$

Notice that we must choose what to use for bw_mem_1 and bw_mem_r. Options include the bandwidths of strided and random accesses to main memory; the bandwidths of strided and random accesses to L1 cache; or, considering the data in Table 1, the bandwidth of strided access to main

Metric	l1(1,r)	mm(1,r)	mm(1), l1(r)
avus	12	21	**9**
cth7	14	80	**9**
gamess	**16**	77	26
hycom	**2**	44	**2**
lammps	107	148	**81**
oocore	**31**	100	44
overflow2	**34**	78	**34**
wrf	63	158	**44**
overall sum	279	706	**249**

Table 2: Sum of the number of thresholded inversions for all numbers of processors for each application, with $\alpha = .01$ and $\beta = .001$. The smallest number (representing the best metric) for each application is in **bold**.

memory and of random access to L1 cache. Furthermore, since we are looking for a single, fixed ranking that can be applied to all applications, we also need to consider which application's m_1, m_r, and f to use in the ranking. In theory we could also ask what processor count of which application to use for the ranking; in practice, we only gathered m_1 and m_r counts for one processor count per application.

Table 2 shows the results of these experiments. Each column shows the number of thresholded inversions for each of the 8 applications using the specified choice of strided and random access bandwidths. In each column the results use the application whose m_1, m_r, and f led to the smallest number of inversions. When using the bandwidths L1, the best application was overflow2; when using the bandwidths to main memory, the best application was oocore; and when using a combination of L1 and main memory bandwidths, avus and hycom generated equally good rankings.

Comparing the results in Table 2 to those in Table 1, we see that partitioning the memory accesses is useful as long as the random accesses are considered to hit in L1 cache. When we use the bandwidth of accesses to main memory only, the quality of the resulting order is between those of rankings based on the bandwidth of random accesses and based on the bandwidth of strided accesses to main memory. Using the bandwidth of random access to L1 cache alone did fairly well, but the ranking is improved by incorporating the bandwidth of strided accesses to L1 cache, and is improved even more by incorporating the bandwidth of strided accesses to main memory.

We believe the combination of mm(1) and l1(r) works well because of a more general fact: applications with a large memory footprint that have many strided accesses benefit from high bandwidth to main memory because the whole cache line is used and prefetching further utilizes the full main memory bandwidth. For many of these codes main memory bandwidth is thus the limiting performance factor. On the other hand, applications with many random accesses are wasting most of the cache line and these accesses do not benefit from prefetching. The performance of these codes is limited by the latency hiding capabilities of the machine's cache, which is captured by measuring the bandwidth of random accesses to L1 cache.

3.1.2 Using more detailed application information

Two changes that might improve the ranking described above are partitioning memory accesses between the different levels of the memory hierarchy and allowing different rankings based on the processor count. The two possibilities are not entirely independent since running the same size problem on a larger number of processors means a smaller working set on each processor and therefore different cache behavior. However, allowing different rankings for different processor counts takes us away from the original goal of finding a single fixed ranking that can be used as a general guideline.

This leaves us with partitioning memory accesses between the different levels of the memory hierarchy. Unfortunately, as noted previously, this requires us to either choose a representative system or to move towards a more complex model that allows for predictions that are specific to individual machines, as is done in [3, 11]. Therefore, given the level of complexity needed for a ranking method that incorporates so much detail, we simply observe that we achieved a ranking with about 28% more thresholded inversions than the brute-force obtainable optimal ranking on our data set without resorting to anything more complex than partitioning each application's memory accesses into strided and random accesses. This represents a significant improvement over the ranking based on flops, which was about 75% worse than the optimal ranking.

Now we shift gears and examine how well current performance prediction methodologies rank the relative performance of different machines.

3.2 Ranking and performance prediction

Up to this point our focus has been on generating a single machine ranking that is evaluated across multiple applications run on several different processor counts. In this section we ask how much better we can do by allowing rankings to vary as a function of the application and the processor count. Of course, if we ran all the applications on all the machines, we could then use the actual runtimes to generate an ideal ranking for each application. In practice, for reasons noted previously, such complete runtime data is almost never available. However, we could use any of a number of performance prediction methodologies and then generate machine rankings based on the predicted runtimes. This approach has the advantage of avoiding gaps in the data and also has the potential for allowing designers to rank systems that have yet to be built.

If we could predict performance exactly, then these rankings would be perfect; in practice, performance predictions are never exact. Furthermore, because methodologies can both overpredict and underpredict the real running time, a more accurate performance prediction methodology might not lead to better rankings. In this section we explore the quality of rankings generated through the performance prediction methodologies described in [3], where the authors evaluate 9 increasingly complex and increasingly accurate ways of using combinations of metrics to predict real runtimes of applications on supercomputers.

Table 3 gives the number of thresholded inversions between the predicted and measured runtimes in [3]. Note that their dataset is different from the one used in the previous sections of this paper: we use 8 applications to their 4, far more processor counts for each application, and 14 machines to their 10. Because of this, the numbers in Table 3 cannot be directly compared with those elsewhere in this paper. However, because the question is how well performance prediction strategies generate application *dependent* rankings, an exact comparison is neither meaningful nor desirable.

We observe that the first 3 cases in [3] should produce rankings that are equivalent to using only

Methodology	1	2	3	4	5	6	7	8	9
# thresh. inversions	165	86	115	165	76	53	55	44	44

Table 3: Sum of the number of thresholded inversions for all applications and numbers of processors, for each of the 9 performance prediction strategies described in [3]. For the measured runtimes we use $\alpha = .01$; for the predicted runtimes we use $\beta = .001$.

flops, only the bandwidth of strided accesses to main memory, and only the bandwidth of random accesses to main memory, respectively. We see that the ranking based on the bandwidth of strided access to main memory is the best of the three, which agrees with the data in our Table 1. As expected from the description in [3], their first and fourth cases produce equivalent rankings.

Case 5 is similar to a ranking based on a simplified version of Equation 1. Case 6 is similar to a ranking based on our Equation 1 using the bandwidths of strided and random accesses to main memory for bw_mem_1 and bw_mem_r, with a different method for partitioning between strided and random accesses. In Table 3 both of these rankings are significant improvements over those produced by the first four cases; however, in our experiments, little improvement was observed. In addition to the fact that we use a more diverse set of applications and a larger set of both processor counts and of machines, a more significant difference is that in [3] the authors change the values of m, m_1, m_r, and f in Equation 1 depending on the application whose running time they are predicting. Not surprisingly, these application dependent rankings outperform application independent rankings.

Cases 7 through 9 in [3] involve considerably more complex calculations than those used in this paper; as expected, they also result in more accurate rankings. It seems a better ranking methodology could certainly resemble better prediction methodology.

4 Conclusion

The Top 500 list is popular in part because it is easy to read, is based on a simple metric that is easy to measure (essentially peak flops), and is easy to update on a regular basis. Any viable alternative must maintain these strengths, but also more accurately rank supercomputer performance on real applications. In particular, no one would be interested in reading, contributing to, or maintaining, a ranking of 500 machines on numerous HPC applications that was generated by brute force (as was the optimal ranking we generated for our dataset in Section 2.3).

In this paper we show that rankings generated using simple benchmarks that measure either the bandwidth of strided accesses to main memory, or the bandwidth of random accesses to L1 cache, are better than the ranking generated by measuring only flops. Furthermore, we show how a combination of the above three metrics can be combined with a small amount of application-specific information in order to significantly improve on a ranking based solely on peak floating point. We are currently looking at the effects of incorporating information about the communication capabilities of different systems into our rankings. This is part of ongoing work towards better understanding the boundary between ranking and performance prediction, which we are beginning to explore in this paper.

We note that exploring the possibility of generating different rankings for different target applications could lead naturally into a study of the relationship between performance prediction and

ranking.

5 Acknowledgments

We would like to thank Michael Laurenzano and Raffy Kaloustian for helping to collect trace data; and Xiaofeng Gao for writing the dynamic analysis tool used in Section 3.1.2. We would like to thank the CRA-W Distributed Mentor Project. We would also like to acknowledge the European Center for Parallelism of Barcelona, Technical University of Barcelona (CEPBA) for their continued support of their profiling and simulation tools. This work was supported in part by a grant of computer time from the DoD High Performance Computing Modernization Program at the ARL, ASC, ERDC, and NAVO Major Shared Resource Centers, the MHPCC Allocated Distributed Center, and the NRL Dedicated Distributed Center. Computer time was also provided by SDSC. Additional computer time was graciously provided by the Pittsburgh Supercomputer Center via an NRAC award. This work was supported in part by a grant from the National Science Foundation entitled "The Cyberinfrastructure Evaluation Center", and by NSF grant #CCF-0446604. This work was sponsored in part by the Department of Energy Office of Science through SciDAC award High-End Computer System Performance: Science and Engineering, and through the award entitled HPCS Execution Time Evaluation.

References

[1] B. Armstrong and R. Eigenmann. Performance forecasting: Towards a methodology for characterizing large computational applications. In *ICPP '98: Proceedings of the 1998 International Conference on Parallel Processing*, pages 518–526, Minneapolis, Minnesota, August 1998.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[3] L. Carrington, M. Laurenzano, A. Snavely, R. Campbell, and L. Davis. How well can simple metrics predict the performance of real applications? In *Proceedings of Supercomputing (SC|05)*, November 2005.

[4] L. Carrington, A. Snavely, N. Wolter, and X. Gao. A performance prediction framework for scientific applications. In *Proceedings of the International Conference on Computational Science (ICCS 2003)*, Melbourne, Australia, June 2003.

[5] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15:1–18, 2003.

[6] J. L. Gustafson and R. Todi. Conventional benchmarks as a sample of the performance spectrum. *The Journal of Supercomputing*, 13(3):321–342, 1999.

[7] IDC balanced rating. http://www.hpcuserforum.com.

[8] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of Supercomputing (SC 2001)*, Denver, Colarado, November 2001.

[9] W. T. C. Kramer and C. Ryan. Performance variability of highly parallel architectures. In *Proceedings of the International Conference on Computational Science (ICCS 2003)*, Melbourne, Australia, June 2003.

[10] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction the the HPC challenge benchmark suite. Available at http://www.hpccchallenge.org/pubs/, March 2005.

[11] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of SIGMETRICS/Performance'04*, New York, NY, June 2004.

[12] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. IEEE Technical Committee on Computer Architecture Newsletter, December 1995.

[13] Department of Defense High Performance Computing Modernization Program. Technology insertion - 06 (TI-06). http://www.hpcmo.hpc.mil/Htdocs/TI/TI06, May 2005.

[14] J. Simon and J. Wierum. Accurate performance prediction for massively parallel systems and its applications. In *Proceedings of 2nd International Euro-Par Conference*, Lyon, France, August 1996.

[15] SPEC: Standard performance evaluation corporation. http://www.spec.org, 2005.

[16] D. Spooner and D. Kerbyson. Identification of performance characteristics from multi-view trace analysis. In *Proceedings of the International Conference on Computational Science (ICCS 2003)*, Melbourne, Australia, June 2003.

[17] STREAM: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream.

[18] Top500 supercomputer sites. http://www.top500.org.

A Tables

Name	Description	Processor counts
avus	CFD calculations on unstructured grids	32, 64, 96, 128, 192, 256, 384
cth7	effects of strong shock waves	16, 32, 64, 96
gamess	general ab-initio quantum chemistry	32, 48, 64, 96, 128
hycom	primitive equation ocean general circulation model	24, 47, 59, 80, 96, 111, 124
lammps	classical molecular dynamics simulation	16, 32, 48, 64, 128
oocore	out-of-core solver	16, 32, 48, 64
overflow2	CFD calculations on overlapping, multi-resolution grids	16, 32, 48, 64
wrf	weather research and forecast	16, 32, 48, 64, 96, 128, 192, 256, 384

Table 4: The applications used in the study and the number of processors on which each was run.

HPC lab location	Processor	Interconnect	# of compute processors
ARL	SGI-03800-0.4GHz	NUMACC	512
ARL	LNX-Xeon-3.6GHz	Myrinet	2048
ARSC	IBM-690-1.3GHz	Federation	784
ASC	SGI-03900-0.7GHz	NUMACC	2032
ASC	HP-SC45-1.0GHz	Quadrics	768
ERDC	SGI-O3900-0.7GHz	NUMACC	1008
ERDC	HP-SC40-0.833GHz	Quadrics	488
ERDC	HP-SC45-1.0GHz	Quadrics	488
MHPCC	IBM-690-1.3GHz	Colony	320
MHPCC	IBM-P3-0.375GHz	Colony	736
NAVO	IBM-655-1.7GHz	Federation	2832
NAVO	IBM-690-1.3GHz	Colony	1328
NAVO	IBM-P3-0.375GHz	Colony	736
SDSC	IBM-IA64-1.5GHz	Myrinet	512

Table 5: Systems used in this study.

Probe name	DoD TI06 Benchmark suite	Machine property measured
flops	CPUBENCH	peak rate for issuing floating-point operations
L1 bw(1)	MEMBENCH	rate for loading strided data from L1 cache
L1 bw(r)	”	rate for loading random stride data from L1 cache
L2 bw(1)	”	rate for loading strided data from L2 cache
L2 bw(r)	”	rate for loading random stride data from L2 cache
L3 bw(1)	”	rate for loading load strided data from L3 cache
L3 bw(r)	”	rate for loading random stride data from L3 cache
MM bw(1)	”	rate for loading strided data from main memory
MM bw(r)	”	rate for loading random stride data from main memory
NW bw	NETBENCH	rate for sending data point-to-point
NW latency	”	startup latency for sending data point-to-point

Table 6: Probes run as part of DoD benchmarking.