

Performance Complexity: An Execution Time Metric to Characterize the Transparency and Complexity of Performance

Erich Strohmaier

Future Technology Group
Lawrence Berkeley National Laboratory
estrohmaier@lbl.gov

Abstract

Performance evaluation of code execution focuses on determining performance and efficiency levels for specific application scenarios. However, there is no measure characterizing how complex it is to achieve performance and how transparent performance results are. In this paper we present an execution time metric called *Performance Complexity (PC)* to capture these important aspects. *PC* is based on performance results from a set of benchmark experiments and related performance models reflecting the behavior of a program. Residual modeling errors are used to derive *PC* as a measure for how transparent program performance is and how complex the performance appears to the programmer. We present a detailed description for calculating compatible *P* and *PC* values and use results from a parametric benchmark to illustrate the utility of *PC* for analyzing systems and programming paradigms.

1 Introduction

During the last decades it has become more difficult to achieve high performance efficiency, portability, and scalability on computer systems. It is becoming increasingly unclear how more complex hardware and software features affect performance and scalability. As we optimize our codes for particular system features, code development time increases and the achieved performance levels are less portable. Coding complexity is also increasing due to insufficient features in programming languages for parallel systems. These problems have been widely recognized and the DARPA HPCS (High Productivity Computing Systems) program is addressing them directly by funding research into new, more productive parallel programming paradigms [1].

Evaluation of code execution has traditionally focused on determining absolute and relative performance and efficiency levels for specific applications. A commonly accepted method uses a set of benchmarks selected to represent a particular workload of interest to measure absolute and relative performance. However, there is no methodology to quantify the transparency of the performance. If we understand the performance behavior of a system, performance sensitive programming is potentially easy. If performance is not transparent, programming becomes difficult and complex. Therefore, a measure for *performance transparency* can also be considered a measure for *programming complexity*.

In this paper we introduce such a code execution metric. We use the accuracy of performance models to derive a quantitative measure for *Performance Complexity - PC*. In the general case, a set of appropriately chosen performance models, is developed for each benchmark in a benchmark set. The residual modeling errors are used to derive measures for how well performance is captured by the models. Performance Complexity is the geometric standard deviation of measured performances relative to predicted performance values.

Developing performance models for a full set of benchmarks can be time consuming. As a first step we use a single tunable synthetic benchmark Apex-MAP [2] and a set of simple performance models. Different descriptive parameter value sets of Apex-MAP are used as replacement of different benchmarks. We also use a set of different performance models to study the influence of model selection on the values of *PC*. We find that our results reflect intuitive qualitative expectations for relative *PC* values surprisingly well.

The rest of this paper is organized as follows: In section 2 we introduce and describe our concept for *Performance Complexity* in detail, section 3 is

a brief introduction to Apex-MAP, in section 4 we develop the used performance models, section 5 and 6 describe results for serial and parallel execution, section 7 discusses related work, and we present our conclusions and future work in section 8.

2 How to characterize Performance Complexity and Transparency

Performance of a system can only be measured relative to a workload description. This is usually achieved by selecting a set of benchmarks representative of a particular workload of interest. Beyond this, there cannot be a meaningful, absolute performance measure valid for all possible workloads. The complexity of programming codes on a given system also depends on the set of codes in question. Therefore, performance complexity (PC) can be defined only relative to a workload and not in absolute terms!

The situation is even more complex when we consider coding the same algorithm in different languages and with different programming styles.

Programming for performance is easier if we understand the influence of system and code features on performance behavior and if unexplained performance variations are small. The level of performance transparency does, however, not indicate any level of performance itself. It only specifies, that we understand performance. This understanding of performance behavior of a code on a system implies that we can develop an accurate performance model for it. Therefore our approach is to use the accuracy of a suitable set of performance models to quantify the performance complexity (PC).

In the ideal case the selected performance model should incorporate the effects of all code features easily controllable in a given programming language and not include any architectural features inaccessible to the programmer. In essence, the performance models for calculating a PC value should reflect the (possible) behavior of a programmer, in which case PC reflect the performance transparency and complexity of performance control this programmer will experience.

Goodness of fit measures for modeling measurements are based on the Sum of Squared Errors (SSE). For a system S_i the performance P_{ij} of a set of n codes C_j is measured. Different measurements j might also be obtained with the same code executed with different problem parameters such as

Operation	Dimension*
Initial data: P_i, M_i	[ops/sec]
Transform P_j, M_j to ideal flat metric	[ops/cycle]
Log-transformation: $P'_i = \log(P_i), M'_i = \log(M_i)$	[log(ops/cycle)]
Basic Calculations	
$\bar{P}' = \frac{1}{n} \sum_j P'_j$	[log(ops/cycle)]
$\overline{SS}' = \frac{1}{n} \sum_j (P'_j - \bar{P}')^2$	[(log(ops/cycle)) ²]
$\overline{SSE}' = \frac{1}{n} \sum_j (P'_j - M'_j)^2$	[(log(ops/cycle)) ²]
$1 - R^2 = \frac{\overline{SSE}'}{\overline{SS}'}$	[]
Back-Transformations	
$\bar{P} = \exp(\bar{P}') = \sqrt[n]{\prod_j P_j}$	[ops/cycle]
Absolute $PC_a = \exp(\sqrt{\overline{SSE}'}) - 1$	[ops/cycle]
Relative $PC_r = \exp\left(\sqrt{\frac{\overline{SSE}'}{\overline{SS}'}}\right) - 1$	[]
Back-transformation of \bar{P}, PC_a to original scale and metrics	[ops/sec]

Table 1: Calculations steps for the values of average performance P and performance complexity PC (absolute and relative) along with a dimensional analysis. The index i for different system is suppressed.

*Inverse dimensions could be chosen as well.

problem sizes and resource parameters such as concurrency levels. A performance model M_{kl} is used to predict performance M_{ikl} for the same set of experiments.

To arrive at a PC measure with the same metric as performance, we have to use standard deviation (SD), which is the square-root of average SSE and cannot use the more widely used total sum. The average SSE_{kl} for each system i and model k across all codes j is then given by

$$\overline{SSE}_{ik} = \frac{1}{n} \sum_j (P_{ij} - M_{ijk})^2.$$

While these SSE values are still absolute numbers (they carry dimension), they are easily transformed into relative numbers by dividing them by the similarly defined Sum of

Squares (SS) of the measured data P : $\overline{SS} = \sum_k (P_k - \overline{P})^2$ and $R^2 = \overline{SSE} / \overline{SS}$.

To further reduce the range of performance values a log-transformation should be applied to the original data. This effectively bases the calculated SSE values on relative errors instead of absolute errors. To obtain numbers in regular dimensions, we transform the final values of SSE back with an exponential function. The resulting numbers turn out to be known as geometric standard deviation, representing multiplicative instead of additive, relative values, which are larger or equal to one. For convenience we transform these values into the usual range of larger than zero by subtracting one. The full sequence of calculation step is summarized in Table 1.

The outlined methodology can be applied equally to absolute performance metrics or to relative efficiency metrics. PC_a represents an absolute metric while PC_r would be the equivalent, relative efficiency metric. PC_r is in the range $[0,1]$ and reflects the percentage of the original variation not resolved in the performance model. Hence, while using PC_r care has to be taken when comparing different systems, as a system with larger relative complexity PC_r might have lower absolute complexity PC_a . This is the same problem as comparing the performance of different systems using efficiency metrics such as ops/cycle or speedup.

3 Apex-MAP

Apex-MAP is a synthetic benchmark generating global address streams with parameterized degrees of spatial and temporal locality. Along with other parameters, the user specifies L and α , parameters related to spatial locality and temporal reuse respectively. Apex-MAP then chooses indices into a data array that are distributed according to α , using a non-uniform random number generator. The indices are most dispersed when $\alpha=1$ (uniform random distribution) and become increasingly crowded toward the starting address as α approaches 0. Apex-MAP then performs L stride 1 references starting from each index. Apex-MAP has been used to map the performance of several systems with respect to L and α (see Figure 1).

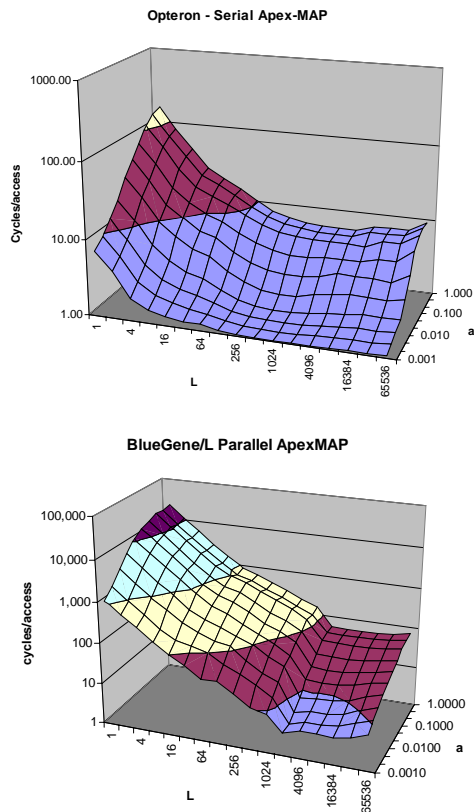


Figure 1: Example results from parameter sweeps of Apex-MAP. Note the change of scale for access times from 2 to 4 magnitudes. L represents spatial and α temporal locality.

4 Performance Models and Modeling Methodology used

In our methodology the selection of the performance models is as important as the selection of the benchmarks. It is widely accepted that there cannot be a single measure for performance, which does not relate to specific codes. Likewise, performance transparency must be related to specific codes. Embarrassingly parallel codes typically will exhibit only small performance complexity, while this is not the case for tightly coupled, irregular scientific problems. In addition the performance complexities programmers experience on a system, also depends on the programming languages and coding styles they use.

Ideally, the features of the performance models should reflect characteristics of our programming languages, which the user can easily control and use to influence performance behavior. An

example would be vector-length as it is easily expressed in most languages as a loop-count or an array-dimension. Unfortunately many programming language do not complement system architectures well, as they do not have appropriate means of controlling hardware features. This situation is exacerbated as many hardware features are actually designed to not be user controllable, which makes performance optimization often a painful exercise in trial and error. Cache usage would be a typical example here as programming languages have little means to control directly which data should reside in the cache or not. In developing performance models we often have to revert to using such non-controllable hardware features.

Apex-MAP is designed to measure the dependency of global data access performance on spatial and temporal locality. This is achieved by a sequence of blocked data accesses of unit stride with a pseudo-random starting address. From this we can expect, that any good performance model for it should contain the features of access length and for access probabilities to different levels of memory hierarchy. The former is a loop length and easily controlled in programs, the later depends on cache hit-rates, which are usually not directly controllable by the programmer. The metric for Apex-MAP performance is [data-access/second] or any derivative thereof.

We use four models to characterize serial benchmark execution and parallel execution of Apex-MAP. The simplest model represents an ideal system on which all data accesses take equal time (flat global memory). Our second model assumes a two level memory hierarchy, with constant access time for each level. M is the global memory utilized by Apex-MAP and c the amount of memory in the faster second level. The probability to find data in c can be derived from the properties of the temporal locality parameter α as $(c/M)^\alpha$. The third model assumes access time depends linearly on block length L with two parameters for latency l and gap g (inverse bandwidth). The fourth model finally combines the previous two by assuming a linear timing model for each level in a hierarchical model. All models and their timing equations are shown in Table 2.

Performance models should be calibrated with hardware specifications or low-level micro-benchmarks. Due to the simplicity of our models, predictions based on hardware specifications can

Model	Time per access	Parameters
0	Flat memory $T = g = \text{constant}$	g
1	Two level memory $T = P(c/M)*g_1 + (1-P(c/M))*g_2$	g_1, g_2
2	Latency, Bandwidth $T = (l+g*(L-1))/L$	l, g
3	L, B for two levels $T = P(c/M)*(l_1+g_1*(L-1))/L + (1-P(c/M))*(l_2+g*(L-1))/L_2$	l_1, g_1, l_2, g_2
Cache-hit probability: $P(c/M) = (c/M)^\alpha$		

Table 2: Performance models used for the average time per data access of Apex-MAP and their performance parameters. M , L and α are Apex-MAP parameters and c is a system parameter reflecting the most important memory or system hierarchy level, which has to be set to different appropriate levels for serial and parallel models.

be expected to be off by large margins. We use a back-fitting procedure to minimize SSE and the prediction error of the models. To check the validity of the models we inspect residual error plots and fitted latency and gap values to rule out any fictitious models or parameter values.

For complex parallel system hierarchies it is not always clear what the second most important hierarchy level is and what the value of c should be. It is therefore advisable to at least use a backfitting approach to confirm an initial choice, or to probe the performance signature of a system, to determine which level is most influential on performance.

5 Serial Execution

As a first example, we calculate the PC values of our four performance models for the serial performance measurements of Apex-Map[2][3]. The memory size used was 512MB and we swept across 10 α and 17 L values. Figure 2 shows for model 0 that performance variation is the highest on the vector processors and the lowest on the PowerPC processor in BlueGene/L, which has a relatively flat memory hierarchy compared to the other superscalar processors. This model reflects the attitude of a programmer, who does not want to take any consideration of performance relevant features in his/her coding style.

The same is basically true for model 1. Complexity values are unchanged on vector processors. PC is slightly reduced for superscalar processor,

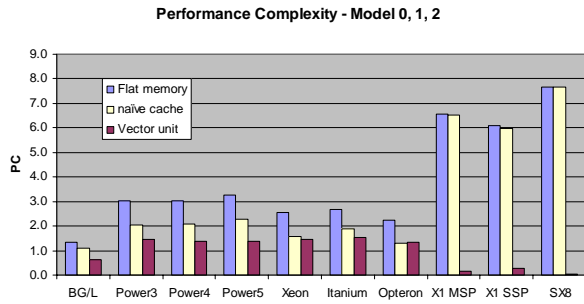


Figure 2: PC values derived with three simple models for serial execution and based on efficiencies [accesses/cycle].

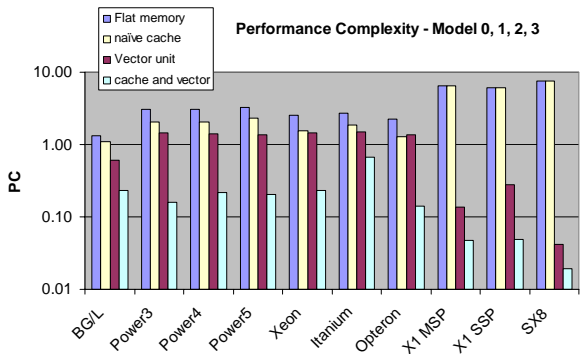


Figure 3: PC values derived with the combined model 3 for serial execution and based on efficiencies [accesses/cycle].

but not as much as one might expect. Inspection of residual error indicates that this is in part due to not considering the effects of fractional cache-line usage for very short L values, which increases error in this parameter region greatly. This model does also not capture any more advanced memory access feature on modern superscalar processors such as pre-fetching.

PC values for model 2, which accounts for such effects are overall substantially better than for the previous models. While this is no surprise at all on vector system, it is a strong indication of the importance of such features as pre-fetching on the performance of superscalar architectures. This model represents a programmer, who is willing to structure their code with long for loops to improve performance.

PC values for the combined model 3 are only visible and shown in Figure 3 where we switched to a logarithmic vertical scale. Clearly, the performance of Apex-MAP is overall much better re-

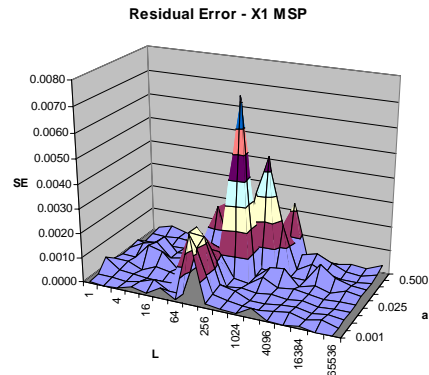
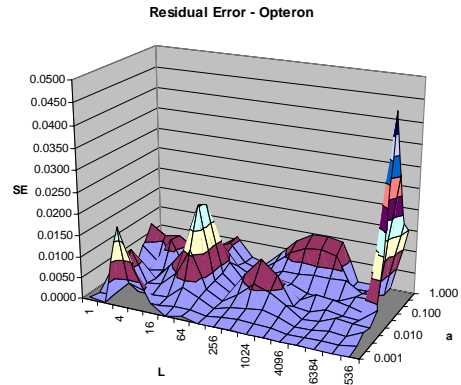
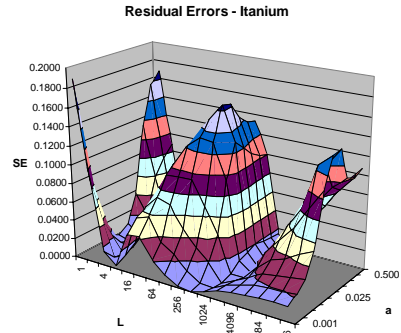


Figure 4: Examples for residual squared errors (SE) of model 3 for Itanium, Opteron, and Cray MSP. Note the differences in scale.

solved in this model. This model reflects a programmer who codes for long loops as well as high cache re-use, where the later feature unfortunately is not easily controllable. The final ranking of processor with PC values gives the NEC SX8 with the lowest complexity followed by the Cray X1 in MSP, and then SSP mode, Opteron, Power3, Power5, Power4, Xeon, PowerPC, and Itanium. Different individuals and groups interviewed have produced very similar rankings of these processors

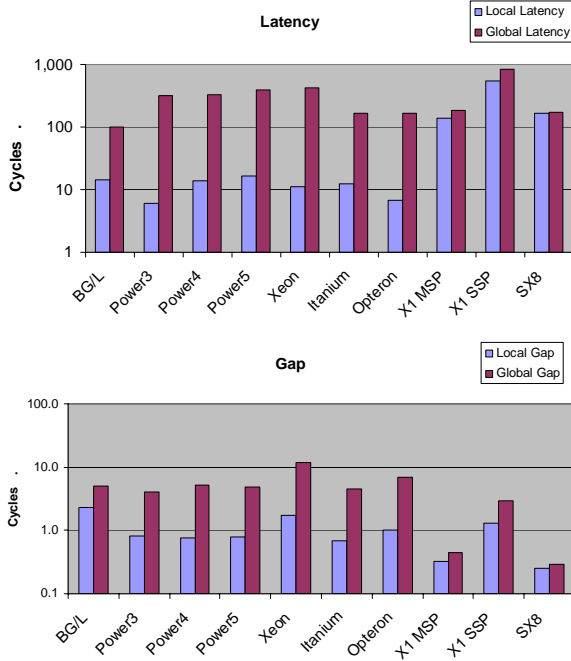


Figure 5: Back-fitted latency and gap values for both memory hierarchies in model 3.

on several occasions. This indicates, that our methodology can produce PC values, which fit intuitive expectations well.

The only processor with highly unresolved PC is the Itanium. Inspecting residual errors in Figure 4, we see high errors for the medium range of loop length, which indicates a more complex behavior, perhaps due to peculiarities of pre-fetching. Residual errors for the Opteron and Cray MSP are on a much smaller scale. For the Opteron there is no discernible structure, while residual on the X1 are somewhat larger for the range of vector length equal to the vector register length.

Latency l and gap g parameters fitted are shown in Figure 5 and show little indication of a memory hierarchy on the vector processors, which is no surprise as the SX8 has none and the E-cache in the X1 has only minor performance impacts in most situations. To fit models well, effective cache size are selected at 256kB for Xeon, Itanium, Opteron, PowerPC, and Cray X1, and 2MB for Power3, Power4, and Power5.

6 Parallel Execution

Analyzing parallel Apex-MAP results for 256 processors and a memory consumption of

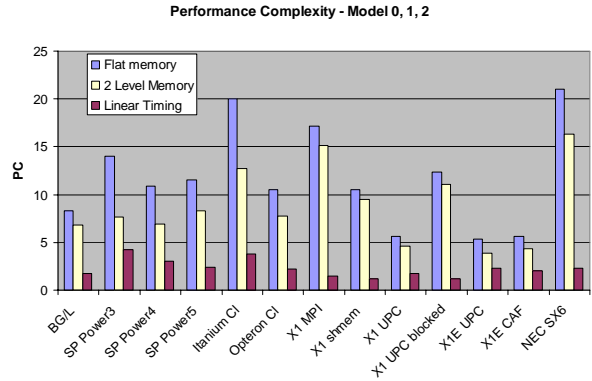


Figure 6: PC values derived with three simple models for parallel execution and based on efficiencies [accesses/cycle].

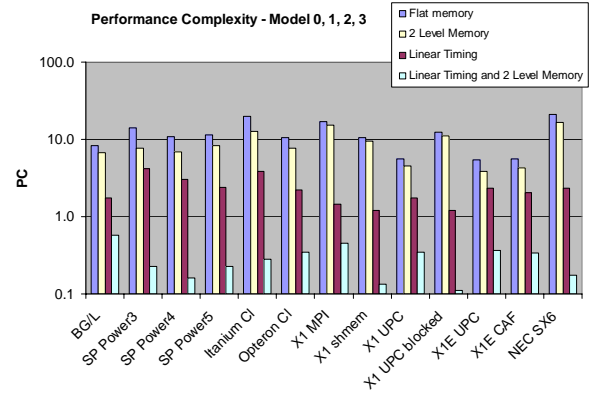


Figure 7: PC values derived with the combined model 3 for parallel execution and based on efficiencies [accesses/cycle].

512MB/process, we face the additional problem of a greatly increased range of raw performance values, which now span 5 orders of magnitude compared to 2 for serial data. This would not be feasible without a sound statistical procedure. This analysis is of special interest as we have obtained performance results with different parallel programming paradigms such as MPI, shmem, and the two PGAS languages UPC, and CoArray Fortran (CAF) on the Cray X1 and X1E systems [2]. In UPC, two different implementations are compared; one for accessing a global shared array element by element and one for a block-transfer access to remote data.

PC values for model 0 in Figure 6 show the highest values for the NEC SX6, an Itanium-Quadrics cluster, and the Cray X1 used with MPI. The lowest complexities are measured for shmem

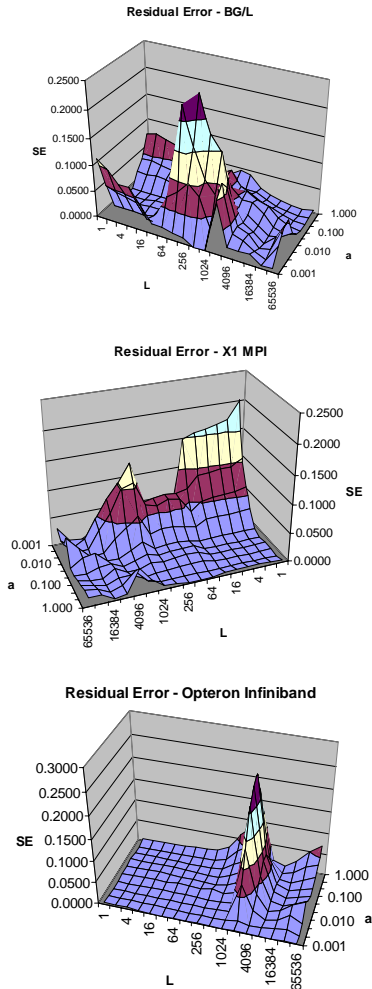


Figure 8: Examples for residual errors of different parallel systems for model 3.

and the PGAS languages on the X1 and X1E. Such a clear separation between these different language groups was surprising to us. The PC value for a global array implementation in UPC is also much lower than for a blocked access.

Model 1 resolves performance better on all tested systems, but again has higher PC values than model 2. The linear timing model is now superior as it fits message exchange on interconnects much better, than a naïve two level memory model. Model 2 represents a programmer coding for long loops and large messages! The combined model 3 resolves performance by far the best. It represents a programmer, who optimizes for long loops, large messages, and high data locality. The lowest complexity PC values are for blocked access in UPC and shmem on the X1.

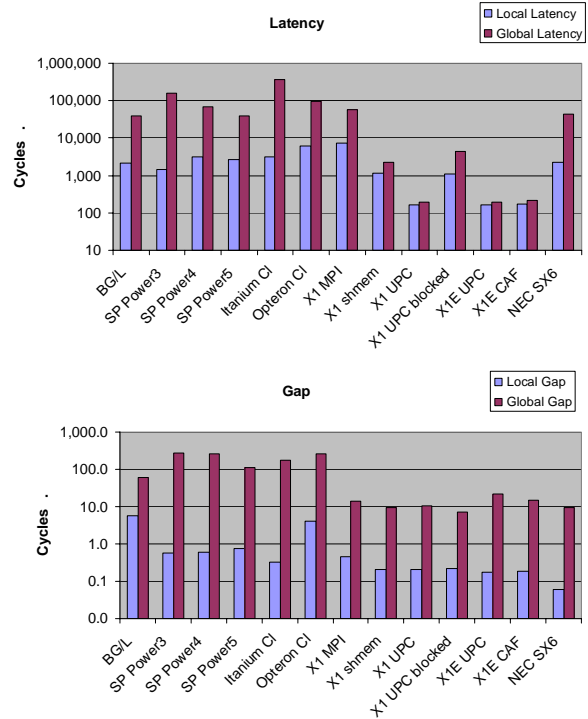


Figure 9: Back-fitted latency and gap values for both memory hierarchies in model 3 for parallel executions.

Inspection of residual errors for BlueGene/L in Figure 8 shows larger deviations for message sizes of 256 Byte and larger with a maximum for 1kB. Flit size on this system is 256Bytes, which suggests influence of the change in protocol once more than one flit is necessary to send a message. The X1 with MPI shows large residual error for high temporal localities. To achieve a best fit the local memory size for this system has to be set to 8GB, which is 16 times the memory used by a single process. These two observations suggest that the second level of system hierarchy has not resolved the local memory access performance. The Opteron Infiniband cluster exhibits a clear signature of a communication protocol change for messages of a few kB size.

The most important level in the memory hierarchy of a system is expressed by the local memory size of c , which for most systems needs to be set to the SMP memory rather than process memory to achieve best model accuracy. While this is not surprising, there are a fair number of systems for which best model fit is achieved for even larger values of c . This is an indication that network con-

tention might be an important factor for the randomized, non-deterministic communication pattern of Apex-MAP, which is not included at all in any of our models. We tested several more elaborate models with a third level of hierarchy and with constant overhead terms for parallel code overhead, but none of these model produced significant improvements and back-fitted parameter values often did not make sense.

Figure 9 shows back-fitted values for l and g in model 3. Latencies for PGAS languages are noticeable lower than for other languages even on the same architecture. In contrast to this, remote gap values seem to be mostly determined by architecture.

7 Related Work

There is a rich collection of research available on the subject of software complexity. Complexity measures discussed range from code size expressed in lines of code (LOC), which is used in various places such as the DARPA HPCS program [4]; Halstead Software Science metrics [5] based on count of operators and operands used; McCabe cyclomatic complexity measure [6] based on the number of linearly-independent paths through a program module, and variants thereof such as design complexity [7]. These software complexity metrics have also been applied and extended to the context of parallel computing [8].

An active area of research within the DARPA HPCS program is productivity metrics, which focus on capturing the complexity of the task of coding itself [4][9][10]. The approach presented here complements research in software complexity and productivity by considering performance complexity, which represent a measure quite different from the former two as it characterizes code execution behavior in a second dimension orthogonal to performance itself. PC is based on performance model accuracy, which has the advantage of depending only on performance measurements and is not based on and does not require code inspection or supervision of coding itself. To our knowledge, no similar concept has been described in the literature.

Apex-MAP is a parameterized, synthetic benchmark designed to cover performance across a whole range of locality conditions [2][3]. Parameterized benchmarks like this have the advantage

of being able to perform parameter sweeps and generating complete performance surfaces.

The HPCC benchmark suite covers a similar locality space with a collection of various benchmarks [11]. This or similar set of benchmarks would provide a good basis for analyzing PC values with more realistic codes. They might however present the additional complication of containing codes, which are measured in different metrics such TF/s and GB/s, between which meaningful averages are very hard to define properly.

8 Conclusions and Future Work

In this paper, we presented a concept for a quantitative measure of performance complexity (PC). The transparency of performance behavior is linked to the complexity of optimizing for performance and can be characterized by the accuracy of performance models. We have presented a definition and detailed description on how to calculate PC based on performance numbers from a set of benchmarks and performance models.

PC is a measure characterizing code execution behavior on a system. To first order, it is independent of performance and serves as second dimension to evaluate systems and programming paradigms.

We demonstrated our methodology by using a parameterized synthetic benchmark, Apex-MAP. The performance results from a large parameter sweep were used as substitute for results from a suite of different benchmarks. This might limit the interpretation of final PC values, but allows us to demonstrate the methodology. Even with this simplified setup we were able to analyze serial and parallel systems in surprising detail.

We are planning on conducting similar studies based on a set of benchmarks such as HPCC. In this context we will also research, how to extend our formalism to situations involving benchmarks measured in different dimensions for which averaging is non-trivial.

9 Acknowledgments

This work was supported in part by the Department of Energy Office of Science and NSA. Computer time was provided by NERSC, LLNL, ORNL, ANL, and HLRS. We also like to thank Hongzhang Shan for helping to collect data, and

Kathy Yelick, Bob Lucas, and Jeremy Kepner for their invaluable input to this work.

References

- [1] DARPA HPCS:
<http://www.darpa.mil/ipto/programs/hpcs/>
- [2] E. Strohmaier and H. Shan. "Apex-Map: A Global Data Access Benchmark to Analyze HPC Systems and Parallel Programming Paradigms", *Proceedings of SC'05* (2005).
- [3] E. Strohmaier and H. Shan. "Architecture independent performance characterization and benchmarking for scientific applications. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Volendam, The Netherlands, 2004.
- [4] A. Funk, V. Basili, L. Hochstein, and J. Kepner, "Application of a Development Time Productivity Metric to Parallel Software Development", *Second International Workshop on Software Engineering for High Performance Computing System Applications at the 27th International Conference on Software Engineering*, May 15, St. Louis, MO
- [5] M. H. Halstead, "Elements of Software Science", *Elsevier North-Holland*, New York, 1977.
- [6] T. J. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, SE-2 No. 4, pp. 308-320, December 1976.
- [7] T. J. McCabe, and C. Butler, "Design Complexity Measurement and Testing", *Communications of the ACM*, 32, pp. 1415-1425, December 1989.
- [8] S. VanderWiel, D. Nathanson, and D. Lilja, "Performance and program complexity in contemporary network-based parallel computing systems". *Technical Report HPPC-96-02*, University of Minnesota, 1996.
- [9] J. Kepner, "HPC Productivity Model Synthesis", *International Journal of High Performance Computing Applications: Special Issue on HPC Productivity J. Kepner (editor)*, Volume 18, Number 4, Winter 2004 (November)
- [10] J. Carver, S. Asgari, V. Basili, L. Hochstein, J. Hollingsworth, F. Shull, and M. Zelkowitz, "Studying Code Development for High Performance Computing: The HPCS Program," *Proceedings of the First International Workshop on Software Engineering for High Performance Computing System Applications at ICSE*, pp. 32-36. Edinburgh, Scotland, May 2004.
- [11] HPC Challenge Benchmark,
<http://icl.cs.utk.edu/hpcc/>