High Productivity Computing Systems and the Path Towards Usable Petascale Computing

Part B: System Productivity Technologies

GUEST EDITOR JEREMY KEPNER

1 A System-wide Productivity Figure of Merit
Declan Murphy, Thomas Nash, Lawrence Votta, Jr., and Jeremy Kepner

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

19 Modelling Programmer Workflows with Timed Markov Models
Andrew Funk, John R. Gilbert, David Mizell, and Viral Shah

27 A Compiler-guided Instrumentation for Application Behavior Understanding
Pedro C. Diniz and Tejus Krishna

35 Symbolic Performance Modeling of HPCS Applications
Sadaf R. Alam, Nikhil Bhatia, and Jeffrey S. Vetter

41 Designing Scalable Synthetic Compact Applications for Benchmarking High Productivity Computing Systems
David A. Bader, Kamesh Madduri, John R. Gilbert, Viral Shah, Jeremy Kepner, Theresa Meuse, and Ashok Krishnamurthy

52 SLOPE - A Compiler Approach to Performance Prediction and Performance Sensitivity Analysis for Scientific Codes
Pedro C. Diniz and Jeremy Abramson

46 Metrics for Ranking the Performance of Supercomputers
Tzu-Yi Chen, Meghan Gunn, Beth Simon, Laura Carrington, and Allan Snavely

68 Large-Scale Computational Scientific and Engineering Project Development and Production Workflows
D. E. Post and R. P. Kendall

Available on-line at http://www.ctwatch.org/quarterly/
High Productivity Computing Systems
and the Path Towards
Usable Petascale Computing
Part B: System Productivity Technologies
Guest Editor Jeremy Kepner
A System-wide Productivity Figure of Merit

1. Introduction

Establishing a single, reasonably objective and quantitative framework to compare competing high productivity computing systems has been difficult to accomplish. There are many reasons for this, not the least of which is the inevitable subjective component of the concept of productivity. Compounding the difficulty, there are many elements that make up productivity and these are weighted and interrelated differently in the wide range of contexts into which a computer may be placed. Significantly improved productivity for high performance government and scientific computing is the key goal of the High Productivity Computing Systems (HPCS) program. Evaluating this critical characteristic across these contexts is clearly essential to attaining and confirming this goal.

This is not entirely a new phenomenon. Anyone who has driven a large scale computing budget request and procurement has had to address the problem of turning a set of preferences and criteria, newly defined by management, into a budget justification and a procurement figure of merit that will pass muster with agency (and OMB) auditors. The process of creating such a procurement figure of merit helps to focus the mind and cut through the complexity of competing user demands and computing options.

Imagining that we are initiating a procurement where Productivity = Utility/Cost will be the criteria, we have developed a total productivity figure of merit. This framework includes such system measurables as machine performance and reliability, developer productivity, and administration overhead and effectiveness of resource allocation. This is all in the context of information from the particular computing environment that may be proposing and procuring an HPCS computer. We note that this framework is applicable across the broad range of environments represented by HPCS mission partners and others with science and enterprise missions that are candidates for such systems.

The value of each variable in our figure of merit is intended to come from a single source, either from one of the HPCS R&D areas or from a mission organization that may procure a system. While we identify the potential source of each value, we do recognize that some of these numbers will not be easy to obtain, particularly those involving the impact of system design on human productivity. Nonetheless, we believe that, at the least, this framework will identify the individual metrics that these efforts should strive to measure. In the end, we will all have to admit that some combination of measurements, informed guesses, and subjective evaluations will be needed to arrive at a figure of merit number.

We also recognize that there is coupling between some of the variables we treat as independent. For example, a user's productivity is impacted by the way jobs are allocated. To deal with this, we suggest assuming an environment in which a particular variable is determined. This means that values for all the other variables, their “operating point,” must be specified for each measurement of a variable. For measurable variables, these operating points come from measurements and studies. One could iterate to a final answer, but we argue that this is unnecessary because the effect of any such coupling, with reasonable operating point guesses, is far smaller than the precision (such as it might be) of any of the measured variables involved.

Not surprisingly, our figure of merit has much in common with ideas expressed in earlier HPCS work. However, we extend beyond this in the following ways:

---

1. Introduction

Establishing a single, reasonably objective and quantitative framework to compare competing high productivity computing systems has been difficult to accomplish. There are many reasons for this, not the least of which is the inevitable subjective component of the concept of productivity. Compounding the difficulty, there are many elements that make up productivity and these are weighted and interrelated differently in the wide range of contexts into which a computer may be placed. Significantly improved productivity for high performance government and scientific computing is the key goal of the High Productivity Computing Systems (HPCS) program. Evaluating this critical characteristic across these contexts is clearly essential to attaining and confirming this goal.

This is not entirely a new phenomenon. Anyone who has driven a large scale computing budget request and procurement has had to address the problem of turning a set of preferences and criteria, newly defined by management, into a budget justification and a procurement figure of merit that will pass muster with agency (and OMB) auditors. The process of creating such a procurement figure of merit helps to focus the mind and cut through the complexity of competing user demands and computing options.

Imagining that we are initiating a procurement where Productivity = Utility/Cost will be the criteria, we have developed a total productivity figure of merit. This framework includes such system measurables as machine performance and reliability, developer productivity, and administration overhead and effectiveness of resource allocation. This is all in the context of information from the particular computing environment that may be proposing and procuring an HPCS computer. We note that this framework is applicable across the broad range of environments represented by HPCS mission partners and others with science and enterprise missions that are candidates for such systems.

The value of each variable in our figure of merit is intended to come from a single source, either from one of the HPCS R&D areas or from a mission organization that may procure a system. While we identify the potential source of each value, we do recognize that some of these numbers will not be easy to obtain, particularly those involving the impact of system design on human productivity. Nonetheless, we believe that, at the least, this framework will identify the individual metrics that these efforts should strive to measure. In the end, we will all have to admit that some combination of measurements, informed guesses, and subjective evaluations will be needed to arrive at a figure of merit number.

We also recognize that there is coupling between some of the variables we treat as independent. For example, a user's productivity is impacted by the way jobs are allocated. To deal with this, we suggest assuming an environment in which a particular variable is determined. This means that values for all the other variables, their “operating point,” must be specified for each measurement of a variable. For measurable variables, these operating points come from measurements and studies. One could iterate to a final answer, but we argue that this is unnecessary because the effect of any such coupling, with reasonable operating point guesses, is far smaller than the precision (such as it might be) of any of the measured variables involved.

Not surprisingly, our figure of merit has much in common with ideas expressed in earlier HPCS work. However, we extend beyond this in the following ways:
A System-wide Productivity Figure of Merit

In a well-balanced HPCS, significant costs will be incurred for resources other than just the CPU cycles that dominate thinking in the commodity cluster architectures. In particular, memory and bandwidth resources will have cost as much or more than CPU, and efficient programs and job allocation will have to optimize use of memory and bandwidth resources as much as CPU. Our framework allows for the inclusion of any set of significantly costly resources.

A single job may be highly optimized, and those in the project it comes from will inevitably believe its utility (or value) approaches infinity. However, a computer center must optimize the total workload, given its organization's evaluation of relative priority (utility or value) for each project and job. The overall utility of the total output of the computer depends on the degree to which the allocation of system resources reflects the institution's priorities and determination of value (or utility). Further, the productivity of the administration staff depends on a system's administration environment and tools, and on its stability.7

The remainder of this article is organized as follows. In Section 2 we define the productivity figure of merit as a combination of factors, where each factor represents a different aspect of productivity that can be evaluated relatively independently of the others. In Section 3 we show how the figure of merit captures the perspectives on productivity of people in different institutional roles and we show how productivity ratios can be used to simplify the evaluation process. In Section 4 we discuss the need for productivity benchmarking and propose the use of operating points to narrow the scope of the analysis needed for a productivity evaluation. We provide a recipe for using the figure of merit in Section 5. We conclude and describe our experience applying the figure of merit using a simple spreadsheet (available from the authors) in Section 6.

2. The Productivity Figure of Merit

In factoring out the productivity contributors, we take an approach that, like the blind men and the elephant, focuses on what can be measured at defined parts of the beast and builds to a picture of the whole productivity equation for a system in a particular environment. This assumes that our elephant is reasonably smooth and predictable between those points we can feel with our measurements.7

We start with

\[ \text{Productivity} = \frac{\text{Utility}}{\text{Cost}} \]

We expand the utility into system level and job level components

\[ P = \frac{U_{\text{sys}} E_{\text{proj}} E_{\text{adm}} E_{\text{job}} A_{\text{sys}} R}{C} \]  \hspace{1cm} (1)

As a convention, we use the letters \( U, E, A, R, C \) to denote variables of utility, efficiency, availability, resources, and cost, respectively. The subscripts indicate the variables that address system level (including administrative and utility) and job level factors.

We recognize that some aspects of the system level efficiency will never be amenable to measurement and will always require subjective evaluation. Only subjective evaluation processes can address the first two variables in the utility numerator. In principle one can measure the last four variables and the HPCS research program is addressing such measurements.
We have been emphasizing that this is to be a figure of merit, including estimates and evaluations of what we expect the productivity output of an installation to be. For clarity, in explaining this formulation, we will start by talking as if we know what happened over the lifetime, as if we are doing a post-mortem. We will mention in passing how the components relate to ongoing work on productivity estimators. We will return to discuss these estimators in more detail in Section 4.

The goal of those optimizing utility at the job level is to maximize resources they can effectively apply to their problem. This will enable them to bring their project to a successful conclusion with higher utility (larger scale or finer granularity solutions or higher throughput of data intensive problems) or more rapidly (allowing more problems of similar utility to be accomplished within the lifetime of the resources). It is “not in their job description” to address the relative utility of their problem compared to others (though they may be inclined to do so). So, we consider utility at the job level, $U_{job}$, to be just the cost ($) of the resources that they have effectively used, and the job level efficiency $E_{job} = U_{job} / C_{sys}$, with $C_{sys}$ the total lifetime system cost. As defined below and in the Appendix, $U_{job}$ and $E_{job}$ are averaged over all jobs.

The government, or the owners or stockholders, establish how activities are to be valued by the institution management. Management valuation appears in the largely subjective $U_{sys}$ variable, which includes, for example, what constitutes “success” for a computing activity. They may well value a successfully completed activity higher than the cost of the resources used. (It would be unusual for them not to do so at budget proposal time.) $U_{sys}$ assumes, as managers might often wish, that all resources are available and fully allocated at all times during the life of their expensive system. We define it as a multiplier to the peak system resources in CPU units. System utilization efficiency and project level effectiveness is included in $E_{adm}$ and $E_{proj}$ as described below.

Once a system is delivered, the system administrators (and vendor) strive to meet management expectations for availability, $A_{sys}$ and system level resource utilization efficiency (including allocation), $E_{adm}$.

More detailed descriptions of the individual variables follow and they are expanded further in the Appendix. We indicate dimensional units in brackets.

$P$, the productivity, is dimensionless, as required by the common economic interpretation of the productivity concept we use, output/input, [$]/[\text{\$}]$.

$C$ is total cost of ownership, including all costs for developing software and running it on the computer, over a defined lifetime of the system ($T$). This is expanded into components in the Appendix. [$\text{\$}]$.

$T$ is the lifetime of the system as defined in the typical budget approval process. It does not show up explicitly in the top-level equation, but it is important to the definitions of the variables as well as for the measurements. It will depend on considerations specific to each environment, including, for example, whether the procurement and justification (and cost) involve continuing upgrades. Those responsible for budget submissions at the proposing institution are the source of this variable. [yrs]

$E_{job}$ is the ratio of $U_{job}$ to the total system cost $C_{sys}$. $U_{job}$ is the total productively used resources over the lifetime by all individual jobs in all project activities, normalized to the assumption of 100% availability and 100% resource allocation. To a large extent, this efficiency measures how well programs use parallel resources. We include other costly resources (e.g., memory, bandwidth, I/O) besides CPU. In order not to favor one class of resources over another, we weight the resources by their relative costs. We say “productively used” so that we only include resources that would

---

8 However, a system approaching 100% allocation will have long queues that would have a deleterious effect on the total utility coming out of the computer. Effects like long queues should be included as part of the environment conditions (the “operating point”) in which job level productivity measurements of $E_{job}$ and $E_{proj}$, defined below, are made.

9 For example $c_{CPU}$ is that fraction of the total lifetime cost, $C_{sys}$, that is attributable to CPU, in $\text{\$ops}$. For CPU, $C_{CPU}$ can be understood (in Sterling’s notation [4]) as $S \times T \times c_{CPU}/C_{sys}$ summed over all jobs during the lifetime $T$, with $S$ the attained performance (ops/sec).

---
be consumed in direct support of maximizing utility for the specified problem. Estimators for 
$E_{job}$ and for $I_{proj}$, the corresponding average total project personnel time (including development and production) per project, are what Job-level Productivity Benchmarks should aim to measure for different environments and systems. More detailed definitions of these can be found in [2]. [Dimensionless]

$A_{sys}$ is the availability; the fraction of total resource time available to the jobs making up $R_{job}$. This accounts for all planned and unplanned downtimes, but does not include job restarts due to failures, which is included in $E_{sys}$ below. Note this is not fraction of system time, but fraction of resource time, so that it accounts for portions of the large system being available when other portions are down. This parameter is based on vendor estimates of MTTF, maintenance requirements, and other RAS considerations. [Dimensionless]

The product $E_{sys} = E_{adm}E_{proj}$ is the effectiveness of the workload resource allocation at meeting the institution’s priorities. System architecture, software, and administrative tools can make a big difference to this important metric. This is where utility (or value) enters the picture and we go beyond just looking at job-level optimization. We assume that the institution will have a process for establishing the priority or the utility or the value of the individual projects and jobs. $E_{sys}$ is the ratio of the actual total value of the jobs run over the lifetime to the optimal total that would come from maximally efficient management of a reference HPCS platform, for a given job mix, a given installed system configuration, and a given time pattern of resource downtime. As shown in [2], we allow for time dependent values, since allocation efficiency that completes a job or project long after it is needed has little or no value. We recognize explicitly (see further discussion in [2]) that not all the factors that go into $E_{sys}$ are amenable, even in principle, to before or after the fact quantitative measurement. Some will need to be evaluated subjectively, and primarily have to do with the effectiveness with which projects (management assigned and valued tasks) are able to accomplish management goals and schedules. This is why we have written $E_{sys}$ as a product with $E_{adm}$ as the measurable part and $E_{proj}$ the project effectiveness part requiring subjective evaluation. $E_{adm}$ is the estimator that, along with the cost of administration, is what System-level and Administration Productivity Benchmarks should aim to measure, for different systems, configurations, and user environments. $E_{proj}$ represents other considerations that result in reduced efficiency at delivering utility. [Dimensionless]

$U_{sys}$ is the summed utility (or value) of all projects or activities, per unit of total available resources, $R$, that would complete successfully during the lifetime $T$ if the system administrators given the systems tools and capabilities of the HPCS Reference Platform were able to attain the optimum level of resource utilization and the system was perfectly available. The values may be assigned by whatever process the institution uses to prioritize its projects and justify its budget proposals. This variable only includes the local institution’s evaluation of the value of the projects it intends for the installation. In some environments, it may be ignored and set to a constant such as $C_{sys}$. We convert all resource units to CPU operations, weighting by their relative costs. [$/ops$]

### 3. Different Perspectives on Productivity

Individuals with different professional responsibilities look at productivity with naturally different perspectives. These include, for example, acquisition decision makers, project managers, individual programmers (“users”), researchers, system administrators, service engineers, operators, vendors, and system designers. It is useful to focus on two of these, project managers, and decision makers.
In principle, project manager perspectives are aligned with their institutional management, the decision makers. In practice, they differ because at least our stereotype of a project manager is only concerned with project personnel costs and not either machine operating or capital costs. And similarly the project manager can only address certain terms in the utility numerator of the productivity ratio. So, the project manager’s perspective on productivity is a subset of the decision maker in Eqs. 1 and 2,

\[ P_{PM} = \frac{U_{sys} E_{proj} E_{job} R}{C_{proj}} \]  

(2)

The decision maker’s (acquisition) productivity we developed in earlier sections is then

\[ P = \frac{E_{adm} A_{sys} C_{proj} P_{PM}}{C} = E_{org} P_{PM} \frac{C_{proj}}{C} \]  

(3)

Here \( E_{org} = E_{adm} A_{sys} \) is the organization’s and system’s multi-project resource utilization efficiency.

We can get considerably more simplification and, perhaps, also more insight, if we now think in terms of comparing the new, next generation (HPCS) system that we are evaluating to a standard reference machine. This reference can be a defined traditional MPI cluster configuration for which there is some level of understanding and experience regarding productivity in its environment. All measurements and terms in Eq. 8 can be ratios of the new system to the reference. In most organizations, for budget and institutional reasons, the ratio of project (development personnel) costs to machine and operations costs is typically a constant. In this situation we conclude that

\[ \bar{P} = \bar{E}_{org} \bar{P}_{PM} = \bar{E}_{adm} \bar{A}_{sys} \bar{E}_{proj} \bar{E}_{job} \bar{R} \]  

(4)

The normalization to the reference system is indicated by bars. We assume that management evaluation of utility obtainable per effectively utilized resource is identical for the two systems \( (\bar{U}_{sys}=1) \). The not-surprising conclusion is that we can consider the relative improvement in productivity to be just the product of productivity measurables for the new system normalized to the old.

4. Productivity Benchmarks and Operating Points

We could get an accurate figure of merit as part of a post-mortem — after the life cycle is over. At that point, we could have access to real experience. But that’s not what we want; we need to predict and estimate in advance. So where do our productivity estimators come from? We have been assuming there are two classes of productivity benchmarking activities: 1) at the job- and project-level, measuring development and production activities; and, 2) at the system-level, measuring administration activities and the effect of the differences in system designs and configuration options on administration overhead and system-level resource utilization.

Development, job- and project-level, productivity benchmarks would aim to measure — for the problem mix of a specific environment — the development time required to attain different levels of productive resource utilization. The simplest example is how much programming time it takes to attain various levels of speedup. Curves of productive resource utilization vs. development and other project personnel time will increase, probably like step functions and will usually present an obvious point of diminishing returns. This can be taken as the “operating point.” Averaged over the work flow and job mix, it provides an estimator of \( \bar{t}_{proj} \) and \( E_{CPU}, E_{mem}, E_{BW}, \) and \( E_{IO} \).
Similarly, administration, system-level, benchmarks can aim to measure \( E_{adm} \). For aspects involving effective scheduling, this could be accomplished, specific to a given environment and system configuration, by creating a simulated scheduling environment and allowing working administrators to attempt to optimize the allocation of a list of prioritized (value assigned) jobs for some simulated time period. An ideal allocation would be defined for each environment as attaining the full value of each project including time considerations. The ratio of the measured to the ideal would give an estimator for this aspect of \( E_{sys} \). Just as for the development benchmarks, this can be treated as a curve where better allocation, more efficient use of resources, and a general increase in output value, uses more administrator time, and a reasonable operating point of diminishing returns is selected. The cost of the administrator time at this operating point would be included as all or part of \( c_{adm} \) (depending on how complete a benchmark simulation is used).

5. Step by Step

Here is a step by step approach to a normalized productivity figure of merit based on workflow productivity measurements:

1. Define a standard reference system for comparison, a “canonical cluster”. Productivity estimates are made on this system and on the new system under evaluation and reported as ratios.

2. Define the set of working environments (application areas, project units, unique program activities) in terms of the resource emphasis (FLOPs, GUPs, bytes, …) and standard personnel rates.

3. For a few (usually just one) activity classes in each environment, describe the workflow as the effort fraction of standard workflow elements. Select a set of representative applications for attained performance (\( E_{job} \)) measurements. Design job-level productivity measurements for the workflow elements and measure the productivity term, \( E_{job} \).

4. Define a typical job mix and the overall administrative workflow, and design administrative-level productivity measurements. Measure \( E_{adm} \).

5. Define subjective scoring criteria for \( E_{proj} \) for each environment, and score the systems with regard to project success rate, accessibility, ease of use effectiveness, …

6. Vendors provide information to identify the relevant resource capability ratios of the systems, such as peak performance (\( R_{CPU} \)) or GUPs, etc. Vendors also provide information on availability of the new system. Reference system availability should be obtained from experience data.

9. Combine factors into the relative productivity figure of merit, \( P \).

10. If absolute Productivity, \( P \), is of interest, costs must be included and a utility constant \( (U_{sys}) \) defined. Costs include project and administrative personnel budgets, and system costs including initial costs and running costs over a defined lifetime, \( T \).
6. Conclusion, Examples, and a Spreadsheet to Learn From

We have tried to demonstrate that the productivity figure of merit we have described here is really much simpler than it may have first appeared. It is no more than a way of getting to a single number that combines what we know or can guess about a system configured for a particular environment into something that approximates a measure of total productivity. It can be used in HPCS design comparisons, subsequent budget justifications, and ultimately, we hope, in procurements.

Many of the numbers needed as input are traditional cost and performance variables. The hard parts, of course, are the benchmarks needed to measure the productivity of humans when confronted with these new systems. We see these productivity benchmarks as simply measuring curves of efficiency of job resource utilization, or system utility optimization, vs. the cost in time of the human effort. These curves should clearly indicate an obvious point of diminishing returns, an operating point.

We recognize that we have made use of words like simply, clearly, obviously, and this may be unfair. We know that getting a number that approaches being a true measure of productivity for a given system is going to be difficult. We need goals for the productivity benchmarking efforts to aim at, and we think this framework provides them.

We have developed a spreadsheet, which may be obtained from the authors, that allows one to gain some intuition into how system-wide productivity comes together from the many components. The first six sheets of the spreadsheet are entry sheets to be completed by the different entities that may be responsible for the information: Vendor Cost, Host Environment and Values, Performance, Job-level Productivity, Administration Productivity, and Subjective Productivity Evaluation. The last sheet summarizes the calculated result.

We see the process of obtaining an informative figure of merit as being incremental. One may start by entering guesses, policies, and goals, and then progress through preliminary measurements and even post-mortem analysis. For each entry, the spreadsheet has a quality descriptor which may be selected from a list, presently including: canonical, policy, wild guess, informed guess, measured 100%, measured 30%, measured 10%, measured 1%, post mortem.

It is instructive to play with the spreadsheet, changing performance, productivity, and cost variables and seeing their effect on the figure of merit and its components on the last sheet of the spreadsheet. In this way, one can learn quickly in a particular environment what matters and what doesn't for the life-cycle productivity, our figure of merit. Even with the simplifications, and possible biases, built into this approach, we believe it goes furthest towards allowing a real understanding of how best to reach the goal of maximizing overall productivity. An example of the use of the spreadsheet to compare the productivity of two systems can be found in [2].

Acknowledgements

We acknowledge with pleasure the important contributions made to the development of a figure of merit, particularly in the critical early stage when the key ideas were being understood, by the following: Susan Squires, Jan Strachovsky, Michael Van De Vanter, and Robertus Van Der Wijngaart.

---

15 In the likely event that host environments wish to specify time-dependent values for projects, these may be parameterized functions or tables. In either case, the spreadsheet will have to be revised to accommodate time-dependent values and system level efficiencies since the present examples do not take time into account.

16 Value judgments may be implicit in some of our simplifications. If these are not consistent with those of a particular environment, they can obviously be adjusted using the basic framework of this approach and the spreadsheet. What is important is to attempt to include as many cost, performance, and productivity components as possible into the best possible single productivity figure of merit.
Appendix
Expansion of Key Terms

Here we expand the high level terms of Eq.1:

A. Cost

\[
C = C_{\text{sys}}^{\text{initial}} + c_{\text{sys}}^{\text{annual}} T + c_{\text{adm}} n_{\text{adm}} T + N c_{\text{proj}} T
\]

Here,

\[
C_{\text{sys}} = C_{\text{sys}}^{\text{initial}} + c_{\text{sys}}^{\text{annual}} T
\]

is the total life cycle cost of the system. [$]

It includes the following costs:

**One Time Costs**

- Initial hardware
- Initial software (system and purchased application software)
- Non-standard or specialized facility equipment (e.g., water cooling)
- Facility preparation (electrical and other building modification)
- Installation

\[
C_{\text{adm}} = c_{\text{adm}} n_{\text{adm}} T = c_{\text{adm}} (n_{\text{adm}}^{\text{base}} + n_{\text{adm}}^{\text{productivity}}) T
\]

is the total lifetime administration costs.

\[
\begin{align*}
C_{\text{proj}} &= N c_{\text{proj}} \bar{t}_{\text{proj}} = c_{\text{proj}} n_{\text{proj}} T
\end{align*}
\]

Note that electrical running costs may be dependent upon the job mix.
B. Productively Used Resources at the Job Level

As noted before we define the job level utility as the cost of the resources productively used

\[ U_{\text{job}} = c_{\text{CPU}}E_{\text{CPU}}R_{\text{CPU}} + c_{\text{mem}}E_{\text{mem}}R_{\text{mem}} + c_{\text{BW}}E_{\text{BW}}R_{\text{BW}} + c_{\text{IO}}E_{\text{IO}}R_{\text{IO}} \]  

and the job level efficiency as

\[ E_{\text{job}} = \frac{U_{\text{job}}}{C_{\text{sys}}} \]  

Here, the subscripts refer to the resource types, CPU, memory, inter-processor bandwidth, and I/O bandwidth resources, respectively. This can obviously be generalized to include other resources with significant costs.

The \( c_r \) are the total costs attributable to each resource per unit of that resource. The total life cost is \( C_{\text{sys}} \) described above,

\[ C_{\text{sys}} = c_{\text{CPU}}R_{\text{CPU}} + c_{\text{mem}}R_{\text{mem}} + c_{\text{BW}}R_{\text{BW}} + c_{\text{IO}}R_{\text{IO}} \]

The \( R_r \) are the total lifetime resources of type \( r \) used by all of the \( N \) project activities.\(^{18}\) The resources are assumed for the purposes of this job-level variable to have been 100% allocated and the time to be 100% available for one of the \( N \) activities.\(^{19}\) Note that costs only come into Eq.5 to provide a relative weight for the different resources.

The \( R_r \) are based on performance measurements and a choice of configuration options. Remember that the \( R_r \) include the lifetime, so that the units for CPU, memory, bandwidth, and IO are [ops], [byte-years], and [bytes], respectively. We weight the resources by their relative costs, so \( U_{\text{job}} \) is just the cost ($) of the resources that the project teams could have used if they were perfectly efficient. The \( c_r \) provide the conversion from resource units to utility as cost in \$/ops, \$/byte-years, \$/bytes, \$/bytes.

The \( E_r \) are the fraction of the total of each resource productively utilized on average by all the jobs in the \( N \) project activities. The variables \( E_r \) are efficiency estimators that Job-level Productivity Benchmark efforts can aim to measure (for specified workflows and job mixes) along with the average effort per activity, \( t_{\text{proj}} \).

By “productively,” we mean that resources used only to support parallelization are not counted, and that the single processor algorithm being used is not wasteful of resources. This can either be a protocol rule in the benchmarking or the benchmarks can down-rate the utilization fractions \( E_r \) for resource usage that is not in direct support of the task or algorithms that are less than optimal.\(^{20}\)

The \( E_r \) are [dimensionless].

C. Project and System Level Efficiency

\( E_{\text{adm}} \), the measurable part of system efficiency, may be understood as the effectiveness of the administrative staff in allocating resources efficiently given the tools and the real environment of their system and the time that they have available, as included in the cost \( c_{\text{adm}} \). An estimator for this traditional measure of system utilization, \( E_{\text{adm}} \), is what System-level and Administration Productivity Benchmarks can aim to measure, as discussed in Section 4.

\( E_{\text{proj}} \) is the subjective component of the figure of merit to allow for evaluation of issues, which might be attributed to system hardware or software, such as project failures or delays and the accessibility of the computing system environment to staff with different levels of computing skills. In general, it includes system or configuration factors that impact the effectiveness of programming teams at accomplishing their goals. This is where utility vs. time considerations may be included, as discussed in [2].

\( E_{\text{adm}} \) and \( E_{\text{proj}} \) are [dimensionless].
Performance Complexity: An Execution Time Metric to Characterize the Transparency and Complexity of Performance

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 I 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. I 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 I 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, I 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 \( PC \). I find that our results reflect intuitive qualitative expectations for relative \( PC \) values surprisingly well.

\(^1\) DARPA HPCS - http://www.darpa.mil/ipto/programs/hpcs/

The rest of this paper is organized as follows: In section 2, I introduce and describe our concept for Performance Complexity in detail, section 3 is a brief introduction to Apex-MAP, in section 4 I develop the used performance models, section 5 and 6 describe results for serial and parallel execution, section 7 discusses related work, and I present my 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, my 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 reflects the performance transparency and complexity of performance control the programmer will experience.

Goodness of fit measures for modeling measurements are based on the Sum of Squared Errors (SSE). For a system S, the performance \( P_{\theta} \) of a set of n codes \( C_i \) is measured. Different measurements \( j \) might also be obtained with the same code executed with different problem parameters such as problem sizes and resource parameters such as concurrency levels. A performance model \( M_{\theta} \) is used to predict performance \( M_{\theta i} \) for the same set of experiments.

To arrive at a PC measure with the same metric as performance, I have to use standard deviation (SD), which is the square-root of average SSE, and I cannot use the more widely used total sum. The average \( SSE_{\theta i} \) for each system \( i \) and model \( k \) across all codes \( j \) is then given by \( SSE_{\theta i} = \frac{1}{n} \sum (P_{\theta i} - M_{\theta i})^2 \). While these SSE values are still absolute numbers (they carry dimension), they are easily transformed

<table>
<thead>
<tr>
<th>Operation</th>
<th>Dimension*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial data: ( P_{\theta}, M_{\theta} )</td>
<td>[ops/sec]</td>
</tr>
<tr>
<td>Transform ( P_{\theta}, M_{\theta} ) to ideal flat metric</td>
<td>[ops/cycle]</td>
</tr>
<tr>
<td>Log-transformation: ( P_{\theta}^{'}, M_{\theta}^{'}, P_{\theta}, M_{\theta} )</td>
<td>[log(ops/cycle)]</td>
</tr>
<tr>
<td>Basic Calculations</td>
<td></td>
</tr>
<tr>
<td>( \overline{P} = \frac{1}{n} \sum P_{\theta j} )</td>
<td>[log(ops/cycle)]</td>
</tr>
<tr>
<td>( \overline{SS} = \frac{1}{n} \sum (P_{\theta j} - \overline{P})^2 )</td>
<td>[log(ops/cycle)^2]</td>
</tr>
<tr>
<td>( \overline{SSE} = \frac{1}{n} \sum (P_{\theta j} - M_{\theta j})^2 )</td>
<td>[log(ops/cycle)^2]</td>
</tr>
<tr>
<td>( 1 - R^2 = \frac{\overline{SSE}}{\overline{SS}} )</td>
<td>[]</td>
</tr>
<tr>
<td>Back-Transformations</td>
<td></td>
</tr>
<tr>
<td>( P = exp(\overline{P}) = \sqrt[n]{\prod P_{\theta j}} )</td>
<td>[ops/cycle]</td>
</tr>
<tr>
<td>Absolute ( PC_{\theta} = exp(\sqrt{\overline{SSE}}) - 1 )</td>
<td>[ops/cycle]</td>
</tr>
<tr>
<td>Relative ( PC_{\theta} = \frac{\sqrt{\overline{SSE}}}{\overline{SS}} - 1 )</td>
<td>[]</td>
</tr>
</tbody>
</table>

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

into relative numbers by dividing them by the similarly defined Sum of Squares (SS) of the measured data \( P \cdot \bar{SS} = \sum_i (p_i - \bar{P})^2 \) and \( R^2 = \frac{SSE}{SS} \).

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, I 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, I transform these values into the usual range of larger than zero by subtracting one. The full sequence of the 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

![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 \( \alpha \) temporal locality.](image)

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 \( \alpha \), parameters related to spatial locality and temporal reuse respectively. Apex-MAP then chooses indices into a data array that are distributed according to \( \alpha \), 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 \( \alpha \) 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 \( \alpha \) (see Figure 1).
4. Performance Models and Modeling Methodology used

In my 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. Eternally 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 depend 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 languages 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.

<table>
<thead>
<tr>
<th>Model</th>
<th>Time per access</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flat memory</td>
<td>$T = g = \text{constant}$</td>
</tr>
<tr>
<td>1</td>
<td>Two level memory</td>
<td>$T = P(c/M)^\alpha g_1 + (1-P(c/M))g_2$</td>
</tr>
<tr>
<td>2</td>
<td>Latency, Bandwidth</td>
<td>$T = (l + g(L-1))/L$</td>
</tr>
<tr>
<td>3</td>
<td>$L, B$ for two levels</td>
<td>$T = P(c/M) + (l_1 + g_1(L-1))/L_1 + (1-P(c/M)) + (l_2 + g_2(L-1))/L_2$</td>
</tr>
</tbody>
</table>

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 $\alpha$ 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.

I 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). My 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 $\alpha$ 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 my models, predictions based on hardware specifications can be expected to be off by large margins. I use a back-fitting procedure to minimize SSE and the prediction error of the models. To check the validity of the models I 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 \( \epsilon \) should be. It is therefore advisable to at least use a back-fitting 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, I calculate the PC values of my four performance models for the serial performance measurements of Apex-Map.\(^1\) The memory size used was 512MB and I swept across 10 \( \alpha \) 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 into consideration the 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 processors, but not as much as we 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 also does not capture any more advanced memory access feature on modern superscalar processors such as pre-fetching.

**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].

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 systems, 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 his or her code with long for loops to improve performance.

PC values for the combined model 3 are only visible and shown in Figure 3 where I switched to a logarithmic vertical scale. Clearly, the performance of Apex-MAP is overall much better resolved 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 processors with PC values shows 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 on several occasions. This indicates that my 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 residuals 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 sizes 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 512MB/processor, I face the additional problem of a greatly increased range of raw performance values, which now span five orders of magnitude compared to two for serial data. This would not be feasible without a sound statistical procedure. This analysis is of special interest as I have obtained performance results with different parallel programming paradigms such as MPI, shmemp, and the two PGAS languages, UPC and CoArray Fortran (CAF) on the Cray X1 and X1E systems. 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 shmemp and the PGAS languages on the X1 and X1E. Such a clear separation between these different language groups was surprising. 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 su-perior as it fits message exchange on interconnects much better than a naïve two level memory model. Model 2 represents a programmer coding for long

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].


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

Inspection of residual errors for BlueGene/L in Figure 8 shows larger deviations for message sizes of 256 Bytes 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.

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

7. Related Work

There is a rich collection of research available on the subject of software complexity. Complexity measures discussed include code size expressed in lines of code (LOC), which is used in various places such as the DARPA HPCS program; Halstead Software Science metrics based on count of operators and operands used; McCabe cyclomatic complexity measure based on the number of linearly-independent paths through a program module, and variants thereof such as design complexity. These software complexity metrics have also been applied and extended to the context of parallel computing.
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. The approach presented here complements research in software complexity and productivity by considering performance complexity, which represents 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 nor requires code inspection or supervision of coding itself. To my 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. 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. 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 as TF/s and GB/s, between which meaningful averages are very hard to define properly.

8. Conclusions and Future Work

In this paper, I 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. I 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. It is independent of performance and serves as a second dimension to evaluate systems and programming paradigms.

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

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

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. I would 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.

11 HPC Challenge Benchmark - http://icl.cs.utk.edu/hpcc/
Modelling Programmer Workflows with Timed Markov Models

Abstract

Software development is a complex process. Many factors influence programmer productivity — experience, choice of programming language, etc. — but comparisons of their effects are primarily anecdotal. We describe a quantitative method to capture programmer workflows using timed Markov models. We fit data collected from programmers in two separate classroom experiments to these timed Markov models, and compare the workflows of programmers using UPC and C/MPI.

1. Introduction

Higher level languages such as Matlab are often thought to be more productive to program in than Fortran or C. PGAS (partitioned global address space) languages are believed to be easier to use than message passing environments. There are several other widely held beliefs about programming and productivity; a well designed IDE might be more productive than command line tools, a debugger might be more productive than debugging by printing, interactive programming environments might allow for quicker development than compiled languages, and so on.

Such hypotheses are often anecdotal — it is hard to prove or disprove them. It should be possible to confirm or refute hypotheses about programmer productivity with a reasonable model of programming workflows coupled with experimental evidence. Quantitative analysis is desirable, but very hard to get.

We believe that our work can lead to an ability to choose a programming environment based on quantitative evaluations instead of anecdotal evidence. Several studies, for example,\(^1\)\(^2\)\(^3\) compare different software engineering processes and programming environments in a variety of ways, mostly qualitative.

Programmers go through an identifiable, repeated process when developing programs, which can be characterized by a directed graph workflow. TMMs (timed Markov models or timed Markov processes) are one way to describe such directed graphs in a quantifiable manner. We describe a simple TMM that captures the workflows of programmers working alone on a specific problem. We then describe an experimental setup in which we instrument the student homeworks in the parallel computing class at UC Santa Barbara. We also describe the tools we developed for instrumentation, modelling, and simulating different what-if scenarios in the modelled data. Using our model and tools, we compare the workflows of graduate students programming the same assignment in C/MPI\(^4\) and UPC\(^5\) — something that is not possible without a quantitative model and measurement tools.

We describe Timed Markov Processes in Section 2. In Section 3, we describe our modelling of programmer productivity with TMM. Section 4 contains a description of our data collection methodology. We compare UPC and C/MPI data in Section 5. In Section 6, we talk about the various tools we are designing to allow anyone else to perform similar analysis. We finish with concluding remarks in Section 7.
2. Timed Markov Processes

The process of software development is iterative and probabilistic. It is iterative in the sense that a programmer often repeats a sequence of steps in the software development process; edit, compile, launch test run, for example. It is probabilistic in that the times in each of the steps of the process can vary, and the number of times a cycle will be repeated before the programmer can move on to the next phase is unpredictable. A timed Markov process can model both aspects.

A timed Markov process is a Markov process, augmented with dwell times for state transitions. Each state transition has both a probability of transition and a dwell time associated with it. Timed Markov processes closely resemble signal flow graphs, for which well known methods exist to estimate time to completion. Iterative design processes and software development processes have been studied using similar techniques.

In Figure 1, \( \text{prob}(B|A) \) is the probability of transitioning to state B given that the process is now in state A. \( \text{time}(A|B) \) is the dwell time spent in state A given that the next state transitioned to is B. \( \text{prob}(C|A) \), which would equal \( 1 - \text{prob}(B|A) \) in this example, is the probability of transitioning to state C given that the process is now in state A.

3. Timed Markov models of programmer workflows

Figure 2. Lone programmer workflow.

---


In our earlier work, we hypothesize a simple timed Markov model of a researcher developing a new application. It represents our assumption that the researcher begins by formulating a new algorithm for solving the problem of interest and then writes a program implementing the algorithm. Following that, the researcher enters a correctness-debugging loop, around which the process cycles until the program is deemed to be free of programming errors. Next is a performance-tuning loop, which cycles until the program has been tuned enough that it gets adequate performance for large problems to be run on an HPC system. This is the workflow shown in Figure 2. In this workflow:

- $T_f$ represents the time taken to formulate the new algorithmic approach.
- $T_p$ is the time necessary to implement the new algorithm in a program.
- $T_c$ is the compile time.
- $T_t$ is the time necessary to run a test case during the debugging phase.
- $T_d$ is the time the programmer takes to diagnose and correct the bug.
- $T_r$ is the execution time for the performance tuning runs. This is the most obvious candidate of a constant that should be replaced by a random variable.
- $T_o$ is the time the programmer takes to identify the performance bottleneck and program an intended improvement.
- $P_p$ is the probability that debugging reveals a necessity to redesign the program.
- $P_d$ is the probability that more debugging is necessary.
- $P_o$ is the probability that more performance optimization is necessary.
- $q_p$, $q_d$, and $q_o$ are $1 - P_p$, $1 - P_d$, and $1 - P_o$, respectively.

This model can also be used to describe the workflow of graduate students programming homework in a parallel computing class. We instrumented their programs, collected workflow data and fit it to such a timed Markov model in our 2004 classroom experiment. While fitting the experimental data to the model, we discovered that in addition to the transitions described above, there were two more transitions. A programmer may introduce or discover a bug while attempting to optimize the program. As a result, there is a transition from the Run state to the Debug state. Sometimes, the last run may not be successful, perhaps because of a failed attempt to optimize. In such a case, an earlier correct run is treated as the final program — all the data we present is from programs that were eventually correct. Hence, there is another transition from Test to Finish. Figure 3 shows the result of our analysis of the 2004 classroom experiment.

Figure 3. Fitting data from the 2004 classroom experiment to a timed Markov model.
4. Instrumentation and data collection

We used a comprehensive framework to collect data in our 2006 classroom experiment. We will describe our 2006 classroom experiment in full detail in a separate paper. Briefly, the programmers programmed the Game of Life on a large grid, which would not fit in the memory of a typical desktop computer. The Game of Life is played on a two dimensional grid. Cells may be alive or dead; their state evolves according to a set of simple rules based on the state of their neighbors. Half the programmers used C/MPI, while the other half used UPC.

Our data collection process gathers enough data at compile time and run time so that programmer experience can be accurately recreated offline. This allows us to replay the sequence of events (every compile and run) and collect specific data that may be required by the modelling process but was not captured while the experiment was in progress. We used such replays very effectively for our 2004 classroom experiment. Our replay capabilities were not perfect then, and we had to use reasonable proxies for run time parameters. We refined our techniques in light of the lessons learned, and achieved perfect replay capabilities for the 2006 classroom experiment.

The programmers were provided with a basic harness that included build infrastructure, data generators, test cases and specific performance targets. We captured timestamps for every compile and run, stored every code snapshot at compile time, and recorded run time information such as number of processors, inputs, correctness and compute time. Due to a glitch in the run time data collection, some of the run time information we needed for the modelling process was missing. However, we were able to gather the missing run time information by replaying every single compile and run for every programmer. The replays took 10 hours to compile, and 950 hours of processor time for all runs.

In our 2004 classroom experiment, the programmers wrote a parallel sorting code using C++/MPI. Since the model based on timed Markov processes was proposed after the 2004 classroom experiment, we used replays to gather the required data and fit it to a timed Markov model. Our experience with data gathering and modelling for the 2004 and 2006 classroom experiments has led us to believe that the replay is the most important facet of the data gathering process.

5. Comparing UPC and C/MPI workflows

Figure 4. TMM fit to UPC workflow data. Edges representing state transitions are labelled as: probability of transition / dwell time in seconds.
Figure 4 shows the workflow of UPC programmers. Figure 5 shows that of C/MPI programmers. These diagrams of the TMMs were prepared using the TMMsim tool described in Section 6.2. This is a preliminary analysis with a small sample size (five programmers using each language). Thus we do not attempt to draw final conclusions comparing the two languages. However, a number of aspects of these TMMs seem encouraging as regards the feasibility of this type of quantitative analysis.

First, the fitted transition times and probabilities from the 2006 classroom experiment are quite similar to those from the 2004 classroom experiment. Not surprisingly, most (92% to 95%) of the test runs lead back into the debug cycle. We see that a “test” run is successful 8% of the time for C/MPI and 5% of the time for UPC; however, in the optimization cycle, 28% of C/MPI runs introduced new bugs compared to only 24% of UPC runs. It is not clear whether these differences are significant for this small sample size. A programmer spends much longer to attempt an optimization (763 seconds for UPC and 883 seconds for C/MPI) than to attempt to remove a bug (270-271 seconds). The time to optimize UPC (763 seconds) is smaller that for MPI (883 seconds), suggesting perhaps that UPC optimization is carried out in a more small-granularity, rapid-feedback way.

6. Tools

The amount of data that needs to be analyzed to produce models of programmer workflows is quite large. We are developing automated tools for visualization, modelling, and simulation of TMMs to facilitate the kind of analysis described in earlier sections.

6.1 A tool for automatic TMM generation from collected data

There are two main types of data that are being collected in the experiments. Physical activities such as code edits, compiles, and executions are automatically captured by the instrumented development environment. During development, in some experiments, the students are also asked to record the time they spend performing logical activities such as thinking, serial coding, parallel coding, and testing. It is these logical activities that we use to create TMMs of the workflows. Alternatively, physical activities can be mapped to logical activities using a set of heuristics.
Whether the logical activities come from student logs\textsuperscript{11} or heuristic mapping,\textsuperscript{9,12} the end result is a list of activities and associated effort (measured in hours), as shown in Figure 6. We have created a Python program that parses this list of activities for each student and counts the transitions and dwell times for each activity. In the example shown, the student starts in the planning stage and then transitions to serial coding. This is represented in the transition matrix as $T_{12} = 1$. Consecutive entries for the same activity are combined. Thus in the dwell time matrix, the amount of time spent in the planning state before transitioning to the serial coding state is represented as $D_{12} = 1 + 3 = 4$. These transitions and dwell times can be aggregated across students and similar assignments to create a larger sample for analysis.

We calculate the probability for each state transition from the transition matrix as:

$$\text{prob}(j|i) = \frac{T_{ij}}{\sum_j T_{ij}}.$$  

Similarly, the average dwell time for each transition is calculated as:

$$\text{time}(j|i) = D_{ij}/T_{ij}.$$  

Once the transition probabilities and dwell times have been computed, the next step is to generate a graph description that can be used to visualize the TMM. Our initial choice for visualization was the Graphviz tool, which uses the DOT language for graph description. Figure 7 shows the student workflow from Figure 6 visualized as a TMM using Graphviz. Using Graphviz we have created a graphical browser for rapid visualization of multiple data sets (see Figure 8).


6.2 A tool for representing and simulating TMMs

Even with the help of automated tools, such as our prototype described in Section 6.1, the mapping from the activity data collected during a software productivity experiment to a workflow diagram and its corresponding TMM will rarely be trivial. Programmers do not necessarily follow the exact same steps when developing or modifying code. The detailed actions programmers take can be abstracted to higher-level workflow steps in a variety of ways, some of which may turn out to be much more faithful models than others. Thus we expect that it may take several tries to obtain an acceptably accurate mapping from experimental data to a workflow.

We therefore wanted a tool with which we could:
- Draw and annotate new TMMs,
- Import, view and modify previously defined TMMs such as those generated by the Python-based tool described in Section 6.1, and
- Run discrete-event simulations of a TMM, in order to estimate the expected time through the workflow that it represents.

We based “TMMsim,” our TMM drawing and simulation software on VGJ, a Java-based tool (“Visualizing Graphs in Java”) for drawing directed and undirected graphs. The key feature we added was the ability to annotate each edge of the graph with a dwell time and a probability of departing the previous state via this edge, as required by a timed Markov model. The tool uses a slightly extended version of the Graph Modeling Language (GML) to input previously defined TMMs, including those generated by our TMM generation tool (Section 6.1). Simple simulation code calculates the average time through the TMM workflow of 100,000 instances of its use. Figure 9 shows the TMM viewer/simulators user interface, along with a dialog box that has been opened in order to annotate the edge of an example TMM with a dwell time and a transition probability. The “Range” buttons in the edge annotation dialog allow a sequence of values to be used instead of a single number for the dwell time or the probability, causing a simulation average to be generated for each of the values, for the sake of sensitivity analysis.

---

Figure 8. TMM visualization GUI.

7. Conclusion

We believe that programmers go through an identifiable, repeated process when developing programs, which can be characterized by a directed graph model such as timed Markov models. We successfully gathered data and fit it to timed Markov models twice, in our 2004 and 2006 classroom experiments. The replay of programmer experience offline was one of the most important aspects of the data gathering process. The timed Markov models clearly indicate the most time intensive parts of the development cycle, quantitatively confirming our intuition — programmers spend most of their time debugging and, once they get a correct program, tuning it for performance is even more difficult. Our data also suggests, in this context, that programmers may introduce slightly fewer bugs in UPC programs and find it easier to optimize them, as compared to C/MPI programs.

Clearly, this is only the beginning. A lot more data needs to be collected before languages can be compared in a meaningful way. Towards this end, we are building various tools for the community at large. These tools will provide a general framework for data collection and model construction to study programmer productivity in a variety of ways.

Acknowledgments
The work of John Gilbert and Viral Shah was partially supported by the Air Force Research Laboratories under agreement number AFRL F30602-02-1-0181 and by the Department of Energy under contract number DE-FG02-04ER25632.
A Compiler-guided Instrumentation for Application Behavior Understanding

Abstract

The standard approach to the problem of application behavior understanding relies on instruction-level instrumentation. This approach generates humongous volumes of data that overwhelm programmers in trying to understand the cause-effect relationships in their applications and thus improve the performance of their codes. This article describes an integrated compiler and run-time approach that allows the extraction of relevant program behavior information by judiciously instrumenting the source code and deriving performance metrics such as range of array reference addresses, array access stride information or data reuse characteristics. This information ultimately allows programmers to understand why the performance is what it is on a given machine as they relate to program constructs that can be reasoned about. We describe the overall organization of our compiler and run-time instrumentation system and present preliminary results for a selected set of kernel codes. The approach allow programmers to derive a wealth of information about the program behavior with a run-time overhead of less than 15% of the original code's execution time making this attractive to instrument and analysis of code with extremely long running times where binary-level approaches are simply impractical.

1. Introduction and Motivation

Understanding the performance of modern high-performance computing machines has become notoriously difficult. These architectures expose to the programmers different hardware mechanisms that often interact in not so predictable ways. Aiming at improving their performance, programmers must resort to low-level instrumentation techniques, such as binary rewriting, to capture run-time execution data from which they hope to understand the performance behavior of the program. This approach, however, generates huge volumes of raw data at a level of abstraction that is seldom adequate for programmers to relate to the source code of their applications. Programmers are thus left guessing which style of coding is adequate for the compiler and the target architecture at hand. Worse, when one of these two components change, the learning investment is wasted, as the process must be restarted.

In this work we describe an alternative approach that relies on high-level source code instrumentation guided by the application of compiler analyses techniques. The basic idea is to judiciously instrument the application source code to extract various execution metrics that can be related to the source code. While some of the aspects of the compiler-architecture interactions are not directly modeled, the fundamental program behavior is retained at a level of abstraction that allows the compiler to relate the observed metrics to source-level program constructs programmers can understand. An added benefit exists in the isolation of the approach from the compiler effects or idiosyncrasies and thus provides a path for program behavior prediction for future architectures for which a compiler or the actual machines do not yet exist.

Consider as an illustrative example the code in figure 1. For each invocation of the innermost loop, the computation accesses \( m \) elements of the array \( A \) with a stride of 1 and \( m \) elements of the array \( B \) with a stride that is dictated by the array size of \( B \). There are several key observations in this example. First, the accesses to the array \( A \) and \( B \) are dependent on the range of values \( n \) and \( m \) which are not known at compile time. Second, the range of addresses to the array \( A \) are repeated across the invocations of the \( i \) loop. Finally, and although some of the values are not known, such as the actual value of \( k \), its value is not modified throughout the execution of both loops and thus can be safely considered to be loop invariant with respect to these two loops.
A Compiler-guided Instrumentation for Application Behavior Understanding

for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
        ... = A[k][i];
        ... = B[j][i];
    }
}

Figure 1. Illustrative example: original source code (left) and instrumented source code (right).

A way to capture the behavior of the program in more detail is to instrument it at the source-code level as shown on the right-hand-side of figure 1. Here we have augmented the original code with calls to instrumentation functions with the appropriate arguments. For instance, saveAddrRange(Ref1, A[k][0], A[k][n-1]); will record that at the particular execution point in the program, the range defined by the two arguments A[k][0],A[k][n-1] will be accessed m times. In this particular instrumentation, we are taking advantage of the fact that the range of addresses is clearly invariant with respect to the i-loop and thus can be hoisted outside the two loops. In this example the loop bounds are symbolically constant and the array access functions are affine with respect to the loop index variables. In the end with the results obtained by this instrumentation, we can learn valuable information regarding the range of addresses the computation accesses and/or the access patterns and thus locality of reference.

Many other authors have addressed the problem of understanding and quantifying the amount of reuse in applications. In the context of scientific application codes, such as the SpecFP benchmark suite, Sherwood et. al.1 developed a low-level instruction instrumentation approach that identifies the phases of a given computation in terms of the similarities between executed instructions. Other authors have focused on cache locality metrics (e.g.,234) whereas other use temporal reuse distance or stack distances.56 Weinberg et. al.7 have implemented the locality scores described in this document using at a very low-level instrumentation approach.

All these efforts have in common the fact that they instrument the addresses and instruction issued by a running executable directly at a very low-level.891011 They offer the promise of high precision, provided one can cope with the enormous data volume they generate or the substantial application slowdown. The approach described in this document explores the opposite end of the instrumentation spectrum. It does not rely on low-level instrumentation, but on high-level source code instrumentation. This approach is faster but comes at a loss in precision, with the additional handicap that it does not take into account the compiler effects – how the compiler has modified the generated code to included hardware-oriented execution transformations such as pre-fetching or speculation. The fundamental premise of this work is that it is possible with little source level instrumentation overhead to capture a large amount of program execution information.

2. Technical Approach

We now describe in more detail the use of compiler analyses and source-level code instrumentation to extract as much information about the program behavior as possible and with the least amount of run-time overhead.

2.1. System Architecture

Figure 2 depicts the overall system architecture. We use the Open64 infrastructure for program internal representation and transformation.12 Given a source file, the compiler performs a series of

12 Open64 Open Source Compiler Infrastructure — http://open64.sourceforge.net/
analyses and generates a set of static information files. These files have some of the information about the program structure such as basic blocks and loop organization, and about the program structure the compiler can determine statically. Next, the compiler instruments the source code with library function calls to capture at run-time the missing pieces of information. At run-time the execution collects these missing pieces of information in internal data structures to be output to selected files when the program executes with real input data. These instrumented files are then compiled with the native compiler to generate an executable that is linked against the instrumentation library.

2.2. Compiler Analyses

We have implemented the following basic compiler analyses in Open64 that form the basis for our instrumentation:

- Basic Block Recognition: The analysis uses the basic control flow information in the Open64 intermediate representation (Whirl) to recognize basic blocks and loop structure in addition to call graph;
- Induction Variable Recognition: We derive induction variables, in some cases capturing induction variables across multiple control flow branches, and as always focusing on scalar variables only;
- Loop Invariants and Symbolic Constant Loop Bounds: Using induction variables and loop index variables (in Fortran) we recognize which bounds of the inner loops are symbolically constant, i.e., their value is unknown at compile time but does not change through the execution of the loop they control;
- Data Access Patterns in Array Expressions: We focus on affine index function recognition to determine, either statically or with the help of run-time information, the stride of the array accesses for a given array reference.

Using the basic information from these analyses the compiler instruments the code to determine the following program execution properties:
- Basic Block Execution Frequency: the absolute number of times a given basic block is executed.
  In many cases with symbolically constant expression, we hoist the expression that evaluates the number of executions outside the loop that encloses the corresponding basic block.
- Loop Bounds: The actual values of the loop bounds and thus the number of loop iterations across all invocations of a given loop construct.
- Virtual Array Address Ranges. For each array reference, the instrumentation keeps track of the range of addresses accessed for each invocation of the loop.
- Array stride Information: Determine stride of consecutive accesses for each given reference measured in terms of array index function and not the virtual address accessed.

### 2.3. Code Generation and Instrumentation

The compiler uses the static information to derive as precise information as possible with as little instrumentation as required. For example, if the loop bounds of a given array are not known at compile time but they depend on variables that do not change through the execution of the loop (i.e., they are symbolically constant), the compiler needs only to capture the value of the bounds once and determine at run-time (with the help of an auxiliary library routine) what the actual bounds are. In extreme cases, however, the compiler may not be able to ascertain if a given expression is symbolically constant, it thus resorts to a very simplistic instrumentation that keeps track of every single loop iteration execution by inserting a counter in the transformed code. The various levels of instrumentation at the source-code level are depicted in figure 3 below.

(a) Original code

```
741:      do 10 i = 1, n
742:         …
743:  10  continue
```

(b) Instrumented code

```
740:      call SaveLoopBounds(1,n)
741:      do 10 i = 1, n
742:         …
743:  10  continue
```

(c) Original code

```
747:      do 10 k=c(j),c(j+1)-1
748:         …
749: 10  continue
```

(d) Instrumented code

```
746:      call saveLoopBounds(c(j),c(j+1)-1)
747:      do 10 k = c(j),c(j+1)-1
748:         …
749: 10  continue
```

(e) Original code

```
747: 100
748:      …
749:      goto 100
```

(f) Instrumented code

```
746:      call setCurentLoop(loopid)
747: 100  call saveLoopIteration()
748:      …
749:      goto 100
```

Figure 3. Illustration of source-code instrumentation levels; Basic with symbolic constant bounds (a,b) and (c,d), and with restricted loop structures (e,f).

This instrumentation focuses on capturing metrics at the basic block level. For this reason, the compiler assigns a unique identifier to each basic block, thus allowing a post-processing tool to gather the various basic block metrics and convolve the corresponding execution frequencies and symbolic array information for analysis. Overall, this source level instrumentation is accomplished by the declaration of extra counter variables and library calls whose placement and arguments are determined by the compiler analysis.
3. Applications to Locality Metrics

As an application of these static and run-time instrumentation techniques, we now describe a specific program behavior model - locality metrics. In particular, we are interested in computing a temporal and spatial locality metric that can quantitatively distinguish between programs with distinct memory access pattern behaviors and thus distinct memory hierarchy performance.

3.1. Spatial and Temporal Locality

A program will exhibit high spatial locality if it tends to visit memory locations that are close to each other in space. For instance, if a program accesses the data organized as an array with a stride of 1, it exhibits very high spatial locality. If the memory access patterns are very random, its spatial locality is low. Conversely, a program will exhibit a high temporal locality if it tends to repeatedly revisit the same memory locations in a small window of accesses. Given these loose concepts in a computation, we plot the memory access behavior of a computation on a two-dimensional axis normalized between zero and one on both temporal and spatial locality axis as depicted in figure 4. A computation that randomly accesses memory will exhibit both very low spatial and very low temporal locality, and will therefore be plotted in the corner of the diagram close to the origin. A computation that scans an extremely large portion of the memory in a stride 1 array access pattern and that never revisits the same location will have high spatial locality but low temporal locality. On the other hand, a computation that revisits data often and scans the memory using a unit stride, as is the case with common matrix-matrix computations, will exhibit high temporal and spatial localities and thus be close to the (1,1) corner of the diagram. Finally, it is possible to conceive a computation that has very random behavior but revisits the same randomly visited locations often. This computation will have high temporal locality but low spatial locality.

These concepts of spatial and temporal locality have been recently explored by Weinberg et al., who developed a series of scoring metrics to approximate more rigorous definitions of spatial and temporal locality. The implementation of their spatial and temporal locality scores is very expensive and calls for the analysis of every memory access a computation issues to determine if the specific location has been visited in the past and if so, how long ago. They also classify the various data access streams trying to uncover the stride information as they define a spatial score as a direct function of the stride associated with a given data stream.

Figure 4. Spatial and temporal locality chart (colored marks from data in [6], black diamonds using the approach presented in this article).
3.2. Spatial and Temporal Scoring

In this work, we define two scoring metrics for the spatial and temporal localities of access streams generated by array references. A stream generated by an array reference that traverses a range of addresses from a lower bound \((lb)\) to an upper bound \((ub)\) with stride \((st)\) exhibits a spatial locality of \(1/st\). Thus a unit stride walk through the elements of an array will correspond to a 1.0 spatial locality scoring and an increasingly longer stride will lead to very low score values. A random scan of the array locations will lead to a spatial score of zero. Intuitively, this model matches the cost model of a cache-supported memory hierarchy, as shorter strides will increase the likelihood of a given data item being on the cache due to the line-size effects. In this work, we use the average stride information that the run-time instrumentation uncovers by using the static and dynamic information to compute a weighted sum over all basic blocks of the spatial locality score, as given below. The weights are simply the relative frequency with which each basic block is executed.

\[
\text{Spatial Score} = \frac{\text{NumStrides} \times \left( \frac{\text{Num Reuse}(\text{stride})}{\text{M}} \right)}{\text{Temporal Score}} = \sum_{i=0}^{\log_2(N)} \text{Reuse}(2^i) \times \frac{\log_2(N) - i}{\log_2(N)}
\]

Figure 5. Spatial and temporal scoring metrics.

The choice of metric to use to capture temporal reuse is far from being widely accepted. A naïve approach relies on keeping track of every accessed address and determining upon a match the distance, in terms of the number of accesses, between repeating address locations. Use of this approach at the lowest level of instrumentation makes this an infeasible approach as suggested by Snavely et al., where the authors report application slowdowns in the range of 10x to 20x. This effectively limits the scope of the instrumentation and forces the implementation to rely on sampling techniques for feasibility.

Instead, we rely on high-level source code instrumentation and program analysis to reduce the number of array references that are instrumented. In our own experiments, we observed a slowdown in the order of 15% much lower than the slowdown reported by the lower-level instrumentation techniques. Specifically, we use the temporal scoring expression shown below as reported originally in [7]. In this scoring, temporal locality is seen as a weighted sum of the fraction of the dynamic references that exhibit a reuse distance of \(2^i\) for decreasing weights with the increased reuse distance. The weight factor \(\text{Weight}(i) = \log_2(N) - i\) for references with a reuse distance of \(2^i\) gives a higher weight to references with a smaller distance and much less to increasing reuse distances. The implementation simply keeps track of how many references do exhibit a reuse of increasingly larger values (as powers of 2) and keeps track of the number of memory references.

4. Experimental Results

We now describe the utilization of the compiler analysis and code instrumentation infrastructure, described in the previous two sections, to several kernels codes, respectively, the CGM kernel from the NAS benchmark suite, a streaming kernel streamadd, and randomaccess from the HPCS benchmark suite. Whereas the spatial and temporal behavior of the HPCS kernels is fairly intuitive, the CGM kernel offers a mix of simple behaviors making it a richer example to experiment with.

4.1. The Kernels

The NAS CGM kernel is written in FORTRAN and implements the conjugate-gradient (CGM) iterative refinements method for a positive-definite input sparse-matrix. At the core of this kernel...
is the matvec subroutine that implements the sparse-matrix vector multiplication computation as depicted in the figure below.

```fortran
subroutine matvec(n, a, rowidx, colstr, x, y)
  integer rowidx(1), colstr(1)
  integer n, i, j, k
  real*8 a(1), x(n), y(n), xj
  do 10 i = 1, n
    y(i) = 0.0d0
  10  continue
  do 200 j = 1, n
    xj = x(j)
    do 100 k = colstr(j), colstr(j+1)-1
      y(rowidx(k)) = y(rowidx(k)) + a(k) * xj
    100  continue
  200  continue
  return
end
```

Figure 6. Source code for the CGM kernel.

The streamadd kernel is written in C and consists of a simple array vector addition loop. The length of the vector and thus of the manipulated arrays is fairly large, say 10,000 elements, to exercise the bandwidth of the memory subsystem. The randomaccess kernel, also known as the gups kernel, performs many updates to randomly selected locations in memory where the addresses of the locations are pre-computed in a loop before the updates take place. Figure 7 below depicts the relevant portions of the C source code for these kernels.

```c
for(i=0; i < N; i++){  24:    for(i=0; i < M; i++) {
09:        c[i] = a[i] + b[i];  25:        loc[adder[i]]++;
10:    }     26:    }
```

Figure 7. Source code for the streamadd (left) and randomaccess (right) kernels.

### 4.2. Analysis Results

For each code, the analysis identifies the number of basic blocks and instruments the code to capture the corresponding frequency of execution. Table 1 below depicts the various results for the various kernels. Row two presents the number of basic blocks and row three the minimum, maximum and the average number of times one of these basic blocks executes. Row four presents the number and type of strides the static analysis determines whereas row five presents the corresponding spatial locality score. Row 6 shows the temporal locality score using the formulation presented in figure 4 above.

<table>
<thead>
<tr>
<th></th>
<th>streamadd</th>
<th>randomaccess</th>
<th>NAS CGM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of Basic Blocks</strong></td>
<td>5</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td><strong>Frequency of Execution (min, avg, max)</strong></td>
<td>[1:200;1,000]</td>
<td>[1:2,000,000]</td>
<td>[1:269,729;1,853,104]</td>
</tr>
<tr>
<td><strong>Stride and Reference Information</strong></td>
<td>3 refs stride 1</td>
<td>1 ref stride 0; 2 refs random</td>
<td>2 refs random; 7 refs stride 1</td>
</tr>
<tr>
<td><strong>Spatial Score</strong></td>
<td>0.999</td>
<td>0.001</td>
<td>0.714</td>
</tr>
<tr>
<td><strong>Temporal Score</strong></td>
<td>0.1</td>
<td>0.000</td>
<td>0.422</td>
</tr>
</tbody>
</table>

Table 1. Experimental results for 3 code kernels.
Notice that the spatial score taking into account the frequency with which each basic block executes and thus although CGM has seven references with stride 1 in the various basic blocks of the core of this computation its spatial score is lower that the number of strided-1 references would otherwise suggest. This is due to the fact that some basic blocks with high spatial score execute very seldom.

4.3. Discussion

The results presented here resulting from the combination of static and dynamic program behavior information are very encouraging. First, the spatial and temporal locality are qualitatively very close to the results for the same kernel codes as reported by Snavely et. al., and are qualitatively identical. This is depicted by the black diamonds in figure 4. The basic difference for the random access kernel stems from the fact that in different implementations and with different compilers they do have slightly different methods to deal with table (array) accesses. Second, they rely on high-level instrumentation with much lower overheads making them feasible for instrumentation in larger scalar applications that can run to completion much more realistic runs that are possible today with low-level instrumentation approaches. Lastly, this approach allows us to derive much more accurate pictures of the behavior of the program without being cluttered or overwhelmed with the amount of data generated by low level approaches. For instance, using the approach described here we can determine why on the spatial or temporal locality, what they are and what are the dynamic references that contribute to the various metrics. Whereas low-level instrumentation approaches can also determine the source of the program behavior they lack the connection to the source code level this approach inherently preserves.

5. Conclusion

Understanding the behavior of programs is a long-standing and sought-after goal. The traditional approach relies on low-level instrumentation techniques to extract performance or execution metrics that can indicate why the performance of a given program is what it is. The high overhead of this approach, the inherent Heissenberg effects of instrumentation and the sheer volume of generated data make it difficult to correlate program behavior with source code constructs. In this article we have described an integrated compiler and run-time approach that allows the extraction of relevant program behavior information by judiciously instrumenting the source code, and deriving performance metrics such as range of array reference addresses and stride and reuse distance information. We have illustrated the application of this approach in instrumenting several kernel codes for spatial and temporal locality scoring. For these kernel codes, this approach derives concrete values for these locality metrics that are qualitatively identical to the scoring obtained by low-level instrumentation of the code but at a much lower execution time cost. This suggests the approach to be extremely valuable in allowing large codes to execute in realistic settings.

Acknowledgments

This research has been sponsored by the US Department of Energy (DoE) under contract number DE-FG02-04ER25597.
Symbolic Performance Modeling of HPCS Applications

1. Introduction

Performance and workload modeling have numerous uses at every stage of the high-end computing lifecycle: design, integration, procurement, installation, tuning, and maintenance. Despite the tremendous usefulness of performance models, their construction remains largely a manual, complex, and time-consuming exercise. Many of these techniques serve the overall purpose of modeling but few common techniques have gained widespread acceptance across the community. In most cases, researchers create models by manually interrogating applications with an array of performance, debugging, and static analysis tools to refine the model iteratively until the predictions fall within expectations. In other cases, researchers start with an algorithm description, and develop the performance model directly from this abstract description. In particular, DARPA’s High Productivity Computing Systems (HPCS) program requires understanding and predicting application requirements almost eight years in advance, when prototype hardware and perhaps even system simulators do not exist. In this light, performance modeling takes on a critical importance because system architects must make choices that match application workloads while DARPA and its HPCS mission partners must set aggressive but realistic goals for performance.

In this article, we describe a new approach to performance model construction, called modeling assertions (MA), which borrows advantages from both the empirical and analytical modeling techniques.1,2 This strategy has many advantages over traditional methods: isomorphism with the application structure; easy incremental validation of the model with empirical data; uncomplicated sensitivity analysis; and straightforward error bounding on individual model terms. We demonstrate the use of MA by designing a prototype framework, which allows construction, validation, and analysis of models of parallel applications written in FORTRAN and C with the MPI communication library. We use the prototype to construct models of NAS CG, SP benchmarks3 and a production level scientific application called Parallel Ocean Program (POP).4

A further advantage of our approach is that the MA symbolic models encapsulate an application’s key input parameters as well as the workload parameters, including the computation and the communication characteristics of the modeled applications. The MA scheme requires an application developer to describe the workload requirements of a given block of code using the MA API in the form of code annotations. These code annotations are independent of the target platforms. Moreover, the MA scheme allows multi-resolution modeling of scientific applications. In other words, a user can decide which functions are critical to a given application and can annotate and subsequently develop detailed performance models of the key functions. Depending on the runtime accuracy of the model, a user can develop hierarchical, multi-resolution performance models of selected functions, for instance, models of critical loop blocks within a time-consuming function. MA models can capture the control structure of an application. Thus, not only an aggregated workload metric is generated, but also the distribution of a given workload over an entire execution cycle can be modeled using the MA framework.

The outline of the paper is as follows: the motivation behind the modeling assertion technique is presented in section 2. Section 3 explains the components of the Modeling Assertions framework. Section 4 describes model construction and validation using the NAS CG benchmarks. Section 5 presents the scalability of the NAS CG and SP benchmarks and POP communication behavior together with an analysis of sensitivity of workload requirements. Section 6 concludes with benefits and contributions of the modeling assertions approach to performance modeling studies.

Sadaf R. Alam
Oak Ridge National Laboratory
Nikhil Bhatia
Oak Ridge National Laboratory
Jeffrey S. Vetter
Oak Ridge National Laboratory

4 Parallel Ocean Program (POP) - http://climate.lanl.gov/Models/POP/
2. The MA Framework

In order to evaluate our approach of developing symbolic models with MA, we have designed a prototype framework. This framework has two main components: a library for instrumenting applications and a post-processing toolset. Figure 1 shows the components of the MA framework. The MA API is used to annotate the source code. As the application executes, the runtime system captures important information in trace files. These trace files are then post-processed to validate, analyze, and construct models. The post-processor currently has three main classes: model validation, control-flow model creation, and symbolic model generation classes. The symbolic model shown in the Figure 1 is generated for the MPI send volume. This symbolic model can be evaluated, and is compatible, with MATLAB and Octave.

![Diagram of MA framework with code snippets and classes](image)

Currently, MA can be used to model computation and communication of applications. The MA API provides a set of functions to annotate a given FORTRAN or C code that uses MPI message-passing for communication. For example, `ma_loop_start`, a MA API function, can be used to mark the start of a loop. Upon execution, the code instrumented with MA API functions generates trace files. For parallel applications, one trace file is generated for each MPI task. The trace files contain traces for `ma_xxx` calls and MPI communication events. Most MA calls require a pair of `ma_xxx_start` and `ma_xxx_end` calls. The `ma_xxx_end` traces are primarily used to validate the modeling assertions against the runtime values. In our current implementation, the assertions for the number of floating-point operations, `ma_flop_start/stop`, invoke the PAPI hardware counter API to collect empirical data and compare it to the prediction. The `ma_mpi_xxx` assertions on the other hand are validated by implementing MPI wrapper functions (PMPI) and by comparing `ma_mpi_xxx` traces to `PMPI_xxx` traces. Additional functions are provided in the MA API to control the tracing activity; for example, to control the size of the trace files, by enabling and disabling the tracing at both compile and run time. At runtime, the MA runtime system (MARS) tracks and captures the actual instantiated values as they execute in the application. MARS creates an internal control flow representation of the calls to the MA library as they are executed. It also captures both the symbolic values and the actual values of the expressions. Multiple calls to the same routines with similar parameters is projected onto the same call graph, therefore, the data volume is reduced.

3. Construction and Validation of Symbolic Models

As a motivating example, we demonstrate MA on the NAS CG benchmark. NAS CG computes an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix, which...
is characteristic of unstructured grid computations. The main subroutine is `conj_grad`, which is called \( n_{iter} \) times. The benchmark results report time in seconds for the time step (do it = 1, \( n_{iter} \)) loop that is shown in Figure 2. Hence, we started constructing the model in the main program, starting from the `conj_loop`, as shown is Figure 2. The first step was to identify the key input parameters, \( na, n_{nonzer}, n_{iter} \) and `nprocs` (number of MPI tasks). Then, using `ma_def_variable_assign_int` function, we declared the essential derived values, which are later used in the MA annotations to simplify the model representation. Figure 2 only shows the MA annotations as applied to the NAS CG main loop. It does not show the annotations in the `conj_grad` subroutine, which are similar except for additional `ma_subroutine_start` and `ma_subroutine_end` calls. The annotations shown in Figure 2 capture the overall flow of the application at different levels in the hierarchy; loops (e.g., `conj_loop`, `mpi`), subroutines (e.g., `conj_grad`), basic loop blocks, etc. These annotations also define variables (e.g., \( na, n_{nonzer}, n_{iter}, n_{procs} \)), which are important to the user in terms of quantities that determine the problem resolution and its workload requirements.

At the lowest level is the `norm_loop` as shown in Figure 2. The `ma_loop_start` and `ma_loop_end` annotations specify that the enclosed loop executes a number of iterations (e.g., `l2npcols`) with a typical number of MPI send and receive operations. More specifically, the `ma_loop_start` routine captures an annotation name (i.e., `l2npcols = log2(num_proc_cols)`), a symbolic expression that defines the number of iterations in terms of an earlier defined MA variable (i.e., `num_proc_cols`).

```fortran
    do it = 1, \( n_{iter} \)
        call conj_grad ( colidx, reduce_recv_lengths )........
        do i = 1, \( l2npcols \)
            call maf_mpi_irecv( 'nrecv','dp*2'. dp*2,1)
            call mpi_irecv( norm_temp2, 2, dp_type ........
        call maf_loop_end('norm_loop',i-1)
    call maf_loop_start('conj_loop', 'niter', niter)
    do it = 1, \( n_{iter} \)
        call conj_grad ( colidx, reduce_recv_lengths )........
        do i = 1, \( l2npcols \)
            call maf_mpi_irecv( 'nrecv','dp*2'. dp*2,1)
            call mpi_irecv( norm_temp2, 2, dp_type ........
    call maf_loop_end('norm_loop',i-1)
```

Figure 2. Annotation of the CG benchmark with MA API calls.

The validation of an MA performance model is a two-stage process. When a model is initially being created, validation plays an important role in guiding the resolution of the model at various phases in the application. Later, the same model and validation technique can be used to validate against historical data and across the parameter space. The model verification output enables us to identify the most floating-point intensive loop block of the code in the CG benchmark. This loop block is shown in Figure 3, which is called twice during a conjugate gradient calculation in the CG benchmark. The symbolic floating point operation cost of the loop is approximately \( 2*na/(num_proc_cols*nonzer*ceiling(nonzer/nprows)) \).

```fortran
    do j=1,lastrow-firstrow+1
        sum = 0.d0
        do k=rowstr(j),rowstr(j+1)-1
            sum = sum + a(k)*p(colidx(k))
        enddo
        w(j) = sum
    enddo
```

Figure 3. The partition submatrix-vector multiply
Using the MA models, we generated the scaling of the floating-point operation cost of the loop block in Figure 3 with the other loop blocks within a conjugate gradient iteration. The model predictions are shown in Figure 4. The total cost of two invocations of the submatrix vector multiply operation contributes to a large fraction of the total floating-point operation cost. Loop L1 is the first loop block in the CG timestep iteration and L2 is the second. Figure 4 shows that the workload is not evenly distributed among the different loop blocks (or phases of calculations), and the submatrix vector multiply loop can be a serious bottleneck. Furthermore, as we scale the problem to a large number of processors, we begin to identify loops that are either the Amdahl’s proportions of the serial code or their loop count is directly proportional to the number of MPI tasks in the system. We found that the loop count of loop number 3 and 8 depend on the number of MPI tasks \( \log_2(\log_2(\text{MPI\_tasks})) \), while loop 1 and 8 scale at a slower rate than loop 2 and 7 (submatrix vector multiply loop), since the cost of loop 2 and 7 is divided twice by the scaling parameters as compared to 1 and 8, which is divided once by the scaling parameter. Another interesting feature is the scaling pattern, which is not linear because the mapping and distribution of workload depends on the ceiling \( \log_2(\text{MPI\_tasks}) \).

We collected the runtime data for the loops blocks in CG time step iterations on XT3\(^8\) and Blue Gene/L\(^9\) processors to validate our workload distribution and scaling patterns. Figure 5 shows the percentage of runtime spent in individual loop blocks. Comparing it with the workload distribution in Figure 4, we observe not only a similar workload distribution but also a similar scaling pattern. Note that the message passing communication times are not included in these runtime measurements. We collected data for the Class D CG benchmark on the XT3 system, which also validates the floating-point message count distribution and scaling characteristics that are generated by the symbolic MA models.

4. Scaling and Sensitivity Analysis

One of the aims of creating the models of scientific applications is to be able to predict the application requirements for the future problem configurations at scale. We use our MA models to understand the sensitivity of floating-point operations, memory requirements per processor, and message volume to applications’ input parameters.


4.1 NAS CG

We begin experiments with a validated problem instance, Class C, for both the NAS CG and SP benchmarks, and scale the input parameters linearly. Note that the MA framework has a post-processing toolset that allows validation of MA model annotations with the runtime values. For instance, the PAPI_FP_OPS (number of floating-point operations) empirical data was compared with the ma_flop predicted value. The validated problem instances, Class C, have na=150000, nonzer=15, for the CG Class C benchmark with 128 MPI tasks. We increase the value of na linearly and generate the floating-point and load-store operation count using the MA symbolic models of the NAS CG benchmark. Figure 6 shows that the floating-point and load-store cost in the CG experiments increase linearly with the na parameter value. Similarly, we generated the growth rates for the floating-point and load-store operation cost for the other input parameter, nonzer. Results in Figure 6 and Figure 7 show that the floating-point and load-store operation cost in CG are relatively more sensitive to the increase in the number of nonzer elements in the array than the array size: na.

4.2 NAS SP

In the second experiment, the NAS SP benchmark has a single application parameter, problem_size, which we have used to represent the workload requirements (floating-point, load-store, memory and communication) in the MA symbolic models. Figure 8 shows the increase in the floating-point and load-store operation count by increasing the problem_size linearly. Note that like CG, the initial set of experiments (Class S, W, A, B, C and D) are validated on the target MPP platforms. Figure 8 shows that the floating-point operation cost increases at a very high rate by increasing the problem_size.
Using the MA models, we not only generate the aggregated workload requirements shown earlier, but we also get an insight into the scaling behavior of the workload requirements within an application as a function of the \texttt{problem\_size} parameter. Figure 9 shows the contribution of different functions in total floating-point operation count in SP time step iterations. The results shown in Figure 9 are generated for a fixed number of MPI tasks and by increasing the \texttt{problem\_size} parameter linearly. The floating-point workload requirements generated by the MA model show that the \texttt{z\_solve} is the most expensive function for runs with large number of processors. The cost of \texttt{x\_solve} and \texttt{y\_solve} are identical and consistent. Moreover, based on the MA model results shown in Figure 9, we can safely ignore the cost of \texttt{txinvr} and add functions in the further analysis.

4.3 Parallel Ocean Program (POP)

Similarly, we studied the communication pattern of a climate application called Parallel Ocean Program (POP). POP is an ocean modeling code developed at the Los Alamos National Laboratory, which executes in a time-step fashion and has a standard latitude-longitude grid with \texttt{km} vertical levels. There are two main processes in a POP time-step: baroclinic and barotropic. Baroclinic requires only point-to-point communication and is highly parallelizable. Barotropic contains a conjugate gradient solver, which requires global reduction operations. Moreover, the discretized POP grid is mapped and distributed evenly on the two-dimensional processor grid. POP has two standard problem instances: \texttt{x1} and \texttt{.01}. Processor grid dimensions are compile-time parameters in POP.

We studied the overall communication volume sensitivity and the individual message distribution in POP by varying the MPI grid sizes. POP uses MPI topology functions to create the two-dimensional virtual topology. All point-to-point communication operations are between the four nearest-neighbors in the 2D grid. Figure 10 shows the increase in overall volume in the two calculation phases, while Figure 11 shows the distribution of the message volume. Note that most messages are less than 1 Kbytes. Using this information, it is now possible to accurately reason about the actual application requirements. In this case, the application requires an interconnect that has low latency and low overhead for small messages.

5. Conclusions and Future Directions

MA is a new technique that combines the benefits from both analytic and empirical approaches, and it adds some new advantages, such as incremental model validation and multi-resolution modeling. Within the HPCS program, these models are useful to perform sensitivity analysis for future problem instances of HPCS applications. Moreover, the symbolic models can be evaluated efficiently and hence provide a powerful tool for application and algorithm developers to identify scaling bottlenecks and hotspots in their implementations. From the perspective of constructing, validating, and evaluating performance models, we believe that MA offers many benefits over conventional techniques throughout the performance lifecycle.
Designing Scalable Synthetic Compact Applications for Benchmarking High Productivity Computing Systems

Abstract

One of the main objectives of the DARPA High Productivity Computing Systems (HPCS) program is to reassess the way we define and measure performance, programmability, portability, robustness and ultimately productivity in the High Performance Computing (HPC) domain. This article describes the Scalable Synthetic Compact Applications (SSCA) benchmark suite, a community product delivered under support of the DARPA HPCS program. The SSCA benchmark suite consists of six benchmarks. The first three SSCA benchmarks are specified and described in this article. The last three are to be developed and will relate to simulation. SSCA #1 Bioinformatics Optimal Pattern Matching stresses integer and character operations (no floating point required) and is compute-limited; SSCA #2 Graph Analysis stresses memory access, uses integer operations, is compute-intensive, and is hard to parallelize on most modern systems; and SSCA #3 Synthetic Aperture Radar Application is computationally taxing, seeks a high rate at which answers are generated, and contains a significant file I/O component. These SSCA benchmarks are envisioned to emerge as complements to current scalable micro-benchmarks and complex real applications to measure high-end productivity and system performance. They are also described in sufficient detail to drive novel HPC programming paradigms, as well as architecture development and testing. The benchmark written and executable specifications are available from www.highproductivity.org.

1. Introduction

One of the main objectives of the DARPA High Productivity Computing Systems (HPCS) program is to reassess the way we define and measure performance, programmability, portability, robustness and ultimately productivity in the High Performance Computing (HPC) domain. An initiative in this direction is the formulation of the Scalable Synthetic Compact Applications (SSCA) benchmark suite. Each SSCA benchmark is composed of multiple related kernels which are chosen to represent workloads within real HPC applications and is used to evaluate and analyze the ease of use of the system, memory access patterns, communication and I/O characteristics. The benchmarks are relatively small to permit productivity testing and programming in reasonable time; and scalable in problem representation and size to allow simulating a run at small scale or executing on a large system at large scale.

Each benchmark written specification presents detailed background and parameters for an untimed data generator and a number of timed application kernels. All of the SSCA benchmarks are intended to be scalable using any of a variety of techniques, a variety of languages, and a variety of machine architectures. Each SSCA includes a number of untimed validation steps to provide checks an implementor can make to gain confidence in the correctness of the implementation.

The SSCA benchmark suite consists of six benchmarks. The first three SSCA benchmarks are specified and described in this article. The last three are to be developed and will relate to simulation.

1. Bioinformatics Optimal Pattern Matching: This benchmark focuses on sequence alignment algorithms in computational biology. It stresses integer and character operations, and requires no floating point operations. It is compute-limited, and most of the kernels are embarrassingly parallel. (Section 2)
2. **Graph Analysis**: SSCA #2 is a graph theory benchmark representative of computations in informatics and national security. It is characterized by integer operations, a large memory footprint, and irregular memory access patterns. It is also relatively harder to parallelize compared to the other two SSCAs. (Section 3)

3. **Synthetic Aperture Radar Application**: This benchmark is characteristic of the computations, communication, and taxing data I/O requirements that are found in many types of sensor processing chains. SSCA #3’s principal performance goal is throughput, or in other words, the rate at which answers are generated. The benchmark stresses large block data transfers and memory accesses, and small I/O. (Section 4)

2 **SSCA #1: Bioinformatics Optimal Pattern Matching**

![Sequence alignment algorithms (SSCA#1) are used for protein structure prediction.](image)

The intent of this SCCA is to develop a set of compact application kernels that perform a variety of analysis techniques used for optimal pattern matching. The chosen application area is from an important optimization problem in bioinformatics and computational biology, namely, aligning genomic sequences. These references provide an introduction to the extensive literature on this problem space, some publicly available programs that address these problems, and the algorithms used in those programs: 3 4 5 6 7 8 9 10 11 12 13 14

2.1 **Bioinformatics**

A genome consists of a linear sequence composed of the four deoxyribonucleic acid (DNA) nucleotides (bases), which forms the famous double helix. The DNA sequence contains the information needed to code the proteins that form the basis for life. Proteins are linear sequences of amino acids, typically 200-400 amino acids in length. Each different protein twists naturally into a specific, complex 3-dimensional shape. This shape is what primarily determines the protein’s function.

---

3. **References**


Three adjacent DNA bases form each of 64 different codons, 61 of which code for the 20 different amino acids, while the three remaining codons indicate a stop to the coding region for the current protein. A particular amino acid may have from one to six different encodings.

Different organisms typically use similar proteins for similar purposes. A slight change in the amino acid sequence can cause anything from a slight to a profound change in the shape of the resulting protein. A slight change in the DNA sequence can cause anything from no change to a profound change in the amino acid sequence. Profound changes are almost always bad for the organism, but smaller changes may be good, bad, or neutral. Such changes (mutations) are continually occurring as a result of radiation, chemical agents, copying errors, etc.

Mutations can change individual bases, or can add or delete sections of DNA. Adding or deleting individual bases almost always produces a profound change, but adding or deleting a sequence of 3n bases may have only a slight effect since the subsequent amino acids remain unchanged.

Automated techniques have produced enormous libraries of DNA sequences identified by organism. Laboratory research has produced enormous libraries of protein sequences identified by organism and function. Today, much biological research depends on finding and evaluating approximate matches between sequences from these libraries.

2.2 Sequence Alignment

In this SSCA, we consider the polynomial time problem of pairwise sequence alignment and the potentially NP-hard problem of multiple sequence alignment. Algorithms that solve these problems are often integer-based and use a variety of algorithmic techniques and heuristics. Optimal algorithms exist and are practical for pairwise alignment. However, approximate or heuristic algorithms are required for multiple sequence alignment; there is no single, obviously best approach to the problem, and the simple algorithms are either NP-hard or approximations.

In biology, multiple sequence alignments are needed to:

- Organize data to reflect sequence homology
- Identify conserved sites/regions
- Identify variable sites/regions
- Uncover changes in gene structure
- Identify probes for similar sequences in other organisms
- Develop PCR primers
- Perform phylogenetic analysis

No one alignment algorithm is suitable for all these applications, but most commonly used alignment programs use variants of a single basic approach, the dynamic programming algorithm for optimal pairwise sequence alignment. The kernels below explore several of these variations.

2.3 Data Generation and Kernels

The first kernel performs a local pairwise alignment of two long codon sequences, finding the end-points of subsequences that are good matches according to the specified criteria. The second kernel identifies actual codon sequences located by the first kernel, working backward from the given end-points. The third kernel uses the interesting subsequences found in the second sequence by the second kernel to search the first sequence for a set of the best complete matches to each interesting
subsequence. The fourth kernel goes through each set of matches found by the third kernel and performs a global pairwise alignment at the nucleotide level for each pair. The fifth kernel performs multiple sequence alignment on each of the sets of alignments generated by the third kernel, using the simple, approximate “center star” algorithm.

2.3.1 Scalable Data Generator

This SSCA requires a scalable data generator to create genomic sequences of lengths from hundreds to potentially billions of nucleotide bases. At least four types of data are commonly used for sequence matching: DNA, RNA, codons, and amino acids.

Usually the division of DNA/RNA into codons is known, and matching at the codon level is the most informative and fastest. Matching at the nucleotide level is interesting for some applications, but is much slower. Matching at the amino acid level is important only if the corresponding DNA/RNA is unknown, and uses almost exactly the same algorithms as codon-level matching. For this SSCA we have chosen to specify DNA codon-level pairwise alignment and DNA nucleotide-level multiple sequence alignment.

2.3.2 Kernel 1: Pairwise Local Alignment of Sequences

Waterman states: “Surprising relationships have been discovered between sequences that overall have little similarity. These are dramatic cases when unexpectedly long matching segments have been located between viral and host DNA. [Smith-Waterman] is a dynamic programming algorithm to find these segments. This is probably the most useful dynamic programming algorithm for current problems in molecular biology. These alignments are called local alignments.” For this first kernel, we are given two sequences and wish to find the subsequences from these two that are most similar to each other as defined by Waterman. There may be several ‘equally similar’ subsequence pairs.

2.3.3 Kernel 2: Sequence extraction

Using the end-point pairs and similarity scores produced by kernel 1, kernel 2 locates actual subsequence pairs with those scores. If there is more than one match for a particular end-point pair, only the best one should be reported. This kernel is specified separately from kernel 1, since keeping track of the actual sequences in kernel 1 would require some extra computation and perhaps a great deal of extra space.

2.3.4 Kernel 3: Locating similar sequences

For the second of each pair of subsequences produced by kernel 2, remove any gaps and search the first full sequence for the 100 best matches to the entire compacted subsequence.

2.3.5 Kernel 4: Aligning pairs of similar sequences

The result of kernel 4 is 100 sets of 100 highly similar subsequences, taken from the first of the two original full sequences.

For each of these sets, this kernel prepares the way for the kernel 5 multiple-sequence alignment algorithm by aligning each pair of subsequences and reporting their alignment score. The scoring algorithm does global matching using a scoring function, which operates at the nucleotide level and does not include any gap-start penalty. Instead of measuring similarity directly, it measures differences between the strings, so computing optimal alignments requires minimizing the difference-score rather than maximizing a similarity-score.
2.3.6 Kernel 5: Multiple Sequence Alignment

The result of kernel 4 is 100 sets of 100 subsequences, pairwise aligned and scored for similarity at the nucleotide level. Kernel 5 then arranges each set of subsequences into a multiple alignment that approximates an alignment, which might be of interest to someone studying relationships between the subsequences within each set.

A Multiple Sequence Alignment (MSA) is defined as follows.

A multiple alignment of strings $S[1], \ldots, S[k]$ is a series of strings $S'[1], \ldots, S'[k]$ with spaces (internal gaps), such that the new sequences $S'[j]$ are all of the same length $n$, and $S'[j]$ is an extension of $S[j]$ by insertion of spaces for $1 \leq j \leq k$. The goal is to find an optimal multiple alignment.

For biological purposes an optimal multiple alignment is one which most clearly shows the relationships of the various sequences. These relationships may be evolutionary and/or structural, and may depend on additional data such as known kinship relationships or 3-dimensional protein shape correlations. Many different approaches are possible and this remains an active research area.

Sum-of-Pairs (SP) is one simple theoretical measure of multiple alignment. Given a specific multiple alignment (including spaces), the global similarity of each pair of sequences is calculated separately. The sum of these pairwise similarities is the SP measure. The minimum of SP over all possible multiple alignments is the theoretically optimal SP alignment.

Finding the optimal SP alignment is NP-hard; using dynamic programming it runs in $O(k^2n^k)$ time, that is, exponential in the number of sequences. For $n = 200$ and $k = 100$, this is prohibitive. A number of different approximate methods are known, of varying performance, quality, and complexity, such as the Star, Tree, and various Progressive alignments. Star alignment runs in polynomial time, but the result can be far from optimal. Tree alignment maps the $k$ sequences to a tree with $k$ leaves, is NP-complete, and requires a tree selection heuristic.

Progressive alignments (such as ClustalW, PileUp, or T-Coffee) are perhaps the most practical and widely used methods, and are a hierarchical extension of pairwise alignments. They compare all sequences pairwise, perform cluster analysis on the pairwise data to generate a hierarchy for alignment (guide tree), and then build the alignment step by step according to the guide tree. The multiple alignment is built by first aligning the most similar pair of sequences, then adding another sequence or another pairwise alignment.

Progressive alignments often work well for similar sequences but are problematic for distantly-related sequences. Probabilistic methods are emerging, such as HMMER, that perform Profile Hidden Markov Models introduced by Gribskov. A profile HMM can be trained from unaligned sequences, if a trusted alignment is not yet known. HMMs have a consistent theory behind gap and insertion scores, and are useful in determining protein families. Many new MSA heuristics have been published in the last few years, including techniques such as MACAW, Clustal W, DIALign, T-Coffee, and POA.

Multiple sequence alignment is a difficult problem; as described the best solutions are either very slow or very complex. For the purposes of this SSCA we choose the “center star” approximation method, as discussed in [8], a simple example of a progressive alignment. When used with an alphabet-weighted scoring scheme that satisfies the triangle inequality, this method produces a sum-of-pairs solution that is within a factor of two of optimal (and is usually much better).
3. **SSCA #2: Graph Analysis**

Graph theoretic problems are representative of fundamental computations in traditional and emerging scientific disciplines like scientific computing and computational biology, as well as applications in national security. This synthetic benchmark consists of four kernels that require irregular access to a large directed and weighted graph.

SSCA #2 is a graph theoretic problem which is representative of computations in the fields of national security, scientific computing, and computational biology. The HPC community currently relies excessively on single-parameter microbenchmarks like LINPACK, which look solely at the floating-point performance of the system, given a problem with high degrees of spatial and temporal locality. Graph theoretic problems tend to exhibit irregular memory accesses, which leads to difficulty in partitioning data to processors and in poor cache performance. The growing gap in performance between processor and memory speeds, the memory wall, makes it challenging for the application programmer to attain high performance on these codes. The onus is now on the programmer and the system architect to come up with innovative designs.

The intent of this SSCA is to develop a compact application that has multiple analysis techniques (multiple kernels) accessing a single data structure representing a weighted, directed graph. In addition to a kernel to construct the graph from the input tuple list, there are three additional computational kernels to operate on the graph. Each of the kernels requires irregular access to the graph's data structure, and it is possible that no single data layout will be optimal for all four computational kernels.

Two versions of this SSCA #2 have been specified. The earlier versions (1.0 and 1.1) used a different data generator and graph algorithm for kernel 4. Here we describe the latest version (2.0) and refer the reader to [16, 17, 18] for details on the older version 1.1.

SSCA #2 includes a scalable data generator that produces edge tuples containing the start vertex, end vertex, and weight for each directed edge. The first kernel constructs the graph in a format usable by all subsequent kernels. No subsequent modifications are permitted to benefit specific kernels. The second kernel extracts edges by weight from the graph representation and forms a list of the selected edges. The third kernel extracts a series of subgraphs formed by following paths of a specified length from a start set of initial vertices. Kernel 3’s set of initial vertices are determined by kernel 2. The fourth kernel computes a centrality metric that identifies vertices of key importance along shortest paths of the graph.

### 3.1 Data Generation

The scalable data generator constructs a list of edge tuples containing vertex identifiers, and randomly-selected positive integers are assigned as weights on the edges of the graph. Each edge is directed from the first vertex of its tuple to the second. The generated list of tuples must not exhibit any locality that can be exploited by the computational kernels. For generating the graphs, we use a synthetic graph model that matches the topologies seen in real-world applications: the Recursive MATrix (R-MAT) scale-free graph generation algorithm. For ease of discussion, the description of this R-MAT generator uses an adjacency matrix data structure; however, implementations may use any alternate approach that outputs the equivalent list of edge tuples. The R-MAT model recursively sub-divides the adjacency matrix of the graph into four equal-sized partitions and distributes edges.


within these partitions with unequal probabilities. Initially, the adjacency matrix is empty and edges are added one at a time. Each edge chooses one of the four partitions with probabilities $a$, $b$, $c$, and $d$, respectively. At each stage of the recursion, the parameters are varied slightly and renormalized. For simplicity in this SSCA, multiple edges, self-loops, and isolated vertices, may be ignored in the subsequent kernels. The algorithm also generates the data tuples with high degrees of locality. Thus, as a final step, vertex numbers must be randomly permuted, and then edge tuples randomly shuffled.

3.2 Kernel 1: Graph Generation

This kernel constructs the graph from the data generator output tuple list. The graph can be represented in any manner, but cannot be modified by subsequent kernels. The number of vertices in the graph is not provided and needs to be determined in this kernel.

3.3 Kernel 2: Classify large sets

The intent of this kernel is to examine all edge weights to determine those vertex pairs with the largest integer weight. The output of this kernel will be an edge list, $S$, that will be saved for use in the following kernel.

3.4 Kernel 3: Extracting subgraphs

Starting from vertex pairs in the set $S$, this kernel produces subgraphs that consist of the vertices and edges along all paths of length less than $\text{subGrEdgeLength}$, an input parameter. A possible algorithm for graph extraction is Breadth-First Search.

3.5 Kernel 4: Graph Analysis Algorithm

This kernel identifies the set of vertices in the graph with the highest betweenness centrality score. Betweenness Centrality is a shortest paths, enumeration-based centrality metric introduced by Freeman.\(^{20}\) This is done using a betweenness centrality algorithm that computes this metric for every vertex in the graph. Let $\sigma_s$ denote the number of shortest paths between vertices $s$ and $t$, and $\sigma_s(v)$ the number of those paths passing through $v$. Betweenness centrality of a vertex $v$ is defined as

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}.\) The output of this kernel is a betweenness centrality score for each vertex in the graph and the set of vertices with the highest betweenness centrality score.

For kernel 4, we filter out a fraction of edges using a filter described in the written specification. Because of the high computation cost of kernel 4, an exact implementation considers all vertices as starting points in the betweenness centrality metric, while an approximate implementation uses a subset of starting vertices ($V_s$).

A straightforward way of computing betweenness centrality would be as follows:

1. Compute the length and number of shortest paths between all pairs $(s, t)$.
2. For each vertex $v$, calculate the summation of all possible pair-wise dependencies

$$\sigma_{s,t}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}.\) $$

Recently, Brandes\(^{21}\) proposed a faster algorithm that computes the exact betweenness centrality score for all vertices in the graph. Brandes noted that it is possible to augment Dijkstra's single-source shortest paths (SSSP) algorithm (for weight graphs) and breadth-first search (BFS) for unweighted


graphs to compute the dependencies. Bader and Madduri give the first parallel betweenness centrality algorithm in [22].

3.5.1 Performance Metric: TEPS

In order to compare the performance of SSCA #2 Version 2.x kernel 4 implementations across a variety of architectures, programming models, and productivity languages and frameworks, as well as normalizing across both exact and approximate implementations, we adopt a new performance metric, a rate called traversed edges per second (TEPS).

4. SSCA #3: Synthetic Aperture Radar Application

Synthetic Aperture Radar (SAR) is one of the most common modes in a RADAR system and one of the most computationally stressing to implement. The goal of a SAR system is usually to create images of the ground from a moving airborne RADAR platform. The basic physics of a SAR system begins with the RADAR sending out pulses of radio waves aimed at a region on the ground that it usually perpendicular to the direction of motion of the platform (see Fig. 3). The pulses are reflected off the ground and detected by the RADAR. Typically the area of the ground that reflects a single pulse is quite large and an image made from this raw unprocessed data is very blurry (see Fig. 4). The key concept of a SAR system is that it moves between each pulse, which allows multiple looks at the same area of the ground from different viewing angles. Combining these different viewing angles together produces a much sharper image (see Fig. 4). The resulting image is as sharp as one taken from a much larger RADAR with a “synthetic” aperture the length of the distance traveled by the platform.

There are many variations on the mathematical algorithms used to transform the raw SAR data into a sharpened image. SSCA #3 focuses on the variation referred to as “spotlight” SAR. Furthermore, SSCA #3 is a simplified version of this algorithm that focuses on the most computationally intensive steps of SAR processing that are common to nearly all SAR algorithms.

The overall block diagram for this benchmark is shown in Fig. 5. At the highest level it consists of three stages:

Figure 3. Basic Geometry of SAR System.  
Figure 4. Unprocessed and processed SAR image.
Figure 5. System Mode Block Diagram. SAR System Mode consists of Stage 1 front end processing and Stage 2 back end processing. In addition, there is significant IO to the storage system.

**SDG**: Scalable Data Generator. Creates raw SAR inputs and writes them to files to be read in by Stage 1.

**Stage 1**: Front-End Sensor Processing. Reads in raw SAR inputs, turns them into SAR images, and writes them out to files.

**Stage 2**: Back-End Knowledge Formation. Reads in several SAR images, compares them and then detects and identifies the difference.

Although the details of the above processing stages vary dramatically from RADAR to RADAR the core computational details are very similar: input from a sensor, followed by processing to form an image, followed by additional processing to find objects of interest in the image.

### 4.1 Operating Modes

This particular SAR benchmark has two operating modes (Compute Only and System) that both reflect different computing challenges. The “Compute Only Mode” represents the processing performed directly from a dedicating streaming sensor (Fig. 6). In this mode, the SDG is meant to simulate a sensor data buffer that is filled with a new frame of data at regular intervals, \( T_{\text{input}} \). In addition, the SAR image created in Stage 1 is sent directly to Stage 2. In this mode, the primary architectectural challenge is providing enough computing power and network bandwidth to keep up with the input data rate.

In “System Mode” the SDG represents an archival storage system that is queried for raw SAR data (Fig. 5). Likewise, Stage 1 stores the SAR images back to this archival system and Stage 2 retrieves pairs of images from this storage system. Thus, in addition to the processing and bandwidth challenges, the performance of the storage system must also be managed. Increasingly, such storage systems are the key bottleneck in sensor processing systems. Currently, the modeling and understanding of parallel storage systems is highly dependent on details of the hardware. To support the analysis of such hardware, the SAR benchmark has an “IO Only Mode” that allows for benchmarking and profiling.
4.2 Computational Workload

The precise algorithmic details of this particular SAR processing chain are given in its written specification. In Stage 1, the data is transformed in a series of steps from a $n \times m_c$ single precision complex valued array to a $m \times n_x$ single precision real valued array. At each step, either the rows or columns can be processed in parallel. This is sometimes referred to as "fine grain" parallelism. There is also pipeline or task parallelism that exploits the fact that each step in the pipeline can be performed in parallel, with each step processing a frame of data. Finally, there is also coarse grain parallelism, which exploits the fact that entirely separate SAR images can be processed independently. This is equivalent to setting up multiple pipelines.

At each step, the processing is along either the rows or the columns, which defines how much parallelism can be exploited. In addition, when the direction of parallelism switches from rows to columns or columns to rows, a transpose (or "cornerturn") of the matrix must be performed. On a typical parallel computer a cornerturn requires every processor to talk to every other processor. These cornerturns often are natural boundaries along which to create different stages in a parallel pipeline. Thus, in Stage 1 there are four steps, which require three cornerturns. This is typical of most SAR systems.

In stage 2, pairs of images are compared to find the locations of new "targets." In the case of the SAR benchmarks, these targets are just $n_{lat} \times n_{font}$ images of rotated capital letters that have been randomly inserted into the SAR image. The Region Of Interest (ROI) around each target is then correlated with each possible letter and rotation to identify the precise letter, its rotation and location in the SAR image. The parallelism in this stage can be along the rows or columns or both, as long as enough overlapping edge data is kept on each processor to correctly do the correlations over the part of the SAR image for which it is responsible. These edge pixels are sometimes referred to as overlap, boundary, halo or guard cells. The input bandwidth is a key parameter in describing the overall performance requirements of the system. The input bandwidth (in samples/second) for each processing stage is given by

$$BW_{input}^1 = \frac{nm_c}{T_{input}}, BW_{input}^2 = \frac{n_{lat}m}{T_{input}}.$$  \hspace{1cm} (1)

A simple approach for estimating the overall required processing rate is to multiply the input bandwidth by the number of operations per sample required. Looking at Table 1, if we assume $n = n_{lat} = 8000$ and $m = m_c = 4000$ the operations (or work) done on each sample can be approximated by

$$W_{sample}^1 = 10\lg(n) + 20\lg(m_c) + 40 \approx 400, \hspace{1cm} W_{sample}^2 = (1/8)n_{lat}n_{lat}n_{font}^2 \approx 1000.$$  \hspace{1cm} (2)
Thus, the performance goal is approximately

\[ R_{\text{goal}} = \frac{W_{\text{sample}}^1 \cdot BW_{\text{input}}^1}{T_{\text{input}}} = 25 \times 10^9 / T_{\text{input}}, \quad R_{\text{goal}} = \frac{W_{\text{sample}}^2 \cdot BW_{\text{input}}^2}{T_{\text{input}}} = 16 \times 10^9 / T_{\text{input}}. \] (3)

\( T_{\text{input}} \) varies from system to, but can easily be much less than a second, which yields large compute performance goals. Satisfying these performance goals often requires a parallel computing system.

The file IO requirements in “System Mode” or “IO Only Mode” are just as challenging. In this case the goal is to read and write the files as quickly as possible. During Stage 1 a file system must read in large input files and write out large image files. Simultaneously, during Stage 2, the image files are selected at random and read in and then many very small “thumbnail” images around the targets are read out. This diversity of file sizes and the need for simultaneous read and write is very stressing often requires a parallel file system.

5. Summary of Current Implementations

Table 1 provides a list of current implementations for each of the three SSCA benchmarks. The benchmarks have been implemented in several languages, with contributions from industry, academia, supercomputing centers and national labs.

Kepner and Meuse from MIT Lincoln Labs maintain the reference executable implementations in Matlab for the three SSCAs. Bader and Madduri have developed a parallel implementation of SSCA #2 in C using the POSIX thread library for commodity symmetric multiprocessors (SMPs). They evaluate the data layout choices and algorithmic design issues for each kernel, and also present execution time and benchmark validation results.\(^{17}\) Gilbert, Reinhardt and Shah describe a StarP implementation of SSCA #2 in \([18]\). The various SSCA implementations have also been compared for productivity studies.

### Table 1. Current SSCA benchmark implementation status (* indicates a completed implementation that has not been released yet, and † indicates work in progress).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>BioInformatics (SSCA #1)</th>
<th>Graph Theory (SSCA #2)</th>
<th>Sensor and IO (SSCA #3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Written Spec</td>
<td>0.5 (GT/LL)</td>
<td>2.0 (GT/LL)</td>
<td>0.8 (LL)</td>
</tr>
<tr>
<td>C</td>
<td>0.561 (PSC)</td>
<td>2.0 (GT)</td>
<td>0.5 (BS)</td>
</tr>
<tr>
<td>C &amp; MPI</td>
<td>0.561 (PSC)</td>
<td>2.0 (GT)</td>
<td>0.5 (BS)</td>
</tr>
<tr>
<td>UPC</td>
<td>0.561* (UNM/ GT/ PSC)</td>
<td>1.0* (UNM/ GT)</td>
<td>0.8* (LL)</td>
</tr>
<tr>
<td>C &amp; Pthreads</td>
<td>0.561* (UNM/ GT)</td>
<td>2.0* (UNM/ GT)</td>
<td>1.0 (LL/ MITRE/ CS)</td>
</tr>
<tr>
<td>C++</td>
<td>2.0* (Sun)</td>
<td>0.5io (LM)</td>
<td></td>
</tr>
<tr>
<td>Fortran</td>
<td>2.0* (Sun)</td>
<td>0.5io (LM)</td>
<td></td>
</tr>
<tr>
<td>Fortran &amp; OpenMP</td>
<td>2.0* (Sun)</td>
<td>0.5io (LM)</td>
<td></td>
</tr>
<tr>
<td>Matlab</td>
<td>0.5 (LL)</td>
<td>2.0 (LL)</td>
<td>0.8 (LL)</td>
</tr>
<tr>
<td>MatlabMPI</td>
<td>1.0 (LL)</td>
<td>1.0 (LL)</td>
<td>0.8 (LL)</td>
</tr>
<tr>
<td>Matlab &amp; mexGA</td>
<td>0.5* (OSC)</td>
<td>1.0* (OSC)</td>
<td>0.8* (LL)</td>
</tr>
<tr>
<td>StarP</td>
<td>2.0* (UCSB)</td>
<td>0.5 (UCSB)</td>
<td></td>
</tr>
<tr>
<td>pMatlab</td>
<td>1.0 (LL)</td>
<td>1.0 (LL)</td>
<td></td>
</tr>
<tr>
<td>Octave</td>
<td>0.8* (OSC)</td>
<td>1.0* (OSC)</td>
<td>0.5 (OSC)</td>
</tr>
<tr>
<td>Octave &amp; mexGA</td>
<td>0.8* (OSC)</td>
<td>1.0* (OSC)</td>
<td>0.5 (OSC)</td>
</tr>
<tr>
<td>Python</td>
<td>0.561 (PSC)</td>
<td>1.0io† (GT)</td>
<td></td>
</tr>
<tr>
<td>Python &amp; MPI</td>
<td>0.561 (PSC)</td>
<td>1.0io† (GT)</td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>0.561 (PSC)</td>
<td>1.0io† (GT)</td>
<td></td>
</tr>
<tr>
<td>Chapel</td>
<td>0.5 (Cray)</td>
<td>1.0io† (Cray)</td>
<td></td>
</tr>
<tr>
<td>X10</td>
<td>0.561 (UNM/ GT/ PSC)</td>
<td>1.0io* (UNM/ GT/ IBM)</td>
<td></td>
</tr>
<tr>
<td>Fortress</td>
<td>2.0* (Sun)</td>
<td>0.5io (LM)</td>
<td></td>
</tr>
</tbody>
</table>

Acknowledgments

David A. Bader’s work was supported in part by DARPA Contract NBCH30390004; and NSF Grants CCF-06-11589, CNS-06-14915, ACI-00-93039, DIB-0420513, and ITR EF/BIO 03-31654. John R. Gilbert and/or Viral Shah’s work was partially supported by Silicon Graphics Inc. and/or by the Air Force Research Laboratories under agreement number AFRL F30602-02-1-0181 and by the Department of Energy under contract number DE-FG02-04ER25632. Jeremy Kepner and Theresa Meuse’s work is sponsored by the Defense Advanced Research Projects Administration under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.

---


SLOPE - A Compiler Approach to Performance Prediction and Performance Sensitivity Analysis for Scientific Codes

Abstract

The lack of tools that can provide programmers with adequate feedback at a level of abstraction the programmers can relate to makes the problem of performance prediction and thus of performance portability in today's or tomorrow's machines extremely difficult. This paper describes SLOPE – the Source Level Open64 Performance Evaluator. SLOPE uses an approach to the problem of performance prediction and architecture sensitivity analysis using source level program analysis and scheduling techniques. In this approach, the compiler extracts the computation's high-level data-flow-graph information by inspection of the source code. Taking into account the data access patterns of the various references in the code the tool uses a list-scheduling algorithm to derive performance bounds for the program under various architectural scenarios. The end result is a very fast prediction of what the performance could be but, more importantly, the reasoning of why the predicted performance is what it is. We have experimented with a real code that engineers and scientists use in practice. The results yield important qualitative performance sensitivity information that can be used when allocating computing resources to the computation in a judicious fashion for maximum resource efficiency and/or help guide the application of compiler transformations such as loop unrolling.

1. Introduction and Background

Modern, high-end computers present a complex execution environment that makes performance understanding and performance portability extremely difficult. Programmers go to extreme lengths to manually apply various high-level transformations, most notably loop-unrolling, in an attempt to expose more Instruction-Level-Parallelism (ILP) and thus take advantage of micro architecture features such as pipelining, super-scalar and multi-core characteristics. The lack of performance prediction and analysis tools that can provide feedback to the programmer about the performance leaves the programmer in an uncomfortable position. Without understanding why the performance is what it is, the programmer is forced to search for the best possible transformation sequences by trial and error. Furthermore, existing performance understanding tools provide feedback at a very low level of abstraction, such as cache miss rates or clocks-per-clock-cycle providing no clue as to what the bottlenecks that lead to such metric values are.

Earlier approaches to performance modeling and understanding were purely empirical. Researchers developed representative kernel codes of large-scale applications such as the NAS Parallel1 and the SPEC. 2 By observing the performance of these kernels on a given machine one could extrapolate in a qualitative fashion, the performance behavior of a real application. More recently researchers have developed models for the performance of parallel applications by examining its memory behavior. 3 4 Other work has focused on modeling the behavior of an application by first accurately characterizing the running time of the sequential portions of the application using analytical modeling based on intimate knowledge of the applications mathematics and empirical observations to extract the corresponding parameter values. 5 On the other end of the spectrum, cycle-level simulators for architecture performance understanding at a very low level are simply too slow for realistic workloads. As a result, the simulations tend to focus on a minute subset of the instruction stream or use sampling techniques, and are thus limited to very focused architectural analyses.

This article describes an alternative approach to the problem of performance prediction and architecture sensitivity analysis using source level program analysis and scheduling techniques. In

---

2 SPEC - http://www.spec.org/
this approach, the compiler first isolates the basic blocks of the input source program and extracts the corresponding high-level data-flow-graph (DFG) information. It then uses the high-level information about the data access patterns of array references to determine the expected latency of memory operations. Once the DFG of each basic block, most notably the ones in the body of nested loops is extracted, the compiler tool uses a list-scheduling algorithm to determine the execution time of the computation. This scheduling makes use of the DFG as well as the functional resources available in the architecture like the number of load/store or functional units and for specific operation latency values.

While this approach has the advantage of being less sensitive to the specific details of an existing machine and not taking into account, with accuracy, the effects of a given compiler, it offers other benefits that instruction-level instrumentation-based performance analysis tools cannot offer. First, it is much closer to the source code and thus can provide feedback to the programmer about which operations (not necessarily instructions) can lead to performance bottlenecks. For example, if the schedule reveals that an indirect array access is accessed randomly, it can determine that this load operation will exhibit a high memory latency and thus stall the pipelining of a functional unit. Under these scenarios the compiler can notify the programmer of the operations that are very likely to cause severe performance degradation. Second, and because it operates at a much higher level we do not require any sort of lengthy low-level instrumentation that requires the code to be executed to extract (sampled) traces. Finally, we are free to emulate future architecture features, such as dedicated custom functional units (e.g., gather-scatter unit), or even emulate some operations in memory by assigning specific costs to specific subsets of the DFG in a computation.

We have experimented with this approach using UMT2K, a synthetic kernel modeled after a real physics photon transport code. Using this computational kernel, our tool determines qualitatively that in the absence of loop unrolling no more than two functional arithmetic units are needed to attain a level of performance that is consistent with the critical path of the computation. When the core is unrolled by a factor of four, no more than four functional arithmetic units are needed. In the context of a multi-core architecture, this information would allow a compiler to schedule and adapt its run-time execution strategy to unroll just the required amount depending on the available units.

The rest of this article is organized as follows. In the next section we describe in more detail the technical approach of our tool and how it allows performance predictions, and we perform architectural sensitivity analysis. Section 3 presents the experimental results for our case study application – the UMT2K kernel code. We present concluding remarks in section 4.

2. Technical Approach

We now describe the details of our technical approach to the problem of performance prediction and sensitivity analysis using high-level source code information using static compiler data and control-dependence analysis techniques.

2.1. Basic Analyses: Data-Flow Graph (CFG) Critical Path (CP) Analysis

This analysis extracts the basic blocks at the source code level by inspection of the compiler intermediate representations in Whirl. Because the front-end of the compiler does perform some deconstruction of high-level constructs, most notably while-loops, it is not always possible to map the intermediate representation constructs back to source code constructs. Despite some of these shortcomings, the front-end does keep track of line number information that allows the tool to provide reasonably accurate feedback to the programmer.
For each basic block, the compiler extracts a data-flow graph accurately keeping track of the dependencies (true-, anti-, input- and output-dependences) via scalar variables and conservatively by considering that any reference to an array variable may induce a dependency. In some cases, the compiler can use data dependence analysis techniques (see e.g., [8]) to disambiguate the references to arrays and thus eliminate false dependences in the DFG. In the current implementation, we make the optimistic assumption that arrays with distinct symbolic names are unaliased. While this assumption is clearly not realistic in the general case, it holds for the kernel code in our controlled experimental results.

Finally, we identify the loops of the code across the various basic blocks of a procedure to uncover basic and derived induction variables. This information is vital in determining array access stride information as explained below.

2.2. Data Access Pattern Analysis

In this analysis, the compiler extracts the affine relations between scalar variables in the array indexing functions whenever possible taking into account the basic and derived induction variables. For example, knowing the fact that the scalar variables $i$ and $j$ are both loop induction variables, the array reference $a[i][j+1]$ will have as affine coefficients the first and second dimension, respectively the values $(1,0,0)$ and $(0,1,1)$, where the last element in each tuple corresponds to the constants 0 and 1 in the expressions $i+0$ and $j+1$. Using this access information and the layout of the array (i.e., either column-wise and row-wise), the analysis determines the stride information for each access and thus makes judgments about the latency of the corresponding memory operations. Regular memory accesses are very likely to hit the cache or reside in registers as a result of an aggressive pre-fetching algorithm whereas an irregular or random memory reference is very likely to miss the cache. The construction of the DFG uses this knowledge to decorate the latency of the individual array accesses as either regular or irregular thus taking into account, to some extent, part of the memory hierarchy effects.

2.3. Scheduling Analysis

Using the extracted DFG, we then develop our own operation scheduler for determining the latency of the execution of each basic block. In this scheduler we can program the latencies of the individual operation as well as if they are executed in a pipelined fashion or not. Our scheduler also allows us to specify the number of functional units for either each individual type of operations or for a generic functional unit. For example, we can segregate the arithmetic and floating-point operations in a single functional unit or allow all of them to be executed in a generic functional unit with integer and floating-point operations. We can also specify multiple load and store units thus modeling the available bandwidth of the target architecture. Finally, we assume the scheduler is an on-line as-soon-as-possible scheduling algorithm with zero-time overhead in scheduling of the various operations in the various functional units.

For simplicity, in this first implementation, we assume the target architecture has enough registers available to support the manipulation of the various arithmetic and addressing operations. This assumption, while not unrealistic for basic blocks with a small number of statements, is clearly unrealistic for larger basic blocks that result from the application of program transformations such as loop unrolling. As such, the performance expectations derived using these assumptions should be viewed as an upper bound. For the same reason, we have folded the information about the data access pattern of array accesses in the scheduling by assuming that accesses that exhibit a very regular, strided data access pattern will have a low memory latency and those that are very irregular in nature will exhibit a high latency. This is because the first set is likely to hit the cache while the second is not.

---


Our scheduler, though simple, allows us to anticipate the completion time of the operations corresponding to a given basic block, along with various efficiency metrics, such as the number of clock cycles a given computation was stalled by while awaiting an available functional unit or awaiting a data dependency of the code to be satisfied. In our reporting, we use generic and architecture independent performance metrics such as the number of operations per clock cycle or number of floating point operations per clock cycle, rather then focusing on FLOPs or CPIs.

3. A Case Study

We now present preliminary experimental results of the performance expectation and sensitivity analysis for a synthetic code, the UMT2K, modeled after a real engineering application, a 3D, deterministic, multigroup, photon transport code for unstructured meshes.\(^6\) We first describe the methodology followed in these experiments and then present and discuss our findings using our analysis approach.

3.1 Methodology

We have built the basic analysis described above in section 2 using the Open64 compilation infrastructure.\(^10\) Our implementation takes an input source program file and focuses on the computationally intensive basic blocks. We used our analysis to extract DFG and generate performance expectation metrics for various combinations of architectural elements. We also applied manual unrolling to the significant loops in the kernel code as a way to compare the expected performance for the various code variants given the potential increase in instruction-level parallelism.

3.2. The Kernel Code

The computationally intensive section of the UMT2K code is located in what is designated the angular-loop located in the \texttt{snswp3D} subroutine. This loop contains a long basic block spanning about 300 lines of C code. This basic block, executed at each iteration of the loop, is the “core” of the computation, and has many high-latency floating point operations, such as divides (4) and multiplies (41) as depicted in Table 1.

<table>
<thead>
<tr>
<th>Total Operations</th>
<th>FP Operations</th>
<th>Integer Operations</th>
<th>Load/Store Operations</th>
<th>FP Multiplies</th>
<th>FP Divides</th>
<th>Integer Multiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>272</td>
<td>93</td>
<td>95</td>
<td>84</td>
<td>41</td>
<td>4</td>
<td>22</td>
</tr>
</tbody>
</table>

In the next section we review some of the experimental results for this basic block in an unmodified form, as well as for manually unrolled versions to explore the performance impact of data dependences and number of arithmetic and load/store units on the projected performance. The unrolled versions will expose many opportunities to explore pipelined and non-pipelined execution of the various functional units and individual operations.

Table 2 below depicts the latencies of the individual operations used in our approach. These latencies are notional rather then being representative of a real system.

<table>
<thead>
<tr>
<th>Load (cache miss)</th>
<th>Load Address</th>
<th>32-bit int. Multiply</th>
<th>32-bit FP Multiply</th>
<th>32-bit FP Divide</th>
<th>Array Address Calculation (Non-affine)</th>
<th>Array Address Calculation (Affine)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>2</td>
<td>1</td>
<td>12</td>
<td>50</td>
<td>60</td>
<td>33</td>
</tr>
</tbody>
</table>
3.3. Experiments

These experiments focus on three major aspects of performance analysis; varying the number of functional units, varying which operations are pipelined, and varying the latency of the load operations. The main goal of these experiments is to understand which aspect of the computation is limiting the performance of its execution, i.e., if the computation is memory or performance bound and how many units should be allocated to its execution in the most profitable fashion.

Figure 1 (left) depicts the impact of additional load and store units for all operations to execute in a pipelined fashion with the exception of the division, square root and reciprocal operations. For one Load/Store unit and one arithmetic unit the schedule of the unrolled version of the main basic block takes 955 clock cycles. As the graph indicates, there are clearly diminishing returns after two arithmetic units. The graph also shows that for this specific set of parameters, varying the number of load/store units does not alter performance significantly. This clearly indicates that computation is not data starved.

Figure 1 (right) depicts the same analysis but this time for the version of the code where the angular loop is unrolled by a factor of four. In this case the length of the schedule is longer as there are more arithmetic and load and store operations and the benefit of adding more functional units tails off for four units rather than for two units as in the previous unrolled case.

![Figure 1. Schedule length for various arithmetic and load/store units configurations for the original version of the code (left) and by unrolling by a factor of four (right).](image)

In either of the cases above, the computation is clearly bounded by the number of arithmetic units. For increasing numbers of load/store units the execution schedule varies little revealing that the computation is compute-bounded.

We have also observed the efficiency of various architecture variations by computing the number of executed operations per clock cycle for each scenario. Table 3 below presents the numerical results for the two unrolling scenarios and one Load/Store unit with one Arithmetic unit versus five of each units.

<table>
<thead>
<tr>
<th>Number of Load/Store units and Arithmetic units</th>
<th>Unrolling Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1.30</td>
</tr>
<tr>
<td>5</td>
<td>2.73</td>
</tr>
<tr>
<td>5</td>
<td>1.015</td>
</tr>
<tr>
<td>5</td>
<td>1.25</td>
</tr>
</tbody>
</table>

Table 3. Executed Operations per Clock.
The results in table 3 clearly show a substantial increase in the number of executed arithmetic operations for the case of a single arithmetic unit and a single load store unit versus the case of five units of each. For the configuration with five units, the efficiency metric drops approximately 25% for the original code and 50% for the code unrolled by a factor of four, thus revealing the later configuration not to be cost effective.

In order to observe the effect of the latency of the memory hierarchy we have analyzed the impact of increasing the latency of the load operations from 20 to 50 clock cycles. The results, depicted in figure 3 reveal that when the code is unrolled by a factor of four the performance is insensitive to the memory latency. This confirms that this code is not memory-bound. The results reveal that there is a 30% increase from one to two arithmetic units and a steady increase of 17% for two to three and three to four arithmetic units. This performance improvement is in stark contrast to the negligible gains obtained by increasing the number of load and store units.

It is also interesting to note that the schedule time for a system with arithmetic units that pipeline all opcodes with a single arithmetic unit is roughly equivalent to the standard code (without any unrolling) with five arithmetic units when both codes are unrolled four times. Also, when the code is not unrolled, increasing the number of arithmetic units has no effect when all opcodes are pipelined.

3.4. Discussion

The architectural model developed in the current SLOPE implementation is rather simple (but not simplistic) in several respects. First, it assumes a zero overhead instruction scheduling. This is clearly not the case although pipelining execution techniques can emulate this aspect. Second, it does not take into account register pressure in the execution. Lastly, it does not yet take into account advanced execution techniques such as software pipelining and multi-threading. Ignoring these techniques and compiler implementation details clearly leads to quantitative results that differ, maybe substantially, from current high-end machines. Last but not least, this approach needs to be validated against a real architecture.

Nevertheless, this approach allows the development of quantitative architectural performance trends and hence allows architecture designers to make informed decisions about how to most efficiently allocate transistors. In the above case, a determination could be made between the complexity and power consumption (for example) of having an arithmetic that pipelines all operations (including divides) and simply replicating standard arithmetic. This information also allows

---

11 Pipelining allows the functional unit allocated to an operation to be available again the cycle after it is obtained.
developers to have an idea of what the performance trend increases will be on a proposed “future” machine for a given code.

4. Conclusion

We have described SLOPE, a system for performance prediction and architecture sensitivity analysis using source level program analysis and scheduling techniques. SLOPE provides a very fast qualitative analysis of the performance of a given kernel code. We have experimented with a real scientific code that engineers and scientists use in practice. The results yield important qualitative performance sensitivity information that can be used when allocating computing resources to the computation in a judicious fashion for maximum resource efficiency and/or help guide the application of compiler transformations such as loop unrolling.
Metrics for Ranking the Performance of Supercomputers

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 latency 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 (e.g., the NAS parallel benchmark suites, the SPEC benchmark, the HINT benchmark, the HPC Challenge benchmark, STREAM, and the ratio of flops to memory bandwidth).

A particularly well-known parallel benchmark is Linpack, which has been used since 1993 to rank supercomputers for inclusion on the Top 500 list (more recently the IDC Balanced Rating 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.

Figure 1. This graph shows the measured runtimes of eight applications on seven 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 eight different High Performance Computing (HPC) codes on seven 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).

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 one 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 (e.g., [11, 12, 13]) 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.14

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.

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; 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 zero and is no larger than $n(n-1)/2$.

In practice, to allow for variance in the measured runtimes, we count only thresholded inversions. Given a threshold $\alpha$ 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 $\alpha$ allows us to handle variations in timing measurements that can be caused for reasons including those discussed in [16].

Furthermore, to allow for variance in the benchmark timings that give the machine metrics, we use a second threshold parameter $\beta$, 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 $\beta$. In general $\beta$ should be chosen to be smaller than $\alpha$ 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.

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.

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:

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 $\beta = .001$. Although this choice of $\beta$ 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, an 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.

---

17 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.
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 eight of the applications in our test suite. Later, in Section 3.2, we discuss application-specific rankings.

### 3.1 Ranking using simple metric-products

We first try using only information gathered on the applications' memory behavior (gathered using the Metasim Tracer\(^{18}\)), 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.\(^{18}\) 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:

\[
r_i = \frac{8m_1}{bw_{mem1}(i)} + \frac{8m_r}{bw_{memr}(i)} + \frac{f_{\text{flops}}(i)}{\text{flops}(i)}, \tag{1}
\]
Metrics for Ranking the Performance of Supercomputers

Notice that we must choose what to use for bw mem, and bw memr. 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 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 $m_i$, $m_r$, 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 $m_i$ and $m_r$ counts for one processor count per application.

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.

<table>
<thead>
<tr>
<th>Metric</th>
<th>$ll(1,r)$</th>
<th>mm(1,r)</th>
<th>mm(1), $ll(r)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>owis</td>
<td>12</td>
<td>21</td>
<td><strong>9</strong></td>
</tr>
<tr>
<td>ch7</td>
<td>14</td>
<td>80</td>
<td><strong>9</strong></td>
</tr>
<tr>
<td>games3</td>
<td>16</td>
<td>77</td>
<td>26</td>
</tr>
<tr>
<td>hycom</td>
<td>2</td>
<td>44</td>
<td>2</td>
</tr>
<tr>
<td>lammps</td>
<td>107</td>
<td>148</td>
<td><strong>81</strong></td>
</tr>
<tr>
<td>oocore</td>
<td><strong>31</strong></td>
<td>100</td>
<td>44</td>
</tr>
<tr>
<td>overflow2</td>
<td><strong>34</strong></td>
<td>78</td>
<td>34</td>
</tr>
<tr>
<td>wrf</td>
<td>63</td>
<td>158</td>
<td>44</td>
</tr>
<tr>
<td>overall sum</td>
<td><strong>279</strong></td>
<td>706</td>
<td><strong>249</strong></td>
</tr>
</tbody>
</table>

Table 2 shows the results of these experiments. Each column shows the number of thresholded inversions for each of the eight applications using the specified choice of strided and random access bandwidths. In each column the results use the application whose $m_i$, $m_r$, 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 $ll(r)$ 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 [10,19]. 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 [10], where the authors evaluate nine 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 [10]. Note that their dataset is different from the one used in the previous sections of this paper: we use eight applications to their four, 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.

<table>
<thead>
<tr>
<th>Methodology</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td># thresh. inversions</td>
<td>165</td>
<td>86</td>
<td>115</td>
<td>165</td>
<td>76</td>
<td>53</td>
<td>55</td>
<td>44</td>
<td>44</td>
</tr>
</tbody>
</table>

Table 3. Sum of the number of thresholded inversions for all applications and numbers of processors, for each of the nine performance prediction strategies described in [10]. For the measured runtimes we use $a = .01$; for the predicted runtimes we use $β = .001$. 

---

We observe that the first three cases in [10] should produce rankings that are equivalent to using only 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 [10], 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 \( bw_{\text{mem}} \) and \( bw_{\text{mem}} \), 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 [10] 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 [10] 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.

---

### Appendix

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Processor counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>awns</td>
<td>CFD calculations on unstructured grids</td>
<td>32, 64, 96, 128, 192, 256, 384</td>
</tr>
<tr>
<td>ctrl</td>
<td>effects of strong shock waves</td>
<td>16, 32, 64, 96</td>
</tr>
<tr>
<td>games</td>
<td>general ab-initio quantum chemistry</td>
<td>32, 48, 64, 96, 128</td>
</tr>
<tr>
<td>hycom</td>
<td>primitive equation ocean general circulation model</td>
<td>24, 47, 59, 80, 96, 111, 124</td>
</tr>
<tr>
<td>lumps</td>
<td>classical molecular dynamics simulation</td>
<td>16, 32, 48, 64, 128</td>
</tr>
<tr>
<td>osecure</td>
<td>out-of-core solves</td>
<td>16, 32, 48, 64</td>
</tr>
<tr>
<td>overlap2</td>
<td>CFD calculations on overlapping, multi-resolution grids</td>
<td>16, 32, 48, 64</td>
</tr>
<tr>
<td>wrf</td>
<td>weather research and forecast</td>
<td>16, 32, 48, 64, 96, 128, 192, 256, 384</td>
</tr>
</tbody>
</table>

Table 5: Systems used in this study.

<table>
<thead>
<tr>
<th>HPC lab location</th>
<th>Processor</th>
<th>Interconnect</th>
<th># of compute processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARL</td>
<td>SGI-0380-0.4GHz</td>
<td>NUMA CC</td>
<td>512</td>
</tr>
<tr>
<td>ARL</td>
<td>LNX-Xeon-3.0GHz</td>
<td>Myrinet</td>
<td>2048</td>
</tr>
<tr>
<td>ARSC</td>
<td>IBM-690-1.0GHz</td>
<td>Federation</td>
<td>783</td>
</tr>
<tr>
<td>ASC</td>
<td>IBM-690-0.7GHz</td>
<td>LNUM ACC</td>
<td>2032</td>
</tr>
<tr>
<td>ASC</td>
<td>HP-SC-15-1.0GHz</td>
<td>Quadrics</td>
<td>768</td>
</tr>
<tr>
<td>ERDC</td>
<td>SGI-0380-0.7GHz</td>
<td>NUMA CC</td>
<td>1008</td>
</tr>
<tr>
<td>ERDC</td>
<td>HP-SC-10-0.83GHz</td>
<td>Quadrics</td>
<td>488</td>
</tr>
<tr>
<td>ERDC</td>
<td>HP-SC-15-1.0GHz</td>
<td>Quadrics</td>
<td>488</td>
</tr>
<tr>
<td>MHPCC</td>
<td>IBM-690-1.4GHz</td>
<td>Colony</td>
<td>320</td>
</tr>
<tr>
<td>MHPCC</td>
<td>IBM-P3-0.5GHz</td>
<td>Colony</td>
<td>736</td>
</tr>
<tr>
<td>NAVO</td>
<td>IBM-655-1.1GHz</td>
<td>Federation</td>
<td>2832</td>
</tr>
<tr>
<td>NAVO</td>
<td>IBM-690-1.3GHz</td>
<td>Colony</td>
<td>1328</td>
</tr>
<tr>
<td>NAVO</td>
<td>IBM-P3-0.375GHz</td>
<td>Colony</td>
<td>736</td>
</tr>
<tr>
<td>SDSC</td>
<td>IBM-IA61-1.5GHz</td>
<td>Myrinet</td>
<td>512</td>
</tr>
</tbody>
</table>

Table 6: Probes run as part of DoD benchmarking.

<table>
<thead>
<tr>
<th>Probe name</th>
<th>DoD TI06 Benchmark suite</th>
<th>Machine property measured</th>
</tr>
</thead>
<tbody>
<tr>
<td>flops</td>
<td>CPUBENCH</td>
<td>peak rate for issuing floating-point operations</td>
</tr>
<tr>
<td>L1 bw(1)</td>
<td>MEMBENCH</td>
<td>rate for loading strided data from L1 cache</td>
</tr>
<tr>
<td>L2 bw(1)</td>
<td>MEMBENCH</td>
<td>rate for loading strided data from L2 cache</td>
</tr>
<tr>
<td>L2 bw(1)</td>
<td>MEMBENCH</td>
<td>rate for loading strided data from L2 cache</td>
</tr>
<tr>
<td>L3 bw(1)</td>
<td>MEMBENCH</td>
<td>rate for loading strided data from L3 cache</td>
</tr>
<tr>
<td>MM bw(1)</td>
<td>MEMBENCH</td>
<td>rate for loading strided data from main memory</td>
</tr>
<tr>
<td>MM bw(1)</td>
<td>MEMBENCH</td>
<td>rate for loading strided data from main memory</td>
</tr>
<tr>
<td>NW bw</td>
<td>NETBENCH</td>
<td>rate for sending data point-to-point</td>
</tr>
<tr>
<td>NW latency</td>
<td>NETBENCH</td>
<td>startup latency for sending data point-to-point</td>
</tr>
</tbody>
</table>

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 NRI 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 CyberInfrastucture 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."
Large-Scale Computational Scientific and Engineering Project Development and Production Workflows

Overview

Computational science and engineering (CSE) is becoming an important tool for scientific research and development and for engineering design. It is being used to make new scientific discoveries and predictions, to design experiments and analyze the results, to predict operational conditions, and to develop, analyze and assess engineering designs. Each application generally requires a different type of application program, but there are important common elements. As computer power continues to grow exponentially, the potential for CSE to address many of the most crucial problems of society increases as well. The peak power of the next generation of computers will be in the range of 1015 floating point operations per second achieved with hundreds of thousands of processors. It is becoming possible to run applications that include accurate treatments of all of the scientific effects that are known to be important for a given application. However, as the complexity of computers and application programs increases, the CSE community is finding it difficult to develop the highly complex applications that can exploit the advances in computing power. We are facing the possibility that we will have the computers but we may not be able to quickly and more easily develop large-scale applications that can exploit the power of those computers.

In support of the Defense Advanced Research Projects Agency’s High Productivity Computing Systems Program (DARPA HPCS) to reduce these software difficulties, we have conducted case studies of many large scale CSE projects and identified the key steps involved in developing and using CSE tools. This information is helping the computer architects for the DARPA HPCS computers understand the processes involved in developing and using large-scale CSE projects, and identify the associated bottlenecks and challenges. This has facilitated their efforts to develop and implement productivity improvements in computer architectures and in the software support infrastructure. This information can also used as a blueprint for new projects.

While CSE workflows share many features with traditional Information Technology (IT) software project workflows, there are important differences. IT projects generally begin with the specification of a detailed set of requirements. The requirements are used to plan the project. In contrast, it is generally impossible to define a precise set of requirements and develop a detailed software design and workplan for the development and application of large-scale CSE projects. This is not because CSE projects have no requirements. Indeed, the requirements for CSE projects, the laws of nature, are very definite and are not flexible. The challenge computational scientists and engineers face is to develop and apply new computational tools that are instantiations of these laws. CSE applications generally address new phenomena. Because they address new issues, they often exhibit new and unexpected behavior. Successful projects identify the properties of nature that are most important for the phenomena being studied and develop and implement computational methods that accurately simulate those properties. The initial set of candidate algorithms and effects usually turns out to be inadequate and new ones have to be developed and implemented. Successful code development is thus a "requirements discovery" process. For these reasons, the development and use of CSE projects is a complex and highly iterative process. While it is definitely not the waterfall model, it does share some of the features of more modern software engineering workflows such as the "spiral" development model.


A typical CSE project has the following steps (Figure 1):

1. **Formulate Questions and Issues**
   - Define the high level requirements and goals (including the phenomenon to be simulated or analyzed); the stakeholders (the application users and customers, the sponsors, the developers, the validation community, and the computer support); the general approach, the important physical effects necessary for the simulation of a particular phenomenon, and the criteria for success.

2. **Develop Computational and Project Approach**
   - Define the detailed goals and requirements; seek input from customers; select numerical algorithms and programming model; design the project including the code architecture; identify the modules and specify interfaces for the individual modules; recruit the team; get the resources; and identify the expected computing environment.

3. **Develop the Program**
   - Write and debug the program, including the individual modules, input and output packages, and code controllers.

4. **Perform Verification & Validation**
   - Define verification tests and methodology; set up regression test suites and run them; define unit tests and execute them; define useful validation experiments; design and conduct validation experiments; and compare the validation data with code results.

5. **Make production runs**
   - Setup problems, schedule runs, execute runs, and store the results.

6. **Analyze computational results**
   - Begin analysis during the production run to optimize it; store, analyze and assess the results at the conclusion of the production run; and document the results, analysis and conclusions. Then develop hypotheses and test them with further runs.

7. **Make decisions**
   - Make decisions based on the analysis of the results; document and justify the decisions; develop plans to reduce uncertainties and resolve open questions; and identify further questions and issues.

These large tasks strongly overlap each other. There is usually a lot of iteration among the steps and within each step. Quite commonly, it turns out that some of the candidate algorithms are not sufficiently accurate, robust, stable or efficient, and new candidate algorithms must be identified, implemented and tested. Similarly, comparison with experimental data (validation) usually shows that the initial set of physical phenomena does not include all of the effects necessary to accurately simulate the phenomenon of interest. The project then needs to identify the effects that were not included in the model but are necessary for accurate simulations, incorporate them in the application, and assess whether the new candidate effects are adequate for simulating the target phenomenon. Often this series of steps will be iterated many times.
Another key aspect of CSE project workflows is the project life cycle (Figure 2). Large-scale CSE projects can have a life cycle of 30 to 40 years or more, far longer than most Information Technology projects. The NASTRAN engineering analysis code was originally developed in the 1960s and is still heavily used today.\(^3\) In contrast, the time between generations of computers is much shorter, often no more than two to four years. A typical major CSE project has an initial design and development phase (including verification and initial validation), that often lasts five or more years (Fig. 2). That is followed by a second phase in which the initial release is further validated, improved and then further developed based on experience by the users running real problems. A production phase follows during which the code is used to solve real problems. If the project is successful, the production phase is often the most active development phase. Once the code enters heavy use, many deficiencies and defects become apparent and need to be fixed, and the users generate new requirements for expanded capability. The new requirements may be due to new demands by the sponsor or user community, to the desire to incorporate new algorithmic improvements, or to the need to port to different computer platforms. Even if no major changes are made during the production phase, substantial code maintenance is usually required for porting the code to different platforms, responding to changes in the computational infrastructure, and fixing problems due to non-optimal initial design choices. The rule of thumb among many major CSE projects is that about one FTE of software maintenance support is needed for each four FTEs of users.

Historically, many, if not most, CSE codes have included only a limited number of effects and were developed by teams of one to five or so professionals. The few CSE codes that were multi-effect generally developed one module for a new effect and added it to the existing code (Figure 3). Once the new module had been successfully integrated into the major application, the developers then started development of the next module. This approach had many advantages. It allowed the developers and users to extensively use and test the basic capability of the code while there was time to make changes in the choices of solution algorithms, data structures, mesh and grid topologies and structures, user interfaces, etc. The users were able to verify and validate the basic capability of the code. Then they were able to test each new capability as it was added. The developers got rapid feedback on every new feature and capability. The developers of new modules had a good understanding of the existing code because many of them had written it. It was therefore possible to make optimum trade-offs in the development of good interfaces between the existing code and new modules. On the other hand, serial development takes a long time. If a code has four major modules that take five years to develop, the full code won't be ready for 20 years. Unfortunately, by then the whole code may be obsolete. Certainly the code will have been ported to new platforms many times.
To overcome these limitations, multi-effect codes are now generally developed in parallel (Figure 4). If a code is designed to include four effects, and the modules for each effect take five years to develop, then the development team will consist of 20 members plus those needed to support the code infrastructure. If all goes well, the complete code with treatments of all four effects will be ready five or six years after the start of the project instead of 20 years.

Because the development teams are much larger, and the individual team members often don't have working experience with the modules and codes being developed by the other module sub-teams, the software engineering challenges are much greater. Parallel development also increases the relative risks. If the development of a module fails, a new effort has to be started. If one out of four module initial development efforts fail, then the impact on total development time is to double it compared to only a twenty-five percent increase with serial development.

Software Development Tool Categories

The code development and production phases of software projects involve many different types of activities. In general each requires different tools and methods. We were able to define four broad categories of tools (and methods) that are typically required in different phases of the scientific
software lifecycle. The potential suppliers for these tools and methods include platform vendors, commercial third party vendors, academic institutions, and open source developers.

1. **Code Development computing environment:**
   This includes the computer operating system (e.g., Linux, AIX, True64, etc.), text editors, interactive development environments (e.g., Eclipse), languages and compilers (Fortran, C, C++, JAVA, etc.) including language enhancements parallel computers (Co-array Fortran, UPC, HPF, OpenMP, Pthreads, etc.), parallel communication libraries (e.g., MPI), symbolic mathematics and engineering packages with a high level of abstraction (Mathematica, Maple, Matlab, etc.), interpretative and compiled scripting languages (PERL, Python, etc.), debuggers (e.g., Totalview), syntax checkers, static and dynamic analysis tools, parallel file systems, linkers and build tools (e.g., MAKE), job schedulers (e.g., LFS), job monitoring tools, performance analysis tools (e.g. Vampir, Tau, Open Speedshop, etc.). This software can either be supplied by the platform vendor or by third parties. For instance, AIX is supplied by IBM. Etnus markets the debugger Totalview.

2. **Production Run computing environment:**
   This includes running the code and collecting and analyzing the results. Many of the tools for the code development environment are required (operating system, job scheduler, etc.). In addition there are specific tasks that involve problem setup (e.g., mesh generation, decomposing the problem domain for parallel runs, etc.), checkpoint restart capability, recovery from faults and component failures (fault tolerance), monitoring the progress of a run, storing the results of the run, and analyzing the results (visualization, data analysis, etc.). Some of this software is supplied by the platform vendor and some by third parties. CEI, for instance, markets Ensight, a massively parallel 3D Visualization tool. Research Systems markets IDL, a data analysis tool. A key task is verification and validation which requires tools for comparing code results with test problem results, experimental data and results from other codes.

3. **Software engineering and software project management tools:**
   These tasks involve organizing, managing and monitoring the code development process. Tools that would help with this task include configuration management tools (e.g., CVS, Perforce, Razor, etc.), code design and code architecture (e.g., UML) although there are few examples of code design tools being used for HPC applications, documentation tools (word processors, web page design and development tools, etc.), software quality assurance tools, project design tools, and project management tools (Microsoft Project, Primavera, etc.). Most of these are supplied by commercial third party vendors. Development of code development collaboration tools for multi-institutional code development teams will also be important in the future (probably a third party task).

4. **Computational algorithms and libraries:**
   These tasks involve development and support of computational algorithms and libraries that are incorporated into working code. These include computational mathematics libraries (e.g., PETSc, NAG, HYPRE, and Trilinos.), physical data libraries, low-level memory management libraries (e.g., MPI), etc. These are supplied by computer platform vendors, commercial vendors, academic and national laboratory institutions, and the open source community.

   For tasks that call for selection of an approach or method, the expectation is that the vendor will provide options and some guidance (documentation and consultation) on which approach or method is most appropriate for a set of specific requirements. In general a formal tool for making the selection is not required.

In the discussion that follows, the categories of software tools defined above will be listed under each major workflow stage.
Development and Production Workflows

The development and production workflow for a typical CSE project is highly iterative and exploratory. Each stage of software development involves many steps that are closely linked. If the steps can be completed successfully, the work proceeds to the next step (Figure 5). For most realistic cases, multiple issues arise at each step and resolution of the issues often requires iteration with prior steps. The detailed architecture of the code evolves as the code is developed and issues are discovered and resolved.

The degree to which each step becomes a formal process depends on the scale of the project. A small project involving only one or two people need not devote a lot of time to each process. Nonetheless, even small projects will go through almost all of the steps defined below. It is thus worthwhile for almost all projects to go through the checklist to ensure that they don't miss a step, which would be simple to address early in the project, but difficult much later in the project.

Throughout this paper we define stakeholders as everyone who has a stake in the project including the sponsors, the users and customers, the project team, the project and institutional management, the groups who provide the computer and software infrastructure, and sub-contractors. Sub-contractors include everyone who develops and supplies crucial modules and software components for the project, but who are not part of the project team and not under the direct control of the project management.

I. Formulate Questions, Issues and General Approach

The time scale for this phase is generally three months to a year. The first step involves assessing the state of the science and engineering, its potential for solving the problem of interest, and the development of a roadmap and high-level plan for the project. A key element is the assessment of prior and existing methods for solving this problem with an analysis of their strengths and weaknesses. Prior and existing computational tools provide highly useful prototypes for the proposed project; they embody the methods and algorithms that have been successful in the past and demonstrate the strengths and weaknesses of those methods. These help potential sponsors, users, stakeholders and domain experts achieve a common view of the problem. For the science community, this phase would result in a proposal for submission to a funding agency (e.g., NSF, DOE SC, etc.). This phase also would provide a document that will be essential for developing a customer base, getting additional support, and communicating the project goals, purpose, and plan to the stakeholders, including prospective project team members.

This stage involves knowledge of all of the development tasks cited above, but emphasizes detailed knowledge of software engineering and software project management, and computational algorithms and libraries. However, a note of caution is appropriate. Extensive use of software tools for project management is premature and can be a serious distraction. Similarly, extensive assessment of algorithms and methods is also premature. A high-level plan and general code architecture is needed before detailed work begins.
II. Develop the Computational and Project Management and Team Approaches for the Code project

General Software infrastructure tool requirements: particularly configuration management (3), project management(3), documentation (3), computational mathematics (4), ...

The time scale for this phase is three months to a year. This is the major planning phase. While some small scale projects may not need much planning, many code development projects ultimately reach cost levels that exceed $100M over the life of the project. In every other type of technical work, sponsoring institutions require detailed plans for how the work will be accomplished, goals met, and progress monitored. They have found that plans and monitoring of progress are essential for minimizing project risks and maximizing project success. CSE is no exception. Developing plans for CSE projects is challenging. The plans must incorporate sufficiently detailed information on the project tasks, schedule and estimated costs for the project to be monitored and judged by the project sponsors and stakeholders. At the same time, the plans must preserve sufficient flexibility and agility that the project can successfully “discover, research, develop and invent” the domain science and solution algorithms need by the project. This is also the time to do a lot of prototyping and testing of candidate modules and algorithms and to explore the issues of integrating modules for different effects, particularly modules for effects that have time and distance scales that differ by many orders of magnitude.
III. Develop the code

General Software infrastructure tool requirements and best practices include: ongoing [documentation of scientific model, equations, design, code, components(3)], configuration management (3), project management(3), component design(3), Compilers(1), Scripts (1), code driver(1), Linker/loaders(1), Syntax and static analyzers(1), Debuggers(1), V&V tools(2), ..

This phase includes the development of the main code (highlighted in Figure 5), runtime controller, individual modules (highlighted in Figure 5), integration of the individual module’s physical databases, problem setup capability, and data analysis and assessment capability. It generally takes five to 10 years for the development of the initial capability of such a project. The steps are summarized below. As the project evolves, the software project management plan will need to be kept current. Risk management is a key issue. If an approach does not work, then alternatives need to be developed and deployed. All of the development should be under strong configuration management. The development of each module should follow a clearly documented plan that describes the domain science, equations, and computational approach. The final module should be thoroughly documented. This is essential for future maintenance and improvements.

IV. Perform V&V

General Software infrastructure tool requirements and best practices: data analysis and visualization tools(2), tools for quantitative comparison of code results with test problem results, other code results and experimental data(2).

Verification provides assurance that the models and equations in the code and the solution algorithms are mathematically correct, i.e., that the computed answers are the correct solutions of the model equations. Validation provides assurance that the models in the code are consistent with the laws of nature and are adequate to simulate the properties of interest. V&V is an ongoing process that lasts the life of the code. However, it is particularly intense during the development of the code and early adoption by the user community. Verification is accomplished with tests that show that the code can reproduce known answers and demonstrate the preservation of known symmetries and other predictable behavior. Validation is accomplished by comparing the code results for an experiment or observation with real data taken from the experiment or observation. A code must first be verified then validated. Without prior verification, agreement between experimental validation data and the code results can only be viewed as fortuitous. Without a successful, documented verification and validation program, there is no reason for any of the project stakeholders to be confident that the code results are accurate.

V. Execute production runs

General Software infrastructure tool requirements and best practices: Data analysis tools, visualization tools, (2), documentation(3), job scheduling(1).

Running a large-scale simulation on a large supercomputer represents a significant investment on the part of the sponsor. Large codes can cost $5M to $10M per year to develop, maintain and run, and more if the costs of validation experiments are included. Large computers are expensive resources. Computer time now costs approximately $1/cpu-hour (2006). A large project will need 5M cpu-hours or more. The total cost is thus in the range of $10M to $20M per year or more and hundreds of millions over the life of the project. It is analogous to getting run time on a large scale experiment...
(e.g., getting beam time at an accelerator, conducting experiments, collecting data, analyzing data). Sponsoring institutions and large scale experimental facilities and teams have learned that research and design activities of this scale require organization and planning to be successful.

VI. Analyze computational results from production runs

Typical production runs from a large-scale computational project can produce TeraBytes or more of data. Analysis of the computational results is not only essential but can also be challenging.

VII. Make Decisions

The whole purpose of the computational system of computers, codes and results analysis is to provide information for the basis of decisions for scientific discovery, engineering design, prediction of operational conditions, etc. Often this is an iterative process. Analysis of the initial results will suggest several conclusions. Further production runs, and possibly further code development, will be needed to confirm those conclusions and suggest modifications. Finally, the decisions and the basis of the decisions need to be documented.

Summary

On the basis of many case studies of successful and unsuccessful Computational Science and Engineering Projects we have identified the steps that such projects follow from initial concept to final conclusions and decisions based on the results of the project. While none of the projects explicitly followed all these steps in an orderly fashion, the successful ones followed these steps either explicitly or implicitly. Key points emerged from the case studies.

1. The level of resources involved in computational science and engineering is becoming large, and the potential impact of the decisions is large. A higher degree of organization and formality is, therefore, inevitable, both for the technical success of the projects and for ensuring that the decisions reached as a result of the projects are correct and accepted by all the stakeholders, especially the sponsors.
2. Development of large scale projects is a highly iterative enterprise and almost always involve an element of research. While the project must be organized and run like a project, highly prescriptive formal designs and processes in the usual Information Technology sense and rigid software management processes are not practical.
3. Development and application of large scale computational science and engineering projects is challenging. Existing development and application support tools are relatively immature compared to the scale of the challenge. Thus there are many opportunities for software and hardware vendors to develop support tools that can reduce the challenge and facilitate code development and production.

Acknowledgments

The authors are grateful for suggestions and support from the members of the DARPA High Productivity Computing Systems (HPCS) Existing Code Analysis Team, Jeffrey Carver, Susan Halversen, Andrew Mark, Dolores Shaffer, and Susan Squires; the leaders of the HPCS productivity team, Jeremy Kepner and Robert Lucas; the members of the various code projects who allowed us to learn from their experiences; Frederick Johnson of the Department of Energy Office of Science; Cray Henry of the Department of Defense High Performance Computing Modernization Program; and Robert Graybill, the leader of the DARPA HPCS program.
HIGH PRODUCTIVITY COMPUTING SYSTEMS AND THE PATH TOWARDS USABLE PETASCALE COMPUTING

PART B: SYSTEM PRODUCTIVITY TECHNOLOGIES

GUEST EDITOR JEREMY KEPNER

AVAILABLE ON-LINE:
http://www.ctwatch.org/quarterly/

E-MAIL CTWatch QUARTERLY:
quarterly@ctwatch.org

CTWATCH IS A COLLABORATIVE EFFORT


CTWATCH IS A PUBLICATION OF THE CYBERINFRASTRUCTURE PARTNERSHIP

http://www.ci-partnership.org/

SPONSORED BY

© 2006 NCSA/University of Illinois Board of Trustees  © 2006 The Regents of the University of California

Any opinions expressed in this publication belong to their respective authors and are not necessarily shared by the sponsoring institutions or the National Science Foundation (NSF).

Any trademarks or trade names, registered or otherwise, that appear in this publication are the property of their respective owners and do not represent endorsement by the editors, publishers, sponsoring institutions or agencies of CTWatch.