Predicting Parallel Applications' Performance Across Platforms Using Partial Execution

Tao Yang[*] Xiaosong Ma[*] [†] Frank Mueller [*]

Abstract

Performance prediction across platforms is increasingly important in today's diverse computing environments. As both programs and their developers face unprecedented wide choices in execution platforms, cross-machine execution time prediction with reasonable accuracy equally benefits scheduling decisions of grid jobs as well as scientists in their research and development planning.

In this paper, we investigate an affordable method approaching **cross-platform performance translation***, based on the notion of* **relative performance** *between two platforms. We argue that relative performance can often be observed without running a parallel application in full. This paper shows that it suffices to observe very short* **partial executions** *of an application since most parallel codes are iterative and behave in a predictable manner after a minimal startup period. This prediction approach is observation-based and does not require program modeling, code analysis, or architectural simulation. Our performance results (using four real-world parallel simulation codes and a total of ten parallel machines with eight distinct architectures) demonstrate that performance prediction derived from partial application executions can yield highly accurate results at a low cost.*

1. Introduction

Users of high-performance computing (HPC) platforms tend to have access to more and more geographically distributed computational resources. Unfortunately, both the resources and the applications in today's distributed computing environment are highly heterogeneous and of great complexity. This makes it difficult to determine the resource usage of a specific application on a wide range of execution platforms. That information is essential in HPC users' decision making when they need to choose a system to obtain access to (or to own) for running their programs, especially for long production executions. For example, it would be helpful to provide scientists with performance estimations of their applications on several large clusters before they decide which cluster to apply for allocations on, and further, how many service units to request on that cluster. Overly inaccurate estimations often cause users to run out of service allocation in the middle of their grant or leave a lot of service units unused when the grant expires, affecting their future allocation application.

The same information is also instrumental in helping a grid resource scheduler to efficiently schedule user jobs. A "meta-scheduler" would like to know how long a job will run on each participating site to improve global load balance and overall job throughput. When the job is scheduled to run at a specific grid-participating site, the local job scheduling system there also needs the job's estimated maximum execution time, which is used as a parameter in their scheduling algorithms [22, 27]. Overly inaccurate estimations here may result in excessive wait times in queues or in forced premature job termination (cancellation) during execution.

For many parallel applications, computational resource usage on a given execution platform is correlated to their execution time on this platform. In this paper, we study *cross-platform execution time prediction* for efficient and accurate resource usage estimation as an affordable *utility* for HPC users and grid schedulers.

There have been numerous studies on parallel programs' performance prediction and many of them can work on multiple platforms (see Section 4). These prior efforts have mostly focused on performance modeling or program simulation. However, the *size*, *diversity*, and *extensibility* of today's HPC environments pose new challenges that traditional performance prediction approaches were not designed to address.

First, it is increasingly difficult to obtain knowledge regarding *both* the applications and the execution platforms. This makes it very challenging for traditional performance prediction to become a *general service*, rather than a case-by-case effort for code optimization. The application developers should not be required to have detailed knowledge about all hardware platforms available. Meanwhile, a job scheduler should not be required to have application-specific knowledge. Second, most traditional performance prediction approaches are too *expensive* for everyday resource usage estimation. For fairly accurate results, the cost of predictions may far exceed the applications' ex-

∗ Department of Computer Science, North Carolina State University, Raleigh, NC, 27695-7534 {tyang2,ma,mueller}@csc.ncsu.edu

† Computer Science and Mathematics Division, Oak Ridge National Laboratory

ecution time. Such approaches normally require both application analysis and system parameter benchmarking, or architecture-specific simulator development.

In contrast, our main approach is ***observation-based*** performance prediction, where we enable very short "test drives" of the applications on multiple candidate platforms to quickly derive the execution time of much longer runs. The results of these test drives can be stored in a database for reusing in future predictions.

One of the key innovations of our work is its reliance on the novel concepts of ***relative performance*** and ***partial execution***. We observe that HPC users often have one or more *reference computers* to develop and test applications. Consequently, our work investigates ***inter-platform performance translation***.

More specifically, this paper aims to answer the following questions through our empirical study:

- Can we predict the overall execution time of a large-scale application on a *target system* using the combination of its known performance on a *reference system* and the *relative performance* between the two systems derived from a very short run of the application?
- How early in this application's execution can we reasonably capture its cross-platform relative performance?
- Will this early observation produce valid predictions for applications with data sizes or complexity that dynamically change as the computation progresses?
- Can we extrapolate the relative performance knowledge collected from partial executions to predict the application's execution time in computing a different problem size or using a different number of processors?
- Will the choice of a reference system be significant? *I.e.*, when an application has known performance data on multiple systems, will we produce similar predictions when using different machines as the reference system?

Our results from four production-scale parallel simulation codes on ten different large parallel computers show very promising prediction accuracy and low prediction overhead: in most cases, with a short execution that takes 1% or less of the total execution time, we obtain an overall execution time prediction accuracy of 97% or higher.

The rest of the paper is organized as follows. Section 2 describes our prediction methods and performance translation algorithms. Section 3 presents experimental results. Section 4 summarizes related work and Section 5 concludes the paper.

2. Methodology and System Design

Scientific applications generally have computationally intensive kernels. The performance of such applications is often constrained by the floating point resources available, memory bandwidth, and the characteristics of inter-processor communication, *via* either shared memory or message passing. Due to these constraints, performance prediction is often challenging. At the same time, scientific applications are characterized by their regularity in the course of executions, specifically with regard to array reference patterns (at the micro level) and alternating phases of computation and communication or I/O (at the macro level). It is this regularity that we exploit for our observation-based performance prediction, in contrast to traditional model- or simulation-based predictions.

The repetitive nature of scientific applications at the macro level is generally a property of the computational model that is based on the notion of convergence in different mathematical approximation methods. Many, if not most, parallel applications are *timestep*-based. In such applications, a timestep is one step of computation followed by inter-processor communication to update data. Timestep computation is repeated until the results converge (*e.g.*, when the simulation object reaches a stable state), or the computation has completed a given number of timesteps.

Traditional model-based or simulation-based approaches to performance prediction generally consider the entire execution of an application and study the interplay between the program and the architecture in a case-by-case manner. For an observation-based approach, executing the entire application takes too long to be acceptable. Instead, our approach utilizes *partial execution* for a limited number of timesteps to capture the *relative performance* across platforms for an application. We argue that *due to the highly repetitive nature of scientific codes, the relative performance observed in this short partial execution is likely to sustain through the entire run*. When used in conjunct with known full execution time of a particular application on a reference platform, this result can then be utilized to perform cross-platform execution time predictions in a very cost-effective way.

2.1. Partial Execution

We have devised an API in support of partial execution for arbitrary applications on clusters. While the design was inspired by the properties of timesteps, the API can also be used in the absence of explicit timesteps — as long as the activities between two consecutive calls closely represent repetitive phases in the application's execution. In the rest of the paper, we use the term "timestep" when referring to these periodic phases. The API for partial execution is as follows:

- init_timestep(): This is an optional call to time-stamp the beginning of an execution. For applications with large start-up overhead, *e.g.*, due to reading large data

sets from secondary storage, this call should be used to separate initialization overhead from subsequent regular timesteps.

- begin_timestep(): This call identifies the beginning of a timestep and allows counters for metrics to be reset between timesteps.
- end_timestep(maxsteps, rampsteps): This call indicates the end of a timestep and implements the logging of metrics pertinent to the timestep work. The first parameter indicates the total number of timesteps before partial execution prematurely terminates the program's execution. The second parameter indicates the number of ramp-up timesteps that should be ignored in the logging activity.

In practice, initialization is a one-time overhead typically with negligible cost to long-running applications, so the initialization call can often be omitted. If the ramp-up parameter is not specified, our implementation uses a default value of 1, assuming the ramp up is finished by the second timestep. Excluding the first timestep allows caches to warm up before timing results enter the performance model. It also allows us to omit the initialization API call since initialization overhead is incurred prior to any timings.

Partial execution utilizing this API allows one to obtain metrics on a per-timestep basis and limits the number of timesteps an application executes. Once this number is exceeded, the application will be terminated prematurely. Hence, the objective of partial execution is not to obtain numerical results from scientific codes but to quickly and cheaply capture their rudimentary execution behavior.

The metrics obtained during partial execution can then be utilized to predict the performance of an application run across different platforms, as detailed below.

2.2. Cross-Platform Performance Prediction

Our approach of observation-based performance prediction is based on two sets of data, one from the *reference platform*, where we have more performance knowledge about the application in question (denoted as A), and one from the *target platform*, where we want to predict the full execution time.

We assume that T_{ref}, the full execution time of A on the reference platform, or num_steps, the total number of timesteps in the full execution, is available. This is reasonable considering that there is at least one "base platform" where the code is developed or tested, and researchers typically keep track of the overall statistics for long-running jobs. In addition, we perform the same partial executions of A on the reference platform as we do on the target platform (see below).

On the target platform, we carry out a set of partial executions of A for a limited number of timesteps. Both the number of partial executions and the number of timesteps per partial execution can be small, as will be demonstrated in the experimental section. The objective of the approach is to inflict minimal time overhead for any executions on both platforms so that performance predictions can be provided quickly. This is especially important when scientists need to select their preferred target from a large set of candidate platforms (based on relative performance, *i.e.*, machine M_1 is x times faster than machine M_2). With known execution times on the reference platform, one can further estimate the absolute performance to supply tight, yet relatively safe bounds on wall-clock time for their submitted jobs, long before having observed a complete run on the target platform, which can takes hours or days.

Base Model: Our predictions are based on the average per-timestep execution time for a set of repeated partial executions on the target platform to obtain the overhead per timestep t_{step} and the initialization t_{init}. The per-timestep averages from multiple runs are then averaged once more over the first n timesteps, *e.g.*, up to the number of timesteps within the partial execution. We call this "prediction using *cumulative averages*" (or running averages). The observed relative performance $\mathbb{R}_{tar_ref}$ between the target and the reference platform can then be calculated using the resulting average per-timestep overhead t_{avg_step} :

$$\mathbb{R}_{tar_ref} = \frac{t_{tar_avg_step}}{t_{ref_avg_step}}$$

Equipped with the above observed relative performance and known execution time T_{ref} on the reference platform, we can estimate absolute performance of A on the target platform T_{tar_est} as:

$$T_{tar_est} = t_{tar_init} + \mathbb{R}_{tar_ref} \times (T_{ref} - t_{ref_init})$$

If T_{ref} is not available but the total number of timesteps, num_steps, is known, we can estimate T_{tar_est} as:

$$T_{tar_est} = t_{tar_init} + t_{tar_avg_step} \times num_steps$$

The accuracy of the above estimation can be assessed by comparing T_{tar_est}, the predicted absolute performance of A, with T_{tar}, the measured performance on the target platform:

$$accuracy = \frac{T_{tar_est}}{T_{tar}}$$

Note that a full execution on the target platform for this metric is only required to assess the model. However, when this prediction technique is applied to a grid job scheduler, one of our target use cases, such full execution time may be obtained at no additional cost from the batch job accounting system. Such "free" information can be conveniently fed back to the prediction model for its self-evaluation and self-adaptation.

Obviously, when the initialization time t_{init} is very small in the overall execution time, the prediction accuracy approximates the observed relative performance *during the short partial executions* against the overall relative performance calculated from actual full executions on both platforms.

Filter Model: The base model of cumulative averages suffices for simple applications and platforms without runtime/OS intervention that affects execution time. To further generalize the model to compensate for fluctuations in execution time, a "filter model" is introduced next. The objective of the filter model is to handle two types of commonly known fluctuations. First, initial fluctuations may occur for multiple timesteps, and not just for initialization code prior to the first timestep. Such fluctuations may be due to warm-up effects of resources, such as caches, but they can also originate from runtime/OS intervention, such as code and/or data migration to better utilize the resources of a given platform. Second, periodic fluctuations are common for additional work performed every k-th timestep, such as checkpointing and I/O for visualization.

The filter model proposed in the following captures both initial and periodic fluctuations. During a partial execution, our API will perform online processing of collected per-timestep timing data. It calculates the ratio between each current timestep with the previous k timesteps and considers the current fluctuation significant if this ratio differs from 1 by a threshold δ or more. Both k and δ are tunable. In practice, we have found that $k = 5$ and $\delta = 0.05$ allow us to identify both one-time anomalies and periodic I/O activities.

Sliding Window Filter Model: We address recurring I/O activities in our prediction by utilizing a *sliding window* of averages instead of cumulative averages. Intuitively, this method uses the average of timestep times collected in a contiguous window of size w. Therefore, to ensure that we include the correct proportion of computation and periodic activities, such as I/O, we need to use a window size that equals the observed period k. The difference between the sliding window vs. the cumulative models materializes after w timesteps. In the presence of I/O, the cumulative approach would be subject to periodic fluctuation while the sliding window provides stability.

With the filter and sliding window models, one of the challenges is to distinguish between random anomalies and periodic fluctuations. We apply heuristics (a) to compare partial execution times from multiple platforms (it is more likely an anomaly if it only occurs on one system) and (b) to detect recurring spikes/dips (it is more likely an anomaly if it only occurs once, especially close to the beginning of execution). If we detect such a one-time anomaly during partial executions, we treat it as a part of the one-time initialization cost and apply our prediction algorithm accordingly as described earlier in this section.

Because this prediction is observation-based, as long as A is executed in the same way across platforms, prediction accuracy will not be affected by various system characteristics, such as different

- processor families, generations and clock frequencies,
- bus interconnects (for shared-memory systems),
- communication interconnects (for networked clusters),
- memory and cache configurations,
- connections from compute nodes to shared disks, and
- system software, such as operating systems and I/O libraries.

Our experiments in Section 3 demonstrate the above. In addition, Section 3 presents irregularities in timestep execution times due to system-dependent anomalies or recurring I/O activities, as well as our enhanced prediction models to handle these situations.

Of course, one major objective of our performance prediction scheme is always to minimize the overhead for partial executions. The question is, *e.g., can we estimate the performance on the target platform using 64 processors with the relative performance observed in partial runs using 8 processors only*? The cheaper and more reusable the partial executions are, the higher we can expect the acceptance of our approach to be by end users.

3. Performance Results

In this section, we present prediction accuracy and other results with our proposed approach using partial execution and the notion of relative performance. Some of the experiments were conducted using the partial execution APIs described in Section 2.1, while others were obtained from scientists who benchmarked per-timestep execution time for their applications. In the second case, we took the first n timesteps to simulate partial executions.

3.1. Experiment Platforms

Our application performance data are collected from ten different parallel computers with eight distinct types of parallel architectures. Table 1 summarizes their technical configurations.

3.2. Base Model and Relative Performance

We evaluated our partial execution method with two benchmarks from the DOE ASCI Purple suite [32], a set of novel applications comprising large-scale parallel codes with inputs resulting in hours of execution. This suite also comprises a mixture of scientific domains, types of meshes, and computation/communication models.

The two applications we successfully ported and tested on four platforms (Datastar-690, Datastar-655, Henry2, and Ram) are Sphot, a 2-D Monte Carlo photon transport code, and sPPM, a 3-D gas dynamics code. For both of them, we chose a problem size that results in hours of execution on

Name	Location	Architecture	CPU	No. nodes	Procs/node	Mem/node	OS	Shared FS
Datastar-690	SDSC	IBM SP4	1.7GHz Power4	8	32	128GB	AIX	GPFS
Datastar-655	SDSC	IBM SP4	1.5GHz Power4	176	8	16GB	AIX	GPFS
Henry2	NCSU	IBM Blade Center	2.8/3.0GHz Xeon	100	2	4GB	Linux	NFS
Ram	ORNL	SGI Altix	1.5GHz Itanium2	256	1	8GB	Linux	XFS
Turing	UIUC	Apple Xserver	2GHz G5	640	2	4GB	Mac OS	NFS
Frost	LLNL	IBM SP3	375MHz Power3	64	16	16GB	AIX	GPFS
Cheetah	ORNL	IBM SP4	1.3GHz Power4	27	32	32/64/128GB	AIX	GPFS
Phoenix	ORNL	Cray X1	vector	512	1	2TB global	UNICOS/mp	StorNext
Seaborg	NERSC	IBM SP3	375MHz Power3	380	16	16/32/64GB	AIX	GPFS
TeraGrid	NCSA	Cluster	1.3/1.5GHz Itanium2	887	2	4/12GB	Linux	GPFS/NFS

Table 1.

the above platforms using 8 processors. These systems all yield fairly small performance variances, and our prediction results are based on 1-2 full executions and 2-5 partial executions.

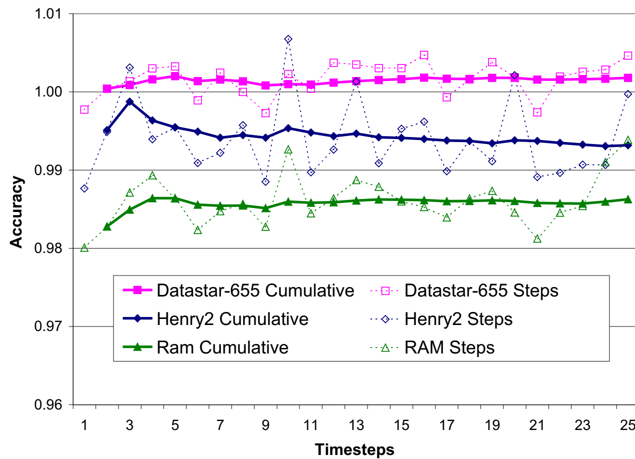

Figure 1. Sphot prediction accuracy using Datastar-690 as the reference platform

Figure 1 shows the prediction results for Sphot, using Datastar-690 as the reference platform and the other three systems as target platforms. For each platform, we portray two prediction methods: *Steps*, where the relative performance used in the ith prediction point is based on the pair of execution time values from timestep i on the reference and target platforms, and *cumulative*, where the relative performance is based on the cumulative average of execution times from timestep 1 to i on both platforms.

In general, Figure 1 demonstrates that our prediction using partial execution yields very accurate results: for all three target systems, the prediction error is within 1.5%. In addition, it demonstrates the following: **(1)** High accuracy can be reached at a very early stage of execution. Even with the first timestep, when initialization and warm-up effects should perturb results on all three target platforms, the prediction accuracy is higher than 98%. Within 5 timesteps, the accuracy on all systems stabilizes at even higher accuracy. **(2)** Partial executions can deliver accurate predictions at a very low cost. In this full execution, Sphot executes thousands of timesteps. On the most time-consuming platform (in this case Ram), the full execution took more than 11 hours while our partial execution of 25 timesteps only took 6 minutes. Moreover, as mentioned earlier our prediction model is accurate even with fewer timesteps as input. Therefore, a partial execution's cost can be bounded by a small maximum wall time for a job. Even when this partial execution itself is terminated prematurely, our model still generates reasonable observation-based predictions. **(3)** The "cumulative" method works better than the "steps" method by smoothing out small irregularities in per-timestep execution times. **(4)** With such uniformly high accuracy, it appears that the selection of a reference platform is not important, at least for this code.

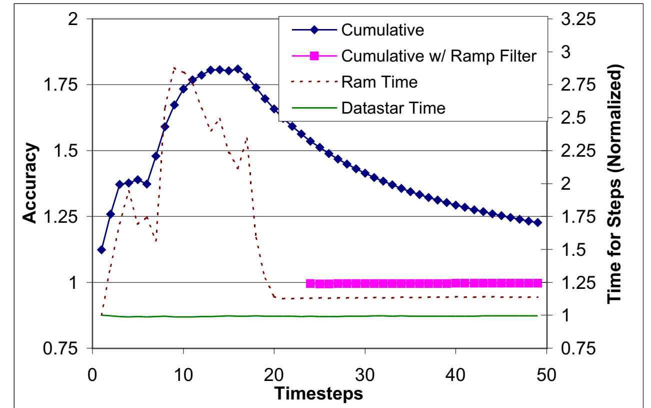

Figure 2. Normalized per-timestep execution time (right axis) and prediction accuracy (left axis) for sPPM

3.3. Filter Model and Initialization Overhead

For sPPM, our filter model is required to obtain accurate predictions. The simple, cumulative model did not suffice for one particular platform, namely RAM. Figure 2 depicts the per-timestep time of the simulation, which increases significantly during the first few timesteps and then drops back before stabilizing after 20 timesteps. This kind of behavior is not observed on Datastar-690, as shown in the same figure, or on any other platform. We suspect that on this NUMA machine, the operating system has been enhanced

to perform page placement based on memory access patterns. As such, pages are moved to the node of most frequently accesses to inflict lower latencies while less frequent accesses my result in longer latencies when being resolved by remote memory accesses.

Our filter model can detect and compensate for this one-time overhead, as explained in Section 2.1. Figure 2 shows the difference in prediction accuracy with and without this "ramp filter". With the non-discriminative cumulative average method, the prediction error can reach 80%, and the effect of misleading relative performance lingers for many timesteps after the anomaly disappears. In contrast, with the improved prediction, the first 23 timesteps will not produce prediction results as they are classified as unstable. Right after that, the prediction instantly yields a consistently high accuracy of over 99%.

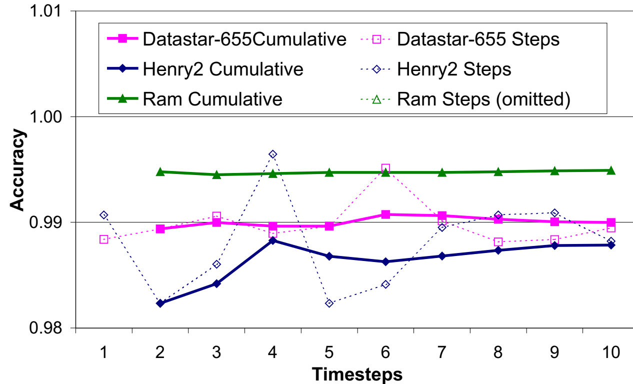

Figure 3. sPPM prediction accuracy using Datastar-690 as the reference platform

Figure 3 depicts the prediction accuracy for sPPM on all three target timesteps. Ram data points represent the first timesteps *after* the relative performance stabilizes using the filter model. sPPM has far more expensive timesteps than Sphot, with each timestep taking around 3 minutes and full runs taking almost 10 hours on Datastar-690 and 655. Therefore, we run only 10 timesteps in our partial executions. Again, the accuracy is remarkably high, at above 98% just after the first timestep.

Comparing the relative performance for Sphot and sPPM also reveals interesting facts. For Sphot, Ram is by far the *worst* platform, where each timestep takes more than 3.5 times as long as on Datastar-690. For sPPM, however, it is by far the *best* platform, where each timestep takes slightly more than 1/6 of the time on Datastar-690. This dramatic contrast is likely due to the different communication and computation patterns of the two codes. For example, sPPM uses frequent large messages, which may benefit from Altix's distributed shared memory architecture. Such phenomena suggest that relative performance across platforms can vary dramatically from application to application (in this case, a 20+ times difference). In addition, system parameters, such as the CPU frequency, do not offer significant information: Ram and Datastar-690 happen to have the same CPU frequency.

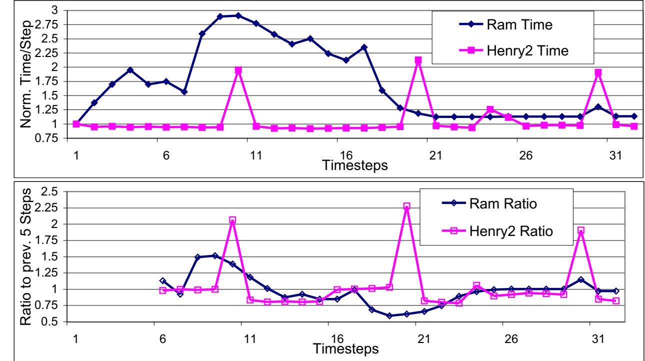

Figure 4. sPPM with high I/O frequency on Ram and Henry2. The top figure shows normalized per-timestep times and the bottom one shows ratios between each current timestep time to the average of previous 5.

3.4. Sliding Window and Periodic I/O

Next, we consider predictions for executions with periodic I/O activities. Such periodic I/O is very common in scientific codes for outputting intermediate results (a) for visualization and analysis and (b) for checkpointing current states to enable efficient restart if the execution is terminated unexpectedly. To save I/O time, most applications choose to periodically generate output every k computational timesteps. Between Sphot and sPPM, the latter provides an easier interface to adjust this I/O frequency. The runs shown above used a default low I/O frequency. Figure 4 depicts sPPM results from runs with a much higher I/O frequency ($m = 10$).

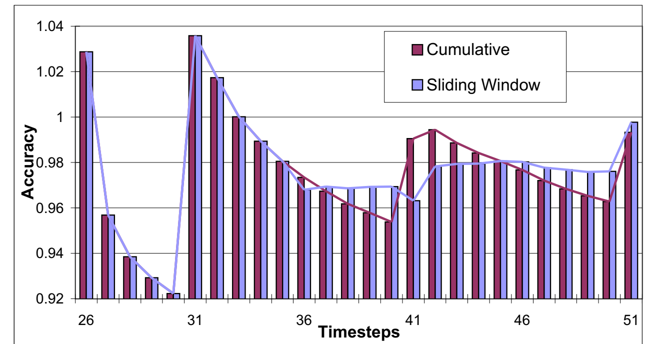

Figure 5. sPPM (with high I/O frequency) prediction results from Ram to Henry2, using the accumulative model and the sliding window model respectively

655 Conf.	8×1	4×2	2×4	1×8
Accuracy	1.002	0.991	1.012	1.004

Table 2. Accuracy in predictions for Datastar-655 runs using different (*number-of-processors-per-node* × *number-of-nodes*) combinations. The total number of processors is fixed at 8.

655 Conf.	2×1	4×1	8×1	8×2	8×4
Accuracy	0.988	0.993	1.002	0.978	0.981

Table 3. Accuracy in predictions for Datastar-655 runs using different (*number-of-processors-per-node* × *number-of-nodes*) combinations. The total number of processors varies from 2 to 32.

Figure 4 and Figure 5 depicts the I/O effect when the performance of sPPM is predicted from Ram to Henry2. This experiment serves a second purpose, namely to show that our ramp filter is valid on a reference platform as well. From the per-timestep timing curves shown in Figure 4, I/O "spikes" can be clearly identified after the execution stabilizes. On Henry2, where computation is faster than on Ram, I/O is significantly slower. Using our filter model that calculates the ratio of each current timestep versus the average of the previous 5 steps, we can successfully identify these I/O spikes as recurring behavior. We can also capture k, the aforementioned periodic I/O frequency in terms of number of timesteps.

We address recurring I/O activities with the *sliding window model*, as discussed in Section 2.2. Figure 5 shows the prediction results (after the initial noise is filtered out on the reference system). For the first 10 timesteps, the sliding window is growing, so the two models perfectly overlap. After that, however, the cumulative algorithm shows a periodic fluctuation in accuracy while the sliding window algorithm is more stable.

We believe that the sliding window model will show more significant advantage if I/O is more frequent and of larger costs. If, however, I/O is sparse and of low cost, it may safely be disregarded by our identification algorithm at all. This is not a big issue though, as the I/O effect would not have a large impact on the overall prediction accuracy in the first place.

Finally, we study different processor/node configurations on the reference and target platforms. As mentioned earlier, our partial execution uses the same number of processors and the same problem size as in the full execution. However, with today's large SMP nodes, a practical configuration may not easily be reproduced on another platform. We subsequently assess the prediction accuracy with the cumulative average method from Datastar-690 (with 32-processor nodes) to Datastar-655 (with 8-processor nodes).

On Datastar-690, we ran all experiments on one node. In the first group of tests, we fixed the number of processors at 8 and varied the number of nodes on Datastar-655 (1, 2, 4, and 8). In the second group of tests, we increased the total number of processors from 2 to 32, where we always tried to minimize the number of nodes to use on Datastar-655. As demonstrated by Table 2 and Table 3, the prediction accuracy remains high in both cases, with no significant variance caused by the different processor/node configurations.

3.5. Application with Varying Timestep Overhead

GENx is a multi-component rocket simulation code developed at the University of Illinois [8]. The performance data we obtained are from model-validation runs simulating lab-scale rockets. We chose this particular simulation since it has an interesting property: the number of particles in its fluid dynamics code increases as time goes on. Therefore, unlike any other codes demonstrated in this paper, this per-timestep execution time grows gradually, reaching a factor of 1.8 at the final point (1550 timesteps in total). We use this code to determine if our model provides reasonable prediction in this situation.

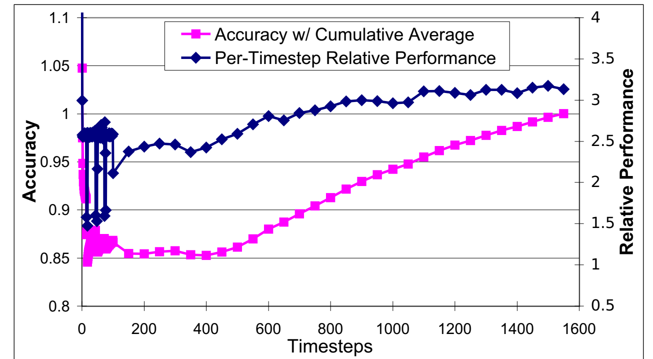

Figure 6. Per-timestep relative performance (right axis) and prediction accuracy with accumulative average (left axis). There is one data point for each of the first 100 timesteps, and one for every 50 timesteps thereafter.

Figure 6 depicts timing and prediction results from two 60-processor runs, which took more than 5 hours on the reference platform (Turing) and 15 hours on the target platform (Frost). It shows that as the simulation progresses, the relative performance between the target and reference platforms gradually increases as well, *i.e.*, the costs per timestep grows *faster* on Frost than on Turing. However, this increase in relative performance (around 24%) is fairly small compared to the increase in per-timestep time (around 45% on Turing and 80% on Frost). A partial execution of 10 timesteps (costing 1.5 minutes on Turing and 4 minutes on Frost) would have yielded a prediction accuracy of 91.6%. Overall, the worst accuracy produced by a partial execution

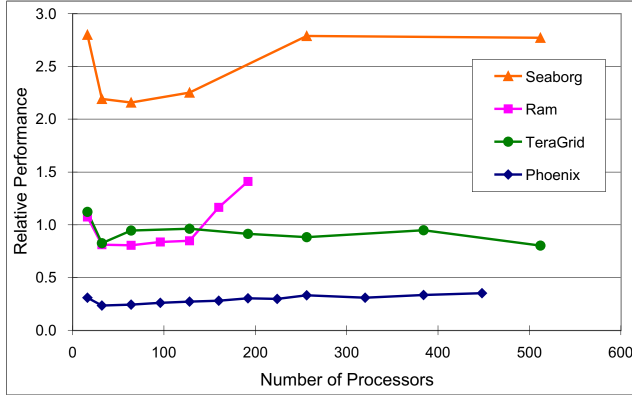

Figure 7. Gyro B1-std relative performance to Cheetah, using different number of processors

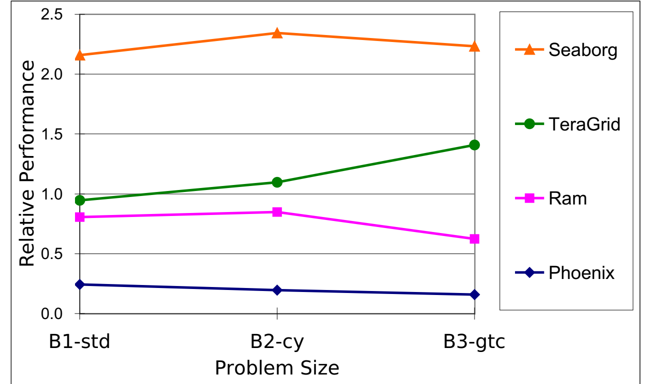

Figure 8. Gyro relative performance to Cheetah, using 64 processors and different input problems

of the first n timesteps ($n = 1, 2, ...1550$) is around 85%. We still consider partial execution capable of delivering reasonable prediction accuracy for this type of codes with dynamic complexity.

3.6. Extension: Application using Varying Number of Nodes and Inputs

GYRO [11] is a code for the numerical simulation of tokamak micro-turbulence solving time-dependent, nonlinear gyrokinetic-Maxwell equations. We obtained a large set of Gyro benchmarking results from ORNL and NCSU researchers who conducted Gyro runs with 3 problem inputs (B1-std, B2-cy, and B3-gtc) on 5 platforms (Cheetah, Ram, Phoenix, Seaborg, and Teragrid) using a variety of processor numbers. Unlike the GENx simulation discussed above, these Gyro runs produce extremely stable per-timestep time, and our prediction easily achieves very high accuracy. Again, the choice of the reference system among the five platforms does not appear to affect the prediction accuracy.

Instead of reporting the accuracy test results, we leverage the abundance of experimental configurations in this case to explore the possibility of *reusing* the relative performance data collected from a pair of partial executions to make predictions for runs using different number of processors or different input data.

	Phoenix	Ram	Seaborg	Teragrid
# Pred.	11	6	5	7
Avg. Error	12.1%	25.5%	16.7%	25.8%

Table 4. Average errors caused by applying the relative performance observed in 16-processor runs to runs using other number of processors

Figure 7 depicts the relative performance of four target platforms against Cheetah using various number of processors (a multiple of 16, limited by hardware availability/configuration on each platform) to compute B1-std. We see that the level of consistency across different numbers of processors varies from platform to platform. Since using a small number of processors is likely to be cheaper (faster to get a job scheduled), we applied the relative performance observed in the run using the fewest processors (16) for each platform when predicting the overall execution time for the other process numbers. Table 4 shows the average prediction error, which varies between 12% and 26%.

	Phoenix	Ram	Seaborg	Teragrid
Avg. Error	37.9%	17.0%	5.6%	23.2%

Table 5. Average errors caused by applying the relative performance observed in B1-std to B2-cy and B3-gtc

Figure 8 plots the relative performance across different problems, and Table 5 shows the average prediction error when we use the relative performance from computing the smallest input problem to predict for the other two problems. This would also reduce the costs of partial executions. *E.g.*, B3-gtc takes up to 3 times longer than B1-std. Here, the average error varies between 5% and 38%. Note that this group of Gyro results is not completely fair to our prediction method, as these several input problems not only bring different amount of computation, but also different computation components to a certain degree.

The above results clearly indicate the trade-off between partial execution overhead and prediction accuracy. In general, the curves in Figure 7 and Figure 8 are smooth, suggesting interpolation may be helpful to mend the gaps between a set of incomplete partial benchmarking results. Depending on the desired level of accuracy, our observation-based prediction approach may demand new partial executions for best accuracy or it may utilize existing application-

specific data to give a quick "ball-park" estimate.

4. Related Work

Grid Job Scheduling: There has been an increasing interest and efforts on grid scheduling (also called *meta-scheduling*) [14, 26, 29, 24], mostly built upon local job schedulers for executing jobs on distributed computing resources. It is recognized that job execution time is an important resource specification item to be translated across machines. However, existing or under-development grid schedulers either do not offer execution time translation, thereby implying that users are responsible for specifying a "safe" maximum wall time value across machines, or they adopt simplified translation methods, such as stretching the execution time with the CPU frequency ratio between two machines [24], which is known to be very inaccurate. Other work on cross-platform performance prediction (*e.g.*, Prophesy [30]) was mostly based on modeling computational kernels instead of complex applications with diverse tasks. Our work can form a building block for future grid schedulers by offering affordable job execution time predictions for diverse applications and platforms.

Parallel Program Performance Prediction: There have been numerous previous studies of performance prediction for parallel programs. Many of these studies are built upon performance modeling techniques (e.g., [1, 7, 10, 15, 21, 28, 31]) requiring either in-depth knowledge of the applications to build analytical models (e.g., [2, 18, 25, 33, 34]) or special compiler/instrumentation tools to infer such knowledge from parallel codes (e.g., [4, 6, 12, 21]). With careful modeling of applications and platforms, many of these previous studies achieved high prediction accuracy. However, detailed modeling often compromises the portability of prediction tools. *E.g.*, some existing approaches are application-specific [2, 16] or language-specific [6, 12]. In addition, a number of prediction techniques are based on simulations (e.g., [3, 4]) where simulators are used to measure the execution time of applications.

Most of the work mentioned above targeted performance prediction for the purpose of *performance optimization*. In contrast, our method targets performance prediction as a means for making resource usage estimation to help application owners in their research planning and daily use of diverse computing resources. In such cases, users may not be able or willing to afford traditional performance prediction techniques, which require a fair amount of work due to model building or instrumentation plus simulation. Especially today's multi-component codes, such as the rocket simulation discussed in this paper, comprise modules from many application domains with diverse computational models/algorithms making analytical modeling very hard. Also, they often use external libraries (MPI, BLAS, PETSc, NetCDF, just to name a few), and/or multiple languages (*e.g.*, mixed C/Fortran/C++ programming). This greatly decreases the feasibility and effectiveness of both analytical and compiler-aided performance modeling. Finally, given the large number of application-platform combinations in future grid environments, simulation-based prediction can consume excessive system resources themselves. In contrast, we develop *observation-based* performance prediction, which does not require in-depth knowledge of parallel codes or systems. This makes our approach application-independent, language-independent, and platform-independent.

Further, many multi-platform performance modeling efforts (e.g., [3, 4, 5, 15, 20, 21]) evaluated their approaches with data collected at multiple supercomputers. However, data from each machine are processed individually, so are predictions and evaluations performed. Our approach, instead, combines benchmarking results from multiple platforms for cross-platform prediction.

Several recent studies addressed performance prediction in heterogeneous grid environments [13]. A few projects addressed relative performance [16] and performance portability [17, 23]. However, we are not aware of work on performance prediction *based on* relative performance.

Finally, the repetitive behavior of applications has been exploited in speeding up run times on architecture simulators [19] and predicting performance metrics based on history information [9]. Such studies exploit repetition at "instruction block" level while we exploit larger-scale and more explicit behavior repetition in high performance scientific codes, based on the iterative nature many of them possess.

5. Conclusion

In this paper, we demonstrated the benefit of a black-box style, observation-based performance prediction approach built on the notion of relative performance and utilizing affordable, short partial application executions. We believe the merits of this approach lie in its *simplicity*, *portability*, and *cost-effectiveness*, making it ideal for performance prediction as a general service to HPC users and grid schedulers.

In addition, we consider a major contribution of this paper to be reporting relative performance measurements and prediction results from multiple production-scale real-world codes on a total of ten large parallel computers. In most cases, we obtained prediction accuracy probably higher than required by our target customers. We acknowledge that these results may not fully generalize from the group of applications currently available to us to arbitrary code. Our model may not exhaustively capture uncommon application behavior. We plan to study more applications, build a relative performance database,and potentially investigate patterns in relative performance across platforms and applications.

6. Acknowledgements

This work was supported in part by NSF grants CNS-0406305, CCF-0429653, CCR-0237570, and Xiaosong Ma's joint appointment between NCSU and ORNL. SDSC, ORNL, and NCSU HPC Center provided computational resources. The authors of the ASCI Purple benchmarks helped during the installation and benchmarking of these applications. Patrick Worley at ORNL, Hongzhang Shan at LBNL, as well as Kumar Mahinthakumar and Sarat Sreepathi at NCSU provided benchmarking data for Gyro. Robert Fiedler and Xiangmin Jiao at UIUC provided the GENx benchmarking data.

References

[1] V. Adve and R. Sakellariou. Application representations for multiparadigm performance modeling of large-scale parallel scientific codes. *The International Journal of High Performance Computing Applications*, 14(4), 2000.

[2] B. Armstrong and R. Eigenmann. Performance forecasting: Towards a methodology for characterizing large computational applications. In *Proceedings of the International Conference on Parallel Processing*, August 1998.

[3] R. Bagrodia, E. Deeljman, S. Docy, and T. Phan. Performance prediction of large parallel applications using parallel simulations. In *Principles Practice of Parallel Programming*, 1999.

[4] J. Bourgeois and F. Spies. Performance prediction of an nas benchmark program with chronosmix environment. In *Proceedings of the 6th International Euro-Par Conference*, 2000.

[5] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of Supercomputing*, 2000.

[6] M. Clement and M. Quinn. Automated performance prediction for scalable parallel computing. *Parallel Computing*, 23(10), 1997.

[7] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[8] W. Dick and M. Heath. Whole system simulation of solid propellant rockets. In *Proceedings of the 38th AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit*, Indianapolis, IN, July 2002.

[9] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[10] M. Faerman, A. Su, R. Wolski, and F. Berman. Adaptive performance prediction for distributed data-intensive applications. In *Proceedings of Supercomputing*, 1999.

[11] M. Fahey and J. Candy. GYRO: A 5-d gyrokinetic-maxwell solver. In *Proceedings of Supercomputing*, 2004.

[12] T. Fahringer and H. Zima. A static parameter based performance prediction tool for parallel programs. In *Proceedings of the International Conference on Supercomputing*, 1993.

[13] S. Jarvis, D. Spooner, H. Keung, G. Nudd, J. Cao, and S. Saini. Performance prediction and its use in parallel and distributed computing systems. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[14] W. Jones, L. Pang, D. Stanzione, and W. Ligon. Bandwidth-aware co-allocating meta-schedulers for mini-grid architectures. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2004.

[15] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of Supercomputing*, 2001.

[16] D. Kerbyson, A. Hoisie, and H. Wasserman. A comparison between the earth simulator and alphaserver systems using predictive application performance models. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[17] W. Ligon. Research directions in parallel I/O for clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[18] C. Lim, Y. Low, B. Gan, and W. Cai. Implementation lessons of performance prediction tool for parallel conservative simulation. In *Proceedings of the 6th International Euro-Par Conference*, 2000.

[19] W. Liu and M. Huang. EXPERT: expedited simulation exploiting program behavior repetition. In *Proceedings of the 18th Annual International Conference on Supercomputing*, 2004.

[20] X. Ma, M. Winslett, J. Lee, and S. Yu. Improving MPI-IO output performance with active buffering plus threads. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[21] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS*, 2004.

[22] A. Mu'alem and D. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6), 2001.

[23] R. Reussner and G. Hunzelmann. Achieving performance portability with SKaMPI for high-performance MPI programs. In *Proceedings of the International Conference on Computational Science*, 2001.

[24] H. Shan, L. Oliker, and R. Biswas. Job superscheduler architecture and performance in computational grid environments. In *Proceedings of Supercomputing '03*, 2003.

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

[26] C. Smith. Open source metascheduling for virtual organizations with the community scheduler framework (CSF). Technical whitepaper, http://www.platform.com/resources/whitepapers/.

[27] W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. In *Proceedings of IPPS Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[28] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Proceedings of Supercomputing*, 2002.

[29] Supercluster.org. SILVER design specification. http://www.supercluster.org/silver/specoverview.shtml.

[30] V. Taylor, X. Wu, and R. Stevens. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *ACM SIGMETRICS Performance Evaluation Review*, 30(4), 2003.

[31] A. van Gemund. Symbolic performance modeling of parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(2), 2003.

[32] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.

[33] A. Wagner, H. Sreekantaswamy, and S. Chanson. Performance models for the processor farm paradigm. *IEEE Transactions on Parallel and Distributed Systems*, 8(5), 1997.

[34] X. Zhang and Z. Xu. Multiprocessor scalability predictions through detailed program execution analysis. In *Proceedings of the 9th ACM International Conference on Supercomputing*, 1995.