2015 IEEE 29th International Parallel and Distributed Processing Symposium

Matching Application Signatures for Performance Predictions using a Single Execution

Anirudh Jayakumar, Prakash Murali, Sathish Vadhiyar
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore, India
jayakumar.anirudh@gmail.com, sercprakash@ssl.serc.iisc.in, vss@serc.iisc.in

***Abstract*—Performance predictions for large problem sizes and processors using limited small scale runs are useful for a variety of purposes including scalability projections, and help in minimizing the time taken for constructing training data for building performance models. In this paper, we present a prediction framework that matches execution signatures for performance predictions of HPC applications using a single small scale application execution. Our framework extracts execution signatures of applications and performs automatic phase identification of different application phases. Application signatures of the different phases are matched with the execution profiles of reference kernels stored in a kernel database. The performance of the reference kernels are then used to predict the performance of the application phases. For phases that do not match significantly, our framework performs static analysis of loops and functions in the application to provide prediction ranges. We demonstrate this integrated set of techniques in our framework with three large scale applications, including GTC, a Particle-in-Cell code for turbulence simulation, Sweep3d, a 3D neutron transport application and SMG2000, a multigrid solver. We show that our prediction ranges are accurate in most cases.**

***Keywords*—Modeling; Prediction; Matching Application Signatures; Kernels; Phase Identification;**

I. Introduction

Performance characterization and predictions of parallel applications are essential and have long been used for various purposes including scalability studies [1], identifying performance bottlenecks [2], projections for future systems [3] and tuning applications and algorithms [4]. A common approach for performance prediction is to execute or benchmark the application for different processors and problem sizes, observe the execution profiles including execution times, and employ curve-fitting and machine learning techniques to map the observed execution profiles to a performance model [5]–[7]. The performance model can then be used for predicting performance for a new problem size and number of processors. In many of the existing strategies, significant number of benchmarks are performed under controlled conditions to obtain performance predictions with reasonable accuracy, resulting in long training times for the model.

Limiting the number of benchmarks needed for building the performance models for predictions will help minimize the time taken for performing the benchmarking experiments and the modeling process. Moreover, in certain constrained environments, the benchmark runs and results are implicitly limited. For example, in some large supercomputer systems, application developers execute their applications with small problem sizes on small number of processors of a special queue called *debug queue* for development, tuning and debugging purposes before performing large scale production runs on large number of processors of *production queues*. The debug runs are limited and are performed for very small number of problem and system size configurations. A performance modeling system for predicting performance of production runs will have to be built using the limited debug runs.

In this work, we have developed a prediction framework for performance predictions of HPC applications using a single small scale application execution. Our framework employs a novel strategy of matching execution profiles of the different phases of the parallel applications to parallel reference kernels stored in a kernel database. The reference kernels are standard benchmarks from diverse application domains as prescribed by Colella's seven dwarfs [8], Berkeley View's thirteen motifs [9], and TORCH testbed of computational reference kernels [10]. Our framework provides a suite, *RK-suite*, of implementations, execution profiles and performance models of reference kernels. Specifically, *RK-suite* consists of 1. a collection, *RK-collection*, of these reference kernel implementations, 2. execution profiles, *RK-profile*, including cache hits and misses, instruction mix etc. obtained using benchmarking runs of the reference kernels for a finite set of problem sizes and number of processors, and 3. a performance model, *RK-model*, that can be used to predict execution times of the kernel implementations for other problem sizes and processors. We claim that such a *RK-Suite* can be useful for a number of purposes including evaluations and comparisons of the high performance computing systems by supercomputer installations, and hardware and software tuning by the vendors.

For a given application executed with a small problem size and number of processors, we collect the execution profiles or execution signatures of the application, automatically detect the significant phases of the application, and match the normalized execution profiles of the phases and the reference kernels. For example, one of the reference kernels in our *RK-collection* is a parallel FFT implemented in the NAS Parallel Benchmark (NPB) [11]. The FFT implementation is executed with different problem sizes and number of processors and the execution profiles, *RK-profile*, are collected for these runs. The execution times are predicted for other problem sizes and number of processors using *RK-model*. An application like Community Earth System Model (CESM) [12] can consist of FFT calculations as one of its phases. The normalized

1530-2075/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPS.2015.20

execution profile of the FFT calculations of the application is compared with the normalized execution profile of the NPB FFT reference kernel.

If there is a strong match, the execution estimates of the reference implementation, *RK-model*, are used to predict the performance of the application phase for large-scale runs on large number of processors. For application phases that do not match significantly with any of the reference kernels, we use static analysis of the variables, loops and functions used in the application phase to give prediction ranges.

We demonstrate this integrated set of techniques in our framework with three large scale applications, including GTC [13], a Particle-in-Cell code for turbulence simulation, Sweep3d [14], a 3D neutron transport application and SMG2000 [15], a multigrid solver. We show that our predictions have errors in the range 0.4-18.7%, which are considered to be good predictions even in controlled prediction environments in which multiple training runs are performed. We show that some of the time-consuming phases in these applications exhibit strong matching with the corresponding reference kernels of completely different implementations. This provides an important insight that the fundamental routines in many of the large-scale applications may exhibit similar coding and execution patterns.

Section II presents the overall design of our prediction framework. Section III and IV describe our approach for performance modeling using signature matching and static analysis techniques, respectively. Section V discusses the phase identification required for our performance modeling framework. Section VI describes our experiments and results in performance modeling and prediction for three large scale applications using a single execution run on a smaller number of processors. In Section VII, we describe relevant work on performance modeling. Section VIII summarizes our work and presents scope for future work.

II. Prediction Framework

The overall working of our prediction framework is shown in Figure 1. Our framework first classifies an application execution into phases based on a single execution training run (Section V), and collects application execution profiles for these phases. It then tries to compare the execution profile of a phase with the reference kernels stored in our reference kernel database (Section III-A). If there is a strong match with a kernel, it then uses the execution performance of the matched kernel and the performance of the application phase during the training run to predict the performance for larger problem size and processor configurations (Section III). If the phase does not strongly match with any of the reference kernels, our framework performs static analysis of the loops and functions in the application phases, finds the critical variables used in these loops and functions, and form composite performance models in terms of the critical variables (Section IV).

III. Performance Prediction using Signature Matching

The fundamental components related to our signature matching approach are a set of reference kernels, and a distance calculation method for finding similarities between two execution profiles.

A. Reference Kernels

Our parallel reference kernels, *RK-collection*, belong to the application classes of Colella's dwarfs [8] and Berkeley's motifs [9], and correspond to the reference kernels maintained by the TORCH testbed [10]. The reference kernels in TORCH belong to various application domains including dense and sparse linear algebra computations, finite difference and finite volume methods on structured grids, spectral methods, particle based interactions, Monte Carlo methods, graph and sorting problems. TORCH also provides serial implementations of these kernels. These kernels and application classes are considered to cover most of the scientific applications. Our current collection consists of parallel implementations of eighteen of the kernels, and is shown in Table I.

For each of these kernels, we perform both strong and weak scaling runs. For strong scaling runs, we choose a particular problem size and execute on different power-of-two number of processors, and also execute for different problem sizes on a particular number of processors. For weak scaling, we perform executions on power-of-two number of processors such that the ratio, N/P, is kept constant, where N is the problem size, and P is the number of processors. We obtain the execution times corresponding to these runs. We then use the actual execution times obtained for a finite set of N and P to build a performance model, *RK-model*, that predicts the execution time of the kernel for different N and P. For performance modeling, we used cubic spline interpolation which constructs a smooth piecewise polynomial curve that passes through the data points. If the new data point for prediction lies within the training data range, cubic spline gives an interpolated value at that point. This enables us to interpolate the kernel execution time for varying problem sizes and processor configurations. When the new data point lies outside the training data, we use model based predictions using either the well known complexities provided by the kernel developers or models based on generalized curve fitting. For example, the complexities of the matrix computations used in the HPCC benchmark are well known. If the complexities are not available, we use generalized curve fitting to obtain generic models in terms of N and P, assuming a fixed set of polynomial and logarithmic complexities of N and P, namely, N, N^2, N^3, $logN$, $1/P$, and $1/logP$, and obtain the best-fit model. Such assumptions of the complexities based on the observations of the complexities in most of the practical parallel applications and algorithms have also been used in earlier efforts [1].

For one of the runs, we also obtain the execution signature for the kernel, *RK-profile*. In particular, we obtain the normalized histogram of the instructions including floating point adds, multiplies, load, stores etc. used in the execution. We use the TAU profiling toolkit [21] to trace the instruction pointer (IP) and obtain the instructions used in a kernel execution. We instruct TAU to collect the kernel trace information at a sampling frequency of ten million instructions. Our framework parses the output from TAU, adds the number of times different instructions were executed across all the sampling intervals and forms the normalized histogram of the instructions. Normalization is performed with the total number of instructions used during the entire execution. Such a normalization helps to compare problems of different sizes.

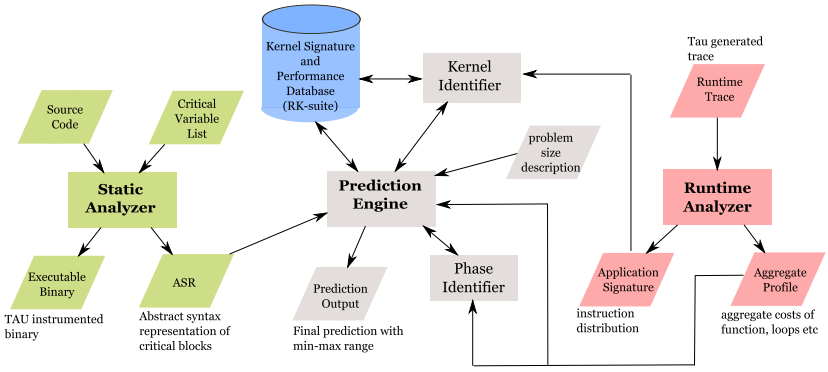


Fig. 1. Prediction Framework

TABLE I. REFERENCE KERNEL DATABASE

Benchmark Suite	Reference Kernel
NAS Parallel Benchmark (NPB) [11]	1. *bt* - block tridiagonal solver 2. *fft* - fast fourier transform 3. *cg* - conjugate gradient 4. *lu* - LU factorization 5. *is* - integer sorting 6. *mg* - multigrid solver 7. *sp* - scalar penta-diagonal solver
Skeleton Particle-in-Cell (PIC) Codes [16]	8. *dep* - deposit kernel 9. *push* - push kernel
Polyhedral Benchmrak Suite, PolyBench [17]	10. *gemv* - matrix-vector multiplication 11. *sym* - symmetric matrix multiplication 12. *trm* - triangular matrix multiplication
ParkBench [18]	13. *dmm* - general dense matrix multiplication 14. *qr* - QR factorization 15. *trd* - matrix tridiagonalization
HPC Challenge (HPCC) benchmark suite [19]	16. *smm* - sparse matrix multiplication 17. em tran - matrix transpose
Berkeley Benchmarking and Optimization (BeBOp) SpMV benchmark [20]	18. *spmv* - sparse matrix-vector multiplication

The actual execution times of the kernel benchmark runs, the performance model, and the execution signatures together constitute our kernel database, *RK-suite*. We claim that such a database for these fundamental kernels can be used for a variety of purposes including evaluation of supercomputer systems by an installation site.

B. Application Signatures and Matching

For predicting the performance of a target large-scale scientific application using the kernel runs, we perform a *small-scale training run* of the application with a given problem size, N_{small}, on a small number of processors, P_{small}. During the execution, we obtain the samples of execution traces including the executed instructions at a sampling frequency of ten million instructions using the TAU profiling toolkit. For a given application phase, we form the normalized histogram of the instructions used during the phase. We also observe the execution time of the application phase as t_{small}.

We then compare the normalized instruction histogram of the application phase with the normalized instruction histograms of the kernels in the kernel database. We choose the most similar kernel for subsequent performance prediction of the application if the similarity measure with the kernel is within a specific threshold. For calculating similarities between two normalized histograms, we use the χ^2 (pronounced "chi-square") distance metric. For two histograms P and Q with K bins, the χ^2 distance is defined as

$$\chi^2(P,Q) = \sum_{i=1}^{K} \frac{(P[i]-Q[i])^2}{P[i]+Q[i]} \quad (1)$$

For each bin, the summation of bin counts in the denominator of Equation 1 implies that χ^2 distance considers small differences between large bins to be less important than a similar difference between small bins. χ^2 distance assumes a value between 0 and 2 with smaller distance implying greater similarity between the application phase and the kernel. We calculate the similarity as $1-\chi^2/2$, thus confining the similarity values to lie between 0 and 1 with higher values denoting higher similarities. We choose the most similar kernel if the similarity value between the application phase and the kernel is at least equal to a threshold. In all our experiments, the most similar kernel matched with a similarity value in the range $[0.85, 0.9]$. Hence for our current work, we chose the threshold value as 0.85. In our future work, we plan to explore smaller thresholds and using kernels with smaller similarity values for predictions.

We use the executed instructions used for matching application phases and kernels since this is effective in matching

executed profiles of the same implementation with different problem sizes and number of processors. Moreover, using instructions for matching also corresponds to our hypothesis that most scientific applications belonging to a common class or implementing the same algorithm exhibit similar coding patterns. Also, the number of time-critical operations for a particular algorithm is invariant for different implementations with the same data structures. Thus, matching using executed instructions can identify different implementations of the same algorithm with the same time complexities. Other metrics including normalized number of loads, stores, cache misses, branch instructions etc. can vary even for the same implementation for different problem sizes due to the inter-play of the application and hardware characteristics. We also experimentally verified that considering these other metrics did not give good similarity measures for the same application executed with different problem sizes and number of processors.

C. Application Prediction using Matched Kernel

We predict the performance or execution time of an application phase for a large problem size, N_{large}, and/or number of processors, P_{large}, using the single small-scale application training run with N_{small} and P_{small}, the corresponding execution time, t_{small}, and the performance model of the matched kernel. We use the performance model of the kernel to find the problem size, N_{kernel}, for which the estimated execution time of the kernel, t_{kernel}, on P_{small} number of processors is the most similar to the actual execution time of the application phase, t_{small}, observed for the small scale training run (Section III-B). i.e., we find N_{kernel} such that $|f(N_{kernel}, P_{small}) - t_{small}|$ is minimum, where $f()$ is the performance modeling function for the kernel.

We match the execution times of the application phase and the kernel since a problem size, N_{small}, of the application phase with the implementation followed in the application may correspond to another problem size, N_{kernel} of the kernel implementation. We calculate the ratio of these two problem sizes as:

$$sFactor_{size} = \frac{N_{small}}{N_{kernel}} \tag{2}$$

We also calculate the ratio of the corresponding execution times of the application phase and the kernel as:

$$sFactor_{time} = \frac{t_{small}}{f(N_{kernel}, P_{small})} \tag{3}$$

We then predict the execution time of the application phase for the larger problem size, N_{large}, and number of processors, P_{large} using the model function for the kernel as:

$$t_{large} = sFactor_{time} \times f(\frac{N_{large}}{sFactor_{size}}, P_{large}) \tag{4}$$

We obtain the predicted total execution time of the entire application for N_{large} and P_{large} as the sum of the predicted execution times of the individual application phases.

TABLE II. MEAN SIMILARITIES BETWEEN NORMALIZED HISTOGRAMS FOR THREE NPB BENCHMARKS

Benchmark	Similarities Across Iterations for the same problem size	Similarities Across Processes for the same problem size	Similarities Across Problem Sizes
CG	0.925	0.93	0.92
FT	0.95	0.95	0.95
MG	0.89	0.89	0.885

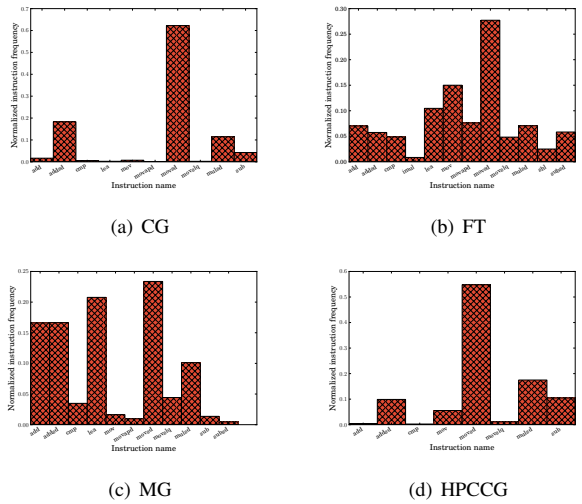
D. Demonstration

We demonstrate the promise and potential of our signature matching techniques for predictions with three NAS parallel benchmarks (NPB), namely, CG, FT and MG. We ran NPB with problem sizes corresponding to classes A, B and C on a dual octo-core Intel Xeon E5-2670 2.6 GHz server with CentOS 6.4. We obtained normalized instruction histogram for each sampling interval in each process. We then computed the mean χ^2 distance across different iterations of the loop of an application for a problem size, across different processes for an execution for a problem size, and across different problem sizes. We computed the corresponding similarity values as $1 - \chi^2/2$. Table II shows the similarity values for the CG, FT and MG benchmarks of NPB. We find the similarity values for a benchmark are close to 90% and show little variations when applied to iterations and processes of a single problem size and also for multiple problem sizes. Thus, our technique of signature matching can be reliably used for performance prediction for larger problem sizes using execution profiles for a small scale run. Comparitively, the similarity values between the different benchmarks are low: the CG-FT, CG-MG and FT-MG similarities are 0.63, 0.71 and 0.79, respectively. Thus, the the signature matching technique can clearly find the most similar kernel and distinguish unrelated kernels.

To demonstrate that the signature matching technique can also be used to match two different implementations, we computed the normalized histogram distances between the three NPB benchmarks and the HPCCG application [22] with a $256 \times 256 \times 256$ grid. HPCCG also performs conjugate gradient calculations like the CG benchmark of NPB, but with a different implementation. Figure 2 shows the normalized instructions for HPCCG and the three NPB benchmarks, CG, FT and MG. We find that the instruction mix in HPCCG is most similar to CG: there are very large number of move operations ($movsd$) and a significant number of double-precision addition, multiplication and subtraction operations. The instruction mix in FT and MG are very different from that of HPCCG. The distances computed between the normalized histograms of HPCCG and CG, FT and MG are 0.875, 0.51 and 0.635, respectively. Thus, our signature matching technique adequately captures the conjugate gradient computations in the HPCCG application using the NPB CG implementation.

IV. STATIC ANALYSIS

For an application phase whose execution signature does not match with an existing reference kernel, we use static compiler-based analyses and use a best-effort approach to derive accurate performance models for the phase. We analyze the individual loops and functions of the phase, derive fine-level performance models for these, and form a composite model of the phase in terms of these loop-level and function-level models. The model due to static analysis generates only



(a) CG (b) FT

(c) MG (d) HPCCG

Fig. 2. Histogram of Instructions used in CG, FT and MG of NBP, and HPCCG

loose bounds for execution times and hence not applicable for predicting large fractions of the entire application.

We require the user to provide the *entry-level* names of the variables in the application that denote the problem size, along with the values of the variables. These are the first set of variables in the application program that are assigned the values for the problem sizes either using an application configuration file or as a command line argument. For example, in a three-dimensional problem, the user may input the x, y and z dimensions for problem size. We also require the user to specify the subset of problem size variables whose values are divided by the number of processors. We denote such variables as *P-dependent variables* and the other variables as *P-independent*. In the 3-D problem, the domain may be decomposed across the processors in the x and y dimensions, but not in the z dimension. In this case, x and y will be P-dependent and z will be P-independent. Our static analyzer first parses the application code to find a list of *critical variables* that are derived from or dependent on the entry-level variables. The static analyzer particularly looks for assignment statements and function arguments to find the critical variables. In addition, the analyzer also evaluates the assignment expressions in terms of critical variables using basic operations of addition, subtraction, multiplication and division. A critical variable may either be P-dependent or P-independent depending on the entry-level problem size variables it was derived from.

The static analyzer then parses the code to find *critical blocks*. We denote the loops, user-level functions and MPI functions that are dependent on the critical variables as critical blocks. A critical block that does not contain any other critical block is denoted as a *fundamental block*. In our work, single-level loops and MPI functions are fundamental blocks. A critical block can nest other critical blocks. We denote such a block as a *super block*. A super block, in addition to nesting critical blocks, can also contain other regions of code denoted as *basic regions*.

We model the time complexity of a single-level loop dependent on a critical variable, C, in terms of the number of iterations using either linear or logarithmic functions. Particularly, we model the number of iterations as C or $logC$, if C is P-independent, and C/P or $logC/P$ if P-dependent. We obtain the total time of the loop corresponding to the single execution training run, and fit the time complexity function of the loop as:

$$a \times iterations = totalTime \tag{5}$$

where $iterations$ can either be C, $logC$, C/P, $logC/P$ and $totalTime$ is the time for the loop corresponding to the single execution. We find the coefficient a and estimate the time for the loop for a prediction run as $a \times iterations_{pred}$ where $iterations_{predicted}$ is the predicted number of iterations for the prediction run obtained using the value of C for the prediction run.

Our static analyzer also parses the MPI functions that are dependent on critical variables. The static analyzer uses a *MPI performance table* to determine the message size as a function of the critical variable, C, used in a MPI function. The MPI performance table contains the times taken for different MPI functions for different message sizes and number of processors. We construct such a performance table using LogGP MPI benchmark suite [23]. Supercomputer installations will be interested in obtaining MPI performance of their systems and will construct the table at the time of the installation. We estimate the time for the MPI function for the prediction run using the MPI performance table.

For modeling the time complexity of a super block containing some critical blocks and basic regions, we first determine the time for the basic regions, $basicCost$, by subtracting the times for the critical blocks from the total time of the super block for the single execution run. We then model the time complexity of the super block using the time complexity functions of the nested critical blocks, $basicCost$ for the basic regions, the critical variable C used in the super block and the total time taken for the super block, $totalTime_{super}$ as:

$$a \times f_{super}(C) \times (basicCost + \sum f_i) = totalTime_{super} \tag{6}$$

where f_i is the performance model for a critical block i nested in the super block, and $f_{super}(C)$ of the superblock can be either one of C, $logC$, C/P, $logC/P$. We find the coefficient a using the time for the single execution run and use it to predict the time for the prediction run.

We construct these performance models in a bottom-up manner for the calling context tree of the application phases, and determine the overall performance model for the entire phase. Since we use either a linear or log function in the performance models of the blocks, we provide a range of time estimates for the prediction run.

For implementation of the static analyzer, we used the TAU profiling toolkit [21] to obtain the total times for the various blocks, and to construct the calling context tree. We used the ROSE framework [24] to obtain the critical variables and blocks based on the entry-level variables. The ROSE framework builds an abstract syntax tree (AST) for the application code, and also provides APIs to traverse the tree and read information in each node. Our static analyzer builds a XML file containing the calling context of the critical blocks along with information for each critical block including its

TABLE III. STATIC ANALYSIS RESULTS FOR NAS PARALLEL BENCHMARKS

Benchmark	Training Run (Class:Procs)	Prediction Run (Class:Procs)	Min Predicted time (secs)	Max Predicted time (secs)	Actual Time (msecs)
CG	B:4	D:32	2618	5764	4900
FT	C:16	D:128	134	754	467
MG	C:4	D:32	213	533	471

type, critical variables used by the block, and the total times taken by the block in the single execution training run. For performance prediction of a prediction run, this XML file is augmented with the values of the critical variables and the number of processors for the single-execution training and the prediction runs, and is given as input to our prediction engine. The prediction engine starts from the inner-most block in the calling context tree and incrementally derives performance estimates. While our static analyzer currently works with the source codes of the applications, it can also be extended to work with application binaries using existing binary analysis frameworks [25].

A. Demonstration

We demonstrate the accuracy of performance prediction ranges obtained using static analysis with NPB. We perform a single execution run with a benchmark for a small problem size and number of processors and predict the performance of the benchmarks for larger configurations. The CG and MG benchmarks were run on a cluster of quad-core AMD Opteron 2218 based 2.64 GHz Sun Fire servers with CentOS 4.3 and connected by Gigabit Ethernet. The FT benchmark was run on a cluster of 64-core AMD Opteron 6274 2.4Ghz servers with CentOS 6.2 and connected using Infiniband. Table III shows the ranges of the predictions and the actual execution times. We find that in all cases, the actual execution times are within the prediction ranges.

V. PHASE IDENTIFICATION

For both our approaches related to matching signatures and static analysis, the phases of an application will have to be identified. Phase identification in application execution has been targeted in earlier efforts for performance prediction, and for runtime adaptations including hardware reconfiguration and changing scheduling and load-balancing strategies [26], [27]. For our purpose of matching application phases with reference kernels and developing performance models for the phases, we identify phases by obtaining aggregate profile generated for the small-scale training run by the TAU toolkit and choosing all functions or subroutines that have consumed more than 5% of the total execution time of the training run. The executions of the chosen functions and subroutines constituted our application phases. Our strategy of phase identification by identifying the time-consuming subroutines is applicable to a large number of practical codes since most applications are written in a modular way with important operations encapsulated as subroutines. By fixing the threshold as 5% for identifying time-consuming subroutines, we consider any medium to long duration functions as potential phases for signature matching, and not consider small duration functions with negligible execution times.

TABLE IV. TRAINING AND PREDICTION CONFIGURATIONS FOR THE THREE APPLICATIONS ON TYRONE AND PARAM CLUSTER

Configuration	Training Run		Prediction Run		
Cores	16	32	128	256	1024
GTC					
No. of particles (10^6)	14.75	73.73	1106	2654	11059
No. of grids	921600	1843200	3686400	3686400	3686400
Runtime (seconds)	293(T)	695(T), 584(P)	2843(T)	3200(T)	4225(P)
Sweep3d					
Grid points (3D)	150^3	200^3	640^3	1000^3	1500^3
Runtime (seconds)	160(T)	172(T), 14(P)	1586(T)	3530(T)	157(P)
SMG2000					
Grid points (3D)	50^3	60^3	100^3	128^3	
Runtime (seconds)	925(T)	1168(T)	8281(T)	26914(T)	

For each of these subroutines or functions, we obtain the function entry/exit timestamps from the TAU trace. From the sampling trace, the instruction pointers between the entry-exit ranges are collected and their corresponding instruction type (*mov,mul* etc.) are retrieved from the *objdump* of the binary. The *objdump* tool displays the assembler mnemonics for the machine instructions from the executable binary. The distribution of the instructions forms the signature of the subroutine.

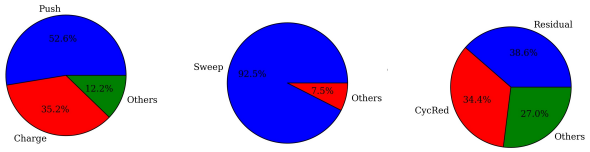
VI. EXPERIMENTS AND RESULTS

We use our overall prediction framework containing kernel matching and static analysis techniques for performance predictions of three large-scale applications, namely, GTC [13], a Particle-in-Cell code for turbulence simulation, Sweep3d [14], a 3D neutron transport application and SMG2000 [15], a multigrid solver. The experiments were carried out on two clusers: a 800-core heterogeneous cluster called *Tyrone* located in our department and a 3600-core cluster called *Param Yuva2* located in Center for Development of Advanced Computing (CDAC), Pune, India. The Tyrone cluster consists of 17 nodes, 9 nodes with 32-cores each and 8-nodes with 64-cores each. Each of the nodes has 2.2 GHz AMD Opteron 6274 processor and 128GB RAM. The cluster nodes are connected using Infiniband. For our experiments, we used a maximum of 4 nodes of 64-core configurations. The Param Yuva2 cluster consists of 225 nodes with 16 cores each. Each of the nodes has 2.6 GHz dual octo-core Intel Xeon E5-2670 CPU and 64 GB RAM. For our experiments, we used 64 nodes of 1024 cores.

Table IV shows the configurations used for single-exectution training run and prediction runs used for the three applications on the two clusters. The qualifiers, *(T)* and *(P)* represent the executions on the Tyrone and the Param clusters, respectively. As can be seen in the table, the problem size is scaled with the number of cores. We also find that we use 4-10 minute training runs to predict for runs that execute for more than an hour.

A. Prediction Times and Errors

In our experiments, we found that our prediction framework takes about 2.5 minutes to obtain data from the training run. This includes about 45 seconds to perform the static analysis of the run, about a minute for extracting the signatures, and about 45 seconds for matching the signatures with the



(a) GTC (b) Sweep3d (c) SMG2000

Fig. 3. Distribution of Times in the Three Applications for the 16-core Training Run

reference kernel database. Given this data from the training run, our framework takes only about 2-3 seconds for prediction for a large-scale run.

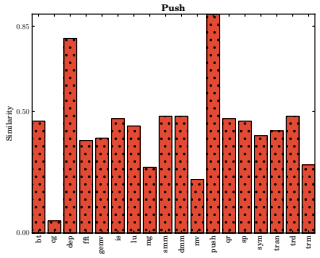
B. GTC

Gyrokinetic Toroidal Code (GTC) is a 3-dimensional code used to study microturbulence in magnetically confined toroidal fusion plasmas via the Particle-In-Cell (PIC) method. The application is composed of six main kernels for each time step: charge deposition from particles onto the grid (*charge*), solving the gyrokinetic Poisson equation on the grid (*poisson*), computing the electric field on the grid (*field*), using the field vector and other derivatives to advance particles (*push*), smoothing the charge density and potential vectors (*smooth*), and moving the particles between processors or toroidal domains (*shift*).

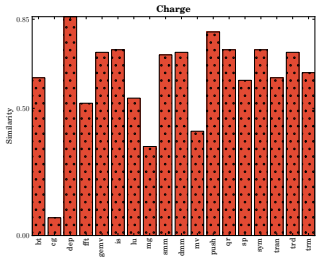
Of the six main kernels of GTC, the *charge* and *push* kernels act on particle data and hence consume more time as compared to other operations that act mainly on grid data. Figure 3(a) shows the distributions of the times consumed in the *charge*, *push* and the remaining kernels for the single-execution training run. We find that the *charge* and *push* kernels together contribute 87.8% of the total runtime.

Our automatic prediction framework matched the *charge* and *push* kernels with the reference kernels by finding similarity measures. Figure 4 shows the similarity values of these two GTC kernels with the kernels contained in our reference kernel database. We find that the two GTC kernels showed the strongest match or highest similarities with the corresponding PIC kernels in our reference kernel collection, even though the PIC kernels are from a completely different software package from a different group. Specifically, the *charge* and *push* kernels of GTC matched with the deposit, *dep*, and *push* kernels of our reference kernel collection with the similarity scores of greater than 0.85. This result and the subsequent similar results with the other two applications reaffirm our hypothesis that most scientific applications implementing common functionalities or methods or algorithms exhibit similar codes.

Our prediction framework then estimated the total time for the GTC application for prediction runs by estimating the times for the *charge* and *push* kernels using signature matching technique described in Section III and the times for the other kernels using static analyses described in Section IV. Figure 5 shows the estimated and actual times for the kernels and the entire applications for the prediction runs on 128 and 256 cores of the Tyrone cluster. To show that our prediction results are not sensitive to the type of single-execution training run, we also show results when the training run was performed



(a) Push



(b) Charge

Fig. 4. Similarities of GTC Kernels with the Reference Kernels

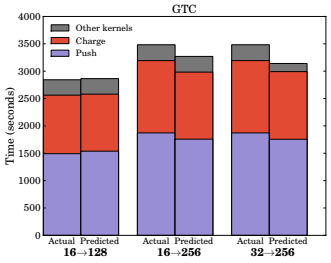


Fig. 5. Predictions of Execution Times for GTC on 128 and 256 Cores on the Tyrone Cluster

on 32 cores. Our predictions for the other kernels denote the maximum of the range predicted by the static analysis. Table V shows the prediction errors for the kernels and the entire application for the prediction runs for all the three applications. We find that for GTC, in all cases, we obtain prediction errors less than 11%, which is significantly accurate considering that only a single training run was used for the predictions. We also find that the prediction accuracy for 256-core execution is slightly better when predicting using 16-core training run than with the 32-core training run. We believe that this accuracy inversion is due to possible extraneous system loads at the time of the executions.

C. Sweep3d

The ASCI Sweep3D is a scientific benchmark solving a three-dimensional neutron transport problem from a scattering source. The numerical solution to the transport equation involves the discrete ordinates (Sn) method and the procedure of source iteration. The solution is by a direct ordered solve known as a "sweep". The *Sweep()* subroutine constitutes the majority of the runtime in Sweep3D. Figure 3(b) shows the distributions of the times consumed in the *sweep* and the

TABLE V. PERCENTAGE PREDICTION ERRORS FOR THE THREE APPLICATIONS ON THE TYRONE CLUSTER

Training Run	16 procs.		32 procs.
Prediction Run	128 procs.	256 procs.	256 procs.
GTC			
charge	2.6	7.3	6.4
push	3.1	6.1	6.2
Entire app	0.4-0.7	6.1-6.7	9.8-10.7
Sweep3d			
sweep	18.1	7.7	2.4
Entire app	17.8-18.7	7.6-8.4	1.2-1.8
SMG2000			
CycRed	11.28	16.06	18.42
Residual	6.28	9.91	13.51
Entire app	2.8-6.0	13.0-15.0	7.5-8.7

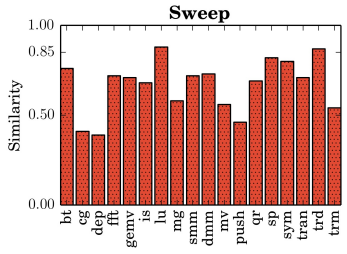


Fig. 6. Similarities of Sweep3d Kernel with the Reference Kernels

remaining kernels for the single-execution training run. We find that the *sweep* subroutine consumed about 92% of the total runtime. Our automatic prediction framework matched the *sweep* kernel with the reference kernels and found the strongest match with the *lu* kernel of the reference collection with a similarity score of 0.88. Figure 6 shows the similarity values of the *sweep* kernel with the kernels contained in our reference kernel database.

Our prediction framework then estimated the total time for the Sweep3d application for prediction runs by estimating the times for the *sweep* kernel using signature matching and the spline curve of the *lu* reference kernel. It predicted the times for the other kernels using static analyses. Figure 7 shows the estimated and actual times for the kernels and the entire applications for the prediction runs on 128 and 256 cores of the Tyrone cluster. Table V shows the prediction errors for the *sweep* kernels and the entire application for the prediction runs of Sweep3d. Our prediction errors are between 1.2% and 18.7% in all cases.

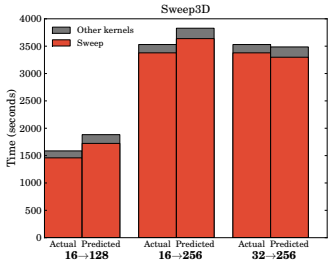
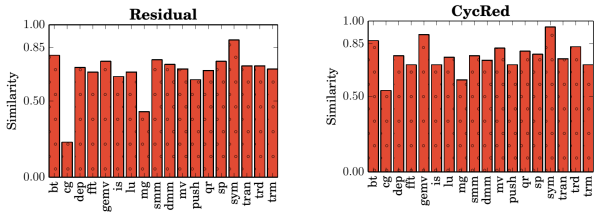


Fig. 7. Predictions of Execution Times for Sweep3d on 128 and 256 Cores on the Tyrone Cluster



(a) Residual (b) CycRed

Fig. 8. Similarities of SMG2000 Kernels with the Reference Kernels

D. SMG2000

SMG2000 is a parallel semicoarsening multigrid solver that solves linear systems that result from hierarchically discretizing differential equations on logically rectangular grids. The 3D algorithm semicoarsens in the z-dimension followed by plane relaxation. Plane solves invoke one V-cycle of the 2D algorithm that semicoarsens in the y-dimension and are followed by line relaxation. To perform the relaxation step for the 3D problem, SMG invokes multiple 2D plane solves which in turn employ multiple 1D line solves. These line solves are performed using *cyclic reduction*, which is a recursive algorithm to solve tridiagonal linear systems. At the end of each iteration SMG2000 invokes the *residual kernel* to compute the error in solution.

The *cyclic reduction (CycRed)* and *Residual* kernels consume most of the execution times. Figure 3(c) shows the distributions of the times consumed in these and the remaining kernels for the single-execution training run. Our automatic prediction framework matched the *Residual* kernel of SMG2000 with the symmetric matrix multiplication (*sym*) reference kernel and the *CycRed* kernel of SMG2000 with the block triagonal solver (*bt*) reference kernel of our RK-suite. Figure 8 shows the similarity values of the two SMG2000 kernels with the kernels contained in our reference kernel database. Even though the *CycRed* kernel of SMG2000 shows higher similarities with the *gemv* and *sym* reference kernels than with the *bt* kernel, our framework matches the parallel *CycRed* with the parallel *bt* kernel since the *gemv* and *sym* are serial kernels. Our framework automatically identifies if an application phase is serial or parallel by looking for MPI functions in the static code analysis. Both the *CycRed* and the *Residual* kernels of SMG2000 match with the corresponding reference kernels with similarity scores of 0.88 and 0.9, respectively.

Our prediction framework then estimated the total time for the SMG2000 application for prediction runs by estimating the times for the *CycRed* and *Residual* kernels using signature matching technique and the times for the remaining kernels using static analyses. Figure 9 shows the estimated and actual times for the two kernels and the entire applications for the prediction runs on 128 and 256 cores of the Tyrone cluster. The times for other kernels in the figure denote the maximum of the ranges predicted by the static analysis. We note that the other kernels which consumed 27% of execution time for the small scale run shown in Figure 3(c), consume a small percentage for the 128 and 256-core runs. Table V shows the prediction errors for the kernels and the entire application for the prediction runs of SMG2000. The total application prediction error ranges

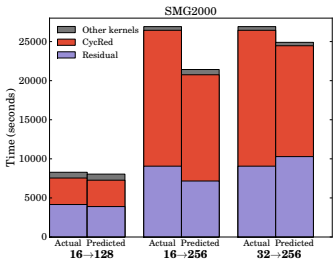


Fig. 9. Predictions of Execution Times for SMG2000 on 128 and 256 Cores

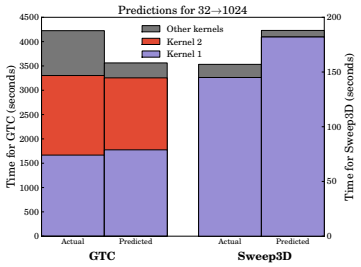


Fig. 10. Predictions of Execution Times on 1024 cores of the Param Yuva2 cluster using 32-core training runs

from 2.8% to 15.0%.

Figure 10 shows the actual and predicted times for the GTC and Sweep3d applications on 1024 cores of the Param Yuva2 cluster using the training run performed only on 32 cores. We find that the prediction errors were less than 20%. These results demonstrate the ability of our framework to predict for large scale runs involving large/very large number of processors using only a small-scale training run involving only a few minutes of execution time. Thus our framework can be used for scalability studies of applications that can help administrators decide on large-scale procurements.

E. Discussion

While our results demonstrated the potential of our prediction framework, the framework has some limitations. Our current methodology considers only the distribution of instructions for matching kernels. Our work assumes that two applications with similar signatures for small scale executions will scale similarly for larger number of processors. However, this assumption does not hold for those codes which have different computation and communication patterns for larger problem sizes and number of processors. Our framework also does not consider different load conditions of the system and network that can exist during the training and prediction runs. Our static analysis can also miss data-dependent paths in the application codes. Finally, our framework is not applicable for languages with dynamic binding.

VII. Related Work

There have been a number of performance modeling and prediction frameworks that perform benchmark runs on small scale systems and problem sizes, and predict for large scale runs. For example, the work by Lee et al. [5] uses statistical analyses between application parameters and performance including clustering, association and correlation analyses, and employ piecewise polynomial regression and artificial neural networks for predicting performance. The work by Singh et al. [6] uses multi-layered neural networks on the sample data obtained from benchmark runs for a subset of parameter spaces to predict for the remaining parameter spaces. Barnes et al. [7] employ regression-based approaches using execution times, per-processor information and global critical paths of execution to predict performance. Many of these approaches train the performance models using a significant number of benchmark runs on controlled environments, and consequently obtain good prediction accuracies with prediction errors in the range 2.2-17.3%. Our work focuses on constrained environments in which the training set can be limited and non-deterministic.

Bailey and Snavley [28] collect application signatures including memory accesses and communication patterns, determine target machine characteristics including memory and communication capabilities in an application-independent manner and use convolution methods to combine application signatures with machine profiles to extrapolate performance for larger systems. This effort also requires conducting a series of experiments on different machines to obtain application signature summaries. They obtain larger prediction errors in the range 13-30%, since their models do not consider all necessary parameters on all the machines. Also, the number of application and machine parameters can be large for modern architectures.

The concept of predicting performance of applications using extensive database of performance data of small fundamental modules appeared in the Prophesy framework by Taylor et al. [29]. They use composition of the models of the modules to build a performance model for an application. In a recent work, He et al. [30] hypothesized that a small set of basic data flow patterns or performance idioms such as stream, transpose, reduction, random access and stencil covered scientific applications. They built a framework that automatically identified the idioms in applications, and used the models of the idioms for predictions of the applications. They demonstrate their method with NAS parallel benchmarks (NPB) as applications and motivate the definition of suitable idioms for approximating performance. They report approximation accuracy in the range 80-96%. Our work extends the concept of idioms to parallel application kernels for predicting performance of much larger scale applications, and obtain similar accuracy.

The work by Calotoiu et al. [1] has developed a tool that automatically generates performance model for each region of the application. They then use small number of runs of the application, perform extrapolations using the models and match the extrapolated performance using their models with the user expectations. The emphasis of their work is not on performance modeling accuracy but to provide a binary scalability indicator to detect scalability bottlenecks of the different regions of the code.

The work by Sharkawi et al. [3] also projects application performance by correlating the performance metrics of the application with the metrics of some fundamental benchmarks on a base system. In particular, they use the benchmarks from SPEC CFP2006 for correlation with the application, identify a

combination of the benchmarks referred to as surrogates that have similar properties with the HPC application. They use this information along with the performance of the surrogates on a target machine to project the performance of the HPC application on the target machine. They report average projection difference to measured runtimes in the range 7-10%. While the emphasis of their work is to project node level performance of a HPC application with a fixed problem size on a target machine using a base machine, our work focuses on projections for larger problem and processor sizes on a system.

VIII. Conclusions and Future Work

In this work, we had developed a prediction framework that predicts performance of large-scale runs of parallel applications using a single small-scale training run. Our framework matches the application phases to reference parallel kernels and uses the performance of matched kernels for predictions. We demonstrated our techniques with three large-scale real scientific applications. We obtained predictions with errors in the range 0.4-18.7%, which are considered to be good predictions even in controlled prediction environments in which multiple training runs are performed with different problem sizes and number of processors. The performance estimates by our prediction framework can be used for batch job submissions and scheduling that employ strategies like backfilling in production HPC environments. In the future, we plan to investigate probabilistic predictions for application phases with low similarities with the reference kernels. We also plan to use our prediction framework in supercomputer sites to predict production runs using debug executions.

IX. Acknowledgements

This work is supported by Department of Science and Technology (DST), India via the grant SR/S3/EECE/0095/2012. The authors would like to thank the National Param Supercomputing Facility (NPSF), Centre for Development of Advanced Computing (CDAC), Pune, India for providing access to their large-scale Param Yuva2 system.

References

[1] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, 2013.

[2] G. Llort, H. Servat, J. González, J. Giménez, and J. Labarta, "On the Usefulness of Object Tracking Techniques in Performance Analysis," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, 2013.

[3] S. Sharkawi, D. DeSota, R. Panda, R. Indukuru, S. Stevens, V. Taylor, and X. Wu, "Performance Projection of HPC Applications using SPEC CFP2006 Benchmarks," in *Proceedings of the International Parallel and Distributed Processing Symposium*, ser. IPDPS '09, 2009.

[4] H. Servat, G. Llort, J. Gimenez, K. Huck, and J. Labarta, "Unveiling Internal Evolution of Parallel Application Computation Phases," in *Proceedings of the International Conference on Parallel Processing*, 2011.

[5] B. Lee, D. Brooks, B. de Supinski, M. Schulz, K. Singh, and S. McKee, "Methods of Inference and Learning for Performance Modeling of Parallel Applications," in *PPOPP*, 2007.

[6] K. Singh, E. Ipek, S. McKee, B. de Supinski, M. Schulz, and R. Caruana, "Predicting Parallel Application Performance via Machine Learning Approaches," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 17, pp. 2219–2235, 2007.

[7] B. Barnes, B. Rountree, D. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz, "A Regression-based Approach to Scalability Prediction," in *Proceedings of the 22nd Annual International Conference on Supercomputing*, ser. ICS '08, 2008.

[8] P. Colella, "Defining Software Requirements for Scientific Computing," DARPA HPCS Presentation, 2004.

[9] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006.

[10] A. Kaiser, S. Williams, K. Madduri, K. Ibrahim, D. Bailey, J. Demmel, and E. Strohmaier, "TORCH Computational Reference Kernels: A Testbed for Computer Science Research," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-144, 2010.

[11] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," NASA, Tech. Rep. NAS-95-020, 1995.

[12] "Community Earth System Model (CESM)," http://www2.cesm.ucar.edu.

[13] D. Quinlan *et al.*, "Gyrokinetic Particle Simulations," http://w3.pppl.gov/theory/proj_gksim.html.

[14] "Sweep3d," http://www.ccs3.lanl.gov/PAL/software/sweep3d.

[15] P. Brown, R. Falgout, and J. Jones, "Semicoarsening Multigrid on Distributed Memory Machines," *SIAM Journal on Scientific Computing*, vol. 21, pp. 1823–1834, 2000.

[16] V. Decyk, "Skeleton PIC codes for parallel computers," *Computer Physics Communications*, vol. 87, no. 1-2, pp. 87–94, 1995.

[17] "PolyBench/C the Polyhedral Benchmark Suite," http://web.cse.ohio-state.edu/~pouchet/software/polybench.

[18] T. Hey and D. Lancaster, "The Development of Parkbench and Performance Prediction," *International Journal of High Performance Computing Applications*, vol. 14, pp. 205–215, 2000.

[19] P. Luszczek, D. Bailey, J. Dongarra, J. Kepner, R. Lucas, R. Rabenseifner, and D. Takahashi, "The HPC Challenge (HPCC) Benchmark Suite," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06, 2006.

[20] "BeBoP SpMV Benchmark," http://bebop.cs.berkeley.edu/spmvbench.

[21] S. Shende and A. D. Malony, "The TAU Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–331, 2006.

[22] "Sandia National Laboratory - Mantevo Project," https://software.sandia.gov/mantevo/.

[23] T. Hoefler, A. Lichei, and W. Rehm, "Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks," in *IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '07, 2007.

[24] D. Quinlan *et al.*, "ROSE Compiler Infrastructure," http://www.rosecompiler.org.

[25] "MAQAO (Modular Assembly Quality Analyzer and Optimizer)," http://www.maqao.org.

[26] C.-H. Chang, P. Liu, and J.-J. Wu, "Sampling-Based Phase Classification and Prediction for Multi-threaded Program Execution on Multicore Architectures," in *International Conference on Parallel Processing (ICPP)*, ser. ICPP 2013, 2013.

[27] C. Ding, S. Dwarkadas, M. Huang, K. Shen, and J. Carter, "Program Phase Detection and Exploitation," in *20th International Parallel and Distributed Processing Symposium*, April 2006.

[28] D. Bailey and A. Snavely, "Performance Modeling: Understanding the Past and Predicting the Future," in *Euro-Par*, 2005.

[29] V. Taylor, X. Wu, J. Geisler, X. Li, Z. Lan, M. Hereld, I. Judson, and R. Stevens, "Prophesy: Automating the Modeling Process," in *Active Middleware Services*, 2001, pp. 3–11.

[30] J. He, A. Snavely, R. van der Wijngaart, and M. Frumkin, "Automatic Recognition of Performance Idioms in Scientific Applications," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011.