Performance Prediction of Parallel Applications Based on Small Scale Executions

Rodrigo Escobar
Computer Science Department
University of Texas at San Antonio
San Antonio, TX, USA
rodrigod.escobar@gmail.com

Rajendra V. Boppana
Computer Science Department
University of Texas at San Antonio
San Antonio, TX, USA
rajendra.boppana@utsa.edu

***Abstract*—Predicting the execution time of parallel applications in High Performance Computing (HPC) clusters has served different objectives, including helping developers to find relevant areas of code that require fine tuning, designing better job schedulers to increase clusters' utilization, and detecting system bottlenecks, among others. In order to predict the run time of a target application, it is usually decomposed into several phases that exhibit different execution characteristics or that are bounded by communication events. Common techniques that have been recently used to extract the phases of an application involve the analysis of complex trace files, extracting the opcode mix of each phase or machine learning-based classification. In this work we present a novel approach for performance prediction that uses empirical analysis of small scale execution times and the relative portion of time spent in significant phases of a target application in order to identify matching kernels from a previously built reference database. Our technique is easy to use and does not require previous knowledge about trace file formats, static analysis or synthetic code generation.**

Our experiments show that running a few times the target application with small scale input sizes and identifying a subset of phases that accounts for a significant portion of the total run time is enough to obtain accurate predictions for parallel scientific applications. We evaluate our methodology using three well known applications: SMG2000, SNAP and HPCG. Our prediction errors range from 1% to 20%.

***Keywords*-performance prediction, High Performance Computing, parallel processing, phase matching**

I. Introduction

Prediction of execution time of parallel applications in High Performance Computing (HPC) clusters serve several purposes. For instance, by being able to project the execution times of applications and their constituent relevant phases, developers can narrow the scope of their effort to fine tune application code. Similarly, by having projected execution times system schedulers can dispatch jobs more efficiently so as to increase overall clusters' utilization. Furthermore, run time predictions can also be used to help on detecting system bottlenecks or to aid on application design before target machines are available.

Several techniques have been used to predict the run time of applications. Most approaches decompose the target application on relevant phases that exhibit different run time characteristics. The identification of such phases have been done with techniques that range from manual instrumentation of source code, to automatic analysis of log traces obtained from running the application with a small input size. In order to predict the performance of the application with a large input size, its phases are identified and a prediction for each phase is performed independently. Then, all predictions are aggregated to produce the overall run time prediction for the application. Common techniques used to predict the performance of application phases include the creation and execution of synthetic code or skeletons [1], static analysis, curve fitting [2], and regression[3].

Recently, Jayakumar et al. [2] introduced a framework for performance predictions that is composed of three steps: 1) construction of a reference kernel database, 2) phase extraction and phase-to-kernel matching based on opcode mix, and 3) run time prediction using curve interpolation. One of the benefits of this approach is that unlike with skeletons, the application does not need to be run after the phases have been matched to the reference kernels. However, in this work we show that the opcode mix of a phase can vary between different compilers and between different compiler optimization flags.

In this paper we present a novel approach for performance prediction that uses empirical analysis of small scale execution times and the relative portion of time spent in significant phases to identify matching kernels from a reference database. Our technique does not require previous knowledge of compiler optimizations or opcode mix extraction. Instead, small scale executions are used to detect and extract relevant performance characteristics of the composing phases of the application and to match each phase with a kernel from a previously built reference database.

In order to evaluate the quality of our predictions, we selected three different applications: SMG2000[4], SNAP[5], and HPCG[6]. We applied our methodology with 8, 32 and 40 processes on a cluster of 48 cores. The remaining cores are left available for the typical tasks that are done regularly by the operating system. From our experiments, we obtain low prediction errors, which range from 1% - 20% while reducing the complexities of previously existing methods.

The rest of this paper is organized as follows: Section II presents previous work on the field of performance prediction. Section III presents our proposed methodology. Our experimental set up and results are presented in section IV. Finally, conclusions are provided in section V.

II. Related Work

Sodhi et al. [1] proposed a technique based on the automatic construction of a scaled down synthetic application that has the same execution characteristics as the application it intends to mimic although it has no semantic relevance. In their approach they run the application and extract the traces necessary to generate a sequence of computation and communication events that are later translated to actual C code. Our approach does not need to analyze trace files. Trace files tend to be large and complex. Instead, we make use of profile information and a set of reference kernels that represent most scientific workloads executed in HPC.

Wong et al. [3] proposed a methodology that creates a signature based on send/receive events extracted from an execution trace. The events are used to identify the phases of the application and then a run time signature is created based on the points where a phase begins and end. The signature then must be executed on the machines where the application will be run in order to perform the prediction. Our approach differs from this technique in that it does not use execution traces and the phases are not identified by send/receive events. Our technique extract the phases of the application based on the run time of each one of its functions, this is a faster method that only needs to profile the application a few times with different small input sizes.

Jayakumar et al. [2] proposed framework for performance predictions for large problem sizes and processors using single small scale runs. In their work, a database of performance models for a set of reference kernels is built. The model of each kernel includes a cubic spline of its runtime and a histogram of its opcode mix. Each bin of the histogram corresponds to a particular opcode. When an application needs to be predicted, a small scale run of the application is performed. Each function of the application that consumes more than 5% of the total execution time is considered a phase. For each phase, an opcode mix histogram is also obtained. The matching procedure of application phases to reference kernels is based on how similar the opcode mix of the small scale run is compared to the reference kernels. A limitation with this approach is that the opcode mix of an application can vary depending on the compiler used. Furthermore, it can be sensitive to different compiler optimization flags even when the same compiler is used. An example of the different opcode mixes that can be obtained for the same code can be seen in Figure 1. Once a match has been found for a phase, an input size correction is done using an textitsFactor (*sF*). SFactor is the size of the matched kernel for which the kernel runs in the same amount of time that the phase does when run with the small scale input size. Once the sFactor is computed, the spline model of the matched kernel is used to predict the runtime of the phase when a large scale input size is specified. For the remaining phases that don't match any kernel in the database, a prediction based on static analysis of source code is performed. Although we follow a similar approach, in our work the matching procedure is based on small scale runs of the application and the identification of the fraction of time spent on relevant phases. This allows us to match phases to kernels without looking at the opcode mix of phases and without having to do static analysis of source codes or binary files.

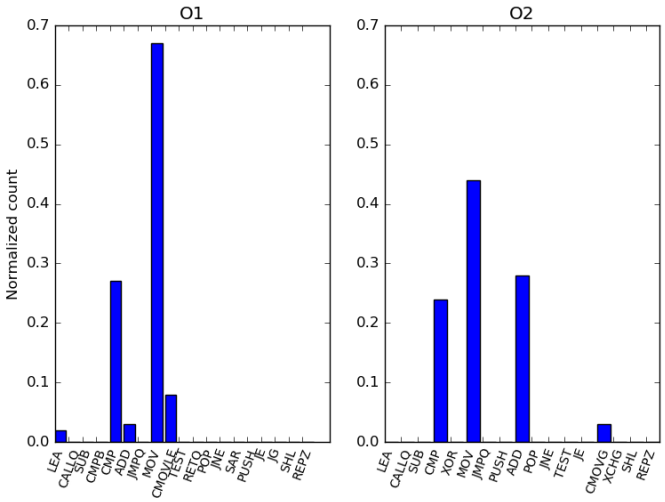

Figure 1. Opcode mix of the Polybench's floyd-warshall program compiled with GCC using O1(left) and O2(right) optimization flags

Yang et al. [7] presented an approach that is based on the observation of partial executions. In their work, they argue that observing a partial execution of an application is sufficient since most parallel codes are iterative. They introduce an API that marks the beginning and end of each phase. In their work, the API calls need to be inserted manually in the code of the application and the phase identification is also manual. In our work, the phase identification is done automatically using the profiles extracted from small executions.

III. Methodology

Our prediction methodology is outlined in Figure 2. Similar to [2], in order to predict the performance of parallel applications on HPC clusters, our methodology is composed of three steps: Construction of a reference kernel database, phase detection and matching, and prediction of run time. These steps are described in detail in sections III-A, III-B, and III-C.

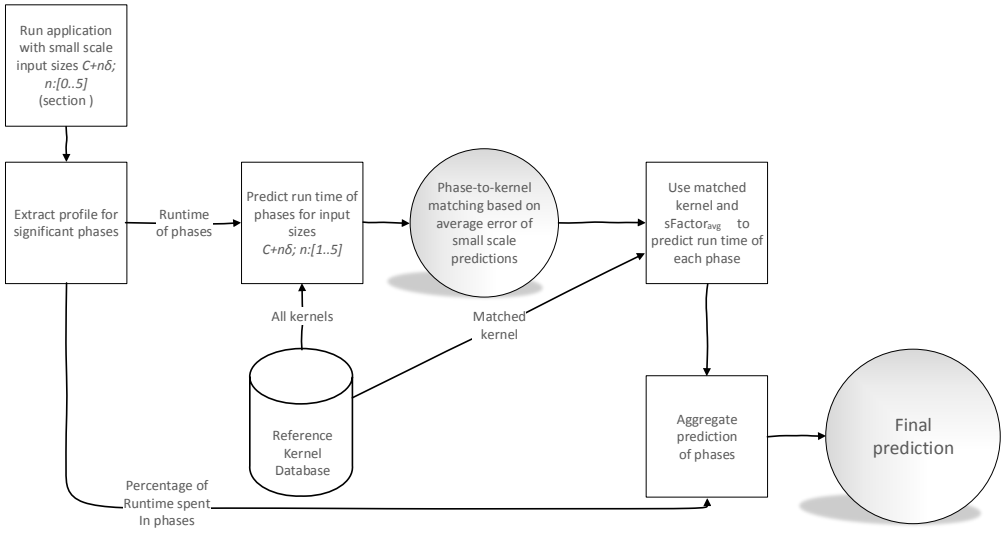

Figure 2. Prediction methodology

Table I
KERNEL IMPLEMENTATIONS

Benchmark suite	**Kernel**
NAS Parallel Benchmarks (NPB)	bt: Block Tri-diagonal solver fft: Discrete Fast Fourier Transform, all-to-all communication cg: Conjugate Gradient, irregular memory access and communication lu: LU decomposition is: Integer sorting mg: Multi-Grid solver, long- and short-distance communication, memory intensive sp: Scalar Penta-diagonal solver
HPC Challenge Benchmark (HPCC)	dgemm: Dense matrix multiplication ptrans: Parallel matrix transpose
Polyhedral Benchmark suite (Polybench)	symm: Symmetric matrix-multiply gemv: Matrix-Vector Multiplication and Matrix Addition trmm: Triangular matrix-multiply
Skeleton Particle-in-Cell (PIC) codes	charge: Particle deposit push: Particle push
Sparse matrix-vector mulitply (SpMV)	spmv: Sparse matrix-vector mulitply

A. Construction of a Kernel Database

To apply our prediction approach we first build a database with a set of 15 computational reference kernels which are representative of the most common computations performed on scientific computing workloads. These kernels were first introduced in [8] and later expanded in [9] and [10]. The main point of using these kernels lies on the fact that the underlying patterns they exhibit, independently of their particular implementation in terms of languages, algorithms or programming model, have persisted through generations and are believed to remain valid in the future[9]. These kernels have been used on various research topics, including code optimization, performance prediction, and power consumption, among others.

We used implementations of the kernels from several different benchmark suites, including the NAS Parallel Benchmarks (NPB)[11], the Skeleton Particle-in-Cell codes[12], the Polyhedral Benchmark suite (Polybench)[13], the HPCC Challenge benchmark suite (HPCC)[14] and the Sparse matrix-vector mulitply (SpMV) benchmark[12]. The list of the parallel kernel implementations we used to build our reference database is shown in Table I. The descriptive data we collect for each kernel is composed of a run time model and a count of different hardware events per kernel.

In order to build the execution time model of the benchmarks, we executed each one of them with different input

sizes and number of processes. For some benchmarks, such as the ones included in the NPB suite, there are predefined sizes that can be specified at compilation time. However, for other benchmarks we had to write scripts that automated the modification of the default input size on their source code. All benchmarks we used were written either in C or Fortran. In addition, some benchmarks such as FT have some restrictions on the valid input sizes that can be used. The number of different input sizes we used for each benchmark ranged from 3 to 10 depending on the restrictions of the benchmarks and the resources of our experimental set up. The input sizes we used gave us enough data to build the run time model of each benchmark. The benchmarks were instrumented with the help of the Tuning and Analysis Utilities (TAU)[15], a set of tools capable of gathering performance information through instrumentation of functions, methods, basic blocks and statements with very small overhead.

Unlike steps textitPhase Detection and Matching and textitPerformance Prediction that are discussed in sections III-B and III-C respectively, which have to be done for every prediction, this step has to be done only once and it will serve to all applications that will be predicted.

B. Phase Detection and Matching

In this step we use the execution of a set of small runs. We first instrument and execute with small problem sizes the application we want to predict. The instrumentation of the application is done using TAU. These small runs produce a profile log that serves two objectives: 1) automatically detecting and extracting the composing phases of the application, and 2) finding a set of small problem sizes that are good enough to characterize the run time of the application. The data collected in this step is the execution time of each phase.

1) Extracting phase information: Our approach uses the profiles obtained from the small runs to identify a small set of functions in which a significant part (70% or more) of the total run time is spent. The percentage of time spent on a function is derived from the information contained in the profile logs.

Typically, scientific workloads spend most of their time in functions that have loops in them. In all three applications we used in our experiments more than 70% of the total execution time was spent in a set of one or two functions. Each function of the set is considered a phase.

2) Finding a set of good small scale problem sizes: Finding the set of small scale input sizes that are used to match each phase of the application with the kernels in our database is crucial. First, we want the small set of input sizes to provide a good characterization of the general execution of the application, and second, at the same time we want them to be as small as possible so that it does not take much time to finalize this step. To that end, we ask the user to provide a small input size σ that may be a good

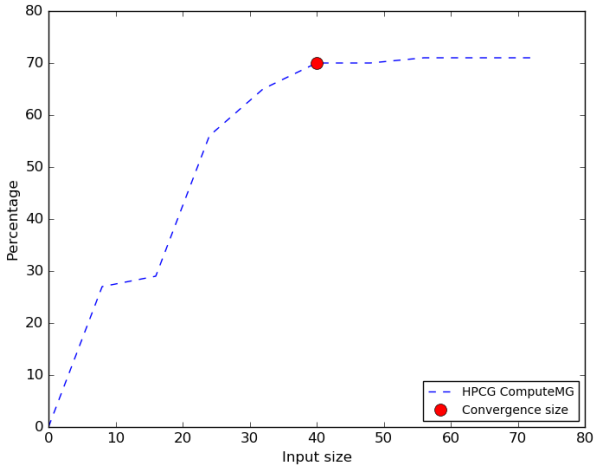

Figure 3. Percentages of phase *Compute MG* in HPCG for different input sizes

characterization of the overall execution of the application. That is the starting point of our search for a good set of small scale input sizes.

We run the application with σ and then iteratively increase the input size by δ noting the percentage of the total execution time of each phase until a convergence input size C is reached. We define the convergence input size of an application as the smallest input size that makes the phases of an application take a similar percentage of the total execution time to $C+\delta$. Figure 3 shows the convergence input size for a phase of HPCG.

Once the convergence input size has been found, we run and record the execution time of the application with N more small input sizes: $C+\delta$,$C+2\delta$,...,$C+N\delta$ and use each one of the kernels in our reference database to predict the phase run time. For each set of N predictions we compute the absolute mean (*AbsAM*), the root mean squared error (*RMSE*), the coefficient of variations of the sFactor ($sFCV$) and the standard deviation (*StdDev*).

From the N small scale executions we found that the sFactor varies depending on the particular input size used as reference. The concept of sFactor was described in the related work section. If a phase takes *tSmall* seconds to run when an input size *nSmall* is specified, and a kernel takes *tSmall* seconds to finish when an input size *nKernel* is specified, then the sFactor is defined as

$$sFactor = \frac{nSmall}{nKernel} \quad (1)$$

In our approach the value of the average sFactor $avsF$ is used as part of the final prediction for each phase. We also use an extrapolation of the sFactor for the target size

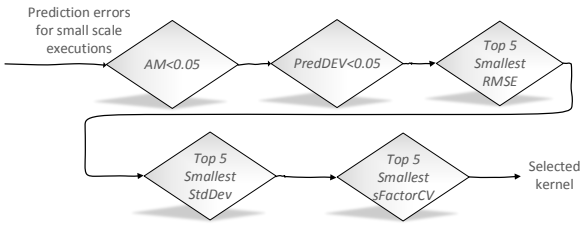

Figure 4. Kernel selection rule

and compute the deviation (*predDev*) of the extrapolation compared to the value of $avsF$. Using those metrics, we use the cascade of selection constraints presented in Figure 4 to select the matching kernel.

Our experiments showed that a value of δ=10 helped reduce the time needed to find the convergence input size and the set of small scale input sizes. Similarly, a value of *N*=5 was enough to get good matching candidates from the reference kernel database. In practice, the value of σ can be any number, including zero, since it does not affect the quality of our predictions. However, providing a good starting point reduces the search space of our technique, which leads to obtaining prediction results faster.

C. Performance Prediction

Our performance prediction is done in two steps. First, a run time prediction for each one of the identified phases is done, then an overall run time prediction for the entire application is performed.

1) Phase prediction: Our phase prediction technique is based on the approach introduced in [2] After having the phase-to-kernel matches, our prediction uses the performance model f of the matching kernel and the convergence size *C* in order to predict the run time t_P of a phase for a large input size *S*.

First, since a problem size *C* may correspond to a different kernel size *K*, the ratio between *C* and *K* is computed as

$$sFactor = \frac{C}{K} \tag{2}$$

The predicted run time of a phase P is

$$t_P = f(\frac{S}{sFactor}) \tag{3}$$

2) Overall prediction: Once we have matched a phase to a kernel from our database, we use the matched kernel's run time model in order to predict the run time of the phase with a large size. Moreover, based on the small scale runs we do a linear extrapolation to obtain the percentage of time that will be spent in the phase when run with a large input size *S*. Assuming that the estimated percentage of the total

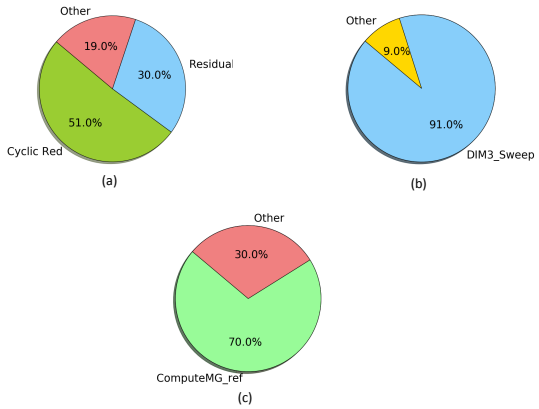

Figure 5. Time distribution for a) SMG2000, b) SNAP, and c) HPCG at convergence input size

execution time spent in phases P_1, P_2, P_3,...,P_h is r_{P1}, r_{P2}, r_{P3},...,r_{Ph}, respectively, and the predicted time for phases P_1, P_2, P_3,...,P_h is t_{P1}, t_{P2}, t_{P3},...,t_{Ph}, respectively, then our prediction for the entire application is:

$$T = \frac{\sum_{i=1}^{h} t_{Pi}}{\sum_{i=1}^{h} r_{Pi}} \tag{4}$$

Our results are in the range of 1% to 15%, indicating that matching and predicting a significant part of an application is enough to obtain an accurate prediction for the run time of the entire application.

IV. Experimental results

In order to evaluate the quality of predictions obtained with our methodology, the reference kernel database and the target applications were run in a cluster of four machines. Table II shows the characteristics of the machines. Our set of target applications is composed of SMG2000[4], SNAP[5], and HPCG[6]. SMG2000 is a semicoarsening multigrid solver for linear systems, SNAP is a proxy application that models the performance of a modern discrete ordinates neutral particle transport, and HPCG is an implementation of a preconditioned Conjugate Gradient method with a local symmetric Gauss-Seidel preconditioner and a three-level multigrid. Figure 5 shows the percentage of time consumed by each phase we used for the predictions when run with the convergence input size.

Our profiling was done using TAU with the TAU_PROFILE option active. Other options such as TAU_TRACING were deactivated since we did not require detailed execution traces. With this configuration, the overhead introduced by the profiler is less than 1%. We test our methodology with two different number of processes: 8 and 40. Since SNAP and HPCG have restrictions on the

number of processes that can be used, they were run with 32 processes instead.

A. 8 processes executions

1) SMG2000: Our profiling of the SMG2000 application reveals that two phases consume most of the time of the application: *Cyclic reduction* and *Residual*. Figure 5 shows that *residual* consumes around 48% whereas *cyclic reduction* takes around 30% of the application, thus contributing to around 78% of the total run time. Moreover, our approach finds that 80 is the convergence size for this application. Therefore, SMG2000 is run with sizes 80 to 130 using steps of δ=10. The time required to run the small sizes is around 1 minute.

After running the application with the set of small scale sizes, our selection rule is applied and *symm* is selected as the best matching reference kernel for the *cyclic reduction* phase. Table IV shows the values obtained from the small scale executions that led us to select *symm*. Table III shows the values obtained for *symm* compared to the selection rules.

Once we have selected a matching kernel for the *cyclic reduction* phase of SMG2000, the prediction of the target size *S=180* is computed using equation 3 and the value of $avSF$. The prediction error obtained when kernel *symm* is selected is close to 1% as presented in Table V. All phase prediction errors are presented in Table VI.

Similarly, *sp* is selected as the best matching kernel from the reference database for the *residual* phase with selection values *AbsAM=0.030*, *PredDev=0.046*, *RMSE=0.041*, *StdDev=0.032* and *SF CV=0.017*. The prediction error for the *residual* phase is 5.4%.

The final step in our methodology uses linear regression to extrapolate the percentage of time consumed by each phase for the target input size. Figure 6 shows the percentage of time consumed by (a) the textitcyclic reduction and (b) *residual* phases for the small scale runs. The overall prediction for the entire application is then computed using

Table II
NODE CHARACTERISTICS

Component	Characteristics
Processors	2 x Intel Xeon X5650 @2.67GHz per node
Cores/Threads	12/24 per node
Cache	L1 192 KB I + 192KB D L2 1.5MB I+D L3 12MB I+D per processor
Memory	24GB RAM DDR3
Disk	800 GB - SATA II
Network	Gigabit Ethernet
Operating system	CentOS 6.5

Table III
SELECTION RULE VALUES FOR SYMM

Rule	Value
Abs AM <0.5	0.003
PredDev <0.5	0.053
Top 5 smallest RMSE	Top 1
Top 5 smallest StdDev	Top 1
Top 5 smallest sFactorCV	Top 1

equation 4 which gives an error in the prediction for the entire application of 9.1%.

2) SNAP: In SNAP, our profiling shows that most of the time of the application is spent in the *DIM3 sweep* phase, which consumes around 91% of the total time. The convergence size is 96. Our small scale input sizes in this case range from 96 to 140 with steps of 8. Similar to HPCG, with a δ=8 we complied with input size restrictions and obtained a difference between run times of adjacent small scale input sizes of more than 50%.

Our methodology matches *sp* from the reference kernels with the *DIM3 sweep* phase. Using *sp's* execution model, the prediction for the target size *S=320* obtains a prediction error of 9.6%. Figure 8 shows the percentage of time consumed by the phase for the small scale runs. The percentage obtained for *S=320* is 100% using linear regression. The overall prediction for the entire application is then computed using equation 4. Using our methodology the error in the

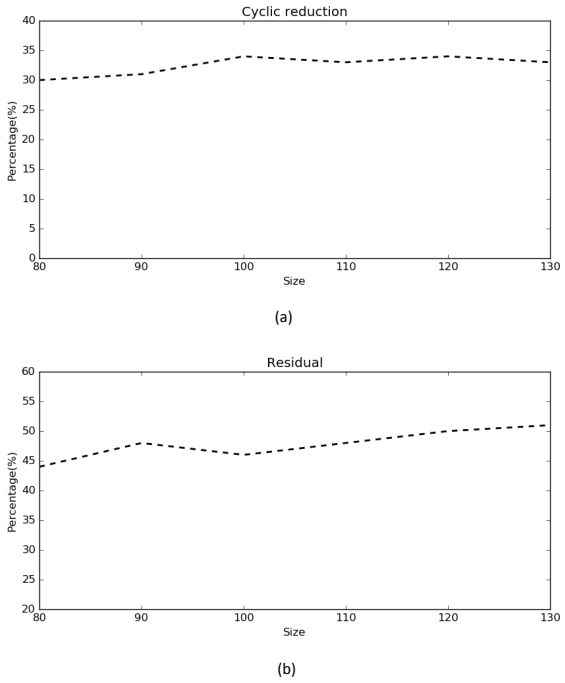

Figure 6. Percentage of time consumed in a) Cyclic reduction, and b) Residual

Table IV
SELECTION RULE STATISTICS EXTRACTED FROM SMALL SCALE RUNS FOR THE CYCLIC REDUCTION PHASE

	Abs AM	**RMSE**	**StdDev**	**sF CV**	**av sF**	**SF 180**	**predDev**
bebop	0.001	0.349	0.39	0.461	0.001	0	1.428
gemve	0.002	0.062	0.069	0.041	0.004	0.004	0.054
dgemm	0.002	0.151	0.169	0.092	0.047	0.034	0.28
symm	**0.003**	**0.037**	**0.041**	**0.022**	**0.088**	**0.083**	**0.053**
bt	0.003	0.09	0.101	0.04	2.118	1.883	0.111
trmm	0.003	0.111	0.124	0.056	0.049	0.041	0.167
mg	0.004	0.1	0.111	0.056	0.134	0.114	0.154
push	0.01	0.212	0.237	0.154	0.012	0.006	0.481
ptran	0.012	0.341	0.381	0.852	0.028	-0.032	2.149
gemm	0.014	0.09	0.1	0.03	0.078	0.072	0.082
sp	0.018	0.068	0.073	0.033	1.726	1.907	0.105
cg	0.049	0.092	0.087	0.065	2.933	2.555	0.129

Table V
CYCLIC REDUCTION PREDICTION ERRORS

kernel	**nSmall**	**nKernel**	**nLarge**	**nLarge/sFactorSize**	**Prediction**	**Real**	**Abs Error**
symm	**80**	**898**	**180**	**2039.683**	**46.752**	**47.123**	**0.008**
trmm	80	1504	180	3726.017	45.238	47.123	0.04
gemm	80	1010	180	2281.262	49.298	47.123	0.046
lu	80	54	180	106.32	43.56	47.123	0.076
bt	80	35	180	85.826	51.022	47.123	0.083
mg	80	537	180	1356.231	51.328	47.123	0.089
dgemm	80	1462	180	3863.385	39.831	47.123	0.155

prediction is 9.6%.

3) HPCG: After profiling the HPCG application, our approach detects that most of the time of the application is consumed by the *Compute MG_ref* phase, which consumes around 70% of the total run time of the application. The convergence size is 40 as shown in Figure 3. The application is then run with small scale input sizes ranging from 40 to 72 with steps of 8. With δ=8 we complied with some input size restrictions imposed by the application while still obtaining a difference between run times for adjacent small scale input sizes of more than 50% in all cases.

Our approach selects *symm* as the best matching kernel for the *Compute MG_ref* phase. After selecting *symm* as the matching kernel, the target size *S=144* is used to perform the prediction based on *symm's* run time model. The prediction error for the *Compute MG_ref* phase is 6.7%.

Finally, after having a prediction for the *Compute MG_ref* phase, we perform the prediction of the overall execution time of the application using equation 4. Figure 7 shows the percentage of time consumed by the matched phase for the small scale runs. After applying linear regression, the percentage of time spent in *Compute MG_ref* for the target size *S=144* is 72%. Thus, using equation 4 the error in the overall execution prediction for HPCG is 8%.

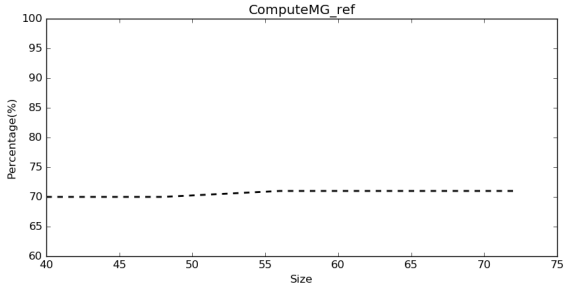

Figure 7. Percentage of time consumed in Compute_MG_ref

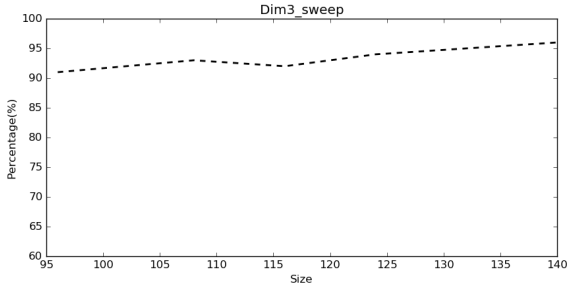

Figure 8. Percentage of time consumed in Dim3_sweep

B. 32 / 40 processes executions

In these experiments we run SMG2000 with 40 processes. SNAP and HPCG are run with 32 processes due to restrictions imposed by the applications on the number of processes that can be used. For each application we apply our prediction methodology as we did with 8 processes. In this case our methodology matched *cyclicred* with the *symm* kernel, *residual* with the *mg* kernel, *dim3_sweep* with

the *bt* kernel, and *computeMG_ref* with the *mg* kernel. The prediction results are presented in Table VII. It can be noted that our phase prediction errors range from 19% to 2%.

C. Discussion

In this work we propose a novel methodology to perform run time prediction of scientific workloads. However, our proposed technique has some limitations. First, our predictions are not valid across different clusters. That is, the predictions need to be done in the same machines where the workload will be executed. Second, our approach does not consider any possible interference created on the computation or the network systems. Our experiments were also run with workloads that stayed at comfortable distance from memory usage saturation.

V. Conclusion

In this work we introduced a novel approach to do performance prediction. Our approach is based on empirical analysis of small scale executions, it is simple to use and does not require involved procedures such as static analysis or manual instrumentation. Instead it only requires a few small executions of the application. Moreover, we show that matching a significant portion of an application with one or more reference kernels is enough to predict the total run time of an application. Our experiments with three applications gave predictions errors in the range of 1% to 20%. In our future work, we plan to explore more features that can improve the accuracy of the predictions. We also plan to include interference factors into our methodology that can be applied to shared environments.

References

[1] S. Sodhi and J. Subhlok, "Skeleton based performance prediction on shared networks," in *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, Conference Proceedings, pp. 723–730.

[2] A. Jayakumar, P. Murali, and S. Vadhiyar, "Matching application signatures for performance predictions using a single execution," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, Conference Proceedings, pp. 1161–1170.

[3] A. Wong, D. Rexachs, and E. Luque, "Parallel application signature for performance analysis and prediction," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 2009–2019, 2015.

Table VI
Prediction errors for 8 processes executions

Application	Phase	Matching kernel	Prediction error
SMG2000	Cyclicred	symm	0.008
SMG2000	Residual	sp	0.054
SNAP	Dim3_sweep	sp	0.096
HPCG	ComputeMGref	bt	0.195

Table VII
Prediction errors for 8 processes executions

Application	Phase	Matching kernel	Prediction error
SMG2000	Cyclicred	symm	0.03
SMG2000	Residual	mg	0.19
SNAP	Dim3_sweep	bt	0.018
HPCG	ComputeMGref	mg	0.032

[4] Lawrence Livermore National Laboratory, "The smg2000 benchmark code," 2015. [Online]. Available: https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/smg/

[5] N. National Energy Reserach Scientific Computer Center, "Snap," 2015. [Online]. Available: http://www.nersc.gov/research-and-development/apex/apex-benchmarks/snap/

[6] The High Performance Conjugate Gradients project, HPCG, "The high performance conjugate gradients (hpcg) benchmark," 2015. [Online]. Available: http://www.hpcg-benchmark.org/index.html

[7] L. T. Yang, M. Xiaosong, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, Conference Proceedings, pp. 40–40.

[8] E. L. Kaltofen, *Numerical and Symbolic Scientific Computing: Progress and Prospects*. Springer Vienna, 2012, pp. 95–104.

[9] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Report UCB/EECS-2006-183, December 18 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[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, Report UCB/EECS-2010-144, December 7 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-144.html

[11] NASA Advanced Supercomputing Division, NAS, "Nas parallel benchmarks," 2015. [Online]. Available: http://www.nas.nasa.gov/publications/npb.html

[12] Particle-in-Cell and Kinetic Simulation Software Center, PICKSC, "Particle-in-cell skeleton code," 2015. [Online]. Available: http://picksc.idre.ucla.edu/software/skeleton-code/

[13] The Ohio State University, OSU, "Polybench c: the polyhedral benchmark suite," 2015. [Online]. Available: http://web.cse.ohio-state.edu/ pouchet/software/polybench/

[14] HPC Challenge, HPCS, "Hpc challenge benchmark," 2015. [Online]. Available: http://icl.cs.utk.edu/hpcc/index.html

[15] Performance Research Lab, PRL, "Tau performance system," 2015. [Online]. Available: http://www.cs.uoregon.edu/research/tau/home.php