Performance Measurement of Novice HPC Programmers' Code

Rola Alameh, Nico Zazworka, Jeffrey K. Hollingsworth
University of Maryland
{rolasa, nico, hollings}@cs.umd.edu

Abstract

Performance is one of the key factors of improving productivity in High Performance Computing (HPC). In this paper we discuss current studies in the field of performance measurement of codes captured in classroom experiments for the High Productivity Computing Project (HPCS). We give two examples of measurements introducing two new hypotheses: spending more effort doesn't always result in improvement of performance for novices; the use of higher level MPI functions promises better performance for novices. We also present a tool - the Automated Performance Measurement System (APMS). APMS helps to partially automate the measurement of the performance of a set of parallel programs with several inputs. The design and implementation of the tool is flexible enough to allow other researchers to conduct similar studies.

1. Introduction

As part of the High Productivity Computing Systems (HPCS) project, the Development Time Working Group captured data from over 20 High Performance Computing (HPC) classes at several universities in the United States [1]. The question we address is: how productive are beginner programmers in HPC? Therefore we are recording different kinds of measurements such as effort data, defect rates, background information and work flow data to find evidence for hypotheses about how student programmers work and lean. The results should allow us to give advice to vendors, create tools to improve productivity and to come up with new learning concepts for HPC.

In previous studies, we have looked at many of these questions [1, 2]. However, the performance of student programs has not previously been measured. This measurement is essential in the field of HPC where the goal is to run programs faster using parallel computation.

The data we collected consists of information manually recorded by participants and data automatically captured by an instrumentation package[1]. This package captures all source code versions for every compile for every student and assignment.

The performance related questions we would like to answer include: what level of performance do novice programmers achieve with their programs, and what variables, if any, affect this performance? The results could support evidence for hypotheses such as: students who spend more effort on their assignments get better performance results. Alternatively, the results could point to common defects that result in performance bottlenecks. We could also compare the results to performance results of code that was written by professional programmers to investigate the gap (if it exists) between them.

Our study approach requires us to run all relevant student programs multiple times, on multiple input sets, on a varying number of processors. To complicate matters, we frequently want to perform longitudinal studies across similar assignments from different semesters. Unfortunately, while the assignments from different classes are similar, details such as command line arguments, and input file formats often change from assignment to assignment. We needed a tool that could automate both running the programs and eliding these minor differences. For this reason, we developed the Automated Performance Measurement System (APMS) which is a web-based tool that automates the process of running a large number of student codes. We describe our APMS tool in the next section. In the subsequent section, we present preliminary studies that illustrate what measurement results look like and what hypotheses they point to.

2. APMS

Capturing the characteristics of a program's performance requires running it on multiple sets of inputs.

[1] http://care.cs.umd.edu:8080/hpcs/software/umdinst

29th International Conference on Software Engineering Workshops(ICSEW'07)
0-7695-2830-9/07 $20.00 © 2007 IEEE

Typically in HPC these inputs include: the number of processors the program should be executed on, and problem-specific parameters such as grid sizes, number of iterations, or convergence thresholds Furthermore even running one set of inputs multiple times can be useful since run time fluctuates between runs due to external factors of the runtime environment. Therefore, measuring performance of codes by hand, on every combination of inputs, can be tedious, time consuming and error prone.

The goal of the APMS is the measurement of performance of parallel codes in an automated way. It is designed to make the job of measuring faster and easier and to provide all information the user would get if the program was executed manually. This includes process exit status, error and warning messages generated and output produced. This goal generates a set of functional requirements, which we will discuss in the following subsection, followed by the system design and the decisions made to achieve the desired functionality.

2.1. System functionality

APMS measures performance metrics for a set of parallel programs on a set of input parameters. The system reads source code from and writes results to the HPCS database, to enable the user later to aggregate performance data with other data that was captured during the classroom studies (e.g. workflow, effort and defect data). The system is fully automatic: once the criteria for programs to run are defined, the user doesn't need to interact with the system while the programs are executing. This feature is useful since although individual runs of programs are typically only a few seconds, a full performance study could require hundreds or thousands of individual program executions.

Alongside the performance measurement, APMS provides the user with the capability of deciding if program outputs are correct. Even if performance is the only variable being studied, correctness must also be verified since performance measurement results are only valid if the program produces the correct result. Judging the program correctness is not done automatically; instead the tool aims to visually display program outputs along side a known good output to facilitate the comparison of their contents, and provides an interface for the users to record their correctness decision.

The tool also allows the user to choose the performance metrics that should be measured. Performance metrics can vary depending on the purpose of measurement. If the goal is to measure overall speedup, the user would need to measure the total execution time. If the user needs more detailed information (like measures related to the machine architecture), lower-level metrics are needed, such as floating point instructions per second, memory and cache accesses.

2.2. System design

APMS was developed in Java using Java Server Pages technology. The system is divided into 4 modules (Figure 1).

2.2.1. Web interface. The system is accessed through a web browser. The user can select source files from the database, define input parameters and a set of possible values for each of them, then start the compilation, execution and performance measurement on a target cluster, and finally observe the results.

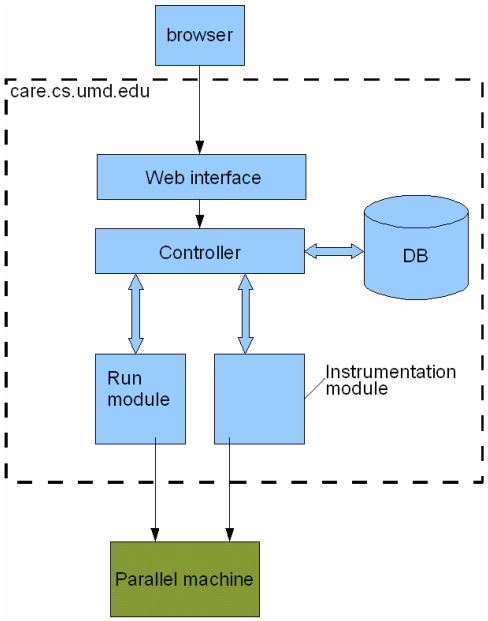

Figure 1: Architecture of APMS

One tricky part of developing such as system is how to specify the adaptation layer to manage mapping program parameters and input file formats between the formats used in different classes. In fact, it turned out that frequently the assignment for a class was under specified, and even within one class students did not use the same pattern to pass input parameters to their programs. For example, some classes expected input options from a specific file name, while, others read them from command line arguments and some had a mix of both techniques.

To handle these combinations we came up with a simple language that allows specifying parameters, their corresponding values, and the format for passing them to the program. We omit the details of this language due to space restrictions.

2.2.2. Instrumentation module. Our ultimate goal for the tool is to support measurement for most programming models in the HPC field. We started with the most widely used programming model, which also happens to be the model from which we have most student codes: MPI based on the C programming language. To collect performance measurement for MPI + C programs, we are using the PAPI library [3], which provides a consistent interface to the performance counter hardware found on most parallel machines.

The tool currently supports collecting the real time, CPU time (time spent in user mode), number of floating point instructions (FLOPS), and FLOPS per second. Additional PAPI events could easily be added.

Since the data collection component of the tool only requires functions to be called at the start and end of a parallel program, it would be easy to add support for additional languages and programming models. The data collection could still be provided via PAPI.

2.2.3. Run module. This module handles compiling and running the programs on the cluster.

The run module handles parallel environment issues. For instance, it handles launching each program through a parallel machine's job scheduler. It also allows the user to input an upper bound of how long the program should take to run. A job that goes beyond this time is automatically killed.

Once a program run is completed, all outputs generated (whether written to standard output or to files) are stored in the database and available to the tool user, who determines the correctness of the program.

2.2.4. Controller module. This module is the central piece that connects all other modules and handles all necessary file and data transfers between the different components.

3. Measurements and analysis

Our overall goal is to characterize the performance of student code focusing on: comparing the performance of code written by novices to the performance of code written by experts; studying the relationship between performance and variables like the effort spent to develop the program, the teaching methods used to train the novices or the experience level of the programmer in areas inside and outside the computer science domain (e.g. math, physics).

As a starting point for our study, we surveyed the range of performance that is achievable by students. For this purpose, we chose from our pool of assignments two problem types: the conjugate gradient problem and the game of life. Due to a delay in storing student submissions into the database, we manually inserted PAPI calls in each program and transferred the files to the target cluster before running the programs.

3.1. Conjugate gradient

For the conjugate gradient, we found 5 student submissions from the same class, written in C based on MPI. One of the 5 versions was incorrect. The remaining four were run on 1, 2, 4 and 8 processors with two sizes (8,000 and 8,000,000) of the input matrix.

Figures 2 and 3 show results from 3 students (s1, s2, s3) because the fourth student program ran for relatively too long (1.4s on 8 processors for a matrix of size of 8,000 x 8,000 and over 2 hours for a matrix of size 8,000,000 x 8,000,000). The plots show that students do achieve speedup (some more than others).

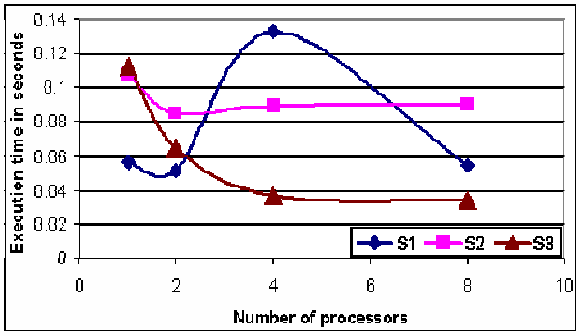

Figure 2: Time to run in seconds using matrix of size 8,000 x 8,000.

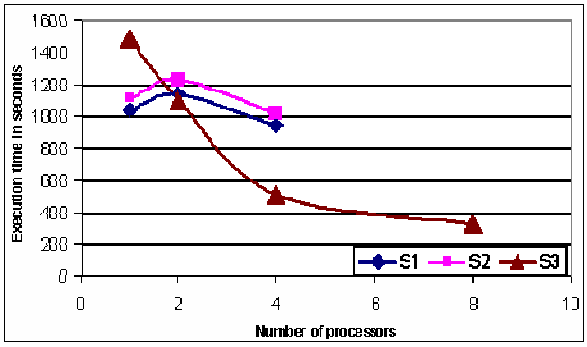

Figure 3: Time to run in seconds using matrix of size 8,000,000 x 8,000,000.

We see that the input size influences the speedup achieved as we scale the number of processors. For the matrix of 8,000, student 2 starts to achieve speedup from 2 processors, whereas with the matrix of 8,000,000, using 2 processors takes more time than 1 processor. The time to run then decreases again with 4 processors but increases with 8 (it took more than 2 hours to run, which is the reason why that point is not shown in Figure 3).

3.2. Game of life

Because of the small number of results, we chose another problem, the game of life. We found three running versions (S3 – S5) from one class and seven from another. These versions were tested using a grid size of 250 x 250 with a relatively high number of life forms. We ran the program for 1,000 generations on 1, 2, 4, 8 and 16 processors. Effort information is available for the 3 students from the first class, and for 2 students from the second class (S11, S12). Code from student S11 didn't run on one processor, but gave the correct output on more processors. The effort is measured in hours. Table 1 summarizes the results.

Table 1: Time to run in seconds for 1, 2 and 4, 8, 16 processors and effort for each student.

	Processors					Effort
	1	2	4	8	16	
S3	0.08	0.06	0.06	0.05	0.04	66
S4	0.07	0.07	0.08	0.19	0.04	47
S5	0.03	4.14	11.4	25.2	33.2	49
S6	0.73	0.37	0.19	0.13	0.08	NA
S7	0.34	0.17	0.09	0.08	0.04	NA
S8	0.34	0.25	0.18	0.39	0.29	NA
S9	1.18	0.67	0.56	0.36	0.25	NA
S10	0.26	0.14	0.09	0.21	0.04	NA
S11	NA	0.17	0.08	0.05	0.05	18
S12	0.27	0.22	0.19	0.91	0.75	14

We noticed that the program belonging to the person who spent the most effort on the assignment got a very good overall performance (it either runs faster or at the same speed as others). However, the person who spent the second most amount of effort on the assignment produced the slowest program (Table 2).

Another interesting fact can be noted from the Figures 4 and 5. Each of these plots shows the performance of programs of two students with similar effort. However, in Figure 4, the student who spent more effort on the assignment wrote a slower program. The situation is reversed in Figure 5. These results suggest that novices who spend more time programming do not necessarily produce faster programs. Of course, to support this hypothesis a larger number of data points is needed.

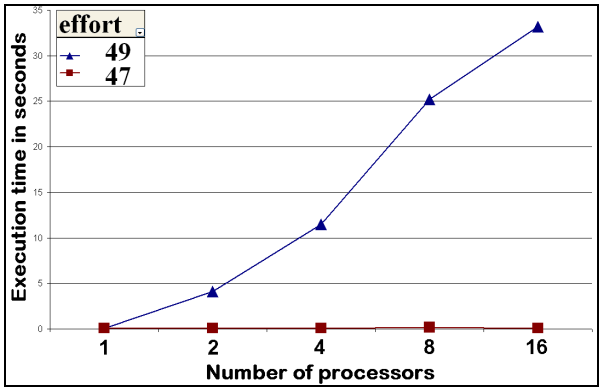

Figure 4: Time to run versus number of processors for students S3 and S2.

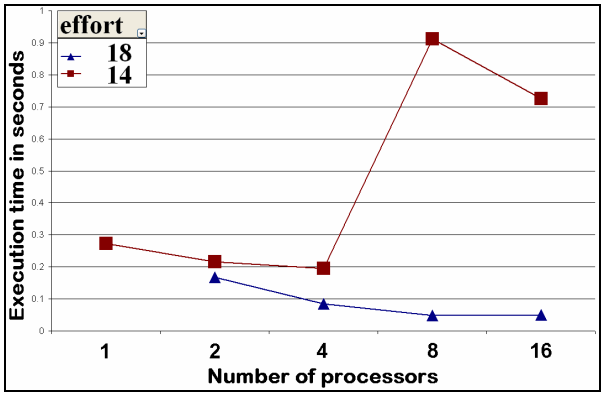

Figure 5: Time to run versus number of processors for students S9 and S10.

Besides the effort, we wondered whether the use of specific MPI functions by novices results in better performance. Therefore we categorized the functions in 3 groups: base functions (MPI_Send, MPI_Recv, MPI_Barrier) that would be enough to solve the problem, non-blocking functions (MPI_Ssend, MPI_Isend, MPI_Irecv, MPI_SendRecv, MPI_Wait) that are asynchronous versions of the low level ones and collective functions (MPI_Bcast, MPI_Reduce, MPI_Scatter, MPI_Gather) that move beyond point to point communication and allow groups of nodes to exchange data.

We ranked all subjects by the performance they achieved on all processors (by ranking each of them on each processor and calculating the mean of all rankings). As Table 2 shows, subjects that used higher level communication functions got better performance than students that used basic ones.

We believe that higher level communication concepts increase the overall performance of novice programs but we have to investigate more data to build

evidence and see if this is also the case for other parallel languages.

Table 2: Student ranking (fastest to slowest run time) and number of various types of MPI routines called.

Rank	ID	Base	Non-Blocking	Collective
1	S3	1	1	1
2	S4	2	1	1
3	S11	2	0	3
4	S10	2	2	0
5	S7	0	3	0
6	S6	2	0	0
7	S8	3	0	0
8	S12	2	1	0
9	S9	2	1	0
10	S5	3	0	0

4. Related Work

There are several other tools that provide automated systems to run programs and to capture output. For example, DART[4], provides a way to automate testing of a software packages on a variety of platforms and compilers. Also tools such as PerfTrack[5] and TAU's Performance Database[6] provide a way to record the performance evolution of a single application over time. However, our work differs in that our focus is gathering performance data and supporting running multiple different implementations of the same functionality.

5. Conclusion

We have developed a prototype of the APMS system. Currently the tool performs automatic instrumentation for MPI+C programs. It can handle programs written in a single source file or in multiple files, as long as they are stored in the database. Future plans for the tool focus on improving the user interface for flexibility, as well as adding functionality. The tool will be developed further to include functions for drawing graphs and diagrams based on the measurement results collected. Support for more programming languages and programming models is required. In addition, we plan to allow uploading source files that are not in the database.

In our research, we are investigating performance from a new perspective: focusing on improving the programmer, rather than the language or machine. Hardware and programming model, however, are not taken out of the equation. On the contrary, we expect that they will play an important role as environment variables. Our long term goal is to understand what works best under which circumstances.

Our preliminary results show that the relationship between effort and performance isn't straightforward as expected. Other variables, such as the type of functions used, might have a greater impact on performance. This suggests that the work flow of programming plays an important role. These hypotheses need further study based on a larger set of data points. We hope with time, APMS will evolve, making more complicated studies, involving a combination of variables, feasible.

6. Acknowledgments

This research was supported in part by Department of Energy contract DE-FG02-04ER25633, NSF grant EIA-0080206, and Air Force grant FA8750-05-1-0100. We would like to thank the HPCS development time working group members for their support.

7. References

[1] Hochstein L., J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, M. Zelkowitz, HPC Programmer Productivity: A Case Study of Novice HPC Programmers, Supercomputing 2005, Seattle, WA, November 2005.

[2] Carver, J., Asgari, S., Basili, V., Hochstein, L., Hollingsworth J. K., Shull, F., and Zelkowitz, M. Studying code development for high performance computing: the HPCS program. In Proceedings of the International Workshop on Software Engineering for High Performance Computing Systems Applications (SE-HPCS '04), May 2004

[3] Browne, S., J. Dongarra, N. Garner, G. Ho, and P. Mucci. "A Portable Programming Interface for Performance Evaluation on Modern Processors", *International Journal of High-Performance Computing Applications* 14:3, fall 2000, pp. 189-204.

[4] A. Memon, I. Banerjee, N. Hashmi, and A. Nagarajan "DART: A Framework for Regression Testing Nightly/daily Builds of GUI Applications", *Proceedings of the International Conference on Software Maintenance*, 2003.

[5] K. L. Karavanic, J. May, K. Mohror, B. Miller, K. Huck, R. Knapp, B. Pugh "Integrating Database Technology with Comparison-based Parallel Performance Diagnosis: The PerfTrack Performance Experiment Management Tool", Proceedings of SC'05, Nov. 2005, Seattle WA.

[6] K. Huck, A.D. Malony, R. Bell, A. Morris, "Design and Implementation of a Parallel Performance Data Management Framework," 2005 International Conference on Parallel Processing (ICPP), Oslo, Norway, 2005.