A Methodology for Scientific Benchmarking with Large-Scale Applications

Brian Armstrong Rudolf Eigenmann

School of Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana, USA

Abstract. The long-term goal of the project described in this paper is to facilitate the use of realistic, large-scale applications in performance evaluation projects. In prior work we have contributed to the creation of a new benchmark suite, called SPEChpc. Here, we introduce a methodology that will (1) help researchers understand the behavior with these large-scale applications, and (2) will guide the benchmark developers in characterizing a new application from a comprehensive and consistent set of perspectives. We will illustrate the proposed methodology by characterizing the *Seismic* benchmark of the SPEChpc suite.

1 Introduction

Importance of performance evaluation with realistic problems: Performance evaluation is an essential part of every scientific project. In computer systems research we generally use a suite of benchmark programs, for which we then quantify the performance difference made by a new concept or prototype. In order for our results to be of high value, the test suite needs to be *representative* and *shared*. *Representative* means that the suite reflects the important properties of realistic problems. It assures that the performance gains measured by the researcher will hold in real-life applications. *Shared* benchmarks are necessary so that we can compare our results with those obtained in related work. It requires that the same test suite be used by the different researchers and that the programs be executed under comparable conditions.

Today, the research community does not have available benchmark suites that are both representative and shared. This is especially true in high-performance computing, where realistic applications can be very large and require substantial machine resources to run. Furthermore, for typical research projects it is often not feasible to run such applications as part of a performance evaluation subproject because of the extensive effort involved.

Making large-scale benchmarks available for scientific study: The present paper addresses both of these issues. In an ongoing effort we have been developing a benchmark suite representative of high-performance computing applications. This effort is associated with the industry-oriented Standard Performance Evaluation Corporation, SPEC. While the primary intent of the industrial SPEC

members is market-oriented benchmarking, the mission of the academic participants is to make the suite available as a shared resource for scientific research. The primary issue in doing so is to facilitate its use for scientific performance evaluation projects, such that executing, gaining an understanding of, and modifying the code become feasible tasks. To this end, we introduce a methodology that will guide the benchmark users in asking the right questions and will give a recipe to the benchmark developer for providing the right information to the user.

Are large-scale application benchmarks necessary? While it is not our intent to replace the many existing programs suites [13, 2, 10, 23, 14, 22], there are numerous situations where large-scale benchmarks are necessary. For example, end-users are interested in the performance of their discipline-specific production application. Even if a researcher proves the value of a new innovation on a scaled-down benchmark, the end user may not appreciate the scientific argument about the equivalence of the scaled-down and realistic benchmarks. More formally, every reduction of a problem is an abstraction that makes assumptions that certain aspects of the application are more important than others. For example, one may reduce the number of time steps in an application, assuming that, if the same data is traversed in every step, the "benchmarking value" of every step is the same. However, our benchmarking objective may be to evaluate adaptive optimization techniques, which, for example, may recompile a part of the application after migrating to a new machine. In this case it is crucial to measure whether the adaptation overhead gets amortized over the remaining time steps of the computation. In general, we can say that, the higher the optimization capabilities of a system, the more important it becomes to use unabstracted, realistic applications for its evaluation.

Contributions: In this paper we will make the following specific contributions.

- We will describe a new benchmarking effort for parallel and high-performance computing. We will describe our goal of creating a research infrastructure that will make it feasible for diverse research groups to use large-scale applications in evaluating and guiding their research.
- We will introduce a methodology for benchmark characterization with the specific goal of facilitating the use of large-scale applications in research projects. This methodology will identify the basic questions commonly asked by the involved research groups. We will define the information that needs to be gathered in order to find answers to these questions.
- To illustrate this methodology we will characterize the seismic processing suite of the SPEChpc benchmarks extensively.

2 SPEChpc: A Performance Evaluation Infrastructure with Realistic Applications

History: the SPEC High-Performance Group The High-Performance Group of the Standard Performance Evaluation Corporation (SPEC/HPG) has

been developing large-scale benchmarks for high-performance compute platforms [12]. It grew out of a joint effort by the Perfect Benchmarks [4, 9] activity and the Standard Performance Evaluation Corporation. The effort also benefited from the participation of members of the *Parkbench* organization [16]. A first suite of the benchmarks is available under the name SPEChpc. SPEC/HPG is searching for applications that are industrially relevant, are being widely-used to solve realistic problems, can be distributed with a simple not-for-profit agreement, are available in a serial and a parallel code variant, and have a sponsor that is able to support the development of the code into a benchmark.

The effort of SPEC/HPG has not only involved the search for such applications, but also the definition of appropriate data sets, validation procedures, and run rules. For example, when generating SPEC-approved benchmark numbers, these rules allow for most source-level code modifications but disallow excessive tuning, such as assembly-level optimizations, and they require the full disclosure of all code modifications.

Table 1. SPEChpc benchmarks

The SPEChpc96 V1.1 benchmark suite			
Code	Area	Programming language	#lines
SPECchem	Molecular modeling	Fortran 77 and C / PVM,MPI,OpenMP	110,000
SPECseis	Seismic processing	Fortran 77 and C / PVM,MPI,OpenMP	20,000
SPECclimate	Weather modeling	Fortran 77 / PVM,MPI,OpenMP	50,000

Current Status of the SPEChpc Benchmarks The current release of the SPEChpc benchmarks includes three codes, SPECseis [19], SPECchem [21], and SPECclimate [15]. They are representative of applications used in the seismic industry, the chemical and pharmaceutical industries, and in atmospheric research, respectively. The codes are listed in Table 1. All benchmarks are available in a serial and a parallel code variant. The parallel codes are available in both message passing (PVM and MPI) and in parallel directive form (OpenMP.) SPEChpc results are published on SPEC's official web page (www.spec.org/hpg/.)

3 Towards a Methodology for Characterizing Large-Scale Application Benchmarks

Goals The goal of our methodology is twofold. The first objective is to give sufficient guidance to the benchmark user, so that it becomes feasible to use the large-scale applications in scientific performance evaluation projects. The second objective is to clearly define the information that the benchmark developer must provide in order to make a benchmark ready for the stated purpose.

Several methodologies for benchmarking and application characterization have been proposed. Benchmark suites typically come with guidelines on how to generate results that conform to the intended benchmarking philosophy. They may describe the metrics and the format of the result presentation. Examples are the SPEC benchmarks, which define run rules and result submission formats, and several PC benchmarks (e.g., [3]), which also define the requirements for reporting results. The Perfect Club [4, 9] proposed a methodology of benchmark diaries, in which the user reported applied code modifications and the resulting performance improvements. The Parkbench effort [16] defines several system levels and provides benchmarks specific to the respective levels. Many projects that study system components (such as cache studies, or compiler analyses) introduce an implicit methodology, which defines metrics and presentation formats that are most meaningful for the shown data. The same holds for the numerous application studies presented in the literature.

Our methodology contrasts with the reporting methods of the commercially-oriented benchmark suites in that we do not specify formal run requirements. This addresses the need of the research-oriented performance analysis projects to look at a benchmark from many different angles and allow modifications of the code. We present brief recommendations on how to reconcile this requirement with the need to have comparable results across many different research contributions in Section 3.1.

Compared to the individual methodologies used by computer systems and application characterization projects, our approach will combine such contributions into an integrated set of information categories that are reported about each benchmark. This information will help a new benchmark user gain an understanding of the overall application as well as the specific aspects that are relevant for the study at hand. It will also provide much of the characterization data (such as algorithmic decomposition, cache statistics, compiler analysis results) that need to be gathered as a basis for every performance evaluation project. Furthermore, the information categories define the data that needs to be collected by the benchmark developers.

In addition, in an ongoing project we are creating tools that make available this information on the Web (www.ece.purdue.edu/ParaMount/Benchmark). This paper presents a subset of the information and the rationales behind our effort. An important guideline for the definition of the proposed information categories is the availability of tools to collect them and the feasibility of creating new tools for this purpose.

Information Categories Tables 2, 3, and 4 list the basic information categories, the research questions addressed, and the target audience. The categories range from the basic problem analysis to details of the benchmark programs and their execution behavior.

Many of these categories include measurements of the benchmarks on actual machines or simulators. Complete information will include both overall application measurements and per-program-section analysis. Program sections can

Table 2. Information Categories of the Benchmark Characterization Methodology and Objectives. Characteristics gathered from the Code.

Information Category	Questions to be Answered and the Target Audience
Application description	A basic understanding of the problem being solved by a benchmark is important for all user classes. For many users it is important to understand how representative of a specific discipline a benchmark is. The latter also helps in selecting the benchmarks and in evaluating the quality of the results.
Program description	Describes programming languages, size, system requirements (cpu, memory, I/O), number of subroutines, and software libraries. The problem is described from a software engineering perspective.
Application component structure	Describes the coarse-grain structure of the application and opportunities for execution across parallel and/or heterogeneous, (globally) distributed systems.
Algorithmic structure	Algorithmic decomposition is important for the general understanding, but specifically aimed at algorithm research. This category includes measured improvements due to algorithmic changes.

be application components (for multi-component benchmarks), subroutines, or individual loops. Uniform naming schemes integrate the diverse information categories into an overall behavioral picture for each benchmark. Many of today's information gathering tools do not provide such uniform naming schemes. For example, instruction analysis tools tend to express their results in terms of program counters, call graphs use subroutine names, and compilers may report information on loops. For the user, it is of paramount importance to be able to compare information across the various categories. Therefore, the development of tools that agree on a uniform naming scheme is an important future goal.

Section 4 characterizes the SPECseis suite using the presented methodology. In doing so, we will also define the categories introduced in Tables 2, 3, and 4 more specifically. The scope of this paper would not allow the full characterization of even a single SPEChpc benchmark. We have begun to develop a facility that makes comprehensive information available on the Web (www.ece.purdue.edu/ParaMount/Benchmark).

3.1 Run Requirements

Flexibility in scientific performance evaluation is important. A conflicting requirement is equally important: results obtained by different researchers must be comparable. We recommend that every evaluation project that uses the SPEChpc benchmarks include the following information: (1) precise execution

Table 3. Information Categories of the Benchmark Characterization Methodology and Objectives. Characteristics gathered from run-time measurements.

Information Category	**Questions to be Answered and the Target Audience**
Measured performance	This addresses basic performance questions of all audiences. It indicated basic scalability, suitability of a code for specific machines, effectiveness of exploiting parallel processors, and sensitivity to changes in input parameters. It also states the baseline for improvements of algorithms, compilers, and architectures.
Analysis with respect to programming models	SPEChpc benchmarks include a message passing parallel and a shared-memory (directive-parallel) code variant. Message-passing profiles answer questions about (1) suitability of a distributed-memory architecture, its interconnections, and message libraries, (2) scalability beyond the measured range, (3) opportunities for communication optimizations, and (4) heterogeneity of the application and coupling of the components. Shared-memory analyses indicate (1) sources of performance loss, (2) data-sharing patterns, and (3) opportunities for shared data optimizations.
I/O characterization	Leads to a basic understanding of the application's I/O component. Shows scalability with respect to I/O, and sensitivity of the application with respect to the machine's I/O subsystems, (e.g., parallel disk organizations and partitioning schemes.)
Cache analysis	Shows the application's locality properties. Gives detailed insight into the data sharing behavior of the shared-memory application version and the data reuse of individual nodes in the message-passing version. The analysis follows the comprehensive cache characterization model introduced in [18].
Instruction analysis	Shows the number and type of instructions executed and the instruction profile (e.g., loads vs. stores.) Indicates detailed timing information, (e.g., pipeline stalls.) This information is the basis for the detailed performance analysis of individual code sections.
Program analysis	This is a large category of primary interest to compiler researchers and a basis for manual code improvements. It includes call-graph information, data use and access patterns, the control-flow structure, and results of compiler analyses, (e.g., applied and failed optimizations and program statistics.)

Table 4. Information Categories of the Benchmark Characterization Methodology and Objectives. Characteristics gathered from modeling the performance.

Information Category	Questions to be Answered and the Target Audience
Simulation Analysis	This is an open-ended category. Simulations can give insight into almost all parameters of an application and its potential execution behavior on a new or idealized machine.
Advanced model analysis	Various advanced performance analysis and prediction models have been proposed in the literature. They can give insight into a code's complex behavior, sources of performance loss, and upper performance limits.

environment (including machine parameters, operating system, compiler flags), (2) all code modifications of the released benchmark versions, (3) if only a subset of the benchmarks is used, the reason must be indicated.

4 A Case Study: Characterization of the SPECseis Suite

In this section we describe a benchmark application in detail, using the introduced methodology. The application is referred to as *Seismic*. As the benchmark of the SPEChpc suite it has the name SPECseis. The application has also been referred to as the *ARCO suite*, indicating its original site of development.

Application Description

The problem being solved: The application processes seismic signals that are emitted from a sound source that moves along a grid, reflected on the earths interior structures, and received by an array of receptors. The signals take the form of a sequence of *seismic traces*, which are processed by applying a sequence of data transformations. Table 5 gives an overview of these data transformation steps. The transformation steps are combined into four phases of the application, referred to as *Phase 1: Data Generation, Phase 2: Stacking of Data, Phase 3: Time Migration*, and *Phase 4: Depth migration*. The problem is described in more detail in [19].

Relevance as a benchmark: The application has been developed specifically as a benchmark for parallel computers that realistically reflects the computational methods used in seismic data processing [19]. The benchmark was created with the intention to avoid the work involved in porting and analyzing proprietary benchmark codes. It is in the public domain and has been included in several benchmark suites.

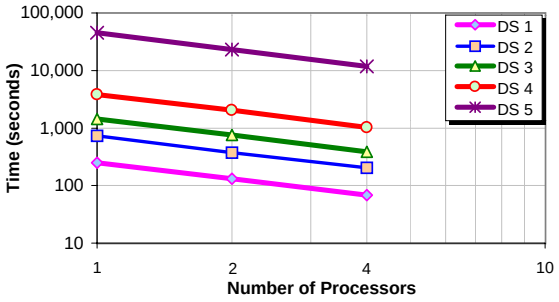

Fig. 1. The performance of *Seismic* on a 4-processor SUN Ultra Enterprise 4000 with five different datasets. The sizes of the datasets are: 16, 48, 96, 96, and 1,536 megabytes. Datasets 4 and 5 both consist of 96 megabytes but with different dimensions and precision. Dataset 5 has twice the number of samples per seismic trace and half the number of traces per group. DS 1 corresponds to the *test* data set of the SPECseis benchmark. DS 3 corresponds to *small* and DS 5 to *medium*.

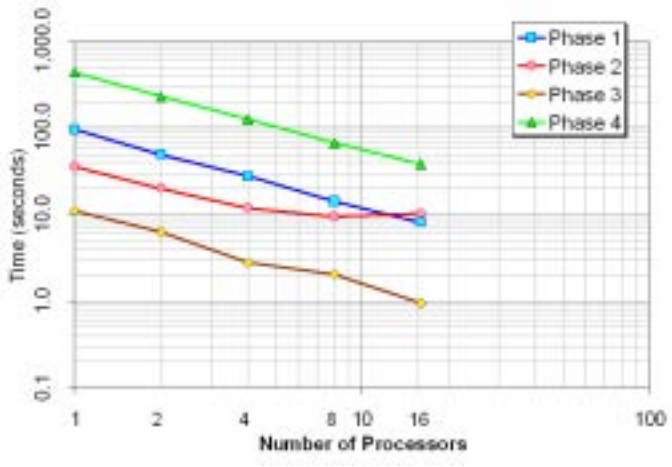

Fig. 2. The performance of a 16-processor SGI PowerChallenge Array with Dataset 3. The figure shows how each of the four seismic processing phases scale with respect to the number of processors. Phase 1 is highly parallel, the communication overhead in Phase 4 is hidden by its large computational load, Phase 3 performs only a fixed number (three) of large communications and file reads, and Phase 2 is characterized by a high communication-to-computation ratio, though it executes a fewer number of communications than Phase 4 does.

Table 5. A brief description of each seismic process which makes up the four processing phases of *Seismic*. Each phase performs all of its processing on every trace in its input file and stores the transformed traces in an output file.

Process	Description
Phase 1: Data Generation	
VSBF	Read velocity function and provide access routines
GEOM	Specify source/receiver coordinates
DGEN	Generate seismic data
FANF	Apply 2-D spatial filters to data via fourier transforms
DCON	Apply predictive deconvolution
NMOC	Apply normal move-out corrections
PFWR	Parallel write to output files
Phase 2: Stacking of Data	
PFRD	Parallel read of input files
DMOC	Apply residual move-out corrections
STAK	Sum input traces into zero offset section
PFWR	Parallel write to output files
Phase 3: Time Migration	
PFRD	Parallel read of input files
M3FK	3-D Fourier domain migration
PFWR	Parallel write to output files
Phase 4: Depth Migration	
VSBF	Data generation
PFRD	Parallel read of input files
MG3D	A 3-D, one-pass, finite difference migration
PFWR	Parallel write to output files

Program Description *Seismic* comes with ample documentation that describes the mechanisms of compiling and executing the program. A starting point to this information is the web site of the SPEC High-Performance Group [8].

The specific Seismic processing steps to be applied in a particular program run are specified via input parameters. An overall driver script defines these parameters for each of the four Seimic phases, writes the parameter file and invokes the application. The application loops through each trace, choosing the appropriate processing routines (listed in Table 5) in sequence.

The program includes 20,000 lines of Fortran and C code, and includes about 230 Fortran subroutines and 120 C routines. The computational parts are written in Fortran. The C routines perform file I/O, data partitioning, and message passing operations. The benchmark includes 5 data sets, ranging from a small set, for testing purposes, to a very large set. The currently largest set would run approximately one day on a 1 GFlops machine and use 100 GB of disk space. The code does not make use of any external libraries.

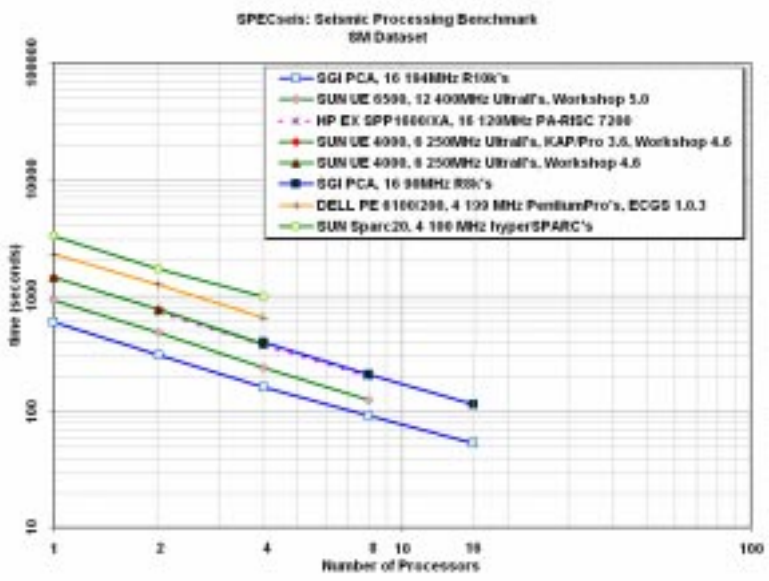

Fig. 3. Comparison of the performance from several architectures as the number of processors is varied. *Seismic* benchmarks shows good scalability on all these architectures.

Application Component Structure *Seismic* runs in four phases, as described earlier. All phases communicate through file I/O. In the current implementation, the phases need to run to completion before the next phase can start, except for Phases 3 and 4, which can run in parallel. This parallelism is practically insignificant since Phase 3 is very short.

More significant is the heterogeneous structure of the four phases. Phase 1 is highly parallel with little communication, while Phases 2 and 4 communicate more intensely. Phase 1 is suitable for a distributed-memory machine while the other two phases scale best on multiprocessor system with low communication latency. Phase 3 performs three communication operations, independent of the size of the input dataset.

Algorithmic Structure The low-level routines involved consist of complex-to-real and real-to-complex FFT transformations, tridiagonal solvers for first-order linear difference equations, which are executed over the two horizontal dimensions, and parallel transposing of 3-D arrays, which are distributed across processors.

Phases 1, 3 and 4 use complex-real FFT transformations. Phase 4 also performs the tridiagonal solvers for the finite difference equations. Phase 2 applies a dip move-out operator and performs common-midpoint stacking of the seismic

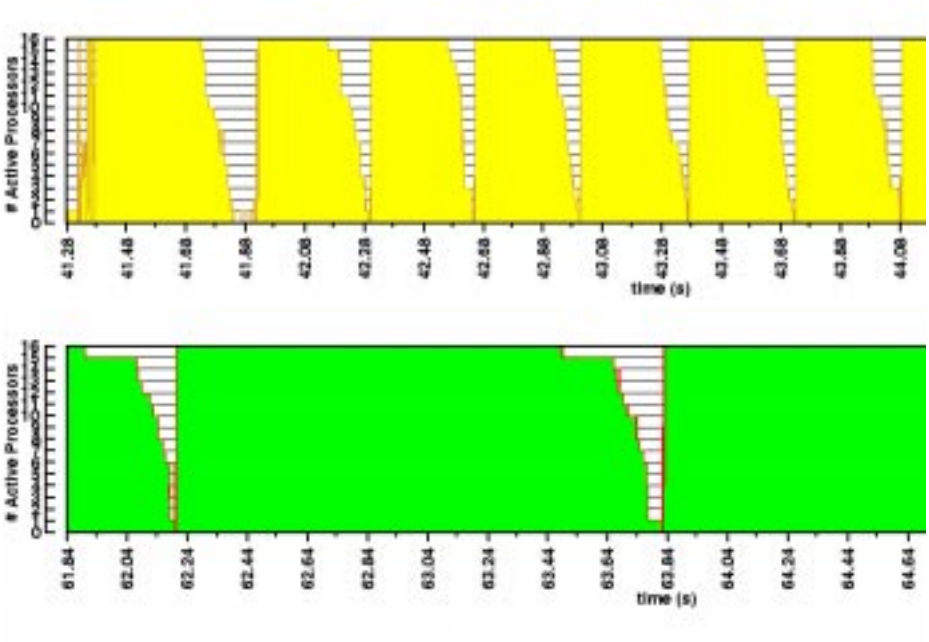

Fig. 4. Short segments (three seconds) of the communication profiles for Phase 2 and Phase 4 with Dataset 3 on an SGI PowerChallengeARRAY using sixteen processors. The graphs show how many processors are waiting for communication at any point in time. One can see that the communication-to-computation ratio in Phase 2 is higher than in Phase 4.

traces [19], which amplifies the true reflections and compresses the data for the following phases.

Measured Performance Figures 1, 2, and 3 give a picture of how *Seismic* scales with respect to increasing the data size, increasing the number of processors, and how it performs across a variety of architectures. Figure 1 shows the overall execution time of *Seismic* on a four-processor SUN Ultra Enterprise 4000 server using five different data sets. Figure 2 shows the performance on an SGI Origin2000 system, broken down into the four application phases. Figure 3 compares the execution obtained by different architectures.

Parallel Message-Passing Analysis Figure 4 gives a close-up view of the communication in Phase 2 and the later part of Phase 4. It provides a detailed view of how communication is performed. The total time spent in communication per processor on sixteen processors is given in Figure 5. The *communication*

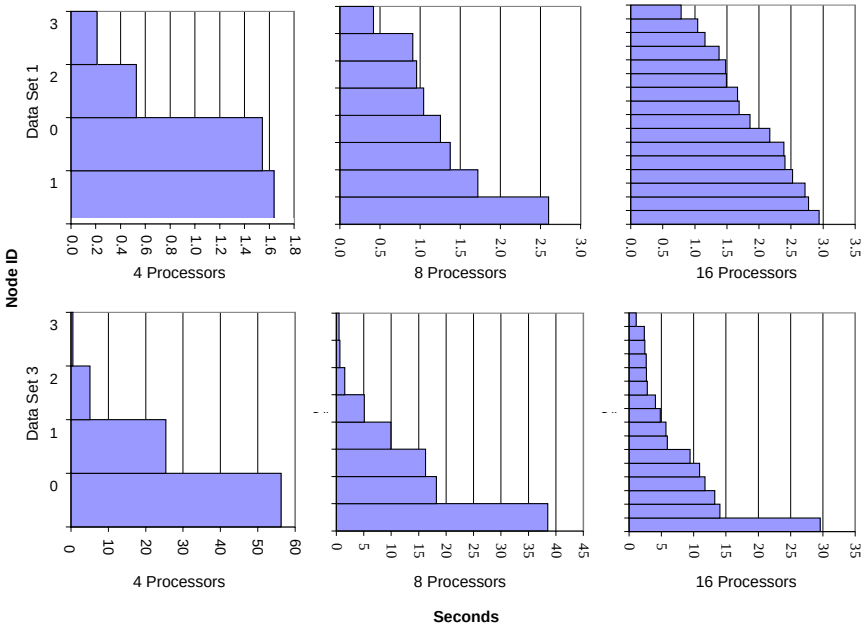

Fig. 5. The total time spent in communication per processor on four, eight, and sixteen processors MIPS R8000 processors with Datasets 1 (upper graph) and 3 (lower graph) on an SGI PowerChallengeARRAY. There is significant imbalance in the amount of communication performed by the different processors.

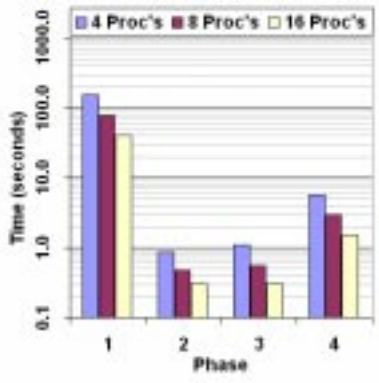

Fig. 6. The average time between communication points for each phase with Dataset 3 on four, eight, and sixteen MIPS R8000 processors of an SGI PowerChallengeARRAY. The communication in the seismic processing phases are regular. Phase 2 has the least amount of computation per communication point.

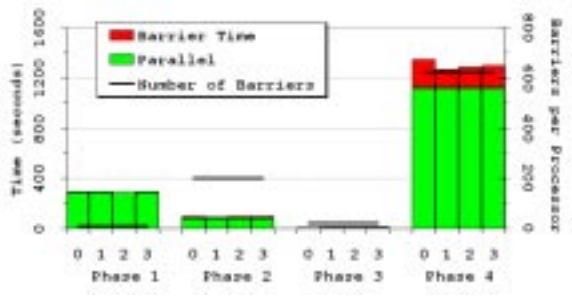

Fig. 7. A parallel execution of the shared-memory version showing the time spent in productive computations and in barrier synchronizations per thread for a 4-processor run. The number of barrier synchronizations is also shown.

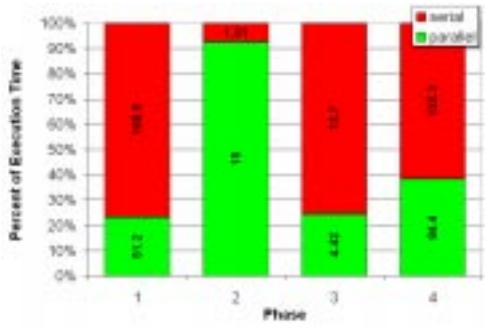

Fig. 8. The percentage of execution time (from a uni-processor run) covered by loops that Polaris determined are parallel. The total time spent in parallel and serial loops is given as a number on the bars.

granularity is given in Figure 6. It shows the average time between two communication points for specific runs of the application.

Parallel Shared-Memory Analysis We have created a shared-memory implementation of *Seismic*, written in the OpenMP parallel directive language. OpenMP offers both loop parallelism and SPMD programming style. For *Seismic* we have used the SPMD style. Except for the initialization section, the entire code is enclosed in one OpenMP *parallel region*. Within this region, each processor executes the same code. Data is statically partitioned. All data, except for data exchange buffers, is localized to the processors, using OpenMP *thread private* common block declarations. All processors exchange data in regular intervals. They do so by copying a data section into a shared buffer, performing a

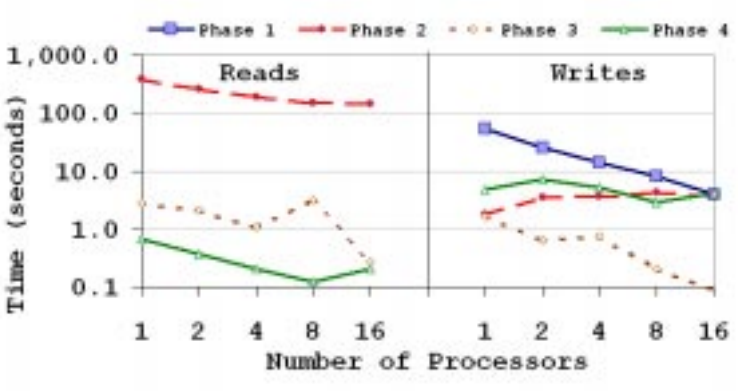

Fig. 9. The disk read/write time for the four seismic phases executed on an HP Exemplar S-Class machine with Dataset 5. Phase 2 is the only phase that reads the seismic data trace by trace instead of in groups. Phase 1 writes the trace dataset to disk, consisting of 1.5 GB of trace data for this data set. Phase 2 reads this file and writes a reduced file, (after performing common-midpoint stacking,) producing a 70 MB data file. Phases 3 and 4 read the reduced file written by Phase 2 and output two other files.

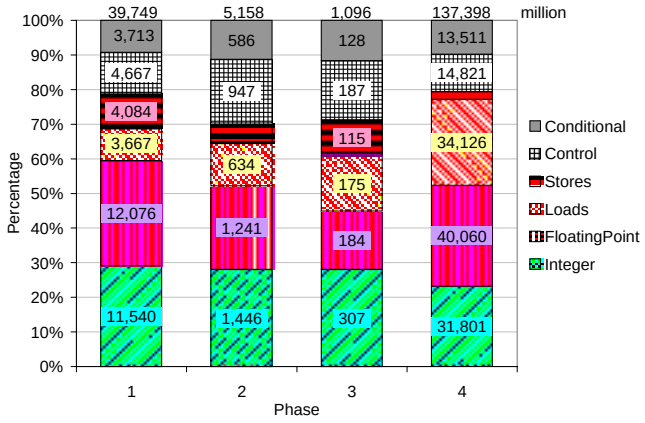

Fig. 10. The instructions executed for the four seismic phases from a serial run on a SUN Ultra Enterprise 4000 with Dataset 3. The instructions are broken down by category.

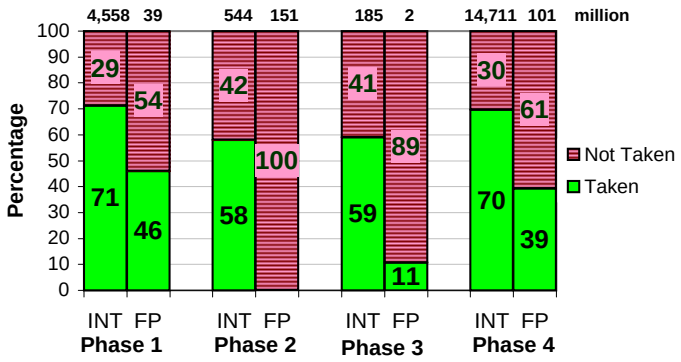

Fig. 11. The branch outcomes for the four seismic phases from a serial execution on a SUN Ultra Enterprise 4000 with Dataset 3. The outcomes for both the integer and the floating-point branches are given.

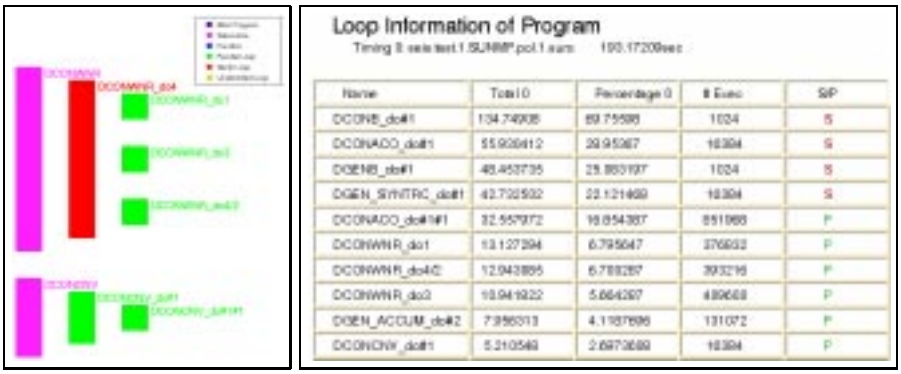

Fig. 12. A call graph for the *DCON* subroutines (part of Phase 1) and a segment of the corresponding loop table information. The left graph shows the nesting of the loops and the location of the subroutine calls within each subroutine. The loops are labeled and marked as serial or parallel. Two subroutines are shown: *DCONWNR*, which consists of a serial loop with three nested parallel loops, and *DCONCNV*, which consists of a parallel loop with a nested parallel loop. The right graph shows the compile-time loop information linked with some of the runtime information from a serial execution of the code on a SUN UltraEnterprise 4000 server. For each loop, the total time spent in the loop over the execution of the application (in both seconds and percentage of total execution time), the number of times the loop is executed, and whether the loop is serial or parallel are given in the columns.

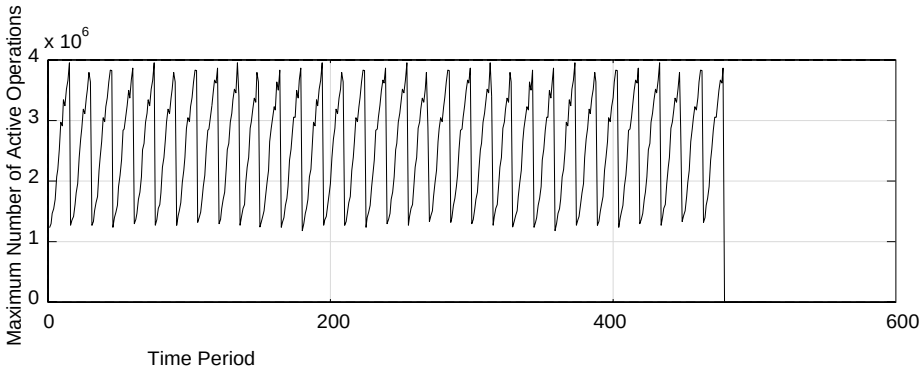

Fig. 13. Maximum parallelism of the MaxPar tool for Phase 2 with Dataset 3. The number of active operations is the number of operations that could be executed concurrently within a certain time segment. For this execution of the code, the maximum number of possible concurrent operations varies between one and four million.

barrier operation across all processors, and then copying from the shared buffer into the private data structures.

Figure 7 shows the breakdown of processor cycles into productive and wait time. The data is collected using the instrumented libraries provided with the KAI OpenMP compiler, *Guide*, which record wait time at barrier operations. The resulting graph shows that the wait times are very small, which is consistent with the good scalability of the code. In Figure 8 we show the amount of execution time covered by parallel loops. These loops were found to be parallel by the Polaris automatically parallelizing compiler [5].

I/O Characterization Figure 9 breaks down the disk I/O time per application phase into read and write operations. Phase 2 reads what Phase 1 writes. Both of these phases spend significantly more time in disk I/O than the other phases. But, the write time of Phases 2 and 4 remain near four seconds while the write time of Phase 1 decreases as the processing is spread across more processors

Cache Analysis In related work we have developed a methodology for cache analysis and a simulator that can capture the program behavior in the comprehensive terms defined by this methodology. For the presentation of results we refer to [18].

Instruction Analysis Figure 10 shows the number of instructions executed in each application phase and the breakdown into several categories. One fourth of the instructions executed in Phase 4 are loads. Though *Seismic* is a scientific application, it executes near the same number of integer instructions as floating-point instructions. Figure 11 shows branch frequencies for both integer and floating-point decisions.

Program Analysis Figure 12 shows data gathered from compile-time tools (Polaris) on code from Phase 1 of *Seismic*. The figure shows the call-graph of two subroutines in the *DCON* seismic process as well as a loop table view, which links the compiler information with timings from runtime instrumentation. The subroutines perform matrix inversion and convolution.

Simulation Analysis Figure 13 shows the results of the simulated execution of the second application phase on an ideal parallel machine. The simulator identifies the maximum degree of parallelism that is inherent an execution of the application [17, 20]. The figure shows the repetitive patterns in *Seismic* that follows the compute and data exchange phases. The maximum number of parallel operations found in this code is up to several million. Although this analysis does not show how realistic it is to exploit this degree of parallelism, it makes clear that the exploited parallelism, shown in the earlier figures, is orders of magnitude below the code's theoretical potential.

Advanced Methodologies The characterization information presented so far shows basic measurements as they are gathered by various tools. Combining and comparing these results in meaningful ways can lead to additional insights into a code's complex behavior. Several advanced performance analysis and prediction methods have been proposed in the literature with this objective. For example, The Hierarchical Performance Bound Model [7, 6] yield a hierarchy of bounds on achievable performance that identify specific causes of performance degradation and the sections of the code and data structures where they occur. The PTOPP model [11] defines overhead factors, such as globalization penalty, parallelization overhead, and spreading overhead, and recipes that guide the user in optimizing a parallel program. The Perfore methodology [1] extracts *Resource Usage Equations* from an applications, which can be used to characterize the scaling behavior of the application with respect to input data and architecture parameters.

The presentation of the results of these methods is beyond the scope of this paper. However, it is an important part of the characterization Web repository of our benchmark suite, which is being created in a related project.

5 Conclusions

Using the proposed methodology we have characterized the *Seismic* application of the SPEChpc benchmark suite. The gathered data is useful for understanding the performance behavior of the application from diverse perspectives.

We have found that the *Seismic* benchmark scales well with respect to the measured data sets and processor numbers, and it ports well between different architectures. However we have also found significant differences in individual phases of the code. These differences are with respect to the communication and the input/output behavior. Significant communication imbalances became

apparent, and the parallel disk I/O scales only in some parts of the application. However, on the measured machines, both the communication and the I/O is still dominated by the computation time, leading to overall good scalability. We have also analyzed the instruction profile of the code and have found that, despite the numerical character of the code, the number of integer instructions is relatively equal to the number of floating-point instructions.

Automatic program analysis for this code is not yet in the position to detect parallelism near the degree available in the manually parallelized code. We have also found that, although the parallel version of the application performs well on parallel machines, there is a theoretical maximum parallelism of several orders of magnitude beyond the currently exploited level.

The presented methodology offers guidelines both for the benchmark user, who needs to grasp the gist of a large application quickly, and for the benchmark developer who needs to know which code characteristics to presented to the user. In this way, the methodology represents a significant step towards our goal of making performance evaluation with realistic, large-scale applications a feasible task that is part of every computer systems research project.

Acknowledgment

This work was supported in part by DARPA contract #DABT63-95-C-0097 and NSF grants #9703180-CCR and #9872516-EIA. This work is not necessarily representative of the positions or policies of the U. S. Government.

References

1. Brian Armstrong and Rudolf Eigenmann. Performance forecasting: Towards a methodology for characterizing large computational applications. In *Proceedings of the International Conference on Parallel Processing*, pages 518–525, August 1998.
2. D. H. Bailey, E. Barszcz, L. Dagum, and H. Simon. NAS parallel benchmark results. In *Proc. Supercomputing '92*, pages 386–393. IEEE Computer Society Press, 1992.
3. BAPCo Benchmarks. Business application performance corporation. http://www.bapco.com/, 1999.
4. M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *Int'l. Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.
5. William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with Polaris. *IEEE Computer*, pages 78–82, December 1996.
6. Eric L. Boyd. *Performance Evaluation and Improvement of Parallel Application on High Performance Architectures.* PhD thesis, University of Michigan, Dept. of Electrical Eng. and Comput. Sci., 1995.

7. Eric L. Boyd, Gheith Abandah, Hsien-Hsin Lee, and Edward S. Davidson. Modeling computation and communication performance of parallel scientific applications: A case study of the IBM SP2. Technical Report CSE-TR-236-95, University of Michigan, Dept. of Electrical Eng. and Comput. Sci., May 1995.
8. Standard Performance Evaluation Corporation. SPEC high-performance group. http://www.spec.org/hpg/, 1996.
9. George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer Performance Evaluation and the Perfect Benchmarks. *Proceedings of ICS, Amsterdam, Netherlands*, pages 254–266, March 1990.
10. J. J. Dongarra. The Linpack benchmark: An explanation. In A. J. van der Steen, editor, *Evaluating Supercomputers*, pages 1–21. Chapman and Hall, London, 1990.
11. Rudolf Eigenmann. Toward a Methodology of Optimizing Programs for High-Performance Computers. *Conference Proceedings, ICS'93, Tokyo, Japan*, pages 27–36, July 20-22, 1993.
12. Rudolf Eigenmann and Siamak Hassanzadeh. Benchmarking with real industrial applications: The SPEC High-Performance Group. *IEEE Computational Science & Engineering*, 3(1):18–23, Spring 1996.
13. C. A. Addison et. al. The GENESIS distributed-memory benchmarks. In J. J. Dongarra and W. Gentzsch, editors, *Computer Benchmarks*, pages 257–271. Elsevier Science Publishers, Amsterdam, 1991.
14. Myron Ginsberg. Creating an automotive industry benchmark suite for assessing the effectiveness of high-performance computers. In *Proc. Ninth Int'l. Conf. on Vehicle Structural Mechanics and CAE*, pages 381–390. Society of Automotive Engineers, Inc. Warrendale, PA, 1995.
15. Philip L. Haagenson, Jimy Dudhia, David R. Stauffer, and Georg A. Grell. The Penn State/NCAR mesoscale model (MM5) source code documentation. http://box.mmm.ucar.edu/mm5/documents/mm5-code-doc.html, Nov. 22 1998.
16. R. W. Hockney and M. Berry (Editors). PARKBENCH report: Public international benchmarking for parallel computers. *Scientific Programming*, 3(2):101–146, 1994.
17. Seon-Wook Kim and Rudolf Eigenmann. *Max/P: detecting the maximum parallelism in a Fortran program.* Purdue University, School of Electrical and Computer, Engineering, High-Performance Computing Laboratory, 1997. Manual ECE-HPCLab-97201.
18. Seon Wook Kim, Michael Voss, and Rudolf Eigenmann. A methodology and a tool for cache characterization of loop-parallel programs. Technical report, HPCLAB, 1998.
19. C. C. Mosher and S. Hassanzadeh. ARCO seismic processing performance evaluation suite, user's guide. Technical report, ARCO, Plano, TX., 1993.
20. Paul Marx Petersen. *Evaluation of Programs and Parallelizing Compilers Using Dynamic Analysis Techniques.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., January 1993.
21. M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elber, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgonery. The general atomic and molecular electronics structure systems. *Journal of Computational Chemistry*, 14(11):1347–1363, 1993.
22. Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. Splash: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, 1992.
23. A. J. van der Steen. The benchmark of the EuroBen group. *Parallel Computing*, 17:1211–1221, 1991.