SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance

Vishal Aslot[1], Max Domeika[2], Rudolf Eigenmann[1], Greg Gaertner[3], Wesley B. Jones[4], and Bodo Parady[5]

[1] Purdue University,
[2] Intel Corp.,
[3] Compaq Computer Corp.,
[4] Silicon Graphics Inc.,
[5] Sun Microsystems

Abstract. We present a new benchmark suite for parallel computers. SPEComp targets mid-size parallel servers. It includes a number of science/engineering and data processing applications. Parallelism is expressed in the OpenMP API. The suite includes two data sets, Medium and Large, of approximately 1.6 and 4 GB in size. Our overview also describes the organization developing SPEComp, issues in creating OpenMP parallel benchmarks, the benchmarking methodology underlying SPEComp, and basic performance characteristics.

1 Introduction

Parallel program execution schemes have emerged as a general, widely-used computer systems technology, which is no longer reserved for just supercomputers and special purpose hardware sytems. Desktop and server platforms offer multithreaded execution modes in today's off-the shelf products. The presence of parallelism in mainstream computer systems necessitates development of adequate yardsticks for measuring and comparing such platforms in a fair manner.

Currently, no adequate yardsticks exist. Over the past decade, several computer benchmarks have taken aim at parallel machines. The SPLASH [7] benchmarks were used by the research community, but have not been updated recently to represent current computer applications. Similarly, the Perfect Benchmarks [2] used to measure high-performance computer systems at the beginning of the 90es. They included standard, sequential programs, which the benchmarker had to transform for execution on a parallel machine. The Parkbench effort [6] was an attempt to create a comprehensive parallel benchmark suite at several system levels. However, the effort is no longer ongoing. The SPEChpc suite [4,3] is a currently maintained benchmark for high-performance computer systems. It includes large-scale computational applications.

In contrast to these efforts, the goal of the present work is to provide a benchmark suite that

– is portable across mid-range parallel computer platforms,

- can be run with relative ease and moderate resources,
- represents modern parallel computer applications, and
- addresses scientific, industrial, and customer benchmarking needs.

Additional motivation for creating such a benchmark suite was the fact that the OpenMP API (www.openmp.org) has emerged as a de-facto standard for expressing parallel programs. OpenMP naturally offers itself for expressing the parallelism in a portable application suite. The initiative to create such a suite was made under the auspices of the Standard Performance Evaluation Corporation (SPEC, www.spec.org), which has been developing various benchmark suites over the last decade. The new suite is referred to at SPEComp, with the first release being SPEComp2001.

The SPEC organization includes three main groups, the Open Systems Group (OSG), best known for its recent SPEC CPU2000 benchmark, the Graphics Performance Characterization Group (GPC), and the High Performance Group (HPG). The SPEComp initiative was taken by the HPG group, which also develops and maintains the SPEChpc suite. Current members and affiliates of SPEC HPG are Compaq Computer Corp., Fujitsu America, Intel Corp., Sun Microsystems, Silicon Graphics Inc., Argonne National Lab, Leibniz-Rechenzentrum, National Cheng King University, NCSA/University of Illinois, Purdue University, Real World Computing Partnership, the University of Minnesota, Tsukuba Advanced Computing Center, and the University of Tennessee. In contrast to the SPEChpc suite, we wanted to create smaller, more easily portable and executable benchmarks, targeted at mid-range parallel computer systems. Because of the availability of and experience with the SPEC CPU2000 suite, we decided to start with these applications. Where feasible, we converted the codes to parallel form. With one exception, all applications in SPEComp2001 are derived from the CPU2000 suite. We also increased the data set significantly. The first release of the suite includes the Medium data set, which requires a computer system with 2GB of memory. An even larger data set is planned for a future release. Another important difference to the SPEC CPU2000 benchmarks is the run rules, which are discussed in section 2.2.

The remainder of the paper is organized as follows. Section 2 gives an overview of the benchmark applications. Section 3 presents issues we faced in developing the OpenMP benchmark suite. Section 4 discusses basic SPEComp performance characteristics.

2 Overview of the SPEComp Benchmarks

2.1 The SPEComp2001 Suite

SPEComp is fashioned after the SPEC CPU2000 benchmarks. Unlike the SPEC CPU2000 suite, which is split into integer and floating-point applications, SPEComp2001 is partitioned into a Medium and a Large data set.

The Medium data set is for moderate size SMP (Shared-memory MultiProcessor) systems of about 10 CPUs. The Large data set is oriented to systems with

Table 1. Overview of the SPEComp2001 Benchmarks

Benchmark name	Applications	Language	# lines
ammp	Chemistry/biology	C	13500
applu	Fluid dynamics/physics	Fortran	4000
apsi	Air pollution	Fortran	7500
art	Image Recognition/neural networks	C	1300
facerec	Face recognition	Fortran	2400
fma3d	Crash simulation	Fortran	60000
gafort	Genetic algorithm	Fortran	1500
galgel	Fluid dynamics	Fortran	15300
equake	Earthquake modeling	C	1500
mgrid	Multigrid solver	Fortran	500
swim	Shallow water modeling	Fortran	400
wupwise	Quantum chromodynamics	Fortran	2200

30 CPUs or more. The Medium data sets have a maximum memory requirement of 1.6 GB for a single CPU, and the Large data sets require up to 6 GB. Run times tend to be a bit long for people used to running SPEC CPU benchmarks. Single CPU times can exceed 10 hours for a single benchmark on a single state-of-the-art processor. Of the twelve SPEComp2001 applications, nine codes are written in Fortran and three are written in C. Table 1 shows basic features of the benchmarks. The suite includes several large, complex modeling and simulation programs of the type used in many engineering and research organizations. The application areas include chemistry, mechanical engineering, climate modeling, physics, image processing, and decision optimization.

2.2 SPEComp Benchmarking Methodology

The overall objective of the SPEComp benchmark suite is the same as that of most benchmarks: to provide the user community with a tool to perform objective series of tests. The test results serve as a common reference in the evaluation process of computer systems and their components.

SPEComp provides benchmarks in the form of source code, which are compiled according to a specific set of rules. It is expected that a tester can obtain a copy of the suite, install the hardware, compilers, and other software described in another tester's result disclosure, and reproduce the claimed performance (within a small range to allow for run-to-run variation).

We are aware of the importance of optimizations in producing the best system performance. We are also aware that it is sometimes hard to draw an exact line between legitimate optimizations that happen to benefit the SPEComp benchmarks and optimizations that specifically target these benchmarks. However, with the list below, we want to increase awareness among implementors and end users towards the issues related to unwanted benchmark-specific optimizations. Such optimizations would be incompatible with the goal of fair benchmarking.

The goals of the SPEComp suite are to provide a reliable measurement of SMP system performance, and also to provide benchmarks where new technology, pertinent to OpenMP performance, can be evaluated. For these reasons, SPEC allows limited source code modifications, even though it possibly compromises the objectivity of the benchmark results. We will maintain this objectivity by implementing a thorough review process.

To ensure that results are relevant to end-users, we expect that the hardware and software implementations used for running the SPEComp benchmarks adhere to the following conventions:

- Hardware and software used to run the SPEComp benchmarks must provide a suitable environment for running typical C and FORTRAN programs.
- Optimizations must generate correct code for a class of programs, where the class of programs must be larger than a single SPEComp benchmark or SPEComp benchmark suite. This also applies to assertion flags and source code modifications that may be used for peak measurements.
- Optimizations must improve performance for a class of programs where the class of programs must be larger than a single SPEComp benchmark or SPEComp benchmark suite.
- The vendor encourages the implementation for general use.
- The implementation is generally available, documented and supported by the providing vendor.

Benchmarking results may be submitted for a *base*, and optionally, a *peak* run. The base run is compiled and executed with a single set of compiler flags, and no source code modification is permitted. For the peak run, separate compiler flags may be used for each program, and limited source code modifications, restricted to the optimization of parallel performance, are permitted.

3 Issues in Developing an OpenMP Benchmark Suite

Several issues had to be resolved in developing the SPEComp suite. These issues include transforming the original codes to OpenMP, resolving portability problems, defining new data sets, creating self-validation code for each benchmark, and developing benchmark run tool.

3.1 Transforming Sequential Applications to OpenMP

A major effort in creating SPEComp was to convert the original, sequential programs into OpenMP parallel form. We give brief descriptions of the major transformation steps in creating OpenMP parallel programs.

To analyze parallelism in the given codes, we used a mix of application-level knowledge and program-level analysis. In several codes we started by manually inlining subroutine calls for easier interprocedural analysis. We then identified variables that are defined within potential parallel regions, and variables that are “live out” (e.g. formal parameters in a subroutine or function call, function return value, and `COMMON` blocks).

We declared as `PRIVATE` scalar variables that are defined in each loop iteration and that are not live out. There were a few instances of `LASTPRIVATE` variables.

We then identified any scalar and array reductions. Scalar reductions were placed on OpenMP `REDUCTION` clauses. Array reductions were transformed by hand (Note, that the OpenMP 2.0 specification supports array reductions).

ammp: ammp was by far the most difficult program to parallelize. In addition to directives, we added 16 calls to various OpenMP subroutines. There were only 13 pragmas added, but extensive revisions of the source base were necessary to accommodate parallelism. A few hundred lines of source code were moved or modified to get acceptable scalability. Key to getting scalability was the organization of the program using vector lists instead of linked lists.

applu: We added a total of 50 directives, with almost all simply a form of `PARALLEL` or `DO`. A `NOWAIT` clause was added for the terminator of one loop to improve scalability.

apsi: The main issue in transforming apsi was to privatize arrays that are sections of larger, shared arrays. We did this by declaring separate arrays in the subroutine scope. The size of the arrays is derived from the input parameters (MAX(nx,ny,nz)). Another way to do this is to `ALLOCATE` the arrays inside `OMP PARALLEL` but before `OMP DO`. Several simple induction variable also had to be substituted in this code. In performing these transformations we followed the parallelization scheme of the same code in [5].

art: For art, one difficulty encountered in defining a large data set with reasonable execution time involved the use of the *-objects* switch. When the *-objects N* switch is invoked (*N* is an integer specifying the number of objects to simulate), the neural network simulates the learning of several more objects than actually trained upon. art's memory requirements scale quadratically with the number of learned objects. Unfortunately, the execution time also scales quadratically due to the F2 layer retry on a mismatch. The art 2 algorithm essentially tests every learned object against the current inputs in a prioritized manner (more probable matches are tested first). To reduce the execution time, the number of F2 layer retries was limited to a small constant value. The reduction still allowed the real objects to be found.

facerec: In this code, pictures in an album are compared against a probe gallery. Both the generation of the album graphs, and the comparison of the album to the probe gallery are parallelized. The parallelization of the probe gallery comparison, which takes up almost all of the computation time, is done on the probe picture level. This is a "shared nothing" parallelization, which is achieved by copying the album graphs into a private array. This method performs well on systems with nonuniform memory access. There still remain many opportunities for parallelism inside the photo gallery comparison code. These opportunities could be exploited using nested parallelism. It would facilitate scaling to larger numbers of processors.

fma3d: fma3d is the largest and most complex code in the suite. There are over 60,000 lines of code, in which we added 127 lines of OpenMP directives. Nearly all directives were of the `PARALLEL` or `DO` variety. There were about a dozen `THREADPRIVATE` directives for a common block, ten reductions, a critical section, and a number of `NOWAIT` clauses. Still, locating the place for these directives was not too difficult and resulted in reasonable scalability.

gafort: We applied three major transformations. First, the "main Generation loop" was restructured so that it is private. It enabled parallelization of the "Shuffle" loop outside this major loop. Inlining of two subroutines expanded this main loop, reducing a large number of function calls. The second transformation concerns the "Random Number Generator". It was parallelized without changing the sequential algorithm. Care was taken to reduce false-sharing among the state variables. Third, the "Shuffle" loop was parallelized using OpenMP locks. The parallel shuffle algorithm differs from the original sequential algorithm. It can lead to different responses for different parallel executions. While this method leads to better scalability and is valid from the application point of view, it made the implementation of the benchmark validation procedure more difficult.

galgel: This code required a bit more attention to parallelization than most, because even some smaller loops in the LAPACK modules need OpenMP directives, and their need became apparent only when a significant number of CPUs were used. A total of 53 directives were added with most being simple `PARALLEL` or `DO` constructs. A total of three `NOWAIT` clauses were added to aid in scalability by permitting work in adjacent loops to be overlapped.

equake: One of the main loops in this code was parallelized at the cost of substantial additional memory allocation. It is an example of memory versus speed tradeoff. Three small loops in the main timestep loop were fused to create a larger loop body and reduce the Fork-Join overhead proportionally. The most time-consuming loop (function smvp) was parallel, but needed the transformation of array reductions. The initialization of the arrays was also parallelized.

mgrid: This code was parallelized using an automatic translator. The code is over 99% parallel. The only manual improvement was to avoid using multiple `OMP PARALLEL/END PARALLEL` constructs for consecutive parallel loops with no serial section in-between.

swim: In this code we added parallel directives to parallelize 8 loops, with a reduction directive needed for one loop. Swim achieves nearly ideal scaling for up to 32 CPUs.

wupwise: The transformation of this code to OpenMP was relatively straightforward. We first added directives to the matrix vector multiply routines for basic OpenMP parallel do in the outer loop. We then added OpenMP directives to the LAPACK routines (`dznrm2.f zaxpy.f zcopy.f zdotc.f zscal.f`). We also inserted a critical section to `dznrm2.f` for a scaling section. Reduction directives were needed for `zdotc.f` and `zscal.f`. After these transformations, wupwise achieved almost perfect scaling on some SMP systems.

3.2 Defining Data Sets and Running Time

An important part of defining a computer benchmark is the selection of appropriate data sets. The benchmark input data has to create an adequate load on the resources of the anticipated test machines, but must also reflect a realistic problem in the benchmark's application domain. In our work, these demands were not easy to reconcile. While we could identify input parameters of most codes that directly affect the execution time and working sets (e.g., time steps and array sizes), it was not always acceptable to modify these parameters individually. Instead, with the help of the benchmark authors, we developed input data sets that correspond to realistic application problems. Compared to the SPEC CPU2000 benchmarks, SPEComp includes significantly larger data sets. This was considered adequate, given the target machine class of parallel servers. The split into a Medium ($< 2GB$) and a Large ($< 8GB$) data set intends to accommodate different machine configurations, but also provides a benchmark that can test machines supporting a 64 bit address space.

3.3 Issues in Benchmark Self-Validation

An important part of a good benchmark is the self validation step. It indicates to the benchmarkers that they have not exploited overly aggressive compiler options, machine features, or code changes. In the process of creating SPEComp we had to resolve several validation issues, which went beyond those arising in sequential benchmarks. One issue is that numerical results tend to become less accurate when computing in parallel. This problem arises not only in the well-understood parallel reductions. We have also observed that advanced compiler optimizations may lead to expression reorderings that invalidate a benchmark run (i.e., the output exceeds the accuracy tolerance set by the benchmark developer) on larger numbers of processors. Another issue arose in benchmarks that use random number generators. The answer of such a program may depend on the number of processors used, making a validation by comparing with the output of a sequential run impossible. To address these issues, we found benchmark-specific solutions, such as using double-precision and identifying features in the program output that are invariant of the number of processors.

3.4 Benchmark Run Tools

Creating a valid benchmark result takes many more steps than just running a code. Procedural mistakes in selecting data sets, applying compilation and execution options, and validating the benchmark can lead to incorrect benchmark reports on otherwise correct programs. Tools that support and automate this process are an essential part of the SPEC benchmarks. The *run tools* for

SPEComp2001 were derived from the SPEC CPU2000 suite. Typically, benchmarkers modify only a small configuration file, in which their machine-dependent parameters are defined. The tools then allows one to make (compile and link) the benchmark, run it, and generate a report that is consistent with the run rules described in Section 2.2.

3.5 Portability Across Platforms

All major machine vendors have participated in the development of SPEComp2001. Achieving portability across all involved platforms was an important concern in the development process. The goal was to achieve functional portability as well as performance portability. Functional portability ensured that the makefiles and run tools worked properly on all systems and that the benchmarks ran and validated consistently. To achieve performance portability we accommodated several requests by individual participants to add small code modifications that take advantage of key features of their machines. It was important in these situations to tradeoff machine-specific issues against performance properties that hold generally. An example of such a performance feature is the allocation of lock variables in gafort. The allocation of locks was moved into a parallel region, which leads to better performance on systems that provide non-uniform memory access times.

4 Basic SPEComp Performance Characteristics

We present basic performance characteristics of the SPEComp2001 applications. Note, that these measurements do not represent any SPEC benchmark results. All official SPEC benchmark reports will be posted on SPEC's Web pages (www.spec.org/hpg/omp). The measurements were taken before the final SPEComp2001 suite was approved by SPEC. Minor modifications to the benchmarks are still expected before the final release. Hence, the given numbers represent performance trends only, indicating the runtimes and scalability that users of the benchmarks can expect. As an underlying machine we give a generic platform of which we know clock rate and number of processors.

Figure 1 shows measurements taken on 2, 4 and 8 processors of a 350MHz machine. The numbers indicate the time one has to expect for running the benchmark suite. They also show scalability trends of the suite. Table 2 shows the *parallel coverage* for each code, which is the percentage of serial execution time that is enclosed by a parallel region. This value is very high for all applications, meaning that these codes are thoroughly parallelized. Accordingly, the theoretical "speedup by Amdahl's Law" is near-perfect on 8 processors, as shown in the third column. Column four shows the "Fork-Join" overhead, which is computed as the $(t_{fj} \cdot N)/t_{overall}$, where $t_{overall}$ is the overall execution time of the code, N is the dynamic number of invocations of parallel regions, and t_{fj} is the Fork-Join overhead for a single parallel region. We have chosen $t_{fj} = 10 + p \cdot 2\ \mu s$, where p is the number of processors. This is a typical value we have observed.

Table 2. Characteristics of the parallel execution of SPEComp2001

Benchmark	Parallel Coverage	Amdahl's Speedup (8 CPU)	% Fork-Join Overhead (8 CPU)	Number of Parallel Sections
ammp	99.2	7.5	0.0008336	7
applu	99.9	7.9	0.0005485	22
apsi	99.9	7.9	0.0019043	24
art	99.5	7.7	0.0000037	3
equake	98.4	7.2	0.0146010	11
facerec	99.9	7.9	0.0000006	3/2[1]
fma3d	99.5	7.7	0.0052906	92/30[1]
gafort	99.9	7.9	0.0014555	6
galgel	96.8	6.5	4.7228800	32/29[1]
mgrid	99.9	7.9	0.1834500	12
swim	99.5	7.7	0.0041672	8
wupwise	99.8	7.9	0.0036620	6

[1] static sections / sections called at runtime

The table shows that the Fork-Join overhead is very small for all benchmarks, except for *galgel*. It indicates that, in all but one codes, the overhead associated with OpenMP constructs is not a factor limiting the scalability. Column five shows the static number of parallel regions for each code. A detailed analysis of the performance of SPEComp2001 can be found in [1].

5 Conclusions

We have presented a new benchmark suite for parallel computers, called SPEComp. We have briefly described the organization developing the suite as well as the development effort itself. Overall, the effort to turn the originally sequential benchmarks into OpenMP parallel codes was modest. All benchmarks are parallelized to a high degree, resulting in good scalability.

SPEComp is the first benchmark suite for modern, parallel servers that is portable across a wide range of platforms. The availability of OpenMP as a portable API was an important enabler. The first release of the suite, SPEComp2001, includes a Medium size data set, requiring a machine with 2 GB of memory. While the codes have been tuned to some degree, many further performance optimizations can be exploited. We expect that the availability of SPEComp will encourage its users to develop and report such optimizations. This will not only lead to improved future releases of the suite, it will also show the value of the new benchmarks as a catalyst for parallel computing technology.

Acknowledgement. We'd like to acknowledge the important contributions of the many authors of the individual benchmark applications. The creation of SPEComp2001 would not have been possible without them.

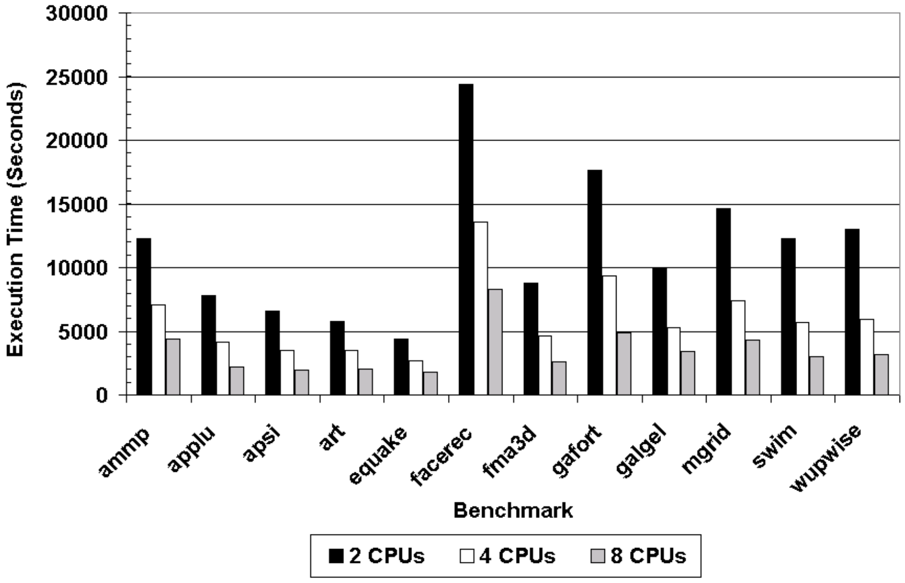

Fig. 1. Execution times of the SPEComp2001 benchmarks on 2,4, and 8 processors of a generic 350MHz machine.

References

1. Vishal Aslot. Performance Characterization of the SPEComp Benchmarks. Master's thesis, Purdue University, 2001.
2. M. Berry, et. al., The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *Int'l. Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.
3. Rudolf Eigenmann, Greg Gaertner, Faisal Saied, and Mark Straka. *Performance Evaluation and Benchmarking with Realistic Applications*, chapter SPEC HPG Benchmarks: Performance Evaluation with Large-Scale Science and Engineering Applications, pages 40–48. MIT Press, Cambridge, Mass., 2001.
4. 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.
5. Rudolf Eigenmann, Jay Hoeflinger, and David Padua. On the Automatic Parallelization of the Perfect Benchmarks. *IEEE Trans. Parallel Distributed Syst.*, 9(1):5–23, January 1998.
6. R. W. Hockney and M. Berry (Editors). PARKBENCH report: Public international benchmarking for parallel computers. *Scientific Programming*, 3(2):101–146, 1994.
7. Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, 1995.