Quantifying Differences between OpenMP and MPI Using a Large-Scale Application Suite*

Brian Armstrong Seon Wook Kim Rudolf Eigenmann

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285

Abstract. In this paper we provide quantitative information about the performance differences between the OpenMP and the MPI version of a large-scale application benchmark suite, SPECseis. We have gathered extensive performance data using hardware counters on a 4-processor Sun Enterprise system. For the presentation of this information we use a *Speedup Component Model*, which is able to precisely show the impact of various overheads on the program speedup. We have found that overall, the performance figures of both program versions match closely. However, our analysis also shows interesting differences in individual program phases and in overhead categories incurred. Our work gives initial answers to a largely unanswered research question: what are the sources of inefficiencies of OpenMP programs relative to other programming paradigms on large, realistic applications. Our results indicate that the OpenMP and MPI models are basically performance-equivalent on shared-memory architectures. However, we also found interesting differences in behavioral details, such as the number of instructions executed, and the incurred memory latencies and processor stalls.

1 Introduction

1.1 Motivation

Programs that exhibit significant amounts of data parallelism can be written using explicit message-passing commands or shared-memory directives. The message passing interface (MPI) is already a well-established standard. OpenMP directives have emerged as a new standard for expressing shared-memory programs. When we choose one of these two methodologies, the following questions arise:

– Which is the preferable programming model, shared-memory or message-passing programming on shared-memory multiprocessor systems? Can we replace message-passing programs with OpenMP without significant loss of speedup?

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

– Can both message-passing and shared-memory directives be used simultaneously? Can exploiting two levels of parallelism on a cluster of SMP's provide the best performance with large-scale applications?

To answer such questions we must be able to understand the sources of overheads incurred by real applications programmed using the two models.

In this paper, we deal with the first question using a large-scale application suite. We use a specific code suite representative of industrial seismic processing applications. The code is part of the SPEC High-Performance Group's (HPG) benchmark suite, SPEChpc96 [1]. The benchmark is referred to as SPECseis, or *Seis* for short. Parallelism in *Seis* is expressed at the outer-most level, i.e., at the level of the main program. This is the case in both the OpenMP and the MPI version. As a result, we can directly compare runtime performance statistics between the two versions of *Seis*.

We used a four-processor shared-memory computer for our experiments. We have used the machine's hardware counters to collect detailed statistics. To discuss this information we use the *Speedup Component Model*, recently introduced in [2] for shared memory programs. We have extended this model to account for communication overhead which occurs in message passing programs.

1.2 Related Work

Early experiments with a message passing and a shared-memory version of *Seis* were reported in [3]. Although the shared-memory version did not use OpenMP, this work described the equivalence of the two programming models for this application and machine class. The performance of two CFD applications was analyzed in [4]. Several efforts have converted benchmarks to OpenMP form. An example is the study of the NAS benchmarks [5, 6], which also compared the MPI and OpenMP performances with that of SGI's automatic parallelizing compiler.

Our work complements these projects where it provides performance data from the viewpoint of a large-scale application. In addition, we present a new model for analyzing the sources of inefficiencies of parallel programs. Our model allows us to identify specific overhead factors and their impact on the program's speedup in a quantitative manner.

2 Characteristics of SPECseis96

Seis 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. We use the 100 MB data set, corresponding to the *small* data set in SPEC's terminology.

The program processes a series of seismic signals that are emitted by a single source which moves along a 2-D array on the earth's surface. The signals are

reflected off of the earth's interior structures and are received by an array of receptors. The signals take the form of a set of *seismic traces*, which are processed by applying a sequence of data transformations. Table 1 gives an overview of these data transformation steps. The seismic transformation steps are combined into four separate seismic applications, referred to as four phases. They include *Phase 1: Data Generation, Phase 2: Stacking of Data, Phase 3: Frequency Domain Migration*, and *Phase 4: Finite-Difference Depth Migration*. The seismic application is described in more detail in [7].

Table 1. Seismic Process. A brief description of each seismic process which makes up the four processing phases of *Seis*. Each phase performs all of its processing on every seismic data trace in its input file and stores the transformed traces in an output file. We removed the seismic process called `RATE`, which performs benchmark measurements in the official SPEC benchmark version of *Seis*.

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.
`VRFY`	Compute average amplitude profile as a checksum.
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.
`VRFY`	Compute average amplitude profile as a checksum.
Phase 3: Fourier Domain Migration	
`PFRD`	Parallel read of input files.
`M3FK`	3-D Fourier domain migration.
`PFWR`	Parallel write to output files.
`VRFY`	Compute average amplitude profile as a checksum.
Phase 4: Finite-Difference 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.
`VRFY`	Compute average amplitude profile as a checksum.

The four phases transfer data through file I/O. In the current implementation, previous phases need to run to completion before the next phase can start,

except for Phases 3 and 4, which both migrate the stacked data, and therefore only depend on data generated in Phase 2. The execution times of the four phases on one processor of the Sun Ultra Enterprise 4000 system are:

Data Generation Phase 1	Data Stacking Phase 2	Time Migration Phase 3	Depth migration Phase 4	Total
272s	62.2s	7.1s	1,201s	1,542s

More significant is the heterogeneous structure of the four phases. Phase 1 is highly parallel with synchronization required only at the start and finish. Phases 2 and 4 communicate frequently throughout their execution. Phase 3 executes only three communications, independent of the size of the input data set, and is relatively short.

Figure 1 shows the number of instructions executed in each application phase and the breakdown into several categories using the SPIX tool [8]. The data was gathered from a serial run of *Seis*. One fourth of the instructions executed in Phase 4 are loads, contributing the main part of the memory system overhead, which will be described in Figure 4. Note that a smaller percentage of the instructions executed in Phase 3 are floating-point operations, which perform the core computational tasks of the application. Phase 3 exhibits startup overhead simply because it executes so quickly with very few computation steps.

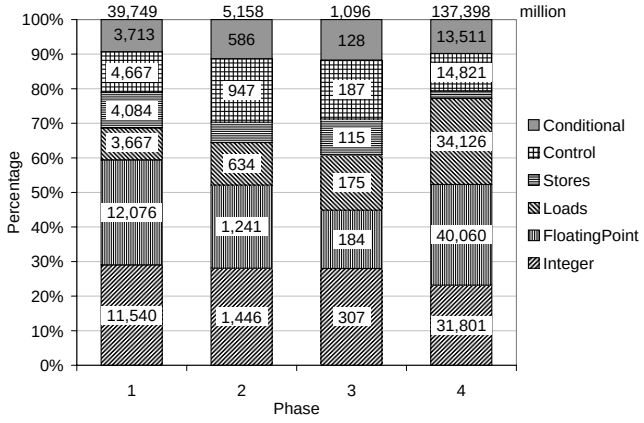
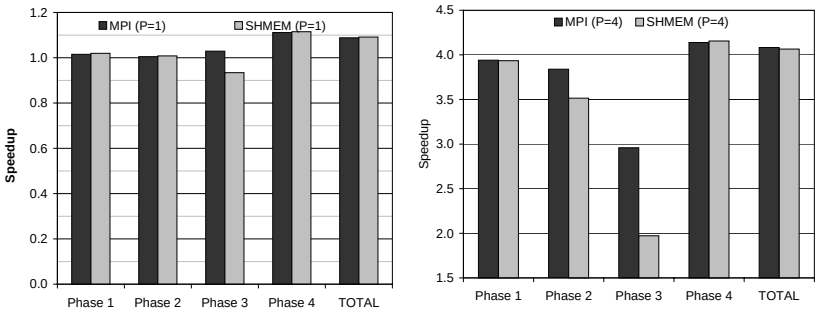


Fig. 1. The Ratio of Dynamic Instructions at Run-Time, Categorized by Type. Instructions executed for the four seismic phases from a serial run were recorded.

Figure 2 shows our overall speedup measurements of MPI and OpenMP versions with respect to the serial execution time. The parallel code variants execute nearly the same on one processor as the original serial code, indicating that negligible overhead is induced by adding parallelism. On four processors, the MPI code variant exhibits better speedups than the OpenMP variant. We will describe reasons in Section 4.



(a) 1 Processor. (b) 4 Processors.

Fig. 2. Speedups of the MPI and OpenMP Versions of *Seis*. Graph (a) shows the performance of each seismic phase as well as the total performance on one processor. Graph (b) shows the speedups on four processors. Speedups are with respect to the one-processor runs, measured on a Sun Enterprise 4000 system. Graph (a) shows that the parallel code variants run at high efficiency. In fact, parallelizing the code improves the one-processor execution of Phase-1 and Phase 4. Graph (b) shows that nearly ideal speedup is obtained, except with Phase 3.

3 Experiment Methodology

3.1 Speedup Component Model

To quantify and summarize the effects that the different compiling and programming schemes have on the code's performance, we will use the *speedup component model*, introduced in [2]. This model categorizes overhead factors into several main components: *memory stalls, processor stalls, code overhead, thread management*, and *communication overhead*. Table 2 lists the categories and their contributing factors. These model components are measured through hardware counters (`TICK` register) and timers on the Sun Enterprise 4000 system [9].

The speedup component model represents the overhead categories so that they fully account for the performance gap between measured and ideal speedup. For the specific model formulas we refer the reader to [2]. We have introduced the communication overhead category specifically for the present work to consider the type of communication used in *Seis*. The parallel processes exchange data at regular intervals in the form of all-to-all broadcasts. We define the communication overhead as the time that elapses from before the entire data exchange (of all processors with all processors) until it completes. Both the MPI and the OpenMP versions perform this data exchange in a similar manner. However the MPI version uses send/receive operations, whereas the OpenMP version uses explicit copy operations, as illustrated in Figure 3.

The MPI code uses blocking sends and receives, requiring processors to wait for the send to complete before the receive in order to swap data with another processor. The OpenMP code can take advantage of the shared-memory space

Table 2. Overhead Categories of the Speedup Component Model.

Overhead Category	Contributing Factors	Description	Measured with
Memory stalls	IC miss	Stall due to I-Cache miss.	HW Cntr
	Write stall	The store buffer cannot hold additional stores.	HW Cntr
	Read stall	An instruction in the execute stage depends on an earlier load that is not yet completed.	HW Cntr
	RAW load stall	A read needs to wait for a previously issued write to the same address.	HW Cntr
Processor stalls	Mispred. Stall	Stall caused by branch misprediction and recovery.	HW Cntr
	Float Dep. stall	An instruction needs to wait for the result of a floating point operation.	HW Cntr
Code overhead	Parallelization	Added code necessary for generating parallel code.	computed
	Code generation	More conservative compiler optimizations for parallel code.	computed
Thread management	Fork&join	Latencies due to creating and terminating parallel sections.	timers
	Load imbalance	Wait time at join points due to uneven workload distribution.	
Communication overhead	Load imbalance	Wait time at communication points.	timers
	Copy operations	Data movement between processors.	
	Synchronization	Overhead of synch. operations.	

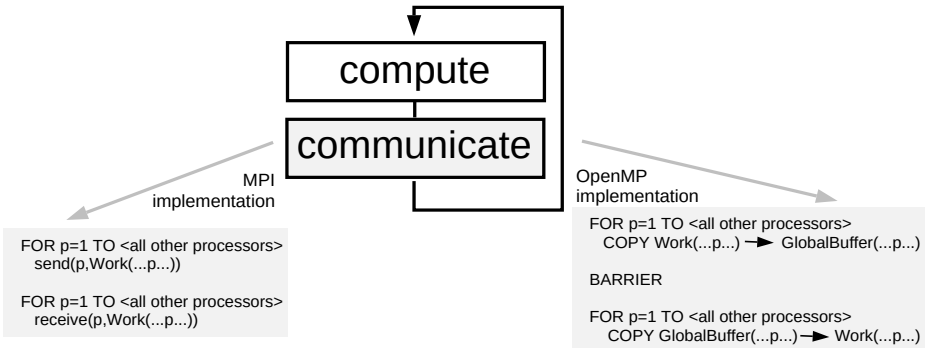


Fig. 3. Communication Scheme in *Seis* and its Implementation in MPI and OpenMP.

and have all processors copy their processed data into the shared-space, perform a barrier, and then copy from the shared-space.

3.2 Measurement Environment

We used a Sun Ultra Enterprise 4000 system with six 248 MHz UltraSPARC Version 9 processors, each with a 16 KB L1 data cache and 1 MB unified L2 cache using a bus-based protocol. To compile the MPI and serial versions of the code we use the Sun Workshop 5.0 compilers. The message-passing library we used is the `MPICH 1.2` implementation of MPI, configured for a Sun shared-memory machine. The shared-memory version of *Seis* was compiled using the KAP/Pro compilers (`guidef77` and `guidec`) on top of the Sun Workshop 5.0 compilers. The flags used to compile the three different versions of *Seis* were `-fast -O5 -xtarget=ultra2 -xcache=16/32/1:1024/64/1`.

We used the Sun Performance Monitor library package and would make 14 runs of the application, gathering hardware counts of memory stalls, instruction counts, etc. Using these measurements we could describe the overheads seen in the performance of the serial code and difference between observed and ideal speedup for the parallel implementations of *Seis*. The standard deviation for all these runs was negligible, except in one case mentioned in our analysis.

4 Performance Comparison between OpenMP and MPI

In this section we first inspect the overheads of the 1-processor executions of the serial as well as the parallel program variants. Next, we present the performance of the parallel program executions and discuss the change in overhead factors.

4.1 Overheads of the Single-Processor Executions

Figure 4 shows the breakdown of the total execution time into the measured overheads. "OTHER" captures all processor cycles not spent in measured stalls. This category includes all productive compute cycles such as instruction and data cache hits, and instruction decoding and execution without stalls. It also includes stalls due to I/O operations. However we have found this to be negligible. For all four phases, the figure compares the execution overheads of the original serial code with those of the parallel code running on only one processor. The difference in overheads between the serial and single-processor parallel executions indicate performance degradations due to the conversion of the original code to parallel form. Indeed, in all but the Fourier Migration code (Phase 3) the parallel codes incur more floating-point dependence stalls than the serial code. This change is unexpected because the parallel versions use the same code generator that the serial version uses, except that they link with the MPI libraries or transform the OpenMP directives in the main program to subroutines with thread calls, respectively.

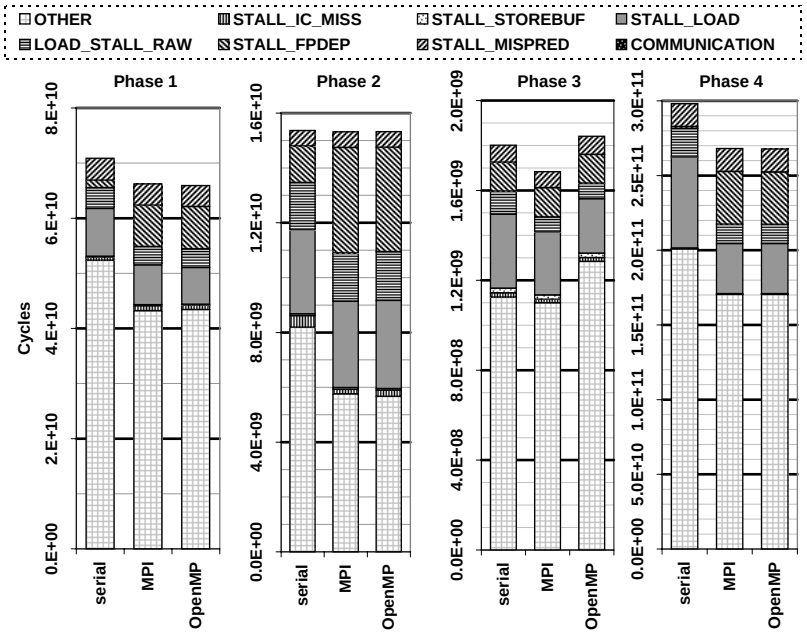


Fig. 4. Overheads for One-Processor Runs. The graphs show the overheads found in the four phases of *Seis* for the serial run and the parallel runs on one processor. The parallel versions of the code cause more of the latencies to be within the FP units than the serial code does. Also, notice that the loads in the Finite-Difference Migration (Phase 4) cause less stalls in the parallel versions than in the serial code. In general, the latencies accrues by the two parallel version exhibit very similar characteristics.

Also from Figure 4, we can see that in Phases 1, 2, and 4 compiling with the parallel environment reduces the "OTHER" category. It means that the instructions excluding all stalls execute faster in the 1-processor run of the parallel code than in the serial code. This can be the result of higher quality code (more optimizations applied, resulting in less instructions) or in an increased degree of instruction-level parallelism. Again this is unexpected, because the same code generator is used.

In Phase 4, the parallel code versions reduce the amount of load stalls for both the one-processor and four-processor runs. The parallel codes change data access patterns because of the implemented communication scheme. We assume that this leads to slightly increased data locality.

The OpenMP and MPI programs executed on one processor perform similarly, except for Phase 3. In Phase 3, the OpenMP version has a higher "OTHER" category, indicating less efficient code generation of the parallel variant. However, Phase 3 is relatively short and we have measured up to a 5% performance variance in repeated executions. Hence, the shown difference is not significant.

4.2 Analysis of the Parallel Program Performance

To discuss how the overheads change when the codes are executed in parallel we use the Speedup Component Model, introduced in Section 3.1. The results are given in Figure 5 for MPI and OpenMP on one and four processors in terms of speedup with respect to the serial run. The upper bars (labeled "P=1") present the same information that is displayed in Figure 4. However, the categories are now transformed so that their contributions to the speedup become clear. In the upper graphs, the ideal speedup is 1. The effect that each category has on the speedup is indicated by the components of the bars. A positive effect, indicated by the bar components on top of the measured speedup, stands for a latency that increases the execution time. The height of the bar quantifies the "lack of ideal speedup" due to this component. A negative component represents an overhead that decreases from the serial to the parallel version. Negative components can lead to superlinear speedup behavior. The sum of all components always equals the number of processors. For a one-processor run, the sum of all categories equals one.

The lower graphs show the four-processor performance. The overheads in Phase 1 remain similar to those of the one-processor run, which translates into good parallel efficiency on our four-processor system. This is expected of Phase 1, because it performs highly parallel operations but only communicates to fork processes at the beginning and join them at the end of the phase. Phase 2 of the OpenMP version shows a smaller improvement due to the *code generation* overhead component, which explains why less speedup was measured than with the MPI version. Again, this difference is despite the use of the same code generating compiler and it shows up consistently in repeated measurements. Phase 3 behaves quite differently in the two program variants. However this difference is not significant, as mentioned earlier.

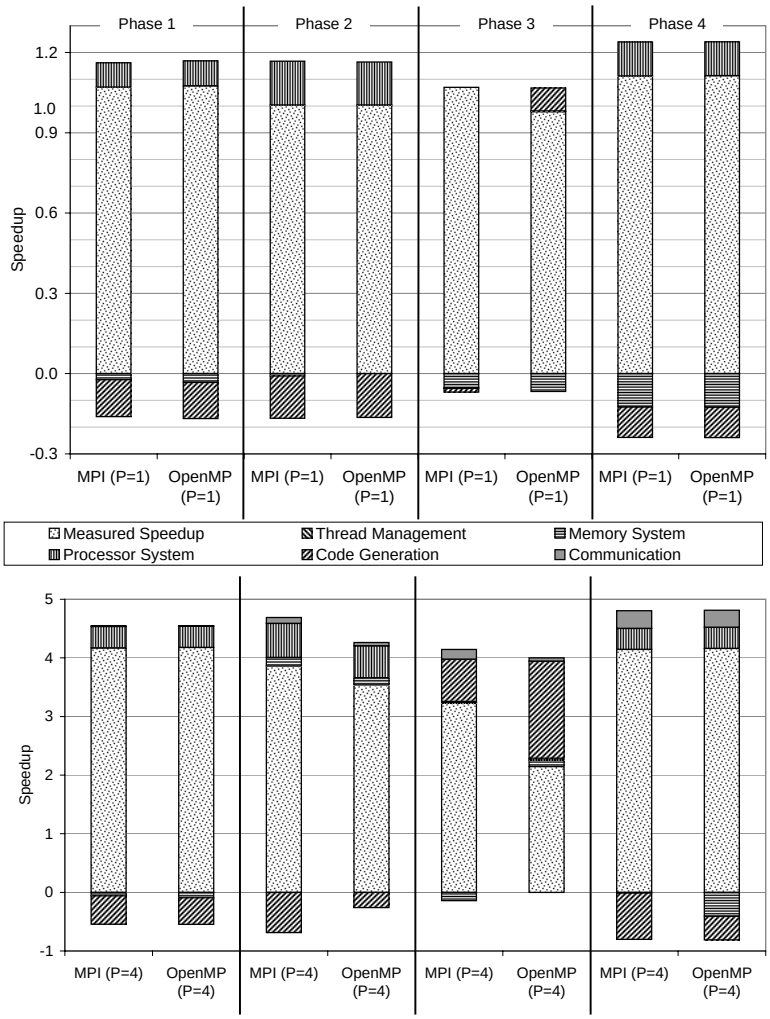


Fig. 5. Speedups Compared Model for the Versions of *Seis*. The upper graph displays the speedups with respect to the serial version of the code and executed on only one processor. The lower graph shows the speedups obtained when executing on four processors. An overhead component represents the amount that the measured speedup would increase (decrease for negative components) if this overhead were eliminated and all other components remained unchanged.

Figure 5 shows several differences between the OpenMP and the MPI implementation of *Seis*. In Phase 4 we can see the number of memory system stalls is less in the OpenMP version than in the MPI version. This shows up in the form of a negative memory system overhead component in the OpenMP versions. Interestingly, the MPI versions has the same measured speedup, as it has a larger negative code generation overhead component. Furthermore, the processor system stalls decrease in the 4-processor execution, however this gain is offset with an increase in communication overheads. These overheads are consistent with the fact that Phase 4 performs the most communication out of all the phases.

Overall, the parallel performance of the OpenMP and the MPI versions of *Seis* are very similar. In the most time-consuming code, Phase 4, the performance is the same. The second-most significant code, Phase 2, shows better performance with MPI than with OpenMP. However, our analysis indicates that the reason can be found in the compiler's code generation and not in the programming model. The communication overheads of both models are very small in Phases 1, 2, and 3. Only Phase 4 has a significant communication component and it is identical for the MPI and OpenMP variants of the application.

5 Conclusions

We have compared the performance of an OpenMP and an MPI version of a large-scale seismic processing application suite. We have analyzed the behavior in detail using hardware counters, which we have presented in the form of the speedup component model. This model quantifies the impact of the various overheads on the programs' speedups.

We have found that the overall performance of the MPI and OpenMP variants of the application is very similar. The two application variants exploit the same level of parallelism, which is expressed equally well in both programming models. Specifically, we have found that no performance difference is attributable to differences in the way the two models exchange data between processors.

However, there are also interesting differences in individual code sections. We found that the OpenMP version incurs more *code overhead* (e.g., the code executes more instructions) than the MPI version, which becomes more pronounced as the number of processors is increased. We also found situations where the OpenMP version incurred less *memory stalls*. However, we do not attribute these differences to intrinsic properties of any particular programming model.

While our studies basically show equivalence of the OpenMP and MPI programming models, the differences in overheads of individual code sections may point to potential improvements of compiler and architecture techniques. Investigating this potential is the objective of our ongoing work.

References

1. Rudolf Eigenmann and Siamak Hassanzadeh. Benchmarking with real industrial applications: The SPEC High-Performance Group. *IEEE Computational Science & Engineering*, III(1):18–23, Spring 1996.

2. Seon Wook Kim and Rudolf Eigenmann. Detailed, quantitative analysis of shared-memory parallel programs. Technical Report ECE-HPCLab-00204, HPCLAB, Purdue University, School of Electrical and Computer Engineering, 2000.
3. Bill Pottenger and Rudolf Eigenmann. Targeting a Shared-Address-Space version of the seismic benchmark Seis1.1. Technical Report 1456, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., September 1995.
4. Jay Hoeflinger, Prasad Alavilli, Thomas Jackson, and Bob Kuhn. Producing scalable performance with OpenMP: Experiments with two CFD applications. Technical report, Univ. of Illinois at Urbana-Champaign, 2000.
5. Abdul Wahed and Jerry Yan. Code generator for openmp. http://www.nas.nasa.gov/Groups/Tools/Projects/LCM/demos/openmp_frame /target.html, October 1999.
6. Abdul Waheed and Jerry Yan. Parallelization of nas benchmarks for shared memory multiprocessors. In *Proceedings of High Performance Computing and Networking (HPCN Europe '98)*, Amsterdam, The Netherlands, apr 21–23 1998.
7. C. C. Mosher and S. Hassanzadeh. ARCO seismic processing performance evaluation suite, user's guide. Technical report, ARCO, Plano, TX, 1993.
8. Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.
9. David L. Weaver and Tom Germond. *The SPARC Architecture Manual, Version 9*. SPARC International, Inc., PTR Prentice Hall, Englewood Cliffs, NJ 07632, 1994.