An Investigation of Tracing Overheads on High End Systems

Kathryn Mohror and Karen L. Karavanic
{kathryn, karavan} @ cs.pdx.edu
Department of Computer Science
Technical Report TR-06-06
Portland State University
P.O. Box 751
Portland, OR 97207-0751

Abstract -- *Although event tracing of parallel applications offers highly detailed performance information, tracing on current leading edge systems may lead to unacceptable perturbation of the target program and unmanageably large trace files. High end systems of the near future promise even greater scalability challenges. In this work we identify and quantify the overheads of application tracing. We report results for two ASC Purple Benchmarks with different communication characteristics: SMG2000, which exhibits an extremely high message rate, and SPhot, an embarrassingly parallel application with relatively little communication. We investigate several different sources of overhead related to tracing: instrumentation, differing trace buffer sizes, periodic buffer flushes to disk, system changes, and increasing numbers of processors in the target application. Our results show that tracing overhead is affected by differences in system software, as well as the choice of trace buffer size. As expected, the overhead of instrumentation correlates strongly with the number of events; however, our results indicate that the overhead of writing the trace buffer increases with increasing numbers of processors.*

1. Introduction

Event tracing of parallel applications offers a highly detailed view of application behavior that can be used to diagnose an important range of performance problems: for example, patterns of communication between processes may indicate a poor choice of MPI operation; slow communications between particular pairs or groups of nodes may indicate a poorly performing communication link or card, or a problem with a queue. A number of well-documented cases demonstrate that tracing downsized versions of large computations to improve the efficiency can miss important behaviors. Kale et al [KaKu03] describe a performance tuning study of the NAMD simulation code that includes several instances where particular performance issues only emerged at runs above 1000 processors. Arnold et al [ArAh06] also describe a study in which scientists at Lawrence Livermore National Laboratory (LLNL) encountered hangs of a Visualization Streams for Ultimate Scalability application on BlueGene/L (BG/L) that only occurred at runs of 8192 processes or more. These case studies show that tracing tools remain important for performance tuning, and that these tools must be able to scale to large numbers of processes.

However, the complexity of modern systems, and the scale of the current generation of simulation codes, stress the limits of tracing as a useful performance measurement approach. Terascale systems of today, and petascale systems of the near future, will employ tens to hundreds of thousands of processors to achieve desired FLOP rates. For example, BG/L, installed at LLNL, includes 131,072 processors. Red Storm, installed at Sandia National Laboratory, includes 26,544. Error and perturbation are inherent in the measurement of computing systems; we cannot completely eliminate measurement perturbation, nor can we exactly quantify the perturbation caused by a particular measurement tool. Pushing tracing technology to higher and higher scales drives up the overall perturbation, yielding unmanageably large trace files and less accurate results, and in many cases breaking the measurement tools completely. Even if the larger trace files can be successfully obtained, many trace analysis and visualization tools fail when trying to process them or display the results.

Determining which approach can be used in different scales requires information about trace tool scaling behavior. To address this need we have conducted a detailed empirical study of trace tool behavior.

We present the results of a series of experiments designed to identify and quantify the overheads incurred when tracing the execution of an application on a representative current architecture. We divide the total trace overhead into the overhead of trace instrumentation and the overhead of writing the trace files, and examine how these overheads change as we scale to larger numbers of events and larger numbers of processors. We present results from runs that used two different freely-available tracing tools, TAU and MPE. We studied two ASC purple benchmark applications: one, SMG2000 with a very high message rate; the other, SPhot, with a fairly low level of communication.

Previous work has shown that the overhead of tracing scales with the number of events; our results confirm this. However, breaking down the total trace overhead allows us to demonstrate that there is an important difference in the scaling characteristics between the instrumentation overhead and the writing overhead; the instrumentation overhead scales with the number of events, but the write overhead scales with the number of processes, even if the number of events remains fairly stable. The results demonstrate that behavior of the parallel file systems typically installed with high end systems today is a significant factor in determining the performance of a tracing tool.

2. Related Work

Because perturbation is intrinsic to measurement [Ga86], research focuses on techniques to lower or limit the overheads, remove the overheads in the resulting data, and to measure and model the overheads.

Researchers have investigated methods to lower the overheads of tracing [Kn03,KrGr96,OgSc93, ReRo93,ShMa06,YaDa00] and to limit the amount of overhead present in the resulting data [HoMi96]. The Event Monitoring Utility (EMU) was designed to allow the user to adjust how much data was collected in each trace record, thereby altering the amount of overhead in each measurement [KrGr96]. The authors found the writing overhead to be the largest monitoring overhead in the measured program. TAU [ShMa06] addresses instrumentation overhead by allowing users to disable instrumentation in routines that are called very frequently and have short duration. TAU also includes a tool that uses profile data to discover which functions should not be instrumented, and feeds this information to the automatic source instrumentor.

Several researchers have developed techniques to attempt to remove overheads from the reported data [FaKe96,Ga94,WoMa05,WiAn93,YaLi93]. Yan and Listgarten [YaLi93] specifically addressed the overhead of writing the trace buffer to disk in AIMS and were able to remove 100% of this overhead. They generate an event marker for these write operations and remove their overhead in a post-processing step.

Overhead studies can be found in the literature, although their focus and content differ from ours. Chung et al [ChWa06] evaluate several profiling and tracing tools on BG/L in terms of total overhead and write bandwidth, and note that the overheads of tracing are high and that the resulting trace files are unmanageably large. They suggest that the execution time overhead is substantially affected by generation of trace file output, but provide no measurements for their claim.

Two research efforts have developed models of the overheads in measurement systems. Malony et al. developed a model to describe the overheads of trace data and describe the possible results of measurement perturbation [MaRe92], then extended it to cover the overheads of SPMD programs [SaMa93]. They noted, as we do, that the execution time of traced programs was influenced by other factors than just the events in each processor independently. However, they did not explore this further. Waheed et al. [WaRo98] explored the overheads of trace buffer flushing and modeled two different flushing policies [WaMe95]. They found that the differences between the policies decreased with increased buffer sizes.

3. Experiment Design

Tracing a parallel application includes a number of steps: execution of code that causes a series of timestamped event records to be created in a local memory trace buffer; periodic trace buffer flushes to disk; and, with some tools, a merge step that combines trace files generated on different nodes into a single trace. In our work we ignore the final merge step, since it is optional in some cases. Our goal is to quantify the tracing overhead in terms of two components: the overhead due to the instrumentation, and the overhead due to writing the trace buffers to disk. We compared the wall clock times of executions that contained trace instrumentation and wrote trace files to disk, executions that contained trace instrumentation, but did not write the trace files to disk, and normal executions with no instrumentation in the program. In addition, we examined how the tracing overheads changed as we scaled the application to larger numbers of processors. We use the following naming convention for the experiments in this paper: the first part of the name describes the trace buffer size used, e.g. *1.5Buff* describes a 1.5 MB buffer; the second part of the name indicates whether or not the program contained trace instrumentation, *instr* or *noInstr*; the third part tells whether or not the executions wrote the trace buffer to disk, *write* or *noWrite*.

3.1 Metrics

The Metrics included in our measurements are:

- *Wall clock time*. Both benchmarks measure wall clock time with `MPI_Wtime`. The resolution of `MPI_Wtime` on MCR is microseconds. The wall clock time is measured after `MPI_Init` and before `MPI_Finalize` in all cases. The instrumentation overhead we do not measure includes any tool setup or finalization overhead, as well as the overhead of measuring any function calls before/after the timer is started/stopped. The writing overhead our measurements do not capture are trace file creations, final trace buffer flushes before file closure, trace file closure, and, in the case of MPE, trace file merging.
- *CPU clock time*. We use the native benchmark measurements for CPU time.
- *Write overhead*. The overhead of writing the buffer was computed by subtracting the average total wall clock time of the runs that did not write the trace buffer from the average total wall clock time of the runs that did write the trace.
- *Instrumentation overhead*. The overhead of instrumentation was computed by subtracting the average total wall clock time of the *noBuff_noInstr_noWrite* runs from the average total wall clock time of the runs that did not write the trace buffer.

3.2 Applications

We studied two applications from the ASCI Purple Benchmark suite, SMG2000 [BrFa00] and SPhot [Sb06]. SMG2000 (SMG) is characterized by an extremely high rate of messages: in our four process runs, SMG executed 434,272 send and receive calls in executions that took approximately 15 seconds. In contrast, SPhot is an embarrassingly parallel application; in a four-process, single-threaded execution of 512 runs with a total execution time of 350 seconds, the worker processes pass 642 messages, and the master process passes 1926 messages.

SMG is a parallel solver of linear systems arising from finite difference, finite volume, or finite element discretizations of the diffusion equation, $-\nabla \cdot (D\nabla u) + \sigma u = f$, in two and three dimensions. Two SMG parameters that are configurable at run time are: the problem size per processor, defined to be $nx * ny * nz$, and the processor topology, $Px * Py * Pz$. For our scaling experiments, we kept the problem size per processor constant, thereby increasing the total problem size. As we increased the numbers of processors, we altered the processor topology to $P * 1 * 1$, where P was the number of processors in the run. SMG reports values for six metrics in its output that correspond to wall and CPU clock times for the three phases of the execution: Struct Interface, SMG Setup, and SMG Solve. We used the values from these metrics to compare the performance of the executions. We used the sum of the wall clock times of each of the phases as the total wall clock time for this application.

SPhot is a two-dimensional photon transport code that tracks photons through a cylindrically symmetric spherical domain on a logically rectilinear, two-dimensional mesh. SPhot uses both MPI and OpenMP for parallelism; we used only one thread for each MPI process. SPhot was compiled with the Intel Fortran compiler, version 9.1. We used two different versions of SPhot runs in our study: SPS is a strong scaling version -- we kept the problem size the same while increasing the number of processes; SPW is a weak scaling version -- we kept the problem size per processor uniform, thus increasing the total problem size with the number of processors. We configured these approaches by changing the Nruns parameter in the input file input.dat, which controls the total amount of work done in a single execution. For strong scaling, we kept Nruns constant at 512 for all processor counts; for weak scaling, we set Nruns equal to the number of MPI ranks. SPhot reports the wall clock time of the execution. We used this measurement to compare the performance of the different executions of SPW and SPS.

Figure 1 shows the scaling behavior of the uninstrumented applications. For SMG, we see that the execution time increases slightly with the number of processes. For SPS, the execution time decreases with increasing numbers of processors; the performance of SPW increases slightly with increasing numbers of processors.

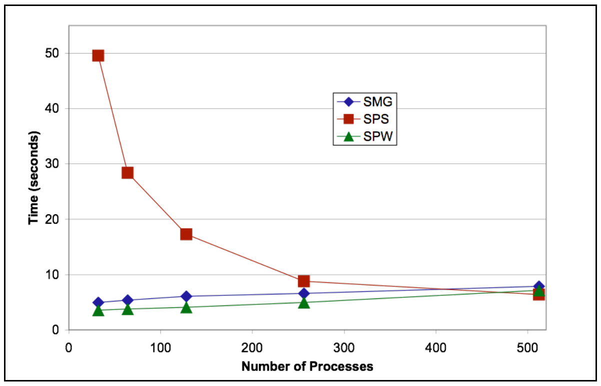

Figure 1. Performance of Uninstrumented Executions

3.3 Generating the Traces

We used two different tracing tools for our experiments: TAU version 2.15.1 [ShMa06] and the MultiProcessing Environment (MPE2) version 1.0.3p1 [ZaLu99]. We built each of these tools from source code using GNU compilers version 3.4.4.

We configured TAU, using the script provided, to do tracing only. We used version 3.4 of the Program Database Toolkit (PDT) [LiCu00] to automatically instrument the source code. We built several versions of TAU: for the *noWrite* versions we commented out the one line in the trace buffer flush routine of the TAU source that actually calls the `write` system call. We altered the number of records stored in the trace buffer between flushes, by changing the #define for TAU_MAX_RECORDS in the TAU source for each size and rebuilding, to test four different buffer sizes: 0.75MB (32,768 TAU events), 1.5 MB (default size for TAU; 65,536 TAU events), 3.0 MB (131,072 TAU events), and 8.0 MB (349,526 TAU events). TAU does not merge the individual trace files at runtime. We used the default level of instrumentation for TAU, which instruments all function entries and exits.

When used for logging MPI events, MPE uses the MPI profiling interface to capture the entry and exit time of MPI functions as well as details about the messages that are passed between processes, such as the communicator used. To produce an MPE library that didn't write the trace buffer to disk, we commented out three calls to `write` in the MPE logging source code. We also had to comment out one call to `CLOG_Converge_sort` because it caused a segmentation fault when there was no data in the trace files. This function is called in the MPE wrapper for `MPI_Finalize`, so it did not contribute to the timings reported in the SMG metrics. We altered the buffer sizes by changing the value of the environment variable CLOG_BUFFERED_BLOCKS. We also set the environment variable MPE_LOG_OVERHEAD to "no" so that MPE did not log events corresponding to the writing of the trace buffer. In MPE, each MPI process writes its own temporary trace file. During `MPI_Finalize`, these temporary trace files are merged into one trace file, and the temporary trace files are deleted. The temporary and merged trace files were written to the Lustre file system. We generated trace files in the CLOG2 MPE format. We used two different buffer sizes: 1.5 MB (24 CLOG buffered blocks), and 8.0 MB (default size for MPE; 128 CLOG buffered blocks). For SPW, we altered the SPhot source to call MPE logging library routines to log events for all function calls, to more directly correspond to the default TAU behavior. We refer to this as "MPc" for MPE with customized logging. For the SPW MPc experiments, we disabled the trace file merge step in MPI_Finalize, because it became quite time consuming with larger trace files.

3.4 Experiment Management

For each application build and run, we collected detailed information about the system and execution using the PerfTrack [KaMa05] experiment management tool. We used PerfTrack wrapper scripts to build the executables and launch the executions. For each set of executions, PerfTrack collected details such as the time of submission, operating system version, environment variable settings, information about dynamic libraries used, and input parameters to the application. After the executions completed, we used PerfTrack to enter the performance data and execution details that had been collected into a database, for later retrieval and analysis.

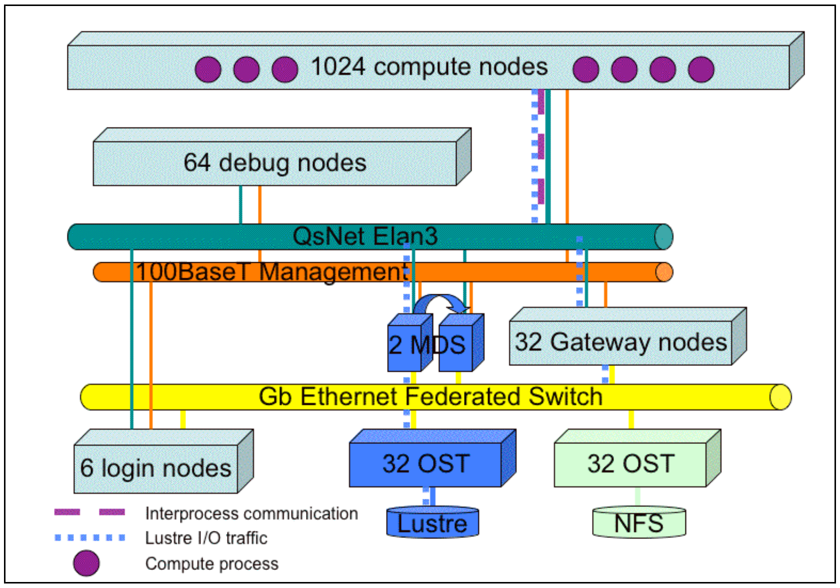

Figure 2. Experiment Environment.

The MPI processes in our experiments, represented by purple circles in the diagram, ran on a subset of the 1024 compute nodes of MCR. MPI communication between the processes traveled over the Quadrics QsNet Elan3 interconnect, shown by the purple dashed line. The I/O traffic for the Lustre file system, represented by the blue dotted line, also traveled over the Quadrics interconnect. Metadata requests went to one of two metadata servers (MDS), a fail-over pair. File data requests first went through the gateway nodes to an object storage target (OST), which handled completing the request on the actual parallel file system hardware.

3.5 Machine Environment

We collected all of our results on MCR, a 1152-node Linux cluster at LLNL running the CHAOS operating system [GaDu02]. (See Figure 2.) Each node comprised two 2.4 GHz Pentium Xeon processors and 4 GB of memory. The MPI implementation used was Quadrics MPI version 1.24-8, which is based on MPICH 1.2.4. All executions ran on the batch partition of MCR. Unless otherwise specified, each of our experiment sets comprised thirty identical executions. The resulting trace files, including any temporary files, were stored using the Lustre file system [Cluster].

4. Preliminary Experiments

This section details four, preliminary experiments we conducted to measure the overheads of tracing. We gathered measurements of the overheads of instrumentation and writing the trace buffer in four-process runs. We also experimented with buffer sizes and noted how changes in system software affected our results.

4.1 Overhead of Instrumentation versus Writing the Buffer Data

Our goal for this experiment was to break the total overhead into two categories: the amount of tracing overhead caused by the trace instrumentation, and the amount caused by writing the trace data to disk. We compared the execution times of three sets of runs: executions that contained trace instrumentation and wrote trace files to disk (*defBuff_instr_write*), executions that contained trace instrumentation, but did not write the trace files to disk (*defBuff_instr_noWrite*), and normal executions with no instrumentation in the program (*noBuff_noInstr_noWrite*).

4.1.1 Approach We compiled SMG for the *defBuff_instr_write* and *defBuff_instr_noWrite* executions using the TAU compiler. We did this by altering the Makefile of the application to include a TAU stub Makefile. The stub Makefile caused the TAU compiler to be used instead of the default compiler. The TAU compiler automatically instrumented the program to generate events for function entries and exits, and message sends and receives. After the program was instrumented, the TAU compiler compiled the program with gcc 3.4.4. For the *noBuff_noInstr_noWrite* executions, we did not use the TAU compiler, but

Table 1 SMG Trace File Sizes Generated by TAU

MPI RANK	TRACE FILE SIZE (MB)	EVENT FILE SIZE (K)
Rank 0	148	21
Rank 1	156	21
Rank 2	166	21
Rank 3	135	21

used gcc 3.4.4 via `mpicc`. For these runs, the version of the operating system on MCR was CHAOS 3.0 and the version of Lustre was 1.4.3.3. Each run for these experiments had four single-threaded MPI processes. The local problem size per processor was 35 x 35 x 35, and the processor topology specified was 4 x 1 x 1. Each process in the execution generated its own trace file.

4.1.2 Results The sizes of the trace files for the executions in the *defBuff_instr_write* set are shown in Table 1. The average execution time for the *defBuff_instr_write* set was approximately16.5 seconds. We show the average values reported for the four SMG metrics for each set of executions in Figure 3. We ran a single-factor ANOVA test for each metric to compare the values returned for the *defBuff_instr_write* and *defBuff_instr_noWrite* executions. We found that the differences between the *defBuff_instr_write* and *defBuff_instr_noWrite* executions for the SMG Setup and SMG Solve metrics were statistically significant at a 95% confidence level. However, there was not a statistically significant difference for the Struct Interface metrics at the 95% confidence level between the *defBuff_instr_write* and *defBuff_instr_noWrite* executions.

To understand the relative differences between the metrics for the execution sets, we counted the number of events generated for each phase by parsing the trace files generated by TAU using the TAU Trace File Reader API. The numbers of events in each of the phases is shown in Table 2. The event counts were identical for each run.

We inspected the TAU source to discover details about the size of each event record and the size of the trace buffer to better understand when trace buffer flushes would have occurred during the executions. The trace buffer was flushed whenever the number of records exceeded TAU_MAX_RECORDS (65,536 records). The size of the trace buffer was 1.5 MB and each event record occupied 24 bytes. From this information, we computed the number of trace buffer flushes in each phase for each rank, shown in Table 2. We see that there were no trace buffer flushes during the first phase of the program, Struct Interface, because the number of events generated in this phase is less than TAU_MAX_RECORDS. This explains why the differences between the *defBuff_instr_write* and *defBuff_instr_noWrite* execution sets were not statistically significant. In contrast, there were 39-48 trace buffer flushes, depending on rank, in the SMG Setup phase and 53-62 in the SMG Solve phase. This large number of trace buffer flushes explains the statistically significant differences for the SMG Setup and Solve phases.

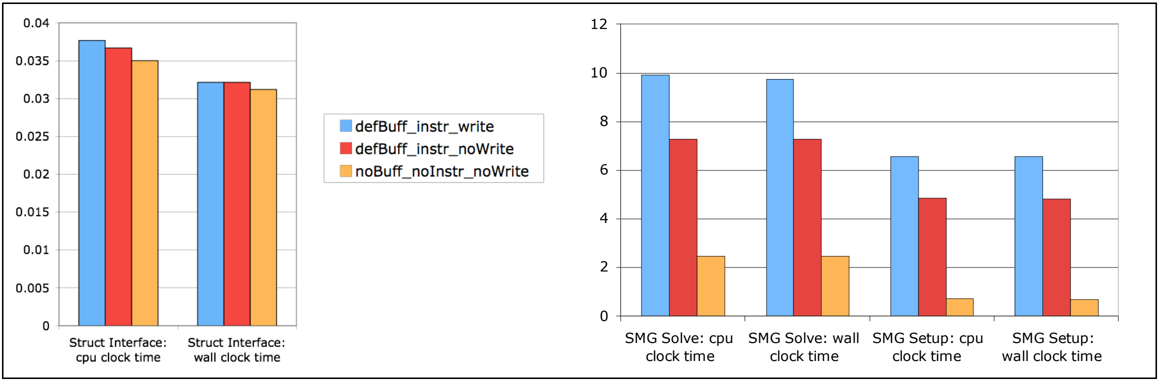

Figure 3. Average SMG Metrics for Four-Process Experiments

The bars shown in the chart are the average values for the metrics in each set of executions. The y-axis in the charts is the time in seconds. The chart on the left shows the Stuct Interface metrics. The chart on the right shows the SMG metric values.

Table 2 Event Counts and Buffer Flushes in SMG Phases

PHASE	RANK 0		RANK 1		RANK 2		RANK 3	
	Events	Flushes	Events	Flushes	Events	Flushes	Events	Flushes
Struct Interface	1,018	0	1,158	0	1,158	0	970	0
SMG Setup	2,598,632	39	3,019,360	46	3,173,376	48	2,419,444	36
SMG Solve	3,851,161	59	3,793,449	57	4,074,823	62	3,455,449	53

Table 3 Overhead Due to Writing the Trace Buffer and Instrumentation in SMG

PHASE	WRITE OVERHEAD (SEC.)	INSTRUMENTATION OVERHEAD (SEC.)	TOTAL OVERHEAD (SEC.)
SMG Solve CPU	2.64	4.79	7.43
SMG Solve wall	2.64	4.79	7.44
SMG Setup CPU	1.72	4.13	5.85
SMG Setup wall	1.74	4.13	5.87

In Table 3, we quantify the amount of overhead due to writing the trace buffer to disk in the executions. We estimated the average overhead due to writing the trace buffer by subtracting the average *defBuff_instr_noWrite* time from the average *defBuff_instr_write* time. The average overhead due to instrumentation was estimated by subtracting the average *noBuff_noInstr_noWrite* time from the average *defBuff_instr_noWrite* time. The total overhead is the sum of the write and instrumentation overheads. We see that the overhead of instrumentation is 2.4 times higher in the SMG Setup phase and 1.8 times higher in the SMG Solve phase.

4.1.3 Discussion From this experiment, we see that the overheads due to instrumentation and writing of the trace data to disk are significant. The overhead of instrumentation in this experiment was higher than the writing overhead.

4.2 TAU Buffer Size Experiment

The goal of this experiment was to determine the effect of trace buffer size on tracing overhead. Using a smaller buffer would cause more frequent small writes of the trace data, while a larger buffer would cause less frequent larger writes.

4.2.1 Approach We compared the metrics reported by SMG for runs where the trace buffer was half the default size (*smBuff_instr_write*) and where the trace buffer was double the default size (*bigBuff_instr_write*) to the *defBuff_instr_write* runs from the previous experiment. The default size of the trace buffer in TAU is 1.5 MB. For the *smBuff_instr_write* runs, we changed the buffer size to 0.75 MB, and for the *bigBuff_instr_write* runs, we changed the buffer size to 3.0 MB. The experimental setup for the executions was exactly the same as before, except for the changed buffer size.

4.2.2 Results and Discussion Figure 4 shows the average values for the Struct Interface, SMG Setup, and SMG Solve metrics for the thirty executions in each of the *defBuff_instr_write*, *smBuff_instr_write*, and *bigBuff_instr_write* sets. We see that the *defBuff_instr_write* and *smBuff_instr_write* sets had approximately the same values for the metrics, while the *bigBuff_instr_write* set values are much larger. A single-factor ANOVA test confirms that the differences between *defBuff_instr_write* and *smBuff_instr_write* are not statistically significant, while the differences between *defBuff_instr_write* and *bigBuff_instr_write* are statistically significant. We also compared the values for the Struct Interface metrics. The differences between *defBuff_instr_write* and *smBuff_instr_write* , and *defBuff_instr_write* and *bigBuff_instr_write* were not statistically significant for these metrics.

Our interest was piqued by the large difference between the *defBuff_instr_write* and *bigBuff_instr_write* runs, so we investigated further. One might guess that larger, less frequent writes to the Lustre file system would give better performance than small, more frequent writes. However, our results showed that the opposite was true. The first thing we noted was that there was large variability in the data for the *bigBuff_instr_write* runs for the SMG Solve metrics. The *bigBuff_instr_write* average value for SMG

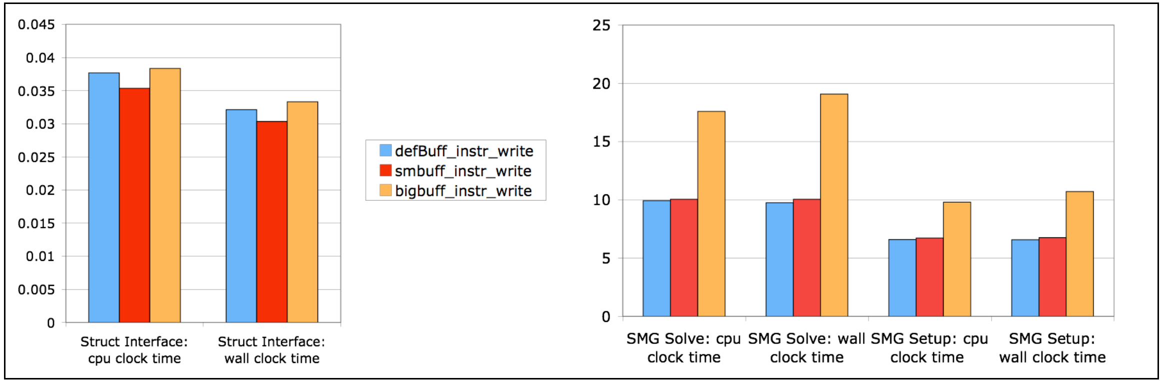

Figure 4. **Average SMG metric values using three trace buffer sizes**

The figure on the left shows the average values for the Struct Interface metrics for three different buffer sizes. The figure on the right shows the average values for the SMG metrics with the three different buffer sizes. The y-axis in the charts is time in seconds.

Solve: cpu time was 17.58 seconds with a standard deviation of 18.54. Closer inspection of the data showed that there was one run with values for the SMG Solve metrics that were approximately 7 times higher than the values for the other runs. Given that the Lustre file system is a shared resource on MCR, we guessed that there could have been some other job heavily using Lustre at the time of this rogue run. To attempt to confirm this, we ran three additional sets of runs, each with ten trials, submitted as separate batch jobs. The sets were named *bigBuff_instr_write_2*, *bigBuff_instr_write_3*, and *bigBuff_instr_write_4*. Figure 5 shows

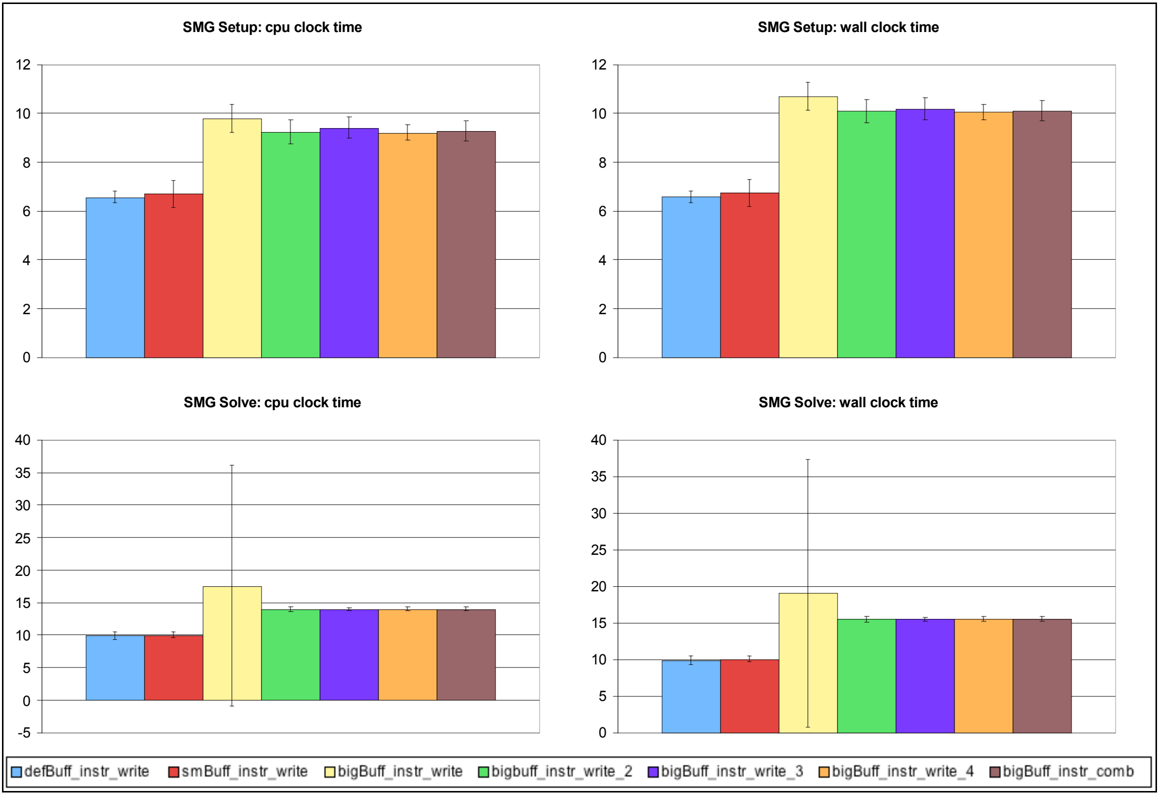

Figure 5. **Average SMG Metrics for Repeated Large Buffer Experiments**

This figure shows the average values for the metrics reported by SMG for three different buffer sizes. The charts have error bars showing one standard deviation. The top charts show the values for the SMG Setup metrics, while the bottom charts show the values for the SMG Solve metrics. The charts on the left are the CPU time measurements. On the right are the wall clock time measurements. The y-axis is the time in seconds.

comparisons of the SMG Setup and Solve metrics for *defBuff_instr_write* and the runs that used the larger buffer. We included error bars that span plus and minus one standard deviation in these charts. The bars labeled *bigBuff_instr_write_comb* are the averages of the values from the combined data of the three additional sets of runs. We see that these new runs do not suffer from high variability, as did the *bigBuff_instr_write* runs. We conclude that heavy use of Lustre by a concurrent user could have indeed caused the high variability in the *bigBuff_instr_write* runs. We performed a single-factor ANOVA test and discovered that there was a statistically significant difference at the 95% confidence level between the SMG Setup and Solve metrics for *defBuff_instr_write* and *bigBuff_instr_write_comb*.

Based on these results, we investigated further to find out why the executions with the larger buffer took longer to execute. First we looked at the trace files themselves to try to understand the differences. We randomly chose one representative execution from each of the *defBuff_instr_write* and *bigBuff_instr_write* groups for further examination. We merged the trace files for the individual processes into a combined TAU trace for each execution. We converted the combined TAU trace files into Open Trace Format (OTF) [KnBr06] format using the TAU utility `tau2otf`. We used Vampir Next Generation (VNG) [BrNa03] to open the OTF trace files, using four VNG processes in parallel on different nodes of MCR. First, we created summary charts for the two executions. The summary charts showed the *bigBuff_instr_write* execution spent much more time in MPI than the *defBuff_instr_write* execution.

Next, we wanted to get an idea of the overall time spent in `MPI_Waitall` by each of these executions. We used the TAU utility `tau2profile` to convert the individual trace files for each of the processes in each execution into TAU profiles. We used the TAU tool paraprof to examine the profiles. The profiles showed that the processes spent most of their time in `MPI_Waitall` for both the *defBuff_instr_write* and *bigBuff_instr_write* sets of runs. However, the *bigBuff_instr_write* executions spent more time in `MPI_Waitall` than the *defBuff_instr_write* executions. On average, the *bigBuff_instr_write* executions spent approximately 2.3 times more time in `MPI_Waitall` than did the *defBuff_instr_write* executions.

4.3 Write Overhead with Larger Buffer

We performed this experiment to measure the overhead due to writing the trace with the larger buffer. We wanted to know if the overhead was due to longer writing times or whether it was due to the size of the buffer itself.

4.3.1 Approach We launched a set of thirty runs called *bigBuff_instr_noWrite* that were configured in the same way was *bigBuff_instr_write*, but did not actually write the trace buffer to disk.

4.3.2 Results We compared the running times of these runs with those from previous experiments; results are presented in Figure 6. We performed single factor ANOVA tests to compare the differences between the sets of executions. The differences for the values reported for the SMG Solve and SMG Setup metrics between *defBuff_instr_noWrite* and *bigBuff_instr_noWrite*, and *bigBuff_instr_noWrite* and *bigBuff_instr_write* were statistically significant at a 95% confidence level. For the metric Struct Interface: cpu clock time, the differences were not statistically significant. For Struct Interface: wall clock time, the differences between *defBuff_instr_noWrite* and *bigBuff_instr_noWrite* were statistically significant.

4.3.3 Discussion We see that for the SMG Solve and SMG Setup phases, the longer running times for the larger buffer size seems to be largely due to the change in the buffer size alone, and only partly due to the time for writing the data to the file. For the Struct Interface phase, there does not seem to be a clear relationship between the different executions.

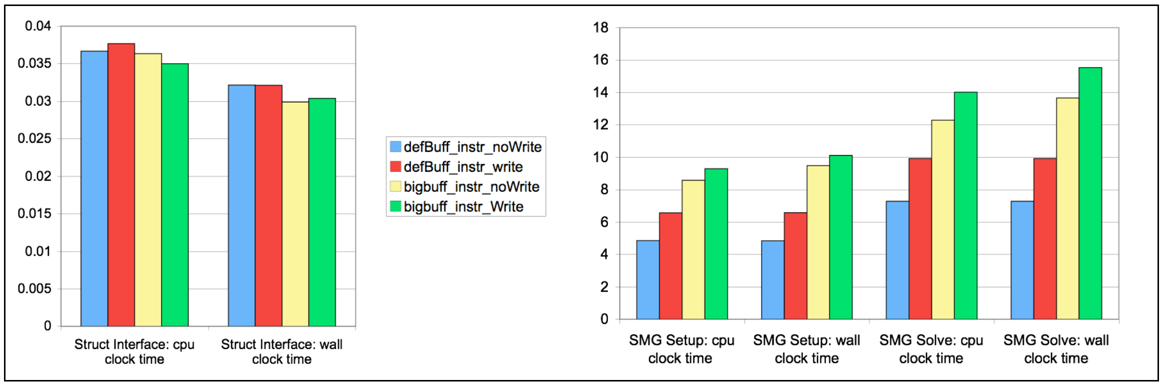

Figure 6. Average Metric Values for Default and Larger Buffers Showing Overhead of Writing

The chart on the left shows the average values for the Struct Interface metrics. The chart on the right shows the values for the SMG metrics. The y-axis is time in seconds.

4.4 Scaling with Larger Buffer and Unexpected Results

In this experiment, our goal was to determine how the overheads of tracing scaled with the number of processes.

4.4.1 Approach We ran new sets of executions, with 8, 16, 32, and 64 processes and three different buffer sizes: 0.75 MB, 1.5MB, and 3.0 MB. We also ran a set of executions with no instrumentation. The local problem size per processor was 35 x 35 x 35, and the processor topology specified was *P* x 1 x 1, where *P* was the number of processes in the execution.

4.4.2 Results and Discussion These experiments yielded surprising results. We found that in this case the executions that used the bigger buffer had *less* tracing overhead than those that used the default buffer size. Because this conflicted with our earlier findings, we ran the experiment yet again with four processes. This time, the executions that used the larger buffer did not have worse performance, and in fact, performed slightly better. Fortunately, we had used PerfTrack to gather and store the system details and performance information. Because of this, we were able to see the differences between our first set of runs and this newer set of runs. It turns out that there were system changes between the set of runs that had poor performance with the larger buffer (Section 4.2), and the latest set of scaling experiments. The system differences between the sets of runs were an operating system upgrade, from CHAOS 3.0 to 3.1, and a file system upgrade, from Lustre 1.4.3.3 to 1.4.5.8. The staff at LLNL ran performance regression tests on the operating system changes only. Their results showed that the operating system upgrade did not significantly affect OS benchmark performance. The benchmarks they used measured aspects such as network and memory bandwidth, message passing performance, and I/O performance (Mike Haskell, personal communication, June 2006). From this, we concluded that the performance differences we saw were due to changes to Lustre.

In Figure 7 we compare the executions when run under CHAOS 3.0/Lustre 1.4.3.3 (denoted by "_1" in the chart legend) and CHAOS 3.1/Lustre 1.4.5.8 (denoted by "_2" in the chart legend). There is no apparent difference between the runs for the Struct Interface metrics. We performed single factor ANVOA tests, using the system change as the factor, and found there were no statistically significant differences between the runs for the Struct Interface metrics. For the SMG metrics, the differences due to the system version between the *smBuff_instr_write* runs are not statistically significant by a single factor ANOVA test at the 95% confidence level. However, the differences due to system version between the *defBuff_instr_write*, *defBuff_instr_noWrite*, *bigBuff_instr_write*, and *bigBuff_instr_noWrite* sets of runs were statistically significant by a single factor ANOVA test. We performed a two-factor ANOVA test for each buffer size, with the two factors being the system differences and whether or not the trace buffer was written to disk. From this, we saw strikingly different results for the two buffer sizes. In Table 4, we see that for the runs that used the default-size buffer, the major contributor (> 90%) to variation between the runs was whether or not the trace buffer was written to disk. We also see that for the runs that used the larger buffer, the major contributor (> 87%) was the system differences.

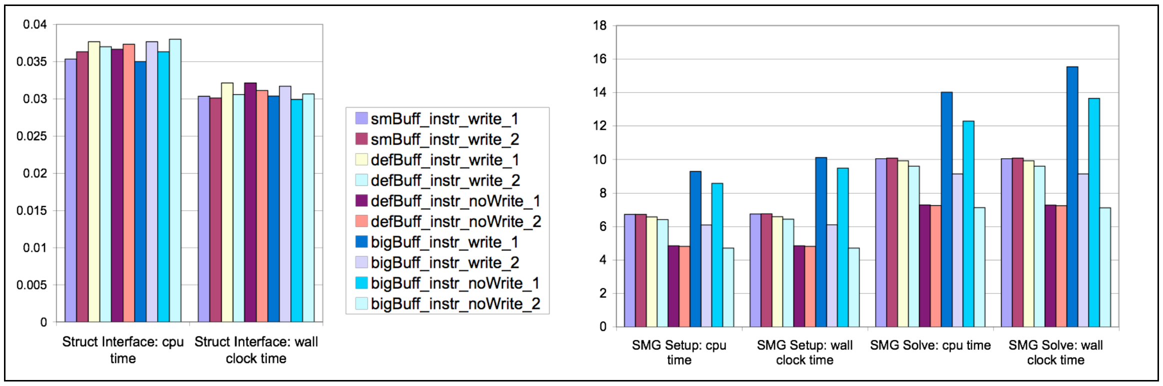

Figure 7. Comparison of executions with different buffer sizes showing system differences

The chart on the left shows the average values for the Struct Inteface metrics. The chart on the right shows the average values for the SMG metrics. The y-axis is time in seconds. The legend shows the system configuration for each bar. The names that end in "_1" were run under CHAOS 3.0/Lustre 1.4.3.3. The names that end in "_2" were run under and CHAOS 3.1/Lustre 1.4.5.8.

Table 4 Percent Contribution to Variation from Two-Factor ANOVA

FACTOR	SMG SETUP: CPU CLOCK TIME		SMG SETUP: WALL CLOCK TIME		SMG SOLVE: CPU CLOCK TIME		SMG SOLVE: WALL CLOCK TIME	
	defBuff	*bigBuff*	*defBuff*	*bigBuff*	*defBuff*	*bigBuff*	*defBuff*	*bigBuff*
Write	***96.12***	7.75	***96.20***	4.88	***93.83***	12.07	***93.83***	8.29
System	0.32	***88.65***	0.29	***92.07***	0.51	***87.16***	0.51	***91.12***
Write:System	0.13	0.78	0.11	0.67	0.32	0.08	0.31	0.01
Residuals	3.43	2.83	3.39	2.38	5.34	0.69	5.34	0.58

Table 5 Percent Contribution to Variation from Three-Factor ANOVA

FACTOR	SMG SETUP: CPU CLOCK TIME	SMG SETUP: WALL CLOCK TIME	SMG SOLVE: CPU CLOCK TIME	SMG SOLVE: WALL CLOCK TIME
Buffer	***21.01***	***23.89***	***20.30***	***23.66***
System	***30.71***	***32.03***	***30.37***	***32.13***
Write	17.02	11.56	21.32	14.39
Buffer:System	***27.55***	***29.44***	***26.24***	***28.65***
Buffer:write	0.87	0.72	0.44	0.22
System:Write	0.17	0.16	0.00	0.00
Buffer:System:Write	0.36	0.30	0.10	0.03
Residuals	2.31	1.91	1.24	0.91

Next, we performed a three-factor ANOVA test using the buffer size, the system differences, and whether or not the trace buffer was written as the factors. Table 5 shows the percentage of contributions to the variation between the runs for each of the factors and interactions between them. We see that the number one factor was the system differences, followed by the interaction between the buffer size and the system differences. On average, the factor with the third largest contribution to the variation was the buffer size, and the fourth largest was whether or not the trace buffer was written to disk.

5. Scaling Experiments

In this set of experiments, our goal was to see how the overheads of tracing scaled with increasing processors, different performance tools, and different applications. For each application: SMG, SPS, and SPW, we varied the buffer sizes: 1.5 and 8.0 MB, numbers of processes: 32, 64, 128, 256, and 512, whether or not the trace buffer was written to disk, and the tracing tool used: TAU or MPE. Each of our experiment sets comprised thirty identical executions. The version of CHAOS was 3.1; the version of Lustre was 1.4.5.8.

5.1 Execution time and Scaling Behavior for Instrumented Runs

We show the average wall clock times for our experiments broken down into time spent in application code, trace instrumentation, and writing the trace buffer in Figure 8. Each bar in a set in these charts represents the average behavior of executions with 32, 64, 128, 256, and 512 processes, respectively. In the left of Figure 8 we show the measurements for SMG with TAU and MPE, and SPW with TAU and MPc. In each run set, we see the same trend; as the numbers of processes increases, the total execution time increases, largely due to the time spent writing the trace buffer. The time spent in the application code and in trace instrumentation remains relatively constant. In the middle of Figure 8, we show the execution times of SPS with TAU. Here, as the numbers of processes increase, the total execution time decreases. However, even though the time spent in writing the trace buffer decreases with increasing processors, it does not decrease as rapidly as the time spent in instrumentation or application code. In the right of Figure 8, we show the results for SPS and SPW with MPE. Here we see that the differences between the *write* and *noWrite* executions are indistinguishable due to the very small amounts of data collected and written.

5.2 Event Counts and Tracefile Sizes

We show the event counts and resulting trace file sizes in Table 6. For executions traced with TAU, the Trace File column refers to the sum of the per-process trace files produced. For those traced with MPE, the Trace File column gives the size of the merged trace file. The data in this table helps us evaluate the results from the previous sections.

For SMG, we see that the counts for TAU and MPE exhibit similar trends, but are different by roughly an order of magnitude. As the numbers of processors double, the per-process event counts and trace data written by each process increase slightly, while the total number of events and resulting trace file sizes double. We see markedly different results between TAU and MPE for SPS; the event counts differ by six orders of magnitude. This is because TAU measures all function entries and exits, while MPE measures only MPI activity. For TAU, as the number of processors double: the per-process event counts decrease by half; and the total event count and resulting trace file sizes remain constant. With MPE, as the number of processes double: the average per-process event count decreases by half; and the maximum per-process

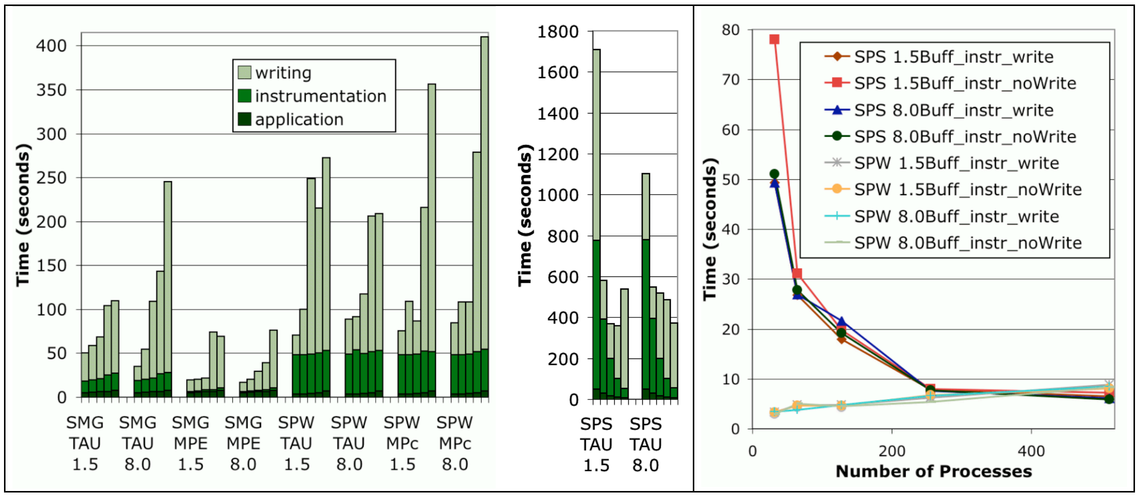

Figure 8. Performance of Instrumented Executions.

On the left, we show the total execution time for the benchmarks when measured with TAU and MPE, with the contributions of the application itself, the trace instrumentation, and writing the trace buffer to the execution time. Each bar in a set represents the average behavior of executions with 32, 64, 128, 256, and 512 processes, respectively. The set labels include (top to bottom): the benchmark name, the measurement tool, and the buffer size. On the right, we show the performance of executions of SPhot using MPE.

event count, the total event count, and resulting trace file sizes increase slightly. For SPW, the counts for TAU and MPc are nearly identical, while the counts for MPE differ. Again, this is because of differences in what was measured by the tools. As the numbers of processors double for TAU and MPc: the per-process event counts remain relatively constant, with slight non-monotonic increases and decreases; the total number of events and resulting trace file sizes double. For MPE, doubling the number of processes means: the average per-process event count increases slightly; the maximum per-process event count, the total event count, and resulting trace file sizes double. The total event count and trace file sizes for MPE are roughly six orders of magnitude less than those of TAU and MPc.

From this information, we can say what we expect in terms of tracing overheads for the different applications and tools. For the weakly-scaled SMG and SPW, we expect that the overheads of tracing would remain relatively constant with increasing numbers of processors because the amount of data being collected and written per-process remains relatively constant. However, for SPW with MPE, we expect to see very little overheads due to the small amount of data collected. For SPS and TAU, we expect the overheads of tracing to decrease with increasing numbers of processors, because the amount of data being collected and written per-process decreases with increasing processes. For SPS with MPE, we expect to see very little overhead because of the small amount of data collected.

5.3 Dominant Contributions

We computed the percentage contribution to variation using three-factor ANOVA, with the buffer size, the number of processes, and whether or not the trace buffer was written to disk as the factors. We show selected results of this analysis in Table 7. For SMG, we show only results for the wall clock time of the dominating phase, SMG Solve. We show the factors or interactions between factors that contributed more than 10% to the variation in the runs. The other factors and interactions are omitted for space and clarity. Across the board, we found that the contribution due to variability within the runs (Residuals) was high. In general, there was quite a bit of variation in the running times of the executions that wrote the trace buffer, which explains the high contribution of the residuals (See Figure 10.). In the rest of this discussion, we only consider contributors to variation outside of the residuals. For SMG measured with TAU and MPE, the largest contributor was whether or not the buffer was written, at 33% and 26%, respectively. The largest contributor for SPS with TAU was the number of processes in the run (19%), followed closely by whether or not the trace buffer was written (14%). SPS with MPE had the number of processes as the dominating factor at 51%. SPW with TAU and MPc both had writing the trace buffer as the largest contributor, at 34% and 24%, while SPW with MPE had the number of processes as the largest, at 81%. The differences in the dominating factors for the SPhot runs with MPE is attributed to the comparatively very small amount of data collected.

5.4 Execution Time vs Event Counts

We computed the correlation of the average total wall clock time of the runs with the average event count per rank and with the maximum event count over all ranks. We show the correlations with maximum event count in Table 8. We found that overall, the total wall clock time was more highly correlated with the maximum event count over all ranks than with the average event count. The single exception was SPS with MPE, where the highest correlations were with the average event count. This is due to the higher relative difference between the event counts in the master and slave ranks due to the algorithm of the application, because the master receives messages from all the slaves in each iteration. SPS with MPE had a relatively weak negative correlation with the maximum event count. The correlation is negative because as the process count increases, the number of messages that the master process receives increases, and the execution time decreases, giving a negative correlation. In general, we noted that the executions that did not write the trace buffer to disk had a higher correlation with the event count than did the executions that did write the trace buffer to disk.

Figure 9 shows the overheads of writing and instrumentation as the maximum number of events in a single rank increases. For SMG with TAU and MPE, we see a clear pattern. The instrumentation overhead appears to vary linearly with the number of events, while the overhead of writing the trace increases much more rapidly, and does not appear to have a linear relationship with the event count. The behavior of SPS is different, because in this application, as the number of events increases, the number of processes decreases. Again, though, the instrumentation overhead appears to have a linear relationship with the event count. The write overhead is high at higher event counts, but also at the low event counts, when the number of writing

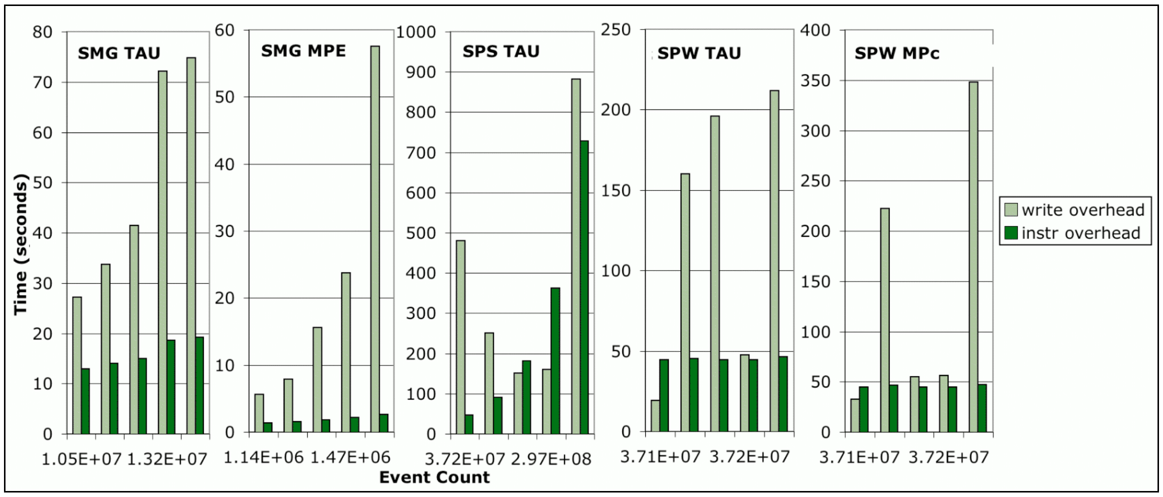

Figure 9. Tracing Overheads as a Function of Maximum Event Count in a Single Rank

For both SPW charts, the order of the bar groupings with respect to processor count is: 32, 128, 256, 64, 512.

processes is higher. For SPW, the number of events does not change much between the run sets, and we see that the instrumentation overhead is relatively constant, too. However, the writing overhead fluctuates widely. The reason for this is that the maximum event count in a rank does not monotonically increase or decrease with increasing processors as it does for SMG or SPS.

6. Discussion

We examined the overheads of instrumentation and writing the trace buffer with increasing numbers of processors with both TAU and MPE. We expected the overheads to scale with the number of events in the execution. For SMG and SPW, we expected the overheads to increase slightly with increasing process counts; for SPS, we expected the overheads to decrease with increasing process counts. For both tools, we found that the overhead of instrumentation scaled with the event counts. However, we found that the overhead of writing the trace buffer scaled with the number of processors in the execution. In addition, we saw that the overhead of writing the trace buffer did not appear to be linearly related to the event counts, suggesting the overhead of writing the trace is not strictly determined only by the number of events.

Our method for measuring the overheads of tracing was to take high-level measurements reported by the benchmarks themselves and compare them. In general, our run sets that wrote the trace buffer to disk had high variability. Was there some way we could have eliminated this variation and derived more precise measures of tracing overheads? We only collected timing data for the whole execution. Could we have gained valuable insight by collecting per-process or per-write performance information? Sources of variability in writing times for the different executions include: contention for file system resources, either by competing processes in the same execution, or by other users of Lustre; contention for network resources, either by other I/O operations to Lustre, or by MPI communication; and operating system or daemon interference during the write. Any user of this system gathering trace data would be subject to these sources of variation in their measurements. We argue that although the variability in our data is high, it represents a true measure of the typical user experience. We present the averages of each run set as a realistic estimate of the overheads a typical user would experience. In order to gather more detailed information about tracing overheads, another possible measurement technique would be to log each buffer flush individually and store them in the trace. This technique would have the advantage of obtaining more precise writing times, information about each individual write, per-process information, and would gather the information about instrumentation overhead and writing overhead in one execution. However, it would artificially inflate the instrumentation overhead, because of the added instrumentation to log the writes; it would add data to the trace file, inflating the buffer writing time and the already problematic size of the trace files; and the difficulty in extracting the added performance information would be high, as the large trace files would need to be parsed and analyzed. In addition, this method would be subject to the same sources of variability in the writing times as discussed above.

An important constraint in the utility of tracing tools for high end systems is storage requirements. For these scaling experiments, we saved just one sample trace file from each run set. The total storage requirements were approximately 8.4 TB. If we had tried to save all trace files, the storage requirements would have been 250 TB. Our user quota on MCR is 16 GB. This space was inadequate, not only for archiving the trace files, but even for storing trace files from a single run in many cases. Livermore Computing provides the Lustre file system as a temporary scratch space for writing/reading large amounts of data. We attempted to store the trace files to the archival tape storage system, but we had difficulties with the TAU 32-process SPS runs because the htar utility can't handle individual files with size greater than approximately 8 GB, due to restrictions in the format of the tar header.

7. Conclusions and Future Work

In our scaling experiments, the execution times of the *noWrite* runs tended to scale with the maximum number of events. However, the execution times of the *write* runs did not scale as strongly with the number of events, and tended to scale with increasing numbers of processors, possibly due to contention caused by sharing the file system resource. In the course of these experiments, we also encountered difficulties storing and analyzing these large trace files. Even though our executions were relatively short, we generated on the order of 250 TB of data for the scaling experiments alone. In addition to storage concerns, the processing time for analyzing and merging the trace files was in some cases considerable.

Our results suggest that the trace writes will dominate the overheads more and more with increasing numbers of processes. They indicate that the trace overheads are sensitive to the underlying file system. They also imply that increasing number of events to be expected with increasing the number of threads will primarily scale up the overheads associated with instrumentation, as opposed to writing, while increasing the number of trace buffers will quickly drive up the cost of writing.

Previously developed models of the overheads of tracing make some simplifying assumptions and do not account for the overheads we saw in our experiments. The early work done by Malony et al. assumes that in the case of programs that do not communicate, the perturbation effect for each processor is only due to the events that occur on that processor [SaMa93]. Our results tend to dispute this assumption. If the application we used did not contain communication, the times for writing the trace file to disk could have been extended due to resource contention on the network by the other processes trying to write the trace file to disk, and could have increased with increasing numbers of processes sharing the resource. The model presented by Waheed et al. also does not account for this interaction when modeling the buffer flushing policies [WaMe95] – instead, they assume a constant latency for all writes of the trace buffer. We plan to develop a new model to address the characteristics of current and near future high end clusters and parallel file systems.

8. Acknowledgements

This work was funded in part by a PSU Maseeh Graduate Fellowship and an Oregon Sports Lottery Graduate Fellowship. The authors acknowledge Mike Haskell and Richard Hedges for their help with CHAOS/Lustre information; Bronis de Supinski for feedback on an early draft; and Lisa Zurk, Michael Rawdon, and our CAT team for emergency office space and tech support during construction.

References

[ArAh06] D. Arnold, D. Ahn, B. deSupinski, G. Lee, B. Miller, and M. Schulz, "Stack Trace Analysis for Large Scale Applications," University of Wisconsin Technical Report, TR-1584, October 2006.

[BrFa00] P. Brown, R. Falgout, and J. Jones, "Semicoarsening multigrid on distributed memory machines." SIAM Journal on Scientific Computing, 21 (2000), pp. 1823-1834. Also available as Lawrence Livermore National Laboratory technical report UCRL-JC-130720.

[BrNa03] H. Brunst, W. Nagel, A. Malony, "A distributed performance analysis architecture for clusters," in Proceedings of IEEE International Conference on Cluster Computing, Hong Kong, pp. 73-81, Dec. 1-4, 2003.

[ChWa06] I. Chung, R. Walkup, H. Wen, and H. Yu, "MPI Performance Analysis Tools on Blue Gene/L," In Proceedings of SC2006, Tampa, Florida, November 11-17, 2006.

[Cluster] Cluster File Systems, Inc., "Lustre: A Scalable, High-Performance File System," Cluster File Systems, Inc. whitepaper, Nov. 11, 2002. (available at: http://www.lustre.org/docs/whitepaper.pdf, June 2006).

[FaKe96] A. Fagot and J. de Kergommeaux, "Systematic Assessment of the Overhead of Tracing Parallel Programs," in Proceedings of 4th Euromicro Workshop on Parallel and Distributed Processing, pp. 179-185, 1996.

[Ga86] J. Gait, "A Probe Effect in Concurrent Programs, " in Software – Practice and Experience 16(3):225-233, 1986.

[Ga94] J. Gannon, K. Williams, M. Andersland, J. Lumpp, Jr., and T. Casavant, "Using Perturbation Tracking to Compensate for Intrusiuon Propagation in Message Passing Systems," in Proceedings of the14th International Conference on Distributed Computing Systems*,* Poznan, Poland, pp. 141-412, June 1994.

[GaDu02] J. Garlick and C. Dunlap, "Building CHAOS: an Operating Environment for Livermore Linux Clusters," Lawrence Livermore National Laboratory, CA, UCRL-ID-151968, February 21, 2002.

[HoMi96] J. Hollingsworth and B. Miller, "An Adaptive Cost Model for Parallel Program Instrumentation," in Proceedings of Euro-Par '96, Lyon, France, pp. 88-97, August 1996.

[KaKu03] L. Kale, S. Kumar, G. Zheng, and C. Wai Lee, Scaling Molecular Dynamics to 3000 Processors with Projections: A Performance Analysis Case Study, Terascale Performance Analysis Workshop, in Proceedings Part IV of the International Conference on Computational Science (ICCS), Melbourne, Australia and St. Petersburg, Russia, pp. 23-32, June 2-4, 2003.

[KaMa05] K. Karavanic, J. May, K. Mohror, B. Miller, K. Huck, R. Knapp, and B. Pugh, "Integrating Database Technology with Comparison-based Parallel Performance Diagnosis: The PerfTrack Performance Experiment Management Tool," in Proceedings of SC2005, Seattle, WA, November 2005.

[Kn03] A. Knüpfer, "A New Data Compression Technique for Event Based Program Traces," in Proceedings Part III of International Conference on Computer Science, Melbourne, Australia and St. Petersburg, Russia, pp. 956-965, June 2-4, 2003.

[KnBr06] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. Nagel, "Introducing the Open Trace Format (OTF)," in Proceedings Part II of International Conference on Computational Science, Reading, UK, pp. 526-533, May 28-31, 2006.

[KrGr96] D. Kranzlmüller, S. Grabner, and J. Volkert, "Monitoring Strategies for Hypercube Systems," in Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing, pp. 486-492, 1996.

[LiCu00] K. Lindlan, J. Cuny, A. Malony, S. Shende, B. Mohr, R. Rivenburgh, and C. Rasmussen, "A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates," in Proceedings of SC2000, Dallas, November 2000.

[MaRe92] A. Malony, D. Reed, and H. Wijshoff, "Performance Measurement Intrusion and Perturbation Analysis," in IEEE Transactions on Parallel and Distributed Systems, 3(4):433-450, July 1992.

[MoKa06] K. Mohror and K.L. Karavanic, "A Study of Tracing Overhead on a High-Performance Linux Cluster," Portland State University Computer Science Technical Report number TR-06-06, December 2006.

[NaAr96] W. Nagel, A. Arnold, M. Weber, H. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," Supercomputer, 12(1):69-80, January 1996.

[OgSc93] D. Ogle, K. Schwan, and R. Snodgrass, "Application-Dependent Dynamic Monitoring of Distributed and Parallel Systems," in IEEE Transactions on Parallel and Distributed Systems, pp. 762-778, 1993.

[ReRo93] D. Reed, P. Roth, R. Aydt, K. Shields, L. Tavera, R. Noe, and B. Schwartz, "Scalable Performance Analysis: the Pablo Performance Analysis Environment," in Proceedings of the Scalable Parallel Libraries Conference, pp.104-113, Mississippi State, MS, USA, October 6-8, 1993.

[SaMa93] S. Sarukkai and A. Malony, "Perturbation Analysis of High Level Instrumentation for SPMD Programs, " in Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, USA, pp. 44-53, 1993.

[Sb06] The SPHOT Benchmark Code, Available at http://www.llnl.gov/asci/purple/benchmarks/limited/sphot/, December 8, 2006.

[ShMa06] S. Shende and A. Malony, "The TAU Parallel Performance System, " in the International Journal of High Performance Computing Applications, 20(2):287-331, Summer 2006.

[WaMe95] A. Waheed, V. Melfi, and D. Rover, "A Model for Instrumentation System Management in Concurrent Computer Systems," in Proceedings of the 28th Hawaii International Conference on System Sciences, pp. 432-441, 1995.

[WaRo98] A. Waheed, D. Rover, and J. Hollingsworth, "Modeling and Evaluating Design Alternatives for an On-line Instrumentation System: A Case Study," in IEEE Transactions on Software Engineering, 24(6):451-470, June 1998.

[WiAn93] K. Williams, M. Andersland, J. Gannon, J. Lummp, Jr., and T. Casavant, "Perturbation Tracking," in Proceedings of the 32nd IEEE Conference on Decision and Control, San Antonio, TX, USA, pp. 674-679, December 1993.

[WoMa05] F. Wolf, A. Malony, S. Shende, and A. Morris, "Trace-Based Parallel Performance Overhead Compensation," in Proceedings of the International Conference on High Performance Computing and Communications (HPCC), Sorrento, Italy, September, 2005.

[YaDa00] K. Yaghmour and D. Dagenais, "Measuring and Characterizing System Behavior Using Kernel-Level Event Logging," in Proceedings of the USENIX Annual 2000 Technical Conference, San Diego, CA, USA, pp. 13-26, June 2000.

[YaLi93] J. Yan and S. Listgarten, “Intrusion Compensation for Performance Evaluation of Parallel Programs on a Multicomputer,” in Proceedings of the Sixth International Conference on Parallel and Distributed Systems, Louisville, KY, October 14-16, 1993.

[ZaLu99] O. Zaki, E. Lusk, W. Gropp, and D. Swider, “Toward Scalable Performance Visualization with Jumpshot,” in High-Performance Computing Applications, 13(2):277-288, 1999.

Table 6 Event Counts and Trace File Sizes[1]

PROCS	AVG EVENT COUNT	MAX EVENT COUNT	TOTAL EVENT COUNT	TRACE FILE [1] (GB)	PROCS	AVG EVENT COUNT	MAX EVENT COUNT	TOTAL EVENT COUNT	TRACE FILE [1] (GB)
SMG WITH TAU					**SMG WITH MPE**				
32	8.71E+06	1.05E+07	2.79E+08	6.2	32	8.25E+05	1.14E+06	2.64E+07	0.86
64	9.06E+06	1.14E+07	5.80E+08	13	64	8.41E+05	1.25E+06	5.38E+07	1.7
128	9.41E+06	1.23E+07	1.20E+09	27	128	8.52E+05	1.36E+06	1.09E+08	3.6
256	9.74E+06	1.32E+07	2.49E+09	56	256	8.59E+05	1.47E+06	2.20E+08	7.4
512	1.01E+07	1.42E+07	5.15E+09	115	512	8.66E+05	1.58E+06	4.43E+08	15
SPS WITH TAU					**SPS WITH MPE**				
32	5.93E+08	5.93E+08	1.90E+10	424	32	8.62E+02	1.30E+04	2.76E+04	0.94**
64	2.96E+08	2.97E+08	1.90E+10	424	64	4.57E+02	1.33E+04	2.93E+04	1.1**
128	1.48E+08	1.48E+08	1.90E+10	424	128	2.52E+02	1.39E+04	3.23E+04	1.2**
256	7.41E+07	7.43E+07	1.90E+10	424	256	1.49E+02	1.49E+04	3.81E+04	1.4**
512	3.71E+07	3.72E+07	1.90E+10	424	512	9.69E+01	1.69E+04	4.96E+04	1.7**
SPW WITH TAU					**SPW WITH MPE-C**				
32	3.70E+07	3.71E+07	1.19E+09	26	32	3.70E+07	3.71E+07	1.19E+09	35
64	3.71E+07	3.72E+07	2.37E+09	53	64	3.71E+07	3.72E+07	2.37E+09	70
128	3.71E+07	3.71E+07	4.74E+09	106	128	3.71E+07	3.71E+07	4.74E+09	141
256	3.71E+07	3.71E+07	9.49E+09	212	256	3.71E+07	3.71E+07	9.49E+09	282
512	3.71E+07	3.72E+07	1.90E+10	424	512	3.71E+07	3.72E+07	1.90E+10	565
SPW WITH MPE									
32	9.61E+01	1.09E+03	3.08E+03	129*					
64	9.66E+01	2.15E+03	6.18E+03	257*					
128	9.68E+01	4.26E+03	1.24E+04	449*					
256	9.69E+01	8.48E+03	2.48E+04	897*					
512	9.69E+01	1.69E+04	4.96E+04	1729*					

* KB
** MB

Table 7 Percentage Contribution to Variation

FACTOR	SMG SOLVE: WALL CLOCK TIME		SPS WALL CLOCK TIME		SPW WALL CLOCK TIME		
	TAU	MPE	TAU	MPE	TAU	MPE-C	MPE
Processes	***17.59***	***12.86***	***19.21***	***50.95***	9.28	***18.71***	***81.33***
Write	***33.35***	***26.19***	***14.20***	0.84	***33.97***	***23.49***	0.10
Processes:Write	***14.72***	***13.96***	0.02	0.66	8.34	***17.48***	0.25
Residuals	7.61	***45.84***	***65.64***	***45.26***	***45.81***	***39.44***	***18.06***

Table 8 Correlation of Total Wall Clock Time with Maximum Event Count in a Rank

BUFFER/WRITE		SMG		SPS		SPW		
		TAU	MPE	TAU	MPE	TAU	MPE-C	MPE
1.5	yes	0.96	0.85	0.91	-0.78	0.69	0.80	0.98
8.0	yes	0.97	0.90	0.95	-0.81	0.61	0.76	0.98
1.5	no	0.98	0.98	0.99	-0.70	0.81	0.55	0.96
8.0	no	0.98	0.98	0.99	-0.79	0.74	0.77	0.95

[1] *Trace File column lists: the sum of the individual tracefile sizes for TAU; the size of the single merged trace file for MPE*

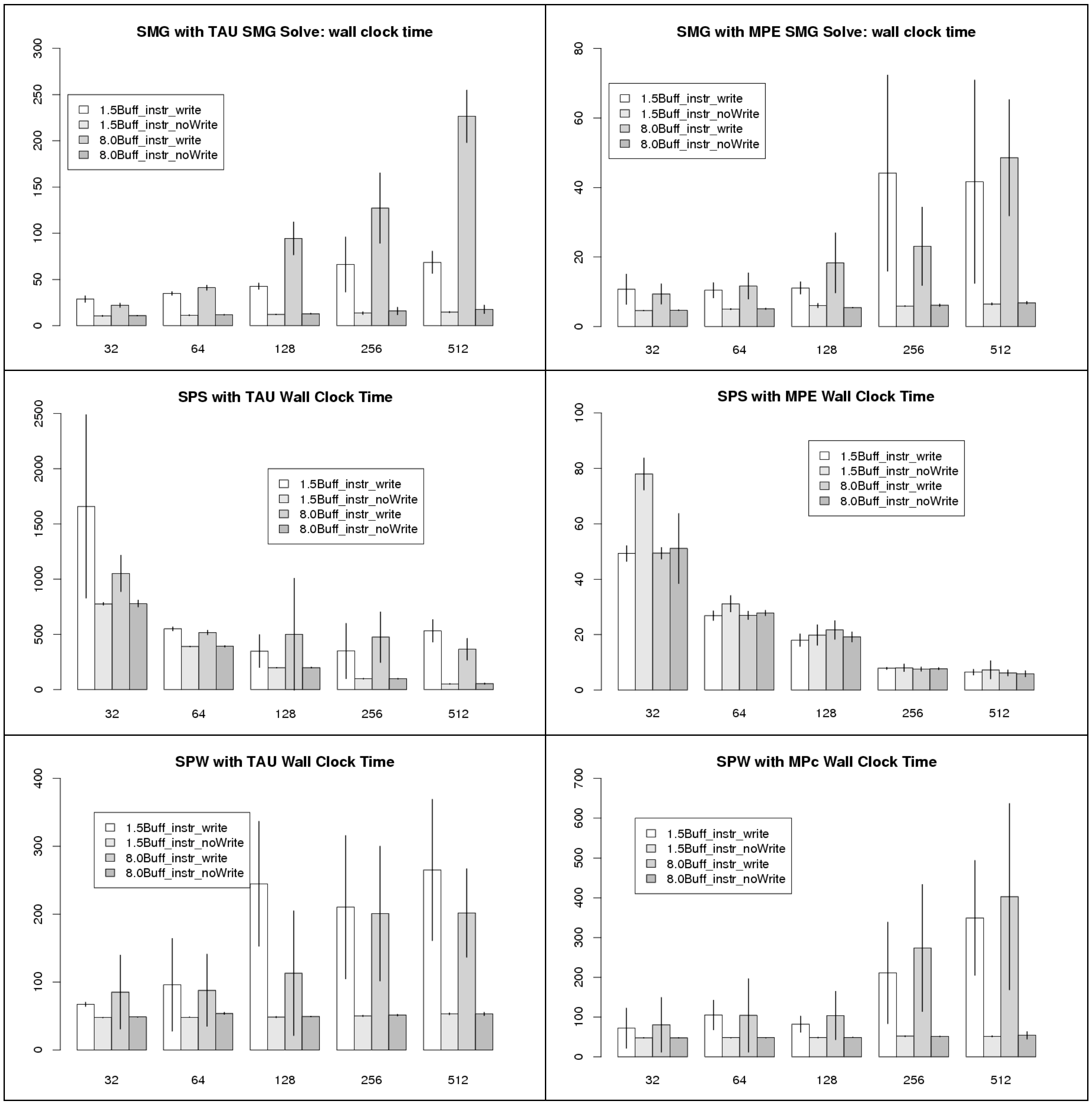

Figure 10. Wall Clock time versus Number of Processors for SMG, SPS, and SPW

This figure shows the wall clock time of the benchmarks as the number of processors increases. The x-axis is the number of processes. The y-axis is time in seconds. We show an error bar of plus and minus one standard deviation for each run set. For SMG, we show the wall clock time of the dominating phase, SMG Solve. For SPS and SPW, we show the total wall clock time reported by the benchmark.