SCALASCA Parallel Performance Analyses of SPEC MPI2007 Applications

Zoltán Szebenyi[1,2], Brian J. N. Wylie[1], and Felix Wolf[1,2]

[1] Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Germany
[2] Aachen Institute for Advanced Study in Computational Engineering Science, RWTH Aachen University, Germany
{z.szebenyi,b.wylie,f.wolf}@fz-juelich.de
http://www.scalasca.org/

Abstract. The SPEC MPI2007 1.0 benchmark suite provides a rich variety of message-passing HPC application kernels to compare the performance of parallel/distributed computer systems. Its 13 applications use a representative cross-section of programming languages (C/C++/Fortran, often combined) and MPI programming patterns (e.g., blocking vs. non-blocking vs. persistent point-to-point communication, with or without extensive collective communication). This offers a basis with which to examine the effectiveness of parallel performance tools using real-world applications that have already been extensively optimized and tuned (at least for sequential execution), but which may still have parallelization inefficiencies and scalability problems. In this context, the SCALASCA toolset for scalable performance analysis of large-scale parallel applications, which has been extended to distinguish iteration/timestep phases, is evaluated with this suite on an IBM SP2 'Regatta' system, and found to be effective at identifying significant performance improvement opportunities.

Keywords: Parallel/distributed systems; Benchmark suite; Performance measurement & analysis tools; Application tracing & profiling.

1 Introduction

Various parallel performance tools studies have considered benchmark suites, such as evaluation of the VAMPIR trace collection and visualization toolset with the 13 applications of the SPEC MPI benchmark suite [1,2,3,4] and the OMPP profiler with the 11 applications of the SPEC OpenMP benchmark suite [5]. Such tools provide in-depth analyses that offer insight into performance and scalability problems indicated by whole execution measurements [6,7]. While tools that aggregate and summarize measurements during execution readily handle long-running complex applications, those that rely on trace collection and analysis are not so fortunate, since trace sizes grow proportionately with the length of measurement (in addition to the orthogonal dimensions of the number of processes/threads, density of traced events and number of metrics associated with each event).

S. Kounev, I. Gorton, and K. Sachs (Eds.): SIPEW 2008, LNCS 5119, pp. 99–123, 2008.

The open-source SCALASCA toolset [8,17] addresses these scalability issues with a compound approach consisting of flexible measurement configuration (including filtering), runtime summarization of measurements during execution, and event trace collection matched with a replay-based trace analysis that exploits the parallelism and distributed-memory resources of the target system [9,10]. From an initial summarization measurement of a fully-instrumented application, an appropriate list of user functions to filter can be determined and specified in subsequent measurements. After verifying that the filter produces an accurate summary measurement (without undue dilation), and that a resulting trace won't be so excessively large as to require highly disruptive intermediate buffer flushing, it can be used for a tracing experiment. Without recompilation or reinstrumentation of the application, straightforward reconfiguration of the measurement runtime system allows traced events to be buffered until measurement completion, after which the trace analyzer replays them in parallel to automatically calculate a rich set of execution performance properties. Both runtime summary and postmortem trace analysis use a common report format, allowing them to be examined with the same interactive analysis report explorer. The library for reading and writing the XML reports also facilitates the development of utilities which process the reports in various ways, such as the extraction of measurements for each process or their statistical aggregation, for the generation of timeline charts and metric graphs, respectively.

This paper presents SCALASCA measurements and analyses of Version 1.0 of the SPEC MPI2007 benchmark suite application kernels on an IBM SP2 'Regatta' system, with particular attention given to the scalability of the applications and the SCALASCA toolset itself, and examination of performance variation between processes and different timesteps/iterations of the applications' executions.

2 Experiment Configuration

2.1 SPEC MPI2007 1.0 Benchmark Suite

Version 1.0 of the SPEC MPI2007 benchmark suite [1,2] was released in June 2007 to provide a standard set of MPI-based HPC application kernels for comparing the performance of parallel/distributed systems' hardware, operating system, MPI execution environment and compilers. The initial release includes 13 applications and a 'medium-sized' reference dataset (MPIm2007) for benchmarking runs requiring up to 2GB of memory per process and configurable for up to 512 processes.

Table 1 summarizes the 13 applications of the MPI2007 suite, showing that they derive from a wide variety of subject areas and are implemented using a representative cross-section of programming languages (C/C++/Fortran, often combined). From the MPI usage breakdown in the table, it can be seen that a variety of MPI functions are used at many locations ('sites') in the source code, however, performance analysis can concentrate on the smaller number of communication and synchronization functions (shown as c&s/used 'funcs')

Table 1. SPEC MPI2007 1.0 applications' coding and subject area

	Program		MPI			
Application code	language	LOC	funcs	sites	paths	Application subject area
104.milc	C	17987	9/18	51	111	Lattice quantum chromodynamics
107.leslie3d	F77,F90	10503	8/13	43	12	Combustion dynamics
113.GemsFDTD	F90	21858	9/16	237	21	Computational electrodynamics
115.fds4	F90,C	44524	8/15	239	8	Computational fluid dynamics
121.pop2	F90	69203	11/17	158	173	Oceanography
122.tachyon	C	15512	8/16	17	8	Computer graphics: ray tracing
126.lammps	C++	6796	12/25	625	41	Molecular dynamics
127.wrf2	F90,C	163462	7/23	132	62	Numerical weather prediction
128.GAPgeofem	F77,C	30935	8/18	58	13	Geophysics finite-element methods
129.tera_tf	F90	6468	9/13	42	17	Eulerian hydrodynamics
130.socorro	F90	91585	11/20	155	147	Quantum chemistry
132.zeusmp2	C,F90	44441	11/21	639	85	Astrophysical hydrodynamics
137.lu	F90	5671	10/13	72	24	Linear algebra SSOR

and the distinct program call-paths on which they are actually executed during benchmark runs ('paths').

Table 2 tallies the MPI functions used by 32-way benchmark executions, and shows that a similarly diverse range of MPI programming patterns are implemented, e.g., blocking, vs. non-blocking vs. persistent point-to-point communication, with or without extensive collective communication, etc. (SPEC rules allow only MPI parallelization, so auto-parallelization capabilities of compilers must be disabled, at least in this initial version of the benchmark suite.) The suite therefore provides a comprehensive test, both for MPI benchmarking purposes, but also for examining the effectiveness of parallel performance tools with real-world applications.

2.2 IBM SP2 Regatta p690+ System

The John von Neumann Institute for Computing 'JUMP' system [11] hosted by Jülich Supercomputing Centre consists of 41 IBM SP2 p690+ frames, each with 16 dual-core 1.7GHz Power4+ processors and 128GB of shared main memory, connected via IBM High Performance Switch. At the time measurements were made, the system was running AIX 5.3, with IBM's POE 4.2 MPI and GPFS filesystem, and use of compute nodes managed via LoadLeveler.

The available IBM XL compiler suites (versions 7.0/8.0 for C/C++ and 9.1/10.1 for Fortran) were unable to compile and/or link some of the SPEC MPI2007 applications when the build was configured using the specification provided for them with the benchmark distribution. In such cases, aggressive optimization options were progressively removed until a viable application executable was produced. Full optimization of the code and run-time environment were neither essential nor particularly desirable for our purposes, as the study

Table 2. MPI function calls used by 32-way SPEC MPIm2007 executions on JUMP

	Irecv	Isend	Recv	Send	Wait	Waitall	Waitany
104.milc	359340	359340			718680		
107.leslie3d	3201600	3201600			320160		
113.GemsFDTD			3316	3316			
115.fds4			35271	35271		151264	
121.pop2	558007700	558007700				319663712	
122.tachyon							
126.lammps	196544		9152	205696	196544		
127.wrf2	6508380		10106	6518486	6508380		
128.GAPgeofem	6099876	6099876				1404288	
129.tera_tf	1989504		360	1989864	1989504		
130.socorro	3286178			3286178			3286178
132.zeusmp2	845056	845056				249888	
137.lu	19000		7600320	7619320	19000		

	Sendrecv	Recv_init	Send_init	Start	Startall	Testsome	Scan
113.GemsFDTD	1240000						
122.tachyon		16158	16158	6536	1	223	
126.lammps							32

	Allgather	Allgatherv	Allreduce	Barrier	Bcast	Gather	Reduce
104.milc			17700	62	122		
107.leslie3d			140832	1088			64
113.GemsFDTD				160	292000		128
115.fds4	303040			320		8512	
121.pop2			26080640	8640	9664		
122.tachyon	32			32			
126.lammps			1696	64	1888		
127.wrf2					67488		
128.GAPgeofem			2016224		352		
129.tera_tf			60352	15520	1184		
130.socorro	512	7936	37536		9696		1088
132.zeusmp2			12864	96	1280		64
137.lu			224	32	288		

	Cart_ create	Comm_ split	Comm_ create	Comm_ free	Comm_ dup	Comm_ group	Group_ range_ incl
104.milc		32					
113.GemsFDTD		32					
121.pop2			96			96	96
126.lammps	32			32			
128.GAPgeofem					32		
130.socorro					224		
132.zeusmp2	32	32					
137.lu		32					

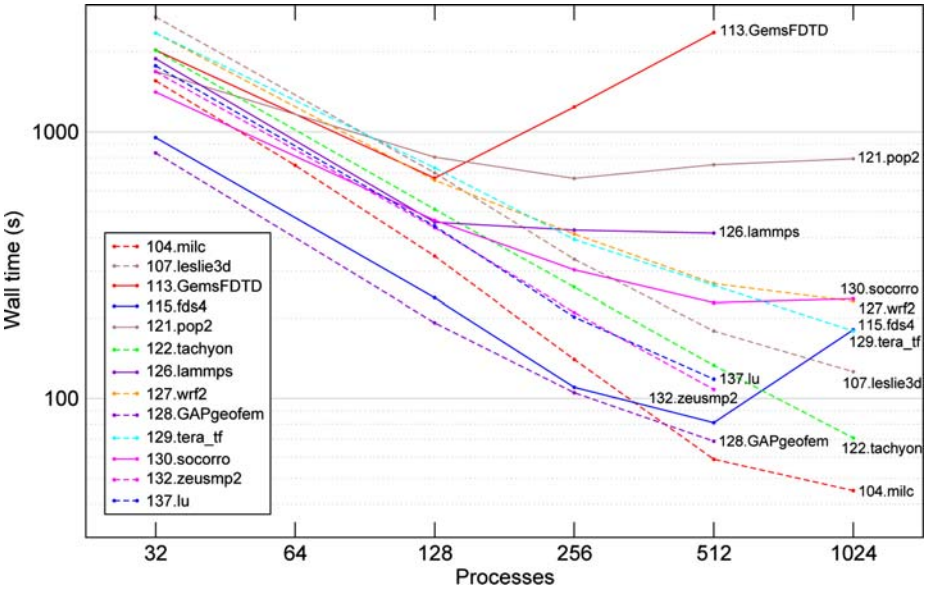

Fig. 1. SPEC MPIm2007 1.0 benchmark execution times with different numbers of processes on the IBM SP2 system 'JUMP.' Eight of the benchmarks (shown with dashed lines) have good speedup, up to 1024 processes when supported by the benchmark and 512 processes otherwise. The remaining five benchmarks (shown with solid lines) have clear scaling problems. *126.lammps* uses a maximum of 140 processes (idling any excess provided) and therefore shows no significant speedup beyond 128 processes. *130.socorro* and *115.fds4* both show good speedup to 512 processes, before respectively having small and significant slowdowns. Finally, *121.pop2* only scales to 256 processes before slowing down and *113.GemsFDTD* only to 128 processes before its dramatic performance breakdown.

is more focussed on 'typical' application performance in a representative HPC environment than benchmarking.

Figure 1 shows a graph of the benchmark execution times with different numbers of processes, on a log–log scale, from which the scalability of each benchmark can be determined. To reduce the impact of variability in run times (due to non-dedicated use of the communication switch and filesystem in the production configuration of the JUMP system), the best run time of several measurements is taken although this is contrary to the SPEC benchmark rules. (Including confidence intervals in the graphs and tables would be appropriate in a comprehensive study, however, these have been omitted to reduce unnecessary clutter and clarify the underlying behaviour.)

While around half of the benchmarks scale well, it is clear that certain others have very limited scalability, before no further speed-up is possible or performance degrades unacceptably. Although no tuning has been done for JUMP, and measurements were taken on a non-dedicated production system, from review of

published benchmark results [1] the same scalability limitations are seen to be common to specially 'tuned' benchmark measurements on dedicated systems.

Of course, analyzing the performance of optimally-tuned applications that scale perfectly has much less value than identifying potential opportunities for improvement of applications with problems, and is key to producing better performing and more scalable applications.

2.3 SCALASCA Toolset

SCALASCA is an open-source toolset for scalable performance analysis of large-scale parallel applications [8,17] developed by Jülich Supercomputing Centre in conjunction with the University of Tennessee. Version 1.0 includes integrated runtime measurement summarization and selective event tracing [9] with automatic trace analysis based on parallel replay [10], to ensure scalability for long-running and highly-parallel MPI, OpenMP and hybrid applications.

When the SCALASCA instrumenter is prepended to each application compile and link command, it produces fully-instrumented executables without modifying or inhibiting compiler optimizations. This exploits capabilities for function entry and exit instrumentation provided by most (but not all) modern compilers, and the standard PMPI library interposition interface. A source preprocessor is also provided for OpenMP pragma/directive and annotated region instrumentation (though not used in this work). Manual annotation of significant code regions (e.g., initialization) can also be done with a macro-based user API, which has been extended for annotating repetitive phases (such as solver iterations or time-steps).

SCALASCA measurement collection and analysis is performed by a nexus that is also prefixed to the normal application execution command-line, whether part of a batch script or interactive run invocation. Experiments with an instrumented executable can be configured to collect runtime summaries and/or event traces (optionally including hardware counters), with the latter traces automatically analyzed with the same number of processes as used for measurement. Both summary and trace analyses are generated in the same profile format, which can be interactively explored with the SCALASCA analysis report examiner GUI (shown in Figures 10&11). Command-line tools are also provided for processing analysis reports, e.g., to produce filters containing lists of functions to ignore for improved measurement configuration, and new prototype tools are being developed for graphing and charting metrics calculated for repetitive phases.

Table 3 shows the SPEC MPIm2007 application execution characteristics determined from SCALASCA runtime summarization experiments. Application programs are seen to typically consist of hundreds to thousands of global timesteps or solver iterations, with the farming-based *122.tachyon* being an exception. Although *130.socorro* only does 20 iterations, it has by far the most complex call-tree and the deepest frame depth (with MPI communication down to depth 18, one further than *127.wrf2*): some highly recursive functions in the initialization phase of *127.wrf2* were filtered out and are not counted here. At tens of gigabytes per process rank, complete traces of either *122.tachyon* or

Table 3. SPEC MPIm2007 1.0 applications' 32-way execution characteristics

	Program execution			RSS	Trace buffer content (MB)				Filter
Application code	steps	depth	callpaths	(MB)	total	MPI	filter	residue	funcs
104.milc	8+243	6/6	255/257	341	2683	1.7	2626	57	4
107.leslie3d	2000	3/3	40/40	1078	1437	15.	1422	16	6
113.GemsFDTD	1000	4/5	166/185	505	3619	5.9	3582	37	1
115.fds4	2363	1/8	149/151	209	122	2.1	117	6	6
121.pop2	9000	6/6	403/403	748	6361	2494.	2606	3841	6
122.tachyon	N/A	3/3	25/27	676	59884	0.7	59809	75	5
126.lammps	500	6/6	162/162	401	291	0.8	290	1	9
127.wrf2	1375	17/22	4951/4975	297	1109	0.4	1106	5	69
128.GAPgeofem	235	4/4	44/44	361	996	33.	971	34	2
129.tera_tf	943	3/4	57/59	74	2459	10.	1628	831	4
130.socorro	20	18/23	10350/10352	148	10703	13.	10587	120	21
132.zeusmp2	200	5/5	171/179	377	5	3.4	—	3	0
137.lu	180	4/4	48/49	384	42	28.	—	28	0

130.socorro would be prohibitively large, however, specifying a few functions to filter reduces their requirements to around 100MB/process. Many of the other applications also benefit from substantial reduction of measurement overheads when one or more of their user functions are filtered. Unfortunately, a full execution measurement of the MPI-dominated *121.pop2* remains intractible even when only MPI functions are traced, therefore it was necessary to reduce the number of steps it does from 9000 to 2000 (by modifying its input file).

Table 4 presents the SPEC MPIm2007 application execution times for uninstrumented runs and for a variety of Scalasca measurement experiments with 32, 128 and 512 processes. When measurements are being collected, run times are naturally longer than the uninstrumented execution times, due to dilation introduced by instrumentation and measurement processing, however, this can be minimized by providing appropriate filters specifying functions to be ignored during measurement (as determined by an initial summarization measurement). When an initial full summarization measurement is not practical, as was the case with *122.tachyon*, a filter could be determined from a shorter or smaller execution. (Although the dilation remains serious, further reduction was not pursued since *122.tachyon* was ultimately not particularly interesting.) For *132.zeusmp2* and *137.lu* filtering was neither necessary nor desirable.

As well as reducing measurement dilation, filtering is also appropriate for reducing the trace buffer capacity requirements, to avoid highly disruptive intermediate flushes of trace buffers to disk during measurement: examples of catastrophic disruption from intermediate trace flushing are detailed in [4]. Furthermore, very large traces are also awkward to analyze, so judicious filtering balances what measurements are collected and analyzed with what is omitted on expediency grounds. Functions that have been filtered in this way are 'invisible' during analysis, as if they had been 'in-lined.' Even with all user functions filtered (i.e., measuring only MPI functions), the 2.5GB/rank trace buffer capacity

Table 4. SPEC MPIm2007 1.0 applications' execution times in wallclock seconds with 32, 128 and 512 processes on the p690+ cluster for a variety of instrumentation and measurement/analysis configurations. 'None' is a reference run with neither instrumentation nor measurement (beyond elapsed time), whereas the additional columns refer to measurements of fully-instrumented versions (i.e., using automatic function instrumentation by the compiler and MPI library interposition instrumentation), sometimes augmented with user-defined phase annotations (p), where measurement was configured for runtime summarization only (Sum) or runtime summarization combined with event tracing (Trace). Measurements marked (f) used filtering of selected user functions with excessive overheads. After trace collection during measurement, additional time is required to dump buffered trace event records to disk (Td) for subsequent automatic trace analysis (Ta), both done in parallel with the total trace data.

	Instrumentation/Measurement					Tracing	Trace
Application code	None	Sum	Sum+f	Sum+pf	Trace	Td+Ta	(GB)
32							
104.milc	1556	2140	1616	—	1611	13+50	1.587
107.leslie3d	2704	2945	2807	2892	2787	43+113	0.403
113.GemsFDTD	2028	2680	2042	2111	2102	57+144	0.634
115.fds4	951	1010	960	957	959	92+141	0.130
121.pop2	1687	2415	2176	2104	N/P	—+—	—
121.pop2 (2000)	398	N/A	514	N/A	518	124+2734	13.613
122.tachyon	2024	N/P	6016	—	6023	1+68	0.007
126.lammps	1883	1988	1899	2001	1963	41+74	0.038
127.wrf2	2352	2945	2499	2475	2550	425+907	18.138
128.GAPgeofem	833	984	879	884	874	14+182	0.670
129.tera_tf	2399	2583	2458	2390	2395	17+71	24.737
130.socorro	1411	3990	1631	1701	1703	120+373	3.420
132.zeusmp2	1683	1727	—	—	1729	28+67	0.113
137.lu	1771	1815	—	—	1910	13+159	1.100
128							
113.GemsFDTD	670	—	1033	—	1038	103+216	0.944
512							
104.milc	59	—	63	—	69	5+7	0.827
107.leslie3d	179	—	193	—	199	310+343	7.037
113.GemsFDTD	2363	—	N/A	—	N/A	—+—	—
115.fds4	81	—	86	—	88	272+743	1.050
121.pop2	752	—	1072	—	N/P	—+—	—
121.pop2 (2000)	182	—	226	—	326	1380+2627	103.646
122.tachyon	133	—	383	—	380	2+27	0.069
126.lammps	416	—	445	—	434	233+360	0.167
127.wrf2	269	—	300	—	310	1878+2535	107.929
128.GAPgeofem	69	—	82	—	87	50+333	15.216
129.tera_tf	265	—	287	—	298	163+316	72.381
130.socorro	228	—	263	—	268	635+913	25.756
132.zeusmp2	108	112	—	—	115	5+19	2.084
137.lu	118	119	—	—	119	36+126	19.493

requirements of *121.pop2* were impractical for tracing a full execution, therefore measurements were repeated with only 2000 rather than the full 9000 steps.

For applications with identifiable repetitive phases, corresponding to global timesteps or solver iterations, additional annotation instrumentation was manually inserted into the source code. This was possible for all except *122.tachyon* which is based on a task-farming parallelization, and *104.milc* which has a complex structure of nested loops and branches. The overhead of this additional instrumentation during measurement is found to be much less than the run-to-run variation of the applications themselves, and the phase markers can be exploited in subsequent analyses.

After an initial set of 32-way measurements, from which appropriate measurement filters could be determined, 128-way and 512-way measurements were then taken. (512-way measurements were skipped for *113.GemsFDTD* due to its adverse scaling.) Although the measurement times for runtime summarization and trace collection are seen to scale in proportion to the uninstrumented application execution time, trace sizes and corresponding trace handling (dumping of buffers and post-mortem analysis) generally grow more expensive. In a few cases, however, traces actually become smaller or the use of parallel I/O decreases trace handling time. For example, *121.pop2* trace sizes and writing times grew by factors of 7.6 and 11 respectively, however, parallel trace analysis time actually slightly improved with 8 times the number of processes.

3 Results and Analyses

The final automatic trace analysis reports for each SPEC MPIm2007 benchmark application execution (with 32 processes), including functions and annotated phases, were postprocessed to extract the aggregate and individual process execution behaviour of each application-specific phase (corresponding to global timesteps or solver iterations as appropriate). *104.milc* and *122.tachyon* are excluded from this analysis.

Scalasca analyses automatically determine a variety of performance metrics for each application call-path and thread of execution, which are concisely presented in hierarchical trees (as shown in Figures 10&11). Simple *Visits* counts and MPI message-passing statistics (e.g., numbers of sends and receives or collective operations and associated *Bytes transferred*) complement metrics derived from measured times. MPI *Communication* and *Synchronization* times can be distinguished from total *Execution* time, and further split into times for *Point-to-point* and *Collective* operations. These summary metrics, which are straightforward to calculate during measurement, can be augmented by specialized metrics that can only be determined from analysis of traces searching for patterns of events indicative of inefficiencies.

Eight of the remaining 11 applications are treated collectively in Figures 2–5, whereas *107.leslie3d*, *129.tera_tf* and *132.zeusmp2* show particularly interesting execution behaviour and are examined in more detail afterwards.

The left column in Figures 2–5 graphs total *Execution* time and MPI *Communication* time for each iteration phase. The values for the process(es) with the largest times are shown red, the median shown blue, and the shortest shown green. In most cases, no significant difference is apparent in total *Execution* time between the fastest and slowest processes, and the graphs appear uniformly green. *137.lu* is one of the exceptions, consistently having an observable difference in every iteration, whereas a difference is only apparent in the first iteration of *113.GemsFDTD* and some of the iterations of *121.pop2*. Variation in MPI *Communication* time is much more pronounced, both between iterations and between processes within iterations, exemplified by *115.fds4* and *113.GemsFDTD* respectively.

Whereas most applications show a stable constant execution time for each iteration (sometimes with the first and/or last iteration being distinguished), some reveal gradually deteriorating performance (e.g., *126.lammps* and *127.wrf2*). Much larger *Execution* time of certain iterations of *126.lammps* at regular intervals are also clearly distinguished, and from further analysis found to correlate to more point-to-point communication every 20th iteration and collective communication every 100th. The execution of *127.wrf2* is clearly dominated by its 1st and 1201th iterations, however, there are also significant iterations with collective communication every 300 iterations.

The right column in Figures 2–5 shows total *Execution* time and MPI *Communication* time for each iteration phase as a timeline chart for each process. In each chart, the value for the largest time is shown in dark red, with the other values on a progressive scale down to light yellow, and white used if there is no value for a particular entry. (This colour scale is shown at the bottom of Figures 10&11.) Globally consistent behaviour is generally apparent, including variation per iteration which appears as peaks in the graphs on the left.

137.lu can again be readily distinguished by its broadly non-deterministic variation of *Execution* time across processes in any iteration, however, MPI *Communication* time reveals a more complex story. Certain processes consistently have much shorter *Communication* times than the others, indicative of load imbalance. More dramatic load imbalance is evident from the horizontal stripes in the MPI *Communication* time chart of *113.GemsFDTD*, where processes with ranks 7, 30 & 31 consistently take longer than the others: the latter are found not to participate in certain local update operations and consequently are always early when they must communicate with partners. Similar striping can also be seen in *128.GAPgeofem* and on odd-numbered process ranks of *126.lammps*. For *121.pop2* it is predominantly higher numbered process ranks that have longer MPI *Communication* times.

Note that the phase annotations do not explicitly synchronize processes, such that the time for a particular iteration on one process can vary significantly from that of its peers, however, inter-process communication results in loose synchronization in those cases where explicit collective synchronization is not used by the application itself in each iteration.

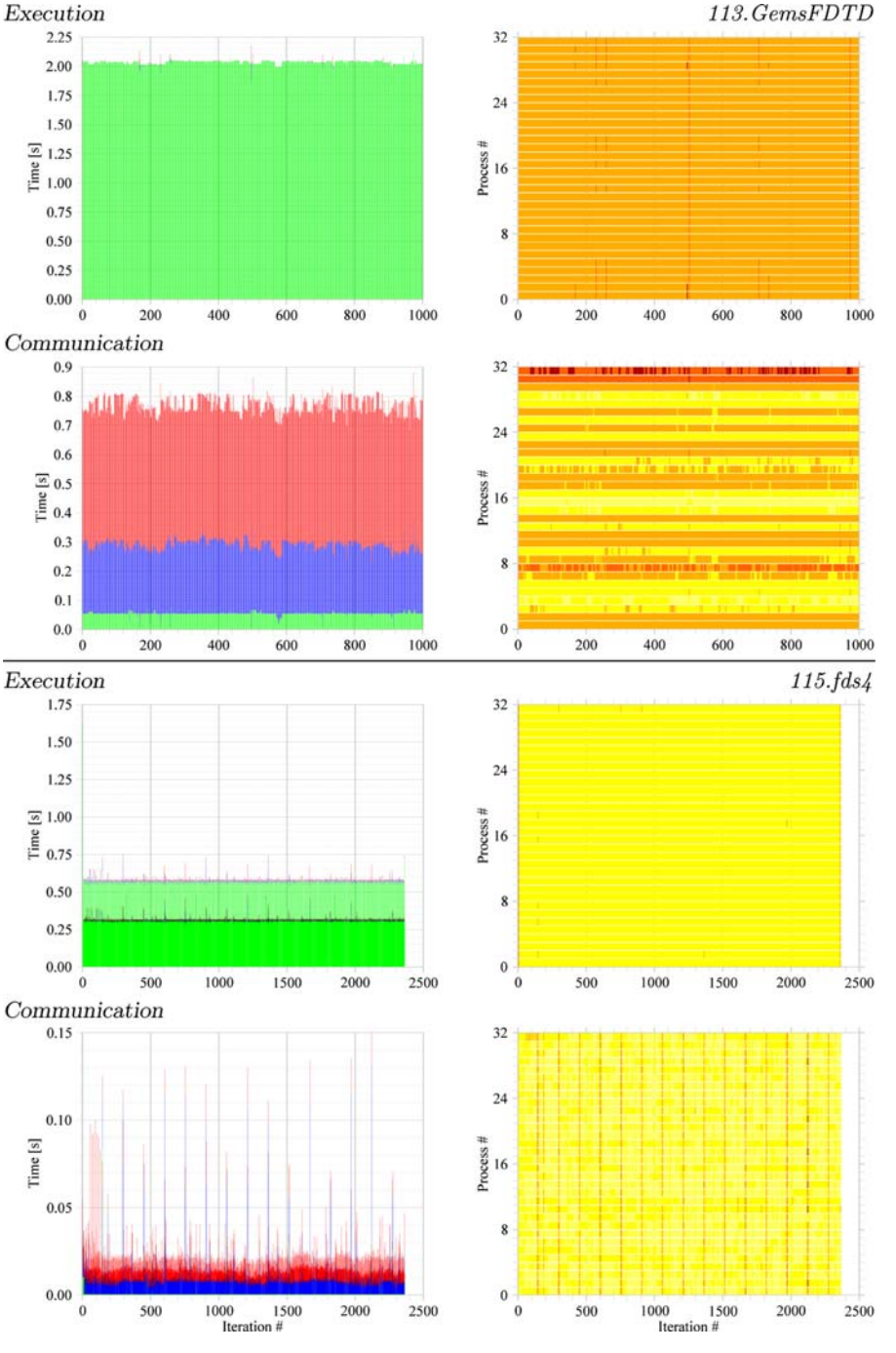

Fig. 2. SPEC MPIm2007 *113.GemsFDTD* and *115.fds4* iteration time metrics

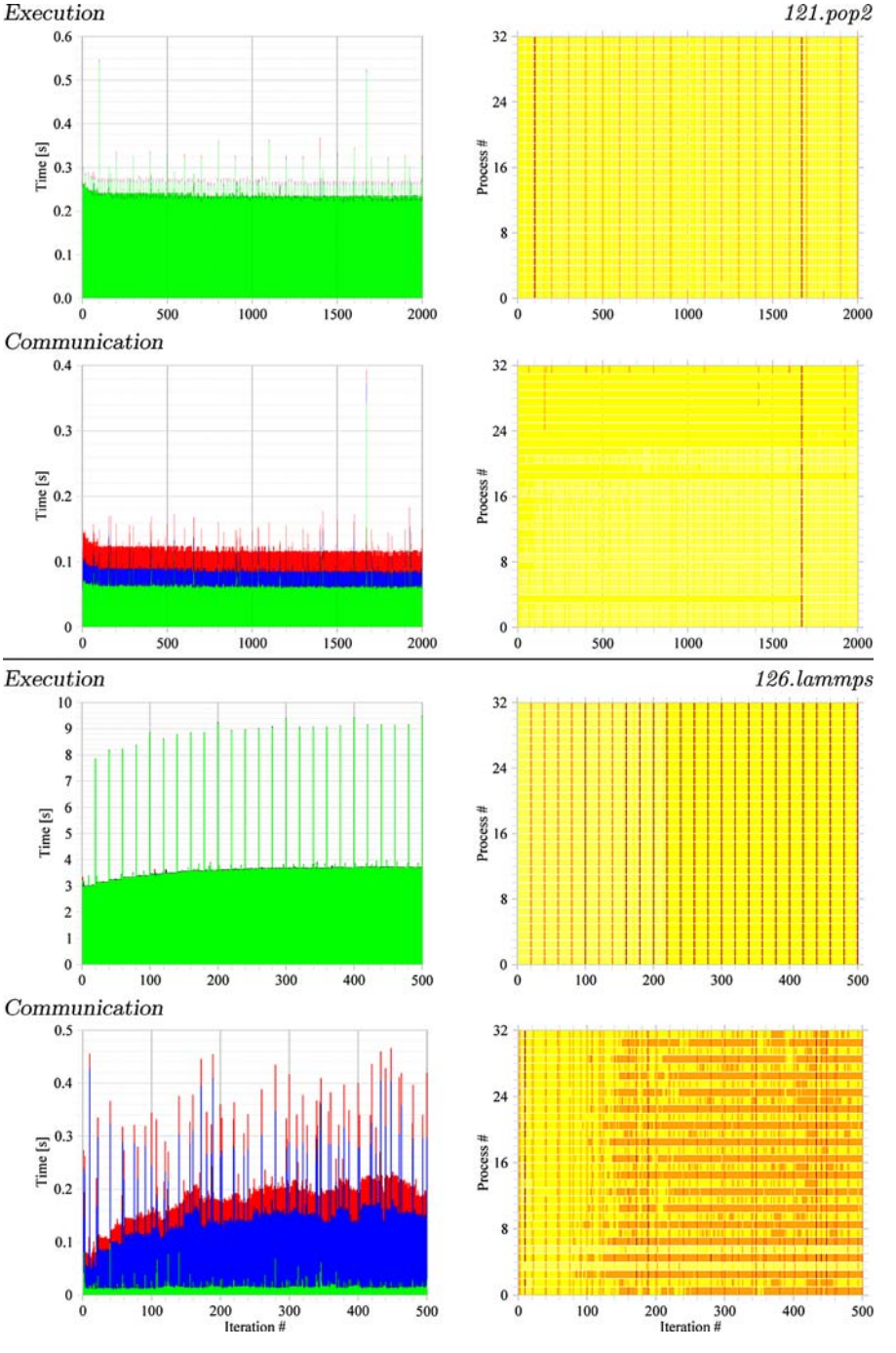

Fig. 3. SPEC MPIm2007 *121.pop2* and *126.lammps* iteration time metrics

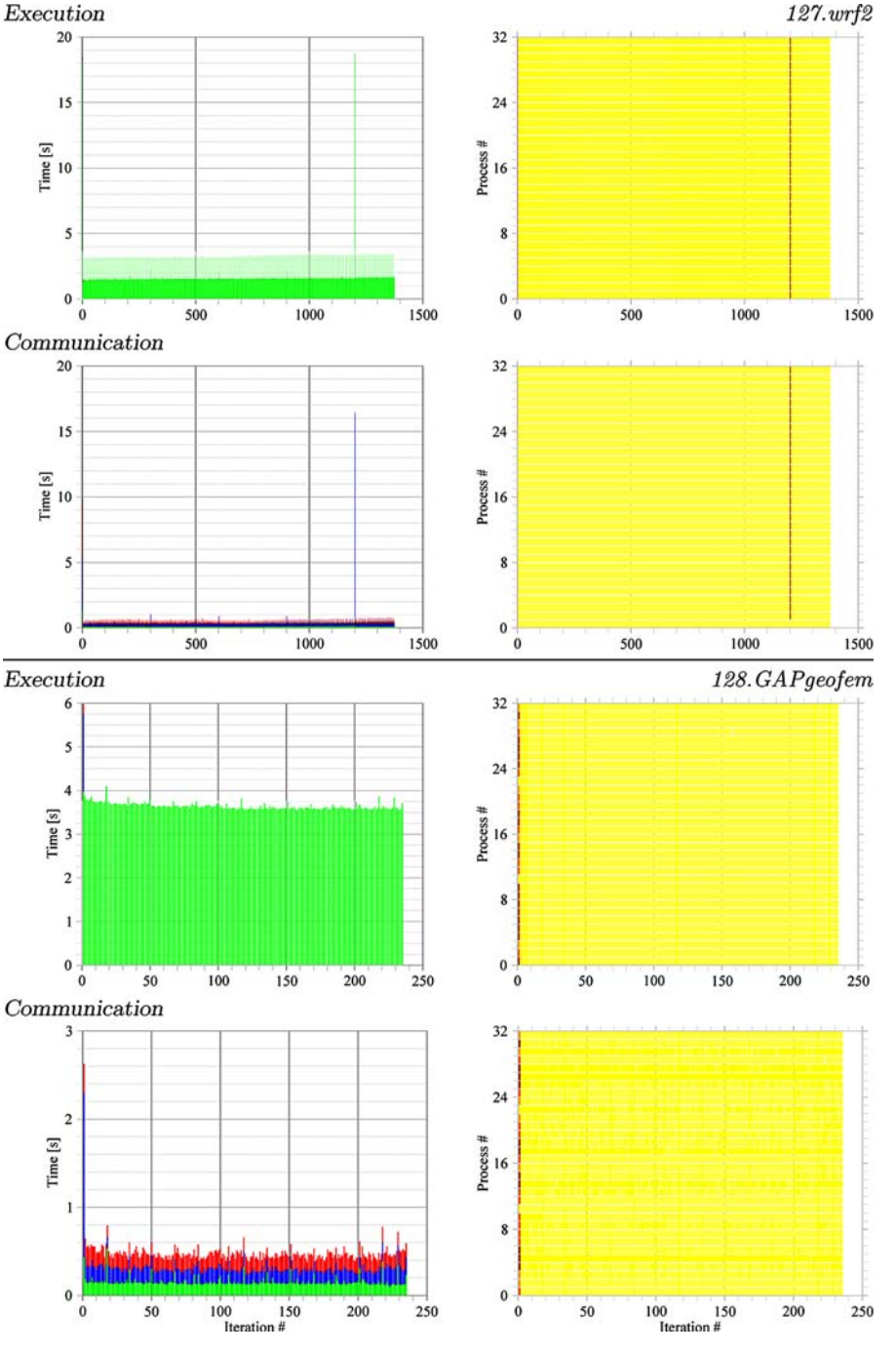

Fig. 4. SPEC MPIm2007 *127.wrf2* and *128.GAPgeofem* iteration time metrics

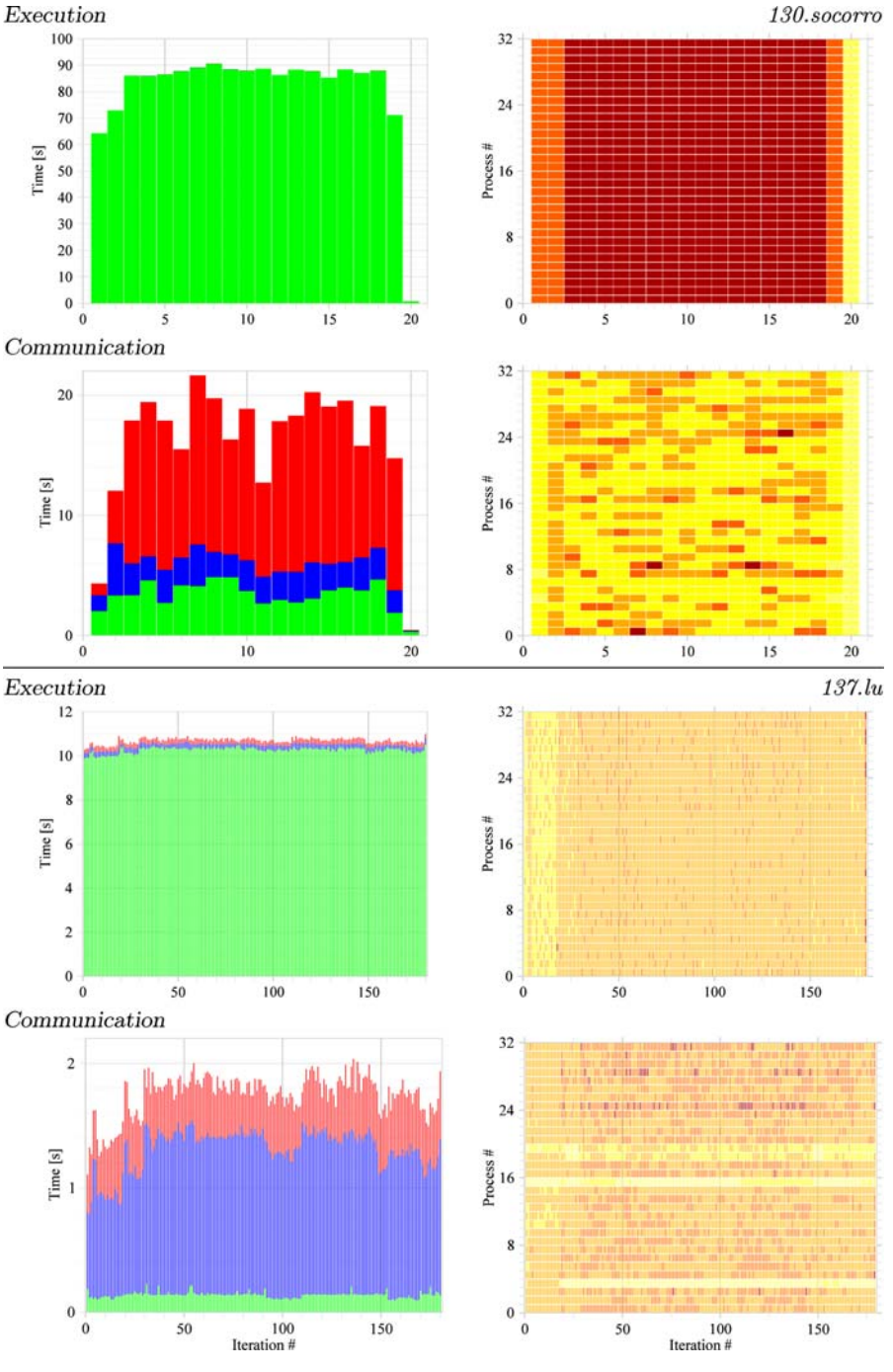

Fig. 5. SPEC MPIm2007 *130.socorro* and *137.lu* iteration time metrics

3.1 *107.leslie3d*

Iteration statistics graphs and timeline charts for a variety of performance metrics measured for *107.leslie3d* are shown in Figure 6. The *Execution* time metric shows a clear transition at iteration number 1015, with iterations taking roughly 1.35s before and 1.37s afterwards. This is seen to correlate with the median *Point-to-point Communication* time metric increasing from 0.10s to 0.13s after iteration 1015. Furthermore, the fraction of *Point-to-point Communication* time considered to be due to early receivers blocked waiting on senders to initiate communication (*Late Sender*) is clearly anti-correlated with the performance degradation and mostly restricted to processes with ranks 10 & 11. They are also found to be receiving messages in non-optimal order: *Late Sender / Wrong Message Order* during that period indicates that a message already in transit could have been received instead of waiting for another not yet initiated. The *Collective Communication* time metric doesn't show a transition, but has a prominent peak value for iteration 1015. Although there is a significant variation in the number of call-path *Visits* and *Bytes transferred* by processes, they remain constant thoughout, and therefore don't explain the dramatic transition. Additional *107.leslie3d* measurement experiments showed similar transitions, though with varying onset, severity, and affected processes, suggesting that an external influence is responsible for this significant disruption in execution performance. While other benchmarks seem less susceptible to this effect, it has also been identified in *121.pop2* and *126.lammps* measurements. One explanation could be process migration away from its local memory within the SP2 SMP node, however, an AIX API to determine processor bindings for processes has not yet been identified to be able to investigate this.

3.2 *129.tera_tf*

Iteration statistics graphs and timeline charts for a variety of performance metrics measured for *129.tera_tf* are shown in Figure 7. The *Execution* time metric shows a progressive increase from 1.2s to 2.9s for iterations, with occasional non-deterministic peaks. This increase is largely explained by the increase in maximum *Point-to-point Communication* time (0.1s growing to 1.5s) during the course of execution: the maximum *Collective Communication* time also grows to 0.4s. Both graphs show intriguing fine-scale variations from iteration to iteration amid larger-scale progressive trends

Blocking time of early receivers waiting for senders to initiate communication, considered *Late Sender* time, is seen to contribute significantly to *Point-to-point Communication* time, and found to affect different processes at different stages of execution. A 'hump' in maximum *Late Sender* time for iterations between 240 and 450 is remarkably prominent. Not shown, *Collective Synchronization* time is insignificant, with only the final iteration containing MPI_Barrier calls, and variation in the number of callpath *Visits* and *Bytes transferred* by processes is clearly evident, but constant throughout.

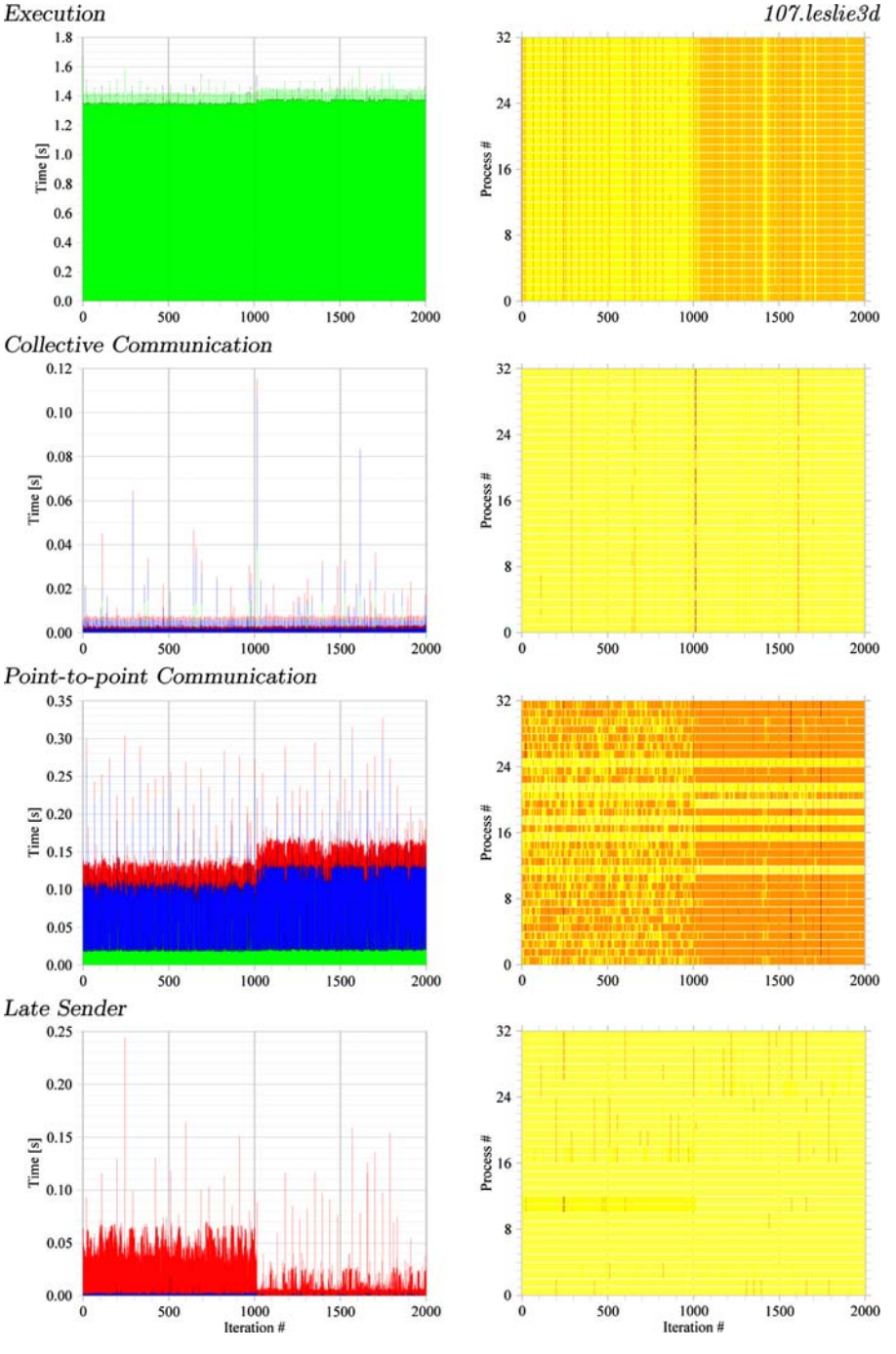

Fig. 6. Graphs and charts of SPEC MPIm2007 *107.leslie3d* iteration time metrics

3.3 *132.zeusmp2*

Although *132.zeusmp2* demonstrated extremely good scalability to 512 processes, SCALASCA analyses of 32-way experiments identified potentially important inefficiencies which warranted further investigation. Further experiments were therefore collected with 512 processes, to examine how these inefficiencies develop at larger scale.

Iteration statistics graphs and timeline charts for a variety of performance metrics measured for 512-way execution of *132.zeusmp2* are shown in Figures 8 and 9. The *Execution* time metric (upper left) shows a progressive increase from 0.43s to 0.49s for timesteps (after the initial timestep), with occasional outliers taking a little longer. Following down the column of metric graphs, this behaviour is explained by the median aggregate *Communication* time, which increases from 0.05s to 1.0s during the course of execution, with occasional iterations taking almost double as long. This is predominantly *Point-to-point Communication* time, with around a fifth due to *Collective Communication* time. Minimum time per iteration for the point-to-point operations fluctuates around 0.02s. Notably, while *Collective Communication* time was negligible during 32-way runs of *132.zeusmp2*, it has grown to be relatively significant in this 512-way experiment.

The bottom graphs of Figure 9 show that blocking time of early receivers waiting for senders to initiate communication, i.e., *Late Sender* time, contributes around half of the *Point-to-point Communication* time, and around half of it is for receiving messages out of order (i.e., *Late Sender / Wrong Message Order*). Multiple iterations are seen to have elevated times across most of the processes, and account for pronounced peaks in the median time, e.g., for iterations 39, 53 & 179, in both of these metrics. These higher communication times also carry through to observable delays in total *Execution* time for those iterations. Variation in the number of callpath *Visits* and *Bytes transferred* by process is clearly evident, but constant throughout, so provide no further insight into this dynamic execution behaviour.

The detailed metric charts and graphs provide a comprehensive view of the execution performance across processes and through time for annotated iterations and timesteps, which complements the profile-oriented SCALASCA analysis presentation.

From the runtime summarization report shown by the SCALASCA analysis report examiner GUI in Figure 10, MPI *Communication time* is found to be 18.4% of total execution time. 70% of this is *Point-to-point Communication time*, however, *Collective Communication time* which was insignificant with 32 processes now contributes the rest: this might be indicative of deteriorating load balance or lower efficiency of collectives using the IBM High Performance Switch when using multiple SMP nodes. MPI *Point-to-point Communication time* is largely concentrated in MPI_Waitall calls in the three routines bvalemf1, bvalemf2 and bvalemf3 on the call-path to hsmoc via ct and transprt. For these MPI_Waitall calls, there is a substantial variation across the 512 processes. with

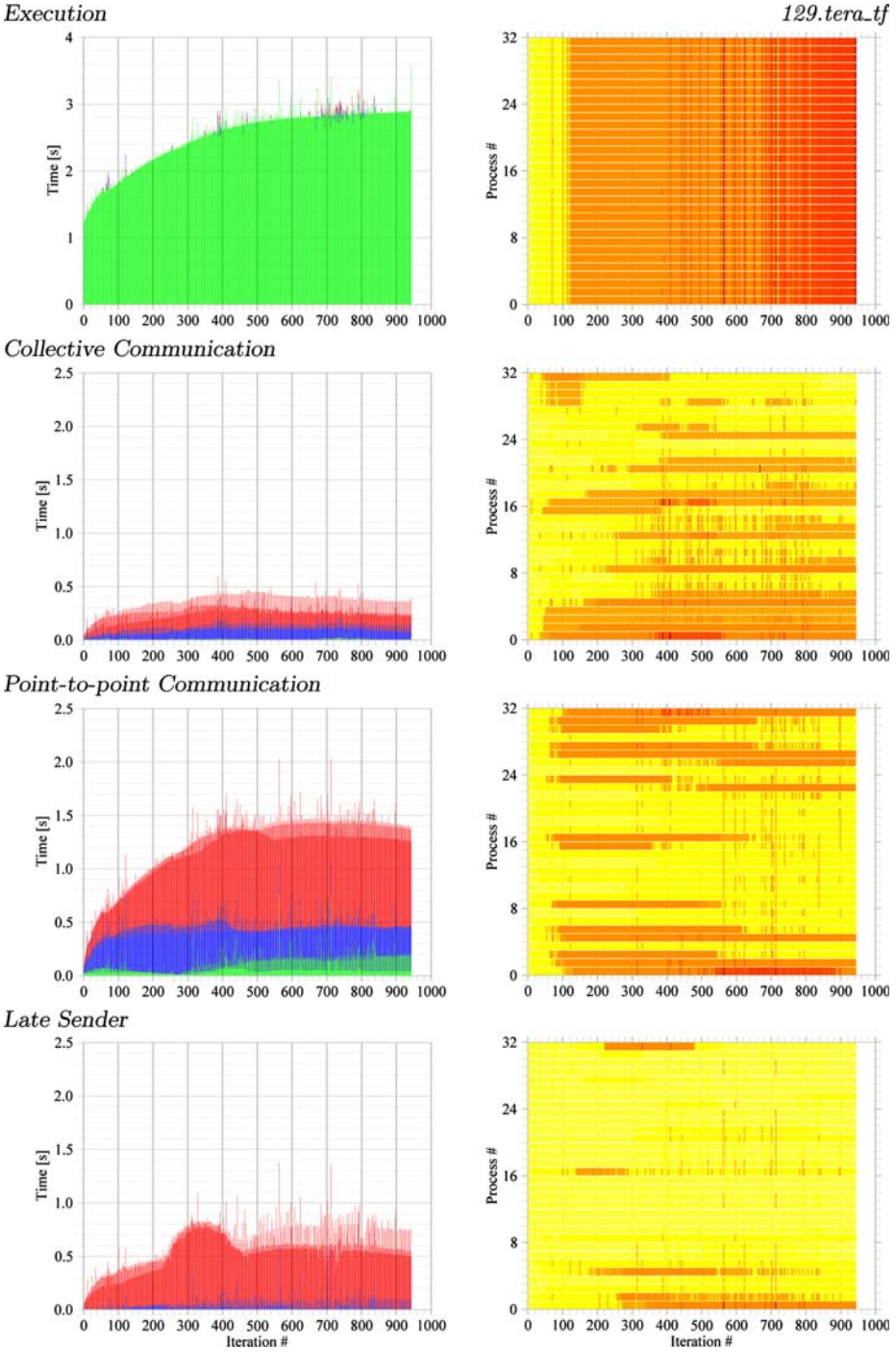

Fig. 7. Graphs and charts of SPEC MPIm2007 *129.tera_tf* iteration time metrics

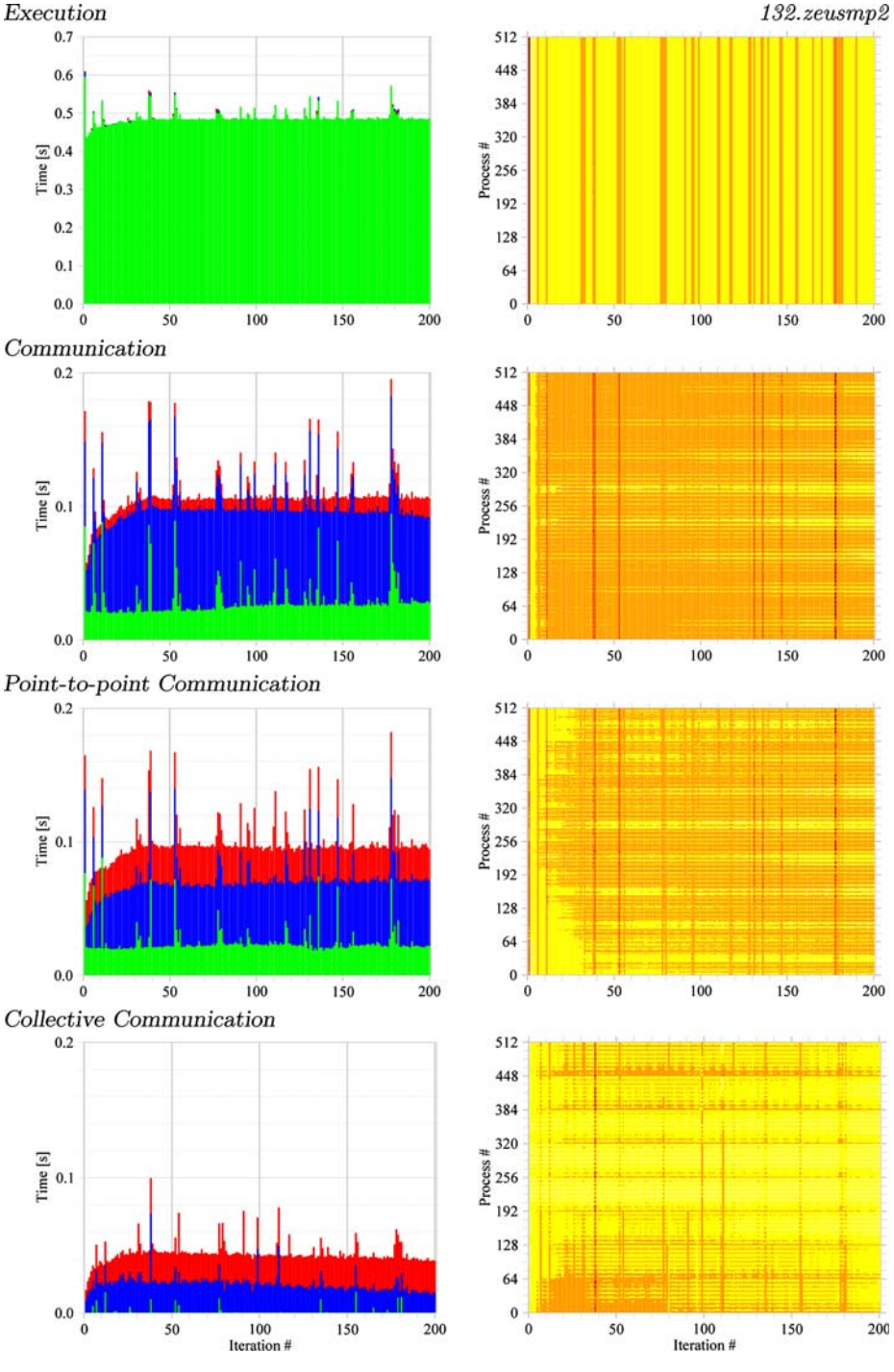

Fig. 8. Graphs and charts of SPEC MPIm2007 *132.zeusmp2* timestep metrics

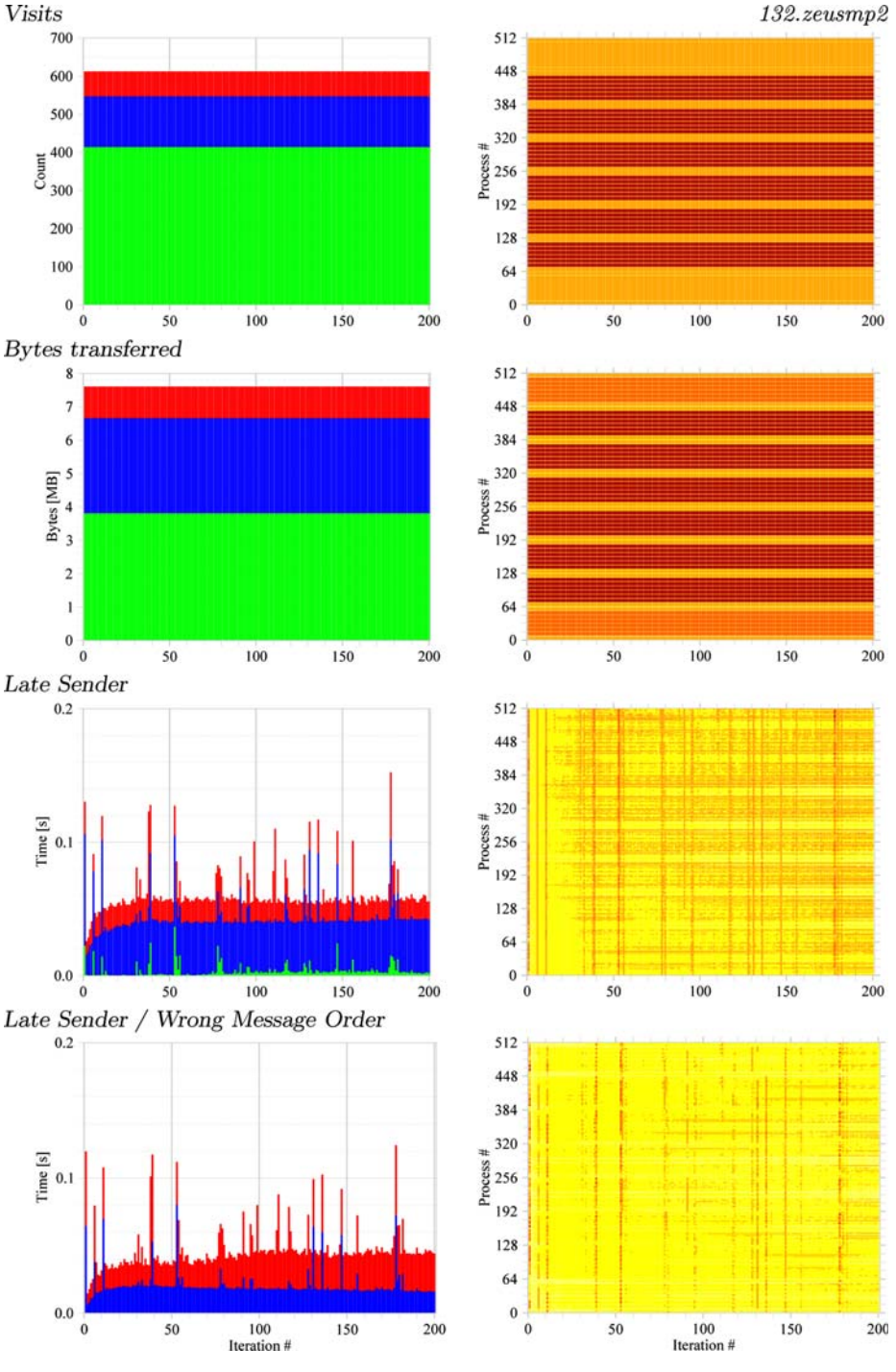

Fig. 9. Graphs and charts of SPEC MPIm2007 *132.zeusmp2* timestep metrics (cont.)

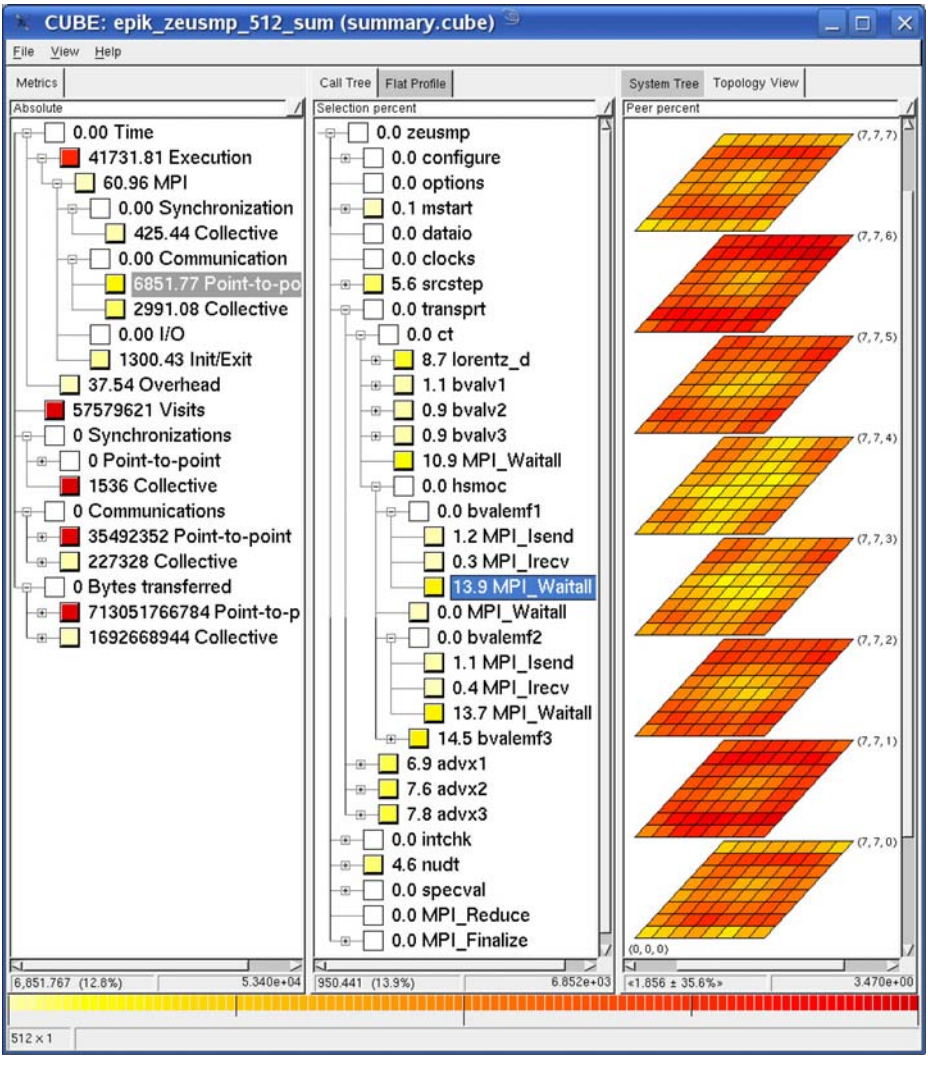

Fig. 10. SCALASCA analysis report of SPEC MPIm2007 *132.zeusmp2* 512-process execution runtime summarization experiment, showing unbalanced distribution of *Point-to-point Communication time* (left pane) on critical call-path to MPI_Waitall calls in function bvalemf1 (central pane). Closed tree nodes show inclusive metric values (including child node values), whereas open tree nodes show exclusive metric values (excluding child values). Numerical metric values are also colour-coded according to the scale at the bottom. Values in each pane are accumulated from those in panes to the right, and selecting a metric or call-path sets that node's metric value as the focus for panes to the right. 12.8% of total execution time is MPI *Point-to-point Communication time*, 13.9% of which is in the MPI_Waitall calls from bvalemf1, with a 35.6% standard variation across the 512 processes, and highest values predominantly for processes in the 2nd and 7th z-planes of the application's $8 \times 8 \times 8$ Cartesian grid (right pane).

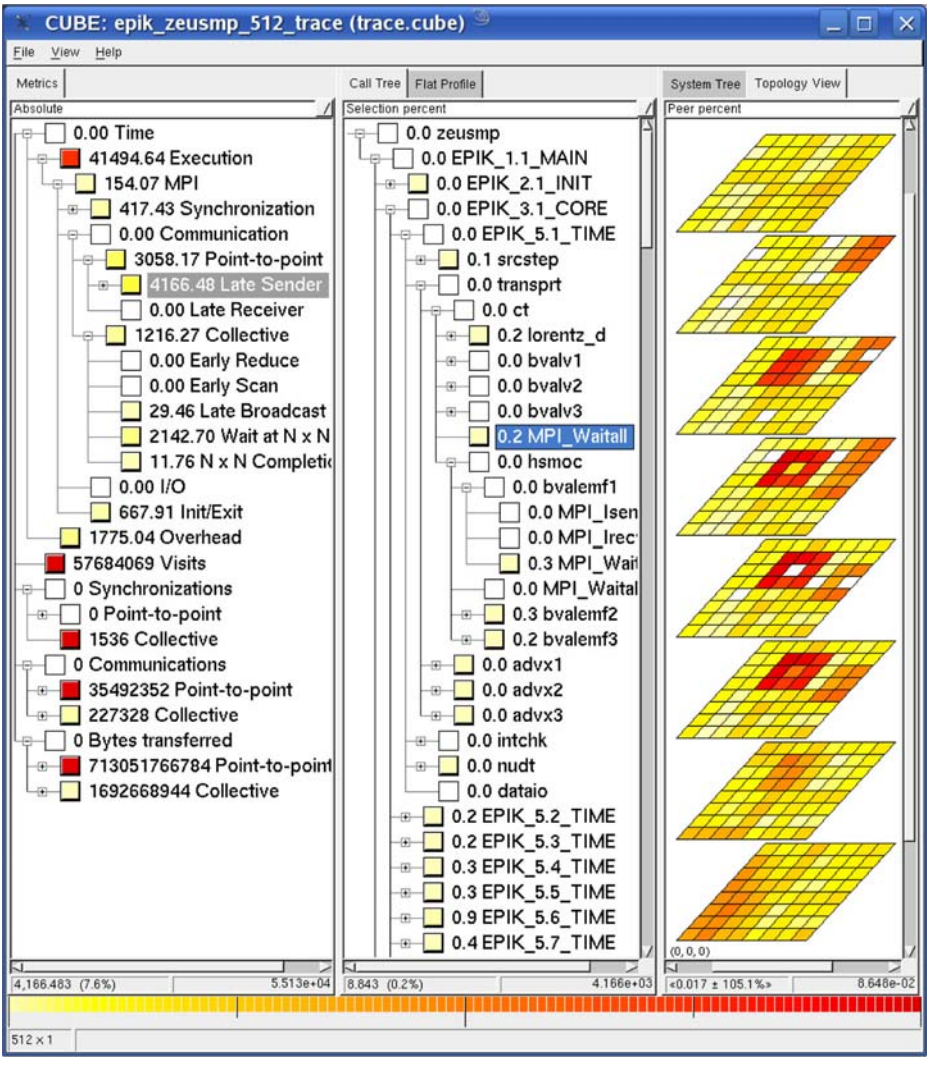

Fig. 11. SCALASCA analysis report of SPEC MPIm2007 *132.zeusmp2* 512-process execution tracing experiment, including manually inserted timestep annotations, showing unbalanced distribution of *Late Sender time* for the MPI_Waitall calls directly from function `ct` during the first timestep. 7.6% of total execution time is due to *Late Sender* situations, which is 57.7% of MPI *Point-to-point Communication time*. This varies considerably from timestep to timestep, and manifests as a 105.1% standard deviation in the MPI_Waitall calls in `ct` during the first timestep, localized to a small number of interior processes of the $8\times8\times8$ Cartesian grid.

highest values localized on certain processes, which can be determined from their locations within the $8\times8\times8$ Cartesian grid used by *132.zeusmp2*.

Additional insight into the origin of this imbalance in MPI *Point-to-point Communication* time can be derived from the automatic trace analysis report shown in Figure 11. Metrics which are only available from trace analysis show that inefficiencies are growing, e.g., *Late Sender* situations are now 57.7% of MPI *Point-to-point Communication time*. The additional timestep annotations distinguish the metric variation between timesteps, which is clearly considerable. Of more concern, however, is the huge variation between processes within each timestep, which is localized to relatively small numbers of interior processes.

3.4 Review of SCALASCA SPEC MPI2007 Benchmark Analyses

SPEC MPI2007 is a substantial suite of application kernels for testing the effectiveness of performance tools. By collecting and analyzing execution measurement experiments with 512 processes for each benchmark, the various Scalasca measurement and analysis techniques have demonstrated that they scale well, and provide insight into significant performance problems. Annotating repetitive execution phases [12,13] and associated timeline charts of those phases [14] support deeper and clearer understanding of those performance issues, to determine which execution intervals and processes are affected. Although the analyses presented here concentrated on MPI communication and synchronization, metrics acquired from processor and network hardware counters can readily be incorporated in measurement experiments for a holistic view of execution performance [15].

Certain dubious coding constructs used in the SPEC MPI2007 applications, however, resulted in analysis problems. For example, a non-void function without an explicit return statement was incorrectly instrumented by the IBM XL compiler, such that exits were not matched with corresponding entry instrumentation. In these rare cases, the offending source code was modified and then the compiler generated correct instrumentation.

The analyses also identified oddities in some of the SPEC MPIm2007 benchmarks, e.g., *115.fds4* makes numerous calls to MPI_Waitall always with an empty list of requests. Although this is a valid test of MPI performance, simple application optimization would skip the MPI_Waitall call in such cases.

122.tachyon and *129.tera_tf* appear to scale perfectly, however, other SPEC MPI2007 applications show performance tailing off with larger numbers of processes, and the Scalasca analyses at large scale provide crucial insight into the governing performance factors, as demonstrated with *132.zeusmp2*. For *137.lu*, *132.zeusmp2* and *126.lammps* the problem size is too small to scale to larger numbers of processes, or there are coded scalability limiters (enforced or implied). Clearly unacceptable scaling of *113.GemsFDTD* appears mainly to be due to its inefficient scheme for distributing data using broadcasts during initialization prior to the update loop.

4 Conclusions and Future Work

Applying established performance analysis techniques for phase annotation, event filtering, runtime summarization, event tracing and analysis presentation, via the Scalasca toolset, to the SPEC MPI2007 benchmark suite applications has revealed a variety of complex execution behaviour and potential opportunities for performance improvement. Although 512 processes is a relatively modest scale for the current generation of HPC applications, the ability to collect and analyze measurements effectively from long-running, real-world applications was demonstrated.

With their limited scalability and significant process memory requirements, the SPEC MPIm2007 benchmarks are clearly not suitable for the largest 'leadership' computer systems, such as IBM BlueGene, Cray XT and Sun Constellation. When a 'large-sized' benchmark configuration becomes available, it will be interesting to repeat the Scalasca analyses at the large-scale for which the toolset was designed and already validated with other HPC applications [16].

Automated classifications of equivalence groups of phases and processes with related behavioural characteristics are currently being investigated with the aim of making measurements and analyses more concise, and thereby more scalable. Future work will also examine how the presentation of such analyses can be scaled adequately for much larger numbers of processes (often in the tens of thousands) and integrated within the Scalasca interactive analysis report explorer GUI.

References

1. Standard Performance Evaluation Corporation, SPEC MPI2007 benchmark suite, `http://www.spec.org/mpi2007/`
2. Müller, M.S., van Waveren, M., Lieberman, R., Whitney, B., Saito, H., Kalyan, K., Baron, J., Brantley, B., Parrott, C., Elken, T., Feng, H., Ponder, C.: SPEC MPI 2007 — An application benchmark for clusters and HPC systems. In: Proceedings of ISC 2007, Dresden, Germany (June 2007) (Also available as internal report ZIH-IR-0708, Technische Universität Dresden, Germany)
3. Müller, M.S.: Applying performance tools to real world applications. In: Proceedings of Seminar 07341 on Code Instrumentation for Massively Parallel Performance Analysis, Dagstuhl, Germany (September 2007)
4. Müller, M.S., Knüpfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., Nagel, W.E.: Developing scalable applications with Vampir, VampirServer and VampirTrace. In: Parallel Computing: Architectures, Algorithms and Applications, Proc. 12th ParCo Conf., Jülich/Aachen, vol. 15, pp. 637–644. IOS Press, Amsterdam (2008)
5. Fürlinger, K., Gerndt, M., Dongarra, J.: Scalability analysis of the SPEC OpenMP benchmarks on large-scale shared-memory multiprocessors. In: Shi, Y., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2007. LNCS, vol. 4488, pp. 815–822. Springer, Heidelberg (2007)
6. Aslot, V., Eigenmann, R.: Performance characteristics of the SPEC OMP2001 benchmarks. In: Proc. 3rd European Workshop on OpenMP, EWOMP 2001, Barcelona, Spain (September 2001)

7. Saito, H., Gaertner, G., Jones, W., Eigenmann, R., Iwashita, H., Lieberman, R., van Waveren, M., Whitney, B.: Large system performance of SPEC OMP2001 benchmarks. In: Proc. Int'l Workshop on OpenMP Experiences and Implementations (WOMPEI 2002) (2002)
8. Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Frings, W., Fürlinger, K., Geimer, M., Hermanns, M.-A., Mohr, B., Moore, S., Pfeifer, M., Szebenyi, Z.: Usage of the Scalasca toolset for scalable performance analysis of large-scale parallel applications. In: Proc. 2nd Int'l Workshop on Tools for High Performance Computing, Stuttgart, Germany, Springer (July 2008) (to appear)
9. Wylie, B.J.N., Wolf, F., Mohr, B., Geimer, M.: Integrated runtime measurement summarization and selective event tracing for scalable parallel execution performance diagnosis. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 460–469. Springer, Heidelberg (2007)
10. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: Scalable parallel trace-based performance analysis. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 303–312. Springer, Heidelberg (2006)
11. John von Neumann Institute for Computing, Jülich Multiprocessor IBM p690+ cluster, `http://www.fz-juelich.de/jsc/jump`
12. Wylie, B.J.N., Gove, D.J.: OMP AMMP analysis with Sun ONE Studio 8. In: Proc. 5th European Workshop on OpenMP EWOMP 2003, Aachen, Germany, September 2003, pp. 175–184. RWTH Aachen University (2003)
13. Malony, A.D., Shende, S.S., Morris, A.: Phase-based parallel performance profiling. In: Parallel Computing: Architectures, Algorithms and Applications, Proc. 11th ParCo Conf., Málaga, Spain, September 2005. NIC Series, vol. 33, pp. 203–210. John von Neumann Institute for Computing, Jülich, Germany (2005)
14. Fürlinger, K., Gerndt, M., Dongarra, J.: On using incremental profiling for the performance analysis of shared-memory parallel applications. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 62–71. Springer, Heidelberg (2007)
15. Wylie, B.J.N., Mohr, B., Wold, F.: Holistic hardware counter performance analysis of parallel programs. In: Parallel Computing: Architectures, Algorithms and Applications, Proc. 11th ParCo Conf., Málaga, Spain, September 2005. NIC Series, vol. 33, pp. 187–194. John von Neumann Institute for Computing, Jülich, Germany (2006)
16. Wylie, B.J.N., Geimer, M., Wolf, F.: Performance measurement and analysis of large-scale parallel applications on leadership computing systems. In: Scientific Programming, special issue on Large-scale Programming Tools and Environments. IOS Press, Amsterdam (to appear, 2008)
17. Jülich Supercomputing Centre, SCALASCA toolset for scalable performance analysis of large-scale parallel applications, `http://www.scalasca.org/`