Deconstructing PARSEC Scalability

Gabriel Southern and Jose Renau
Dept. of Computer Engineering, University of California, Santa Cruz
{gsouther,renau}@soe.ucsc.edu

Abstract

PARSEC is a popular benchmark suite designed to facilitate the study of CMPs. It is composed of 13 parallel applications, each with an input set intended for native execution, as well as three reduced-size simulation input sets. Each benchmark also demarcates a Region of Interest (ROI) that indicates the parallel code in the application. The PARSEC developers state that users should model only the ROI when using simulation inputs; in other cases the native input set should be used to obtain results representative of full program execution.

We analyzed the runtime scalability of PARSEC using real multiprocessor systems and present our results in this paper. For each benchmark we analyzed the scalability of both the ROI and full execution for all the input sets. We found that for 7 of the benchmarks the ROI scalability matches that of the full program regardless of the input set used. For the remaining 6 benchmarks, for at least some of the input sets there is significant divergence between the scalability of the ROI and the full program. Three of these benchmarks have much lower scalability for the full program than the ROI, even when run with the native input set. Finally, we found that for most of the benchmarks the scalability of the simulation inputs differs from that of the native input set, both for the ROI and the full program.

1. Introduction

The PARSEC benchmark suite is "designed to provide parallel programs for the study [of] CMPs" [6]. It was introduced in 2008 and has been widely used for computer architecture research since then. Developed with the needs of researchers in mind, it has features that make it easier to use with architectural simulators. Each benchmark has multiple input sets, including three that are intended to run with simulators (*simsmall, simmedium, simlarge*), and one that is intended to be representative of a real application (*native*).[1] This allows users to simulate a smaller workload but obtain results representative of a real workload. Each benchmark also defines a *Region of Interest* (ROI) indicating which part of the benchmark executes in parallel. By simulating only the ROI, PARSEC users can reduce simulation time. The ROI is also important for ensuring that results obtained using simulation inputs are representative of real program behavior [5].

Choices in input set size and whether to model the whole program, or only the ROI, can lead to different interpretations when analyzing benchmark results. For instance, Figure 1 shows the reported scalability of blackscholes (one of the PARSEC benchmarks) from Bienia [5] and from Pusukuri et al. [22]. Bienia used the simlarge input set and only measured the scalability of the ROI, concluding that blackscholes has the potential for nearly linear scaling. But using the native input set and measuring the runtime of the entire application, Pusukuri et al. concluded that the scalability of blackscholes is limited by the serial portion of the application.

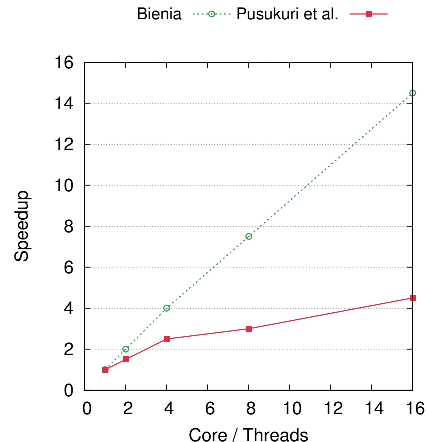

Figure 1: Scalability of blackscholes as reported by Bienia [5] and by Pusukuri et al. [22]. Bienia reports nearly linear scalability with increasing number of cores, while Pusukuri et al. reports a maximum speed-up of less than 5 times.

Several papers characterize the behavior of the PARSEC benchmark suite [1, 2, 4, 5, 6, 7, 8, 10, 22] but none of them compare the runtime scalability of the native input sets with that of the simulation input sets, nor do they compare the runtime scalability of the ROI to that of the full program. This paper provides this missing characterization, but a greater concern is that users of PARSEC rarely state which portion of the application is modeled or what input set is used. Table 1 lists how many papers from ISCA 2010 – 2014 used PARSEC in their evaluation, which input set was used and whether the paper analyzed the ROI only or the full program execution. In most cases this information is not provided in the paper, but as the differing results from Bienia and Pusukuri et al. illustrate, these parameters are important for interpreting benchmark data.

In this paper we measure the runtime scalability of the four main PARSEC input sets, and we compare the scalability of

[1] Two additional input sets (*test* and *simdev*) are included for simulator testing and are not appropriate for scientific studies.

Input Set	Full	ROI	Not Stated
native	1	1	4
simlarge	0	5	9
simmedium	0	2	7
simsmall	0	1	10
not stated	1	1	20

Table 1: Input sets and program region modeled for 55 papers[2] published in ISCA from 2010 – 2014 that use PARSEC. Most papers do not state what region is modeled.

the ROI with that of the whole program. We do so by running the benchmarks on real multicore systems, varying the number of threads, and measuring the *runtime* of the ROI and the full program for the four different input sets. Our contributions are as follows:

- First systematic analysis of the runtime scalability of all PARSEC input sets.
- First systematic analysis of the runtime scalability of the ROI compared with full-program execution for all input sets.
- We identify 7 benchmarks where the ROI and full program execution have similar scalability for all input sets, and 6 where the scalability differs for at least some of the input sets.
- We show that for most of the benchmarks the scalability of the simulation input sets differs significantly from the scalability of the native input set for both the ROI and full program execution.

The rest of this paper is organized as follows: Section 2 provides background about PARSEC; Section 3 describes our experiment setup; Section 4 provides detailed results; Section 5 surveys related work; and Section 6 concludes.

2. Background

PARSEC was developed between 2005 and 2009 as part of a collaboration between Princeton and Intel [5]. The developers' goal was to create a benchmark suite of emerging parallel workloads that would help architects and researchers design emerging multicore and multiprocessor systems. After its initial release in 2008, PARSEC quickly became popular among computer architecture researchers and has since been widely used in published research.[3] Several versions of PARSEC have been released, but the 13 core benchmarks listed in Table 2 were finalized in PARSEC 2.0, which was released in February 2009.

Six different input sets are defined for each benchmark: *test, simdev, simsmall, simmedium, simlarge, native*. Test and simdev should only be used to test that the benchmark can run. Native is intended to approximate realistic input indicative of how the benchmark application would be used in practice. The remaining three simulation inputs were created by scaling down the native input sets in a way that maintained a representative mix of instructions. The inputs were selected so that serial execution of the native input sets on a real machine should complete in 15 minutes or less, while the simlarge, simmedium, and simsmall inputs should complete execution within 15 seconds, 4 seconds, and 1 second respectively.

The inputs set scaling process skews the amount of time spent in serial phases compared to parallel phases. As a result PARSEC defines an ROI for each benchmark that marks the parallel phase of the benchmark. Bienia [5] states:

> The skew should be compensated for by either excluding the serial initialization and shutdown phases and limiting all measurements to the Region-of-Interest (ROI) of the program, which was defined to include only the representative parallel phase, or by measuring the phases of the program separately and manually weighing them correctly. It is safe to assume that the serial initialization and shutdown phases are negligible in the real inputs, which allows one to completely ignore them for experiments. Benchmark users who do not wish to correct measurements in such a way should limit themselves to the `native` input set, which is a much more realistic description of real program behavior that exhibits these scaling artifacts to a much lesser extent.

PARSEC supports three different threading models: pthreads, OpenMP, and Intel Thread Building Blocks (TBB). PARSEC also allows users to specify the *minimum* number of threads run with the benchmark by setting a parameter (n) when the benchmark is started. Table 2 shows the correspondence between the user specified number of threads and how many threads the benchmark actually spawns.

Benchmark	Threads
blackscholes	$1+n$
bodytrack	$2+n$
canneal	$1+n$
dedup	$3+3n$
facesim	$1+n$
ferret	$3+4n$
fluidanimate	$1+n$
freqmine	n
raytrace	$1+n$
streamcluster	$1+2n$
swaptions	$1+n$
vips	$3+n$
x264	$1+2\times frames$

Table 2: Benchmarks in PARSEC along with the number of threads spawned by each benchmark, where n is the PARSEC *minimum threads* parameter.

[2]Total is more than 55 because some papers used multiple input sets.

[3]The main PARSEC paper [6] has over 1,500 citations according to Google Scholar.

In most cases there is a single main thread which spawns n worker threads, but some of the benchmarks use a pipelined parallelization model and spawn multiple threads for each one the users specifies. In addition x264 spawns twice as many threads as there are frames in its input (native has 512 frames), but it uses the parameter n to limit how many threads run in parallel. Most of the benchmarks allow n to range from 1 up to at least 128; however, there are a few restrictions:

- Facesim is limited to the values 1, 2, 3, 4, 6, 8, 16, 32, 64, 128.
- Swaptions is limited to the number of entries in its input set (16 for simsmall, 32 for simmedium, 64 for simlarge, and 128 for native).
- Fluidanimate requires the number of threads to be a power of 2.
- x264 is limited by the number of frames in its input set.[4] We restricted n to 1–8 for simsmall and 1–32 for simmedium in our experiments.

In this paper we analyze the 13 benchmarks and input sets first released with PARSEC 2.0.[5] We use the native, simlarge, simmedium, and simsmall input sets, and we use pthreads for all benchmarks except freqmine (which requires OpenMP). We vary the number of threads using the minimum threads parameter n and we report n as the parameter of interest in our results instead of reporting how many threads were actually spawned.

3. Experiment Setup

We analyzed the scalability of PARSEC input sets and ROI by running the benchmarks on three different real multi-core/multiprocessor systems and measuring the runtime. We focused our evaluation on wall-clock runtime because this is the most important metric for determining the scalability of parallel applications.

The scalability results are specific to the systems that we used for our evaluation and, in theory, alternative system configurations could produce different results. However, we expect that the relative scalability trends for inputs sets and ROI compared to full are applicable to many systems that use PARSEC. The three systems that we used for our evaluation are:

- A single CPU system with 4 cores, 2 threads per core, for a total of 8 logical processors, along with 16 GB of RAM.
- A dual socket system with 8 cores per socket, 2 threads per core, for a total of 32 logical processors, along with 64 GB of RAM.
- A quad socket system with 12 cores per socket, 1 thread per core, for a total of 48 logical processors, along with 64 GB of RAM.

The detailed system specifications are shown in Table 3.

System	Configuration
M8	1 x Intel Xeon E3-1275 v3 (4 core, 2-way SMT) 32 KB L1, 256 KB L2, 8 MB L3 cache 16 GB DRAM
M32	2 x Intel Xeon E5-2689 (8 core, 2-way SMT) 32 KB L1, 256 KB L2, 20 MB L3 cache 64 GB DRAM
M48	4 x AMD Opteron 6172 (12 core) 64 KB L1, 512 KB L2, 5 MB L3 cache 64 GB DRAM

Table 3: Specifications of systems used for experiments. M8, M32 and M48 respectively have 8, 32, and 48 logical processors.

In the rest of this paper we refer to the 8-logical processor system as *M8*, the 32-logical processor system as *M32*, and the 48-logical processor system as *M48*.

All of the systems used the x86_64 version of Arch Linux with version 3.18.6-1 of the Linux kernel. All benchmarks were compiled with version 4.9.2 of gcc/g++. We disabled ASLR but did not do any other special tuning. The OS and hardware were allowed to schedule threads and control CPU frequency using default scheduling algorithms. We used PARSEC hooks to identify the ROI, and for each configuration we recorded the runtime of the ROI and full benchmark execution.

We repeated each experiment at least 10 times and calculated the mean and the confidence interval at a 95% confidence level. For configurations where the initial confidence interval after 10 runs was not within 5% of the mean we repeated the experiment until the confidence interval was within 5% of the mean. The speedup results we present are computed by dividing the mean execution time of a system, input set, and ROI or full configuration with a single thread by the mean execution time of the same configuration with multiple threads. In total we performed over 300,000 experiments across the 3 different machines while sweeping through a range of parameters and gathering statistically stable results.

4. Results

This section presents the results of our characterization of PARSEC's scalability. In Section 4.1 we measure the fraction of time each benchmark spends executing the ROI. Section 4.2 describes how we measured scalability and we present our results in a set of figures that spans three pages. Section 4.3 summarizes the scalability by comparing maximum speedup, and Section 4.4 quantifies the similarity of different configurations. Finally Section 4.5 provides insight about benchmark behavior.

4.1. ROI Percentage

The ROI is the only part of the PARSEC benchmarks that can be executed in parallel. Figure 2 shows the percentage of time each benchmark spent in the ROI when running on M8 with the number of threads $n = 1$. Seven of the benchmarks (dedup, ferret, freqmine, streamcluster, swaptions, vips, x264) have

[4] This limitation is not reported when the benchmark is launched, but in our experiments we observed the output was not correct for n greater than 9 for simsmall and n greater than 33 for simmedium.

[5] We used PARSEC 3.0 downloaded from the PARSEC website, but there are minimal changes between 2.0, 2.1 and 3.0 for the benchmarks we analyzed.

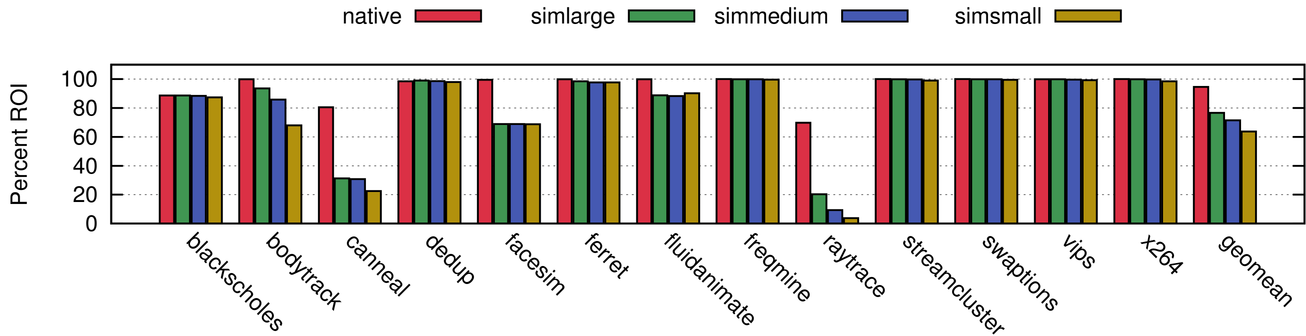

Figure 2: Percentage of full execution time that is in the ROI measured by running benchmarks on M8 with a single thread. Seven of the benchmarks spend nearly 100% of their execution time in the ROI, while the other six spend significantly less for at least some input sets.

almost all of the execution time included in the ROI for both the full and simulation input sets. The other six benchmarks spend less than 90% of their execution time in the ROI and consequently have a maximum theoretical speedup of less than 10 times for at least some of their input sets. We provide more detail in the following list and state the maximum theoretical speedup (calculated using Amdhal's law) in parentheses when listing ROI percentages.

- **Blackscholes** spends approximately 89% (9X) of its time executing the ROI for all four input sets. The time outside of the ROI is spent initializing the input array and writing out the results. This amount of work scales linearly with the input set size; consequently a larger input sets does not improve the scalability. We confirmed this experimentally by creating an input set 10 times larger than the native input set included with PARSEC and measured the same ROI percentage for this larger input set.
- **Bodytrack** has over 99% ROI for the native input set, but the ROI drops to 94% (17X) for simlarge, 86% (7X) for simmedium, and 68% (3X) for simsmall. Bodytrack uses helper threads to load image data for the next frame concurrently with the threads processing the current frame. However, the thread pool must be created and the first image loaded before any parallel computation can occur. This initialization is done before the ROI starts. A negligible percentage of total execution time is consumed for the native input set, but a significant fraction of time is consumed for the two smallest input sets.
- **Canneal**'s ROI percentage is 80% (5X) for native, approximately 30% (1.5X) for simlarge and simmedium, and 23% (1.3X) for simsmall. The majority of the time outside of the ROI is spent initializing the netlist. This initialization time is proportional to the size of the netlist. However, there are two ways to increase the work done by the benchmark. Either the netlist size can be increased, or the number of temperature steps can be increased. We tested using 10 times as many temperature steps for the native input set with the same netlist, which increased the ROI percentage to 98%.
- **Facesim** has over 99% ROI for native, but approximately 69% (3.2X) for the simulation input set.[6] There is a fixed amount of work done outside the ROI based on the facial features that will be animated. The work done to animate each frame is in the ROI, and this work scales with the number of frames. The native input set processes 100 frames, while the simulation inputs process only a single frame.
- **Fluidanimate** also have over 99% ROI for native, but approximately 89% (9X) for the simulation inputs. The benchmark simulates fluid dynamics for use in animation sequences. The work outside of the ROI is mostly involved with partitioning how a single animation frame is processed. The work in the ROI scales with the number of frames in the workload, and adding frames adds more work in the ROI. The native input set has 500 frames, while the simulation inputs only have 5 frames.
- **Raytrace** has the worst speedup potential with only 70% (3.3X) ROI for native inputs, and 20% (1.25X), 9% (1.1X), and 3% (1.03X) for simlarge, simmedium, and simsmall respectively. However, it is possible to increase the amount of work in the ROI by rendering more frames. The native input set renders 200 frames; when we increased this to 2,000 frames the ROI increased to 95%.

4.2. Measured Scalability

The ROI percentages detailed in the previous section show how the parallel and serial sections of the workload can impact the maximum theoretical speedup. However, there are typically other bottlenecks that limit an application's maximum speedup. We measured the actual speedup for all of the benchmarks on our three systems: M8, M32, M48. For each system we varied the number of threads from 1 to the number of logical processors in the system and measured the runtime of full execution and ROI for all benchmarks using all input

[6] Facesim includes files for three simulation input sets, but they are all identical.

sets. We measured the runtime for over 300,000 benchmark executions and after averaging and summarizing the data we had 7,736 data points characterizing the measured scalability of the PARSEC benchmarks. We organized this data by benchmark and system and present it in a series of 39 graphs. The graphs are grouped into columns by system and into rows by benchmark. In order to make the data legible it is split into Figures 4, 5, and 6.

4.3. Maximum Speedup

Figure 3 shows the maximum speedup obtained for each of the benchmarks for both the full execution and the ROI. As expected the six benchmarks we identified with low ROI percentage show a big difference between the speedup of full and the ROI. PARSEC was designed so that researchers using the simulation inputs should only collect data from the ROI, and our results show the relevance this recommendation.

The divergence between the speedup of ROI for each of the native inputs and the speedup of ROI of each of the simulation inputs is a problem because results obtained using simulation inputs may not be representative of actual application behavior. For many benchmarks the maximum speedup of the ROI obtained using simulation inputs is different from the maximum speedup obtained with the native input set. The geometric mean of these differences is 62%, 47%, and 31% for simlarge, simmedium, and simsmall respectively.

Benchmark	Native	Simlarge	Simmedium	Simsmall
blackscholes	25.6	17.0	12.8	9.5
bodytrack	0.0	1.0	1.5	1.3
canneal	12.4	19.4	16.7	11.6
dedup	0.2	0.4	0.3	0.5
facesim	0.1	3.0	2.9	3.0
ferret	0.2	0.9	1.2	0.4
fluidanimate	0.2	4.7	4.7	4.5
freqmine	0.0	0.0	0.1	0.1
raytrace	27.2	23.8	18.0	10.3
streamcluster	0.0	0.0	0.0	0.0
swaptions	0.0	0.3	0.4	0.4
vips	0.1	0.5	0.8	0.8
x264	0.1	0.2	0.3	0.1
geomean	0.2	0.9	1.1	0.9

Table 4: Average Euclidean distance between speedup of ROI and full for each input set.

4.4. Quantifying Similarity

Visual inspection of the scalability results presented in Figures 4, 5, and 6 provides an intuitive sense of how the scalability compares between ROI and full for all of the input sets. We also attempted to quantify the similarity by calculating the average Euclidean distance between the measured speedup at each point. Table 4 shows the results for comparing the ROI and full execution for each of the input sets. The closer the values are to 0, the more similar the speedup results. The 6 benchmarks where we measured less than 90% ROI all have an average Euclidean distance of 1.0 or more; for the remaining 7 benchmarks the distance is less than 1.0.

We also calculated the average Euclidean distance between each of the simulation inputs and the native input for both ROI and full. In all cases the average difference was greater than 1.0 and the geometric mean for each category was greater than 5.0. This did not reveal any other clear trends so the data is not included. Instead we refer readers to the detailed graphs in Figures 4, 5, and 6 as well as the benchmark insights in the following section to interpret benchmark behavior.

4.5. Benchmark Insights

Blackscholes has the best scalability of any benchmark for the ROI with the native input set. The maximum speedup of the full execution is limited to less than 9 times because of the serial portion of the benchmark. It is also noteworthy that even for the ROI, the simulation inputs do not scale as well as the native input set for large numbers of threads. On M48 the execution time of the ROI for the simsmall input when running with 48 threads is 14.4 ms. We tested a modified version where the worker threads spawn and return immediately without doing any work and the ROI time dropped to 4 ms. Thus it does not appear that the thread creation overhead prevents further scaling of the benchmark. Determining the precise cause of the scalability limits for the simulation inputs in the ROI is a topic of future work for us.

Bodytrack has relatively good similarity between ROI and full for native and simlarge. For simmedium and simsmall speedup of full and ROI drops noticeably, particularly on M48 with many threads.

Canneal has limited scalability for full for all of the input sets with a maximum speedup of less than 5 times for native input. Even for ROI only, the total scalability is limited and on M48 the speedup of ROI drops after roughly 30 threads. Canneal uses atomic operations to synchronize data between threads; as a result adding more threads increases the chance of conflicts between threads. There is a tradeoff between the size of the netlist and the number of temperature steps. We think this is why the simlarge input has higher scalability than native. When we tested with an input with more time steps than native we found that the scalability of full improved, but the scalability of the ROI dropped.

Dedup has an erratic speedup pattern due to a work distribution imbalance between threads. Dedup creates queues for partitioning work between threads, and the number of queues created is $n\ threads/4 + n\ threads \bmod 4$. Afterwards each queue is assigned 4 worker threads, except the last queue, which has *n threads* mod 4 threads. This workload imbalance causes the scalability to be best when the number of threads is a multiple of 4. It is also noteworthy that the simlarge input set has a maximum speedup with 12 user threads while the other inputs sets have a maximum speedup with 8 user threads, and the maximum speedup of simlarge input is much higher than that of native for both ROI and full. We suspect this is

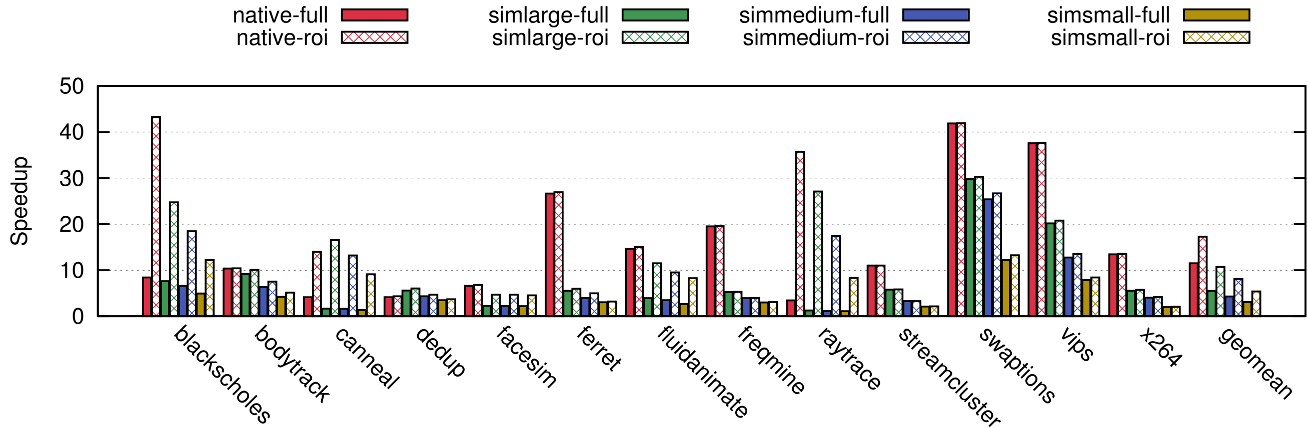

Figure 3: Maximum speedup measured on M8, M32, and M48, for each benchmark, region, and input set combination. The best is Blackscholes ROI with native input that has a 43 times speedup. The worst is Raytrace full with simsmall that has a 1.1 times speedup.

caused by differences in the input set data. Dedup performs deduplication, and simlarge achieves 2.38X compression factor, while native, simmedium, and simsmall achieve 1.05X, 1.06X, and 1.09X compression respectively.

Facesim has much lower scalability for the simulation input than for the native input set for both full and ROI. The lower scalability of full is explained by the fraction of the benchmark that is in the ROI. Since simulation and native inputs both process the same frame we expect the differences between the scalability of the two inputs are related to additional initialization overhead that is included in the ROI time but not amortized over multiple frames.

Ferret also has much lower scalability for the simulation inputs than for native. It uses pipelined parallelization, but the first and last stages in the pipeline only spawn a single thread. We experimented with removing these stages from the ROI, but that did not improve the ROI scalability for the simulation inputs. Bienia [5] notes that the simulation inputs for Ferret have less opportunity for parallel execution, while Pusukuri et al. [22] found that the speedup for the native input set was limited by lock contention. Although we do not have a definitive explanation for the scalability difference, we do note that the simulation inputs reach their maximum speedup much earlier than when using the native input, and this maximum speedup is much lower than when using native inputs.

Fluidanimate's low scalability for full execution of the simulation inputs is explained by the lower fraction of the benchmark in the ROI. However, even for the ROI the scalability of the simulation inputs is lower than that of the native input set. We suspect this is due to the thread communication overhead that is proportional to the number of particles in the input set, and so the overhead is more for the smaller input sets. It is also noteworthy that the speedup of simsmall on M48 *drops* when increasing from 16 to 32 threads.

Freqmine is another benchmark with much lower scalability for the simulation inputs than the native input set. Both the native and simulations inputs have a high percentage of their work included in the ROI. However, freqmine uses OpenMP and we suspect that the smaller simulation inputs have their scalability constrained by the serial portions of the workload and that the parallel loops are too small to provide much speedup.

Raytrace has extremely limited scalability for the full execution because of the low fraction of the benchmark in the ROI. Increasing the number of frames from the 200 used for the native input set to 2,000 increases the maximum speedup from 3.5X to 16X. Consequently we recommend that any studies using full execution should use a larger input set than native. It is also noteworthy that the ROI for the simulation inputs does not scale nearly as well as the native input set, particularly for simmedium and simsmall.

Streamcluster uses barrier based synchronization, and the scalability for simmedium and simsmall is much worse than for native. On M48, running simsmall with 48 threads is 68 times *slower* than running with 1 thread. Of even greater concern, the maximum speedup for simsmall on M48 occurs when the PARSEC minimum thread parameter is 2, and afterwards the performance worsens as more threads are added. Roth et al. [24] observed similar behavior and attributed it to inefficiency in the barrier synchronization.

Swaptions has very similar scalability for the ROI and the full program. The simulation inputs also match native scalability at some points but diverge at others. The application itself has a stairstep type of scalability caused by an imbalance in workload distribution between the threads. Earlier papers [22, 24] also identified this problem and our results add further confirmation to their observations.

Vips also uses a pipelined parallel programming model with two threads for performing I/O and then *n* threads for processing the data, and we suspect this is what causes worse

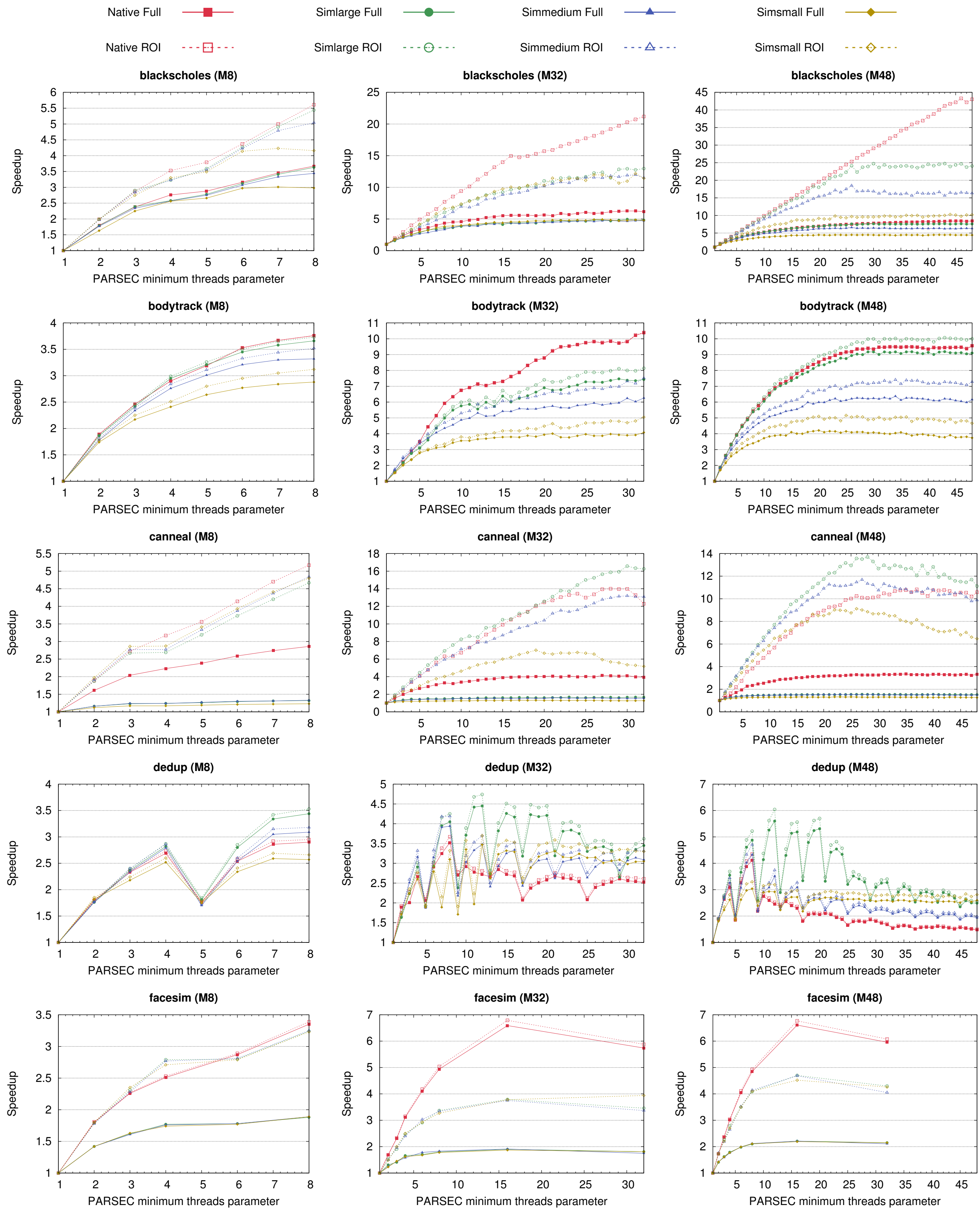

Figure 4: Speedup of blackscholes, bodytrack, canneal, dedup, and facesim for all input sets for both ROI and full on M8, M32, and M48. Benchmarks are grouped in rows and systems are grouped in columns. Native, simlarge, simmedium, and simsmall inputs are marked with a square, circle, triangle, and diamond mark respectively, and full are solid while ROI are hollow.

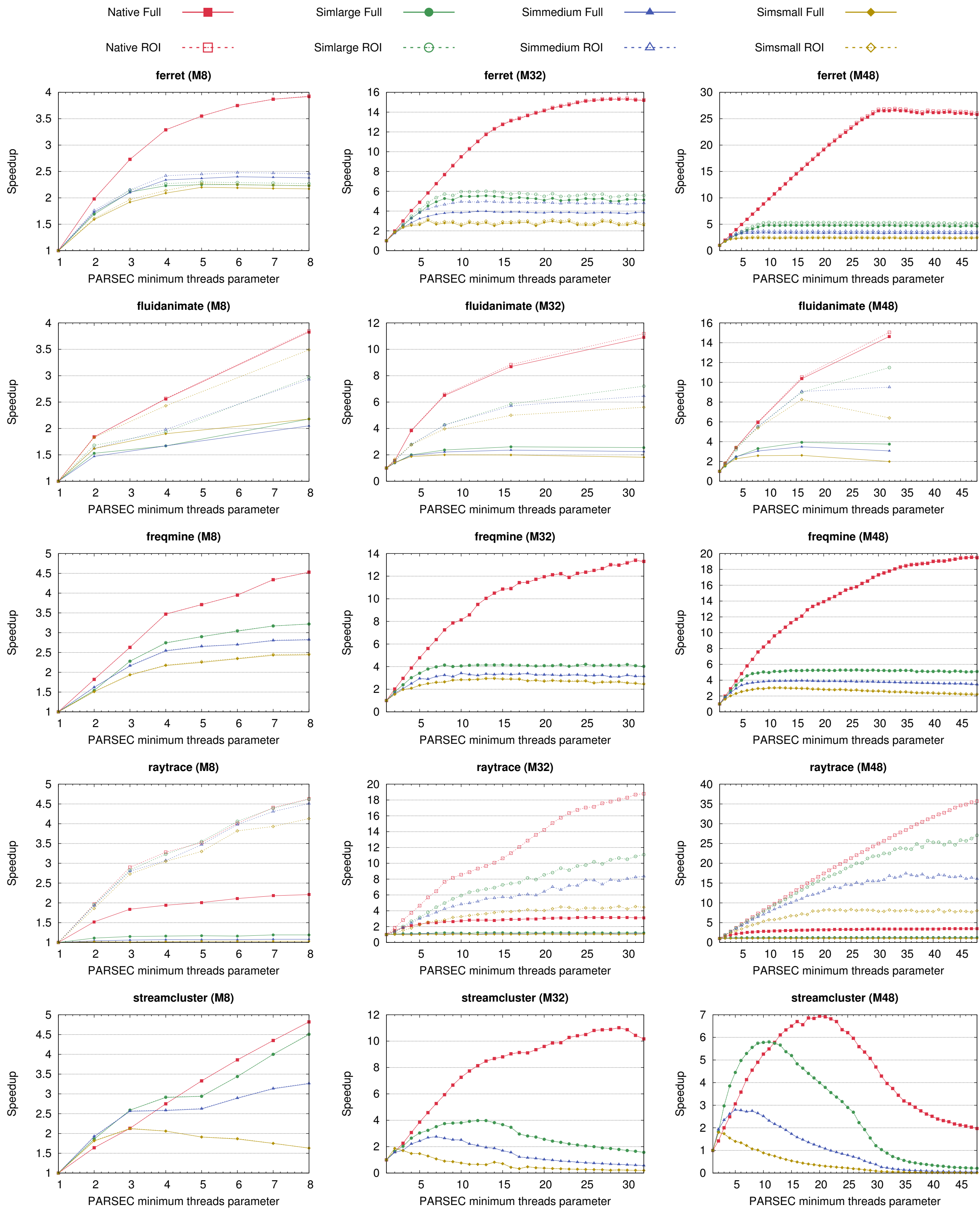

Figure 5: Speedup of ferret, fluidanimate, freqmine, raytrace, and streamcluster. The data is presented using the same formatting as explained for Figure 4. In some cases the full and ROI speedup for an input set is the same, so only one set of markers is visible because they completely overlap.

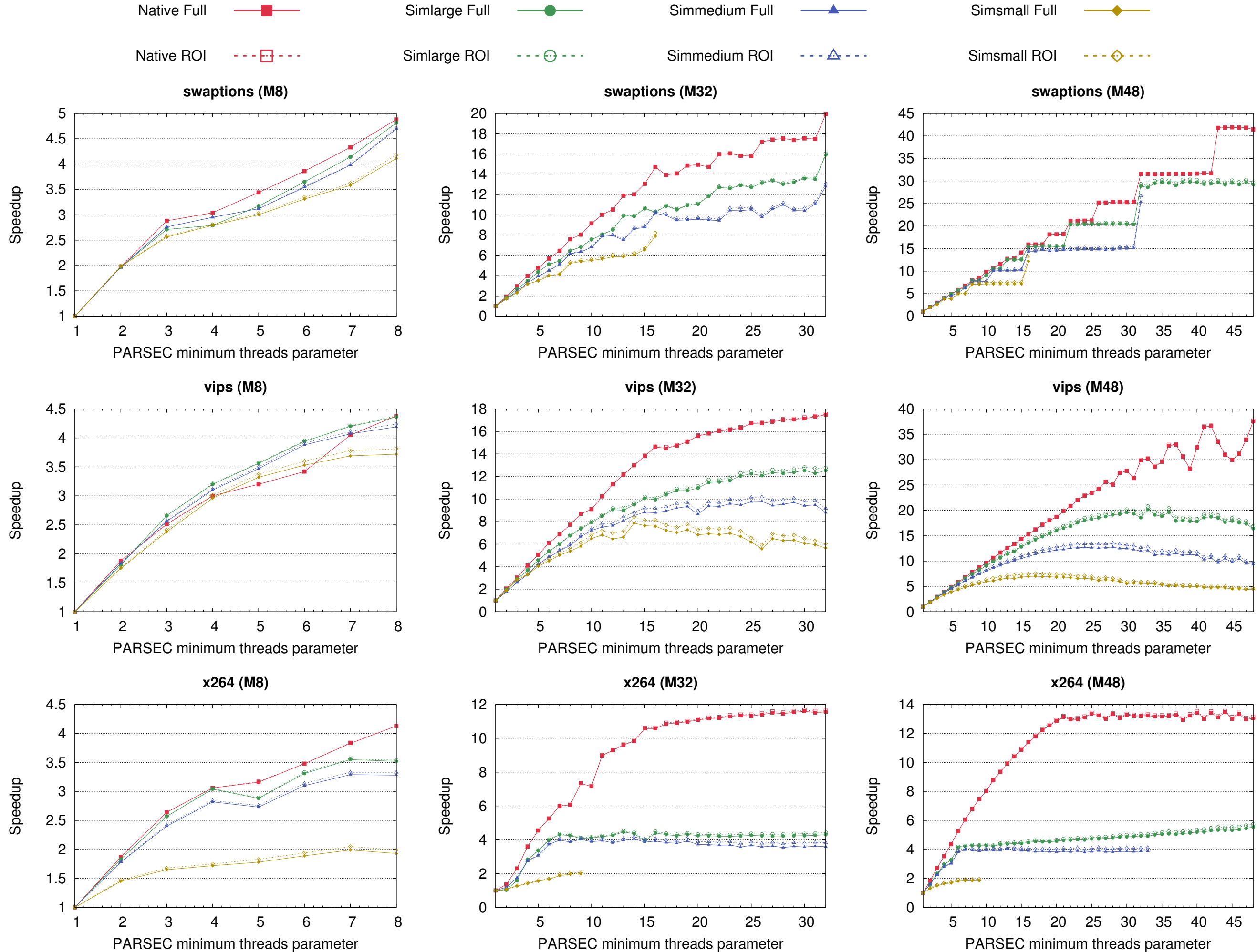

Figure 6: Speedup for swaptions, vips, x264, presented using same formatting explained in Figure 4.

scalability for smaller input sets. Bienia's dissertation [5] notes that the size of the output buffers can limit parallelism, and that this problem may be corrected in future versions of PARSEC. Although most of our tests used the benchmark code from PARSEC 3.0, for vips we reverted to PARSEC 2.1 because the source code for PARSEC 3.0 was missing ROI annotations. After noting Bienia's comment we also experimented with the vips source code in PARSEC 3.0, but our preliminary results show the same behavior as PARSEC 2.1.

x264 also has much lower scalability for the simulation inputs than for the native input set. The x264 application compresses an input video stream, and the simulation inputs have fewer and smaller frames than the native input. The smaller inputs have more dependencies between frames, and this limits the overall potential for achieving parallel speedup.

5. Related Work

Christian Bienia's 2011 dissertation [5] is the most comprehensive study of the PARSEC benchmarks and extends material published earlier [4, 6, 7, 8]. The characterization of PARSEC in Bienia's work relies on simulation and is intended to be machine independent. In contrast, our characterization is done using real machines and we focus on runtime as performance metric of interest.

Pusukuri et al. [22] developed *Thread Reinforcer* to pick an optimal number of threads for a parallel application. They evaluated their proposal using 8 of the 13 PARSEC benchmarks running on a 24-core system. They used the native input sets for their evaluation, and like us, found that the maximum speedup of the full execution of blackscholes and canneal was limited due to the fraction of serial code. Part of the motivation for our study was noting differences in Bienia and Pusukuri et al.'s characterization of PARSEC's scalability.

Several other papers have also studied the problem of thread scheduling and included characterization of some of the PARSEC benchmarks as part of their evaluation [17, 18, 19, 20, 21, 23, 25].

There are also several papers that characterized the performance of PARSEC. Like us, Bhadauria et al. [2] studied the scalability of PARSEC workloads using real machines, but they did not compare different input sets, or ROI and full program execution. Barrow-Williams et al. [1] analyzed commu-

nication patterns between threads in PARSEC and SPLASH using Simics. Bhattacharjee and Martonosi [3] analyzed TLB behavior of PARSEC benchmarks using a combination of native execution and simulation. Ferdman et al. [15] analyzed the single threaded performance of a variety of benchmark suites including PARSEC. Cebrián et al. [10, 11] proposed extending PARSEC with better support for SIMD hardware. Bryan et al. [9] examined how synchronization overhead and other system-level effects limited the potential scalability of PARSEC 1.0 benchmarks.

Several papers have also analyzed the scalability of some of the PARSEC benchmarks while developing techniques to find performance bottlenecks in parallel applications [12, 13, 14, 16, 24].

6. Conclusion

Benchmarks are a critical part of the quantitative approach to computer architecture research. But the complex interaction of the many layers of the computing stack, coupled with the slow speed of architectural simulators, forces architects to make approximations when simulating benchmark execution. PARSEC provides reduced size input sets and demarcates a ROI as ways to reduce simulation time while still approximating the behavior of the actual workload. In this paper, we presented the first systematic analysis of the runtime scalability of all of the input sets for all of the PARSEC benchmarks for both full program execution and ROI only.

Our results show significant variation in benchmark scalability depending on parameter selection. When calculating the maximum benchmark speedup on a 48-core system the average of all 13 benchmarks (calculated using geometric mean) ranges from 17.3X when measuring the ROI of the native input down to 3.1X for the full execution of the simsmall input set. Even when using the ROI only, as suggested by the PARSEC developers, the average scalability varies (17.3X for native, 10.7X for simlarge, 8.1X simmedium, and 5.4X simsmall). We also found that the scalability of full and ROI only varies for 6 benchmarks and is nearly the same for the other 7.

The PARSEC developers state that the native input sets can be used in cases where users want to measure the entire benchmark execution rather than ROI only. This is true for most of the benchmarks, but not for blackscholes, canneal, and raytrace, which have significant variation in speedup between ROI and full when using the native input sets. The PARSEC developers also suggest using as large an input set as possible, and relying on sampling to accelerate simulation. We agree that the larger input sets tend to offer better scalability. However, for several benchmarks the scalability of all the simulation inputs, including simlarge, is much less than that of the native input set, even when measuring ROI only.

Users of PARSEC should be aware of the differing behavior of PARSEC depending on input and region selection and should provide information about what parameters were selected when writing research papers, for the benefit of reviewers and other readers. In addition several of the scalability anomalies that we observed have been identified in the past and possible solutions have been proposed [24, 22]. However, the proposed fixes have not yet been incorporated into PARSEC. We think the research community as a whole would benefit from having a way to more quickly update PARSEC with bug fixes and performance improvements.

References

[1] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterization of SPLASH-2 and PARSEC," in *IISWC 2009*, October 2009.

[2] M. Bhadauria, V. M. Weaver, and S. A. McKee, "Understanding PARSEC performance on contemporary CMPs," in *IISWC 2009*, October 2009.

[3] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors," in *PACT 2009*, October 2009.

[4] C. Bienia, S. Kumar, and K. Li, "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *IISWC 2008*, September 2008.

[5] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *PACT 2008*, October 2008.

[7] C. Bienia and K. Li, "PARSEC 2.0: A new benchmark suite for chip-multiprocessors," in *MoBS 2009*, June 2009.

[8] ——, "Fidelity and scaling of the PARSEC benchmark inputs," in *IISWC 2010*, December 2010.

[9] P. Bryan, J. Beu, T. Conte, P. Faraboschi, and D. Ortega, "Our many-core benchmarks do not use that many cores," in *WDDD 2009*, June 2009.

[10] J. Cebrian, M. Jahre, and L. Natvig, "Optimized hardware for suboptimal software: The case for simd-aware benchmarks," in *ISPASS 2014*, March 2014.

[11] ——, "Parvec: vectorizing the parsec benchmark suite," *Computing*, pp. 1–24, 2015.

[12] S. Dutta, S. Manakkadu, and D. Kagaris, "Classifying performance bottlenecks in multi-threaded applications," in *MCSoc 2014*, September 2014.

[13] D. Eklov, N. Nikoleris, and E. Hagersten, "A software based profiling method for obtaining speedup stacks on commodity multi-cores," in *ISPASS 2014*, March 2014.

[14] S. Eyerman, K. Du Bois, and L. Eeckhout, "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," in *ISPASS 2012*, April 2012.

[15] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *ASPLOS 2012*, March 2012.

[16] W. Heirman, T. Carlson, S. Che, K. Skadron, and L. Eeckhout, "Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads," in *IISWC 2011*, November 2011.

[17] J. Lee, H. Wu, M. Ravichandran, and N. Clark, "Thread tailor: Dynamically weaving threads together for efficient, adaptive parallel applications," in *ISCA 2010*, June 2010.

[18] R. Moore and B. Childers, "Inflation and deflation of self-adaptive applications," in *SEAMS 2011*, May 2011.

[19] ——, "Using utility prediction models to dynamically choose program thread counts," in *ISPASS 2012*, April 2012.

[20] ——, "Program affinity performance models for performance and utilization," in *DATE 2014*, March 2014.

[21] A. Navarro, R. Asenjo, S. Tabik, and C. Caşcaval, "Load balancing using work-stealing for pipeline parallelism in emerging applications," in *ICS 2009*, June 2009.

[22] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread reinforcer: Dynamically determining number of threads via OS level monitoring," in *IISWC 2011*, November 2011.

[23] ——, "Thread tranquilizer: Dynamically reducing performance variation," *TACO*, January 2012.

[24] M. Roth, M. J. Best, C. Mustard, and A. Fedorova, "Deconstructing the overhead in parallel applications," in *IISWC 2012*, 2012.

[25] S. Sridharan, G. Gupta, and G. S. Sohi, "Adaptive, efficient, parallel execution of parallel programs," in *PLDI*, June 2014.