Performance scalability and dynamic behavior of Parsec benchmarks on many-core processors

Oved Itzhak
Technion
ovedi@tx.technion.ac.il

Idit Keidar
Technion
idish@ee.technion.ac.il

Avinoam Kolodny
Technion
kolodny@ee.technion.ac.il

Uri C. Weiser
Technion
uri.weiser@ee.technion.ac.il

ABSTRACT

The Parsec benchmark suite is widely used in evaluation of parallel architectures, both existing and novel, the latter through simulation. In particular, it is used for evaluation of highly parallel architectures. It is well known that parallelism bottlenecks occur both in the architecture, (e.g., shared-resource contention) and in the algorithm, (e.g., data-dependency). In this paper we study the latter, i.e., the inherent parallelism scalability and the dynamic behavior of the benchmark programs themselves, independently of the architecture.

To this end, we present a new simulator that performs efficient, functionally accurate, simulation of a hypothetical ideal parallel architecture with no parallelism bottlenecks, where any measured parallelism limitation is necessarily due the benchmark itself. By applying this methodology to a continuum of simulated machines, ranging from a few processors to thousands of processors, we characterize the dynamic behavior and scalability of different benchmarks. We find that only a quarter of the Parsec benchmarks truly scale well to hundreds of processors. Moreover, somewhat surprisingly, we find the Amdahl effects are responsible for lack of scaling in only about half the non-scalable benchmarks. The rest are limited by their inability to produce sufficient work for all cores, and the others benchmarks' scalability is limited by Amdahl effects.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance attributes.

General Terms

Performance.

Keywords

Parsec; many-core; architecture; simulation.

1. INTRODUCTION

The Parsec benchmark suite [1] is widely used in parallel architectures research. While the benchmarks in this suite are implemented for mainstream operating systems (Linux and Microsoft Windows), the workloads are relevant to parallel architectures regardless of the OS. The primary goal of our work is to study the inherent behavior of multithreaded application programs on many-core machines, excluding effects of the operating system.

Parallel benchmarks are compute-intensive, hence I/O and kernel space processing is not only unessential for performance measurements, it actually constitutes noise because of compute resource consumption by background processing (e.g., interrupts, daemons). Indeed the benchmarks are typically used in settings that attempt to prevent OS-level scheduling [2] and sometimes even filtering out kernel-space computation from the measurements [3]. Furthermore, the Parsec benchmarks define the concept of Region-Of-Interest (ROI), separating the compute part of the benchmark from setup and cleanup, which typically include environment-specific operations such as resource acquisitions, (e.g., memory, synchronization objects, thread creation) and I/O (for both input data sets and final results).

This work was partially supported by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI) by an Intel Heterogeneous Computing research grant and by G. S. Elkin Research Fund.

In order to study the scalability of Parsec benchmarks, we develop a novel simulator that can execute Linux programs, and thus supports any Linux benchmark, but takes into account only user-space code such that it is oblivious to the OS' scheduler. This saves the need to strip down the OS to minimize background processing interference, and also allows us to simulate a larger number of cores than supported by the underlying OS. In addition, the simulator has special provisioning to restrict measurements to the ROI.

We use our new simulator to measure how performance scales as the number of cores is increased. Moreover, we extract the dynamics of the number of active threads over the course of the execution. This illustrates phases in the benchmark's life cycle, and highlights where parallelism bottlenecks lie within the code.

We use an ideal parallel architecture model, one with no contention on shared resources, (e.g., caches, interconnect), a perfect core (1 cycle per instruction), and a perfect memory hierarchy (taking 1 cycle per memory access). This allows us to isolate the inherent performance properties of the benchmarks.

We profile the Parsec benchmarks with a range of threads count and corresponding numbers of processors. Of the 12 benchmarks we study, we find that only three scale well to many hundreds of processors, whereas the rest peak between tens to a couple of hundreds of processors. Of the latter, four are limited by their inability to provide enough work for all available processors, and five are limited due to a classical Amdahl law effect [4].

2. RELATED WORK

The Parsec benchmark suite is implemented on top of the Linux OS and is popular in Linux environments, running both on physical machines [5] and full-system simulators [6][7][8]. The benchmarks themselves, however, represent emerging parallel workloads, and are not inherently dependent on Linux. One technique to profile the benchmarks on a novel architecture or on a many-core architecture not supported by Linux is to collect traces in a Linux environment with the benchmark running as many threads as there are cores in the target architecture [9][10] and using these in a trace-driven simulation. However, this technique loses timing-dependent functional effects, which may lead to inaccurate performance prediction. In contrast, our simulator employs a feedback loop between the trace collection and the trace-driven timing simulation for accurate performance modeling.

A similar mechanism was used in the Graphite simulator [11]. Like our simulator, Graphite also collects traces between inter-thread

interaction points. However, Graphite's main requirement is to minimize single simulation latency, to which end it sacrifices the accuracy of inter-thread dependency effects, e.g., by employing lax synchronization and analytical modeling. In our simulator the tradeoff is the opposite – our main requirement is accurate simulation of inter-dependency effects and we sacrifice simulation latency for that. Specifically, our simulations can exploit only a single physical core, as it runs one thread at a time. The high latency of a single simulation is mitigated by the fact that studies typically involve many independent simulations of different data points and with different parameters, which can run simultaneously on numerous physical cores.

The general characteristics of the Parsec suite on contemporary architectures were studied in earlier work [12][13]. The software's scalability assuming ideal, perfect hardware architecture was studied by standard system simulation for up to 16 cores [2]. In this paper we push the envelope much further – we evaluate the workload on architectures including even thousands of processors. We do so by leveraging our simulator's scalability and efficiency, which allow it to simulate architecture with more cores than supported by the operating system.

3. THE SIMULATOR

Our simulator's architecture resembles that of Graphite [11]. It employs binary instrumentation to observe the dynamic instruction stream, whereas the functional execution is performed by the underlying physical processor. Like Graphite, it uses the Pin binary instrumentation tool from Intel [14] and the simulator is implemented as a Pintool. This means that only the user-space code of the benchmark is simulated.

3.1 Timing simulation

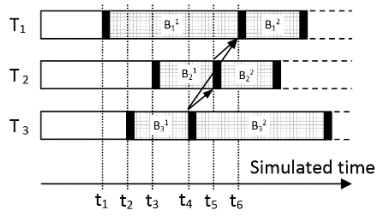
To maintain the timing effect in the functional execution, (and thus achieve the effect of an execution-driven simulation), the instructions are scheduled to the physical processor in the order imposed by the timing model of the simulated architecture. For example, when two threads contend on changing the same memory location (typically using atomic read-modify-write instructions), the one that would execute first according to the timing model is scheduled first to the physical processor. We control the functional execution scheduling through per-thread semaphores. To delay functional execution of a thread, the instrumentation code (simulator's code) blocks on the respective thread's semaphore. When the timing simulation determines the thread whose next instruction should execute, it signals its semaphore. This unblocks the thread, allowing its next instruction to functionally execute. This means that the Operating system's scheduler and other processes in the system do not affect the timing simulation, thus eliminating potential performance measurement noise.

3.2 Batching functional execution

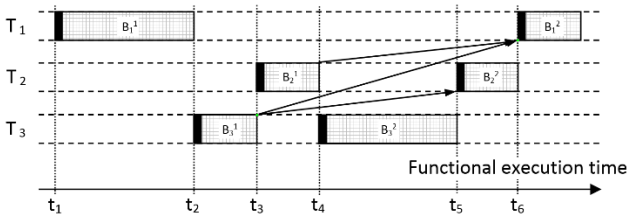
Thread blocking and signaling incurs significant runtime overhead. Therefore it is critical for simulation performance to minimize such synchronization. A key observation towards reducing this overhead is that in order to preserve the effect of simulated timing on the computation, it is not necessary to functionally execute instructions in the global order imposed by the timing model – it suffices to execute them in dependency order. Inter-thread instructions dependency occurs only when there are shared operands. With Linux supporting processors with relaxed memory consistency models [15], sharing operands requires the use of memory fences to make visibility order guarantees. Hence, it can be assumed that memory operands are not shared unless the sharing threads delineate these accesses explicitly using fence instructions. It therefore suffices to execute only fence instructions in the order imposed by the timing model.

This observation allows us to functionally execute multiple instructions of a single thread continuously without blocking as long as no memory fence is encountered. Once a memory fence instruction occurs, its functional execution is delayed until all memory fence instructions in other threads that precede it according to the timing model have been functionally executed. This way subsequent instructions see the results of other threads' instructions that precede them according to the timing model.

An example functional execution is illustrated in **Figure 1**. It shows the instruction streams of three threads: T_1, T_2 and T_3. The small black rectangles denote memory fence instructions. **Figure 1(a)** shows the execution timing in the simulated architecture. The arrows show dependency order, from instructions that are ordered earlier to ones that are ordered later. **Figure 1(b)** shows a functionally equivalent execution: memory fences are executed in the same order as in **Figure 1(a)**, preserving inter-thread instructions ordering.



(a) Timing model on the simulated architecture



(b) Valid functional execution of (a) under a weak memory consistency model

Figure 1: Functional execution scheduling

The simulator exploits this degree of freedom in the functional execution order, and batches the execution of consecutive instructions preceding a memory fence while collecting traces. It then executes the timing model simulation on the traces to determine the thread that first executes its next instruction (a memory fence). This thread is then allowed to execute and collect a new trace and so on. The simulation process thus alternates between two phases: trace collection from a single thread until a fence is encountered, and timing model simulation.

It should be emphasized that this functional execution batching is just a simulation runtime optimization. It has no effect on the timing simulation because the simulated architecture has strong memory consistent model and no pipeline or out-of-order execution, hence a memory fence is equivalent to NOP instruction.

3.3 Thread blocking

Since kernel-space code is not simulated, the simulator needs to know when a thread is blocked inside the kernel in order to exclude it from performance counters and from scheduling.

We use a simple scheme: every syscall is assumed to be blocking. If it is not blocking, it will return very quickly in terms of simulation time, because the kernel is not simulated. If it is indeed blocking, it will return when another thread performs the unblocking operation, which occurs at the correct simulation time.

3.4 Region-Of-Interest (ROI) Detection

The Parsec benchmarks mark the beginning and end of the ROI by calling functions **`__parsec_roi_begin`** and **`__parsec_roi_end`**, respectively. The former is called just before spawning the worker threads, after setting up the data set to operate on. The latter is called just after waiting for all threads to exit and any output data is generated after it. It would have been simple to detect the calls to these functions in the simulator and restrict the measurement accordingly. However, with hundreds and thousands of threads, just the creation of the threads and waiting for their exit, which occur inside the ROI, may affect the performance results. Indeed, two benchmarks (***blackscholes*** and ***raytrace***) use a pthreads barrier at the beginning of the worker threads to ensure that worker threads do not make progress before all are created. However, the pthreads barrier implementation, whose user-space part of the code is included in the simulation, has the last thread that enters the barrier unblock all other threads in a loop. This is a serial operation whose length is proportional to the number of threads, which has a significant impact on performance results.

We solve this predicament by introducing the concept of a simulator-level barrier. The simulator can be given a name of a function in the simulated program that when entered is considered as a wait on the simulator's barrier, meaning that the thread gets blocked by the simulator itself, i.e., it is excluded from simulation scheduling. Only when the designated function is entered the specified number of times (the expected number of worker threads) all are unblocked and it is taken as the ROI-begin mark. Since this is done in the simulator, unblocking is instantaneous in simulated time, i.e. takes zero simulation time for all threads.

Complementary to the ROI-begin detection, which minimizes serial effects during startup, we select the ROI-end time so as to minimize serial effects at the end of the computation. If one chooses the ROI-end as the point where all threads have exited, the measurements become sensitive to tail effects – different threads may finish at different times, and the parallelism degree is reduced towards the end of the execution. To avoid this tail effect, we take the ROI-end to be the time when the first worker threads exits. The simulator detects this simply by detecting the first time the benchmark exits from the function that is designated to be the simulator's barrier entry function, (which is typically the worker thread entry function).

4. EVALUATION METHODOLOGY

The metric we use for performance is average number of *instructions-per-cycle* (*IPC).* This metric is more appropriate than the conventional *total execution-time* metric because the latter allows meaningful comparison only when the size of the problem is constant across different parallelism degrees. In contrast, IPC is appropriate also when benchmarks may adjust the size of the problem according to the parallelism degree, as some of the Parsec benchmarks do. A weakness of the IPC metric is that it is not appropriate when programs perform busy loops (e.g., spin-locks) or speculative computations (e.g., transactional memory programs), since it counts instructions that do not contribute to the task [16]. This is not a problem with Parsec benchmarks, since they do not perform speculative computations. The IPC is calculated from the simulator's performance counters – the *total-instructions* counter and the *execution time* counter.

We use up to 1984 worker-threads. This upper limit is not inherent: it is derived from a limitation of the Pin binary instrumentation framework, which supports programs with up to 2048 threads. Since most of the benchmarks have a control thread in addition to the worker-threads, these benchmarks cannot be instantiated with 2048 worker-threads under Pin. 1984 was chosen because it is the largest multiple of 64 that is smaller than 2048.

Not all benchmarks can create that many threads. Some benchmarks have hard-coded limits on the number of threads; we increased these limits to 2048. Some benchmarks impose constraints on the number of worker-threads, e.g., requiring them to be a power of two. Some do not run properly with thousands of worker-threads. Some spawn multiple threads per worker-thread so they hit the 2048 thread limit of the Pin binary instrumentation framework with a lower number of threads-count parameter. We indicate the relevant limitations in the results of every specific benchmark.

4.1 Simulation environment

We used Parsec suite version 2.1[1], compiled with gcc 4.6.3 in **gcc-hooks** mode. The ***freqmine*** benchmark does not support this mode and therefore was not studied.

The input set used was **simmedium**.

The simulation was executed on an HP Proliant DL785 G5 machine, with 8 AMD Opteron™ Processor 8356 Quad-Core (total of 32 cores) with 128GB RAM, running Linux Ubuntu 12.04 LTS, 64-bit.

4.2 Results presentation

For each benchmark, we show the graph of its performance scalability and a graph of its dynamic behavior.

The performance is plotted (in *IPC* units) vs. the number of cores. We include two curves – the actual performance (solid line) and the theoretical maximum performance, i.e., if threads were never blocked (dotted line). The latter is provided by **equation (1)**. $\boldsymbol{CPI_{exe}}$ is the Cycles-Per-Instruction for the compute part of the instruction, $\boldsymbol{t_m}$ is the average memory operand access latency (in cycles) $\boldsymbol{r_m}$ is the average number of memory operands per instruction and $\boldsymbol{n}$ is the number of cores (and threads).

$$\textbf{(1)}\ \boldsymbol{Perf[IPC]} = \frac{\boldsymbol{n}}{\boldsymbol{CPI_{average}}} = \frac{\boldsymbol{n}}{\boldsymbol{CPI_{exe}+r_m \cdot t_m}}$$

In our ideal parallel architecture model both CPI_{exe} and t_m are one cycle. n is a parameter of a specific execution and the respective r_m is extracted from the simulator's performance counters.

The dynamic behavior graph is plotted as the number of running threads, (i.e., threads that are not blocked waiting for other threads) in the vertical axis vs. the time (in cycles) in the horizontal axis. This shows the pattern of inter-thread interaction over time. This graph is different for executions with different numbers of threads. We choose one representative graph for each benchmark, and explain it.

5. SIMULATION RESULTS

We identify three families of benchmark parallelism behavior: Linear scalability, scalability limited due to increasingly dominating serial phase and scalability limited due to bounded degree of parallelism.

5.1 Linear scalability

The ***fluidanimate*** benchmark has all worker threads running most of the time across the range of parallelism that was profiled (up to 1024 cores) as shown in Figure 2. Still, there are phases of reduced parallelism which make the performance less than ideal as shown

[1] With three patches: ***Syntax error in open()***, ***Deadlock in ferret*** and ***Missing barrier in streamcluster***.

in Figure 3. However, the speedup is linear, indicating that the reduced parallelism part is proportional to the overall run time rather than increasingly dominating it.

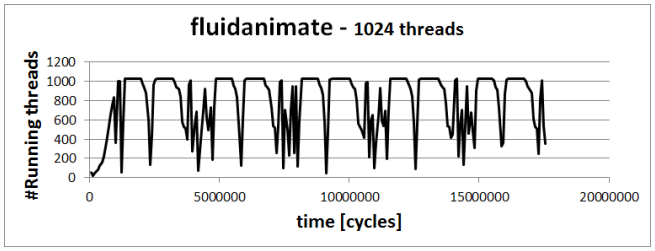


Figure 2: *fluidanimate* dynamics – good parallelsim

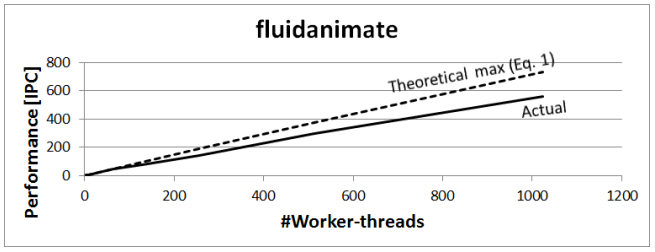


Figure 3: *fluidanimate* performance – linear speedup

The ***streamcluster*** benchmark does not have a serial phase as shown in Figure 4. Also, the average number of running threads is proportional to the number of worker threads hence it demonstrates linear scalability as shown in Figure 5.

It should be noted that this benchmark uses an input data-set that is proportional to the number of threads[2]. Therefore, it fits the parallelism scalability model used in Gustafson Law [17].

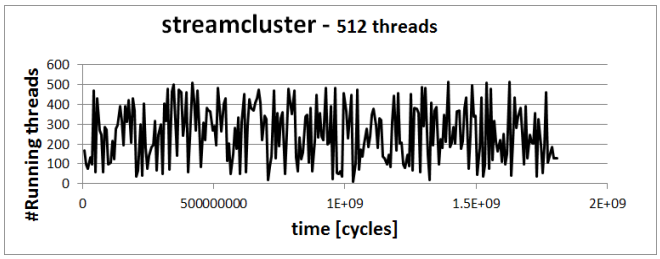


Figure 4: *streamcluster* dynamics – average running threads proportional to the number of worker-threads

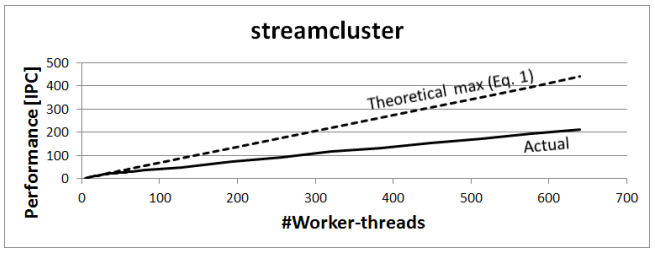


Figure 5: *streamcluster* performance – linear speedup

The ***blackscholes*** benchmark is embarrassingly parallel, meaning it operates on a large dataset and there is no dependency between the operations on different part of the set. Therefore, its threads do not communicate, hence never block, thus achieving perfect scalability – the actual performance and the theoretical maximum performance curves overlap (the graph is omitted due to space considerations).

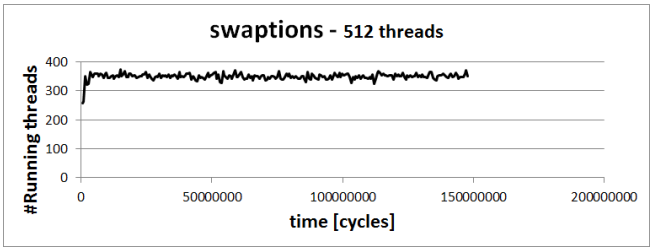


Figure 6: *swaptions* dynamics – bounded parallelism

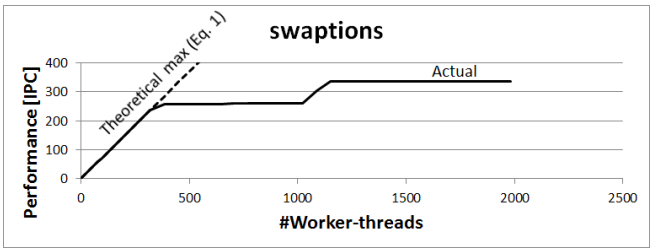


Figure 7: *swaptions* performance – perfect up to 384 threads

The ***swaptions*** benchmark is essentially *embarrassingly parallel* (see ***blackscholes*** in 5.1). However, it employs dynamic memory allocation, which constitutes a shared-resource contention – the heap. Therefore, it has limited parallelism (albeit quite high – 350 threads) as shown in Figure 6. Indeed it demonstrates perfect scalability up to that limit and then it saturates at 384 thread (the smallest simulated configuration with #threads >= 350) as shown in Figure 7. Thus, we effectively measure the scalability of the memory allocator rather than the benchmark itself. There is a second performance plateau that requires further investigation.

5.2 Scalability limited by a serial phase

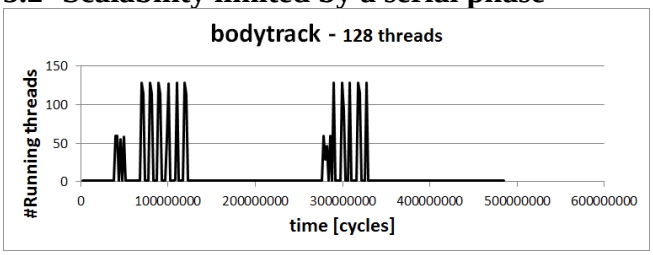


Figure 8: *bodytrack* dynamics – fixed serial phase

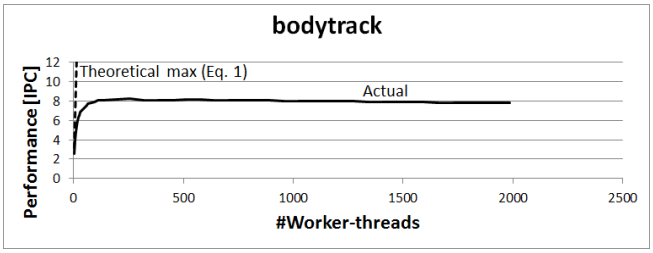


Figure 9: *bodytrack* performance – saturation at ~128 threads

The ***bodytrack*** benchmark has a serial phase which increasingly dominates the execution time as shown in Figure 8. This is classical Amdahl's law effect. The performance saturates at ~128 threads as shown in Figure 9.

The ***facesim*** benchmark has a fixed serial phase which increasingly dominates the execution time (the graph is omitted due to space considerations). This benchmark was profiled only up to 128 threads because there is no suitable input data set for a larger number, so merely increasing the hard-coded limit of 128 is

[2] For this reason its simulation time is proportional to the number of worker threads. For this reason this benchmark was profiled only up 640 threads.

insufficient. Hence the performance scalability graph in Figure 10 does not strictly saturate but still show quickly diminishing performance increase with the increase in number of cores.

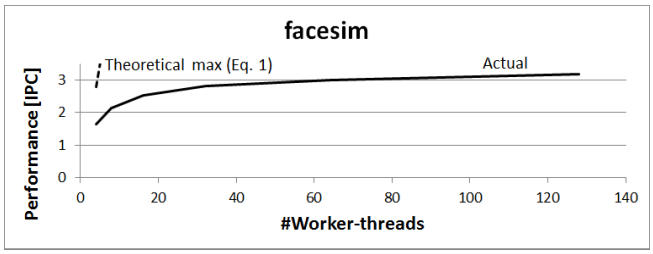


Figure 10: *facesim* performance – diminishing growth

The ***canneal*** benchmark has a serial phase that increasingly dominates the execution time as shown in Figure 11. However, this phase is almost linear in the number of threads (graphs omitted due to space considerations). Indeed the performance scalability graph in Figure 12 shows an effect that is worse than saturation – the performance degrades significantly after peaking at ~256 threads.

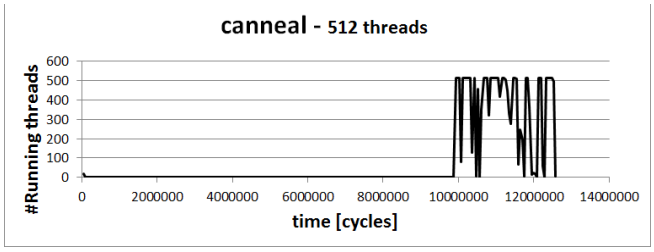


Figure 11: *canneal* dynamics – increasingly long serial phase

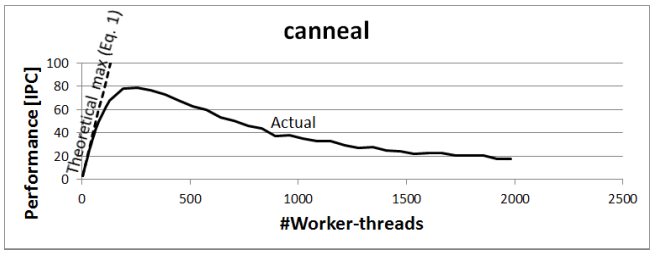


Figure 12: *canneal* performance – peak at ~256 threads

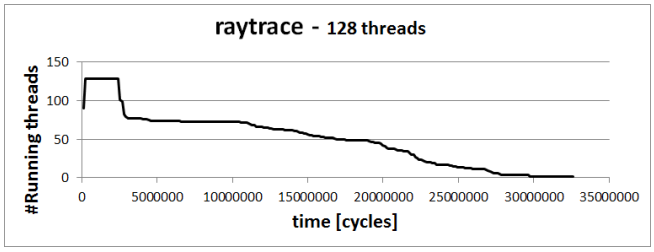


Figure 13: *raytrace* dynamics – fixed bounded parallelism phase

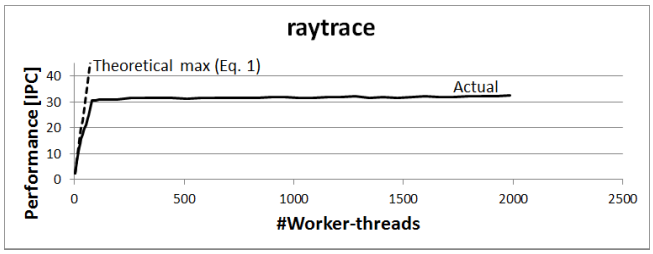


Figure 14: *raytrace* performance – saturation at ~80 threads

The ***raytrace*** benchmark's dynamic behavior is shown in Figure 13. It shows a leading phase with as much parallelism as there are threads but a trailing phase with reduced parallelism that increasingly dominates the execution time[3]. The performance saturates at ~80 worker threads as shown in Figure 14.

5.3 Scalability limited by bounded parallelism

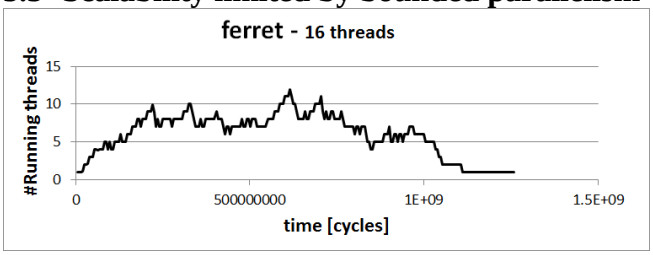


Figure 15: *ferret* dynamics – bounded parallelism

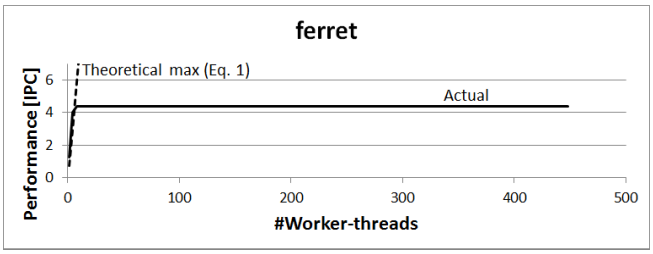


Figure 16: *ferret* performance – saturation at ~8 threads

The ***ferret*** benchmark fails to provide enough parallelism for more than 12 threads to be running simultaneously, as shown in Figure 15. This means that any worker thread beyond that doesn't provide any performance increase. It should be noted that this benchmark is implemented as a 4 stage pipeline, each stage having as much worker threads as specified in the invocation. Hence, invocation with 16 threads can theoretically get up to 64 running threads. However, we see that no more than 12 threads are running at any given time, which explains the performance saturation at 8 threads as shown in Figure 16.

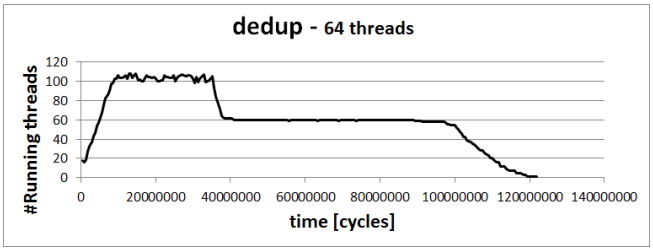


Figure 17: *dedup* dynamics – bounded parallelism

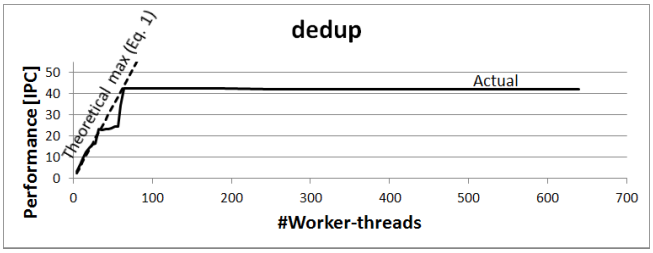


Figure 18: *dedup* performance – saturation at ~64 threads

The ***dedup*** benchmark fails to make all worker threads to run simultaneously, as shown in Figure 17. This benchmark is implemented as a 3 stage pipeline, each having as much worker threads as specified in the invocation. Hence, invocation with 64

[3] This is after the exclusion of thread termination tail effects as described in 3.4.

threads can theoretically get up to 192 running threads. This explains how we get to 100 running threads in an invocation with 64 worker threads. There are 3 different phases: parallelism limited to ~100 threads, to ~60 threads and a tail. This explains the performance saturation at 64 threads as shown in Figure 18.

Although the tail was supposed to be excluded as described in 3.4, for this specific benchmark the tail exclusion method could not be employed because worker threads of different pipeline stages exit at different times, some very early. Thus for this benchmark the ROI-end was taken to be the exit of the last worker thread rather than the first exit.

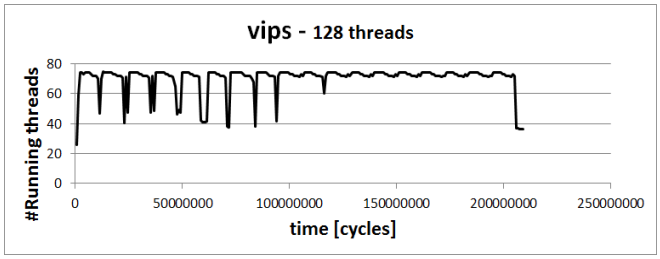


Figure 19: *vips* dynamics – bounded parallelism

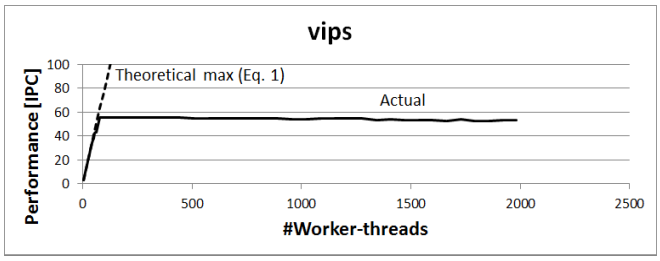


Figure 20: *vips* performance – saturates at 80 threads

The ***vips*** benchmark fails to get more than ~74 threads to run simultaneously as shown in Figure 19. The performance saturates at 80 threads (the smallest simulated configuration with #threads >= 74) as shown in Figure 20.

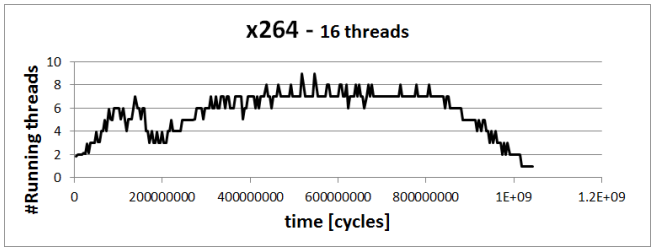


Figure 21: *x264* dynamics – bounded parallelism

The ***x264*** benchmark fails to provide enough parallelism for all the worker threads to be running simultaneously - no more than ~8 threads are running at any given time as shown in Figure 21. For this reason the performance saturates at ~8 worker threads (graph omitted due to space considerations).

6. CONCLUSIONS

In this paper, we used a perfect parallel architecture model to capture the inherent parallelism characteristics of the Parsec benchmarks. To this end, we developed a functionally-accurate simulator that can scale to thousands of threads.

Our scalability study has identified three parallelism families:

1. Linear scalability - linearly scale up to many hundreds of cores.
2. Parallelism scalability limited by a serial (or bounded parallelism) phase that increasingly dominates the execution time (Amdahl's Law effect).
3. Parallelism scalability limited by bounded parallelism – worker threads are never all active simultaneously.

We further presented dynamic behavior graphs, showing the number of running threads over time, in order to capture the effect of data dependency on the parallelism and to identify phases with different parallelism characteristics. This can be used to guide parallelism improvement of benchmarks.

REFERENCES

[1] C. Bienia and K. Li, "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors", In Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation, June 2009.

[2] Bryan, Paul D., et al. "Our many-core benchmarks do not use that many cores." System 6 (2009): 8.

[3] Keramidas, Georgios, Nikolaos Strikos, and Stefanos Kaxiras. "Multicore Cache Simulations Using Heterogeneous Computing on General Purpose and Graphics Processors." Digital System Design (DSD), 2011 14th Euromicro Conference on. IEEE, 2011.

[4] G.M. Amdahl, "Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities," Proc. Am. Federation of Information Processing Societies Conf., AFIPS Press, 1967, pp. 483-485.

[5] Bhadauria, Major, Vincent M. Weaver, and Sally A. McKee. "Understanding PARSEC performance on contemporary CMPs." Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. IEEE, 2009.

[6] Trivino, Francisco, et al. "Self-related traces: An alternative to full-system simulation for nocs." High Performance Computing and Simulation (HPCS), 2011 International Conference on. IEEE, 2011.

[7] Tan, Zhangxi, et al. "RAMP gold: an FPGA-based architecture simulator for multiprocessors." Proceedings of the 47th Design Automation Conference. ACM, 2010.

[8] Patel, Avadh, et al. "MARSS: A full system simulator for multicore x86 CPUs." Proceedings of the 48th Design Automation Conference. ACM, 2011.

[9] Moeng, Michael, Sangyeun Cho, and Rami Melhem. "Scalable Multi-cache Simulation Using GPUs." Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on. IEEE, 2011.

[10] Chang, Kuei-Chung, Ming Liao, and Chiu-Han Liao. "Improving performance of multi-core NUCA coherent systems using NoC-assisted mechanisms." The Journal of Supercomputing 62.3 (2012): 1318-1337.

[11] J. E. Miller et al, "Graphite: A distributed parallel simulator for multicores", HPCA-16, January 2010.

[12] Bhadauria, Major, Vincent M. Weaver, and Sally A. McKee. "Understanding PARSEC performance on contemporary CMPs." Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. IEEE, 2009.

[13] Bienia, Christian, et al. "The PARSEC benchmark suite: characterization and architectural implications." Proceedings of the 17th international conference on Parallel architectures and compilation techniques. ACM, 2008.

[14] Luk, Chi-Keung, et al. "Pin: building customized program analysis tools with dynamic instrumentation." ACM SIGPLAN Notices. Vol. 40. No. 6. ACM, 2005.

[15] Adve, Sarita V., and Kourosh Gharachorloo. "Shared memory consistency models: A tutorial." *computer 29.12* (1996): 66-76.

[16] Alameldeen, Alaa R., and David A. Wood. "IPC considered harmful for multiprocessor workloads." IEEE Micro 26.4 (2006): 8-17.

[17] Gustafson, John L. "Reevaluating Amdahl's law." Communications of the ACM 31.5 (1988): 532-533.