PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors

Christian Bienia and Kai Li
Department of Computer Science, Princeton University
{cbienia,li}@cs.princeton.edu

ABSTRACT

The second version of the Princeton Application Repository for Shared-Memory Computers (PARSEC) has been released. PARSEC is a benchmark suite for Chip-Multiprocessors (CMPs) that focuses on emerging applications. It includes a diverse set of workloads from different domains such as interactive animation or systems applications that mimic large-scale commercial workloads.

The next version of PARSEC features several improved and one new workload. It also supports an additional parallelization model. Many patches and changes were included which simplify the use of PARSEC in practice. The benchmarks of the new suite have higher scalability and cover a larger number of emerging applications. In this paper we discuss the major changes in detail and provide the information necessary to interpret results obtained with PARSEC 2.0 correctly.

Categories and Subject Descriptors

D.0 [**Software**]: [benchmark suite]

General Terms

Performance, Measurement, Experimentation

Keywords

benchmark suite, performance measurement, multithreading, shared-memory computers

1. INTRODUCTION

The goal of the first version of PARSEC was to provide a selection of next-generation workloads to enable the development of future CMPs. The PARSEC team believes that such a benchmark suite should satisfy the following five criteria [2]:

- It is composed of multithreaded applications.
- It focuses on emerging workloads.
- It is diverse enough to represent the increasingly heterogeneous ways in which multiprocessors are used.
- Its workloads employ state-of-art techniques.
- It should support research.

An assessment of existing benchmark suites showed that no single suite before PARSEC satisfied all five conditions. That is why PARSEC was created. A statistical comparison of PARSEC and SPLASH-2 revealed significant, systematic differences [1]. These findings suggest that parallel programs have changed considerably since the first wave of commercially available multiprocessor machines, which means that the use of PARSEC instead of older suites will impact the results obtained.

Until today PARSEC has been downloaded over 1,500 times. It is now in use at virtually all major research institutions worldwide, as the list of acknowledgments in Section 8 suggests. The first publications using PARSEC have been released.

The initial version of PARSEC was made publicly available in January 2008. The second version followed in February 2009. Its major improvements over the original release are several new and enhanced workloads as well as additional parallelization models. How does it differ from the first version of the benchmark suite?

To answer this question this paper makes the following contributions:

- We motivate the need for the second version of the benchmark suite.
- We describe some common emerging applications and how well PARSEC covers them.
- We provide the information necessary to understand results obtained with PARSEC 2.0.

2. MOTIVATION

The initial version of PARSEC was an improvement over existing benchmark suites. However, despite the fact that it made several new types of workloads available for performance measurement and research, it had its shortcomings. Important emerging application domains remained uncovered. New parallelization models for CMPs are being developed and should be considered in a state-of-art benchmark suite. And the increasing use of PARSEC workloads continues to reveal bugs and other issues that should be addressed. The second version of PARSEC was created with the following goals:

- Improve its existing workloads.
- Increase the application coverage of the suite.
- Simplify its use in practice.

The new suite covers a wider spectrum of emerging applications: One new application, `raytrace`, was added. It represents important tasks commonly found in emerging interactive animations such as frame rendering or visibility detection. We will investigate this area of emerging applications in Section 3.

The improvements make a substantial difference. All PARSEC 1.0 benchmark programs were altered in some way, in four cases the changes are so significant that the characteristics of the workloads are noticeably affected. Support for a new parallelization model, Intel Threading Building Blocks (TBB), was added to several PARSEC benchmarks. For example, the original version of the `bodytrack` program suffered from limited scalability because it contained an important unparallelized kernel. It supported two parallelization models: OpenMP and pthreads. The scalability issue was addressed for the second version of the benchmark suite. Figure 1 shows a direct comparison of the two `bodytrack` versions

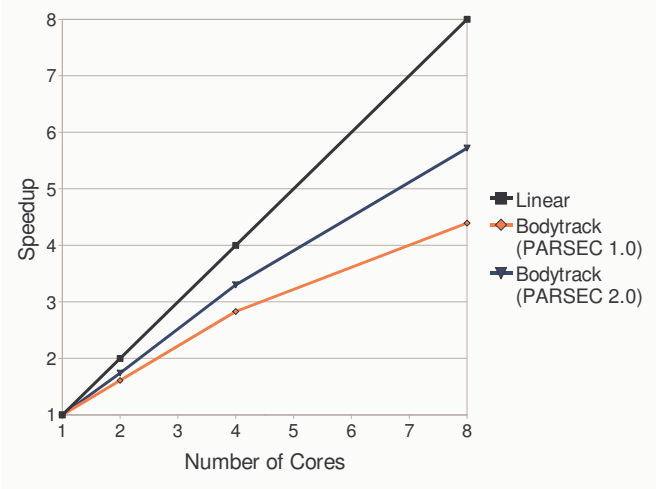

Figure 1: Comparison of the speedups of the `bodytrack` workloads of PARSEC 1.0 and 2.0. The improved parallelization achieves 30% more performance with 8 threads on a real 8-way multiprocessor machine.

on a real 8-way multiprocessor machine. The `bodytrack` benchmark of PARSEC 2.0 exhibits a 30% higher performance with 8 threads. The memory requirements of the feature extraction phase continue to limit its speedup in practice. The new version of the workload now also supports TBB. Major changes like these will be discussed in Section 5.

The handling of PARSEC 2.0 in practice has been simplified. All PARSEC workloads now support the Solaris/Sparc platform. This feature increases the number of simulators that can be used with PARSEC. Numerous bugs were fixed. An online documentation system was added which will help PARSEC users to leverage the full potential of the suite.

3. EMERGING APPLICATIONS

In this section we discuss two types of emerging applications that were a main focus during the development of PARSEC 1.0 and 2.0: Video games and virtual worlds. We will begin with an explanation of what an emerging application is, which will be followed by an analysis of these two application areas. We conclude this section with a coverage analysis of the PARSEC benchmark suite to illustrate how well it represents the different components of the presented emerging applications and why the inclusion of the new `raytrace` workload is an improvement.

What defines an emerging application and what not has an inherent degree of subjectivity to it. Opinions can differ without necessarily invalidating each other. The definition used by the PARSEC benchmark suite is derived from the scientific Grand Challenge problems:

1. The application is hard to run well, it requires significant performance increases.

2. It is feasible to run the application, the required performance is considered to be attainable.

3. There is strong demand for the application.

The motivation for this analogy to the Grand Challenges is to obtain a set of benchmarks that can drive research. In fact, parallel computing itself is considered a Grand Challenge due to its significance for many applied sciences, and several PARSEC workloads solve problems that are directly related to Grand Challenges, such as computational fluid dynamics for example. Just like the Grand Challenges help to focus research efforts, emerging applications can channel computer architecture research by describing a common goal that puts the need of computer users into the center of attention.

3.1 Video Games

Video games have become a driver for the entire computer industry. They are arguably the most important and demanding type of application on clients. Computer processors are now being developed with the primary goal of improving the gaming experience [11]. No performance analysis should omit modern computer game workloads from its benchmark selection.

The focus of game development has traditionally been on the creation of more realistic graphics. However, as the visual quality of games is approaching photorealism this is no longer sufficient for a successful game. Players are increasingly demanding a more immersive environment and are asking for more realistic physics and improved artificial intelligence (AI).

Figure 2 shows a breakdown of the computational kernels of next-generation games. After processing the user input a game needs to update the virtual world by calculating any necessary physical effects, such as object deformations or fluid movement. It also needs to determine the new strategies of the Non-Player Characters (NPCs). After these decisions the game world can be animated. During the course of these steps collisions of impenetrable objects can occur, which need to be detected and resolved. Finally, the screen can be updated by rendering the output of the game. Online games work in a very similar fashion but perform several of these steps on the server. The game client only animates and renders the frames. A typical feature of games is that the game world data is static and available to both the client and the server. It can therefore be distributed and installed offline. Only the position and number of objects need to be updated during runtime. Some of the most popular online games to date are *World of Warcraft* and *Eve Online*, which typically simultaneously serve thousands of clients per server.

3.2 Virtual Worlds

Virtual worlds are highly immersive, simulated realities. On first glance they resemble online games, but they typically allow a wider degree of actions and accept user-generated content. Some virtual worlds like *Second Life* function like a 3D Internet and aim to create another interface for online content [7]. Virtual worlds are considered one of the most important emerging applications. As generic online games become more complex and realistic, they become increasingly similar to virtual worlds. For example, the space combat game *Eve Online* has announced an extension named *Ambulation* that will allow players to leave their ships at space stations and interact with each other. No form of combat will be possible outside of space ships, but players will be able to socialize in bars or casinos, use mission rooms for briefings or sell self-made clothes to other players.

The fact that virtual worlds allow user-generated content creates the problem of distributing this content efficiently to the clients. Unlike in the case of games, clients only have partial information about the world. In Figure 2 we compare virtual worlds to traditional games. Three differences can be identified: First, virtual world servers must employ some form of visibility detection to determine which content must be sent to the clients. Virtual worlds

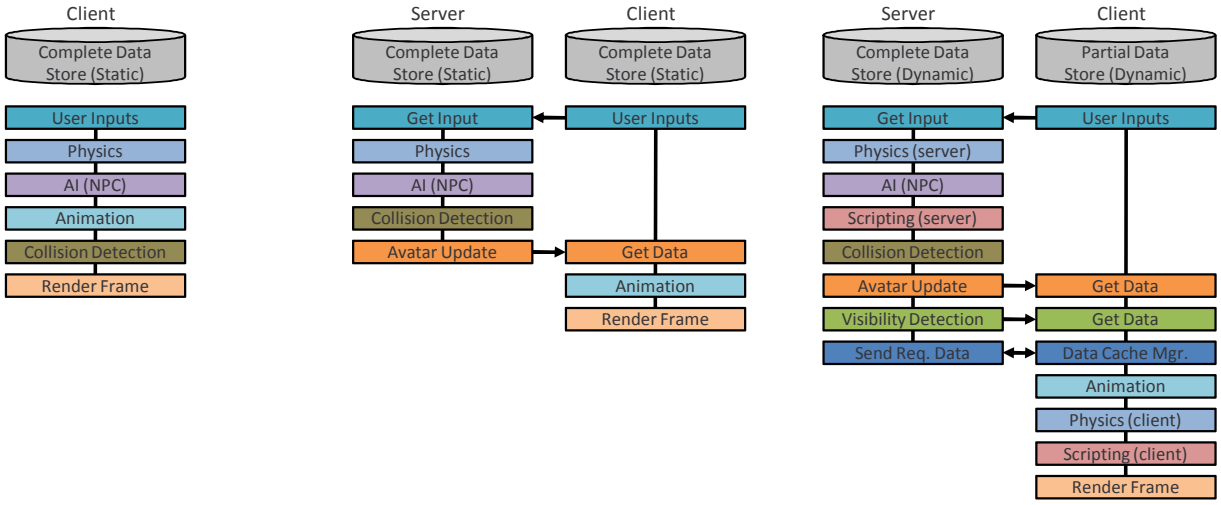

Figure 2: Breakdown of computational kernels in computer games (left), massive multiplayer online games (middle) and virtual worlds (right). Online games must perform some of the client operations centrally on the game servers. Virtual worlds furthermore allow user-generated content, which poses the additional problems of animating it with scripts and determining when it is visible and must thus be sent to the user. Virtual world clients do not have the entire world stored locally, unlike computer game clients.

can also execute scripts to give behavior to user-generated in-world objects. *Second Life* offers the *Linden Scripting Language* (LSL) that its users can use for this purpose. A third difference compared to online games is that virtual world clients typically take on more work to create additional realism. Like the server they also simulate physical effects and execute scripts. These differences allow virtual worlds to create a more immersive reality than current online games, at the cost of significantly increased performance requirements: A server for a virtual world like *Second Life* can usually handle no more than a few dozen clients. This is several orders of magnitudes less than the typical numbers for an online game server.

3.3 Coverage Analysis

In this section we will briefly discuss how well PARSEC covers video games and virtual worlds. A direct comparison of the computational kernels of the presented emerging applications with the relevant PARSEC workloads can be seen in Table 1. The original release of the benchmark suite already contained `facesim` and `fluidanimate` which perform physics simulations and animations. Realistic animation often requires simulation of physical effects to obtain natural behavior, which means that both tasks use similar computations.

Application Kernel	PARSEC Benchmark
Physics, Animation	`facesim`, `fluidanimate`
AI (NPC)	
Scripting	
Collision Detection	`raytrace`
Visibility Detection	`raytrace`
Render Frame	`raytrace`

Table 1: Computational kernels of virtual worlds and which PARSEC 2.0 kernels they are covered by.

For PARSEC 2.0 the `raytrace` workload was included. This benchmark renders a 3D scene so that it can be seen on the screen by a human observer. The basic idea of the ray tracing method is to shoot rays into a scene and compute where they hit objects. A new set of rays is then created at each intersection point to simulate effects such as reflections and refractions. To accelerate this process ray tracers usually use a data structure that is called a Bounding Volume Hierarchy (BVH). A BVH organizes the entire scene in a tree structure, which means that by descending down from the root ray tracers can find ray-surface intersection points extremely fast. A more detailed description of the `raytrace` workload with its core algorithms and data structures can be found in Section 5.1.4.

The ray tracing method can also be used for collision and visibility detection. Identifying the set of visible surfaces is a subproblem of visualization. It is hence automatically computed for every frame. A visibility detection algorithm related to ray tracing is ray casting [10]. It resembles ray tracing but only casts the primary rays without following reflections and refractions. Ray casting can be thought of as a non-recursive version of ray tracing. This means that the set of visible surfaces is a side product of the ray tracing method.

Likewise, the ray tracing method must be able to handle collision detection efficiently because the ray-surface intersections that a ray tracer has to compute are nothing but ray-surface collisions. Collision detection methods usually need to find surface-surface intersections instead of the ray-surface intersections that are computed by the `raytrace` workload. This requires different computations for the actual intersection test in the inner-most loop. However, collision detection methods usually implement BVHs to quickly reduce the set of surfaces to consider, which will result in similar access patterns and sharing behavior.

Another difference of collision detection methods is that a collision response must be computed once a collision is detected. Examples are object deformations or a "bouncing off" effect for which ragdoll physics are frequently employed. Realistic collision behavior takes advantage of physics simulation, which is already included in the PARSEC suite as discussed earlier.

Because of these similarities we conclude that PARSEC covers next-generation online games and virtual worlds well. The `raytrace` workload was a significant addition to PARSEC 2.0 that greatly improves the coverage of the suite. The only two major components of emerging virtual world applications that are left are next-generation AI and scripting.

4. METHODOLOGY

All measurements presented in this paper were conducted on an 8-way multiprocessor machine. The operating system was Linux, the processors were AMD Opterons. Statistics were collected using Pin [9]. Pin is a tool for dynamic instrumentation of programs. It is similar to the ATOM toolkit for Alpha machines, but unlike ATOM the instrumentation and analysis code is injected by Pin during runtime.

We used the precompiled PARSEC 2.0 binaries for our experiments that are available from the PARSEC website. The binaries were compiled with `gcc` 4.2.1.

5. THE PARSEC 2.0 BENCHMARK SUITE

The second release of the PARSEC benchmark suite extends and improves the original program selection of PARSEC. All workloads were modified in some way so that researchers should not automatically assume that results obtained with PARSEC 1.0 and 2.0 are comparable. In most cases the impact on the characteristics should be limited so that some degree of continuity can be expected, however four benchmarks - `bodytrack`, `canneal`, `dedup` and `x264` - were overhauled significantly and are now very different from their original version. One new benchmark - `raytrace` - has been added to the program selection, increasing the total number of workloads that are part of the PARSEC benchmark suite to 13. The impact of these changes is summarized in Tables 2 and 3.

Another important area in which PARSEC has been improved is support of additional parallelization models. The programming model of parallel systems has not matured yet, and no satisfactory standard method to parallelize programs currently exists. This makes it important to consider the impact of alternate parallelization models. The original release supported pthreads and OpenMP. With TBB a third parallel programming paradigm has been added. It is currently supported by five workloads. Support for OpenMP has also been extended so that researchers have more choice when they decide on a set of benchmarks and parallelization models to use.

PARSEC has also been improved in many other ways besides these major changes so that the suite is now much easier to use in practice. All PARSEC tools and the source code of the hooks instrumentation library are now fully documented. Important concepts are also explained in detail. The documentation is accessible at the command line in the form of `man` pages. An HTML version of the `man` pages has been made available online on the PARSEC website. One of the most requested features that has been added is improved program portability, especially support for big-endian architectures. All workloads now auto-detect the endianness of their architecture at runtime and automatically apply any transformations necessary. Numerous changes to the syntax of the programs and their build systems make PARSEC easier to compile on new architectures. Due to these updates PARSEC now fully supports at least the following three platforms: Linux/x86, Linux/Itanium and Solaris/Sparc. Binary distributions for Linux/x86 and Solaris/Sparc with precompiled binaries for all build configurations have been made available on the PARSEC website.

We describe the five workloads that are new or have been significantly overhauled in Section 5.1. The parallel programming models available are discussed in Section 5.2.

5.1 New and Improved Workloads

This section summarizes the most important changes to the workloads included in PARSEC 2.0. In all cases listed here the updates affected fundamental properties of the program such as instruction count, scalability or the locality of memory references.

5.1.1 bodytrack

`Bodytrack` is an application which tracks the body pose of a human with multiple cameras. The program uses computer vision algorithms to extract all necessary image features from the video streams. An annealed particle filter is employed to pin down the exact location and pose of the body on the images. The use of an annealing algorithm makes the body tracking program robust and flexible enough to solve the problem without further help such as markers or any constraints. The program was included in the PARSEC benchmark suite because computer vision algorithms play an increasingly important role in many application areas.

The program works as follows: First, `bodytrack` extracts the needed image features from the set of images that form the current observation. The program then makes an annealing run through all layers of the annealed particle filter. Each layer is initialized with the particles which are the result of the previous filter update, and each particle is an instance of the model configuration that describes the location and state of the tracked body. The layer computes a weight for each particle that encodes how likely it is that this particle is a good solution. A new, random set of particles is then drawn. A particle is selected with a probability equal to its weight. After that step the particles are resampled to obtain the final particle set for the next layer. This process is repeated for all layers. After the processing steps for the last layer have been completed the estimated model configuration is computed by calculating the weighted average of all particles. A more detailed description of the workload can be found in [3].

For the second release of the PARSEC benchmark suite the `bodytrack` benchmark was substantially improved. A new TBB version of the workload was implemented. It introduces pipelining to the program to increase the amount of concurrency. This change is achieved implicitly by leveraging the task concept of the TBB library which executes all available tasks concurrently. The pthreads version uses an alternative approach to achieve higher scalability. It loads the input images that form the individual observations using asynchronous I/O so that disk I/O and computations are overlapping. Finally, the particle resampling phase has been parallelized, making it the fourth parallel kernel employed by the program. These changes address the scalability limitations that were reported before [3, 2]. On a real 8-way multiprocessor machine the cumulative effects of the improvements result in 30% higher performance with 8 threads.

`Bodytrack` now uses a total of four parallel kernels:

Edge detection This kernel implements a gradient based edge detection mask. Spurious edges are elimianted by a comparison with a threshold. This functionality can be found in function `GradientMagThreshold`.

Edge smoothing A separable Gaussian filter is used to smooth the edges. The result is converted to a pixel map which encodes the distance of each pixel from an edge. The filter is implemented in function `GaussianBlur`.

Calculate particle weights This kernel analyzes the image features extracted earlier and determines how likely the individual particles are to describe the correct body pose and location. The kernel is executed once for each annealing layer. It is the hot spot of the `bodytrack` workload.

Particle resampling This kernel resamples particles by adding normally distributed random noise to them, thereby effectively creating a new set of particles. This function is implemented in `GenerateNewParticles`. The random number generator used for the task is given by class `RandomGenerator`.

Program	Application Domain	Parallelization		Working Set	Data Usage	
		Model	Granularity		Sharing	Exchange
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Enterprise Storage	pipeline	medium	unbounded	high	high
facesim	Animation	data-parallel	coarse	large	low	medium
ferret	Similarity Search	pipeline	medium	unbounded	high	high
fluidanimate	Animation	data-parallel	fine	large	low	medium
freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
raytrace	Rendering	data-parallel	medium	unbounded	high	low
streamcluster	Data Mining	data-parallel	medium	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
vips	Media Processing	data-parallel	coarse	medium	low	medium
x264	Media Processing	pipeline	coarse	medium	high	high

Table 2: Qualitative summary of the inherent key characteristics of PARSEC benchmarks. Working sets that are 'unbounded' are large and have the additional qualitative property that there is significant application demand to make them even bigger. In practice they are typically only constrained by main memory size.

The input sets for bodytrack did not change:

- test: 4 cameras, 1 frame, 5 particles, 1 annealing layer
- simdev: 4 cameras, 1 frame, 100 particles, 3 annealing layers
- simsmall: 4 cameras, 1 frame, 1,000 particles, 5 annealing layers
- simmedium: 4 cameras, 2 frames, 2,000 particles, 5 annealing layers
- simlarge: 4 cameras, 4 frames, 4,000 particles, 5 annealing layers
- native: 4 cameras, 261 frames, 4,000 particles, 5 annealing layers

5.1.2 canneal

The canneal kernel employs a simulated annealing (SA) algorithm to minimize the routing cost of a chip design. This optimization method tries to pseudo-randomly swap netlist elements. If the swap would decrease the routing cost it is automatically accepted. With a certain probability that is decreasing over time a swap that would increase the routing cost is also accepted so that the design can escape from local minima. The design converges as the number of swaps that can be successfully executed decreases until it is fully stable. Canneal has been included in the PARSEC suite to represent engineering workloads and because of its demanding memory access behavior. A more comprehensive description of the algorithm and the characteristics of the program is available in [3].

For the second generation of the PARSEC suite the program mechanism controlling how the design converges have been changed. Originally each thread of the program kept track of the number of successful swaps for itself. The program would terminate if one of the threads determines that the design has converged. The new version of the benchmark tracks the swaps globally, which slightly increases communication between threads. Moreover, the program now has the additional ability to terminate before the chip design has converged. This behavior can be controlled by passing a new argument to the program that specifies the maximum number of steps to run. This new feature can be used to reduce the run time of the benchmark to some extent without changing its working set sizes or other characteristics much.

The changes of the algorithm permitted a modification of the input sets of canneal to reduce the run time of the benchmark. Its input sets are now defined as follows:

- test: 5 swaps per temperature step, 100° start temperature, 10 netlist elements, 1 temperature step
- simdev: 100 swaps per temperature step, 300° start temperature, 100 netlist elements, 2 temperature steps
- simsmall: 10,000 swaps per temperature step, 2,000° start temperature, 100,000 netlist elements, 32 temperature steps
- simmedium: 15,000 swaps per temperature step, 2,000° start temperature, 200,000 netlist elements, 64 temperature steps
- simlarge: 15,000 swaps per temperature step, 2,000° start temperature, 400,000 netlist elements, 128 temperature steps
- native: 15,000 swaps per temperature step, 2,000° start temperature, 2,500,000 netlist elements, 6,000 temperature steps

5.1.3 dedup

Dedup is a kernel which uses a next-generation data compression method called 'deduplication'. It combines local and global compression to achieve very high compression ratios. This workload was included in the PARSEC benchmark suite because deduplication is becoming a standard method for backup storage systems and bandwidth optimized network appliances.

The benchmark streams the input through a pipeline. First, the program breaks the input stream into coarse-grained chunks that can be processed independently. The second stage employs rolling fingerprinting to fragment the chunks into even smaller pieces. The next stage computes a hash value for each chunk that uniquely identifies its content. The fourth stage builds a global database of chunks that is indexed with the hash values. If a chunk has not been encountered before it is compressed using the Ziv-Lempel algorithm and added to the database. The final stage assembles the output stream that consists of compressed chunks and hash values so that each chunk occurs exactly once in its compressed form. The algorithm is described in more detail in [3].

Program	Problem Size	Instructions (Billions)				Synchronization Primitives		
		Total	FLOPS	Reads	Writes	Locks	Barriers	Conditions
blackscholes	65,536 options	4.90	2.32	1.51	0.79	0	8	0
bodytrack	4 frames, 4,000 particles	14.04	6.08	3.26	0.80	28,538	2,242	518
canneal	400,000 elements, 128 temperature steps	7.00	0.45	1.76	0.88	34	1,024	0
dedup	184 MB data	41.40	0.23	9.85	3.77	258,381	0	291
facesim	1 frame, 372,126 tetrahedra	30.46	17.17	9.91	4.23	14,566	0	3,327
ferret	256 queries, 34,973 images	25.90	6.58	7.65	1.99	534,866	0	1273
fluidanimate	5 frames, 300,000 particles	13.54	4.30	4.46	1.07	9,347,914	320	0
freqmine	990,000 transactions	33.22	0.08	11.19	5.23	990,025	0	0
raytrace	3 frames, $1,920 \times 1,080$ pixels	46.48	8.12	11.07	9.28	105	0	38
streamcluster	16,384 points per block, 1 block	22.15	16.49	4.26	0.06	183	129,584	115
swaptions	64 swaptions, 20,000 simulations	16.81	5.66	5.62	1.54	23	0	0
vips	1 image, 2662×5500 pixels	31.30	6.34	6.69	1.62	33,920	0	7,356
x264	128 frames, 640×360 pixels	14.42	7.37	3.88	1.16	16,974	0	1,101

Table 3: Breakdown of instructions and synchronization primitives of PARSEC 2.0 workloads for input set `simlarge` on a system with 8 cores. All numbers are totals across all threads. Numbers for synchronization primitives also include primitives in system libraries. "Locks" and "Barriers" are all lock- and barrier-based synchronizations, "Conditions" are all waits on condition variables.

To fragment the data `dedup` uses rolling fingerprints that split the input stream at content-dependent locations. This approach ensures that splits do not hide redundancy in the input data. For the second version of PARSEC the fragmentation process was further improved. The algorithm now guarantees that the input buffer does not introduce additional data-independent splits. In the previous program version the algorithm would introduce an artificial split when it reached the end of the input buffer by starting a new data chunk at the beginning of the next buffer, resulting in two data chunks instead of a single chunk if the buffer had been sufficiently large.

The new version of `dedup` also has a significantly improved serial algorithm. The new algorithm moves data chunks through the entire deduplication process at once. This behavior results in significantly improved cache locality and thus higher performance.

The input sets for `dedup` did not change:

- `test`: 10 KB
- `simdev`: 1.1 MB
- `simsmall`: 10 MB
- `simmedium`: 31 MB
- `simlarge`: 184 MB
- `native`: 672 MB

5.1.4 raytrace

The `raytrace` application is an Intel RMS workload which renders an animated 3D scene. Ray tracing is a technique that generates a visually realistic image by tracing the path of light through a scene [13]. Its major advantage over alternative rendering methods is its ability to create photorealistic images at the expense of higher computational requirements because certain effects such as reflections and shadows that are difficult to incorporate into other rendering methods are a natural byproduct of its algorithm. Ray tracing leverages the physical property that the path of light is always reversible to reduce the computational requirements by following the light rays from the eye point through each pixel of the image plane to the source of the light. This way only light rays that contribute to the image are considered. The computational complexity of the algorithm depends on the resolution of the output image and the scene. The `raytrace` benchmark program uses a variety of the ray tracing method that would typically be employed for real-time animations such as computer games because it is optimized for speed rather than realism. The `raytrace` benchmark was included in PARSEC because of the continuing trend towards more realistic graphics in video games and other forms of real-time animation. As of 2009 all major graphics card vendors have announced plans to incorporate ray tracing into their products in one form or another. Commercial computer games adapted to employ ray tracing instead of rasterization have already been demonstrated.

All rendering methods try to solve the rendering equation [6], which uses the physical law of conservation of energy to describe the total amount of outgoing light L_o at location x, direction ω and time t with wavelength λ:

$$L_o(x,\omega,\lambda,t) =$$

$$L_e(x,\omega,\lambda,t) + \int_\Omega f_r(x,\omega',\omega,\lambda,t) L_i(x,\omega',\lambda,t)(\omega' \cdot n) d\omega'$$

The total amount of outgoing light L_o is the sum of the emitted light L_e and an integral over all inward directions ω' of a hemisphere that gives the amount of reflected light. f_r is the bidirectional reflectance distribution function which describes the propor-

Figure 3: The `native` input set of the `raytrace` benchmark is a 3D model of a Thai statue with 10 million polygons, which is about the amount of triangles that need to be rendered per frame for modern video games.

tion of the incoming light L_i that is reflected from ω' to ω at position x and time t with wavelength λ. The term $\omega' \cdot n$ is the attenuation of inward light. Solving the rendering equation gives theoretically perfect results because all possible flows of light are included[1], but because of the high computational demand it is only approximated in practice. The ray tracing method does so by sampling the object surfaces at discrete locations and angles as given by the scatter model.

The scatter model describes what happens when a ray hits a surface. In that case the ray tracing method can generate up to three new types of rays: Reflection rays, refraction rays and shadow rays. Reflection rays are created if the surface of the object is shiny. A reflected ray continues to traverse the scene in the mirrored direction from the surface. The closest surface it intersects will be visible as a mirror image on the surface of the reflecting object. If the object is transparent a refraction ray is generated. It is similar to a reflection ray with the notable exception that it enters and traverses the material. Shadow rays are the method that is used by the ray tracing algorithm to determine whether an intersection point is visible or not. Every time a ray intersects a surface, shadow rays are cast into the directions of every light source in the scene. If a shadow ray reaches its light source then the intersection point is illuminated by that light. But if the shadow ray is blocked by an opaque object then the intersection point must be located in its shadow with respect to that light, resulting in a lower light intensity.

To find intersection points quickly ray tracers store the scene graph in a Bounding Volume Hierarchy (BVH). A BVH is a tree in which each node represents a bounding volume. The bounding volume of a leaf node corresponds to a single object in the scene which is fully contained in the volume. Bounding volumes of intermediate nodes fully contain all the volumes of their children, up to the volume of the root node which contains the entire scene. If the bounding volumes are tight and partition the scene with little overlap then a ray tracer searching for an intersection point can eliminate large parts of the scene rapidly by recursively descending in the BVH while performing intersection tests until the correct surface has been found.

The entry point for the rendering algorithm of the `raytrace` benchmark is the `renderFrame` method of the `Context` class. In the parallel case this function merely unblocks all threads, which start executing the `task` method of the `Context` class for each work unit. Work units correspond to tiles on the screen. The work is distributed using the task queue in the `MultiThreadedTaskQueue` class so that the program is dynamically load balanced. The BVH containing the scene is stored in the `m_bvh` object, which is an instance of the `BVH` class. It uses arrays to store the BVH nodes in a compact way so they can be traversed quickly.

For each frame the program starts traversing this scene graph with the `renderTile_With_StandardMesh` method. The method creates the initial rays and then calls `TraverseBVH_with_StandardMesh` to handle the actual BVH traversal. This function is the hot spot of the `raytrace` workload. It is a recursive function by nature, but to eliminate the recursive function calls a user-level stack of BVH nodes is used. The stack is implemented as an array and accessed with the `sptr` pointer. To further optimize the intersection tests the function considers the origins and directions of rays and handles the different cases with specialized code.

Figure 3 shows the rendered `native` input. The input sets for the `raytrace` workload are defined as follows:

- `test`: 1×1 pixels, 8 polygons (octahedron), 1 frame
- `simdev`: 16×16 pixels, 68,941 polygons (Stanford bunny), 3 frames
- `simsmall`: 480×270 pixels ($\frac{1}{4}$ HDTV resolution), 1 million polygons (Buddha statue), 3 frames
- `simmedium`: 960×540 pixels ($\frac{1}{2}$ HDTV resolution), 1 million polygons (Buddha statue), 3 frames
- `simlarge`: $1,920 \times 1,080$ pixels (HDTV resolution), 1 million polygons (Buddha statue), 3 frames
- `native`: $1,920 \times 1,080$ pixels (HDTV resolution), 10 million polygons (Thai statue), 200 frames

5.1.5 x264

`X264` is a lossy video encoder based on the ITU-T H.264 standard. H.264 improves over previous video encoding standards with many new features that allow it to achieve a higher output quality at the expense of a significantly increased compression time. Next-generation Blu-ray video players already use H.264 video compression, but many other application areas are equally supported by the H.264 standard. The flexibility and wide-spread use of H.264 video compression is the reason for the inclusion of the `x264` application in the PARSEC benchmark suite. We provide a more detailed description of the workload in [3].

For the second release of the PARSEC benchmark suite the `x264` benchmark was updated to a significantly overhauled program version. The H.264 standard only specifies how to decode a compressed video stream. This leaves a large degree of freedom for H.264 encoder development, which makes video encoding an active area of research and development. The updates to `x264` since its first release as part of PARSEC 1.0 reflect the progress that was made in that field since then.

[1]The rendering equation does not consider certain physical effects such as phosphorescence, fluorescence or subsurface scattering.

The large number of updates and changes makes it difficult to summarize them all. Some of the most important updates are support for additional prediction block sizes, a completely overworked subpixel estimation system, the use of low-resolution lookahead motion vectors as an extra predictor, an improved B-frame decision method based on a Viterbi algorithm, additional inline assembly kernels as well as numerous bugfixes and performance optimizations. A comparison of the two encoders included in PARSEC 1.0 and 2.0 with the default settings shows that the new version of the program achieves a 1% higher peak signal-to-noise (PSNR) ratio, at the expense of a 26% longer run time.

The significant updates to the code base of `x264` made it necessary to select a different set of encoding features for the benchmark, but the input files did not change:

- `test`: 32×18 pixels, 1 frame
- `simdev`: 64×36 pixels, 3 frames
- `simsmall`: 640×360 pixels ($\frac{1}{3}$ HDTV resolution), 8 frames
- `simmedium`: 640×360 pixels ($\frac{1}{3}$ HDTV resolution), 32 frames
- `simlarge`: 640×360 pixels ($\frac{1}{3}$ HDTV resolution), 128 frames
- `native`: $1,920 \times 1,080$ pixels (HDTV resolution), 512 frames

5.2 New Parallelization Models

Parallel programming paradigms are a focus of computer science research due to their importance for making the large performance potential of CMPs more accessible. The first version of the PARSEC benchmark suite supported POSIX threads (pthreads) and OpenMP. The second version also includes support for Intel Threading Building Blocks (TBB). Table 4 summarizes which workloads support which programming models. Besides the constructs of these parallelization models, atomic instructions are also directly used by a few programs if synchronized low-latency data access is necessary.

Program	Pthreads	OpenMP	TBB
`blackscholes`	X	X	X
`bodytrack`	X	X	X
`canneal`	X		
`dedup`	X		
`facesim`	X		
`ferret`	X		
`fluidanimate`	X		X
`freqmine`		X	
`raytrace`	X		
`streamcluster`	X		X
`swaptions`	X		X
`vips`	X		
`x264`	X		

Table 4: Parallelization models supported by PARSEC 2.0.

POSIX threads [12] are one of the most commonly used threading standards to program contemporary shared-memory Unix machines. Pthreads requires programmers to handle all thread creation, management and synchronization issues themselves. It was officially finalized by IEEE in 1995 in section 1003.1c of the Portable Operating System Interface for Unix (POSIX) standard in an effort to harmonize and succeed the various threading standards that industry vendors had created themselves. The parallelization model is supported by all PARSEC workloads except `freqmine`.

OpenMP [8] is a compiler-based approach to program parallelization. To parallelize a program with OpenMP the programmer must annotate the source code with the OpenMP `#pragma omp` directives. The compiler performs the actual parallelization, and all details of the thread management and the synchronization are handled by the OpenMP runtime. The first version of the OpenMP API specification was released for Fortran in 1997 by the Architecture Review Board (ARB). OpenMP 1.0 for C/C++ followed the subsequent year. The standard keeps evolving, the latest version 3.0 was released in 2008. In the original release of the PARSEC benchmark suite OpenMP was supported by `bodytrack` and `freqmine`. The second version also adds OpenMP support to `blackscholes`.

TBB is a high-level alternative to pthreads and similar threading libraries [5]. It can be used to parallelize C++ programs. The TBB library is a collection of C++ methods and templates which allow to express high-level, task-based parallelism that abstracts from details of the platform and the threading mechanism. The first version of the TBB library was released in 2006, which makes it one of the more recent parallelization models. PARSEC did not support TBB when it was originally released. The second version of the benchmark suite adds TBB support to five of its workloads: `blackscholes`, `bodytrack`, `fluidanimate`, `streamcluster` and `swaptions`.

Researchers that use the PARSEC benchmark suite for their work must be aware that the different versions of a workload that use the various parallelization methods can behave drastically different at runtime. Contreras et al. studied the TBB versions of the PARSEC workloads in more detail [4]. They conclude that the dynamic task handling approach of the TBB runtime is effective at lower core counts, where it efficiently reduces load imbalance and improves scalability. However, with increasing core counts the overhead of the random task stealing algorithm becomes the dominant bottleneck. In current TBB implementations it can contribute up to 47% of the total per-core execution time on a 32-core system. Results like these demonstrate the importance of choosing a suitable parallelization model for performance experiments.

6. FUTURE WORK

Emerging applications continue to evolve, and so should a benchmark suite of emerging applications like PARSEC. We plan to offer future versions of the suite that include new and improved workloads. Benchmarking needs to continue to develop in order to mirror real-world computer usage accurately. OS effects are typically already included in most performance experiments by the use of full system simulation, but new trends such as cloud computing or virtualization have introduced additional software layers and interactions that are not reflected in current workloads. It might become necessary to include effects such as TCP/IP communication or I/O activity to obtain accurate benchmarks.

7. CONCLUSIONS

In this paper we discussed improvements introduced with the second version of the PARSEC benchmark suite and explained major differences to the original version. The main features of PARSEC 2.0 are four significantly improved workloads, one new benchmark, support for the Intel Threading Building Blocks parallelization model and improvements to ease the use of the suite in practice. The new workload selection increases the coverage of the computational kernels typically found in emerging video games and virtual worlds. The explanations of the updates that we provide will help researchers to interpret PARSEC 2.0 results correctly.

8. ACKNOWLEDGMENTS

It is the goal of the PARSEC team to fully document every contribution to the benchmark suite. The following researchers submitted source code that was included in PARSEC 2.0: Gilberto Contreras (Princeton University), Christian Fensch (University of Cambridge), Saugata Ghose (Cornell University), Wim Heirman (University of Gent), Nikolay Kurtov (Intel Corp.) and Jiaqi Zhang (Tsinghua University). Details can be found in the `CHANGELOG` that is part of the PARSEC distribution. We would also like to thank the users who provided us with bug reports and other forms of feedback. We are very grateful for all contributions.

9. REFERENCES

[1] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *Proceedings of the 2008 International Symposium on Workload Characterization*, September 2008.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical Report TR-811-08, Princeton University, January 2008.

[4] G. Contreras and M. Martonosi. Characterizing and Improving the Performance of the Intel Threading Building Blocks Runtime System. In *International Symposium on Workload Characterization*, September 2008.

[5] Intel. Threading Building Blocks. `http://www.threadingbuildingblocks.org/`, 2008.

[6] J. T. Kajiya. The Rendering Equation. In *SIGGRAPH '86: Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, pages 143–150, New York, NY, USA, 1986. ACM.

[7] S. Kumar, J. Chhugani, C. Kim, D. Kim, A. Nguyen, P. Dubey, C. Bienia, and Y. Kim. Second Life and the New Generation of Virtual Worlds. *Computer*, 41(9):46–53, 2008.

[8] OpenMP Architecture Review Board. OpenMP Application Program Interface. `http://www.openmp.org/`, 2008.

[9] Pin. `http://rogue.colorado.edu/pin/`.

[10] S. Roth. Ray Casting for Modeling Solids. *Computer Graphics and Image Processing*, 18(2):109–144, 2 1982.

[11] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[12] The Open Group and IEEE. IEEE Std 1003.1, 2004.

[13] T. Whitted. An Improved Illumination Model for Shaded Display. *Commun. ACM*, 23(6):343–349, 1980.