PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite

DIMITRIOS CHASAPIS, MARC CASAS, MIQUEL MORETÓ, RAUL VIDAL, EDUARD AYGUADÉ, JESÚS LABARTA, and MATEO VALERO, Barcelona Supercomputing Center and Universitat Politecnica de Catalunya—BarcelonaTech, Barcelona, Spain

In this work, we show how parallel applications can be implemented efficiently using task parallelism. We also evaluate the benefits of such parallel paradigm with respect to other approaches. We use the PARSEC benchmark suite as our test bed, which includes applications representative of a wide range of domains from HPC to desktop and server applications. We adopt different parallelization techniques, tailored to the needs of each application, to fully exploit the task-based model. Our evaluation shows that task parallelism achieves better performance than thread-based parallelization models, such as Pthreads. Our experimental results show that we can obtain scalability improvements up to 42% on a 16-core system and code size reductions up to 81%. Such reductions are achieved by removing from the source code application specific schedulers or thread pooling systems and transferring these responsibilities to the runtime system software.

Categories and Subject Descriptors: D.1.3 [**Software**]: Concurrent Programming

General Terms: Performance, Measurement, Experimentation

Additional Key Words and Phrases: Parallel Applications, scalable applications, parallel benchmarks, parallel architectures, parallel runtime systems, task-based programming models, concurrency, synchronization

ACM Reference Format:
Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. PARSECSs: Evaluating the impact of task parallelism in the PARSEC benchmark suite. ACM Trans. Archit. Code Optim. 12, 4, Article 41 (December 2015), 22 pages.
DOI: http://dx.doi.org/10.1145/2829952

1. INTRODUCTION

In the past few years, processor clock frequencies have stagnated, whereas exploiting instruction-level parallelism (ILP) has already reached the point of diminishing returns. Multicore designs arose as a solution to overcome some of the technological constraints that uniprocessor chips have, but they exacerbated some others as a

This work has been partially supported by the European Research Council under the European Union 7th FP, ERC grant agreement 321253; the Spanish Ministry of Science and Innovation under grants TIN2012-34557 and TIN2015-65316-P; the Severo Ochoa Program, awarded by the Spanish government, under grant SEV-2011-00067; and the HiPEAC Network of Excellence. M. Moreto has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva post-doctoral fellowship JCI-2012-15047, and M. Casas is supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Co-fund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (contract 2013 BP B 00243).
Authors' addresses: D. Chasapis, M. Casas, M. Moreto, and R. Vidal, Nexus II - Planta 2, C/ JORDI GIRONA, 29, Barcelona, 08034; emails: {dimitrios.chasapis, marc.casas, miquel.moreto, raul.vidal}@bsc.es; E. Ayguade and J. Labarta, K2M - Planta 3, C/ JORDI GIRONA, 1-3, BARCELONA, 08034; emails: {eduard.ayguade, jesus.labarta}@bsc.es; M. Valero, Nexus II - Planta 1, C/ JORDI GIRONA, 29, BARCELONA, 08034; email: mateo.valero@bsc.es.

DOI: http://dx.doi.org/10.1145/2829952

counterpart.Multicore architectures can potentially provide the desired performance by exploiting thread-level parallelism (TLP) of large-scale parallel workloads on chip. Such a large amount of parallelism is managed by the software, which means that the programmer needs to implement highly efficient and architecture-aware parallel codes to achieve the expected performance. This is obviously much harder than programming a uniprocessor chip, which is commonly referred as the programmability wall [Chapman 2007]. Moreover, dealing with this wall will be even harder in the near future with the arrival of many-core systems with tens or hundreds of heterogeneous cores and accelerators on chip.

Threading is the most common way to program many-core processors. POSIX threads (Pthreads) [Butenhof 1997] and OpenMP [Chapman et al. 2007] are two of the most common programming models to implement threading schemes. Additionally, MPI [Nagle 2005] can be incorporated to threading codes to handle parallelism in a distributed memory environment. However, to develop efficient threading codes can be a really hard job due to the increasing amount of concurrency handled by many-core processors and the current trend toward more heterogeneity within the chip. Synchronization points are often needed in threading codes to control the dataflow and to enforce correctness. However, the cost of these schemes increases with the amount of parallelism handled on chip, seriously hurting performance due to issues like load imbalance or NUMA effects. In addition, relaxing synchronization costs often involves significant programming efforts, as it requires the deployment of complex and application-specific mechanisms like thread pools.

Task parallelism [Fatahalian et al. 2006; Blumofe et al. 1995; Bellens et al. 2006; Ayguadé et al. 2009; Tzenakis et al. 2012; Jenista et al. 2011; Planas et al. 2009; Duran et al. 2011] is an alternative parallel paradigm where the load is organized into tasks that can be asynchronously executed. Additionally, some task-based programming models allow the programmer to specify data or control dependencies between the different tasks, which allows synchronization points relaxation by explicitly specifying the data involved in the operation [Jenista et al. 2011; Ayguadé et al. 2009; Tzenakis et al. 2012; Duran et al. 2011].

The task-based execution model requires tracking the dependencies among tasks, which can be explicitly specified by the programmer [Jenista et al. 2011; Zuckerman et al. 2011] or dynamically handled by an underlying runtime system [Duran et al. 2009; Tzenakis et al. 2012; Duran et al. 2011]. When dependencies are detected among tasks, a deterministic execution order is applied by the runtime system to enforce correctness. In this way, all potential parallelism of the code is exposed to the runtime system, which can exploit it depending on the available hardware. Additional optimizations such as load balancing or work stealing [Blumofe et al. 1995; Duran et al. 2011] can be applied at the runtime system layer without requiring any platform-specific consideration from the programmer.

The potential of task-based programming models is expected to be significant in a wide range of areas. The evaluations made so far consider microbenchmarks [Appeltauer et al. 2009; Podobas and Brorsson 2010] or are limited to specific application domains like graph analysis codes [Adcock et al. 2013] or high performance computing (HPC) kernels [Ayguadé et al. 2008; Podobas and Brorsson 2010]. In this work, we aim to evaluate the benefits of task-based parallelism beyond the scope of HPC applications, focusing on a set of parallel applications representative of a wide range of domains from HPC to desktop and server applications. To do so, we apply task-parallelism strategies to the PARSEC benchmark suite [Bienia 2011] and compare them in terms of programmability and performance with respect to the fork-join versions contained in the suite. The main contributions of this article are the following:

—We apply task-based parallelization strategies to 10 PARSEC applications.
—We fully evaluate them in terms of performance, considering different scenarios (from 1 to 16 cores) and achieving average improvements of 13%. In some particular cases, the improvements reach 42%.
—We provide detailed programmability metrics based in lines of code (LOCs), achieving an average reduction of 28% and reaching a maximum of 81%.

The remaining of this document is organized as follows. Section 2 offers background information on the task-based model we chose and the PARSEC benchmark suite. In Section 3, we discuss how these applications are parallelized in Pthreads/OpenMP and then explain our task-based approach. Section 4 shares our experience in programming these applications: the required effort, the versatility of the task-based model, and its current limitations. Section 5 evaluates the performance of the Pthreads/OpenMP codes and our task-based implementations. Section 6 summarizes the related work, and in Section 7, we give our closing remarks.

2. BACKGROUND

2.1. The PARSEC Benchmark Suite

With the prevalence of many-core processors and the increasing relevance of application domains that do not belong to the traditional HPC field comes the need for programs representative of current and future parallel workloads. The PARSEC [Bienia 2011] features state-of-the art, computationally intensive algorithms and very diverse workloads from different areas of computing. PARSEC is composed of 13 benchmark programs. The original suite makes use of the Pthreads parallelization model for all of these benchmarks, except for `freqmine`, which is only available in OpenMP. The suite includes input sets for native machine execution, which are real input sets. Table I describes the different benchmarks included in the suite along with their respective native input and the LOCs of each application. We apply tasking parallelization strategies to 11 out of its 13 applications: `blackscholes`, `bodytrack`, `canneal`, `dedup`, `facesim`, `ferret`, `fluidanimate`, `freqmine`, `streamcluster`, `swaptions`, and `x264`. We leave 2 applications out of this study: `raytrace` and `vips`. `Vips` is a domain-specific runtime system for image manipulation. Since vips is a runtime itself, it is not reasonable to implement it on top of another runtime system. Therefore, we do not include this code in our evaluations. `Raytrace` code has the same extension as `ferret`, `facesim`, and `bodytrack`, and the same parallel model as `blackscholes` [Cook et al. 2013]. Therefore, since it does not offer any new insight, we do not consider the `Raytrace` code in the article.

We have a preliminary task-based implementation of the `x264` encoder, which scales up to 14x on a 16-core machine, the same as the Pthreads version. Since we just emulate the same parallel model as the original Pthreads version and obtain the same performance, we do not include this code in the rest of this work, as it provides no insight.

2.2. Asynchronous Tasks and Dataflow Model

Tasks offer an easy and abstract way to express parallelism. OpenMP 4.0 [OpenMP Architecture Review Board 2013], a widely used programming standard for shared memory machines, allows the user to annotate functions that can be run asynchronously. It also supports dataflow annotations that describe data dependencies among tasks. This information can be used by the runtime system to synchronize task execution. Standard synchronization schemes are also available (locks, atomics, barriers, etc). In this work, we chose to use the OmpSs [Duran et al. 2011] programming model, which

Table I. PARSEC Benchmark Suite

Benchmark	Description	Native Input	LOC
blackscholes	Intel RMS benchmark that calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation	10,000,000 options	404
bodytrack	Computer vision application that tracks a 3D pose of a markerless human body with multiple cameras through an image sequence	4 cameras, 261 frames, 4,000 particles, 5 annealing layers	6,968
canneal	Simulated cache-aware annealing to optimize routing cost of a chip design	2,500,000 elements, 6,000 temperature steps	3,040
dedup	Compresses a data stream with a combination of global compression and local compression to achieve high compression ratios	672MB data	3,401
facesim	Intel RMS workload that takes a model of a human face and a time sequence of muscle activation and computes a visually realistic animation of the modeled face	100 frames, 372,126 tetrahedra	34,134
ferret	Content-based similarity search of feature-rich data such as audio, images, video, and 3D shapes	3,500 queries, 59,695 images database, finds top 50 images	10,552
fluidanimate	Intel RMS application that uses an extension of the smoothed particle hydrodynamics method to simulate an incompressible fluid for interactive animation purposes	500 frames, 500,000 particles	2,348
freqmine	Intel RMS application that employs an array-based version of the frequent pattern growth method for frequent itemset mining	250,000 HTML documents, minimum support 11,000	2,231
raytrace	Intel RMS workload that renders an animated 3D scene	200 frames, 1,920×1,080 pixels, 10 million polygons	13,751
streamcluster	Solves the online clustering problem	200,000 points per block, 5 block	1,769
swaptions	Intel RMS workload that uses the Heath-Jarrow-Morton framework to price a portfolio of swaptions	128 swaptions, 1,000,000 simulations	1,225
vips	VASARI Image Processing System (VIPS), which includes fundamental image processing operations.	18,000×18,000 pixels	127,957
x264	H.264/AVC (Advanced Video Coding) video encoder.	512 frames, 1,920×1,080 pixels	29,329

is a forerunner of the OpenMP 4.0 tasks. Despite the fact that OmpSs offers advanced features like socket-aware scheduling for NUMA architectures or pragma annotations to handle multiple dependence scenarios, in this article we only use features already available in the OpenMP 4.0 standard. The two models have virtually the same syntax, thus porting OpenMP code to OmpSs and vice versa is straightforward.

Figure 1 shows a simplified version of the `ferret` benchmark implemented in OmpSs, an application that is parallelized with a pipeline model. The programmer can use pragma directives to identify functions that should run asynchronously. These task pragmas can have dataflow relations expressed with the use of `in`, `out`, and `inout` annotations. These declare whether a variable is going to be read, written, or both by the task. An underlying runtime system is responsible for scheduling tasks, tracking dependencies, balancing the load among available threads, and ensuring correct order of execution, as dictated by the dataflow relations. In our example, data dependencies will force tasks spawned in the same iteration to run in sequential order, whereas tasks from different iterations can run concurrently. An exception is `t_out`, which shares a common output between all instances, `outstream`, to store the final results of `ferret`.

```
void load() {
  int i = 0;
  while( load_image(image[i]) ) {
    #pragma omp task in(image[i])
                     out(seg_images[i])
    seg_images[i] = t_seg(image[i]);
    #pragma omp task in(seg_images[i])
                     out(extract_data[i])
    extract_data[i] = t_extract(seg_images[i]);
    #pragma omp task in(extract_data[i])
                     out(vectoriz_data[i])
    vectoriz_data[i] = t_vec(extract_data[i]);
    #pragma omp task in(vectoriz_data[i])
                     out(rank_results[i])
    rank_results[i] = t_rank(vectoriz_data[i]);
    #pragma omp task in(rank_data[i])
                     out(outstream)
    t_out(rank_data[i], outstream);
    i++;
  }
  #pragma omp taskwait
}%
```

Fig. 1. `Ferret` implementation in OmpSs.

3. APPLICATION PARALLELIZATION

In this section, we discuss how the PARSEC applications are parallelized in Pthreads/OpenMP2.0 and how they can be implemented efficiently using a task-based approach. When possible, we exploit dataflow relations to take advantage of implicit synchronization (as described in Section 2.2). If it is not possible, we use conventional synchronization primitives such as locks, atomics, and barriers.

Blackscholes. This application solves a Black-Scholes partial differential equation [Black and Scholes 1973] to calculate the prices for a portfolio of 10 million European options.

Pthreads. This version simply divides the portfolio into work units by the number of available threads and stores them into the `numOptions` array. Each thread calculates the prices for its corresponding options and waits in a barrier until all threads have finished executing. The algorithm is run multiple times to obtain the final estimation of the portfolio.

Task based. In the case of the task-based version, we divide the work into units of a predefined block size. This block size allows having much more task instances than threads, which implies a much better load balance, as this is an embarrassingly parallel application with no dependencies among tasks in the same run.

Bodytrack. Computer vision application that tracks a markerless human body using multiple cameras through an image sequence. The application employs an annealed particle filter to track the body using edges and the foreground silhouette as features of interest.

Pthreads. `Bodytrack` applies the same algorithm on each frame of the image sequence to track the different poses of the body. The human body is modeled as a tree-basedstructure, consisting of seven conic cylinders. It reads four images taken from several cameras to capture a scene from four different angles, and thus each frame consists of these four images. These images are read and encoded to a single data structure. For each frame, `bodytrack` extracts the edges and silhouette features

for each of these four images. In this feature extraction stage, we have three different kernels:

(1) *Edge detection*: Gradient based edge detection
(2) *Edge smoothing (phase 1)*: Gaussian filter used to smooth edges applied on array rows
(3) *Edge smoothing (phase 2)*: Gaussian filter used to smooth edges applied on array columns.

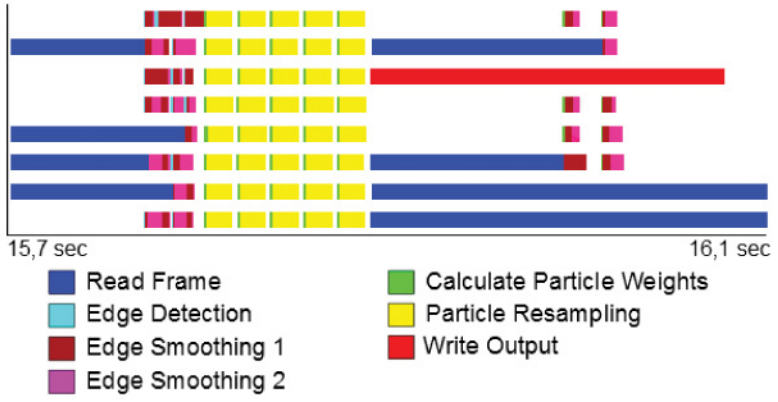
Afterward, `bodytrack` goes through an annealed particle filter stage, which consists of M annealing layers over a set of N particles. The particles are multivariate configurations of the state and location of the tracked body. Given the image features, the particles are assigned weights, which increase or decrease the chance that a particle represents a body part. N particles are then chosen, depending on the probability dictated by their weights. Random noise is added to this set of particles, creating a new set. This process is repeated for all annealing layers. `Bodytrack` then picks one of the M configurations—the one which has the highest weighted average. This process has two parallel kernels:

(4) *Calculate particle weights*: Computes weights for the particles, using the edges and silhouette produced from the previous stages
(5) *Particle resampling*: Adds Gaussian random noise to the particles, thus creating a new set of particles.

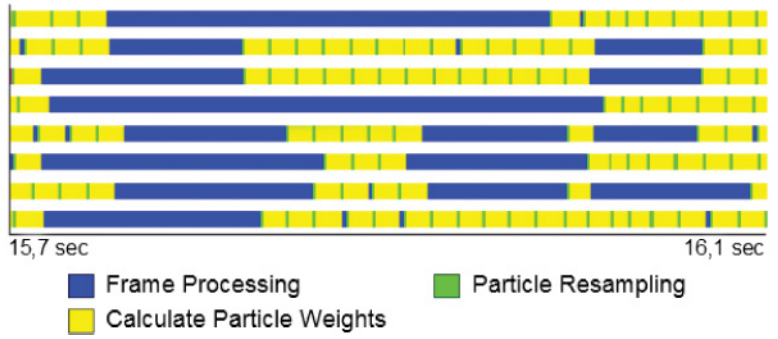
In the case of Pthreads, the four images of a frame are read and processed in parallel using one thread per image. The Pthreads implantation is limited by the four images that it can process concurrently, as there is no other candidate work at this point. A specific asynchronous I/O implementation is required to read the files in parallel. Then, the features extraction stage is executed using all available threads, with a synchronization barrier at the end of each phase. The same structure is followed in the annealed particle filter stage, with two barriers at the end of each phase. Between the two stages, serial code has to be executed, which leaves only one thread busy and the rest idle. Finally, the output results are written sequentially in one file.

Task based. In the case of the task-based version, we adopt a more coarse grain approach. We do not parallelize the feature extraction stage; instead, we taskify the whole frame processing, allowing concurrent execution of all frames. The parallel kernels of the annealed particle filter stage are taskified in our version, and synchronization is achieved by dataflow annotations. Each frame needs to be written when calculations are completed. In our version, we can do this asynchronously while the threads are busy with the processing stage of another frame. Thus, output I/O is effectively overlapped with computation stages.

Figure 2 shows parallel executions of two different task-based implementations: the first one just mimics the Pthreads behavior (2(a)), and the second is an optimal task-based implementation (2(b)). Different colored boxes represent different task types, as well the duration of that task type on each core. In both cases, the white gaps denote the time each thread spends idle. Both figures show the same duration for each execution. In the optimal version, all functionality is implemented within the frame-processing task, thus execution time for read-frame, edge-detection, and edge-smoothing is represented with blue color (frame-processing). Tasks particle-resampling and calculate-particle-weights are also implemented as nested tasks. They are displayed with different colors (green and yellow, respectively). We can observe that the Pthreads-like version suffers from greater idle time compared to its optimal task-based counterpart. Work is distributed more efficiently in the optimal implementation by processing



(a) Trivial task-based strategy



(b) Optimal task-based strategy

Fig. 2. Parallel execution of Pthreads and task-based versions of `bodytrack` on an eight-core machine and native input size. Different parallel regions correspond to different colors. White gaps in the figure represent idle time.

different frames concurrently. This allows us to overlap I/O and serial code segments of one with available work from another one.

Canneal. This kernel uses a cache-aware simulated annealing [Banerjee 1994] to optimize routing cost of a chip design. `Canneal` progressively swaps elements that need to be placed in a large space, eventually converging to an optimal solution. The problem is stored as a list with routing costs between nodes.

Pthreads. This version compares random element pairs of the graph concurrently and swaps them until it converges to an optimal solution. No locks are used to protect the list from concurrent accesses/writes, but swaps are done atomically instead. However, the evaluation of the elements to be swapped is not atomic. This means that disadvantageous swaps may occur, which will require the algorithm to eventually swap them again. This method has provided better results than the alternative algorithm with locks [Bienia et al. 2008].

Task based. Our task-based version follows the same paradigm. Several tasks are spawned without any dependencies between themselves. We use the same atomics as with the Pthreads version. Since tasks work with an arbitrary number of list elements, it is not possible to describe which elements of the list a task is going to randomly access.

We also try an alternative fine-grain implementation, where a task is spawned for each random pair of list elements. This would allow the runtime to know if two tasks are working on the same list of elements. However, this implementation implied fine-grain tasks. Each task would merely do a single swap between two list elements. The overhead of the dynamic scheduling is a problem in this scenario. A more complex but more efficient solution is suggested by Symeonidou et al. [2013] with the use of memory regions. Adopting this method in a task-based model would allow the programmer to describe parts of the list (or other pointer based data structures) and express dataflow relations as abstract memory regions. This solution also implies fine-grain tasking and is not evaluated in the work of Symeonidou et al.

Dedup. The `dedup` kernel is used to compress data streams using local and global compression to achieve higher compression rates. This method is called *deduplication* [Quinlan and Dorward 2002].

Pthreads: Dedup is parallelized using a pipeline model with the following stages:

—*Fragment*: First, the data stream is read and partitioned at fixed positions into coarse-grain data chunks. Each chunk can be processed individually by the rest of the stages. This stage is executed on a single thread.
—*Fragment Refine*: A new data chunk initiates the second pipeline stage, where it is further partitioned into smaller fine-grain chunks. The portioning is done by using the rolling-fingerprint algorithm.
—*Deduplication*: This stage eliminates duplicate fine-grain chunks. Unique chunks are stored in a hash table. Locks are used here to protect each bucket from concurrent accesses.
—*Compress*: At this stage, chunks are compressed in parallel. Identical chunks are compressed only once as duplicates are removed at the deduplication stage.
—*Reorder*: This stage writes the final compressed output data to a file. It writes only unique chunks' compressed data; for the duplicates, it stores their hash values. However, this stage needs to reorder the data chunks as they are received to match the original order of the uncompressed data.

The Pthreads version maintains a queue and a thread pool dedicated to each stage. When a chunk becomes available at one stage, it is moved to the queue of the next stage. Each stage polls at its queue for available chunks to process. The reorder stage is done sequentially with a devoted thread that can be in an idle loop waiting for previous stages to finish. Each thread pool is composed of a number of threads equal to the number of available cores. The only exceptions are `Fragment` and `Reorder` stages, which are served by a single thread each.

Task based. In our implementation, we taskify each pipeline stage and express data dependencies using static arrays and dataflow relations, one for each pipeline stage. `FragmentRefine`, however, partitions the data chunks into very fine grain segments, ranging from a few hundreds to thousands. For such granularity, our approach suffers from high overheads due to dynamic scheduling overhead. The same is observed in Vandierendonck et al. [2013], where an alternative approach is adopted. In their approach, two pipelines are identified. The outer pipeline consists of stages `Fragment`, `InnerPipeline`, and `Reorder`. The inner pipeline consists of `FragmentRefine`, `Deduplicate`, and `Compress`. To reduce the dynamic scheduling overhead, they merge together `Deduplicate` and `Compress`. By doing so, the available parallelism is limited, but still there is enough work not to harm performance and scalability. In our approach, we merge together the inner pipeline, creating one sequential function, exploiting only the parallelism available in the outer pipeline. Even in this

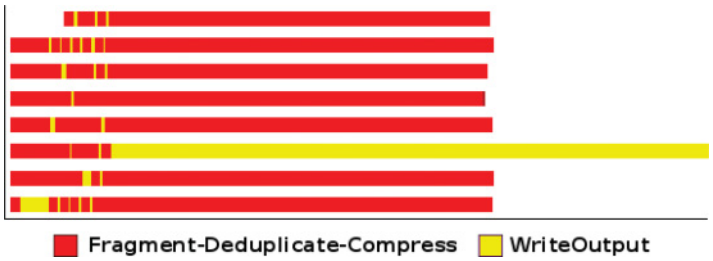


Fig. 3. Parallel execution of the task-based version of dedup on an eight-core machine and native input size. Different task types correspond to different colors.

scenario, the available parallelism is still abundant, as the application is bound by the writing of the output file, which is sequential. Figure 3 shows a trace of the task-based version. We can see that the communication stage (in yellow) is effectively overlapped with the computation stage (in red); however, there is not enough work to keep all of the threads busy, until the end of the execution.

Furthermore, we modify the `Reorder` stage by replacing it with a simple stage where the chunk is simply written to file (`WriteOutput`). Using dataflow relations and a shared output resource between the `WriteOutput` tasks, we ensure that chunk `N-1` will be written before chunk `N`. Thus, we do not need to reorder data chunks in this task type. Moreover, the scheduler makes sure that chunks are written as soon as they become available by the `InnerPipeline` task, an improvement over the Pthreads version, where `Reorder` instances need to wait until all previous chunks have been written. Another difference between the two versions is that Pthreads oversubscribes threads to cores for each pipeline stage, whereas in our implementation, we only assign one thread to each core.

Facesim. Facesim computes a visually realistic human face animation by simulating the underlying physics. As input, it uses a 3D model of a human face containing both a tetrahedra mesh and triangulated surfaces for the flesh and bones, respectively. Additionally, it uses a time sequence of muscle movement [Sifakis et al. 2005].

Pthreads. The application statically decomposes the original tetrahedron mesh into smaller partitions, equal to the number of available threads. There are three main parallel kernels:

—*Update State*: Calculates the steady properties of the mesh, constrains such as stress and stiffness
—*Add Forces*: Computes the force contribution between vertices acting on the 3D model
—*Conjugate Gradient (CG)*: An iterative method that solves the linear system produced by the other two previous kernels and finds the final displacement of the vertices for the current frame.

`Update_State` and `Add_Forces` kernels consist of one and two parallel loops, respectively, whereas `CG` has three. Synchronization between loops and kernels is achieved by barriers. The corresponding force computations from the skeleton are also done in `Update_State` and `Add_Forces`, but after the parallel computations on the tetrahedra mesh have been made. In Pthreads, a master thread is assigning work to all threads in a round-robin fashion through a queuing system. Each thread maintains its private queue, which is protected by locks.

Task based. In the task-based version, the application-level queuing system is completely replaced by the OmpSs runtime. In our initial implementation, all parallel loops

are taskified. Additionally, in `Update_State`, there is a sequential code segment: `Update_Collision_Penalty_Forces`. This code segment operates on the bones while the parallel loop of `Update State` does so on the tetrahedra. By taskifying it and adding dataflow relations between this section and the following `Add_Forces` kernel, we can overlap `Update_Collision_Penalty_Forces` with the rest of `Update_State`.

To improve performance, we refactor tasks' creation in `CG` by nesting the first task creation loop inside another task. This enables us to overlap task creation time with computation, which contributes to increase `Facesim's` task-based implementation performance. Although we achieved better scalability than the original code, task creation still imposed overheads. To address this issue, we replace tasks in `CG` with the OmpSs parallel loops construct (equivalent to the OpenMP one), which implements loop worksharing with a task. Even though this approach limits the available parallelism (barrier synchronization, no dataflow annotations), the overhead associated with task creation and scheduling is greatly reduced and overall performance is improved.

Ferret. Content similarity search server for feature-rich data [Lv et al. 2006] such as audio, video, and images. The benchmark application is configured for image similarity search.

Pthreads. Ferret is parallelized using a pipeline model. A serial query is broken down into six pipeline stages:

—*Load*: This stage loads an image that is going to be used as a query.
—*Segmentation*: At this stage, the image is decomposed into the different objects displayed on it. Different weight vectors are to be assigned on each object to achieve better results.
—*Extract*: At this stage, a 14-dimensional vector is computed for each object from the segmentation stage, describing features such as color, area, and state.
—*Vectorization*: This is the indexing stage that tries to find a set of candidate images in the database.
—*Rank*: This stage ranks the results found, using the EMD metric for each query object's vector and the database's image vectors.
—*Output*: This outputs the result of the ranking stage. Multiple instances of this stage need to run serially, as they all share the same output stream.

In the Pthreads version, every stage is served by a dedicated thread pool of N threads each, where N is the number of available cores. The only exceptions are the `Load` and `Output` stages that are executed by a respective single thread. Each stage polls on its corresponding queue for available work. When a stage finishes, it pushes the results to the next stage's queue.

Task based. In this version, we implement a variation of this pipeline model. As soon as the first stage, `Load`, finds a new image, it spawns all stages of a pipeline for that image, thus reducing the pipeline to five stages. We model the dataflow relations between different stages as simple one dimension arrays, as shown in Figure 1. Tasks working on different image queries do not share any dependencies. An exception is task `t_out`, which shares the same output file between all pipelines, and thus sequential execution is forced between all instances of this task. The pipeline stages and dependencies are constructed a priori, which is good enough for this application, but this is not always the case. Lee et al. [2013] proposes a system that can handle dynamic pipeline creation by constructing a DAG with the stages using indexes and the `cilk_continue` and `cilk_wait` keywords. Indexes are used to define the different pipeline stages, whereas `cilk_continue` creates a stage that can run once all previous stages in the same pipeline iteration are done, and `cilk_wait` creates a stage that will

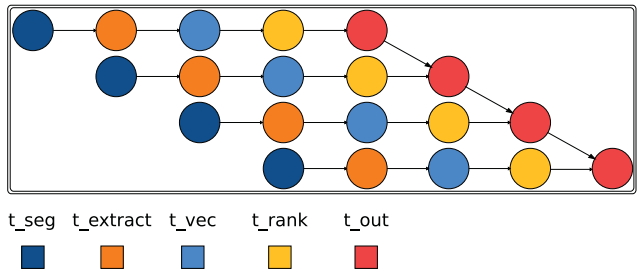


Fig. 4. Task graph of ferret showing the pipelined execution model. Edges show data dependencies among different tasks.

wait for its stage counterpart of the previous iteration to finish. A strategy based on versioning the dependency objects between the stages has been proposed [Vandierendonck et al. 2011]. Output dependencies are renamed and privatized, and thus the static array for privatization is not required.

Figure 4 shows the task graph of the `ferret` application. Colored nodes denote the concurrent tasks (each color matches a specific task type). Tasks that have data dependencies are connected by directed edges. By inspecting the task graph, we can see a pipeline pattern of execution. Despite the fact that the task-based approach does not significantly improve the overall performance, as we can see in Section 5, it significantly reduces the effort required to express the pipeline parallelism, compared to its Pthreads counterpart, as is shown in detail in Section 4.1.

Fluidanimate. This application simulates incompressible fluid interactive animation using the smoothed particle hydrodynamics method [Müller et al. 2003].

Pthreads. `Fluidanimate` uses five special kernels that are responsible for rebuilding the spatial index, computing fluid densities and forces at given points, handling fluid collisions with the scene geometry, and finally updating particle locations The fluid surface is partitioned, and each thread works on its own grid segment. The kernels are parallelized as do-all loops, separated by barriers. Moreover, there are cases where these threads need to update values beyond their partition, which are handled using locks.

Task based. The task-based implementation follows the same approach: we apply a loop tiling transformation for each parallel loop and taskify each iteration. We maintain the same barrier and lock synchronization scheme using the OmpSs synchronization primitives.

Freqmine. This is a data mining application that makes use of an array-based version of the frequent pattern (FP) growth method for frequent itemset mining [Grahne and Zhu 2003].

Pthreads. The application uses a compact tree data structure, denoted *FP-tree* [Han et al. 2000], to store information about frequent patterns of the transaction database. The FP-tree is coupled with a header table, which is a list of database items sorted by decreasing order of occurrences. The FP-growth algorithm traverses the FP-tree structure recursively, constructing new FP-trees until the complete set of frequent itemsets is generated. There are three parallel kernels. The `Build_FP-tree_header_table` kernel performs a database scan and counts the number of occurrences of each item. The result is the FP-tree header table. `Build_Prefix_tree` kernel performs a second database scan required to build the prefix tree, and the `Data_Mining` kernel obtains the frequent itemset information by using the previous two structures. It creates an

additional lookup table, which allows faster traversals on sparse itemsets. The original PARSEC benchmark uses OpenMP2.0 for loop parallelization inside each kernel.

Task based. In our implementation, we taskify each iteration. We do not use any dataflow relations in this application, and we resolve to adopt the locking and barrier synchronization used in the original OpenMP version.

Streamcluster. Streamcluster is a kernel that solves the online clustering problem. It takes a stream of points and then groups them in a predetermined number of clusters with their respective centers.

Pthreads. Up to 90% of total execution time is spent in function `pgain`, computing whether opening a new center is advantageous or not. For every new point, function `pgain` calculates the cost of making it a new center by reassigning some points to it and comparing it to the minimum distance $d(x, y) = |x - y|^2$ between all points x and y. The result is accepted if found to favor the new center. Data points are statically partitioned by a given block size, which determines the level of parallelism in the application. In the Pthreads version, this is equal to the number of threads.

Task based. In our implementation, we follow a different decomposition strategy, making the number of tasks independent of the number of partitions. Barriers are employed to synchronize accesses to a partition in both Pthreads and the task-based implementation. In the case of Pthreads, an additional user-implemented library is used for the barriers. This library is not required in the case of the OmpSs implementation, as the runtime already has a generic barrier implementation.

Swaptions. This is an economics application that uses the Heath-Jarrow-Morton [Heath et al. 1992] for pricing of a portfolio of swaptions. To calculate prices, it employs the Monte Carlo simulation.

Pthreads The application stores the portfolio into an array. In the Pthreads version, this array is divided by the number of available threads, with each thread working on its own part of the array.

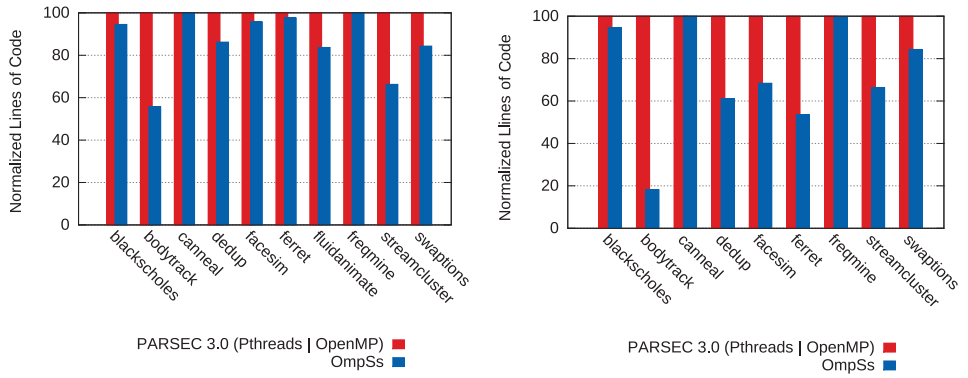
Task based. We use the exact same strategy, where each task works on a part of the array. No data dependencies exist between the tasks.

4. PROGRAMMABILITY

Different models and languages offer diverse ways to express concepts, such as parallelism or asynchrony. In this section, we evaluate how successful and easy it is to express parallelism using task-based models. A good proxy to evaluate the complexity of a particular implementation is the number of LOCs it takes. Despite being a metric proposed some decades ago, comparing different programming models in terms of the total number of code lines is still a valid metric. Indeed, recent publications make extensive use of it [Vandierendonck et al. 2011; Dongarra et al. 2008].

4.1. Lines of Code

The reduction of the LOCs attests to a more compact and readable code. In some of our PARSEC task-based implementations, a simple pragma directive replaced application-specific schedulers, scheduling queues, thread pooling mechanisms, and lock synchronization. We do not change the algorithm in any of the task-based parallel strategy implemented in the PARSEC suite. Figure 5(a) shows a normalized comparison between the LOCs of our task-based implementations and the original Pthreads/OpenMP implementations of the PARSEC 3.0 distribution. The PARSEC 3.0 versions to which we refer are always the Pthreads versions, except in the case of `freqmine`, where since



(a) Comparison between all source files.

(b) Comparison between only source files containing parallel code.

Fig. 5. Comparison of LOCs between our task-based implementations and the original Pthreads or OpenMP versions.

there is no Pthread version available, the OpenMP 2.0 version is taken as reference. We preprocess all source files so that they only contain LOCs relevant to the respective programming model.[1] Figure 5(b) shows the total LOCs comparison when we only consider files that are relevant to the parallel implementation—that is, files that contain calls to Pthreads or task invocations, asynchronous I/O implementations, atomic primitives, and so forth. In this graph, we see that the reductions in terms of LOCs of our task-based strategies are significant. In the case of `bodytrack`, we are able to remove 81% of the code lines. Since `Bodytrack` implements its own scheduler to deal with load balancing, there is much room for code reductions by replacing this ad hoc mechanism for a few pragma annotations.

By using tasks and dataflow relations, it is very easy to implement pipelines. We adopt this approach for both `dedup` and `ferret`, which result in a significant decrease in LOC (38% and 46%, respectively). Figure 1 shows the pipeline code for `ferret`. All that is required is to taskify the different pipeline stages and make sure that dataflow relations force in-order execution of tasks in the same pipeline instance. The Pthreads version requires the implementation of queues between each stage, which must also be safe to use by multiple threads and concurrent accesses. `Streamcluster` and `fluidanimate` task-based versions also reduce LOCs by 33% and 21%, respectively, by removing the need for an additional, user-implemented, barrier library. `Blackscholes` and `swaptions` are relatively simple applications, containing only one do-all parallel loop each. In these cases, the LOC difference is minimal (0.5% and 15%, respectively). In the cases of `canneal` and `freqmine`, we see no difference in LOCs. `Canneal` is not a data parallel application, and in both cases Pthreads and tasks are used merely as thread launching mechanism, whereas the synchronization effort is essentially the same.

It is worth noting that conventional synchronization primitives can still be used with tasks but without penalizing the programmer. `Freqmine` is implemented in OpenMP, which excels at parallelizing loops with very little effort from the programmer and is the ideal programming model for this application. In our implementation, we simply taskify the loops, essentially not affecting LOCs. `Facesim` also benefits from the

[1]PARSEC benchmarks contain mixed serial, Pthreads, OpenMP, and TBB source code, and make use of macros to enable conditional compilation for only one programming model at a time.

tasks-based approach by 37%, as the queues required to schedule work have been completely removed. Overall, we see that the task-based model reduces code size by 28% on average.

5. PERFORMANCE EVALUATION

In this section, we compare our task-based implementations to the original PARSEC implementations in Pthreads or OpenMP.

5.1. Experimental Setup

The experiments are performed on an IBM System X server iDataPlex dx360 M4, composed of two eight-core Intel Sandy Bridge processors (E5-2650), with 2.60Hz and 20MB of shared last-level cache. There are eight 4GB DDR3 DIMMs running at 1.6GHz (a total of 32GB per node and 2GB per core). The hard drive is an IBM 500GB 7.2K 6Gbps NL SATA 3.5. We make use of the OmpSs programming model [Duran et al. 2011] and its associated toolflow: nanos++ runtime system (version 0.8a), Mercurium source-to-source compiler (version 1.99), and gcc 4.7 as the back-end compiler. We run all benchmarks using their respective native inputs as described in Table I.

5.2. Scalability

Figure 6 shows how the task-based codes scale compared to the PARSEC Pthread/OpenMP versions. Results are shown individually per benchmark as we increase the number of cores assigned to the application and are normalized to the execution time of the serial implementation[2] of the application. Nearly all applications scale linearly up to four cores.

In the case of `bodytrack`, as described in Section 3, by concurrently executing different frames, there is always enough work for all threads; by taskifying the output stage of each frame, we overlap this I/O bottleneck with other computation stages. The speedup when run on 16 cores is 12.1x, whereas the Pthreads implementation reaches a poor 6.8x speedup when run on 16 cores. The `dedup` application has a very expensive stage that writes the compressed data to the output file. Our task-based implementation is very effective in overlapping this time with computation from the compression stage. In addition, the task-based version does not have to reorder the data chunks, as the I/O execution takes place in order as dictated by dataflow relations. This results in an impressive 30% performance improvement of the OmpSs version with respect to Pthreads when run on 16 cores. The Pthreads `facesim` implementation is burdened by barriers that limit available parallelism. By using dataflow relations, we taskify sequential segments of significant cost, and we effectively synchronize them with parallel sections preceding and following it. The performance improvements comes from the overlap of sequential computations with parallel sections. The task-based parallelization of `facesim` reaches a speedup of 10.2x when run on 16 cores, whereas the PARSEC code only reaches a 6.4x speedup.

In the cases of `blackscholes`, `canneal`, `ferret`, `fluidanimate`, `freqmine`, and `swaptions`, the Pthreads/OpenMP versions already achieve good scalability results. With the exception of `ferret`, the task-based codes have very close resemblance to their Pthreads/OpenMP counterparts and have offered reduced opportunities for OmpSs to dynamically exploit additional parallelism. The parallel implementation in these applications, with the exception of `ferret`, is limited to parallel do-all loops with

[2]The PARSEC benchmark suite provides a serial implementation for `blackscholes`, `bodytrack`, `dedup`, `ferret`, `freqmine`, and `swaptions`. For the other benchmarks, the original Pthreads parallel implementation executed on a single core is considered as the baseline.

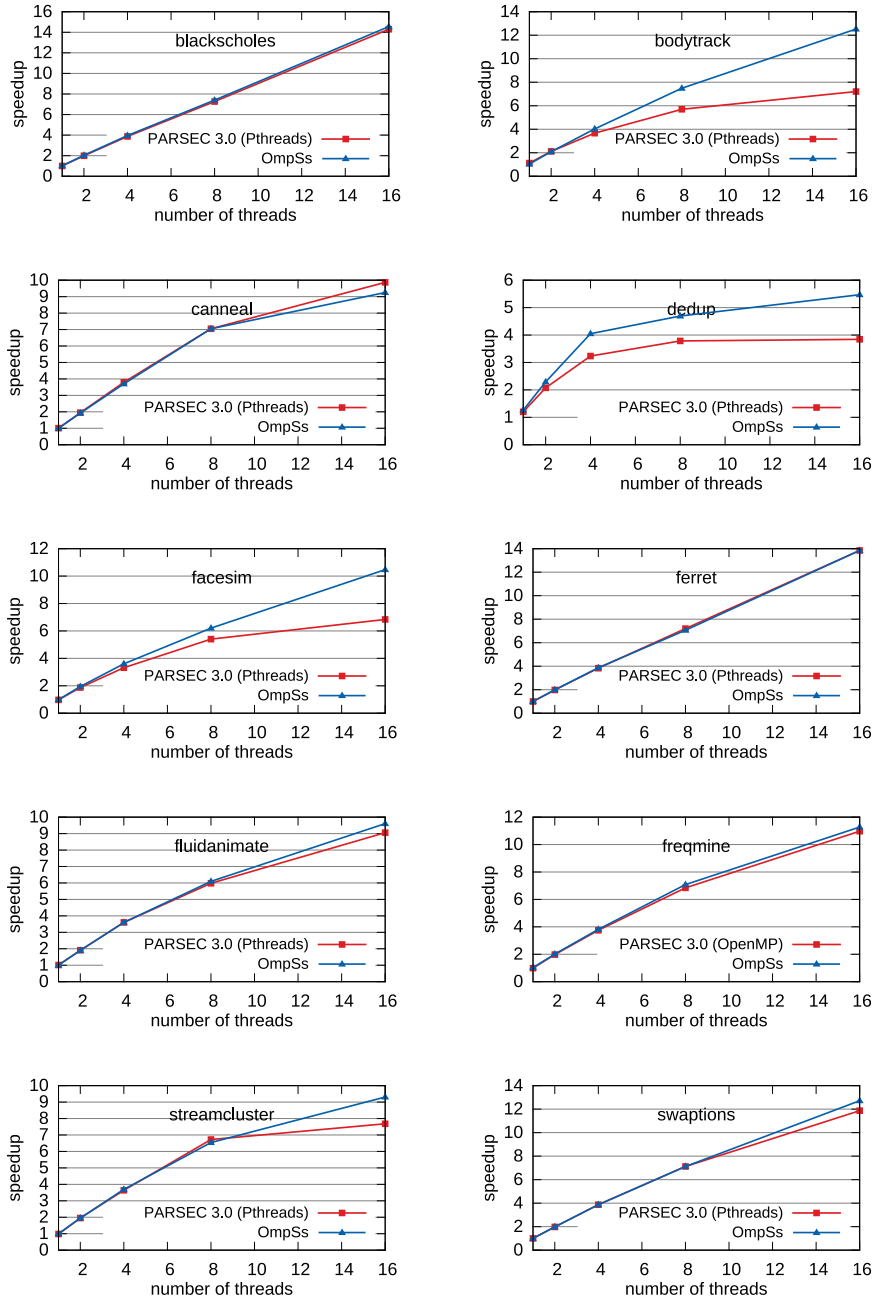


Fig. 6. Comparison of scalability between the task-based implementations and the original (Pthreads/OpenMP) versions.

barrier synchronization, essentially exploiting the same amount of parallelism among all versions (OmpSs/Pthreads/OpenMP).

In the case of `ferret`, although the code is substantially different, both versions employ the same pipeline model and deliver the same level of parallelism, which is already

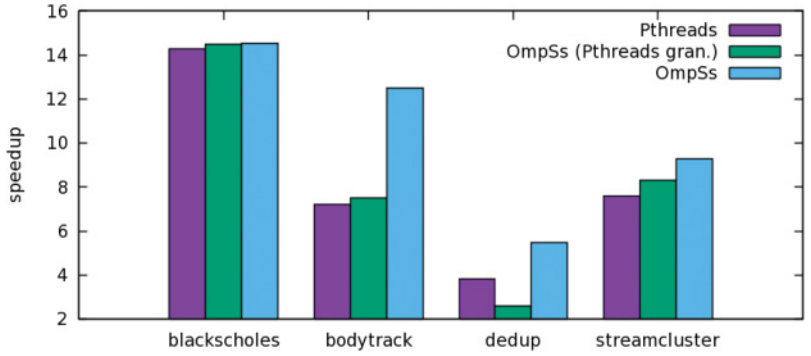


Fig. 7. Speedup comparison for Pthreads and OmpSs with the same granularity, as well as our optimized OmpSs version, when run on 16 cores. Results are normalized to the sequential version of the original code.

high in the Pthreads version. We express a bit of extra parallelism by extending the pipeline with multiple stages, which write to the output file, effectively overlapping some communication with computation. However, the final impact in the total execution time is limited, as the time needed to write the output file is a very small fraction of the total execution time. Finally, we observe performance gain (18%) in `streamcluster`, which can be partly attributed to the more efficient barrier implementation of OmpSswhen compared to the user-implemented barriers of the Pthreads version. However, the most important performance drawback from which the original Pthreads implementation suffers is the negative NUMA effects. This issue is observed when we run our experiments on a two-socket system. The Pthreads code partitions the working set by the number of available cores. We employ a different partition scheme to counter the NUMA effects. Through experimentation, we observe that the best results can be obtained when using 80 blocks.

5.3. Task Granularity Impact

The granularity of individual tasks is an important factor that needs to be considered when parallelizing an application. Small task granularity can reduce load imbalance, but such performance benefits can be neglected by the overhead of the runtime system, as it has to create and schedule more tasks. Results in Section 5.2 show how tuning the task granularity brings performance benefits in some cases (`blackscholes`, `bodytrack`, `dedup`, and `streamcluster`), whereas in others it is better to keep the same parallel granularity as the PARSEC distribution codes (`canneal`, `facesim`, `ferret`, `freqmine`, and `swaptions`).

To provide a more comprehensive comparison, this section examines the performance of `blackscholes`, `bodytrack`, `dedup`, and `streamcluster` when using exactly the same granularity as in the PARSEC distribution code. Figure 7 shows the speedup of these benchmarks when run on 16 cores. The purple bar shows the speedup of the Pthreads version, and the green one shows the speedup of a task-based implementation that has the same parallel granularity as its Pthreads counterpart. Finally, the light blue bar shows the speedup of the optimal task-based implementations discussed in Sections 3 and 5.2.

For the cases of `blackscholes` and `streamcluster`, the parallelization scheme followed in the three codes (Pthreads and the two OmpSs versions) is the same. The difference between the two OmpSs versions is the granularity of the block sizes that are processed per task. In the case of `blackscholes`, the OmpSs implementation with the same granularity as Pthreads does not perform better, as the parallelism of this

benchmark follows a fork-join model. In the case of `streamcluster`, the task-based implementations always improve the Pthreads performance, even if they operate following the same parallelization scheme and granularity as the Pthreads version. These improvements come from the NUMA effects correction that the OmpSs versions carry out.

In the case of `bodytrack`, the optimal OmpSs implementation follows a quite different parallelization scheme than the original Pthreads code, as explained in Section 3. We consider a trivial implementation in OmpSs where we follow the same parallelization strategy as in Pthreads. As shown in Figure 7, we do not observe any significant difference in performance among Pthreads and the equivalent OmpSs implementation. However, the new parallelization scheme is not applicable to Pthreads, as it requires synchronizing the workload by explicit dependencies, which are not available in the Pthreads API.

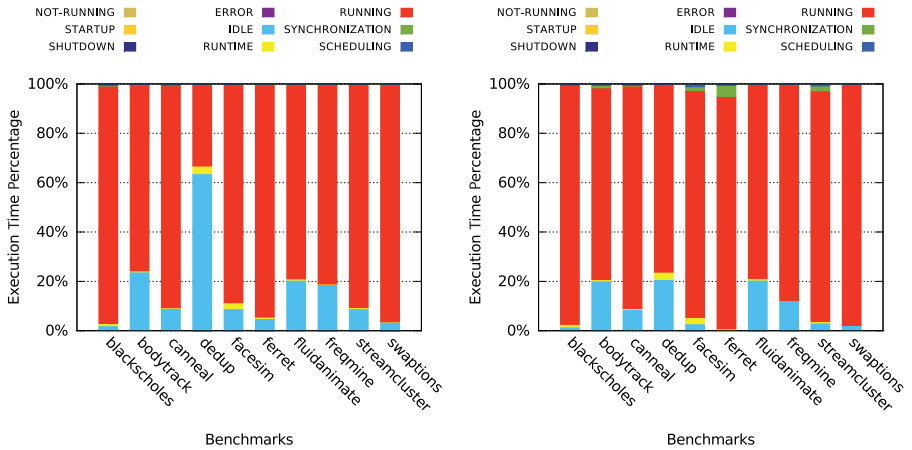
In the case of `Dedup`, the trivial Pthreads-like implementation performs poorly, achieving a speedup of 2.6x. In this implementation, each pipeline stage is taskified following the Pthreads approach. Each large chunk is partitioned into smaller chunks that will spawn three new tasks (`Compress`, `Deduplicate`, and `WriteOutput`). This level of granularity creates hundreds of thousands of tasks, increasing the runtime's overhead significantly. In contrast, the optimized task-based version operates at the granularity of the large chunks, creating only a few hundred tasks, effectively reducing the runtime overhead.

In some cases, OmpSs can overperform Pthreads even if the same parallelization scheme and granularity are followed, as the `streamcluster` results demonstrate. In some other cases (`dedup` and `bodytrack`), the performance improvements come from an optimized parallelization scheme. Such new schemes could be hardly implemented in Pthreads, as they require a direct synchronization via explicit dependencies, which is not available in the Pthreads API. Finally, in case of simple fork-join applications (i.e., `blackscholes`), our performance benefits just come from further optimizing the parallel granularity.

5.4. Runtime System Overhead

Task creation, scheduling, and data dependencies tracking are all handled by the OmpSs runtime system. In this section, we evaluate the impact of these activities over the final parallel performance. Figure 8(a) shows a breakdown of the total execution time of each application. Each bar shows the breakdown of one application after averaging the values over eight concurrently executing threads. The red color representsthe portion of time dedicated in running tasks—that is, in running user code. All applications, excluding dedup, spend more than 75% of the time doing useful work. The cyan bar represents idle time, which corresponds to the time a thread is waiting for some work to become available and is caused by load imbalance and sequential code phases. In most cases, this time is low, except for `dedup`, where it reaches 60% of the total execution time. In Figure 8(a), we also represent the time spent in other activities, such as synchronization and scheduling. None of these activities represent more than 5% of the total execution time.

Figure 8(b) shows the same breakdown of execution time but only for the main thread of execution, which is the one that runs serial parts of the code besides parallel tasks. `Dedup` has significantly lower idle time in the main thread, which indicates that there is not enough parallel work to keep all threads busy. This issue has been reported in the past [Vandierendonck et al. 2013]. In general, we see that the overhead of the runtime system is low, with only a few cases that show some time spent in synchronization, scheduling, and miscellaneous runtime overhead (in light yellow). Synchronization time can be time spent waiting for a barrier or acquiring/releasing locks. Scheduling



(a) Average time over eight threads (b) Main thread runtime breakdowns

Fig. 8. Runtime breakdowns when running on an eight-core configuration.

Table II. PARSEC Parallelization Model and Properties Characterization

Benchmark	Parallel Model	I/O Heavy	Synchronization	LOC Reduction	Performance Improvement
blackscholes	data parallel	✗	dataflow	5.4%	0%
bodytrack	pipeline	✓	dataflow	81.0%	42.0%
canneal	unstructured	✗	locks/atomics	0%	–6.2%
dedup	pipeline	✓	dataflow/locks	38.0%	30.0%
facesim	pipeline	✗	dataflow/barrier	31.0%	34.0%
ferret	pipeline	✗	dataflow	46.0%	0%
fluidanimate	data parallel	✗	dataflow/barrier	21.0%	5.7%
freqmine	data parallel	✗	barrier/locks	0%	2.7%
streamcluster	data parallel	✗	dataflow/barrier/ atomics	33.0%	18.0%
swaptions	data parallel	✗	dataflow	15.0%	6.6%

includes time needed to resolve dependencies and make scheduling decisions, whereas other runtime overheads are related to activity that cannot be associated with task scheduling and creation. Overall, we have seen that our implementations improve scalability considerably (by 13% on average), whereas runtime overhead remains low.

5.5. Characterization of the Applications

In Table II, we characterize the considered applications in terms of parallelization model, I/O intensity, and synchronization scheme. The table also shows code reductions and performance improvements achieved on a 16-core Sandy Bridge system. This table summarizes the properties of applications that make them good candidates for adopting a task-based model.

Applications characterized as data parallel are limited to loop parallelism, where tasks are merely emulating an OpenMP loop construct. In these cases, there is no performance gain, and the programming effort involved either with Pthreads, OpenMP, or tasks is similar. Pipeline applications are better candidates, as they separate the application into discrete abstract stages. Implementing this paradigm with tasks implies taskifying the functionality of each stage and describing the data or control dependencies between them. In Pthreads, the programmer has to implement application-specific

thread pools and queuing systems to achieve the same performance. Additionally, in many cases, task-based models offer an opportunity to easily expand the pipeline stages of the application with sequential and I/O-intensive codes (e.g., `facesim` and `bodytrack`, respectively). Indeed, by replacing locks and barriers, the runtime can discover additional dynamic parallelism and eliminate the cost of acquiring locks. Our task-based parallelization strategies successfully scale up the pipeline applications with a poorly scaling Pthreads version (`bodytrack`, `dedup`, and `facesim`) while reducing the code complexity in all of them. In the case of `ferret`, the task-based version does not perform better than the Pthreads counterpart, as its scalability is already very good (14x on a 16-core machine). However, the reduction in terms of LOCs is dramatic: 46%. In the case of unstructured programs (e.g., `canneal`), task-based programming does not offer any advantage over threading approaches.

Overall, we conclude that task-based parallelism can be used effectively to reduce the effort required to implement pipeline parallelism, and there are o important performance benefits to be gained if the application has no specific thread-pooling mechanisms or I/O-intensive serial regions. In this scenario, the pipeline can be easily expanded to include the I/O region and overlap it with a computation stage of the pipeline.

6. RELATED WORK

Few studies exist that examine the performance of task parallelism compared to other models. Ayguadé et al. [2008] evaluate OpenMP tasks by implementing a few small kernel applications using the new OpenMP task construct. Their evaluation tests the model's expressiveness and flexibility as well as performance. Podobas and Brorsson [2010] compare three models that implement task parallelism, Wool, Cilk++, and OpenMP. They compare their performance using small kernels, as well as some microbenchmarks aimed to measure task creation and synchronization costs. They show that Cilk++ and Wool have similar performance, whereas they outperform OpenMP tasks for fine-grain workloads. On coarser-grain loads, all models have matching performance, with OpenMP gaining in one case due to superior task scheduling.

BDDT [Tzenakis et al. 2012] is a task-based parallel model, very similar to OmpSs, that also uses a runtime to track data dependencies among tasks. BDDT uses block-level argument dependence tracking, where task arguments are processed into blocks of arbitrary size, which is defined by the user. BDDT is shown to outperform loop constructs implemented using OpenMP 2.0.

Other studies exist that compare parallel programming models in the literature. Although these studies do not focus on task parallelism, they employ benchmarks and similar methodology to evaluate their target models. Coarfa et al. [2005] study and compare the performance of UPC and Co-array Fortran—two PGAS languages. They use select benchmarks from the NAS benchmark suite. Appeltauer et al. [2009] use microbenchmarks to measure and compare the performance of 11 context-oriented languages. Their study shows that they all often manifest high overheads.

Although all of the works that we mention try to evaluate various programming models, in terms of performance, and sometimes on usability and versatility, they are all limited to small kernels or even just microbenchmarks. We find that this approach is not sufficient to give us an insight on how a model will impact actual large-scale applications. Karlin et al. [2013] use a proxy application in their work to evaluate several different programming models (OpenMP, MPI, MPI+OpenMP, CUDA, Chapel, Charm++, Liszt, Loci). However, their approach is limited to only one application. Different application domains can be very different and may require different parallelization techniques to get good scalability and performance. A programming model could fail to even provide a way to express a parallelization scheme, let alone deliver

performance. It is important to have an in-depth understanding of a model's behavior and limitation to make an educated decision whether research should direct its efforts to adopt and further expand it.

Pipeline parallelism has been the subject of study in some recent studies. This programming idiom is found often in streaming and server applications and goes far beyond the HPC domain. Lee et al. [2013] propose an extension to the Cilk model for expressing pipeline parallelism on-the-fly without constructing the pipeline stages at their dependencies a priori. It offers a performance comparison between the proposed model, Pthreads, and thread building blocks (TTB) for three PARSEC benchmarks: ferret, dedup, and x264.

7. CONCLUSIONS

In this work, we evaluate the benefits of task-based parallelism by applying it to the PARSEC benchmark suite. We discuss and compare our implementations to their PARSEC Pthreads/OpenMP counterparts. We show how task parallelism can be applied on a wide range of applications from different domains. In fact, by comparing the LOCs between our implementations and the original versions, we make a strong case that task-based models are actually easier to use. The asynchronous nature of task-based parallelism, along with data dependence tracking through dataflow annotations, allows us to overlap computation with I/O phases. The underlying runtime system can take care of issues like scheduling and load balancing without significant overhead.

Our experimental results demonstrate that the task model can be easily applied on a wide range of applications beyond the HPC domain. Although not all applications can benefit from a task-based approach, there are cases where it can greatly improve scalability. The programs that benefit most are those that present pipeline execution model, where different stages of the application can run concurrently. Finally, we plan to make a public release of our task-based implementations to stimulate research on these novel programming models.

ACKNOWLEDGMENTS

The authors are grateful to the reviewers for their valuable comments, to the people from the Programming Models Group at BSC for their technical support, to the RoMoL team, and to Xavier Teruel, Roger Ferrer, and Paul Caheny for their help in this work.

REFERENCES

Aaron B. Adcock, Blair D. Sullivan, Oscar R. Hernandez, and Michael W. Mahoney. 2013. Evaluating OpenMP tasking at scale for the computation of graph hyperbolicity. In *OpenMP in the Era of Low Power Devices and Accelerators*. Lecture Notes in Computer Science, Vol. 8122. Springer, 71–83. DOI:http://dx.doi.org/10.1007/978-3-642-40698-0_6

Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. 2009. A comparison of context-oriented programming languages. In *Proceedings of the International Workshop on Context-Oriented Programming (COP'09)*. ACM, New York, NY, Article No. 6. DOI:http://dx.doi.org/10.1145/1562112.1562118

Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. 2009. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems* 20, 3, 404–418. DOI:http://dx.doi.org/10.1109/TPDS.2008.105

Eduard Ayguadé, Alejandro Duran, Jay Hoeflinger, Federico Massaioli, and Xavier Teruel. 2008. An experimental evaluation of the new OpenMP tasking model. In *Languages and Compilers for Parallel Computing*. Springer-Verlag, 63–77. DOI:http://dx.doi.org/10.1007/978-3-540-85261-2_5

Prithviraj Banerjee. 1994. *Parallel Algorithms for VLSI Computer-Aided Design*. Prentice Hall.

Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. 2006. CellSs: A programming model for the cell BE architecture. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'06)*. Article No. 86. http://doi.acm.org/10.1145/1188455.1188546

Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University, Princeton, NJ.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM, New York, NY 72–81. DOI:http://dx.doi.org/10.1145/1454115.1454128

Fischer Black and Myron S. Scholes. 1973. The pricing of options and corporate liabilities. *Journal of Political Economy* 81, 3, 637–654. http://ideas.repec.org/a/ucp/jpolec/v81y1973i3p637-54.html.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. *ACM SIGPLAN Notices* 30, 8, 207–216. DOI:http://dx.doi.org/10.1145/209937.209958

David R. Butenhof. 1997. *Programming with POSIX Threads*. Addison Wesley Longman.

Barbara Chapman. 2007. The multicore programming challenge. In *Advanced Parallel Processing Technologies*. Springer. DOI:http://dx.doi.org/10.1007/978-3-540-76837-1_3

Barbara Chapman, Gabriele Jost, and Ruud van der Pas. 2007. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. MIT Press, Cambridge, MA.

Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. 2005. An evaluation of global address space languages: Co-array fortran and unified parallel C. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*. ACM, New York, NY, 36–47. DOI:http://dx.doi.org/10.1145/1065944.1065950

Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. 2013. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. *ACM SIGARCH Compututer Architecture News* 41, 3, 308–319. DOI:http://dx.doi.org/10.1145/2508148.2485949

Jack Dongarra, Robert Graybill, William Harrod, Robert F. Lucas, Ewing L. Lusk, Piotr Luszczek, Janice McMahon, Allan Snavely, Jeffrey S. Vetter, Katherine A. Yelick, Sadaf R. Alam, Roy L. Campbell, Laura Carrington, Tzu-Yi Chen, Omid Khalili, Jeremy S. Meredith, and Mustafa M. Tikir. 2008. DARPA's HPCS program—history, models, tools, languages. *Advances in Computers* 72, 1–100. DOI:http://dx.doi.org/10.1016/S0065-2458(08)00001-6

Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters* 21, 2, 173–193. DOI:http://dx.doi.org/10.1007/978-3-642-37658-0_7

Alejandro Duran, Roger Ferrer, Eduard Ayguadé, Rosa M. Badia, and Jesus Labarta. 2009. A proposal to extend the OpenMP tasking model with dependent tasks. *International Journal of Parallel Programming* 37, 3, 292–305. DOI:http://dx.doi.org/10.1007/s10766-009-0101-1

Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06)*. ACM, New York, NY, Article 83. DOI:http://dx.doi.org/10.1145/1188455.1188543

Gosta Grahne and Jianfei Zhu. 2003. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the Workshop on Frequent Itemset Mining Implementations (FIMI'03)*. http://dblp.uni-trier.de/db/conf/fimi/fimi2003.html#GrahneZ03.

Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. *SIGMOD Record* 29, 2, 1–12. DOI:http://dx.doi.org/10.1145/335191.335372

David Heath, Robert Jarrow, and Andrew Morton. 1992. Bond pricing and the term structure of interest rates: A new methodology for contingent claims valuation. *Econometrica* 60, 1, 77–105. http://ideas.repec.org/a/ecm/emetrp/v60y1992i1p77-105.html.

James Christopher Jenista, Yong Hun Eom, and Brian Charles Demsky. 2011. OoOJava: Software out-of-order execution. *ACM SIGPLAN Notices* 46, 8, 57–68. DOI:http://dx.doi.org/10.1145/2038037.1941563

Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary Devito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles H. Still. 2013. Exploring traditional and emerging parallel programming models using a proxy application. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS'13)*. IEEE, Los Alamitos, CA, 919–932. DOI:http://dx.doi.org/10.1109/IPDPS.2013.115

I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. 2013. On-the-fly pipeline parallelism. In *Proceedings of the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'13)*. ACM, New York, NY, 140–151. http://doi.acm.org/10.1145/2486159.2486174

Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2006. Ferret: A toolkit for content-based similarity search of feature-rich data. *SIGOPS Operating Systems Review* 40, 4, 317–330. DOI:http://dx.doi.org/10.1145/1218063.1217966

Matthias Müller, David Charypar, and Markus Gross. 2003. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA'03)*. 154–159. http://dl.acm.org/citation.cfm?id=846276.846298

Dan Nagle. 2005. *MPI—The Complete Reference*, vol. 1, *The MPI Core*, 2nd ed., Scientific and Engineering Computation Series, by Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra. *Scientific Programming* 13, 1, 57–63. DOI:http://dx.doi.org/10.1155/2005/653765

OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface Version 3.0. http://www.openmp.org/mp-documents/openmp-4.5.pdf.

Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. 2009. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications* 23, 3, 284–299. DOI:http://dx.doi.org/10.1177/1094342009106195

Artur Podobas and Mats Brorsson. 2010. A comparison of some recent task-based parallel programming models. In *Proceedings of the 3rd Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG'10)*.

Sean Quinlan and Sean Dorward. 2002. Awarded best paper! Venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*. Article No. 7. http://dl.acm.org/citation.cfm?id=1083323.1083333.

Eftychios Sifakis, Igor Neverov, and Ronald Fedkiw. 2005. Automatic determination of facial muscle activations from sparse motion capture marker data. *ACM Transactions on Graphics* 24, 3, 417–425. DOI:http://dx.doi.org/10.1145/1073204.1073208

Christi Symeonidou, Polyvios Pratikakis, Angelos Bilas, and Dimitrios S. Nikolopoulos. 2013. DRASync: Distributed region-based memory allocation and synchronization. In *Proceedings of the 20th European MPI Users' Group Meeting (EuroMPI'13)*. ACM, New York, NY, 49–54. DOI:http://dx.doi.org/10.1145/2488551.2488558

George Tzenakis, Angelos Papatriantafyllou, John Kesapides, Polyvios Pratikakis, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2012. BDDT: Block-level dynamic dependence analysis for deterministic task-based parallelism. *ACM SIGPLAN Notices* 47, 8, 301–302. DOI:http://dx.doi.org/10.1145/2370036.2145864

Hans Vandierendonck, Kallia Chronaki, and Dimitrios S. Nikolopoulos. 2013. Deterministic scale-free pipeline parallelism with hyperqueues. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13)*. ACM, New York, NY, Article No. 32. http://doi.acm.org/10.1145/2503210.2503233

Hans Vandierendonck, Polyvios Pratikakis, and Dimitrios S. Nikolopoulos. 2011. Parallel programming of general-purpose programs using task-based programming models. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism (HotPar'11)*. 13. http://dl.acm.org/citation.cfm?id=2001252.2001265.

Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. 2011. Using a "Codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT'11)*. ACM, New York, NY, 64–69. http://doi.acm.org/10.1145/2000417.2000424.

Received March 2015; revised September 2015; accepted September 2015