A Benchmark Suite for Evaluating Parallel Programming Models
Introduction and Preliminary Results

Michael Andersch
Technische Universität Berlin
Einsteinufer 17
10587 Berlin
andersch@cs.tu-berlin.de

Ben Juurlink
Technische Universität Berlin
Einsteinufer 17
10587 Berlin
juurlink@cs.tu-berlin.de

Chi Ching Chi
Technische Universität Berlin
Einsteinufer 17
10587 Berlin
cchi@cs.tu-berlin.de

***Abstract*—The transition to multi-core processors enforces software developers to explicitly exploit thread-level parallelism to increase performance. The associated programmability problem has led to the introduction of a plethora of *parallel programming models* that aim at simplifying software development by raising the abstraction level. Since industry has not settled for a single model, however, multiple significantly different approaches exist. This work presents a benchmark suite which can be used to classify and compare such parallel programming models and, therefore, aids in selecting the appropriate programming model for a given task. After a detailed explanation of the suite's design, preliminary results for two programming models, Pthreads and OmpSs/SMPSs, are presented and analyzed, leading to an outline of further extensions of the suite.**

I. Introduction

Due to physical limitations of the process technology and the depletion of ILP sources, improvement of single threaded performance has mostly stopped. This has led to a paradigm shift in the development of new processor architectures and is expected to lead to highly parallel compute architectures ("many-cores") with hundreds or thousands of cores in the future [1].

The move towards multi-core architectures, however, changes the programmer's view of the architecture and introduces yet unresolved programmability issues. Increasing performance now requires the explicit exploitation of thread-level parallelism. The development of parallel programs is generally not a trivial task since an appropriate parallel decomposition of the algorithms needs to be found. If the parallel 'pattern' is not chosen well, it complicates thread synchronization and data management which in turn increases code complexity. Additionally, the programmer usually has to perform architecture-specific optimizations such as thread-to-core mapping, page placement or optimizations regarding the bandwidth of the interconnection network and cache performance, which could lead to different optimal parallelization strategies for different platforms. Naturally, this imposes both decreased performance and portability to machines featuring different architectural approaches. It must be questioned if scalability to larger core counts can be ensured. Furthermore, the verification and debug processes of such programs introduce additional difficulties caused by the complexity associated with sophisticated threads running in parallel.

All this has led to the introduction of several *programming models* in an attempt to relieve developers partly or completely from such parallel programming issues. These models, however, differ significantly in the provided underlying parallel principles, abstraction levels, semantics, and syntax. Some recently introduced models are OpenMP [2], POSIX threads [3], Cilk [4], Intel TBB [5], CUDA [6], and OpenCL [7]. All of them focus on or develop towards shared memory computer architectures. Thus, it can be imagined that more parallel programming models for such architectures will be introduced in the future. Therefore, some means for evaluating these shared memory programming models, against each other and against software project development constraints, are needed. This way, comparability is introduced which will allow developers to choose the programming model that fits their requirements best.

This work aims at providing such means by introducing an *evaluation suite* consisting of several applications to examine and classify the features and qualities of shared memory parallel programming models. These applications will not only be used as benchmarks to measure performance levels of programs developed in a particular model, they also allow hands-on examination of the usability and features of that model while actually being ported. Additionally, a first version of this suite is presented along with preliminary results for two currently relevant models. The contributions of this work can be summarized as follows:

- We propose a benchmark suite specifically targeted at the evaluation of performance and usability of parallel programming models rather than parallel machines.
- We focus on modularity and portability for the benchmarks incorporated into the suite.
- We perform a case study, evaluating the novel OmpSs programming model [8], using POSIX threads as a reference for comparison.

This paper is structured as follows. Section II encompasses the top-level design decisions made in creating the evalua-

tion suite. In Section II-A, we discuss general requirements the suite must fulfill. In Section II-B, these requirements are utilized to create a preliminary selection of benchmarks which are then presented in a more detailed fashion. A case study using the benchmark suite is presented and analyzed in Section III, leading to an preliminary comparison between the two programming models considered. Section IV discusses related work. Finally, in Section V, conclusions are drawn and future perspectives are given.

II. Suite Design

On a high level, the benchmark suite must fulfill several critical requirements which can be derived from its objective as a suite to evaluate the programmability and performance of parallel programming models. These criteria will then function as guidelines for the selection of benchmarks and benchmarking practice. In the following section, we identify six such criteria, explaining for each how it contributes to the suite's objectives.

A. General Requirements

In order to provide a wide basis for comparison, the first requirement the suite must fulfill is:

1) A broad range of application domains must be covered.

Examples of application domains are image processing, financial computations, physical simulations, and computer vision. Especially, the suite should include and be extended to newly explored computational fields, such as online data mining and real-time ray tracing, which only become feasible through the proliferation of multi-core architectures and parallel software. This ensures the usefulness of the suite for a wide range of developers.

Different domains, however, do not necessarily ensure that different types of parallelism are covered. Thus, a second requirement is:

2) Various parallel patterns and characteristics must be covered.

A parallel *pattern* is a scheme of how and where parallelism can be found in an algorithm [9]. Well-known patterns are, e.g., fork/join, divide-and-conquer or pipeline parallelism. By collecting programs covering several of these patterns, it is possible to evaluate the suitability of a programming model for different types of algorithmic approaches. In addition, because these patterns are universal, it is likely that their importance persists over time. This way, we capture characteristics of programs, not programs themselves.

The parallelization style and software domain are not the only important classifications, however. A third requirement which must be fulfilled is:

3) Various application sizes must be covered.

By including benchmarks of multiple, different sizes, ranging from small, extracted kernels up to large, full-scale applications with a multitude of interacting subsystems, we assure the possibility to judge whether a specific programming model can easily be applied in the development of applications of a certain size. Additionally, this increases the suite's usability since there are both small, easily portable programs, ideal to quickly mediate a first impression of a programming model's handling and behavior, and large, more sophisticated ones, to acquire in-depth information.

To examine a key property of the programming models, a fourth requirement was identified:

4) The suite must include input data sets of varying size.

Usually, for any programming model, the features as well as the additional abstraction come at the cost of longer initialization phases, used to allocate memory for the runtime, create threads and initialize data structures. Naturally, for programs which spend more time processing data, the impact of this overhead on the total execution time decreases. To be able to examine this effect and test with varying amounts of parallelism, for all benchmarks input data sets of various sizes and complexities need to be provided. To maintain significance of the suite in the future, it should be possible to easily extend the given data sets to include ones which are larger or display different characteristics.

The fifth requirement addresses the portability requirement induced by a suite which targets programming models:

5) Simplicity, modularity and portability must be ensured.

Previous benchmark suites (Section IV) have focused on the analysis of processor performance. However, the evaluation of a programming model is a different process, necessarily consisting of more than only comparing raw performance numbers achieved by the ported benchmarks. While performance is clearly important, the experiences obtained during the course of their creation are also of major importance. To be able to investigate this, it must be as easy as possible to port the suite to new, upcoming programming models, an approach comparable to the work done in [10]. However, all of the aforementioned benchmark suites incorporate their benchmarks into strictly structured frameworks and do therefore not provide easy access to the program cores. This makes them unsuitable for the changeability this work's design requires.

The last criterion we discuss is

6) The parallelization approach must be fixed and well documented.

Generally, different parallel algorithms for a specific problem can be designed. A single, optimal design does neither always exist nor is immediately obvious. Without artificially limiting parallelism, it still is necessary to set up guidelines on how the problem is supposed to be parallelized to ensure comparability. Otherwise, higher performance or easier development could also be caused by the utilization of a different parallelization approach, not only by features and properties of the programming model. The benchmark suite must therefore provide a clear description of the parallelism in each application that is supposed to be exploited.

B. Benchmark Suite

The current benchmark suite is presented in Table I. The **K**, **W**, and **A** identifiers in the second column are a realization of

TABLE I
BENCHMARKS

Name	Type	Application	Domain	Problem Sizes	Code Size
c-ray	K	Offline Raytracing	Computer Graphics	18 / 192 objects	500 LOC
md5	K	MD5 Calculation	Cryptography	various	1000 LOC
rgbcmy	K	Color Conversion	Image Processing	3.8 / 30.5 MP	700 LOC
rotate	K	Image Rotation	Image Processing	3.8 / 30.5 MP	1000 LOC
kmeans	K	k-Means Clustering	Artificial Intelligence	various	600 LOC
rot-cc	W	rotate + rgbcmy	Combined Workload	3.8 MP / 30.5 MP	1400 LOC
ray-rot	W	c-ray + rotate	Combined Workload	18 / 192 objects	1300 LOC
h264dec	A	H.264 Decoding	Video Processing	Full HD / QHD video	20000 LOC

the third requirement, grouping benchmarks into one of three different categories, **K**ernels, **W**orkloads, and **A**pplications.

A kernel consists of (a part of) the extracted core of a real-world application. Kernels are, therefore, comparably small ($<$ 1000 LOC) and exhibits only a single, isolated parallelization pattern.

Workloads are either derived by combining several kernels, thereby introducing additional data dependencies and covering more parallelization patterns, or it is a program considered too large to fit in the kernel class, but still only an extracted part of a real application.

Applications are full-grown software products which are widely used in industry or science and therefore feature the highest number of subsystems, of dependencies and combinations of parallelization patterns. Nonetheless, applications are as well simplified as far as possible, containing only the parts relevant for use in this suite.

Following is, in the order depicted in Table I, a short description of each benchmark, the parallel pattern, and an explanation of why it is considered relevant. It should be kept in mind this is a preliminary selection which shows the current and not the final state of the suite and is subject to change and extension.

1) c-ray: c-ray is a simple, brute-force ray tracer [11]. It is small (ca. 500 LOC for the serial version) and renders an image in PPM binary format from a simple scene description file. As is common for ray tracing, it exhibits data-level parallelism since all pixel colors can be computed fully independent due to the lack of post-processing and optimization. This is illustrated on using three threads in Figure 1. It is shown how the target image is divided into three *work units* of several pixel scanlines and how the threads process the work units. Grouping scanlines into blocks is usually required to coarsen the task granularity, thus reducing threading overhead. Since the processing of pixels grouped to a scanline is common, the parallelism in c-ray is supposed to be exploited this way. Nevertheless, all threads need shared access to the complete geometric input data since the tracing of any ray may lead to an intersection with any scene object.

Despite its simplicity, c-ray is a very compute-intensive benchmark, featuring a high computation-to-communication ratio. It also requires a load balancing mechanism since the computational effort required to compute pixel colors depends on the location of the pixel in the scene.

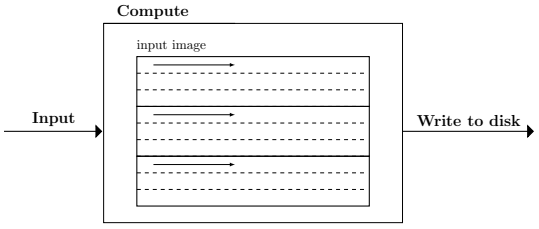


Fig. 1. Parallel pattern for c-ray, rotate and rgbcmy kernels

2) md5: md5 is a benchmark utilizing a standard implementation of the MD5 hash algorithm [12] to produce hash values. It is also present in the EEMBC MultiBench benchmark suite [13]. Since there is no exploitable thread-level parallelism in the block cipher construction used in MD5, the benchmark uses multiple input buffers consisting of predefined raw binary data which it processes in parallel. The MD5 digests are then written into one consecutive output buffer. Since there are no dependencies between the input buffers, this program can be classified as fully data parallel and is relatively straightforward to parallelize. Its structure is shown in Figure 2; parts drawn shaded illustrate exploitable parallelism.

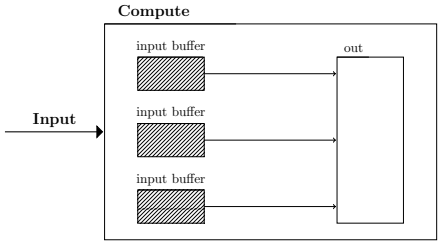


Fig. 2. Parallel pattern for md5 kernel

3) rgbcmy: The *rgbcmy* kernel converts an input RGB PPM image to the CMYK color space used for image printing. Like md5, it is also present in the EEMBC MultiBench benchmark suite [13]. As in c-ray, parallelism is found in the different pixels (which can be converted independently), visualized in Figure 1, and can be classified as data-level parallelism. Compared to c-ray, however, rgbcmy features a lower computation-to-communication ratio, combining a color transformation calculation with the write-back of pixels into memory. Work unit granularity, as in c-ray, is controlled by grouping image scanlines into (fixed-size) blocks, yielding the

same fixed parallelization strategy as previously described for c-ray.

4) rotate: *rotate* is a benchmark, as well present in the EEMBC MultiBench benchmark suite [13], which rotates an RGB image in binary representation by 0, 90, 180 or 270 degrees. Similar to c-ray and rgbcmy, it exhibits data-level parallelism in the independence of all image pixels. The parallelization approach can thus also be visualised by Figure 1. The fixed rotational angle allows rotating the image by mapping the pixels indices only and does not require a matrix rotation or shear. Rotate is different from c-ray and rgbcmy, however, since it features relatively few computations but stresses the memory subsystem instead.

5) kmeans: The *kmeans* kernel executes the k-Means clustering algorithm [14] used in the domains artificial intelligence and data mining. It is derived from the correspondent benchmark in the NU-MineBench benchmark suite [15]. For a given set of input points in an n-dimensional space, it iteratively selects k cluster centers, computes the cluster membership of each input point and averages all cluster points to obtain new cluster centers until a stable state is reached. A stable state is reached when the percentile of points changing membership in an iteration is below a fixed threshold. Kmeans, therefore, consists of two iteratively repeated phases, a clustering phase in which the cluster closest to each point is computed and a reduction phase in which the previously obtained information is used to compute new cluster centers. This resembles the common MapReduce pattern. The algorithm is visualized in Figure 3.

This benchmark can be parallelized by exploiting the independence of all input points. We focus, however, on exploiting this independence for the distance computation phase only since it is the computationally most demanding one.

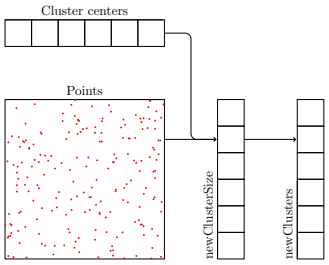


Fig. 3. Algorithm for kmeans kernel

6) rot-cc: As mentioned before, workloads consist of combinations of kernels. The first workload combines the *rotate* and *rgbcmy* kernels and is therefore called *rot-cc* (for rotation + color conversion). By feeding the output from the image rotation into the color conversion, it introduces function-level parallelism between the two kernels which can be exploited in addition to the parallelism in each kernel. The parallelization structure is illustrated in Figure 4. Interesting cases are those where the rotation changes the image orientation (90 and 270 degrees) since this leads to a strided memory access pattern for the color conversion kernel.

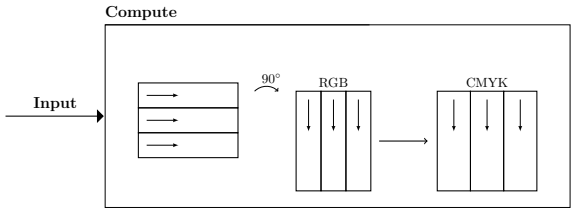


Fig. 4. Parallel patterns for rot-cc workload

7) ray-rot: By chaining the *c-ray* and *rotate* kernels, we obtain the *ray-rot* workload. It exhibits additional function-level parallelism and is especially interesting because the two phases are highly different in characteristics and must, therefore, be load balanced to achieve high performance. *ray-rot* is illustrated in Figure 5.

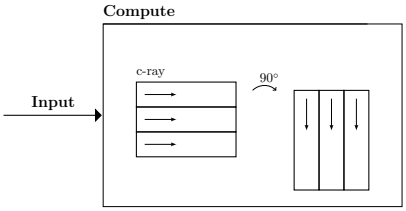


Fig. 5. Parallel patterns for ray-rot workload

8) h264dec: *h264dec* is an H.264 decoder [16], derived from FFmpeg, a free, cross-platform H.264 transcoder [17].

For the H.264 decoder benchmark, parallelism is exploited at two levels: function-level and data-level parallelism (DLP). First, in the decoder stages, each stage can be performed in parallel in a pipeline fashion on different frames as shown in Figure 6.

Additionally, DLP is exploited within the entropy (ED) and macroblock decoding (MBD) stages. In the ED stage, macroblocks in different frames can be decoded in parallel as long as the co-located macroblock in the previous frame has been decoded before. Only one macroblock at a time can be decoded per frame because each macroblock depends on its predecessor. In the MBD stage, DLP can be exploited within a single frame. Macroblocks, for which the upper right and left neighboring macroblocks have already been decoded, can then be decoded in parallel. The DLP of the ED and MBD stages is illustrated in Figure 7. Here, hatched blocks denote data that can be processed in parallel.

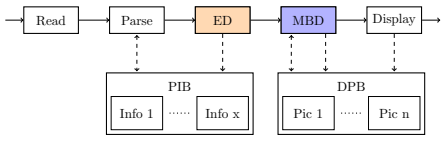


Fig. 6. Pipeline parallelism in h264dec

III. Case study: Pthreads vs. OmpSs

We now present performance results and the documentation of usage experiences comparing two programming models,

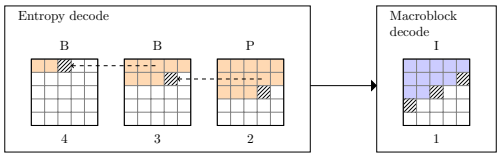


Fig. 7. DLP parallelism in ED and MD stages

Pthreads and OmpSs/SMPSs [8]. In Section III-A, we describe the features of the programming models in comparison. After the experimental setup is presented in Section III-B, Section III-C compares the speedup characteristics of the different benchmarks. From this comparison, we derive information about the benchmark behavior and gain a first impression on how the two models compare to each other. Then, in Section III-D, we study in more detail the performance differences observed in two example benchmarks taken from the suite. Finally, Section III-E specifies the distinct experiences with the two programming models from a programmers perspective, giving examples of criteria which need to be compared besides performance.

A. Evaluated Programming Models

The POSIX thread library [3] provides basic threading support for the C programming language. Synchronization is achieved using mutexes to protect critical sections and condition variables to achieve thread synchronization. The threads themselves have to be created, managed (i.e., set to a certain priority or in a detached state) and terminated explicitly. Pthreads thus fully leaves the management of the parallel algorithm to the programmer, enforcing him or her to consider dependencies, synchronization points, and possible race conditions in a direct, exposed way.

OmpSs/**SMPSs** [8] is the **SMP** instance of the *OpenMP SuperScalar model* (OmpSs). It is a novel task-based programming model which consists of a runtime library and a source-to-source compiler. SMPSs requires the programmer to annotate functions as tasks using *#pragma css task* directives and label every task argument as an *input*, *output*, *inout*, or *reduction* parameter. These keywords declare an argument either read-only, write-only, read-write or as part of a reduction operation. Once such a task is created, it will be added to a runtime data structure, called the *task dependency graph*. The task graph is maintained and populated by the underlying runtime system which performs the dependency resolution and the scheduling of tasks on worker threads. This is similar to the way a superscalar processor dispatches instructions to available execution units. The only additional synchronization constructs SMPSs provides to the programmer are a *barrier* directive, which requires all previously created tasks to finish, a *wait on* directive, used to wait for a certain task to complete, and a mutex, which currently is required for reduction operations. An advantage of SMPSs is that the serial base code is maintained, allowing profiling and debugging of the sequential code with established tools. Its functionality can easily be regained by compiling an SMPSs program with a compiler not recognizing the preprocessor pragmas.

B. Experimental Setup

All available results have been obtained during the development of this benchmark suite and are therefore neither specifically optimized nor have been analyzed in detail. Their main objective at this stage is to classify and analyze the early benchmark behavior. Due to this early state of the described benchmarks, results for kmeans and h264dec are not yet included. Our evaluation platform is a 64-core cc-NUMA system with the following features:

- 8x Xeon X7560 (Nehalem EX architecture),
- 2.26 GHz clock frequency,
- HyperThreading disabled,
- 2 TB RAM,
- 204.8 GB/s aggregate memory bandwidth.

Each reported result is the average of three runs. Timing is done using timestamps inside the benchmarks and always excludes the I/O-phases (i.e., loading the input from disk into memory and cleaning up). Additionally, the execution time of all programs has been measured using both a small and a large input data set (see Table I for details).

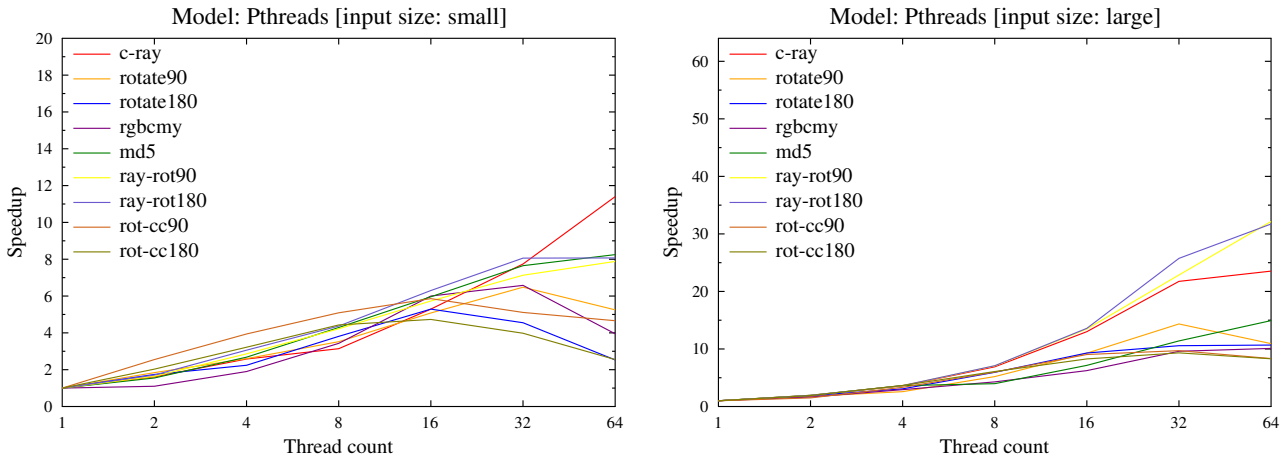
C. Preliminary Scaling Results

In this section, the preliminary results for the Pthreads and SMPSs versions of the benchmarks are discussed. The speedup has been obtained by dividing the execution time on one processor by the execution time on n processors *for the same program*, thus normalizing the speedup factor for a single core to one.

The speedup results for up to 64 cores for the Pthreads programming model using small input data sets are shown in Figure 8(a), the ones for large input data in Figure 8(b). The corresponding ones for SMPSs can be found in Figures 9(a) and 9(b).

The figures show that the behavior varies widely across the applications. For the highest thread count of 64, the Pthreads benchmarks achieve speedup factors ranging from 2.53x to 11.4x for the small and from 10.1x to 31.7x for the large input data sets. For the same number of threads, the SMPSs benchmarks achieve speedups between 1.7x and 33.4x for the small and 3.0x and 52.5x for the large input data sets. The differences observed between small and large input sizes are caused by the naturally higher amounts of DLP, leading to a coarsened granularity of the work units and thus diminishing the impact of the threading overhead. The results show that regarding only speedup and not real execution time, for these benchmarks, SMPSs performs on average two times better than Pthreads. For SMPSs, the runtime must be initialized before and shut down after any calls to it are made. This is excluded from the timing, while for Pthreads, thread creation is mostly tightly coupled with the actual execution and is therefore generally included. This is one of the reasons for the higher speedups of SMPSs.

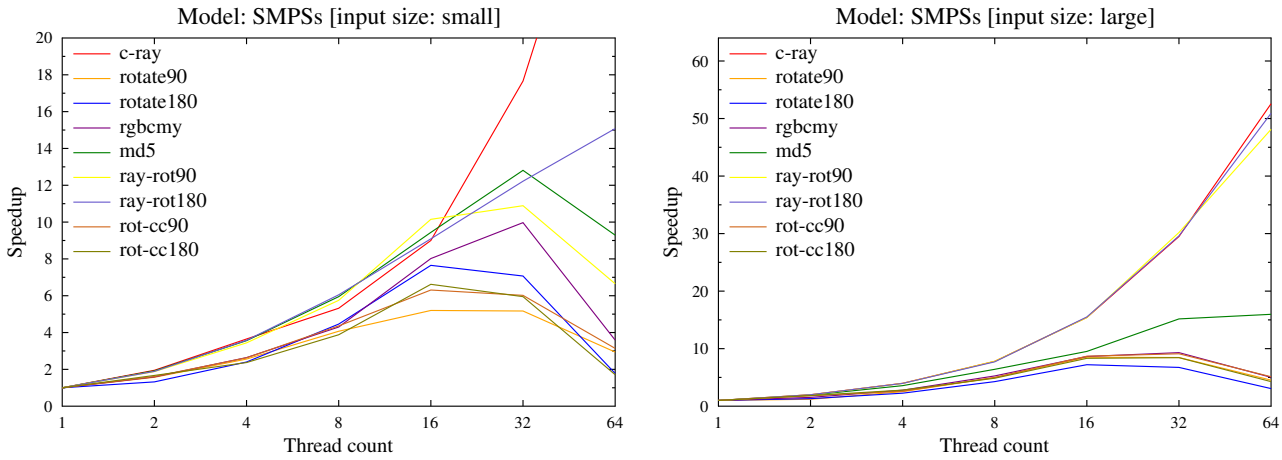
The 90 and 180 suffixes for the rotation benchmarks denote the rotational angle. Experimentation has shown that different



(a) Small input size

(b) Large input size

Fig. 8. Baseline performance for Pthreads



(a) Small input size

(b) Large input size

Fig. 9. Baseline performance for OmpSs/SMPSs

angles potentially influence the cache behavior since memory is accessed linearly for 180 degrees and in a strided pattern for 90 degrees of rotation. This leads to higher execution times for the 90 degrees case.

The highest speedups are achieved for benchmarks which include ray tracing. This is expected since c-ray has a high computation-to-communication ratio. The performance of the Pthreads version of the c-ray kernel for large inputs saturates, however. In this case, performance increases by only 8% when going from 32 to 64 threads, compared to an average 70% for the other test cases (Figures 8(a), 9(a), 9(b)). This is fully reproducable and will be investigated further.

The lowest speedups are achieved by benchmarks which include the rotate kernel (and do not also include ray tracing). The reason for this supposedly is the cc-NUMA evaluation platform. Because there is only a small amount of computation in the rotate kernel, increasing the thread count does not improve performance but instead leads to memory contention, causing a high amount of (coherence) traffic. This is especially the case for the transition from 32 to 64 threads where four additional processor sockets are used for 64 threads, resulting in a lower speedup than for 32 threads.

Other observations that can be made from the graphs are:

- The md5 kernel should scale perfectly since the tasks are independent and it features only small amounts of I/O, but actually it performs poorly.
- Even though in c-ray the parallelism is easy to exploit, SMPSs generally shows a much higher performance than Pthreads for this kernel, especially for high thread counts. It needs to be noted here again that the Pthreads versions of the benchmarks have not been optimized yet.
- The rotate kernel, in contrast to the other benchmarks, performs better with Pthreads than with SMPSs.
- For Pthreads, the 90 degrees rotate kernel is slower than for 180 degrees but the achieved speedup of the 90 degrees variant is higher.
- For large input data, the ray tracing phase in the ray-rot workload masks the influence of the rotation phase

completely, scaling almost as good as the ray tracing kernel.
- None of the benchmarks achieves optimal speedup.

A deeper analysis of these observations and further machine-specific optimizations are future work.

D. Preliminary Performance Comparison

In this section we select two benchmarks in order to perform a direct performance comparison between the two programming models. The main difference between the following and previous figures is their normalization. Here, instead of comparing relative *speedup*, *performance* is compared by normalizing to the *fastest* execution time, regardless of which programming model it is produced by. Therefore, higher numbers not only resemble higher speedup but higher absolute performance. For each benchmark, we first explain in detail how parallelization has been performed and then discuss the performance characteristics.

1) c-ray kernel: The first benchmark examined is the c-ray kernel. Results are shown in Figures 10(a) and 10(b).

The figures show that the speedups achieved by the SMPSs implementation, as shown in Section III-C, directly translate to a performance advantage for c-ray.

In the Pthreads variant, the image vertical resolution y is divided by the thread count t, distributing y/t scanlines to each thread for processing. After they have been created, threads wait until a start flag is raised to exclude the thread creation time from the overall timing. Thread termination as performed by a join, however, is included in the timing. This static work distribution approach features a very low overhead but no load balancing is performed between threads.

In the SMPSs variant, for every image scanline a task is created which calculates all pixel colors and writes its output to the respective location in the pixel color buffer. The desired task granularity as specified by the SMPSs documentation [18] is $250\mu s$ which is larger than the time it takes to process an empty image scanline. To detect which scanlines are empty prior to processing requires significant changes to the sequential base code and reduces performance and, therefore, has not been performed. Termination of all tasks is assured using a barrier. Runtime startup and shutdown are not included in the timing.

While Pthreads work units are larger and in consequence incur less overhead, the static mapping of work units to threads creates a load imbalance. The dynamic task scheduling capability of SMPSs results in significantly improved load balancing, leading to higher performance.

2) rot-cc workload: The second benchmark considered is the rot-cc workload. Performance results are given in Figures 11(a) and 11(b).

In the Pthreads variant the workload is parallelized by maintaining the parallelization structure of both kernels and placing an implicit barrier in between, enforced by joining the threads of the rotation phase once it is completed and creating new ones for the second phase. In the SMPSs variant, parallelization is performed the same way, creating tasks for both rotation and color conversion with a barrier in between.

For SMPSs, the rotation of a fixed number of image rows and the color conversion of some image rows/columns (depending on the rotation angle) are declared as tasks. However, 90 and 270 degrees of rotation result in a strided memory access pattern for the color conversion. In its current state, the SMPSs compiler and runtime cannot handle dependencies on such memory "regions" (however, such an extension to the programming model is currently in development [19]). To circumvent extensive remapping of the intermediate image buffer to accommodate these shortcomings, a barrier directive is inserted between the two workload phases, effectively eliminating pipeline parallelism, but assuring correctness.

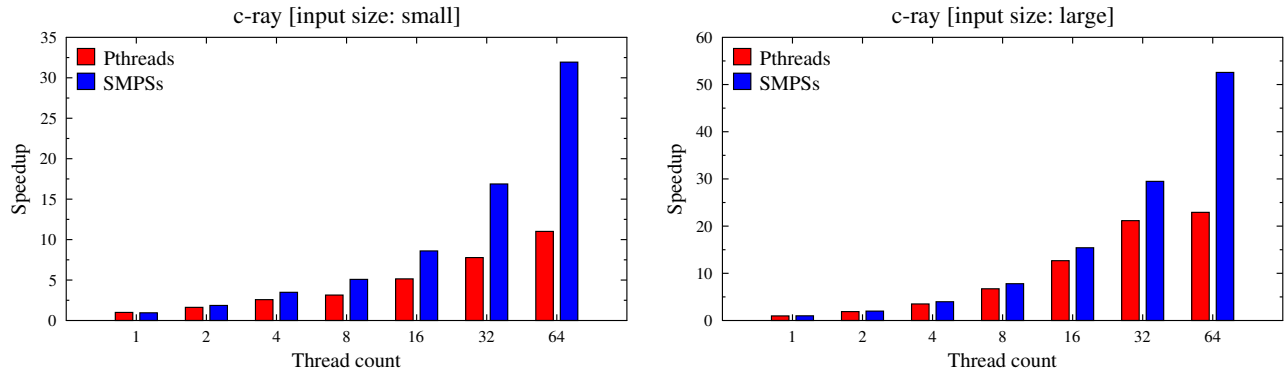
The figures show again a significant real performance advantage for SMPSs. Especially surprising is the almost doubled performance of SMPSs over Pthreads for a single thread. This is remarkable since the SMPSs concept in this case enforces the runtime to run on the same core as the actual computation. On the contrary, the Pthreads version of the program is architected to (for each phase) launch a single worker thread processing the whole image while the main thread is waiting for it to finish. The performance difference is caused by a thread scheduling issue. For the Pthreads version, a speedup of almost two is achieved by binding the whole program to a single processor core. Further investigation of the cause of this behavior (cache organization, prefetching) is future work. Another characteristic to be observed is the higher *relative* performance of Pthreads for large inputs. Both models also slow down noticeably when going from 32 to 64 threads, an anomaly that can be observed for many benchmarks.

E. Development Experiences

As remarked in Section II-A, performance is only a part of the evaluation of a programming model. While higher abstraction levels are targeted by most parallel programming models, it must be questioned whether low-level control can be traded for abstraction. It is a field of active research whether a higher abstraction level can keep the same expressiveness as the usually more complicated, basic counterpart [20]. Therefore, in this section, we want to give exemplary descriptions of notable experiences obtained in the process of parallelizing our benchmarks with the two considered programming models. For each programming model we will discuss the following qualitative and quantitative metrics:

- Code size,
- Code transformations required,
- Expressiveness,
- Tool chain,
- Verification and debugging.

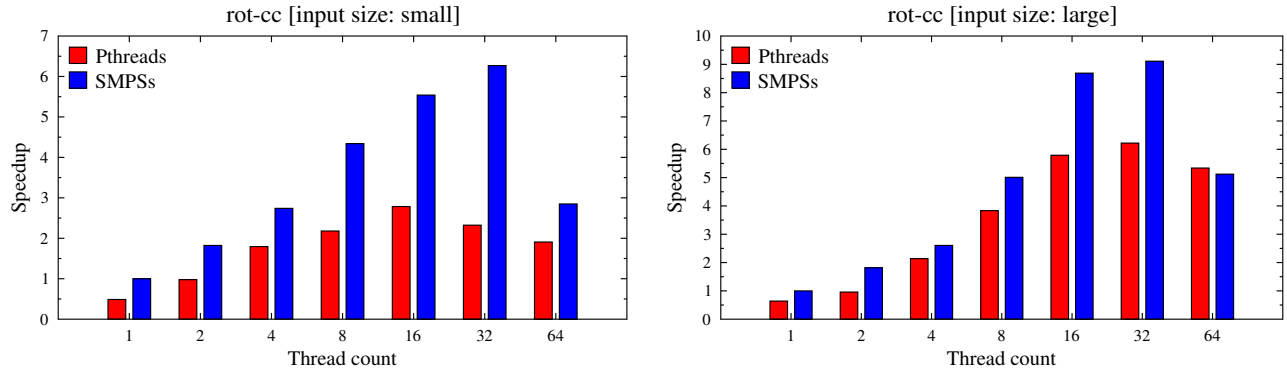
The Pthreads benchmarks are in general larger than their serial counterparts. This increase is due to the necessity of using specialized threading data structures and multiple function calls. SMPSs has a clear advantage in this respect since the only additions necessary to the code are pragmas to start and end the runtime and declare tasks. This advantage,



(a) Small input size

(b) Large input size

Fig. 10. Direct performance comparison of the Pthreads and SMPSs implementations of the c-ray kernel



(a) Small input size

(b) Large input size

Fig. 11. Direct performance comparison of the Pthreads and SMPSs implementations of the rot-cc workload

however, only holds if no manual code transformations have to be applied to expose the parallelism. For Pthreads, such transformations were necessary several times. The programmer must reorganize the code into thread functions and thread inputs must be grouped, if not global, into a single parameter structure. SMPSs, given a serial program that can be straightforwardly parallelized, only enforces the programmer to decompose the function arguments into parts (because structure member access and pointer dereference is impossible in SMPSs directives) to allow the direct annotation of these parts as inputs or outputs. If the program cannot be straightforwardly parallelized, fundamental changes to the serial base code might be required.

Regarding the expressiveness of the two models, with Pthreads, any kind of parallelism can be expressed. However, since fine-grained control over all condition and lock variables is required, the program code quickly becomes unintuitive to read. This is especially true for large programs where more parallel threads are interacting. Expressing parallelism in SMPSs is, on the contrary, much more straightforward. However, our experiences are that several well known parallel programming patterns cannot be conveniently expressed using SMPSs. For example, pipeline parallelism requires buffers between two consecutive pipeline stages to decouple them. With Pthreads, this can be implemented using any kind of dynamic buffers like queues or ringbuffers to communicate between stages. With SMPSs, currently this is not possible.

The use of Pthreads only requires to link the library to include by the C preprocessor and therefore does not impose changes in the toolchain employed. SMPSs, on the contrary, provides a complete runtime system and a specialized compiler, leading to the issue of missing support for several established compilation parameters (e.g. $-$ffast$-$math). Another consequence of the changed compiler is the occurrence of syntactic constructs that the SMPSs compiler does not support while conventional C compilers do (example: **const char**$*$ foo = {”hello\n” ”you”}).

A well-known problem when developing parallel programs is debugging. It is difficult to debug programs where several things happen at the same time. This applies to Pthreads as well as SMPSs. Debugging an SMPSs program, however, is significantly more difficult for three reasons: First, the source-to-source compiler of the SMPSs programming model makes debugging with a conventional debugger difficult since the code which is debugged bears only a faint resemblance to the code that was actually written. Second, in the OmpSs programming model the programmer has only an abstracted view of the underlying task execution system. In the case of

program misbehavior, the raised abstraction level could turn into a serious issue since a clear understanding of the program behavior is required to efficiently locate and eliminate bugs. Third, established tools like gdb lack support for SMPSs-specific language primitives and can therefore only be used to debug SMPSs programs in a naive way. Tools to help debugging SMPSs programs are currently being developed [21].

IV. Related Work

Benchmark suites have been developed previously, including several proprietary, domain-specific products [22]–[25]. PARSEC [26], [27] is a recent benchmark suite consisting of 12 programs. It includes industry-level workloads, covering a multitude of application domains. The target platforms of PARSEC originally are chip multiprocessors, however, the programs included in the suite are not inherently limited to this usage scenario. The stated goal of PARSEC is to discover new trends in the research and development of parallel machines, algorithms and applications. They focus on development and applicability insights, not on sheer performance numbers. This is achieved by not only covering multiple significantly distinct applications, but by also featuring different parallelization models for each. Featured in PARSEC are variants for Pthreads, OpenMP and Intel Thread Building Blocks. Support also includes more than one architecture type, extending to big-endian machines. The suite contains parallelization patterns such as full data parallelism and pipeline parallelism. However, PARSECs set goal is also to provide a fix framework for benchmark execution, input data size control and installation. This complicates the processes of extending the suite quickly or extracting an application out of it for further, isolated use.

ALPBench [28], [29] is a fully multithreaded benchmark suite derived from the media processing application domain. It consists of five benchmarks: Face recognition, speech recognition, ray tracing and MPEG-2 encode and decode. The parallelization model employed to achieve this is Pthreads. ALPBench additionally incorporates a different style of parallelism exploitation by making extensive use of SSE2 instructions if available. The goal of the benchmark suite was to exemplify how increasingly complex media workloads can be sufficiently parallelized to provide further usability on future multicore platforms. Despite the quality of the programs included, the limitation to media makes ALPBench a suite with different goals than our work. However, it might be considered to include one ore more of their benchmarks into our benchmark suite.

Apart from benchmark suites, previous work also includes attempts particularly targeted at the evaluation of parallel programming models.

Podobas et al. [30] performed an evaluation of three task-based parallel programming models, OpenMP, Cilk++ and Wool. They focus on leveraging the performance characteristics of these parallel programming models, studying in detail the cost of creating, spawning and joining tasks as well as overall performance. For the latter, they use a small set of widely known kernels (FFT, NQueens, Multisort, SparseLU and Strassen). The results are limited to only three programming models, only kernel-type programs and only performance characteristics. Moreover, the work excludes the extension to new programming models and therefore is, in contrast to our work, not portable.

Duran et al. [31] present a similar (kernel-type programs and performance characteristics only) approach. They specifically target the OpenMP tasks programming model, investigating different implementation mechanisms provided by the model (such as tiedness of tasks).

Ravela [32] presents an evaluation of Intel TBB, Pthreads, OpenMP and Cilk++, containing results for both achieved performance and the time required to develop the respective versions of the benchmarks. All used benchmarks are, however, taken from the domain of high performance computing, resulting in limited relevance for different application domains.

Schneider et al. [33] provide an evaluation of three recent parallel programming models, Sequoia, CellGen and the Cell SDK, and for each investigate programming effort and performance in comparison with hand-tuned code. The work's focus is on machines with explicitly managed memory hierarchies, not on shared-memory machines.

V. Conclusions and Future Work

In this paper, we presented a benchmark suite to evaluate the programmability and performance of emerging parallel programming models. To achieve this, we focused on a structured, portability-focused, fixed-parallelism approach. We analyzed the intended usage of the suite, thereby compiling a set of requirements which must be met by a benchmark suite aimed at evaluating parallel programming models. We presented and described an early collection of such benchmarks, covering a wide range of application domains, and used them in case study, comparing an established with an emerging programming model that is actively being developed. In this comparison, we presented results for both performance and programmability. For the latter, we compiled a set of possible categories which must be investigated.

The preliminary experimental results obtained in this process have shown a wide range of characteristics for the chosen benchmark set, especially giving insights about the behavioral properties of those benchmarks and producing valuable information on the scaling and speedup characteristics of the two analyzed programming models.

For two selected benchmarks, we demonstrated that further investigation is needed to fully understand the characteristics we observed. It will, therefore, be the goal of our further research on this topic to classify and examine the behavior of our applications in detail (including kmeans and h264dec). This study must also be extended to additional types of parallel machines, for example heterogeneous architectures or large chip multiprocessors. Such an investigation could also include a detailed analysis of the statistical features of each benchmark, resulting in concrete measures for properties such

as bandwidth usage, arithmetic complexity or memory size requirements.

Naturally, our benchmark suite will be subject to extension. As mentioned in Section IV, porting suitable, existing open-source benchmarks from other collections to this suite is ongoing work. Furthermore, we seek to extend the suite with additional, industry-relevant applications in order to gain key insights on how modern programming models fare when used in large, real-world applications. Benchmarks currently being considered are POV-Ray [34] or a game engine. Aside from adding more benchmarks to the suite, another goal is to evaluate more programming models to gather more experiences in using the suite. Potential additions could be OpenMP [2], a Cilk derivative [4], the Intel Threading Building Blocks [5] or other new, emerging programming models. Evaluating a larger number of programming models is naturally an advantage because it will provide more references to compare to when evaluating new programming models.

For the two programming models already evaluated with the suite, further investigations on the particular threading management overheads, depending on the input data size and specific to each benchmark, will be performed. This can be achieved by changing the timing mechanisms of the benchmarks to separately measure the startup and shutdown phases, thereby comparing the useful execution time to the parallelization overhead. This becomes especially relevant for models targeted at machines connecting cores over a slow network where communication becomes a significant overall execution time fraction.

ACKNOWLEDGMENT

This research has been supported in part by the European Community's Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (www.encore-project.eu), grant agreement n° 248647 [35], and the Future SOC Lab of Hasso-Plattner-Institute Potsdam [36].

REFERENCES

[1] S. Borkar, "Thousand Core Chips: A Technology Perspective," in *Proc. 44th annual Design Automation Conf.*, 2007.

[2] L. Dagum and R. Menon, "OpenMP: A Proposed Industry Standard API for Shared Memory Programming," *IEEE Computing in Science and Engineering*, 1997.

[3] IEEE Opengroup, "Portable Operating System Interface," 2004.

[4] K. H. Randall, "Cilk: Efficient Multithreaded Computing," Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1998.

[5] K. Wooyoung and M. Voss, "Multicore Desktop Programming with Intel Threading Building Blocks," *Software, IEEE*, 2011.

[6] NVIDIA, "CUDA: Compute Unified Device Architecture," 2007, http://developer.nvidia.com/object/gpucomputing.html.

[7] Khronos Group, "OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems," 2009, http://www.khronos.org/opencl/.

[8] J. Perez, R. Badia, and J. Labarta, "A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures," in *Cluster Computing, 2008 IEEE International Conference on*, 2008.

[9] T. Mattson, B. Sanders, and B. Massingill, *Patterns For Parallel Programming*. Pearson Education, 2004.

[10] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," 2006.

[11] J. Tsiombikas, 2010. [Online]. Available: http://www.futuretech.blinkenlights.nl/c-ray.html

[12] R. Rivest, "The MD5 Message-Digest Algorithm," 1992.

[13] S. Gal-On, "Multicore Benchmarks Help Match Programming to Processor Architecture," 2008, initial Presentation.

[14] J. B. MacQueen, "Some Methods for Classification and Analysis of MultiVariate Observations," in *Proc. of the fifth Berkeley Symp. on Mathematical Statistics and Probability*, 1967.

[15] R. Narayanan, B. Özisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "MineBench: A Benchmark Suite for Data Mining Workloads," in *Proc. Int. Symp. on Workload Characterization (IISWC)*, 2006.

[16] C. Ching Chi and B. Juurlink, "A QHD-Capable Parallel H.264 Decoder," in *Proceedings of the 25th Int. Conf. on Supercomputing*, 2011.

[17] FFmpeg group, 2010. [Online]. Available: http://www.ffmpeg.org/ffmpeg.html

[18] D. S. School, "An Overview Of StarSs," 2010. [Online]. Available: www.deisa.eu/Summer-School/StarSs-DEISA-2010-SummerSchool.pdf

[19] J. M. Perez, R. M. Badia, and J. Labarta, "Handling Task Dependencies Under Strided and Aliased References," in *Proceedings of the 24th ACM Int. Conf. on Supercomputing*, 2010.

[20] H. Vandierendonck, P. Pratikakis, and D. Nikolopoulos, "Parallel Programming of General-Purpose Programs Using Task-Based Programming Models," 2011, to be published.

[21] P. Carpenter, A. Ramirez, and E. Ayguade, "Starsscheck: A Tool to Find Errors in Task-Based Parallel Programs," in *Euro-Par 2010 - Parallel Processing*, 2010.

[22] M. Berry, "Public International Benchmarks for Parallel Computers: PARKBENCH Committee: Report-1," *Sci. Program.*, 1994.

[23] Standard Performance Evaluation Corporation, "SPEC Benchmark Suite," http://www.spec.org/index.html.

[24] NASA, "NAS Parallel Benchmarks," http://www.nas.nasa.gov/Resources/Software/npb.html.

[25] R. Jha, R. C. Metzger, B. VanVoorst, L. S. Pires, W. Au, M. Amin, D. A. Castanon, and V. Kumar, "The C31 Parallel Benchmark Suite - Introduction and Preliminary Results," in *Proc. 1996 ACM/IEEE Conf. on Supercomputing*, 1996.

[26] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, 2011.

[27] C. Bienia and K. Li, "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors," in *Proc. 5th Annual Workshop on Modeling, Benchmarking and Simulation*, 2009.

[28] M.-L. Li, R. Sasanka, S. Adve, Y.-K. Chen, and E. Debes, "The ALPBench Benchmark Suite for Complex Multimedia Applications," in *Proc. IEEE Int. Symp. on Workload Characterization (IISWC)*, 2005.

[29] R. Sasanka, M.-L. Li, S. V. Adve, Y.-K. Chen, and E. Debes, "ALP: Efficient Support for all Levels of Parallelism for Complex Media Applications," *ACM Transactions on Architecture and Code Optimization*, 2007.

[30] A. Podobas, M. Brorsson, and K.-F. Faxén, "A Comparison of some recent Task-based Parallel Programming Models," in *Third Workshop on Programmability Issues for Multi-Core Computers*, 2010.

[31] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in *Int. Conf. on Parallel Processing*, 2009.

[32] S. C. Ravela, "Comparison of Shared memory based parallel programming models," Master's thesis, Biekinge Institute of Technology, 2010.

[33] S. Schneider, J.-S. Yeom, B. Rose, J. C. Linford, A. Sandu, and D. S. Nikolopoulos, "A comparison of programming models for multiprocessors with explicitly managed memory hierarchies," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2009.

[34] "Persistence of vision ray tracer," http://www.povray.org/.

[35] Encore Project, "ENabling technologies for a future many-CORE." [Online]. Available: http://www.encore-project.eu/

[36] Hasso-Plattner-Institut Potsdam, "Future SOC Lab," http://www.hpi.uni-potsdam.de/forschung/future_soc_lab.html.