This is the author's version of the work. The definitive work was published in *Proceedings of the 24th International Conference on Architecture of Computing Systems (ARCS), Lake Como, Italy, February 22-25, 2011.*

Frameworks for Multi-core Architectures: A Comprehensive Evaluation using 2D/3D Image Registration

Richard Membarth[1], Frank Hannig[1], Jürgen Teich[1],
Mario Körner[2], and Wieland Eckert[2]

[1] Hardware/Software Co-Design, Department of Computer Science,
University of Erlangen-Nuremberg, Germany.
{richard.membarth,hannig,teich}@cs.fau.de
[2] Siemens Healthcare Sector, H IM AX,
Forchheim, Germany.
{mario.koerner,wieland.eckert}@siemens.com

Abstract. The development of standard processors changed in the last years moving from bigger, more complex, and faster cores to putting several more simple cores onto one chip. This changed also the way programs are written in order to leverage the processing power of multiple cores of the same processor. In the beginning, programmers had to divide and distribute the work by hand to the available cores and to manage threads in order to use more than one core. Today, several frameworks exist to relieve the programmer from such tasks. In this paper, we present five such frameworks for parallelization on shared memory multi-core architectures, namely OpenMP, Cilk++, Threading Building Blocks, RapidMind, and OpenCL. To evaluate these frameworks, a real world application from medical imaging is investigated, the 2D/3D image registration. In an empirical study, a fine-grained data parallel and a coarse-grained task parallel parallelization approach are used to evaluate and estimate different aspects like usability, performance, and overhead of each framework.

Key words: Parallelization, Frameworks, Evaluation, Medical Imaging, 2D/3D Image Registration, OpenMP, Cilk++, Threading Building Blocks, RapidMind, OpenCL.

1 Introduction and Related Work

Multi-core processors have become mainstream within the last years. Today, processors with two, four, or even more cores are ubiquitous in personal computers and even in portable devices such as notebooks and mobile phones. Most times, the massive computing power offered by these architectures, is only used at a very coarse-grained level of parallelism, namely *task-level parallelism* or *application-level parallelism*. That is, the different processor cores are employed by the operating system, more specific by the scheduler, which assigns applications or tasks to the available cores. In order to accelerate the execution of a large number of running tasks, for instance, in a data base server or in network processing, such a work-load distribution might be ideal.

However, if the performance of a single algorithm shall be increased, more fine-grained parallelism (for example, *data parallelism* or *instruction-level parallelism*) has to be considered. In this case, the parallelization cannot be left to the

operating system, but the software developer has to take care of the parallelization by hand or by using appropriate tools and compilers. Here, techniques like *loop unrolling*, *affine loop transformations*, or *loop tiling* [5, 10, 17] are often the agent of choice in order to parallelize an algorithm onto several cores, to adapt to a cache hierarchy, or to achieve a high resource utilization at instruction-level, with the overall goal to speed up the execution of one algorithm (latency minimization) or to achieve a higher data throughput. Arisen from these needs, a number of *parallelization frameworks*[3] have been evolved that assist a developer when writing parallel programs.

In this paper, we present five such frameworks (Section 3) for parallelization on standard shared memory multi-core architectures, namely OpenMP, Cilk++, Threading Building Blocks, RapidMind, and OpenCL. This selection is by no means complete, but concentrated rather on frameworks that are widely used and well recognized than on relatively new academic frameworks with support for only a small range of applications. In a previous study [9], we have shown low level implementation details for some of the herein presented frameworks. The implementation of histogram generation—which is also used within the 2D/3D image registration (Section 2)—was therefore used as an example. In comparison, this study focuses on a more general comparison (Section 4) on a higher abstraction level of the frameworks.

While each of the frameworks has been studied in detail on its own, there is only little related work comparing different parallelization frameworks from a usability perspective. Most studies compare the performance of parallelized applications using different frameworks and compare low level framework details like scheduling [12]. The study most similar to ours from Kegel et al. compares different aspects of Threading Building Blocks to OpenMP [4]. However, their work considers only two frameworks while we give a broader overview of five different frameworks.

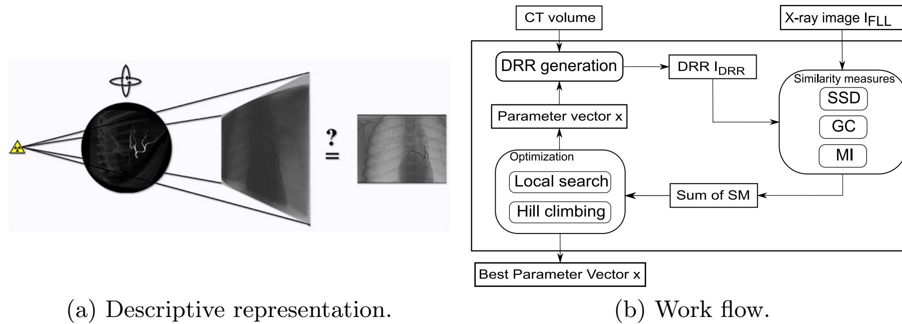
2 2D/3D Image Registration

In medical settings, images of the same modality or different modalities are often needed in order to provide precise diagnoses. However, a meaningful usage of different images is only possible if the images are beforehand correctly aligned. Therefore, an image registration algorithm is deployed.

2.1 Application

In the investigated 2D/3D image registration, a previously stored volume is registered with an X-ray image [6, 16]. The X-ray image results from the attenuation of the X-rays through an object from the source to the detector. Goal of the registration is to align the volume and the image. Therefore, the volume can be translated and rotated according to the three coordinate axes. For such a transformation an artificial X-ray image is generated by iterating through the transformed volume and calculating the attenuated intensity for each pixel.

[3] The term *parallelization framework* is used within this paper as follows: A parallelization framework is a software solution that abstracts from the underlying threading concepts of the operating system and provides the programmer the possibility to express parallelism without worrying about the implementation details.



(a) Descriptive representation. (b) Work flow.

Fig. 1: Descriptive representation and work flow of the 2D/3D image registration.

In order to evaluate the quality of the registration, the reconstructed X-ray image is compared with the original X-ray image using various similarity measures. This process is depicted in Fig. 1(a).

The work flow of the complete 2D/3D image registration as shown in Fig. 1(b) consists of two major computationally intensive parts: Firstly, a digitally reconstructed radiograph (DRR) is generated according to the transformation vector $\mathbf{x} = (t_x, t_y, t_z, r_x, r_y, r_z)$ describing the translation in millimeters along the axes and the rotation according to the Rodrigues vector [15]. *Ray casting* is used to generate the radiograph from the volume, casting one ray for each pixel through the volume. On its way through the volume, the intensity of the ray is attenuated depending on the material it passes. A detailed description with mathematical formulas on how the attenuation is calculated can be found in [6]. Secondly, intensity-based similarity measures are calculated in order to evaluate how well the digitally reconstructed radiograph and the X-ray image match. We consider three similarity measures: sum of square differences (SSD), gradient correlation (GC), and mutual information (MI). These similarity measures are weighted to assess the quality of the current transformation vector $\mathbf{x}$.

To optimally align the two images, optimization techniques are used to find the best transformation vector. Therefore, we use two different optimization techniques. In a first step, *local search* is used to find the best transformation vector evaluating randomly changed transformation vectors. The changes to the transformation vector have to be within a predefined range, so that only transformation vectors similar to the input transformation vector are evaluated. The best of these transformation vectors is used in the second optimization technique, *hill climbing*. In hill climbing only one of the parameters in the transformation vector is changed—once increased by a fixed value and once decreased by the same value. This is done for all parameters and the best transformation vector is taken afterwards for the next evaluation, now, using a smaller value to change the parameters. Once the change to the parameters is below a given threshold, the evaluation stops.

2.2 Parallelization Strategies

For parallelization on the software side, there exist mainly two prominent strategies, namely fine-grained data parallelization and coarse-grained task paralleliza-

tion. Both approaches are applicable for the 2D/3D image registration and described in detail here.

Fine-grained Parallelization describes the parallel processing that is focused on data elements, that is, typically one thread operates on one data element. This means for the 2D/3D image registration that one thread per pixel is used to generate the radiograph in parallel or to calculate the quality measures in parallel. The threads are lightweight and do only little work before they finish compared to coarse-grained parallelization approaches. More precisely, each iteration of a loop processing an image will be executed in parallel.

Coarse-grained Parallelization describes the parallel execution of different tasks. Compared to the fine-grained data parallelism, each thread executes not only few operations on single data elements, but performs typically more complex operations on the whole data set. This means for the 2D/3D image registration that one thread performs the evaluation of one parameter vector $\mathbf{x}$. Different parameter vectors are evaluated in parallel by different threads.

3 Frameworks

The parallelization frameworks are introduced in this section, including a brief background, framework's evolution, and the parallelization approaches taken.

OpenMP. Open Multi-Processing (OpenMP) is a standard that defines an application programming interface to specify shared memory parallelism in C, C++, and Fortran programs [3]. The OpenMP specification is implemented by all major compilers such as Microsoft's compiler for Visual C++ (MSVC), the GNU Compiler Collection (gcc), or Intel's C++ compiler (icc). OpenMP provides preprocessor directives, so called *pragmas* to express parallelism. These pragmas specify which parts of the code should be executed in parallel and how data should be shared. The basic idea behind OpenMP is a fork-join model, where one master thread executes throughout the whole program and forks off threads to process parts of the program in parallel. OpenMP provides different work-sharing constructs, which allow to express task parallelism and data parallelism. The main source of data parallelism are loop programs iterating over a data set, performing the same operation on each element.

Cilk++. Cilk++ [7] is a commercial version of the Cilk [2] language developed at the MIT for multithreaded parallel programming. Cilk++ was recently acquired by Intel and is since then available as Intel Cilk Plus. It is an extension to the C++ language, adding three basic keywords to process loops in parallel, to launch new tasks, and to synchronize between tasks. These keywords allow to express task as well as data parallelism. In addition, Cilk++ provides *hyper-objects*, constructs that avoid data race problems created by parallel access of global variables without locks. For code generation, Cilk++ provides two compilers, one based on MSVC for Windows platforms and one based on gcc for GNU/Linux. Cilk++ provides also tools to detect race conditions and to estimate the achievable speedup and inherent parallelism of *cilkified* programs. Its run-time system implements an efficient work-stealing algorithm that distributes the workload to idle processors.

Threading Building Blocks. Threading Building Blocks (TBB) [14] is a template library for C++ developed by Intel to parallelize programs and is available as an open source version as well as a commercial version providing further support. Parallel code is encapsulated in special classes and invoked from the program. TBB allows to express task and data parallelism. All major compilers can be used to generate binaries from TBB programs. Instead of encapsulating the parallel code in classes, *lambda functions* can be used to express parallelism in a more compact way. There are, however, only few compilers supporting *lambda functions* of the upcoming *c++0x* standard. TBB provides also concurrent container classes for hash maps, vectors, or queues as well as own mutex and lock implementations. The run-time system of TBB schedules the tasks using a work-stealing algorithm similar to Cilk++.

RapidMind. RapidMind [13] is a commercial solution that emerged from a high-level programming language for graphics cards, called Sh [8]. RapidMind was recently acquired by Intel and is integrated into Intel's upcoming Array Building Blocks technology. While Sh was targeting originally only graphics cards, RapidMind takes a data parallel approach that maps well onto many-core hardware as well as on standard shared memory multi-core architectures and the Cell Broadband Engine. RapidMind programs follow the single program, multiple data (SPMD) paradigm where the same function is applied data parallel to all elements of a large data set. These programs use an own language and data types, are compiled at run-time, and called from standard C/C++ code. Through dynamic compilation, the code can be optimized for the underlying hardware at run-time and the same code can be executed on different back ends like standard multi-core processors and graphics cards. All this functionality is provided by libraries and works with the standard compilers on Windows, GNU/Linux, and Mac OS X. RapidMind programs are by design free of deadlocks and race conditions. This is possible, since the changes to the output variables of a RapidMind program are only visible after the program has finished and each output data location can only be written once.

OpenCL. The Open Computing Language (OpenCL) is a standard for programming heterogeneous parallel platforms [11]. OpenCL was initiated by Apple and created and maintained by the Khronos Group. OpenCL is a platform independent specification for parallel programming like OpenGL is for graphics programming. OpenCL supports currently programming standard multi-core processors, graphics cards as well as the Cell Broadband Engine with planned support for other accelerators like DSPs. OpenCL allows to express data and task parallelism. The functionality is provided by a library and OpenCL programs are just-in-time compiled from the run-time environment like in RapidMind. The *kernels* are stored in strings, just like in OpenGL. It is also possible to share resources between OpenCL and OpenGL. The OpenCL standard is implemented by hardware vendors like AMD, IBM, and NVIDIA, but also by operating systems like Apples Mac OS X. All major compilers can be used to link against the OpenCL library.

Framework Comparison. Table 1 compares selected features of the frameworks like their availability, the approach taken to parallelize code, the supported architectures, as well as supported memory architectures.

Table 1: Comparison of selected framework features, including supported memory architecture, availability of the framework, parallelization approach, and supported architectures.

	Memory Architecture		Availability			Approach				Architectures		
	Shared	Distributed	Commercial	OSS	Consortium	Language	Library	Compiler	Comp. Ext.	x86	GPUs	Others
OpenMP	√				√				√	√		√
Cilk++	√		√	(√)		√		√	(√)	√		
TBB	√		√	√			√			√		√
RapidMind	√	√	√			(√)	√	(√)		√	√	√
OpenCL	√	√			√	(√)	√	(√)		√	√	√

4 Framework Evaluation

The evaluation of the frameworks is done in three steps. Firstly, our observations using a certain framework are utilized to assess the framework. Then, the performance and scaling of the implementation on a cluster is compared, and, finally, the overhead incurred for each framework is estimated.

4.1 Usability Evaluation

In an empirical study, the reference implementation written in C/C++ was parallelized using each of the frameworks. For each of the frameworks, two weeks were spent for parallelization. Only for RapidMind and OpenCL more time was spent, since the algorithms had to be rewritten in a different language. During that time the progress on parallelization as well as difficulties, drawbacks, and advantages of each of the frameworks were documented. The relevant aspects common to all frameworks included:

- **Familiarization time**: Time to get familiar with the framework and to understand the parallelization approaches' concepts.
- **Documentation, community**: Available resources to solve problems, for example by an online community.
- **Mapping effort**: Effort to map the reference implementation to the framework. This includes the time required as well as the fraction of the code that has to be changed.

– **Integration**: How well does the framework integrate into the existing environment?
– **Data parallelism**: Support of data parallel processing by the framework.
– **Task parallelism**: Support of task parallel processing by the framework.
– **Image processing**: Special support of the framework for image processing like pipelining of different algorithms.

These criteria are summarized for all frameworks in Table 2. It shows that in particular the frameworks that target also graphics cards require more familiarization time, higher parallelization effort, and provide poor support for task parallelization. However, they provide the best support for image processing and in particular RapidMind provides a concise syntax that matches image processing perfectly. The initialization of OpenCL code is—compared to the other frameworks—notably complicated and requires a lot of different steps.

Although OpenCL supports task parallelism, we were not able to get a task parallel implementation up and running (see Table 3 for used OpenCL version). During execution our task parallel implementation hung, while the same code run in debug mode. Likewise, the data parallel RapidMind implementation yielded wrong results and it was not possible to debug it with all the means at our disposal. Even for the data parallel implementation different iterations of the optimizer with the same parameter set yielded varying results in RapidMind. This could be traced back to the just-in-time compiler. Not using JIT compilation gave us the desired results. Cilk++ integration as suggested by the Programmer's Guide (i. e., using Cilk++ bindings for all functions/files) resulted in a slow program execution. Using only Cilk++ bindings for *cilkified* functions, however, resulted in 30 % faster execution. Having no vibrant online community to narrow down and solve such problems is clearly a drawback for Cilk++ and even more for RapidMind. The only framework that provides support to detect race conditions is Cilk++. Its *Race Detector* executes a cilkified program on one core and reports potential race conditions. Furthermore, Cilk++'s *Scalability Analyzer* provides a profile of parallelism and parallelization overhead of an executed program. This includes also speedup estimates for different core counts.

Table 3 lists the compiler and framework version used for each of the investigated frameworks.

Table 2: Usability evaluation of the frameworks, considering familiarization time, available documentation and community support, mapping effort, integration, as well as support for data parallelism, task parallelism, and image processing.

	Famil. time	Doc. community	Mapping effort	Integration	Data parallel	Task parallel	Image processing
OpenMP	+ +	+ +	+ +	+ +	+ +	+	− −
Cilk++	+ +	−	+ +	−	+ +	+ +	− −
TBB	+	+ +	+	+ +	+ +	+ +	+
RapidMind	− −	− −	− −	−	+ +	− −	+ +
OpenCL	− −	+ +	− −	+	+ +	−	+ +

4.2 Performance Evaluation

For evaluation, a system consisting of four Intel Xeon *Dunnington* CPUs was used, each hosting six cores running at 2.67 GHz. This allowed to use up to 24 cores. Only for OpenCL, a different system had to be chosen due to incompatibility of the OpenCL library with the glibc version on the Dunnington system. Therefore, a system consisting of two Intel *Nehalem-EP* CPUs, each hosting four cores running at 2.66 GHz was used. The CPUs support hyperthreading (simultaneous multithreading), the ability to run two threads per core, and, hence, up to 16 threads in total. For the 2D/3D image registration a fixed number of iterations with the same parameter vector are evaluated. 100 iterations in local search and 60 iterations in hill climbing. This ensures that the same computation is done for each framework.

Figure 2 shows the execution times of the fine-grained implementation of each framework on up to 24 cores for a typical volume size of $256 \times 256 \times 189$ voxels and image resolutions of 320×240 pixels. The times of the OpenCL implementation have been normalized according to the baseline execution times. It can be seen that in particular OpenMP and RapidMind have an enormous initialization overhead when only few threads are used. This effect is even more prevalent for smaller volume and image resolutions, but decreases also with lower resolutions.

On a single multi-core processor with four cores, these frameworks barely match the baseline implementation. With an increasing number of cores, however, these frameworks scale well and catch up with the performance of the other frameworks, notably OpenMP. The framework that scales best and achieves the best results for all problem sizes is TBB. Cilk++ scales just as well, but is slightly slower than TBB. OpenCL shows also good performance, though, only for the middle and large (not shown) volume.

Similarly, Fig. 3 shows the execution times of the coarse-grained implementation of each framework on up to 24 cores. For the reasons mentioned in the usability evaluation, there are no coarse-grained implementations for RapidMind and OpenCL. The graphs show that there are no major differences between the OpenMP, Cilk++, and TBB for the coarse-grained implementation. All frameworks scale well and achieve good performance.

4.3 Overhead Estimation

In order to evaluate and estimate the parallelization overhead of each framework, we calculate the sequential part of the fully parallelized part of the 2D/3D image

Table 3: Compiler version and framework version used for evaluation.

Framework	Version
OpenMP	gcc/4.4.2, OpenMP 3.0
Cilk++	gcc/4.2.4, Cilk++ 1.10
TBB	icc/11.1, TBB 3.0
RapidMind	gcc/4.4.2, RapidMind 4.0.1
OpenCL	gcc/4.4.2, OpenCL 1.0, AMD Stream SDK 2.0.1/2.1

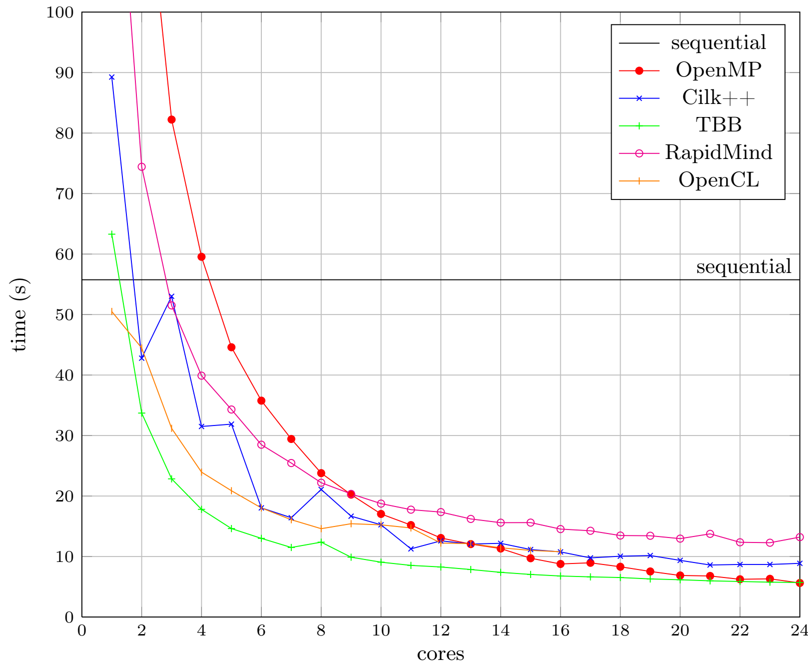


Fig. 2: Execution times of all parallelization frameworks for the fine-grained implementation on up to 24 cores for a volume of $256 \times 256 \times 189$ voxels compared with the sequential implementation.

registration using curve fitting. Therefore, we use the formula of Amdahl's law [1] as in Equation (1):

$$S(n) = \frac{1}{\alpha + \frac{1-\alpha}{n}} \tag{1}$$

The sequential part of the application is denoted by α in the formula and the number of processors by n. Using the measured speedup $S(n)$, α can be estimated using curve fitting. Figure 4 shows exemplarily the measured speedups for the fine-grained and coarse-grained implementations using Cilk++ for one volume size as well as the speedup curve after Amdahl using the alpha value we obtained using curve fitting. It can be seen that the coarse-grained implementation has a much lower sequential part of 1.87 % compared to 13.68 % for the fine-grained implementation. This sequential part is to a large extent caused by synchronization as well as by communication overheads to distribute and share data. Since the implementation strategy for all frameworks was the same, the sequential part should be in the same region, too. Only for RapidMind and OpenCL different approaches had to be used for algorithms with a random scatter patterns like histogram generation. Table 4 shows this sequential part for all investigated frameworks and for different volume sizes. While the sequential part for all coarse-grained implementations is relatively low, it differs significantly for the fine-grained implementations. In particular, TBB has a relatively low over-

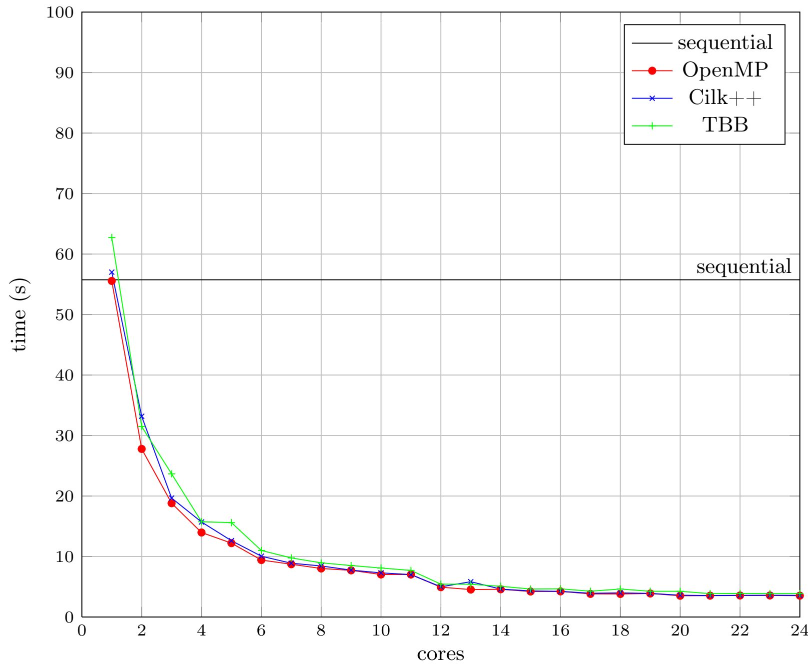


Fig. 3: Execution times of all parallelization frameworks for the coarse-grained implementation on up to 24 cores for a volume of $256 \times 256 \times 189$ voxels compared with the sequential implementation.

head here. The same applies for OpenCL, however, only for the large volume. In particular for the small volume size, huge overheads are involved.

5 Conclusions

In this paper, different parallelization frameworks for standard shared memory multi-core processors have been evaluated. The evaluation criteria were not only limited on pure performance numbers, but also other aspects like usability or

Table 4: Sequential part in % after Amdahl for different volume sizes and parallelization approaches.

Volume	$128 \times 128 \times 94$		$256 \times 256 \times 189$		$512 \times 512 \times 378$	
Approach	data	task	data	task	data	task
OpenMP	24.55 %	2.55 %	10.14 %	1.70 %	7.29 %	1.94 %
Cilk++	30.97 %	2.12 %	13.68 %	1.87 %	8.79 %	2.32 %
TBB	17.93 %	2.85 %	6.48 %	2.58 %	4.04 %	3.90 %
RapidMind	105.66 %	n/a	22.31 %	n/a	8.82 %	n/a
OpenCL	89.88 %	n/a	15.92 %	n/a	3.51 %	n/a

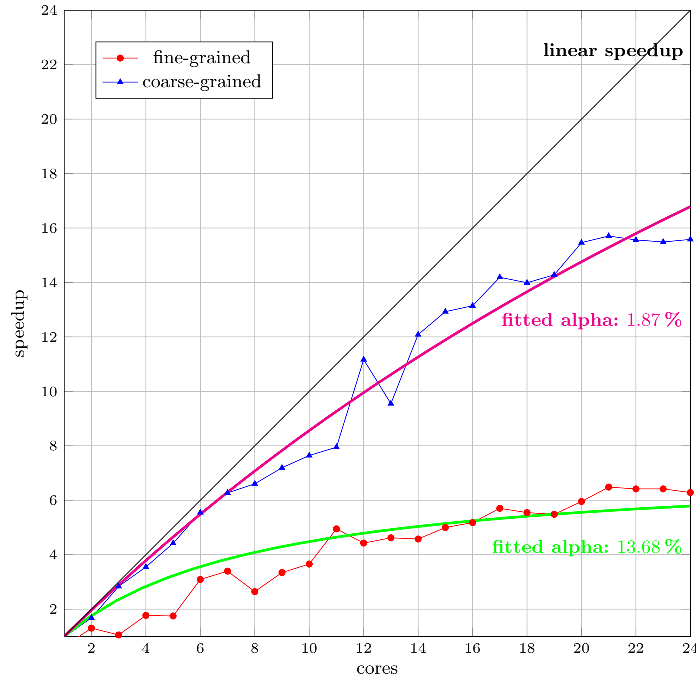


Fig. 4: Speedup of fine-grained and coarse-grained Cilk++ implementation on up to 24 cores for the $256 \times 256 \times 189$ volume compared with the sequential implementation.

productivity of the frameworks were considered. For evaluation, a computationally intensive application from medical imaging, 2D/3D image registration, has been chosen.

Two parallelization strategies were evaluated, namely fine-grained data parallelism and coarse-grained task parallelism. From the considered frameworks, only OpenMP, Cilk++, and Threading Building Blocks support both approaches, while only the fine-grained data parallelism could be realized using RapidMind and OpenCL. While for the coarse-grained approach all frameworks yield equally good results, the overhead for the fine-grained approach differs a lot. OpenMP and RapidMind come along with much overhead for a small number of cores, while the other frameworks have only little overhead here. The best results for the fine-grained approach were obtained using TBB and OpenMP, followed by Cilk++. OpenCL was outstanding, though, only for large problem sizes. In terms of productivity, OpenMP and Cilk++ required only little modifications of the source code, and Threading Building Blocks major restructuring of the source code. In contrast, RapidMind and OpenCL required a complete restructuring of the source code and also the algorithm had to be expressed differently.

The *best* framework depends on the existing environment, the application domain and also on political decisions. Therefore, we will not give precedence to any of the frameworks, but we hope to give others assistance when they have to chose the framework appropriate to their needs.

Acknowledgments. We are indebted to the RRZE (Regional Computing Center Erlangen) and their HPC team for granting computational resources and providing access to preproduction hardware.

References

1. Amdahl, G.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: Proceedings of the AFIPS Spring Joint Computing Conference. pp. 483–485. ACM (1967)
2. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. ACM SigPlan Notices 30(8), 207–216 (1995)
3. Dagum, L., Menon, R.: OpenMP: An Industry Standard API for Shared-memory Programming. Computational Science & Engineering, IEEE 5(1), 46–55 (2002)
4. Kegel, P., Schellmann, M., Gorlatch, S.: Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-cores. Euro-Par 2009 Parallel Processing pp. 654–665 (2009)
5. Kejariwal, A., Nicolau, A., Banerjee, U., Veidenbaum, A., Polychronopoulos, C.: Cache-Aware Iteration Space Partitioning. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). pp. 269–270. ACM, Salt Lake City, UT, USA (2008)
6. Kubias, A., Deinzer, F., Feldmann, T., Paulus, S., Paulus, D., Schreiber, B., Brunner, T.: 2D/3D Image Registration on the GPU. International Journal of Pattern Recognition and Image Analysis 18(3), 381–389 (2008)
7. Leiserson, C.: The Cilk++ Concurrency Platform. In: Proceedings of the 46th Annual Design Automation Conference. pp. 522–527. ACM (2009)
8. McCool, M., Du Toit, S.: Metaprogramming GPUs with Sh. AK Peters, Ltd. (2004)
9. Membarth, R., Hannig, F., Teich, J., Körner, M., Eckert, W.: Comparison of Parallelization Frameworks for Shared Memory Multi-Core Architectures. In: Proceedings of the Embedded World Conference. Nuremberg, Germany (Mar 2010)
10. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
11. Munshi, A.: The OpenCL Specification. Khronos OpenCL Working Group (2009)
12. Olivier, S., Prins, J.: Comparison of OpenMP 3.0 and Other Task Parallel Frameworks on Unbalanced Task Graphs. International Journal of Parallel Programming pp. 1–20 (2010)
13. RapidMind: RapidMind Development Platform Documentation. RapidMind Inc. (June 2009)
14. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media, Inc. (2007)
15. Trucco, E., Verri, A.: Introductory Techniques for 3-D Computer Vision. Prentice Hall New Jersey (1998)
16. Weese, J., Penney, G., Desmedt, P., Buzug, T., Hill, D., Hawkes, D.: Voxel-Based 2-D/3-D Registration of Fluoroscopy Images and CT Scans for Image-Guided Surgery. IEEE Transactions on Information Technology in Biomedicine 1(4), 284–293 (1997)
17. Wolfe, M.: High Performance Compilers for Parallel Computing. Addison-Wesley (1996)