A Methodology for Porting Sequential Software to the Multicore Platform
Considering Technical and Economical Aspects of Software Parallelization

Constantin Christmann, Jürgen Falkner and Anette Weisbecker
Fraunhofer Institut für Arbeitswirtschaft und Organisation IAO, Nobelstraße 12, 70569 Stuttgart, Germany

Keywords: Software Development, Parallelization, Multicore Platform, Economical Aspects, Auto Tuning.

Abstract: Today multicore processors are ubiquitous in desktop computers, servers and various other devices. In order to take advantage of such multicore processors many of today’s existing applications, which typically are sequential applications, need to be ported to the multicore platform. However, the adoption of multicore technology in software applications is still restrained by technical and economical obstacles. The contribution of this paper is a methodology for porting sequential software to the multicore platform. It takes into account the technical specifics of parallel programming and multicore technology offering developers orientation during the porting process. In addition to that (and in contrast to existing methodologies) it also addresses the economical obstacles of multicore adoption in software development by (1) supporting planning and cost control to counteract high development costs and by (2) utilizing auto tuning in order to cope with uncertainty due to varying processor architectures.

1 INTRODUCTION

Due to diminishing returns of traditional techniques in processor design recent developments showed a clear trend towards multicore architectures, where two or more individual processor cores are integrated on a single chip (Borkar and Chien, 2011). Today multicore processors are ubiquitous in desktop computers, servers and also mobile devices like smartphones or tablets (UBM Tech, 2011; Jainschigg, 2012).

However, the increasing performance of such multicore processors can only be exploited by software applications that take advantage of parallelism (Pankratius and Tichy, 2008; Rauber and Rünger, 2012). The consequence is that new software applications should be designed and developed as parallel programs right from the start. Furthermore, many existing applications have to be adapted to this new paradigm, due to the fact, that with an increasing number of available cores the compute intensive parts of a sequential application might lose a growing factor compared to a parallelized implementation (Singler and Konsik, 2008; Creeger, 2005).

Today the adoption of multicore technology in software applications is still restrained by various obstacles (Christmann, Hebisch and Strauß, 2012a). On the one hand there exist technical challenges of adopting the technology: many developers do not have the necessary know-how to use existing parallel programming environments effectively, developers are in need of adequate tools supporting the different tasks of parallel programming and software engineering methods are required which give developers orientation regarding parallel software development.

On the other hand there exist obstacles which originate more in an economical perspective:

- High development costs: Parallel programming is generally associated with a higher development effort than the development of sequential programs (Hochstein et al., 2005). This higher effort has a direct effect on the development cost. Furthermore the parallel development is more costly due to the special expertise which is needed by developers (Diggins, 2009).
- Varying Architectures: There exists a high uncertainty regarding many aspects of the multicore processor design. Examples are the

Christmann C., Falkner J. and Weisbecker A..
A Methodology for Porting Sequential Software to the Multicore Platform - Considering Technical and Economical Aspects of Software Parallelization.
DOI: 10.5220/0004988505510559
In *Proceedings of the 9th International Conference on Software Engineering and Applications* (ICSOFT-EA-2014), pages 551-559
ISBN: 978-989-758-036-9
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

number of hardware threads per core, the organization of caches or heterogeneous processor cores (Sodan et al., 2010; Borkar and Chien, 2011). The optimization of a parallel program for a specific architecture may have a negative effect on performance on a slightly different system (Karcher, Schaefer and Pankratius, 2009). So, in order to support different processors multiple optimizations must be established and maintained, which increases the complexity of the development process.

Such an economical perspective is highly relevant in the context of parallelization. This is pointed out by the results of a survey of 254 software developers where 30 percent of the participants are seeking *efficient* techniques for porting sequential applications to parallel (Jainschigg, 2012). Albeit this relevance, existing methodologies for porting of sequential applications mostly focus on the technical obstacles mentioned above, leaving out the economic aspects (see Section 2).

Thus, the contribution of this paper is a methodology for porting sequential software to the multicore platform which (in contrast to existing methodologies) also addresses the economical obstacles of multicore adoption in software development by (1) supporting planning and cost control to counteract high development costs and by (2) utilizing auto tuning in order to cope with uncertainty due to varying processor architectures.

The remainder of this paper is structured as follows: Section 2 presents the related work. In Section 3 the design considerations are described that form the foundation for the development of the presented methodology. In Section 4 the methodology is described in detail and Section 5 presents the results of applying the methodology to the porting of an image processing application. Section 6 contains a detailed comparison between the manual parallelization approach used by the methodology and the classical approach of existing parallelization methodologies. Section 7 concludes with a discussion and an outlook on upcoming research activities.

2 RELATED WORK

The porting of an existing sequential application is related to software development in general and more specifically to the topics software maintenance and migration. For these topics various methodologies and approaches are available, which in general do rather focus on processes than technical details; hence, they do not address the specifics of multicore technology and parallelization.

A methodology presented by Christmann, Hebisch and Strauß (2012b) acts as a link between the general software development process and the specifics of parallel programming. This was achieved by describing central activities which need to be incorporated in the classical development process in order to develop software for the multicore platform.

Regarding the specifics of software parallelization more literature can be found as this is a classical research area in high performance computing (HPC) that now is foregrounded due to the increasing proliferation of multicore processors. Most methodologies focus either on the shared memory model or on message passing, which both are the dominating models for parallel programming nowadays.

A method for the message passing model was presented by Ramanujam and Sadayappan (1989), which allows the parallelization of nested loops. Foster (1995) described the abstract steps for developing a parallel algorithm - also with focus on message passing. Sundar et al. (1999) presented a step-wise parallelization for sequential vector programs using the Message Passing Interface (MPI).

Other methodologies address the shared memory programming model: One is the methodology described by Park (2000), which supports the parallelization of a whole program while fostering the utilization of various tools within this process. Intel (2003) and Tovinkere (2006) also cover the parallelization of a full sequential program with focus on the shared memory model. Both methods utilize specific (and somewhat out dated) Intel tools; however, the individual steps of the methodology as well as the utilization of different tools are still relevant today. Donald and Martonosi (2006) describe a method for the parallelization of a specific simulation code.

Addressing the shared memory model as well as message passing, Mattson, Sanders and Massingill (2004) created a set of design patterns covering different aspects of parallel programming (i.e. decomposition, organization of dependencies, …).

In Table 1 these methodologies are compared. The criterions for comparison are:

- Independence of a specific domain

- Support of the shared memory model as well as the message passing model
- Addressing of higher development costs and uncertainty due to varying architectures

Table 1: Comparison of existing methodologies.

	Domain Independence	Shared Memory	Message Passing	High Development Costs	Varying Architectures
Ramanujam and Sadayappan (1989)	◑	○	●	○	○
Foster (1995)	●	○	●	◔	○
Sundar et al. (1999)	○	○	●	◑	○
Park (2000)	●	●	○	◑	○
Mattson, Sanders and Massingill (2004)	●	●	●	◔	○
Intel (2003)	●	●	○	◑	○
Tovinkere (2006)	●	●	○	◑	○
Donald and Martonosi (2006)	○	●	○	◑	○
Christmann, Hebisch and Strauß (2012b)	●	●	○	◑	●

Legend: ● applies ◕ applies mostly ◑ applies partly ◔ applies slightly ○ does not apply

As the comparison shows, most methods only support one of the two predominant programming models. Regarding the development costs some methods just give the advice to developers to consider complexity and software engineering efforts when making decisions regarding the manual parallelization (Foster, 1995; Mattson, Sanders and Massingill, 2004). Other methods do focus development efforts on compute intensive parts of the program (Park, 2000; Tovinkere, 2006; Intel, 2003; Christmann, Hebisch and Strauß, 2012b). Donald and Martonosi (2006) achieve a simplified parallelization by exploiting characteristics of the specific application domain and Sundar et al. (1999) reduce the complexity of the parallelization by addressing data partitioning issues and communication issues in separate development stages. However, none of these methods allows a planning of the parallelization process while taking into account the development costs of alternatives.

Also, varying processor architectures are considered unsatisfactorily - in (Christmann, Hebisch and Strauß, 2012b) this aspect is addressed by integrating auto tuning into the development process. But to sum up, none of these methods fully counteracts the economic obstacles described in the introduction.

3 DESIGN CONSIDERATIONS

The following considerations served as foundation for the development of the methodology:

- Parallelization: To achieve a parallelization without any manual effort the methodology should consider the utilization of a parallelizing compiler - if such a compiler is available for the given programming environment. However, satisfactory performance is often not possible using a parallelizing compiler alone, so an (additional) manual parallelization will be necessary also (Pankratius and Tichy, 2008; Asanovic et al., 2009).
- Planning: In order to support planning and cost control during a complex and time-intensive porting process the methodology should be organized as a phase model (Ludewig and Lichter, 2007) with well-defined results for each individual phase. Furthermore, regarding the manual parallelization an optimization method (Christmann, Falkner and Weisbecker, 2012) should be integrated in the methodology.
- Auto tuning: Due to the heterogeneity of processor architectures an individual adaption of the parallel program may be necessary. To minimize the manual effort the methodology should utilize an automatic tuning for this. Besides the advantage of reducing the manual effort this also may lead to better performance, as non-intuitive parameter combinations may be reached (Asanvoic et al., 2006).

4 THE METHODODLOGY

The methodology divides the process of porting a sequential application to the multicore platform into four phases. The individual phases will be described in the next sections.

4.1 Phase 1: Preparation

The objective of the first phase is the **selection and preparation of a suitable programming environment** for the parallelization of the sequential program.

The applicability of a parallel programming environment is highly dependent on the existing program code of the sequential program. Furthermore, the methodology does require certain tools (mandatory: timer, profiler, parallel debugger, auto tuner; optional: parallelizing compiler, regression test tool, analysis tools), so the availability of these tools is also an important constraint. After selecting a suitable programming environment the provisioning of this environment (including the required tools) must take place.

In particular, if a parallelizing compiler is available for the programming environment and if a test shows that it does not have a negative (but possibly positive) effect on the performance it should be utilized for compilation.

4.2 Phase 2: Analysis

The objective of the second phase is the detailed analysis of the sequential program and based on this analysis the **determination of an optimal strategy for the manual parallelization**.

The underlying approach of this phase is based on the optimization method for the manual parallelization presented in (Christmann, Falkner and Weisbecker, 2012). The method perceives a manual parallelization as a combination of one or more local parallelizations in individual partitions of the sequential program code and allows the estimation of overall implementation effort and speedup based on such local parallelizations. These estimates are then used to determine an optimal combination of local parallelizations based on the economic principle (Kampmann and Walter, 2009). Following this optimization method this phase is divided into three steps (see Figure 1):

1. **Initialization:** First the partitions of the program code which contribute significantly to the execution time of the program are identified using a profiler.
2. **Analysis of partitions:** Then each of these compute intensive partitions is analyzed regarding potentials for local parallelization. This involves the identification of dependencies relevant for parallelization as well as the creative process of identifying and documenting parallelization opportunities (regarding the identification of such parallelization opportunities see i.e. Foster, 1995; Mattson, Sanders and Massingill, 2004; Rauber and Rünger, 2012). In particular for each opportunity the expected speedup as well as the associated development effort must be methodically estimated and documented.

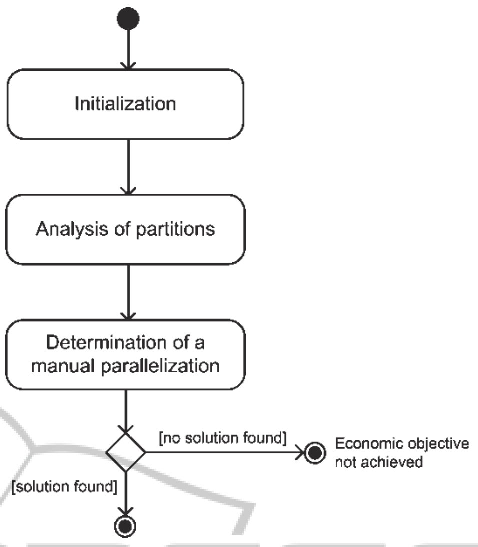
Initialization
Analysis of partitions
Determination of a manual parallelization
[no solution found]
Economic objective not achieved
[solution found]

Figure 1: Activity diagram of the analysis phase.

3. **Analysis of partitions:** Then each of these compute intensive partitions is analyzed regarding potentials for local parallelization. This involves the identification of dependencies relevant for parallelization as well as the creative process of identifying and documenting parallelization opportunities (regarding the identification of such parallelization opportunities see i.e. Foster, 1995; Mattson, Sanders and Massingill, 2004; Rauber and Rünger, 2012). In particular for each opportunity the expected speedup as well as the associated development effort must be methodically estimated and documented.
4. **Determination of a manual parallelization:** Then one must decide which economic objective should be applied to this optimization of the manual parallelization:
   - Speedup maximization: Achieving a maximal speedup while not exceeding a certain budget of development hours A^*.

- Effort minimization: Achieving a certain speedup S^* while requiring the fewest number of development hours.

As described in (Christmann, Falkner and Weisbecker, 2012) the underlying optimization problem of selecting ideal parallelization opportunities within *independent subsets* of program partitions can be formulated as multiple choice knapsack problem (MCKP) (Kellerer, Pferschy and Pisinger, 2010). Hence, for each independent subset the MCKP must be solved and the optimal solution over all independent subsets must be selected. The result of this optimization is the basis for the following project decision:

- If a solution exists then it does represent the optimal manual parallelization with regard to the chosen economic objective.
- Otherwise, if no solution can be found, the project must be aborted due to the expectation that the economic objective cannot be accomplished.

4.3 Phase 3: Implementation

In this phase the **individual local parallelizations become implemented** which are the result of the optimization in the previous analysis phase.

The implementation of a single local parallelization is further divided into multiple steps (see Figure 2):

1. **Implementation:** The first step involves the implementation of the local parallelization following the documentation created in the previous analysis phase.
2. **Test and debugging:** The objective of this step is to make sure that the original functionality of the program is ensured. As with the sequential program a program version exists, which produces correct results, regression tests (Frühauf, Ludewig and Sandmayr, 2004) can be used for program verification. If testing does indicate errors this errors must be resolved. The identification of causes for errors introduced by parallelization (like race conditions or dead locks) is typically not a simple task but special analysis tools for automatic error detection can help. However, if such analysis tools cannot help to determine all errors then the program execution must be retraced using a debugger suited for debugging parallel programs until all causes for errors are identified and resolved.
3. **Evaluation:** This step involves the evaluation and if necessary the optimization of the local parallelization. Therefore, the overall execution of the program should be measured to find out if the recent parallelization had a positive effect on the execution time in the first place. Also an analysis tool can be utilized for judging if the parallel execution of the local parallelization does perform as expected.
4. **Rollback:** If the local parallelization does not improve the execution time of the program then the sequential program flow in this part of the program must be restored.

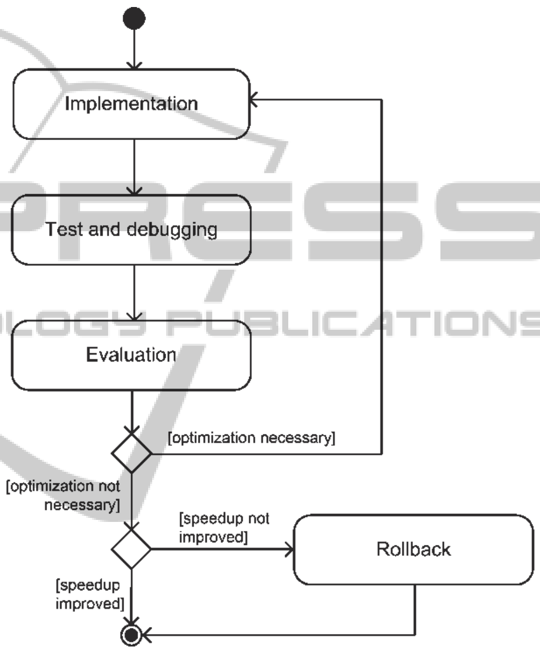

Figure 2: Activity diagram of the implementation phase.

4.4 Phase 4: Adaption

The objective of the last phase is to **optimally adapt the parallel program to one or more target systems**. Such adaption can be done manually; however, as this might be associated with a high development effort, it is intended by the methodology to utilize auto tuning for this task. First, the auto tuning must be integrating the parallel program. Afterwards, the tuning must be performed for every target system. The result is a set of one or more parameter configurations, which optimally adjust the program to the various target systems.

5 APPLICATION

The presented methodology was applied to the porting of OpenJPEG, which is an open source implementation of the JPEG 2000 standard. The scenario we used for benchmarking was the encoding of a given set of bitmap images.

The economic objective for the parallelization was to achieve a maximum speedup within a given time frame of one week (ca. 40 developer hours). The whole porting was performed by a single developer who already had some experience in parallel programming.

As primary system during the manual parallelization a server with Dual Clovertown processor (2.33 GHz, 64 bit) with 8 cores was used. The second system we wanted to support was a laptop with Intel Mobile Core 2 Duo processor (2.40 GHz, 32 bit) equipped with 2 cores. On both target systems Windows 7 was installed as operating system.

After selecting a suitable parallel programming environment (Visual Studio 2010, Intel C/C++ Compiler XE, OpenMP) and preparing optional and mandatory tools the analysis of the sequential code was conducted following the steps described in Section 4.2: At first a profiling run using the sampling profiler of Visual Studio did point out 11 partitions in the code which were significantly compute intensive. These partitions were analyzed regarding parallelization opportunities. This detailed analysis of partitions was rather time-consuming and did require a total of 19 developer hours. In the final step of the analysis an optimal manual parallelization was chosen which maximized the speedup under the constraint of the remaining developer hours.

This manual parallelization was implemented which took slightly more than the expected 4 hours. After some analysis and slight optimization of the implementation a speedup of 1.95 was achieved (the expectation was a speedup around 2.2). While this speedup did not meet the expectation it does not necessarily indicate a wrong decision regarding the manual parallelization – due to the fact that estimates for other manual parallelizations contain the same potential for deviation than the chosen parallelization.

Thereafter, the implementation was adapted to the two target systems using auto tuning. More specifically different parameters of the parallel implementation were adjusted dynamically by a genetic algorithm (Goldberg, 1989) based on the execution time of many repeated program runs. The parameter optimization of the auto tuning further reduced the execution time so that a final speedup of 2.06 was realized.

After adapting the program to the server system the auto tuning was also conducted for the mobile system. Worth mentioning is that at first the execution time of the parallel program on the mobile system was significantly worse than the execution time of the sequential implementation – this shows how the performance of a given parallelization depends on the architecture it becomes executed on and how an individual optimization for varying systems is necessary. After applying the auto tuning a final speedup of 1.21 was achieved - without requiring any manual analysis or optimization of the implementation. Figure 3 gives an overview how the execution time of the application was improved for both systems over the different phases of the methodology.

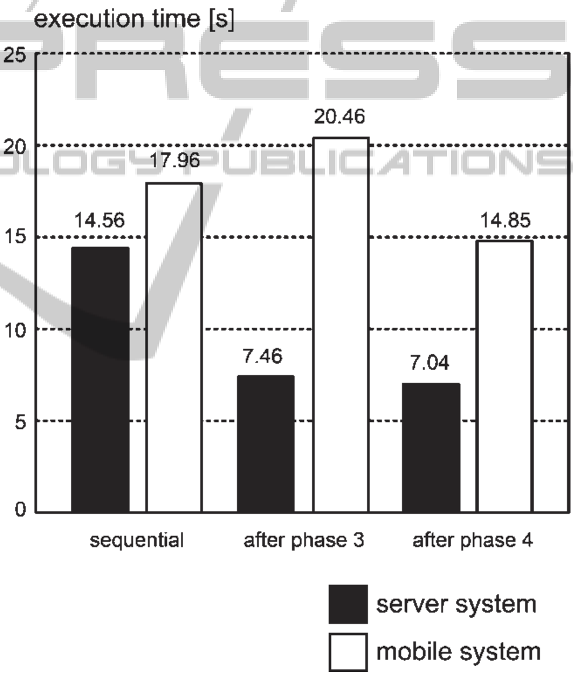

Figure 3: Improvement of the execution time for both systems over the different phases of the methodology.

6 EVALUATION OF THE MANUAL PARALLELIZATION

Subject of this section is the comparison between the manual parallelization approach used by our methodology and the approach of existing parallelization methodologies. In particular the focus of this comparison lies on the ability to support planning activities and to consider an economic objective/constraint regarding the manual parallelization.

In general many of the existing methodologies from Section 2 do only cover the technical aspects of parallelizing a single algorithm or a given part of the program. Some methods (Park, 2000; Intel, 2003; Tovinkere, 2006; Christmann, Hebisch and Strauß, 2012b) help the programmer to focus development efforts onto promising parts of the program. The underlying approach of these methods can be generalized in the following form, which we will denote as the *classical approach*:

1. **Identification of partitions:** First the partitions of the program code which contribute significantly to the execution time of the program are identified. The following steps 2 to 4 are traversed for each of these compute intensive parts individually. The sequence is determined by the share of the execution time a partition has, so that the one with the largest share is parallelized first.
2. **Analysis of the partition:** The current partition is analyzed regarding opportunities for local parallelizations.
3. **Determination of a local parallelization:** Then a local parallelization is chosen – typically the one which maximizes the speedup in the partition.
4. **Implementation of the local parallelization:** The local parallelization becomes implemented, tested and eventually optimized. Then steps 2 to 4 are traversed for the next partition.

An abort criterion for this successive parallelization is not explicitly declared by any methodology but we assume the following: Of course the parallelization ends if steps 2 to 4 have been traversed for all compute intensive partitions. Furthermore, due to the economic principle (Kampmann and Walter, 2009) we can assume that the cycle continues until either a demanded speedup for the program has been accomplished (effort minimization) or the available budget for the parallelization has been used up (speedup maximization).

The first step of the approach used by our methodology (which will be denoted in the following as *analyzing approach*) and the classical approach is very similar. As well is the creative process of analyzing a partition the same for both approaches. A first difference is that in the analyzing approach the development effort for every local parallelization must be estimated methodically whereas the classical approach has no such requirement. However, the central difference is that in the classical approach the decision for implementing a local parallelization is made subsequently to the analysis of an individual partition whereas in the analyzing approach at first all compute intensive partitions are analyzed and the selection of one or more local parallelizations is based on this exhaustive analysis.

Both approaches have their strengths and weaknesses. The advantage of the classical approach is that the selection of a local parallelization only requires a simple comparison of the parallelization opportunities in the current partition. Furthermore, in the best case only partitions which actually get parallelized are analyzed, due to the fact that the decision for analyzing the next partition depends on the parallelization objective being already accomplished or not.

In contrast to that does the presented methodology follow the approach of selecting a manual parallelization for the whole program after performing an exhaustive analysis of possibly multiple partitions. The overall analysis effort of the analyzing approach depends directly on the threshold for rating partitions as being significantly compute intensive in step 1 of the analysis phase. Hence, this effort can be higher than the effort spent for analysis when following the classical approach. On the other hand, if this step leads to more partitions being analyzed than in the classical approach then the data basis for finding an optimal parallelization is larger, too – it comprises at least the same parallelization opportunities which would be identified when following the classical approach.

Hence, the manual parallelization determined in step 3 of the analysis phase is at least the same or possibly even better regarding the given economic objective as the parallelization that becomes implemented by the classical approach. Another advantage is that in our methodology a cancellation of the project is possible in an early stage of the project. And if the project continues after the analysis phase the estimates regarding speedup and effort of the selected manual parallelization can be used for the further project planning.

7 CONCLUSION & OUTLOOK

The results of applying our methodology to some real world code (see Section 5) show that the methodology is suitable for porting sequential software to the multicore platform. Furthermore, the results show that following the methodology it also was possible to comply with a given economic objective as well as achieving a significant speedup on two varying target systems without much additional development effort – hence, the methodology did help to address and overcome the economic obstacles associated with the scenario.

In Section 6 we have carved out similarities as well as fundamental differences between the analyzing approach used by our methodology and the classical approach used by many existing methodologies. The comparison showed how our approach allows project planning and cost control and how in some cases this even can lead to better quality (higher speedup/less development effort) for the manual parallelization.

As both approaches do have pros and cons the question arises, under which circumstances each of the approaches is suitable the most? Due to the higher complexity we think the approach of our methodology might be better suited for rather large porting projects, which can benefit the most from better planning capabilities and cost control. In contrast to that the more "ad hoc" doing of the classical approach might be the right choice for rather small porting projects. However, despite this argumentation this is still an open question that we want to pursue. Furthermore, we intend to apply our methodology to upcoming porting projects. Based on these experiences we want to develop the methodology further to achieve an optimal use in practice.

REFERENCES

Asanovic, K. et al., 2006. The landscape of parallel computing research: a view from Berkeley. University of California at Berkeley, Technical report.

Asanovic, K. et al., 2009. A view of the parallel computing landscape. In *Commun. ACM*, Vol. 52, No. 10, pp. 56-67.

Borkar, S., Chien, A. A., 2011. The future of microprocessors. In *Commun. ACM*, Vol. 54, No. 5, pp. 67-77.

Christmann, C., Falkner, J., Weisbecker, A., 2012. Optimizing the efficiency of the manual parallelization. In *Int. Conf. on Software & Systems Engineering and their Applications (ICSSEA 2012)*.

Christmann, C., Hebisch, E., Strauß, O., 2012a. *Einsatzszenarien für die multicore-technologie*, Fraunhofer Verlag. Stuttgart.

Christmann, C., Hebisch, E., Strauß, O., 2012b. *Vorgehensweise für die multicore-softwareentwicklung*, Fraunhofer Verlag. Stuttgart.

Creeger, M., 2005. Multicore cpus for the masses. In *ACM Queue*, Vol. 3, No. 7, pp. 64 ff.

Diggins, C., 2009. *Three reasons for moving to multicore*. Dr. Dobb's Journal. www.drdobbs.com/parallel/216200386 [2014-02-25].

Donald, J., Martonosi, M., 2006. An efficient, practical parallelization methodology for multicore architecture simulation. In *IEEE Computer Achitecture Letters*, Vol. 5, No. 2, pp. 14 ff.

Foster, I., 1995. *Designing and building parallel programs: concepts and tools for parallel software engineering*, Addison-Wesley. Boston.

Frühauf, K., Ludewig, J., Sandmayr, H., 2004. *Software-prüfung – eine anleitung zum test und zur inspektion*, vdf Hochschulverlag. Zurich, 5th edition.

Goldberg, D. E., 1989. *Genetic algorithms in search, optimization, and machine learning*, Addison-Wesley. Reading.

Grötker, T. et al., 2008. *The developer's guide to debugging*, Springer. Berlin.

Hochstein, L. et al., 2005. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. In *Proc. of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society. Washington, pp. 35 ff.

Intel, 2003. *Threading methodology: principles and practices*. Intel Corp., White paper.

Jainschigg, J., 2012. *Parallel programming: goals, skills, platforms, markets, languages*. Slashdot Media, Report.

Kampmann, R., Walter, J., 2009. *Mikroökonomie: markt, wirtschaftsordnung, wettbewerb*, Oldenbourg. Munich.

Karcher, T., Schaefer, C., Pankratius, V., 2009. Auto-tuning support for manycore applications: perspectives for operating systems and compilers. In *ACM SIGOPS Operating Systems Review*, Vol. 43, No. 2, pp. 96-97.

Kellerer, H., Pferschy, U., Pisinger, D., 2010. *Knapsack problems*, Springer. Berlin.

Ludewig, J., Lichter, H., 2007. *Software-engineering: grundlagen, menschen, prozesse, techniken*, Dpunkt. Heidelberg, 1st edition.

Mattson, T. G., Sanders, B. A., Massingill, B. L., 2004. *Patterns for parallel programming*, Addison-Wesley. Boston.

Park, I., 2000. *Parallel programming methodology and environment for the shared memory programming model*. Purdue University, PhD thesis.

Pankratius, V., Tichy, W. F., 2008. Die Multicore-Revolution und ihre Bedeutung für die Softwareentwicklung. In *Objektspektrum*, No. 4, pp. 30-32.

Ramanujam, J., Sadayappan, P., 1989. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proc. 1989 ACM/IEEE Conf. on Supercomputing*. ACM. New York, pp. 637-646.

Rauber, T., Rünger, G., 2012. *Parallele Programmierung*, Springer. Berlin, 3rd edition.

Singler J., Konsik, B., 2008. The gnu libstdc++ parallel mode: software engineering considerations. In *Proc. 1st Int. Workshop on Multicore Software Engineering (IWMSE '08)*. ACM. New York, pp. 15-22.

Sodan, A. C. et al., 2010. Parallelism via multithreaded and multicore cpus. In *IEEE Computer*, Vol. 43, No. 3, pp. 24-32.

Sundar, N. S. et al., 1999. An incremental methodology for parallelizing legacy stencil codes on message-passing computers. In *Proc. 1999 Int. Conf. on Parallel Processing (ICPP '99)*. IEEE Computer Society. Washington, pp. 302-310.

Tovinkere, V., 2006. *A methodology for threading serial applications*. Intel Corp., White paper.

UBM Tech, 2011. *The state of parallel programming*. Report.