

Software Engineering in the Era of Parallelism

Victor Pankratius

Multicore Software Engineering Young Investigator Group
Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
pankratius@ipd.uka.de
<http://www.victorpankratius.com>

Abstract. We are in the era of parallelism: Multicore processors with several cores on the same chip are standard. Fundamental changes in mainstream software development are required, because parallelism is now the key to better performance on every PC, laptop, or embedded device. Every performance-critical application – not just supercomputing applications – has to be parallel in order to exploit the hardware potential. The great challenge is now how to make parallel programming easier for average programmers. This paper presents an overview of my research directions, thoughts, and visions for the field of multicore software engineering, to make parallelism accessible to a large number of developers, scalable, portable, and applicable in novel scenarios. In particular, I will highlight contributions to auto-tuning, programming models, debugging techniques, as well as insights from empirical studies, working towards the goal of making parallel programming easier.

1 Parallelism Changes the Game

The computer science community agrees that we are at an inflection point [1]: Parallelism is available on every desktop at low cost. Every PC, laptop, or embedded device is a truly parallel machine. Regular processors have 4, 8, or 12 cores. Intel presented the Single Chip Cloud Computer prototype with 48 cores on one chip; CISCO had already in 2005 a packet processing chip with 192 cores. Recent graphics cards have over 300 cores that can be used on every PC to speed up data parallel computations. Looking at these developments, it is not hard to believe projections that mainstream processors will soon have hundreds of cores on one chip. We need to take advantage of this horsepower!

In the past, parallel computers used to be expensive and typically accessible to scientists only. Why are they now affordable for everyone? Processor clock rate increases have reached technical limits, mainly due to heat and power consumption, as shown in Figure 1 (a). We have to go parallel in hardware to increase performance; this is possible because the number of transistors on the same chip can still grow.

What is different from parallel computing in the past? It's not only the ubiquity of parallel hardware, but also some other subtle improvements. Multicore processors share cores as well as other resources, such as cache memory and

--

Published in Victor Pankratius, Samuel Kounev (Eds.),
"Emerging Research Directions in Computer Science. Contributions from the Young Informatics
Faculty in Karlsruhe", KIT Scientific Publishing, 2010

busses, all on on the same chip (see Figure 1 (b)). This difference has important implications for communication and other overhead, ultimately influencing when parallelization pays off. We are given a new chance in which the break-even sweet spot of parallel performance is moved to a potentially more favorable location, and parallelism becomes applicable in scenarios and fields in everyday life that were never considered before. In addition, we also have new opportunities to exploit parallelism in-the-small, e.g., on a PC or in the pocket cell phone, not just in large-scale distributed systems or supercomputer clusters.

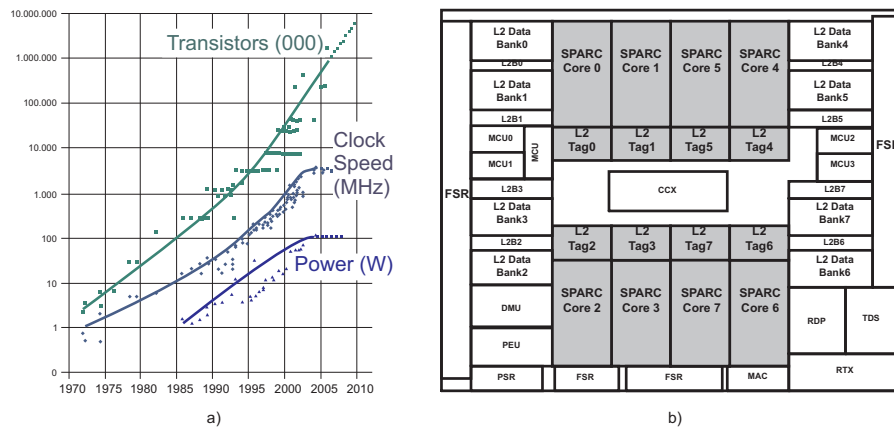


Fig. 1. a) Hardware developments [14]; (b) Sketch of the Sun Niagara II chip [10].

2 Why We Can't Ignore the Trend

Major software engineering methods and tools currently focus on sequential software development. However, every developer is now confronted with parallel programming of applications such as server applications, desktop applications (e.g., compression programs, multimedia programs, games, Web browsers), office applications, or business applications. In addition, parallelism can be used to improve application-level fault-tolerance and human-computer interaction.

Software engineers can't ignore the trend for several reasons. If a particular application is not parallel, it uses just one core, so performance cannot be improved by additional cores. As more and more cores are integrated on the same chip, clock rates may decrease, so if no other techniques are employed sequential applications could be slower on every new processor generation! Needless to say, customers won't buy new systems that have a lower performance than their old ones. Industry projects that all major processor product lines will increase the number of cores in the future. Exploiting parallelism in software becomes an unavoidable necessity.

The whole spectrum of software engineering – from design over testing to maintenance – has to be revisited in the light of parallelism. Nondeterminism adds a new dimension of complexity.

Compared to scientific applications (which I view as the traditional area of parallelism), everyday parallel applications emphasize different requirements. This working hypothesis is grounded on various studies I did in collaboration with industry (e.g., Intel, SAP, and Agilent). For example, exploiting the last few percent of available hardware performance is not so important, while scalability with the number of cores and portability to other platforms is really necessary. Performance can thus be sacrificed in exchange for gaining code that is easier to write, understand, debug, and maintain by teams of developers. Robustness of parallel programs plays a greater role in business-critical or safety-critical systems; scientific simulations can be restarted if something goes wrong, but incorrect financial transactions or flawed security systems may cause irreversible damage in real life. Nevertheless, the trustworthiness of scientific software results would also benefit from better software engineering.

Many parts of the existing software stack require revisions, each providing challenges and opportunities for parallelism exploitation. These parts include programming languages and compilers, libraries, middleware, and operating systems. Although my focus is on the upper layers, all parts are interdependent and have to be adapted and optimized in concert. Software engineering research cannot be looked at in isolation at a higher abstraction level, because the methods and approaches to be developed often rely on certain guarantees enforced in some lower levels. Coping with this interdependency is yet another challenge.

3 Addressing the Multicore Software Engineering Challenge

In the long run, my research aims to make the development of parallel software easier for the average developer by advancing concepts, methods, and tools. My work currently focuses on shared-memory multicore machines. Based on results from empirical studies that will be discussed later, I hypothesize that fully “automagic” parallelization of large, real-world applications is not generally promising. Software engineers need to be involved – more or less directly – in the parallelization process. Automation and tools are meant to support software engineers, assuming they know what to do.

My research takes place in the major areas depicted in Figure 2, which I consider key areas in addressing the difficulties of parallel programming: Automatic performance tuning, parallel programming models, and debugging techniques for parallelism. In addition, empirical studies are an important cross-cutting area serving two purposes: (1) learning about the key issues for each area, based on parallelizing complex real-world applications (most of them in collaboration with industry) as well as studies with programmers; (2) validating new approaches and theories proposed in each research area.

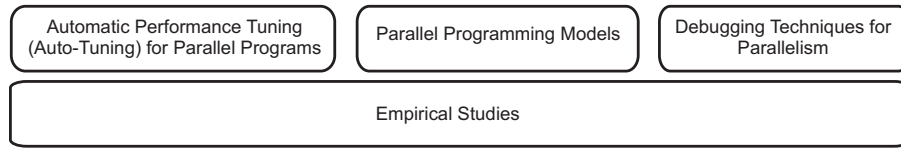


Fig. 2. Key areas of my multicore software engineering research.

3.1 Auto-Tuning

Performance is the main reason why we parallelize software. A complex parallel program can have many parameters influencing performance. Figure 3 (a) shows a biological data analysis application employing nested parallelism with tunable parameters on several layers [9]; for example, tunable parameters include the number of threads in different parts of the program, the size of various data structures that influences cache misses, but also the number of pipeline stages, the choice of algorithms and load-balancing techniques, or the number and size of data partitions that are processed in parallel.

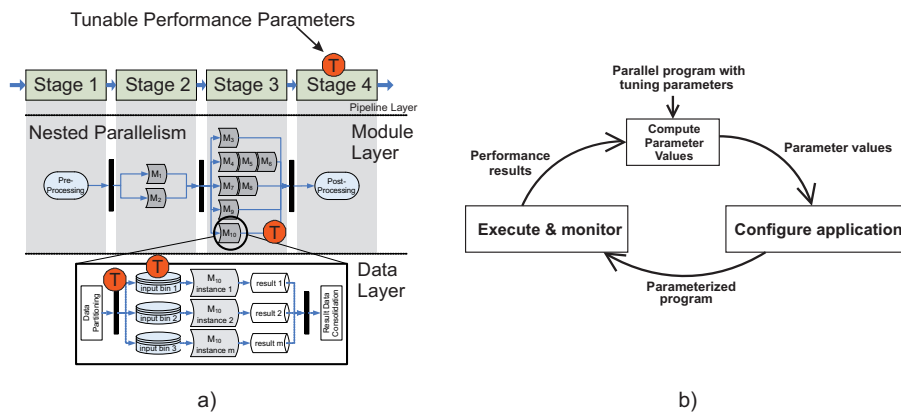


Fig. 3. (a) An example application with tunable performance parameters [9]; (b) Working principle of an auto-tuner based on [4].

Such parameters typically need to be adapted to the target hardware executing the program, because the characteristics of multicore platforms can vary, e.g., with respect to number of parallel hardware threads, cache architectures and sizes, memory bandwidth, or operating system. Adaptations are often done by hand and machine-dependent aspects are hard-coded to achieve best performance. This harms the portability of everyday software – imagine a user’s

disappointment of an application running fast on a laptop, but slow on a desktop PC. We cannot sacrifice portability in favor of performance.

This problem can be tackled using an auto-tuner, which is an external, dynamic optimizer working iteratively as shown in Figure 3 (b). In an offline tuning approach, a parallel application exposes tunable parameters, parameters ranges, as well as probes for feedback measurements to an auto-tuner. In one iteration, the auto-tuner calculates new parameter values, executes the application, and gathers feedback data. This cycle is repeated until some convergence criteria is satisfied (e.g., minimizing a given target function such as application run-time). The auto-tuning algorithms are a sub-area of my research.

We developed Atune-IL [11] as an instrumentation language to connect an auto-tuner to general-purpose multicore applications. A particular improvement are language features that exploit software architecture knowledge to avoid trying out unnecessary parameters combinations, thus significantly reducing the search space. Earlier experiments have also shown that fine-granular parallelism does not always provide enough leverage for good performance [9, 8]. As an improvement, we developed an approach to describe software architecture variants using parallel patterns; an auto-tuner finds the best parallel software architecture for a particular hardware platform [12].

My vision is that in the long-run, every application will be auto-tuned; every operating system will provide appropriate services and interfaces to make parallel application portability easy [4]. We successfully integrated an auto-tuner into the Linux operating system, which is capable of tuning several applications online while they are executing. In contrast to offline tuning, this approach can find the global, system-wide performance optimum. Important application scenarios include server applications, multimedia, games, and databases.

Why is auto-tuning important for software engineering? Because it separates performance tuning concerns from code development concerns. It can also find non-intuitive parameters values for the best performance. The tedious optimization is left to an auto-tuner, and programmers can write parallel code in a more generic way. Auto-Tuning will belong to the standard set of multicore software engineering tools, because it saves development time and improves portability. Parallel code becomes less complex, easier to read, understand, and maintain.

3.2 Parallel Programming Models

Programming models are a key area to improve developer's productivity and making parallelism easier to handle. Many industrial multicore applications have a complex structure and require the exploitation of several types of parallelism at the same time, such as task, data, or pipeline parallelism – very often in a nested fashion. In addition, current programming models have many pitfalls even for experts, leading to race conditions or deadlocks that go unnoticed for a long time.

One of my goals is to reduce the cognitive distance between the programmer's intentions and the actual implementation. I see a great potential in multi-paradigm languages; for example, declarative and imperative aspects can be

combined in the same language to better handle parallelism and easily express programmer's thoughts. Declarative parts offer more potential for automatic performance tuning, while imperative parts offer more precise control. The ideologic disputes of the past – whether a functional, logic, or imperative approach is better for parallel programming – asked the wrong question; the real question is how to combine the best of all worlds in a meaningful way.

A particular direction I initiated and contributed to was the combination of stream-orientation with object-orientation in the XJava language [6, 5], which allows writing a Java program partly expressed in a stream-oriented way. Among others, introducing an operator similar to Unix pipes in Java makes the exploitation of pipeline parallelism easier and introduces new opportunities for automatic thread handling and tuning. Data parallelism can be automatically exploited through replication of stateless filters, and task parallelism can be deduced based on program structure. The new language constructs added to Java make parallel programming easier, because the potential to introduce parallel programming errors is reduced. For example, programmers don't have to care explicitly about buffers between pipeline stages, synchronization, or signalling; instead, they just use one operator.

3.3 Debugging Techniques for Parallelism

Debugging multicore programs is difficult because of nondeterministic executions; compared to sequential programs, developers also have to deal with race conditions and deadlocks.

The exact solution to finding all races in an arbitrary parallel program is equivalent to the halting problem. However, this is such an important field that we cannot afford to give up, so we need to work with heuristics. Of course, the accuracy of race detectors used at development time can be improved [3]. As all heuristics have trade-offs, I also approach this problem from additional angles: (1) already in the programming model, e.g., by introducing parallel constructs that don't allow too many wrong usages; (2) race detectors that work at run-time [13], after the application has been deployed in a productive environment. To be usable at run-time, our prototype online detector only selectively captures some of the most common race patterns, but on the other hand it can automatically repair actually occurring races by delaying accesses of conflicting threads. (3) by employing data mining techniques on call graphs to detect, among others, wrong parallel program behavior that is caused by the wrong usage of non-parallel constructs [2]. This is a promising approach not only to automatically identify what causes incorrect results, but also what causes excessive resource usage (e.g., memory leaks) and performance degradation.

3.4 Empirical Studies

We are closely collaborating with industry partners to study novel applications areas for parallelism in realistic contexts with real data. The results show that multicore performance can indeed be successfully exploited in many scenarios.

For example, a parallelization of the route planner in one of SAP's business modules could reduce for real customer data (with hundreds of trucks and thousands of constraints) the computation time from hours to minutes. Other successes were achieved parallelizing a biological data analysis application, or extending Sun Microsystems's Electric Java application with parallel algorithms that speed up the placement optimization of cells on microchips. In other case studies, we parallelized a project management application, BZip compression, encryption (GnuPG), virus scanning (ClamAV), desktop search engines, and database query processing (PostgreSQL).

We also analyzed the potential and programmability of graphics cards for signal processing algorithms in an industrial context at Agilent, concluding that data parallelism can lead to significant speedups if the problems are suitable and have the right size, but that programmability and performance optimization are still a big obstacle. Other case studies used several teams working on the same programming problem, but with different programming approaches, e.g., Transactional Memory vs. Pthreads, or OpenMP vs. Pthreads; the results are summarized in [7, 8].

All studies show that despite the existing body of knowledge in parallel scientific computing, we need to extend our repertoire of tools for general-purpose parallel applications. Better software engineering support will benefit both scientific computing as well as everyday parallel programming.

4 Conclusion

Parallel computing has arrived on every desktop – it is not an exotic niche any more. These are exciting times for software engineering researchers, but also for practitioners to improve applications to scale their problem size and make innovative use of parallelism. The multicore approach has far-reaching implications for everyday applications that will go far beyond performance improvement. I envision that multiple cores will not be used to improve performance, but also to make everyday applications fault-tolerant and crash-proof, more secure, and to improve human-computer interaction in unprecedented ways.

Acknowledgments. Many thanks to the Excellence Initiative, the DFG, and the Landesstiftung Baden-Wuerttemberg for their support.

References

1. K. Asanovic et al. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.
2. F. Eichinger, V. Pankratius, P. W. L. Große, and K. Böhm. Localizing defects in multithreaded programs by mining dynamic call graphs. In *to appear in Proc. Testing: Academic & Industrial Conference Practice and Research Techniques*. Springer LNCS, 2010.
3. A. Jannesari, K. Bao, V. Pankratius, and W. Tichy. Helgrind+: An efficient dynamic race detector. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–13, 2009.

4. T. Karcher, C. Schaefer, and V. Pankratius. Auto-tuning support for manycore applications: perspectives for operating systems and compilers. *ACM SIGOPS Oper. Syst. Rev.*, 43(2):96–97, 2009.
5. F. Otto, V. Pankratius, and W. Tichy. High-level multicore programming with xjava. In *31st ACM/IEEE International Conference on Software Engineering*, pages 319–322, 2009.
6. F. Otto, V. Pankratius, and W. Tichy. Xjava: Exploiting parallelism with object-oriented stream programming. In *Euro-Par 2009*, volume 5704 of *LNCS*, pages 875–886. Springer, 2009.
7. V. Pankratius, A.-R. Adl-Tabatabai, and F. Otto. Does transactional memory keep its promises? Results from an empirical study. Technical report, Technical Report 2009-12, IPD, University of Karlsruhe, Germany, September 2009.
8. V. Pankratius, A. Jannesari, and W. Tichy. Parallelizing bzip2: A case study in multicore software engineering. *Software, IEEE*, 26(6):70–77, Nov.-Dec. 2009.
9. V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy. Software engineering for multicore systems: an experience report. In *Proc. ACM IWMSE '08*, pages 53–60, New York, NY, USA, 2008.
10. V. Pankratius and W. F. Tichy. Die Multicore-Revolution und ihre Bedeutung für die Softwareentwicklung. *Objektspektrum*, 4:30, 2008.
11. C. Schaefer, V. Pankratius, and W. Tichy. Atune-IL: An instrumentation language for auto-tuning parallel applications. In *Euro-Par 2009*, volume 5704 of *LNCS*, pages 9–20. Springer, 2009.
12. C. Schaefer, V. Pankratius, and W. F. Tichy. Engineering parallel applications with tunable architectures. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)*, Cape Town, South Africa, 2010.
13. J. Schimmel and V. Pankratius. Tachorace: Exploiting performance counters for run-time race detection. Technical report, Karlsruhe Institute of Technology, Technical Report 2010-01, 2010.
14. H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3), 2005.



Biography. Dr. Pankratius heads since 2007 the Multicore Software Engineering young investigator group at KIT. He serves as the elected chairman of the "Software Engineering for Parallel Systems (SEPARS)" international working group that has more than 100 members. His current research concentrates on how to make parallel programming easier for the average programmer and covers a range of research topics including auto-tuning, language design, debugging, and empirical studies. Dr. Pankratius received for his work the Intel Leadership Award, the Sun Microsystems Concurrent Computing Community Award, and the Microsoft Research Faculty Fellowship Finalist Award.

He received in 2007 a Ph.D. with distinction from the University of Karlsruhe, Germany, a Diplom degree (M.S.) in Business Computer Science best of class 2003 from the University of Münster, Germany, and a Bachelor of Science in Information Systems from the same university. He initiated and co-organized the series of Multicore Software Engineering workshops co-located with ICSE, the ACM/IEEE flagship conference on software engineering. He is a member of the ACM, IEEE, GI, and the elite program for postdocs of the Landesstiftung Baden-Wuerttemberg. Contact him at <http://www.victorpankratius.com>.