Software Engineering for Multicore Systems – An Experience Report

Victor Pankratius
University of Karlsruhe
76131 Karlsruhe, Germany
pankratius@ipd.uka.de

Christoph Schaefer
University of Karlsruhe
76131 Karlsruhe, Germany
cschaefer@ipd.uka.de

Ali Jannesari
University of Karlsruhe
76131 Karlsruhe, Germany
jannesari@ipd.uka.de

Walter F. Tichy
University of Karlsruhe
76131 Karlsruhe, Germany
tichy@ipd.uka.de

ABSTRACT

The emergence of inexpensive parallel computers powered by multicore chips combined with stagnating clock rates raises new challenges for software engineering. As future performance improvements will not come "for free" from increased clock rates, performance critical applications will need to be parallelized. However, little is known about the engineering principles for parallel general-purpose applications.

This paper presents an experience report with four diverse case studies on multicore software development for general-purpose applications. They were programmed in different languages and benchmarked on several multicore computers. Empirical findings include:

- Multicore computers deliver: Real speedups are achievable, albeit with significant programming effort and speedups that are typically lower than the number of cores employed.
- Massive refactoring of sequential programs is required, sometimes at several levels. Special tools for parallelization refactorings appear to be an important area of research.
- Autotuning is indispensable, as manually tuning thread assignment, number of pipeline stages, size of data partitions and other parameters is difficult and error prone.
- Architectures that encompass several parallel components are poorly understood. Tuneable architectural patterns with parallelism at several levels need to be discovered.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming* ; D.2.11 [**Software Engineering**]: Software Architectures —*Patterns*

General Terms

Experimentation, Performance, Design, Algorithms

Keywords

Multicore Systems, Design Patterns, OpenMP, Autotuning

1. INTRODUCTION

Inexpensive multicore chips (chips with several processors) are pushing parallel computing out of the relative niche of high performance computing into the mainstream. Already in 2005, affordable dual-core laptops, quad-core PCs, and eight-core servers were available on the market. Largely unnoticed went the fact that Cisco, also in 2005, developed a packet routing chip with 188 (!) processors [10]. The roadmaps of the semiconductor industry predict several hundreds of cores per chip in future generations [25, 30]. This development presents an opportunity that the software industry cannot ignore.

The bad news is that the era of doubling performance every 18 months has come to an end [23]. This means that the implicit performance improvement "for free" with every chip generation has also ended. Thus, future performance gains, required for new or improved applications, will have to come from parallelism.

Unfortunately, one cannot rely solely on compilers to perform the parallelization work [6], as the choice or parallelization strategy has a significant impact on performance and often requires massive program refactorings. Software engineering now faces the problem of developing parallel applications, while keeping cost and quality of software constant [6].

This paper takes stock of the current situation in multicore programming and suggests areas for future research and development. What are the tools and techniques we have right now to develop general-purpose software for multicore systems? What are the problems and difficulties? Is multicore programming worth the additional effort? Where do we need extensions and future research? To answer these questions, we conducted four case studies with applications from different areas, written in different programming lan-

Technical Report
Institute for Program Structures
and Data Organization (IPD)
University of Karlsruhe, Germany
December 2007

guages (C++ with OpenMP [8], Java, C#), and tested them on multicore computers manufactured by Intel and Sun Microsystems.

The paper is organized as follows. Section 2 discusses related work. The first application is a commercial biological data analysis program (Section 4). It is the most complex application in this study, as it requires parallelization at several levels. Parallelization is at a course level, viewing individual algorithms as black boxes. The next case study deals with fine-grained parallelism. It dissects a Monte Carlo simulation for project management. This case study demonstrates that an application that initially appears easy to parallelize nevertheless requires careful attention to shared data structures. It also shows that the choice of compiler can cause dramatic performance differences at different thread numbers. The next two case studies drill down to the algorithmic level: A parallelization of the shortest path problem demonstrates that significant algorithm engineering may be required to achieve speedup and that the available parallelism can be limited by the problem itself, not the number of available processors. The final case study, Section 7, briefly describes the parallelization of another classic algorithm, the traveling salesman problem. This problem provides ample parallel work, but the caches and memory accesses become a bottleneck as the number of threads increases. Section 8 distills the lessons learned.

2. RELATED WORK

Parallel programming is difficult, because it adds synchronization as a new problem area to be dealt with. Synchronization defects such as race conditions, deadlocks, and livelocks are known to be difficult to detect [4]. Furthermore, software developers need a thorough understanding of parallel algorithms. Although previous work in algorithms [14], operating systems [29], database systems [12], and high-performance computing (including cluster computing) [24, 32] dealt with these problems, the novel challenge now is to develop general-purpose engineering approaches for assisting ordinary programmers with the creation of potentially large, parallel applications. These approaches must not only target algorithmic levels, but also higher abstraction levels such as design patterns [20]. Strategic engineering aspects on how to approach parallelization on different abstraction levels are not well-developed. At the moment, development environments offer only low-level support for design, testing, or debugging. Other approaches with different underlying paradigms, such as transactional memory [28] or stream programs [13], are active areas of research. Asanovic et al [6] identified further gaps in the current landscape of parallel computing research. They collect elementary computation kernels with characteristic patterns of computation and communication that may occur in parallel programs. These kernels (nicknamed "dwarfs") include numerical and non-numerical problems and are meant primarily for performance evaluation of multicore architectures. Our case study covers the following dwarfs: Monte Carlo computations (as a special case of the Map/Reduce pattern), graph traversal, dynamic programming, backtracking, and branch and bound. Numerical dwarfs are not covered, as their parallelization is well understood in the HPC community.

3. CASE STUDIES OF MULTICORE SOFTWARE APPLICATIONS

Following the guidelines of [34], we conducted four independent case studies, carried out by different researchers, to asses the research question concerning the present state of multicore software development, as seen from the perspective of an ordinary software engineer. All case studies started with a sequential program version that needed to be parallelized. The case studies have a descriptive and explanatory character as they describe how the parallelization was done with the available languages and tools, and explain the observed behavior or performance issues where appropriate. Validity is constructed by collecting evidence from the application context, the created software artifacts, the languages/tools documentations, from qualitative observations during the development process, and from quantitative results (such as run-time data). The evaluation is done in two stages: the first stage appears at the end of each case study; the second stage appears in Section 8, combining the lessons learned from all four case studies.

The programs for the case studies were carefully selected. The first two are complete applications, a large, complex one and a short, simple one. Both applications include all processing steps, including all I/O. We also selected two classic (non-HPC) algorithms, again a complex and a simple one, because for some applications, it is enough to parallelize one or a few algorithms where most of the runtime is spent. Because of the diverse choices, and because we started with sequential algorithms in all cases, we can expect to achieve adequate coverage of the dominant phenomena that arise when parallelizing sequential applications in the context of this limited study.

4. MULTICORE BIOLOGICAL DATA ANALYSIS

The first case study subject is a commercial biological data analysis application (Agilent Technolgies' MassHunter Metabolite ID [2]). The focus is on coarse-grained parallelism at higher abstraction levels and on parallel design patterns. The sequential version of the application is written in C#.NET 2.0 and runs on ordinary desktop PCs.

4.1 Field of Application

The application performs a so-called metabolite identification. Metabolites are the intermediate and final products of metabolism; metabolism is the sum of chemical reactions that take place within the cells of a living organism.

Metabolite identification is an important method for testing drugs. For example, it can test how drugs are actually taking effect and helps with assessing the impact or adverse effects of drugs. The identification process begins with samples of body fluids, taken at certain points in time after the application of a drug. From these samples, mass spectrograms are produced. The problem studied here is to compare each of the mass spectrograms of the samples with a control sample taken before the application of the drug, in order to identify the metabolites caused by the drug. The drug's chemical structure is also used in this process.

4.2 How it Works

The application's main module is the metabolite identification unit, which executes a series of algorithms that iden-

tify and extract the metabolite candidates. Figure 1 shows the relevant processing steps. In the original program, the entire identification process runs sequentially; potential parallelism is not exploited.

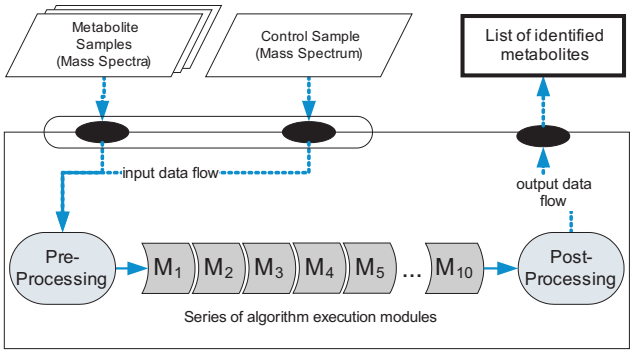


Figure 1: Conceptual structure of the sequential application.

The input data consists of the metabolite samples and the control sample. After preprocessing the samples (i.e., formatting the raw data and initializing the application's data structures), the algorithms try to identify metabolite candidates using various criteria and strategies. As depicted in Fig. 1, the algorithms are each wrapped by an execution module $(M_1, \ldots, M_n)$ that provides the required input data for the respective algorithm and retrieves the results. When the identification process is finished, a post-processing step consolidates and presents all identified metabolite candidates to the user. The algorithm sequence is repeated for each metabolite sample.

4.3 Experience

Our main point of interest was how this complex application can be parallelized at higher abstraction levels without touching the internals of the algorithms contained in the executions modules $M_1, \ldots, M_n$. This way, we wanted to assess whether speedup gains can be achieved by restructuring rather than by algorithmic improvements.

We were faced with the following challenges:

- Choosing appropriate levels of parallelization and identifying logical application layers was the first sticking point, especially as the application's architecture and structure was never designed for parallel execution.
- As parallel sections continued to grow and cover more program logic, we had to keep track of a larger number of data and task dependencies.
- The application with many parallel sections on different application levels was hard to tune because of interdependencies. Parameters for one particular parallel section, such as the number of threads or the size of data partitions, had to be set in concert with parameter values of other sections in order to achieve the best possible performance. This task was too complex to be managed manually.

We used the .NET threading library to implement thread-related functionality. We addressed the challenges by using the Data Decomposition Pattern, Task Decomposition Pattern, and Pipeline Pattern described in [20]. In addition, we tackled the tuning problem by constructing an autotuner [6]. The details are presented next.

Parallel Design Patterns.

We followed a methodical approach using the Parallel Pattern Language described in [20]. This pattern language is designed to systematically guide a software engineer through the process of developing parallel applications. We employed a bottom-up strategy to identify different parallelization layers. This approach was also useful for testing, as the layers at higher levels were parallelized only after the parallelization of the levels below was working. Figure 2 outlines the various sources of parallelism.

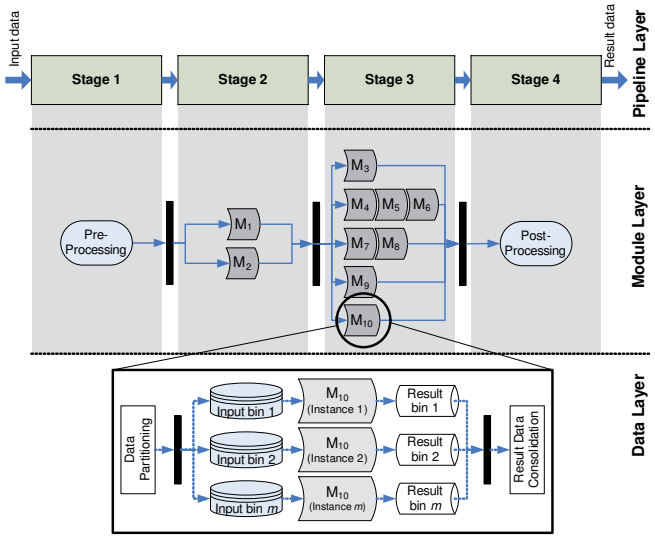


Figure 2: Conceptual structure of the parallelized application.

The **data layer**, on the lowest abstraction level, applies the Data Decomposition Pattern [20]. Using code inspections, we figured out that it is possible to divide the input data into several partitions and let different instances of an execution module work in parallel on each partition. As sketched in Fig. 2, the data layer partitions the input data into different bins for each execution module, starts the parallel execution of several instances of the module, and finally collects and consolidates the results.

The **module layer** applies the Task Decomposition Pattern [20] and ensures the correct initialization and execution order of the modules. Where possible, it forks the control flow to execute the algorithms concurrently. In our context, however, only certain orderings were possible. Also, we had to replicate the respective data structures and introduce barriers for joining the control flow back to a single thread (cf. Fig. 2).

The **pipeline layer** uses the Parallel Pipeline Pattern [20]. To increase throughput, the pipeline can be fed a new sample as soon as the first stage is idle. Thus, this layer processes several samples at once in a true pipeline fashion.

We made the following observations throughout the usage of parallel patterns:

- Using the Parallel Pattern Language enabled us to take a systematic approach to converting the sequential application to a parallel one.
- Converting a sequential application to a parallel version comes, however, with a high refactoring effort. The sequential application consists of about $100k$ of non-commented source lines, not including the identification algorithms and data access modules. The regions of code relevant for parallelization sum up to about $8,000$ lines. All of those had to be analyzed, about $1,000$ were changed and $1,500$ added. Tools for this process are definitely needed, and parallelizing compilers are not sufficient for this task.
- It would have been helpful to have predefined code templates for parallelization patterns. These should be extensible by the programmer and provide switches for configuring the template not only during programming, but also during performance tuning.

These observations influenced our approach to autotuning, which is described next.

Autotuning.

After converting the application from sequential to parallel, tuning all parallel sections was required to get the best possible performance. The most influential parameter was the number of threads used for a parallel section. In addition, the size of data partitions, the number of input samples as well as the number of pipeline stages had an impact on the overall performance (cf. Fig. 2). The identification of the optimal values for these parameters – especially for the number of threads – posed the following challenges:

- From a local point of view, a parallelized module could obtain considerable speed-up compared to its sequential version. However, from a global perspective, the parameters under which each parallelized module achieves its optimal speedup, differs from module to module. Due to the pipeline structure, one inadequately configured stage slows down the entire application. So a homogeneous thread assignment does not work well.
- The number of available processors differs from computer to computer. Therefore, the upper bound of threads that could be active varies as well, and this number does not necessarily equal the number of cores. Porting the application to another machine with a different number of cores or different memory organization requires re-tuning.

To overcome these obstacles, we created an autotuner – a program that automatically executes the parallel application within a predefined search space of parameter values, in order to find the parameter configurations that yields optimal performance. Even though we have prior experience with autotuning [33], no suitable autotuner was available that fully suited our needs, so we built our own.

Our autotuner was implemented as a .NET library. We defined the tuning parameters in a configuration file, and implemented the parallel patterns in a configurable way by making their behavior dependent on predefined parameters. The novel aspect is that the autotuner can vary not only numerical values (such as the number of threads or the size of data partitions), but also performs architectural variations (e.g., configuring the pipeline pattern with different numbers of stages).

The tuning results were not intuitive. With the help of the autotuner, we noticed that our manual assignments tended to use too many threads. Due to critical sections, I/O operations, or the total number of threads which were simultaneously active, the optimal value was often lower than expected. The performance gain between the worst and the best parameter combination on one machine could be as high as 40% (when using the max. number of cores).

The application was tuned on two different machines with two and eight cores, resp. On the two-core machine (Intel Core 2 Duo E6600 at 2.4 GHz, 2 GB RAM, Windows Vista) we achieved a total speedup of 1.7. On the eight-core machine (2x Intel XEON E5320 Quadcore at 1.86GHz, 8 GB RAM, Windows 2003 Enterprise Edition R2) the total speedup was 2.9. In both cases, about 30% of the achieved speedup resulted from the restructuring of the data and module layer, whereas the remaining 70% came from the pipeline layer. The reason why we had no linear speedup with respect to the number of cores was because of unavoidable locks and critical sections. As we have no access to the metabolite identification algorithms, we do not know whether parallelizing them beyond the data partitioning would speed up the application further.

In sum, this case study demonstrates the need for parallelization patterns, refactoring, and autotuning.

5. MULTICORE MONTE CARLO SIMULATION IN PROJECT MANAGEMENT

This case study focuses on a Monte Carlo simulation that uses random numbers to compute project completion times. It offers fine-grained parallelism and is thought to be trivially parallelizable, as the computation pattern consists of individual computations that are independent of each other.

The application was written in C++, and the parallelization was done with OpenMP [8]. The Microsoft Visual C++ 2005 Compiler and the Intel C++ 10 Compiler, which both have built-in support for OpenMP, were used under the Windows operating system.

5.1 Field of Application

The Monte Carlo simulation computes a probability distribution for the completion time of a project, given a schedule for the project. A project schedule (e.g., for building software or a skyscraper) consists of several tasks linked in a dependency relation that specifies the partial order of the tasks. As the durations of the individual tasks are not exactly known, one uses probability distributions instead (cf. Fig. 3).

The computational problem is to determine the probability distribution of the finishing times for the entire project. As an analytical solution is infeasible, one employs Monte Carlo simulation. The basic idea of this approach is to draw a random number for each task according to its probability distribution and to compute the total completion time with the given set of random numbers, taking into account the partial order of the tasks. This process is repeated thousands of times, and the computed durations are accumulated in a histogram. This histogram approximates the true distribution of completion times, i.e., for a given number of days

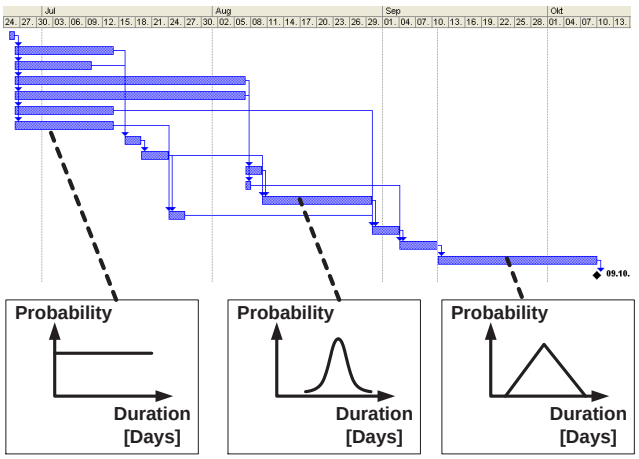


Figure 3: Example for an assignment of probability distributions to tasks of a project schedule.

one can determine the probability of the project completing in this many days. One can also compute the expected completion time and the variance.

5.2 How it Works

We use Microsoft Project to specify input files with probabilistic project schedules. For each task, user-defined attributes are set to specify the distribution type and distribution parameters. The following distribution types are available: Uniform, Triangle, Exponential, Weibull, Gauss, Gamma, Beta, or Erlang (see [16] for details). The completed schedule is piped into the Monte Carlo simulator.

The sequential version of the C++ application uses an array for representing the partial task order as an adjacency matrix, as well as other arrays for earliest and latest completion times, the distribution type and the distribution parameters for each task.

During a simulation step, the program assigns each task a randomly computed duration from its distribution, using the methods described in [15, 16]. Then it computes the earliest and latest completion times for each task (cf. [21]). At the end of a simulation step, the program collects the earliest completion time of the last task and updates the histogram of total completion times. At the end, the histogram is written to disk.

5.3 Experience

The parallelization of the sequential C++ application was done using OpenMP, a de-facto standard for programming shared-memory multiprocessors. Its philosophy is to insert directives into regular C++ code, which specify the creation of threads for parallel computations. For example, such directives may define parallel loops or critical sections [8, 19]. It is left to the operating system to schedule threads onto different cores or processors. OpenMP has a fork/join model of programming and assumes that a program has one master thread. When a parallelization directive is encountered, the control flow is forked into additional worker threads. Upon their completion, the control flow is joined again to the master thread. The OpenMP language is embedded into other languages, such as C++. Non-OpenMP compilers treat parallelization directives as comments and generate sequential programs.

Parallelizing this application was simple, as arbitrarily many instance of the main computational step can be executed in parallel and there are no dependencies. (This is also called an embarrassingly parallel computation). It is also completely straight forward to express this parallelization with OpenMP. We simply create a certain number of threads with an OpenMP directive. Each thread draws random numbers for each task, computes the completion time, and repeats. However, there were two issues that needed to be addressed: parallel random number generation and collecting the results in the completion time histogram.

If we had started the random number generator for each thread with the same seed, then each thread would operate on the same sequence, which would be useless. Following the approach proposed by Anderson [5] we use an initial run of the generator to produce one random number per thread, which is then used as the seed for the thread. In a separate run, we analyzed the total set of generated random numbers for independence and randomness with appropriate tests and plots [31].

The second problem, updating the shared histogram, permits three different solutions, which were all implemented.

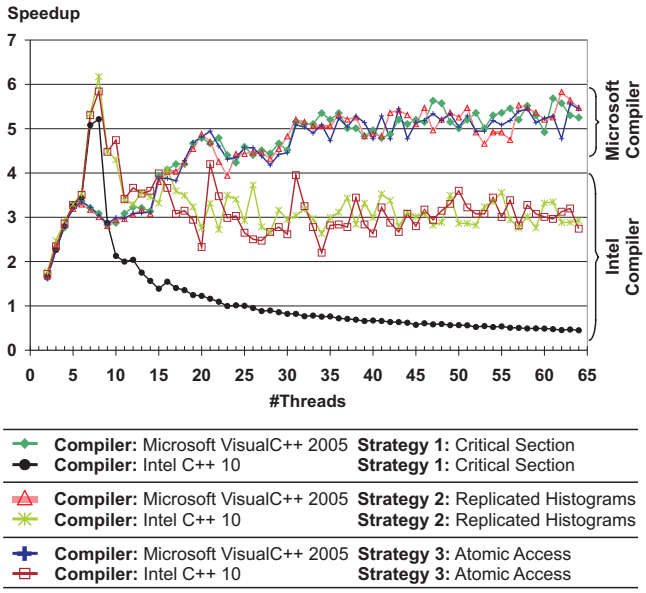
- **Strategy 1**: the update of the histogram vector is enclosed by a critical section directive, which locks the whole array (i.e., only one thread at a time can access the array).

- **Strategy 2**: every thread owns a private histogram for accumulating results. This approach has the advantage that no locking is needed during the computation stage. At the end, however, the private histograms must be summed into a final histogram. This step is a simple, repeated vector addition and easily parallelized: each thread sums a section of the vectors.

- **Strategy 3**: histogram update uses the *atomic* directive. In contrast to strategy 1, this directive locks vector elements individually, rather than the whole vector. This has the effect that update operations can proceed in parallel, as long as the threads write to different vector elements.

We varied the number of threads used in parallel sections and measured the execution times (including reading in the schedule and writing the histogram to disk) of each strategy on different multicore computers. The resulting speedups on an eight-core machine are shown in Fig. 4. We used all appropriate compiler optimizations, and in particular the special options offered by the Intel compiler for its own processors.

As can be seen, respectable speedups can indeed be achieved. For the Microsoft compiler, the three strategies do not differ much. For the Intel compilers, locking the entire histogram has a high cost for large numbers of threads. Above 25 threads, the parallel program is slower than the sequential one! It is also interesting to note that the Intel compiler delivers the best application performance when the number of threads is about equal with the number of cores. At higher thread numbers, performance drops markedly, while the Microsoft compiler produces a more or less steady performance increase as the number of threads goes up. The best absolute execution time (not shown in the Figure) was



All programs were compiled and run on a machine with 2x Intel XEON E5320 Quadcore at 1.86GHz, 8 GB RAM, Windows 2003 Standard x64 R2. The simulation used 1 million steps, a project schedule graph with 16 tasks, and 40 bins for the result histogram.

Figure 4: Performance results for the parallel Monte Carlo application with different implementation strategies.

obtained with strategy 2, eight threads, and the Intel Compiler.

Further experiments were conducted on two dual-core machines: a Toshiba Tecra M5 Laptop (Intel Core Duo T2500 at 2 GHz, 2 GB RAM, Windows XP) and a Dell Desktop PC (Intel Core Duo E6400 at 2.13 GHz, 3 GB RAM, Windows Vista). With the same simulation parameters, the speedup on these machines was in a corridor between 1.4 and 2; the Intel compiler with strategy 1 behaved in a similar way as in Fig. 4. The best absolute execution times were obtained with strategies 2 and 3 and the Intel Compiler. We also observed marked temperature increases on the processor chips when all processor cores were working at full capacity. The laptop chip was the hottest with $85°C$, pointing to a potential cooling problem.

Other Observations.

- Overall, OpenMP had the advantage that the existing sequential program could be parallelized incrementally. In addition, the parallelization was expressed on a higher abstraction level than threading.
- A drawback was the transparency of the OpenMP language for the C++ compiler. Compiler and debugger messages are not mapped back to the OpenMP source, but to some intermediate representation, which makes the error messages unintelligible and debugging complicated. The integration among OpenMP, underlying language, debuggers, and development environments should be seamless.
- Some refactoring was necessary for parallel random number generation and histogram access. For example, the initial sequential application grew from about 650 lines of code to 870 lines of code for strategy 2 (approx. 850 for strategies 1 and 3). About 40% of the parallel code was obtained by refactoring. Only a few lines of OpenMP (2–4) were needed for each version. There appears to be a vast potential for semi-automatic refactoring or redesign of data structures, which could work in conjunction with static program analysis.
- As shown in Fig. 4, the achieved speedup depends on different factors: number of threads, parallelization strategy, and compiler. Again, autotuners would help find the optimum.

6. MULTICORE SHORTEST PATH COMPUTATIONS IN GRAPHS

This case study analyzes the parallelization of computations on road network graphs, and discusses a parallel implementation of the computation of shortest paths.

The employed compiler was the SUN C++ 5.8 compiler with the SUN implementation of OpenMP. The experiments were conducted under the Solaris 10 operating system. To achieve comparable results, we turned off the Java garbage collection and took care of it explicitly within or application.

6.1 Field of Application

The Single-Source Shortest Path (SSSP) problem is defined as follows: Given a directed graph with nodes, weighted edges, and a distinguished start node, the problem is to find the path with the minimum sum of weights from the start node to all other nodes. The SSSP problem is important for navigation systems or finding train connections.

We used the SSSP problem to analyze the parallelization of a graph traversal application. For a realistic context, we used the following road networks:

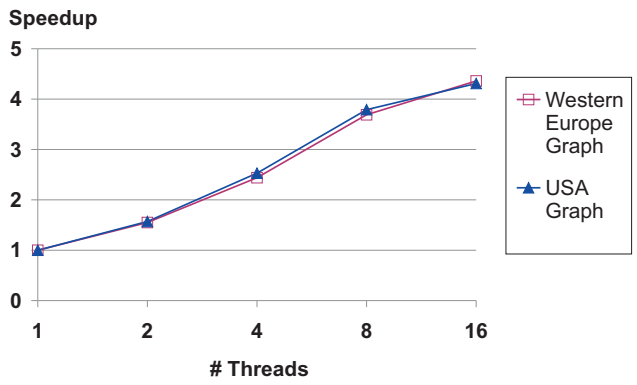
Graph	Nodes	Edges	Source
Western Europe	18,598,616	41,836,920	PTV Co.
USA	23,947,347	58,333,344	9th DIMACS Challenge [1]

6.2 How it Works

The SSSP problem is well-studied in the literature [18]. There are two major classes of algorithms: label-setting and label-correcting. Label-setting algorithms, such as the one proposed by Dijkstra [9], partition the node set into three subsets: 1) settled nodes whose distance from the start node is definitely known; 2) queued nodes that have a tentative distance, but which still can be modified; 3) unreached nodes, whose distance from the start node is not known. Edge relaxations are performed on nodes in subset 2) to improve the tentative distances (see [22] for details). By contrast, label-correcting algorithms do not designate settled nodes, and all distance computations are considered temporary until the final step [22]. For our case study, we focused on algorithms that do not preprocess the data.

6.3 Experience

Due to the special characteristics of the road network graphs (e.g., the graph of Western Europe has an average outdegree of 2.25) which differ from random graphs, first



SunFire T2000 T1-processor alias "Niagara" machine with 8 cores at 1 GHz, 16 GB RAM, Solaris 10 operating system

Figure 5: Performance results for the parallel shortest path computation on an eight-core system.

parallelization experiments showed that label-correcting algorithms were likely to perform better than label-setting algorithms, because the set on which relaxations could be performed was larger [11].

We used C++ and OpenMP to implement a label-correcting, parallel shortest path algorithm based on the Δ-stepping approach proposed in [22]. We extended that algorithm to support variable deltas instead of fixed deltas, to keep the number of nodes relaxed in each step constant. There is not enough space available here to explain the algorithm adequately, but the reader is referred to the M.S. thesis on this topic [11]. A few points about the implementation follow.

The data structure used to store nodes to be relaxed is a distributed radix heap [3]. Each thread has its own radix heap, and each node is owned by exactly one thread. Each thread is allowed to relax only its own nodes, which enhances locality effects. When a thread has to modify another thread's node, it generates a relaxation request instead of modifying it directly; every thread must also process the relaxation requests generated by other threads. Due to the data structures and various assumptions, the program runs with a number of threads that is a power of two.

The total implementation was 2200 lines of code, only about 10 were related to OpenMP. The core of the serial version is about two dozen lines of code.

As one of the machines this algorithm was measured on was the SUNFire T2000 with the first Niagara chip, and since this chip is known to have only modest floating point power (only one floating point processor), we implemented the algorithm with integer arithmetic. The performance of our implementation is depicted in Fig. 5; a speedup of 4.3 on an eight-core SUNFire T2000 machine is reasonable. By contrast, an experimental study of the Δ-stepping approach of [18] with fixed deltas on a Cray MTA-2 with 40 Processors had a speedup of only 2.95 for road networks (for random graphs, the speedup was 31).

This case study is an example for a problem that offers little opportunity for parallelism, mainly because of the small amount of work that could be processed in parallel. Extracting this work was extremely difficult (an entire Master's thesis [11]).

The case study also shows that parallel algorithms can behave differently on realistic data than on synthetic data.

7. MULTICORE TRAVELING SALESMAN

We focus in this case study on an multicore implementation of the Traveling Salesman Problem (TSP) in Java, using a branch and bound algorithm (integer version).

We used the Java 6 compiler. The parallel program was studied on two different eight-core machines, one under Windows Server 2003 and one under the Solaris 10 operating system.

7.1 Field of Application

The traveling salesman problem is a well-known NP-complete problem in combinatorial optimization. For its large number of applications see [17].

7.2 How it Works

Generally speaking, a branch and bound approach successively divides the set of possible solutions into smaller subsets, calculates bounds of the objective function value for these subsets, and uses these bounds to exclude appropriate subsets from further consideration. The process stops when either each subset has produced a feasible solution, or when it has been detected that a particular solution is not better than the one available so far. In the TSP context, the branch and bound algorithm repeatedly divides the solution space into two parts: a part with a given edge and a part without it, and evaluates them with the objective function on the length of the tours (see also [7]).

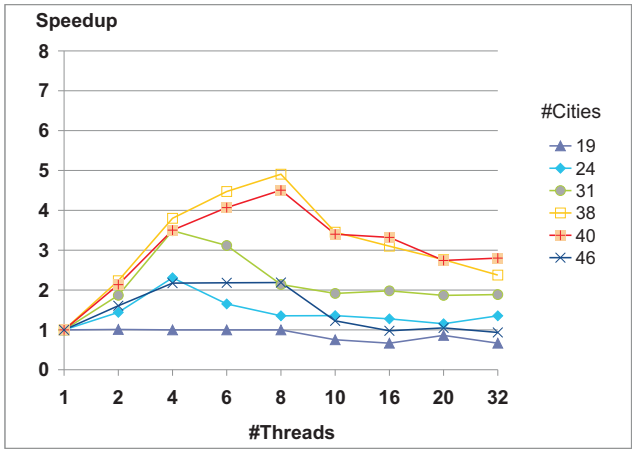
7.3 Experience

We briefly sketch the key points of the parallelized version of the TSP program. It has a total of 1300 lines of code; about 50 of them contain parallelization constructs. The partial problems resulting from the branching operations are stored in a thread-safe task queue, which is a priority queue sorted by the lower bound of the problems. The lower bound for each branch is computed as described in [27].

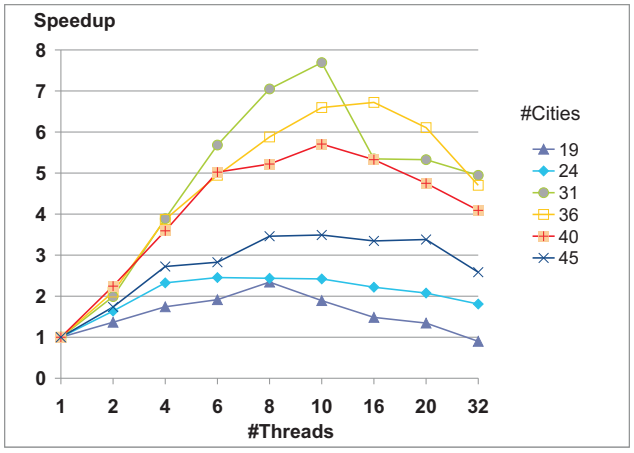
A thread processes and evaluates a branch with a given edge, and pushes the other branch without the given edge on the task queue. The task queue is a means of distributing work between cores and to achieve dynamic load balance. Branches with a higher lower bound are processed first, as they have a higher probability to get pruned and thus might not create new branches.

Figure 6 shows the performance results on two different eight-core machines. It can be observed that as the problem grows and the amount of data increases, the speedup improves as well. This means that the cores and the memory bandwidth are well-exploited when there are enough data and threads running in parallel. However, when the problem scale passes a certain point, speedup drops.

Due to the task queue with many partial solutions, the branch and bound algorithm is memory intensive. Therefore, it is important to implement this data structure in such a way that it can cope with this situation. In addition, when comparing the results of Fig. 6, it becomes obvious that the SUN processor can cope much better with the heavy use of memory, presumably because of the cache memory architecture. The SUN T1 processor has 3 MB shared L2 cache for all 8 cores, whereas the Intel Xeon processor has its cores



Machine with 2x Intel XEON E5320 Quadcore at 1.86GHz, 8 GB RAM, Windows 2003 Standard x64 R2



SunFire T2000 T1-processor alias "Niagara" machine with 8 cores at 1 GHz, 16 GB RAM, Solaris 10 operating system

Figure 6: Performance results for the Traveling Salesman Problem on eight-core machines.

grouped into pairs on one die, and a pair of cores share a 4 MB L2 cache.

The results imply that a developer currently needs to be aware of the cache architecture when designing the data structures or communication patterns in multicore programs. However, from a longer-term perspective, automated or semi-automated support is needed.

Another difficulty is that Java only supports parallelization constructs on a low level. The programmer is burdened with manually creating, destroying, or synchronizing threads. In many cases, developers might not need such a fined-grained control over parallelization. OpenMP-like extensions of Java would help.

8. LESSONS LEARNED FOR MULTICORE SOFTWARE ENGINEERING

Based on the empirical results and observations of all of the presented case studies, we summarize the lessons learned for multicore software engineering and discuss opportunities for future research.

- Our performance results show that parallelization works on multicore and is worth the effort.
- The parallelization strategy is important. Multicore performance issues are not merely a matter of compilers or operating systems.
- Threading is currently the dominating concept for parallelization. The mainstream programming languages, such as C# or Java, support threading concepts. However, managing threads explicitly is tedious and error prone.
- OpenMP eliminates explicit thread management. Sequential programs can be parallelized incrementally. Compared to alternatives (e.g., threading APIs or libraries, cf. [4]) OpenMP programs are easier to port to different platforms. Due to the poor integration of OpenMP into a host language (e.g., C++), developers get unspecific error messages if something goes wrong in the parallel program sections. Debugging is currently difficult as well. Better integration of OpenMP into host languages and debuggers is needed, or programming languages in which parallel constructs are "first-class citizens".
- Refactoring/restructuring are crucial for parallelization. This is also the case when an application uses thread-safe libraries, such as [26]. Future research seems promising for semi-automatic techniques which allow developers to consistently modify data structures or introduce certain access patterns with the click of a mouse. There is additional automation potential when restructuring techniques are used in conjunction with program analysis tools.
- There are typically several sources of parallelism. Sequential applications can be parallelized on different abstraction levels, and speedups can be gained from each of these levels.
- Parallel patterns are useful. Configurable parallel patterns appear to have significant potential. Parameters for such patterns could be set at design-time in order to configure a predefined pattern to a certain context. In addition, setting pattern parameters at run-time (e.g., which communication rules are to be used) will ease application tuning and make it possible to try out different architectures in a systematic way.
- Synchronization defects arise when developers work at low abstraction levels (e.g., the locking of data structures) and are not aware of the emergent behavior on higher abstraction levels (e.g., on a protocol level). Parallel patterns can help in this context to reduce such errors, as they can be pre-tested and pre-configured.
- I/O is in many cases the limiting performance factor. Using parallel patterns, developers can create initial program skeletons that show how to spread computations while waiting for I/O.
- Autotuning is indispensable for multicore software engineering. We definitely need future work in this area,

especially on intelligent heuristics that reduce the parameter space. It might be possible to prune the search space by analyzing a parallel program with static or dynamic program analysis techniques.

- Though not a result of these case studies, we can report from our experience with parallel programming on clusters that a shared address space model is vastly simpler to program than message passing, for example à la MPI. Though the increasing number of cores per chip will make it difficult to impossible to provide shared memory, a machine model in which programmers must explicitly send messages to access data should be avoided at all costs.

9. CONCLUSION

"The future is parallel" [25] - modern processors already have multicore architectures that offer true hardware parallelism at affordable cost, and the number of cores will continue to grow. This development will have fundamental impacts on software engineering theory, practice, and education, as every programmer will be confronted with programming parallel systems. The software engineering and research community needs to prepare for that situation. The case studies presented in this paper that, on a rudimentary level, multicore programming already works and is worth the effort, but that we also need significant advances before parallel programming becomes routine.

Acknowledgements

We thank Agilent Technologies Inc. for providing the source code of Masshunter Metabolite ID for study as well as Agilent Technologies Foundation for the financial support.

10. REFERENCES

[1] Challenge benchmarks of the 9th DIMACS implementation challenge - shortest paths. http://www.dis.uniroma1.it/~challenge9/download.shtml, 2006.

[2] Agilent Technologies. MassHunter Metabolite ID software. http://www.chem.agilent.com/Scripts/PDS.asp?lPage=57806, 2007. last accessed September 12, 2007.

[3] R. K. Ahuja, K. Mehlhorn, J. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, 1990.

[4] S. Akhter and J. Roberts. *Multi-Core Programming*. Intel Press, 2006.

[5] S. L. Anderson. Random number generators on vector supercomputers and other advanced architectures. *SIAM Review*, 32(2):221–251, June 1990.

[6] K. Asanovic et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

[7] E. Balas and P. Toth. Branch and bound methods. In Lawler et al. [17].

[8] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerical Mathematics*, 1:269–271, 1959.

[10] W. Eatherton. The push of network processing to the top of the pyramid. keynote address at Symposium on Architectures for Networking and Communications Systems, Oct. 26–28 2005. Slides available at: http://www.cesr.ncsu.edu/ancs/slides/eathertonKeynote.pdf.

[11] C. Frommeyer. Parallelisierung von Kürzeste-Wege-Algorithmen für die Anwendung auf Straßengraphen. Master's thesis, IPD Institute, University of Karlsruhe, Germany, 2007.

[12] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems. The complete book.* Prentice Hall, 2002.

[13] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM Press.

[14] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to parallel computing.* Addison-Wesley, 2nd edition, 2003.

[15] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley, 3rd edition, 1997.

[16] A. Law and W. D. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, 3rd edition, 1999.

[17] E. L. Lawler et al., editors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* John Wiley and Sons, 1985.

[18] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. Parallel shortest path algorithms for solving large-scale instances. In *9th DIMACS Implementation Challenge - Shortest Paths*, November 2006.

[19] A. Marowka. Parallel computing on any desktop. *Communications of the ACM*, 50(9):74–78, 2007.

[20] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming.* Addison-Wesley Professional, 2004.

[21] J. R. Meredith. *Project Management: A Managerial Approach.* John Wiley and Sons, 2003.

[22] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[23] D. A. Patterson. Computer science education in the 21st century. *Communications of the ACM*, 49(3):27–30, 2006.

[24] G. F. Pfister. *In Search of Clusters.* Prentice Hall, 1995.

[25] J. R. Rattner. Tera-scale computing - a parallel path to the future. http://softwarecommunity.intel.com/articles/eng/1inebreak[0]1275.htm, May 15 2007.

[26] J. Reinders. *Intel Threading Building Blocks.* O'Reilly, 2007.

[27] E. M. Reingold, J. Nievergelt, and N. Deo.

Combinatorial algorithms. Prentice-Hall, 1977.

[28] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, V10(2):99–116, February 1997.

[29] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts*. John Wiley and Sons, 7th edition, 2005.

[30] B. Smith. Reinventing computing. In *Keynote at the Manycore Computing Workshop*, Seattle, WA, June 20–21 2007.

[31] The R Development Core Team. R: A language and environment for statistical computing. http://www.r-project.org, June 27 2007.

[32] T. M. Warschko, J. M. Blum, and W. F. Tichy. The parastation project: using workstations as building blocks for parallel computing. *Information Sciences*, 106(3-4):277–292, 1998.

[33] O. Werner-Kytölä and W. F. Tichy. Self-tuning parallelism. In *8th International Conference High Performance Computing and Networking (HPCN)*, volume 1823 of *LNCS*, pages 300–312, Amsterdam, The Netherlands, May 2000. Springer Verlag.

[34] R. K. Yin. *Case Study Research: Design and Methods*. Sage Publications, 3rd edition, 2002.