Parallelizing BZip2

A Case Study in Multicore Software Engineering

Victor Pankratius
University of Karlsruhe
76131 Karlsruhe, Germany
pankratius@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

This paper presents a case study on parallelizing the sequential version of the BZip2 compression program for usage on multicore computers. We describe the encountered software engineering problems, discuss the tradeoffs of different parallelization strategies, and present empirical performance results.

The study was conducted during the last three weeks of a multicore software engineering course. Eight students, working in teams of two, were assigned the task to parallelize BZip2 in a team competition. Before starting with BZip2, all students had three months of extensive training in parallelization with POSIX threads and OpenMP, as well as knowledge of profiling strategies and tools.

Our empirical findings show that considerable speedups can be gained by exploiting parallelism on higher abstraction levels through parallel patterns, which are more significant than speedups obtained from a fine-granular parallelization. Another key issue we identify is the systematic refactoring of existing sequential code to prepare it for parallelization; the time needed for such refactorings can be significantly longer than for the actual insertion of parallelization constructs. The team who mastered these tasks well won the contest with a speedup above 10 on an eight-core SUN Niagara T1, while the weakest team produced a parallel version that was even slower than the sequential one.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.0 [**Software Engineering**]: [General]

Keywords

multicore systems, bzip, concurrency, synchronization, patterns, OpenMP, POSIX

University of Karlsruhe, Germany
Technical Report 2008-11
Institute for Program Structures and Data Organization (IPD)
First published: April 2008, updated December 2008
A modified version of this report will appear in IEEE Software Magazine

1. INTRODUCTION

With the emergence of affordable multicore processors that integrate several computing cores on a single chip, parallel programming becomes a concern for more developers than ever before. Despite the existing body of parallelization knowledge in scientific computing, numerics, operating systems, or databases, we still have many application areas in everyday computing that were not attractive for parallelization so far. Consequently, the software engineering of such applications also did not receive much attention. However, we are now at an inflection point where this is changing, as ordinary users possess multicore computers and demand software that exploits the full hardware potential. Unfortunately, previous empirical studies often focused on the mentioned areas, on architectures not comparable to current multicore systems, or on selected algorithms [13, 8, 7].

We now have a new situation where parallelism need no longer be confined to a narrow application range, but be employed in many different types of applications that are used in everyday life. We thus need to improve the software engineering for multicore applications and fill the existing gaps. One important building block for systematic engineering are empirical case studies that investigate in detail which parallelization approaches work and which do not work for certain applications. This must be complemented by an analysis of the reasons for success or failure of parallelization, along with an identification of the issues that require further research.

This paper addresses at this point and presents an empirical case study of the parallelization of the sequential BZip2 compression program [12]. This program was chosen for several reasons: it is widely used, it is a real application that is relevant in everyday life, the source code is available as open source, and the functioning principles are well-documented. In addition, the algorithms are not trivial and the size of the available code is just large enough to be manageable for students in an extended course exercise.

The parallelization of BZip2 was done as a 3-week course exercise at the end of a multicore software engineering course, after all students had three months of intensive training in parallelism. Eight students, working in teams of two since the beginning of the course, were assigned the parallelization task as a competition. We chose to conduct the study this way because we expected a larger diversity of solutions that would help triangulate the relevant engineering issues. To make the study more realistic, we set a 3-week time frame and allowed the students to pursue any strategy they considered promising, use any tools or languages they wanted,

or reuse any code or parallel library, as long as the binary compatibility of the compressed/decompressed files was preserved. We created standardized benchmark files for compression, and compared the parallel implementations on an eight-core SUN Niagara T1 with 16GB of RAM.

The paper is organized as follows. Section 2 presents the details of the multicore software engineering course and the context of the study. Section 3 outlines the fundamentals of the compression mechanisms of the sequential BZip program. Section 4 focuses on the parallelization of BZip, shows the details of how each team approached parallelization, and makes quantitative comparisons of the resulting programs. Detailed performance results of each team are presented in Section 5, and the winning team is identified. Thereafter, we discuss in Section 6 the lessons learned and make suggestions for the improvement. Threats to validity are discussed in Section 7. The novel insights are summarized in Section 8.

2. THE MULTICORE SOFTWARE ENGINEERING COURSE

The course was part of the computer science curriculum at the University of Karlsruhe, Germany, and was held during the winter semester 2007/2008. The course started October 2007, ended February 2008, and had a total duration of 15 weeks. It was supervised by a post-doc, a PhD student, and a student assistant. All participants were graduate students who voluntarily chose to participate because they were interested in the topic. All students already took classes on software engineering, good knowledge of C/C++/Java, and were highly motivated to develop practical skills in the area of parallelization.

The course had alternating teaching parts and practical parts (one week teaching, followed by two weeks of practice). The teaching part presented the concepts needed in the practical part. In the practical part, the students were given different tasks, such as writing parallel programs and testing them on the SUN Niagara T1. The last three weeks of the course were reserved for the final challenge: the parallelization of the sequential BZip2 compression program.

2.1 The Teaching Part

The teaching part covered the following topics:

- Basics of multicore programming
  (processor architecture, thread programming models, process models, design patterns, speedup considerations)
- POSIX Threads (PThreads)
  (concepts, thread management, synchronization with mutexes, condition variables)
- OpenMP
  (fork/join model, constructs for parallelization/ synchonization, thread scheduling strategies)
- Performance
  (caching issues, PThreads/OpenMP issues)
- Profiling
  (concepts and tools, e.g., Intel VTune [9], gprof [5], KProf [10], Valgrind (esp. Callgrind / KCachegrind) [15])

2.2 The Practical Part

The students were given assignments to write parallel programs. They worked in pairs that were formed at the beginning of the course. After two weeks, the students had to submit their solutions for pass/fail grading. All teams passed all assignments. We briefly outline the contents of the assignments:

- Introductory examples: try out the working environment on SUN Niagara, makefiles, Subversion
- Assignment 1 (PThreads): performance comparison of process vs. thread creation, find data dependencies in loops, parallel histogram computation for a list of integers
- Assignment 2 (PThreads): manual implementation of semaphores, implementation of the sleeping barber problem, implementation of the dining philosophers problem, parallel implementation of mergesort
- Assignment 3 (OpenMP): HelloWorld in OpenMP, parallel computation of π (numerical integration and Monte Carlo approach), parallel search for the smallest element of an array, parallel Ranksort
- Assignment 4 (OpenMP): parallel insertion/deletion of elements in a linked list, parallel Bucketsort, parallel application of a smoothing filter on a digital image
- Assignment 5 (OpenMP): parallel Quicksort, parallel matrix-vector and matrix-matrix multiplication, iterative Jacobi and Gauss-Seidel solvers

2.3 The Final Challenge: Parallelizing BZip2

After working on the topics described in Sect. 2.1 and 2.2, the students had a sense of achievement and were very confident of their skills. They were highly motivated when the parallelization of BZip2 was proposed as a final challenge, and happy to parallelize a larger program. BZip2 was available as open-source written in C. For the final three weeks, the teams competed against each other to obtain a parallel version of BZip2 with the shortest execution time on the 8-core SUN Niagara T1.

The rules of the competition were as follows. The students were told to try to parallelize on all fronts and at all possible levels. Everything was allowed, as long as the binary compatibility of compressed/decompressed files was preserved (i.e., for a given input file, the parallel version had to produce the same file as the sequential version of BZip2). It was permitted to use any program, tool (e.g., profiler, debugger), or library that a team found useful. Furthermore, the teams were allowed to rewrite any portion of the code or integrate any open-source code available on the Internet. The students were given an article describing the operation principles of the compression algorithm [14]. We also communicated to the students that a speedup of at least 2 was obtained for a parallel version in our previous feasibility study.

Various parallel implementations of BZip2 existed [16, 1, 6], and the students were aware of them. It was allowed to look into the code and re-use parts of it. However, [6] was not binary compatible with the sequential version of BZip2 for file with sizes greater than 900,000 bytes. The

other implementations [16, 1] were based on older versions of BZip2.

We prepared special input files with varying sizes and different suitability for compression. The intention was to make the execution times of the parallel compression programs comparable and to simplify the testing process. For example, some files contained random bytes, long sequences of zeroes, or were real-world archives obtained from the Internet. On of these files, the Eclipse package for Linux (eclipse-java-europa-fall2-linux-gtk.tar, 79MB, obtained from [2]) was designated as the benchmark for the competition. To simplify the benchmarking process, the teams had to make their programs parameterizable from the command line, e.g., with the number of threads to be used in parallel. The final benchmarking was done separately by a student assistant after the submission deadline for the parallel programs.

The students were asked to document their work, such as their initial strategies and expectations, the difficulties encountered during parallelization, and their actual approach.

3. FUNDAMENTALS OF COMPRESSION WITH BZIP

The relevant background on BZip is outlined next. We begin with the algorithmic principles of the sequential BZip and summarize some key points of the existing open source implementation that was used for parallelization.

3.1 Algorithmic Principles

BZip uses a combination of different techniques to compress data in a lossless way. An input file to be compressed is divided into fixed-sized blocks that can be processed independently. Each block is fed into a pipeline of algorithms. The most important stages of this pipeline are depicted in Fig. 1. The compressed blocks obtained at the end of the pipeline are stored in the original order in an output file. All transformations are reversible, and the stages are passed in the opposed direction for decompression.

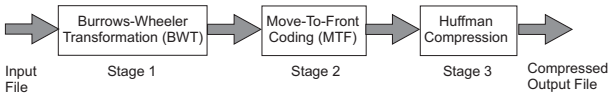


Figure 1: Operation principle of the sequential BZip2 algorithm.

The *first pipeline stage* performs a Burrows-Wheeler Transformation (BWT) on a block [4]. This transformation reorders the characters in such a way that similar characters have a higher probability of being closer to one another. Neither the length of the block nor the characters are changed. Conceptually, a string S of N characters taken from an alphabet X is transformed in the following way:

- form N cyclic shifts of S (imagine each shifted string written in a new line) and sort the lines lexicographically
- from each shifted S, extract its last character
- construct a string L whose ith character L_i is the last character of the ith sorted shift
- output (L, I), where I is the index of the original S in the list of shifts; index counting starts with 0

For example, $S =$ '*abraca*' yields $L =$ '*caraab*' and $I = 1$. The transformation can be reversed by using L and I to reconstruct the shifts and obtain a lookup vector that is used to map the characters to their original positions. Reference [4] presents further details and explains why this transformation is reversible.

The *second stage* performs a Move-To-Front (MTF) Coding [3] on (L, I). The string L is transformed as follows [4]:

- define a vector $R = (r_0, \ldots, r_{N-1})$
- create a list Y. Initialize it to contain each character of the alphabet X
- for $i = 0 \ldots N-1$
  - set r_i = number of characters in Y preceding character L_i
  - in Y, move the character L_i to the front of Y
- output (R, I)

For example, MTF produces for $L =$ '$caraab', I = 1$ the vector $R = (2, 1, 3, 1, 0, 3)$ and $I = 1$. This locally-adaptive algorithm assigns low integer values to symbols that reappear more frequently, and replaces long runs of the same character with zeroes. The resulting vector can be compressed more efficiently.

The *third stage* applies the well-known Huffman compression (cf. [11]) to the vector obtained in the previous stage.

3.2 Open Source Implementation

An open source implementation called BZip2 was developed by Seward [12]. It employs the algorithms described in the previous Section and allows the definition of a block size from a range of 100.000 Bytes – 900.000 Bytes.

The compression program is built on top of a library that provides low-level and high-level interfaces to the compression algorithms.

The low-level interface consists of functions that compress and decompress data in main memory. The sorting algorithm needed for the BWT (cf. Sect. 3.1) has a sophisticated fallback mechanism to improve performance.

The high-level interface provides wrappers for the low-level functions and adds functionality for dealing with I/O, such as streams and files.

4. PARALLELIZING BZIP2

We now describe the parallelization strategies of the competing teams and present quantitative comparisons of their parallel programs.

Evidence [17] is collected from the written reports of the teams, personal observations, the submitted code, the final presentations, and from interviews with the students after their presentations.

4.1 Strategies of the Competing Teams

4.1.1 Team 1

The first team had different parallelization strategies that were radically changed under time pressure. In principle, they started with their own approach to parallelize algorithms on a low level, using a mixture of OpenMP and PThreads. As the submission deadline approached, they decided to revert to an earlier snapshot and port the parallelization ideas of BZIP2SMP [1] to their code.

Their planned strategy was to understand the code base (1 week), parallelize it (1 week), and test/debug the parallel version (1 week).

Their actual work soon departed from their original plans. At the beginning, the team tried about 2 hours to get an overview of the code and find the files that were relevant for parallelization. In another 3-4 hours, they created execution profiles with gprof, KProf, and Valgrind. The group realized that input data had to be chosen carefully in order to find the critical path and keep the data sizes manageable. Another 2 hours were invested in understanding interesting code along the critical path. General code understanding as well as studying article [14] took about 6 hours. Thereafter, they found the parallel processing of data blocks to be most promising, but they had problems to unravel existing data dependencies.

The team continued with a parallelization on a low abstraction level, taking about 12 hours. In particular, frequently called code fragments were parallelized with OpenMP, and a sorting routine was exchanged for a parallel Quicksort implementation using PThreads. However, the achieved speedup was small.

The team decided to refactor the code to OOP and improve its readability. After 8 hours, they were surprised to see that the execution times did not differ much from the previous version, but the code was now easier to understand. The restructured code also eased the work on the parallel processing of data blocks, which took about 12 hours. Although only a few lines had to be changed to introduce parallelism, it was difficult for the team to assess the impact of those changes.

Although the refactoring approach was promising, the group ran out of time as the deadline approached and decided to abandon this strategy. They reverted to the version before the OOP refactoring and began to integrate some parallelization ideas obtained from the BZIP2SMP [1] code basis (2 hours). Additional 3 hours were used to make the files ready for submission.

In the end, the team reported that fine-granular parallelization was inappropriate. It would have been necessary to restructure the code even more in order to be able to parallelize effectively.

4.1.2 Team 2

The second team focused on extensive restructuring of the sequential program before starting with the parallelization.

Their plan was to analyze and profile the code (1 week), refactor it (1 week), and parallelize it (1 week).

The group spent about 2x50 hours of work in total. The first week was dominated by code analysis and profiling of the sequential Bzip2 with Valgrind and gprof. In the remaining time, their work concentrated on restructuring the sequential version of BZip2 and preparing the code for parallelization. Two days before submission, the team was still refactoring; the actual parallelization was done in the last day before submission.

In particular, the entire library used in BZip2 as well as the I/O routines were rewritten using a producer-consumer pattern. Thereafter, PThreads were used to introduce parallelism. The team realized early that a fine-granular parallelization, e.g., of the sorting algorithm, did not yield sufficient speedups. Therefore, they tried to achieve parallelism on higher abstraction levels. The massive refactorings were indispensable to resolve data dependencies and enable blockwise compression in their producer-consumer approach.

Although the team identified several other hotspots for parallelization, they did not have enough time to tackle them. For example, no time was left for fine-tuning the parallel version obtained so far. The team also planned to try pipeline to improve throughput.

The group reported that they drastically underestimated the time needed for refactoring. Refactoring was accompanied by long periods of frustration, until an executable parallel version existed. Nevertheless, the team was aware that such drastic restructuring was indispensable.

4.1.3 Team 3

The third team started with a fine-granular parallelization strategy using OpenMP and abandoned it later in favor of a master-worker approach using PThreads.

Their initial strategy was to begin with program and algorithm understanding, followed by a parallelization with OpenMP.

The students mentioned that they worked 6 or more hours a day. During the first 10 days, 2-3 hours a day were spent on understanding code and algorithms, as well as on trying out OpenMP directives. The sequential code was profiled with gprof to find performance bottlenecks.

After trying out different ways of fine-granular parallelization with different OpenMP directives (e.g., `parallel for`), the team realized that the speedups were not promising. They were aware that the changes would have to be much more invasive. However, they did not want to make large modifications to the library used by BZip2. They decided to focus on data parallelism on a higher abstraction level and implement a master-worker approach in which different blocks of a file were compressed independently. The master filled a buffer with blocks, while workers took blocks from the buffer to compress them.

The thread synchronization mechanism between master and workers was considered difficult. The students used sequence diagrams to design it and conditional variables and mutexes to implement it. Another difficulty was the file output, which required adjustment of the sequence of the compressed blocks to obtain the original order.

Unfortunately, the team did not finish the parallel version by the deadline, and it was therefore excluded from the final competition. Although the main reason was a trivial bug in an I/O routine, the students said that they were too tired to find and fix it. However, the group submitted a working version one week after the deadline, and we used this version for benchmarking.

4.1.4 Team 4

This team had a trial-and-error approach for parallelization, working from the bottom up.

Their strategy was to create execution profiles of the sequential code with gprof/KProf, find the critical path, and parallelize the code along this path. OpenMP was chosen as a means for parallelization, as the students considered it to be simpler and superior to PThreads.

The team reported that their actual work was dominated by trying out ideas. In sum, they spent about 70% of their time on implementing and debugging ideas, and only 30% on actually reading and understanding the sequential code. Program understanding was perceived as one of the most difficult tasks. Large parts of the code were misunderstood during the first parallelization attempts, and the team failed to gain a thorough understanding of the compression algorithm.

Another difficulty was that many parts of the sequential code were not parallelizable right away, due to data dependencies, side effects of function calls, and tricky optimizations for faster sequential execution. In addition, many loops in the sequential version were realized in a `while(true){...}` style that did not allow enclosing them with parallel for loops with OpenMP.

Consequently, the group started to refactor the loops. They focused on loops with no function calls, thus avoiding the handling of side effects in the parallel case. Data dependencies were unraveled, leading to code that could be wrapped by parallel OpenMP for loops. Unfortunately, this effort was rewarded only with minor speedups.

The group explained that – from its point of view – OpenMP would be a good and scalable approach for parallelization in general. However, the students reported that in this case study, parallelizing BZip2 would have required a much more fine-granular synchronization between individual threads in order to preserve data dependencies. The usage of OpenMP required massive refactorings to make the sequential code parallelizable. This was difficult to do within the given time. Given the opportunity to start over, they would have resorted PThreads instead.

4.2 Quantitative Comparisons

The quantitative comparisons of the parallel BZip2 code produced by the four teams provide some interesting insights.

Table 1 shows the total lines of code (LOC), the LOC without blank lines and comment lines, and the number of lines containing parallel constructs (e.g., `pthread_create`, `pthread_mutex_lock`, `\#pragma omp`, etc.). Compared to the sequential BZip2, the LOC of the parallel versions vary about ±15%. There are only a few lines (less than 2%) expressing parallelism.

Table 2 shows that although the total LOC do not vary widely, the number of modified lines can be large (e.g., 49% for team 1). Team 2 and 3 modified about 12% and 17% of the lines of the sequential BZip2, also pointing to a significant refactoring effort. Even team 4 that tried an incremental parallelization with OpenMP had to modify about 3% of the original code.

5. AND THE WINNER IS...

Team 2 won the competition and obtained a speedup of 10.3, which was the highest of all teams. Speedups greater than the number of eight cores are possible, as the SUN T1 processor can execute up to 4 threads per core. Team 3 had a speedup of 9.8. A reasonable maximum speedup of 8.8 was achieved by team 1. Finally, team 4 obtained a speedup of 0.9.

Given the benchmark file of 79 MB mentioned in Sect. 2, team 2 had the lowest execution time of 25.8 seconds at 51 threads. They are closely followed by team 3 with 27.3 seconds at 127 threads, and team 1 with 30.2 seconds at 32 threads. By contrast, team 4 had their lowest execution time of 308.7s at 16 threads - this is about 15% slower than the sequential program that executes in 267.1 seconds! Moreover, their parallel program was designed in such a way that only 1, 2, 4, 8, or 16 parallel threads were allowed.

The detailed performance comparisons of the four teams are depicted in Fig. 2. All programs were compiled on the SUN Niagara machine with GCC 4.2.1. The execution times are shown in Fig. 2 a). A particular data point represents the time in seconds that was needed to compress the input file (including I/O) when the program was called with a particular number of threads. All data points are averages of 5 executions. The speedups in Fig. 2 b) are calculated with respect to the execution time of the sequential BZip2.

6. RECOMMENDATIONS

Several lessons learned from this case study are presented next, along with suggestions for improvement.

6.1 Parallelize on Several Abstraction Levels

The case study shows that a parallelization on low abstraction levels alone might not be enough.

In the beginning, it looked promising to profile the sequential code, determine the critical execution path, and parallelize the application along this path. Unfortunately, this approach does not produce respectable speedup results, because the resulting parallel version is too tightly bound to the sequential implementation. Many parallelization opportunities are not exploited, as the original program is heavily optimized for sequential performance.

Just parallelizing loops, even if they are on the critical execution path, yields only minor speedups. Due to data dependencies introduced by "tricky" programming in the sequential version, a lot of effort may be required to prepare the loops for parallelization (cf. team 1 and 4). Other low-granular replacements of calls to a sequential sorting routine with calls to a parallel sorting routine did not lead to significant improvements either (cf. team 1).

The most successful strategies tackled parallelization on higher abstraction levels by introducing a producer-consumer pattern (team 2) or master-worker pattern (team 3). Team 2 won the competition even though they did not have the time to do other fine tuning. In addition, the usage of high-level patterns improved the understanding of the parallel program.

Suggestions for improvement: *Further research is needed on how to provide developers with the appropriate support for parallelization on several abstraction levels. For example, useful parallel patterns must be identified and made reconfigurable, and techniques must be developed that simplify the integration of such patterns into existing code.*

6.2 Use Several Sources to Find Parallelism

The sequential implementation of BZip2 is not the only source for parallelism.

Of course, understanding and profiling the sequential code

Program	Total Bytes	Total Lines of Code (LOC)	Total LOC without comments or blank lines	#lines with constructs for parallelism; % w.r.t. column 4
Sequential BZip2	236,150	8,090 (100%)	5,102	0
Team 1	204,704	7,030 (87%)	4,228	49 (1.2%)
Team 2	246,079	8,515 (105%)	5,356	48 (0.9%)
Team 3	266,897	9,270 (115%)	5,915	82 (1.4%)
Team 4	239,387	8,207 (101%)	5,170	8 (0.2%)

Table 1: Comparison of total lines of code and lines with constructs for parallelism.

Program	#lines modified	# lines added	# lines removed
Team 1	2476 (49%)	801 (15.7%)	1675 (32.8%)
Team 2	600 (12%)	427 (8.4%)	173 (3.4%
Team 3	861 (17%)	837 (16.4%)	24 (0.5%)
Team 4	156 (3%)	112 (2.2%)	44 (0.9%)

Table 2: Total modified lines of code compared to sequential version of BZip2 (comments and blank lines are excluded; additions/removals with respect to the sequential version of BZip2).

gives some hints at existing performance bottlenecks. However, these bottlenecks may be closely related to the way the sequential version implements the algorithm.

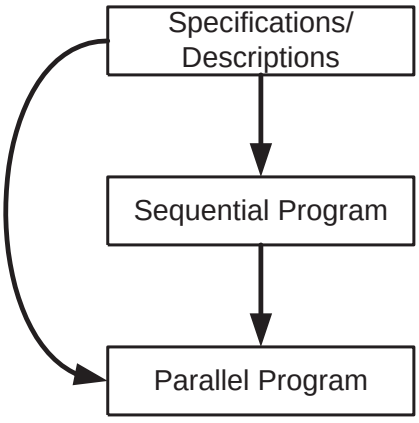


Figure 3: Identify potential parallelism using different sources.

Other sources for parallelism may be found in the specification or description of the compression algorithm (cf. Fig. 3). They are important because the sequential implementation of a program may omit information that does not improve sequential performance, but which is valuable for improving parallel performance.

In addition to documentation available online [12], the students were given an article describing the general compression principles used in BZip2 [14]; the article also contains references to papers on the employed algorithms. A thorough program understanding using different sources helps to assess the potential for parallelization, and is an intermediate step towards the identification of potential parallelism on higher abstraction levels. Teams 1, 2, 3 invested more time in these tasks and performed better than team 4 that mainly focused on the sequential code.

Suggestions for improvement: *Productivity can be increased by tools that assist developers at parallelizing a sequential program. For example, an interactive analysis of data dependencies might help decide on how to modularize code for parallelization. Building upon this, an automated proposal of parallelizable parts would help as well. Both program code and specification could be used as a source to improve the quality of suggestions.*

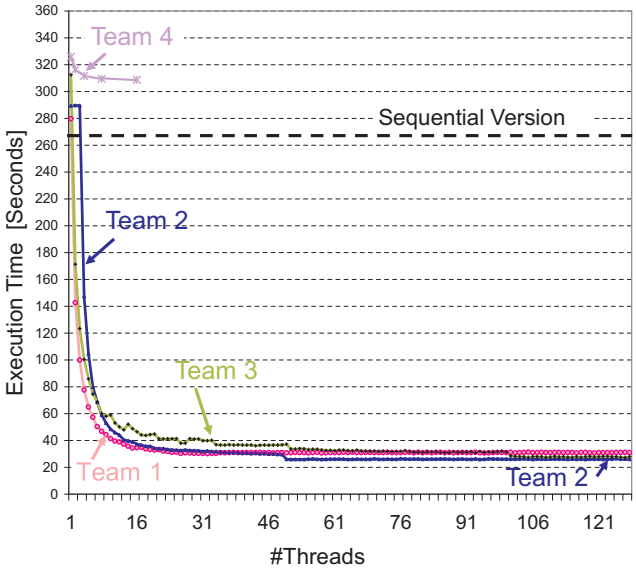
6.3 Don't Loose Your Nerve While Refactoring

In certain cases, parallelizing a sequential program is not possible right away, as massive refactorings are required to prepare the code for parallelization (cf. Sect. 4.2). Refactorings may be targeted at the improvement of code modularity or readability, the elimination of side effects of function calls, or the removal of unnecessary data dependencies.

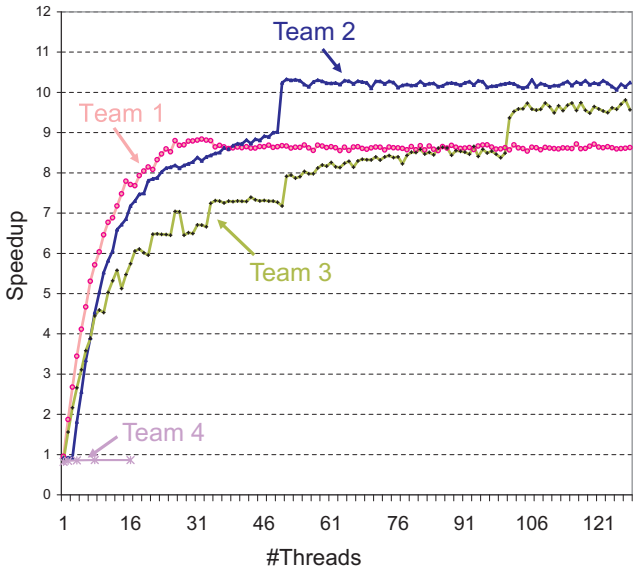
The case study shows that even simple loop parallelizations that are easy to do in classroom examples may be difficult to realize in practice. This becomes evident for team 4 that focused on parallelizing loops with OpenMP. Although parallelization constructs accounted for 0.2% of the total code (Tab. 1), 3% of the code – that is about 10 times more – had to be refactored (Tab. 2).

At another extreme, team 1 refactored almost half the code, while team 2 and 3 refactored 12% and 17%, respectively (Tab. 2). However, the strategies of teams 1, 2, 3 were much more invasive than the strategy of team 4.

Although teams 1, 3, 4 favored OpenMP at first, they



(a) Comparison of execution times



(b) Comparison of speedups

Figure 2: Performance comparisons of the four teams. The benchmarked programs use a block size of 900.000 Bytes and eclipse-java-europa- fall2-linux-gtk.tar (79MB, obtained from [2]) as an input file.

soon realized that it required even more refactoring. Teams 1 and 3 turned to PThreads instead, trading a more explicit (and potentially more error-prone) thread programming for less refactoring.

Team 2 reported that refactoring for parallelization was a frustrating experience, because it took a long time to see the results and have a sense of achievement. In the end, however, refactoring was an important success factor for winning the competition. Team 1 also got frustrated by refactoring, but stopped doing it under time pressure.

Suggestions for improvement: *Refactoring comprises many tedious tasks. If they are well-structured, tools could be designed to help with the preparation of sequential programs for parallelization. An interactive automation might relieve stress from the developers. This could help them focus on the relevant issues for parallelization, increasing productivity and reducing errors at the same time. Further research is needed to find out what the typical refactoring tasks are, and how they can be automated.*

6.4 Work in a Systematic Way

For successful parallelization, it is important to have a strategy on how to proceed. A strategy helps to set up long-term goals and short-term milestones, and allows developers to stay on the right track. A structured process is key to success for systematic engineering. Even though the strategy or the milestones might be changed in the course of action, it nevertheless important to define them.

Team 4 neglected these guidelines as well as the "big picture". The team did not work systematically enough, as the work was dominated by trying out ideas. For this reason, it was difficult to measure progress or decide whether to abandon certain approaches. This finally led to the worst result.

By contrast, Team 1 and 3 changed their initial strategies during their work, but they always had a plan. Both of them delivered a parallel BZip2 version with acceptable speedups. Although team 3 did not submit their results on time, they continued to work systematically and handed in a version that performed better than that of team 4.

Team 2 worked systematically and stuck to their refactoring strategy, even though they did not have a working version two days before the deadline. They did not panic under time pressure. We consider this to be an additional success factor.

Suggestions for improvement: *Future research can help to find a systematic process model for the parallelization of sequential programs. Such a model would form the basis for systematic engineering and helps developers to plan their steps and control progess. More empirical studies are needed to identify and validate the relevant phases as well as the structure of such a model. In addition, we also need metrics to measure the complexity of multicore programs.*

6.5 Current Industry Approaches

Current industry approaches seem to try to convey the idea that incremental parallelization of sequential programs can be done easily by simply inserting some parallelization constructs into sequential code. Furthermore, many solutions focus on fine-granular parallelization. Based on the experience from this case study, the following questions need to be asked: is the reality of multicore programming different from what the industry vendors tell us at the moment? Are the currently offered software solutions, such as languages or libraries for multicore programming, suitable?

At least for this case study, fine-granular parallelization does not seem to be promising. The achieved speedups are minor compared to what is possible when parallelization is complemented on higher abstraction levels. This means that a superficial parallelization of sequential code by incremen-

tally enclosing some portions with parallelization constructs may not yield acceptable performance in the long run. Moreover, this approach does not seem to be scalable due to the required refactoring effort.

Another industry trend is to offer parallel libraries for multicore. However, this study shows that in certain contexts, just exchanging library calls with parallel implementations might not yield acceptable performance. Other approaches, such as patterns (cf. Sect. 6.1) appear to be more promising.

Suggestions for improvement: *If these experiences are made also in other studies, we must fundamentally rethink our approach to parallelization for general-purpose applications. In particular, parallelization on several abstraction levels and refactoring must be addressed.*

6.6 Educational Aspects

The topics and tools that were taught in the course are standard in parallel programming. The students did not have many difficulties while they worked on the classroom exercises. In addition, they were familiar with software engineering techniques.

However, things changed when the larger BZip2 program had to be parallelized. Despite extensive training in parallelization, the results of the teams varied greatly. The students said that there was a difference between parallelizing small "toy" programs and real-world applications. Many concepts, such as loop parallelization or data partitioning could not be applied right way. Not only was BZip2 more complex, but it was also heavily optimized for sequential performance, making parallelization a difficult task.

Suggestions for improvement: *As future software engineers will be confronted with the parallelization of real-world applications, we need to prepare them adequately and address parallelization in computer science curricula. We obviously need to extend the repertoire of available techniques and tools that we teach. In addition, combining techniques from different field of computer science (e.g., covering parallel programming and software engineering) is unavoidable. From a practical point of view, case studies in parallelizing real-world programs are a suitable vehicle to train the skills needed in every-day situations, and at the same time gain valuable insights for research.*

7. THREATS TO VALIDITY

Within our course, all students had the same parallelization training before the BZip project started. However, individual skills could have influenced the results. To be sure that this case study was not dominated by such effects, we asked all students about other courses or labs they have taken before in the area of software engineering or parallelism. We found out that all of them had a similar experience. In addition, the skill levels we observed throughout the course were comparable – no student performed significantly worse than others. This is objectively backed by the fact that all students passed all assignments.

Some of the teams did not log their activities accurately enough. Team 4 could not give accurate effort numbers due to their trial-and-error approach, but provided estimates when asked during the interview. We checked the numbers reported by all teams for plausibility, based on the delivered code, comparisons between teams, and our own experience from parallelizing BZip in a feasibility study.

This case study can't be representative for all multicore software in general; it aims to present our experience with parallelizing a real program and develop an understanding for similar problems in practice.

8. CONCLUSION

In order to exploit the potential of current multicore hardware, applications of all sorts need to be parallelized. However, we are currently lacking empirical results in the area of multicore software engineering. This case study adds a piece to the body of knowledge and reports on the experience gained from the parallelization of a compression program.

A remarkable result is that many of the key activities for successful parallelization are software engineering activities beyond "mere programming". This claim is supported by several clues. Parallelization on higher abstraction levels using patterns improved speedups. Contrary to our initial expectations, this aspect was even more important here than more fine-granular parallelizations on an algorithmic level or loop-level. Moreover, just exchanging calls to sequential library functions with calls to parallel counterparts did not produce acceptable speedups, as the structure of the sequential program was highly optimized for sequential execution and acted like a tight corset. Parallelization on higher abstraction levels help to break through such barriers, and might become even more important when large applications – with millions of lines of code – are parallelized. Furthermore, a careful preparation of the sequential code for parallelization through refactoring was indispensable and represented another key factor to success.

Many of the engineering aspects for parallel software that were touched in this study are largely neglected at the moment. Research must fill these gaps quickly if multicore programming is to be successful on a large scale.

Acknowledgements
We would like to thank our student assistant Kai-Bin Bao, and the course students for their support.

9. REFERENCES

[1] BZIP2SMP v. 1.0. http://bzip2smp.sourceforge.net/, December 2 2005. last accessed August, 2008.

[2] Eclipse. http://www.eclipse.org/downloads/, Fall 2007. last accessed August, 2008.

[3] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, 1986.

[4] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, May 10 1994.

[5] J. Fenlason and R. Stallman. GNU gprof. http://www.gnu.org/software/binutils/, 2008. last accessed August, 2008.

[6] J. Gilchrist. Parallel BZIP2 v 1.0.2. http://compression.ca/pbzip2/, July 25 2007. last accessed August, 2008.

[7] L. Hochstein and V. R. Basili. The asc-alliance projects: A case study of large-scale parallel scientific code development. *Computer*, 41(3):50–58, 2008.

[8] L. Hochstein et al. Parallel programmer productivity: A case study of novice parallel programmers. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2005. IEEE Computer Society.

[9] Intel. Intel®VTune™Performance Analyzer 9.0. http://www.intel.com. last accessed April 2008.

[10] F. Pillet. KProf. http://kprof.sourceforge.net/, 2002. last accessed August, 2008.

[11] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 3rd edition, 2005.

[12] J. Seward. bzip2 v. 1.0.4. http://www.bzip.org/, December 20 2006. last accessed August, 2008.

[13] D. Szafron and J. Schaeffer. An experiment to measure the usability of parallel programming systems. *Concurrency: Practice and Experience*, 8(2):147–166, 1996.

[14] M. Tamm. Packen wie noch nie. Datenkompression mit dem BWT-Algorithmus. *C't*, 16:194, 2000.

[15] Valgrind. Valgrind tools. http://valgrind.org/, 2007. Last accessed August, 2008.

[16] N. Werensteijn. SMP bzip2. http://home.student.utwente.nl/n.werensteijn/smpbzip2/, May 2003. last accessed August, 2008.

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