Parallelizing Bzip2: A Case Study in Multicore Software Engineering

Victor Pankratius, Ali Jannesari, and Walter F. Tichy, *University of Karlsruhe*

As multicore computers become mainstream, developers need to know which approaches to parallelism work. Four teams competitively parallelized the Bzip2 compression algorithm. The authors report lessons learned.

Multicore chips integrate several processors on a single die, and they're quickly becoming widespread. Being affordable, they make it possible for every PC user to own a truly parallel computer, but they also make parallel programming a concern for more software developers than ever before. Not only is parallel programming considered difficult, but experience with parallel software is limited to a few areas, such as scientific computing, operating systems, and databases. Now that parallelism is within reach for new application classes, new software engineering questions arise.

In the young field of multicore software engineering, many fundamental questions are still open, such as what language constructs are useful, which parallelization strategies work best, and how existing sequential applications can be reengineered for parallelism. At this point, there is no substitute for answering these questions than to try various approaches and evaluate their effectiveness. Previous empirical studies focused on either numeric applications or computers with distributed memory,[1–3] but the resulting observations don't necessarily carry over to nonnumeric applications and shared-memory multicore computers.

We conducted a case study of parallelizing a real program for multicore computers using currently available libraries and tools. We selected the sequential Bzip2 compression program for the study because it's a computing-intensive, widely used, and relevant application in everyday life. Its source code is available, and its algorithm is well-documented (see the sidebar "Bzip Compression Fundamentals"). In addition, the algorithm is nontrivial, but, with 8,000 LOC, the application is small enough to manage in a course.

The study occurred during the last three weeks of a multicore software engineering course. Eight graduate computer science students participated, working in independent teams of two to parallelize Bzip2 in a team competition. The winning team received a special certificate of achievement.

Competing Team Strategies

Prior to the study, all students had three months' extensive training in parallelization with Posix Threads (PThreads) and OpenMP (see the sidebar, "Parallel Programming with PThreads and OpenMP") and in profiling strategies and tools. The teams received no hints for the Bzip2 parallelization task. They could try anything, as long as they preserved compatibility with the sequential version. They could reuse any code—even from existing Bzip2 parallel implementations,[4–6] although these implementations were based on older versions of the sequential program and weren't fully compatible with the current version.

0740-7459/09/$26.00 © 2009 IEEE

Bzip Compression Fundamentals

Bzip uses a combination of techniques to compress data in a lossless way. It divides an input file into fixed-sized blocks that are compressed independently. It feeds each block into a pipeline of algorithms, as depicted in Figure A. An output file stores the compressed blocks at the pipeline's end in the original order. All transformations are reversible, and the stages are passed in the opposite direction for decompression.

- Pipeline stage 1. A Burrows-Wheeler transformation (BWT) reorders the characters on a block in such a way that similar characters have a higher probability of being closer to one another.[1] BWT changes neither the length of the block nor the characters.
- Pipeline stage 2. A move-to-front (MTF) coding applies a locally adaptive algorithm to assign low integer values to symbols that reappear more frequently.[2] The resulting vector can be compressed efficiently.
- Pipeline stage 3. The well-known Huffman compression technique is applied to the vector obtained in the previous stage.

Julian Seward developed the open source implementation of Bzip2 that we used in our case study.[3] It lets block sizes vary in a range of 100 to 900 Mbytes. A low-level library comprises functions that compress and decompress data in main memory. The sorting algorithm that's part of the BWT includes 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.

References

1. M. Burrows and D.J. Wheeler, *A Block-Sorting Lossless Data Compression Algorithm*, tech. report 124, Digital Equipment Corp., 10 May 1994.
2. J.L. Bentley et al., "A Locally Adaptive Data Compression Scheme," *Comm. ACM*, vol. 29, no. 4, 1986, pp. 320–330.
3. J. Seward, Bzip2 v. 1.0.4, 20 Dec. 2006; www.bzip.org.

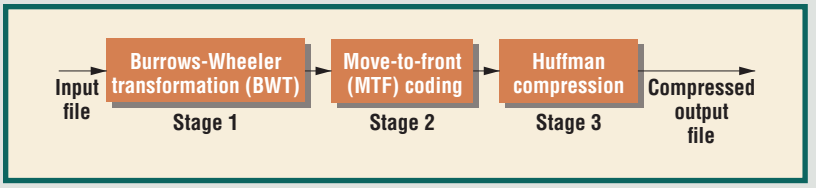


Figure A. The Bzip2 stages. The input file is divided into fixed-block sizes that are compressed independently in a pipeline of techniques.

We asked the teams to document their work from the beginning—including their initial strategies and expectations, the difficulties they encountered during parallelization, their approach, and their effort. In addition to these reports, we collected evidence from personal observations, the submitted code, the final presentations, and interviews with the students after their presentations.[7]

Because of space limitations, we omit a number of details here, but more information (including threats to validity) is available elsewhere.[8]

Team 1

The first team tried several strategies. They started with a low-level approach, using a mixture of OpenMP and PThreads. Then they restructured the code by introducing classes. As the submission deadline approached, they reverted to an earlier snapshot and applied some ideas from the BzipSMP parallelization.[5]

Team 1's plan was to understand the code base (one week), parallelize it (one week), and test and debug the parallel version (one week). Actual work quickly diverged from the original plan. At the beginning, the team invested two hours to get a code overview and find the files that were relevant for parallelization. They spent another three to four hours to create execution profiles with gprof (www.gnu.org/software/binutils), KProf (http://kprof.sourceforge.net), and Valgrind (http://valgrind.org).

The team realized that they had to choose input data carefully to find the critical path and keep the data sizes manageable. They invested another two hours in understanding code along the critical path. Understanding the code generally and studying the algorithm took another six hours.[9] Thereafter, they decided that parallel processing of data blocks was the most promising approach, but they had problems unraveling existing data dependencies.

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

The team decided to refactor the code and

Parallel Programming with PThreads and OpenMP

PThreads and OpenMP add parallelism to C in two different ways. PThreads is a thread library, while OpenMP extends the language.

PThreads

Posix Threads (PThreads) is a threading library with an interface specified by an IEEE standard. PThreads programming is quite low level. For example, `pthread_create(...)` creates a thread that executes a function; `pthread_mutex_lock(l)` blocks lock l. For details, David Butenhof has written a good text.[1]

OpenMP

OpenMP defines pragmas—that is, annotations—for insertion in a host language to indicate the code segments that might be executed in parallel. Effectively, OpenMP thus extends the host language. In contrast to PThreads, OpenMP abstracts away details such as the explicit creation of threads. However, the developer is still responsible for correctly handling locking and synchronization.

With OpenMP, you parallelize a loop with independent iterations by inserting a pragma before the loop. The following example illustrates a parallel vector addition:

```
#pragma omp parallel for      //OpenMP annotation
  for(i=0; i<N; i++) {        //usual C code
    c[i] = a[i]+b[i];
  }
```

In this example, OpenMP creates several threads that handle iterations of the loop in parallel. The example also illustrates OpenMP's idea of incrementally parallelizing a sequential program by inserting one pragma after the other in the code. When a sequential host compiles the code, it simply ignores the pragmas and runs the code as a sequential version. In our real-world study, OpenMP had limited applicability (see the lessons learned in the main text, under the subhead "Incremental Parallelization Doesn't Work").

OpenMP is standardized and available for C and Fortran.[2] Porting OpenMP to other languages is ongoing.

References

1. D.R. Butenhof, *Programming with Posix Threads*, Addison-Wesley, 2007.
2. B. Chapman et al., *Using OpenMP: Portable Shared Memory Parallel Programming*, MIT Press, 2008.

improve its readability by introducing classes. After eight hours of work, the execution times didn't differ much from the previous version, but the code was now easier to understand. The restructured code also made it easier to implement parallel-data-block processing, which took about 12 hours. Only a few lines had to be changed to introduce parallelism, but the team found it difficult 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. The team reverted to the version without classes and began to integrate some parallelization ideas from Bzip2SMP (two hours).[5] Additionally, they spent three hours making the files ready for submission.

In the end, Team 1 reported that fine-grained parallelization was inappropriate. It would have required too much effort to restructure the code for higher speedups.

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 (one week), refactor it (one week), and parallelize it (one week). The team spent about 2 × 50 hours of work in total. The first week went mostly to analyzing code and profiling the sequential Bzip2 with Valgrind and gprof. In the remaining time, they concentrated on restructuring and preparing the code for parallelization. Two days before submission, they were still refactoring. They performed the actual parallelization on the last day.

The team rewrote the entire Bzip2 library as well as the I/O routines using a producer-consumer pattern. Thereafter, they used PThreads to introduce parallelism. They realized early on that a fine-grained parallelization wouldn't yield sufficient speedup, so they tried to achieve parallelism on higher abstraction levels. 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 didn't have enough time to tackle them. For example, they had no time left for fine-tuning the parallel version or for a plan to improve throughput with pipelining.

Team 2 reported that it drastically underestimated the time needed for refactoring. They found refactoring to be frustrating because it took a long time before an executable parallel version was available. Nevertheless, the team knew that drastic restructurings were indispensable.

Team 3

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

The team initially planned to begin with program and algorithm understanding, followed by parallelization with OpenMP. They reported working six or more hours a day. During the first 10 days, they spent two to three hours a day on understanding code and algorithms and trying out OpenMP directives. They profiled the sequential code with gprof to find performance bottlenecks.

After trying different ways of fine-grained parallelization with different OpenMP directives (for example, `parallel for`), the team realized that the speedups weren't promising and that changes would have to be much more invasive. However, they didn't want to make large modifications to the Bzip2 library, so they decided to focus on parallelism at a higher abstraction level and implemented a master-worker approach in which they compressed different file blocks independently. In this design, the master fills a buffer with blocks, while workers take blocks from the buffer to compress them.

The team had difficulties with the thread-synchronization mechanism between master and workers. They used sequence diagrams to design the mechanism and conditional variables and locks to implement it. Another difficulty was the file output, which required a sequence adjustment of the compressed blocks to obtain the original order.

Unfortunately, Team 3 didn't finish the parallel version by the deadline, so they were excluded from the final competition. The main reason was a trivial bug in an I/O routine, but the team said they were too tired to find and fix it. However, they submitted a working version one week after the deadline, which we used for benchmarking.

Team 4

This team used a trial-and-error approach for parallelization, working from the bottom up. Their plan was to create execution profiles of the sequential code with gprof and KProf, find the critical path, and parallelize the code along this path. They chose OpenMP as a means for parallelization, which they considered to be simpler and superior to PThreads.

Team 4 reported that its actual work was dominated by trying out spontaneous ideas, which was why they didn't accurately log their effort in terms of person hours. During the post-competition interview, they estimated to have spent about 70 percent of their time implementing and debugging ideas and only 30 percent actually reading and understanding the sequential code. They perceived program understanding as one of the most difficult tasks. The team misunderstood large parts of the code during their first parallelization attempts, and they failed to gain a thorough understanding of the compression algorithm.

Another difficulty was that many parts of the sequential code weren't parallelizable right away, due to data dependencies, function-call side effects, and sequential-execution optimizations. In addition, the sequential version implemented many loops in a `while(true){...}` style that didn't permit enclosing them with the parallel loops of OpenMP. Consequently, the team started to refactor the loops. They focused on loops with no function calls, thus avoiding side effects in the parallel case. They unraveled data dependencies, which led to code that could be wrapped by parallel OpenMP loops. Unfortunately, this effort resulted in only minor speedups.

Team 4 explained that they thought OpenMP would be a good, scalable approach for parallelization in general. However, parallelizing Bzip2 would have required a much more fine-grained synchronization between individual threads to preserve data dependencies. The use of OpenMP required massive refactorings to make the sequential code parallelizable. This work was difficult to complete within the given time. Given the opportunity to start over, they said they would have resorted to PThreads instead.

Quantitative Comparisons

Quantitative comparisons of the parallel Bzip2 code produced by the four teams reveal some interesting points. Table 1 shows the total LOC without blank lines and comment lines, along with the number of lines containing parallel constructs, such as `pthread_create`, `pthread_mutex_lock`, and `#pragma omp`. Compared to sequential Bzip2, the LOC of the parallel versions vary about ±15 percent. Only a few lines—less than 2 percent—express parallelism.

Although the total LOC doesn't vary widely, the number of modified lines is quite high in the case of Team 1 (49 percent). Teams 2 and 3 modified 12 and 17 percent of the original code, also a significant refactoring effort. Team 4 modified about 3 percent of the original code, but failed to produce a speedup.

Team 2 won the competition by obtaining an

Team 2 won the competition by obtaining an impressive 10.3 speedup using 51 threads on a Sun Niagara T1.

Table 1

LOC comparisons for Sequential Bzip2 and four team efforts to parallelize its code

Program	Total LOC	Total LOC without comments or blank lines	LOC from previous column with parallelism constructs	LOC modified	LOC added	LOC removed	Total effort in person-hours
Sequential Bzip2	8,090 (100%)	5,102	0	—	—	—	—
Team 1	7,030 (87%)	4,228	49 (1.2%)	2,476 (49%)	801 (15.7%)	1,675 (32.8%)	~2∗50†
Team 2	8,515 (105%)	5,356	48 (0.9%)	600 (12%)	427 (8.4%)	173 (3.4%)	~2∗50†
Team 3	9,270 (115%)	5,915	82 (1.4%)	861 (17%)	837 (16.4%)	24 (0.5%)	~2∗30†
Team 4	8,207 (101%)	5,170	8 (0.2%)	156 (3%)	112 (2.2%)	44 (0.9%)	N/A

†*~2∗x represents the effort of two people working for x hours each.*

impressive 10.3 speedup using 51 threads on a Sun Niagara T1 (8 processors, 32 hardware threads in total). Speedups greater than the number of cores on the Niagara are possible, as the Sun T1 processor provides four hardware threads per core and the machine switches to a different hardware thread if an active thread waits for data. In general, it can pay off to have many more software than hardware threads—that is, more than 32 threads—to keep the processors busy.

Figure 1 compares performance results for the four teams. We compiled the benchmarked programs with the Gnu Compiler Collection (GCC) 4.2, using a 900-Mbyte block size and eclipse-java-europa-fall2-linux-gtk.tar (79 Mbytes) as an input file. We executed each algorithm five times and averaged the results. The figure shows that Team 4 achieved a speedup of less than 1. Their execution time was about 15 percent slower than the sequential program. Moreover, their program's design allowed only 1, 2, 4, 8, or 16 parallel threads.

Lessons Learned

We offer eight lessons learned from our experience.

Don't Despair While Refactoring

Parallelizing a sequential program might not be possible right away. It might require massive refactorings to prepare the code for parallelization. Refactorings improve code modularity, eliminate function-call side effects, and remove unnecessary data dependencies.

Team 1 refactored almost half the code, while Teams 2 and 3 refactored 12 and 17 percent, respectively (see Table 1). Team 4 refactored only 3 percent, but compared to a mere 0.2 percent parallelization constructs in its code, the refactoring is still significant.

Although Teams 1, 3, and 4 favored OpenMP at first, they soon realized that it required more refactoring than PThreads. Teams 1 and 3 turned to PThreads instead, trading more explicit (and potentially more error-prone) thread programming for less refactoring. The teams chose a suboptimal parallelization strategy because of the high refactoring cost that they would incur to do it right.

Team 2 reported that refactoring for parallelization was frustrating because it took a long time to see the results. However, refactoring was important for winning the competition. Team 1 also felt frustrated by refactoring and stopped doing it under time pressure.

Obviously, developers need some tools to help prepare sequential programs for parallelization. Automated support tools could increase productivity and reduce error rates. At the same time, tools could relieve stress and help developers focus on parallelization issues. Further research must define the typical refactoring tasks for parallelization and how to automate them.

Incremental Parallelization Doesn't Work

A purported strength of OpenMP is its enabling of incremental parallelization, which means you can start with sequential code and parallilize it by simply adding pragmas, one by one. This might be possible for simple cases, but we have yet to encounter a real program where the incremental approach works. Real programs like Bzip are full of side effects, data dependencies, optimizations for

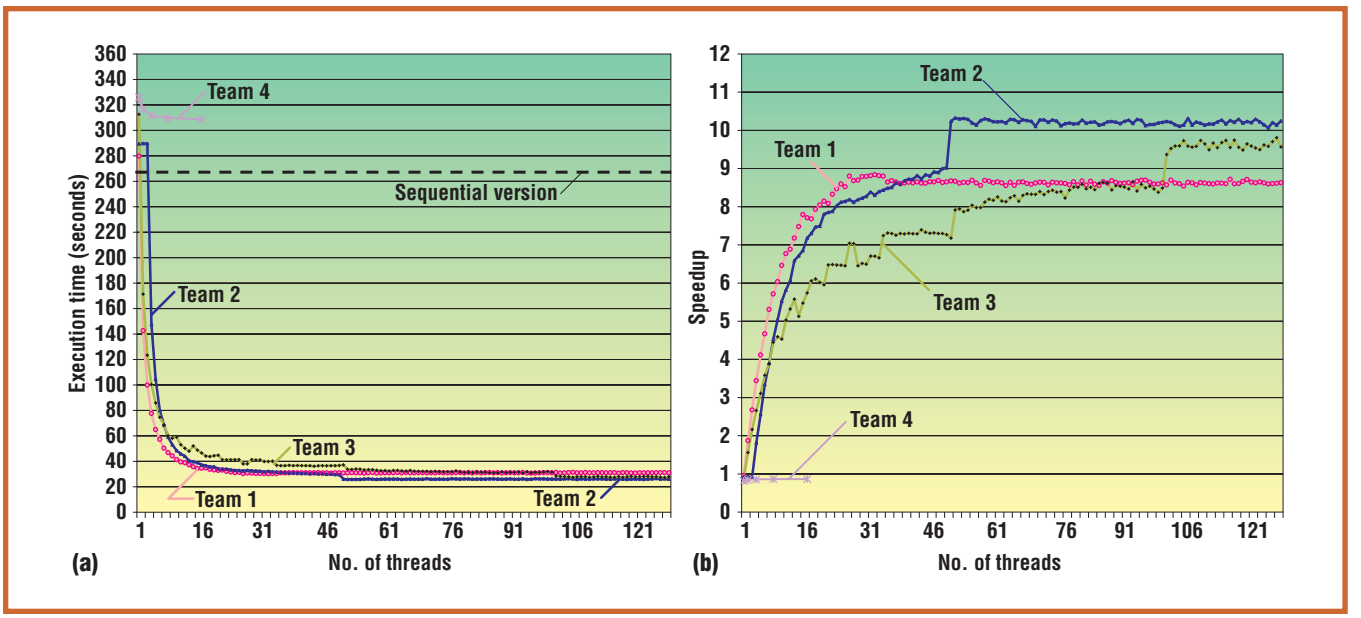


Figure 1. Performance comparisons for the four teams: (a) execution times and (b) speedups for the parallelized Bzip2 programs. All data points are averages of five total executions including I/O.

sequential execution, and tricky code. Significant refactoring is necessary.

PThreads might require less refactoring than OpenMP because it supports more explicit parallel programming, but the effort can still be significant.

On the basis of this and other case studies,[10] we believe the idea that sequential code can be parallelized without significant modification is a myth.

Look beyond the Critical Path

Some parallel programming textbooks advocate using a profiler to find the critical execution path and parallelizing along that path.[11] This advice sounds reasonable, but with hindsight, we don't think that it's a good approach. Parallelizing the critical path worked for none of the four teams in this study, nor for our other case studies.[10]

Profiling sequential code certainly provides information about performance bottlenecks. However, these bottlenecks might be closely related to the way the sequential version implements the specification. A sequential implementation typically involves design choices that preclude others representing degrees of freedom that effective parallelization might require. It's not enough to study the sequential implementation. You must also study the specification as well and consider alternative, parallel algorithms for the parallel version. This is why we gave the students an article that included Bzip's specification.[9]

Fine-Grained Parallelization Isn't the Only Choice

Parallelizing loops, even if they're on the critical execution path, yielded only minor speedups (see the results for Teams 1 and 4). Successful parallelizations constructed larger tasks for parallel execution.

Replacing a sequential sorting routine with a parallel version didn't significantly improve performance either, as Team 1's experience showed. We don't have enough data at this time, but we suspect that replacing standard library routines with parallel versions might not generate significant speedups for some real applications. If this is true, developing parallel libraries might remain an interesting exercise with little effect on practice.

Evaluate High-Level Parallelism's Potential

The most successful teams tackled parallelization on high abstraction levels by introducing a producer-consumer pattern (Team 2) or master-worker pattern (Team 3).

You might assume that high-level parallelization would be easier because it would involve less detail. In our experience, the opposite is true. The original program had nonlocal data dependencies to unravel, and the functions had side effects to understand before any changes could be made. Team 4 modified the code at loop level, which was actually easier but less successful. High-level improvements affected more parts of the program and were much more difficult to realize, as documented by Team 2's refactoring effort.

The students reported a big difference between parallelizing small "toy" programs and a real-world application.

We need further research on how to give developers parallelization support at appropriate abstraction levels. For example, we need to identify useful parallel patterns, as Timothy Mattson and his colleagues have done,[12] and make these patterns configurable. We also need techniques to simplify the integration of such patterns into existing code.

Trial and Error Is Risky

Team 3 started with trial and error in OpenMP and quickly realized that it wasn't the way to go. Team 4 didn't extricate themselves from this mode. It became difficult to measure progress or decide whether to abandon certain choices. In short, this approach failed.

It's easy to be seduced into a trial-and-error mode when working with code, but it's especially dangerous when parallelizing because you face many new choices and unknowns. Instead, you need a plan of attack. First, figure out the "big picture." Then develop some hypotheses about where parallelization might yield speedups. Then start identifying the most promising choices and eliminating poor ones. Some back-of-the-envelope calculations might help.

Parallel Programming Isn't a Black Art

Most people still think of parallel programming as a black art. However, our graduate students had only one semester's experience with parallel programming and didn't find it overwhelming.

Dealing with parallelism requires new concepts, new algorithms, and special attention to new types of defects. However, we believe that parallel system design and parallel programming aren't black arts that only the elite can master. We're hopeful that, with appropriate concepts, languages, tools, and training, competent programmers will handle multicore system challenges just fine. Given some time, the professional community will identify the engineering principles to support general-purpose, parallel software and the ways to teach and apply these principles successfully.

Other Issues

Beyond this study, the following advice might be helpful.

Consider Multiple Levels for Parallelization

Jon Louis Bentley identified different levels to consider when writing efficient, sequential programs:[13] "In most systems there are a number of design levels at which we can increase efficiency, and we should work at the proper level to avoid being penny-wise and pound-foolish." This is excellent advice for parallelization as well.

If You Start from Scratch ...

When you're writing a new sequential program from scratch, ask whether it might be parallelized in the future. If so, write it in a way that saves future refactoring work. For example, keep computational tasks modular and free of side effects so that individual threads can execute them. You should parameterize functions right away to simplify the division of work. This controls the parts of data to which the functions apply and enables domain decomposition. It's also important to write functions in a thread-safe manner that allows many instances to run simultaneously.

It might help to think in terms of parallel design patterns when writing a multicore program from scratch.[12] For example, the master-worker pattern manages independently running, parallel computations, while the pipeline and producer-consumer patterns break computations into phases that can be overlapped. You can combine parallel patterns—for instance, the master-worker pattern combines with the domain decomposition pattern—and you can split individual worker computations in pipeline fashion. Ideally, you would design all computational steps so that they can be packaged into threads. We recommend trying out prototypes early to check feasibility and performance. Prototypes are useful because you might not know cache effects or overhead costs of various building blocks (such as thread creation, communication, or I/O) ahead of time. These effects and costs might also deviate from what you expect.

Practical Training

None of the students experienced difficulties in creating small parallel programs from scratch. However, things changed when they had to parallelize Bzip2. Despite intensive training in parallelization, the students reported a big difference between parallelizing small "toy" programs and a real-world application. They couldn't readily apply concepts such as loop parallelization or data partitioning. Bzip2 was not only more complex but also heavily optimized for sequential performance, making parallelization a difficult task.

It's well known that computer science students need exposure to real systems, and multicore in no way diminishes this need. Fortunately, many students are excited about the current shift to parallel computation and eager to learn and

explore. They are further motivated by the prospect of a virtual job guarantee from being able to work with multicore/manycore software.

Multicore software isn't merely a matter of programming in the small. Many aspects of classical software engineering apply and must be adapted, such as design patterns, performance modeling, prototyping, refactoring, and tool support. Multicore software engineering is an area that seeks to build a core of systematic, empirically validated engineering principles for general-purpose, parallel software. We hope this limited study helps open the door to this research.

Acknowledgments

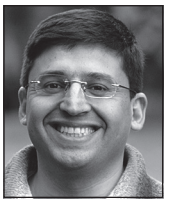
We thank our student assistant Kai-Bin Bao and the course students for their support. We also appreciate the support of the excellence initiative at the University of Karlsruhe.

References

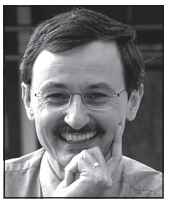
1. D. Szafron and J. Schaeffer, "An Experiment to Measure the Usability of Parallel Programming Systems," *Concurrency: Practice and Experience*, vol. 8, no. 2, 1996, pp. 147–166.
2. L. Hochstein et al., "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers," *Proc. 2005 ACM/IEEE Conf. Supercomputing*, IEEE CS Press, 2005, p. 35.
3. L. Hochstein and V.R. Basili, "The ASC-Alliance Projects: A Case Study of Large-Scale Parallel Scientific Code Development," *Computer*, vol. 41, no. 3, 2008, pp. 50–58.
4. N. Werensteijn, Smpbzip2, May 2003; http://home.student.utwente.nl/n.werensteijn/smpbzip2.
5. Bzip2smp v. 1.0, Dec. 2005; http://bzip2smp.sourceforge.net.
6. J. Gilchrist, Parallel Bzip2 v. 1.0.2, 25 July 2007; http://compression.ca/pbzip2.
7. R.K. Yin, *Case Study Research: Design and Methods*, 3rd ed., Sage Publications, 2002.
8. V. Pankratius et al., "Parallelizing *Bzip2: A Case Study in Multicore Software Engineering*, tech. report, IPD Inst., Univ. of Karlsruhe, Apr. 2008; www.multicore-systems.org/research.
9. M. Tamm, "Data Compression with the BWT Algorithm," *C't*, vol. 16, 2000, pp. 194–201 (in German).
10. V. Pankratius et al., "Software Engineering for Multicore Systems: An Experience Report," *Proc. 1st Int'l Workshop Multicore Software Eng.* (IWMSE 08), ACM Press, 2008, pp. 53–60.
11. S. Akhter and J. Roberts, *Multi-Core Programming*, Intel Press, 2006.
12. T.G. Mattson et al., *Patterns for Parallel Programming*, Addison-Wesley, 2004.
13. J.L. Bentley, *Writing Efficient Programs*, Prentice Hall, 1982.

About the Authors

Victor Pankratius is the head of the young investigator group on multicore software engineering at the University of Karlsruhe. In multicore software engineering, his interests include autotuning, language extensions, debugging, empirical studies, and making multicore programming accessible to many developers. Pankratius has a Dr.rer.pol. from the University of Karsruhe. He's a member of the IEEE Computer Society, the ACM, and the German Computer Science Society. Contact him at pankratius@ipd.uka.de.



Ali Jannesari is a PhD student in the University of Karlsruhe's Multicore Software Engineering group. His interests include debugging parallel programs. He received a master's degree in computer science from the University of Stuttgart. Contact him at jannesari@ipd.uka.de.



Walter F. Tichy is a professor of computer science at the University of Karlsruhe. He's also director of the Forschungszentrum Informatik, a technology transfer institute. His primary research interests are software engineering and parallelism. Tichy has a PhD in computer science from Carnegie Mellon University. He's a cofounder of ParTec, a company specializing in cluster computing, and a member of the IEEE Computer Society, the ACM, and the German Computer Science Society. Contact him at tichy@ipd.uka.de.