Parallelizing an Index Generator for Desktop Search

David J. Meder, Walter F. Tichy

Parallelizing an Index Generator for Desktop Search

David J. Meder and Walter F. Tichy

Institute for Program Structures and Data Organization (IPD)
University of Karlsruhe
76131 Karlsruhe, Germany
{meder, tichy}@ipd.uni-karlsruhe.de

Abstract. Experience with the parallelization of an index generator for desktop search is presented. Several configurations of the index generator are compared on three different Intel platforms with 4, 8, and 32 cores. The optimal configurations for these platforms are not intuitive and are markedly different for the three platforms. For finding the optimal configuration, detailed measurements and experimentation were necessary. Several recommendations for parallel software design are derived from this study.

Keywords: index generator, multicore, parallel software design, parallelization

1 Introduction

Developing multi-threaded applications for multicore computers is challenging. Numeric applications have been parallelized for a wide range of platforms, starting with the Cray vector computer in 1976. Over the years, numeric applications have been ported to all kinds of parallel platforms, including SIMD computers, shared-memory multiprocessors, clusters, grids, multicores, and GPUs. Consequently, the parallelization of numeric applications is well understood, at least in principle. However, little is known about parallelizing the vast number of applications that are non-numeric in nature. Multicore chips now power servers, desktop, and laptop computers and these are primarily used for non-numeric applications. To learn how to parallelize these applications, it is appropriate to perform case studies. By documenting these studies, such as the BZip2 example [1] or the applications covered by Schäfer et al. [2], an effective process for parallel software design might emerge. To add to this growing set, we conducted and documented an in-depth study of parallelizing desktop search. Desktop search is performed on PCs, laptops, smart phones, or similar. In its simplest form, it returns a list of files that contain a given combination of search terms. The search uses an inverted index that lists the files in which a given term occurs.

We chose desktop search for several reasons: It is a wide-spread, non-numeric application available on virtually every computing device with a file system, and

it is worth parallelizing. The application is simple enough to permit experimenting with alternatives, yet challenging enough in that the optimal solution is not obvious. It is also an I/O intensive application. In an earlier study, Pankratius et al. [3] conducted a competition among student teams to parallel desktop search. Surprisingly, the team with the best performance used software transactional memory, while the team in second place used locks. However, the two solutions were incomparable, because the teams stored different amounts of data in the index. Furthermore, the competition was performed under time pressure. Quite naturally the question arose what the best performance would be, given enough time to try out several alternatives.

We present our approach on how to parallelize the index generation of desktop search using locks. We provide results for three different platforms: a 4-core and a 8-core Intel platform in our lab and a 32-core Intel platform made available through Intel's public manycore testing lab [4].

2 What to Parallelize and how?

Before writing any concurrent code, one has to identify the components that can and should be parallelized. In case of the index generator, there are at least three parts that could be parallelized independently or in combination.

Filename generation: Traverse the directory hierarchy to generate the names of the files to be indexed.
Term extraction: Scan files and extract terms.
Index update: Add the extracted terms to an index structure.

At the outset of the project, we faced a number of questions: Which of those parts are the dominant ones and worth parallelizing? Is it traversing the file system from some root, opening and scanning individual files, or building the index? Or is the disk the slowest part, in which case there is no hope for significant speed up? None of these questions was answerable without measurement. Furthermore, it was unclear how to parallelize the filename generation and the index update.

2.1 Filename Generation

Traversing the directory hierarchy from a given root is an I/O intensive process, whose performance depends on variables such as the number of directories, the number of files contained in directories, the number, transfer rate, and seek times of the installed drives, and the buffering of the operating system. Parallelizing directory traversal is difficult, because directory trees are unbalanced. Another problem is how to distribute the filenames to multiple term extractors in a balanced way, since the file lengths are uneven. Work queues, round-robin distribution, assignment based on file lengths, or work stealing are the main options considered. Concurrent access to the filename data structure or the work queues was likely to slow everything down. We didn't even know whether it was worth parallelizing filename generation.

2.2 Term Extraction

The most I/O intensive job of the index generator is reading the files. It was unclear how many threads could be employed for this job before the file system bottlenecked; furthermore, the best configuration was likely to be dependent on platform characteristics, such as clock rate of cores, size of caches, and I/O performance. A single configuration was not going to be optimal for all platforms. Another question was how to handle duplicates of terms: Terms typically appear multiple times in a given document. Should a term be entered into the index every time it is found, or should the term extractor construct a condensed word list without duplicates from each file and then insert the list of terms all at once? The former technique might overwhelm the index with locking requests, while the latter approach might simply duplicate work that the index was well prepared to handle anyway.

2.3 Index Update

The main question concerns the relative speeds of index update and term extraction. Would it be enough to let the extractor threads update the index with a synchronized update method, or would it pay to have a separate process for index update that received sets of terms via a buffer? Is synchronization the bottleneck? If so, there is a way to avoid synchronization entirely, by applying a pattern we call "Join Forces". The idea behind this pattern is to let each term extractor build its own index and join the indices at the end. This approach would eliminate all synchronization, except for a barrier before the join operation. Would it be enough to join the indices with a single thread, or should a parallel reduction setup with multiple joining processes be used?

3 Parallelization

To answer some of the questions posed in the previous section, we needed to get some facts about the performance of the three parts named above. The first step was to set up a benchmark. It consists of about 51.000 ASCII text files and the file size distribution corresponds with file size distribution on a typical personal computer (see [5] and [6]). On the whole the file set contains about 869 MB of data, created by extracting plain text versions from word processor files. Handling complex word processor formats directly in the term extractor would have been too distracting at the time, even though it would be an interesting extension now. Plain text made scanning faster, but it also made the parallelization problem harder: the faster the term extractor runs, the less opportunity for speedup exists.

Next, we implemented a sequential version of the index generator and timed the individual parts. The execution times are shown in Table 1. Generating filenames only takes 5 seconds, or between 2 to 5 percent of total runtime. With this information, it was clear that parallelizing the file system traversal was

unnecessary. To avoid synchronization operations, we decided to use a single thread for the filename generation, which would generate the complete set of filenames in main memory before starting term extraction.

Table 1. Execution times for sequential index generation.

	Execution time (s)			
	filename generation	**read files**	**read files and extract terms**	**index update**
4-core platform	5.0	77.0	88.0	22.0
8-core platform	4.0	47.0	61.0	29.0
32-core platform	5.0	73.0	80.0	28.0

The next question was whether scanning the files was worth parallelizing, or whether the whole program was I/O-bound. To decide this, we built an empty scanner, i.e., a loop that simply reads each file byte by byte, but without any term extraction. Reading the benchmark from start to finish takes between 50-80 seconds on the three platforms. Extracting the terms adds another 7-14 seconds. (For more complex formats, this part would take longer.) Now it was obvious that the sequential version was not I/O bound. For confirmation, a back-of-the envelope comparison with disk transfer and seek times confirmed that there was enough I/O bandwidth for reading multiple files in parallel. However, we still needed a balanced work distribution. After trying a distribution that took file sizes into account, we found that simply assigning files round-robin was the fastest approach. Given k term extractors, the filename generator fills k vectors with filenames in round-robin fashion. Each term extractor then processes its private vector of filenames without any interference or synchronization. Running the filename generator concurrently with the term extractors proved to be highly inefficient, because of a pair of lock operations for every filename inserted or removed from a shared buffer.

The most difficult part concerned the interaction with the index. With a few tests, it became clear that having a single index for all threads was not always a good choice, but we didn't know what the right balance was. It was unclear not only how many threads to use, but also whether the filename generation and the index update should use the same or a different thread count. There are multiple system properties that affect this choice. Filename generation and term extraction largely depend on the number of drives and disk performance. The more drives and the better their performance, the more threads can be employed to scan files. Index update, on the other hand, is mainly influenced by the clock rate of the CPU. A fast CPU can write more terms into the index, so we need fewer threads for this part. The number of available processors, the scheduling policies of the operating system and the size of the caches affect the total number of threads that can be used profitably. Using too many threads typically slows

an application down. Only experimentation would answer these questions. The next section provides some of the data points. In some of the experiments, we used the auto-tuner by Schäfer et al. [2], but couldn't use it throughout, because this auto-tuner was built for C#, while our implementation was written in C++ for extra speed.

The problem of how to handle term duplicates was not answered by measurement, but by analysis. The question was whether each term extractor should implement a private index for eliminating duplicates, or whether term extractors should insert terms immediately (and potentially repeatedly) into a shared index. The latter solution would be similar to the distributed map-reduce implementation in [7]. But we thought that the former solution had higher performance potential. The lookup time would be about the same for both methods. However, the shared index must also store the filename associated with the term. This requirement means that once a term has been looked up in the index, a search must check whether the pair (term, filename) had been added previously (duplicate). This (linear) search for duplicates is eliminated entirely if the term extractor first builds a list of terms for a given file, without duplicates. This list is then entered en bloc, with the filename added. Since each file is scanned exactly once, we need not check whether the filename already exists in this case. We chose to implement this approach in all configurations. This choice also has the benefit of passing large chunks of data from term extractor to index, which reduces the number of buffering and locking operations. Perhaps the distributed map-reduce implementation of index generation would also benefit from this technique.

We implemented the index with a hash map provided by the Boost C++ Library. The duplicate elimination in the term extractors uses a hash set. Both data structures use the FNV1 hash function [8] to calculate the hash values.

4 Performance Results

The following three alternative designs of the index generator were compared:

Design 1: Use a single shared index and lock it on update.
Design 2: Replicate the shared index and join the replicates at the end.
Design 3: Same as Design 2, but don't join indices (because the search can work with multiple indices in parallel).

We ran implementations of those designs on three systems: A 4-core Intel machine (Intel Core2Quad Q6600, 2.4 GHz, 4 GB RAM, Windows 7 64 bit), a 8-core Intel machine (Intel Xeon E5320, 1.86 GHz, 8 GB RAM, Ubuntu 8.10 64 bit), and a 32-core Intel machine (Intel Xeon X7560, 2.27 GHz, 8 GB RAM, RHEL 4 64 bit). Each of the implementations was run using different numbers of threads for term extraction, index update, and index joining, as discussed in section 2. Any combination of thread counts – for example Design 2 running with 3 threads for term extraction, 3 threads for index update, and 1 thread for

joining indices – was run 5 times on each system. We report the averages per platform. The efficiency presented in Tables 2, 3, and 4 is defined as

$$E(p) = \frac{S(p)}{p}$$

with $S(p)$ is the achieved speed-up using p cores and

$$p = \min(\text{Number of Threads}, \text{Number of Cores}).$$

The sequential implementation on the 4-core machine takes about 220 seconds. All three parallel designs achieve nearly the same speed-up of about 4.7 (see Table 2). Note the super-linear speed-up, probably due a larger effective cash.

Table 2. Execution time (seconds), speed-up, and efficiency for the best configurations on the 4-core machine. Each configuration tuple (x, y, z) describes the number of threads used in term extraction, index update, and index join, resp.

	4-core Intel machine			
	config.	**exec. time**	**speed-up**	**efficiency**
Sequential	-	220.0	-	-
Design 1	(3, 1, 0)	46.7	4.71	1.18
Design 2	(3, 5, 1)	46.9	4.70	1.18
Design 3	(3, 2, 0)	46.4	4.74	1.19

The 8-core machine executes the sequential implementation in about 105 seconds, which is almost twice as fast as the 4-core machine. The parallel designs achieve different speed-ups as shown in Table 3. Design 1 takes the most time to execute whereas design 3 achieves the best speed-up of about 2.12 on this machine. For all three, there are more term extractor threads than index threads, compensating for the slower speed of term extraction.

On the 32-core machine, the sequential implementation takes about 90 seconds, which is significantly faster than on the 4-core machine. The reason for the runtime difference may be the better I/O performance. In contrast to the 4-core system, the parallel designs achieve different speed-ups, see Table 4. The performance results for this system show that design 1 takes longest to execute with a speed-up of about 1.96, while design 3 achieves a speed-up of 3.5. Because more processors are being used, efficiency is down.

What would happen if all three machines used the same configurations? Table 5 shows the results when using the configurations that proved best on the 4-core computer. Not surprsingly, designs 1 and 2 on the 32-core machine now differ by only 1 second from the 4-core machine; design 3 takes 4.5 seconds less than on

Table 3. Execution time (seconds), speed-up, and efficiency for the best configurations on the 8-core machine. Each configuration tuple (x, y, z) specifies the number of threads used in term extraction, index update, and index join, resp.

	8-core Intel machine			
	config.	**exec. time**	**speed-up**	**efficiency**
Sequential	-	105.0	-	-
Design 1	(3, 2, 0)	59.5	1.76	0.35
Design 2	(6, 2, 1)	57.7	1.82	0.20
Design 3	(6, 2, 0)	49.5	2.12	0.26

Table 4. Execution time (seconds), speed-up, and efficiency for the best configurations on the 32-core machine. Each configuration tuple (x, y, z) specifies the number of threads used in term extraction, index update, and index join, resp.

	32-core Intel machine			
	config.	**exec. time**	**speed-up**	**efficiency**
Sequential	-	90.0	-	-
Design 1	(8, 4, 0)	45.9	1.96	0.16
Design 2	(8, 4, 1)	36.4	2.47	0.19
Design 3	(9, 4, 0)	25.7	3.50	0.27

the 4-core machine. On the 8-core machine, all three designs run 14 to 28 seconds slower than on the other two machines. By comparison with the preceding tables, the optimal configurations for the 8 and 32 core platforms run significantly faster, so the choice of the optimal configuration matters. One configuration does not fit all platforms!

Table 5. Comparison of the execution time (seconds) and speed-up for identical configurations on the 4-core, 8-core, and 32-core Intel machine. The last column shows the execution time differences (expressed as a percentage) of the 8-core machine and 32-core machine compared to the 4-core machine.

	4-core vs. 8-core vs. 32-core Intel machine			
	config.	**exec. time**	**speed-up**	**exec. time difference (%)**
Design 1	(3, 1, 0)	46.7; 61.1; 47.2	4.71; 1.71; 1.91	+30.84; +1.07
Design 2	(3, 5, 1)	46.9; 75.2; 47.3	4.70; 1.39; 1.90	+2.13; +0.85
Design 3	(3, 2, 0)	46.4; 61.5; 41.9	4.74; 1.71; 2.15	+32.54; -9.70

5 Lessons Learned and Conclusion

The main lessons from this case study is that even in a simple application such as index generation, the choreography of the various parallel steps is important. There are typically many ways to parallelize a given application, and their performance characteristics can be quite different. Moreover, there is no such thing as the optimal parallel configuration for all platforms. A single, fixed configuration that runs optimally on one platform might run sub-optimally on others, especially when the number and speed of the cores and I/O performance differ.

To arrive at a fast parallel design, one should proceed as follows:

1. Use benchmarks and measurement to identify the components with the highest parallelization potential.
2. Beware of bottlenecks, such as I/O operations and shared data structures with locks.
3. Develop alternative parallel designs, especially those that reduce or eliminate locking.
4. Use back-of-the-envelope analysis with data from 1. to explore alternatives.
5. Experiment with alternatives, where necessary. In particular, test different thread allocations.
6. Use an auto-tuner to speed up exploring the design space.

We presented a rational design process; however this is not how it really happened. We went through a number of dead ends caused by some of the reasons pointed out by Parnas et al. [9]: A lot of design details emerged while implementing the application, and we were influenced by design ideas from previous experience. But presenting a rational process is beneficial nevertheless, as Parnas pointed out [9]:

> "Those who read the software documentation want to understand the programs, not to relive their discovery. By presenting rationalized documentation we provide what they need."

By contributing this case study we hope to help make parallel software design a more rational and goal-oriented process.

In the future we will analyze how to integrate the search query functionality and parallelize it as well, for instance by using multiple indices. Better work distribution strategies, more file formats, larger benchmarks, and more platforms are additional work items as well as the optimization at runtime.

Acknowledgments. We thank Dr. Victor Pankratius for many fruitful discussions and Christoph Schäfer for experiments with his auto-tuner. We also thank Intel for making the Manycore Testing Lab available.

References

1. Pankratius, V., Jannesari, A., Tichy, W.F.: Parallelizing BZip2: A Case Study in Multicore Software Engineering, IEEE Softw., vol. 26, 70–77, IEEE Computer Society Press, Los Alamitos, CA, USA (2009)

2. Schäfer, C.A., Pankratius, V., Tichy, W.F.: Engineering Parallel Applications with Tunable Architectures, ICSE 2010, 405–414, ACM (2010)
3. Pankratius, V., Adl-Tabatabai, A., Otto, F.: Does Transactional Memory Keep Its Promises? Results from an Empirical Study, Technical Report 2009-12, IPD, University of Karlsruhe, Germany (September 2009)
4. Intel Manycore Testing Lab, http://software.intel.com/en-us/articles/intel-manycore-testing-lab/ (March 2010)
5. Agrawal, N., Bolosky, W.J., Douceur, J.R., Lorch, J.R.: A five-year study of file-system metadata, Trans. Storage, vol. 3, ACM, New York, NY, USA (2007)
6. Douceur, J.R., Bolosky, W. J.: A Large-Scale Study of File-System Contents, SIGMETRICS '99, vol. 27, 59–70, ACM, New York, NY, USA (1999)
7. Dean J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, Commun. ACM, vol. 51, 107–113, ACM, New York, NY, USA (2008)
8. Noll, L.C.: FNV hash http://isthe.com/chongo/tech/comp/fnv/ (March 2010)
9. Parnas, D.L., Clements, P.C.: A rational design process: How and why to fake it. IEEE Trans. Softw. Eng., vol. 12, 251–257, IEEE Press, Piscataway, NJ, USA (1986)