Karlsruhe Reports in Informatics 2010,9
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Parallelizing an Index Generator for Desktop Search

David J. Meder ; Walter F. Tichy

2010

Fakultät für **Informatik**

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 were compared on Intel platforms with 4, 8 and 32 cores. The optimal configurations were not intuitive and markedly different for the three platforms. For finding the optimum, detailed measurements and experimentation were necessary. Several recommendations for parallel software design follow from this case study.

1 Introduction

Developing multi-threaded applications for multicore computers is challenging. Since little knowledge about parallelizing non-numeric applications is available, it is appropriate at this time to perform case studies about parallelizing a wide range of applications. By documenting these case studies, such as the BZip2 case study [4] or the examples covered in [7], a rational process for parallel software design might emerge. In this vain, we conducted and documented an in-depth study of parallelizing desktop search. Desktop search is performed on a PC, laptop, smart phone, 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 for every term the files in which the 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 many 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. [5] conducted a competition among student teams to parallelize 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 not comparable, because the teams stored different amounts of data in the

Technical Report 2010-9, Institute for Program Structures and Data Organization (IPD), University of Karlsruhe, Germany, May 11, 2010.

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 present 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 [2].

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 which could be parallelized independently or in combination.

Filename generation: Traverse the directory hierarchy to generate the names of the files to be indexed (Stage 1).
Term extraction: Scan files and extract terms (Stage 2).
Index update: Add the extracted terms to an index structure (Stage 3).

At the outset of the project, we faced a number of questions: Which of those stages 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 stages 1 and 3.

2.1 Stage 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 Stage 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 in stage 2 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 Stage 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 various stages. The first step was to set up a benchmark. It consists of about 51.000 ASCII text files, containing many small files and five large text files. 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 stage 1, which would generate the complete set of filenames in main memory before starting term extraction.

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 77-80

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

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 safety, 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 generated and consumed.

The most difficult part was 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. Only experimentation would answer this question. The next section provides some of the data points. In some of the experiments, we used the auto-tuner by Schäfer et al. [7], 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 [1]. But we thought that the former solution had a 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 in turn 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 enters the list of terms per file en bloc, without duplicates: 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 [3] to calculate the hash values.

4 Performance Results

The following three alternative implementations of the index generator have been compared:

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

We ran those implementations 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 Implementation 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 sequential implementation on the 4-core machine takes about 220 seconds. All three parallel implementations achieve nearly the same speed-up of about 4.7 (see Table 2).

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

	4-core Intel machine			
	best config.	**exec. time (s)**	**speed-up**	**variance**
Sequential	-	220.0	-	-
Implementation 1	(3, 1, 0)	46.7	4.71	0.0%
Implementation 2	(3, 5, 1)	46.9	4.70	-0.21%
Implementation 3	(3, 2, 0)	46.4	4.74	+0.85%

The 8-core machine executes the sequential implementation in about 105 seconds which is almost twice as fast as on the 4-core machine. The different implementations achieve different speed-ups as shown in Table 3. Implementation

1 takes the most time to execute whereas Implementation 3 achieves the best speed-up of about 2.12 on this machine.

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

	8-core Intel machine			
	best config.	**exec. time (s)**	**speed-up**	**variance**
Sequential	-	105.0	-	-
Implementation 1	(3, 2, 0)	59.5	1.76	0.0%
Implementation 2	(6, 2, 1)	57.7	1.82	+3.4%
Implementation 3	(6, 2, 0)	49.5	2.12	+16.5%

On the 32-core machine, the sequential implementation takes about 90 seconds, which is significantly faster than on the 4-core machine. In contrast to the 4-core system, the different implementations achieve different speed-ups as shown in Table 4. The performance results for this system show that Implementation 1 takes longest to execute with a speed-up of about 1.96 while Implementation 3 achieves a total speed-up of 3.5.

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

	32-core Intel machine			
	best config.	**exec. time (s)**	**speed-up**	**variance**
Sequential	-	90.0	-	-
Implementation 1	(8, 4, 0)	45.9	1.96	0.0%
Implementation 2	(8, 4, 1)	36.4	2.47	+26.0%
Implementation 3	(9, 4, 0)	25.7	3.50	+78.6%

5 Lessons Learned and Conclusion

There are typically numerous ways to parallelize an application, and index generation is no exception. To arrive at a fast parallel implementation, one should proceed as follows:

1. Use benchmarks and measurements 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.
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 development 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. [6]: 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 [6]:

> "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.

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. Dean J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, Sixth Symposium on Operating Systems Design and Implementation (2004)
2. Intel Manycore Testing Lab, http://software.intel.com/en-us/articles/intel-many-core-testing-lab/ March 2010.
3. Noll, L.C.: FNV hash http://isthe.com/chongo/tech/comp/fnv/ March 2010
4. Pankratius, V., Jannesari, A., Tichy, W.F.: Parallelizing BZip2: A Case Study in Multicore Software Engineering, IEEE Software, 70-77, November 2009
5. Pankratius, V., Adl-Tabatabai, A., Otto, F.: Does Transactional Memory Keep Its Promises? Results from an Empirical Study. Technical Report 2009-12, University of Karlsruhe (2009)
6. Parnas, D.L., Clements, P.C.: A rational design process: How and why to fake it. IEEE Trans. Softw. Eng. 12, 251–257 (1986)
7. Schäfer, C.A., Pankratius, V., Tichy, W.F.: Engineering Parallel Applications with Tunable Architectures. International Conference on Software Engineering (2010)