A High-throughput Computing Scheme on Many-core Processors

Yiqing Zhu Shunsuke Takano Wei Chen
Mizuki Oka Kazuhiko Kato

This paper proposes a high-throughput computing scheme for many-core processors: cave and court computing (CCC). To achieve high concurrency, this scheme allows data inconsistency in order to perform lock-free operations in the *cave* mode and merge the results to solve the data inconsistency in the *court* mode. The scheme supports applications demanding high throughput such as big-data processing. We study CCC with B-Tree to perform a simple big-data process: counting the occurrence of each word. By evaluating the result, we show this scheme has an acceptable performance and specific scalability. We argue that given careful implementation, the CCC scheme is competent for high-throughput computing.

1 Introduction

Recently, big-data researchers have mainly focused on how to scale out big-data processing across a large number of servers. However, big-data-oriented software schemes for high-throughput computing on a single computer have not been studied much. Making good use of many CPUs has become an important issue since processor industry is experiencing a transition from the exponential increase in per-core CPU speed to the exponential increase in the number of cores. By taking advantage of many-core, we believe applying thread-level parallel processing in a single server obviously enjoys many benefits such as lower complexity and lower price.

In this paper, we propose a high-throughput processing software scheme on many-core, which we call cave and court computing (CCC). In our CCC scheme, threads process data in private memory without any data synchronization in the *cave* mode, to increase concurrency, and merge the results into public memory to solve data inconsistency in the *court* mode. During the execution, CCC switches between cave mode and court mode. Combining the CCC scheme with an appropriate data structure such as B-Tree supports high concurrent operations. We propose this novel scheme greatly increases the use of many-core, thus achieving high throughput, making the scheme appropriate for big-data processing.

The rest of the paper is organized as follows. In Section 2, the research background and related work are introduced. In Section 3, a brief introduction is given to the preliminary knowledge. In Section 4, an overview of CCC is provided, while in Section 5, the CCC implementation on B-Tree is described. In Section 6, the experiment results are presented. In Section 7, the paper is concluded and future work is discussed.

A High-throughput Computing Scheme on Many-core Processors
Zhu, Takano, Chen, Oka, Kato, , Dept. of Information and Computer Science, University of Tsukuba.

2 Background and Related Work

The amount of data has been exploding. Analyzing big data, is becoming an inscreasingly important basis of competition. Since big data is very large and complex, it requires many techniques and experiments to process. A common solution in industry recently is to scale out big-data processing among a large number of servers because the computational ability of a single computer is limited. Many mature programming models or schemes have been proposed. For example, MapReduce [4], which was first developed by Google in 2004, is based on this " scaling out " approach.

In the last decade, the processor industry has expanded. The increase in computational ability makes it theoretically possible to process big data on a single server. The benefits are obvious, such as lower computational complexity and lower maintenance cost. However, the per-core CPU speed is reaching a physical limit, as Herb Sutter stated in 2005 [8], which indicates this increase cannot be translated directly into faster computation. Meanwhile, increasing the number of cores exponentially is the current focus of the processor industry rather than the pursuit of faster per-core speed. This change had a profound impact on software performance, which reaps the benefits of multicores. Therefore the traditional multi-thread approach is worth a second look for big-data processing jobs.

Parallelizing the big-data process requires very high throughput to keep the process within a tolerable time. In thread-level parallelizing, concurrency occurs in the shared memory. High-throughput concurrency control requires important features. For example, the concurrency should be extremely high, no rollback or wasted work occurs if possible, and little copying memory if possible.

A MapReduce library in shared memory such as Phoenix [7] demonstrated the MapReduce model can perform competitively with those written with Pthreads. Further work on Phoenix like [9] and [3] provided different optimization on it. Other MapReduce implementation such as Mars [6] shows that an implementation on GPU is also an alternative choice to cluster-based MapReduce. In addition to MapReduce, other GPU-focused studies provide other models or schemes on the GPU for data processing.

3 Preliminary

3.1 Concurrency Control Mechanisms in Shared Memory

The traditional concurrency control mechanism in shared memory can be mainly divided into two categories: the pessimistic approach and the optimistic approach. A well-known example of the pessimistic approach is the lock. In the lock-based method, data consistency is always maintained during execution, by blocking the threads that may cause data inconsistency. The optimistic approach, for example, software transactional memory [5], assumes multiple threads can be completed without affecting each other. After a piece of code is executed, the transactional memory library checks the data consistency and rolls back if it finds inconsistent data.

Unfortunately, neither approach offers high throughput on many-core computers. The limitation of the pessimistic approach is that it blocks threads thus decreasing the concurrency; the limitation of the optimistic approach is that when it finds inconsistent data, it has to roll back, which is a big waste.

3.2 B-Tree

B-Tree is a type of balanced tree in indexing data structure, which efficiently supports operations, such as search, insertion or deletion, in logarithmic time. B-Tree was first introduced by Bayer

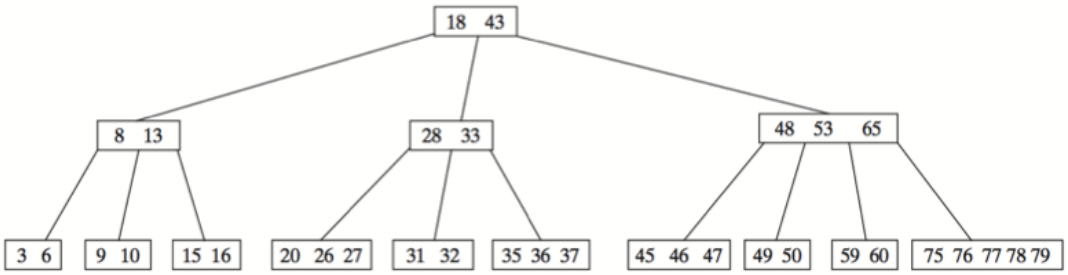

図 1 An example of a B-Tree of order 6.

and McCreight [1] and is often used to store data in secondary storage to reduce disk I/O. A notable property of B-Tree is that all leaf nodes are in the same level. Figure 1 shows a typical B-Tree of order 6, where keys are represented as integers.

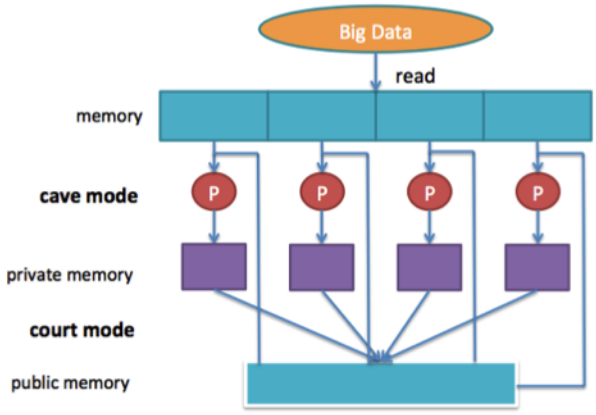

図 2 Overview of the cave and court computing scheme.

4 Cave and Court Computing Scheme

The limitation of traditional approaches comes from the fact that the pessimistic approach and the optimistic approach require extra work to handle data consistency. Whether blocking threads or rolling back, these two approaches try to maintain in a data-consistent condition. However, in big-data processing, people seldom care about intermediate conditions. Since the scope of big-data processing is much smaller, we can allow data inconsistency to occur during the execution. Without data synchronization, very high concurrency is possible. Then the inconsistency is solved, that is the results are merged from asynchronous computing, to get a synchronized state.

Our main idea comes from the concept called “ cave and court ” in an office design. The term “cave” means a private working area for every individual in an office; “court” a means public meeting or discussion area. In design, an office with “cave and court” is a good design. This concept is based on the characteristics of human cooperation. We propose cave and court computing scheme, applying this method of cooperation to one of the threads.

Cave and court computing has two execution modes, the cave mode and the court mode. In the cave mode, every thread performs normal operations high-concurrently without any data synchronization, saving the results in private regions. Most of the time, the threads are in the the cave mode. In the court mode, the threads do not perform normal operations, but merge the results in the private regions. Execution switches between the cave mode and the court mode during the execution. The cave-court circle loops hundreds or

thousands of times in a processing application, processing data asynchronously with high throughput in the cave mode and then merging the results in the court mode. The merged results in the court mode can be used for future computation in the next circle.

5 Cave and Court Computing with B-Tree

Unlike MapReduce, which only works on a particular data structure, the key/value pair, the CCC scheme can be easily combined with commonly used data structures. In this research, we implement the CCC scheme with B-Tree.

Figure 3 shows the structure of the cave and court computing with B-Tree. Different from standard B-Tree, in each leaf node, there are separate private regions for each thread. The private regions are invisible to other threads, so that, in the cave mode, threads perform read operations throughout the whole tree and write operations only in one 's shared mmory even though data inconsistency may occur in the cave mode.

Search: In search, a thread searches the public memory, which includes the internal nodes and public regions of the leaf nodes, as well as the private regions corresponding to the thread in the leaf nodes.

Insert: In insertion, a thread finds the proper leaf node according to the B-Tree semantics. The thread inserts the key into the private region in this leaf node. The new keys are merged in the public region in the court mode.

In the court mode, threads pause normal operations to merge the results in each leaf node. We name this operation " reorganize ".

Reorganize: In reorganization, keys both in the private regions and the public region of one node are sorted together and then put in the node. Similar to the normal insert operation on standard B-Tree, nodes may split because of node overflow. The splitting strictly follows B-Tree semantics but performs differently with normal B-Tree because there can be many keys in the private region such that the original node may split into more than two nodes. Concurrent reorganization is possible and necessary for high throughput. However the traditional lock algorithm [2] on B-Tree cannot maintain high throughput, because so many new leaf nodes must be produced that reorganization on a leaf node may affect very upper-level internal nodes, or even the root. As a consequence, the locks on the upper-level nodes are difficult to release.

In the court mode, almost all the node in the B-Tree are modified and many internal nodes would be modified more then once due to the large amount of splitting in their children. To improve the concurrency and efficiency, we use a bottom-up method to perform an "accumulating" reorganization level up level rather than node-by-node reorganization separately. In detail, all the nodes in the same level are reorganized at the same time, thus merging keys and splitting nodes without modification on the parents. After this level is completed, the new produced nodes are used for reorganization in next level up to it.

6 Experiments

We apply our CCC scheme to process big data on a rather simple example: the word count. The word-count program counts the occurrence for each word in a big file. The key/value pairs of (word, count) are stored in the cave and court B-Tree in sorted order. The B-Tree acts as the index of words occurring.

We perform our experiment on a Dell PowerEdge R815. It has four 16-core 2.1GHz AMD Opeteron 6272 processors, that is, 64 cores in total, and 64G memory. We used the Wikipedia English version data with all the non-alphanumeric characters removed. The data is about 36G.

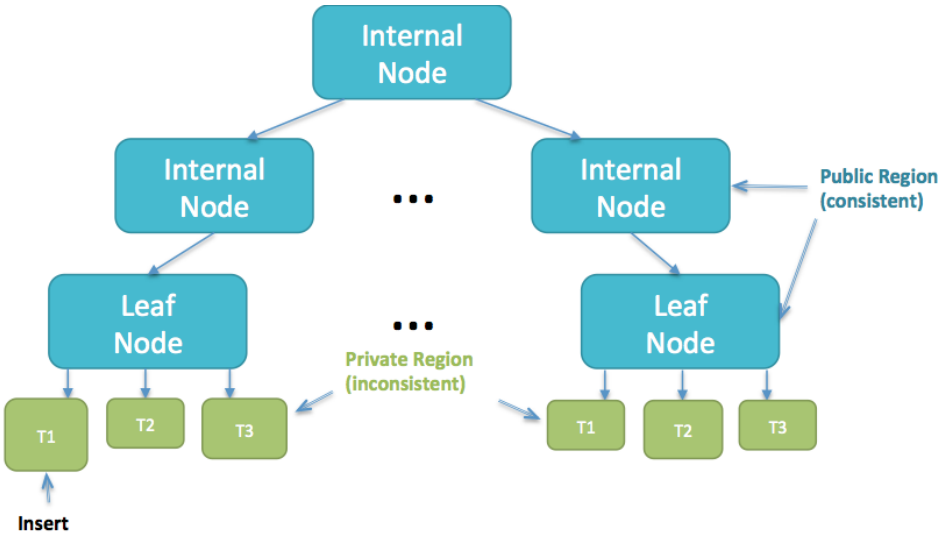

図 3 Structure of the cave and court computing with B-Tree.

Figure 4 is the experimental result. The x-axis is the number of threads and the y-axis is the time required to complete the task. The word count job can be completed in about 30 minutes, an acceptable result considering the lower computational complexity and maintenance cost compared to processing data across a large number of servers, typically on a cluster-based MapReduce implementation. However, the performance peaks when the number of threads exceeds 24. One possible bottlenecks originates from the IO bandwidth. That is, in the cave mode, threads perform lock-free operations so fast that they have to wait for the read-file thread to fill in the input buffer.

7 Conclusion and Future Work

In this paper, we propose a novel software scheme on many-core processors called cave and court computing (CCC). By switching between the cave mode and the court mode, this scheme allows data inconsistency in order to perform lock-free operations in the cave mode, and merge the results to solve the data inconsistency in the court mode. The scheme supports high throughput applications such as big-data processing. We first introduced the basic concept of the CCC scheme. Then we introduced our CCC implementation with B-Tree, as the search, insertion and reorganization algorithm. Finally, we performed a word count experiment on Wikipedia data.

The current experimental results shows the scheme scales out until the number of threads is equal to 24 and saturates when the number is greater than 24. We assume this is due to the I/O bandwidth limitation. In future work, We will examine our implementation more closely to check whether there is room for achieving a better performance. Last, we want to compare our software scheme to the Amazon Elastic MapReduce service †1 with the same instance number as our core number to test if our proposed scheme is an alternative choice to MapReduce for big-data processing.

参 考 文 献

[1] Bayer, R. and McCreight, E.: *Organization and maintenance of large ordered indexes*, Springer,

†1 http://aws.amazon.com/jp/elasticmapreduce/

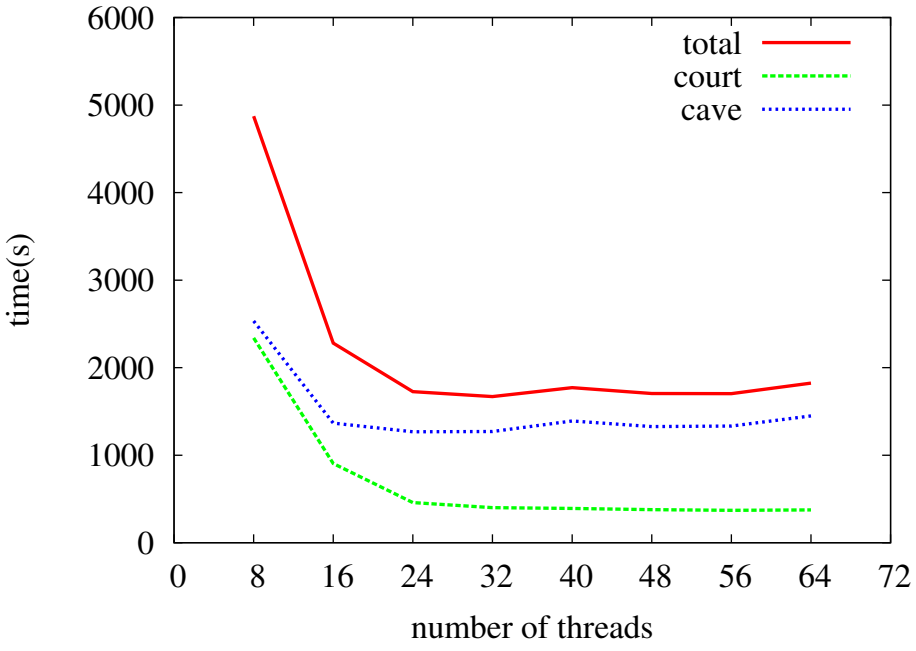

図 4 Wordcount time cost on English Wikipedia (36G). The "total" is the total time cost and the "cave" and the "court" are the cave mode time and court mode time, respectively.

2002.

[2] Bayer, R. and Schkolnick, M.: Concurrency of operations on B-trees, *Acta informatica*, Vol. 9, No. 1(1977), pp. 1–21.

[3] Chen, R., Chen, H., and Zang, B.: Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling, *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, New York, NY, USA, ACM, 2010, pp. 523–534.

[4] Dean, J. and Ghemawat, S.: MapReduce: a flexible data processing tool, *Communications of the ACM*, Vol. 53, No. 1(2010), pp. 72–77.

[5] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M.: Composable memory transactions, *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM, 2005, pp. 48–60.

[6] He, B., Fang, W., Luo, Q., Govindaraju, N. K., and Wang, T.: Mars: a MapReduce framework on graphics processors, *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ACM, 2008, pp. 260–269.

[7] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C.: Evaluating mapreduce for multi-core and multiprocessor systems, *IEEE 13th International Symposium on High Performance Computer Architecture*, IEEE, 2007, pp. 13–24.

[8] Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software, *Dr. Dobb 's Journal*, Vol. 30, No. 3(2005), pp. 202–210.

[9] Talbot, J., Yoo, R. M., and Kozyrakis, C.: Phoenix++: modular MapReduce for shared-memory systems, *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, New York, NY, USA, ACM, 2011, pp. 9–16.