IEEE distributed systems Expert-authored articles and resources ONLINE
September 2007 (vol. 8, no. 9), art. no. 0709-mds2007090003

Published by the IEEE Computer Society

Distributed Wisdom

Concurrency and the Principle of Data Locality

Hagit Attiya • Technion

Because clock frequency has hardly advanced in recent years, major chip manufacturers are shifting their focus from improving the speed of individual processors to increasing parallel-processing capabilities. *Multicore* technology refers to a processor with more than one engine, allowing for greater efficiency because the processor workload is essentially shared. With multicore and multiprocessing architectures becoming common, it's imperative to devise effective software tools for managing the difficulty of concurrent programming.

At the same time, applications must be designed to exploit parallelism and avoid the perils of sequential execution. To utilize the architecture's capabilities, it's critical to allow many operations to make progress concurrently and to complete without interference.

Multiword synchronization

A good example of the challenge in obtaining high throughput is *multiword synchronization* operations, such as *k*-compare-and-swap (*k*CAS). Such operations allow reading the contents of several memory locations, comparing them with specified values, and if they all match, updating the locations—all in one atomic operation. Multiword synchronization facilitates the design and implementation of concurrent data structures, making this process more effective and easier than when using only single-word synchronization.

In reality, however, today's multicore and multiprocessing architectures support in hardware only single-word synchronization operations such as CAS (compare-and-swap) or LL/SC (load-link, store-conditional), or at best, DCAS (double compare-and-swap). So, much research has focused on providing *k*CAS in software. In typical *k*CAS implementations, an operation tries to lock all the words it needs, one by one. If another operation already holds the lock on a word, the operation waits for the lock to be released, often while helping the conflicting operation make progress. If a third operation then holds a lock needed by the conflicting operation, the operation helps it as well.

Clearly, when several *k*CAS operations need to simultaneously access the same words, an inherent "hot spot" is created and operations must be delayed. A worse situation happens in these typical *k*CAS implementations when an operation's progress is also hindered because of operations that don't contend for the same memory words. The recursive helping in these schemes causes chains of operations, where each operation is waiting for the next operation in the chain. In these scenarios, an operation is delayed a number of steps proportional to the chain's total length, causing a lot of work to be invested while only a few operations complete.

Better throughput through shorter conflict chains

Understanding these notions is easier if we break the data into disjoint items—for example, elements of a linked list—and use a *conflict graph* to visualize the relations among operations that overlap in time. In this graph, nodes correspond to data items, and edges connect data items if they're accessed by the same operation. So, two simultaneous operations that contend for the same data item have adjacent edges. Many synchronization algorithms guarantee that operations delay each other if and only if a path exists between them in the conflict graph. That is, operations proceed in parallel if they

access disjoint parts of the data structure. This property, called *disjoint access parallel*, essentially lets operations help or delay each other only if a path exists between them. The definition is in fact more specific; each operation may take a number of steps proportional to the total number of operations in its connected component.

One way to bound the lengths of delay chains and improve the concurrency of multiword synchronization is to restrict these paths' lengths. However, even when operations randomly choose the words they access, these paths' lengths depend on the total number of operations, and paths of significant length might be created in the conflict graph (as shown by Phuong Hoai Ha and his colleagues).[1] This means that the connected components have a nonconstant diameter, implying that "distant" operations can delay an operation.

The adverse effect of waiting and delay chains can be mitigated, greatly improving the concurrency, if operations are delayed only because of operations that are at a constant distance from them. This means that operations accessing distant data don't interfere, even if they're connected (but at a nonconstant distance). So, the algorithm's throughput is localized in components of constant diameter, which are effectively isolated from operations at a larger distance.

Measuring concurrency

You can use various measures to evaluate how much concurrency an algorithm provides. For algorithms that might block when an operation fails, Manhoi Choy and Ambuj Singh suggested evaluating the *failure locality*—namely, to what distance (in the conflict graph) an operation's failure can prohibit other operations from completing.[2] When the algorithm is *nonblocking*, you can measure its locality by the diameter in which it's guaranteed that some operation completes, provided that operations in this diameter take enough steps. Yehuda Afek and his colleagues suggested two quantitative measures that bound the distance among operations that influence each other's step complexity or access the same (low-level) memory word.[3]

Despite the importance of improving throughput by increasing concurrency, few algorithms have provable (and good) locality properties. Eyal Dagan and I showed how to simulate DCAS in software from CAS, with $O(\log^* n)$-locality properties.[4] Building on this algorithm, Afek and his colleagues presented a wait-free kCAS algorithm, for a fixed k, with $O(k + \log^* n)$-locality properties.[3] Recently, Eshcar Hillel and I implemented a *doubly linked list*, which allows concurrent insertions and deletions as long as they're at least three operations apart (including at the ends).[5]

This area offers many opportunities for interesting research. Besides further development of specific concurrent data structures and applications with good locality properties, three other fundamental challenges exist. The first is to understand the problem: how delay chains are created and how they affect concurrency and locality. The second is to develop an accompanying theory by devising appropriate measures and finding algorithmic ideas, perhaps even proving some lower bounds. Last, but certainly not least, it's imperative to estimate these measures' and algorithms' practical relevance.

References

1. P.H. Ha et al., "Efficient Multi-word Locking Using Randomization," *Proc. 24th Ann. ACM Symp. Principles of Distributed Computing* (PODC 05), ACM Press, 2006, pp. 249–257.
2. M. Choy and A.K. Singh, "Efficient Fault-Tolerant Algorithms for Distributed Resource Allocation," *ACM Trans. Programming Languages and Systems*, vol. 17, no. 3, 1995, pp. 535–559.
3. Y. Afek et al., "Disentangling Multi-object Operations," *Proc. 16th Ann. ACM Symp. Principles of Distributed Computing* (PODC 97), ACM Press, 1997, pp. 111–120.
4. H. Attiya and E. Dagan, "Improved Implementations of Binary Universal Operations," *J. ACM*, vol. 48, no. 5, 2001, pp. 1013–1037.
5. H. Attiya and E. Hillel, "Built-In Coloring for Highly-Concurrent Doubly-Linked Lists," *Proc. 20th Int'l Symp. Distributed Computing* (DISC 06), LNCS 4167, Springer, 2006, pp. 31–45.

Hagit Attiya is a professor in the Technion's Department of Computer Science. Contact her at (mailto:hagit@cs.technion.ac.il).

Related Links

- DS Online's Parallel Processing Community
- "Reactive Multi-Word Synchronization for Multiprocessors," *Proc. 12th Int'l Conf. Parallel Architectures & Compilation Techniques* (http://doi.ieeecomputersociety.org/10.1109/PACT.2003.1238014)
- "Efficient Wait-Free Implementation of Multiword LL/SC Variables," *Proc. 25th IEEE Int'l Conf. Distributed Computing Systems* (http://doi.ieeecomputersociety.org/10.1109/ICDCS.2005.29)
- "High Performance Computing in the Multi-core Area," *Proc. 6th Int'l Symp. Parallel & Distributed Computing* (http://doi.ieeecomputersociety.org/10.1109/ISPDC.2007.28)

Cite this article:
Hagit Attiya, "Concurrency and the Principle of Data Locality," *IEEE Distributed Systems Online*, vol. 8, no. 9, 2007, art. no. 0709-o9003.