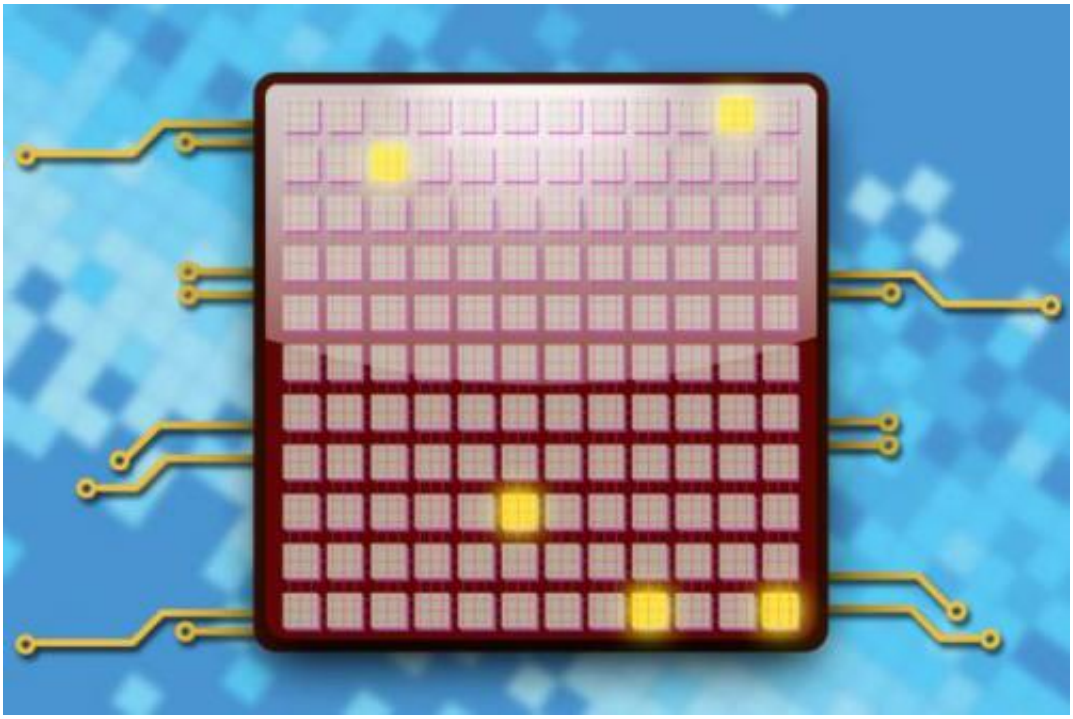
Researchers parallelize a common data structure to work with multicore chips

January 30 2015, by Larry Hardesty



Credit: Christine Daniloff/MIT

Every undergraduate computer-science major takes a course on data structures, which describes different ways of organizing data in a computer's memory. Every data structure has its own advantages: Some are good for fast retrieval, some for efficient search, some for quick insertions and deletions, and so on.

Today, hardware manufacturers are making computer chips faster by giving them more cores, or processing units. But while some data structures are well adapted to multicore computing, others are not. In principle, doubling the number of cores should double the efficiency of a computation. With algorithms that use a common data structure called a priority queue, that's been true for up to about eight cores—but adding any more cores actually causes performance to plummet.

At the Association for Computing Machinery's Symposium on Principles and Practice of Parallel Programming in February, researchers from MIT's Computer Science and Artificial Intelligence Laboratory will describe a new way of implementing priority queues that lets them keep pace with the addition of new cores. In simulations, algorithms using their data structure continued to demonstrate performance improvement with the addition of new cores, up to a total of 80 cores.

A priority queue is a data structure that, as its name might suggest, sequences data items according to priorities assigned them when they're stored. At any given time, only the item at the front of the queue—the highest-priority item—can be retrieved. Priority queues are central to the standard algorithms for finding the shortest path across a network and for simulating events, and they've been used for a host of other applications, from data compression to network scheduling.

With multicore systems, however, conflicts arise when multiple cores try to access the front of a priority queue at the same time. The problem is compounded by modern chips' reliance on caches—high-speed memory banks where cores store local copies of frequently used data.

"As you're reading the front of the queue, the whole front of the queue will be in your cache," says Justin Kopinsky, an MIT graduate student in electrical engineering and computer science and one of the new paper's co-authors. "All of these guys try to put the first element in their cache

and then do a bunch of stuff with it, but then somebody writes to it, and it invalidates everybody else's cache. And this is like an order-of-magnitude slowdown—maybe multiple orders of magnitude."

Loosening up

To avoid this problem, Kopinsky; fellow graduate student Jerry Li; their advisor, professor of computer science and engineering Nir Shavit; and Microsoft Research's Dan Alistarh, a former student of Shavit's, relaxed the requirement that each core has to access the first item in the queue. If the items at the front of the queue can be processed in parallel—which must be the case for multicore computing to work, anyway—they can simply be assigned to cores at random.

But a core has to know where to find the data item it's been assigned, which is harder than it sounds. Data structures generally trade ease of insertion and deletion for ease of addressability. You could, for instance, assign every position in a queue its own memory address: To find the fifth item, you would simply go to the fifth address.

But then, if you wanted to insert a new item between, say, items four and five, you'd have to copy the last item in the queue into the first empty address, then copy the second-to-last item into the address you just vacated, and so on, until you'd vacated address five. Priority queues are constantly being updated, so this approach is woefully impractical.

An alternative is to use what's known as a linked list. Each element of a linked list consists of a data item and a "pointer" to the memory address of the next element. Inserting a new element between elements four and five is then just a matter of updating two pointers.

Road less traveled

The only way to find a particular item in a linked list, however, is to start with item one and follow the ensuing sequence of pointers. This is a problem if multiple cores are trying to modify data items simultaneously. Say that a core has been assigned element five. It goes to the head of the list and starts working its way down. But another core is already in the process of modifying element three, so the first core has to sit and wait until it's done.

The MIT researchers break this type of logjam by repurposing yet another data structure, called a skip list. The skip list begins with a linked list and builds a hierarchy of linked lists on top of it. Only, say, half the elements in the root list are included in the list one layer up the hierarchy. Only half the elements in the second layer are included in the third, and so on.

The skip list was designed to make moving through a linked list more efficient. To find a given item in the root list, you follow the pointers through the top list until you identify the gap into which it falls, then move down one layer and repeat the process.

But the MIT researchers' algorithm starts farther down the hierarchy; how far down depends on how many cores are trying to access the root list. Each core then moves some random number of steps and jumps down to the next layer of the hierarchy. It repeats the process until it reaches the root list. Collisions can still happen, particularly when a core is modifying a data item that appears at multiple levels of the hierarchy, but they become much rarer.

More information: "The SprayList: A Scalable Relaxed Priority Queue": www.mit.edu/~jerryzli/SprayList-CR.pdf

This story is republished courtesy of MIT News

(web.mit.edu/newsoffice/), a popular site that covers news about MIT research, innovation and teaching.

Provided by Massachusetts Institute of Technology