Why is uRiKA So Fast on Graph-Oriented Queries?

Posted on **November 14, 2012** by **david**

Amar

If you look at some of the benchmarks we've run, the uRiKA system's performance really stands out when the structure of the SPARQL query is particularly complex. One example is the triangular query structure of Query 9 in the LUBM benchmark queries: "Find all graduate students who attend a class taught by their advisor." As simple as this query may look, it is one of the more complex of the LUBM queries, and it runs slowly on many existing SPARQL query engines, because of the three-way connection, from students to classes to professors and back to students, that must be established among the intermediate results. While many query engines bog down on this query, uRiKA blasts through it in a few seconds, and it scales very close to linearly as you increase the dataset size.

The uRiKA system is built around a parallel supercomputer architecture, previously known as the Cray XMT. Is this performance advantage due to parallelism? No, other systems we compare against also use multiple processors. uRiKA is a memory-resident database – is it the absence of file I/O during the processing of a query that gives us a performance advantage? Well, that certainly helps performance, but it doesn't explain why we're faster than other parallel systems that also have enough memory to keep the database memory-resident. What's the difference, then? It boils down to the processor architecture.

A little background: the uRiKA system is made up of two kinds of processor boards: the "service nodes" are pretty conventional: AMD Opterons running Linux. The "compute nodes" contain Cray custom-designed processors – each node consists of a "Threadstorm IV" processor, directly connected to up to 64GB of RAM, and all are connected in a Cray custom-designed 3D torus network. The service nodes support uRiKA's user interface, launch jobs on the system, control the file system, and other system-management tasks. The compute nodes contain the database and run the query engine.

The Threadstorm IV processor is the secret sauce. The supercomputer's original name, XMT, stood for "eXtreme MultiThreading." Threadstorm is heavily multithreaded. Each Threadstorm processor has 128 hardware copies of the register set, program counter, stack pointer, etc. necessary to hold the current state of one software thread that is executing on the processor. The processor's instruction execution hardware essentially does a context switch every instruction cycle, finding the next thread that is ready to issue an instruction into the execution pipeline. Any thread that is not currently ready to issue, typically because it's waiting for data to arrive from memory into its registers, is skipped. So when some threads are stalled waiting for the data they need to arrive from memory, other threads probably have something to do, and so somebody can issue an instruction nearly every instruction cycle, and that keeps the processor busier than it would be, otherwise. That's not the important thing, though. The important thing is that the processor can stay busier, issuing memory requests.

Basically, the architect of the Threadstorm processor, Dr. Burton J. Smith, now of Microsoft Research, was making a fundamental assumption about the nature of the computations this processor was going to run – and this assumption was the opposite of the one usually made by processor designers. The Threadstorm design uses the real estate of the processor chip for all these extra sets of registers, enough to hold the states of 128 threads. In a conventional Intel or AMD processor, that same real estate is typically taken up by a hierarchy of caches. When an architect puts caches in a processor, he/she is assuming that there will be locality in most of the computations that processor will run. It can be spatial locality ("the next item of data the application will need is very close in memory to the one it is using now"), or temporal locality ("the application is going to re-use the current item of data several times in rapid succession before it is through with it"). When the application you're running has lots of locality, caches drastically increase performance, because the next item of data the processor needs to operate on is likely to already be in the cache. A heavily multithreaded processor is essentially designed on the assumption that the applications it runs will have little or no locality. Suppose that an application is structured in such a way that practically every data reference is independent of the previous one, and the cache in a conventional processor will be sitting unused most of the time. When a conventional processor runs out of data items to work on, it will stall until more data items arrive from memory. While it is stalled, it won't be issuing more memory requests. The Threadstorm processor, on the other hand, tends not to stall, because it has so many other threads that may have something to do, particulary issue more memory requests. A typical commodity processor can issue 4 to 8 memory requests until it has to stall, waiting for some of them to be fulfilled. One Threadstorm processor can have more than 1000 memory requests in flight at once. Thus, if your application has no locality, and the processor is going to have to issue a great many unrelated memory requests, a multithreaded processor can issue many more per second than a conventional processor, as the application executes.

What kinds of applications tend to have lots of locality? Well, most of them, but in the supercomputer context that I'm most familiar with, we can say that a great many scientific applications have quite a bit of locality, because it's often dictated by the physics: molecules directly influence other molecules that they bump into, and so on. What kinds of applications don't have any locality? Well, those where you're de-referencing pointers a lot, and there isn't any predictability as to where in memory, or even in which node's memory bank, the next referenced item is going to reside. So, it's big, pointer-riddled data structures that a multithreaded multiprocessor is ideally suited for. I can think of an example: graph-oriented databases. Graphs that are created from real-world data are notorious for the irregularity of their structure, and the inability of their structure to be partitioned – divided up in such a way that most of the references at least stay in the same node. So when you do graph applications on a parallel computer, your code is going to be making seemingly random references to data items, all over the machine. A system that keeps issuing those references instead of stalling, has a huge advantage. We typically see a factor of 10 to 100 better performance of a graph application on the uRiKA system than the same application, on the same data set, implemented on a conventional parallel system with the same number of processors.

On any parallel system, for any parallel application, when you've tuned the app as well as you can, something is going to be the bottleneck. It might be the processor throughput, or it might be the memory bandwidth. On principle, if you can keep that bottleneck saturated, you're doing as well as you can possibly do. On graph applications, because of the seeming randomness of their references, the bottleneck is usually the interprocessor communication network. Because uRiKA's Threadstorm processor can keep so many remote memory references in flight throughout its communication network, it runs faster and scales better on large graph problems than any other system out there – and the scaling is along either axis, increasing the problem size, or increasing the number of processors used. We bad.

This entry was posted in **Graph Analytics**. Bookmark the **permalink [https://web.archive.org/web/20150214173827/http://www.cray.com/yarcdata/blog/?p=182]** .