Converting an X86 HTA BFS to Scratchpad

Austin Gibbons – September 6th, 2011

An unfinished draft

The BFS implementation works as follows:

Build Graph

While(not done)

Pop work from local queue

Check for completion

Expand neighbors

Check if neighbor is found

If undiscovered, add to new neighbors

Add new neighbors to communication structure

Synchronize (barrier / lock / flag)

Take from communication structure

Add to local queue

Theory

This implementation lends itself to natural Scratchpadization. The following data structures are good candidates for scratchpadery:

1. Communication. This is a series of *p x p* of one-producer-one-consumer queues, where p is the number of threads. Storing this queue in the consumers is a natural use of a scratchpad.  
   **Problems with implementation:** The problem is the implementation of the queue. If it is fixed length then we can just allocated it once and db\_mem / db\_release from one to another. But, in order to scale to any graph size, we need to allow our queue to grow, which forces to confront allocation/free as well as the potential of running out of space.

**Solution:** Multicast. Being able to allocate the same structure in more than one location with either (1.1) duplication or (1.2) smart partitioning with a TLB of sorts would greatly improve the ease of programmability.

1. Local Queue Bookkeeping. Another natural scratchpadual. Simply store the front of your queue in your available local memory and offload the rest to exterior memory, bringing it into your scratchpad as appropriate.

**Problems with implementation:** A lot of bookkeeping and memory movement. Also, my implementation uses ***WAY*** too much meta-data, which is really my fault. It is further difficult to handle dynamic data structures in C, and this difficulty is amplified by AFL’s space requirements & limitations.  
**Solution:** Building out of Pradeep’s version is an overall smarter approach. I feel that toggling the allocated hardware is an acceptable solution as well. It still is wasteful to use this memory though, as it will waste power and time. Greater empirical analysis would be needed to compare performance versus programmability versus other implementations.

1. Graph Nodes. Because I am using a static mapping to determine “ownership” of a node and I am pre-loading the graph into one giant memory structure, I should allocate the node into the thread which will eventually own it.  
   **Problems with implementation:** Any graph which is larger than 64k \* numEngines will not fit into this structure. Plus, it feels like cheating and is not that flexible.  
   **Solution:** Scaling your hardware with your problem size. Similar to (1.1), made easier with multicasting.
2. Neighbors. Neighbors can be spawned locally before being shared through the communication structure. Because we are controlling the amount of work between Communication, this should directly fit into the Local memory  
   **Problems with implementation:** The problem remains having dynamic data structures as in (2), as well as the shortcuts taken in my implementation. My implementation allocates a new node *for every edge*. This is because of the speculation support. Namely, in order to ensure that the correct height is the final height a node is left with, an instance of the node is generated with the associated height by a *spawner thread*. Then the spawners hand off the work to the *owners*. The owners then do the height comparison to ensure only the lowest height is preserved. Correctness is ensured because given a node id, the owner thread is the same, so comparisons are not racy. This can work well with a scratchpad in a similar vein as (1). Indeed, the elements of the Communication queue are these nodes. Just as it would be ideal to have a fixed length queue, it would be better to have the instances of these nodes fixed in local memory and changing their values rather than allocating new memory.  
   **Solution:** Creating a dynamic queue template solves this problem. Further, I have a better implementation which does not so grossly over-allocate. This over-allocation context is still a fair analysis, because it is not such large an issue when using malloc (though still would be preferable not to perform such allocations). Indeed, that the Scratchpad version forced me to realize this inefficiency is a proponent of its usefulness in HPC.
3. Checking if a node has been previously explored can be optimized in the following way. If you have some smart partitioning of your graph, then you can rely on spatial locality and check in a hierarchical fashion Namely, if you know the (preferably static) partitioning infrastructure, you can avoid re-discovering nodes by creating one bit for a node at each level of the hierarchy. Then, you can make comparisons at each level of the hierarchy, stopping when either a bit is set (previously discovered) or you exhaust the structure (first discovery).

**Problems with implementation:** This approach will generate a good deal of network traffic. It will also cause data duplication, which wastes power and space. Implementation is also more taxing to program than a naïve approach, as placing it into each level of the hierarchy currently requires a manual effort.

**Solution:** Empirical analysis to determine if it was useful. If this feature was often useful (and indeed it would be, I readily argue) than a template version would greatly help programmability.   
*via Justin Teller*

Analysis

From a programmability standpoint, the existing interface into the scratchpad model is functioning but missing a few key features. The most important among this is the “multicast” idea. I believe automatically determining the amount of hardware needed is possible as well, but may require some hints from the programmer or may prohibit the programmer from doing anything too un-patterned.  
A *Cache-aware* implementation (optimizations specifically targeting a memory hierarchy with a fast, hardware coherent smaller cache hierarchy and a slower, fully-associative memory) has the disadvantage of succumbing to *adversary* graphs. At a high level, a cache-optimized version will perform better given some spatial locality. One can construe a graph which has such poor spatial locality as to make these optimizations a negative effect. Consider a cache with a cache line size of 256 bits. Our visited array (alluded to in (4)) which allows us to check if a node has been previously explored using only one bit can go wasted if no vertices within 256 bits of each other share an edge. The effect will be bringing the cache line into the cache, changing the one bit, and then not using the line again during this iteration. The performance is then relying that the data persists in the cache through iterations and some spatial locality is still observed.

General complaints for the Scratchpad model

This model requires you to know how much memory you are using. For the most interesting problems in the world, this is a good thing, as understanding the problem will lead to greater performance. For a general purpose Breadth First Search, this is unfortunate. In the memory-blind version I felt that over-allocating\* was not good but I was not paying any real penalty. This issue is related more to the effort required to build dynamic data structures in C than a true issue with scratchpads, but it does create a tax on the programmer that they must be willing to accept.

\*I use a series of three hash tables per thread to maintain the execution order. This is an *extremely* superfluous amount and stems from the fact that my version supports speculation and does not exhaust an entire level before continuing execution. There is no hash table support in C, so to avoid implementing a hash table which could grow and shrink properly; I simply over-allocate a large enough hash table at each instance. I pay no drastic penalty in the traditional model, but in a scratchpad model this needlessly consumes space in the local memories, and causes their quick exhaustion.

Comparison

The cache model is arguably less scalable than the Sunshine+ hierarchy. Let us observe how the cache model and the scratchpad model scale with (1) number of vertices and (2) number of edges

1. As the number of vertices grows, the metadata which encompassed the entire set of vertices in the cache will no longer be able to fit, and collision misses will impact the performance. In general, a large enough adversary graph will see poor cache performance and generate significant network traffic.   
   The scratchpad model will not have this cache exhaustion issue. It will instead be bounded by the total number of execution units available. At some critical point, the graph + associated meta data will no longer fit into the local memories, and eventually will require storage into DRAM. This is as bad as the cache model, but does not suffer the same loss of useless cache fetches. The scaling of the network traffic will tentatively be worse than in the cache model, as the probability of communicating with a node belonging to a block on another chip will increase as the number of vertices increases.
2. As the number of edges grows, the cache version can expect to have a greater hit rate in its visited array. It will of course ultimately have concerns of hitting too well and experiencing thrashing.   
   The Scratchpad model will speculatively not see this performance increase, and will have space concerns much more quickly.

Conclusion

**Programmability**: I am of the professional opinion that it is not overly-taxing to program for the Scratchpad model within the framework of a memory management library. Given a system to easily associate a pointer with a location or series of locations in the hierarchy, then the ability to assess where data should be allocated is not over-burdening.   
**Performance:** It is not merited through significant empirical demonstration, but the theoretical model I hold misses only on optimizations which utilize the hardware (using the coherency scheme as communication, allocating data to avoid collisions, etc.) have not seemed important\* and do not reflect any algorithmic improvement.

\*A quick test using the visited/checking scheme did not demonstrate significant speed-up.

**Optimization:** An implementation which caters to a cache version pays a mindset to the size, structure; et al of the cache is in essence performing the scratchpad optimization through a more challenging lens. Optimizations which rely on the aforementioned hardware structure could further be created in a scratchpad model, but do not seem necessary.   
The scratchpad model offers an explicit version of the analysis needed to fit data into cache. The memory management task is pushed into the software level but does not necessarily reside on the top of the software stack.