Pen & Paper Memory Access Analysis for BFS

Austin Gibbons – September 7th, 2011

The BFS Implementation:

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

Data Structures:

* 1. Global structure to hold the graph (as its parsed from a file)
  2. Communication structure – a numThreads x numThreads array of linked lists used for inter-thread communication
  3. A queue of nodes to explore
  4. Three thread local hash structures.
     1. Hash by node identity (into the working queue)
     2. Hash by height to the first element with height *h*
     3. Hash by height to the last element with height *h*

This version was written to support speculative execution (to make load-balancing & spatio-temporal locality optimizations), and thus the hash tables are part of this infrastructure.

Scratchpad Behavior:

This analysis assumes the following structure: We are given a graph which can fit into the local memory of our cluster. Namely, for a graph with *numPoints* such that each point in total requires *point\_size* memory given that we have *engineHeap* kilobytes of engine heap, *nXEs* per block, *nBlocks* per chip, we have

*nChips* >= (numPoints \* point\_size) / (engineHeap \* nXEs \* nBlocks)

I am further assuming that the memory loaded into the block\_memory is not the limiting factor. I assume this because in my implementation it is not.

My implementation is, admittedly, not focused on minimizing the memory footprint. Indeed, the cost of a point on the local memory is:

Θ(Sizeof(node)\*3 + sizeof(edgeList)\*average\_number\_of\_edges) bytes  
(The \*3 being the hash tables described in the data structures section)

With (in bytes)

sizeof(node) = 32

sizeof(new\_node) = 24

sizeof(edgeList) = 16

Thus, we can approximate it as being 96 + 16\*edge\_average.  
Suppose the average edges is .1% of the graph. Then, solving for the above equation using our known values of (in bytes)

engineHeap ~= 64k

nXEs = 8

nBlocks = 16

We can see that allocating our edgelists directly into local memory already requires Θ(64) chips for a graph as small as 2^14 vertices and 2^25 edges. And this number grows by a factor of four as we double the vertices.  
For comparison, the bottom of the June Graph 500 rankings uses 32 cores and runs on 2^26 edges.  
I claim the expected performance and performance-per-watt from an implementation rooted in local memory in the scratchpad hierarchy warrants investigation into this model. Further, there are graphs bounded by these sizes which are interesting, and readily available across literature.

**Analysis:**During the execution, the thread will pop work from its local queue which resides in its local memory. It will then traverse the edgeLists, which exist in its local memory. It then loads the neighbors it discovers into the block local memory of the respective *owner*s. The owners then perform some computation before adding it into the local queues in their own local memories, and repeating the process.   
Thus, no data is shared with DRAM. Data which is transmitting across nodes goes to the block local memory, to simplify network traffic. Data which is being used is kept local to the XE performing the computation.

We could greatly reduce network traffic through contextual-based work stealing algorithm. By keeping the nodes local to the block through a dynamic partitioning scheme, we will greatly reduce power used through network communication.

Caching Behavior:

In a cache system, the graph is loaded into memory. Threads pop work from their local queue. We will assume, generously, that **all of these are cache hits**, leftover from the previous iterations push. We are assuming this because we are looking at graphs which fit into our local scratchpads, and presumably our system has comparable cache space. These nodes’ edge lists are then loaded, which will involve **mostly cache hits, and** **some cache misses**, as I have implemented this as a linked list, but the links were allocated in order, and we are assuming malloc() is not doing anything boisterous. Then if we have *lineSize* byte cacheLines and our sizeof(edgeList) = 16 as before, we will see (16 / lineSize)% misses. These will then be placed into the communication structure, which a smart optimization could have cache’d at all times, and thus we should attribute it **no cache misses**. We then perform some computation and find the corresponding node from our global graph structure, and add it to our queue. This will have **un-patterned cache misses.**Thus in total, the raw version should expect to see a good deal of cache misses.  
We should also examine a cache-friendly implementation. For example, a simple optimization is to check if a node has already been visited before we re-explore it. This is easy to do with hardware cache-coherence, as we can use the coherency structure as a communication mechanism. We can now represent each node with a single bit, and set it to one to indicate that it has been checked. We can even let this happen racily and observe only a small increase in duplicate explored nodes. Then, our (16 / lineSize)% is decreased to **un-patterned cache misses**, as the cache misses are now dictated by whether the collection of edges we explore have neighbors with good or bad spatial locality. We should still see at least (number\_of\_vertices / bits\_in\_a\_line) total misses, but expect much more than that for an average graph.