

Running Brook Scalar on SSS

I. Buck

June 6, 2002

This document proposes a method of how to run Brook scalar code on the SSS architecture.

- Having one node run the scalar code and trying to coordinate all of the other 16K nodes is clearly too much work for a single node. The alternative discussed in this document runs a copy of the scalar code on each node in the system. Each node basically mirrors the scalar computation.
- All static memory (global variables, local variables, constants) is replicated at each node. Any read or write is resolved by local memory. No synchronization is required.
- Heap memory is divided in two groups, local and global. Local memory, allocated by regular malloc, allocates memory within the local node. This operation is replicated on all the nodes just like static memory. No synchronizations are required for local heap memory since all nodes perform the same writes and reads to their local memory.
- Global heap memory, allocated with a special alloc function “globalalloc” (or something), is NOT replicated. When the code performs a read to global alloc’ed memory, each node fetches the value from a single memory location. This is a costly operation since all the nodes are doing a read to a single mem location. However global heap memory can be allocated across the entire machine which can be much larger than regular malloc’ed memory.
- Writing to global memory only needs to be performed by a single node since all the nodes would write the same value. We can label one node as the “master” node which is responsible for performing the writes to global memory. The other option is that nodes only write to global memory locations if they fall into its local memory space. This minimizes the network traffic for writes.
- Synchronization is also required for reads/writes to global memory. If a node performs a read to global memory but the node responsible for that location has not yet written the value, the result of the read is not valid. Furthermore, if the writing process is ahead of the read process, the read could return a result which is the value of a future write. One way to deal with this is to insert barriers before and after every write operation to global memory. A costly solution but does provide correctness.

- Caching should be fine on global memory since the nodes all know what the values of global locations should be.
- The cost of synchronization is pretty huge since we'll need to synchronize at every global write op. One alternative is to allow OpenMP style directives to allow a single node to update a region of global memory under release consistency. A synchronization would occur at the start and end of the OMP directives. I recommend reading the OpenMP spec at <http://www.openmp.org/> and borrowing some of these pragma directives for dealing with updating global memory.
- We should allow threads in our environment since we have a large number of scalar processors which all could be doing different tasks rather than the same scalar code. Introduction of a basic threading api or OMP style multi-threading would be useful however we may want to limit whether threads can kick off kernel operations since we still want these to be global operations.
- With thousands of nodes, it may be beneficial to segment the system into separate environments allowing for each segment to perform its own streaming operations and separate threads. This allows the application designer to better load balance a large number of nodes which may have to operate on a large number of short streams.
- I still would also like to not give up on implementing an MPI layer for the SSS. Though it goes against the shared memory model, many existing applications are already implemented using MPI and would benefit greatly from the faster SSS network. This would provide a killer demo and could springboard into introducing streaming to make things even run faster.