Proposal

Mark Madler

1 Design

This work will consist of modifying an existing compiler (gcc or llvm**) to support LOCO as a backend for disaggregated memory. Ideally allowing programs written for OpenMP to work seamlessly in a disaggregated setting. These modifications include modifying dynamic memory allocation to use special virtual addresses for LOCO objects and modifying load/store behavior for these special addresses. There will also be a system to determine whether a remote address is cached or not. All cached data will be tagged with a dirty delta to allow for seamless release consistency. On releases data will be written back to its home-node and only modified data will be written back (scatter). If two nodes wish to write back the same byte-area they are in a data race and it is the programmer's fault.

The first iteration of this design should include all heap memory in a LOCO hashtable and simply perform read/write for each load/store and an insert/delete for each new/delete. Since all shared data for OpenMP is on the heap, there is a nice analog to insert/delete for each allocation or deallocation of shared data. Tentatively it seems that stack accesses can be ignored by leaving calls to alloca alone and ensuring that "regular" memory addresses behave as normal.

1.1 New and Malloc

Calls to malloc and new need to be handled correctly in the frontend. So in clang there should be a call to a special LOCO runtime function to allocation LOCO memory. e.g. locomalloc. (basically just an insert at a new special memory address)

1.2 Load / Store Behavior

Loads and stores to new LOCO memory locations need to be handled differently. We will create new functions remoteload and remotestore which will replace loads and stores in LLVM IR based on the virtual address of the operands. This can be done in a special pass after the LLVM IR has been created which will maintain the modularity that LLVM has. There may also be a need to modify atomic loads and stores differently than others, but lock-guarded accesses should behave identically.

the load store modification might look something like Figure 1

```
for (auto &I : instructions(F)) {
  if (auto *Load = dyn_cast<LoadInst>(&I)) {
    if (isRemoteAddress(Load->getPointerOperand())) {
      // Replace with remote_load
      IRBuilder<> Builder(Load);
      Value *RemoteLoadCall = Builder.CreateCall(
          RemoteLoadFunc, {Load->getPointerOperand()});
      Load->replaceAllUsesWith(RemoteLoadCall);
      Load->eraseFromParent();
      }
  }
}
```

Figure 1. IR modification code

1.3 Synchronization Primitives

For this backend to work seamlessly with LOCO there may be a need to replace synchronization primitives with LOCO analogs. I am unsure of how these primitives might work with things like task synchronization and thread numbering across nodes.

1.4 Memory Consistency

The memory consistency model in this design should be relatively simple. There is release consistency. I would assume that on each write release all local changes are pushed to their remote locations ensuring all previous writes are visible before and subsequent writes. If doing an acquire load it may be necessary to pull remote changes. I assume it will be unless we broadcast releases to all nodes somehow

1.5 Caching

To cache data while all loads and stores are handled through this new hashtable mechanism we will require a scheme to save all read-accesses temporarily. Since we are not modifying how the stack behaves this could be where local cache is, this also works because it is assumed that cached data is not shared between threads. I think the most logical thing to do is to store all hashtable accesses on the stack and to add them to some cache-table which will be accessed first on lookups. If there is a acquire load or a write release RDMA operations are required.

1.6 OpenMP Integration

LOCO supports seemingly all OpenMP synchronization primitives, below is a table of analogous operations:

omp-Barrier	LOCO-Barrier (either local or Global)
omp-Flush	write + LOCO fence
omp-critical	use LOCO locks
omp-atomic	use LOCO atomic writes
omp-ordered	use barriers?

2 outstanding questions

some outstanding questions to answer:

- how does the local node know what data is local? – hashtable lookup for local heap may be somewhat costly
- does a system like this exist for TCP/IP? – not to my knowledge
- specifics of design implementation. i.e does loco have all openMP primitives? It seems that LOCO has a corresponding synchronization primitive for each of openMP's, but it may be difficult to determine when to use each primitive.
- should GCC or LLVM be used? It looks like GCC has hooks for malloc/realloc/free which would allow for easy integration. But then I am not sure where the insertion point for modifying load/stores is. LLVM on the other hand is very modular and it would be easy to add a shim layer to modify IR.

3 My TODO list / other notes on project progression

- get cmatose.c working on r320s to diagnose issue. (is it librdmacm?). cmatose works. The issue must be in our configuration – but why does hardware impact this?
- invalid args from loco on r320s, seems to not in in hostname lookup, but errors on `rdma_connect()` as called from `lf_peer` l.329 from `lf_network::init()` and manager. It appears that there is some issue with how the endpoint is setup or in the connection parameters. Maybe this issue is due to the fact that the NIC may not support IPv4? It really seems like this is because of hostname issues. I think that looking up addresses from hostname is not working in this context. The IP address associated with the destination is not even found in the dns server?? It is some address in Korea..
- change all config scripts in Odyssey. Changes seem to be mostly made, but still there is issue where copy-run.sh stalls after building
- Finish this proposal.