General Thoughts on APIs for Fine-Grained Multi-Threading

Erik Schnetter

Spectre telecon, 2014-06-05

Evolution of Programming Models for Distributed-Memory Parallelism

1. Message Passing ("Sequential Communicating Processes"): the nodes run independent processes that explicitly exchange messages
2. GAS (Global Address Space, aka shared memory): all pointers can point to all nodes' memory; locality is only an optimization
3. PGAS (Partitioned Global Address Space): explicit distinction between local and global (possibly remote) data; locality is exposed to programmer
4. AGAS (HPX terminology, Active Global Address Space): objects can move between localities without changing their address

Why Not Shared Memory?

- Idea: Memory already has a hierarchy (cache levels, main memory) – treat remote memory as just another level!
- Shared memory programming (OpenMP, C++11 threads) is easy
- Address spaces (64 bit) are large enough for whole system
- Hardware (Infiniband etc.) supports efficient RDMA (Remote Direct Memory Access)
- Caching can be implemented in software (still faster than RDMA latency)

Why Not Shared Memory?

- Problem: Cache coherence!
- On a modern system, all caches of all CPUs and main memory are always in sync (cc-NUMA)
  - For each memory location, there exists a global ordering for read/write accesses
  - Very important for ease of programming
- Cache coherence requires a lot of communication between caches and memories (“snooping”)
- 10…100 cores seem to be a practical limit

Minimum Ingredients for PGAS

1. Global pointers, i.e. pointers that can point to other localities
   - Should probably involve automatic memory management
2. Execute routines on other localities, e.g. call member function via global pointer
   - Should probably execute asynchronously since communication has high latency

Non-Ingredients

- Do not send messages to other threads; instead, start a new thread executing a function
- Do not expect other objects/threads to know what data one object/thread needs (push model); instead, each object/thread requests the data that it needs (pull model)
- No cache coherence:
  - Try to avoid locks and mutable state
  - Use atomic transactions and constant objects

On Global Pointers

- Implementation could be as simple as tuple (MPI rank, local pointer value)
  - HPX uses a more complex model involving a database lookup
- Need conversion from/to local (regular) pointers
  - Probably want easy interaction with shared_ptr etc.
- Will need futures of global pointers (“clients”)
  - Probably want to communicate these futures

On Remote Execution

- Could be as simple as remote async (see HPX)
- Will require serializing function arguments/return values
- Communication requires copying -- references, move semantics not well defined
  - Pointers and copying is easier
- Detached execution?
- Will want some parallel algorithms (broadcast, generate, map, reduce – MapReduce?)
- Info output? Debug output?

Summary

- The features and properties described earlier should be generic, valid for any extension of the C++11 threading model
  - If we cannot choose a communication library, then we should base our algorithms on those generic features
  - Can we design a clean and efficient infrastructure based on these?
- Alternative would probably be some flavour of message passing