MPI and OpenMP

Pavan Balaji

Computer Scientist and Group Lead

Argonne National Laboratory
MPI+X for effective programming?

- MPI is only for inter-address-space (node) communication
  - Within an address space, we need OpenMP, OpenACC, CUDA, etc.
- Hybrid programming vs. a single unified programming model
  - The number of models we program to should not be too large, but a small collection of standardized programming models which interoperate with each other is not a bad thing
  - MPI+OpenMP has demonstrated this successfully

*Why is this:* better than this?
MPI+Threads Programming

- Interoperability requires the right semantics
  - Neither MPI nor OpenMP meet these requirements just yet
  - But they can be fixed

- As of MPI-3 and OpenMP-4, interaction semantics were limited
  - MPI semantics required sharing of MPI object information across all threads in the address space
    - Sharing data structures across the entire node is expensive
    - Memory consistency is expensive
  - OpenMP semantics do not expose concurrency
    - Execution in two “threads” does not mean concurrent execution
State of Hybrid MPI+OpenMP Programming

- MPI+OpenMP interoperability can happen in multiple ways
  - Funneled and Serialized modes are most common where a single thread makes MPI calls at a time
  - THREAD_MULTIPLE is becoming increasingly common where multiple threads can make MPI calls simultaneously ("fully multi-threaded")
    - Functional implementation provided by almost all implementations

- What’s missing?
  - Performance of fully multi-threaded communication is not entirely optimized
    - Some research has already made its way into production supercomputers
  - MPI specification needs new capabilities
  - Threading and tasking models need new capabilities
MPI Implementation Optimizations
Contestion in a Multithreaded MPI Model

Multithreaded MPI
- Threads can make MPI calls concurrently
- Thread-safety is necessary

Thread-safety can be ensured by:
- **Critical Sections** (Locks)
  → Possible Contention!
- Using **Lock-Free** algorithms
  → Non trivial!
- **Still does memory barriers**

```c
MPI_Init_thread(...,MPI_THREAD_MULTIPLE,...);

#pragma omp parallel
{
    /* Do Work */
    MPI_Put();
    /* Do Work */
}
```

**Enter_CS()**
**Exit_CS()**

**Thread1**
**MPI_Put()**
**ENTER_CS()**
**EXIT_CS()**

**Thread2**
**MPI_Put()**
**ENTER_CS()**
**EXIT_CS()**

Thread Sleeping/Polling

Pavan Balaji, Argonne National Laboratory

DOE Runtime Workshop (03/12/2015)
Coarse Grained Locking

- Single global mutex
- Mutex is held between entry and exit of most MPI_functions
- No concurrency in communication
Lock Granularity

- Using mutexes can affect concurrency between threads
  - Coarser granularity → less concurrency
  - Finer granularity → more concurrency

- Fine grained locking
  - Decreasing size of critical sections increases concurrency
    - Only hold mutex when you need it
  - Using multiple mutexes can also increase concurrency
    - Use separate mutexes for different critical sections
  - Acquiring and releasing mutexes has overhead
    - Increasing granularity increases overhead
  - Shrinking CS, using multiple mutexes increases complexity
    - Checking for races is more difficult
    - Need to worry about deadlocks
Levels of Granularity

- **Global**
  - Use a single global mutex, held from function enter to exit
  - Existing implementation

- **Brief Global**
  - Use a single global mutex, but reduce the size of the critical section as much as possible

- **Per-Object**
  - Use one mutex per data object: lock *data* not *code sections*

- **Lock-Free**
  - Use no mutexes
  - Use lock-free algorithms
  - Future work
Fine-grained Multi-threaded Communication

- Non-blocking sends access global data
  - Requests are allocated from a global request pool
  - Alloc and free of req requires updating reference count on communicator
- Per-object tlp: thread-local request pool
- Per-object tlp atom: use tlp and atomic ref count updates
- Still some contention exists
Dimensions of Thread-Safety

• Critical Section
  Granularity/Length
  • Shorter is better but more complex

  Critical Section Arbitration
  – Many blocked threads
  – Who goes first?
    • Random
    • FIFO
    • Priority
Unfairness May Occur!

Flat memory

Hierarchical Memory

Access should be random

Access biased by the proximity to the cache containing the mutex
SWAP-Assembler

- Blocking Send/Recv
- Two threads per process
  - One sending
  - The other receiving
- Strong scaling with 1 millions reads, each with 36 nucleotides
MPI Specification Changes (proposed)
Endpoints Proposal Status

- Endpoints is proposed for MPI 4.0
- Hybrid WG has completed formal proposal
- Formal reading scheduled for December ‘14 meeting
  - Then on to voting!

- Further reading:
  - [https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/380](https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/380)
**MPI Endpoints Semantics**

- Creates new MPI ranks from existing ranks in parent communicator
  - Each process in parent comm. requests a number of endpoints
  - Array of output handles, one per local rank (i.e. endpoint) in endpoints communicator
  - Endpoints have MPI process semantics (e.g. progress, matching, collectives, ...)

- Threads using endpoints behave like MPI processes
  - Provide per-thread communication state/resources
  - Allows implementation to provide process-like performance for threads

```c
MPI_Comm_create_endpoints(MPI_Comm parent_comm, int my_num_ep, MPI_Info info, MPI_Comm out_comm_handles[])
```
MPI in ULT and Tasking Environments (MPI-5?)

- THREAD_MULTIPLE mode requires MPI to do two things
  - Locking (for thread safety)
  - Yielding (for deadlock avoidance)

- In ULT environments, if concurrency of threads/tasks (or lack of it) is exposed to the MPI library, locking is unnecessary
  - pthreads
    - Everything is concurrent
  - qthreads/MassiveThreads
    - Don’t know what is concurrent
  - Need a new threading/tasking library (or more information exposed from current threading/tasking libraries) to make this effective
New Threading/Tasking Libraries and Dynamic Runtime Systems
User-level Threads (ULTs) and Tasks

- **User-level Thread (ULT)**
  - Lightweight thread with low context-switch overhead
  - More than one ULT can be mapped to a single OS thread
    - May not be executed concurrently

- **Why ULTs?**
  - Conventional threads (e.g., pthreads) are too expensive to express massive parallelism
  - If we create pthreads more than # of cores (i.e., oversubscription):
    - context-switch and synchronization overhead will increase dramatically
  - ULTs can mitigate high overhead of pthreads but need explicit control

- **Where to use?**
  - To better overlap computation and communication/IO
    - Low context-switching overhead of ULTs can give more opportunities to hide communication/IO latency
  - To exploit fine-grained task parallelism
    - Lightweight ULTs are more suitable to express massive task parallelism
Motivation
Message Passing with User-level Threads

Overlap communication with computation using ULT

- Lightweight
  - ULT does not execute concurrently using additional hardware resource, but takes turn to run by context switching
  - No lock needed between two ULTs in the same kernel thread

- Asynchronous communication
  - Helps turn a MPI blocking call to a nonblocking one
  - Decouples the operation of “send start” and “send complete”

- Dynamic Tasking
  - Providing automatic overlap based on task-graph dependencies
Qthreads

- Lightweight portable ULT library
- Locality-aware workstealing scheduler
- Supports multiple programming models: Chapel, OpenMP
- Pros and cons when integrated with MPI:
  - Pros: provides an easy interface to use ULT
  - Cons: lacks an explicit semantic for parallelism. All ULTs can potentially run in parallel thus lock is needed everywhere

Qthreads Architecture
Source: [Friar Tuck's Chapel]
Argobots

- **Execution Streams (ES)**
  - Sequential instruction stream
    - Can consist of one or more work units
  - Mapped efficiently to a hardware resource
  - Implicitly managed progress semantics
    - One blocked ES cannot block other ESs

- **Work Units**
  - User-level Threads (ULTs)
  - Tasklets

- **Scheduler**

- **Synchronization primitives**
  - Mutex
  - Condition variable
  - Future
  - Barrier
MPI+Argobots: Data Movement in Distributed Memory Systems with Lightweight Threads

- Hybrid runtime of MPI and Argobots
  - lightweight and dynamically adapt to the hardware resources
- Two level of threads provide an explicit semantic for concurrency
  - Execution Stream (ES) provides concurrent execution
  - User Level Thread (ULT) provides fast context switch
- vs. MPI+Qthreads
  - Qthreads share its ULT among workers, so it can not specify which ULTs run in parallel. Argobots binds ULT to ES for explicit scheduling
  - Highly optimized context switch in Argobots
Application: HPCG

- High Performance Conjugate Gradient (HPCG)
  - Solves $Ax=b$, large and sparse matrix
  - Models 3D PDE. 27-point stencil, communicate with up to 26 neighbors

- Key Communication Patterns
  - Global collective communication
    - MPI_Allreduce of local dot product
  - Neighborhood communication
    - halo exchange with neighbors

for $k = [1: \text{max\_iter}]$ && $\text{normr} > \text{tolerance}$:

MG(A, r, z);
if $k == 1$:
  WAXPBY(1.0, z, 0.0, z, p);
  DDOT(r, z, rtz);
else
  DDOT(r, z, rtz);
  WAXPBY(1.0, z, beta, p, p);
SpMV(A, p, Ap);
DDOT(p, Ap, pAp);
WAXPBY(1.0, x, alpha, p, x);
WAXPBY(1.0, r, -alpha, Ap, r);
DDOT(r, r, normr);
normr = sqrt(normr);

Simplified main loop in HPCG
Overlapping Communication and Computation using ULT

- **Hiding Global Collective Communication**
  - **ult_ddot**: a wrapper of DDOT
  - overlap communication and computation between iterations: ult_ddot (in iteration i) and MG (in iteration i+1)
  - fork a ULT to do ult_ddot and join in the next iteration

- **Hiding Neighborhood Communication**
  - **ult_spmv**: spmv for one neighbor
  - for each neighbor, fork a ULT to do halo exchange and a small part of SpMV (communication)
  - main ULT computes local spmv (computation)

```c
for k = [1: max_iter]:
    MG(A, r, z);
    if k > 1:
        ult_join (thread);
        if (normr <= tolerance) break;
    ....
    ult_fork(ult_ddot, &param, &thread)
```

```
SpMV(A, x, &y):
    for each neighbor:
        ult_fork(es, ult_spmv, &t[i]);
    for i in [0: nRows]:
        ult_yield();
        for each j in row i:
            y[i] += val[j] * x[idx[j]];
    for each neighbor i:
        ult_join(t[i]);
```
On 2,048 cores, HPCG using MPI+Qthreads shows performance improvement of 19.8% over MPI-only version, or 34.9% over MPI+Pthreads version.

- As core number increases, the benefit of communication hiding begins to reveal. DDOT% increases from 0.62% on 16 cores to 36.8% on 2,048 cores.
Preliminary Results: SpMV w/ MPI+Argobots

- On 1,024 cores (grid size=64^3 per process), MPI+Argobots shows an improvement of 51.3% while MPI+Qthreads shows 27.7% compared against MPI-only version.
Take Away

- MPI+X is becoming an increasingly popular model
  - The big problem is neither “MPI” nor “X”, but rather the “+”

- MPI+X is an evolving model
  - Far from perfect, but changes are underway for both the implementations of “MPI” and “X”, as well as the standards to make “MPI+X” bigger than the sum of the parts

- The work is not done, still a long way to go
  - New “complementary” programming models are needed in some cases, but they must not ignore the pieces that MPI has gotten right
  - Orthogonal technologies that work with MPI (MPI+X) are important