Generalized Communication Pattern Offload with OFI

Sayantan Sur, Intel

(in collaboration with Jithin Jose and Charles Archer)

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Motivation

MPI defines twelve types of collective operations, along with their corresponding vector, datatype, blocking/non-blocking and some neighborhood variants

There are a multitude of algorithms that can be used depending on topology

Increasing desire to reduce processor overhead and offload the collectives to the fabric

EXPLOSION in underlying APIs trying to expose every use case!

The Open Fabrics Interfaces (libfabric) way

The OFIWG is creating extensible, open source framework that is aligned with application needs for high-performance

There is no dedicated collectives offload interface defined yet

If we did the desired solution should be:

- Derived from application requirements
- Make use of the existing framework
- Avoid specialized interfaces if possible
- Should be generic in nature as far as possible
- *Should be high-performance!*

Summary of Collective Implementation Techniques

Using MPI Send and Receive

- Limited Offload
- Can compose all algorithms
- Algorithm innovation resides in MPI library

Using high-level Collective Library

- **No visibility of algorithm in MPI**
- **Does not compose with other operations**
- **Problem simply pushed down**
- Easy for developers

Using Fabric Specific Features

- Good performance on one generation of hardware
- Disruption/panic on next generation
- Hard for developers

😳

Generalized Pattern Offload

- Can use to compose many algorithms
- True offload when vendor supports it, otherwise similar to MPI send/receive
- Algorithm innovation remains in MPI

💯

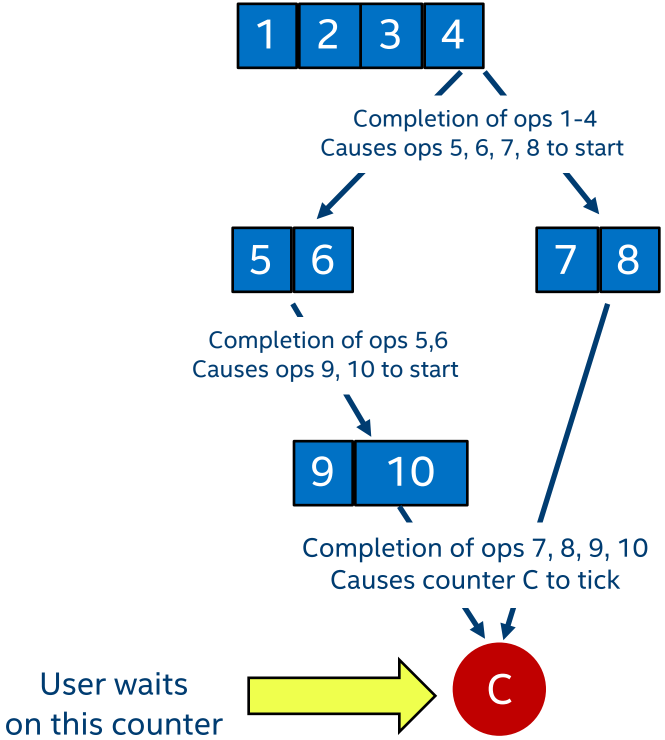
Tell me more about these Generalized Patterns!

Communication patterns such as: A sends to B, B sends to C, C sends to A

User can describe an operation and then schedule it for future execution while specifying dependencies

Fabric can make progress on the schedule

User can just wait for the completion of the entire schedule (not on a per operation basis)



Completion of ops 1-4
Causes ops 5, 6, 7, 8 to start

Completion of ops 5,6
Causes ops 9, 10 to start

Completion of ops 7, 8, 9, 10
Causes counter C to tick

User waits
on this counter

What is new?

Collective communication primitives proposed by DK Panda's group

- "Design and Evaluation of Communication Primitives with Overlap Using ConnectX-2 Offload Engine"
  - Raw CORE-Direct APIs were not sufficiently composable
  - The work proposed operations such as: one-to-many multi-send, many-to-one receive, receive replicate, receive-reduce, etc.
  - *The OFI proposal allows composition of 1:N, and N:1 operations using 1:1 operations, and proposes send-reduce as the fundamental new op*

Current proposal is inspired by Torsten Hoefler's cDAG and GOAL

- *The OFI proposal adds persistence, and techniques to update operations in place (such as changing send/receive buffer pointers)*

Changes proposed to OFI for primitive offloading

1. Prepare a command for future execution
2. Arrange commands in required dependency
3. Call into OFI to create the schedule structures
4. Run the schedule
5. A way to “update” commands that are already in a schedule
6. A send atomic function
7. Flags to optimize schedule execution

Preparing a command for future execution

Define a new flag FI_SCHEDULE to prepare a command

A command is referenced by struct fi_context

```
fi_sendmsg(struct fid_ep *ep,
           const struct fi_msg *msg,
           flags | FI_SCHEDULE);
```

FI_SCHEDULE flag causes provider not to send out message, rather return some opaque data filled in the context structure

The returned context structure is used later to execute the message

Works similarly for recv, rma, all other types of transactions

Arranging the commands

Schedule is expressed as a tree

Child nodes are executed after parent nodes are done executing

```
struct fi_sched_ops {
        struct fi_context *ops;
        uint32_t num_ops;
        struct fi_sched_ops *edges;
        uint32_t num_edges;
        uint64_t flags;
        void *reserved[8];
}
```

Concurrent set of ops
(context structures obtained
using FI_SCHEDULE)

Sub-trees that are executed
When the ops are complete

Useful for the provider
when BFS parsing the tree

Updating the commands and Send Atomic

In collective operations, the arguments change between calls, but the schedule remains the same

- Simply call sendmsg/recvmsg with FI_SCHEDULE flag
- The command will be updated “in place”

Define a new sendmsg() call in fi_ops_atomic

```
fi_atomic_send(struct fid_ep *ep, const struct fi_msg_tagged *msg,
        enum fi_op op, uint64_t flags);
```

The semantics of this call are just like send, with the addition that after the match occurs, op is applied to the matching receive buffer

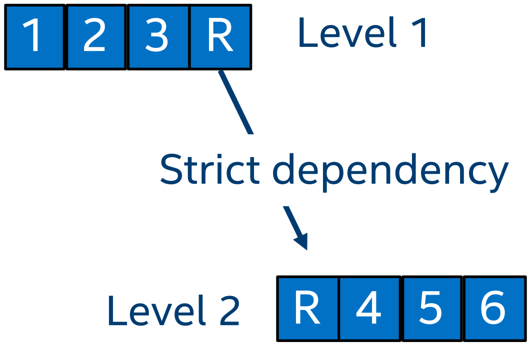
Optimizing the Schedule Execution

Schedules are offloaded, and one of the primary issues to deal with is managing early arrivals – sender arrives before receive is posted

- There may be situations where the same receive buffer (R) appears in multiple levels of the tree, the second receive cannot be posted before the first is complete

Useful if app can declare that there are no such dependencies and schedule has been globally pre-posted

- FI_SCHEDULE_RECV_DISJOINT: all receive buffers are disjoint
- FI_SCHEDULE_REMOTE_READY: the remote side is already executing this schedule



Together, they help avoid early arrivals

Example of how to use Eager mode in blocking collectives

MPI Communicator Creation:

1. Allocate three* sets buffers for small message allreduce
2. Create three fid_sched with flags DISJOINT | REMOTE_READY
3. The schedules start with a receive operation, so until the first receive is matched, it doesn't advance
4. Run two schedules – they will wait until the first matches

MPI Allreduce invocation:

1. Issue the first matching send operation
2. Post a schedule for the next invocation
3. Wait for current schedule to end

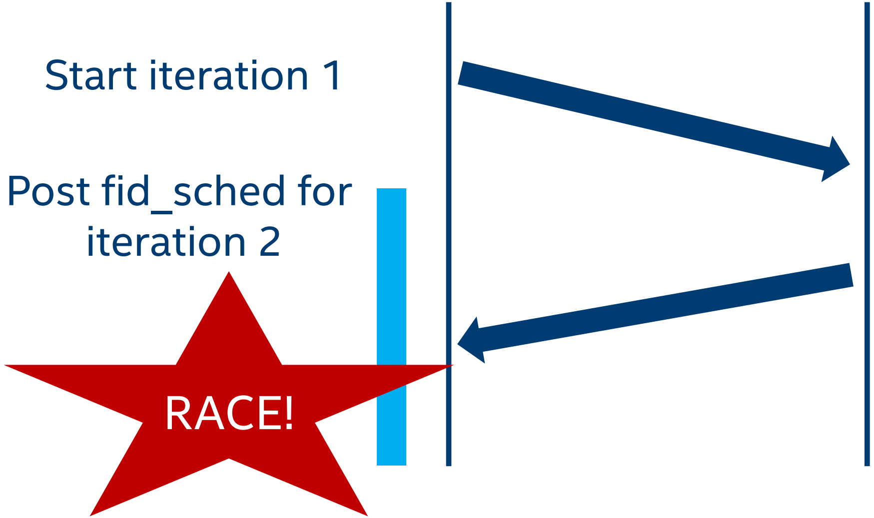
Posting of schedule overlapped with execution of Allreduce!

* = wait for next slide

Why do we need three sets of buffers?

Want to overlap the posting of schedule with execution of Allreduce

If we had only two sets of buffers and posted the schedule AFTER starting



Start iteration 1

Post fid_sched for iteration 2

Waiting for end of iteration 1

Start iteration 2

RACE!

Example of how to use for non-blocking collectives

MPI Communicator Creation:

1. Create N fid_sched with flags zeroed
2. Use NULL as buffer pointers

Pre-allocated fid_sched allow for certain number of concurrent non-blocking collectives

MPI Allreduce invocation:

1. Choose i^{th} among N fid_sched
2. Update the buffer pointer in the schedules with the user buffer provided in invocation
3. Post the schedule

Conclusions and future work

A prototype implementation available on top of libfabric sockets provider:

- https://github.com/sayantansur/libfabric/tree/schedule

Work is progressing on expressing MPI collective algorithms into DAGs

Proposal will be brought up for discussion within OFIWG

- Participation is completely open and interested folks are welcome to contribute to the discussion

Performance evaluation with capable libfabric providers

Legal Notices and Disclaimers