Scheduling Data Streams for Low Latency and High Throughput on a Cray XC40 Using Libfabric

Farouk Salem, Florian Schintke, Thorsten Schütt, Alexander Reinefeld
{salem,schintke, schuett, ar}@zib.de
Zuse Institute Berlin
Berlin, Germany

Cray User Group 2019, 5-9 May, Montreal Canada
Introduction: Research Project

- **The Compressed Baryonic Matter (CBM) Experiment**
  - observes high-energy nucleus-nucleus collisions
  - at FAIR in Darmstadt, Germany
    - appr. 150,000 m² and up to 24 m depth
    - 20 buildings

- **Many sensors surround the experiment**
  - hundreds of data streams
  - > 1 TiB/s of aggregated generated data
Research Project

• Data is received in high performance computing cluster called **First-Level Event Selector (FLES)**
  - builds time-slices for analysis using a software called **FLESnet**

• FLESnet logically consists of:
  - Input nodes: divide data streams into micro time-slices (MTSs)
  - Compute nodes: group MTSs to form complete time-slices
Status Quo: FLESnet Communication Pattern

- The communication follows best-effort approach
  - using round robin schema
  - complete time-slices are given to analysis

- Input nodes continuously transmit MTSs without coordination
  - Local buffer space
  - Ticket-based flow control mechanism is used

- Input nodes maintain a local buffer to receive data from sensors

- The communication pattern is challenging
Research Challenges

• Compute nodes process only complete time-slices
  o High dependency between nodes
  o Time to complete a time-slice increases with scalability

• Input nodes use part of network links at a time
  o Inefficient network/compute node utilization
  o Network/endpoint congestion

• Compute nodes have a limited local memory when scaling up
  o Less tickets / input node
  o Implicit higher synchronization
  o Higher influence of stragglers
Research Goals

• Achieve good aggregate bandwidth in large systems by
  o saturating all links at all times
  o avoiding endpoint congestion
  o requiring buffer space sparingly

• Gather all micro-timeslices for a time-slice in a short duration by
  o reuse memory buffers quickly
  o keep input nodes synchronized
Our approach: Data-Flow Scheduler (DFS)

• Offset-based round-robin schema
  o avoid endpoint bandwidth sharing
  o use all network links

• Coordinate input nodes in time
  o divide time into intervals
  o via collecting timing data from compute nodes
  o schedule injection rate at input nodes
  o let stragglers follow and catch up
  o consider different clock drifts

• Adopt to network changes (available overall bandwidth)
  o avoid network congestion
Data-Flow Scheduler: Architecture

- **DFS consists of two engines**
  - **Distributed Deterministic Engine**
    - proposes start-time and duration (meta-data) for each interval
  - **Input Engine**
    - follows DD Engines
    - broadcasts the actual meta-data to DD engines

\[
j : \text{interval index} \\
t_m : \text{time measured} \\
dur_m : \text{duration measured} \\
t_p : \text{time proposed} \\
dur_p : \text{duration measured}
\]
• **Distributed Deterministic (DD) Engine** consists of three modules:
  - **History Manager:**
    - collects the actual meta-data
    - calculates statistics
  - **Clock Sync.:**
    - triggers the clock drift
    - calculates the clock offsets between nodes
  - **Proposer:**
    - synchronizes the Input Engines
    - calculates interval meta-data of upcoming intervals
    - applies a Staged-SpeedUp (SSU) mechanism
    - broadcasts interval meta-data to Input Engines

---

**Data-Flow Scheduler: DD Engine**

![Diagram of Data-Flow Scheduler: DD Engine]

\[
\begin{align*}
    j & : \text{interval index} \\
    tm_j & : \text{time measured} \\
    tp_j & : \text{time proposed} \\
    durm_j & : \text{duration measured} \\
    durp_j & : \text{duration measured}
\end{align*}
\]
Data-Flow Scheduler: Input Engine

- **Input Engine**
  - transmits first intervals using best-effort approach
  - divides each interval into rounds
  - uses a non-blocking mechanism
  - broadcasts the actual meta-data to DD engines

\[ j \quad : \text{interval index} \]
\[ tm \quad : \text{time measured} \]
\[ tp \quad : \text{time proposed} \]
\[ durm \quad : \text{duration measured} \]
\[ durp \quad : \text{duration measured} \]
Data-Flow Scheduler: Workflow

- prop\[x\] = \{y , z\} \rightarrow \text{proposed meta-data [interval index] = \{tp, durp\}}

- act\[x\] = \{y , z\} \rightarrow \text{Actual meta-data [interval index] = \{tm, durm\}}

- mdurm\_x..y \rightarrow \text{median measured durations from interval x to y (sliding window of hist\_cnt)}
Data-Flow Scheduler: Fault Tolerance

- DD Engines are replicas to each other
  - same meta-data for each interval
  - Input Engines use the first received meta-data

- When a compute node fails
  - Build of history is required
    - Inconsistent meta-data prevention
  - Bandwidth recovery $\rightarrow$ SSU mechanism

- When an input node fails
  - Failure detection
  - Proposing to use fewer Input Engines

- DFS is able to recover the failures in both input and compute nodes
Implementation

• We implemented DFS on top of FLESnet
  o FLESnet used Infiniband Verbs API
  o We ported FLESnet to Libfabric to support modern interconnects

• Input and compute processes run on a separate single cores (single threaded)

• Two types of messages to communicate
  o Remote Direct Memory Access (RDMA): write MTSs
  o Message Passing (SYNC): coordinate between processing nodes

• MPI Barrier to synchronize input and compute processes
Cray XC40 using up to 384 nodes, each equipped with
  - two Intel Xeon E5-2680v3
  - 64 GiB of main memory

- CRAY mpich v7.5.1
- Libfabric v1.6.2
- Cray GCC v6.2.0
- micro-timeslice size 64 KiB
- Buffer size per node 32 MiB

We implemented Libfabric and MPI micro-benchmark with the communication pattern of FLESnet
  - To examine the maximum achievable bandwidth without any dependencies or buffer limitations

- FLESnet is tested **with** and **without** the DFS
Libfabric/MPI Micro-Benchmark (1)

- The benchmark uses
  - `fi_write` and `MPI_Put`
  - 2 MB of HugePages
  - `MPI_Barrier` before start transmission
  - Half of nodes as senders and half as receivers
- Each sender finishes writing data independently of other senders

- **MPI achieves shorter duration than Libfabric when Message Size > 128KiB:**

| Message Size | 64 KiB | 128 KiB | 1 MiB |
|--------------|--------|---------|-------|
| 64 nodes     |         |         |       |
| 128 nodes    |         |         |       |
| 192 nodes    |         |         |       |
| 384 nodes    |         |         |       |

| Message Size | 64 KiB | 128 KiB | 1 MiB |
|--------------|--------|---------|-------|
| 64 nodes     | 53.50% | 29.84%  | 65.73%|
| 128 nodes    | 28.65% | 08.70%  | 43.11%|
| 192 nodes    |        |         |       |
| 384 nodes    |        |         |       |

![Graph showing comparison between Libfabric and MPI for different message sizes](image)

- **smaller is better**
A significant performance drop with messages larger than 8 KiB using Libfabric

Libfabric achieves better bandwidth than MPI when message size at most 128KiB:

|      | Libfabric |      |      |      |
|------|-----------|------|------|------|
|      | 128n      | 192n | 384n |      |
| 64 KiB | + 53.52 %  | + 53.38 %  | + 65.76 %  |      |
| 128 KiB | + 28.68 %  | + 27.76 %  | + 43.17 %  |      |
| 1 MiB  | - 33.55 %  | - 34.44 %  | - 43.35 %  |      |

More bandwidth - Less bandwidth
Libfabric/MPI Micro-Benchmark (3)

- A mix of message sizes on Libfabric and MPI for comparison with FLESnet:
  - For each big message (size is variable)
    - One message of 16 bytes
    - Two message of 64 bytes each
  - 192 nodes are used

- Libfabric shows better aggregated bandwidth than MPI when message size < 1 MiB:

| Message Size | Libfabric |
|--------------|------------|
| 64 KiB       | ↑ 77.00 %  |
| 128 KiB      | ↑ 60.98 %  |
| 512 KiB      | ↑ 15.96 %  |
| 1 MiB        | ↓ 11.18 %  |
The Data Flow Scheduler vs FLEsNet
Achieved Throughput

• FLESnet uses a mix of message sizes:
  o MTS size → variable
  o MTS descriptor = 20 bytes
  o SYNC message = 87 bytes

• Libfabric Benchmark is with a mix of same message sizes

• DFS achieves better aggregated bandwidth than FLESnet compared to the Libfabric aggregated bandwidth:

|        | 128n  | 192n  | 384n  |
|--------|-------|-------|-------|
| FLESnet| 66 %  | 55 %  | 54 %  |
| DFS    | 99.5 %| 77 %  | 83 %  |
Synchronization Overhead (1)

- DFS shortens the duration to receive a complete time-slice at the compute nodes

- At least 30x reduction

- To receive a complete timeslice (median in ms):

|         | 64n  | 128n | 192n  | 384n  |
|---------|------|------|-------|-------|
| FLESnet | 267.72 | 957.44 | 1674.30 | 4223.63 |
| DFS     | 5.76  | 21.29 | 49.54  | 138.93  |
Synchronization Overhead (2)

Completion duration [ms] vs. Timeslice no. [1] (192 nodes)

Completion duration [ms] vs. Timeslice no. [1] (384 nodes)

Min, Max (bar: 10th - 90th percentile, median) (192 nodes)

Min, Max (bar: 10th - 90th percentile, median) (384 nodes)

FLESnet vs. DFS
Bandwidth Recovery

- Input Engines follow the actual proposed meta-data
- When network is stable and SSU is turned off, the duration of intervals stabilizes
- We simulated 25% artificial bandwidth drop for some intervals
  - Recovery of optimal duration
The buffer fill level aggregated across compute nodes shows the significant advantage of DFS:
- the buffer fill level is ordered in descending order.

With FLESnet: buffers are filled up
- Input nodes run out of tickets

With DFS: buffer fill level is only ~10% for all connections (~100% with FLESnet)
Buffer Usage (2)

Connections sorted by buffer fill level [1] (192 nodes)

Connections sorted by buffer fill level [1] (384 nodes)
Conclusion

- CBM expected data rate > 1 TiB/s

- The Data-Flow Scheduler (DFS)
  - synchronizes input nodes
  - utilizes the usage of the underlying network
  - reduces endpoint/network congestion

- Libfabric outperforms MPI with small messages
- DFS completes time-slices 30x faster
- DFS needs only 10% of the buffer space
- DFS adapts to network changes