MarlinMT - parallelising the Marlin framework.

CHEP 2019
Adelaide, Australia

Rémi Ete, F. Gaede
DESY
November 04, 2019
iLCSoft in a nutshell

https://github.com/iLCSoft

- Software stack the ILC experiment
- Nowadays used by many other experiments/collaborations
  → e.g: CLICdp, CEPC, CALICE, LCTPC, EU-Telescope
- Maintained by FLC @ DESY and CLICdp @ CERN

Main components
- **DD4hep**: Geometry description for simulation (Geant4) and reconstruction
- **LCIO**: Linear Collider IO and EDM (2003)
- **Marlin**: Reconstruction framework based on LCIO
The Marlin framework

[Link to Marlin GitHub repository](https://github.com/iLCSoft/Marlin)

Standard HEP event processing framework

- **LCIO** as event data model
- **LCCD** for data condition handling
- **AIDA** for histogram handling

Main components

- **Processor**: Main user module
- Application steered via [XML](https://example.com) files
- Plugin mechanism
- Standard event processing pipeline.
  - **All sequential**
  - **Parallelization at process level**
Choice of parallelism?

- **Event parallelism only**, one thread per event
- No task parallelism
  - Introduces thread-safety additional requirements at user/framework level
  - Not mandatory for a first prototype
- Use only STL library for multi-threaded scheduler
  - Keep implementation simple
  - Keep dependencies minimal
  - May use TBB later...

- Processing pipeline replicated for each thread
  - Modules are cloned or shared
  - Depends on memory / thread-safety requirements
The PEP scheduler

Scheduler architecture

![Diagram of the PEP scheduler]

- Workers (Thread pool)
- Input event queue (default size = 2xN)
- MT::LCReader
- Read event:
  1. disk IO
  2. uncompress buffer
  3. unpack buffer
- Push event
- Pop event
- LCIO files
- Output event pool
The PEP scheduler
First scaling performances

CPU crunching: crunch numbers for n ms

Amdahl’s law:

\[ S_{\text{latency}} = \frac{1}{1 - p + \frac{p}{s}} \]

- Parallel code is CPU crunching → Limited by sequential code fraction
- Main sequential code: → LCIO file reading!
- Need to reduce sequential code

AMD EPYC 7451 24-Core (2 x 24 x 2 threads)
The SIO library
https://github.com/iLCSoft/SIO

- SIO is the IO layer of LCIO
- Re-implemented with thread-safety in mind
- Features:
  - Schema evolution (block versioning)
  - Pointer chasing (e.g., linked-list)
  - Fast random access of records
  - Endianness agnostic (always write in big endian)
- LCIO event written in two records:
  - *Event header*: event summary (simple metadata)
  - *Event record*: all event collections

LCIO event structure

R. Ete — DESY — November 04, 2019 — Page 7
LCIO lazy unpacking with new SIO

**LCIO lazy unpacking**

1. Extract event record in a buffer
2. Move the buffer in LCEvent
3. Trigger full event decoding on first collection access

---

**Fast operations**
- Read from disk (IO)
- Uncompress record (zlib)
- Decode record

**Event header**
- Read from disk (IO)
- Decode record

**Full event data**
- Read from disk (IO)
- Uncompress record (zlib)
- Decode record

**CPU intensive !!**

---

**LCIO event decoding**

**Trigger CPU intensive decoding only on demand!**
MarlinMT scheduler
Parallel event processing - *With lazy unpacking*

- **Workers** (Thread pool)
  - **W1**
  - **W2**
  - **WN**

**Input event queue**
(default size = 2xN)

**Output event pool**

**MT::LCReader**

- **Read event:**
  1. disk IO

- **Push event**

- **Pop event**

**LCIO files**

- **Output event pool**

**Module 1**
**Module 2**
**Module N**

- **(2) uncompress buffer**
- **(3) unpack buffer**
MarlinMT scheduler
Scaling performances with lazy unpacking

- With LCIO lazy unpacking

AMD EPYC 7451 24-Core (2 x 24 x 2 threads)
Histogramming and multi-threading

Common dilemma of histograms in multi-threaded frameworks:

- One histogram in N threads? Nice for memory! Killing CPU!
- N histogram copies in N threads? Nice for CPU! Killing memory!

What to choose?
Histogramming and multi-threading

Common dilemma of histograms in multi-threaded frameworks:

- One histogram in N threads? Nice for memory! Killing CPU!
- N histogram copies in N threads? Nice for CPU! Killing memory!

What to choose?

Why not both in a framework?
Let the user choose where to win (and to lose)!
Histogramming and multi-threading

Common dilemma of histograms in multi-threaded frameworks:
• One histogram in N threads? Nice for memory! Killing CPU!
• N histogram copies in N threads? Nice for CPU! Killing memory!
What to choose?

Why not both in a framework?
Let the user choose where to win (and to lose)!

Currently designing a book store MarlinBook
• ROOT7 histograms ROOT::Experimental::RHist
• Both histogram copy/shared at runtime
• Shared histograms: use thread local buffering
• ROOT6 or ROOT7 files
• Evaluating new RNTuple instead of TTree
Conclusion and outlook

Conclusion

• A prototype of MarlinMT has been developed:
  • Schedule parallel event processing in worker threads
  • Benchmarking doesn't show big issues → keep this implementation for now

• New SIO implementation
  • Full control on read/write steps: disk I/O, un/compression, en/de-coding
  • Allow for lazy data en/de-coding → Speeds up MarlinMT!
  • New SIO implementation used in other projects: see talk by F. Gaede on PODIO

Outlook

• Implement the book store MarlinBook (started)
• Porting existing reconstruction code will be a long journey
Backups
## LCIO reading performance

Comparison of extract and decode times

<table>
<thead>
<tr>
<th></th>
<th>Extract time</th>
<th>Decode time</th>
<th>Read fraction</th>
<th>Event size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SIM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single photon</td>
<td>2.583</td>
<td>33.116</td>
<td>12.821</td>
<td>1.34</td>
</tr>
<tr>
<td>uds 200 GeV</td>
<td>9.399</td>
<td>79.45</td>
<td>8.453</td>
<td>3.05</td>
</tr>
<tr>
<td>4f WW had</td>
<td>20</td>
<td>171</td>
<td>8.55</td>
<td>5.85</td>
</tr>
<tr>
<td><strong>REC</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single photon</td>
<td>1.916</td>
<td>26.25</td>
<td>13.7</td>
<td>2.5</td>
</tr>
<tr>
<td>uds 200 GeV</td>
<td>9.749</td>
<td>156.5</td>
<td>16.053</td>
<td>1</td>
</tr>
<tr>
<td>4f WW had</td>
<td>20.1</td>
<td>331.7</td>
<td>16.502</td>
<td>55.6</td>
</tr>
<tr>
<td><strong>DST</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single photon</td>
<td>0.085</td>
<td>0.088</td>
<td>1.035</td>
<td>0.0016</td>
</tr>
<tr>
<td>uds 200 GeV</td>
<td>0.179</td>
<td>1.51203</td>
<td>8.447</td>
<td>0.052</td>
</tr>
<tr>
<td>4f WW had</td>
<td>0.334</td>
<td>3.118</td>
<td>9.335</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table: LCIO file read performances (single thread). Read times in *milli-seconds*. Event size in MG. Read fraction defined as decode / extract. Results may vary on different machines.
Benchmarking machine

- Architecture: x86_64
- Byte Order: Little Endian
- CPU(s): 96
- Thread(s) per core: 2
- Core(s) per socket: 24
- Socket(s): 2
- NUMA node(s): 8
- CPU family: 23

- Model name: AMD EPYC 7451 24-Core Processor
- CPU MHz: 1200.000
- CPU max MHz: 2300.0000
- CPU min MHz: 1200.0000
- L1d cache: 32K
- L1i cache: 64K
- L2 cache: 512K
- L3 cache: 8192K