Multi-Threading for Baikal-GVD Core Software Framework

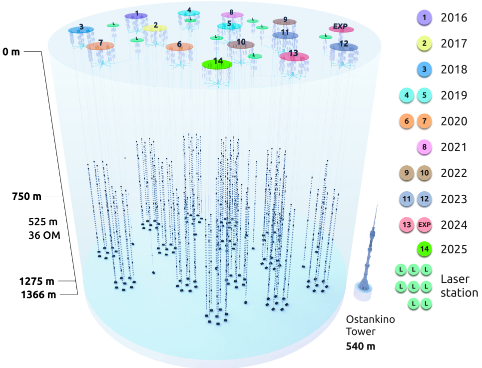
A.G. Solovjev
on behalf of the Baikal-GVD Collaboration

Joint Institute for Nuclear Research, Dubna, Russia, 141980

Distributed Computing and Grid-technologies
in Science and Education, 7-11 July 2025

Introduction

Baikal-GVD neutrino telescope in Lake Baikal



14 clusters (0.7 km^3), 8 strings/cluster, 36 OMs/string.
Expanding yearly (+1–2 clusters/year).

Data Processing System

- **Core**: BARS framework (C++)
- **Management**: Python workflow

Parallelism Levels

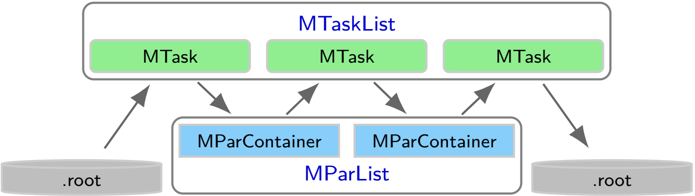
- Cluster-level (distributed VMs)
- Workflow-level (parallel processing sections):
  - Fast (2-13 min/file)
  - Offline (1-5 hr/run)
- Thread-level (this work)

A. G. Solovjev et al. (Baikal-GVD), "Review of Software for Automatic Processing and Analysis of Data from the Baikal-GVD Neutrino Observatory". In: *Parallel Computational Technologies*. Ed. by L. Sokolinsky et al. Springer, 2024, pp. 80–91. https://doi.org/10.1007/978-3-031-73372-7_6

BARS Event Processing Architecture

Processing Workflow

1. **Pre-processing**:
   - Register containers
   - Load parameters
2. **Processing**:



3. **Post-processing**:
   - Finalize results
   - Release resources

Event Loop Code

```
Int_t rc = kTRUE;
// Phase 1: Pre-process all tasks
for (MTask *task : fTaskList) {
  rc = task->PreProcess(fParList);
}
// Phase 2: Main processing loop
if (rc) {           Parallelization Target
  while (true) {
    fParList->Reset();
    for (MTask *task : fTaskList) {
      rc = task->Process();
    }
    if (!rc) break;
  }
}
// Phase 3: Post-process all tasks
for (MTask *task : fTaskList) {
  rc = task->PostProcess(fParList);
}
return rc;
```

Parallelization Requirements and Performance

Key Constraints

- **Event Processing Order** must be strictly preserved
- **Task Execution Order** within tasklist must be maintained
- **Task Exclusivity**: identical tasks must not run simultaneously

Theoretical Speedup

$$T_{\text{sequential}} = \bar{t}_{\text{tasklist}} \cdot N_{\text{events}}$$

$$T_{\text{parallel}} \approx (\bar{t}_{\text{tasklist}} + \bar{t}_{\text{idle}}) \cdot \frac{N_{\text{events}}}{n_{\text{threads}}}$$

$$S \approx \frac{\bar{t}_{\text{tasklist}}}{\bar{t}_{\text{tasklist}} + \bar{t}_{\text{idle}}} \cdot n_{\text{threads}}$$

OpenMP Tasking

```
// Thread pool setup
#pragma omp parallel
#pragma omp single
while (true) {
  #pragma omp task
  {
    // Process tasklist
    fParList->Reset();
    for (MTask *task : fTaskList) {
      omp_set_lock(task->fLock);
      rc = task->Process();
      omp_unset_lock(task->fLock);
    }
    if (!rc) break;
  }
}
```

Amdahl's law

$$S_{\text{total}} = \frac{1}{(1-p) + \frac{p}{S}}, \quad p = \frac{T_{\text{Process}}}{T_{\text{total}}}$$

Optimization Strategies

Core Approaches to Reduce Idle Time ($\bar{t}_{idle}$)

- **Task Decomposition**
  - Split long task into smaller subtasks
  - Advantages:
    - Enables dynamic load balancing
    - Minimizes synchronization overhead
- **Task Optimization**
  - Optimize individual task execution time
  - Methods:
    - Profile and optimize critical paths
    - Implement strategic caching

Approaches to Increase Parallel Portion (p)

- **Pre/Post-Process Optimization**
  - Offload critical paths to separate processes
  - Pre-cache frequently used data

Why Many Small Tasks Work Better

- **Few large tasks** (causes thread starvation)

- **Many small tasks** (maximizes core utilization)

- **Advantages**
  - More tasks $\Rightarrow$ Better load balancing
  - Smaller tasks $\Rightarrow$ Less idle time

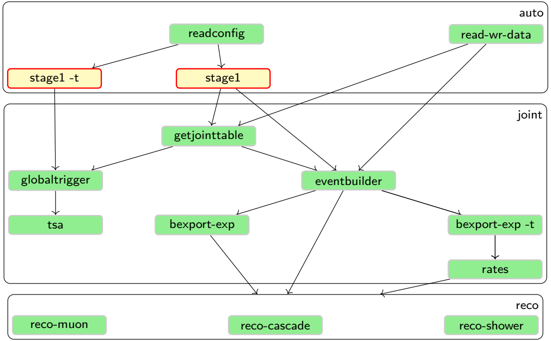
Optimized Programs by Processing Stage

Basic parallelization does not speed up enough

Per-File Workflow

Key optimized program:

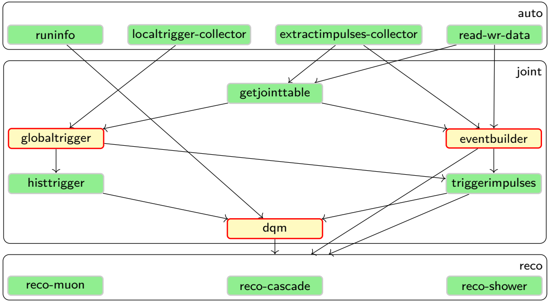
- `stage1`



Per-Run Workflow

Key optimized program:

- `globaltrigger/eventbuilder`
- `dqm`



Flame Graph Analysis for `stage1`

Profiled Command

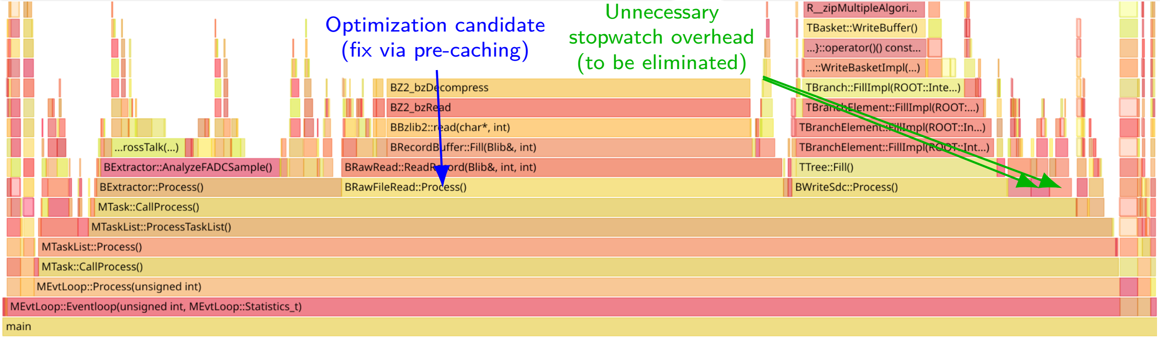
```
stage1
--season 2024 --cluster 2 --run 230 --file 1
--create-chain --absolute-time --raw-write-additional
```

(261124 events)

Flame graphs

Intuitive visualization of profiling:

- element width — execution time
- hierarchy — call stacks



Parallelizable region

Parallelization Strategies for `stage1`

`stage1 --season 2024 --cluster 2 --run 230 --file 1 --create-chain ...`

- **Original parallel execution of independent instances** (MT faster, 4 cores)
  - `--absolute-time --raw-write-additional` *(extr.events)*
  - `--mt 2`
  - `--loc-trigger-write-only` *(loc.trigger)*
  - `--mt 2`
- **Parallel execution with optimized options combinations** (MT faster, 4 cores)
  - `--absolute-time` *(extr.events)*
  - `--mt 2`
  - `--loc-trigger-write-only --raw-write-additional` *(loc.trigger)*
  - `--mt 2`
- **Merged execution** (MT no slowdown, 2 cores)
  - `--absolute-time --raw-write-additional --loc-trigger-write` *(merged)*
  - `--mt 2`

0 10 s 20 s 30 s

Parallelization Bottlenecks in `globaltrigger/eventbuilder`

Execution Details

- **Command:**
  `globaltrigger`
  `--season 2024 --cluster 2 --run 230`
  `--mt 2` [2 threads]
- **Execution Log:**
  Main Tasks:

```
88.9% BGlobTriggerBuilder (idle 47.4%)
 9.0% BWriteTree          (idle 36.3%)
```

Performance

- Efficiency: $\approx$ **54%**
- Speedup: —

Current Implementation

```
class BGlobTriggerBuilder : public MTask {
  Int_t Process(int onThread) {
    for (auto sdc : SDCs) {
      // Processes ALL SDCs sequentially
    }
  }
};
```

Core Problem

- **Sequential** SDC processing in a loop
- **Self-blocking** in multi-threaded execution

Solution

- Decompose monolithic SDC processing into per-SDC tasks

Task Decomposition in globaltrigger/eventbuilder

Improved Implementation

```
// Per-SDC parallel tasks
class BGlobTriggerBuilderSdc : public MTask {
  Int_t Process(int onThread) {
    // Processes individual SDC
  }
};

// Main container
class BGlobTriggerBuilder : public MTaskList {
  BGlobTriggerBuilder() {
    for (auto sdc : SDCs) {
      AddToList(new BJointEventBuilderSdc(sdc));
    }
  }
};

// User code
- tasks.AddToList(&globtrigger);
+ tasks.AddToList(*globtrigger.GetList());
```

Execution Log

SDC Tasks:

```
2.5% BGlobTriggerBuilder190 (idle 1.3%)
2.0% BGlobTriggerBuilder192 (idle 1.7%)
1.9% BGlobTriggerBuilder193 (idle 1.5%)
...
3.2% BGlobTriggerBuilder215 (idle 1.3%)
```

Parallel Performance

	2 cores	3 cores	4 cores
• Efficiency:	**92%**	**86%**	**81%**
• Speedup:	**1.7×**	**2.3×**	**2.8×**

Flame Graph Analysis for DQM

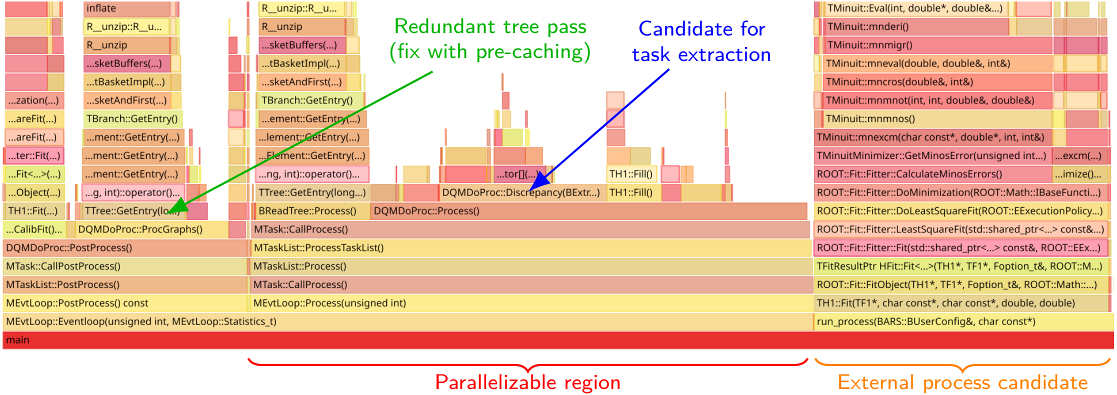
Profiled Command

```
dqm-cli
--season 2024 --cluster 9 --run 112
--exponent --poisson --uniformity --tdqm --charge
```

Execution Time

≈ **4 minutes**

(729886 events)



Profiled Command

dqm-cli
--season 2024 --cluster 9 --run 112
~~--exponent~~ --poisson --uniformity --tdqm --charge
--mt 2

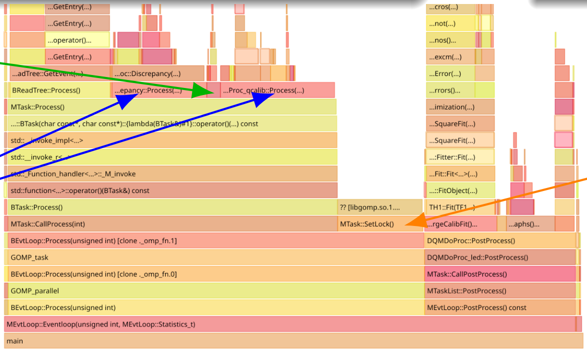
[2 threads]

Speedup

≈ 2× faster
(4 min ↓ 2 min)

Pre-caching tree pass

Long task decomposition



Remaining synchronization overhead

Parallelized region (efficiency ≈75%)

Conclusion

Key Achievements

- Successfully implemented multi-threading in Baikal-GVD's core framework:
  - **Eventloop parallelization** via OpenMP tasking
  - **Optimization strategies** via task granularity
- Achieved significant speedups across processing stages:
  - `stage1`: stable performance during combined execution without increased resource consumption
  - `globaltrigger/eventbuilder`: good scaling via parallel detector section processing
  - `dqm`: $2\times$ speedup via improved load balancing and elimination of sequential bottlenecks

Future Directions

- **Refactoring processing graphs** to enable multithreading support
- **Eliminating exclusivity requirement** through tasks refactoring

Conclusion

Key Achievements

- Successfully implemented multi-threading in Baikal-GVD's core framework:
  - **Eventloop parallelization** via OpenMP tasking
  - **Optimization strategies** via task granularity
- Achieved significant speedups across processing stages:
  - `stage1`: stable performance during combined execution without increased resource consumption
  - `globaltrigger/eventbuilder`: good scaling via parallel detector section processing
  - `dqm`: $2\times$ speedup via improved load balancing and elimination of sequential bottlenecks

Future Directions

- **Refactoring processing graphs** to enable multithreading support
- **Eliminating exclusivity requirement** through tasks refactoring

Thank you!