

МНОГОПОТОЧНОСТЬ ДЛЯ БАЗОВОГО ПРОГРАММНОГО ФРЕЙМВОРКА Baikal-GVD

А. Г. Соловьев^{1, *}

от имени коллаборации Baikal-GVD

¹ Объединенный институт ядерных исследований, Дубна, Россия

Представлена реализация многопоточности в программном фреймворке нейтринного телескопа Baikal-GVD. Описан метод параллелизации цикла обработки событий с использованием OpenMP, обеспечивающий сохранение порядка выполнения. Решение демонстрирует повышение производительности при сохранении целостности данных.

This paper presents multi-threading implementation in the Baikal-GVD neutrino telescope software framework. We describe an OpenMP-based parallelization method for event processing that maintains execution order. The solution shows performance gains while preserving data integrity.

PACS: 07.05.Kf; 29.85.-c

ВВЕДЕНИЕ

Baikal-GVD — нейтринная обсерватория в озере Байкал [1]. Ядро ее ПО основано на фреймворке BARS (Baikal Analysis and Reconstruction Software) [2] и реализует все этапы обработки данных. Система автоматизации организует обработку через два последовательных рабочих процесса: *быструю обработку* (2–13 мин/файл) и *офлайн-обработку* (1–5 ч/ран), оба с внутренним параллелизмом [3].

В работе представлена реализация многопоточности в BARS, дополняющая существующее распараллеливание. Основной целью является снижение времени обработки данных, в первую очередь на ранних этапах, что важно для многоканальной астрономии в онлайн.

1. АРХИТЕКТУРА ОБРАБОТКИ СОБЫТИЙ В BARS

BARS использует две ключевые абстракции для обработки данных:

- **контейнеры** (MParContainer) хранят экспериментальные данные и параметры обработки, создаются до обработки;
- **задачи** (MTask) преобразуют данные контейнеров, выполняя элементарный шаг обработки.

* E-mail: solovjev@jinr.ru

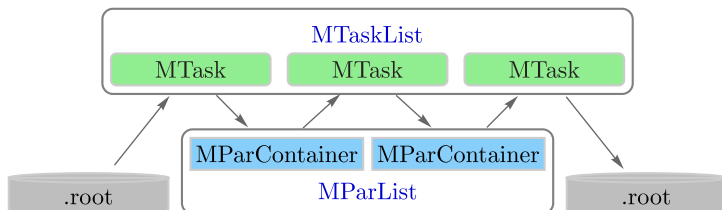


Рис. 1. Схема цикла обработки событий

Процесс обработки организован в три этапа, каждый из которых включает полный проход по всем задачам списка **MTaskList**:

1. **Pre-processing**: в **MTask::PreProcess** задачи регистрируют контейнеры и загружают параметры.

2. **Processing**: цикл обработки событий (рис. 1) последовательно выполняет задачи для каждого события через **MTask::Process**. Задачи взаимодействуют через общий список контейнеров (**MParList**).

3. **Post-processing**: завершение в **MTask::PostProcess**, где задачи финализируют результаты и освобождают ресурсы.

Ошибки (**kFALSE**) на любом этапе прерывают выполнение. Эта архитектура с четкими ответственностями и строгим порядком выполнения позволяет надежно реализовать многопоточность.

2. РЕАЛИЗАЦИЯ МНОГОПОТОЧНОЙ ОБРАБОТКИ

Цикл событий, показанный на рис. 1, является целью для параллелизации. Для многопоточности требуется дублирование контейнеров между потоками и соблюдение ключевых условий:

- **порядок обработки событий** должен строго сохраняться;
- **порядок выполнения задач** в списке задач должен соблюдаться;
- **эксклюзивность задач**: идентичные задачи не должны выполняться одновременно.

Первые два условия обеспечивают воспроизводимость, а третье — предотвращает состояние гонки при параллельном выполнении из-за наличия внутреннего состояния **MTask**.

В реализации используется стандартный подход на основе OpenMP:

```

#pragma omp parallel
#pragma omp single
while (true) {
    #pragma omp task
    {
        // Tasklist processing
    }
}
  
```

Для удовлетворения указанных требований каждой задаче **MTask** в списке задач назначается блокировка, гарантирующая как эксклюзивное выполнение, так и порядок обработки событий.

Параллелизм достигается по принципу конвейера: обработка следующего события начинается сразу после завершения первой задачи предыдущего события, образуя «лесенку» из перекрывающихся задач.

Ускорение оценивается как

$$S \approx \frac{\bar{t}_{\text{tasklist}}}{\bar{t}_{\text{tasklist}} + \bar{t}_{\text{idle}}} n_{\text{threads}}, \quad (1)$$

где дробь отражает эффективность распараллеливания. Меньшее время простоя $\bar{t}_{\text{idle}}$ означает лучшую производительность. Для его минимизации используются:

- разделение длинных задач на отдельные подзадачи;
- оптимизация времени выполнения отдельных задач.

Более короткое время выполнения задач уменьшает время простоя и дает больше возможностей для автоматической балансировки нагрузки.

Оценка (1) применима только к этапу Process. Согласно закону Амдала* общее ускорение, включая PreProcess и PostProcess, составляет

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

т. е. время выполнения в Pre/Post-Process ограничивает общее ускорение.

3. ОПТИМИЗАЦИЯ КРИТИЧЕСКИХ ЭТАПОВ ОБРАБОТКИ

Базовая параллелизация часто оказывается недостаточной. В этом разделе показаны примеры более глубоких оптимизаций.

Программа stage1 остается узким местом в пофайловой обработке, несмотря на параллельное выполнение с разными параметрами. Кроме того, сжатые входные данные читаются дважды.

После оптимизации с помощью flame graph** (рис. 2) параллельные экземпляры **stage1** достигают ускорения, но с более высокими затратами ресурсов. Оптимизированная версия позволяет выполнение в одном вызове с параллелизмом на 2 ядра, сохраняя производительность и устраняя избыточные чтения.

Программы globaltrigger и eventbuilder имеют схожую структуру, что позволяет анализировать их совместно. Профилирование первой выявило критическое узкое место — доминирующую длинную задачу

* Закон Амдала гласит, что максимальное ускорение, достигаемое за счет распараллеливания программы, принципиально ограничено той ее частью, которая должна выполняться последовательно.

** Flame graph предоставляет интуитивно понятную визуализацию профилирования: ширина элемента отражает время выполнения, а иерархия — стек вызовов.

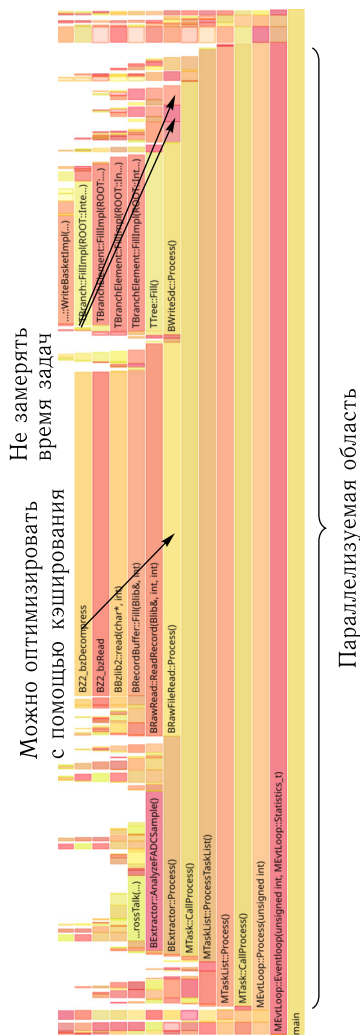
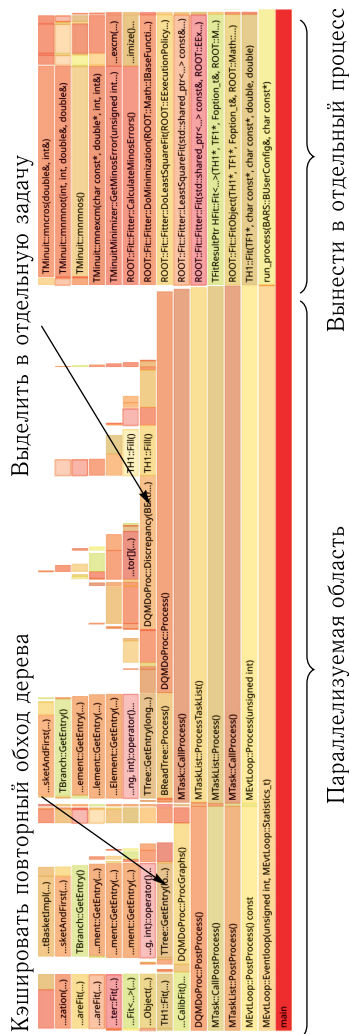


Рис. 2. Анализ flame graph выполнения stage1



BGlobTriggerBuilder, самоблокирующуюся при параллельном выполнении из-за последовательной обработки секций детектора (SDCs).

Оптимизация включает архитектурный рефакторинг:

- введен новый класс для изолированной обработки отдельных SDC;
- основной класс преобразован в контейнер на основе **MTaskList**, управляющий экземплярами этого класса.

Параллельная реализация с использованием **MTaskList::GetList** демонстрирует хорошую масштабируемость, с ускорениями $1,7\times$ (2 ядра), $2,3\times$ (3 ядра) и $2,8\times$ (4 ядра) и эффективностью распараллеливания 81–92%.

Программа DQM является одним из ключевых узких мест в порановой обработке. Анализ flame graph (рис. 3) выявил:

- значительную часть времени выполнения в последовательном коде;
- доминирующую длительную задачу в цикле событий.

Оптимизация дала ускорение в 2 раза: самая длинная вариация **dqm** теперь эффективно параллелизуется на 2 потока (75% эффективности), превосходя оригинал. Дополнительные накладные расходы (2 процесса + многопоточность) допустимы, так как на этом этапе нет дефицита ядер.

ЗАКЛЮЧЕНИЕ

В работе успешно реализована многопоточность в базовом фреймворке Baikal-GVD с использованием:

- **параллелизации цикла событий** через задачи OpenMP;
- **стратегий оптимизации** через гранулярность задач.

Предложенный подход показал эффективность при ускорении критических компонентов пофайловой и порановой обработки. Полученные результаты создают основу для будущего рефакторинга графов обработки с целью поддержки многопоточности и ускорения всего конвейера.

СПИСОК ЛИТЕРАТУРЫ

1. *Allakhverdyan V. A. et al. (Baikal-GVD Collab.). Status of the Baikal-GVD Neutrino Telescope and Main Results // Bull. Russ. Acad. Sci. Phys. 2023. V. 87, No. 7. P. 1059–1062.*
2. *Avorin A. D. et al. (Baikal-GVD Collab.). Automatic Data Processing for Baikal-GVD Neutrino Observatory // PoSI. 2021. V. CRC2021. P. 1040; arXiv:2107.13939.*
3. *Solovjev A. G. et al. (Baikal-GVD Collab.). Review of Software for Automatic Processing and Analysis of Data from the Baikal-GVD Neutrino Observatory // Parallel Computational Technologies / Ed. by L. Sokolinsky et al. Cham: Springer Nature Switzerland, 2024. P. 80–91.*