

Санкт-Петербургский государственный университет

Программная инженерия

Группа 24.M71-мм

Оптимизация производительности пакета BmNRoot

Ли Цзишэн

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
доцент кафедры вычислительной физики, к. ф.-м. н., Немнюгин С. А.

Санкт-Петербург
2025

Оглавление

Введение	3
1. Обзор	5
1.1. Ключевые факторы, влияющие на производительность	5
1.2. Анализ производительности и план оптимизации	5
2. Анализ производительности	8
2.1. Цели анализа производительности	8
2.2. Процедура анализа производительности	8
2.3. Результаты анализа производительности	9
3. Практика оптимизации производительности	12
3.1. Разработка и оценка стратегий	12
3.2. Реализация схемы параллелизма данных	13
3.3. Реализация схемы параллелизма данных	15
Заключение	18
Приложение 1 Автоматизированный тестовый скрипт	20
Список литературы	22

Введение

BM@N (Baryon Matter at Nuclotron) - первый эксперимент по ядерной физике с фиксированной мишенью в рамках программы исследований на ускорительном комплексе NICA, и его успешная реализация в значительной степени зависит от эффективной и надежной программной поддержки[1]. Пакет [BmnRoot](#), являющийся основным инструментом обработки данных для эксперимента BM@N, основан на среде [CERN ROOT](#), библиотеке моделирования [Geant4](#) и объектно-ориентированной среде [FairRoot](#). предоставляет мощные инструменты для исследования работы детектора, моделирования событий, реконструкции примеров и анализа физических данных, а также для выполнения других функций[2]. С помощью BmnRoot исследователи могут моделировать и калибровать подсистемы детектора BM@N, а также реконструировать и анализировать события столкновений, тем самым поддерживая задачи экспериментальной физики.

Однако текущее программное обеспечение BmnRoot имеет ряд проблем с точки зрения вычислительной производительности. Во-первых, существующий фреймворк обнаруживает узкие места в вычислительной производительности при больших наборах данных и высоких требованиях к пропускной способности. Когда Geant4, ROOT и FairRoot последовательно ввели поддержку многопоточности (например, Geant4MT), код BmnRoot пришлось модифицировать, чтобы адаптировать к требованиям многопоточной параллельной работы[2]. Это говорит о том, что отсутствие распараллеливания является одной из причин длительного времени вычислений. Во-вторых, есть недостатки и в управлении памятью. При длительном крупномасштабном моделировании объем памяти BmnRoot может постоянно увеличиваться, могут возникать проблемы с несвоевременным освобождением ресурсов или утечкой памяти. Подобные проблемы с управлением памятью могут привести к излишнему расходу ресурсов и снижению стабильности программы.

Поэтому проведение оптимизационных исследований узких мест

производительности и проблем управления памятью в BmnRoot имеет большое практическое значение и прикладную ценность.

1. Обзор

1.1. Ключевые факторы, влияющие на производительность

BMNROOT — это широко используемый фреймворк для моделирования в экспериментах физики высоких энергий, производительность модуля генерации событий которого напрямую влияет на эффективность всего экспериментального моделирования [1]. Проведя углубленный анализ, мы выявили два ключевых фактора, влияющих на эффективность работы модуля генерации событий:

Во-первых, вычислительная нагрузка, обусловленная большим масштабом событий. Модуль генерации событий должен обрабатывать значительное количество событий, для каждого из которых требуется создание множества экземпляров объектов, что приводит к существенным накладным расходам на выделение памяти и инициализацию. Исследования показывают, что в моделировании детекторов частиц создание и управление объектами может потреблять более 30% общего времени вычислений.

Во-вторых, высокая вычислительная сложность. Каждое событие включает в себя множество физических расчетов, связанных с поведением частиц, причем эти расчеты содержат структуры с множественными вложенными циклами. В частности, в моделировании детектора GEM такие процессы, как перенос частиц, депозиция энергии и генерация сигналов, связаны со сложными математическими операциями, формирующими основное вычислительное узкое место[3].

1.2. Анализ производительности и план оптимизации

1.2.1. Методология анализа производительности

Для комплексной оценки производительности системы была принята многоуровневая стратегия анализа:

1. Анализ памяти: Использование набора инструментов Valgrind для анализа использования памяти. [Valgrind](#) предоставляет функции детектирования утечек памяти и идентификации недопустимых обращений к памяти; его уникальная технология динамического бинарного инstrumentирования позволяет обнаруживать проблемы с памятью во время выполнения без необходимости перекомпиляции программы[4].
2. Профилирование производительности: Использование интегрированного в ядро Linux инструмента [perf](#) для системного анализа производительности. [perf](#), основанный на аппаратных счетчиках производительности, позволяет собирать данные о производительности CPU с очень низкими накладными расходами (обычно менее 5%), поддерживает мониторинг аппаратных и программных событий, анализ сэмплирования и генерацию флеймграфов.
3. Тестирование в различных сценариях: Проведение сравнительного анализа производительности для трех генераторов частиц (ION, BOX и PART), поддерживаемых модулем генерации событий, при одинаковом количестве событий с целью выявления характеристик производительности в различных условиях.

1.2.2. Разработка стратегии оптимизации

На основе результатов анализа производительности были разработаны три схемы параллельной оптимизации с разной гранулярностью:

1. Параллелизм на уровне событий: Реализация параллельной обработки на уровне событий с использованием [OpenMP](#). Путем введения параллелизма в цикл по событиям fRun->Run(nEvents) в макросе run_sim_bmn.C данный метод обеспечивает простоту программирования, но требует учета проблем безопасности потоков[5]. Исследования показывают, что параллелизм на уровне событий в моделировании физики частиц обычно обеспечивает близкое к линейному ускорение.

2. Параллелизм на уровне процессов: Реализация параллелизма на уровне процессов с использованием MPI. MPI позволяет упростить проблемы потокобезопасности за счет избегания разделяемого доступа к памяти, подходит для обработки событий в крупном масштабе и обладает хорошей отказоустойчивостью[6]. Однако данный метод требует значительной переработки кода, и каждый процесс должен независимо инициализировать экземпляры, такие как геометрия и магнитное поле, что приводит к дополнительным накладным расходам.
3. Параллелизм данных: Использование OpenMP для тонкозернистого распараллеливания вычислений, связанных с поведением частиц. Для циклических структур, вычислительно интенсивных и с минимальной конкуренцией за данные, OpenMP предоставляет простой и эффективный путь параллелизации[7]. В моделировании физики высоких энергий было доказано, что параллелизм данных эффективно повышает вычислительную эффективность.

2. Анализ производительности

2.1. Цели анализа производительности

Данный анализ производительности был сосредоточен на следующих трех аспектах:

1. Обнаружение и идентификация потенциальных утечек памяти.
2. Точное определение «горячих точек» функций (hot spots) путем сэмплингового анализа.
3. Определение ключевого пути оптимизации на основе анализа кода.

2.2. Процедура анализа производительности

2.2.1. Анализ памяти с помощью Valgrind

Был проведен углубленный анализ памяти макроса run_sim_bmn.C с использованием Valgrind:

```
valgrind --leak-check=full --show-leak-kinds=all \
--log-file=valgrind_10events.log \
root -b -q ~/bmnroot/macro/run/run_sim_bmn.C
```

2.2.2. Профилирование производительности с помощью Perf

Для системной оценки характеристик производительности был разработан автоматизированный скрипт анализа (Полный код представлен в Приложении 1):

```
GENERATOR="ION"
echo "Запуск perf record..."
perf record -g --call-graph=fp --freq=997 -o \
"${GENERATOR}_perf.data" \
root -b -q "/home/vodka/bmnroot/macro/run/run_sim_bmn.C"
```

```
echo "Генерация флеймграфа..."  
perf script -i "${GENERATOR}_perf.data"  
stackcollapse-perf.pl > out.perf-folded  
flamegraph.pl out.perf-folded > "${GENERATOR}_flamegraph.svg"
```

2.3. Результаты анализа производительности

2.3.1. Результаты анализа Valgrind

Отчет анализа Valgrind показал:

```
==2422== Memcheck, a memory error detector  
==2422== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.  
==2422== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info  
==2422== Command: root -b -q /home/vodka/bmnroot/macro/run/run_sim_bmn.C  
==2422== HEAP SUMMARY:  
==2422== in use at exit: 0 bytes in 0 blocks  
==2422== total heap usage: 4 allocs, 4 frees, 73,848 bytes allocated  
==2422== All heap blocks were freed -- no leaks are possible  
==2422== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Результаты свидетельствуют:

1. Отсутствие утечек памяти, все блоки кучи корректно освобождены.
2. Отсутствие ошибок памяти, программа не выполняла недопустимых операций с памятью.
3. Высокая эффективность использования памяти, осуществлено лишь небольшое количество операций выделения/освобождения памяти.

2.3.2. Результаты анализа производительности с помощью Perf

С помощью perf были сгенерированы флеймграфы для трех генераторов частиц (Рис. 1-3). Ширина флеймграфа отражает долю времени CPU, затрачиваемого на цепочку вызовов функций, а вертикальное направление показывает отношения вызовов[8]. Анализ показал,

что функция `BmnGemStripModule::AddRealPointFull` является основным "горячим" участком, потребляющим большую часть вычислительных ресурсов.

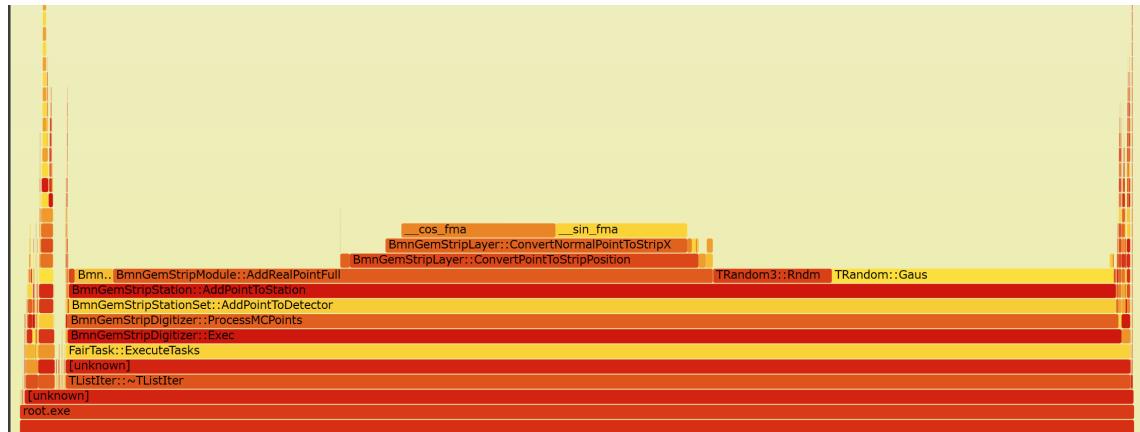


Рис.1 ION_flamegraph

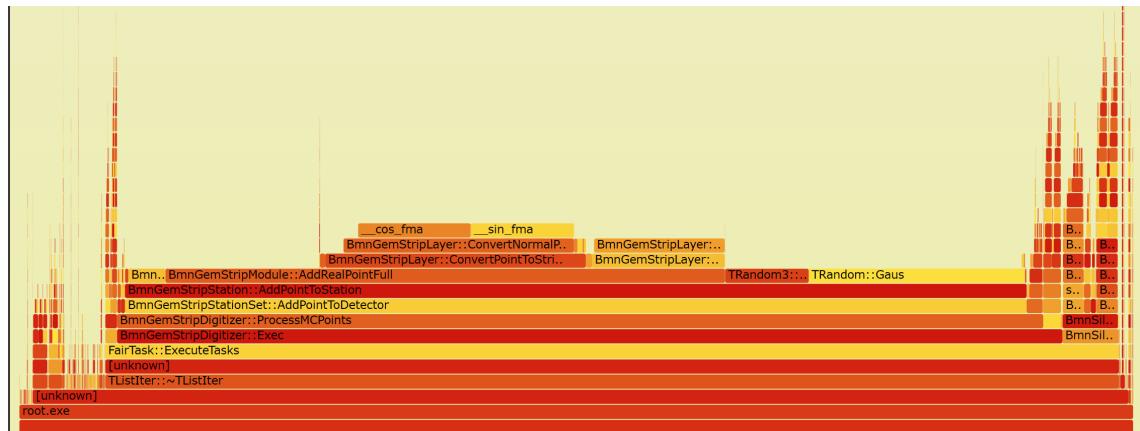


Рис.2 PART_flamegraph

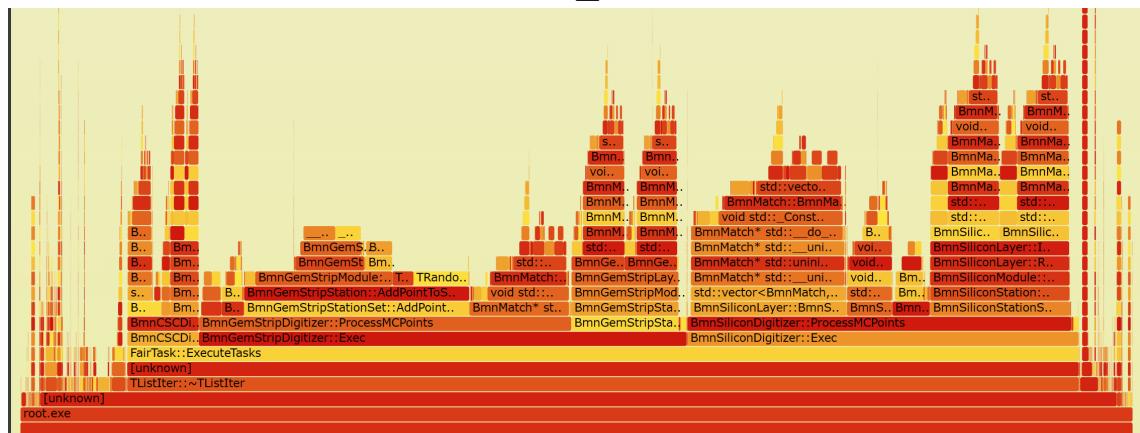


Рис.3 BOX_flamegraph

2.3.3. Выводы анализа производительности

Обобщая результаты анализа производительности, мы идентифицировали ключевое узкое место: структуру многоуровневых вложенных циклов в функции BmnGemStripModule::AddRealPointFull:

```
// Цикл по точкам столкновения
for(Int_t icoll = 0; icoll < collision_points.size(); ++icoll) {
    // Цикл по электронам
    for(Int_t ielectron = 0; ielectron < n_electrons_cluster; ++ielectron) {
        // Цикл по усилению
        for(int igain = 0; igain < total_gain; ++igain) {
            // Цикл по слоям
            for(Int_t ilayer = 0; ilayer < n_strip_layers; ++ilayer) {
                // Сложные вычисления...
            }
        }
    }
}
```

Данная циклическая структура вызывается в макросе run_sim_bmn.C через функцию Run(nEvents) экземпляра FairRunSim, формируя основное вычислительное узкое место.

3. Практика оптимизации производительности

3.1. Разработка и оценка стратегий

Первоначальная схема предполагала реализацию параллелизма на уровне событий для fRun->Run(nEvents) с помощью OpenMP:

```
// После fRun->Init(), добавить следующий код:  
// Создание нескольких экземпляров FairRunSim (по одному на поток)  
std::vector<FairRunSim*> runs;  
std::vector<FairPrimaryGenerator*> generators;  
  
// Начало параллельной обработки на уровне событий  
#pragma omp parallel for num_threads(4)  
// Настроить в зависимости от числа ядер CPU  
for (int iEvent = 0; iEvent < nEvents; iEvent++) {  
    #pragma omp critical  
    {  
        // Создание независимого экземпляра для выполнения для каждого потока  
        if (runs.size() <= omp_get_thread_num()) {  
            // Копирование конфигурации исходного fRun...  
            FairRunSim* newRun = new FairRunSim();  
  
            // Копирование конфигурации исходного primGen...  
            FairPrimaryGenerator* newPrimGen = new FairPrimaryGenerator();  
  
            runs.push_back(newRun);  
            generators.push_back(newPrimGen);  
            newRun->Init();  
        }  
    }  
}  
  
// Обработка одиночного события  
runs[omp_get_thread_num()]->Run(1);  
}
```

Однако тестирование выявило следующие проблемы данной схемы:

1. Параметры конфигурации fRun включают множество экземпляров, содержащих std::vector, не поддерживающие многопоточные операции.
2. Некоторые модули FairRoot не гарантируют потокобезопасность.
3. Ручная настройка многочисленных параметров чрезвычайно подвержена ошибкам.

Следовательно, схема параллелизма на уровне событий требовала значительной переработки кода и была сопряжена с высокой сложностью.

Рассматриваемые альтернативные схемы: Параллелизм на уровне процессов с использованием MPI и параллелизм данных для функции BmnGemStripModule::AddRealPointFull. Параллелизм на уровне процессов с MPI мог бы избежать конфликтов данных за счет создания независимых экземпляров в каждом процессе, однако это привело бы к значительному увеличению потребления памяти, возможным трудностям с объединением данных, а также требовало бы существенной переработки кода. Поэтому в качестве окончательной стратегии оптимизации была выбрана схема параллелизма данных для функции BmnGemStripModule::AddRealPointFull, как наиболее целенаправленная и умеренная по сложности реализации.

3.2. Реализация схемы параллелизма данных

Для оптимизации функции BmnGemStripModule::AddRealPointFull с использованием параллелизма данных были приняты следующие ключевые меры:

1. Использование независимых хранилищ cluster и генераторов случайных чисел в каждом потоке для избежания конкуренции.
2. Динамическое планирование для адаптации к несбалансированной нагрузке.

3. Защита критической секции при объединении результатов.

Реализующий код представлен ниже:

```
// Strip cluster in each strip layer
Int_t n_strip_layers = StripLayers.size();
vector<StripCluster> cluster_layers(n_strip_layers, StripCluster());

// Начало параллельной секции -----
#pragma omp parallel
{
    // Каждый поток имеет свое собственное хранилище cluster
    vector<StripCluster> local_clusters(n_strip_layers, StripCluster());
    // Получение ID потока,
    // создание независимого генератора случайных чисел для каждого потока
    int thread_id = omp_get_thread_num();
    TRandom3 local_random(gRandom->GetSeed() + thread_id);

    // Electron avalanche producing for each collision point at the track
    #pragma omp for schedule(dynamic, 10) nowait
    for(Int_t icoll = 0; icoll < collision_points.size(); ++icoll) {
        /* Сложные вычисления и вложенные циклы, где clasters
           и генераторы случайных чисел должны использовать
           экземпляры, созданные для каждого потока независимо. */
    }
    // Объединение результатов
    #pragma omp critical
    {
        for(Int_t ilayer = 0; ilayer < n_strip_layers; ++ilayer) {
            for(int i = 0; i < local_clusters[ilayer].GetClusterSize(); ++i) {
                Int_t strip_num = local_clusters[ilayer].Strips[i];
                Double_t strip_signal = local_clusters[ilayer].Signals[i];
                cluster_layers[ilayer].AddStrip(strip_num, strip_signal);
            }
        }
    }
}
// Конец параллельной секции -----
```

3.3. Реализация схемы параллелизма данных

Для системной оценки эффекта параллельной оптимизации в данном исследовании была разработана схема сравнительного эксперимента: в условиях фиксированного масштаба событий (1000 событий) тестировалась производительность трех генераторов частиц: ION, PART и BOX. Экспериментальная среда использовала многоядерную процессорную архитектуру, а анализ масштабируемости параллелизации проводился путем контроля количества потоков.

3.3.1. Сравнение параллельной производительности различных генераторов

При 1000 событиях и 8 потоках были протестированы генераторы событий ION, PART и BOX. Записывалось время выполнения в однопоточном режиме и в параллельном режиме, а также рассчитывался коэффициент ускорения. Результаты представлены в Таблице 1:

Таблица 1: Сравнение производительности трех генераторов в конфигурации с 8 потоками

Генератор	Однопоточное время(s)	Параллельное время(s)	Ускорение
ION	4953	1077	4.6
PART	768	533	1.4
BOX	119	108	1.1

Результаты эксперимента показывают, что эффект параллельной оптимизации существенно зависит от генератора. Генератор ION демонстрирует наилучшую параллельную производительность с коэффициентом ускорения 4.6, что приближается к 57.5% эффективности от идеального линейного ускорения (теоретический максимум равен 8). Генератор PART показывает умеренное ускорение (коэффициент 1.44), в то время как генератор BOX из-за изначально низкой вычислительной нагрузки получает ограниченную выгоду от распараллеливания (коэффициент 1.1).

3.3.2. Анализ масштабируемости по потокам

Для дальнейшей оценки масштабируемости параллельной схемы был проведен тест для генератора ION при 1000 событиях и количестве потоков от 4 до 12. Было записано время выполнения и коэффициент ускорения. Результаты представлены в Таблице 2:

Таблица 2: Сравнение производительности трех генераторов в конфигурации с 8 потоками

потоков	Параллельное время(с)	Ускорение	эффективность
1	4953	—	—
4	1601	3.1	77.5%
6	1223	4.0	66.7%
8	1077	4.6	57.5%
10	1073	4.6	46.0%
12	1078	4.6	38.3%

Анализ данных показывает, что параллельная производительность генератора ION с увеличением количества потоков демонстрирует характерную кривую насыщения. На интервале от 1 до 8 потоков коэффициент ускорения приблизительно линейно растет, а эффективность параллелизма сохраняется на уровне выше 70% (эффективность 77.5% для 4 потоков и 57.5% для 8 потоков). Когда количество потоков превышает 8, производительность выходит на плато, а коэффициент ускорения стабилизируется на уровне около 4.6. Это явление соответствует прогнозу закона Амдала, указывая на наличие в системе последовательного "узкого места" приблизительно в 20%[5].

Насыщение масштабируемости по потокам в основном объясняется следующими факторами: Однако тестирование выявило следующие проблемы данной схемы:

1. Ограничение пропускной способности памяти: Конкуренция за пропускную способность подсистемы памяти при одновременном доступе множества потоков.
2. Накладные расходы когерентности кэша: Дополнительная задержка, вызванная синхронизацией данных между ядрами.

3. Дисбаланс нагрузки: Сложность полного выравнивания вычислительной нагрузки даже при динамическом распределении задач.
4. Сериализация в критической секции: Последовательные накладные расходы, вносимые операцией объединения результатов.

Заключение

Итоги работы

В данном исследовании был системно проанализирован профиль производительности модуля генерации событий программного обеспечения BMNROOT, выявлены ключевые узкие места и успешно реализована оптимизация на основе параллелизма данных. Комбинированное использование Valgrind и perf обеспечило целенаправленность и эффективность оптимизации. Результаты экспериментов показывают, что данная схема параллельной оптимизации обеспечила значительное повышение производительности для генератора ION, подтвердив эффективность стратегии распараллеливания данных для сценариев вычислений интенсивного моделирования частиц. Эффект оптимизации положительно коррелирует с вычислительной сложностью генератора, и это открытие предоставляет важное руководство для последующей целенаправленной оптимизации. Примечательно, что текущая схема имеет потенциал для улучшения в области масштабируемости по потокам.

Направления будущей работы

1. Гибридная параллельная архитектура: Углубленное исследование структуры кода для реализации гибридной модели, сочетающей параллелизм на уровне событий и параллелизм данных, с использованием преимуществ как OpenMP, так и MPI.
2. Интеграция параллельных возможностей Geant4: Полноценное использование встроенных параллельных возможностей Geant4 в составе FairRoot, исследование стратегий их интеграции с существующими оптимизированными решениями[9].
3. Передовые технологии оптимизации: Исследование таких современных методов, как векторное преобразование (SIMD) и параллелизация на [CUDA](#), для дальнейшего повышения вычислительной эффективности[10]. В частности, изучение алгоритмов переноса

частич с использованием GPU для решения задач моделирования еще большего масштаба.

4. Оптимизация алгоритмов: Глубокое исследование и анализ алгоритмов, оптимизация структур данных и других компонентов, способных влиять на производительность.

Приложение 1 Автоматизированный тестовый скрипт

```
#!/bin/bash

# Установка базовых переменных
OUTPUT_DIR="perf_results_1000"
FLAMEGRAPH_DIR="/home/vodka/FlameGraph"
GENERATOR="ION"
# Создание выходной директории
mkdir -p "$OUTPUT_DIR"

# Создание директории для текущего теста
TEST_DIR="$OUTPUT_DIR/${GENERATOR}"
mkdir -p "$TEST_DIR"

echo "Начало тестирования генератора: $GENERATOR"
# Переход в директорию теста
cd "$TEST_DIR"

echo "Запуск perf record..."
perf record -g --call-graph=fp --freq=997 -o "${GENERATOR}_perf.data" \
    root -b -q "/home/vodka/bmnroot/macro/run/run_sim_bmn.C" \
    ("DCMSMM_XeCsI_3.9AGeV_mb_10k_142.r12", \
    "$VMCWORKDIR/macro/run/bmnsim.root", 0, 1000, ION)"

# Проверка успешного создания perf.data
if [ ! -f "${GENERATOR}_perf.data" ]; then
    echo "Ошибка: ${GENERATOR}_perf.data не был создан, пропуск этого теста"
    cd - > /dev/null
    exit 1
fi

# Запуск perf stat для получения общей статистики
echo "Запуск perf stat..."
perf stat -e cycles,instructions,cache-misses,branch-misses,cache-references \
    -o "${GENERATOR}_perf_stat.txt" \
    root -b -q "/home/vodka/bmnroot/macro/run/run_sim_bmn.C" \
    ("DCMSMM_XeCsI_3.9AGeV_mb_10k_142.r12", \
    "$VMCWORKDIR/macro/run/bmnsim.root", 0, 1000, ION)"
```

```

# Генерация текстового отчета
echo "Генерация текстового отчета..."
perf report --stdio -i "${GENERATOR}_perf.data" \
> "${GENERATOR}_perf_report.txt"

# Генерация отчета с графиком вызовов
echo "Генерация отчета с графиком вызовов..."
perf report -g fractal,0.5,caller --stdio -i "${GENERATOR}_perf.data" \
> "${GENERATOR}_perf_callgraph.txt"

echo "Генерация флеймграфа..."
perf script -i "${GENERATOR}_perf.data"
stackcollapse-perf.pl > out.perf-folded
flamegraph.pl out.perf-folded > "${GENERATOR}_flamegraph.svg"

# Очистка промежуточных файлов
rm -f out.perf-folded

# Возврат в исходную директорию
cd - > /dev/null

echo "Завершение теста: $GENERATOR"
echo "Результаты сохранены в: $TEST_DIR"
echo "-----"

```

Список литературы

- [1] Software Development for the BM@N Experiment at NICA / Konstantin Gertsenberger, Sergey Merts, Ilnur Gabdrakhmanov et al. // EPJ Web of Conferences. — Vol. 226. — Dubna, Russia : EDP Sciences, 2020. — P. 03008. — Mathematical Modeling and Computational Physics 2019. URL: <https://doi.org/10.1051/epjconf/202022603008>.
- [2] Multithreaded Event Simulation in the BmnRoot Package / M. M. Stepanova, A. V. Driuk, S. P. Mertz et al. // Proceedings of the 9th International Conference "Distributed Computing and Grid Technologies in Science and Education" (GRID'2021). — Dubna, Russia : Joint Institute for Nuclear Research, 2021. — July 5-9. — P. 58–63. — URL: <http://nica.jinr.ru>.
- [3] Brun Rene, Rademakers Fons. ROOT – An object oriented data analysis framework // Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. — 1997. — Vol. 389, no. 1–2. — P. 81–86. — URL: <https://www.sciencedirect.com/science/article/abs/pii/S016890029700048X>.
- [4] Dynamic binary analysis and instrumentation : Rep. : UCAM-CL-TR-606 / University of Cambridge, Computer Laboratory ; Executor: Nicholas Nethercote : 2004. — . — URL: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf>.
- [5] Barbara Chapman Gabriele Jost, van der Pas Ruud. Portable Shared Memory Parallel Programming. — The MIT Press, 2007.
- [6] MPI: A Message-Passing Interface Standard : Rep. ; Executor: Message P Forum. — USA : 1994.
- [7] L. Dagum R. Menon. OpenMP: an industry standard API for shared-memory programming // IEEE Computational Science and Engineer-

- ing. — 1998. — Vol. 5, no. 1. — P. 46–55. — URL: <https://ieeexplore.ieee.org/abstract/document/660313>.
- [8] Brendan Gregg Netflix. The Flame Graph: This visualization of software execution is a new necessity for performance profiling and debugging. // Queue. — 2016. — Vol. 14, no. 2. — URL: <https://queue.acm.org/detail.cfm?id=2927301>.
- [9] J. Allison K. Amako J. Apostolakis H. Araujo P. Arce Dubois M. Asai. Geant4 developments and applications // IEEE Computational Science and Engineering. — 2006. — Vol. 53, no. 1. — P. 270–278. — URL: <https://ieeexplore.ieee.org/abstract/document/1610988>.
- [10] Corporation NVIDIA. CUDA C++ Programming Guide. — URL: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/> (дата обращения: 27 сентября 2025 г.).