УДК 004.45

СРАВНЕНИЕ МНОГОПОТОЧНЫХ МОДЕЛЕЙ В C++, GO И RUST В ЗАДАЧАХ ПАРАЛЛЕЛЬНОЙ ОБРАБОТКИ ДАННЫХ

Е. А. Азаров, А. Н. Марков, С. Н. Барсукевич

Белорусский государственный университет информатики и радиоэлектроники, Минск, Беларусь, georga399@gmail.com

Данное исследование сравнивает эффективность применения многопоточных моделей в C++, Go и Rust в задачах параллельной обработки данных. Анализируются производительность, безопасность и удобство разработки. Наибольшая производительность достигается в C++, Go продемонстрировал лучшую масштабируемость, а Rust показал лучшую безопасность. Результаты подкреплены бенчмарками производительности.

Ключевые слова: многопоточность; C++; Go; Rust; параллельная обработка; производительность; безопасность.

COMPARATIVE ANALYSIS OF MULTITREADING MODELS IN C++, GO AND RUST FOR PARALLEL DATA PROCESSING TASKS

E. A. Azarov, A. N. Markov, S. N. Barsukevich

Belarusian State University of Informatics and Radioelectronics, Minsk, Belarus, georga399@gmail.com

This study compares the effectiveness of using multithreaded models in C++, Go and Rust in parallel data processing tasks. It analyzes performance, safety, and development convenience. The highest performance is achieved in C++, Go demonstrated the best scalability and Rust showed the best safety. The results are supported by performance benchmarks.

Keywords: multithreading; C++; Go; Rust; parallel processing; performance; safety.

1. Введение

Многопоточность является ключевым инструментом для параллельной обработки данных, позволяя эффективно использовать многоядерные процессоры в задачах с большими объемами данных. Такие языки программирования, как *C++*, *Go* и *Rust*, предлагают различные механизмы реализации многопоточности, отличающиеся по производительности, безопасности и удобству разработки. Данное исследование анализирует их

возможности в контексте задач обработки данных, таких как вычисления и операции ввода-вывода, и предоставляет результаты бенчмарков для оценки производительности, а также определяет оптимальные сценарии использования [1].

2. Многопоточность в C++, Go, Rust

C++ предоставляет низкоуровневые механизмы многопоточности через стандартную библиотеку, начиная с *C++11*. Основные инструменты включают потоки *std::thread*, *std::atomic*. Эти примитивы обеспечивают точный контроль над потоками, но требуют внимательного управления для предотвращения гонок данных и взаимных блокировок. Внешние библиотеки, такие как *OpenMP* и *Intel TBB*, упрощают реализацию параллельных алгоритмов, особенно для задач с неоднородной нагрузкой [2].

Go использует модель конкурентности, основанную на горутинах – легковесных потоках, управляемых средой выполнения. Планировщик *Go* эффективно распределяет горутины по системным потокам, что делает его подходящим для задач с высокой конкуренцией, таких как обработка сетевых запросов. Каналы *chan* обеспечивают безопасную передачу данных между горутинами, но сборка мусора может вызывать паузы в вычислительно интенсивных задачах [3].

Rust применяет уникальную систему владения и заимствования, которая предотвращает гонки данных на этапе компиляции. Типы *Send* и *Sync* обеспечивают безопасную передачу данных между потоками. Библиотеки, такие как *Rayon* и *Tokio*, расширяют возможности для параллельных и асинхронных вычислений, делая *Rust* подходящим для критически важных систем, где безопасность имеет приоритет [4].

3. Разработка тестовых сценариев

Для сравнительного анализа были составлены следующие тестовые сценарии: параллельная обработка независимых задач (*Coro-Prime-Sieve*), синхронизированный доступ к общему ресурсу, конвейерная обработка с передачей данных.

Тесты проводились в операционной системе *Linux Ubuntu 22.04* на вычислительной машине с процессором *Intel Core i5-1135G7* (8 ядер) с использованием компиляторов *g++ 12.2 (-O3)*, *Go 1.21* и *Rust 1.68*.

4. Сравнительный анализ параллельной обработки независимых задач

Задача *Coro-Prime-Sieve* разделяет вычисление простых чисел на подзадачи. *C++* использует *std::thread*, *Go* – горутины, *Rust – std::thread*. *C++* показывает лучшее время выполнения (450 мс) для конкретной

конфигурации (8 потоков), однако *Go* показал лучшую масштабируемость, при увеличении числа потоков, благодаря легковесным горутинам.

Сравнение производительности в задаче вычисления простых чисел до заданного предела (вычисления простых чисел меньших 4000) с использованием решета Эратосфена [5], разделенного на независимые подзадачи, представлено на рис. 1.

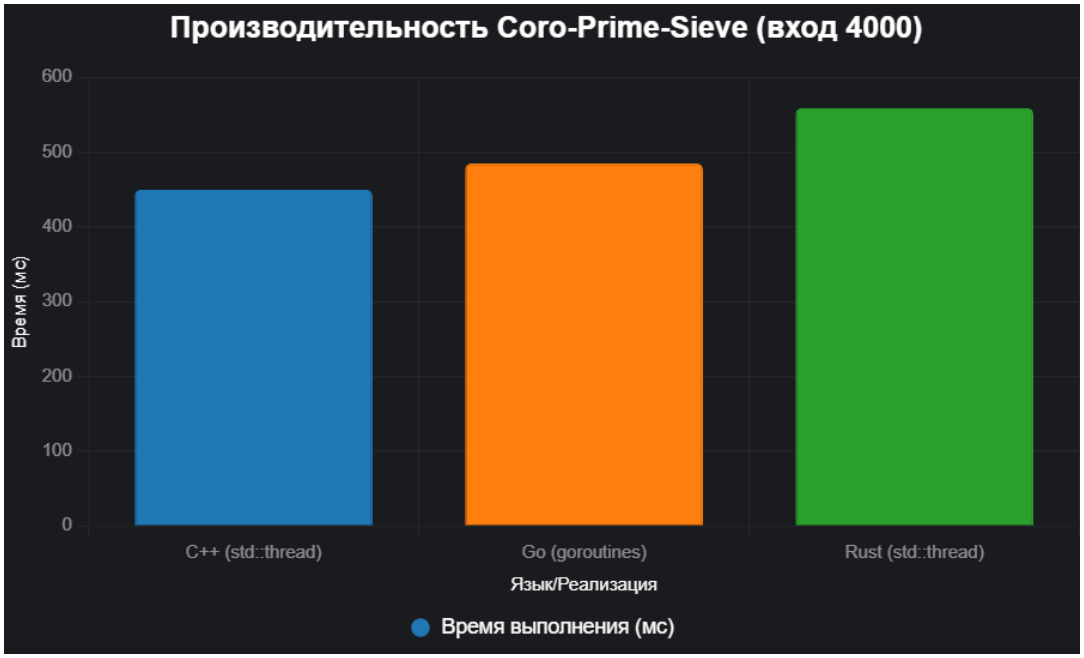

Рис. 1. График производительности в задаче *Coro-Prime-Sieve*

C++ использует явное создание потоков (*std::thread*) со статическим распределением диапазонов чисел между потоками. Каждый поток обрабатывает заранее выделенный сегмент решета Эратосфена независимо. Синхронизация между потоками отсутствует, так как подзадачи полностью независимы.

Go применяет динамическое распределение подзадач (проверка чисел на простоту) между горутинами через пул воркеров (набор горутин-обработчиков, каждая из которых последовательно выполняет задачи из общей очереди) или каналы. Горутины, завершившие свою задачу, немедленно получают новую из общего канала. Лучшая масштабируемость обеспечивается за счёт низкой стоимости создания горутин (0,2–0,5 мкс), эффективной работой планировщика *Go* (*M:N* модель), динамически балансирующего нагрузку на системные потоки, отсутствием перегрузки при создании большого количества горутин.

Rust реализует подход, аналогичный *C++,* со статическим распределением диапазонов по потокам (*std::thread*). Ключевая особенность – гарантии

безопасности на этапе компиляции. Система владения *Rust* проверяет отсутствие гонок данных при передаче сегментов данных в потоки (тип *Send*), исключая необходимость в *runtime*-проверках или неявных копиях.

5. Сравнительный анализ задачи с синхронизированным доступом к общему ресурсу

Задача заключается в инкременте общего счетчика (10^6 итераций) несколькими потоками с использованием средств синхронизации [6].

График сравнения производительности в задаче с синхронизированным доступом изображен на рис. 2.

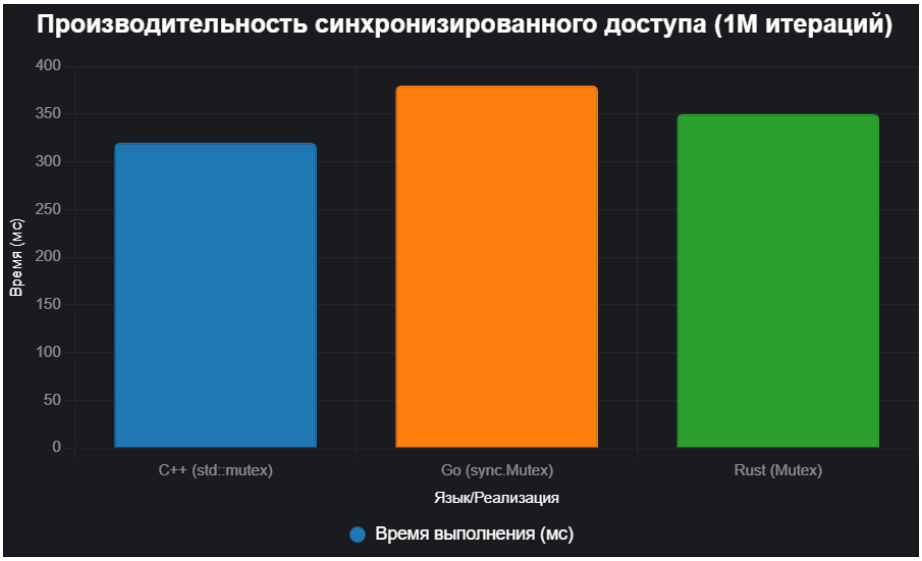

Рис. 2. Сравнение производительности в задаче с синхронизированным доступом

C++ использует *std::mutex* с явным блокированием (*lock()/unlock()*), *Go – sync.Mutex*, *Rust – Mutex*. *C++* показал лучшее время (320 мс) благодаря низким накладным расходам. В *Rust* потенциальные ситуации, когда два или более потока бесконечно ожидают освобождения ресурсов, заблокированных друг другом (*deadlock'u*), обнаруживаются только на этапе выполнения (как и в *C++/Go*), но гарантируется отсутствие гонок данных (*data races*) на этапе компиляции.

6. Сравнительный анализ задачи конвейерной обработки с передачей данных

Задача заключается в реализации конвейера, который обрабатывает 10^3 элементов в три этапа (вычисление квадрата, фильтрация, суммирование) [7]. *Go* показал лучшее время (260 мс) благодаря каналам.

График сравнения производительности конвейерной обработки показан на рис. 3.

Рис. 3. Сравнение производительности конвейерной обработки

Программа тестового сценария на C++ реализуется с использованием очередей (*std::queue* или *std::deque*) для хранения элементов между этапами, мьютексов (*std::mutex*) и условных переменных (*std::condition_variable*) для синхронизации доступа к очередям и уведомления потоков о появлении новых данных или свободного места, явных потоков (*std::thread*) для каждого этапа конвейера.

Go использует каналы (*chan*) как основной механизм передачи данных между горутинами-этапами. Лучшая производительность (260 мс) достигается за счет семантики *CSP* (*Communicating Sequential Processes*): каналов – первоклассной конструкции языка, глубоко интегрированной в планировщик и легковесности горутин. Передача данных через канал часто эффективнее, чем через разделяемую память с мьютексом.

Rust использует каналы *std::sync::mpsc*, а именно *Multi-Producer*, *Single-Consumer* каналы (похожи на каналы *Go*, но с одним получателем). Передача данных обычно требует перемещения (*move* семантика). Гарантии владения обеспечивают безопасную передачу. Производительность высока (близка к *Go*) благодаря эффективной реализации.

7. Сравнение накладных расходов

Накладные расходы при реализации многопоточности представляют собой существенный фактор, влияющий на общую производительность параллельных систем. Величина этих расходов детерминируется как

архитектурными особенностями языка программирования, так и выбранной моделью параллелизма.

Ниже (табл. 1) приведено сравнение накладных расходов многопоточных механизмов в *C++, Go*, *Rust* [7].

Таблица 1

Сравнение накладных расходов многопоточных механизмов в *C++*, *Go*, *Rust*

Операция	*C++*	*Go*	*Rust*
Создание потока	10–100 мкс	0,2–0,5 мкс	10–100 мкс
Переключение	1–3 мкс	0,1–0,3 мкс	1–3 мкс
Блокировка мьютекса	20–50 нс	30–70 нс	40–80 нс
Атомарная операция	10–30 нс	–	15–40 нс

Примечание. Для языка *Go* показатели атомарных операций не включены ввиду преобладания канальной коммуникации как основного метода синхронизации в идиоматических паттернах. Замеры проводились для стандартных примитивов синхронизации (*std::mutex*, *sync.Mutex*, *std::sync::Mutex*).

По данным таблицы можно сделать вывод, что *Go* лидирует в переключении контекста. *C++* минимален по накладным расходам, *Rust* обеспечивает безопасность с умеренными накладными расходами.

8. Заключение

Сравнительное тестирование показало, что *C++*, *Go* и *Rust* демонстрируют разную производительность и накладные расходы в параллельных вычислениях, что определяет их применимость для различных классов задач. *C++* обеспечивает максимальную производительность в задачах с интенсивными вычислениями и синхронизацией, что делает его предпочтительным для высокопроизводительных систем, таких как научные вычисления или игровые движки. *Go* упрощает разработку высококонкурентных приложений, таких как веб-серверы или системы обработки потоковых данных, благодаря легковесным горутинам и каналам. *Rust* гарантирует безопасность памяти, что критически важно для надежных систем, например, в телекоммуникациях или блокчейн-приложениях.

Результаты бенчмарков в проведенном исследовании помогут разработчикам выбрать язык в зависимости от требований проекта.

Библиографические ссылки

1. Benchmarking low-level I/O: C, C++, Rust, Golang, Java, Python. URL: https://medium.com/star-gazers/benchmarking-low-level-i-o-c-c-rust-golang-java-python-9a0d505f85f7 (date of access: 05.06.2025).

2. C++ Reference: Concurrency support library. URL: https://en.cppreference.com/w/cpp/thread (date of access: 05.06.2025).

3. The Go Programming Language. URL: https://go.dev/doc/ (date of access: 05.06.2025).

4. The Rust Programming Language. URL: https://doc.rust-lang.org/book/ (date of access: 05.06.2025).

5. Prime number generator using Sieve of Eratosthenes. URL: https://felspar.com/coro/examples/primes-1-generator.cpp (date of access: 05.06.2025).

6. *Herlihy M., Shavit N.* The Art of Multiprocessor Programming. Morgan Kaufmann, 2012.

7. *Mattson T. G., Sanders B. A., Massingill B. L.* Patterns for Parallel Programming. Addison-Wesley, 2004.