

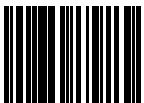
Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
GRADUATION THESIS

Разработка алгоритма быстрого распределения работы для компокуемых
планировщиков задач

Обучающийся / Student Воркожоков Денис Вадимович
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования
Группа/Group М34391
Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика
Образовательная программа / Educational program Информатика и программирование 2019
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Бакалавр
Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")
Консультант не из ИТМО / Third-party consultant Малахов Антон Александрович, ООО "Техкомпания Хуавей", Ведущий инженер ключевых проектов

Обучающийся/Student

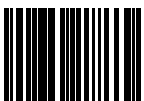
Документ подписан	
Воркожоков Денис Вадимович	
17.05.2023	

(эл. подпись/ signature)

Воркожоков
Денис
Вадимович

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Аксенов Виталий Евгеньевич	
17.05.2023	

(эл. подпись/ signature)

Аксенов
Виталий
Евгеньевич

(Фамилия И.О./ name
and surname)

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /
OBJECTIVES FOR A GRADUATION THESIS**

Обучающийся / Student Воркожоков Денис Вадимович
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования
Группа/Group М34391
Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика
Образовательная программа / Educational program Информатика и программирование 2019
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Бакалавр
Тема ВКР/ Thesis topic Разработка алгоритма быстрого распределения работы для компокуемых планировщиков задач
Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")
Консультант не из ИТМО / Third-party consultant Малахов Антон Александрович, ООО "Техкомпания Хуавей", Ведущий инженер ключевых проектов

Основные вопросы, подлежащие разработке / Key issues to be analyzed

Требуется разработать алгоритм распределения задач параллельного алгоритма для компокуемого планировщика задач, производительный вне зависимости от паттерна нагрузки, в том числе при наличии вложенного параллелизма в программе. В рамках этого необходимо:

1. Разработать бенчмарки с различным характером нагрузки, использующие алгоритм `parallel_for`
2. Сравнить современные решения (OneTBB и OpenMP) в различных конфигурациях на разработанных бенчмарках
3. Разработать прототип алгоритма распределения задач, определить проблемные места, предложить их решения
4. Провести сравнение разработанного алгоритма с аналогами

Форма представления материалов ВКР / Format(s) of thesis materials:

Программный код, презентация, пояснительная записка

Дата выдачи задания / Assignment issued on: 01.04.2023

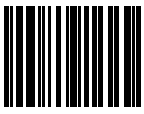
Срок представления готовой ВКР / Deadline for final edition of the thesis 15.05.2023

Характеристика темы ВКР / Description of thesis subject (topic)

Тема в области прикладных исследований / Subject of applied research: да / yes

СОГЛАСОВАНО / AGREED:

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Аксенов Виталий Евгеньевич	
09.05.2023	

Аксенов
Виталий
Евгеньевич

(эл. подпись)

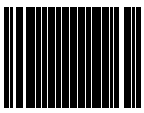
Задание принял к
исполнению/ Objectives
assumed BY

Документ подписан	
Воркожиков Денис Вадимович	
09.05.2023	

Воркожиков
Денис
Вадимович

(эл. подпись)

Руководитель ОП/ Head
of educational program

Документ подписан	
Станкевич Андрей Сергеевич	
22.05.2023	

Станкевич
Андрей
Сергеевич

(эл. подпись)

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
SUMMARY OF A GRADUATION THESIS**

Обучающийся / Student Воркожоков Денис Вадимович

Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования

Группа/Group М34391

Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика

Образовательная программа / Educational program Информатика и программирование 2019

Язык реализации ОП / Language of the educational program Русский

Статус ОП / Status of educational program

Квалификация/ Degree level Бакалавр

Тема ВКР/ Thesis topic Разработка алгоритма быстрого распределения работы для компонуемых планировщиков задач

Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

Консультант не из ИТМО / Third-party consultant Малахов Антон Александрович, ООО "Техкомпания Хуавей", Ведущий инженер ключевых проектов

**ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
DESCRIPTION OF THE GRADUATION THESIS**

Цель исследования / Research goal

Разработка алгоритма распределения задач параллельного алгоритма для компонуемого планировщика задач, производительного вне зависимости от паттерна нагрузки, в том числе при наличии вложенного параллелизма в программе

Задачи, решаемые в ВКР / Research tasks

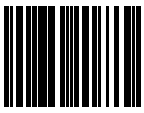
Разработка прототипа алгоритма распределения задач, определение проблемных мест и предложения по их решению. Разработка бенчмарков с различными паттернами нагрузки, использующих `parallel_for`. Сравнение получившегося алгоритма с известными реализациями параллельного исполнения (`OneTBB` и `OpenMP`) на разработанных бенчмарках.

Краткая характеристика полученных результатов / Short summary of results/findings

Был разработан алгоритм распределения работы, порожденной алгоритмом `parallel_for`, для компонуемого планировщика задач. Для оценки производительности алгоритма был разработан комплекс бенчмарков с различным характером нагрузки, позволяющих сравнить получившееся решение с аналогами. Было выявлено, что алгоритм уступает на сбалансированной нагрузке только статическому распределению из `OpenMP`, а на

несбалансированной нагрузке имеет производительность лучше аналогов, участвовавших в сравнении.

Обучающийся/Student

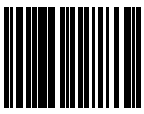
Документ подписан	
Воркожоков Денис Вадимович	
17.05.2023	

(эл. подпись/ signature)

Воркожоков
Денис
Вадимович

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Аксенов Виталий Евгеньевич	
17.05.2023	

(эл. подпись/ signature)

Аксенов
Виталий
Евгеньевич

(Фамилия И.О./ name
and surname)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
1. Обзор предметной области	7
1.1. Используемые термины и понятия	7
1.2. Описание аналогов	9
1.2.1. OneTBV	9
1.2.2. OpenMP	10
1.3. Проблемы существующих решений	11
1.4. Постановка задачи	12
Выводы по главе 1	14
2. Предлагаемое решение	15
2.1. Решаемые подзадачи	15
2.2. Начальное распределение задач	15
2.3. Интеграция с планировщиком задач	19
2.4. Адаптивный размер балансировочных задач	22
2.5. Тестирование	23
Выводы по главе 2	23
3. Бенчмарки и сравнение с аналогами	24
3.1. Описание бенчмарков	24
3.1.1. Умножение разреженной матрицы на вектор (SPMV)	24
3.1.2. Свёртка (reduce)	26
3.1.3. Сканирование (scan)	26
3.1.4. Вложенный параллелизм	27
3.1.5. Время распределения задач	27
3.2. Конфигурация системы	28
3.3. Результаты бенчмарков	29
3.3.1. SPMV	30
3.3.2. Reduce	35
3.3.3. Scan	35
3.3.4. Вложенный параллелизм	37
3.3.5. Время распределения задач	39
Выводы по главе 3	40
ЗАКЛЮЧЕНИЕ	42
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	43

ВВЕДЕНИЕ

С ростом объема данных возникает необходимость в их быстрой обработке. В прошлом производительность одного ядра процессоров росла экспоненциально, однако, к началу 2000-х годов этот рост замедлился и индустрии пришлось искать другие способы увеличивать производительность [2]. Один из таких способов — горизонтальное масштабирование, которое достигается за счет использования параллельных алгоритмов, использующих несколько ядер или даже процессоров одновременно. Такие алгоритмы позволяют значительно увеличить скорость обработки данных, в лучшем случае — повысить её кратно количеству ядер процессоров.

Для достижения максимального ускорения алгоритма, необходимо эффективно распределять работу между ядрами процессоров, избегая простоя ресурсов. Например, для базового алгоритма `parallel_for` необходимо учитывать, что различные его итерации могут занимать разное время и нагрузка не всегда будет равномерно распределена, поэтому подход статического равномерного деления работы не всегда является оптимальным.

Однако более сложные алгоритмы распределения задач, адаптирующиеся под текущую нагрузку, ожидаемо имеют накладные расходы по сравнению с наивным подходом. Это, в частности, заметно при сравнениях современного планировщика задач OneTBB и статического подхода из OpenMP. Связано это с тем, что в основе планировщиков задач лежит подход `work-stealing` [1], позволяющий простаивающим потокам «воровать» работу у других в процессе выполнения алгоритма, но приводящий к конфликтам за общие ресурсы между потоками.

В то же время, классический OpenMP при условии равномерной нагрузки имеет более высокую производительность за счёт уменьшения накладных расходов на распределение задач, но он практически не поддерживает вложенный параллелизм и балансировку нагрузки, что не позволяет использовать его в качестве универсального решения.

Таким образом, существует потребность в алгоритме, который будет совмещать разные парадигмы распределения работы и иметь одинаково высокую производительность вне зависимости от паттерна нагрузки. Вместе с тем, чтобы алгоритм был универсально применяем, он должен поддерживать композиемость — к примеру, внутри итерации параллельного региона может быть ещё

один вызов `parallel_for`, который будет выполняться теми же потоками и тем самым порождать вложенный параллелизм.

Цель этой работы — разработать алгоритм распределения задач, интегрированный с компонуемым планировщиком задач, производительный вне зависимости от паттерна нагрузки, в том числе при наличии вложенного параллелизма в программе. Несмотря на то, что параллельных алгоритмов существует достаточно много, в работе будет рассматриваться распределение работы, порождаемой вызовом `parallel_for`, как одного из основных примитивов параллелизма.

Для достижения этой цели в рамках работы были решены следующие задачи:

- Разработка прототипа алгоритма распределения задач, определение проблемных мест и предложения по их решению.
- Разработка бенчмарков с различными паттернами нагрузки, использующих `parallel_for`.
- Сравнение получившегося алгоритма с известными реализациями параллельного исполнения (OneTBB и OpenMP) на разработанных бенчмарках.

В первой главе работы представлен обзор предметной области, анализ существующих подходов и постановка задачи. Во второй главе описан предлагаемый алгоритм и нюансы реализации. В третьей главе описаны реализованные бенчмарки и приведено сравнение производительности между современными решениями и предложенным алгоритмом.

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

В этой главе определены используемые в работе термины, подробно сформулирована решаемая задача, а также приведен обзор аналогов, решающих такую же или похожую задачу.

1.1. Используемые термины и понятия

В рамках модели параллельных вычислений мы будем оперировать понятиями потока и ядра процессора. Ядро процессора — самостоятельный физический вычислительный блок процессора, выполняющий заданную последовательность инструкций. Современные процессоры, используемые в серверах, могут насчитывать порядка 128 ядер. Поток — это логическая сущность внутри программы, имеющая свой стек, набор регистров и последовательность выполняемых инструкций. Планировщик операционной системы для каждого ядра процессора определяет, какой поток в данный момент времени будет на нём выполняться, переключая контекст выполнения между ними.

В этой работе мы фокусируемся только на работе с потоками. На практике в программах, большая часть работы которых это выполнение параллельных алгоритмов, прибегают к привязке потоков к ядрам. Например, на Linux это достигается выставлением каждому потоку соответствующей битовой маски, задающей набор ядер процессора, на которых он может запускаться, с помощью системного вызова `sched_setaffinity` [11]. Несмотря на то, что однозначное соответствие ядер потокам не даёт планировщику более оптимально распределять процессорное время между потоками разных процессов, это более чем оправданно, если на сервере параллельно не работают другие процессы, требовательные к вычислительным ресурсам. Такой механизм позволяет избежать миграции потоков между ядрами, которая приводит к дорогостоящей смене контекстов и может повлиять на стабильность замеров производительности [13].

Также регулярно упоминается пул потоков — это механизм управления множеством потоков, позволяющий переиспользовать потоки для выполнения задач вместо пересоздания новых. Обычно пул потоков реализован в виде простого интерфейса, приведённого в листинге 1, поэтому альтернативно пул потоков можно называть и планировщиком задач — он принимает задачу и планирует её выполнение на одном из потоков. Задачей, в свою очередь, можно

просто называть функцией без аргументов, которую необходимо выполнить в любом из потоков.

Листинг 1 – Интерфейс пула потоков

```
class ThreadPool {  
    ThreadPool(size_t threadNum);  
    void Submit(F func);  
};
```

Обычно планировщик задач реализован в виде какой-то очереди, а созданные потоки в цикле вытаскивают из очереди задачу и выполняют её. Для уменьшения конфликтов потоков за общие ресурсы (англ. contention) используют несколько очередей — например, по одной локальной очереди на поток. В таком случае, задача попадает в случайную из очередей, а потоки забирают задачу только из назначенной ему локальной очереди. Чтобы повысить утилизацию ресурсов потоков в ситуациях, когда у одного из потоков задачи кончились, а другого нет, применяется механизм захвата работы (англ. work-stealing) — поток, у которого кончились задачи, итерируется по очередям других потоков и пробует забрать задачу оттуда.

В работе мы будем фокусироваться на распределении работы, порожденной параллельными алгоритмами, поэтому стоит дать следующие определения. Параллельным алгоритмом будем называть алгоритм, исполнение которого разбивается на вычисление независимых подзадач, которое можно делать параллельно. Обобщённо такие подзадачи будем называть нагрузкой или работой, которую порождает этот алгоритм. Нагрузка может быть сбалансированной или несбалансированной. Нагрузка считается сбалансированной, если подзадачи, порождаемые алгоритмом, в среднем занимают одинаковое время на выполнение при идентичных условиях, в противном случае, мы считаем, что нагрузка является несбалансированной.

Для оценки производительности реализованы бенчмарки — специальные программы, целью которых является измерение производительности программ путём замера времени работы их на одних и тех же или сгенерированных похожим образом данных. Для устойчивости результата бенчмарков такое измерение проводится многократно и результат усредняется [7].

1.2. Описание аналогов

На данный момент в индустрии распространены две библиотеки для параллельных вычислений на C++, которые имплементируют два различных подхода — Intel OneTBB и OpenMP, с ними в этой работе и будут проводиться сравнения. Несмотря на то, что мы ограничиваемся библиотеками на одном языке (в том числе для удобства тестирования гипотез), фактически для высоко-производительных вычислений у C++ сейчас нет аналогов, поэтому рассматривать библиотеки и фреймворки на других языках излишне.

1.2.1. OneTBB

OneTBB — это библиотека, предоставляющая интерфейсы различного уровня абстракций, от низкоуровневого интерфейса задач до параллельных алгоритмов таких как `parallel_for`, `parallel_reduce` и другие. Она подключается как обычная C++ библиотека, поэтому не требует какой-либо специфичной поддержки от компилятора.

Стоит отметить, что весь дизайн OneTBB ориентирован на порождение и выполнении задач (Task-Based [12]) — благодаря этому достигается высокая производительность за счёт возможности балансировать нагрузку перемещением задач, а также появляется возможность оптимально реализовать различные сценарии композиции параллельности, такие как вложенность и конкурентность [14]. Задачи, порождаемые программой, выполняются потоками из глобального пула потоков, который инициализируется на старте программы. В такой модели параллельные алгоритмы, в частности `parallel_for`, должны порождать задачи, распределять их по потокам и дожидаться завершения. Для `parallel_for` это можно выразить совсем естественным образом — каждая итерация представляется в виде одной задачи, вызывающей тело цикла на одном значении индекса. Однако очевидно, что в таком случае накладные расходы на создание и перемещение по памяти задачи могут оказаться выше, чем время на выполнение самого тела функции. Альтернативно, можно было бы разбить весь диапазон на число задач, равное числу потоков, но в таком случае при несбалансированной нагрузке некоторые задачи могли бы требовать в разы больше времени, чем другие, и вычислительные ресурсы потоков не были бы использованы наиболее оптимальным образом.

В OneTBB задачу оптимального разбиения диапазона работы на задачи решают так называемые разделители (англ. *partitioner*). Кроме собственно

разделения диапазона на меньшие диапазоны, они определяют стратегию распределения этих диапазонов по потокам.

OneTBB предоставляет следующие разделители [3]:

- `auto_partitioner` — стратегия по умолчанию, она изначально делит диапазон на большие поддиапазоны и по необходимости дробит дальше, основываясь на информации о том, была ли захвачена работа другим потоком через механизм `work-stealing`.
- `simple_partitioner` — делит диапазон на меньшие, пока их размер не меньше заданной заранее константы.
- `affinity_partitioner` — распределяет задачи на основе информации о распределении с предыдущей итерации, чтобы уменьшить процент кеш-промахов.
- `static_partitioner` — равномерное распределение итераций без балансировки нагрузки, имеет смысл использовать, если заранее известно, что нагрузка равномерно распределена.

1.2.2. OpenMP

OpenMP же, в отличие от OneTBB, требует поддержки компилятора, так как преобразовывает исходный код, основываясь на директивах, написанных в нём. В частности, для параллельного исполнения цикла `for` необходимо добавить директиву `#pragma omp parallel for`, по необходимости определив стратегию планирования, которую нужно применять (см. листинг 2).

OpenMP поддерживает следующие стратегии планирования (распределения работы) [9]:

- **static** — статическое распределение работы диапазонами одинакового размера (по числу потоков).
- **dynamic** — динамическое распределение работы через общую очередь диапазонов одинакового размера.
- **guided** — аналогичен **dynamic**, но размеры диапазонов не одинаковы, а убывают пропорционально числу нераспределённых итераций.

Вдобавок стратегии `dynamic` и `guided` имеют модификатор `monotonic` (или `nonmonotonic`), специфицирующие порядок исполнения — если установлен модификатор `monotonic`, то каждый из потоков будет после итерации с индексом i выполнять только итерации с большим номером.

Листинг 2 – Параллельный цикл `for` в OpenMP со стратегией `static`

```
#pragma omp parallel
#pragma omp for schedule(static)
for (int i = from; i < to; ++i) {
    func(i);
}
```

OpenMP для вычисления параллельных алгоритмов использует **fork-join** модель — когда исполнение «доходит» до директивы `#pragma omp parallel`, поток создаёт по необходимости (в зависимости от уровня параллелизма, который можно указать вызовом функции `omp_set_num_threads` или через переменную окружения `OMP_NUM_THREADS`) дополнительные потоки, которые будут выполнять код параллельного региона. Если в регионе есть директива `#pragma omp for` и цикл `for` после него, то диапазон такого цикла будет каким-либо образом разделен между потоками таким образом, что каждая итерация будет выполнена ровно один раз в каком-либо из потоков. В конце параллельного региона находится неявный с точки зрения пользовательского кода барьер. С точки зрения реализации все эти директивы заменяются на этапе компиляции на соответствующий код в промежуточном представлении компилятора (создание потоков, барьер и т.д.).

У OpenMP есть несколько реализаций, для сравнений в этой работе была выбрана реализация LLVM как наиболее активно развивающаяся по части улучшения производительности и используемая в аналогичных статьях и исследованиях [8].

1.3. Проблемы существующих решений

Несмотря на то, что уже есть два популярных в индустрии решения, в них не решены следующие проблемы:

- Статический подход к распределению работы, реализованный в OpenMP, показывает низкую производительность на несбалансированной нагрузке.
- Подход к планированию задач через захват работы (*work-stealing*), реализованный в OneTBB, показывает низкую производительность при сбалансированной нагрузке за счёт накладных расходов на излишнюю балансировку.

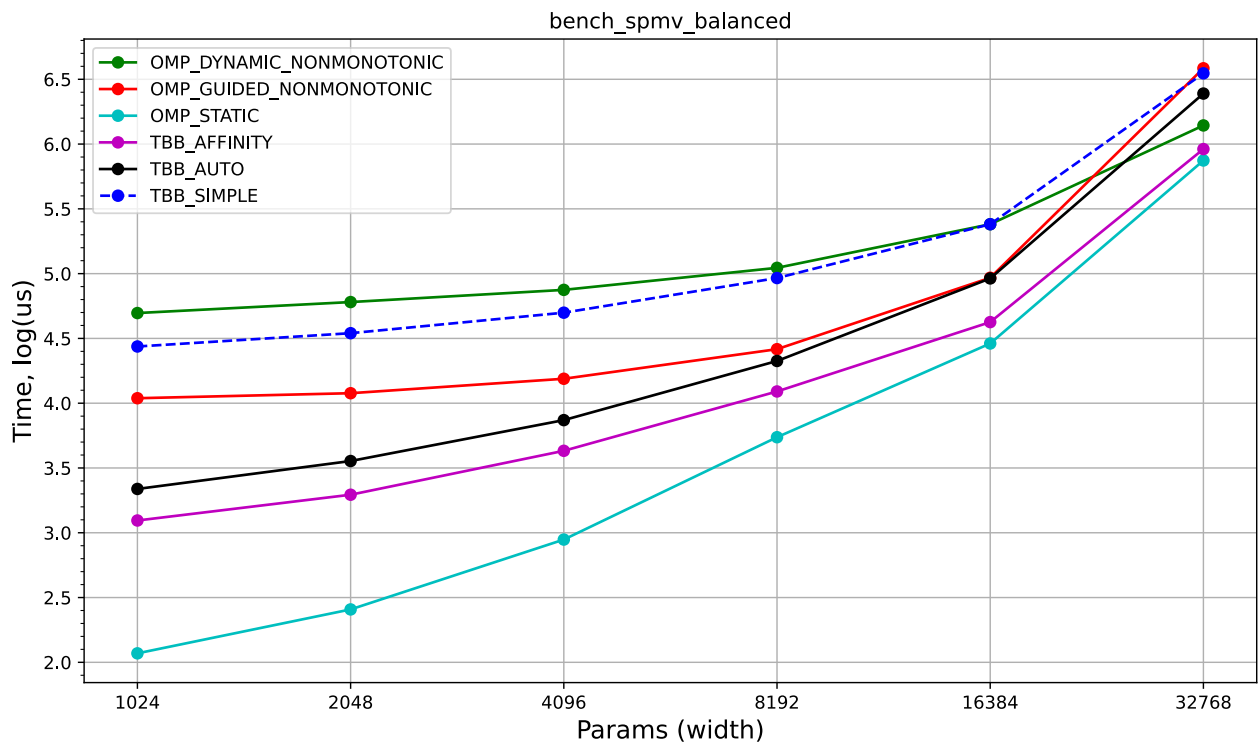


Рисунок 1 – Результаты бенчмарка SPMV Balanced, OpenMP и TBB

— OpenMP фактически не поддерживает компонентность и вложенный параллелизм, так как каждый вложенный вызов порождает новые потоки. Разница в производительности на сбалансированной и несбалансированной нагрузкой заметна на рисунках 1 и 2 — OpenMP Static сильно проигрывает TBB на несбалансированной, но показывает лучшую производительность на сбалансированной нагрузке. Подробнее приведённые бенчмарки описаны в третьей главе.

1.4. Постановка задачи

Как можно понять из предыдущего раздела, актуальна потребность в универсальном решении, которое бы имело удовлетворительную производительность вне зависимости от характера нагрузки, так как он может зависеть от входных данных и не всегда можно выбрать оптимальный подход на этапе написания кода. Помимо этого, необходимо решить и проблему компонентности — универсальное решение должно поддерживать вложенный параллелизм без многократной деградации производительности, как это происходит в OpenMP из-за создания новых потоков.

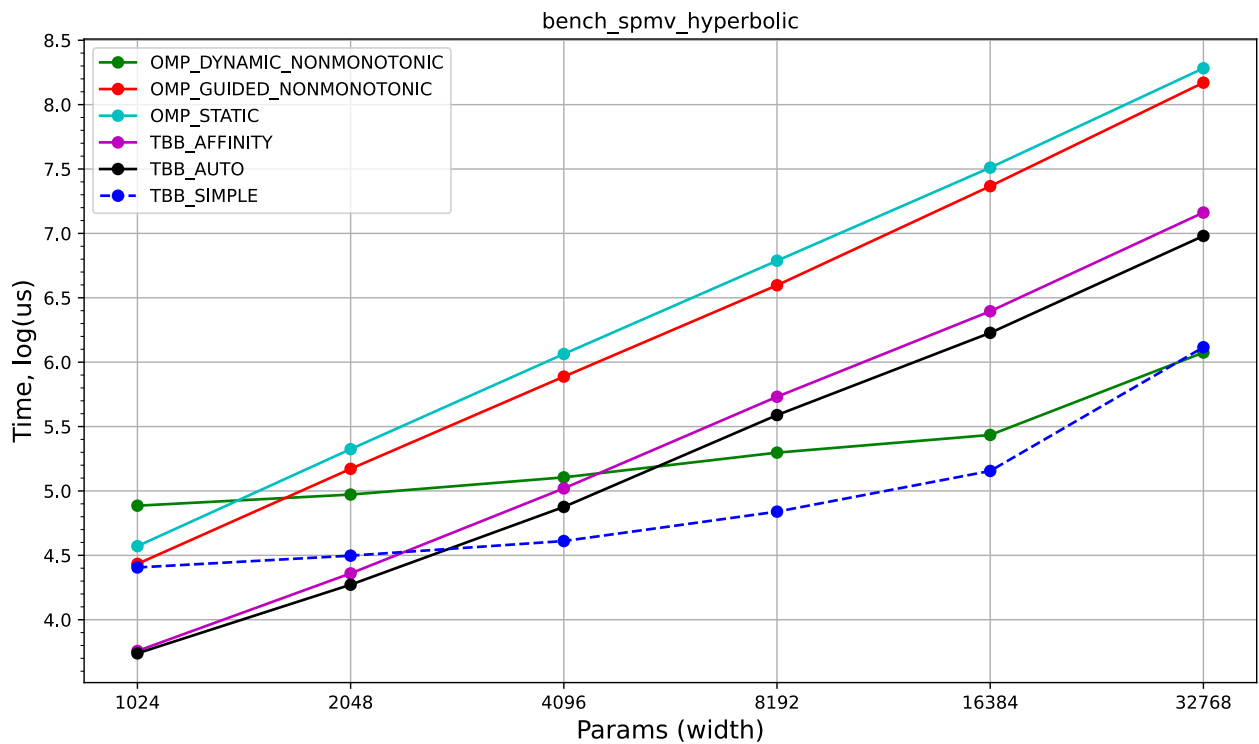


Рисунок 2 – Результаты бенчмарка SPMV Unbalanced, OpenMP и TBB

Понятно, что задача оптимального распределения работы для параллельного алгоритма специфична для конкретного алгоритма, поэтому в работе рассматривается преимущественно распределение работы для `parallel_for`, как наиболее часто используемого примитива, в том числе позволяющего строить поверх него другие, более сложные алгоритмы.

Реализовывать с нуля пул потоков с захватом работы нецелесообразно, учитывая наличие множества оптимально работающих решений и то, что это не является целью работы. Вместо этого, был использован готовый пул потоков из библиотеки Eigen, разработанный Дмитрием Вьюковым, который также проектировал планировщик для языка Go [6]. Он устроен следующим образом: в конструкторе создаются потоки, у каждого из которых есть локальная очередь задач. Метод `Submit` добавляет задачу в очередь случайного из потоков, а сами потоки регулярно проверяют свои очереди на наличие задач и, если в очереди что-то появилось, достают задачу и выполняют её. При этом для балансировки задач между потоками реализован механизм `work-stealing`: после того, как поток , он обходит в случайном порядке очереди других потоков и пытается взять задачу оттуда.

Необходимо реализовать поверх выбранного планировщика задач оптимальный алгоритм распределения работы, порождаемой `parallel_for`, по необходимости внося изменения в сам планировщик.

Выводы по главе 1

В этой главе был приведен обзор предметной области выполнения параллельных алгоритмов, в частности объяснены используемые далее термины и понятия, а также приведено описание двух популярных в индустрии подходов к выполнению параллельных алгоритмов. Также была сформулирована задача, которая решается в этой работе.

ГЛАВА 2. ПРЕДЛАГАЕМОЕ РЕШЕНИЕ

В этой главе описан алгоритм предлагаемого решения, а также разобраны проблемные места в распределении задач, которые были решены. Помимо того, приведены результаты бенчмарков, сравнивающие получившуюся реализацию с аналогами (OpenMP и OneTBB).

2.1. Решаемые подзадачи

Как можно понять из предыдущей главы, для реализации оптимального компонента `parallel_for` нужно решить следующие задачи:

- а) Эффективно распределить изначально работу по потокам.
- б) Во время исполнения по необходимости перераспределять работу между потоками, если начальное распределение оказалось неоптимальным из-за несбалансированного характера нагрузки.

На примере OneTBB можно убедиться, что удобно эти проблемы решать, декомпозировав алгоритм на две составляющих — разделитель работы (`partitioner`) и планировщик задач с механизмом захвата работы (`work-stealing` пул потоков). В таком случае мы получаем производительную компоненту — любой вложенный или конкурентный параллелизм будет работать на том же пуле потоков, не приводя к `oversubscription` (ситуации, когда число активных потоков превышает количество доступных для их выполнения ядер в системе) и связанным с ним негативным эффектам в виде падения производительности. Другими словами, мы переносим задачу планирования нагрузки из слоя операционной системы (планирование выполнения потоков между ядрами) в пользовательский код и переходим к планированию задач между потоками, чем снижаем накладные расходы за счёт меньшего переключения контекста потоков и переходов в пространство ядра.

Однако самого по себе распределения задач посредством `work-stealing` в пуле потоков не достаточно. Как мы видим на примере `simple_partitioner` из TBB, такой подход оказывается не самым оптимальным из-за описанных далее недостатков.

2.2. Начальное распределение задач

Распределение задач по потокам с помощью `work-stealing` плохо масштабируется — если потоки вынуждены искать задачи в очередях других потоков,

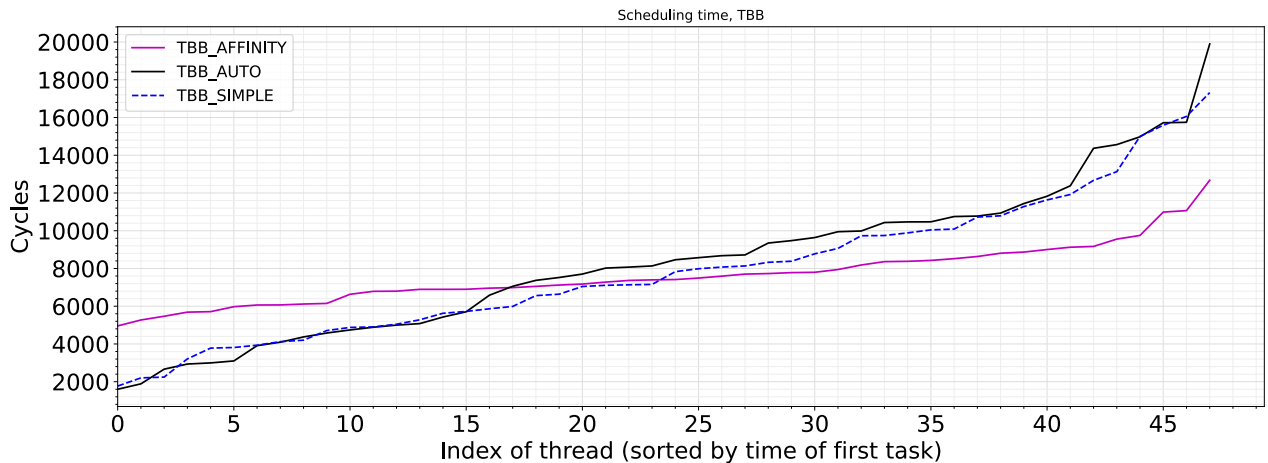


Рисунок 3 – Время от вызова `parallel_for` до начала выполнения первой задачи в OneTBB

время на поиск задач потоком задач растёт по мере уменьшения числа оставшихся задач, так как потокам приходится проверить большее число очередей. В частности, такая проблема заметна в OneTBB на небольшом числе задач, что видно по результатам замера времени от вызова `parallel_for` до старта выполнения задач на конкретном потоке, представленном на рисунке 3 — особенно выделяется на графике «хвост» для последних потоков.

Альтернативно начальному распределению задач через механизм случайного `work-stealing` предлагается использовать явное распределение задач по потокам (`work-sharing`) — будем явно и детерминированно распределять задачи по очередям потоков, не полагаясь на `work-stealing`. При этом очевидно, что если распределять будет один поток, то время распределения зависит линейно от числа потоков в системе — например, такая проблема наблюдается у `simple_partitioner` и `auto_partitioner` в TBB. Здесь предлагается следующая оптимизация: распределять изначально задачи по потокам в виде K -ичного дерева. В зависимости от числа ядер в системе значение параметра K может варьироваться, в простейшем случае можно использовать $K = 2$ — по результатам бенчмарка на рисунке 4 заметна разница между линейным распределением и деревьями с разным количеством детей. Распределение будет работать следующим образом:

- 1) Поток с номером L получает задачу с диапазоном итераций $[l, r)$ и диапазоном потоков $[L, R)$, задачи по которым ещё не были распределены.

- 2) Если $R - L \neq 1$, то есть остался один поток, то из диапазона $[L + 1, R)$ выбираются K потоков и на них запускается та же задача с равномерно разбитым по ним диапазоном итераций $[l + \frac{r-l}{R-L}, r)$.
- 3) Остальные $[l, l + \frac{r-l}{R-L})$ итераций выполняются в этом потоке, по мере их выполнения создаются балансировочные задачи и добавляются в очередь этого потока.

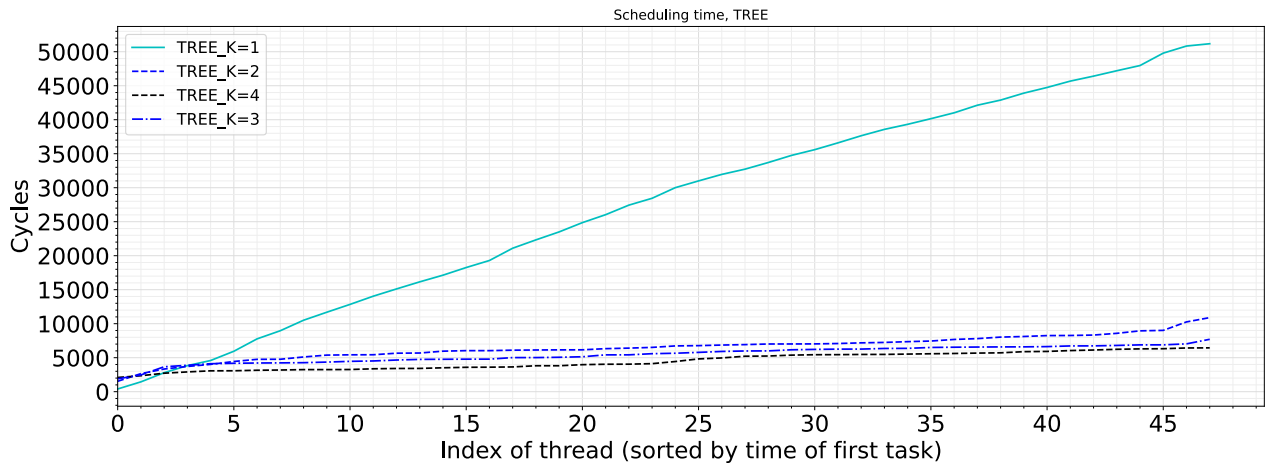


Рисунок 4 – Зависимость времени распределения задач от количества детей в дереве

В результате получается дерево исполнения глубины $\log N$, где N — число задач. То есть время распределения задач растёт логарифмически с увеличением числа потоков. При этом разумно изначально делить задачи оптимистично на равные диапазоны по потокам — в таком случае при сбалансированной нагрузке мы получим идеальное распределение.

После того, как поток поделил изначальный диапазон на задачи с одинаковыми меньшими диапазонами и добавил их в очереди других потоков, он делит свой диапазон на задачи, которые будем называть балансировочными и добавляет их в свою локальную очередь. Затем наконец поток начинает выполнение своей части диапазона из локальной очереди.

Однако в этом месте возникает конфликт парадигм распределения работы через её явную раздачу (work-sharing) и механизма захвата работы (work-stealing), за счёт которого выполняется балансировка. Он проявляется следующим образом: поток может захватить балансировочные задачи из другого потока до того, как к нему придёт его основной диапазон работы. Произойдёт преждевременный захват работы, который приводит к несбалансированному

изначальному распределению нагрузки даже в случае идеально сбалансированных входных данных, а также к недетерминированному распределению, что плохо сказывается на взаимодействиях с памятью и кешами.

Пример такого исполнения приведён на рисунке 5: поток T_0 добавил потокам T_1 и T_2 в очередь на исполнение задачи с диапазонами $[\frac{n}{4}, \frac{n}{2})$ и $[\frac{n}{2}, n)$, создал балансировочные задачи и начал выполнение. Поток T_3 до того, как в его очередь попала задача с диапазоном $[\frac{3n}{4}, n)$, которую создает поток T_2 , захватывает балансировочную задачу из очереди потока T_0 и начинает её выполнение. Как результат, изначальная равномерность изначального распределения нарушена. При этом, если в системе больше потоков, задача переданная потоком T_2 потоку T_3 не будет разделена и распределена дальше, пока он выполняет захваченную балансировочную задачу.

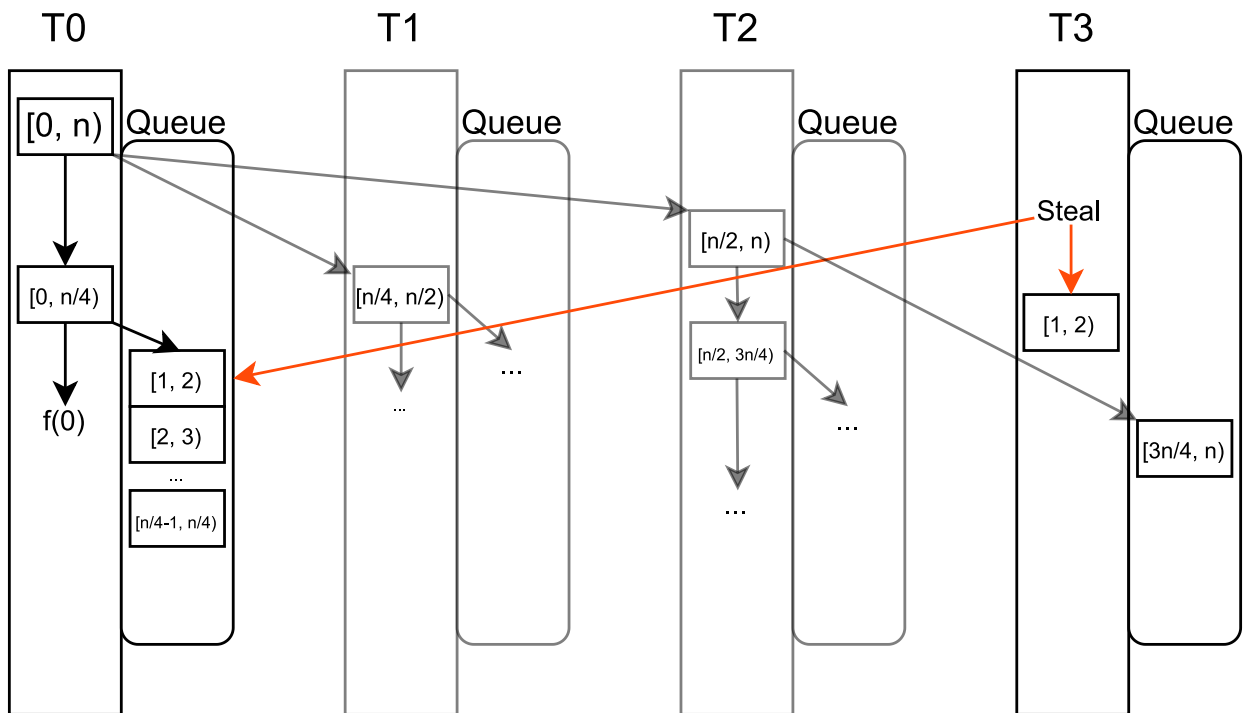


Рисунок 5 – Пример исполнения с преждевременным захватом работы

Решать эту проблему предлагается следующим образом: после того, как задача из изначального распределения начинает выполняться потоком, он не должен создавать балансировочные задачи до того момента, пока изначальное распределение работы не завершится, то есть пока все потоки не получают через алгоритм, описанный выше, свои задачи. Это можно было бы реализовать в виде некоторой общей точки синхронизации, такое решение плохо масштабируется из-за конфликтов потоков на общей памяти (contention).

Был выбран альтернативный вероятностный подход. Он основан на времени, за которое на конкретной конфигурации на выбранном числе потоков задачи успевают распределиться по всем потокам. Сводится он к тому, что поток не сразу создаёт балансировочные задачи, а только спустя заранее заданный промежуток времени.

Понятно, что такой подход не гарантирует, что за этот промежуток времени задачи успеют распределиться по всем потокам, но вероятность этого можно увеличить правильным подбором константы. Для этого реализована отдельная программа (`timespan_tuner`), запускающая большое число итераций (10000), состоящих из запуска `parallel_for` из `num_threads` задач, каждая из которых блокируется в ожидании запуска оставшихся. По каждой итерации считается статистика времени, которое ушло на распределение задач, а потом по всем итерациям берётся 99-я перцентиль — при выборе такой константы ожидается, что в 99% запусков распределение работы будет укладываться в этот промежуток времени. Время замеряется инструкцией `rdtsc` на x86 [10] (и её альтернативой `mrs %0, cntvct_el0` на ARM [4]).

Аналогичное предыдущему исполнению, но с отложенным выполнением задач, изображено на рисунке 6 — попытка потока T_3 захватить задачу из чужой очереди заканчивается неуспехом, так как очереди ещё пусты и он начинает выполнение явно переданной ему после этого части диапазона.

2.3. Интеграция с планировщиком задач

Учитывая, что проектируемое решение должно работать с компоновым планировщиком задач, распределение работы нужно делать в виде одной из задач, которую планировщик принимает и выполняет. Для интеграции этого алгоритма с планировщиком задач, взятым из библиотеки Eigen, потребовались некоторые доработки.

Во-первых, при вызове `parallel_for` вызывающий поток обязан дожидаться завершения работы, но делать это ожидая на барьере не эффективно — поток вместо этого может выполнять часть задач из диапазона, который нужно выполнить в `parallel_for`, как это происходит в OpenMP и TBB. Для этого в планировщике был реализован метод `JoinMainThread()`, позволяющий основному потоку программы, то есть не созданному планировщиком, подключиться к выполнению задач.

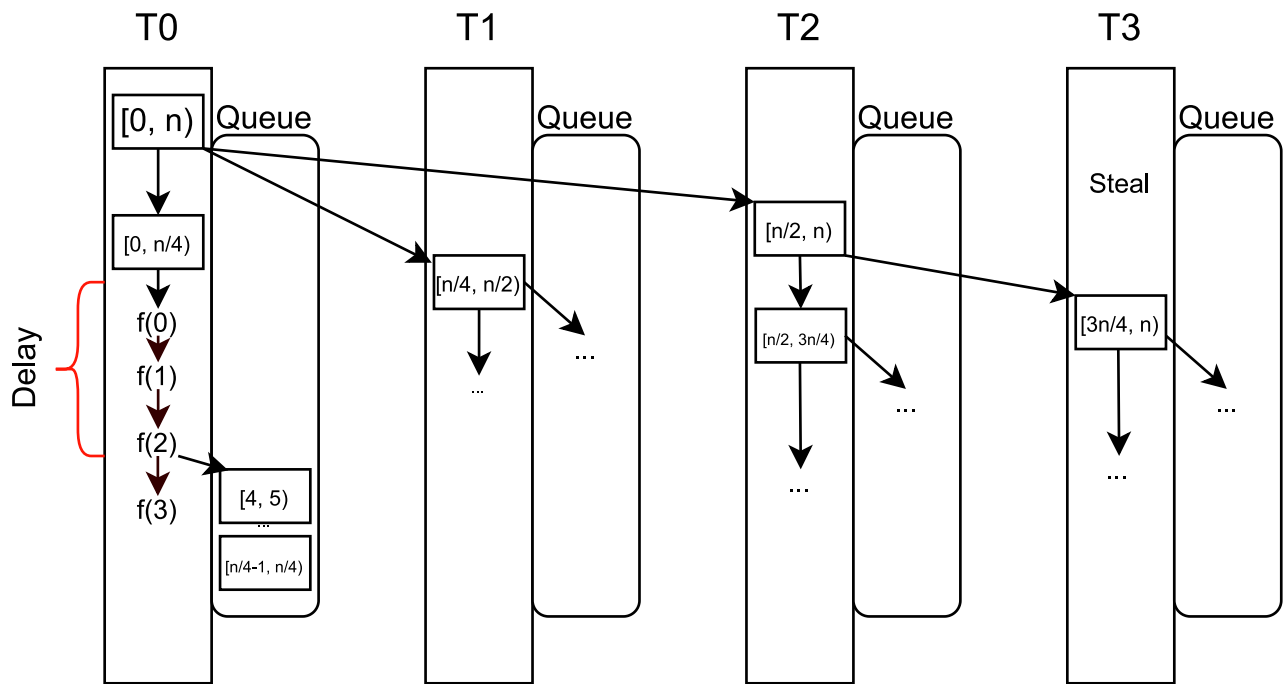


Рисунок 6 – Пример исполнения с отложенным созданием балансирующих задач

Во-вторых, для оптимальной работы недостаточно простого метода `Submit(F Func)` у планировщика, потому что в таком случае алгоритм мог бы добавлять задачу в очередь только случайного потока и не было бы контроля над тем, на каких потоках предпочтительно будут исполнены задачи — для поддержки такого в планировщик был добавлен метод `Submit(F func, ThreadId idx)`, добавляющий задачу в очередь указанного потока, а не случайного.

Итоговый интерфейс планировщика приведён на листинге 3.

Листинг 3 – Интерфейс модифицированного пула потоков

```
class ThreadPool {
    ThreadPool(size_t threadNum);
    void Submit(F func);
    void Submit(F func, ThreadId idx);
    size_t CurrentThreadId();
    void JoinMainThread();
};
```

Распределение работы реализовано с помощью задач, которые порождают другие задачи и передают их в планировщик. Упрощенный код задачи представлен на листинге 4. Методы `DistributeInitialWork` и `CreateBalancingTasks` создают дополнительные задачи, по алгорит-

му описанному в предыдущем разделе и добавляют их в очереди соответствующих потоков (`DistributeInitialWork` — в очереди других потоков, `CreateBalancingTasks` — в очередь текущего потока). Методы `ExecuteForTimespan` и `ExecuteCurrentRange` выполняют задачи из текущего диапазона — первый выполняет их только в течение заранее заданного промежутка времени, а второй — оставшуюся часть диапазона.

Листинг 4 – Задачи, запускаемые в планировщике

```
template<typename F, bool Initial>
struct Task {
    void operator() () {
        if constexpr (initial) {
            DistributeInitialWork ();
        }
        ExecuteForTimespan ();
        CreateBalancingTasks ();
        ExecuteCurrentRange ();
        CurrentNode_.Reset ();
    }
private:
    Scheduler &Sched_;
    Range Range_;
    size_t End_;
    Func Func_;
    IntrusivePtr<TaskNode> CurrentNode_;
};
```

Ожидание завершения в главном потоке было реализовано с помощью создания явного дерева задач из интрузивных указателей — для каждой задачи аллоцируется узел в дереве со счётчиком ссылок и указателем на предка, в задаче содержится указатель на соответствующий ей узел. Когда задача завершается, счётчик ссылок уменьшается на 1 и, если все дети этой задачи уже закончили выполнение, ссылок на этот узел дерева не остаётся и он разрушается, автоматически уменьшая счётчик ссылок у предка. Таким образом, к концу выполнения задач остаётся только один узел — корень дерева, созданный в вызове функции `ParallelFor`. Чтобы дождаться завершения всех итераций, достаточно в теле функции `ParallelFor` дождаться, пока у корня не останется других ссылок (то есть счётчик ссылок не станет равен 1).

2.4. Адаптивный размер балансировочных задач

Ещё одна проблема заключается в том, что заранее невозможно выбрать универсальное разделение диапазона итераций на задачи (в терминах планировщика) оптимального размера, так как невозможно предугадать время исполнения каждой итерации. Реализации в `simple-partitioner` из OneTBB и `Dynamic` из OpenMP по сути не решают эту проблему и отдают её пользователю, предоставляя возможность указать минимальный размер диапазона (по умолчанию равный 1), что не всегда подходит, если заранее не известны параметры нагрузки. Низкое значение этого параметра при небольшом размере задач приводит к тому, что накладные расходы начинают превышать затраты на выполнение непосредственно задач, а слишком большое значение не даёт оптимально балансировать нагрузку. Механизм адаптивного подбора granularity задач реализован в `auto_partitioner` из OneTBB, однако он показывает не лучшую производительность по результатам бенчмарков, представленных в следующей главе — возможно, из-за того, что используется механизм проверки, была ли захвачена задача из очереди другим потоком, что приводит к дополнительной синхронизации на общем атомарном флаге.

В реализованном алгоритме создание балансировочных задач работает после того, как поток распределил исходный диапазон по остальным потокам. В простейшем случае весь оставшийся диапазон итераций можно поделить на задачи по одной итерации ($k = 1$), но в таком случае, в зависимости от нагрузки, накладные расходы на добавление задачи в очередь и вытаскивание из неё (текущим потоком или же другим потоком при захвате работы) могут оказаться вычислительно затратнее, чем само выполнение задачи. Для оптимизации этого процесса было применено следующее решение: во время задержки балансировки подсчитываем, сколько задач успел выполнить поток (назовём эту величину `grainsize` по аналогии с параметром в OneTBB), а затем полученное значение используется в качестве минимального числа итераций в одной балансировочной задаче. Таким образом, в условиях нагрузки из «лёгких» итераций заметно снижается суммарное время выполнения за счёт увеличения размера задач.

2.5. Тестирование

Для проверки корректности разработанного алгоритма были реализованы тесты, проверяющие основные сценарии использования `parallel_for`, в частности:

- Несколько базовых запусков `parallel_for` подряд.
- Конкурентные вызовы `parallel_for` из разных потоков.
- Тест, проверяющий, что `parallel_for` использует все потоки.
- Вложенные вызовы `parallel_for`.
- Тест, проверяющий работоспособность `work-stealing`.
- Тест, проверяющий равномерность изначального распределения.

Вдобавок был реализован стресс-тест, запускающий `parallel_for` из разных потоков со случайными задержками в задачах.

Выводы по главе 2

В этой главе было представлено решение поставленной задачи, включающее в себя несколько аспектов:

- Оптимальное начальное распределение задач, порожденных алгоритмом, по потокам.
- Адаптивный подбор размера для создаваемых балансировочных задач.
- Интеграция алгоритма с `work-stealing` планировщиком задач для поддержки компонуемости.

Также были описаны реализованные тесты алгоритма.

ГЛАВА 3. БЕНЧМАРКИ И СРАВНЕНИЕ С АНАЛОГАМИ

В этой главе описаны бенчмарки, разработанные в рамках работы для оценки производительности разных реализаций алгоритма `parallel_for` на различных типах нагрузки. Также приведено описание систем, результаты запуска на которых используются в работе. Помимо того, проведен анализ производительности существующих популярных решений и сравнение с ними разработанного решения, описанного в предыдущей главе.

3.1. Описание бенчмарков

В этом разделе описано устройство и характеристики бенчмарков. Краткий обзор их свойств представлен также в таблице 1.

3.1.1. Умножение разреженной матрицы на вектор (SPMV)

Умножение матрицы на вектор — это часто используемая операция в высокопроизводительных вычислениях, особенно для разреженных матриц, что часто встречается, например, в машинном обучении. В этом бенчмарке мы используем сжатое представление матрицы в формате Compressed Sparse Row (CSR), где каждая матрица представлена тремя массивами: ненулевыми значениями, индексами их столбцов и массивом из указателей в массив индексов столбцов по каждой строке. Умножение матрицы и вектора производится путем итерации по строкам матрицы и умножения каждой строки на вектор. Эта задача хорошо параллелизуется, так как каждую строку можно умножать независимо. Код параллелизации такого умножения представлен в листинге 5.

Мы будем фокусироваться на двух интересных случаях: сбалансированных и несбалансированных матрицах. В контексте этой задачи матрица считается сбалансированной, если количество ненулевых элементов в каждой строке примерно одинаково — размер всех задач будет примерно одинаковым. В противном случае мы будем называть матрицу несбалансированной. Во всех случаях использовалась матрица из $\text{num_threads} \cdot 2^9$ строк с плотностью элементов $\frac{1}{27}$, проверялись разные варианты ширины строки от 2^{10} до 2^{15} . Параметры были подобраны таким образом, чтобы бенчмарк занимал адекватное время на выполнение, но при этом был заметен выигрыш от его параллелизации.

Таблица 1 – Обзор характеристик бенчмарков

Название	Нагрузка	Вызовов <code>parallel_for</code> и итераций
SPMV-balanced	Сбалансирована	1 по N
SPMV-triangle	Несбалансирована	1 по N
SPMV-hyperbolic	Несбалансирована	1 по N
Scan	Сбалансирована	$2 \log N$ по $\frac{N}{2^i}, i = 1.. \log N$
Reduce	Сбалансирована	1 по $\frac{N}{1024}$
Scheduling time	Сбалансирована	1 по <code>num_threads</code>
MatrixTranspose	Сбалансирована	Вложенный: 1 вызов из 16 итераций, в каждой ещё вызов из 16 итераций
MatrixMultiply	Сбалансирована	Вложенный: 1 вызов из <i>Rows</i> итераций, в каждой ещё 1 вызов из <i>Cols</i> итераций

Листинг 5 – Параллельное умножение разреженной матрицы на вектор

```

double MultiplyRow(SparseMatrix m, Vector column, size_t row) {
    double res = 0;
    for (size_t i = m.RowIndex[row]; i < m.RowIndex[row + 1]; ++i) {
        res += m.Values[i] * column[m.ColumnIndex[i]];
    }
    return res;
}

Vector MultiplyMatrix(SparseMatrix m, Vector column) {
    Vector out(m.Rows);
    ParallelFor(0, m.Rows, [&](size_t i) {
        out[i] = MultiplyRow(m, column, i);
    });
    return out;
}

```

3.1.1.1. Сбалансированная версия

В этом бенчмарке были сгенерированы сбалансированные матрицы фиксированного размера, в которых число элементов равномерно распределено между строками и столбцами. Ожидается, что на этом бенчмарке будут наиболее эффективны подходы статического распределения работы, так как он более предсказуем и может быть легко распараллелен путем равномерного деления между потоками.

3.1.1.2. Несбалансированная версия

Несбалансированная матрица может быть несбалансирована по-разному: с одинаковым числом элементов размер задач может распределяться по-разному. Мы будем сравнивать два случая: с равномерно убывающим количеством элементов в строке (назовем его **spmv_triangle** бенчмарк) и с гиперболически убывающим количеством элементов (**spmv_hyperbolic** бенчмарк).

От динамического подхода распределения ожидается большая эффективность для несбалансированных матриц. При равномерном статическом планировании некоторые потоки закончат работу раньше и будут простаивать, пока другие потоки все еще работают. Динамическое распределение позволит лучше балансировать нагрузку между потоками, но оно более вычислительно затратное из-за накладных расходов на захват работы или другие виды синхронизации.

3.1.2. Свёртка (reduce)

Этот бенчмарк вычисляет сумму массива целых чисел, разбивая его на блоки заранее заданного размера и считая сумму в каждом блоке параллельно. Этот бенчмарк представляет случай полностью сбалансированной нагрузки, но по сравнению с бенчмарком умножения разреженной матрицы на столбец размер каждой порции данных в этом бенчмарке больше. Вследствие чего мы ожидаем низкие накладные расходы относительно времени выполнения самих задач и высокую производительность как для статического алгоритма планирования, так и для динамического.

3.1.3. Сканирование (scan)

Scan — это параллельный алгоритм, который вычисляет все префиксные суммы массива целых чисел. Мы используем реализацию с двумя фазами: Up-Sweep (Reduce) и Down-Sweep, как описано в статье «Parallel Prefix Sum (Scan) with CUDA» от NVIDIA [5]. Этот бенчмарк интересен тем, что он содержит множество ($2 \log N$) вызовов `parallel_for`, часть из которых выполняет небольшое число лёгких задач, что позволяет пронаблюдать накладные расходы на фазы инициализации и завершения алгоритма.

3.1.4. Вложенный параллелизм

Для проверки производительности решений в условиях применения вложенного параллелизма было реализовано два похожих бенчмарка: `MMultiple` (умножение двух матриц) и `MTranspose` (транспонирование матрицы). Отличаются эти бенчмарки тем, что умножение двух матриц имеет большие диапазоны (внешний `parallel_for` итерируется по строкам матрицы, а внутренний — по столбцам), когда транспонирование матриц делит матрицу на фиксированное число блоков и итерируется только по ним. Псевдокод этих бенчмарков представлен на листингах 6 и 7 соответственно.

Листинг 6 – Параллельное умножение двух матриц

```
DenseMatrix MultiplyMatrix(DenseMatrix A, DenseMatrix B) {
    DenseMatrix out(A.Rows, B.Columns);
    ParallelFor(0, out.Rows, [&](size_t row) {
        ParallelFor(0, out.Columns, [&](size_t col) {
            double sum = 0;
            for (size_t j = 0; j != A.Columns; ++j) {
                sum += A[row][j] * B[j][col];
            }
            out[row][col] = sum;
        });
    });
}
```

Листинг 7 – Параллельное транспонирование матрицы

```
static constexpr size_t BLOCKS = 16;
void TransposeMatrix(DenseMatrix &input, DenseMatrix &out) {
    ParallelFor(0, BLOCKS, [&](size_t row) {
        ParallelFor(0, BLOCKS, [&](size_t column) {
            // calc fromRow, toRow, etc...
            for (size_t i = fromRow; i != toRow; ++i) {
                for (size_t j = fromCol; j != toCol; ++j) {
                    out[j][i] = input[i][j];
                }
            }
        });
    });
}
```

3.1.5. Время распределения задач

Также был разработан специальный бенчмарк (будем называть его `scheduling_time`) для наблюдения за скоростью распределения задач между

потоками. Основное внимание в этом бенчмарке уделяется измерению времени между моментом вызова `parallel_for` и моментами, когда каждый поток начинает выполнять свою первую задачу. Результаты этого бенчмарка помогают понять, сколько времени планировщик тратит на распределение задач. Ожидается, что в этом случае статический подход к планированию будет более эффективен, поскольку не требуется никакой синхронизации между потоками.

3.2. Конфигурация системы

Все бенчмарки были запущены на двух системах: на виртуальной машине с 48 ядрами в облачном окружении с двумя процессорами Intel Xeon Gold 6338 (с отключенным HyperThreading) и на 48-ядерном Huawei Kunpeng 920 на архитектуре ARM. Для компиляции использовался компилятор языка C++ Clang версии 14.

Вычислительно интенсивные бенчмарки были выполнены с использованием библиотеки Google Benchmark. Бенчмарк времени распределения задач планировщиком выполнялся с помощью собственной реализации сбора данных, а время выполнения было измерено в тактах процессора с использованием инструкции `rdtsc` для x86 [10] и инструкции `mrs %0, cntvct_el0` для ARM [4].

Для каждого бенчмарка были протестированы различные конфигурации библиотек. Реализация OpenMP использовалась из библиотеки LLVM OpenMP v16.0.2 с различными политиками планирования: `static`, `dynamic` и `guided`. Политики планирования `guided` и `dynamic` были проверены в комбинации с параметром `nonmonotonic` и `monotonic`. Потоки привязывались к ядрам, используя переменную среды `OMP_PROC_BIND=close`, чтобы избежать миграции одного потока на другое ядро, отличное от того, на котором он изначально был запущен.

Кроме того, были установлены значения `OMP_WAIT_POLICY=active` и `KMP_BLOCKTIME=infinite`, чтобы гарантировать, что потоки не «засыпают» во время ожидания задач, что могло бы привести к нестабильным результатам. Для включения вложенного параллелизма использовалось выставление параметра `OMP_MAX_ACTIVE_LEVELS=8`, по умолчанию в OpenMP вложенный параллелизм отключен как раз по причине его низкой производительности — так как он создает новые потоки, это часто приводит к тому, что

потоков становится больше чем ядер, в результате чего стремительно деградирует производительность из-за их переключений. Помимо этого, в принципе операция старта новых потоков занимает существенное время.

Для OneTBB использовался планировщик по умолчанию с настройками по умолчанию, все потоки были привязаны к ядрам с использованием `observer`. [13]

Чтобы убедиться, что при измерении производительности будут задействованы все потоки, перед запуском измеряемой части бенчмарков проводился прогрев блокирующими задачами, активно ожидающими, пока начнется выполнение на всех потоках. Такая процедура помогает бороться с тем, что библиотеки (и TBB, и OpenMP) создают потоки «лениво», что могло бы сказаться на воспроизводимости результатов.

3.3. Результаты бенчмарков

В этом разделе представлены результаты бенчмарков трех различных решений — OneTBB, OpenMP и алгоритма, представленного в предыдущей главе. Для увеличения читабельности графиков с них были скрыты результаты OpenMP с модификатором `monotonic`, так как они оказались хуже, чем аналогичные стратегии планирования с `nonmonotonic`, который применяется по умолчанию.

На легенде графиков эти алгоритмы имеют следующие названия:

- `OMP_DYNAMIC_NONMONOTONIC` - Стратегия планирования `Dynamic` из OpenMP с модификатором `nonmonotonic`
- `OMP_GUIDED_NONMONOTONIC` - Стратегия планирования `Guided` из OpenMP с модификатором `nonmonotonic`
- `OMP_STATIC` - Статическое распределение из OpenMP
- `TBB_AFFINITY` - `parallel_for` из TBB с разделителем `affinity_partitioner`
- `TBB_AUTO` - `parallel_for` из TBB с разделителем `auto_partitioner`
- `TBB_SIMPLE` - `parallel_for` из TBB с разделителем `simple_partitioner`
- `TIMESPAN_GRAINSIZE` - реализованный в работе алгоритм

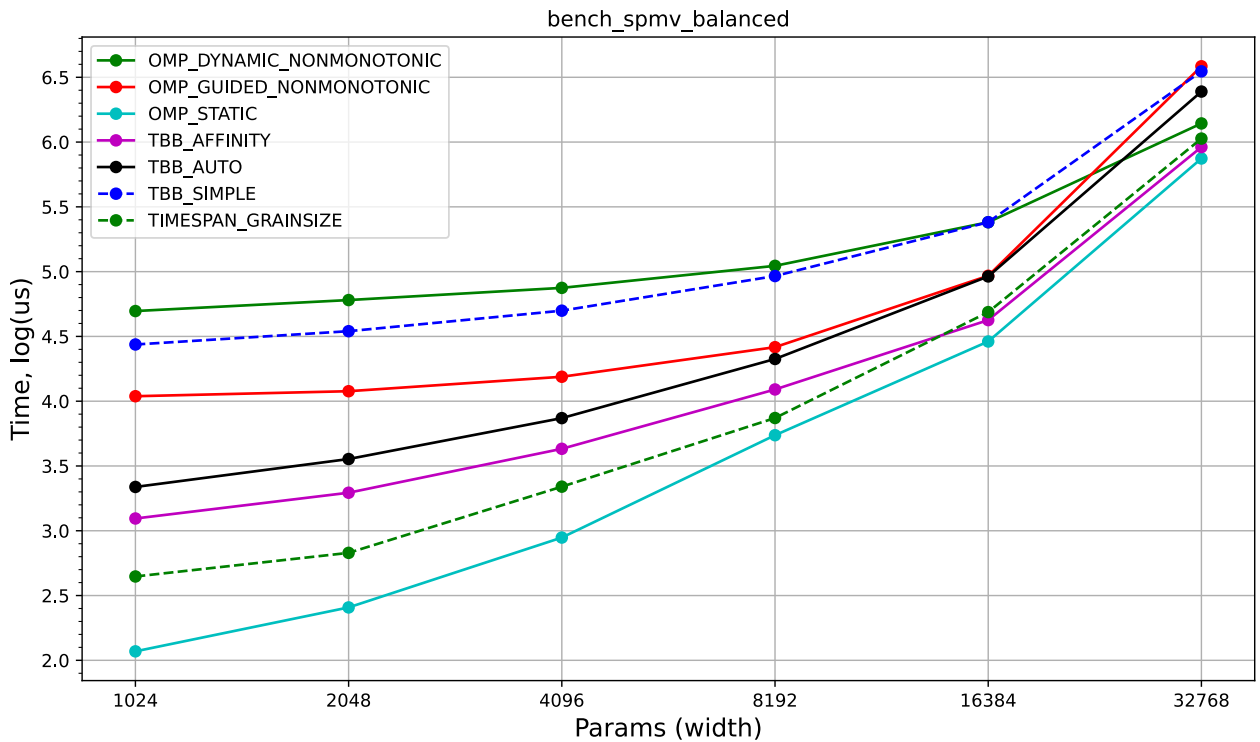


Рисунок 7 – Результаты бенчмарка SPMV Balanced, система с x86

3.3.1. SPMV

Результаты бенчмарка SPMV представлены на рисунках 7, 8 и 9. Для демонстрации зависимости производительности от размера задач бенчмарк запускался с разным параметром `width`, отвечающим за размер строки — от неё прямо пропорционально число действий на каждой итерации цикла. По оси X отмечена ширина строки матрицы, а по оси Y — время на выполнение бенчмарка в микросекундах по логарифмической шкале. Заметно, что с увеличением размера задач сокращается разрыв в производительности между различными алгоритмами, что в целом объяснимо тем, что на задачах большего размера накладные расходы на распределение задач менее заметны на фоне времени, затраченного на выполнение тела задач — особенно это заметно при сбалансированной нагрузке.

Также на рисунках 10 и 11 представлены различные вариации разработанного алгоритма поверх одного и того же планировщика задач (Simple – задачи по одной итерации, Static – статическое распределение, Timespan – с отложенной балансировкой, Timespan Grainsize – с отложенной балансировкой и адаптивной гранулярностью задач, финальная версия). Отчетливо видно, насколько каждая из оптимизаций вносит выигрыш в итоговый результат алгоритма.

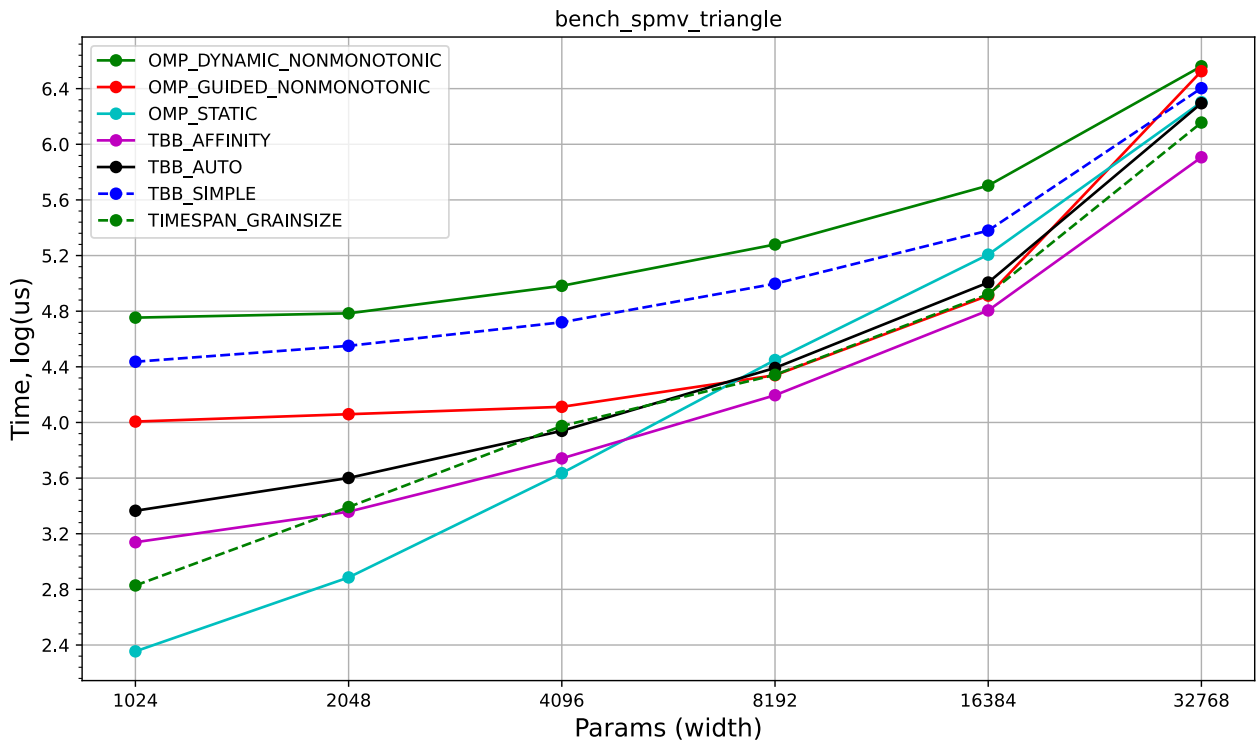


Рисунок 8 – Результаты бенчмарка SPMV Triangle, система с x86

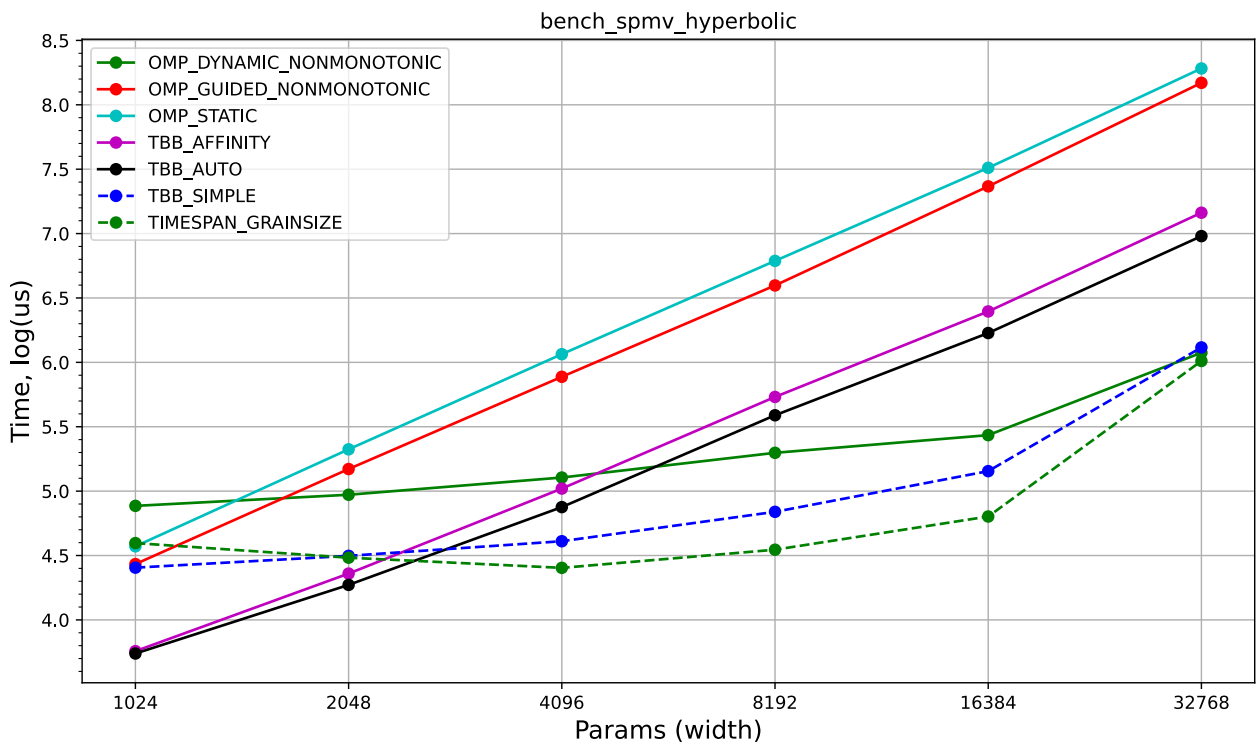


Рисунок 9 – Результаты бенчмарка SPMV Hyperbolic, система с x86

На системе с ARM результаты получились похожими (рисунки 12, 13, и 14), но при этом выигрыш в производительности заметнее, чем на системе с x86 — предположительно, из-за более низкой производительности на одно

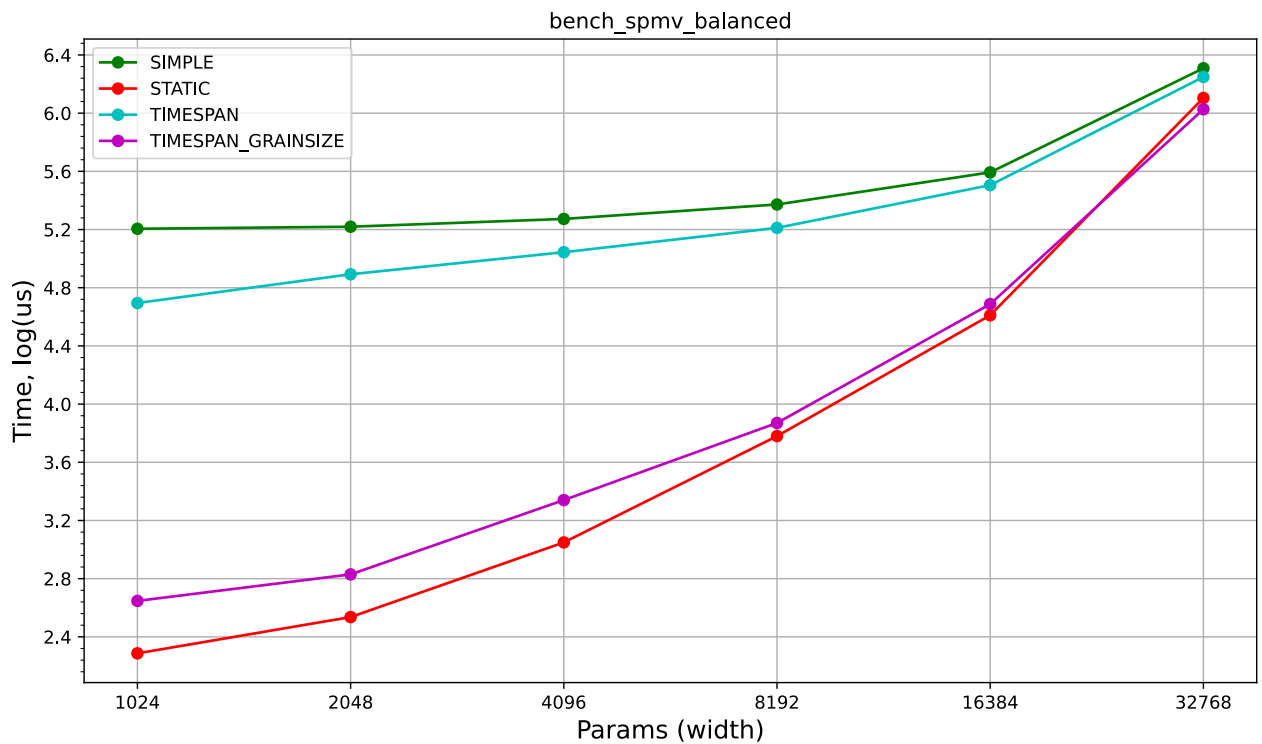


Рисунок 10 – Результаты бенчмарка SPMV Balanced, только разработанный алгоритм, система с x86

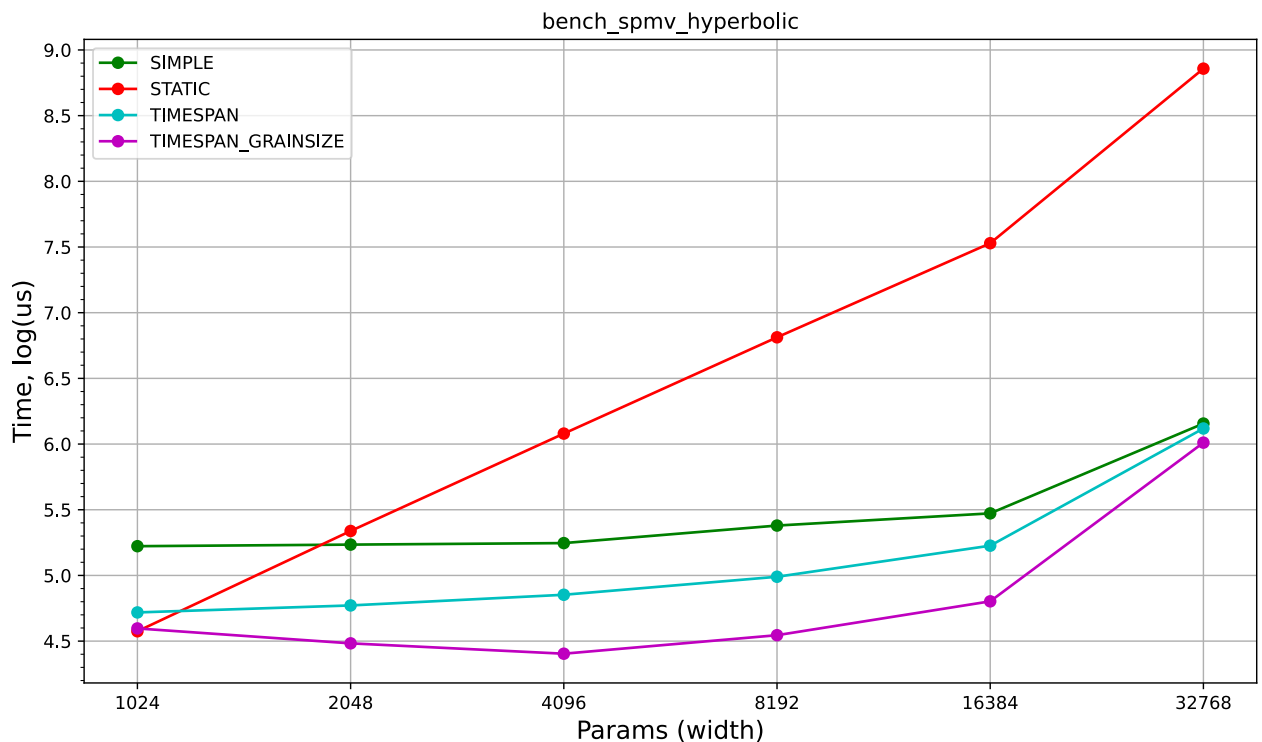


Рисунок 11 – Результаты бенчмарка SPMV Hyperbolic, только разработанный алгоритм, система с x86

ядро, при которой важнее утилизация ресурсов всех потоков и становятся заметнее накладные расходы.

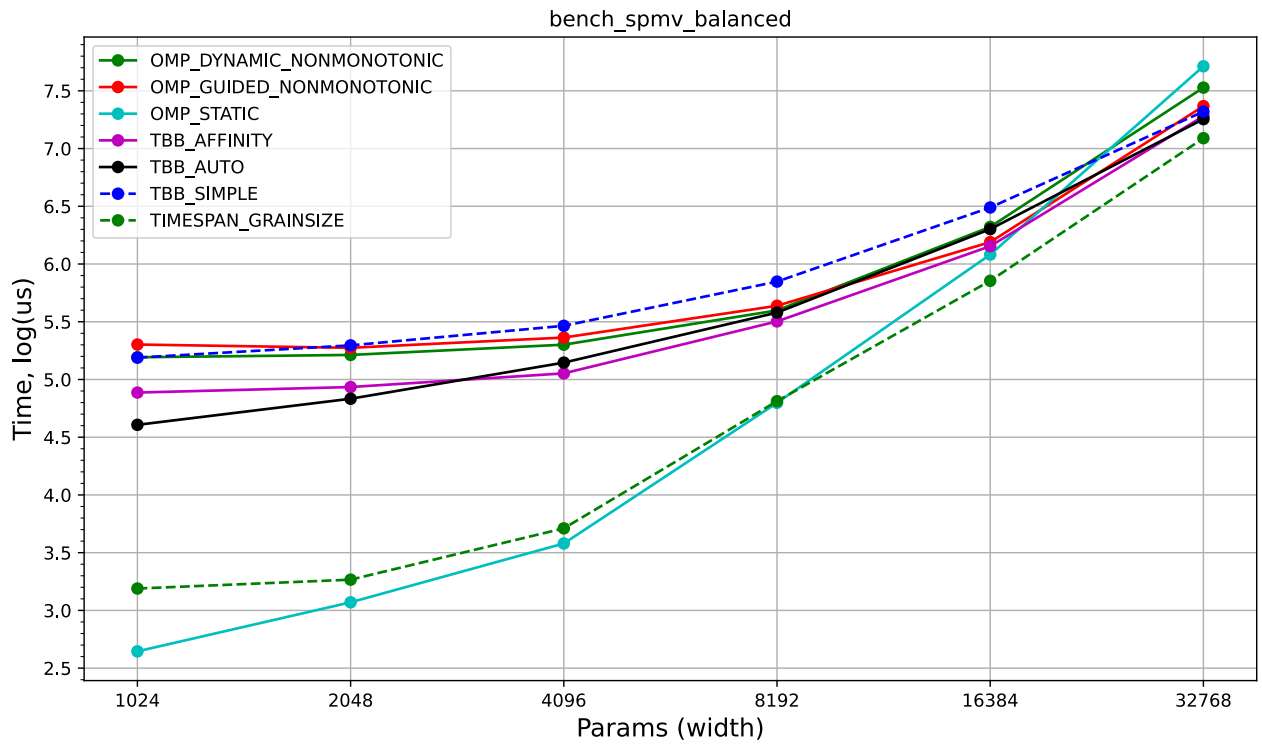


Рисунок 12 – Результаты бенчмарка SPMV Balanced, система с ARM

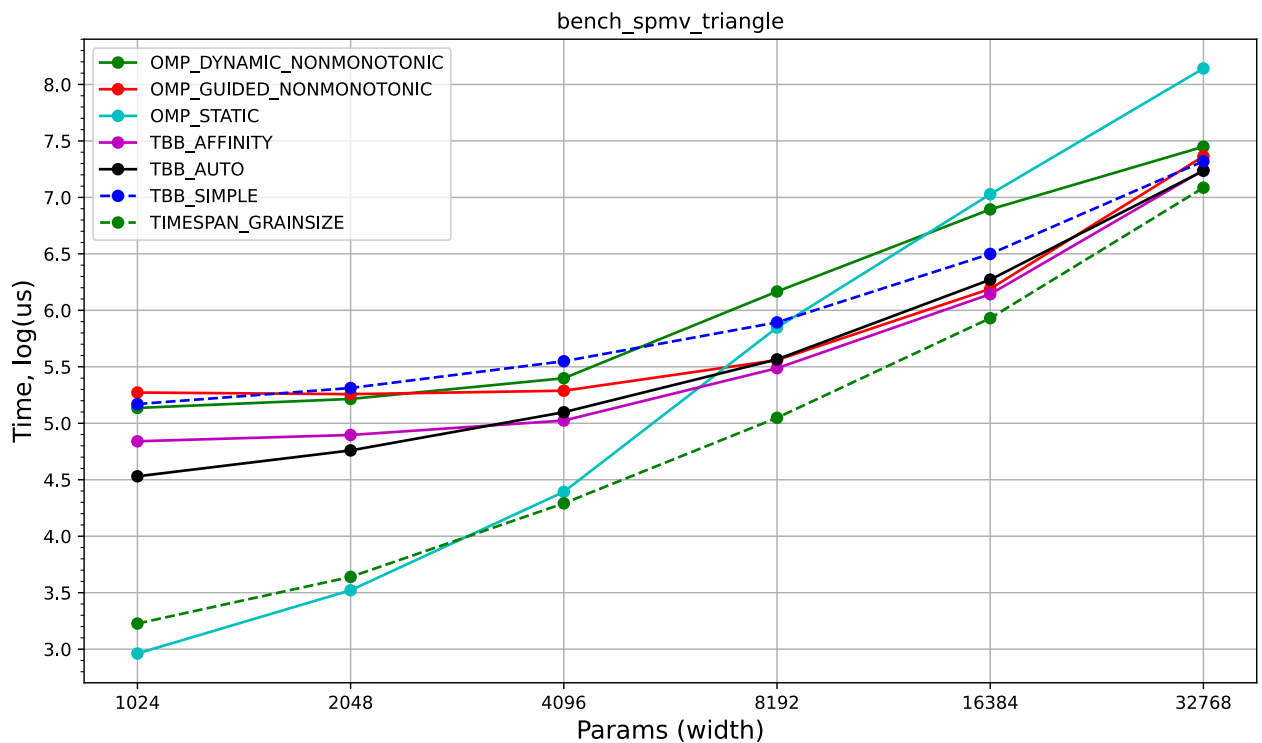


Рисунок 13 – Результаты бенчмарка SPMV Triangle, система с ARM

3.3.1.1. Сбалансированная матрица

Как ожидалось, в этом случае статический подход к планированию оказался более эффективным, чем динамический. Это особенно заметно при срав-

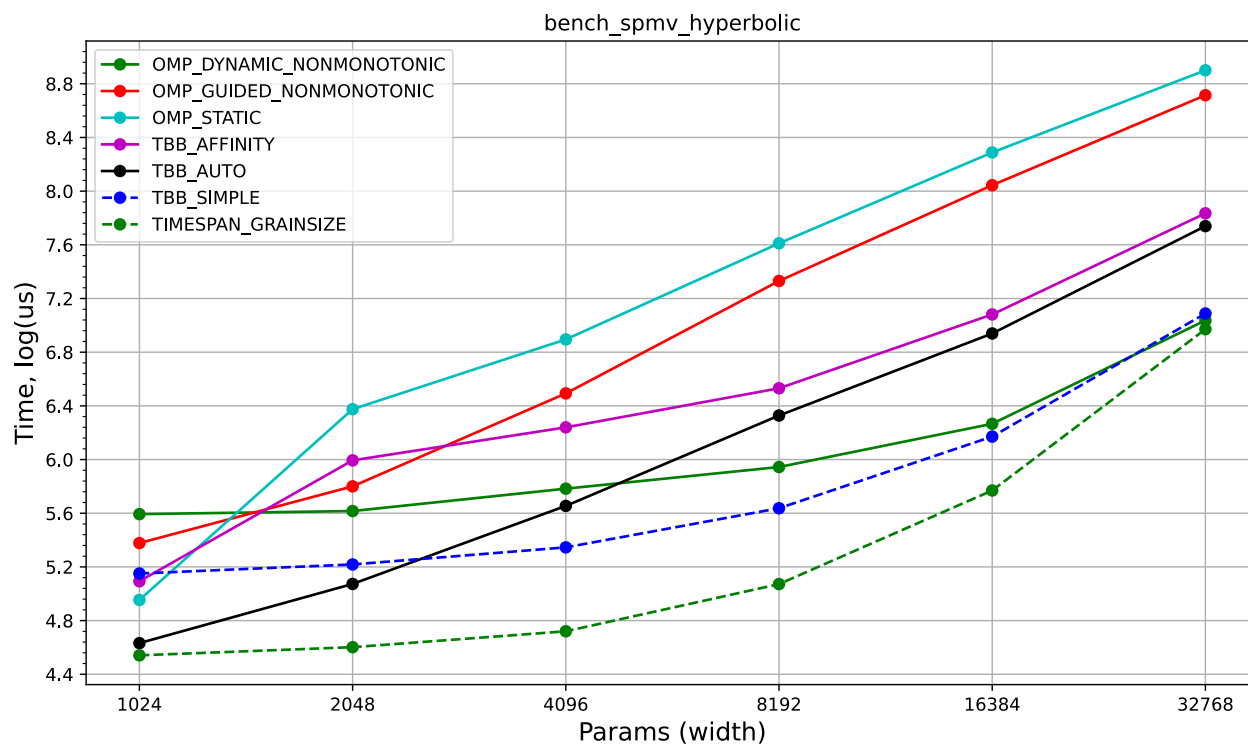


Рисунок 14 – Результаты бенчмарка SPMV Hyperbolic, система с ARM

нении различных политик планирования одной и такой же библиотеки: статическое распределение работы в OpenMP оказалось более эффективным, чем динамическое – среднее время выполнения не меньше, чем на 25% ниже, чем у аналогов. Результаты TBB показали, что `affinity_partitioner` в этом случае лидирует, среднее время выполнения у него на 24% ниже, чем у `auto_partitioner`. Эта разница в производительности может быть связана с устойчивым распределением итераций между запусками в `affinity_partitioner`, что благоприятно влияет на переиспользование данных из кеша, как указано в документации oneTBB [3].

Алгоритм, разработанный в рамках работы, стабильно уступает по производительности только `OMP_static`, а также немного уступил на больших размерах задач `affinity_partitioner` из OneTBB. В целом, это ожидаемый результат — статические подходы оказываются лучше за счёт меньших накладных расходов, но при этом предложенный алгоритм показывает лучшую производительность, чем классические динамические подходы.

3.3.1.2. Несбалансированная матрица

В отличие от варианта со сбалансированной нагрузкой, в этом случае динамический подход к планированию выполнения задач оказался более эффек-

тивным, чем статический. На бенчмарке с сильно несбалансированной матрицей (`spmv_hyperbolic`) из аналогов наибольшая эффективность оказалась у `TBB Simple`, который делит диапазон на задачи по одной итерации, а на бенчмарках с треугольной матрицей — `affinity_partitioner` из `TBB` (и `OMP_static` на задачах меньшего размера).

Можно заметить, что в случае гиперболически изменяющегося распределения элементов матрицы разница в производительности между динамической и статической стратегиями планирования более заметна, чем в случае треугольной матрицы — это объясняется тем, что в треугольной матрице для задач меньшего размера накладные расходы куда больше влияют на время выполнения бенчмарка, из-за чего даже на такой несбалансированной нагрузке `OMP Static` может показывать удовлетворительный результат.

Предложенный в работе алгоритм показывают наилучшую производительность на бенчмарке `spmv_hyperbolic`, кроме бенчмарка со строками матрицы наименьшего размера — там за счёт этого преимущество оказывается у `TBB`. При этом заметно, что у статического `OMP_static` и у `TBB_affinity` производительность деградирует наиболее заметно, а ближайшим к разработанному алгоритму оказывается `TBB_simple`, который в разы уступал на сбалансированной нагрузке.

3.3.2. Reduce

Результаты бенчмарка `Reduce` представлены на рисунке 15. Результаты ожидаемо похожи на сбалансированную версию бенчмарка `SPMV` — лучшие результаты у `OMP Static` и `TBB Affinity`, что обоснованно для сбалансированной нагрузки, но внезапно схожую производительностью показывает и `OpenMP Dynamic`. Разработанное решение оказалось в разы лучше чем `TBB Auto` и `TBB Simple`, с которым сравнивалось на несбалансированной нагрузке. На системе с `ARM` (рисунок 16) результаты аналогичные.

При этом на обеих системах разброс в результатах куда ниже, чем на других бенчмарках за счёт того, что задачи в этом бенчмарке выполняются дольше и накладные расходы на этом фоне менее заметны.

3.3.3. Scan

Результаты бенчмарка `Scan` представлены на рисунке 17. По оси `X` отмечен логарифм от размера массива (размер изменялся от 2^{10} до 2^{24}), а по оси

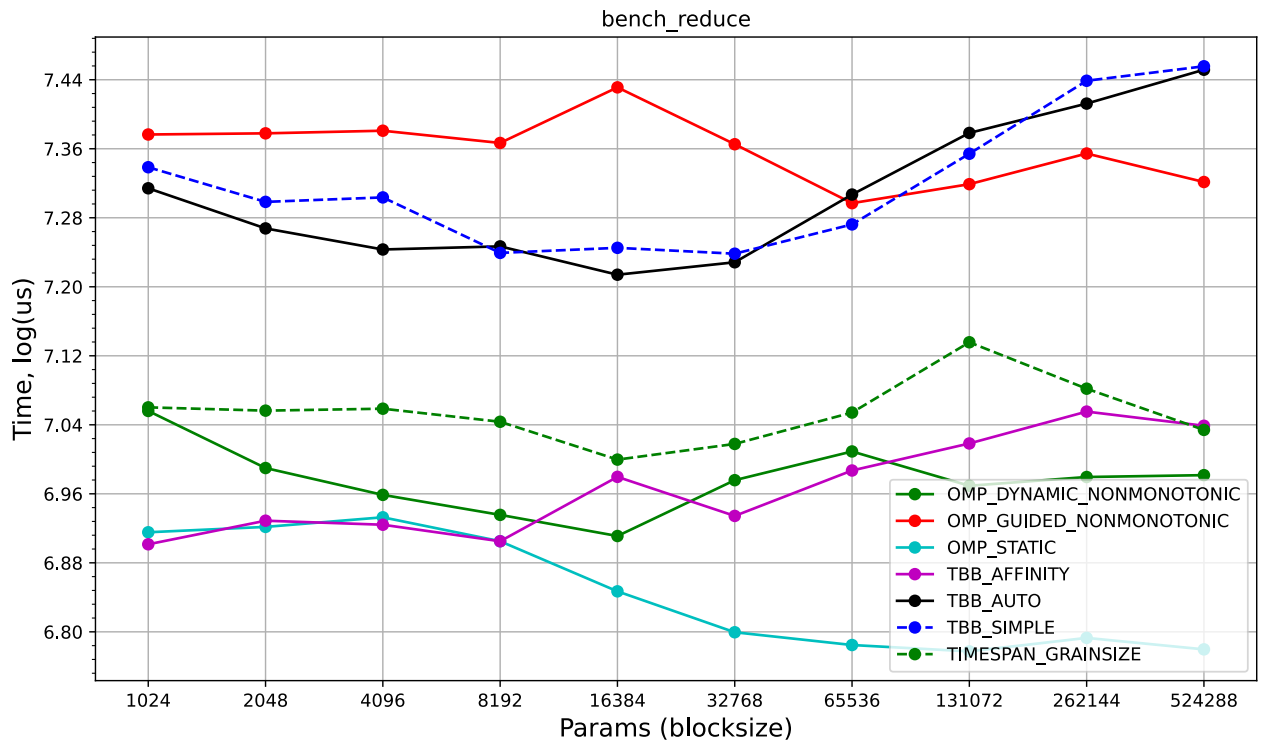


Рисунок 15 – Результаты бенчмарка Reduce, система с x86

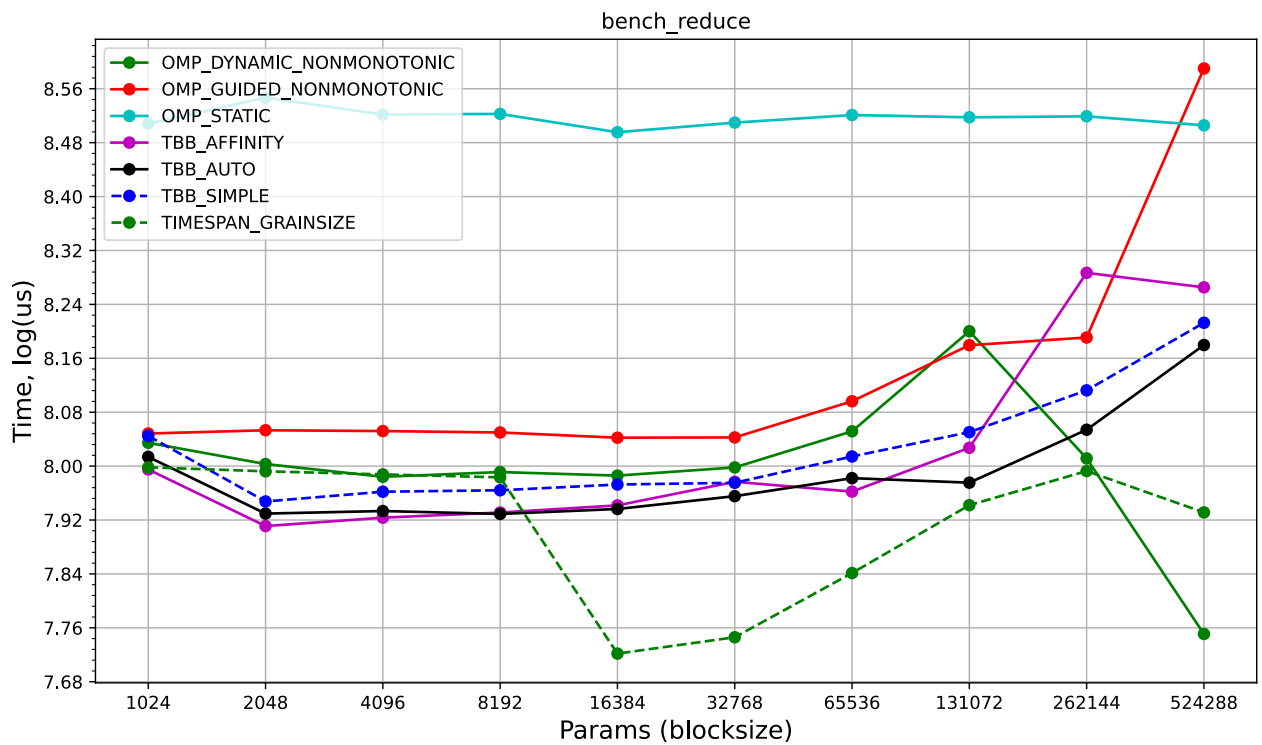


Рисунок 16 – Результаты бенчмарка Reduce, система с ARM

У — время на выполнение бенчмарка в микросекундах по логарифмической шкале.

Динамические конфигурации в OpenMP и TBB показывают самую низкую эффективность. Статическая политика планирования из OpenMP оказалась наиболее эффективной, демонстрируя почти в 2 лучшую производительность по сравнению с другими вариантами. Разработанный в работе алгоритм оказался не хуже большинства аналогов, уступает только OpenMP static и на большем размере массива сравним с TBB affinity. Это может быть связано с накладными расходами на начальное распределение задач и захват работы при малых размерах задач. Подтверждение этому видно на бенчмарке времени распределения работы (рисунок 23) — полученный алгоритм распределяет работу быстрее, чем алгоритмы из OneTBB, но медленнее реализации из OpenMP, и приблизительно такой же порядок виден в результате этого бенчмарка.

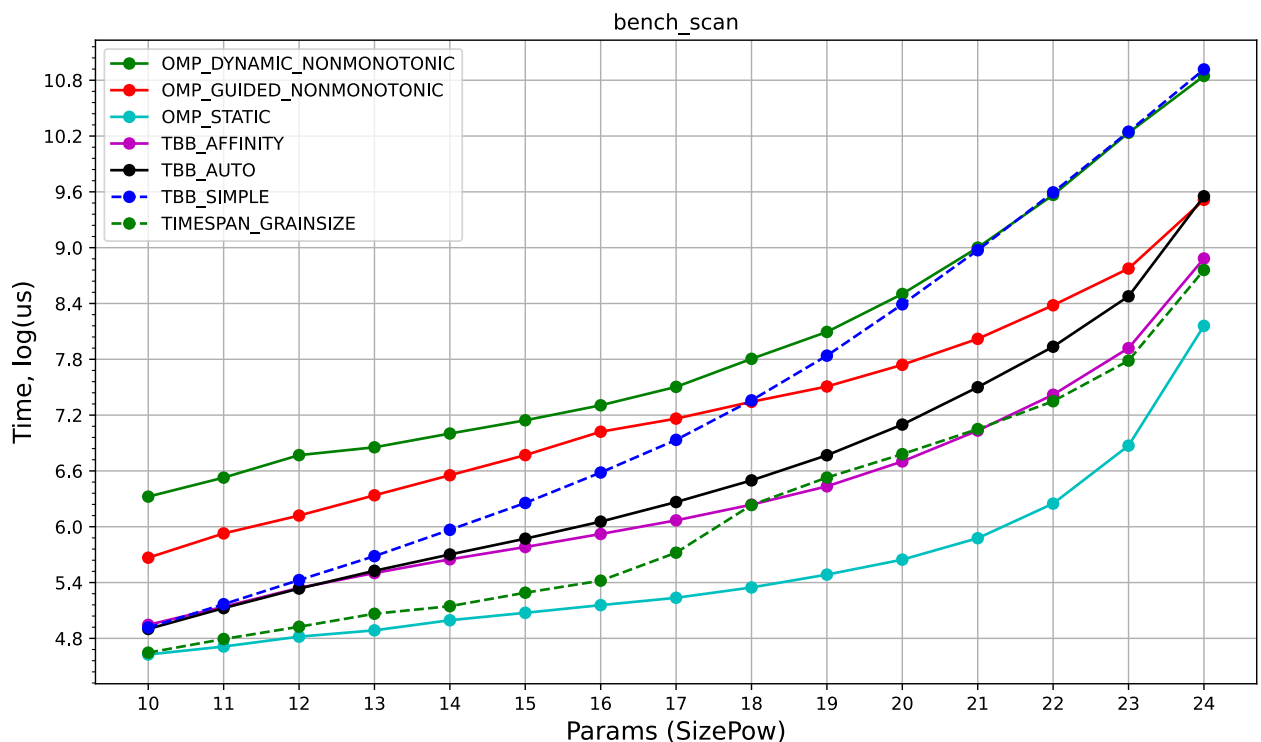


Рисунок 17 – Результаты бенчмарка Scan, система с x86

На системе с ARM (рисунок 18) результаты аналогичные, но как и с бенчмарком SPMV заметнее отрыв разработанного решения от аналогов.

3.3.4. Вложенный параллелизм

По результатам этих бенчмарков, представленных на рисунках 19 и 22, особенно заметно, что OpenMP не поддерживает вложенный параллелизм — производительность деградирует по сравнению с OneTBB и предложенным

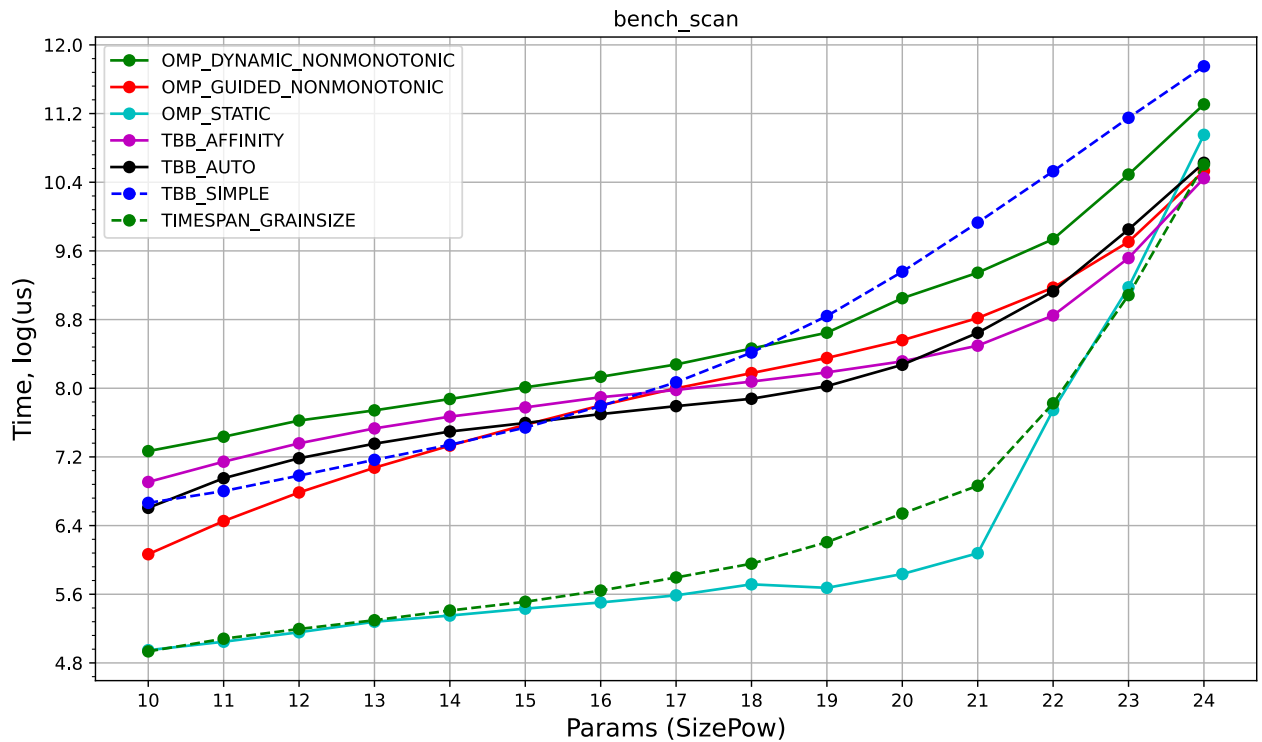


Рисунок 18 – Результаты бенчмарка Scan, система с ARM

алгоритмом на порядок, что объясняется как раз тем, что OpenMP создаёт новые потоки на каждый вложенный вызов. При этом разрыв в транспонировании матриц в разы сильнее из-за меньшего размера задач, на фоне которого больше выделяются накладные расходы на создание потоков при вложенности. Разработанный алгоритм `TIMESPAN_GRAINSIZE` показывает совместимость со вложенным параллелизмом, хоть и уступает решениям из TBB, но не на порядок, как это происходит с OpenMP.

Аналогичные результаты получились и на системе с архитектурой ARM (рисунки 21 и ??).

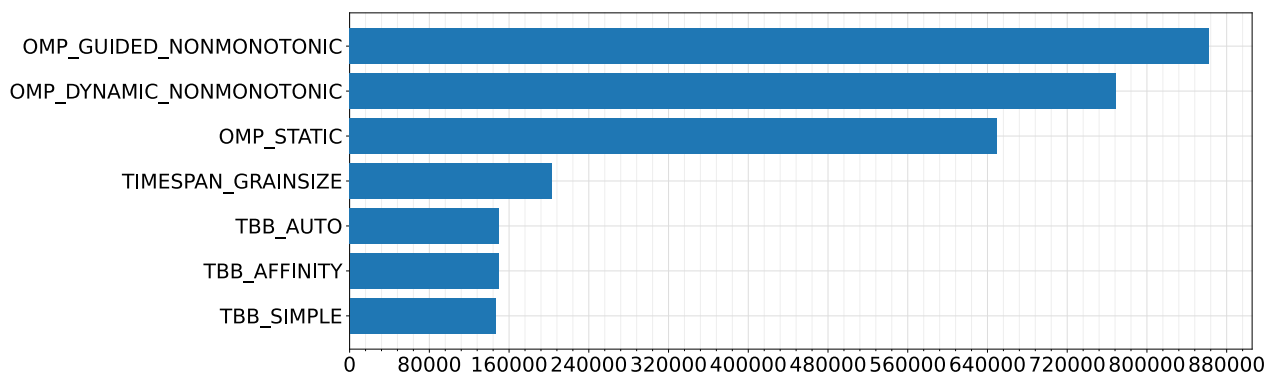


Рисунок 19 – Результаты бенчмарка MMultiply, система с x86

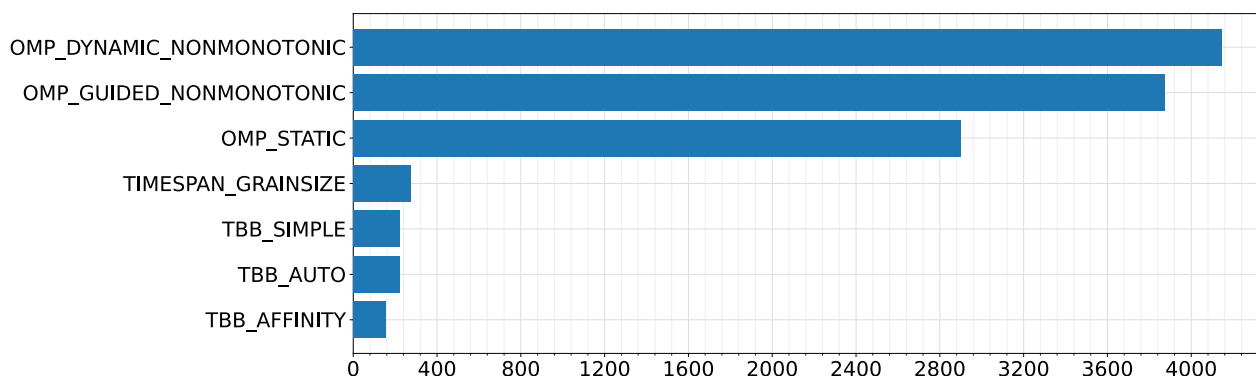


Рисунок 20 – Результаты бенчмарка MTranspose, система с x86

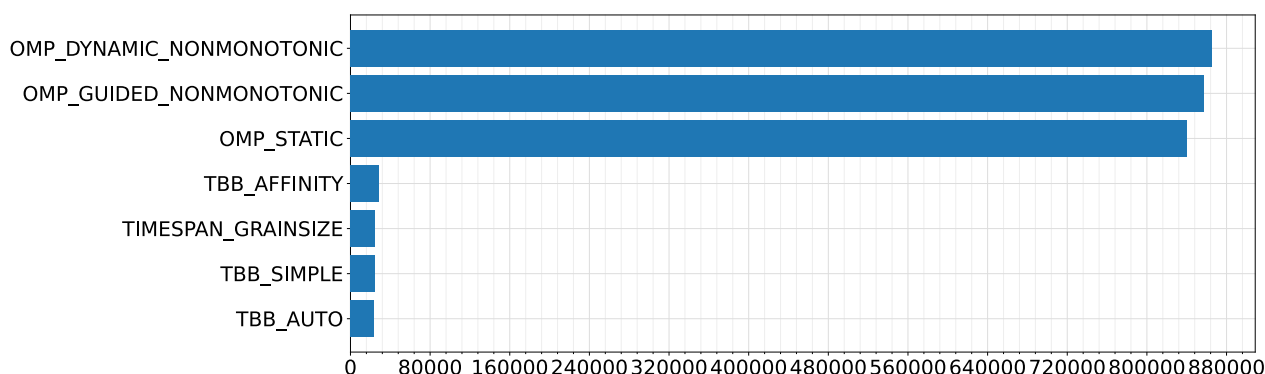


Рисунок 21 – Результаты бенчмарка MMultiply, система с ARM

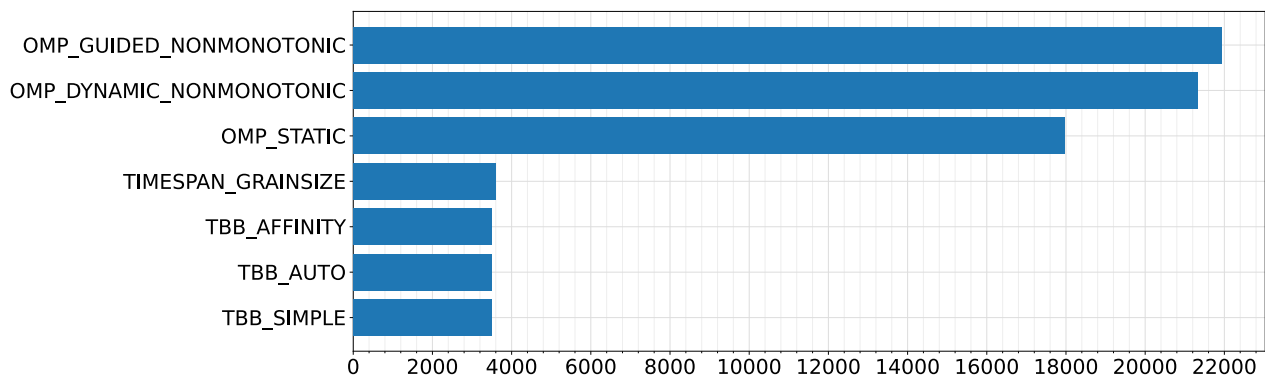


Рисунок 22 – Результаты бенчмарка MTranspose, система с ARM

3.3.5. Время распределения задач

В этом бенчмарке сравнивается время, необходимое планировщику для распределения работы между потоками. Результаты представлены на рисунке 23. На оси X показаны номера потоков (отсортированных по времени выполнения первой задачи), а на оси Y — время, затраченное на планирование задачи (в циклах).

Результаты показали, что стратегия статического планирования в OpenMP является наиболее эффективной. Это ожидаемый результат, так как

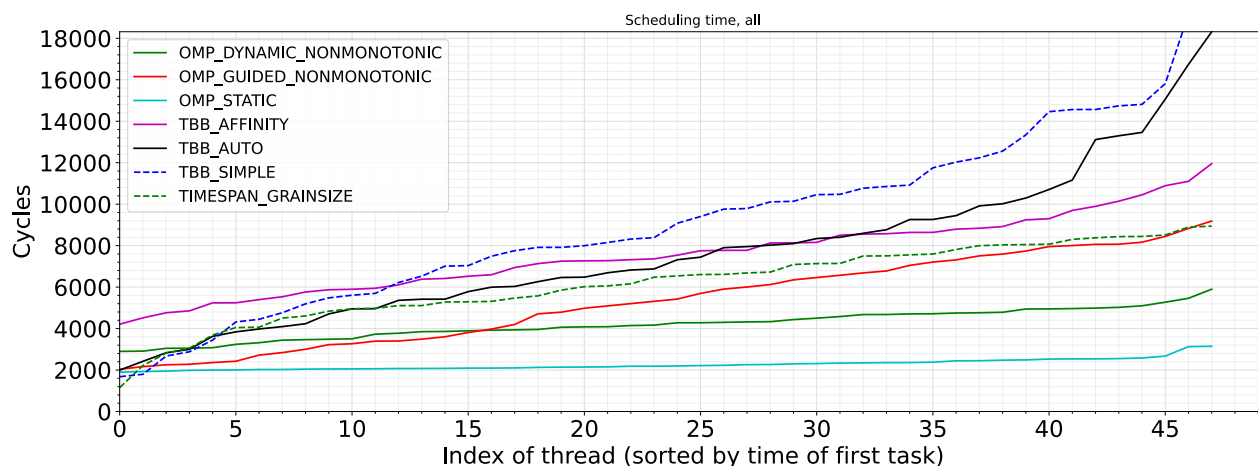


Рисунок 23 – Время от вызова `parallel_for` до начала выполнения первой задачи, система с x86

статическое планирование требует только деления работы между потоками, в то время как динамическое планирование требует дополнительной синхронизации потоков при работе с очередями задач. Кроме того, среднее время планирования оставалось примерно равным для всех потоков.

Почти все разделители из ТВВ имели похожую друг на друга производительность, при этом `affinity_partitioner` оказался наиболее эффективным. Однако время распределения задач между потоками в ТВВ неодинаково для всех потоков. В частности, последние потоки имели значительно более высокое время планирования, чем первые – это может происходить из-за накладных расходов на захват работы, так как последним потокам нужно проверять больше очередей других потоков при захвате задач.

Разработанный алгоритм оказался быстрее аналогов из ТВВ, но ожидается не лучше чем OpenMP.

Похожие тренды видны на результате бенчмарка на системе с архитектурой ARM (рисунок 24).

Выводы по главе 3

В этой главе было приведено описание разработанных бенчмарков, а также результаты запуска на этих бенчмарках популярных аналогов решаемой задачи и предложенного в работе решения.

В целом, различные конфигурации OneTBB и OpenMP показывают ожидаемые результаты на разных по характеристикам бенчмарках. Так, статический вариант из OpenMP (`OMP_STATIC`) имеет наилучшую производительность на бенчмарках с равномерно распределенной нагрузкой

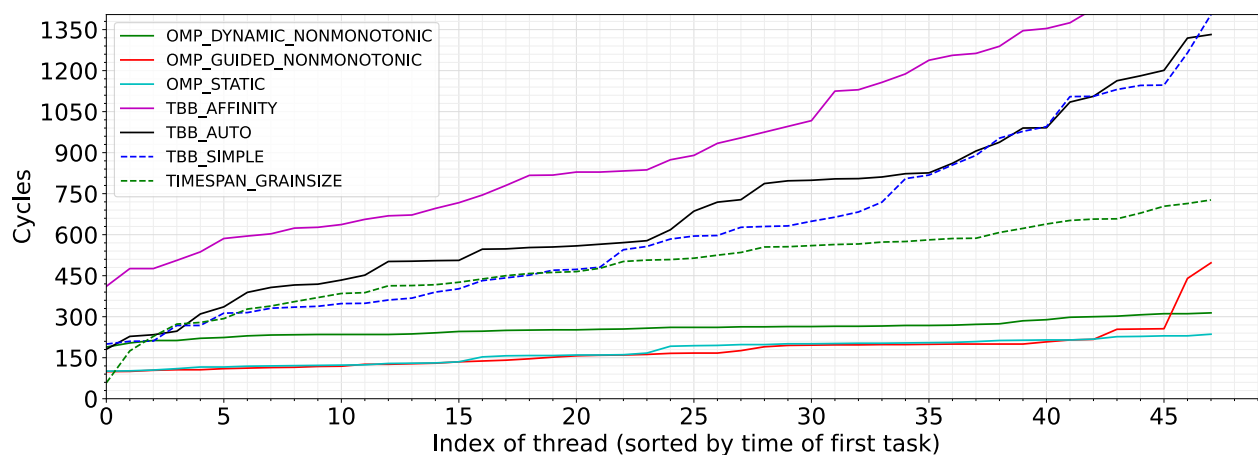


Рисунок 24 – Время от вызова `parallel_for` до начала выполнения первой задачи, система с ARM

(`spmv_balanced`, `reduce`) и на бенчмарке, требовательном к времени старта и завершения задач (`scan`), но при этом отсутствие балансировки нагрузки проявляется на бенчмарках с несбалансированной нагрузкой. Динамический подход к распределению задач, как в TBB, что в OpenMP показывает неплохие результаты на несбалансированной нагрузке и ожидаемо проигрывает статическому подходу на сбалансированной, при этом реализация из OpenMP заметно уступает OneTBB.

Как итог, ни один из аналогов нельзя назвать универсальным — при непредсказуемом паттерне нагрузки, который не известен заранее, производительность любого из них может заметно деградировать, так как разные парадигмы (статическая и динамическая) подходят для разных сценариев нагрузки.

При этом представленное в работе решение показывает наилучшую производительность на бенчмарке с несбалансированной нагрузкой (`spmv_hyperbolic`) и ожидаемо уступает на бенчмарках со сбалансированной нагрузкой только статическим подходам к распределению работы. Кроме того, решение совместимо со вложенным параллелизмом, хоть и уступает на нём OneTBB, но в разы превосходит OpenMP.

ЗАКЛЮЧЕНИЕ

В рамках работы был разработан алгоритм распределения работы, порожденной алгоритмом `parallel_for`, для компонуемого планировщика задач. Разработанный алгоритм фокусируется на оптимизации двух фаз распределения работы — оптимальном равномерном начальном распределении, а также на последующем эффективном создании балансировочных задач для планировщика потоков. Для реализации равномерного распределения было применено отложенное создание балансировочных задач, а для эффективного выбора размера — адаптивный подсчет параметра, задающего гранулярность задач при балансировке.

Для оценки производительности алгоритма был разработан комплекс бенчмарков с различным характером нагрузки, позволяющих сравнить получившееся решение с аналогами.

В ходе сравнения было выявлено, что реализованный алгоритм на сбалансированной нагрузке уступает только простому OpenMP Static, на несбалансированной — лучше чем аналоги из OneTBB и OpenMP. При этом решение компоуемо и поддерживает вложенный параллелизм, показывая в бенчмарках с ним производительность сравнимую с OneTBB, что в разы превосходит OpenMP.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 *Blumofe R. D., Leiserson C. E.* Scheduling multithreaded computations by work stealing // Journal of the ACM (JACM). — 1999. — Т. 46, № 5. — С. 720–748.
- 2 *Sutter H.* The free lunch is over: A fundamental turn toward concurrency in software // Dr. Dobbs's journal. — 2005. — Т. 30, № 3. — С. 202–210.
- 3 Algorithms — oneAPI Specification 1.2-rev-1 documentation [Электронный ресурс]. — 2023. — URL: <https://spec.oneapi.io/versions/latest/elements/oneTBB/source/algorithms.html#partitioners>.
- 4 Arm Armv8-A Documentation: Architecture Registers [Электронный ресурс]. — 2021. — URL: <https://developer.arm.com/documentation/ddi0595/2021-03/AArch64-Registers/CNTVCT-EL0--Counter-timer-Virtual-Count-register>.
- 5 Chapter 39. Parallel Prefix Sum (Scan) with CUDA | NVIDIA Developer [Электронный ресурс]. — 2023. — URL: <https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda>.
- 6 Eigen C++ Library, Non Blocking Thread Pool [Электронный ресурс]. — 2023. — URL: <https://gitlab.com/libeigen/eigen/-/blob/master/Eigen/src/ThreadPool/NonBlockingThreadPool.h>.
- 7 Google Benchmark User Guide, Runtime and Reporting Considerations [Электронный ресурс]. — 2023. — URL: https://github.com/google/benchmark/blob/main/docs/user_guide.md#runtime-and-reporting-considerations.
- 8 *Korndörfer J. H. M., Ahmed Eleliemy A. M., Ciorba F. M.* LB4OMP: A Dynamic Load Balancing Library for Multithreaded Applications. — 2021. — URL: <https://arxiv.org/abs/2106.05108>.
- 9 OpenMP Application Programming Interface Specification Version 5.2 [Электронный ресурс]. — 2021. — URL: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf>.

- 10 RDTSC: Read Time-Stamp Counter (x86 Instruction Set Reference) [Электронный ресурс]. — 2021. — URL: https://c9x.me/x86/html/file_module_x86_id_278.html.
- 11 sched_setaffinity(2) — Linux manual page [Электронный ресурс]. — 2021. — URL: https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html.
- 12 Task-Based Programming — oneTBB documentation [Электронный ресурс]. — 2023. — URL: https://oneapi-src.github.io/oneTBB/main/tbb_userguide/Task-Based_Programming.html.
- 13 *Voss M., Asenjo R., Reinders J.* Creating Thread-to-Core and Task-to-Thread Affinity, Pro TBB Chapter [Электронный ресурс]. — 2019. — URL: https://link.springer.com/chapter/10.1007/978-1-4842-4398-5_13.
- 14 *Voss M., Asenjo R., Reinders J.* The Pillars of Composability , Pro TBB Chapter [Электронный ресурс]. — 2019. — URL: https://link.springer.com/chapter/10.1007/978-1-4842-4398-5_9.