МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Тверской государственный технический университет»

(ТвГТУ)

Кафедра “Программного обеспечения”

**Лабораторная работа №2**

по дисциплине «Параллельные вычисления»

Тема: «Определение количества потоков CPU»

Выполнил: студент группы

Б.ПИН.РИС 21.06

Пономарев В.С.

Проверил:

Желтов С.А.

Тверь 2024

Оглавление

[Цель работы 3](#_Toc180870075)

[Задачи 4](#_Toc180870076)

[Создание приложения 5](#_Toc180870077)

[Последовательный алгоритм 5](#_Toc180870078)

[Пирамидальный алгоритм 6](#_Toc180870079)

[Кусочный алгоритм 6](#_Toc180870080)

[Определение фактического времени выполнения программы 6](#_Toc180870081)

[Анализ 7](#_Toc180870082)

[Вывод 9](#_Toc180870083)

[Источники 9](#_Toc180870084)

# Цель работы

Целью лабораторной работы является определить оптимальное количество потоков CPU в вычислительной системе при организации параллельных вычислений.

# Задачи

1. Создать многопоточное приложение.
2. Реализовать последовательный алгоритм вычисления суммы элементов массива целых чисел.
3. Описать и реализовать параллельный алгоритм вычисления суммы элементов массива целых чисел.
4. Определить время фактического выполнения программы для разных наборов входных данных и различного количества потоков.
5. Провести анализ зависимости времени фактического выполнения программы от количества потоков и размерности массива.

# Создание приложения

Для выполнения данной работы было разработано консольное приложение на языке программирования python версии 3.12.1 в IDE Visual Studio Code. Необходимые библиотеки: numpy, matplotlib и их зависимости. Основные характеристики системы на которой выполнялся код:

OC: Windows 10

CPU: AMD Ryzen 7 7700

RAM: DDR5 32Gb 6000MHz

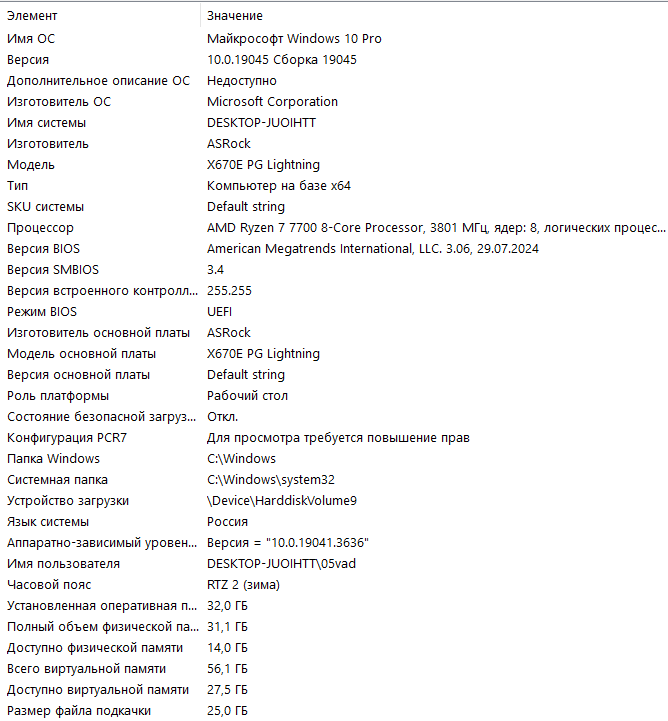


Рисунок 1 – Аппаратное обеспечение

# Последовательный алгоритм

Для получения суммы всех элементов в массиве, итерируемся по нему и складываем каждое значение с предыдущим.

Данный алгоритм реализован в функции sum\_array [[1](#first)] принимающей массив на вход и возвращающей сумму элементов этого массива.

# Пирамидальный алгоритм

Пирамидальный алгоритм [[2](#second)] заключается в распараллеливании процесса суммирования, для этого мы складываем пары элементов массива и на каждом последующем шаге выполняем n/2 кол-во параллельных сложений.

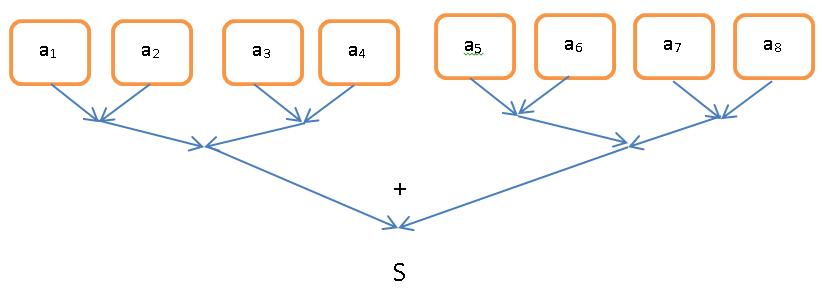


Рисунок 2 – Представление пирамидального алгоритма

Этот алгоритм состоит из двух частей: выяснение что суммируем (номер числа в массиве) и само суммирование.

Для реализации этого алгоритма была написана функция parallel\_sum\_array\_piramid [[1](#first)] принимающая на вход массив и опционально, кол-во потоков, если не указать это значение, то будет выбрано максимальное количество. Функция возвращает сумму всех элементов массива.

# Кусочный алгоритм

Данный алгоритм предполагает деление массива на части и распараллеливание суммирования этих частей. В программе этот алгоритм реализован в функции parallel\_sum\_array\_batch [[1](#first)] принимающей на вход массив и опционально, кол-во потоков. Функция возвращает сумму всех элементов массива.

## Определение фактического времени выполнения программы

Для вычисления времени выполнения программы был написан вспомогательный метод task [[1](#first)], принимающий на вход размер массива (который заполняется случайными числами) и кол-во потоков. Он запускает 3 алгоритма подряд: последовательный, пирамидальный и кусочный, и печатает время выполнения.

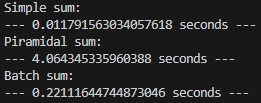


Рисунок 3 – Результат вызова функции на массиве размером 300000

## Анализ

При тестировании алгоритмов было замечено, что пирамидальный алгоритм в разы медленнее остальных. Хоть операции сложения и происходят “одновременно” этого сэкономленного времени недостаточно что бы покрыть затраты на этапы алгоритма предшествующие суммированию.

Выясним эффективна ли кусочная функция по сравнению с последовательной и есть ли зависимость скорости вычисления от кол-ва потоков.

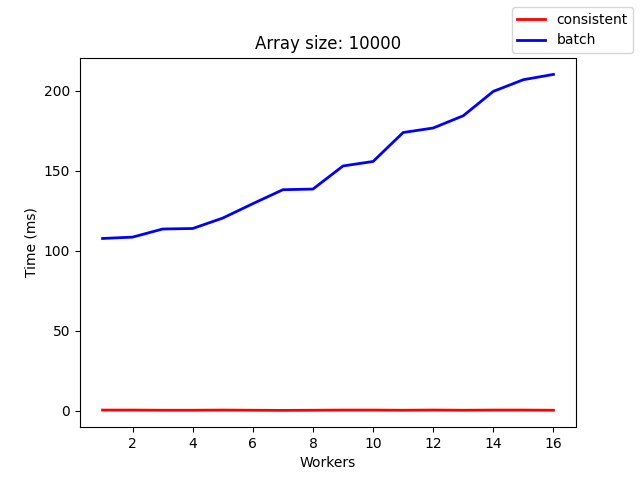


Рисунок 4 – График зависимости времени выполнения от кол-ва потоков на массиве размером 10000



Рисунок 5 – График зависимости времени выполнения от кол-ва потоков на массиве размером 1000000

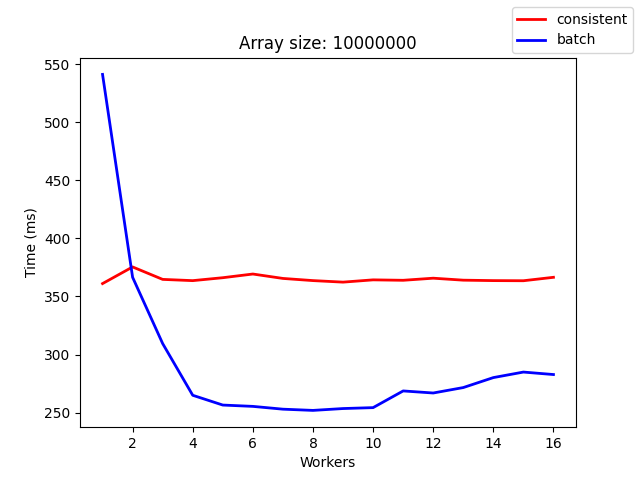


Рисунок 6 – График зависимости времени выполнения от кол-ва потоков на массиве размером 10000000

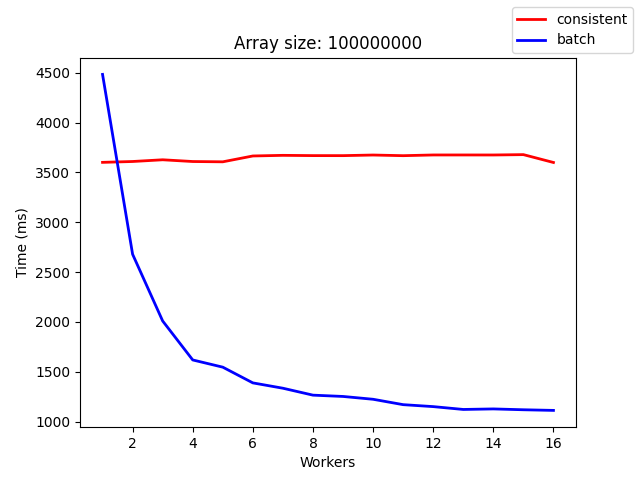


Рисунок 7 – График зависимости времени выполнения от кол-ва потоков на массиве размером 10000000

Как можно заметить на графиках, представленных выше, кусочный алгоритм выигрывает по времени только на большом объеме данных, так же заметно, что оптимальное кол-во потоков увеличивается с объемом.

## Вывод

Мною были реализованы 3 алгоритма нахождения суммы элементов массива: последовательный, пирамидальный и кусочный. Пирамидальный алгоритм продемонстрировал самую низкую эффективность, кусочный алгоритм дробящий массив на части (меньшие или равные кол-ву потоков процессора) и вычисляющий их параллельно эффективнее пирамидального т.к. не производятся лишние операции по созданию потоков (т.к. массив огромен, то кол-во операций сложения значительно превышает имеющееся кол-во потоков процессора и из-за этого они все-равно выполняются “последовательно”), и вся скорость выполнения уходит на последовательное сложение в потоках. Также была выявлена зависимость, что при увеличении размера массива и кол-ва потоков кусочный алгоритм становится эффективнее последовательного.

## Источники

1. Код программы на GitHub

URL: <https://github.com/vadimyt/Multi-processing>

1. Академия Microsoft: Параллельные вычисления и многопоточное программирование

URL: <https://intuit.ru/studies/courses/10554/1092/lecture/27091?page=2>