

**Министерство науки и высшего образования Российской  
Федерации ФГАОУ ВО «УрФУ имени первого Президента России  
Б.Н. Ельцина»**

Отчет по дисциплине:

## **«Компьютерные науки»**

**Студентка:** Кандрина Анна Дмитриевна

**Группа:** МЕНМ-140112 (МГМТ-1)

**Преподаватель:** Мисилов Владимир Евгеньевич к.ф.-м.н

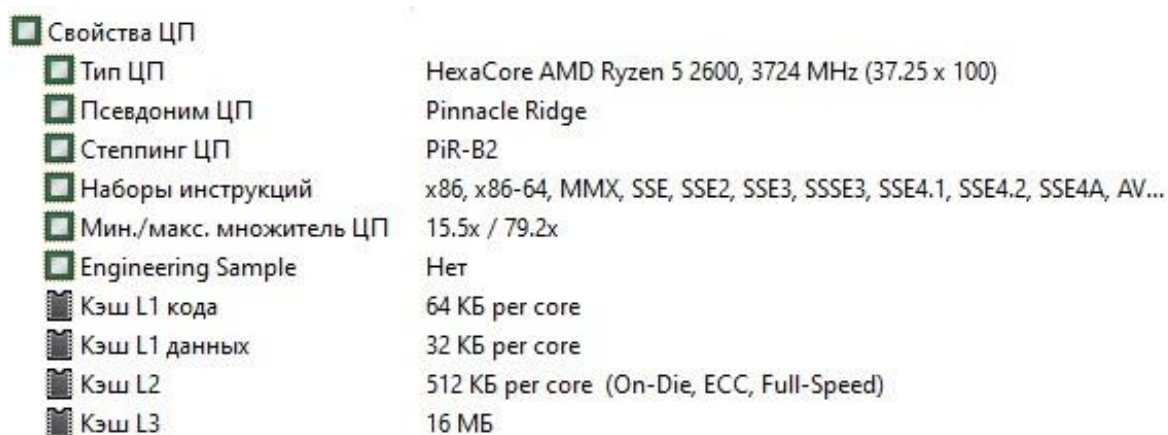
Екатеринбург 2025

## Оглавление

Справка.....	3
Задача 1. Обедаящие философы .....	4
Задача 2. OpenMP .....	8
Задача 3: MPI .....	10
Заключение.....	13

## Справка

Все представленные ниже замеры скорости выполнения работы программ и другие расчёты производились в системе со следующей конфигурацией см. Рис. 1 и Рис 2.



Свойства ЦП	
Тип ЦП	HexaCore AMD Ryzen 5 2600, 3724 MHz (37.25 x 100)
Псевдоним ЦП	Pinnacle Ridge
Степпинг ЦП	PiR-B2
Наборы инструкций	x86, x86-64, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A, AV...
Мин./макс. множитель ЦП	15.5x / 79.2x
Engineering Sample	Нет
Кэш L1 кода	64 КБ per core
Кэш L1 данных	32 КБ per core
Кэш L2	512 КБ per core (On-Die, ECC, Full-Speed)
Кэш L3	16 МБ

Рис. 1. Конфигурация ЦП.



Тип корпуса	1331 Pin uOPGA
Размеры корпуса	40 mm x 40 mm
Технологический процесс	13M, 12 nm CMOS, Cu
Напряжение питания ядра	0.825 - 1.137 V
Типичная мощность	65 W

Рис. 2. Физическая информация о ЦП.

Основные сведения об операционной системе представлены ниже (см. Рис. 3).



Выпуск	Windows 10 Домашняя
Версия	22H2
Дата установки	20.04.2021
Сборка ОС	19045.3086
Взаимодействие	Windows Feature Experience Pack 1000.19041.1000.0

Рис. 3. Характеристики Windows.

## Задача 1. Обедающие философы

### Требования:

#### Необходимо обеспечить:

1. Защиту от взаимоблокировок (deadlocks)
2. Защиту от динамических взаимоблокировок (livelock)
3. Справедливость – все философы едят примерно одинаковое количество времени

#### Бонус:

1. Масштабируемость (число философов может быть большим)

#### Ход решения:

Программа написана на языке C#. Использовалась стандартная библиотека System.Threading для работы с потоками. В качестве механизма синхронизации потоков задействован класс Mutex, собственно он и обеспечивает одновременное выполнение несколькими потоками одного и того же участка кода.

В начальной реализации возникала ситуация взаимоблокировки (deadlock), когда все философы одновременно пытаются взять левую вилку и ожидают освобождения правой вилки. Это происходит потому, что каждый философ пытается захватить вилки в порядке слева-направо, и если все философы одновременно возьмут левую вилку, они будут ожидать, пока правая вилка освободится, чего не произойдет.

Для того чтобы избежать такой ситуации, можно использовать разные стратегии. Одна из возможных стратегий, принятая здесь, это изменение порядка взятия вилок у одного из философов. Например, можно изменить порядок у первого из философов с **слева-направо** на **право-налево**: сначала

брать правую вилку, а затем левую. Это изменение порядка поможет предотвратить циклическую зависимость между философами.

При достаточно большом ( $> 50$ ) количестве философов **нарушается принцип справедливости**. При запусках программы получалось так, что кто-то ест 34% от общего времени, а кто-то только 15%. В качестве решения данной проблемы и балансировки времени работы была принята следующая стратегия: ограничить время еды для каждого философа. Причём, ограничение сначала представляло из себя случайное целое число, такой подход себя не оправдал, поэтому длительность «принятия пищи» для каждого из философов стало задаваемым параметром. Эмпирически было выявлено наилучшее значение в 5 секунд (брались только целые секунды).

Ниже представлены таблицы с временем «приёма пищи» относительно общего времени работы программы философами при разных входных данных  $P$  и  $D$ , где  $P$  – количество философов, а  $D$  – ограниченная длительность приёма пищи. Данные значения указаны в скобках после номера таблицы (см. Табл. 1, Табл. 2, Табл. 3, Табл. 4)

Табл. 1 ( $P = 5, D = 5$ )

Номер философа	Процент времени от общего времени
1	27%
2	27%
3	26%
4	26%
5	27%

Табл. 2 ( $P = 5, D = 10$ )

Номер философа	Процент времени от общего времени
1	27%
2	27%

3	28%
4	27%
5	28%

Табл. 3 ( $P = 10, D = 5$ )

Номер философа	Процент времени от общего времени
1	28%
2	28%
3	26%
4	28%
5	27%
6	26%
7	28%
8	27%
9	28%
10	27%

Табл. 4 ( $P = 50, D = 5$ )

Номер философа	Процент времени от общего времени
1	26%
2	28%
3	26%
4	28%
5	26%
6	27%
7	28%
8	26%
9	26%
10	31%
11	25%
12	29%
13	26%
14	30%
15	28%
16	26%
17	28%
18	29%
19	29%
20	28%

21	26%
22	27%
23	28%
24	31%
25	28%
26	26%
27	27%
28	32%
29	25%
30	25%
31	26%
32	29%
33	28%
34	27%
35	29%
36	31%
37	25%
38	27%
39	28%
40	30%
41	26%
42	30%
43	28%
44	29%
45	27%
46	29%
47	28%
48	27%
49	26%
50	30%

Если проанализировать результаты Табл. 4, то можно посчитать средний процент (с учетом среднеквадратичного отклонения) приёма пищи  $P = 50$  философов:

$$0,28 \pm 0.01525$$

## Задача 2. OpenMP

**Постановка задачи:** Найти площадь многоугольника, заданного списком координат вершин.

**Необходимо осуществить:**

- Составить параллельную программу, решающую задачу с использованием OpenMP.
- Откомпилировать и запустить программу.
- Замерить времена решения с разным числом потоков (от 1 до сколько получится).
- Объяснить результаты

**Ход решения:**

Программа написана на C++. В качестве инструментария для использования функционала OpenMP была выбрана библиотека `omp.h`.

Достаточно долгое время программа выполнялась, по сути, в одном потоке. Проверка данного факта осуществлялась при помощи вызова метода `omp_get_thread_num()`. Была гипотеза о том, что скорость программы не растёт из-за достаточно малых по объёму данных для выполнения программы. Однако после настройки соответствующих зависимостей, всё заработало, как и предполагалось.

Число потоков для тестирования было от 1 до 6 (непонятно, правда, почему при увеличении количества потоков свыше четырех, производительность не растёт). Результаты тестирования представлены в Табл. 5.

Табл 5.

Количество потоков OpenMP	Время выполнения программы (в мс.)
1	5082
2	2677
3	1834
4	1202



5	1217
6	1218

При достаточно больших объемах для вычисления ускорение происходит, примерно в 4,22 раза.

В качестве выводов следует отметить:

1. OpenMP эффективен для подобного рода задач (правда ускорение происходит не в N раз, т.к. некоторые части кода не параллелятся)
2. Видимо, оптимальное число потоков примерно равно числу физических ядер.

### Задача 3: MPI

**Постановка задачи:** Реализуйте параллельную версию игры «Жизнь» с использованием MPI

**Требования:**

- Загрузка начальной конфигурации клеток из файла, выполнение заданного количества шагов эволюции, сохранение полученной конфигурации в файл
- Визуализация не требуется
- Размеры поля должны быть достаточно большими и продолжительность в ходах должна быть достаточно большая

**Ход решения:**

Программа написана на C++. В качестве инструментария использовалась библиотека mpi.h. После настройки окружения и библиотеки удалось добиться прироста производительности.

Данные расчётов величин приведены в следующей таблице (см. Табл.6).

Расчёт времени работы программы производился по следующей формуле:

$$\sum_{i=1}^N \frac{t_i}{N} \pm \sigma,$$

где  $t_i$  – время при  $i$ -ом замере,  $N$  – количество замеров (в данном случае  $N = 10$ ), а  $\sigma$  – среднеквадратичное отклонение, которое рассчитывается по следующей формуле:

$$\sigma = \sqrt{\frac{\sum_{i=1}^N (t_i - \sum_{j=1}^N t_j)^2}{N-1}}$$

Табл 6.

Количество процессов MPI	Время работы, мс
2	9650±67
3	6996±59
4	5322±62

При запуске программы с количеством процессов больших, чем 4 выдавалось следующее исключение об ограничении количества одновременно используемых процессов: **Invalid rank has value 4 but must be nonnegative and less than 4.**

Замеры в Табл. 6 велись при значении количества итераций (EPOCHS) перехода в следующее состояние клеточного автомата в 100 штук. Интересно было наблюдать зависимость возрастающей вычислительной сложности данной программы (при зафиксированном количестве процессов MPI):

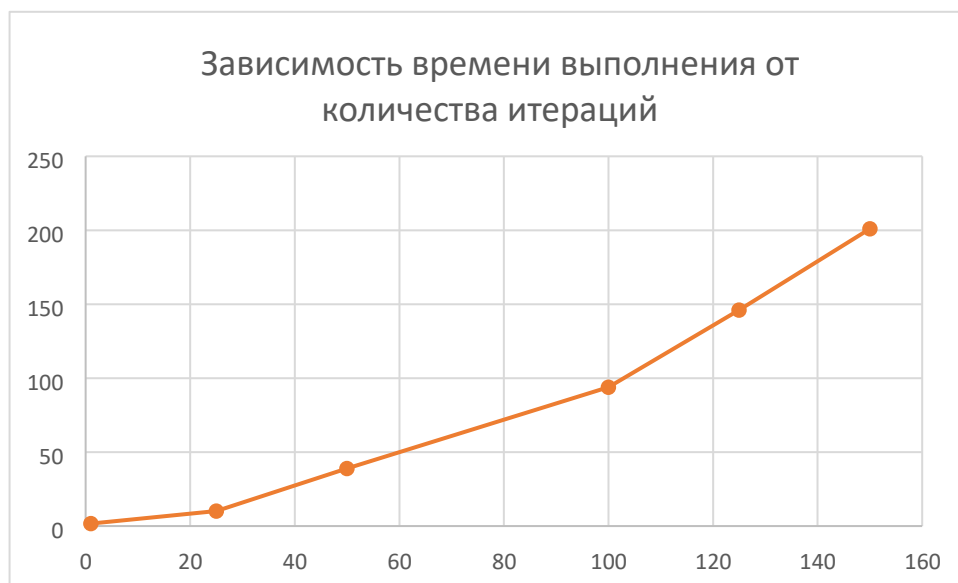


Рис. 4. Зависимость времени выполнения от количества итераций

Из Рис. 4 видно, что зависимость напоминает своим видом полиномиальную кривую: нелинейный, но плавный рост, а также нет резких скачков.

Возможно, что оптимизация алгоритма снизит вычислительную сложность задачи.

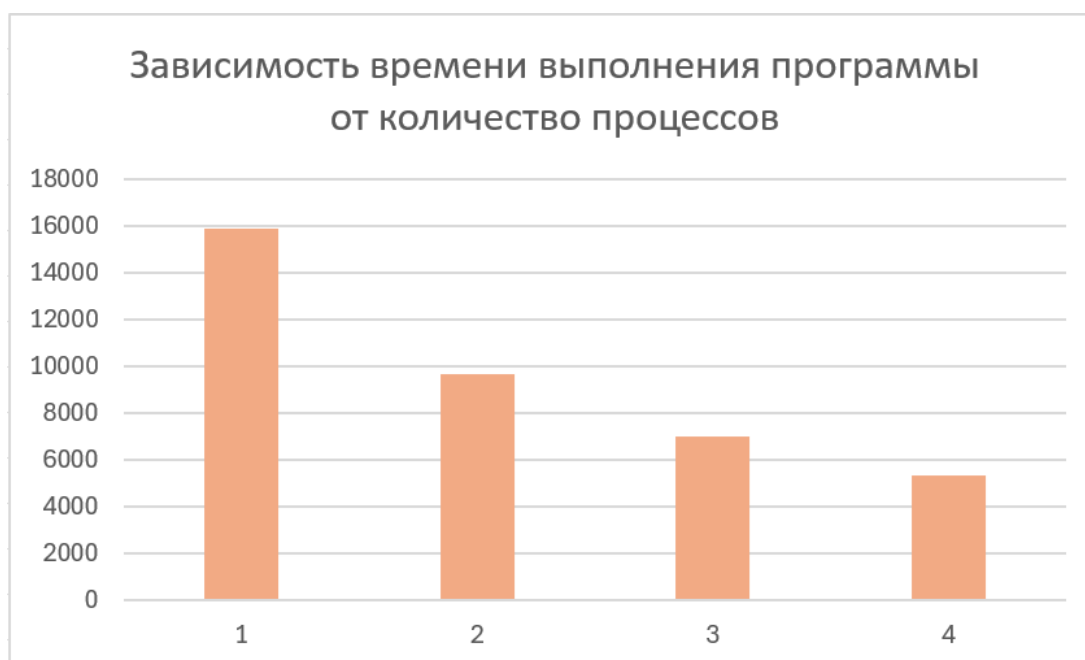


Рис 5. Зависимость времени выполнения программы от количество процессов

На рис.5 наглядно продемонстрированы результаты работы программы, представленные в Табл. 6.

## **Заключение**

Распараллеливание кода играет важную роль в современных вычислительных системах и программировании. Оно позволяет эффективно использовать ресурсы многоядерных процессоров, кластеров или суперкомпьютеров для выполнения задач параллельно.

Параллельные вычисления применяются для увеличения производительности, решения сложных и высоконагруженных задач, масштабируемости кода. Что напрямую влечёт актуальность параллельных вычислений в наши дни.