Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Архитектура вычислительных систем

*К защите допустить:*

И.О. Заведующего кафедрой информатики

\_\_\_\_\_\_\_\_\_\_\_\_\_С. И. Сиротко

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему

**ОПТИМИЗАЦИЯ ПРОИЗВОДИТЕЛЬНОСТИ ВЫЧИСЛЕНИЙ**

**НА МНОГОЯДЕРНЫХ ПРОЦЕССОРАХ С ПОМОЩЬЮ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ**

БГУИР КП 1-40 04 01 012 ПЗ

Студент Д. С. Кончик

Руководитель А. А. Калиновская

Нормоконтролер А. А. Калиновская

Минск 2023

СОДЕРЖАНИЕ

[Введение 5](#_Toc153177136)

[1 Архитектура вычислительной системы 6](#_Toc153177137)

[1.1 Платформы параллельных вычислений 6](#_Toc153177138)

[1.2 Параллельные вычисления в процессорах 8](#_Toc153177139)

[1.3 Разница между многоядерной архитектурой и гиперпоточностью 10](#_Toc153177140)

[1.4 Выбор вычислительной системы 12](#_Toc153177141)

[2 Платформа программного обеспечения 14](#_Toc153177142)

[2.1 Структура и архитектура платформы 14](#_Toc153177143)

[2.2 История, версии и достоинства 15](#_Toc153177144)

[2.3 Обоснование выбора платформы 17](#_Toc153177145)

[3 Теоретическое обоснование разработки программного продукта 18](#_Toc153177146)

[3.1 Обоснование необходимости разработки 18](#_Toc153177147)

[3.2 Технологии программирования, используемые для решения поставленных задач 19](#_Toc153177148)

[3.3 Связь архитектуры вычислительной системы с разрабатываемым программным обеспечением 23](#_Toc153177149)

[4 Проектирование функциональных возможностей программы 25](#_Toc153177150)

[4.1 Анализ алгоритмов 25](#_Toc153177151)

[4.2 Метод Гаусса 26](#_Toc153177152)

[4.3 Описание функциональной схемы программы 29](#_Toc153177153)

[4.4 Описание блок схемы алгоритма программы 29](#_Toc153177154)

[4.5 Реализация программы 30](#_Toc153177155)

[5 Сравнение производительности 33](#_Toc153177156)

[5.1 Понятие производительности 33](#_Toc153177157)

[5.2 Основные сведения о тестировании 35](#_Toc153177158)

[5.3 Результат работы программы 36](#_Toc153177159)

[5.4 Анализ результатов 37](#_Toc153177160)

[Заключение 39](#_Toc153177161)

[Список литературных источников 40](#_Toc153177162)

[Приложение А (обязательное) Листинг программного кода 42](#_Toc153177163)

[Приложение Б (обязательное) Функциональная схема алгоритма,  
 реализующего программное средство 45](#_Toc153177164)

[Приложение В (обязательное) Блок схема алгоритма, реализующего  
 программное средство 46](#_Toc153177165)

[Приложение Г (обязательное) Скриншоты сравнений производительности 47](#_Toc153177166)

[Приложение Д (обязательное) Ведомость документов 48](#_Toc153177167)

Введение

В современном мире вычислительные мощности имеют решающее значение в решении разнообразных задач, начиная от научных исследований и заканчивая бизнес-приложениями. Однако, с увеличением вычислительных мощностей в виде многоядерных процессоров, встает вопрос о том, как эффективно использовать их потенциал для максимизации производительности. В этом контексте, параллельное программирование становится ключевой стратегией для оптимизации вычислений на многоядерных системах.

Многоядерные процессоры стали стандартом для большинства современных компьютеров и серверов. Они предоставляют значительный потенциал для ускорения выполнения задач, однако этот потенциал не всегда реализуется в полной мере. Эффективное использование многоядерных процессоров требует грамотного параллельного программирования, что делает данную тему актуальной для исследования и практической реализации.

Целью данного курсового проекта является исследование и практическая реализация методов оптимизации производительности вычислений на многоядерных процессорах с использованием параллельного программирования.

Для достижения данной цели предусмотрены следующие задачи:

1. Выбор конкретной задачи или приложения для оптимизации, которая будет служить основой для исследования.
2. Разработка параллельной версии выбранной задачи с использованием соответствующих инструментов и технологии параллельного программирования.
3. Оценка производительности параллельной версии приложения и сравнение ее с последовательной версией для выявления улучшений.
4. Анализ результатов экспериментов с целью определения эффективности оптимизации и идентификации возможных узких мест.

Кроме того, в ходе курсового проекта будет проведен анализ архитектуры многоядерных процессоров, что позволит лучше понять и оценить возможности и ограничения использования параллельного программирования на данной аппаратной платформе.

Пояснительная записка оформлена в соответствии с  
СТП 01-2017 [1].

1. Архитектура вычислительной системы
   1. Платформы параллельных вычислений

Параллельность в программах – это способ организации выполнения задач таким образом, чтобы они могли выполняться одновременно. Этот подход направлен на более эффективное использование вычислительных ресурсов компьютера [2].

Чтобы обеспечить параллельную работу программ, аппаратная платформа должна поддерживать возможность одновременного выполнения нескольких программных потоков. В целом, компьютерные архитектуры могут быть классифицированы в двух разных измерениях.

Первое измерение – это количество потоков команд, которые данная компьютерная архитектура способна выполнить в единицу времени.

Второе измерение – это количество потоков данных, которые могут быть обработаны в единицу времени.

Таким образом, любая компьютерная система может быть описана в терминах обработки команд и данных. Эта система классификации известна как таксономия Флинна (*Flynn* 1972), и ее графическое представление показано на рисунке 1.1.

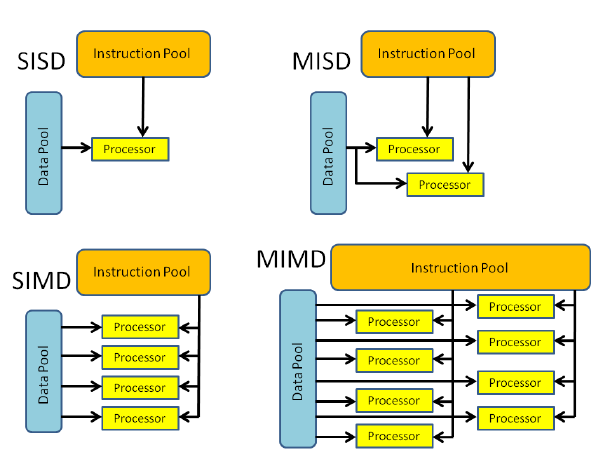


Рисунок 1.1 – Таксономия Флинна

Согласно таксономии Флинна любая компьютерная платформа попадает в одну из четырех категорий:

1. Компьютер с одним потоком команд и одним потоком данных   
   (*SISD, Single Instruction, Single Data,*) – это традиционный последовательный компьютер, который не обеспечивает параллельности в своей аппаратной части. Команды выполняются последовательно. Только один поток данных обрабатывается процессором в такте. Примеры таких платформ – это *IBM PC*, устаревшие мэйнфреймы и множество домашних 8-разрядных компьютеров, которые были популярны в начале 1980-х.
2. Компьютер с несколькими потоками команд и одним потоком данных

(*MISD, Multiple Instruction, Single Data*) способен обрабатывать один поток данных, одновременно используя несколько потоков команд. Однако в большинстве случаев нескольким потокам команд нужно несколько потоков данных (чтобы от них была польза), так что этот класс параллельных компьютеров обычно применяется лишь как теоретическая модель, а не как реальный компьютер массового производства.

1. В компьютере с одним потоком команд и несколькими потоками данных (*SIMD, Single Instruction, Multiple Data*) один поток команд одновременно может обрабатывать несколько потоков данных. Первоначально суперкомпьютеры были выполнены по этой архитектуре. Почти все компьютеры сегодня реализуют некую форму набора команд *SIMD*. Процессоры Intel реализуют наборы команд MMX, Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), Streaming SIMD Extensions 3 (SSE3), которые могут обрабатывать несколько элементов данных за один такт. Элементы данных хранятся в регистрах с плавающей точкой.
2. Компьютер с несколькими потоками команд и несколькими потоками данных (*MIMD, Multiple Instruction, Multiple Data*) способен выполнять несколько потоков команд при обработке каждого независимого потока данных. Сегодня это самый часто встречающийся тип параллельного компьютера, современные многоядерные платформы попадают именно в эту категорию.

Учитывая, что современные компьютеры относятся либо к категории *SIMD*, либо к категории *MIMD*, разработчики программ имеют возможность эксплуатировать в своем программном обеспечении параллелизм на уровне данных и на уровне заданий [3].

* 1. Параллельные вычисления в процессорах

Современные процессоры широко используют инструкционный параллелизм (*ILP, Instruction Level Parallelism*), что позволяет одновременно выполнять несколько инструкций программы внутри аппаратной архитектуры процессора. Механизм *ILP* достигается за счет выдачи, выполнения и завершения нескольких инструкций за один тактовый цикл, что позволяет увеличить производительность без необходимости вмешательства разработчика программного обеспечения [4].

Программный поток можно считать элементарной единицей использования процессора. Каждый поток содержит программный счетчик, который указывает на текущую инструкцию, выполняемую в контексте этого потока, информацию о состоянии процессора, включая значения регистров, свой собственный стек, который используется для управления вызовами подпрограмм, передачи параметров и хранения локальных переменных.

Физический процессор (рисунок 1.2) включает в себя различные компоненты, такие как регистры общего назначения, регистры контроллера прерываний, кэши, шины, исполнительные блоки и блоки предсказания ветвлений. Однако, чтобы описать программный поток, достаточно данных состояния архитектуры, включающих информацию о значениях в регистрах и других ключевых элементах. Логический процессор может быть создан путем дублирования области, занимаемой данными состояния архитектуры [5].

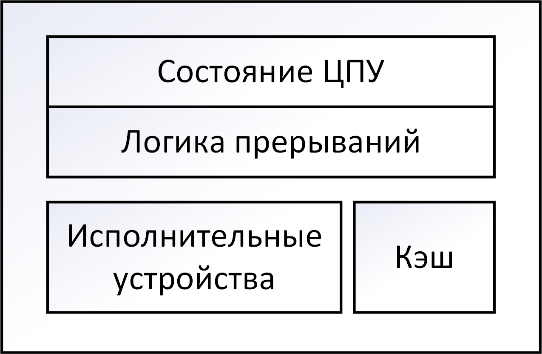


Рисунок 1.2 – Одноядерная архитектура

Затем исполнительные ресурсы распределяются между разными логическими процессорами. Этот подход известен как одновременная многопоточность (*Simultaneous Multi-Threading, SMT*). Реализация SMT от компании *Intel* известна как технология гиперпоточности (*Hyper-Threading Technology, HT Technology*) [6].

За счет технологии гиперпоточности (рисунок 1.3) один процессор выглядит (с точки зрения программного обеспечения) как несколько логических процессоров. Это позволяет операционным системам и приложениям планировать выполнение несколько программных потоков на разных логических процессорах точно так же, как в случае многопроцессорной системы.

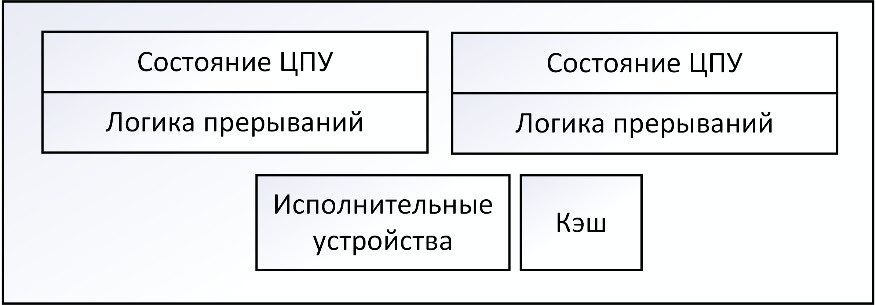


Рисунок 1.3 – Одноядерная архитектура с гиперпоточностью

С точки зрения микроархитектуры команды от логических процессоров поступают постоянно и выполняются одновременно на разделяемых исполнительных ресурсах. Другими словами, можно планировать несколько программных потоков, но поскольку исполнительные ресурсы используются совместно, то именно микроархитектура определяет, когда и как будет чередоваться выполнение двух потоков. Когда один программных поток останавливается, другому потоку разрешается продолжить выполнение. В число таких остановок входят обработка кэш-промахов и неверных предсказаний ветвлений.

Многоядерные процессоры (рисунок 1.4) поддерживают мультипроцессорную обработку на кристалле (*Chip Multiprocessing, CMP*) [7].

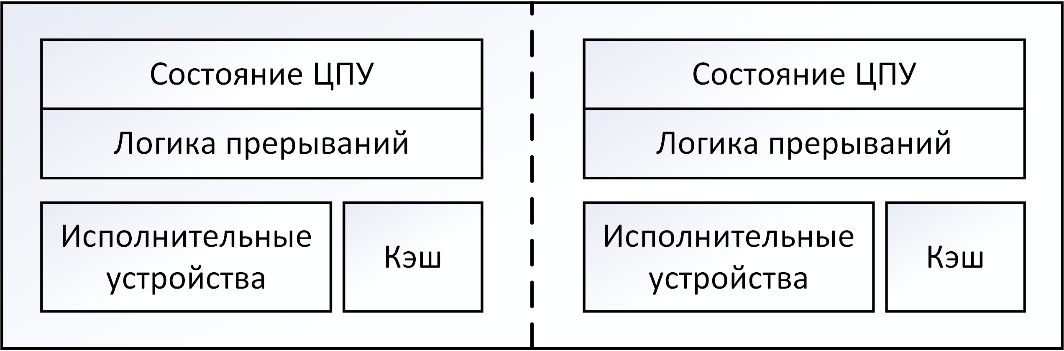


Рисунок 1.4 – Многоядерная архитектура

Вместо постоянного повторного использования ресурсов одноядерного процессора реализовано два или более исполнительных ядра в одном процессоре. Можно сказать, что эти ядра – отдельные процессоры на одном кристалле. Исполнительные ядра имеют собственный набор исполнительных и архитектурных ресурсов. В зависимости от конструкции эти процессоры могут совместно использовать (или не использовать) большой кэш на том же кристалле.

Кроме того, каждое ядро может задействовать механизм *SMT*(рисунок 1.5)*,* увеличивая таким образом количество логических процессоров в два раза по сравнению с количеством исполнительных ядер.

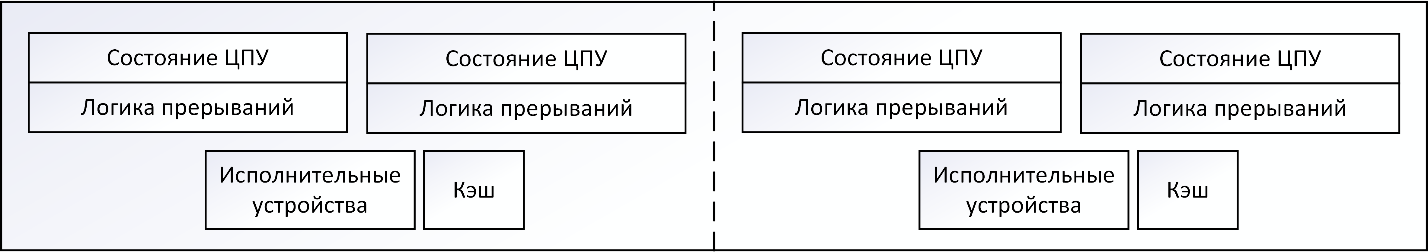


Рисунок 1.5 – Многоядерная архитектура с гиперпоточностью

* 1. Разница между многоядерной архитектурой и гиперпоточностью

Благодаря гиперпоточности некоторые части процессора разделяются между программными потоками, а другие части дублируются между ними. Один из самых важных разделяемых ресурсов – это исполнительная система. Она одновременно работает на оба программных потока путем выполнения команд одного потока на ресурсах, которые не заняты вторым потоком. Когда выполняются оба потока, механизм гиперпоточности буквально чередует команды в исполнительном конвейере. Какая именно команда и когда вставляется, целиком зависит от того, какие исполнительные ресурсы процессора свободны во время исполнения. Более того, если один программный поток занят чтением большого файла данных с диска или ждет от пользователя ввода с клавиатуры, то другой программный поток без переключения задач операционной системой захватывает все ресурсы процессора до того момента, как первый поток будет готов продолжить обработку. Таким образом каждый поток получает максимум доступных ресурсов, и процессор загружается настолько, насколько это возможно [8].

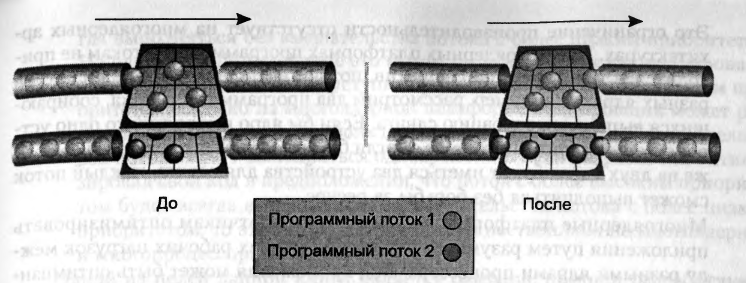
Гиперпоточность повышает производительность, скрывая задержки. По существу, единственное исполнительное ядро используется совместно несколькими программными потоками. То есть выполнение потоков реально не происходит параллельно. В результате рост производительности зависит от приложения и аппаратной платформы. При помощи технологии гиперпоточности в некоторых приложениях можно добиться повышения производительности процессора в среднем на 30 %. Другими словами, в некоторых случаях процессор может выполнять в 1.3 раза больше команд, чем он выполнил бы при наличии только одного потока. Выигрыш в производительности при использовании гиперпоточности зависит от того, как много задержек можно нейтрализовать в приложении.

В то же время многоядерные процессоры содержат два или более независимых исполнительных ядра в «одной упаковке». В результате каждая последовательность команд (программный поток) получает аппаратную среду выполнения в свое исключительное распоряжение. Это позволяет каждому потоку выполняться истинно параллельным образом.

Пример выполнения двух программных потоков на двухъядерном процессоре и на одноядерном с поддержкой *SMT* представлен на рисунке 1.6.



а



б

а – два программных потока выполняются одноядерным

процессором, поддерживающим технологию гиперпоточности;

б – два программных потока на двухъядерном процессоре,

каждый поток выполняется независимо

Рисунок 1.6 – Сравнение гиперпоточности и многоядерности

Двухъядерный процессор обеспечивает истинно параллельное выполнение каждого программного потока, в то время как одноядерный процессор работает с программами последовательно, переключаясь между потоками временно (контекстно) для эмуляции параллелизма.

Необходимо отметить, что технология гиперпоточности не направлена на то, чтобы достичь теоретического прироста производительности многоядерных систем в 100 % (то есть двукратного роста производительности в случае двухъядерной системы). Гиперпоточность – это средство, позволяющее программисту задействовать свободные ресурсы процессора для выполнения дополнительной работы. В сочетании с технологией многоядерности гиперпоточность способна дать мощные ресурсы для оптимизации, существенно повышающей производительность системы [9].

* 1. Выбор вычислительной системы

Для решения задачи оптимизации производительности вычислений на многоядерных процессорах был проведен анализ ряда доступных процессоров. В результате анализа были выбраны процессоры   
*Intel Core i7-1165G7* и *AMD Ryzen 5 5600X.*

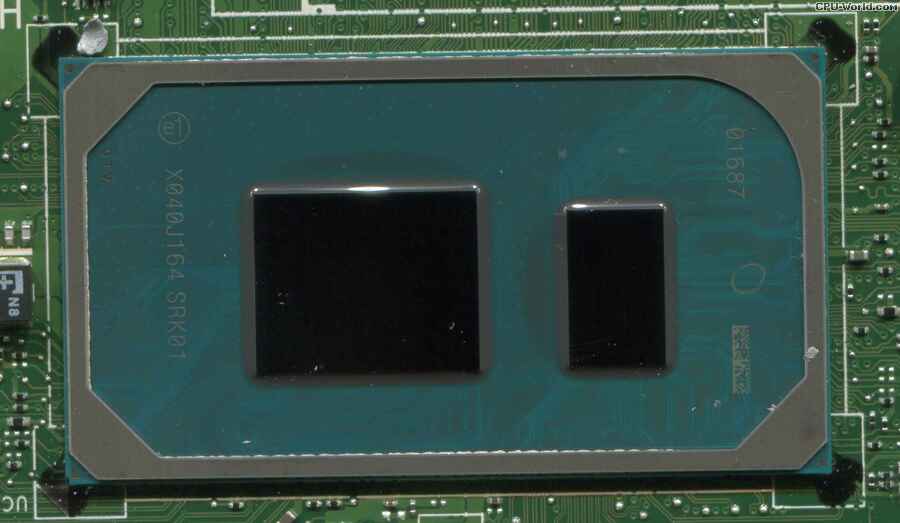
Основные технические характеристики процессоров представлены в таблице 1.1 [10].

Таблица 1.1 – Основные характеристики выбранных процессоров

|  |  |  |
| --- | --- | --- |
|  | *AMD Ryzen 5 5600X* | *Intel Core i7-1165G7* |
| Кодовое имя архитектуры | *Vermeer (Zen 3)* | *Tiger Lake-UP3* |
| Физические ядра | 6 | 4 |
| Логические ядра | 12 | 8 |
| *L1* кэш | 384 *KB* | 320 *KB* |
| *L2* кэш | 3 *MB* | 5 *MB* |
| *L3* кэш | 32 *MB* | 12 *MB* |
| Тактовая частота | 3.7 *GHz* / 4.6 *GHz* | 2.8 *GHz* / 4.7 *GHz* |
| Техпроцесс | 7 *nm* | 10 *nm* |
| Разрядность | 64 *bit* | 64 *bit* |
| Встроенная графика | – | *Intel Iris Xe Graphics* |
| *TDP* | 28 *Watt* | 65 *Watt* |

Выбор (рисунок 1.7) обосновывается следующими факторами:

1. Оба процессора обладают многоядерной структурой. Каждое ядро может выполнять свою собственную задачу, что увеличивает общую производительность системы при работе с параллельными приложениями.
2. Поддержка гиперпоточности. Оба выбранных процессора, обладают технологией *Hyper-Threading* (в случае *Intel*) или ее эквивалентом – *Simultaneous Multithreading* *(SMT)* (в случае *AMD*). Рассмотрение данного аспекта важно для анализа и сравнения производительности в условиях параллельного программирования.
3. Совместимость с технологиями параллельного программирования. Это обеспечивает гибкость в выборе инструментов и подходов при решении задач оптимизации производительности.
4. Наличие оборудования – обладая обоими устройствами, есть удобная возможность провести сравнительный анализ и оценить производительность каждого из них в контексте поставленной задачи оптимизации.

а б

а – процессор Intel Core i7-1165G7;

б – процессор AMD Ryzen 5 5600X

Рисунок 1.7 – Изображения выбранных процессоров

Исходя из вышеуказанных факторов, выбор процессоров обоснован их многоядерной архитектурой, совместимостью с технологиями параллельного программирования, и наличием оборудования для проведения сравнительного анализа. Этот выбор предоставляет возможность изучить прирост производительности в многопоточных сценариях по сравнению с однопоточными, что является ключевым аспектом для разработки эффективных параллельных вычислительных алгоритмов.

1. Платформа программного обеспечения
   1. Структура и архитектура платформы

Для выполнения данного курсового проекта была выбрана платформа программного обеспечения *Windows 10*. Ниже рассмотрены основные аспекты и характеристики этой платформы, разработанной корпорацией *Microsoft*, а также её роль в разработке и выполнении прикладных программ, с учетом оптимизации производительности на многоядерных процессорах при помощи параллельного программирования.

Современная платформа программного обеспечения *Windows* имеет модульную структуру, что позволяет ей эффективно управлять ресурсами многоядерных процессоров и обеспечивать высокую производительность при выполнении параллельных вычислений. ОС *Windows* состоит из следующих основных компонентов [11]:

1. Ядро *Windows* *(Windows Kernel)* – центральная часть операционной системы, обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, память, внешнее аппаратное обеспечение, внешнее устройство ввода и устройства вывода. Также обычно ядро предоставляет сервисы файловой системы и сетевых протоколов. Как основополагающий элемент ядро представляет собой наиболее низкий уровень абстракции для доступа приложений к ресурсам системы, необходимым для своей работы. Как правило, ядро предоставляет такой доступ исполняемым процессам соответствующих приложений за счёт использования механизмов межпроцессного взаимодействия и обращения приложений к системным вызовам ОС.
2. Оболочка *Windows (Windows Shell)* – это графический пользовательский интерфейс операционной системы *Windows*, который предоставляет пользователям средства для взаимодействия с компьютером и управления файлами и приложениями. Она является ключевой частью пользовательского опыта в *Windows* и предоставляет удобный способ доступа к функциям ОС.
3. Службы *Windows (Windows Services)* – представляют собой фоновые приложения, которые работают в операционной системе *Windows* независимо от активных пользовательских сессий. Они выполняют разнообразные задачи и функции, которые помогают обеспечивать стабильную и безопасную работу операционной системы. Например, служба обновлений *Windows*, служба антивирусной защиты.
   1. История, версии и достоинства

Операционная система *Windows* является одной из самых значимых и влиятельных операционных систем в истории компьютерной индустрии. Её история началась в далеком 1985 году с выпуска первой версии, и с тех пор *Windows* прошла долгий путь развития, став одной из самых популярных ОС в мире [12].

В 1985 году *Microsoft* выпустила первую версию операционной системы *Windows*, *Windows 1.0.* Эта версия представляла собой графическую оболочку для *MS-DOS*, который был распространенной операционной системой на персональных компьютерах в то время. *MS-DOS (Microsoft Disk Operating System)* представлял собой командную строку, где пользователи могли вводить команды текстовым образом для выполнения задач. Однако *Windows 1.0* внесла революцию, добавив элементы графического интерфейса, такие как окна, иконки, меню и диалоговые окна, что сделало использование компьютеров более интуитивным и доступным для широкой аудитории. Несмотря на ограниченные возможности по сравнению с более поздними версиями *Windows, Windows 1.0* заложила важные основы для будущего развития графических пользовательских интерфейсов в операционных системах.

*Windows 3.0,* выпущенная в 1990 году, стала первой версией *Windows*, которая стала самостоятельной операционной системой, а не просто оболочкой для *MS-DOS*. Она включала поддержку *TrueType* шрифтов (формат компьютерных шрифтов, разработанный корпорацией *Apple* в конце 1980-х годов), что улучшило качество текста, а также предоставила пользователю больше возможностей для одновременного выполнения приложений.

*Windows 95* (1995 год) была одной из самых значимых версий Windows. Она представила новый пользовательский интерфейс с кнопкой "Пуск" и более удобное управление файлами. Одной из ключевых особенностей *Windows 95* была поддержка *Plug and Play (PnP).* Эта технология позволяла компьютеру автоматически обнаруживать новое аппаратное обеспечение, устанавливать необходимые драйверы и настраивать устройства. Это существенно упростило процесс добавления нового оборудования и сделало его более доступным для пользователей.

*Windows XP,* выпущенная в 2001 году, стала одной из самых популярных версий *Windows* в истории. Она предоставила стабильность и удобство использования, а также улучшила поддержку сетей, что сделало её идеальным выбором как для домашних, так и для корпоративных пользователей.

*Windows 7,* выпущенная в 2009 году, заслужила признание благодаря своей высокой стабильности, улучшенной производительности и интуитивно понятному интерфейсу. Эта операционная система была одной из наиболее популярных версий *Windows* и нашла широкое применение как среди домашних пользователей, так и среди бизнес-клиентов. Улучшенная поддержка аппаратного обеспечения, удобная организация рабочего стола и стабильная работа сделали *Windows 7* предпочтительным выбором для многих пользователей, и она оставалась востребованной долгое время до выпуска последующих версий операционной системы *Windows*.

*Windows 10* внедрила "сервисную" модель обновлений, которая предполагает регулярные обновления операционной системы. Это означает, что *Microsoft* регулярно выпускает крупные обновления, обогащая функциональность и улучшая безопасность ОС. Вместо того чтобы выпускать новые версии *Windows* каждые несколько лет, компания предоставляет обновления с новыми функциями и исправлениями прямо через интернет. Это позволяет пользователям оставаться на актуальной версии ОС и получать новые возможности.

*Windows 11* представила совершенно новый пользовательский интерфейс с улучшенными анимациями, более современным дизайном и улучшенной оптимизацией для сенсорных устройств. Этот интерфейс создает более современный и стильный вид операционной системы. Одной из значимых нововведений *Windows 11* является поддержка приложений для *Android*. Пользователи могут устанавливать и запускать приложения из магазина приложений *Android* на своих устройствах с *Windows 11,* что расширяет функциональность и доступность приложений.

Обе версии, *Windows 10* и *Windows 11,* продолжают совершенствовать и развивать пользовательский опыт, предоставляя новые возможности и усовершенствования. Эти операционные системы оставляют прочный след в истории *Windows* и удовлетворяют потребности современных пользователей.

Весь этот путь развития *Windows* свидетельствует о постоянной эволюции операционной системы и стремлении *Microsoft* к улучшению производительности, безопасности и удобства использования для миллионов пользователей по всему миру. *Windows* остается одной из наиболее популярных и важных операционных систем на сегодняшний день.

* 1. Обоснование выбора платформы

В ходе курсового проекта выбор программной платформы играет ключевую роль. В данном случае, было принято решение использовать операционную систему *Windows 10* в качестве платформы для проведения исследований и разработки параллельных программ. Существует несколько важных аргументов, которые обосновывают выбор *Windows 10:*

1. Широкое распространение. Операционная система *Windows* является одной из наиболее распространенных операционных систем в мире. Это означает, что множество пользователей и организаций работают на этой платформе. Такой широкий пользовательский базис обеспечивает актуальность и практическую значимость исследования.
2. Поддержка многих процессоров. *Windows* обладает высокой совместимостью с различными аппаратными конфигурациями, включая многоядерные процессоры. Это предоставляет отличную возможность для исследования и оптимизации вычислений на многопроцессорных системах.
3. Богатая среда разработки. *Windows* предоставляет разработчикам обширный набор инструментов для разработки прикладных программ. Включая *Visual Studio* и поддержку для различных языков программирования. Эти инструменты значительно упрощают процесс создания и тестирования параллельных приложений.
4. Доступность библиотек и фреймворков. На платформе *Windows* доступны множество библиотек и фреймворков для параллельного программирования, такие как *PPL (Microsoft Parallel Patterns Library)* и *OpenMP (Open Multi-Processing).* Эти инструменты облегчают создание параллельных приложений и оптимизацию производительности.
5. *Windows* предоставляет интуитивный интерфейс и легко настраивается для разработки и тестирования параллельных приложений. Это позволяет сосредоточиться на исследовании и оптимизации вычислений, а не на сложностях настройки среды разработки.

Таким образом, выбор операционной системы *Windows 10* в качестве платформы для проведения исследований и разработки параллельных программ обоснован на основе ее широкой распространенности, совместимости с многоядерными процессорами, наличии разнообразных инструментов разработки, богатстве библиотек и простоте использования. Эта платформа обеспечивает удовлетворение требований исследования и позволяет провести обширное исследование в области оптимизации производительности вычислений.

1. Теоретическое обоснование разработки программного продукта
   1. Обоснование необходимости разработки

В современном информационном обществе, где высокая производительность вычислений играет ключевую роль, разработка программного продукта, направленного на оптимизацию производительности вычислений на многоядерных процессорах с использованием параллельного программирования, является важным шагом в обеспечении эффективного использования вычислительных ресурсов.

Современные компьютеры все чаще оснащаются многоядерными процессорами для обеспечения параллельной обработки данных. Это создает потенциал для существенного увеличения производительности при правильном использовании параллельных вычислений. Таким образом, с постоянным ростом сложности и объема вычислительных задач, многоядерные процессоры становятся стандартом, предоставляя множество физических и логических ядер для выполнения задач.

Основные выгоды разработки программного продукта:

1. Улучшение производительности. Разработка программы, способной эффективно использовать многоядерные процессоры, обеспечит значительный прирост производительности вычислений.
2. Параллельное программирование позволяет более эффективно распределять вычислительные задачи между ядрами процессора, что приводит к сокращению времени выполнения программы и более эффективному использованию ресурсов.
3. Повышение конкурентоспособности. В условиях быстро меняющегося рынка эффективность и скорость разработки программного обеспечения становятся ключевыми факторами конкурентоспособности. Разработка оптимизированного продукта способствует укреплению позиций в конкурентной среде.

Обобщая вышеизложенное, разработка программного продукта, направленного на оптимизацию производительности вычислений на многоядерных процессорах, оправдана с точки зрения повышения эффективности использования ресурсов, сокращения времени выполнения вычислений и повышения конкурентоспособности в современном информационном обществе.

* 1. Технологии программирования, используемые для решения поставленных задач

Выбор языка программирования – это важное решение, которое оказывает существенное влияние на характер и эффективность разработки программного продукта. Для создания программного продукта, направленного на оптимизацию производительности вычислений, был выбран *C++*. Этот выбор обосновывается несколькими ключевыми факторами, представленными ниже, подчеркивающими превосходство C++ в контексте задач такого типа.

*C++* обладает высокой производительностью, сравнимой с языком ассемблера. Это обеспечивает возможность создания высокоэффективных программ, что является критически важным в контексте оптимизации производительности вычислений.

Несмотря на свою близость к ассемблеру, *C++* остается высокоуровневым языком программирования. Это обеспечивает удобство и легкость написания кода, что является важным фактором при разработке сложных программных продуктов. Программистам предоставляются удобные средства абстракции, способствующие повышению производительности труда и снижению вероятности ошибок.

*C++* обладает обширным экосистемой библиотек и технологий, которые значительно упрощают процесс разработки. От стандартных библиотек, таких как *STL (Standard Template Library),* до различных фреймворков и инструментов, C++ предоставляет разработчикам множество возможностей для ускорения и улучшения процесса создания программ.

Язык *C++* предоставляет разработчикам полный контроль над управлением памятью. Это особенно важно при разработке высокопроизводительных вычислительных алгоритмов, где эффективное использование ресурсов играет ключевую роль.

*C++* является расширением языка *C*, что обеспечивает обратную совместимость с существующим кодом на *C*. Это означает, что разработчики могут легко интегрировать *C++* в проекты, созданные на *C*, что является важным фактором при работе с существующими системами.

В итоге, выбор *C++* представляет собой компромисс между легкостью написания кода и высокой производительностью, а также обеспечивает доступ к богатому набору библиотек и технологий. Этот язык становится идеальным выбором для разработки программного продукта, где требуется эффективность, высокая производительность и легкость разработки [13].

В процессе разработки программного продукта с целью оптимизации выполнения вычислительных задач применялась технология *OpenMP*.

*OpenMP* – это библиотека для параллельного программирования вычислительных систем с общей памятью. Официально поддерживаются языки *С, С++* и *Fortran*, однако можно найти реализации для некоторых других языков, например *Pascal* и *Java*. Библиотека активно развивается, в настоящий момент актуальный стандарт версии 4.5 (выпущен в 2015 году), однако уже идет разработка версии 5.0. В тоже время, компилятор *Microsoft C++* поддерживает только версию 2.0, а *GCC* –версию 4.0.

*OpenMP* часто используется в математических вычислениях, т.к. предоставляет удобные и гибкие механизмы для внедрения параллельных вычислений в код и позволяет очень быстро и без особого труда распараллелить программу [14].

Основной инструмент *OpenMP* – это директивы. Директивы *OpenMP* внедряются в исходный код и указывают компилятору на участки, которые могут быть выполнены параллельно. Основными директивами и компонентами являются:

1. *`#pragma omp parallel`* – определяет участок, который будет выполняться параллельно несколькими потоками. Это позволяет эффективно использовать многопроцессорные системы.
2. *`#pragma omp for`* – применяется к циклам, разбивая их выполнение между потоками, что также ускоряет обработку данных. Дополнительно, с использованием параметра *`schedule`* можно уточнить, как именно итерации цикла будут распределены между потоками.
3. *#pragma omp critical`* и *`#pragma omp atomic`* – обеспечивают механизмы синхронизации доступа к общим ресурсам в параллельных участках кода.
4. *`num\_threads`* – используется для указания количества потоков, которые будут созданы и участвуют в параллельной секции кода. Это позволяет программисту управлять числом потоков, используемых для выполнения параллельного участка программы.
5. *`shared`* – применяется к переменным, которые должны быть общими для всех потоков и доступными им одновременно. Это означает, что все потоки будут использовать одну и ту же копию переменной в памяти.
6. *`private`* – применяется к переменным, которые должны иметь локальные копии для каждого потока. Каждый поток будет иметь свою собственную версию переменной, что предотвращает конфликты при одновременном доступе разных потоков к одной переменной.

После перечисления основных директив *OpenMP*, важно обратить особое внимание на параметр *`schedule`* в директиве *`#pragma omp for`*, так как он играет значительную роль в оптимизации параллельных вычислений. Опция `*schedule`* изменяет алгоритм распределения итераций между потоками. Всего поддерживается 3 таких алгоритма. Пусть потоков выполняют итераций. Опции планирования:

1. *`schedule(static)`* –статическое планирование. При использовании такой опции итерации цикла будут поровну (приблизительно) поделены между потоками. Нулевой поток получит первые итераций, первый – вторые и т.д.
2. *`schedule(static, 10)`* – блочно-циклическое распределение итераций. Каждый поток получает заданное число итераций в начале цикла, затем (если остались итерации) процедура распределения продолжается. Планирование выполняется один раз, при этом каждый поток «узнает» итерации, которые должен выполнить.
3. *`schedule(dynamic)`*, *`schedule(dynamic, 10)`* – динамическое планирование. По умолчанию параметр опции равен 1. Каждый поток получает заданное число итераций, выполняет их и запрашивает новую порцию. В отличие от статического планирования, выполняется многократно (во время выполнения программы). Конкретное распределение итераций между потоками зависит от темпов работы потоков и трудоемкости итераций.
4. *`schedule(guided)`*, *`schedule(guided, 10)`* – разновидность динамического планирования с изменяемым при каждом последующем распределении числе итераций. Распределение начинается с некоторого начального размера, зависящего от реализации библиотеки до значения, задаваемого в опции (по умолчанию 1). Размер выделяемой порции зависит от количества еще нераспределенных итераций.

Для измерения времени выполнения операций в программном продукте используется библиотека *<chrono>*, которая является частью библиотеки стандартных шаблонов *(STL)* языка *C++*. Эта библиотека предоставляет эффективные и точные инструменты для работы с временем. Основные возможности *<chrono>* включают в себя [15]:

1. *`std::chrono::high\_resolution\_clock`*.Этот класс представляет механизм измерения времени с высокой точностью, используя наилучшие доступные часы на системе. Он предоставляет наивысшую возможную разрешающую способность для замеров времени, что особенно полезно при измерении малых временных интервалов.
2. *`std::chrono::duration\_cast`*.Этот инструмент используется для преобразования одного типа *`std::chrono::duration`* в другой с изменением единиц измерения времени. Например, можно преобразовать разницу во времени между двумя моментами в наносекунды, миллисекунды, секунды и так далее.
3. *`std::chrono::steady\_clock::time\_point`.* Этот класс позволяет представлять конкретные моменты времени.

Библиотека *`nlohmann/json`* используется в проекте с целью обеспечения удобного сохранения результатов вычислений и последующего анализа данных в формате *JSON*. Эта библиотека является эффективным инструментом для работы с *JSON* в языке программирования C++ и обладает несколькими важными преимуществами [16]:

1. Гибкость и интуитивно понятный интерфейс для создания, обработки и манипулирования *JSON*-структурами, что упрощает встраивание библиотеки в проект и облегчает ее использование.
2. Простота в использовании. Создание *JSON*-структур и добавление данных в них осуществляется с легкостью, что упрощает кодирование результатов вычислений.
3. Эффективность. Благодаря эффективным механизмам парсинга и сериализации *JSON*, библиотека обеспечивает оптимальное использование ресурсов, что важно при обработке больших объемов данных.
4. Удобное сохранение результатов для последующего анализа.

Таким образом, для создания программного продукта выбраны инструменты, обеспечивающие высокую эффективность и удобство работы. Использование языка C++ обусловлено его возможностью эффективной работы с разнообразными структурами данных, высокой производительностью и обширным набором библиотек. Технология *OpenMP* позволяет эффективно распараллеливать вычисления, обеспечивая ускорение выполнения задач за счет использования нескольких потоков. Для измерения времени выполнения операций применяется библиотека *<chrono>*, а сохранение результатов осуществлялось в формате *JSON* при помощи библиотеки *`nlohmann/json`*, обеспечивая последующее удобство анализа и визуализации данных.

* 1. Связь архитектуры вычислительной системы с разрабатываемым программным обеспечением

В качестве основных архитектурных компонентов выступают многоядерные процессоры *AMD Ryzen 5 5600X* и *Intel Core i7-1165G7*.

Главным инструментом разработки программного обеспечения является язык программирования *C++* и технология *OpenMP*.

*OpenMP* – интерфейс прикладного программирования *(API)* для масштабируемых *SMP*-систем (симметричные мультипроцессорные системы) в модели общей памяти.

*SMP (Symmetric Multiprocessing) –* это архитектура многопроцессорных систем, в которой каждый процессор имеет равный доступ к памяти и одинаковую степень управления системой. В симметричной многопроцессорной системе нет главного процессора, и каждый процессор выполняет те же функции и имеет равный статус. Это означает, что схожие процессоры могут выполнять параллельные задачи независимо друг от друга.

Исполняемый процесс в памяти может состоять из множественных нитей, которые имеют общее адресное пространство, но разные потоки команд и раздельные стэки. В простейшем случае, процесс состоит из одной нити, выполняющую функцию *main*. Нити иногда называют также потоками, легковесными процессами, *LWP (light-weight processes)*. *OpenMP* основан на существовании множественных потоков в общедоступной памяти. Схема процесса представлена на рисунке 3.1.

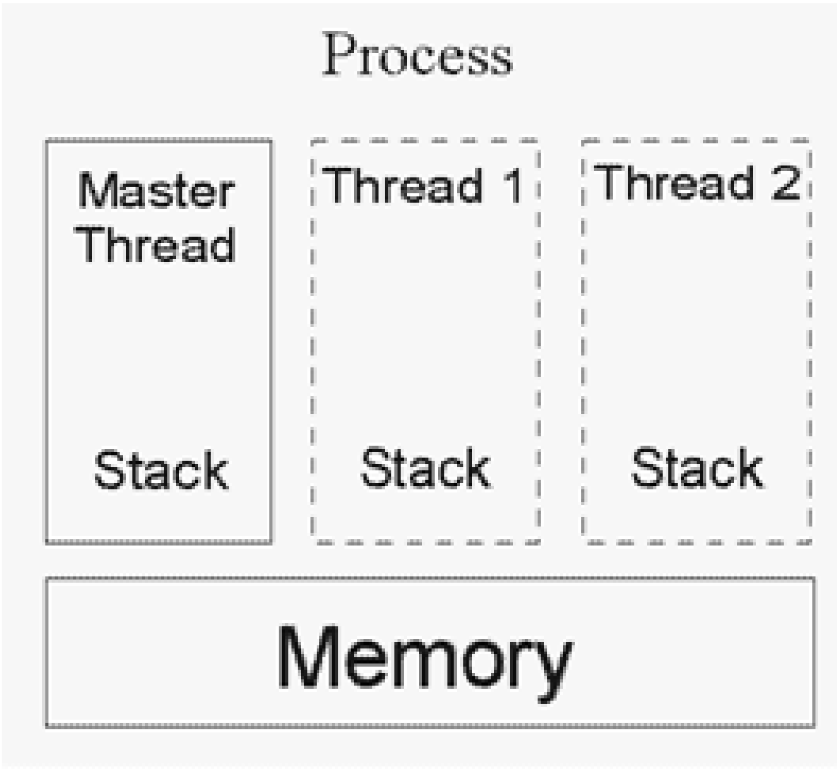


Рисунок 3.1 – Схема существования нескольких потоков

Все программы *OpenMP* начинаются как единственный процесс с главным потоком. Главный поток выполняется последовательно, пока не сталкиваются с первой областью параллельной конструкции. Создание нескольких потоков *(FORK)* и объединение *(JOIN)* проиллюстрировано на рисунке 3.2.

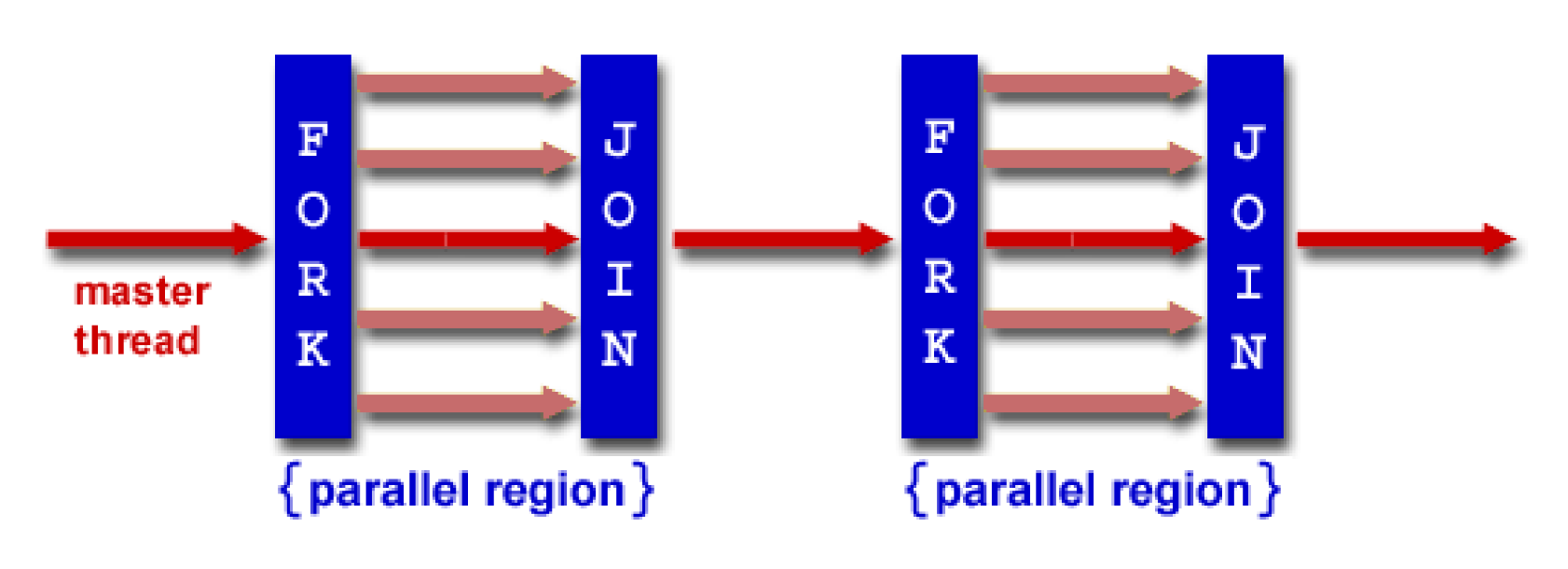


Рисунок 3.2 – Создание и объединение потоков

Как было сказано ранее, директивой создания параллельной области является *`#pragma omp parallel`*. Более наглядно процесс порождения и завершения потоков можно увидеть на рисунке 3.3. Черными линиями показано время жизни потоков, а красными — их порождение.

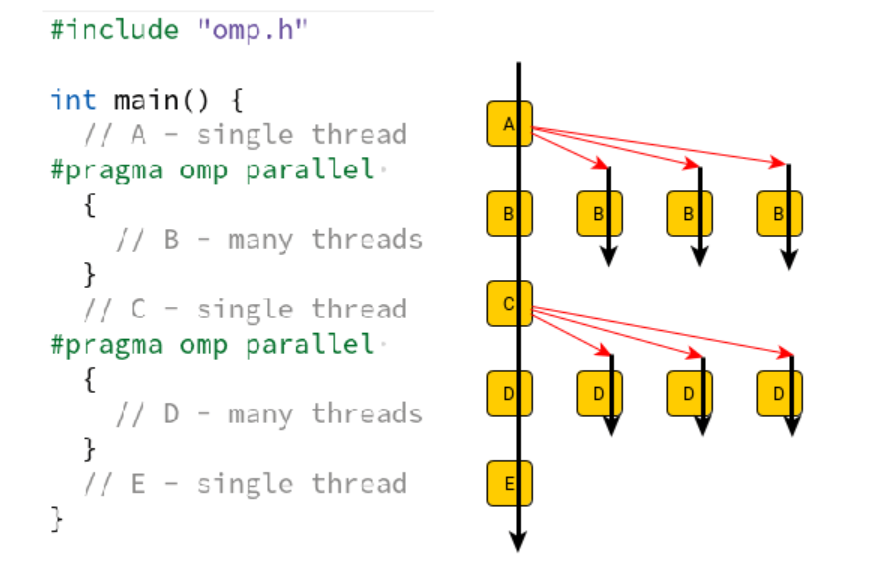


Рисунок 3.3 – Директива *`omp parallel`*

Стоит отметить, что директивы *`parallel`* могут быть вложенными, при этом в зависимости от настроек могут создаваться вложенные потоки.

1. Проектирование функциональных возможностей программы
   1. Анализ алгоритмов

В настоящее время, в условиях постоянного развития технологий, одним из важнейших аспектов в области вычислений является повышение производительности на многоядерных процессорах. Для достижения эффективного распараллеливания вычислений на таких системах необходимо тщательно выбирать соответствующие алгоритмы. Этот раздел посвящен анализу алгоритмов с целью определения их пригодности для параллельного выполнения.

Перед тем как приступить к выбору конкретного алгоритма, необходимо понять основные параметры, которые должны соответствовать алгоритму для успешного распараллеливания. Важными факторами являются:

1. Возможность разделения задачи – алгоритм должен предоставлять возможность разделения задачи на подзадачи, которые могут быть выполнены параллельно. Это обеспечивает эффективное использование многоядерных процессоров.
2. Минимизация зависимостей данных – чем меньше зависимостей между данными, тем легче достигнуть параллелизма. Алгоритмы, минимизирующие зависимости между вычислениями, более подходят для распараллеливания.
3. Управление конфликтами при доступе к ресурсам – эффективное распараллеливание требует управления конфликтами при доступе к общим ресурсам, таким как общая память. Алгоритм должен минимизировать ситуации, когда несколько потоков пытаются изменить одни и те же данные одновременно, чтобы предотвратить «гонку данных» (ситуацию, при которой несколько потоков одновременно пытаются изменить общие данные, что может привести к непредсказуемым результатам из-за неопределенного порядка выполнения операций).

С учетом вышеуказанных параметров, особое внимание уделяется методу Гаусса, широко применяемому для решения систем линейных алгебраических уравнений. В дальнейшем будет рассмотрено, как данный метод может быть эффективно распараллелен с использованием технологии *OpenMP*.

* 1. Метод Гаусса

Применяемые в настоящее время методы решения систем можно разделить на две группы: точные и приближенные. К точным относятся такие методы, которые в предположении, что вычисления ведутся без округлений, позволяют получить точные значения неизвестных. К приближенным относятся методы, которые даже в предположении, что вычисления ведутся без округлений, позволяют получить решение системы лишь с заданной точностью.

Метод Гаусса, широко используемый для решения систем линейных алгебраических уравнений (СЛАУ), представляет собой эффективный алгоритм, основанный на матричных преобразованиях, относится к точным методам решения. В данном разделе будет рассмотрен его математический аспект, начиная с базовых понятий [17].

Линейное алгебраическое уравнение (ЛАУ) – это уравнение первой степени, в котором неизвестные входят линейно. Общий вид линейного алгебраического уравнения можно представить следующим образом:

где – неизвестные переменные, а – известные коэффициенты.

Система линейных алгебраических уравнений (СЛАУ) – это совокупность уравнений, в которых неизвестные входят линейно. Общий вид СЛАУ:

где – коэффициенты, – переменные и – правые части уравнений.

Для системы удобно использовать расширенную матрицу:

где – матрица коэффициентов, а – столбец правых частей.

Элементарные преобразования, такие как умножение строки на константу, прибавление к одной строке другой и перестановка строк, применяются к расширенной матрице, не меняя решений системы [18].

Приведение расширенной матрицы системы к верхнетреугольному   
виду – ключевой этап метода Гаусса. Этот процесс заключается в последовательном выполнении элементарных преобразований, направленных на устранение нижних элементов матрицы. После приведения матрицы к верхнетреугольному виду, последующий обратный ход становится более простым.

Для каждого столбца определяется главный элемент, который представляет собой первый ненулевой элемент в данном столбце. После выбора главного элемента осуществляется процесс приведения к нулю нижних элементов в текущем столбце. Этот процесс включает в себя вычитание из каждой последующей строки соответствующей кратной строки, полученной умножением главной строки на подходящий коэффициент.

Процесс выбора главного элемента и приведения к нулю нижних элементов повторяется для каждой строки и каждого столбца, начиная с первой строки. Каждый новый шаг уменьшает количество свободных переменных, приближая матрицу к верхнетреугольному виду.

После завершения приведения к треугольному виду, система принимает более удобную форму для последующего обратного хода и нахождения значений неизвестных переменных:

Обратный ход – последний этап метода Гаусса, который применяется после приведения расширенной матрицы системы к верхнетреугольному виду. Этот процесс направлен на нахождение значений неизвестных переменных системы линейных алгебраических уравнений (СЛАУ). Рассмотрим этот этап более подробно.

Обратный ход начинается с последнего уравнения системы, в котором у нас есть всего одна неизвестная переменная. Решение для этой переменной находится просто делением правой части уравнения на коэффициент при этой переменной в последней строке.

После нахождения значения последней неизвестной, это значение подставляется в предпоследнее уравнение системы, где теперь остается две неизвестные. Таким образом, можно найти значение предпоследней неизвестной.

Этот процесс продолжается, пока не найдутся значения всех неизвестных переменных. На каждом шаге значения известных переменных подставляются в предыдущее уравнение, и таким образом, находятся значения всех неизвестных в «обратном» порядке.

Асимптотическая сложность алгоритма составляет , где  
 – размерность системы уравнений.

Таким образом, метод Гаусса представляет собой мощный инструмент для решения систем линейных алгебраических уравнений (СЛАУ), который обеспечивает точные значения неизвестных переменных. В данном разделе были рассмотрены математические аспекты этого метода, начиная с базовых понятий линейных алгебраических уравнений и систем.

Преимущества метода Гаусса [19]:

1. Универсальность: Метод Гаусса применим к широкому спектру систем линейных уравнений, включая системы с разным количеством уравнений и неизвестных.
2. Точность: Метод Гаусса является точным методом, предоставляющим точные значения неизвестных в случае отсутствия округлений.
3. Простота реализации: Простота алгоритма и легкость его понимания делают метод Гаусса доступным для широкого круга пользователей.

Недостатки метода Гаусса:

1. Чувствительность к вырожденным случаям. В некоторых случаях, если система близка к вырожденной (существует линейная зависимость между уравнениями), метод может стать неустойчивым или дать неточные результаты.
2. При работе с большими матрицами метод Гаусса может потребовать значительных вычислительных ресурсов.

В целом, метод Гаусса остается важным инструментом в численных методах и приложениях линейной алгебры, но при выборе метода решения необходимо учитывать особенности конкретной задачи.

* 1. Описание функциональной схемы программы

Функциональная схема программы представлена в приложении Б.   
В рамках разработки функциональной схемы были выделены четыре ключевых блока:

1. Инициализация начальных значений. На данном этапе происходит создание объекта, необходимого для хранения результатов тестирования, и инициализация массива, содержащего значения размеров тестируемых матриц.
2. Тестирование функций. В этом блоке происходит измерение времени выполнения последовательной и параллельной версии алгоритма решения СЛАУ методом Гаусса, а также сохранение этой информации в ранее созданный объект.
3. Вывод результатов в консоль. Здесь производится вывод результатов предыдущего этапа в консоль в формате таблицы, содержащей следующие заголовки: *size, sequential, parallel, speedup*, где *size –* размер протестированной матрицы, *sequential* – время выполнения последовательного алгоритма решения СЛАУ, *parallel* – время выполнения параллельного алгоритма решения СЛАУ, *speedup* – отношение *sequential* к *parallel*, т. е. ускорение.
4. Запись результатов в файл – занесенные на втором этапе результаты тестирования записываются в файл для дальнейшего анализа.
   1. Описание блок схемы алгоритма программы

Блок схема алгоритма программы представлена в приложении В.   
Блок-схема состоит из пяти ключевых функций:

* *main*: точка входа в программу, ответственная за вызов всех функций и обработку их результатов;
* *test*: определяет среднее время выполнения переданной функции (последовательной или параллельной реализации решения СЛАУ методом Гаусса) за определенное количество итераций;
* *generate*: генерирует заполненную случайными числами расширенную матрицу некоторого размера;
* *solution*: последовательная реализация решения системы линейных алгебраических уравнений (СЛАУ) методом Гаусса;
* *solutionOMP*: параллельная реализация решения СЛАУ методом Гаусса с использованием *OpenMP*.
  1. Реализация программы

В данном подразделе представлена детальная реализация метода Гаусса для решения систем линейных алгебраических уравнений (СЛАУ) с использованием языка программирования *C++*. Код включает как последовательную, так и параллельную версии алгоритма, а также инструменты для замера времени, вывода подробной информации в консоль и сохранения результатов в формате *JSON*.

Для успешной компиляции и выполнения программы используются следующие библиотеки и инструменты (листинг А.1):

* *<iostream>*: библиотека ввода-вывода;
* *<vector>*: контейнер из стандартной библиотеки *C++*, представляющий собой динамический массив с автоматическим управлением размера;
* *<chrono>*: библиотека, предоставляющая возможности для работы с временем;
* *<iomanip>*: библиотека, позволяющая управлять форматированием вывода данных в консоль;
* *<fstream>*: библиотека для работы с файлами;
* *<fstream>*: библиотека для работы с файлами;
* *<nlohmann/json.hpp>*: библиотека для работы с форматом *JSON*.

Функция генерации матрицы (листинг А.2) – предназначена для генерации расширенной матрицы размера . В контексте решения системы линейных уравнений методом Гаусса, каждая строка матрицы представляет собой коэффициенты уравнения и свободный член.

Шаги функции:

1. Инициализация генератора случайных чисел. *`srand(0)`* устанавливает начальное значение для генератора случайных чисел. Это делается для обеспечения одинаковых исходных данных для соответствующих случаев тестирования.
2. Создание матрицы. Инициализация двумерного вектора размера , представляющего собой расширенную матрицу.
3. Заполнение матрицы случайными значениями.
4. Возврат сгенерированной матрицы.

Функция решения СЛАУ (листинг А.3) – последовательная реализация метода Гаусса решения системы линейных алгебраических выражений. Принимает на вход расширенную матрицу и возвращает вектор решения.

Шаги функции:

1. Начальная инициализация переменных (определение числа строк и столбцов в расширенной матрице).
2. Приведение матрицы к верхнетреугольному виду с помощью нескольких циклов. Внешний цикл осуществляет проход по всей матрице сверху вниз, внутренние – осуществляют соответствующие операции с элементами.
3. Обратный ход.
4. Возврат вектора решения.

Функция решения СЛАУ с использованием *OpenMP* (листинг А.4) – параллельная реализация метода Гаусса решения системы линейных алгебраических выражений. Принимает на вход расширенную матрицу и возвращает вектор решения.

Шаги функции:

1. Инициализация переменных, определяющих количество строк и столбцов в матрице.
2. Приведение матрицы к верхнетреугольному виду. Внешний цикл проходит по каждой строке, за исключением последней. Внутренний цикл распараллеливается с использованием директивы *`#pragma omp parallel for schedule(dynamic)`.* Каждый поток выполняет часть итераций внутреннего цикла. Внутренний цикл рассчитывает коэффициент, на который нужно умножить элементы текущей строки для обнуления соответствующих элементов нижних строк. После этого производится вычитание из нижних строк текущей строки, умноженной на рассчитанный коэффициент.
3. Обратный ход. Внешний цикл проходит по строкам матрицы в обратном порядке. Внутренний цикл также распараллеливается с использованием *`#pragma omp parallel for`.* Каждый поток выполняет часть итераций внутреннего цикла. Внутренний цикл рассчитывает значение текущего элемента решения путем деления соответствующего элемента свободного столбца на соответствующий диагональный элемент. Затем производится вычитание из верхних строк текущей строки, умноженной на рассчитанное значение решения.
4. Возврат вектора решения.

Важным моментом является использование во втором пункте динамического распределения итераций. *`schedule(dynamic)`* гарантирует, что каждый поток получит новую итерацию сразу после завершения предыдущей. Такой подход уменьшает простои потоков и обеспечивает более равномерное распределение нагрузки между ними.

При этом во втором распараллеливании используется стандартное статическое распределение, которое предоставляет каждому потоку определенное количество итераций единожды на этапе компиляции. Такой ход объясняется более простыми вычислениями в распараллеленном цикле. Если бы использовалось динамическое распределение, то расходы, затраченные на выделение итераций, превысили бы потенциальную выгоду от распределения такого типа.

Также для инструкций распараллеливания не указывается число используемых потоков для того, чтобы дать *OpenMP* возможность автоматически адаптироваться к конфигурации системы и использовать все доступные ресурсы процессора.

Шаблонная функция измерения времени выполнения (листинг А.5) представляет собой инструмент для измерения времени выполнения алгоритма. Эта функция используется для сравнения производительности последовательной и параллельной реализаций метода Гаусса.

В целях сравнения производительности, эта функция использует стандартную библиотеку *<chrono>*, предоставляющую средства измерения времени с высоким разрешением. Она включает цикл, внутри которого испытуемая функция вызывается заданное количество раз, и для каждого вызова фиксируется момент начала и завершения. Таким образом, удается получить усредненные данные о времени выполнения.

Точка входа в программу (листинг А.6) является ключевым компонентом управления выполнением программы. Программа итерируется по определенным значениям переменной *n*. Для каждого значения *n* производится вызов функции измерения времени выполнения с соответствующими параметрами (функция реализации метода Гаусса обычная или распараллеленная, само число *n* и количество итераций для усреднения значения). Результаты измерений выводятся в консоль в форматированной таблице, а также записываются в формат *JSON* для последующего анализа.

Таким образом, представленный раздел охватывает детальную реализацию метода Гаусса для решения систем линейных алгебраических уравнений (СЛАУ) на языке программирования *C++*. Включены последовательная и параллельная версии алгоритма, сопровождаемые инструментами для замера времени, вывода информации в консоль и сохранения результатов в формате *JSON*, что обеспечивает удобство для последующего анализа и интерпретации полученных данных.

1. Сравнение производительности
   1. Понятие производительности

При оценке выигрыша в производительности при параллельном программировании используется показатель, известный как ускорение. Ускорение рассчитывается как отношение времени выполнения лучшего последовательного алгоритма к времени выполнения соответствующей параллельной программы. Это отношение характеризует, насколько быстрее программа выполняется при распараллеливании, и выражается как

Ускорение определяется в зависимости от количества физических потоков , использованных при параллельной организации.

С учетом предыдущего определения ускорения представляется возможным вычислить теоретический предел повышения производительности при увеличении числа процессорных ядер (и, следовательно, физических потоков).

Исследование этого вопроса начинается с работы Джина Амдала, выполненной в 1967 году. Закон Амдала, известный как правило Амдала, описывает максимальный теоретический выигрыш в производительности параллельного решения по отношению к лучшему последовательному решению [20].

Под лучшим последовательным решением понимается оптимальный вариант алгоритма или программы, который выполняется последовательно, то есть в одном потоке, и служит базовым эталоном для сравнения с производительностью параллельных версий

Амдал начал с интуитивно понятного утверждения, что ускорение программы зависит как от части программы, которая поддается ускорению, так и от степени ускорения этой части программы. Математическое выражение данного утверждения:

Амдал далее продвинулся в анализе, чтобы продемонстрировать, как данное уравнение выражается для тех частей программы, которые выполняются последовательно и тех, которые выполняются последовательно.

Закон Амдала.

Пусть необходимо решить некоторую вычислительную задачу. Предположим, что её алгоритм таков, что доля от общего объёма вычислений может быть получена только последовательными расчётами, а, соответственно, доля может быть распараллелена идеально (то есть время вычисления будет обратно пропорционально числу задействованных узлов ). Тогда ускорение, которое может быть получено на вычислительной системе из процессоров, по сравнению с однопроцессорным решением не будет превышать величины

Верхний предел ускорения для приложения, выполняющего последовательный код с долей , ограничен самой последовательной частью кода и равен

Однако для того, чтобы закон Амдала отражал реальности многоядерных систем, а не теоретический максимум, следует учесть накладные расходы (издержки) на добавление программных потоков. Тогда ускорение равно

где – накладные расходы.

Издержки состоят из двух частей: фактические издержки операционной системы и издержки на межпоточные процессы, такие как синхронизация и прочие формы взаимодействия между потоками. Стоит обратить внимание, что если издержки достаточно велики, то они перевешивают выигрыш от параллельной части программы. Следовательно величина ускорения может стать меньше единицы, а это значит, что поточная обработка фактически снижает производительность по сравнению с однопоточным решением. Эта ситуация часто встречается в плохо разработанных многопоточных приложениях. Отсюда важное следствие: потери вследствие внедрения поточной обработки должны быть сведены к минимуму.

Ранее обсуждалось применение закона Амдала к многопроцессорным и многоядерным системам. При использовании гиперпоточности в сравнении с отсутствием ее использования важно отметить, что распределение ресурсов между программными потоками может как увеличивать, так и уменьшать величину прироста производительности. Важно предложить такую формулу закона Амдала, которая будет учитывать исполнительную среду с чередованием (гиперпоточность). Однако создание универсальной формулы для учета гиперпоточности усложняется из-за индивидуальных потребностей и особенностей каждого приложения.

* 1. Основные сведения о тестировании

Далее будут рассмотрены результаты экспериментов по сравнению производительности алгоритма решения систем линейных уравнений (СЛАУ) методом Гаусса на многоядерных процессорах (были упомянуты ранее, *AMD Ryzen 5 5600X и Intel Core i7-1165G7*). Основной акцент будет сделан на анализ прироста производительности при использовании распараллеливания.

Для проведения экспериментов использовалась технология параллельного программирования *OpenMP*. Замер времени выполнения алгоритма осуществлялся с использованием библиотеки *<chrono>.* Результаты экспериментов были сохранены в файл в формате *JSON* для последующего анализа. Размеры матриц, на которых проводились измерения, варьировались от до с шагом 1 и от до с шагом . Такой выбор размеров матриц позволяет исследовать влияние размера матрицы на эффективность распараллеливания алгоритма решения систем линейных уравнений методом Гаусса.

Исследования начались с проведения тестов на обоих компьютерах с включенной гиперпоточностью. После этого функциональность гиперпоточности была отключена, и тесты были повторно запущены для получения дополнительных данных.

Для вычисления ожидаемого прироста производительности необходимо применить формулу Амдала (5.4). Так как параллельная функция вычисления решения СЛАУ достаточно хорошо распараллелена, то при стремлении (размера матрицы) к достаточно большому значению число (доля последовательно выполненных операций) стремится к нулю. Следовательно, формула вычисления прироста производительности упрощается   
до , где – число ядер (только для больших ).

* 1. Результат работы программы

Результаты работы программы (при включенной гиперпоточности) представлены в виде скриншотов консольного приложения, запущенного на обоих компьютерах (рисунки 5.1, 5.2).

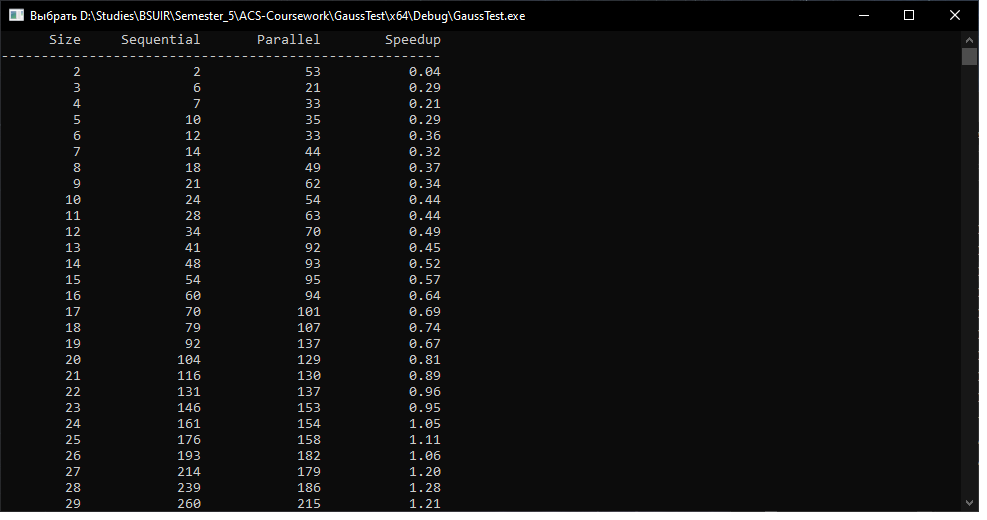


Рисунок 5.1 – Часть вывода программы на *AMD Ryzen 5 5600X*

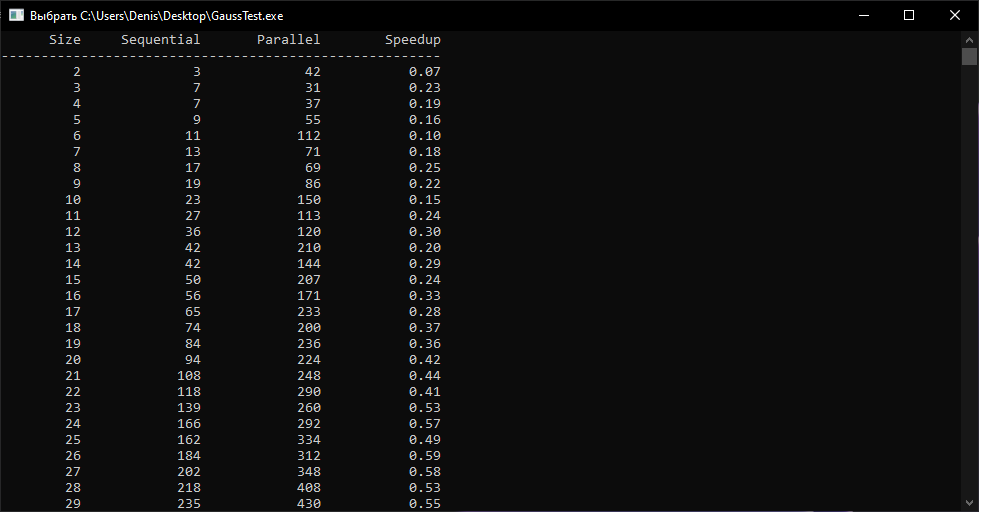


Рисунок 5.2 – Часть вывода программы на *Intel Core i7-1165G7*

Каждый скриншот включает в себя вывод программы, отображающий замеры времени выполнения в микросекундах (*sequential –* для последовательной версии алгоритма,   
*parallel* – для распараллеленной) для различных размеров матриц (*size*) с величиной прироста (*speedup*), вызванной использованием распараллеленной версии алгоритма.

Эти данные, также как и данные, полученные после выключения гиперпоточности и повторного запуска программ, были сохранены в соответствующие файлы в формате *JSON* для последующего анализа и использования. После этого данные были экспортированы в таблицы программы *Microsoft Excel*, где были построены соответствующие графики для более детального анализа и визуализации полученных результатов.

* 1. Анализ результатов

Графики сравнения производительности обоих процессоров представлены в приложении Г. На листе 1 находятся графики, относящиеся к результатам тестов процессора *AMD Ryzen 5 5600X*, на листе 2 – к на *Intel Core i7-1165G7.* Для каждого процессора представлены по два полных графика тестирования: один с использованием гиперпоточности и один без нее. Также для выявления особенностей поведения алгоритмов при небольших системах дополнительно приведены увеличенные графики (при не столь больших значениях ).

На оси абсцисс отображены размеры тестируемых матриц . Пунктирный график (*sequential)* представляет собой время выполнения в последовательной версии алгоритма. Сплошной график (*parallel*) представляет собой время выполнения в распараллеленной с помощью *OpenMP* версии алгоритма.

Влияние размера матрицы на эффективность распараллеливания метода Гаусса с использованием *OpenMP* является важным аспектом анализа. На небольших матрицах (например, ) при внедрении параллельных вычислений наблюдается увеличение времени выполнения в несколько раз. Это может быть объяснено дополнительными накладными расходами, связанными с организацией параллельных потоков и их управлением, которые превосходят выигрыш от самого параллелизма.

Однако, с увеличением размера матрицы, дополнительные затраты, связанные с параллельными вычислениями, начинают становиться незначительными по сравнению с общим временем выполнения расчетов. При некотором размере матрицы наступает момент, когда дополнительные расходы, связанные с распараллеливанием, становятся сопоставимыми с выигрышем от параллельных вычислений (точка пересечения графиков *sequential* и *parallel*). При дальнейшем увеличении размера матрицы наблюдается рост эффективности применения параллелизма. Коэффициент прироста, то есть отношение времени выполнения последовательной версии алгоритма к параллельной, увеличивается и упирается в некоторое значение, определяемое характеристиками аппаратного обеспечения.

Так, на матрицах размером от до процессор *AMD Ryzen 5 5600X* продемонстрировал средний прирост производительности  
 раза с использованием гиперпоточности и без нее, в то время как процессор *Intel Core i7-1165G7* показал улучшение в и раз соответственно. Эти значения представляют собой существенное улучшение эффективности выполнения алгоритма.

Как отмечено в первом разделе, *AMD Ryzen 5 5600X* обладает 6 физическими и 12 логическими ядрами, а конфигурация   
*Intel Core i7-1165G7 –* 4/8*.* Стоит отметить, что, несмотря на наличие технологии гиперпоточности на обоих процессорах, прирост производительности не достигает количества логических ядер, предоставляемых процессорами. Конкуренция за ресурсы является основной причиной, объясняющей этот факт. Она возникает из-за разделения ресурсов, таких как кэш и исполнительные блоки, между логическими ядрами. Такое разделение может привести к конфликтам за доступ к данным и исполнительным ресурсам, уменьшая общую эффективность каждого ядра при условии гиперпоточности.

У *AMD* наблюдается более значительный прирост производительности, вызванный использованием гиперпоточности, составляющий , по сравнению с *Intel*, где этот прирост составил . Это различие может объясняться несколькими факторами. Во-первых, архитектурные особенности процессора *AMD* могут обеспечивать более эффективное использование гиперпоточности и распределение ресурсов между логическими ядрами.   
Во-вторых, у процессора *Intel*, по умолчанию, меньше физических ядер, что также может сказываться на максимальном приросте производительности при использовании гиперпоточности. Кроме того, различия в технических характеристиках компьютеров, таких как тип и объем оперативной памяти, могут влиять на общую производительность систем.

В итоге, проведенный анализ на процессорах *AMD Ryzen 5 5600X* и  
*Intel Core i7-1165G7* с применением и без применения гиперпоточности подчеркивает эффективность параллельных вычислений на больших матрицах. Процессор *AMD* проявил более существенное улучшение производительности при использовании гиперпоточности по сравнению с *Intel*, возможно, благодаря архитектурным особенностям и большему числу физических ядер. Указанные результаты подчеркивают важность выбора аппаратной конфигурации и оптимизации для эффективного использования параллелизма в вычислительных задачах.

Заключение

В рамках данного курсового проекта было проведено исследование и реализация методов оптимизации производительности вычислений на многоядерных процессорах с использованием параллельного программирования. Объектом исследования стало решение систем линейных уравнений методом Гаусса, что позволило проанализировать производительность алгоритма на многоядерных системах   
*AMD Ryzen 5 5600X* и *Intel Core i7-1165G7*.

В процессе исследования были изучены особенности многоядерных процессоров, включая их архитектуру и внутреннее устройство. Кроме того, были детально рассмотрены основные принципы и концепции технологии параллельного программирования *OpenMP*. Этот анализ включал в себя рассмотрение ключевых методов и средств, предоставляемых библиотекой для эффективного распараллеливания программ.

Была проведена разработка программной реализации алгоритма решения систем линейных алгебраических уравнений методом Гаусса, использующей технологию *OpenMP* для параллельного выполнения вычислений. Основным намерением было создать эффективную и оптимизированную версию алгоритма, способную полностью использовать вычислительные возможности многоядерных процессоров.

Анализ данных, полученных в ходе экспериментов, выявил, что на данных небольших размеров распараллеливание может привести к увеличению времени выполнения из-за накладных расходов, вызванных организацией параллельных потоков и их управлением. Однако, с увеличением размера данных, выигрыш от параллельного выполнения становится существенным.

Таким образом, данное исследование предоставляет важные выводы о применении параллельного программирования на многоядерных процессорах, выделяя плюсы и минусы данного подхода, что имеет значение для дальнейших исследований в области оптимизации производительности вычислений на современных аппаратных платформах.

Список литературных источников

1. Доманов, А. Т. Стандарт предприятия / А. Т. Доманов,   
   Н. И. Сорока. – Минск : БГУИР, 2017. – 167 с.
2. Эхтер, Ш. Многоядерное программирование / Ш. Эхтер, Дж. Робертс. – СПб. : Питер, 2010. – 316 с.
3. Классификация Флинна [Электронный ресурс]. – Режим доступа: https://parallel.ru/computers/taxonomy/flynn.html. – Дата доступа: 21.09.2023.
4. Параллелизм на уровне команд [Электронный ресурс]. – Режим доступа: https://studref.com/524588/informatika/parallelizm\_urovne\_komand. – Дата доступа: 21.09.2023.
5. Что такое центральный процессор [Электронный ресурс]. – Режим доступа: https://aws.amazon.com/ru/what-is/cpu/. – Дата доступа: 21.09.2023.
6. Hyper-Threading [Электронный ресурс]. – Режим доступа: https://www.alterbit.ru/glossary179.html/. – Дата доступа: 23.09.2023.
7. Многоядерный процессор [Электронный ресурс]. – Режим доступа: http://db4.sbras.ru/elbib/data/show\_page.phtml?77+743. – Дата доступа: 23.09.2023.
8. Обзор архитектуры современных многоядерных процессоров [Электронный ресурс]. – Режим доступа: https://intuit.ru/studies/courses/10611/1095/lecture/22904?page=3. – Дата доступа: 23.09.2023.
9. Что такое гиперпоточность [Электронный ресурс]. – Режим доступа: https://zawindows.ru/что-такое-hyperthreading-гиперпоточность/. – Дата доступа: 23.09.2023.
10. Сравнение процессоров [Электронный ресурс]. – Режим доступа: https://technical.city/ru/cpu. – Дата доступа: 24.09.2023.
11. Архитектурные модули Windows NT   
    [Электронный ресурс]. – Режим доступа: https://studref.com/549707/informatika/arhitekturnye\_moduli\_windows. – Дата доступа: 25.09.2023.
12. История развития Microsoft Windows [Электронный ресурс]. – Режим доступа: https://tproger.ru/translations/microsoft-windows-history. – Дата доступа: 25.09.2023.
13. Страуструп, Б. Язык программирования С++ / Б. Страуструп. –   
    СПб.: Невский диалект, 1999. – 369 с.
14. Учебник по OpenMP [Электронный ресурс]. – Режим доступа: https://pro-prof.com/archives/4335/. – Дата доступа: 15.10.2023.
15. Date and time utilities [Электронный ресурс]. – Режим доступа: https://en.cppreference.com/w/cpp/chrono. – Дата доступа: 15.10.2023.
16. JSON for Modern C++ [Электронный ресурс]. – Режим доступа: https://json.nlohmann.me/. – Дата доступа: 15.10.2023.
17. Самарский, А. А. Численные методы / А. А. Самарский,   
    А. В. Гулин. – Москва : Наука, 1989. – 432 с.
18. Решение систем линейных уравнений методом Гаусса [Электронный ресурс]. – Режим доступа: https://prog-cpp.ru/gauss/. – Дата доступа: 16.10.2023.
19. Преимущества и недостатки метода Гаусса   
    решения СЛАУ [Электронный ресурс]. – Режим доступа: https://studbooks.net/2395779 /preimuschestva\_nedostatki\_metoda\_gaussa. – Дата доступа: 16.10.2023.
20. Закон Амдала [Электронный ресурс]. – Режим доступа: https://medium.com/german-gorelkin/amdahls-law-79a8edb040e2. – Дата доступа: 20.10.2023.

ПРИЛОЖЕНИЕ А  
(обязательное)  
Листинг программного кода

Листинг А.1 – Зависимости программы

#include <iostream>

#include <vector>

#include <chrono>

#include <iomanip>

#include <fstream>

#include <nlohmann/json.hpp>

using namespace std;

using json = nlohmann::json;

#define ITERATIONS 10

Листинг А.2 – Функция генерации матрицы

vector<vector<double>> generateExtendedMatrix(int n) {

srand(0);

vector<vector<double>> matrix(n, vector<double>(n + 1));

for (int i = 0; i < n; i++) {

for (int j = 0; j < n + 1; j++) {

matrix[i][j] = rand();

}

}

return matrix;

}

Листинг А.3 – Функция решения СЛАУ

vector<double> gaussianSolution(vector<vector<double>> matrix) {

int rows = matrix.size();

int cols = matrix[0].size();

// Приведение матрицы к верхнетреугольному виду

for (int i = 0; i < rows - 1; i++) {

for (int k = i + 1; k < rows; k++) {

double factor = matrix[k][i] / matrix[i][i];

for (int j = i; j < cols; j++) {

matrix[k][j] -= factor \* matrix[i][j];

}

}

}

// Обратный ход

vector<double> solution(rows);

for (int i = rows - 1; i >= 0; i--) {

solution[i] = matrix[i][cols - 1] / matrix[i][i];

for (int k = i - 1; k >= 0; k--) {

matrix[k][cols - 1] -= matrix[k][i] \* solution[i];

}

}

return solution;

}

Листинг А.4 – Функция решения СЛАУ с использованием *OpenMP*

vector<double> gaussianSolution\_OMP(vector<vector<double>> matrix) {

int rows = matrix.size();

int cols = matrix[0].size();

// Приведение матрицы к верхнетреугольному виду

for (int i = 0; i < rows - 1; i++) {

#pragma omp parallel for schedule(dynamic)

for (int k = i + 1; k < rows; k++) {

double factor = matrix[k][i] / matrix[i][i];

for (int j = i; j < cols; j++) {

matrix[k][j] -= factor \* matrix[i][j];

}

}

}

// Обратный ход

vector<double> solution(rows);

for (int i = rows - 1; i >= 0; i--) {

solution[i] = matrix[i][cols - 1] / matrix[i][i];

#pragma omp parallel for

for (int k = i - 1; k >= 0; k--) {

matrix[k][cols - 1] -= matrix[k][i] \* solution[i];

}

}

return solution;

}

Листинг А.5 – Шаблонная функция измерения времени выполнения

template <typename TimeUnits>

long long Test(vector<double>(\*solution)(vector<vector<double>>), int size,  
 int iterations) {

vector<vector<double>> matrix = generateExtendedMatrix(size);

long long sum = 0;

for (int i = 0; i < iterations; i++) {

auto start\_time = chrono::high\_resolution\_clock::now();

solution(matrix);

auto end\_time = chrono::high\_resolution\_clock::now();

sum += chrono::duration\_cast<TimeUnits>(  
 end\_time - start\_time).count();

}

return sum / iterations;

}

Листинг А.6 – Точка входа в программу

int main() {

cout << setw(10) << "Size" << setw(15) << "Sequential" << setw(15)

<< "Parallel" << setw(15) << "Speedup" << endl;

cout << setfill('-') << setw(55) << "-" << setfill(' ') << endl;

json resultArray;

vector<int> testValues;

for (int n = 2; n < 50; n++)

testValues.push\_back(n);

for (int n = 50; n <= 1000; n += 10)

testValues.push\_back(n);

for (int n : testValues) {

auto duration = Test<chrono::microseconds>

(gaussianSolution, n, ITERATIONS);

auto durationOMP = Test<chrono::microseconds>

(gaussianSolution\_OMP, n, ITERATIONS);

cout << setw(10) << n << setw(15) << duration

<< setw(15) << durationOMP

<< setw(15) << fixed << setprecision(2)

<< (double)duration / durationOMP << endl;

json resultObject;

resultObject["n"] = n;

resultObject["duration"] = duration;

resultObject["durationOMP"] = durationOMP;

resultArray.push\_back(resultObject);

}

ofstream file("results.json");

file << setw(4) << resultArray << endl;

return 0;

}

ПРИЛОЖЕНИЕ Б  
(обязательное)  
Функциональная схема алгоритма,   
реализующего программное средство

ПРИЛОЖЕНИЕ В  
(обязательное)  
Блок схема алгоритма,   
реализующего программное средство

ПРИЛОЖЕНИЕ Г  
(обязательное)  
Скриншоты сравнений производительности

ПРИЛОЖЕНИЕ Д  
(обязательное)  
Ведомость документов