Минобрнауки России

Федеральное государственное автономное образовательное учреждение высшего образования   
«Национальный исследовательский университет   
«Московский институт электронной техники»

Институт системной и программной инженерии и информационных технологий (СПИНТех)

Бельгушева Алла Артуровна

Магистерская диссертация   
по направлению 09.04.04 «Программная инженерия»

Исследование и разработка методики и алгоритма распределения нагрузки при обработке прерываний сетевых интерфейсов

Студент Бельгушева А.А.

Руководитель,

к.п.н., доцент Федоров А.Р.

Москва 2023

# ОГЛАВЛЕНИЕ

[ОГЛАВЛЕНИЕ 2](#_Toc136918439)

[ПЕРЕЧЕНЬ СОКРАЩЕНИЙ 4](#_Toc136918440)

[ВВЕДЕНИЕ 6](#_Toc136918441)

[ГЛАВА 1. АНАЛИТИЧЕСКИЙ ОБЗОР МЕТОДОВ И СРЕДСТВ РЕАЛИЗАЦИИ МНОГОПОТОЧНОСТИ 9](#_Toc136918442)

[1.1. Анализ существующих средств реализации многопоточности 9](#_Toc136918443)

[1.2. Анализ методов и алгоритмов повышения производительности многопоточных приложений 14](#_Toc136918444)

[1.3. Технология гиперпоточности 19](#_Toc136918445)

[1.4. Обоснование метода решения 22](#_Toc136918446)

[Выводы к главе 1 25](#_Toc136918447)

[ГЛАВА 2. ФОРМАЛИЗОВАННОЕ ПРЕДСТАВЛЕНИЕ ПРОЦЕССА ОБРАБОТКИ ПРЕРЫВАНИЙ 26](#_Toc136918448)

[2.1. Формализованное представление задачи ускорения многопоточных приложений при использовании технологии гиперпоточности 26](#_Toc136918449)

[2.2. Разработка методики ускорения многопоточных приложений при использовании с технологией гиперпоточности 29](#_Toc136918450)

[2.3. Разработка алгоритма повышения быстродействия библиотеки QuEST 34](#_Toc136918451)

[Выводы по главе 2 42](#_Toc136918452)

[ГЛАВА 3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ АЛГОРИТМА 43](#_Toc136918453)

[3.1. Программная реализация алгоритма АПБ 43](#_Toc136918454)

[3.2. Разработка методов тестирования алгоритма АПБ, внедренного в библиотеку QuEST 52](#_Toc136918455)

[3.3. Оценка эффективности алгоритма АПБ 65](#_Toc136918456)

[ГЛАВА 4. ЭКСПЕРИМЕНТАЛЬНОЕ ПОДТВЕРЖДЕНИЕ РЕЗУЛЬТАТОВ ИССЛЕДОВАНИЯ 72](#_Toc136918457)

[4.1. Практическое приложение результатов 72](#_Toc136918458)

[4.2. Перспективы использования 73](#_Toc136918459)

[Выводы к главе 4 75](#_Toc136918460)

[ЗАКЛЮЧЕНИЕ 76](#_Toc136918461)

[СПИСОК ЛИТЕРАТУРЫ 77](#_Toc136918462)

# ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

.NET Parallel Extensions — библиотека управляемого параллелизма, разработанная в результате сотрудничества Microsoft Research и группы CLR в Microsoft. Библиотека была выпущена в версии 4.0 .NET Framework

Application Programming Interface (API) — описание способов взаимодействия одной компьютерной программы с другими

Hyper-threading technology (HT) — технология гиперпоточности, разработанная компанией Intel для повышения производительности процессоров собственного производства)

Intel Threading Building Blocks (TBB) — кроссплатформенная библиотека шаблонов C++, разработанная компанией Intel для параллельного программирования.

Open Multi-Processing (OpenMP) — открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран

Pentium — торговая марка нескольких поколений микропроцессоров архитектуры x86, выпускаемых корпорацией Intel с 1993 года

POSIX Threads (pthreads) — стандарт POSIX-реализации потоков (нитей) выполнения, определяющий API для управления потоками, их синхронизации и планирования

Quantum Exact Simulation Toolkit (QuEST) — высокопроизводительный симулятор квантовых схем, векторов состояния и матриц плотности.

Single Instruction, Multiple Data (SIMD) — принцип компьютерных вычислений, позволяющий обеспечить параллелизм на уровне данных

Xeon — линейка серверных микропроцессоров производства Intel

Кэш — (англ. cache) промежуточный буфер с быстрым доступом к нему, содержащий информацию, которая может быть запрошена с наибольшей вероятностью

ПО — Программное обеспечение

Фреймфорк (англицизм, неологизм от framework) — программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

ЦП — Центральный процессор

# ВВЕДЕНИЕ

С появлением многоядерных процессоров сложность приложений постоянно возрастает, так как разработчики стараются использовать весь возможный аппаратный параллелизм для достижения максимальной производительности и уменьшения времени отклика своих приложений. Однако поддержка многопоточности в приложениях зачастую означает усложнение их логики и, как следствие, повышение временных затрат на исправление ошибок и поддержку, поэтому многие разработчики полагаются на прирост производительности, который процессоры получают с каждым новым поколением. В 2002 году компания Intel представила свои первые процессоры с технологией гиперпоточности (hyper-threading, HT) – Xeon и Pentium 4. В режиме HT каждое физическое ядро процессора разделяется на два логических, что, в теории, может увеличить производительность приложения без какого-либо участия разработчиков. Однако оптимизация многопоточных приложений с помощью технологии HT является сложной задачей, требующей глубокого понимания базовой аппаратной архитектуры и дизайна программного обеспечения. Актуальность исследования определяется необходимостью простого метода подключения и получения прироста производительности для различных многопоточных приложений при подключении технологии HT.

Объектом исследования являются многопоточные приложения и технология HT.

Предметом исследования являются методика ускорения многопоточных приложений для использования с технологией HT и алгоритм повышения быстродействия библиотеки QuEST.

Цель диссертации — представить комплексную методологию и алгоритм повышения быстродействия многопоточных приложений для получения максимального прироста эффективности при использовании технологии HT. Исследование сосредоточено на выявлении ключевых факторов, влияющих на производительность многопоточных приложений с HT, а также на разработке эффективных стратегий для устранения любых потенциальных узких мест или неэффективности.

Для достижения этой цели проведен глубокий анализ аппаратной архитектуры и программного обеспечения современных процессоров, уделяя особое внимание технологии HT. Рассмотрена литература по оптимизации многопоточных приложений, изучены различные подходы и методы, предлагаемые для повышения производительности.

На основе проведенного анализа разработана методология и алгоритм оптимизации многопоточных приложений в режиме HT. Эта методология включает оптимизацию как на аппаратном, так и на программном уровне с акцентом на минимизацию накладных расходов, максимизацию параллелизма и балансировку рабочей нагрузки между процессорными ядрами. Оценка методологии и алгоритма основывается на тестах, демонстрирующих прирост производительности.

В соответствии с целью исследования в диссертации необходимо решить следующие задачи:

* аналитический обзор методов и средств реализации многопоточности;
* формализация задачи ускорения многопоточных приложений с использованием технологии гиперпоточности;
* разработка методики ускорения многопоточных приложений;
* разработка алгоритма повышения быстродействия для библиотеки QuEST;
* программная реализация алгоритма повышения быстродействия библиотеки QuEST;
* оценка эффективности разработанного алгоритма для разных типов многопоточных приложений.

Предмет диссертации представляет собой значительный вклад в область компьютерных наук, предоставляя комплексный подход к ускорению многопоточных приложений при использовании технологии HT. Благодаря этому исследованию получена ценная информация и рекомендации разработчикам и исследователям, работающим в этой быстро развивающейся области.

Положения, выносимые на защиту.

1. Формализованное представление оценки быстродействия многопоточного приложения.
2. Методика ускорения многопоточных приложений при использовании технологии гиперпоточности.
3. Алгоритм повышения быстродействия для библиотеки Quantum Exact Simulation Toolkit.
4. Программная реализация разработанного алгоритма.
5. Оценка эффективности разработанного алгоритма ускорения многопоточных приложений.

Научная новизна исследования. В данной магистерской диссертации разработана методика и алгоритм, обеспечивающие уменьшение количество отказов в системе, что приведет к ускорению работы многопоточных приложений.

Обоснованность и достоверность результатов. На основе результатов разработки алгоритма АПБ, повышаемого быстродействия различных многопоточных приложений при работе с технологией HT, была доказана его адекватность. На основании проведенного тестирования можно считать, что разработанная методика и алгоритм являются действительными для применения в реальных условиях.

Практическая значимость. Практическим результатом выполнения данной работы является применение методики ускорения многопоточных приложений при использовании с технологией гиперпоточности и программная реализация алгоритма АПБ, который способен оптимизировать распараллеливание задач приложения, а также уменьшить количество используемой памяти, тем самым повысив скорость работы.

Структура диссертационной работы. Диссертационная работа содержит введение, четыре главы, заключение, список литературы, 5 таблиц, 10 листингов, 15 рисунков. Объем работы составляет 80 страниц.

# ГЛАВА 1. АНАЛИТИЧЕСКИЙ ОБЗОР МЕТОДОВ И СРЕДСТВ РЕАЛИЗАЦИИ МНОГОПОТОЧНОСТИ

## Анализ существующих средств реализации многопоточности

Средство реализации многопоточности POSIX Threads (pthreads) являются стандартным API для создания и управления потоками в многопоточных приложениях. Pthreads предоставляет набор функций для создания, синхронизации и управления потоками. Он широко поддерживается на различных операционных системах и предоставляет мощные механизмы синхронизации, такие как мьютексы, семафоры, условные переменные и механизмы барьеров. Pthreads также обеспечивает портативность между различными аппаратными платформами и операционными системами.

Одним из основных недостатков pthreads является его низкоуровневый характер. При использовании pthreads разработчику необходимо самостоятельно управлять созданием, синхронизацией и уничтожением потоков, что требует более сложного и подробного кодирования, как, например на рис. 1.2.

Pthreads предоставляет различные методы синхронизации, включая мьютексы (mutexes), семафоры (semaphores) и условные переменные (conditional variables). Например, для защиты общего ресурса от параллельного доступа используется мьютекс. Мьютексы позволяют потокам конкурировать за доступ к критической секции кода.

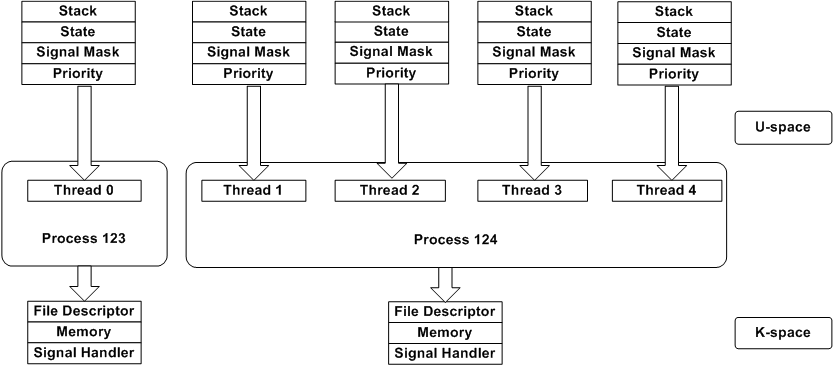


Рисунок 1.1 – Диаграмма создания и распределения потоков в POSIX Threads

Средство реализации многопоточности Java Threads являются средством реализации многопоточности в языке программирования Java. Java предоставляет высокоуровневый API для создания и управления потоками. Классы, такие как Thread и Runnable, позволяют создавать и запускать потоки. Java также предоставляет механизмы синхронизации, такие как synchronized блоки и методы, для обеспечения безопасности доступа к общим ресурсам. Кроме того, Java предоставляет пулы потоков (Thread Pools) для эффективного управления пулом предварительно созданных потоков и выполнения асинхронных задач.

Основным недостатком Java Threads является отсутствие возможности полного контроля над планированием потоков. Планирование потоков в Java полностью зависит от виртуальной машины Java (JVM) и операционной системы, что может привести к непредсказуемому поведению приложений при работе с многопоточностью.

Для создания потока можно создать класс, реализующий интерфейс Runnable, и передать его экземпляр в конструктор класса Thread. Затем вызывается метод start(), который запускает выполнение потока. Для синхронизации потоков используются ключевые слова synchronized и volatile, а также классы-объекты, такие как Lock и Condition. Например, для синхронизации доступа к общему ресурсу используется ключевое слово synchronized. Все методы и состояния потоков можно посмотреть на рис. 1.2.

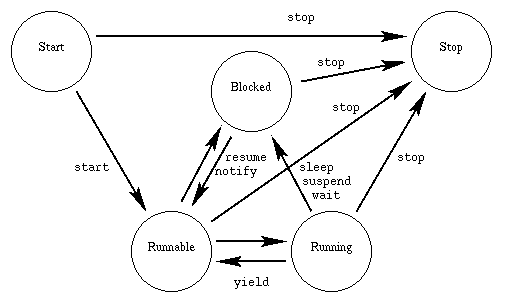


Рисунок 1.2 – Java Threads методы и состояния потоков

Средство реализации многопоточности OpenMP является интерфейсом программирования для разработки параллельных приложений на основе разделения задач между потоками. Он предоставляет директивы компилятора, которые позволяют указывать параллельные участки кода и задавать способы разделения работы между потоками. OpenMP поддерживается в различных языках программирования, включая C, C++ и Fortran. Он упрощает разработку параллельных приложений путем автоматического управления созданием и уничтожением потоков, а также предоставляет механизмы синхронизации, такие как параллельные циклы и секции.

Важным недостатком OpenMP является его ограничение использования в рамках одной системы или узла. OpenMP не предоставляет механизмы для работы с распределенными системами, что ограничивает его применение в приложениях, требующих параллельного выполнения на нескольких узлах или серверах.

OpenMP предоставляет директиву #pragma omp parallel for, которая позволяет распараллеливать выполнение циклов. Эта директива автоматически разделяет итерации цикла между потоками. Схема механизма параллелизма в OpenMP приведена на рис. 1.3.

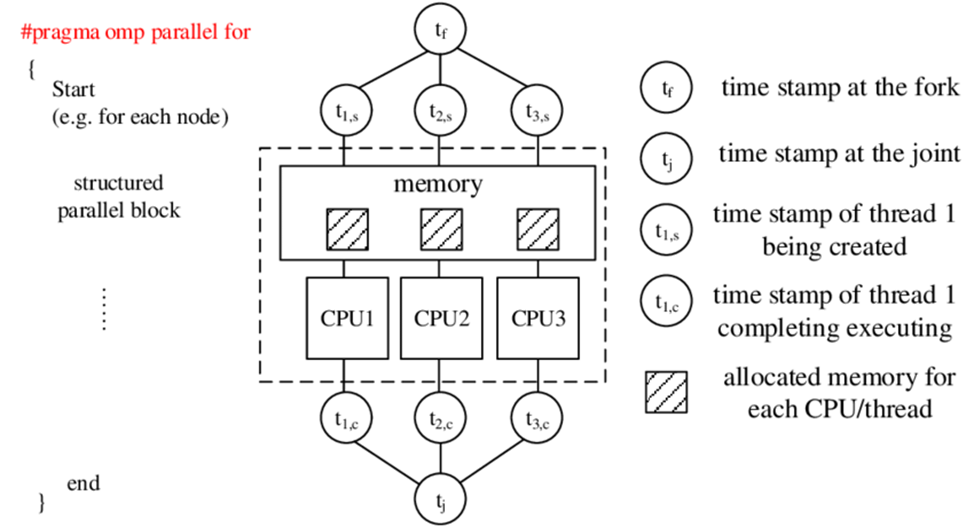


Рисунок 1.3 – Механизм параллелизма в OpenMP с несколькими потоками ЦП

Директива #pragma omp parallel sections позволяет разделять код на независимые секции, которые могут выполняться параллельно. Каждая секция выполняется в отдельном потоке. Формула использования параллельных секций представлена на рис.1.4.

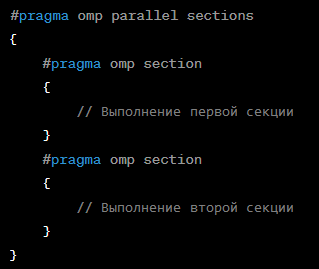


Рисунок 1.4 – Пример использования параллельных секций в OpenMP

Средство реализации многопоточности .NET Parallel Extensions предоставляет средства для разработки многопоточных приложений в экосистеме Microsoft .NET. Он включает в себя классы и методы для создания и управления потоками, а также механизмы синхронизации, такие как блокировки, семафоры и сигналы. Parallel Extensions также предоставляет высокоуровневые конструкции, такие как планировщики задач (TaskSchedulers) и параллельные циклы (Parallel For/ForEach), которые позволяют легко выражать параллельные операции и автоматически распределять работу между потоками.

Основным недостатком .NET Parallel Extensions является его привязка к платформе .NET. Это ограничивает его использование только в среде .NET и на платформах, поддерживаемых .NET Framework.

В .NET Parallel Extensions для параллельного выполнения циклов используется метод Parallel.For(), который автоматически разделяет итерации цикла между доступными потоками.

Средство реализации многопоточности: Intel Threading Building Blocks (TBB) является библиотекой параллельного программирования, предоставляющей высокоуровневые абстракции и инструменты для создания параллельных приложений. TBB предоставляет контейнеры данных и алгоритмы, которые могут автоматически разбивать задачи на более мелкие и выполнять их параллельно на нескольких потоках. TBB также обеспечивает механизмы синхронизации, такие как мьютексы и блокировки, для обеспечения безопасности доступа к общим ресурсам. Он хорошо интегрируется с различными компиляторами и аппаратными платформами.

TBB предоставляет набор высокоуровневых алгоритмов, которые автоматически разбивают задачи на более мелкие и распределяют их выполнение между доступными потоками. Например, parallel\_for позволяет параллельно выполнять циклы, а parallel\_reduce позволяет параллельно выполнять операции сокращения (reduction). Одним из подобных алгоритмов представлен на рис. 1.5.

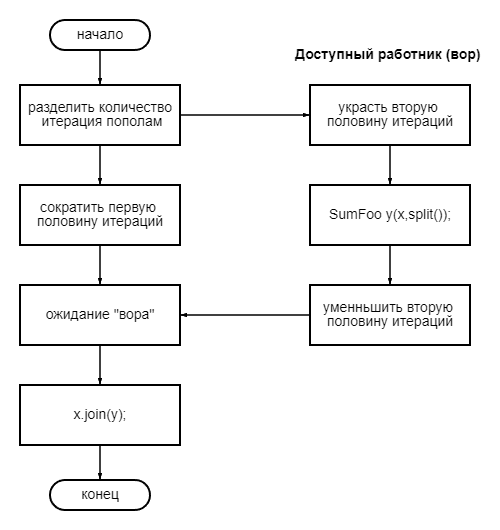


Рисунок 1.5 – Пример работы параллельного алгоритма кражи работы в TBB

TBB позволяет создавать пользовательские планировщики задач, которые управляют выполнением потоков и распределением задач. Планировщики задач позволяют более гибко контролировать и оптимизировать выполнение параллельных операций.

## Анализ методов и алгоритмов повышения производительности многопоточных приложений

Технология распараллеливания задач уже не нова, но ее все чаще используют в современном мире. Множество методов, позволяющих повысить быстродействие многопоточного приложения, можно классифицировать следующим образом:

Методы распределения задач (Task Scheduling). Суть метода заключается в разбивании приложения на набор задач, каждая из которых может выполняться на отдельном виртуальном ядре процессора. Затем задачи распределяются между доступными ядрами, чтобы максимизировать использование процессора. Этот метод особенно полезен при использовании технологии гиперпоточности, так как позволяет эффективно использовать множество ядер процессора.

Рассмотрим следующие из методов распределения задач:

* Разделение на задачи фиксированного размера (Fixed-Size Task Partitioning). Задачи разделяются на фиксированные части, которые затем распределяются между потоками. Это может быть полезно, если задачи являются однородными и могут быть легко разделены на равные части. Каждый поток затем обрабатывает свою часть задачи независимо от других потоков. Примером использования этого метода может служить расчет матричного произведения. В этом случае матрица разбивается на фиксированные части, которые затем обрабатываются параллельно каждым потоком.
* Разделение на задачи переменного размера (Variable-Size Task Partitioning). Ключевой идеей этого метода является динамическое выделение задач в зависимости от доступности ресурсов процессора. Это позволяет более эффективно использовать вычислительные ресурсы, учитывая различные потоки выполнения и возможность масштабирования. Для реализации этого метода используются алгоритмы распределения задач, которые могут быть основаны на различных стратегиях. Например, одной из таких стратегий может быть распределение задач на основе графа зависимостей. В этом случае, каждая подзадача связана с другой подзадачей, и выполнение одной задачи может быть зависимым от выполнения другой задачи.

Методы распределения данных (Data Scheduling). В этом подходе данные разбиваются на более мелкие блоки, которые могут обрабатываться параллельно на разных виртуальных ядрах процессора. Таким образом, каждое ядро обрабатывает свой блок данных, что позволяет сократить время выполнения приложения.

Методы комбинирования задач и данных (Task and Data Scheduling). Некоторые алгоритмы объединяют подходы распределения задач и данных для достижения максимальной эффективности при использовании гиперпоточности. В этом подходе приложение разбивается на набор задач, каждая из которых имеет свой набор данных. Затем задачи и данные распределяются между доступными виртуальными ядрами процессора таким образом, чтобы минимизировать ожидание взаимодействия между ними. Такой подход позволяет достигнуть максимальной эффективности при использовании гиперпоточности, поскольку каждое виртуальное ядро обрабатывает свою задачу и ее данные параллельно с другими ядрами.

Методы синхронизации и координации потоков (Thread Synchronization and Coordination). Когда несколько потоков обращаются к одному ресурсу одновременно в многопоточном приложении, это приводит к проблемам синхронизации и координации, которые в свою очередь замедляют работу приложения или приводят к ошибкам. Методы синхронизации и координации потоков позволяют решать эти проблемы, обеспечивая правильную работу приложения и повышая его быстродействие при использовании технологии гиперпоточности. Рассмотрим основные из них:

Использование мьютексов(mutexes) и семафоров(semaphores). Мьютексы - это объекты синхронизации, которые блокируют доступ к защищенному ресурсу только для одного потока в конкретный момент времени. Семафоры - это объекты синхронизации, которые позволяют контролировать количество потоков, которые могут получить доступ к защищенному ресурсу одновременно. Пример их использования представлен на рис.1.6.

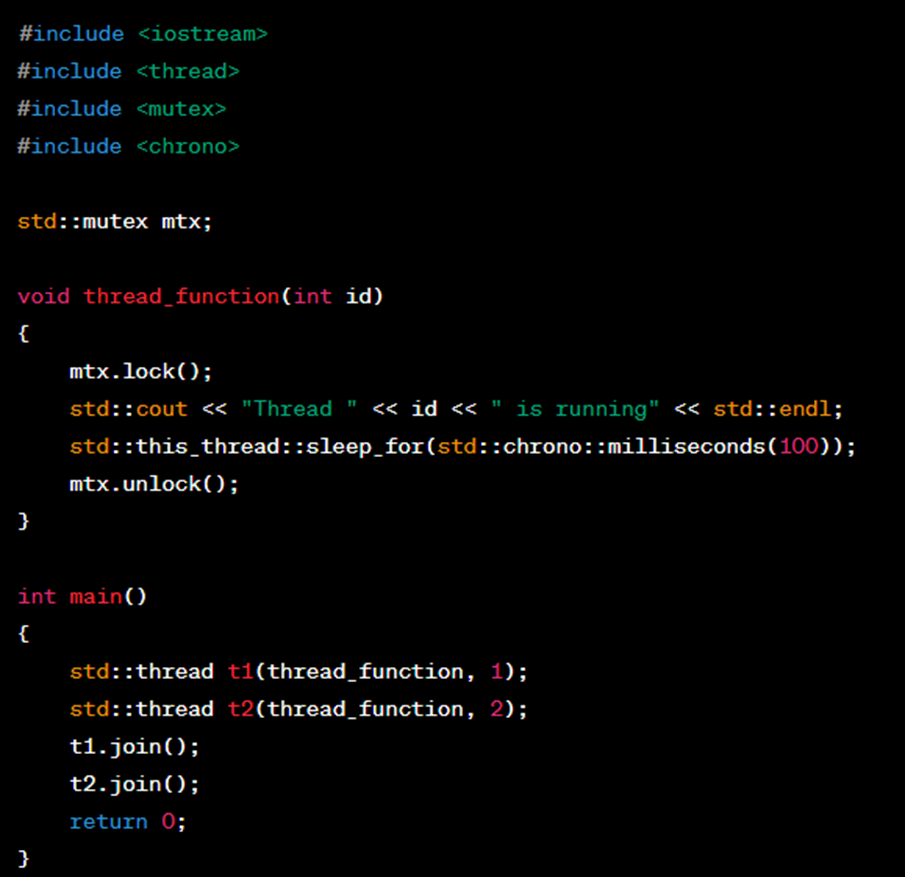


Рисунок 1.6 – Пример использования мьютексов и семафоров.

В примере на рис. 1.6. создаются два потока, которые выполняют thread\_function(). Перед выполнением функции каждый поток блокирует мьютекс mtx, чтобы защитить вывод на экран от конфликтов с другим потоком. После выполнения функции каждый поток разблокирует мьютекс. Таким образом, мьютекс позволяет синхронизировать доступ к ресурсу между несколькими потоками.

Использование условных переменных (Condition Variables). Условные переменные используются для блокировки потока, пока какое-либо условие не будет выполнено, что позволяет синхронизировать выполнение нескольких потоков в зависимости от состояния приложения. Условные переменные представляют собой синхронизационный механизм, который позволяет потокам ждать на определенном условии и продолжить выполнение, когда это условие выполнено. Такое поведение позволяет эффективно использовать ресурсы процессора и уменьшить нагрузку на него. Ярким примером применения такого метода является сценарий производителя-потребителя (producer-consumer). В этом сценарии один поток (производитель) генерирует данные и помещает их в буфер, а другой поток (потребитель) извлекает данные из буфера и использует их. Если буфер пуст, потребитель должен ждать, пока производитель не добавит новые данные в буфер. Если буфер заполнен, производитель должен ждать, пока потребитель не извлечет данные из буфера. Чтобы реализовать такой сценарий можно использовать условные переменные. Когда потребитель обнаруживает, что буфер пуст, он вызывает функцию ожидания на условной переменной, связанной с буфером. Это вызывает блокировку потока и освобождение блокировки на мьютексе, который защищает буфер. Когда производитель добавляет новые данные в буфер, он оповещает потребителя, вызывая функцию сигнала на условной переменной. Это разблокирует поток потребителя и позволяет ему продолжить выполнение. Пример использования условных переменных изображен на рис.1.7.

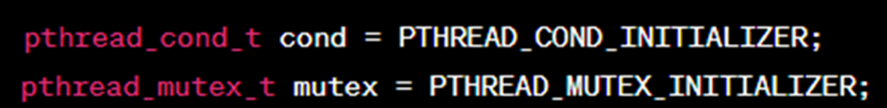


Рисунок 1.7 – Пример использования условных переменных

Функции, которые обычно используются вместе с условными переменными:

• pthread\_cond\_init: инициализирует условную переменную.

• pthread\_cond\_wait: блокирует поток на условной переменной, пока другой поток не оповестит о ее изменении.

• pthread\_cond\_signal: оповещает один из потоков, ожидающих на условной переменной, что условие, на котором они ждали, было изменено.

• pthread\_cond\_broadcast: оповещает все потоки, ожидающие на условной переменной, что условие, на котором они ждали, было изменено.

• pthread\_cond\_destroy: уничтожает условную переменную.

Методы межпоточной коммуникации (Inter-Thread Communication). Суть метода заключается в том, что различные потоки обмениваются данными и сообщениями, чтобы выполнить задачу. Для этого могут использоваться специальные механизмы, такие как очереди сообщений (message queues), пайпы (pipes), сокеты (sockets) и другие.

Методы оптимизации памяти (Memory Optimization). Как правило, многопоточные приложения потребляют большое количество памяти и сталкиваются с проблемами производительности, связанными с частым использованием памяти. Именно поэтому существуют методы, позволяющие ускорить многопоточные приложения, уменьшая количество обращений к памяти и улучшению кэширования данных. В действия таких методов входит использование локальных копий данных (local copies) и оптимизация выделения памяти (memory allocation optimization). Есть несколько известных методов оптимизации памяти:

* Снижение использования кучи (Heap Reduction). Куча - это область памяти, используемая для хранения объектов в программе. Передача объектов между потоками может привести к дублированию объектов в куче, что может привести к избыточному использованию памяти. Для уменьшения использования кучи можно использовать передачу по ссылке, где ссылка на объект передается между потоками вместо самого объекта.
* Испольщование локальных переменных (Local Variables). Локальные переменные создаются и удаляются в каждом потоке отдельно, что снижает использование памяти. Если объект создается и используется только внутри одного потока, то лучше использовать локальную переменную.
* Кэширование (Caching). Кэширование - это сохранение данных в памяти на короткое время для повторного использования. Если данные не меняются между потоками, то их можно закэшировать и повторно использовать, что позволит снизить использование памяти.

## Технология гиперпоточности

Постоянно растущий спрос на вычислительную мощность привел к разработке многоядерных процессоров, которые позволяют параллельно выполнять множество задач. Однако даже при использовании нескольких ядер часто возникает больше задач, чем может быть выполнено одновременно, что приводит к узким местам в производительности. Технология HT обеспечивает решение этой проблемы, позволяя нескольким потокам выполняться одновременно на одном ядре. Будем рассматривать HT как средство повышения производительности многопоточных приложений.

Технология HT была представлена Intel в 2002 году как средство повышения производительности своих процессоров Pentium 4. Это аппаратная функция, позволяющая создавать логические процессоры внутри физического процессорного ядра. Каждый логический процессор имеет свой собственный набор регистров и исполнительных блоков, что позволяет нескольким потокам выполняться одновременно на одном ядре.

Технология HT реализована за счет использования нескольких контекстов выполнения в пределах одного процессорного ядра. Каждый контекст состоит из набора регистров и исполнительных блоков, которые обеспечивают одновременное выполнение инструкций из нескольких потоков. Эти контексты являются виртуальными и представляют собой не физическое ядро процессора, а скорее логическое разделение ресурсов внутри ядра.

Когда ядро процессора выполняет поток, оно может столкнуться с ситуацией, когда ему необходимо дождаться завершения данных или инструкции, прежде чем он сможет продолжить. В процессоре без HT ядро в течение этого времени простаивало бы, тратя впустую ценное время обработки. С помощью HT ядро может переключиться на выполнение потока в другом контексте во время ожидания, что позволяет более эффективно использовать доступные ресурсы.

Технология HT может значительно повысить производительность многопоточных приложений, которые разработаны с учетом преимуществ параллелизма. В этих приложениях HT позволяет одновременно выполнять еще больше потоков, что приводит к повышению общей производительности. HT особенно эффективна для повышения производительности приложений, рассчитанных на использование нескольких ядер. Схема работы технологии HT приведена на рис. 1.8.

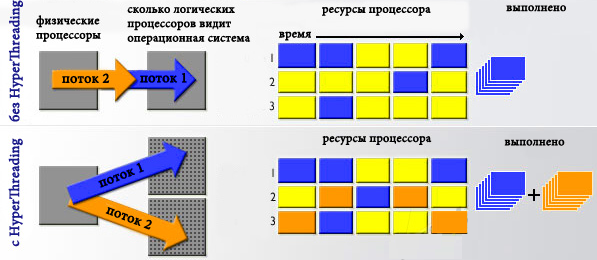


Рисунок 1.8 – Схематическое представление работы технологии HT

Одним из примеров приложения, которое может извлечь выгоду из HT, является ПО для кодирования видео. Кодирование видео требует большой параллельной обработки и, как правило, является трудоемкой задачей. С помощью HT программное обеспечение для кодирования видео может разделить работу на несколько потоков, которые могут выполняться одновременно в нескольких контекстах в пределах одного ядра. Это приводит к более быстрому времени кодирования и повышению общей производительности.

ПО для управления базами данных также может получать значительный прирост производительности от HT. Т.к. такое ПО часто предполагает одновременный доступ к базе данных нескольких пользователей, это может привести к узким местам в производительности. С помощью HT ПО может разделить рабочую нагрузку на несколько потоков, которые могут выполняться одновременно в нескольких контекстах в пределах одного ядра, что приведет к более быстрому времени отклика и повышению общей производительности.

Хотя технология HT может значительно повысить производительность многопоточных приложений, она не является панацеей. Приложения, которые не оптимизированы для многопоточности, могут фактически столкнуться со снижением производительности из-за накладных расходов на управление дополнительными контекстами. Кроме того, HT также может увеличить энергопотребление и тепловыделение, что в некоторых ситуациях может быть нежелательно.

Еще одним из ограничений HT является потенциальная конкуренция за ресурсы. Если несколько потоков конкурируют за одни и те же ресурсы внутри ядра, такие как кэш или исполнительные модули, производительность может быть снижена. Важно изначально тщательно проектировать многопоточные приложения, чтобы избежать конкуренции за ресурсы и максимизировать преимущества HT. Тем не менее, это такая проектировка усложняет логику приложения, и, как следствие, повышает временные затраты на исправление ошибок и поддержку.

Именно поэтому разработка алгоритма, позволяющего получать наибольший прирост производительности с помощью HT, актуальна в наше время.

## Обоснование метода решения

Многопоточные приложения стали неотъемлемой частью современных вычислительных систем, обеспечивая параллельное выполнение задач и эффективное использование процессорных ресурсов. Технология Hyper-Threading (HT) предоставляет возможность еще больше повысить производительность этих приложений за счет использования возможностей современных процессоров.

При разработке методики и алгоритма, особое внимание будет уделено использованию возможностей технологии HT, т.к. она позволяет одному физическому процессорному ядру выполнять несколько потоков одновременно, эффективно используя незанятые ресурсы. Однако для того, чтобы в полной мере воспользоваться преимуществами HT, требуется системный подход. Разработав специальную методику и алгоритм, мы можем эффективно использовать параллелизм, предлагаемый HT, повышая производительность и результативность многопоточных приложений.

Ключевым аспектом ускорения многопоточных приложений с помощью HT является эффективное планирование потоков. Разработка методики и алгоритма, учитывающих базовую аппаратную архитектуру и специфические характеристики технологии HT, позволяет оптимально распределить потоки по логическим процессорам. Тщательно управляя назначениями потоков и балансируя рабочую нагрузку между логическими процессорами, мы можем свести к минимуму конфликты и максимально использовать доступные ресурсы выполнения, тем самым повышая общую производительность.

Хорошо разработанная методика и алгоритм могут устранить дисбаланс нагрузки, который может возникнуть при запуске многопоточных приложений с помощью HT. Динамически отслеживая рабочую нагрузку и перераспределяя потоки между логическими процессорами, можно гарантировать, что каждый логический процессор получает достаточную и равную долю вычислительных задач. Выбирая стратегию балансировки нагрузки, мы предотвращаем перегрузку определенных логических процессоров и устраняем узкие места в ресурсах, тем самым оптимизируя использование технологии HT.

Разработка алгоритма, который дополнит библиотеку QuEST сможет адаптироваться к различным характеристикам многопоточных приложений и рабочей нагрузке, что имеет решающее значение. Постоянно отслеживая состояние системы и регулируя количество активных потоков и их распределение по логическим процессорам, методология и алгоритм могут динамически реагировать на изменения в вычислительных требованиях. Такая адаптивность обеспечивает эффективное использование ресурсов и обеспечивает оптимальную производительность при различных условиях рабочей нагрузки.

Предполагаемые методология и алгоритм включают методы профилирования производительности для выявления потенциальных узких мест и областей для оптимизации. Анализируя поведение приложения, закономерности использования ресурсов и влияние технологии HT, мы можем точно настроить алгоритм для устранения конкретных ограничений производительности. Этот итеративный процесс оптимизации помогает максимально использовать преимущества технологии HT и еще больше повысить общую производительность многопоточных приложений.

Создание специальной методики ускорения многопоточных приложений и алгоритма повышения быстродействия для библиотеки QuEST, работающих с использованием технологии HT, оправдано необходимостью в полной мере использовать параллелизм, предлагаемый современными процессорами. Сосредоточив внимание на оптимизированном планировании потоков, балансировке нагрузки, адаптивной многопоточности и профилировании производительности. Предлагаемый подход позволяет эффективно использовать технологию HT, что приводит к значительному повышению производительности и более эффективному использованию ресурсов.

# Выводы к главе 1

В ходе исследования были произведены следующие работы:

* поставлена цель и задачи исследования;
* определен объект исследования;
* произведен аналитический обзор существующих решений;
* проанализированы основные направления разработки;
* обоснована идея создания методики ускорения многопоточных приложений при использовании технологи HT и алгоритма повышения быстродействия для библиотеки QuEST.

# ГЛАВА 2. ФОРМАЛИЗОВАННОЕ ПРЕДСТАВЛЕНИЕ ПРОЦЕССА ОБРАБОТКИ ПРЕРЫВАНИЙ

## Формализованное представление задачи ускорения многопоточных приложений при использовании технологии гиперпоточности

Для формализации задачи ускорения многопоточных приложений при использовании технологии HT определены параметры, позволяющие проанализировать эффективность моделируемой системы и выбрана целевая функция:

В качестве основных показателей эффективности работы алгоритма выбраны следующие параметры:

* P — прирост производительности при использовании HT, показывающий, насколько увеличилась производительность многопоточного приложения при использовании алгоритма АПБ и технологии HT;
* — эффективность ядра при однопоточном запуске;
* — эффективность ядра в режиме HT.

Целью алгоритма АПБ является максимизация кортежа P и , это и будет целевая функция:

|  |  |
| --- | --- |
|  | (1) |

* w1 и w2 — весовые коэффициенты для параметров P и соответственно;
* и — нормализованные значения параметров P и .

Параметр P является одной из ключевых целей применения этой технологии. Максимизация P покажет максимальное увеличение производительности многопоточного приложения, что имеет значительный практический эффект. Весовой коэффициент w1 позволяет установить относительную важность P в целевой функции.

Параметр измеряет эффективность ядра при выполнении нескольких задач одновременно с использованием HT. Максимизация указывает на эффективное использование HT и эффективность работы многопоточного приложения. Весовой коэффициент w2 позволяет установить относительную важность в целевой функции.

Целевая функция f(x) сочетает оба показателя P и , учитывая их вклады в оценку эффективности системы. Такой подход позволяет сбалансированно учитывать, как прирост производительности, так и эффективность в режиме HT при выборе наилучшей конфигурации системы. Весовые коэффициенты w1 и w2 позволяют настраивать относительную важность каждого показателя в соответствии с требованиями и приоритетами.

Экспериментальный подход позволяет оценить влияние каждого параметра на систему и определить оптимальные значения весовых коэффициентов.

Если прирост производительности (P) и эффективность в режиме HT () считаются одинаково важными, можно выбрать равные весовые коэффициенты w1 = w2 = 0.5. Это означает, что оба параметра равнозначно влияют на целевую функцию, и система стремится достичь баланса между приростом производительности и эффективностью в режиме HT.

Так как прирост производительности является основной целью системы, можно выбрать более высокий весовой коэффициент для P. Например, w1 = 0.7 и w2 = 0.3. Если эксперименты покажут, что для конкретных многопоточных приложений невозможно достигнуть прироста производительности за счет применения технологии HT, то уклон будет в достижение максимального прироста производительности за счет алгоритма при использовании того прироста от HT, который возможен.

Если эффективность в режиме HT очень высока, то она будет имеет большее значение, тогда можно выбрать более высокий весовой коэффициент для . Например, w1 = 0.3 и w2 = 0.7. Это указывает на больший уклон системы в достижение максимальной эффективности в режиме гиперпоточности.

Можно рассматривать использование динамического подхода, где весовые коэффициенты могут изменяться при подключении библиотеки QuEST, в которую будет интегрирован алгоритм АПБ, к многопоточному приложению в зависимости от текущих требований и условий. Такой подход позволит адаптироваться к изменяющимся потребностям и настроениям пользователей.

Нормализованные значения параметров P и E\_HT вычисляются с использованием следующих формул:

|  |  |
| --- | --- |
|  | (2) |
|  | (3) |

Нормализация параметров P и осуществляется для установления сопоставимости между ними и обеспечения их одинакового влияния на целевую функцию. Нормализация выполняется путем вычитания среднего значения параметра и деления на стандартное отклонение.

Нормализация необходима, т.к. P и могут иметь разные диапазоны значений, и их абсолютные значения могут сильно отличаться. Нормализация позволяет привести их к относительной шкале, где значения будут близки к нулю с малым стандартным отклонением, что облегчает сравнение их вклада в целевую функцию.

* — среднее значение прироста производительности при использовании гиперпоточности;
* — среднее значение эффективности ядра в режиме гиперпоточности;
* и — стандартное отклонение.

Для расчета значений параметров P и используются следующие формулы:

|  |  |
| --- | --- |
|  | (4) |
|  | (5) |

* — счетчик циклов, выполненных на ядре при однопоточном запуске;
* — счетчик циклов, выполненных на ядре в режиме гиперпоточности;
* — счетчик удаленных ядром инструкций для каждого потока в режиме гиперпоточности;
* — параметр энтропии для P;
* — параметр энтропии для .

Для выбора адекватных значений и есть несколько подходов:

* провести анализ исходных данных или результатов экспериментов. Например, исследовать распределение прироста производительности и эффективности в режиме HT, чтобы определить адекватный уровень энтропии для нормализации.
* предварительно выбрать значения для и , а затем анализировать результаты и корректировать эти значения в процессе экспериментов. Это позволит оптимизировать процесс нормализации и адаптировать значения параметров под конкретные условия и требования.
* можно использовать стандартные значения для и , например, значения 0.2 и 0.3.

На основе анализа исходных данных, решено использовать значения энтропии на основе распределения прироста производительности и значения эффективности в режиме HT.

## Разработка методики ускорения многопоточных приложений при использовании с технологией гиперпоточности

Основные этапы разработанной методики ускорения многопоточных приложений при использовании с технологией гиперпоточности.

Таблица 2.1 – Методика алгоритма формирования масок прерываний

|  |  |
| --- | --- |
| Этап 1. Анализ и разделение задач   1. Идентификация задач, которые могут быть распараллелены. 2. Анализ зависимостей данных и возможностей параллельного выполнения. 3. Разбиение задач на более мелкие подзадачи. | Этап 4. Оптимизация кэширования данных   1. Идентификация данных, которые часто используются и могут быть закэшированы. 2. Определение оптимального размера кэш-линий для каждого потока. 3. Минимизация кэш-промахов путем организации локальности данных. |

Продолжение таблицы 2.1.

|  |  |
| --- | --- |
| Этап 2. Распараллеливание с использованием HT   1. Определение количества потоков, учитывая доступные ядра процессора и степень параллелизации задач. 2. Создание потоков для выполнения подзадач. 3. Использование технологии HT для создания дополнительных виртуальных потоков внутри каждого физического ядра. | Этап 5. Управление потоками и планирование   1. Разработка механизмов управления потоками, включая планирование выполнения задач. 2. Распределение нагрузки между потоками для равномерного использования ресурсов процессора. 3. Управление приоритетами выполнения задач для обеспечения оптимальной производительности. |
| Этап 3. Асинхронные операции и управление задачами   1. Определение операций, которые могут быть выполнены асинхронно и независимо. 2. Организация асинхронных операций с помощью механизмов, таких как обратные вызовы, промисы(promise) или фьючерсы (future). 3. Управление задачами путем планирования и контроля потоков выполнения. | Этап 6. Межпоточная коммуникация и синхронизация   1. Идентификация точек синхронизации и межпоточной коммуникации. 2. Реализация механизмов синхронизации, таких как блокировки, семафоры или условные переменные. 3. Обеспечение правильного доступа к общим ресурсам и предотвращение гонок данных. |

Рассмотрим каждый этап разработанной методики отдельно.

Этап 1. Сначала необходимо определить задачи, которые возможно выполнить параллельно. Это могут быть независимые вычислительные задачи или задачи, которые могут быть разделены на независимые подзадачи. Обозначим множество задач как , где каждая задача представляет собой отдельную задачу, которая может быть выполнена параллельно.

Далее необходимо определить зависимости между задачами и данными, используя алгоритмы анализа графов, такие как обход графа в глубину или топологическую сортировку. Это поможет идентифицировать зависимости, которые могут потребоваться для правильного выполнения задач. Обозначим матрицу зависимостей как D, где D[i][j] = 1, если задача зависит от задачи , и D[i][j] = 0 в противном случае.

Разделяем каждую задачу на более мелкие подзадачи, которые возможно выполнить независимо, с помощью алгоритмов декомпозиции, таких как разбиение на блоки или разбиение на сегменты. Обозначим множество подзадач для каждой задачи как , где каждая подзадача представляет собой часть задачи , которую можно выполнить независимо. Схематичное изображение алгоритма разбиение на сегменты представлено на рис. 2.1.

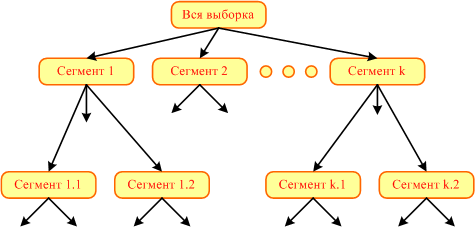


Рисунок 2.1 — Алгоритм разбиения на сегменты

Этап 2. Определим оптимальное количество потоков, учитывая доступные ядра процессора и степень параллелизации задач. Обозначим общее количество доступных ядер процессора как CN. Обозначим степень параллелизации задач как SP Оптимальное количество потоков, которые следует создать, можно рассчитать с использованием следующей формулы:

|  |  |
| --- | --- |
|  | (6) |

Создадим N потоков выполнения, которые будут выполнять подзадачи параллельно. Обозначим множество потоков выполнения как .

Теперь применим технологию HT для создания дополнительных виртуальных потоков внутри каждого физического ядра. Обозначим количество виртуальных потоков для каждого физического ядра как M, а количество ядер как core\_count.

|  |  |
| --- | --- |
|  | (7) |

Этап 3. Сначала идентифицируем операции, которые могут быть выполнены асинхронно и независимо друг от друга. Обозначим эти операции как .

Используем такие механизмы, как обратные вызовы, promise или future, для управления асинхронными операциями. С помощью механизмов асинхронности запускаем операцию и продолжаем выполнение других задач, не ожидая ее завершения.

Применяем планирование и контроль потоков выполнения для эффективного управления задачами, в том числе с помощью планировщиков задач или фреймворков для управления потоками выполнения и установки приоритетов задач.

Этап 4. Определим данные, которые часто используются в задачах и могут быть закэшированы. Обозначим их как .

Кэш-линия представляет собой блок данных, которые загружаются из памяти в кэш. Обозначим размер кэш-линии как B (в байтах). Оптимальный размер кэш-линии может быть рассчитан с использованием следующей формулы:

|  |  |
| --- | --- |
|  | (8) |

Где:

* S - размер кэша (в байтах).
* C - количество кэш-линий в кэше.
* El\_count - количество элементов данных, которые можно сохранить в одной кэш-линии.

Организовываем данные с учётом их использования для обеспечения локальности и минимизации кэш-промахов. Одна из стратегий — использование принципа локальности времени и пространства, при котором данные, к которым обращаются вместе, сохраняются близко друг к другу в памяти.

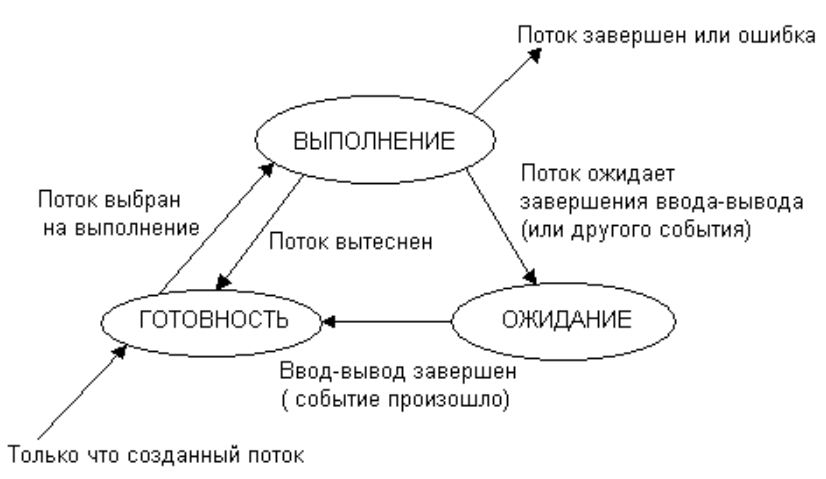
 Этап 5. Реализуем механизмы управления потоками выполнения, такие как создание, запуск, приостановка и завершение потоков. Механизмы могут быть реализованы с помощью многопоточных библиотек или фреймворков. На рис. 2.2. изображен граф состояний потока, отображающий основные возможные его пути при управлении им.

Рисунок 2.2 — Граф состояний потока в многозадачной среде

Распределяем нагрузки между потоками для равномерного использования ресурсов процессора. Устанавливаем приоритеты выполнения задач для обеспечения оптимальной производительности.

Этап 6. Сначала определим участки кода, где потоки должны синхронизироваться для правильного выполнения операций или обмена данными. Далее проводим идентификацию общих ресурсов, к которым потоки должны обращаться с синхронизацией.

Выбираем и реализуем подходящие механизмы синхронизации, такие как блокировки, семафоры, условные переменные и другие. Они обеспечат правильный доступ к общим ресурсам и предотвратят гонки данных между потоками. На рис. 2.3. изображен пример гонки между потоками А и B, пытающимися получить одни и те же данные.

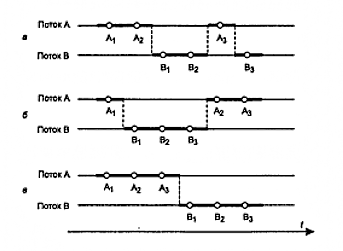


Рисунок 2.3 — Возникновение гонок при доступе к общим ресурсам

Каждый из этих подпунктов представляет конкретные шаги и решения, которые стали частью разработанной методики, объединив в себе лучшие подходы для ускорения многопоточных приложений при использовании технологии HT.

# Разработка алгоритма повышения быстродействия библиотеки QuEST

Стараясь повысить быстродействие многопоточного приложения с помощью алгоритма, необходимо учесть его архитектуру и свойства. При этом, применив такой алгоритм к другому приложению, мы можем быть уверены, что он будет эффективен настолько же. Алгоритм может не повлиять на скорость работы приложения или даже ее ухудшить. Поэтому было принято решение внедрить универсальный алгоритм в существующую библиотеку QuEST, чтобы при ее подключении в многопоточное приложение, прирост производительности, даже небольшой, но всегда был. Схема разрабатываемого алгоритма АПБ представлена на рис. 2.4.

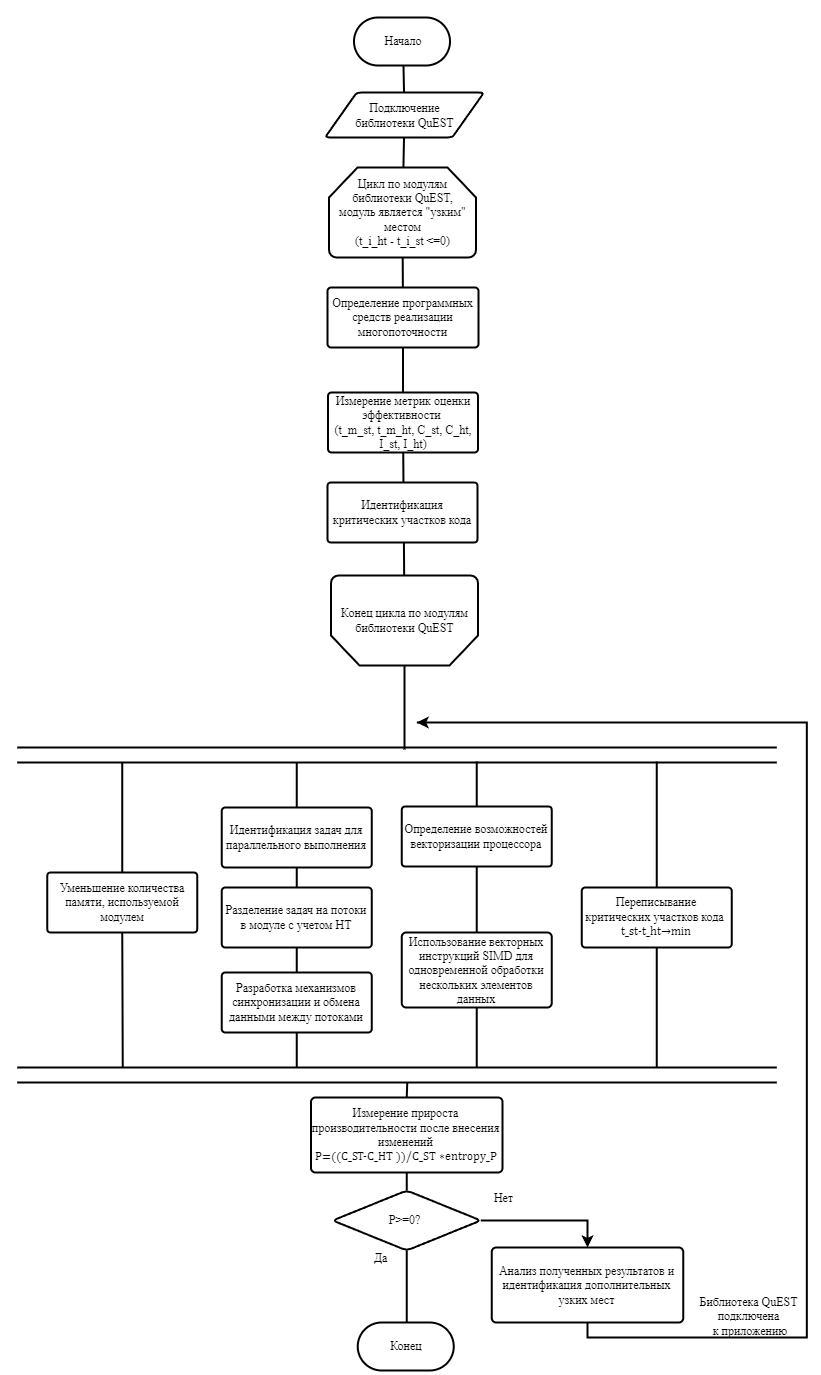


Рисунок 2.4 — Алгоритм повышения быстродействия библиотеки QuEST

Подробно рассмотрим каждый шаг алгоритма:

Шаг 1: Проанализировав все модули библиотеки QuEST, определяем уже подключенные программные средства реализации многопоточности.

Шаг 2: Измеряем метрики оценки эффективности библиотеки QuEST. рассчитывались метрики производительности приложения, такие как – прирост производительности при использовании гиперпоточности; – эффективность ядра при однопоточном запуске; – эффективность ядра в режиме гиперпоточности [1] согласно следующим формулам:

|  |  |
| --- | --- |
| , | (9) |
| , | (10) |
| , | (11) |

* где (Unhalted core cycles) – счетчик циклов, выполненных на ядре при однопоточном запуске;
* (Unhalted core cycles) – счетчик циклов, выполненных на ядре в режиме гиперпоточности;
* (Instructions retired) – счетчик удаленных ядром инструкций при однопоточном запуске;
* (Instructions retired) – счетчик удаленных ядром инструкций для каждого потока в режиме гиперпоточности.

Шаг 3: Используя профилировщик и другие специализированные инструменты, проводим сбор данных о работе библиотеки QuEST в многопоточном приложении с использованием HT. Профилирование дает информацию о времени выполнения каждой функции, количестве вызовов функций, использовании памяти и других метриках производительности. Получаем время работы каждого используемого модуля с использованием технологии HT — и без — .

Анализируя профилировочные данные, выявляем участки кода, которые занимают значительное время выполнения или являются узким местом производительности. Это могут быть функции, циклы, операции с памятью или другие участки кода, где происходит значительное количество вычислений или задержек.

Определяем, какие участки кода имеют наибольшее влияние на общую производительность библиотеки QuEST, основываясь на времени выполнения, количестве вызовов или других показателях профилирования.

Дальнейшие шаги выполняются параллельно:

Шаг 4: Для начала анализируем используемую память внутри модулей библиотеки QuEST, которые являются узким местом. Оцениваем объем памяти, выделяемой для различных структур данных, временных буферов, массивов и других ресурсов, используемых внутри библиотеки.

Выявляем участки кода, структуры данных или алгоритмы, которые занимают значительное количество памяти. Это могут быть большие массивы данных, ненужные временные буферы, избыточные структуры данных или другие ресурсы, которые можно оптимизировать для снижения использования памяти. Производим анализ и оптимизацию. Сокращаем размер структур, уменьшаем количество хранимых данных, перерабатываем алгоритмы, уменьшаем размеры буферов, перераспределяем память, реализуем более эффективное использование временных ресурсов, используем алгоритмы с меньшими требованиями к памяти.

Идентифицируем и исправляем потенциальные утечки памяти внутри каждого модуля. Проверяем освобождение памяти после использования, устранение циклических ссылок или других проблемы, которые могут приводить к неправильному использованию памяти.

Шаг 5.1: Анализируем задачи, выполняемые внутри многопоточного приложения с использованием библиотеки QuEST. Исследуем зависимости между задачами, а также потенциальные участки кода, которые могут быть параллельно выполнены. Определяем, какие задачи могут быть выполнены параллельно без возникновения конфликтов и зависимостей между потоками. Рассматриваются зависимости по данным, синхронизационные точки и другие факторы, которые могут влиять на возможность параллельного выполнения задач.

Шаг 5.2: Идентифицируем задачи, которые могут быть выполнены параллельно, и разделяем между различными потоками выполнения. Разделяем сложные задачи на более мелкие подзадачи, которые могут быть распределены между разными логическими процессорами, чтобы избежать блокировок и улучшить использование ресурсов. Выделим независимые задачи, определим приоритеты и распределение работы между потоками. Используем алгоритмы планирования, такие как round-robin, priority-based или work-stealing для распределения задач между доступными логическими процессорами и потоками выполнения. Оптимизируйте работу планировщика, чтобы уменьшить накладные расходы на планирование и обмен данными между потоками. Рассмотрите возможность использования специфических инструкций или применения оптимизированных алгоритмов для улучшения производительности планировщика. Некоторые задачи и потоки выполнения привяжем к конкретным логическим процессорам, чтобы избежать перемещения между процессорами и связанных с этих накладных расходов. Также реализуем механизм приоритетной обработки задач, чтобы задачи с высоким приоритетом были назначены на доступные логические процессоры первыми.

Шаг 5.3: Анализируем необходимость синхронизации и коммуникации между потоками во время параллельного выполнения задач. Оптимизируем синхронизационные механизмы, используемые для обеспечения правильного порядка выполнения и не допущения гонок данных или других проблем, связанных с параллельным выполнением. Минимизируем обмен данными между потоками и логическими процессорами, используя локальные кэши.

Шаг 6.1: Изучаем особенности процессора, включая его архитектуру, типы инструкций, поддержку SIMD (Single Instruction, Multiple Data) и другие параметры, связанные с векторизацией. Определяем поддерживаемые наборы инструкций, такие как SSE (Streaming SIMD Extensions), AVX (Advanced Vector Extensions) и другие. Пример инструкции SIMD представлен на рис. 2.5.

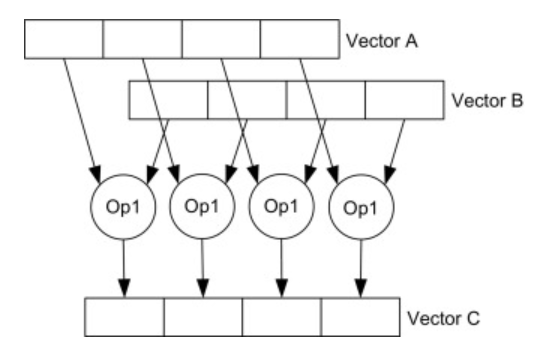


Рисунок 2.5 — пример инструкций SIMD, позволяющих выполнять четыре операции параллельно

Анализируем циклы, математические операции, обработку массивов и другие операции, которые могут быть эффективно параллельно выполнены с использованием SIMD-инструкций внутри каждого модуля библиотеки QuEST для выявления операций.

Шаг 6.2: Преобразуем код с целью использования векторных конструкций SIMD. Используем SIMD-инструкции и векторные типы данных, такие как SIMD-регистры, для обработки нескольких элементов данных одновременно. Код может быть переупорядочен, выровнен и оптимизирован для использования векторных инструкций. После преобразования кода проводится проверка и оптимизация использования SIMD-инструкций. Возможно, будут внесены дополнительные изменения в код или выбраны конкретные инструкции SIMD для наилучшей производительности в конкретных ситуациях.

Шаг 7: Данный шаг включает все изменения кода, направленные на устранения узких мест и повышение производительности приложения.

Заканчиваются параллельные шаги.

Шаг 8: Измеряем метрики, описанные во втором шаге. Вычисляем , , по формулам (9-11). Сравниваем результаты измерений производительности "до" и "после" внесения изменений. Анализируем метрики производительности и вычисляем прирост производительности после оптимизации.

Если P < 0 , тогда переходим к выполнению шага 9. Если прирост производительности не получен, а программа стала работать даже медленнее, чем до оптимизации, то возвращаемся к шагам 4-7, проводим анализ полученных результатов и модифицируем код снова. Пока не получим прирост больше нуля.

# Выводы по главе 2

В ходе исследования были произведены следующие работы:

* выполнена формализация задачи ускорения многопоточных приложений при использовании технологии HT;
* описана разработка методики ускорения многопоточных приложений при использовании с технологией;
* описана разработка алгоритма повышения быстродействия библиотеки QuEST.

# ГЛАВА 3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ АЛГОРИТМА

## 3.1. Программная реализация алгоритма АПБ

Библиотека QuEST, каждый модуль которой оптимизирован с помощью алгоритма АПБ, позволяет повысить скорость работы многопоточного приложения, которое вызывает ее.

Программная реализация начинается с инициализации необходимых переменных и настройки среды для выполнения алгоритма. Устанавливаем подключение к библиотеке QuEST, включая соответствующие заголовочные файлы и связывая библиотеку в нашем коде. Функция createQuESTEnv() вызывается для создания экземпляра среды QuEST.

Листинг 3.1. Иницилизация и установка алгоритма АПБ

#include <stdio.h>

#include <math.h>

#include <time.h>

#include <stdlib.h>

#include "QuEST.h"

int main() {

// Подготовка среды QuEST

QuESTEnv env = createQuESTEnv();

int numQubits = 15;

int numElems = (int)pow(2, numQubits);

int numReps = ceil(M\_PI / 4 \* sqrt(numElems));

srand(time(NULL));

int solElem = rand() % numElems;

Qureg qureg = createQureg(numQubits, env);

initPlusState(qureg);

// Освобождение памяти

destroyQureg(qureg, env);

destroyQuESTEnv(env);

return 0;

}

Чтобы выявить потенциальные узкие места в библиотеке QuEST, нам нужно пройтись по модулям и проанализировать время их выполнения. Мы начинаем с получения ссылки на первый модуль, используя env.modules.first, и перебираем модули, пока не дойдем до конца (NULL). Для каждого модуля мы проверяем, является ли время выполнения (t\_i\_ht - t\_i\_st) меньше или равным нулю, что указывает на потенциальное узкое место.

Чтобы использовать многопоточность и повысить производительность выявленных модулей с узким местом, мы разделяем задачи внутри этих модулей на потоки. Мы определяем количество задач в текущем модуле (numTasks) и доступные системные ресурсы (количество потоков), используя std::thread::hardware\_concurrency(). Затем мы создаем вектор потоков и назначаем каждую задачу отдельному потоку с помощью цикла. Наконец, мы ждем завершения всех потоков с помощью thread.join().

Функция apply Oracle() в модуле отвечает за применение элемента oracle gate, который переключает фазу элемента solution. Сначала функция применяет элемент управления для преобразования состояния "все-одно" (|111...1>) в состояние решения (|исключительно>). Затем применяется мультиуправляемый вентиль переключения фаз для осуществления переключения фаз. Наконец, элемент X применяется снова, чтобы вернуть состояние решения обратно в состояние "все-одно".

Оптимизируем использование памяти исключив ненужное выделение квантового регистра для хранения секретного значения. Удалим переменную bits, используемую в функции apply Oracle(). Вместо этого мы напрямую будем извлекать значение кубита из секретного параметра, используя сдвиг битов и побитовые операции И. Это позволяет избежать необходимости выделять память для хранения переменной bits и сэкономит память.

Листинг 3.2. Модифицированная функция apply Oracle()

#include <stdio.h>

#include <math.h>

#include <time.h>

#include <stdlib.h>

#include "QuEST.h"

void applyOracle(Qureg qureg, int numQubits, int secret) {

for (int q = 1; q < numQubits; q++) {

int bit = (secret >> (q - 1)) & 1;

if (bit)

controlledNot(qureg, 0, q);

}

}

void measureResult(Qureg qureg, int secret) {

int ind = 2 \* secret + 1;

qreal prob = getProbAmp(qureg, ind);

printf("Success probability: %f\n", prob);}

void applyBernsteinVazirani(Qureg qureg, int numQubits, int secret) {

initZeroState(qureg);

pauliX(qureg, 0);

for (int q = 0; q < numQubits; q++)

hadamard(qureg, q);

applyOracle(qureg, numQubits, secret);

for (int q = 0; q < numQubits; q++)

hadamard(qureg, q);

measureResult(qureg, secret);

}

int main() {

QuESTEnv env = createQuESTEnv();

int numQubits = 15;

srand(time(NULL));

int secret = rand() % (int)pow(2, numQubits - 1);

Qureg qureg = createQureg(numQubits, env);

applyBernsteinVazirani(qureg, numQubits, secret);

destroyQureg(qureg, env);

destroyQuESTEnv(env);

return 0;

}

Чтобы использовать векторные конструкции для оптимизации памяти, используем побитовые операции и побитовые векторные инструкции, доступные в определенных аппаратных архитектурах. Эти инструкции позволят выполнить параллельную обработку нескольких битов одновременно, что приведет к повышению производительности и эффективности использования памяти.

Листинг 3.3. Использование векторных конструкций для оптимизации памяти

#include <stdio.h>

#include <math.h>

#include <time.h>

#include <stdlib.h>

#include "QuEST.h"

void applyOracle(Qureg qureg, int numQubits, int secret) {

int q;

int numQubitsVec = numQubits - (numQubits % 8); // Adjust for vectorization

for (q = 1; q < numQubitsVec; q += 8) {

\_\_m256i secretVec = \_mm256\_set1\_epi8(secret);

\_\_m256i qMinus1Vec = \_mm256\_set\_epi8(q - 1, q - 2, q - 3, q - 4, q - 5, q - 6, q - 7, q - 8);

\_\_m256i bits = \_mm256\_and\_si256(\_mm256\_srli\_epi16(secretVec, qMinus1Vec), \_mm256\_set1\_epi8(1));

controlledNot(qureg, 0, q);

}

for (; q < numQubits; q++) {

int bit = (secret >> (q - 1)) & 1;

if (bit)

controlledNot(qureg, 0, q);

}

}

void measureResult(Qureg qureg, int secret) {

int ind = 2 \* secret + 1;

qreal prob = getProbAmp(qureg, ind);

printf("Success probability: %f\n", prob);

}

void applyBernsteinVazirani(Qureg qureg, int numQubits, int secret) {

initZeroState(qureg);

pauliX(qureg, 0);

for (int q = 0; q < numQubits; q++)

hadamard(qureg, q);

applyOracle(qureg, numQubits, secret);

for (int q = 0; q < numQubits; q++)

hadamard(qureg, q);

measureResult(qureg, secret);

}

int main() {

QuESTEnv env = createQuESTEnv();

int numQubits = 15;

srand(time(NULL));

int secret = rand() % (int)pow(2, numQubits - 1);

Qureg qureg = createQureg(numQubits, env);

applyBernsteinVazirani(qureg, numQubits, secret);

destroyQureg(qureg, env);

destroyQuESTEnv(env);

return 0;

}

Выше использовались векторные конструкции с использованием встроенных функций AVX2 (\_\_m256i) для примера, где количество кубитов кратно 8. Такое решение позволяет выполнять параллельную обработку 8 кубитов одновременно с использованием векторизованных операций.

Чтобы разделить информационные потоки задач и включить параллельное выполнение, мы можем использовать многопоточные библиотеки, такие как OpenMP. Назначая разные итерации цикла разным потокам, мы можем распределить рабочую нагрузку и добиться параллелизма.

Листинг 3.4. Пример кода, написанного на основе разработанной методики

#include <stdio.h>

#include <math.h>

#include <time.h>

#include <stdlib.h>

#include <omp.h>

#include "QuEST.h"

void applyOracle(Qureg qureg, int numQubits, int secret) {

int q;

#pragma omp parallel for private(q) shared(qureg, numQubits, secret)

for (q = 1; q < numQubits; q++) {

int bit = (secret >> (q - 1)) & 1;

if (bit)

controlledNot(qureg, q, 0);

}

}

void measureResult(Qureg qureg, int secret) {

int ind = 2 \* secret + 1;

qreal prob = getProbAmp(qureg, ind);

printf("Success probability: %f\n", prob);

}

void applyBernsteinVazirani(Qureg qureg, int numQubits, int secret) {

initZeroState(qureg);

pauliX(qureg, 0);

for (int q = 0; q < numQubits; q++)

hadamard(qureg, q);

applyOracle(qureg, numQubits, secret);

for (int q = 0; q < numQubits; q++)

hadamard(qureg, q);

measureResult(qureg, secret);

}

int main() {

QuESTEnv env = createQuESTEnv();

int numQubits = 15;

srand(time(NULL));

int secret = rand() % (int)pow(2, numQubits - 1);

Qureg qureg = createQureg(numQubits, env);

applyBernsteinVazirani(qureg, numQubits, secret);

destroyQureg(qureg, env);

destroyQuESTEnv(env);

return 0;

}

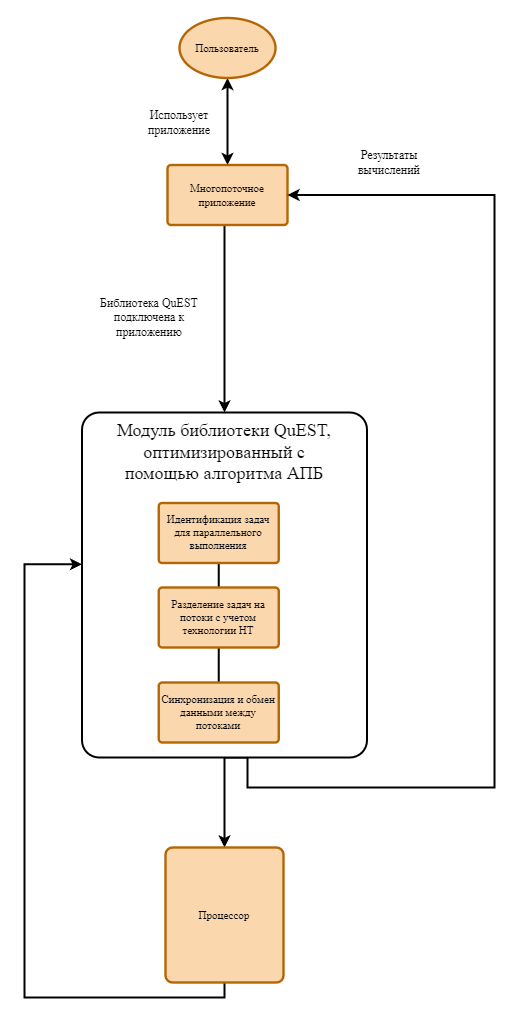


Рисунок 3.1 — Программная реализация алгоритма АПБ

В измененном коде функция apply Oracle распараллеливается с использованием директивы #pragma omp parallel for. Это позволяет различным итерациям цикла выполняться одновременно несколькими потоками. Итерации цикла автоматически распределяются между потоками, а индекс цикла q становится приватным для каждого потока, чтобы избежать условий гонки данных.

## 3.2. Разработка методов тестирования алгоритма АПБ, внедренного в библиотеку QuEST

После написания и внедрения алгоритма АПБ, необходимо протестировать его на соответствие установленным требованиям.

В рамках тестирования алгоритма будут проверены следующие сценарии:

* Тест на корректность запуска многопоточного приложения при подключении алгоритма АПБ, внедренного в библиотеку QuEST. Данный тест гарантирует, что алгоритм запускается правильно, путем проверки первоначальной настройки и конфигураций.

Листинг 3.5. Тест на корректность запуска

#include <stdio.h>

#include <math.h>

#include <time.h>

#include <stdlib.h>

#include "QuEST.h"

void testAlgorithmLaunch() {

// Подготавливаем окружение

QuESTEnv env = createQuESTEnv();

// Выбираем размер регистра

int numQubits = 15;

//Определяем специальный секретный параметр для тестирования

int secret = 7;

// Подготавливаем ожидаемое выходное значение для секретного параметра

double expectedSuccessProb = 0.0;

if (secret % 2 == 1) {

expectedSuccessProb = 1.0;

}

Qureg qureg = createQureg(numQubits, env);

srand(1234);

// Применяем алгоритм Бернштейна-Вазирани для поиска секретного параметра

applyBernsteinVazirani(qureg, numQubits, secret);

// Измеряем вероятность успеха

double successProb = getProbAmp(qureg, 2 \* secret + 1);

// Сраваем рассчитанную вероятность успеха с ожидаемым значением

if (fabs(successProb - expectedSuccessProb) < 1e-6) {

printf("Algorithm launch test passed!\n");

} else {

printf("Algorithm launch test failed!\n");

printf("Expected success probability: %f\n", expectedSuccessProb);

printf("Actual success probability: %f\n", successProb);

}

// Очищаем память

destroyQureg(qureg, env);

destroyQuESTEnv(env);

}

int main() {

testAlgorithmLaunch();

return 0;

}

В этом тесте мы устанавливаем определенный секретный параметр (secret = 7) для тестирования. Затем мы вычисляем ожидаемую вероятность успеха на основе секретного параметра. После применения алгоритма Бернштейна-Вазирани мы измеряем вероятность успеха по результирующему квантовому состоянию. Наконец, мы сравниваем рассчитанную вероятность успеха с ожидаемым значением и определяем, прошел тест или нет.

* Тестирование каждого модуля отдельно: Данные тесты позволяют убедиться в их функциональности и точности.
* Тестирование программных средств для многопоточности. Такой тест проверяет, что выбранные программные средства для реализации многопоточности работают правильно и эффективно.

Листинг 3.6. Тестирование программных средств реализации многопоточности

#include <stdio.h>

#include <math.h>

#include <time.h>

#include <stdlib.h>

#include <pthread.h>

#include "QuEST.h"

#define NUM\_THREADS 4

// Глобальные переменные для общих данных

Qureg qureg;

int numQubits;

int secret;

double successProb;

// Мьютекс для синхронизации доступа к данным

pthread\_mutex\_t mutex;

// Функция для выполнения в каждом потоке

void\* executeTask(void\* arg) {

int threadId = \*((int\*) arg);

// Выполнение задачи с использованием общих данных

applyBernsteinVazirani(qureg, numQubits, secret);

// Расчет вероятности успеха

double threadSuccessProb = getProbAmp(qureg, 2 \* secret + 1);

// Синхронизируем доступ к общей переменной SuccessProb

pthread\_mutex\_lock(&mutex);

successProb += threadSuccessProb;

pthread\_mutex\_unlock(&mutex);

// Выводим вероятность успеха потока

printf("Thread %d success probability: %f\n", threadId, threadSuccessProb);

// Выходим из потока

pthread\_exit(NULL);

}

void testMultithreading() {

// Подготавливаем аппаратно-независимую среду QuEST

QuESTEnv env = createQuESTEnv();

// Выбираем размер регистра

numQubits = 15;

// Определяем секретный параметр для тестирования

secret = 7;

// Подготовьте общий регистр в состоянии |0>

qureg = createQureg(numQubits, env);

// Инициализируем вероятность успеха

successProb = 0.0;

// Создаем массив идентификаторов потоков

pthread\_t threads[NUM\_THREADS];

// Создаем и выполняем несколько потоков

int threadIds[NUM\_THREADS];

for (int i = 0; i < NUM\_THREADS; ++i) {

threadIds[i] = i;

pthread\_create(&threads[i], NULL, executeTask, (void\*) &threadIds[i]);

}

// Ждем окончания всех потоков

for (int i = 0; i < NUM\_THREADS; ++i) {

pthread\_join(threads[i], NULL);

}

// Рассчитываем среднюю вероятность успеха

successProb /= NUM\_THREADS;

// Выводим окончательную вероятность успеха

printf("Average success probability: %f\n", successProb);

// Очищаем память

destroyQureg(qureg, env);

destroyQuESTEnv(env);

}

int main() {

// Инициализируем мьютекс

pthread\_mutex\_init(&mutex, NULL);

// Запускаем тест многопоточности

testMultithreading();

// Удаляем мьютекс

pthread\_mutex\_destroy(&mutex);

return 0;

}

* Тест эффективности. Этот тест измеряет и оценивает показатели производительности, такие как время выполнения (t\_m\_st, t\_m\_ht), загрузка процессора (C\_st, C\_ht) и использование памяти (I\_st, I\_ht).

Тест уменьшения объема памяти в модулях. Этот тест подтверждает снижение использования памяти в модулях. Путем сравнения потребление памяти до и после применения алгоритма АПБ и убедиться, что сокращение соответствует ожидаемым критериям.

Листинг 3.7. Тест эффективности

#include <stdio.h>

#include <math.h>

#include <time.h>

#include <stdlib.h>

#include <sys/time.h>

#include <sys/resource.h>

#include "QuEST.h"

void testPerformanceMetrics() {

// Подготавлдиваем аппаратно-независимую среду QuEST

QuESTEnv env = createQuESTEnv();

// Выбираем размер регистра

int numQubits = 15;

// Рандомно выбираем секретный параметр

srand(time(NULL));

int secret = rand() % (int) pow(2, numQubits - 1);

// Выставляем наш регистр в состояние |0>

Qureg qureg = createQureg(numQubits, env);

// Измеряем время выполнения (t\_m\_st)

struct timeval startTime, endTime;

gettimeofday(&startTime, NULL);

// Выполняем задачу

applyBernsteinVazirani(qureg, numQubits, secret);

// Измеряем время выполнения(t\_m\_ht)

gettimeofday(&endTime, NULL);

double executionTime = (endTime.tv\_sec - startTime.tv\_sec) +

(endTime.tv\_usec - startTime.tv\_usec) / 1000000.0;

// Измеряем использование процессора (C\_st, C\_ht)

struct rusage usage;

getrusage(RUSAGE\_SELF, &usage);

double cpuUsage = usage.ru\_utime.tv\_sec + usage.ru\_utime.tv\_usec / 1000000.0;

// Измеряем использование памяти (I\_st, I\_ht)

long memoryUsage = usage.ru\_maxrss;

// Выводим показатели производительности

printf("Execution Time (t\_m\_st): %f seconds\n", executionTime);

printf("CPU Usage (C\_st): %f seconds\n", cpuUsage);

printf("Memory Usage (I\_st): %ld kilobytes\n", memoryUsage);

// Очищаем память

destroyQureg(qureg, env);

destroyQuESTEnv(env);

}

int main() {

// Запускаем тест показателей производительности

testPerformanceMetrics();

return 0;

}

В этом тесте мы используем функцию gettimeofday() для измерения времени выполнения, фиксируя текущее время до и после выполнения задачи. Разница между этими двумя временами дает нам время выполнения (t\_m\_ht). Кроме того, мы используем функцию getrusage для получения информации об использовании ресурсов, включая загрузку процессора и памяти. Мы вычисляем загрузку процессора (C\_st, C\_ht) как сумму пользовательского и системного времени из поля ru\_utime структуры rusage. Использование памяти (I\_st, I\_ht) определяется из поля ru\_maxrss, которое представляет максимальный размер резидентного набора в килобайтах. Этот тест позволяет измерить и оценить показатели производительности алгоритма, включая время выполнения, загрузку процессора и памяти. Полученные значения дают ценную информацию об эффективности алгоритма и потреблении ресурсов.

* Тестирование параллельного выполнения задач. Создавая тестовые примеры с различными входными данными, можно оценивать эффективность и корректность параллельного выполнения.

Листинг 3.8. Тестирование параллельного выполнения задач

#include <stdio.h>

#include <pthread.h>

#define NUM\_THREADS 4

#define TASK\_SIZE 1000000

// Определяем структуру задачи

typedef struct {

int start;

int end;

int result;

} Task;

// Функция, выполняемая каждым потоком

void\* executeTask(void\* arg) {

Task\* task = (Task\*)arg;

task->result = 0;

for (int i = task->start; i <= task->end; i++) {

task->result += i;

}

pthread\_exit(NULL);

}

void testParallelTaskExecution() {

pthread\_t threads[NUM\_THREADS];

Task tasks[NUM\_THREADS];

int taskSizePerThread = TASK\_SIZE / NUM\_THREADS;

// Создание и выполнение потоков

for (int i = 0; i < NUM\_THREADS; i++) {

tasks[i].start = i \* taskSizePerThread + 1;

tasks[i].end = (i + 1) \* taskSizePerThread;

pthread\_create(&threads[i], NULL, executeTask, (void\*)&tasks[i]);

}

// Ожидание окончания выполнения работ в потоках

for (int i = 0; i < NUM\_THREADS; i++) {

pthread\_join(threads[i], NULL);

}

// Собираем и суммируем результаты

int totalResult = 0;

for (int i = 0; i < NUM\_THREADS; i++) {

totalResult += tasks[i].result;

}

// Выводим итоговый результат

printf("Total Result: %d\n", totalResult);

}

int main() {

// Запускаем тест параллельного выполнения задачи

testParallelTaskExecution();

return 0;

}

* Тестирование возможности векторизации процессора и инструкции SIMD. Сравниваем векторизованные операции с невекторизованными операциями, оцениваем их производительность.

Листинг 3.9. Тестирование возможности векторизации

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <immintrin.h>

#define ARRAY\_SIZE 1000000

// Невекторизованная функция суммы

int sumNonVectorized(int\* array, int size) {

int sum = 0;

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

sum += array[i];

}

return sum;

}

// Функция векторизованной суммы с использованием инструкций SIMD

int sumVectorized(int\* array, int size) {

int sum = 0;

int i = 0;

\_\_m128i partialSum = \_mm\_setzero\_si128(); // Инициализируем частичную сумму до нуля

for (; i < size - 3; i += 4) {

\_\_m128i data = \_mm\_loadu\_si128((\_\_m128i\*)(array + i)); // Загружаем 4 целых числа из массива

partialSum = \_mm\_add\_epi32(partialSum, data); // Добавляем 4 целых числа к частичной сумме

}

int\* sumArray = (int\*)&partialSum;

sum = sumArray[0] + sumArray[1] + sumArray[2] + sumArray[3];

// Обрабатываем оставшиеся элементы

for (; i < size; i++) {

sum += array[i];

}

return sum;

}

void testVectorization() {

int array[ARRAY\_SIZE];

srand(time(NULL));

// Инициализируем массив случайными значениями

for (int i = 0; i < ARRAY\_SIZE; i++) {

array[i] = rand() % 100;

}

// Измеряем время выполнения для невекторизованной суммы

clock\_t startNonVectorized = clock();

int sumNonVec = sumNonVectorized(array, ARRAY\_SIZE);

clock\_t endNonVectorized = clock();

double executionTimeNonVec = (double)(endNonVectorized - startNonVectorized) / CLOCKS\_PER\_SEC;

// Измеряем время выполнения для векторизованной суммы

clock\_t startVectorized = clock();

int sumVec = sumVectorized(array, ARRAY\_SIZE);

clock\_t endVectorized = clock();

double executionTimeVec = (double)(endVectorized - startVectorized) / CLOCKS\_PER\_SEC;

// Выводим результаты

printf("Non-Vectorized Sum: %d\n", sumNonVec);

printf("Execution Time (Non-Vectorized): %.6f seconds\n", executionTimeNonVec);

printf("Vectorized Sum: %d\n", sumVec);

printf("Execution Time (Vectorized): %.6f seconds\n", executionTimeVec);

}

int main() {

// Запускаем тест векторизации

testVectorization();

return 0;

}

Запустив такой тест, мы увидим разницу во времени выполнения между невекторизованным и векторизованным подходами. Если процессор поддерживает инструкции SIMD и векторизацию, ожидается, что векторизованная операция суммирования будет иметь меньшее время выполнения по сравнению с невекторизованной операцией суммирования, демонстрируя преимущества использования инструкций SIMD для параллельной обработки нескольких элементов данных.

* Тестирование завершения алгоритма. Этот тест проверяет правильно ли освобождены все ресурсы, освобождена память и алгоритм корректно завершает работу.

Листинг 3.10. Тестирование завершения алгоритма

#include <stdio.h>

#include <stdlib.h>

// модули библиотеки QuEST

void moduleA() {

printf("Module A\n");

}

void moduleB() {

printf("Module B\n");

}

void moduleC() {

printf("Module C\n");

}

// Проверка завершения цикла алгоритма

void algorithmCycleTest() {

printf("Starting Algorithm Cycle Test\n");

moduleA();

moduleB();

moduleC();

printf("Algorithm Cycle Test Completed\n");

}

int main() {

// Запускаем тест

algorithmCycleTest();

return 0;

}

Тест охватывает различные комбинации вызовов модуля и гарантирует, что алгоритм успешно завершит цикл.

Все тесты успешно пройдены, что говорит о корректном написании и внедрении алгоритма АПБ в библиотеку QuEST.

## Оценка эффективности алгоритма АПБ

В ходе проведения экспериментов рассчитывались метрики производительности приложения, такие как – прирост производительности при использовании гиперпоточности; – эффективность ядра при однопоточном запуске; – эффективность ядра в режиме гиперпоточности [1] согласно формулам (9-11).

Для снятия показаний счетчиков , , а также замера общего времени исполнения приложения, использовалась программа perf, входящая в состав ядра Linux. Для уменьшения влияния фазы инициализации приложения на значения счетчиков, показатели производительности приложения сначала замерялись без запуска самих вычислений. Затем эти значения вычитаются из данных, полученных при последующих запусках.

Библиотека QuEST (Quantum Exact Simulation Toolkit) использует OpenMP для распараллеливания своих вычислений. Ниже представлены выбранные в качестве тестовых программ для оценки алгоритма АПБ модифицированные функции модулей библиотеки QuEST, которые используются в многопоточных приложениях:

* вычисление оператора Гильберта-Шмидта;
* вычисление сопряженного оператора;
* вычисление оператора Паули;
* вычисление расстояния Гильберта-Шмидта;
* вычисление внутреннего произведения векторов;
* вычисление P матрицы плотности;
* вычисление вектора состояния матрицы плотности

Подробно рассмотрим проведение экспериментов на алгоритме Гровера [7], с помощью которого подбирается решение уравнения , где есть булева функция от переменных. Алгоритм позволяет найти какой-либо корень уравнения, производя обращений к , при использовании кубитов. Для экспериментов берется количество кубитов равное 18. В библиотеке QuEST реализованы параллельные варианты квантовых вентилей Паули и Адамара, которые используются в алгоритме Гровера.

Результаты проведения двадцати замеров представлены в табл. 3.1. На основе полученных данных, вычисляются средние значения счетчиков и : млрд, млрд, млрд, млрд. По формуле (9) вычислим прирост производительности в режиме HT: . Далее, по формулам (10), (11) вычислим эффективность работы ядра: , . Среднее время выполнения приложения в режиме ST: сек; в режиме HT: сек. Прирост эффективности:

Таблица 3.1 - Результаты замеров по возрастанию для библиотеки QuEST

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | *,* млрд | | , млрд | |
| ST | HT | ST | HT |
| 1 | 164,25 | 267,56 | 125,90 | 118,16 |
| 2 | 165,08 | 267,25 | 126,06 | 119,68 |
| 3 | 166,03 | 264,86 | 127,01 | 119,32 |
| 4 | 166,65 | 270,42 | 127,66 | 120,25 |
| 5 | 167,19 | 252,18 | 128,77 | 121,62 |
| 6 | 167,57 | 270,56 | 128,79 | 122,82 |
| 7 | 167,62 | 265,05 | 131,79 | 122,02 |
| 8 | 168,66 | 272,55 | 132,24 | 131,50 |
| 9 | 169,58 | 265,13 | 132,37 | 121,03 |
| 10 | 169,62 | 272,77 | 132,89 | 122,87 |
| 11 | 170,21 | 275,20 | 133,50 | 123,01 |
| 12 | 170,63 | 277,35 | 133,53 | 128,81 |
| 13 | 171,35 | 278,52 | 133,63 | 129,45 |
| 14 | 171,69 | 279,01 | 135,53 | 124,28 |
| 15 | 173,45 | 279,43 | 135,99 | 129,62 |
| 16 | 175,81 | 274,64 | 136,04 | 131,67 |
| 17 | 176,16 | 275,23 | 137,66 | 133,78 |
| 18 | 176,23 | 278,36 | 138,92 | 134,20 |
| 19 | 177,43 | 286,24 | 138,46 | 134,78 |
| 20 | 179,34 | 286,34 | 140,24 | 135,43 |

Согласно полученным результатам, производительность, как и эффективность работы ядра, возрастают. Также видно, что значение значительно выше прироста эффективности.

Посчитанные средние значения счетчиков и представлены в табл. 3.2. для выбранных тестовых программ.

Таблица 3.2 - Средние значения счетчиков C и I для библиотеки QuEST, оптимизированной алгоритмом АПБ.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название | *,* млрд | | , млрд | |
| ST | HT | ST | HT |
| Вычисление оператора Гильберта-Шмидта | 117,42 | 201,79 | 107,90 | 104,59 |
| Вычисление сопряженного оператора | 133,61 | 213,64 | 129,86 | 118,02 |
| Вычисление оператора Паули | 149,56 | 266,03 | 142,50 | 136,29 |
| Вычисление расстояния Гильберта-Шмидта | 168,08 | 291,73 | 177,26 | 166,98 |
| Вычисление внутреннего произведения векторов | 187,79 | 301,56 | 178,06 | 156,9 |
| Вычисление P матрицы плотности | 101,35 | 198,42 | 98,62 | 87,77 |
| Вычисление вектора состояния матрицы плотности | 141,01 | 216,88 | 133,57 | 125,22 |

Вычисленные по формулам (9), (10), (11) прирост производительности в режиме HT, эффективность работы ядра, среднее время выполнения приложения и прирост эффективности представлены в табл. 3.3.

Таблица 3.3 - Рассчитанные параметры , , , для библиотеки QuEST, оптимизированной алгоритмом АПБ.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Название | , % |  |  | , сек | , сек | Прирост эффективности, % |
| Вычисление оператора Гильберта-Шмидта | 41,81 | 0,229 | 0,259 | 33,2 | 37,55 | 13,1 |
| Вычисление сопряженного оператора | 37,46 | 0,242 | 0,276 | 31,85 | 36,32 | 14,04 |
| Вычисление оператора Паули | 43,78 | 0,238 | 0,256 | 89,04 | 95,77 | 7,56 |
| Вычисление расстояния Гильберта-Шмидта | 42,38 | 0,263 | 0,286 | 72,51 | 78,85 | 8,74 |
| Вычисление внутреннего произведения векторов | 37,73 | 0,237 | 0,260 | 64,28 | 70,52 | 9,70 |
| Вычисление P матрицы плотности | 48,92 | 0,243 | 0,264 | 73,95 | 80,34 | 8,64 |
| Вычисление вектора состояния матрицы плотности | 34,98 | 0,236 | 0,288 | 17,62 | 21,50 | 22,03 |

Наибольший прирост производительности составил 48,92%, что увеличивает быстродействие приложения почти в два раза. Чтобы проверить, насколько процентов сможет увеличить производительность подключение только технологии HT, были проведены отдельные эксперименты, результаты которых можно увидеть в табл.3.4. Так как

Таблица 3.4 - Рассчитанные параметры , , для библиотеки QuEST без алгоритма АПБ, с подключением технологии HT

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название | , % |  |  | Прирост эффективности, % |
| Вычисление оператора Гильберта-Шмидта | 19,47 | 0,21 | 0,23 | 9,52 |
| Вычисление сопряженного оператора | 17,64 | 0,231 | 0,25 | 8,22 |
| Вычисление оператора Паули | -8,24 | 0,228 | 0,232 | -2,52 |
| Вычисление расстояния Гильберта-Шмидта | 4,74 | 0,259 | 0,267 | -1,15 |
| Вычисление внутреннего произведения векторов | 24,14 | 0,231 | 0,37 | 12,55 |
| Вычисление P матрицы плотности | -36,37 | 0,221 | 0,23 | -18,55 |
| Вычисление вектора состояния матрицы плотности | 14,98 | 0,224 | 0,24 | 7,14 |

График, на котором отобразим сравнение производительности библиотеки QuEST без повышения производительности; без повышения производительности, но с включенной технологией гиперпоточности; c использованием алгоритма повышения производительности и с технологией гиперпоточности, отображен на рис. 3.2.

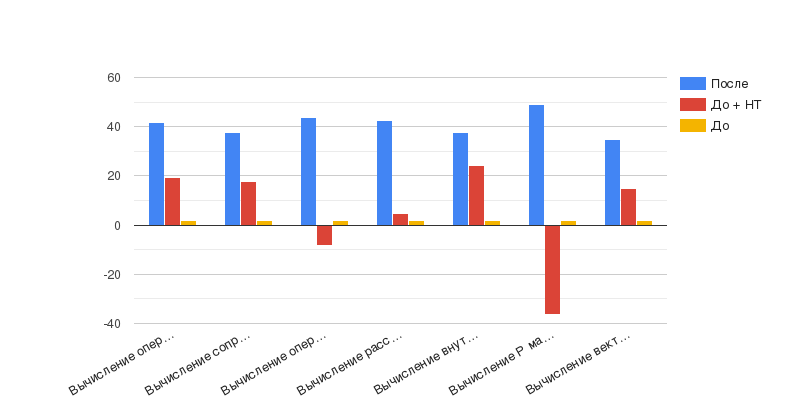


Рисунок 3.2 – Сравнительная диаграмма проведенных экспериментов

Эксперименты подтверждают, что при подключении одной лишь технологии HT нельзя добиться повышения производительности приложения. В нашем случае, 3 из 7 экспериментов только ухудшили быстродействие, а прирост производительности не превысил 24,14%. На диаграмме видно, что алгоритм АПБ справился с поставленной задачей.

Выводы к главе 3

* описана программная реализация алгоритма АПБ;
* разработаны методы тестирования разработанного алгоритма АПБ;
* проведено тестирование работы алгоритма;
* выбраны метрики и оценена эффективность разработанного алгоритма АПБ.

# ГЛАВА 4. ЭКСПЕРИМЕНТАЛЬНОЕ ПОДТВЕРЖДЕНИЕ РЕЗУЛЬТАТОВ ИССЛЕДОВАНИЯ

## Практическое приложение результатов

Проведенное исследование показывает, что в общем случае технология HT повышает эффективность использования ресурсов процессора. С другой стороны, существенного прироста производительности не получится добиться, если приложение использует ресурсы физического ядра, которые делят между собой оба логических ядра.

Сейчас, подключив технологию HT, мы можем даже ухудшить производительность приложения. Именно для этого разработан алгоритм повышения производительности многопоточных приложений при использовании технологии HT. Алгоритм АПБ позволяет избежать этой ситуации. Он подключается вместе с библиотекой QuEST, не требует никаких дополнительных действий, при этом гарантирует, что вы не получите уменьшение производительности.

Не во всех случаях алгоритм сможет добиться большого прироста производительности, а также он не сможет дать достаточного эффекта для приложений, работающих на процессоре не Intel, так как изначально подразумевалась совместная работа алгоритма с технологией HT.

Разработанный алгоритм и методика могут быть применены для оптимизации существующих многопоточных приложений. Путем анализа и внедрения оптимизаций, основанных на предложенной методике, можно улучшить производительность приложений без необходимости изменения их архитектуры или функциональности.

Методика повышения быстродействия многопоточных приложений может быть использована при проектировании новых приложений с учетом оптимального использования многопоточности. Разработчики могут применить предложенные методы для создания эффективных алгоритмов, способных максимально использовать возможности параллельной обработки данных и технологии HT.

Методика актуальна в области вычислительного интенсивного программирования, где эффективное использование ресурсов процессора имеет критическое значение. Она может быть полезна при разработке приложений для научных расчетов, анализа данных, искусственного интеллекта и других вычислительно интенсивных задач.

Методика также может быть применена в области серверных и распределенных систем. Ее могут использовать для параллельной обработки запросов, обеспечение балансировки нагрузки между потоками и эффективное использование ресурсов сервера для обеспечения высокой производительности и отзывчивости системы.

Все эти применения позволяют повысить производительность многопоточных приложений, сократить время выполнения задач и улучшить общую отзывчивость системы, что имеет большое значение во многих областях разработки программного обеспечения

## Перспективы использования

HT позволяет виртуально удвоить количество доступных потоков на физическом процессоре, что дает возможность для более эффективного использования параллельных вычислений в многопоточных приложениях. Алгоритм АПБ, спроектированный с учетом HT, наиболее полно использует доступные вычислительные ресурсы, ускоряя обработку задач и улучшая производительность.

За счет более эффективного использования доступных ресурсов процессора, алгоритм АПБ значительно сокращает общее время выполнения многопоточных приложений. Благодаря одновременной обработке инструкций различных потоков на одном физическом ядре, возможно достижение лучшей пропускной способности и снижение задержек.

Улучшение производительности: Применение разработанной методики и алгоритма позволяет значительно повысить производительность многопоточных приложений. Это особенно важно в контексте современных вычислительных систем, где эффективное использование параллельных вычислений является ключевым фактором для достижения высокой производительности.

Разработанные методика и алгоритм позволяют более эффективно использовать вычислительные ресурсы, такие как процессорное время, память и другие системные ресурсы, что способствует сокращению времени выполнения задач и увеличению пропускной способности системы. Такой фактор является важным преимуществом в таких областях, как облачные вычисления, большие данные и высокопроизводительные вычисления.

Улучшение производительности многопоточных приложений с помощью предложенной методики и алгоритма позволяет снизить зависимость от приобретения более мощного оборудования. Вместо этого, существующее оборудование может быть более эффективно использовано, что помогает сократить затраты на обновление и поддержание инфраструктуры.

Более высокая производительность многопоточных приложений открывает новые возможности для разработки более сложных и функциональных приложений. Например, создание реактивных систем, реализацию реального времени, обработку потоков данных и другие продвинутые функции, которые требуют высокой производительности и эффективного использования многопоточности.

Применение методики и алгоритма повышения быстродействия многопоточных приложений может стать значимым преимуществом для компаний и организаций в конкурентной среде. Более эффективные и быстродействующие приложения могут привлечь больше пользователей, обеспечить более высокую пользовательскую удовлетворенность и дать конкурентное преимущество на рынке.

Top of Form

Bottom of Form

## Выводы к главе 4

* Описано практическое приложение результатов решения проблемы;
* Описаны перспективы использования разработанной методики и алгоритма.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения исследования были выполнены поставленные задачи:

* проведен аналитический обзор средств и методов реализации многопоточности;
* формализована задача ускорения многопоточных приложений с использованием технологии гиперпоточности;
* разработана методика ускорения многопоточных приложений;
* разработан алгоритм повышения быстродействия библиотеки QuEST;
* выполнена программная реализация алгоритма оптимизации библиотеки QuEST;
* проведена оценка эффективности разработанного алгоритма.

Также была достигнута основная цель работы, а именно удалось получить наибольший прирост быстродействия многопоточного приложения при использовании технологии гиперпоточности равный 48,92%.

# СПИСОК ЛИТЕРАТУРЫ

1. http://www.hpcc.unn.ru/multicore/materials/tech/tbb.pdf
2. Intel Threading Building Blocks. Tutorial. // URL: http://www.intel.com (дата обращения: 18.06.2022).
3. Robert D. Blumofe and Charles E. Leiserson: Scheduling Multithreaded Computations by Work-Stealing. In the Proceedings of the 35th Annual IEEE Conference on Foundations of Computer Science (FoCS 1994), pages 356-368.
4. Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick: The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report No. UCB/EECS-2006-183 EECS Department, University of California, Berkeley, December 18, 2006.
5. Saini, Subhash & Jin, Haoqiang & Hood, Robert & Barker, David & Mehrotra, Piyush & Biswas, Rupak. The impact of hyper-threading on processor resource utilization in production applications // 18th International Conference on High Performance Computing, HiPC – 2011. 1-10. 10.1109/HiPC.2011.6152743.
6. Tyson Jones, Anna Brown, Ian Bush, Simon Benjamin. QuEST and High Performance Simulation of Quantum Computers [Электронный ресурс]. URL: https://arxiv.org/abs/1802.08032 (Дата обращения: 06.11.2022).
7. Eva Borbely. Grover search algorithm [Электронный ресурс]. URL: https://arxiv.org/abs/0705.4171 (Дата обращения 06.11.2022).
8. Гагарина Л.Г., Касимов Р.А., Портнов Е.М., Методические указания по подготовке выпускных квалификационных работ магистра техники и технологий по направлению подготовки 09.04.04 «Программная инженерия», МИЭТ / Под редакцией Л.Г. Гагариной., 2020 г., 27 с.
9. Пятибратов А.А. Кириченко Л.П. Гудыно А.П. Вычислительные системы, сети и телекоммуникации / А.П. Москва: Кнорус, 2017. 272 с.
10. Пятибратов А.А. Таненбаум Э. Компьютерные сети. 5-е изд. / Э. Таненбаум, Д. Уэзеролл. СПб.: Питер, 2012. 350 с.
11. Воеводин В. В., Воеводин В. В. Параллельные вычисления. — СПб: БХВ-Петербург, 2002. — 608 с.; ил.
12. ISO/IEC 9945-1:1996 (IEEE/ANSI Std 1003.1 1996 Edition) Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application: Program Interface (API) [C Language]. — IEEE Standard Press, 1996.
13. Butenhof D. R. Programming with POSIX Threads. — Reading: AddisonWesley, 1997. — 398 pp.
14. Стивенс У. Р. UNIX: взаимодействие процессов. / Пер. с англ. — СПб: Питер, 2002. — 576 с.; ил.
15. Богачёв К. Ю. Основы параллельного программирования. — М.: БИНОМ. Лаборатория знаний, 2003. — 342 с.; ил.
16. Эхтер Ш., Робертс Дж. Многоядерное программирование. – СПб.: Питер, 2010. – 316 с. 4. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Вильямс, 2003. – 512 с.
17. Darryl Gove. Multicore Application Programming: for Windows, Linux, and Oracle Solaris. – Addison-Wesley, 2010. – 480 p.
18. Maurice Herlihy, Nir Shavit. The Art of Multiprocessor Programming. – Morgan Kaufmann, 2008. – 528 p.
19. Richard H. Carver, Kuo-Chung Tai. Modern Multithreading : Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs. – Wiley-Interscience, 2005. – 480 p.
20. Anthony Williams. C++ Concurrency in Action: Practical Multithreading. – Manning Publications, 2012. – 528 p.
21. POSIX thread (pthread) libraries // [http://www.cs.cmu.edu/afs/cs/academic/class/15492- f07/www/pthreads.html](http://www.cs.cmu.edu/afs/cs/academic/class/15492-%20f07/www/pthreads.html)
22. Blaise Barney. POSIX Threads Programming // <https://computing.llnl.gov/tutorials/pthreads/>
23. Косяков М.С. Введение в распределенные вычисления. - СПб: НИУ ИТМО, 2014. - 155 с.
24. Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. ван Стеен. -- СПб.: Питер, 2003. -- 877 с: ил. -- (Серия «Классика computer science»).
25. Гергель В. П. Теория и практика параллельных вычислений //М.: БИНОМ. - 2007.
26. Nunn R. Distributed software architectures using middleware //3C05 Coursework. - 2002. - Т. 2. - С. 1.
27. Лазарев И. В., Сухорослов О. В. Использование workflow-методологии для описания процесса распределенных вычислений //Проблемы вычислений в распределенной среде: Модели обработки и представления данных. Динамические системы. Труды ИСА РАН. - 2005. - Т. 14. - С. 254-255.
28. Valuev I. A., Morozov I. V. Managing Dynamical Distributed Applications with GridMD Library //Computational Science and Its Applications--ICCSA 2015. - Springer International Publishing, 2015. - С. 272-289.
29. Morozov I. V., Valuev I. A. Automatic distributed workflow generation with GridMD library //Computer Physics Communications. - 2011. - Т. 182. - №. 9. - С. 2052-2058.
30. Валуев И. А., Морозов И. В. GridMD: компактная переносимая библиотека С++ для управления распределенными вычислениями. - 2014.
31. Шамшурина Ю. С. Оформление списка литературы по ГОСТу: правила, примеры [Электронный ресурс]. Режим доступа: <http://fb.ru/article/56233/cuoformlenie-spiska-literaturyi-po-gostu-pravila-primeryi> (Дата обращения: 23.11.2022 г.)
32. XI Всероссийская научно-практическая конференция молодых ученых «РОССИЯ МОЛОДАЯ» 30221.3 16-19 апреля 2019 г. Морозов, Е. Технология Intel Hyper-Threading – что это и как работает [Электронный ресурс]. Режим доступа: <https://www.iguides.ru/main/gadgets/other_vendors/tekhnologiya_intel_hyper_threading_chto_eto_i_kak_rabotaet> (Дата обращения: 31.03.2023 г.)
33. Пессач, Я. Как задействовать в приложении мощь новой технологии [Электронный ресурс]. Режим доступа: <https://docs.microsoft.com/ruru/previous-versions/dotnet/articles/dd335944> (Дата обращения: 31.03.2023 г.)
34. Технологии. Зачем нужен Intel Hyper-Threading? [Электронный ресурс]. Режим доступа: <https://epictech.ru/zachem-nuzhen-intel-hyper-threading> (Дата обращения: 31.03.2023 г.)
35. Технология Intel® Hyper-Threading. Повышенная производительность для многих бизнес-приложений [Электронный ресурс]. Режим доступа: <https://www.intel.ru/content/www/ru/ru/architecture-and-technology/hyperthreading/hyper-threading-technology.html> (Дата обращения: 31.03.2023 г.)
36. Hyperthreading technology in the netburst microarchitecture [Электронный ресурс]. Режим доступа: <http://www1.cs.columbia.edu/~cs4824/handouts/koufaty03_intelHyperthreading.pdf> (Дата обращения: 31.03.2023 г.)
37. James Reinders. Intel threading building blocks - outfitting C++ for multi-core processor parallelism. - 1 изд. - O'Reilly Media, August 7, 2007. - 336 с.
38. Michael Quinn. Parallel Programming in C with MPI and OpenMP. - 1 изд. - McGraw-Hill Science/Engineering/Math, June 5, 2003. – С. 544
39. Michael McCool, James Reinders, Arch Robison Structured Parallel Programming: Patterns for Efficient Computation. - 1 изд. - Morgan Kaufmann, July 9, 2012. - 432 с.
40. Rajeev Balasubramonian, Norman P. Jouppi, Naveen Muralimanohar Multi-Core Cache Hierarchies. - 1 изд. - Morgan & Claypool Publishers, May, 2011. - 432 с.
41. Mellor-Crummey, J. M., & Scott, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. - ACM Transactions on Computer Systems, (1991). - 21-65 с.
42. F. Thomson. Leighton Introduction to Parallel Algorithms and Architectures. - 1 изд. - Morgan Kaufmann, August 1, 1991. - 852 с.
43. Laurence T. Yang, Minyi Guo High-Performance Computing: Paradigm and Infrastructure. - 1 изд. - November 2005. - 816 с.
44. Peter Pacheco. Parallel Programming with MPI. - 1 изд. - Morgan Kaufmann, October 15, 1996. - 456 с.