МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«ТЮМЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ НАУК
Кафедра программного обеспечения

РЕКОМЕНДОВАНО К ЗАЩИТЕ
В ГЭК И ПРОВЕРЕНО НА ОБЪЕМ
ЗАИМСТВОВАНИЯ
Заведующий кафедрой
к.т.н., доцент
М. С. Воробьева
29.06. 2019 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(магистерская диссертация)

ИССЛЕДОВАНИЕ ЭФФЕКТИВНОСТИ ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ ДЛЯ РЕШЕНИЯ СЛАУ С ЛЕНТОЧНЫМИ МАТРИЦАМИ

02.04.03. Математическое обеспечение и администрирование информационных систем
Магистерская программа «Разработка, администрирование и защита вычислительных систем»

Выполнила работу
Студентка 2 курса
очной формы обучения
Гаврилова
Наталия
Михайловна

Научный руководитель
к.ф.–м.н., доцент
Ступников
Андрей
Анатольевич

Рецензент
доцент кафедры
информационных систем
ТюмГУ, к.ф.–м.н., доцент
Моор
Павел
Климентьевич

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Совершенствование вычислительной техники и сетевых технологий обуславливает повышенный интерес к технологиям параллельных вычислений.

Многие ведущие IT компании (Microsoft, Intel, AMD, NVIDIA, Google, и т.д.) занимаются проектированием и разработкой высокопроизводительных систем, основанных на алгоритмах параллельной обработки данных.

Одним из важнейших путей повышения скорости решения сложных задач является использование многопроцессорных вычислительных систем.

Практически все современные компьютеры оснащены многоядерными процессорами. Ускорение «долгих» расчетов требует подключения нескольких ядер для решения подзадач, которые могут выполняться независимо друг от друга. Это делает необходимым использование параллельного программирования и дает возможность использовать преимущества многоядерных процессоров и многопроцессорных систем [1].

Приближенное решение сложных научно-технических задач, особенно формализованных методами математического моделирования, часто сводится к решению уравнений в частных производных. Подобные модели в обязательном порядке изучаются студентами технических и естественно-научных направлений в рамках дисциплин, связанных с численными методами. Основное содержание лабораторных работ по этим дисциплинам требует разработки программ для численного решения задач уравнений математической физики, поэтому студент должен обладать компетенциями в области численных методов решения данного класса задач. С учётом широкого распространения многопроцессорной архитектуры современных компьютеров к этим компетенциям следует отнести и умение распараллеливать вычислительные алгоритмы. В связи с этим возникает необходимость рассмотрения различных подходов и алгоритмов для распараллеливания вычислений.

Магистерская диссертация посвящена описанию применимости технологий параллельных вычислений для решения задач вычислительной

математики с использованием многопроцессорных вычислительных систем с общей памятью.

Распараллеливание рассмотрено на примере численного решения краевой задачи для уравнения теплопроводности, описывающего процесс распространения тепла в одномерном однородном стержне с граничными условиями первого рода и заданными начальными условиями.

Данный подход сводится к решению системы линейных алгебраических уравнений с трехдиагональной матрицей коэффициентов при неизвестных. Для решения таких систем используется метод исключения неизвестных – метод прогонки.

Рассматриваемая задача может быть примером использования технологий параллельной обработки данных. Это объясняется тем, что при большом объеме начальных данных или именно при высокой требуемой точности вычисления могут занимать длительное время, что потребует распараллеливания вычислений для ускорения выполнения программы.

Существуют различные подходы для организации параллельных вычислений на вычислительных системах с общей памятью. В выпускной квалификационной работе рассмотрены технологии организации параллельных вычислений на основе потоков (класс Thread из библиотеки .NET), предполагающее ручное управление аспектами распараллеливания и с использованием технологии OpenMP, использующей встроенную библиотеку организации распараллеливания.

Организация параллельной обработки данных с использованием потоков предполагает, что программист сам устанавливает сколько данных организуется в тот или иной поток, как он работает. Для того, чтобы распараллелить данные существует единая технология потоков (thread). Для этого в языке программирования определен специальный класс Thread. Это трудоемкий процесс, но в некоторых случаях это позволяет четко контролировать процесс вычислений и производить его мониторинг.

Организация распараллеливания расчетов с использованием технологии OpenMP предполагает использование библиотечных процедур, специально

предназначенных для программирования многопоточных приложений на многопроцессорных системах с общей памятью (SMP-системах).

Таким образом, с одной стороны, в работе рассмотрены различные алгоритмы для распараллеливания метода трехдиагональной прогонки, с другой стороны эти алгоритмы реализованы с использованием различных параллельных вычислительных технологий.

Алгоритмы распараллеливания трехдиагональный прогонки интересны сами по себе, но с практической точки зрения интересно и сравнение этих алгоритмов с точки зрения их эффективности и применения для решения задач, возникающих при моделировании физических процессов, сводимых к решению уравнений в частных производных.

Цель работы – разработать приложения, решающие систему линейных уравнений с ленточной матрицей коэффициентов с использованием различных технологий распараллеливания вычислений.

Для достижения цели необходимо решить следующие задачи:

1. Изучить:
   - задачи, сводимые к решению систем линейных уравнений с ленточными матрицами;
   - алгоритмы распараллеливания для подобных систем с 3-хдиагональной матрицей коэффициентов при неизвестных;
   - технологии параллельных вычислений в системах с общей памятью.
2. Реализовать:
   - однопоточный алгоритм метода трехдиагональной прогонки;
   - двухпоточный алгоритм встречной прогонки;
   - многопоточные варианты метода прогонки.

В качестве алгоритмов распараллеливания метода трехдиагональной прогонки рассматриваются:

- алгоритм встречной прогонки (для организации расчетов на 2-х потоках);
- горизонтально-блочный параллельный алгоритм (для организации расчетов на p потоках);

- алгоритм параметрической прогонки (для организации расчетов на p потоках).

3. Провести вычислительный эксперимент для оценки эффективности различных подходов к распараллеливанию вычислений для решения систем линейных уравнений с трехдиагональной матрицей коэффициентов при неизвестных.

Объект исследования – алгоритмы для параллельных вычислений и методики программирования на многопроцессорных вычислительных системах с общей памятью.

Метод исследования – теоретическое исследование, практическая реализация и вычислительный эксперимент.

Область применения – изучение и практическое решение задач, сводимых к решению систем линейных уравнений с трехдиагональными матрицами коэффициентов при неизвестных.

В частности, полученные результаты могут быть использованы для численного решения краевых одномерных задач для уравнения теплопроводности, требующих быстрой обработки данных больших размерностей и допускающих распараллеливание алгоритма.

Результаты работы – разработка приложений, с использованием технологий распараллеливания вычислений таких, как технологии потоков Thread и OpenMP, в частности, реализация алгоритмов распараллеливания прогонки для решения систем трехточечных уравнений, возникающих при моделировании решений краевых задач для уравнений в частных производных.

Результаты исследования могут быть использованы в учебном процессе с целью формирования навыков и умений выстраивать и анализировать параллельные алгоритмы решения основных задач вычислительной математики.

ГЛАВА 1. ОПИСАНИЕ ТЕХНОЛОГИИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

1.1. Оценка эффективности параллельных вычислений

Параллельное программирование – это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (*multithreading*) [1].

Параллельная программа - программа, допускающая параллельное (распределённое) исполнение, то есть, исполнение отдельных частей программы (так называемых задач) на более, чем одном исполняющем устройстве одновременно.

Параллельная программа (алгоритм) может быть разработана как параллельная изначально, либо быть преобразована (параллелизация) из последовательной программы (алгоритма).

Программное обеспечение, используемое на последовательных компьютерах может быть преобразовано путем выделением параллельных ветвей, ускоряющих их выполнение на параллельных вычислительных системах.

При разработке параллельных программ выделяют четыре основных этапа [2]:

1. Декомпозиция задачи на подзадачи, которые реализуются независимо.

Существует два основных подхода к декомпозиции задач это параллелизм данных (data parallel) и параллелизм задач (message passing). В первом случае происходит разбиение данных, с которыми связываются операции алгоритма их обработки, а во втором – сначала преобразуется сам алгоритм обработки данных.

2. Определение для сформированного набора подзадач информационных взаимодействий.

На этом этапе устанавливаются коммуникации (связи) между фундаментальными подзадачами. Определяется коммуникационная модель

передачи данных между подзадачами. Выбираются алгоритмы и методы коммуникаций.

3. Масштабирование подзадач, определение количества процессоров.

Масштабирование разработанной параллельной вычислительной схемы необходимо проводить в случае, если количество имеющихся подзадач отличается от числа используемых процессоров. Если количество процессоров меньше, чем число назначенных задач, то следует выполнить укрупнение или агрегацию вычислений.

4. Определение архитектуры системы, закрепление подзадач за процессорами.

На этом этапе осуществляется назначение укрупненных подзадач определенным процессорам в соответствии с требованиями снижения коммуникационных затрат и обеспечения параллелизма [3].

На рис. 1.1 представлена возможная схема взаимосвязи типовых этапов разработки алгоритмов параллельных вычислений.

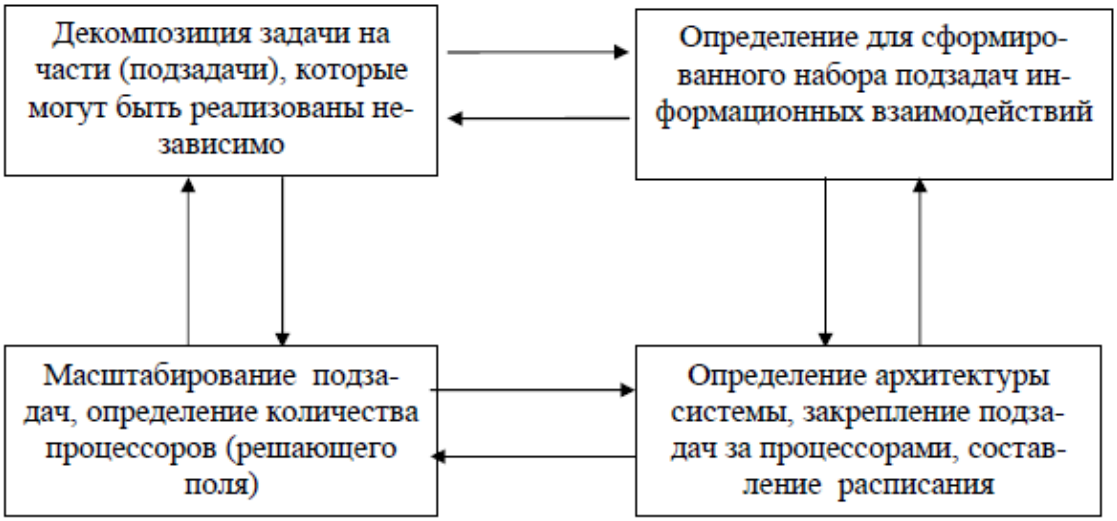

Рис. 1.1. Общая схема взаимосвязи этапов разработки параллельных алгоритмов

Последовательная программа может быть переработана в параллельный вариант для переноса на параллельную ЭВМ, что не всегда приводит к ускорению вычислений. Усилия, затрачиваемые на эту переработку, в

значительной степени зависят от типа решаемой задачи, а именно: допускает ли задача распараллеливание по данным (параллелизм данных) или имеет место лишь параллелизм задач.

Производительность компьютерной программы – величина, обратная процессорному времени, затрачиваемому на выполнение программы.

Степень параллелизма компьютера – количество информации, которое может быть обработано одновременно. Наибольший параллелизм высокопроизводительной вычислительной системы достигается при обработке количества информации, кратного степени параллелизма компьютера.

Степень параллелизма алгоритма – число арифметических операций, которое можно выполнять независимо, т.е. параллельно [4].

Для оценки эффективности параллельной программы рассматриваются такие показатели, как ускорение и эффективность.

Ускорение параллельного алгоритма при наличии в системе p процессоров, по сравнению с последовательным вариантом выполнения, определяется по формуле:

$$S_p = T_1 / T_p \ ,$$

т.е. как отношение времени выполнения последовательного алгоритма к времени выполнения параллельного алгоритма. T_1 – время выполнения программы на одном процессоре; T_p –время выполнения программы на системе из p процессоров.

Эффективность использования параллельной программой ресурсов многопроцессорной вычислительной системы при решении задачи определяется соотношением:

$$E_p = \frac{T_1}{p \cdot T_p} = S_p / p$$

Эффективность показывает, насколько используются вычислительные ресурсы системы. Теоретическое значение эффективности равно единице в идеальном случае.

При создании высокопроизводительных программ необходимо стремиться к тому, чтобы $S_p \to p$ и $E_p \to 1$.

Максимальное значение ускорения зависит от доли в выполняемых вычислениях последовательных расчетов, которые не могут быть распараллелены. Джин Амдал (Gene Amdahl) показал, что верхняя граница для ускорения определяется долей последовательных вычислений алгоритма:

$$S_p \le \frac{1}{f + (1-f)p},$$

где f – доля последовательных вычислений в применяемом алгоритме обработки данных, p – число процессоров [3].

Например, если доля последовательных вычислений составляет 25%, то максимально достижимое ускорение для параллельного алгоритма равно:

$$S^* \le \frac{1}{f} = \frac{1}{0{,}25} = 4$$

В «законе Амдала» допускается, что доля последовательных расчетов является постоянной величиной и не зависит от вычислительной сложности решаемой задачи.

Но в большинстве случаев величина f является убывающей функцией от n, где n – параметр сложности задачи. В этом случае ускорение может быть увеличено при увеличении вычислительной сложности задачи.

Архитектурные особенности параллельной вычислительной системы могут нарушать выводы «закона Амдала». Например, параллельный алгоритм уменьшает объем данных, используемых каждым процессором, и повышает эффективность использования кэш-памяти каждого процессора. Оптимальная работа с кэш-памятью может сильно увеличить быстродействие алгоритма.

В [4] приведены теоретические оценки T_1 и T_p для задачи определения скалярного произведения векторов с использованием последовательного и параллельного алгоритмов.

Операция покомпонентного умножения координат векторов выполнялась параллельно, сложение полученных произведений по обычному

последовательному алгоритму было бы неэффективно для параллельных вычислений, поэтому применялся алгоритм сдваивания при суммировании, когда одновременно с покомпонентными произведениями суммируются соседние пары слагаемых, а затем их суммы.

Данный подход является показательным для оценки ускорения и эффективности параллельных вычислений.

На рис.1.2 представлены графики приближенной оценки ускорения и эффективности параллельного алгоритма скалярного произведения двух векторов (размер задачи n = 1000). Сравниваются два алгоритма: последовательный алгоритм суммирования и алгоритм сдваивания. Из рисунка следует, что применение алгоритма сдваивания повышает ускорение и эффективность алгоритма скалярного произведения.

При увеличении количества процессоров и неизменном размере задачи ($n = const$) темп роста ускорения $S_p = S_p(n, p)$ снижается. Это можно объяснить увеличением доли коммуникационных затрат в общем времени выполнения параллельного алгоритма.

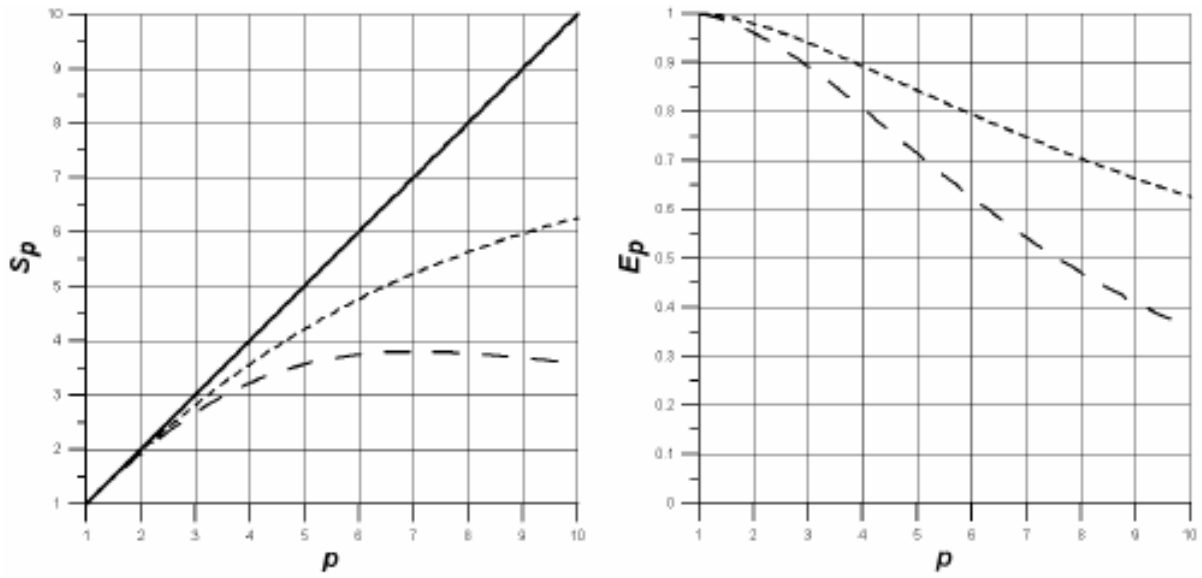

Рис.1.2. Графики приближенной оценки ускорения и эффективности параллельного алгоритма скалярного произведения двух векторов ($n = 1000$) при использовании последовательного алгоритма суммирования (крупный штрих) и алгоритма сдваивания (мелкий штрих).

График ускорения имеет максимум, в котором достигается наибольшее ускорение параллельного алгоритма и дальнейшее увеличение числа используемых процессоров становится нецелесообразным (наблюдается точка насыщения $S_p = S_p(n, p)$ параллельного алгоритма) [4].

1.2. Параллельная обработка данных с использованием класса Thread

В системах с общей памятью, включая многоядерные архитектуры, параллельные вычисления могут выполняться как при многопроцессном выполнении, так и при многопоточном выполнении.

Многопроцессное выполнение предполагает, что для каждая подзадача оформляется в виде отдельной программы (процесса). В этом случае бывает сложно организовать взаимодействие подзадач.

Каждый процесс функционирует в своем виртуальном адресном пространстве, не пересекающемся с адресным пространством другого процесса. Для взаимодействия подзадач необходимо использовать специальные средства межпроцессной коммуникации (интерфейсы передачи сообщений, общие файлы, объекты ядра операционной системы) [4].

Использование потоков для организации параллельных вычислений позволяет выделить подзадачи в рамках одного процесса.

Все потоки одного приложения работают в рамках одного адресного процесса и, тем самым, разделяют данные программы. Для взаимодействия потоков не нужно применять какие-либо средства коммуникации. Поток – это независимый путь исполнения, способный выполняться одновременно с другими потоками.

Общность данных, с одной стороны, существенно упрощает организацию взаимодействия потоков (результат, вычисленный одним потоком, сразу становится доступным всем остальным потокам программы), но, с другой стороны, требует соблюдения определенных правил использования разделяемых данных [10].

Работа потоков с общими переменными может приводить к изменению работы других потоков, поэтому для регулирования порядок работы потоков с данными существую средства синхронизации работы потоков.

Многопоточность является важным средством многозадачного программирования среды .NET. Язык программирования C# поддерживает параллельное выполнение кода через многопоточность.

Для одновременно выполняющихся потоков операционная система частично «забирает» процессорное время у собственных задач и других приложений, выделяя его новым потокам.

Поток состоит из нескольких структур [10].

Ядро потока – содержит информацию о текущем состоянии потока: приоритет потока, программный и стековый указатели. Программный и стековые указатели образуют контекст потока и позволяют восстановить выполнение потока на процессоре.

Блок окружения потока - содержит заголовок цепочки обработки исключений, локальное хранилище данных для потока и некоторые структуры данных, используемые интерфейсом графических устройств (GDI) и графикой OpenGL.

Состояния потоков (рис.1.3).

Каждый поток может находиться в одном из нескольких состояний. Поток, готовый к выполнению и ожидающий предоставления доступа к центральному процессору, находится в состоянии «Готовый».

Поток, который выполняется в текущий момент времени, имеет статус «Выполняющийся».

При выполнении операций ввода-вывода или обращений к функциям ядра операционной системы, поток снимается с процессора и находится в состоянии «Ожидает».

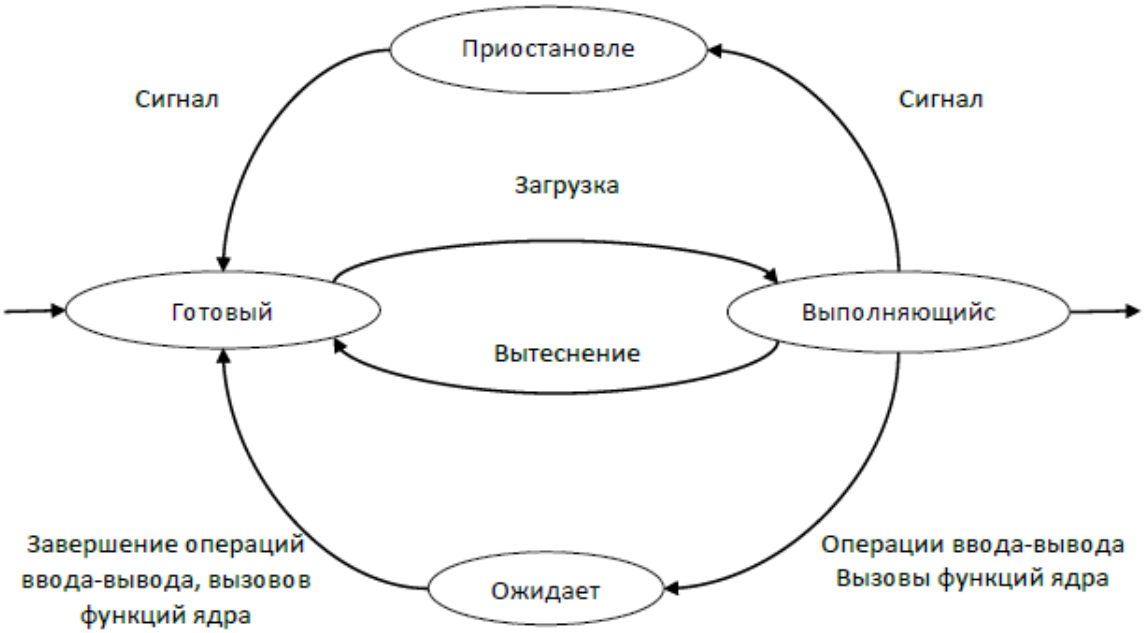

Рис.1.3. Состояния потоков.

При завершении операций ввода-вывода или возврате из функций ядра поток помещается в очередь готовых потоков. При переключении контекста поток выгружается и помещается в очередь готовых потоков.

В среде NET Framework существует два типа потоков: основной и фоновый (вспомогательный). В целом отличие между ними одно – если первым завершится основной поток, то фоновые потоки в его процессе будут также принудительно остановлены, если же первым завершится фоновый поток, то это не повлияет на остановку основного потока – тот будет продолжать функционировать до тех пор, пока не выполнит всю работу и самостоятельно не остановится. Обычно при создании потока ему, по - умолчанию, присваивается основной тип [12].

Программа на языке программирования C# запускается как единственный поток, автоматически создаваемый CLR и операционной системой («главный» поток), и становится многопоточной при помощи создания дополнительных потоков.

Для работы с потоком в C# имеется пространство имен System.Threading, которое содержит в себе классы, поддерживающие многопоточное программирование. Для создания объекта потока и управления отдельным

потоком, необходимо организовать отдельный объект класса Thread из пространства имён System.Threading.

Объект этого класса сопровождает функционирование каждого потока, дополнительно главному потоку процесса.

При создании данного бъекта указывается имя метода, который будет выполняться параллельно с главным потоком или имя делегата.

Созданный объект будет существовать до тех пор, пока выполняется этот метод, затем он автоматически уничтожается.

При создании объекта автоматически создаётся отдельный поток. При завершении потока автоматически разрушается связанный с ним объект.

В приложении всегда выполняется хотя бы один поток – основной поток данного приложения. Этим потоком тоже можно управлять.

Все потоки одного приложения работают в рамках одного адресного процесса. Потоки могут непосредственно обращаться к общим переменным, которые могут быть изменены другими потоками.

Организация многопоточной обработки данных используется для более эффективного использования вычислительных ресурсов компьютера при решении отдельной задачи, когда она разбивается на подзадачи и решение каждой подзадачи выполняется отдельным методом в отдельном потоке.

1.3. Организация параллельной обработки данных с использованием технологии OpenMP

OpenMP – это технология написания параллельных программ для систем с общей памятью. Технология OpenMP использует встроенную библиотеку для организации распараллеливания вычислений, состоит из набора директив компилятора и библиотечных функций и дает возможность создавать многопоточные приложения на языках C/C++, Fortran. Поддерживается производителями аппаратуры (Intel, HP, IBM, и т.д.), разработчиками компиляторов (Intel, Microsoft, и т.д.).

Одним из популярных средств программирования для компьютеров с общей памятью, базирующихся на традиционных языках программирования и использовании специальных комментариев, в настоящее время является технология OpenMP (Open Multi-Processing).

OpenMP – это интерфейс прикладного программирования для создания многопоточных приложений, предназначенных в основном для параллельных вычислительных систем с общей памятью (SMP-систем).

OpenMP включает в себя набор директив для компиляторов и библиотек специальных функций [8].

С помощью средств OpenMP можно создавать многопоточные приложения на алгоритмических языках Fortran и C/C++. При этом директивы OpenMP аналогичны директивам препроцессора для языка C/C++. Это позволяет в любой момент разработки параллельной программы при необходимости вернуться к последовательному варианту.

Основой программы с использованием технологии OpenMP за является последовательная программа. Для создания параллельной версии пользователь может использовать:

1) директив компилятора,

2) процедур (вспомогательных функций),

3) переменных окружения (среды).

Важным достоинством технологии OpenMP является возможность реализации так называемого инкрементального программирования, когда программист постепенно находит участки в программе, содержащие ресурс параллелизма, с помощью предоставляемых механизмов делает их параллельными, а затем переходит к анализу следующих участков [9].

Программа, написанная с использованием технологии OpenMP, состоит из последовательных и параллельных (однопоточных и многопоточных) участков и представляется моделью распараллеливания «Fork/Join» (Разветвление/Слияние).

Вначале программы имеется один единственный поток, его называют начальным потоком или ещё основной нитью (Master thread).

В точке распараллеливания вычислительного процесса выполнение операции FORK организует создание дополнительных нитей, которые нумеруются последовательными натуральными числами.

Все порожденные нити выполняют одну и ту же программу, соответствующую параллельной области.

При выходе из параллельной области основная нить дожидается завершения работы остальных нитей (выполняется операция JOIN), и дальнейшее выполнение программы осуществляется основной нитью [11].

На рисунке 1.4 изображена модель распараллеливания «Fork/Join».

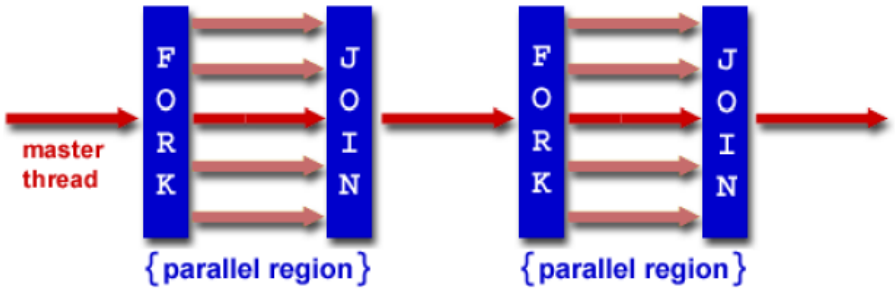

Рис. 1.4. Схема распараллеливания «Fork/Join».

Контроль числа потоков выполняющих параллельную часть программы можно осуществлять разными способами, например использовать OMP_NUM_THREADS или вызвать процедуру omp_set_num_threads().

OpenMP доступна для всех нитей, а локальная - только для одной.

Сформулируем основные возможности и преимущества технологии распараллеливания OpenMP [7]:

1. OpenMP предоставляет возможность реализации максимально эффективных многопроцессорных вычислительных систем с общей памятью, используя для этого общие данные для параллельно выполняемых потоков, без каких-либо затруднений для межпроцессорных передач данных.
2. Возможность поэтапной разработки параллельных вычислительных программ позволяет получить параллельную программу, используя директивы OpenMP, которые могут добавляться в последовательную программу постепенно.

3. В OpenMP несложный набор директив для изучения. Разработка относительно простых параллельных программ не вызовет трудностей, порой достаточно включить пару директив в код последовательной программы. Но нужно помнить, что для написания сложных программ требуется наличие углубленных знаний [8].

Структура OpenMP

В составе структуры OpenMP можно выделить следующее:

- Директивы.
- Функции.
- Переменные окружения.

В общем виде формат директив можно представить в виде:

#pragma omp <Имя директивы> [<параметр>[[,] <параметр>]…], где #pragma omp <Имя директивы> - это фиксированная часть, а параметром может быть хоть сколько угодно.

Основные директивы **#pragma omp**:

- parallel – используется для выделения параллельной области;
- DO/for, section – используется для автоматического распределения работы между потоками (распараллеливания циклов);
- shared, private – используется для определения классов переменных;
- critical, atomic, barrier – для синхронизации потоков.

Общий вид для первых двух директив:

#pragma omp parallel[<параметр>[[,] <параметр>]…]
<код программы>
#pragma omp for[<параметр>[[,] <параметр>]…]
<Цикл for>

Основные функции:

- omp_get_num_threads() – возвращает количество потоков в текущей параллельной области;

- omp_get_thread_num() – возвращает номер текущей нити (все нити нумеруются с 0).

Функции управления свойствами параллельных областей:

- omp_set_nested(true|false),
- omp_set_dynamic(true|false)

Переменные окружения:

- OMP_DYNAMIC – значение говорит о том, разрешено ли программе менять количество процессов динамически.
- OMP_NUM_THREADS – количество процессов в параллельной области по умолчанию, и пр.

OpenMP может использоваться совместно с другими технологиями параллельного программирования, например, с MPI. Обычно в этом случае MPI используется для распределения работы между несколькими вычислительными узлами, а OpenMP затем используется для распараллеливания на одном узле [8].

ГЛАВА 2. ОПИСАНИЕ ПОДХОДОВ К ЧИСЛЕННОМУ РЕШЕНИЮ СИСТЕМ С ЛЕНТОЧНОЙ МАТРИЦЕЙ

2.1. Организация вычислений при решении линейных систем с трехдиагональными матрицами методом прогонки

Особенность решения систем линейных уравнений с ленточной матрицей состоит в том, что в каждом уравнении содержится только часть искомых неизвестных, расположенных последовательно. В ленточной матрице все ее ненулевые элементы расположены вблизи главной диагонали.

Известно, что задача решения системы линейных уравнений методом Гаусса имеет сложность $O(n^2)$ и при большом объеме данных распараллеливается классическим подходом передачей пересчета различных строк разным исполнителям [5]. Очевидно, что в случае ленточных матриц подобный подход будет неэффективным, так как большая часть коэффициентов при неизвестных равна нулю.

Для ленточных, также, как и для сильно разреженных матриц, нет простого решения. Свойство разреженности предполагает существование алгоритма, использующего разреженность матрицы коэффициентов системы, что позволяет выполнять вычисления эффективнее по сравнению со стандартными алгоритмами, с другой стороны, распараллеливание расчетов ведет к значительному усложнению алгоритма, что может привести к отрицательной эффективности распараллеливания.

Рассмотрим ряд подходов к распараллеливанию метода трехдиагональной прогонки, позволяющих получить приемлемую эффективность для ленточных, в частности, трехдиагональных матриц.

Классический метод прогонки входит в базовый курс «Численные методы», при этом итерации алгоритма выполняются последовательно. Однако, алгоритм расчета системы уравнений с трехдиагональной матрицей можно видоизменить, есть модификации данного метода, позволяющие организовать вычисления отдельных частей алгоритма параллельно.

В работе рассмотрены несколько вариантов метода трехдиагональной прогонки:

- классический последовательный метод прогонки;
- параллельные варианты метода прогонки:
  - встречная прогонка (число потоков $p = 2$);
  - блочная прогонка (число потоков $p > 2$);
  - параметрическая прогонка (число потоков $p > 2$).

Система линейных алгебраических уравнений с трехдиагональной матрицей коэффициентов при неизвестных имеет вид (рис.2.1):

$$\begin{pmatrix} b_0 & -c_0 & 0 & 0 & 0 & \dots & 0 \\ -a_1 & b_1 & -c_1 & 0 & 0 & \dots & 0 \\ 0 & -a_2 & b_2 & -c_2 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -a_{n-2} & b_{n-2} & -c_{n-2} & 0 \\ 0 & \dots & 0 & 0 & -a_{n-1} & b_{n-1} & -c_{n-1} \\ 0 & \dots & 0 & 0 & 0 & -a_n & b_n \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{n-2} \\ f_{n-1} \\ f_n \end{pmatrix}.$$

Рис. 2.1. Вид системы линейных уравнений с трехдиагональной матрицей коэффициентов при неизвестных.

К системам такого вида часто сводится разностная аппроксимация дифференциальных уравнений, описывающих физические процессы. Например, краевые задачи для уравнений теплопроводности. Также такие системы возникают при построении кубических сплайнов.

Метод прогонки является модификацией метода Гаусса, использующий специальный вид матрицы системы, и привлекателен своей экономичностью и простотой реализации на компьютерах с последовательной архитектурой.

Приведем формулы алгоритма прогонки для последовательных вычислений [6].

Система линейных уравнений с трехдиагональной матрицей коэффициентов при неизвестных записывается в виде:

$$a_i x_{i-1} - b_i x_i + c_i x_{i+1} = f_i \quad i = 0, \ldots, n, \tag{2.1}$$

$$a_0 = 0, \quad c_n = 0,$$

Алгоритм метода прогонки можно организовывать разными способами. Разница в том, как происходит вычисления прогоночных коэффициентов и в каком порядке находятся неизвестные системы.

Алгоритм правой прогонки:

Алгоритм метода прогонки разделяется на прямой и обратный ход:

1. Прямой ход – вычисляются специальные прогоночные коэффициенты α и β по формулам:

$$\alpha_1 = \frac{-b_0}{c_0}, \qquad \beta_1 = \frac{f_0}{c_0},$$

$$\alpha_{i+1} = \frac{-b_i}{a_i \alpha_i + c_i} \qquad \beta_{i+1} = \frac{f_i - a_i \beta_i}{a_i \alpha_i + c_i}, \quad i = 1,\ldots,n-1. \tag{2.2}$$

2. Обратный ход – вычисляются неизвестные x_i.

$$x_n = \frac{f_n - a_n \beta_n}{a_n \alpha_n + c_n}, \quad x_i = \alpha_{i+1} x_{i+1} + \beta_{i+1}, \quad i = n-1,\ldots,0. \tag{2.3}$$

Для вычислительной устойчивости метода прогонки необходимо выполнение условия диагонального преобладания [7]: $|b_0| \geq |c_0|$, $|b_n| \geq |a_n|$, $|b_i| > |a_i + c_i|$, $i = 1,..n-1$. Формулы прогонки можно применять, если знаменатели дробей (2.2- 2.3) не обращаются в ноль.

В [9] приведены оценки трудоемкости метода прогонки. Прямой ход метода прогонки по формулам (2.2) реализуется за $8(n-2)+2$ операций. Обратный ход по формулам (2.3) потребует $2(n-1)+5$ операций.

Общее число операций оценивается величиной *10n +const*. Время решения системы методом прогонки при больших значения размерности системы n

будет определяться как $T_1 = 10n\tau$, где τ время выполнения одной операции. Таким образом, при увеличении размера системы, время расчета по последовательному алгоритму возрастает линейно.

Алгоритм трехдиагональной прогонки, в котором прогоночные коэффициенты α_i, β_i находятся в прямом ходе ($i = 1,\ldots,n$) по формулам (2.2), а при обратном ходе определяются неизвестные x_i ($i = n,\ldots,1$) по формулам (2.3), называется правой прогонкой.

Симметричный метод левой прогонки производит вычисление прогоночных коэффициентов справа налево ($i = n,\ldots,1$), а значения неизвестных x_i определяются слева направо ($i = 1,\ldots,n$).

Формулы левой прогонки.

Предполагая, что $x_{i+1} = \xi_{i+1}x_i + \eta_{i+1}$, использует данное равенство для исключения переменной x_{i+1}, из i-го уравнения системы (2.1):

$a_i x_{i-1} - b_i x_i + c_i(\xi_{i+1}x_i + \eta_{i+1}) = f_i$ или

$$x_i = \frac{a_i}{b_i - c_i\xi_{i+1}}x_{i-1} + \frac{f_i - c_i\eta_{i+1}}{b_i - c_i\xi_{i+1}}, \qquad (2.4)$$

Сравнивая (2.4) с формулой $x_i = \xi_i x_{i-1} + \eta_i$ получим расчетные формулы прогоночных коэффициентов для левой прогонки (2.5).

Прямой ход:

$$\xi_n = \frac{a_n}{c_n}, \qquad \xi_i = \frac{a_i}{b_i - c_i\xi_{i+1}}, \quad i = n-1,\ldots,1. \qquad (2.5)$$

$$\eta_n = \frac{f_n}{c_n}, \qquad \eta_i = \frac{f_i - c_i\eta_{i+1}}{b_i - c_i\xi_{i+1}}, \quad i = n-1,\ldots,1.$$

Значение x_1 определяются из первого уравнения исходной системы и условия $x_1 = \xi_1 x_0 + \eta_1$. Далее, используя условие $x_i = \xi_i x_{i-1} + \eta_i$ и известные коэффициенты ξ_i, η_i можно найти все остальные значения неизвестных.

Обратный ход:

$$x_0 = \frac{f_0 - c_0\eta_1}{b_0 - c_1\xi_1} \qquad x_{i+1} = \xi_{i+1}x_i + \eta_{i+1}, \quad i = 0,\ldots,n-1. \qquad (2.6)$$

Общее число операций левой прогонки составляет также $10n + const$ [7].

Методы правой и левой прогонки предполагают последовательное исполнение вычислений, что не допускает распараллеливания алгоритма.

2.2. Метод встречной прогонки и его распараллеливание

Метод встречной прогонки дает возможность организовать параллельный расчет в двух потоках. В первом потоке определяются прогоночные коэффициенты α_i, β_i метода правой прогонки для уравнений с номерами $1 \le i \le p$. Второй поток находит прогоночные коэффициенты метода левой прогонки ξ_i, η_i для уравнений с номерами $p \le i \le n$, $p = \left[\frac{N}{2}\right]$

При $i = p$ проводится сопряжение решений в виде (2.3) и (2.6): определяется значение x_p из системы двух уравнений:

$$\begin{cases} x_p = \alpha_{p+1} x_{p+1} + \beta_{p+1} \\ x_{p+1} = \xi_{p+1} x_p + \eta_{p+1} \end{cases} \qquad x_p = \frac{f_p - a_p \beta_p - b_p \eta_{p+1}}{c_p + a_p \alpha_p + b_p \xi_{p+1}} \qquad (2.7)$$

Определив значение x_p, можно найти все x_i, при $1 \le i < p$ в первом потоке, и все значения x_i, при $p < i \le N$ во втором потоке.

Время на выполнение метода параллельной встречной прогонки можно оценить как

$$T_1 = 5n\tau + \delta, \qquad (2.8)$$

где δ – время, необходимое на организацию и закрытие параллельной секции. Следует отметить, что расчеты и при прямом, и при обратном ходе производятся независимо, теоретическое ускорение здесь должно быть равно двум [4].

При реализации встречной прогонки необходимо создать два потока. Первый поток обрабатывает половину СЛАУ по формулам (2.2) и (2.3), а второй поток обрабатывает вторую половину СЛАУ по формулам (2.5) и (2.6).

На рис 2.2 представлена схема параллельного расчета для метода встречной прогонки. Во время его работы в параллельном режиме создается два потока (за исключением этапа разрешения встречной прогонки, когда работает только один процессор).

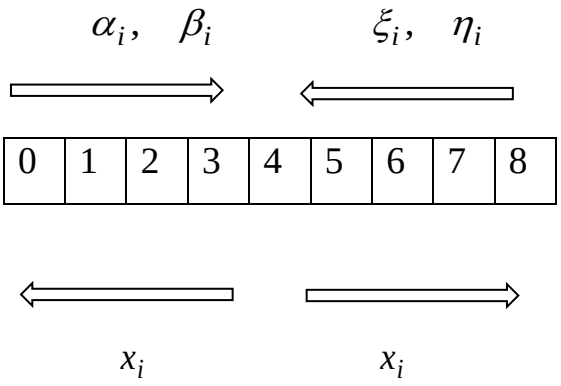

Рис. 2.2. Метод встречных прогонок, n = 8

Для равномерной загрузки процессоров необходимо, чтобы n было четным.

2.3. Горизонтально-блочный параллельный алгоритм

Идея метода заключается в том, что каждый поток будет обрабатывать $m = [N / p]$ строк матрицы A, т.е. k-й поток обрабатывает строки с номерами $1 + (k-1)m \le i \le km$.

Рассмотрим случай системы из 12 уравнений, число потоков p =3 (рис.2.3) [9].

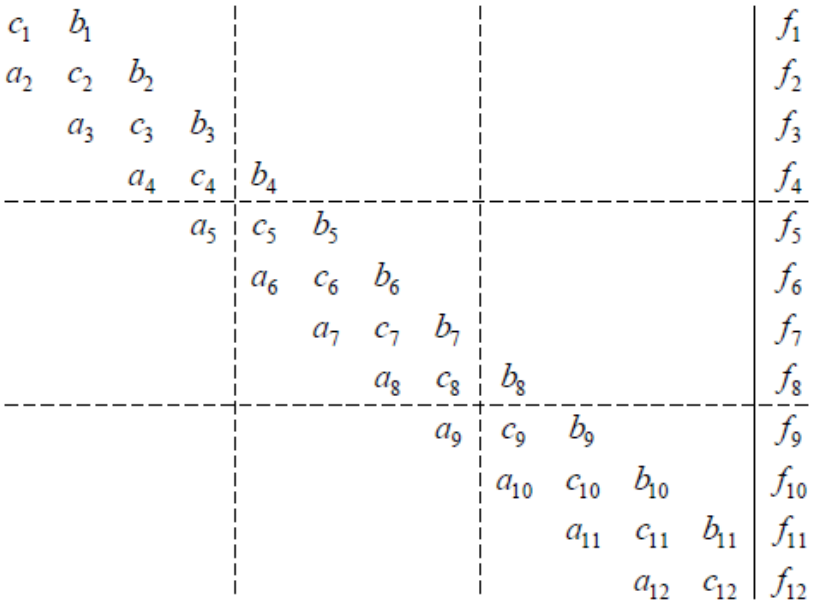

Рис.2.3 Вид матрицы и вектора правых частей для системы уравнений при $n = 12$

В каждой полосе матрицы, соответствующей потоку с номером k, организуется исключение поддиагональных элементов. Для этого производится вычитание строки с номером i, умноженной на константу a_{i+1} / c_i из строки с номером $i + 1$ с тем, чтобы коэффициент при неизвестной x_i в $(i + 1) -$ й строке стал нулевым.

В результате, в каждой рассматриваемой полосе, за исключением первой, появится новый столбец коэффициентов.

Число ненулевых элементов в строке останется тем же, но изменится структура уравнений (рис.2.4).

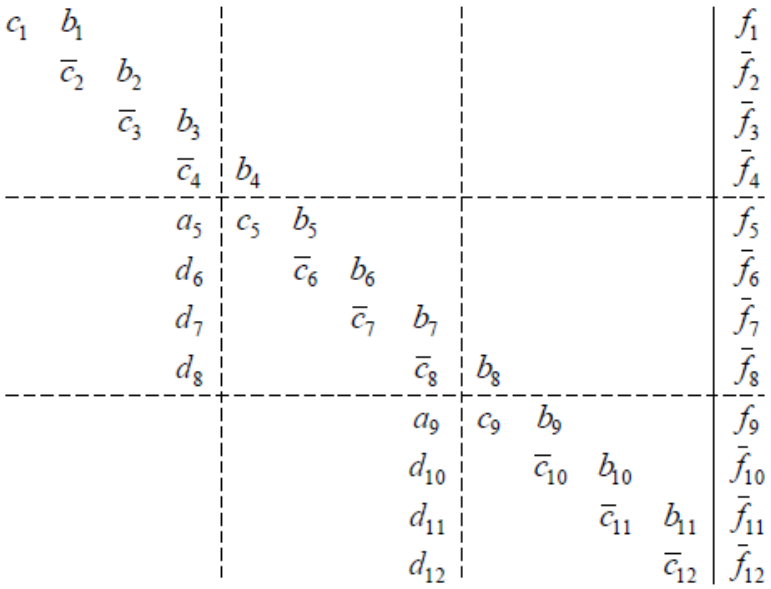

Рис.2.4. Вид матрицы и вектора правых частей после прямого хода

Обратный ход алгоритма организует в каждом потоке исключение наддиагональных элементов, начиная с предпоследнего и заканчивая последним для предыдущего потока. В итоге матрица коэффициентов исходной системы уравнений принимает блочный вид, представленный на рис.2.5):

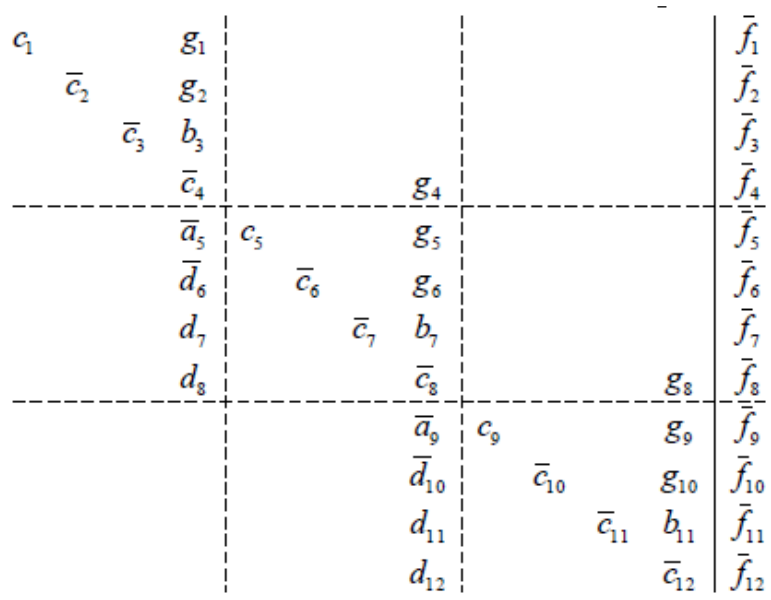

Рис.2.5. Вид матрицы и вектора правых частей после обратного хода

Далее рассматривается система уравнений, содержащая последнее уравнение каждой полосы. Эта система будет содержать p уравнений, и будет трехдиагональной (рис.2.6).

$$\left(\begin{array}{ccc|c} \bar{c}_4 & g_4 & & \bar{f}_4 \\ d_8 & \bar{c}_8 & g_8 & \bar{f}_8 \\ & d_{12} & \bar{c}_{12} & \bar{f}_{12} \end{array}\right)$$

Рис.2.6. Вид системы уравнений, состоящий из граничных уравнений каждого блока

Для решения полученной системы уравнений применяется последовательный метод прогонки. После этого, с помощью найденных граничных значений неизвестных определяются значения внутренних переменных в каждом блоке [9].

Формулы прямого хода алгоритма горизонтально - блочной прогонки.

Прямой ход метода – вычисляются прогоночные коэффициенты α, β и γ для каждой полосы размера $m = [N / p]$ по формулам:

$$\alpha_1 = \frac{-b_0}{c_0}, \qquad \beta_1 = \frac{f_0}{c_0}, \qquad \gamma_1 = \frac{-a_0}{c_0},$$

$$\alpha_{i+1} = \frac{-b_i}{a_i\alpha_i + c_i} \qquad \gamma_{i+1} = \frac{-a_i\gamma_i}{a_i\alpha_i + c_i} \qquad \beta_{i+1} = \frac{f_i - a_i\beta_i}{a_i\alpha_i + c_i}, \quad i = 1,\ldots,m-1. \quad (2.9)$$

Обратный ход метода – вычисляются прогоночные коэффициенты m, l, k:

$$m_{m-1} = \alpha_{m-i}, \qquad l_{m-1} = \beta_{m-i}, \qquad k_{m-1} = \gamma_{m-i}$$

$$m_i = \alpha_i m_{i+1}, \quad l_i = \alpha_i l_{i+1} + \beta_i, \quad k_i = \alpha_i k_{i+1} + \gamma_i, \quad i = m-2,\ldots,1. \qquad (2.10)$$

Затем, (в соответствии с рис.2.6) с помощью найденных коэффициентов формируется трехдиагональная матрица для системы уравнений размера $(p \times p)$, состоящая из уравнений на нижних границах каждой полосы:

$$A_i = -\gamma_{m(i+1)},\ B_i = -\alpha_{m(i+1)} \cdot m_{m(i+1)+1},$$

$$C_i = 1 - \alpha_{m(i+1)} \cdot k_{m(i+1)+1},$$

$$F_i = \alpha_{m(i+1)} \cdot l_{m(i+1)+1} + \beta_{m(i+1)}, \quad i = 2, \dots p-1.$$

$$A_1 = 0\ ,\ \ B_1 = -\alpha_m \cdot m_{m+1} + \beta_m\ ,\ \ C_1 = 1 - \alpha_m \cdot k_{m+1}, \qquad (2.11)$$

$$F_1 = \alpha_m \cdot l_{m+1} + \beta_m\ ,$$

$$A_p = -\gamma_{p \cdot m}\ ,\ \ B_p = 0\ ,\ \ C_p = 1, \quad F_{\text{p}} = \beta_{p \cdot m}$$

Ее решение находится обычным последовательным методом прогонки.

Далее, остальные неизвестные (внутренние переменные в каждом потоке) находятся по формулам:

$$j = 1, \dots p\ ,\ i = (j-1)m, \dots, jm - 1$$

$$x_i = m_{i+1} \cdot x_{j \cdot m-1} + k_{i+1} \cdot x_{(j-1)m-1} + \cdot\, l_{i+1} \qquad (2.12)$$

Эти значения опять можно вычислять независимо в каждом потоке.

Блок-схема расчета горизонтально-блочной прогонки представлена на рис.2.7:

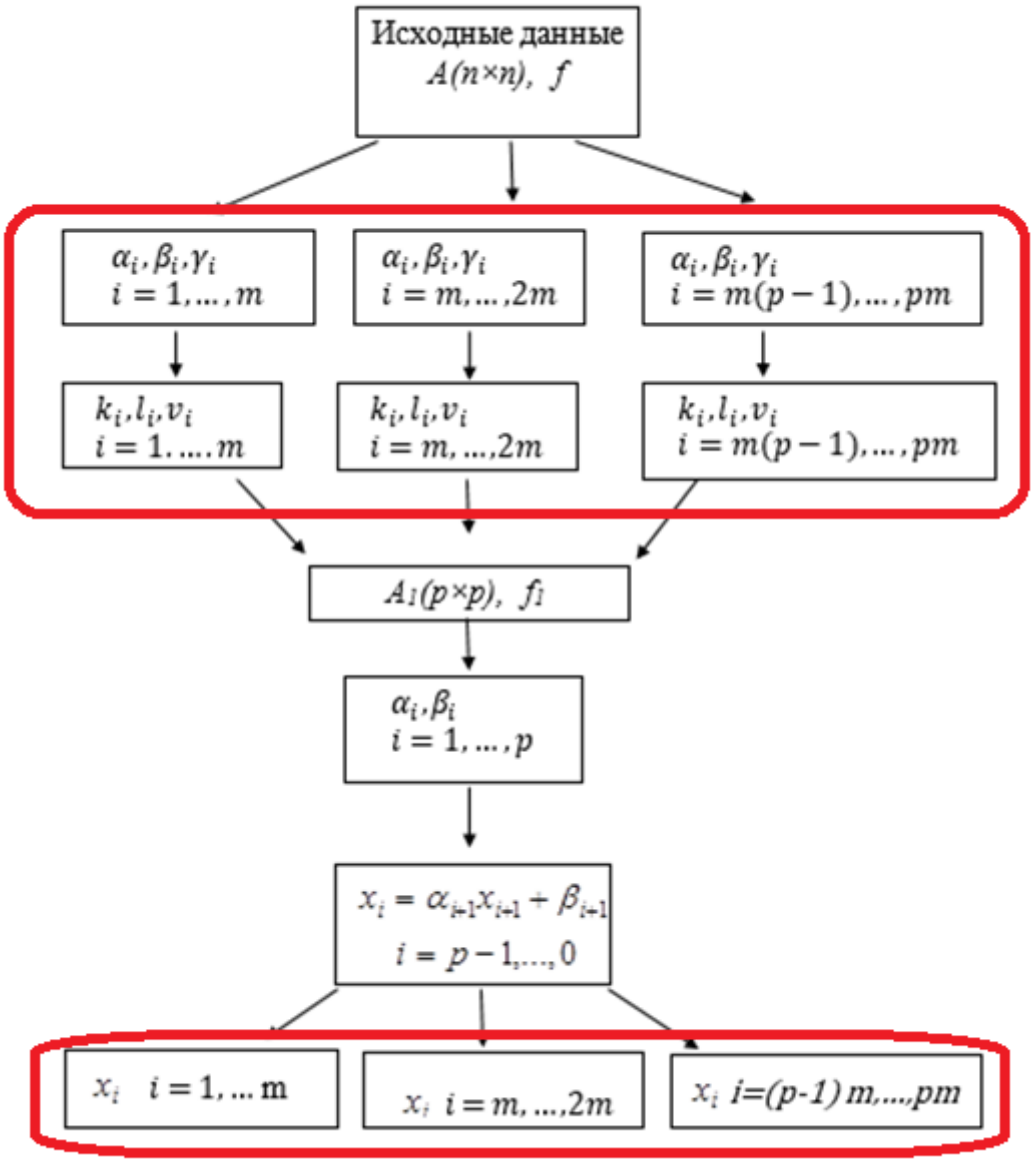

Рис.2.7. Блок-схема метода горизонтально-блочной прогонки

Красными прямоугольниками отмечены области, поддающиеся распараллеливанию.

В [9] приведены оценки трудоемкости параллельного варианта метода блочной прогонки.

Исходные данные для системы уравнений имеют следующие обозначения: матрица коэффициентов A имеет размер $n \times n$, число потоков равно p ($p \le n$), соответственно, $m = n / p$ - размер полосы матрицы A на каждом процессоре.

При выполнении прямого хода алгоритма исключение поддиагональных элементов в каждой полосе оценивается как $8(m-1)$ операций. При выполнении обратного хода – исключение наддиагональных элементов в каждой полосе потребует $7m$ операций.

После формирования вспомогательной трехдиагональной системы уравнений размера $p \times p$, выполняемой в одном потоке, реализуется последовательный метод правой прогонки, который потребует порядка $10p$ операций.

На заключительном этапе для нахождения внутренних переменных каждый процессор выполняет обратный ход алгоритма, который потребует $5(m-1)$ *)* операций. Таким образом, общую трудоемкость параллельного метода прогонки можно оценить как $T_p = 20m + 10p$.

С помощью полученных оценок можно оценить показатели ускорения и эффективности параллельного варианта метода прогонки [9]:

$$S_p = \frac{T_1}{T_p} = \frac{10n}{20\frac{n}{p} + 10p} = p\frac{10n}{20n + p^2}, \qquad E_p = \frac{S_p}{p} = \frac{10n}{20n + p^2}$$

Приведенные соотношения позволяют сделать вывод, что в случае решения системы уравнений с большим числом неизвестных, при котором $p << n$, показатели ускорения и эффективности будут определяться как

$$S_p \approx = \frac{p}{2}, \quad E_p \approx 0.5. \tag{2.13}$$

2.4. Параметрическая прогонка

Идея метода встречных прогонок на случай многопроцессорной системы была обобщена в работе [15] и получила название параметрической прогонки. Основу предлагаемого метода составляют следующие этапы:

1. выделение части искомых неизвестных $\{x_i\}$ в качестве параметрических (обозначены как $\{x_\mu\}$);

2. построение системы линейных уравнений относительно этих неизвестных; - решение этой системы и определение $\{ x_\mu \}$;

3. нахождение по $\{ x_\mu \}$ остальных искомых неизвестных $\{ x_i \}$.

Для осуществления этой идеи система уравнений (2.1) разбивается на подсистемы, количество которых будет совпадать с количеством используемых процессорных элементов p . Причем декомпозиция по данным осуществляется таким образом, что каждый μ -й процессорный элемент (ПЭ) обрабатывает $m+1 = n/p+1$ уравнений (рис. 2.8), а для неизвестных системы вводятся локальные индексы для получения однородного алгоритма [14].

$$x_i = x_{j+\mu m} = x_j^{\{\mu\}}, i = 0,\ldots,n;\ j = 0,\ldots,m;\ \mu = 0,\ldots,p-1$$

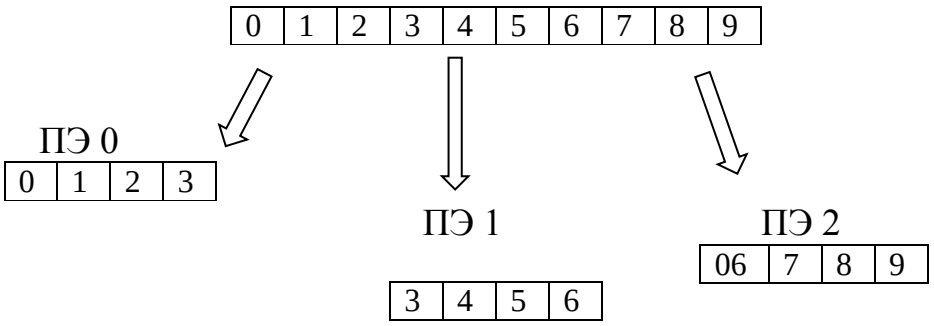

Рис. 2.8. Пример декомпозиции неизвестных между тремя процессорами (число уравнений $n = 9$)

На границах соседних процессорных элементов выполняется:

$$x_m^{\{\mu-1\}} = x_0^{\{\mu\}} = \bar{x}_\mu, \quad \mu = 1,\ldots,p-1; \quad \bar{x}_0 = x_0; \quad \bar{x}_p = x_n.$$

Решение системы можно записать в виде рекуррентных формул:

$$x_m^{\{\mu\}} = v_j^{\{\mu\}} \cdot \bar{x}_\mu + z_j^{\{\mu\}} \cdot \bar{x}_\mu + w_j^{\{\mu\}}, \quad j = 0,\ldots,m, \quad \mu = 1,\ldots,p-1, \qquad (2.14)$$

где $\{v_j^{\{\mu\}}\}$, $\{z_j^{\{\mu\}}\}$, $\{w_j^{\{\mu\}}\}$ решения следующих систем уравнений с трехдиагональной матрицей:

$$\begin{cases} -a_{j+\mu m} v_{j-1}^{\{\mu\}} + b_{j+\mu m} v_j^{\{\mu\}} - c_{j+\mu m} v_{j+1}^{\{\mu\}} = 0, \quad j = 1,\ldots m-1 \\ v_0^{\{\mu\}} = 1, \quad v_m^{\{\mu\}} = 0. \end{cases} \qquad (2.15)$$

$$\begin{cases} -a_{j+\mu m} z_{j-1}^{\{\mu\}} + b_{j+\mu m} z_j^{\{\mu\}} - c_{j+\mu m} z_{j+1}^{\{\mu\}} = 0, \quad j = 1,...m-1 \\ v_0^{\{\mu\}} = 0, \quad v_m^{\{\mu\}} = 1. \end{cases} \tag{2.16}$$

$$\begin{cases} -a_{j+\mu m} w_{j-1}^{\{\mu\}} + b_{j+\mu m} w_j^{\{\mu\}} - c_{j+\mu m} w_{j+1}^{\{\mu\}} = 0, \quad j = 1,...m-1 \\ v_0^{\{\mu\}} = 0, \quad v_m^{\{\mu\}} = 0. \end{cases} \tag{2.17}$$

Системы (2.15 - 2.17) получаются в результате подстановки (2.14) в (2.1) и решаются независимо для каждой μ -ой подобласти обычным методом прогонки. На этом этапе возможно сокращение количества арифметических операций за счет использования одной и той же трехдиагональной матрицы коэффициентов в (2.15 - 2.17).

Решения этих трех систем можно найти методом прогонки, причем независимо на каждом процессоре.

После вычисления необходимых коэффициентов производится построение системы линейных уравнений для параметрических неизвестных $\{x_\mu\}$. Для этого рассматриваются неиспользованные ранее уравнения системы (2.1) при $i = 0, m, 2m,.., pm$, в которых с помощью формулы (2.14) исключаются все «непараметрические» неизвестные.

В результате формируется система с трехдиагональной матрицей ($a_0 = 0, \quad c_n = 0$):

$$\begin{cases} -a_{\mu m} v_{m-1}^{\{\mu\}} \bar{x}_{\mu-1} + (b_{\mu m} - a_{\mu m} z_{m-1}^{\{\mu\}} - c_{\mu m} v_1^{\{\mu\}}) \bar{x}_\mu - c_{\mu m} \bar{x}_{\mu+1} = \\ f_{\mu m} + a_{\mu m} w_{m-1}^{\{\mu-1\}} + c_{\mu m} w_1^{\{\mu\}}, \quad \mu = 1,... p. \end{cases} \tag{2.18}$$

которая решается обычным методом прогонки, в результате определяются искомые $\{x_\mu\}$.

Размерность системы уравнений (2.18) равна количеству процессоров, что существенно меньше количества неизвестных рассматриваемой задачи.

Для определить остальных неизвестные задачи вычисляются по формуле (2.14).

Таким образом, параметрический метод содержит три этапа [13]:

1) нахождение предрешений,

2) нахождения гранично–процессорных решений,

3) восстановление решения.

Первый и третий этапы выполняются независимо на каждом устройстве, для второго этапа потребуется один сопроцессор (на этом этапе применяется метод правой прогонки.).

Число арифметических операций в методе параметрической прогонки складывается как сумма операций на всех этапах: число операций на этапе нахождения предрешений, число операций в расчете коэффициентов матрицы системы для нахождения гранично-процессорных решений, число операций в методе прогонки на этом этапе, число операций при восстановлении решения исходной системы.

В [13] приведены оценки трудоемкости метода параметрической прогонки.

На первом этапе (решение систем (2.15) – (2.17)) каждому процессору требуется около $13m$ арифметических операций, которые выполняются одновременно и независимо от других процессоров. Затем решается система (2.18) одним из процессоров. Для расчета коэффициентов системы (2.18) и нахождения ее решения требуется около $18p$ арифметических операций.

На завершающем этапе каждый процессор одновременно и независимо ведет расчет по формуле (2.14), для чего понадобится $4m$ арифметических операций. В итоге получается оценка

$$T_p = 17m + 18p$$

Показатели ускорения и эффективности метода параметрической прогонки могут быть определены при помощи соотношений следующего вида:

$$S_p = \frac{T_1}{T_p} = \frac{10n}{17\frac{n}{p} + 18p} = p\frac{10n}{17n + 18p^2}, \qquad E_p = \frac{S_p}{p} = \frac{10n}{17n + 18p^2}$$

Из приведенных соотношений видно, что в случае решения системы уравнений с большим числом неизвестных, при котором $p<<n$, показатели ускорения и эффективности будут определяться как

$$S_p \approx 0{,}6p, \quad E_p \approx 0.6 .$$

2.5. Применение алгоритмов параллельной прогонки для решения краевой задачи для уравнения теплопроводности

В качестве применения различных подходов к организации параллельных вычислений рассмотрено определение численного решения краевой задачи для дифференциального уравнения в частных производных параболического типа, уравнения теплопроводности, описывающего процесс распространения тепла в одномерном однородном стержне с граничными условиями первого рода и заданными начальными условиями. Численное решение этой задачи сводится к решению системы алгебраических уравнений с трехдиагональной матрицей коэффициентов при неизвестных.

Дифференциальное уравнение в частных производных, описывающее процесс распространения тепла в одномерном однородном стержне имеет вид [7]:

$$\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2} + f(x,t) \tag{2.19}$$

Без ограничения общности можно считать $\alpha^2 = 1$. Рассмотрим первую краевую задачу в области $D = \{0 \le x \le 1, \quad 0 \le t \le T\}$.

Требуется найти непрерывное в D решение $u = u(x,t)$ задачи

$$\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2} + f(x,t), \quad 0 < x < 1, \quad 0 < t \le T.$$

$$\begin{aligned} &u(x,0) = u_0(x), \quad 0 \le x \le 1, \\ &u(0,t) = u_1(t), \quad u(1,t) = u_2(t), \quad 0 \le t \le T. \end{aligned} \tag{2.20}$$

Для построения разностной схемы в области D вводится сетка:

$$\omega_{h\tau} = \left\{ \begin{array}{l} (x_i, t^n); x_i = ih, \quad 0 \le i \le N, \quad h = 1/N, \\ t^n = n\tau, \quad 0 \le n \le m, \quad \tau = T/m. \end{array} \right\} \tag{2.21}$$

с шагами h по оси x и τ по оси t.

Неявная разностная схема с весами, аппроксимирующая дифференциальное уравнение (2.20) имеет вид []:

$$\frac{u_i^{n+1} - u_i^n}{\tau} = \sigma \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{h^2} + (1-\sigma)\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h^2} + f_i^n.$$

$$\begin{array}{l} u_i^0 = u_0(x_i), \quad 0 \le i \le N, \\ u_0^j = u_1(t^n), \quad u_n^n = u_2(t^n), \quad 0 \le n \le m. \end{array} \tag{2.22}$$

где u_i^n - сеточная функция, являющаяся решением разностной схемы и аппроксимирующая точное решение дифференциального уравнения в узлах сетки; σ - параметр, называемый весом, а f_i^n некоторая правая часть.

Шаблон разностной схемы (2.22) представлен на рис. 1.3.

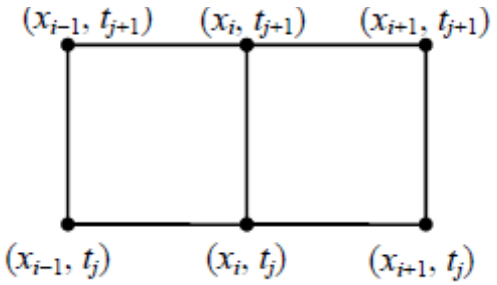

Рис. 2.9. Шаблон разностной схемы с весом $\sigma = \frac{1}{2}$

Частный случай неявной схемы (2.22) при $\sigma = \frac{1}{2}$ называется схемой Кранка - Николсона имеет вид:

$$\frac{u_i^{n+1} - u_i^n}{\tau} = \frac{\alpha^2}{2}\left[\frac{u_{i+1}^{n+1} - u_i^{n+1} + u_{i-1}^{n+1}}{h^2} + \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h^2}\right] + f_i^{n+1/2}. \tag{2.23}$$

$$u_i^0 = u_0(x_i), \quad u_0^n = u_1(t^n), \quad u_N^n = u_2(t^n), \quad 0 \le i \le N, \quad 0 \le n \le m.$$

Система уравнений относительно значений неизвестных на новом $n+1$-ом слое является трехточечной и ее решение находится методом трехдиагональной прогонки:

$$\frac{\alpha^2\tau}{2h^2}u_{i-1}^{n+1}-(1+\frac{\alpha^2\tau}{h^2})u_i^{n+1}+\frac{\alpha^2\tau}{2h^2}u_{i+1}^{n+1}=(1-\frac{\alpha^2\tau}{h^2})u_i^n+\frac{\alpha^2\tau}{2h^2}(u_{i-1}^n+u_{i+1}^n)+\tau f_i^{n+1/2}$$
$$i=1,\ldots,N-1,\quad u_0^n=u_1(t^n),\quad u_N^n=u_2(t^n).$$

(2.24)

Разностная схема Кранка - Николсона (2.24) является абсолютно устойчивой, ее погрешность аппроксимации оценивается как $O(\tau^2+h^2)$ [5]. Данная схема с весом $\sigma=\frac{1}{2}$ обеспечивает хорошую точность при не слишком малых шагах сетки.

Значения неизвестных на *(n+1)*-м слое можно найти, зная значения на предыдущем - *n*-м слое, т.е. вычисления проводятся «по слоям». Таким образом, в качестве подзадачи, допускающей распараллеливание, можно рассмотреть вычисление значений на очередном *(n+1)*-м слое [14].

Неявная конечно-разностная схема (2.24), записанная в виде:

$$a_i\,u_{i-1}^{n+1}-\;b_i\,u_i^{n+1}+\;c_i\,u_{i+1}^{n+1}=\;\;f_i^n\;\;i=0,\ldots,N-1\,,\;n=0,1,2,\ldots \quad (2.25)$$

может быть решена методом прогонки в его параллельном варианте.

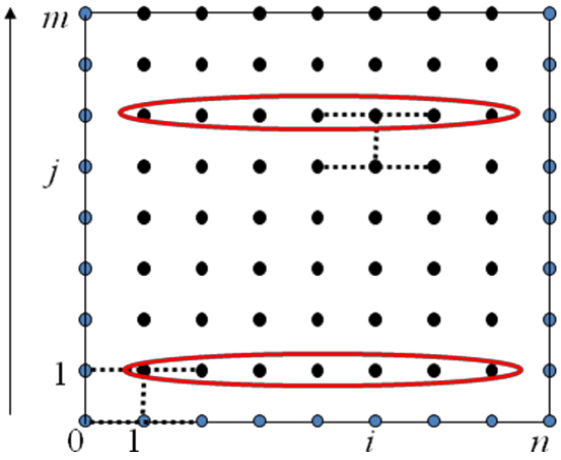

Рис. 2.10. Распараллеливание разностной схемы Кранка-Николсона

На рисунке 2.10 представлен схема решения задачи. Светлыми точками отмечены узлы сетки, значения в которых известны из начальных и граничных условий, темными точками отмечены узлы искомого *n+1* -го слоя, пунктиром изображен шаблон разностной схемы.

Полученная система уравнений на каждом временном $n+1$-м слое имеет вид $Ax = f$, где A -матрица , f - вектор свободных членов.

Элементы матрицы A являются коэффициентами при неизвестных системы линейных алгебраических уравнений:

$$a_i\, u_{i-1}^{n+1} - b_i\, u_i^{n+1} + c_i\, u_{i+1}^{n+1} = f_i^n \quad i = 0, \dots, N-1\,,\ n = 0,1,2, \dots \qquad (2.26)$$

где $a_i = c_i = \dfrac{\alpha^2\tau}{2h^2}$, $b_i = (1 + \dfrac{\alpha^2\tau}{h^2})$,

$$f_i = (1 - \frac{\alpha^2\tau}{h^2})u_i^n + \frac{\alpha^2\tau}{2h^2}(u_{i-1}^n + u_{i+1}^n) + \tau f_i^{n+1/2}, \quad 0 < i < N-1.$$

Для решения системы уравнений с трехдиагональной матрицей коэффициентов при неизвестных были использованы следующие методы:

- классический последовательный метод прогонки

параллельные варианты метода прогонки:

- встречная прогонка (число потоков $p = 2$);
- блочная прогонка (число потоков $p > 2$);

ГЛАВА 3. ЧИСЛЕННЫЕ ИССЛЕДОВАНИЯ ЭФФЕКТИВНОСТИ РАСПАРАЛЛЕЛИВАНИЯ МЕТОДА ПРОГОНКИ

Вычислительные эксперименты для оценки эффективности параллельных вариантов метода прогонки для решения систем линейных уравнений с трехдиагональной матрицей проводились на устройстве со следующими техническими характеристиками (Таблица 3.1)

Таблица 3.1.

Тестовая инфраструктура

Процессор	четырехъядерный процессор Intel Xeon E5520 (2.27 GHz) Intel®Core™ i5-825OU
Память	8 Gb
Операционная система	Microsoft Windows 10
Среда разработки	Microsoft Visual Studio 2017
Компилятор, профилировщик, отладчик	IntelParallelStudioXE

Для тестирования используемых алгоритмов расчеты проводились для системы линейных уравнений с заданной трехдиагональной матрицей коэффициентов с диагональным преобладанием (элементы главной диагонали являлись удвоенной суммой элементов в строке).

Правая часть задавалась из соображений существования точного решения.

Рассматривались задачи размера N, кратного 12 и, чтобы размер блоков был одинаков для всех потоков. Время расчетов T измерялось в миллисекундах. Каждый алгоритм выполнялся 10 раз, после чего рассчитывается среднее время работы алгоритма.

Для оценки разброса получаемых значений строился доверительный интервал для полученного среднего времени расчета:

Статистические показатели рассчитывались по следующим формулам:

$\bar{T} = \frac{1}{n}\sum_{i=1}^{n} T_i$ – среднее арифметическое значение времени расчета.

Дисперсия – $S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(T_i - \bar{T})^2$.

Стандартное отклонение $S_0 = \sqrt{\frac{\sum_{i=1}^{n}(T_i-\bar{T})^2}{n-1}}$.

Доверительный интервал (при 95% значимости) – $T_{1,2} = \bar{T} \pm \frac{S_0}{\sqrt{n}} \cdot c_\gamma$, где T_1 и T_2 – соответственно нижняя и верхняя граница доверительного интервала, S_0 –среднеквадратичное отклонение по выборке, n – размер выборки, γ – доверительная вероятность (обычно равна 0,9, 0,95 или 0,99), $c_\gamma = \Phi^{-1} \cdot (1+\gamma)/2$, c_γ – обратное значение функции нормального распределения (функции Лапласа), т.е. количество стандартных ошибок от средней арифметической до нижней или верхней границы (указанным трем вероятностями соответствуют значения 1,64, 1,96 и 2,58).

Например, для 10 испытаний ($n=10$) при среднем значении $\bar{T}$=248,8 получен такой 224,2- 273,4 доверительный интервал,

Таким образом, влияние внутренних процессов на компьютере на время расчета по рассматриваемым моделям составляет порядка 10%. Учитывая это, в дальнейшем будет использоваться не интервальная, а точечная оценка времени.

На основании полученных результатов было рассчитано ускорение S работы параллельных алгоритмов по формуле:

$$S = \frac{T_1}{T_p}, \text{где}$$

T_1 –время работы последовательного алгоритма, T_p – время работы параллельного алгоритма.

Для выполнения численного эксперимента было создано две программы в среде программирования Visual Studio 2017:

- Приложение на языке программирования C# с использование технологии Thread.
- Приложение на языке программирования C++ с использованием технологии OpenMP.

Приложения реализованы в консольном виде с окном, отображающим результаты расчета.

В приложении на языке программирования C# реализованы варианты алгоритма прогонки:

- Последовательный расчет с помощью левой, правой и встречной прогонок.
- Параллельный расчет с помощью встречной и горизонтально-блочной прогонки.

В приложении на языке программирования C++ реализованы варианты алгоритма прогонки:

- Последовательный расчет с помощью правой и встречной прогонки.
- Параллельный расчет с помощью встречной и горизонтально-блочной прогонки.

На рис.3.1 приведена диаграмма классов приложения на C#.

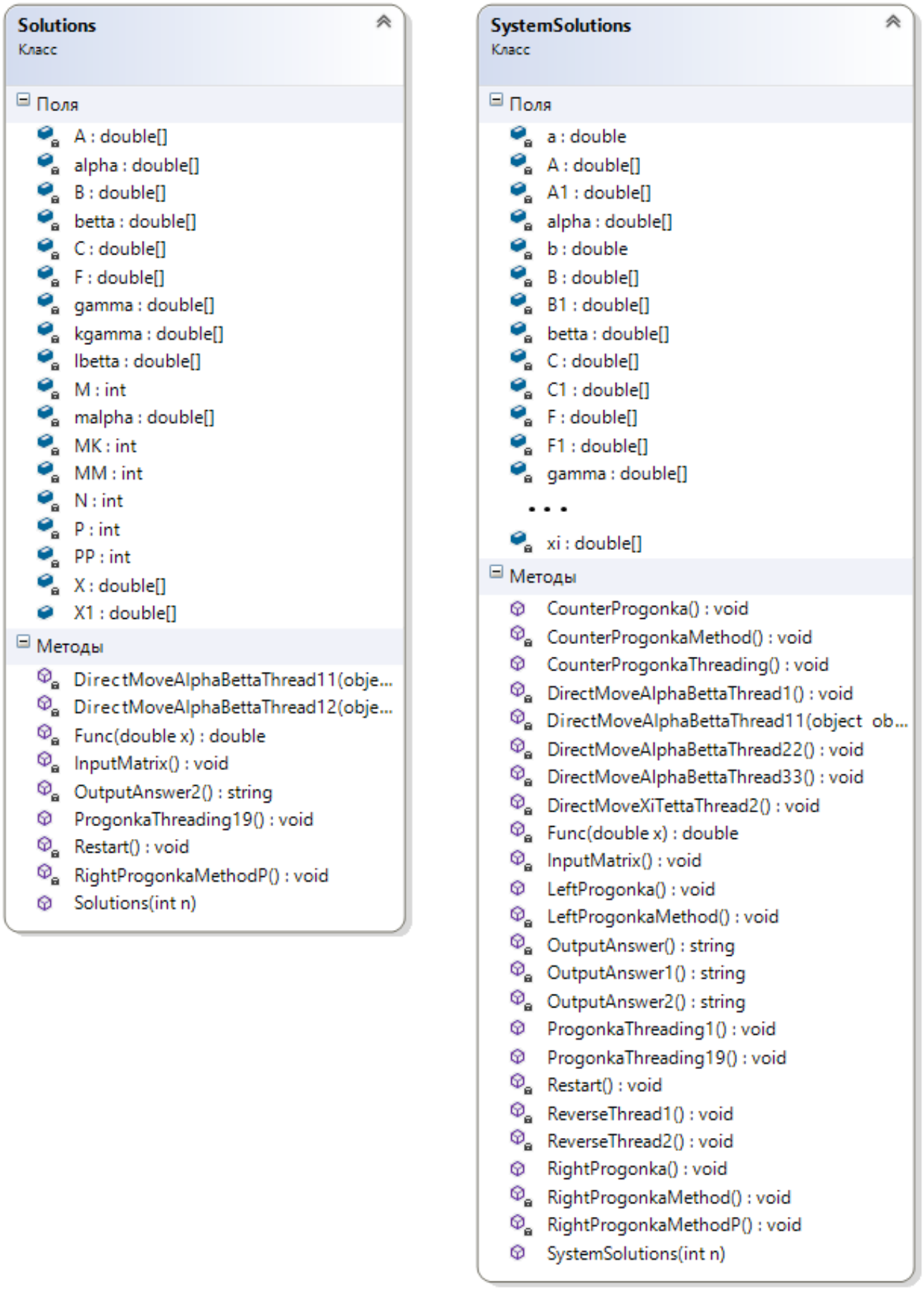

Рис.3.1. Диаграмма классов программы на языке программирования C#

Код методов, реализующих алгоритмы горизонтально–блочной прогонки, представлен в приложении 1.

3.1. Результаты расчета на основе многопоточного программирования .NET (Thread)

В данном разделе приводятся результаты численных экспериментов для определения следующих характеристик рассмотренных методов в последовательном и параллельном вариантах:

- Измерение времени работы параллельных алгоритмов T (сек, мсек).
- Ускорение работы программ S относительно последовательного алгоритма.

Тестирование проведено на одномерной задаче теплопроводности с использованием неявной схемы. Размерность задачи варьировалась от 1.0E+04 до 4.2E+07. Задача была рассчитана в различных сочетаниях количества thread - потоков.

Для проверки правильности алгоритма получены расчеты погрешности численного решения путем сравнения результатов численного решения с аналитическим решением.

На рис. 3.2 представлен вариант вывода результатов расчета погрешности численного решения относительно точного решения.

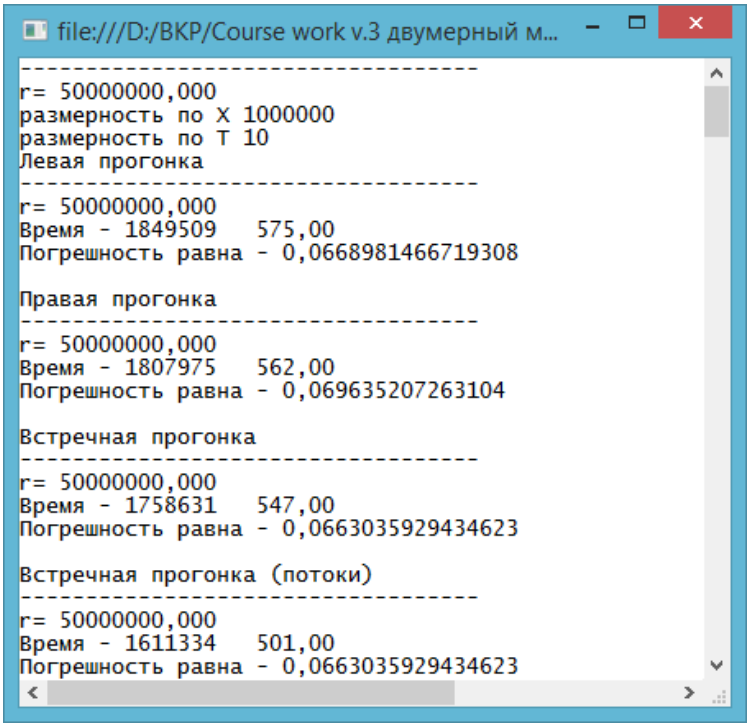

Рис.3.2 Результат расчета погрешности численного решения

Из рисунка видно, что погрешность расчета методом прогонки (левой, правой, встречной последовательной и встречной в двух потоках) практически одинаковы. Время расчета в случае параллельного варианта встречной прогонки уменьшается, полученная величина погрешности расчета для заданной сетки (τ= 0,1. h=10^{-6}) согласуется с оценкой $O(\tau^2 + h^2)$.

Для сравнения времени работы алгоритмов были проведены расчеты методом прогонки в последовательном варианте и методом встречной прогонки в параллельном варианте для двух потоков.

На рис.3.3 представлен участок кода приложения, реализующий алгоритм встречной прогонки в параллельном варианте.

```
public void CounterProgonkaThreading()
        {
            double S1 = 0;
            Console.WriteLine("Встречная прогонка (2 потока)");
            for (int k = 1; k <= 10; k++)
            {
                Restart();
                Stopwatch st = new Stopwatch();
                st.Start();
                Thread th1 = new Thread(DirectMoveAlphaBettaThread1);
                Thread th2 = new Thread(DirectMoveXiTettaThread2);
                th1.Start();
                th2.Start();
                th1.Join();
```

```
            th2.Join();
            Console.Write("++++++++++Встречная 2 Потока - {0}
              {1,8:f2}", st.ElapsedTicks, st.ElapsedMilliseconds);
            X[P] = (alpha[P + 1] * tetta[P + 1] + betta[P + 1]) /
               (1 - xi[P + 1] * alpha[P + 1]);
            th1 = new Thread(ReverseThread1);
            th2 = new Thread(ReverseThread2);
            th1.Start();               th2.Start();
            th1.Join();                th2.Join();

            Console.Write("++++++++++Встречная 2 Потока - {0}
                {1,8:f2}", st.ElapsedTicks, st.ElapsedMilliseconds);
            st.Stop();
            Console.WriteLine();
            //Console.WriteLine(OutputAnswer());
            S1 += st.ElapsedMilliseconds;
        }
       Console.WriteLine("Время среднее-  {0,8:f2}", S1 / 10);
    }
```

Рис.3.3 Метод, реализующий алгоритм встречной прогонки (параллельный вариант) на C#

В таблице 3.2 представлены результаты измерения времени работы (в сек) алгоритмов правой, левой, встречной прогонок (последовательный вариант) и алгоритма встречной прогонки (параллельный вариант $p = 2$) в зависимости от размера задачи N.

Таблица 3.2

Результаты измерения времени работы алгоритмов T(сек) в зависимости от размера задачи N

N-число элементов массива ($\times 10^4$)	T-сек			
	Левая прогонка	Правая прогонка	Встречная прогонка	Встречная прогонка (2 потока)
60	0,0163	0,0162	0,0155	0,0264
72	0,0186	0,016	0,0174	0,039
240	0,0611	0,0654	0,0645	0,0655
360	0,0971	0,0986	0,0992	0,0821
480	0,1297	0,1373	0,1354	0,1075
600	0,1658	0,1694	0,168	0,1275
720	0,2029	0,1993	0,2061	0,1462
840	0,2728	0,2677	0,2589	0,1608
960	0,3051	0,3057	0,2973	0,1948
1200	0,3706	0,3802	0,378	0,2354

2400	0,8401	0,8181	0,7576	0,4988
3600	1,2139	1,188	1,1827	0,7173
4800	1,6145	1,8779	1,5771	1,0224
6000	2,0221	2,0754	2,2451	1,6098
7200	2,5421	2,5493	2,5299	1,6168

Результаты демонстрируют, что время работы алгоритмов с помощью правой, левой и встречной прогонок в последовательном варианте практически одинаковы. Поэтому для сравнения с параллельным вариантом алгоритма прогонки можно выбрать любой, например, правый. Как видно из таблицы, при малой размерности массива N <2.4·10^6 параллельный алгоритм более затратен по времени. При больших N эффективность распараллеливания очевидна и стремится к теоретическому максимуму.

На рис 3.4 приведены результаты сравнения времени расчета алгоритмов, представленных в табл.3.2, в зависимости от размерности задачи N, демонстрирующие эффект, который дает использование встречной прогонки в параллельном варианте (для больших N >8.0·10^6).

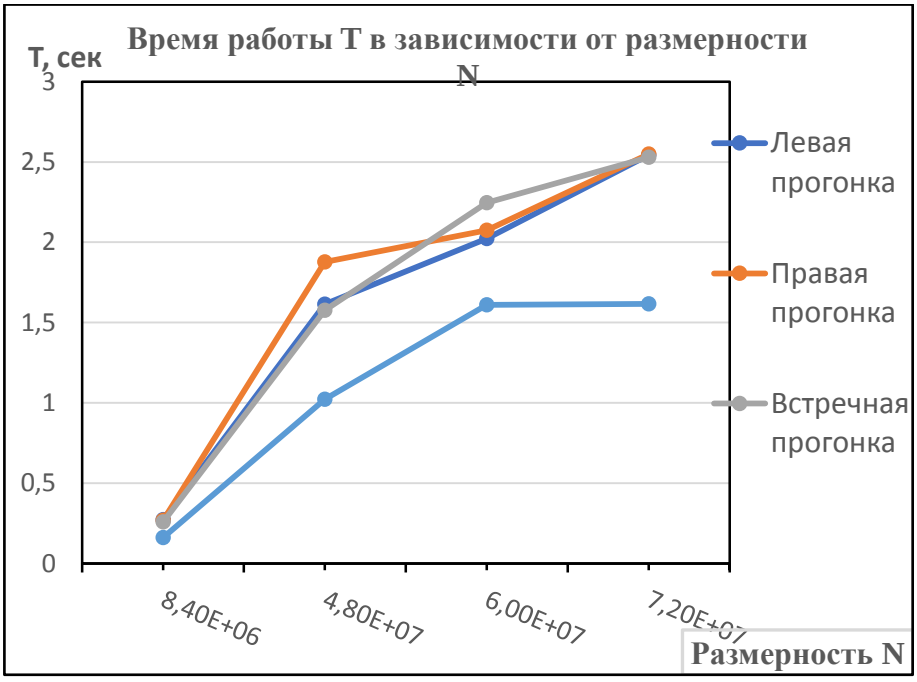

Рис.3.4 Время расчета последовательных и параллельного алгоритмов ($p = 2$)

При значениях N> 2·10^6 встречная прогонка в распараллеливании на двух потоках реализуется за меньшее время.

Результаты, отражающие зависимость ускорения S работы параллельного алгоритма метода встречной прогонки от размерности задачи N, приведены на рис.3.5.

Рис.3. 5 Зависимость ускорения S встречной прогонки от размера задачи

Величина ускорения S >1начинается со значений N> $2.4 \cdot 10^6$ и становится равной ~1,6 для N> $9.0 \cdot 10^6$. Дальнейшее увеличение размерности массива не приводит к увеличению величины ускорения расчетов.

В Таблице 3.3 приведены результаты вычислительных экспериментов (T - время работы алгоритма в сек), полученные при использовании параллельного метода прогонки (горизонтально-блочный алгоритм) с числом потоков $p > 2$.

Таблица 3.3

Время работы алгоритма горизонтально-блочной прогонки T(сек) в зависимости от размера задачи N и числа потоков p

N-число элементов массива ($\times 10^4$)	Число потоков p				
	1	4	6	8	10
720	331	364	417	510	594
1200	594	683	691	763	788
2400	907	1111	1145	1383	1286
3600	1843	2563	2296	1829	1717
4200	2912	3436	2947	2185	2000

При увеличении количества блоков (подзадач), на которые разбивается исходная система, растет время расчета для значений $N<3.6\cdot10^7$. При больших значениях размерности N наблюдается сокращение времени расчета.

На рис.3.6 представлено значение времени расчета (соответственно данным табл.3.3) методом горизонтально-блочной прогонки в зависимости от числа заданных потоков для различных значений размерности N.

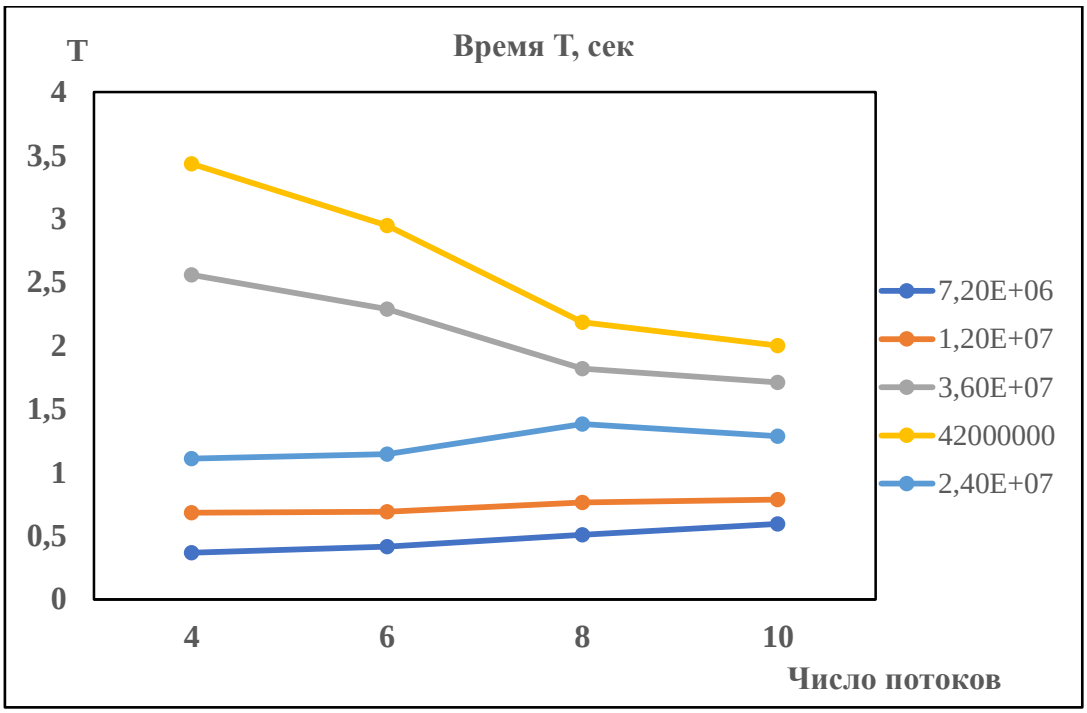

Рис. 3.6 Зависимость времени расчета от числа потоков для метода горизонтально-блочной прогонки

В таблице 3.4 и на рис.3.7, соответственно, представлены результаты расчета ускорения S (рис.3.7) методом горизонтально-блочной прогонки в зависимости от числа заданных потоков p для различных значений размерности N.

Таблица 3.4

Результаты расчета ускорения S в зависимости от размера задачи N и числа потоков p

	N-число элементов матрицы

P-число потоков	7,20E+06	1,20E+07	3,60E+07	4,20E+07
1	1	1	1	1
4	0,909341	0,869693	0,528528	0,721506
6	0,793765	0,859624	0,752402	0,988124
8	0,64902	0,778506	0,946703	1,332723
10	0,557239	0,753807	1,007602	1,456

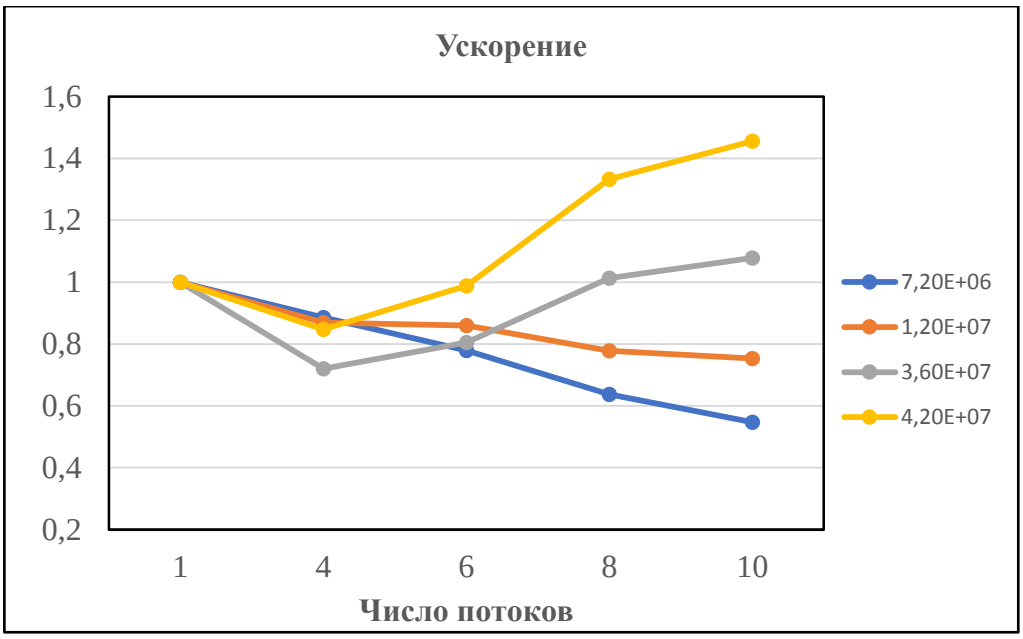

Рис. 3.7 Зависимость ускорения от числа потоков для метода горизонтально – блочной прогонки

Из рисунка можно заметить, что ускорение расчетов S>1 наблюдается при N>3.6·107 и количества потоков p>5.

3.2. Результаты расчета при использовании технологии OpenMP

Для организации параллельных алгоритмов встречной и горизонтально-блочной прогонки также была использована технология OpenMP-потоков.

Для реализации распределенных вычислений на центральном процессоре следует подключить библиотеку omp.h, добавить директиву #pragma omp parallel, тем самым обозначив параллельную область вычислений, а также добавить директиву #pragma omp for перед циклами вычислений.

#include <omp.h>

#pragma omp parallel for

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

{...}

На рис. 3.8 представлен участок кода, отвечающий за работу алгоритма горизонтально-блочной прогонки. Здесь *ThreadCount* – требуемое количество потоков при распараллеливании. Полный код данного метода и остальных методов на C++, реализующих различные алгоритмы прогонки, представлен в приложении 2.

```
double BlockSystemSolution::BlockProgonka(int j, double tau, double r,
double h)
{
	Restart(j, tau, r, h);
	auto start = steady_clock::now();
	int time = 0;
#pragma omp parallel for
	for (int i = 1; i <= ThreadCount; i++)
	{
		int b0 = (i - 1)*BlockN;
		int b1 = i * BlockN;
		DirectAlphaBettaGamma(b0, b1);
	}
	auto end = steady_clock::now();
	cout << "Время-1-" << duration_cast<std::chrono::milliseconds>(end -
start).count() << " ";
	time += duration_cast<std::chrono::milliseconds>(end - start).count();
	start = steady_clock::now();
	RightProgonkaMethodP();
	end = steady_clock::now();
	time += duration_cast<std::chrono::milliseconds>(end - start).count();
	cout << "Время-2-" << time << " ";
	start = steady_clock::now();
#pragma omp parallel for
	for (int i = 0; i <= ThreadCount - 1; i++)
	{
		X1[BlockN * (i + 1) - 1] = X[i];
	}
	end = steady_clock::now();
	time += duration_cast<std::chrono::milliseconds>(end - start).count();
	cout << "Время-3-" << time << " ";
	start = steady_clock::now();
//#pragma omp parallel for
	for (int i = 0; i <= BlockN - 2; i++)
	{
		X1[i] = malpha[i + 1] * X1[BlockN - 1] + lbetta[i + 1];
	}
	end = steady_clock::now();
	time += duration_cast<std::chrono::milliseconds>(end - start).count();
	cout << "Время-4-" << time << " ";
```

```
      start = steady_clock::now();
#pragma omp parallel for
      for (int k = 2; k <= ThreadCount; k++)
      {
            for (int i = BlockN * (k - 1); i <= k * BlockN - 2; i++)
            {
                  X1[i] = malpha[i + 1] * X1[k * BlockN - 1] + kgamma[i + 1] *
X1[(k - 1)
                              * BlockN - 1] + lbetta[i + 1];
            }
      }
      end = steady_clock::now();
      time += duration_cast<std::chrono::milliseconds>(end - start).count();
      cout << "Время-5-" << time << " ";
      for (int i = 0; i < N + 1; i++)
      {
            U[j][i] = X1[i];
      }
      return time;
```

Рис.3.8 Метод, реализующий алгоритм горизонтально-блочной прогонки на C++

Время расчета измерялось после каждого из 5-ти этапов алгоритма горизонтально-блочной прогонки с целью оценить временные затраты каждого этапа и выявить наиболее длительный.

На рис.3.9 приведен вариант вывода результатов работы программы.

Рис.3.9 Окно вывода результатов

Во всех испытаниях время счета в параллельном режиме более, чем в 1,5 раза меньше последовательного варианта. При этом, наиболее затратным по времени является первый этап метода блочной прогонки, в котором выполняется прямой и обратный ход алгоритма блочной прогонки для каждой полосы (потока).

В таблице 3.5 представлены результаты измерения времени работы (Т в мсек) алгоритмов правой прогонки (последовательный вариант) и алгоритма встречной прогонки (параллельный вариант $p = 2$) в зависимости от размера задачи N.

Таблица 3.5

Результаты измерения времени работы алгоритмов Т(сек) в зависимости от размера задачи N

N-число элементов массива ($\times 10^3$)	Т мсек	
	Правая прогонка	Встречная прогонка 2 потока
24	1,6	1,4
36	2,5	2,1
48	3,5	2,4
60	5,6	4,6
72	6,5	7,4
240	18,9	16,1
360	28,1	25,3
480	34,7	39,1
600	47,1	45,9
720	60,3	43,8
2400	154,2	137,2
3600	249,5	207,7
4800	302,2	277,5
6000	356,2	335,1
7200	415,2	404,5
8400	480,9	468,1
9600	575,3	533,9

Результаты демонстрируют, что время работы алгоритмов с помощью встречной прогонок в последовательном и параллельном вариантах практически

одинаковы до N <7.0·10^5. При больших N эффективность распараллеливания очевидна, соответствует данным, полученным с помощью технологии thread–потоков, и стремится к теоретическому максимуму.

В Таблице 3.6 приведены результаты вычислительных экспериментов (T - время работы алгоритма в мсек), полученные при использовании параллельного метода прогонки (горизонтально-блочный алгоритм) с числом потоков $p > 2$.

Таблица 3.6

Время работы алгоритма горизонтально-блочной прогонки Т(мсек) в зависимости от размера задачи N и числа потоков p

N-число элементо в массива ($\times 10^3$)	Параллельный алгоритм Т в зависимости от числа потоков						
	1	2	3	4	5	6	8
2400	154,2	137,2	82,4	84,9	86,2	90,2	89,7
3600	249,5	207,7	126,2	125,8	127,7	135	137,9
4800	302,2	277,5	178,3	171,2	183,1	160,2	179,0
6000	356,2	335,1	221,9	217,9	215,8	183,8	222,3
7200	415,2	404,5	257,1	252,6	259,7	259,6	264,5
8400	480,9	468,1	296,2	294,5	306,2	302,2	313,1
9600	575,3	533,9	341,8	343,8	349,1	359,6	353,9

При увеличении количества блоков (подзадач), на которые разбивается исходная система (число потоков p изменяется от 2 до 6), растет время расчета для значений N <4.8·10^6. При больших значениях размерности N наблюдается сокращение времени расчета. Увеличение числа потоков до p =10 приводит к увеличению времени расчета.

На рис.3.10 приведены результаты, соответствующие табл.3.6.

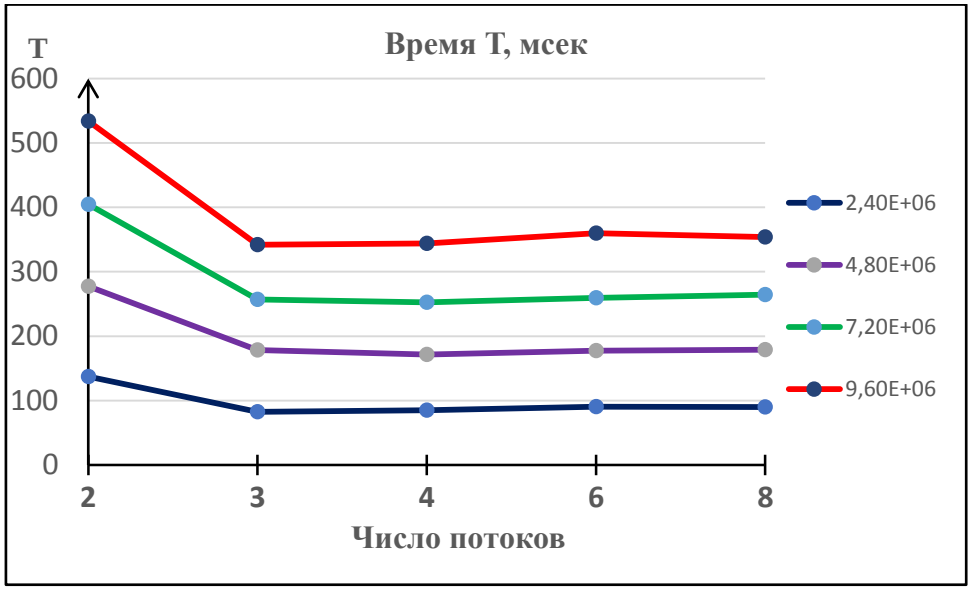

Рис. 3.10 Время работы алгоритма блочной прогонки в зависимости от числа потоков p и размерности задачи N

При изменении числа потоков p от 2 до 3 время расчета уменьшается более, чем в 2 раза. При дальнейшем увеличении числа потоков (от 3 до 8) расчетное время практически стабилизируется.

В Таблице 3.7 приведены результаты расчета величины ускорения S в соответствии с данными табл.3.6, полученные при использовании параллельного метода прогонки (горизонтально-блочный алгоритм) с числом потоков $p > 2$.

Таблица 3.7

Ускорение S работы алгоритма горизонтально-блочной прогонки в зависимости от размера задачи N и числа потоков p

N-число элементов массива ($\times 10^4$)	Параллельный алгоритм ускорение S в зависимости от числа потоков p						
	1(время)	2	3	4	5	6	8
240	154,2	1,12	1,87	1,82	1,79	1,71	1,72
360	249,5	1,20	1,98	1,98	1,90	1,84	1,81
480	302,2	1,09	1,67	1,77	1,70	1,89	1,69
600	356,2	1,06	1,60	1,63	1,65	1,94	1,60
720	415,2	1,03	1,61	1,64	1,61	1,61	1,57
840	480,9	1,03	1,62	1,63	1,57	1,59	1,54
960	575,3	1,08	1,68	1,67	1.60	1,62	1,62

Результаты Табл.3.7 демонстрируют ускорение расчета во всех испытаниях. При увеличении количества блоков (подзадач), на которые разбивается исходная система (число потоков p изменяется от 3 до 8) и значений размерности N $>6.0 \cdot 10^6$, значение величины ускорения выравнивается и устанавливается равным ~1,6.

На 3.11 представлены результаты расчета ускорения S в соответствии с данными табл.3.7 методом горизонтально-блочной прогонки в зависимости от числа заданных p потоков для различных значений размерности N с использованием технологии.

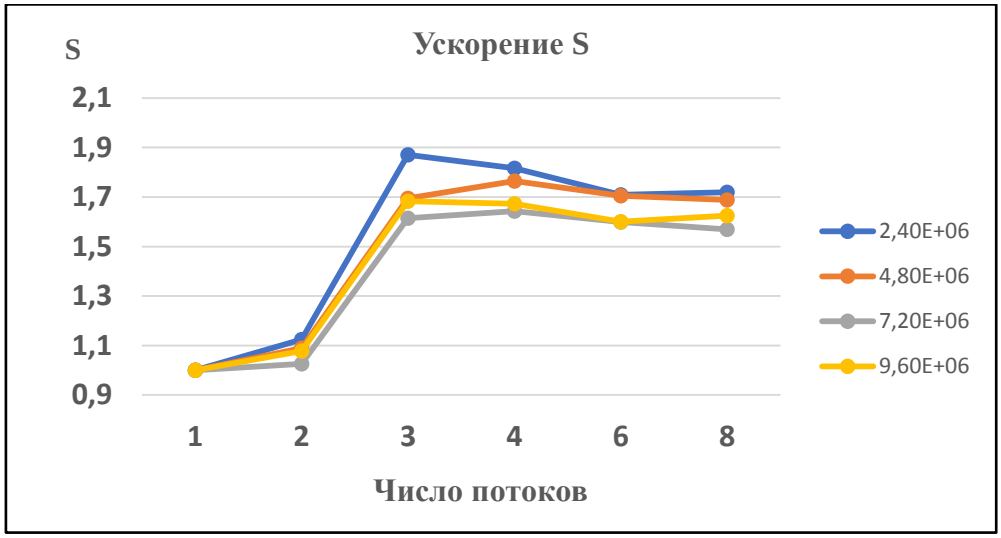

Рис.3.11 Ускорение S алгоритма блочной прогонки

Так, для случая N=2,40E+06 и использовании 4 – х потоков время расчёта T составляло 84,9 мсек, последовательный вариант потребовал 154,2 мсек, а достигнутое ускорение S=1,82, т.е., выигрыш во времени 82% по сравнению с применением классического последовательного подхода.

Видно, что при числе используемых потоков $p < 6$ ускорение несколько ниже, но в целом соответствует оценке (2.13), полученной с помощью подсчета числа операций, необходимых для работы метода. Падение можно объяснить тем, что расчеты выполнялись на компьютере с 4-мя ядрами, а обслуживание дополнительных потоков требует дополнительных временных затрат.

3.3. Оценка эффективность распараллеливания для разных ситуаций

В результате выполненного вычислительного эксперимента по исследованию эффективности параллельных алгоритмов для решения систем линейных уравнений с трехдиагональной матрицей коэффициентов при неизвестных были измерены время работы T (сек, мсек) и ускорение S работы параллельных алгоритмов относительно последовательных алгоритмов.

Результаты расчетов, полученные на основе многопоточного программирования с помощью Thread технологии и технологии OpenNP демонстрируют, что время работы параллельных алгоритмов (встречная прогонка и горизонтально-блочная прогонка) при малых размерностях массива N <2.4·10^6 более затратны по времени. При больших N эффективность распараллеливания очевидна и стремится к теоретическому максимуму. При этом расчеты, проведенные с использованием OpenNP–потоков затрачивают меньше времени, чем при использовании thread–потоков примерно в 5 раз.

На основе многопоточного программирования с помощью Thread технологии можно сделать следующие выводы:

- В случае встречной прогонки (p=2) при значениях N> 2·10^6 величина ускорения S >1 и становится равной ~1,6 для N> 9.0·10^6. Дальнейшее увеличение размерности массива не приводит к увеличению величины ускорения расчетов.
- При использовании горизонтально-блочного алгоритма с числом потоков $p > 2$ получено, что при увеличении количества блоков (подзадач), на которые разбивается исходная система, растет и время расчета для значений N <3.6·10^7. При больших значениях размерности N наблюдается сокращение времени расчета. Ускорение расчетов S >1 наблюдается при N>3.6·10^7 и количестве потоков p>5.

На основе многопоточного программирования с помощью OpenNP технологии можно сделать следующие выводы:

- Время работы алгоритмов с помощью встречной прогонок в последовательном и параллельном вариантах практически одинаковы до N

$<7.0\cdot10^5$. При больших N эффективность распараллеливания очевидна, соответствует данным, полученным с помощью технологии thread–потоков, и стремится к теоретическому максимуму.

- При использовании горизонтально-блочного алгоритма во всех испытаниях при изменении числа потоков p от 2 до 3 время расчета уменьшается более, чем в 2 раза. При дальнейшем увеличении числа потоков (от 3 до 8) расчетное время практически стабилизируется. Увеличение же числа потоков до p =10 приводит к увеличению времени расчета.
- Ускорение расчетов S >1 во всех испытаниях для N $>2.4\cdot10^6$. При этом (число потоков p изменяется от 3 до 8) для значений размерности N $>6.0\cdot10^6$ значение величины ускорения выравнивается и устанавливается равным ~1,6. Так для $N=2{,}4\cdot10^6$ и использовании 4 – х потоков время расчёта T составляло 84,9 мсек, последовательный вариант потребовал 154,2 мсек, а достигнутое ускорение S=1,82, т.е., выигрыш во времени 82% по сравнению с применением классического последовательного подхода.

Таким образом, использование OpenNP– технологии демонстрирует эффективность использования ресурсов $E_p = S/p$ равной 0, 48 для $N = 2.4\cdot10^6$ при p=4 и E_p =0,63 при p=3, а использование Thread -технологии показало значение E_p =0,25 при $N = 7.2\cdot10^6$ и p=4.

Полученные значения можно объяснить тем, что расчеты выполнялись на компьютере с 4-мя ядрами, а обслуживание дополнительных потоков требует дополнительных временных затрат.

ЗАКЛЮЧЕНИЕ

В результате выполнения выпускной квалификационной работы было проведено исследование применимости технологий параллельных вычислений для решения задач вычислительной математики с использованием многопроцессорных вычислительных систем с общей памятью.

В качестве объекта применения различных подходов к организации параллельных вычислений рассмотрена задача определения численного решения системы линейных уравнений с ленточной матрицей, возникающей, например, при решении краевой задачи для дифференциального уравнения в частных производных параболического типа, уравнения теплопроводности.

Для численного решения краевой задачи использована неявная конечно-разностная схема Кранка-Николсона., предполагающей применение метода прогонки для решения системы линейных алгебраических уравнений с трехдиагональной матрицей коэффициентов при неизвестных.

В качестве средства программирования для компьютеров с общей памятью использовались следующие технологии организации параллельных расчетов:

- технологии многопоточного программирования Threads, реализующей использование потоков с ручной настройкой работы каждого потока;
- технологии распараллеливания OpenMP, в рамках которой вычисления выполняются под управлением автоматического планировщика заданий, выбирающего оптимальный с его точки зрения режим параллельной обработки.

Разработаны две программы в среде программирования Visual Studio 2017 с использованием технологий параллельного программирования на основе стандартов OpenMP, и класса Thread для многопоточного программирования .NET.

Приложения реализованы в консольном виде с окном, отображающим результаты расчета.

В каждом приложении реализованы два подхода к распараллеливанию процедуры организации параллельных вычислений для случая трёхдиагональной матрицы:

- алгоритм метода встречной прогонки (организация вычислений на 2-х потоках);
- горизонтально-блочный параллельный алгоритм (организация вычислений на $p>2$ потоках).

В результате проведенного вычислительного эксперимента для оценки эффективности параллельных вариантов получены результаты сравнения времени расчета и ускорения в зависимости от размера задачи и количества используемых потоков.

Общее время вычислений для технологии Thread оказалась в 5 раз больше, чем на OpenMP – потоках.

При распараллеливании на Thread-потоках получено ускорение в 1.6 раза, на OpenMP-потоках – в 1.9 раза. Увеличение числа используемых потоков до величины большей, чем число ядер процессора,PP не приводит к ускорению расчетов.

Таким образом, показана практическая эффективность использования технологии распараллеливания при выполнении расчётов для задачи определения решения систем линейных уравнений с трехдиагональной матрицей.

Полученные результаты будут способствовать модернизация содержания базового учебного курса «Численные методы» для студентов -программистов с учетом применения обучающимися современных технологий параллельных вычислений.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Введение в параллельное программирование. [Электр. ресурс]. Режим доступа https://www.intuit.ru/studies/courses/4807/1055/lecture/16369 (дата обращения: 22.12.2018).
2. Старченко А. В., Берцун В. Н. Методы параллельных вычислений. [Электр. ресурс]. Режим доступа http://www.math.tsu.ru/sites/default/files/mmf2/e-resources/parallel_comp_meth.pdf (дата обращения: 22.12.2018).
3. Вержбицкий В.М. Численные методы (математический анализ и обыкновенные дифференциальные уравнения). – М.: Высшая школа, 2001.
4. Бахвалов Н.С., Жидков Н.П., Кобельков Г.М. Численные методы. – М.: Наука, 1987.
5. Самарский А.А. Введение численные методы. – СПб.: Лань, 2005.
6. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных машин. Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского, 2003.
7. Гергель В.П. Теория и практика параллельных вычислений. – М.: БИНОМ, 2007.
8. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. – М.: Изд-во МГУ, 2009.
9. Баркалов К.А Образовательный комплекс «Параллельные численные методы» Параллельные численные методы http://www.hpcc.unn.ru/?doc=491 (дата обращения: 20.12.2018).
10. Бурова И.Г., Демьянович Ю. К. Алгоритмы параллельных вычислений и программирование Курс лекций. - СПб.: Изд-во С. -Пб. ун-та, 2007.
11. Гергель В.П. Параллельное программирование с использованием OpenMP; ДМК Пресс - Новосибирск, 2007.
12. Metanit.com Сайт о программировании [Электр. ресурс]. – Режим доступа https://metanit.com/sharp/tutorial/12.2.php (дата обращения: 05.04.2019).
13. Федоров А. А., Быков А. Н. Метод двухуровневого распараллеливания прогонки для решения трехдиагональных линейных систем на гибридных

ЭВМ с многоядерными сопроцессорами //Вычислительные методы и программирование. 2016. Т. 17, стр. 234-244.

14. Рычков А.Д. Параллельные вычислительные технологии. Учебный курс. Кафедра вычислительных систем. Сибирский государственный университет телекоммуникаций и информатики [Электр. ресурс]. – Режим доступа https://ita.sibsutis.ru/node/78 (дата обращения: 05.04.2019).

15. Яненко Н.Н., Коновалов А.Н., Бугров А.Н., Шустов Г.В. Об организации параллельных вычислений и «распараллеливании» прогонки // Численные методы механики сплошной среды. 1978.Т.9. №7. С.139-146.

ПРИЛОЖЕНИЕ 1

РЕАЛИЗАЦИЯ АЛГОРИТМОВ БЛОЧНОЙ ПРОГОНКИ ПРИ ИСПОЛЬЗОВАНИИ КЛАССА THREAD

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace Course_work_v._3
{
```

1. Метод, реализующий блочную прогонку

```
        public void ProgonkaThreading19()
        {
            //DateTime S = DateTime.Now;
            Console.WriteLine("Блочная прогонка (P -потоков)");
            //double S = 0;
            double S1 = 0;
            for (int k = 1; k <= 10; k++)
            {
                Restart();
                // InputMatrix();
                //Console.WriteLine("число итераций {0}", k);
                Thread[] tha = new Thread[PP];
                Console.Write("----------Время - ");
                Stopwatch st = new Stopwatch();
                //DateTime t0 = DateTime.Now;
                st.Start();
                for (int i = 1; i <= PP; i++)
                {
                    int[] a = new int[2];
                    a[0] = (i - 1) * MM;
                    a[1] = i * MM;
                    tha[i - 1] = new Thread(DirectMoveAlphaBettaThread11);
                    tha[i - 1].Start(a);
                }
                for (int i = 1; i <= PP; i++)
                {
                    tha[i - 1].Join();
                }
                //st.Stop();
                Console.Write("{0,8:f2}", st.ElapsedMilliseconds);
               // st.Restart();
                //TimeSpan dt1 = DateTime.Now - t0;
                RightProgonkaMethodP();
               // st.Stop();
                Console.Write("{0,8:f2}", st.ElapsedMilliseconds);
               // st.Restart();
                //TimeSpan dt2 = DateTime.Now - t0;
                //Console.WriteLine(OutputAnswer1());
                for (int i = 0; i <= PP - 1; i++)
                {
                    X1[MM * (i + 1) - 1] = X[i];
                }
                //st.Stop();
                Console.Write("{0,8:f2}", st.ElapsedMilliseconds);
               // st.Restart();
                //TimeSpan dt3 = DateTime.Now - t0;
                for (int i = 0; i <= MM - 2; i++)
```

```
            {
                X1[i] = malpha[i + 1] * X1[MM - 1] + lbetta[i + 1];
            }
            //st.Stop();
            Console.Write("{0,8:f2}", st.ElapsedMilliseconds);
            //st.Restart();
            //TimeSpan dt4 = DateTime.Now - t0;
             for (int i = 2; i <= PP; i++)
             {
                tha[i - 1] = new Thread(DirectMoveAlphaBettaThread12);
                tha[i - 1].Start(i);
              }
              for (int i = 1; i <= PP; i++)
              {
                 tha[i - 1].Join();
              }
            //st.Stop();
            Console.Write("{0,8:f2}", st.ElapsedMilliseconds);
            //st.Restart();
            // st.Stop();
            //TimeSpan dt5 = DateTime.Now - t0;
            st.Stop();
            Console.WriteLine();
            S1 += st.ElapsedMilliseconds;
        }
        //Console.WriteLine(OutputAnswer2());
        Console.WriteLine("-----Время среднее-  {0,8:f2}", S1/10);
    }
```

2. Метод, реализующий прямой ход блочной прогонки

```
    void DirectMoveAlphaBettaThread11(object obj)
    {
        int[] b = (int[])obj;
        int nn = b[0];
        int mm = b[1];
        //Console.WriteLine("nn={0}, mm={1}", nn,mm);
        //int kk = Convert.ToInt32(t);
        // int pp = Convert.ToInt32(t1);
        alpha[nn + 1] = -B[nn] / C[nn];
        betta[nn + 1] = F[nn] / C[nn];
        gamma[nn + 1] = -A[nn] / C[nn];
        for (int i = nn + 1; i <= mm - 1; i++)
        {
            alpha[i + 1] = -B[i] / (C[i] + A[i] * alpha[i]);
            betta[i + 1] = (F[i] - A[i] * betta[i]) / (C[i] + A[i] * alpha[i]);
            gamma[i + 1] = -A[i] * gamma[i] / (C[i] + A[i] * alpha[i]);
        }
        malpha[mm - 1] = alpha[mm - 1];
        lbetta[mm - 1] = betta[mm - 1];
        kgamma[mm - 1] = gamma[mm - 1];
        for (int i = mm - 2; i >= nn + 1; i--)
        {
            malpha[i] = alpha[i] * malpha[i + 1];
            lbetta[i] = alpha[i] * lbetta[i + 1] + betta[i];
            kgamma[i] = alpha[i] * kgamma[i + 1] + gamma[i];
        }
    }
```

3. Метод, реализующий обратный ход блочной прогонки

```
    void DirectMoveAlphaBettaThread12(object obj)
    {
        int j = (int)obj;
```

```
            for (int i = MM * (j - 1); i <= j * MM - 2; i++)
            {
                X1[i] = malpha[i + 1] * X1[j * MM - 1] + kgamma[i + 1] * X1[(j - 1) *
                MM - 1] + lbetta[i + 1];
            }
        }
```

4. Метод, реализующий итоговую прогонку с остаточной матрицей

```
        void RightProgonkaMethodP()//(double[] lbetta, double[] malpha, double[]
kgamma)
        {
            double[] A1 = new double[PP + 1];
            double[] B1 = new double[PP + 1];
            double[] C1 = new double[PP + 1];
            double[] F1 = new double[PP + 1];
            for (int i = 1; i <= PP - 2; i++)
            {
                A1[i] = -gamma[(i + 1) * MM];
                F1[i] = alpha[MM * (i + 1)] * lbetta[MM * (i + 1) + 1] + betta[MM *
                (i + 1)];
                B1[i] = -alpha[MM * (i + 1)] * malpha[MM * (i + 1) + 1];
                C1[i] = 1 - alpha[MM * (i + 1)] * kgamma[MM * (i + 1) + 1];
            }
            B1[PP - 1] = 0;
            A1[PP - 1] = -gamma[PP * MM];
            C1[PP - 1] = 1;
            F1[PP - 1] = betta[PP * MM];
            A1[0] = 0;
            F1[0] = alpha[MM] * lbetta[MM + 1] + betta[MM];
            B1[0] = -alpha[MM] * malpha[MM + 1];
            C1[0] = 1 - alpha[MM] * kgamma[MM + 1];
            alpha[1] = -B1[0] / C1[0];
            betta[1] = F1[0] / C1[0];
            for (int i = 1; i <= PP - 2; i++)
            {
                alpha[i + 1] = -B1[i] / (A1[i] * alpha[i] + C1[i]);
                betta[i + 1] = (F1[i] - A1[i] * betta[i]) / (A1[i] * alpha[i] + C1[i]);
            }
            X[PP - 1] = (F1[PP - 1] - A1[PP - 1] * betta[PP - 1]) / (A1[PP - 1] *
            alpha[PP - 1] + C1[PP - 1]);
            for (int i = PP - 2; i >= 0; i--)
            {
                X[i] = alpha[i + 1] * X[i + 1] + betta[i + 1];
            }
        }
    }
}
```

РЕАЛИЗАЦИЯ РАЗЛИЧНЫХ АЛГОРИТМОВ ПРОГОНКИ ПРИ ИСПОЛЬЗОВАНИИ OPEN MP–ПОТОКОВ

```
#include "stdafx.h"
#include "SystemSolution.h"
#include <omp.h>
#include <chrono>
#include <cmath>
#include <iostream>
using namespace std;
using namespace std::chrono;

SystemSolution::SystemSolution(double _a, double _b, int _M, int _N, double _T, double
_coef)
{
	a = _a;
	b = _b;
	M = _M;
	N = _N;
	T = _T;
	coef = _coef;
	P = N / 2;
	A = new double[N + 1];
	B = new double[N + 1];
	C = new double[N + 1];
	F = new double[N + 1];
	alpha = new double[N + 1];
	betta = new double[N + 1];
	xi = new double[N + 1];
	tetta = new double[N + 1];

}
SystemSolution::~SystemSolution()
{
	delete[]A,B,C, alpha, betta, xi, tetta;
}

ouble SystemSolution::G1(double t)
{
	return 0;
}
double SystemSolution::G2(double t)
{
	return 0;
}
double SystemSolution::Phi(double x)
{
	return 0;
}
double SystemSolution::Function(double x, double t)
{
	return 1 - x * x;
}
void SystemSolution::DefaultValues(double h)
{
	for (int j = 0; j < M + 1; j++)
	{
		U[j] = new double[N + 1];
		if (j == 0)
		{
			for (int i = 0; i < N + 1; i++)
				U[j][i] = Phi(a + i * h);
```

```
                U[0][0] = G1(0);
                U[0][N] = G2(0);
            }
        }
}

void SystemSolution::Restart(int j, double tau, double r, double h)
{
        C[N] = -(1 + 2 * r);
        C[0] = -(1 + 2 * r);
        A[0] = 0;
        A[N] = 0;
        B[0] = 0;
        B[N] = 0;
        F[0] = -U[j][0] - tau * Function(a, j * tau);
        F[N] = -U[j][N] - tau * Function(b, j * tau);
        for (int i = 1; i < N; i++)
        {
            A[i] = r;
            B[i] = r;
            C[i] = -(1 + 2 * r);
            F[i] = -U[j][i] - tau * Function(a + h * i, j * tau);
        }
}
void SystemSolution::DirectMoveAlphaBettaThread1()
{
        alpha[1] = -B[0] / C[0];
        betta[1] = F[0] / C[0];
        for (int i = 1; i <= P; i++)
        {
            alpha[i + 1] = -B[i] / (C[i] + A[i] * alpha[i]);
            betta[i + 1] = (F[i] - A[i] * betta[i]) / (C[i] + A[i] * alpha[i]);
        }

}
void SystemSolution::DirectMoveXiTettaThread2()
{
        xi[N] = -A[N] / C[N];
        tetta[N] = F[N] / C[N];
        for (int i = N - 1; i >= P; i--)
        {
            int ind = i + 1;
            xi[i] = -A[i] / (C[i] + B[i] * xi[ind]);
            tetta[i] = (F[i] - B[i] * tetta[i + 1]) / (C[i] + B[i] * xi[i + 1]);
        }
}
void SystemSolution::ReverseThread1(int j)
{
        for (int i = P - 1; i >= 0; i--)
        {
            U[j][i] = alpha[i + 1] * U[j][i + 1] + betta[i + 1];
        }
}
void SystemSolution::ReverseThread2(int j)
{
        for (int i = P; i <= N - 1; i++)
        {
            U[j][i + 1] = xi[i + 1] * U[j][i] + tetta[i + 1];
        }
}
void SystemSolution::RightProgonka(int j)
{
        alpha[1] = -B[0] / C[0];
        betta[1] = F[0] / C[0];
        for (int i = 1; i <= N; i++)
        {
            alpha[i + 1] = -B[i] / (C[i] + A[i] * alpha[i]);
```

```
			betta[i + 1] = (F[i] - A[i] * betta[i]) / (C[i] + A[i] * alpha[i]);
		}
		U[j][N] = (F[N] - A[N] * betta[N]) / (A[N] * alpha[N] + C[N]);
		for (int i = N - 1; i >= 0; i--)
		{
			U[j][i] = alpha[i + 1] * U[j][i + 1] + betta[i + 1];
		}
}
void SystemSolution::CounterProgonkaMethod(int j, bool parallel_flag)
{
#pragma omp parallel if(parallel_flag)
		{
#pragma omp sections
			{
#pragma omp section
				{
					DirectMoveAlphaBettaThread1();
				}
#pragma omp section
				{
					DirectMoveXiTettaThread2();
				}
			}
		}
		U[j][P] = (alpha[P + 1] * tetta[P + 1] + betta[P + 1]) / (1 - xi[P + 1] * alpha[P +
1]);
#pragma omp parallel if(parallel_flag)
		{
#pragma omp sections
			{
#pragma omp section
				{
					ReverseThread1(j);

				}
#pragma omp section
				{
					ReverseThread2(j);
				}
			}
		}
}
void SystemSolution::ImplictSchema(double h, double tau, double r, bool parallel_flag)
{
		for (int j = 0; j < M; j++)
		{
			Restart(j, tau, r, h);
			CounterProgonkaMethod(j + 1, parallel_flag);
			U[j + 1][0] = G1((j + 1) * tau);
			U[j + 1][N] = G2((j + 1) * tau);
		}
}
void SystemSolution::ImplictSchema1(double h, double tau, double r)
{
		for (int j = 0; j < M; j++)
		{
			Restart(j, tau, r, h);
			RightProgonka(j + 1);
			U[j + 1][0] = G1((j + 1) * tau);
			U[j + 1][N] = G2((j + 1) * tau);
		}
}
double SystemSolution::SystemSolutionTime(bool parallel_flag)
{
		int iteration = 10;
		double time = 0;
		double tau = T / M;
```

```
	double h = (double)(b - a) / N;
	//double r = coef * tau / (h*h);
	double r = coef;
	for (int t = 0; t < iteration; t++)
	{
		U = new double *[M + 1];
		DefaultValues(h);
		auto start = steady_clock::now();
		ImplictSchema(h, tau, r, parallel_flag);
		auto end = steady_clock::now();
		cout << "Время*****" << duration_cast<std::chrono::milliseconds>(end -
start).count() << "\n";
		time += duration_cast<std::chrono::milliseconds>(end - start).count();
		/*cout << OutputAnswer(U);*/
		for (int i = M; i >= 0; i--)
		{
			delete U[i];
		}
		delete[]U;
	}
	return time / iteration;
}
double SystemSolution::RightProgonkaTime()
{
	int iteration = 10;
	double time = 0;
	double tau = T / M;
	double h = (double)(b - a) / N;
	//double r = coef * tau / (h*h);
	double r = coef;
	for (int t = 0; t < iteration; t++)
	{
		U = new double *[M + 1];
		DefaultValues(h);
		auto start = steady_clock::now();
		ImplictSchema1(h, tau, r);
		auto end = steady_clock::now();
		cout << "Время*****" << duration_cast<std::chrono::milliseconds>(end -
start).count() << "\n";
		time += duration_cast<std::chrono::milliseconds>(end - start).count();
		/*cout << OutputAnswer(U);*/
		for (int i = M; i >= 0; i--)
		{
			delete U[i];
		}
		delete[]U;
	}
	return time / iteration;
}
string SystemSolution::OutputAnswer(double **U)
{
	string s = "Размерность по пространсву N: " + to_string(N) + "\n" + "Количесвто
веременных отрезков M: " + to_string(M) + "\n" + "Ответ:\n";
	for (int j = 0; j < M + 1; j++)
	{
		for (int i = 0; i < N + 1; i++)
			s += to_string(U[j][i]) + "\n";
		s += "\n\n";
	}
	return s;
}
```

```
#include "stdafx.h"
#include "BlockSystemSolution.h"
#include <chrono>
#include <iostream>

using namespace std;
using namespace std::chrono;
BlockSystemSolution::BlockSystemSolution(double _a, double _b, int _M, int _N, int
_ThreadCount, double _T, double _coef)
{
	a = _a;
	b = _b;
	M = _M;
	N = _N;
	Time = _T;
	coef = _coef;

	ThreadCount = _ThreadCount;
	BlockN = (N + 1) / ThreadCount;

	A = new double[N + 1];
	B = new double[N + 1];
	C = new double[N + 1];
	F = new double[N + 1];

	X = new double[N + 1];
	X1 = new double[N + 1];

	alpha = new double[N + 1];
	betta = new double[N + 1];
	gamma = new double[N + 1];
	malpha = new double[N + 2];
	lbetta = new double[N + 2];
	kgamma = new double[N + 2];
	Atmp = new double[ThreadCount + 1];
	Btmp = new double[ThreadCount + 1];
	Ctmp = new double[ThreadCount + 1];
	Ftmp = new double[ThreadCount + 1];
}
BlockSystemSolution::~BlockSystemSolution()
{
	delete[]A, B, C, alpha, betta, malpha, lbetta, kgamma, X, X1, Atmp, Btmp, Ctmp, Ftmp;
}

double BlockSystemSolution::G1(double t)
{
	return 0;
}
double BlockSystemSolution::G2(double t)
{
	return 0;
}
double BlockSystemSolution::Phi(double x)
{
	return 0;
}
double BlockSystemSolution::Function(double x, double t)
{
	return 1 - x * x;
}

void BlockSystemSolution::DefaultValues(double h)
{
	for (int j = 0; j < M + 1; j++)
	{
		U[j] = new double[N + 1];
		if (j == 0)
```

```
		{
			for (int i = 0; i < N + 1; i++)
				U[j][i] = Phi(a + i * h);
			U[0][0] = G1(0);
			U[0][N] = G2(0);
		}
	}
}
void BlockSystemSolution::Restart(int j, double tau, double r, double h)
{
	C[N] = -(1 + 2 * r);
	C[0] = -(1 + 2 * r);
	A[0] = 0;
	A[N] = 0;
	B[0] = 0;
	B[N] = 0;
	F[0] = -U[j][0] - tau * Function(a, j * tau);
	F[N] = -U[j][N] - tau * Function(b, j * tau);
	for (int i = 1; i < N; i++)
	{
		A[i] = r;
		B[i] = r;
		C[i] = -(1 + 2 * r);
		F[i] = -U[j][i] - tau * Function(a + h * i, j * tau);
	}
}
void BlockSystemSolution::DirectAlphaBettaGamma(int b0, int b1)
{
	int nn = b0;
	int mm = b1;
	alpha[nn + 1] = -B[nn] / C[nn];
	betta[nn + 1] = F[nn] / C[nn];
	gamma[nn + 1] = -A[nn] / C[nn];
	for (int i = nn + 1; i <= mm - 1; i++)
	{
		alpha[i + 1] = -B[i] / (C[i] + A[i] * alpha[i]);
		betta[i + 1] = (F[i] - A[i] * betta[i]) / (C[i] + A[i] * alpha[i]);
		gamma[i + 1] = -A[i] * gamma[i] / (C[i] + A[i] * alpha[i]);
	}
	malpha[mm - 1] = alpha[mm - 1];
	lbetta[mm - 1] = betta[mm - 1];
	kgamma[mm - 1] = gamma[mm - 1];
	for (int i = mm - 2; i >= nn + 1; i--)
	{
		malpha[i] = alpha[i] * malpha[i + 1];
		lbetta[i] = alpha[i] * lbetta[i + 1] + betta[i];
		kgamma[i] = alpha[i] * kgamma[i + 1] + gamma[i];
	}
}
void BlockSystemSolution::RightProgonkaMethodP()
{
	for (int i = 1; i <= ThreadCount - 2; i++)
	{
		Atmp[i] = -gamma[(i + 1) * BlockN];
		Ftmp[i] = alpha[BlockN * (i + 1)] * lbetta[BlockN * (i + 1) + 1] +
betta[BlockN * (i + 1)];
		Btmp[i] = -alpha[BlockN * (i + 1)] * malpha[BlockN * (i + 1) + 1];
		Ctmp[i] = 1 - alpha[BlockN * (i + 1)] * kgamma[BlockN * (i + 1) + 1];
	}
	Btmp[ThreadCount - 1] = 0;
	Atmp[ThreadCount - 1] = -gamma[ThreadCount * BlockN];
	Ctmp[ThreadCount - 1] = 1;
	Ftmp[ThreadCount - 1] = betta[ThreadCount * BlockN];
	Atmp[0] = 0;
	Ftmp[0] = alpha[BlockN] * lbetta[BlockN + 1] + betta[BlockN];
	Btmp[0] = -alpha[BlockN] * malpha[BlockN + 1];
	Ctmp[0] = 1 - alpha[BlockN] * kgamma[BlockN + 1];
```

```
	alpha[1] = -Btmp[0] / Ctmp[0];
	betta[1] = Ftmp[0] / Ctmp[0];
	for (int i = 1; i <= ThreadCount - 2; i++)
	{
		alpha[i + 1] = -Btmp[i] / (Atmp[i] * alpha[i] + Ctmp[i]);
		betta[i + 1] = (Ftmp[i] - Atmp[i] * betta[i]) / (Atmp[i] * alpha[i] +
Ctmp[i]);

	}
	X[ThreadCount - 1] = (Ftmp[ThreadCount - 1] - Atmp[ThreadCount - 1] *
betta[ThreadCount - 1]) / (Atmp[ThreadCount - 1] * alpha[ThreadCount - 1] + Ctmp[ThreadCount
- 1]);
	for (int i = ThreadCount - 2; i >= 0; i--)
	{
		X[i] = alpha[i + 1] * X[i + 1] + betta[i + 1];
	}
}
double BlockSystemSolution::BlockProgonka(int j, double tau, double r, double h)
{
	//	auto start1 = steady_clock::now();
	Restart(j, tau, r, h);
	//	cout << "Время (Restart)/*/*/*/*/*" <<
duration_cast<std::chrono::milliseconds>(steady_clock::now() - start1).count() << " ";
	auto start = steady_clock::now();
	int time = 0;
#pragma omp parallel for
	for (int i = 1; i <= ThreadCount; i++)
	{
		int b0 = (i - 1)*BlockN;
		int b1 = i * BlockN;
		DirectAlphaBettaGamma(b0, b1);
	}
	auto end = steady_clock::now();
	cout << "Время-1-" << duration_cast<std::chrono::milliseconds>(end - start).count()
<< " ";
	time += duration_cast<std::chrono::milliseconds>(end - start).count();
	start = steady_clock::now();
	RightProgonkaMethodP();
	end = steady_clock::now();
	time += duration_cast<std::chrono::milliseconds>(end - start).count();
	cout << "Время-2-" << time << " ";
	start = steady_clock::now();
#pragma omp parallel for
	for (int i = 0; i <= ThreadCount - 1; i++)
	{
		X1[BlockN * (i + 1) - 1] = X[i];
	}
	end = steady_clock::now();
	time += duration_cast<std::chrono::milliseconds>(end - start).count();
	cout << "Время-3-" << time << " ";
	start = steady_clock::now();
//#pragma omp parallel for
	for (int i = 0; i <= BlockN - 2; i++)
	{
		X1[i] = malpha[i + 1] * X1[BlockN - 1] + lbetta[i + 1];
	}
	end = steady_clock::now();
	time += duration_cast<std::chrono::milliseconds>(end - start).count();
	cout << "Время-4-" << time << " ";
	start = steady_clock::now();
#pragma omp parallel for
	for (int k = 2; k <= ThreadCount; k++)
	{
		for (int i = BlockN * (k - 1); i <= k * BlockN - 2; i++)
		{
			X1[i] = malpha[i + 1] * X1[k * BlockN - 1] + kgamma[i + 1] * X1[(k - 1)
* BlockN - 1] + lbetta[i + 1];
```

```
			}
		}
		end = steady_clock::now();
		time += duration_cast<std::chrono::milliseconds>(end - start).count();
		cout << "Время-5-" << time << " ";
		for (int i = 0; i < N + 1; i++)
		{
			U[j][i] = X1[i];
		}
		return time;
}
double BlockSystemSolution::ImplictSchema(double h, double tau, double r)
{
		double time = 0;
		for (int j = 0; j < M; j++)
		{
			time += BlockProgonka(j + 1, tau, r, h);
			U[j + 1][0] = G1((j + 1) * tau);
			U[j + 1][N] = G2((j + 1) * tau);
		}
		return time;
}
double BlockSystemSolution::BlockSystemSolutionTime()
{
		omp_set_num_threads(ThreadCount);
		int iteration = 10;
		double time = 0;
		double tau = Time / M;
		double h = (double)(b - a) / N;
		//double r = coef * tau / (h*h);
		double r = coef;
		for (int t = 0; t < iteration; t++)
		{
			U = new double *[M + 1];
			DefaultValues(h);
			//auto start = steady_clock::now();
			//ImplictSchema(h, tau, r);
			//auto end = steady_clock::now();
			//cout << "Время------" << duration_cast<std::chrono::milliseconds>(end -
start).count() << "\n";
			//time += duration_cast<std::chrono::milliseconds>(end - start).count();
			double iterationTime = ImplictSchema(h, tau, r);
			time += iterationTime;
			cout << "Время------" << iterationTime << "\n";
			/*cout<<OutputAnswer(U);*/
			for (int i = M; i >= 0; i--)
			{
				delete U[i];
			}
			delete[]U;
		}
		return time / iteration;
}
string BlockSystemSolution::OutputAnswer(double **U)
{
		string s = "Размерность по пространсву N: " + to_string(N) + "\n" + "Количесвто
веременных отрезков M: " + to_string(M) + "\n" + "Количество потоков: " +
to_string(ThreadCount) + "\n" + "Ответ:\n";
		for (int j = 0; j < M + 1; j++)
		{
			for (int i = 0; i < N + 1; i++)
				s += to_string(U[j][i]) + "\n";
			s += "\n\n";
		}
		return s;
}
```