МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение
высшего образования «Самарский национальный исследовательский
университет имени академика С.П. Королева (Самарский университет)»

Институт информатики, математики и электроники

Факультет информатики

Кафедра программных систем

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту по дисциплине «Параллельное программирование»

по теме «Решение СЛАУ методом Крамера»

Студент группы 6413-020302D В.Д. Гижевская

Студент группы 6414-020302D М.С. Сусликова

Руководитель В.В. Жидченко

Самара 2021

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Потребность решения сложных прикладных задач с большим объёмом вычислений и принципиальная ограниченность максимального быстродействия "классических" - по схеме фон Неймана - ЭВМ привели к появлению многопроцессорных вычислительных систем (МВС). Особую значимость параллельные вычисления приобрели с переходом компьютерной индустрии на массовый выпуск многоядерных процессоров.

Суперкомпьютерные технологии и высокопроизводительные вычисления с использованием параллельных вычислительных систем становятся важным фактором научно-технического прогресса; их применение принимает всеобщий характер [1].

К решению систем линейных алгебраических уравнений сводятся многочисленные практические задачи (по некоторым оценкам более 75% всех задач). Можно с полным основанием утверждать, что решение линейных систем является одной из самых распространённых и важных задач вычислительной математики. Конечно, существует много методов и современных пакетов прикладных программ для решения СЛАУ, но для того, чтобы их успешно использовать, необходимо разбираться в основах построения методов и алгоритмов, иметь представления о недостатках и преимуществах используемых методов [2].

Методы решения большинства задач численных методов (в том числе задач решения линейных алгебраических систем) можно разделить на два типа: прямые и итерационные. Метод решения задачи называется прямым, если он даёт её точное решение за конечное число действий. Метод решения задачи называется итерационным, если он даёт точное решение как предел последовательности приближений , вычисляемых по единообразной схеме, то есть $x = \lim_{k\to\infty} x^k$, где *x* - точное решение задачи, *x(k)* – *k*-тое приближение (итерация) к решению. К прямым методам решения СЛАУ относятся

известный из алгебры метод Крамера, метод Гаусса и его модификации и другие [3].

1 Постановка задачи

Цель работы: Закрепление теоретического материала курса и практических навыков параллельного программирования на примере решения одной из задач вычислительной математики.

Задание:

1. Разработать последовательный алгоритм решения задачи.
2. Реализовать разработанный алгоритм в виде программы на языке C++, провести вычислительные эксперименты для различных параметров задачи в соответствии с вариантом.
3. Разработать параллельный алгоритм решения задачи, для чего:
   - Проанализировать возможные способы декомпозиции задачи, выполнить декомпозицию.
   - Определить информационные зависимости между выделенными фрагментами, принять решение о необходимости синхронизации.
   - Выполнить масштабирование фрагментов задачи с учётом характеристик целевой вычислительной системы.
4. Реализовать параллельный алгоритм в виде программы на языке C++ с использованием технологии OpenMP.
5. Реализовать параллельный алгоритм в виде программы на языке C++ с использованием библиотеки MPI.
6. Провести вычислительные эксперименты для различных параметров задачи и различного количества параллельных процессов в соответствии с вариантом. Параметры задачи должны соответствовать параметрам, используемым в п.3. Количество процессов в программах на OpenMP и MPI должно совпадать.
7. Провести сравнение продолжительности работы различных вариантов программы (последовательной, OpenMP, MPI), вычислить

достигаемое ускорение и эффективность, результаты измерений и вычислений свести в таблицу. Построить графики по данным из таблицы.

8. Объяснить полученные закономерности.

2 Описание вычислительной системы

Вычисления будут проводиться на ПК HP Pavilion 15-cs0015ur с 4-ядерным процессором Core i5 8250U с поддержкой Hyper-Threading Technology.

3 Ход работы

3.1 Описание метода Крамера

Метод Крамера или так называемое правило Крамера – это способ поиска неизвестных величин из систем уравнений. Его можно использовать только если число искомых значений эквивалентно количеству алгебраических уравнений в системе, то есть образуемая из системы основная матрица должна быть квадратной и не содержать нулевых строчек, а также её детерминант не должен являться нулевым [4].

Теорема Крамера:

Если главный определитель D основной матрицы, составленной на основе коэффициентов уравнений, не равен нулю, то система уравнений совместна, причём решение у неё существует единственное. Решение такой системы вычисляется через так называемые формулы Крамера для решения систем линейных уравнений: $x_i = \frac{D_i}{D}$.

Суть метода Крамера в следующем [4]:

1. Чтобы найти решение системы методом Крамера, первым делом вычисляем главный определитель матрицы D. Когда вычисленный детерминант основной матрицы при подсчёте методом Крамера оказался равен нулю, то система не имеет ни одного решения или имеет нескончаемое количество решений. В этом случае для нахождения общего или какого-либо базисного ответа для системы рекомендуется применить метод Гаусса.

2. Затем нужно заменить крайний столбец главной матрицы на столбец свободных членов и высчитать определитель D_1.

3. Повторить то же самое для всех столбцов, получив определители от D_1 до D_n, где n - номер крайнего справа столбца.

4. После того как найдены все детерминанты $D_1 \dots D_n$, можно высчитать неизвестные переменные по формуле $x_i = \frac{D_i}{D}$.

Для вычисления определителя матрицы следует использовать метод, известный как метод Гаусса, также иногда этот метод называют понижением порядка определителя. В этом случае матрица преобразуется и приводится к треугольному виду, а затем перемножаются все числа, стоящие на главной диагонали. Следует помнить, что при таком поиске определителя нельзя домножать или делить строчки или столбцы на числа без вынесения их как множителя или делителя. В случае поиска определителя возможно только вычитать и складывать строки и столбы между собой, предварительно помножив вычитаемую строку на ненулевой множитель. Также при каждой перестановке строчек или столбцов матрицы местами следует помнить о необходимости смены конечного знака у матрицы [4].

3.2 Последовательный алгоритм

Определим массив векторов coefficientsSLAU и вектор constantTermsSLAU размера n и заполним их значениями от -30 до 30 и от -100 до 100 соответственно, генерируемыми в методе random. Данный метод принимает границы a и b промежутка значений для генерации и возвращает случайное целочисленное число из указанного промежутка, используя генератор псевдослучайных чисел rand(). Сочетание srand(time(NULL)) устанавливает в качестве базы текущее время, чтобы при разных запусках генератора была всякий раз разная база и, соответственно, разный ряд получаемых значений.

Далее вызываем метод metodCramer. Данный метод по ссылке принимает массив векторов с коэффициентами системы coefficientsSLAU вектор значений свободных членов constantTermsSLAU, количество уравнений в системе n; возвращает вектор решений системы solution.

Зададим неопределенный вектор solution размером n и два массива векторов basicMatrix и tempMatrix со скопированными значения вектора coefficientsSLAU. Первый массив будем использовать для вычисления определителя основной матрицы. Запускаем цикл по n, в котором последовательно меняем соответствующие столбцы матрицы tempMatrix на столбец свободных членов constantTermsSLAU, вычисляем определитель получившейся матрицы векторов. Если он равен нулю, то немедленное завершаем работу программы. В противном случае вычисляем соответствующее решение системы посредством деления значения получившегося определителя на значение определителя основной матрицы. Заносим данное решение в вектор solution .

Метод findDetByGauss используется для вычисления определителя системы методом Гаусса. Он принимает по ссылке массив векторов системы matrix и количество уравнений в системе n, а возвращает переменную det типа double, хранящую значение определителя. После преобразования исходной матрицы к треугольному виду путем перестановок все числа, стоящие на главной диагонали, последовательно перемножаются и заносят в переменную det. Ее знак остается прежним, если было проведено четное число перестановок строк, и изменяется на противоположный при нечетном количестве перестановок.

Метод toTriangularMatrix приводит матрицу к треугольному виду для вычисления определителя методом Гаусса. Он принимает по ссылке массив векторов системы matrix и количество уравнений в системе n, а возвращает количество перестановок swapCount. В данном методе с помощью цикла элементарных преобразований (перестановка строки с максимальным элементом с текущей, домножение строки на значение и сложение их меж собой) все элементы ниже главной диагонали обнуляются.

Метод findMaxInColumn находит максимальный по модулю элемент в столбце и возвращает его позицию maxPos. Он принимает по ссылке массив векторов системы matrix, номер столбца и количество уравнений в системе n.

Метод swapCoefficientsWithConsts меняет столбец матрицы tempMatrix на столбец свободных членов constantTermsSLAU. Он имеет тип возвращаемого значения void и принимает по ссылке массив векторов с коэффициентами системы matrix вектор значений свободных членов constantTermsSLAU, номер столбца в массиве для замены i, количество уравнений в системе n.

Реализация данного алгоритма представлена в приложении А.

3.3 Параллельный алгоритм

В нашей программе можно выделить 2 блока, на которых можно осуществить распараллеливание: метод триангуляции и метод Крамера.

Метод триангуляции с помощью цикла элементарных преобразований (перестановка строки с максимальным элементом с текущей, домножение строки на значение и сложение их меж собой) все элементы ниже главной диагонали обнуляются. Данный этап можно распараллелить таким образом, чтобы обработка i-той строки распределялась между потоками. Это позволит уменьшить время обработки матрицы.

В методе Крамера мы вычисляем n определителей. Эти вычисления также можно распараллелить. Сам метод требует вычисления n определителей, соответственно, эти вычисления также будем производить параллельно. Чтобы не возникло никаких ошибок при распределении вычислений между потоками, каждый поток должен иметь собственную копию исходной матрицы.

3.3.1 Параллельный алгоритм с использованием OpenMP

OpenMP – механизм написания параллельных программ для систем с общей памятью. Он состоит из набора директив компилятора и библиотечных функций и позволяет достаточно легко создавать многопоточные приложения на C/C++, Fortran [5].

Параллельные регионы являются основным понятием в OpenMP. Именно там, где задан этот регион программа исполняется параллельно. Как только компилятор встречает прагму omp parallel, он вставляет инструкции для создания параллельных потоков.

Количество порождаемых потоков для параллельных областей контролируется через переменную окружения OMP_NUM_THREADS, а также может задаваться через вызов функции внутри программы.

Каждый порождённый поток исполняет блок код в структурном блоке. По умолчанию синхронизация между потоками отсутствует и поэтому последовательность выполнения конкретного оператора различными потоками не определена.

После выполнения параллельного участка кода все потоки, кроме основного завершаются, и только основной поток продолжает исполняться, но уже один.

Каждый поток имеет свой уникальный номер, который изменяется от 0 (для основного потока) до количества потоков – 1. Идентификатор потока может быть определён с помощью функции omp_get_thread_num().

Зная идентификатор потока, можно внутри области параллельного исполнения направит потоки по разным ветвям.

Директива #pragma omp parallel for указывает на то, что данный цикл следует разделить по итерациям между потоками [5].

Реализация данного алгоритма представлена в приложении Б.

3.3.2 Параллельный алгоритм с использованием MPI

Любая прикладная MPI-программа должна начинаться с вызова функции инициализации MPI_Init(), инициализирующей группу процессов и создающую область памяти, определённой коммуникатором MPI_COMM_WORLD. Эта область связи объединяет все процессы-приложения [6].

Функция MPI_Comm_size() определяет число параллельных процессов, передавая значение в выходной параметр rank [6].

Функция MPI_Comm_rank() определяет номер процесса в группе, передавая значение в выходной параметр size [6].

Каждый процесс перед завершением должен вызывать MPI_FINALIZE, прежде убедившись в том, что все ожидающие неблокирующие взаимодействия завершены (локально) перед вызовом [6].

В метод, где будут вычисляться решения СЛАУ методом Крамера, помимо значений параметров, описанных в последовательном алгоритме, передадим также значения параметров rank и size и количество выполняемых операций вычисления numOfOperations для дальнейшего вычисления среднего времени работы алгоритма.

Для дальнейшей работы с функциями передачи сообщений между процессами в методе зададим структура MPI_Status status.

Зададим вектора типа double basicSolution и solution. В них будут храниться общее решение СЛАУ и решения СЛАУ каждого процесса соответственно. Для равномерного распределения вычислений между процессами также зададим 2 вектора типа int размером size: в первый вектор localCounts будем заносить количество обрабатываемых элементов каждым процессом, во второй offsets – смещение элементов для каждого процесса.

Перед подсчетом элементов вектора localCounts вычислим остаток от деления n на size и занесем его в переменную remainder. Далее в цикле по процессам получаем первоначальное количество элементов делением n на size. В случае наличия остатка, прибавляем еще 1 элемент и выполняем декремент переменной remainder. Здесь же в цикле через переменную sum вычисляем значение смещения и заносим его в вектор offsets.

В цикл по количеству выполняемых операций вычисления numOfOperations занесем всю основную логику программы. Если процесс имеет порядковый номер 0, то задаем размер вектора basicSolution равным n (так как этот процесс будет заполнять данный вектор), копируем переданное

в метод по ссылке значение coefficientsSLAU в массивы векторов basicMatrix и tempMatrix, вычисляем определитель системы и заносим его значение в переменную mainDet. Значения определителя mainDet и массива векторов tempMatrix как раз и будут передаваться другим процессам. Так как значения должны быть переданы всем процессам, то наиболее оптимальным вариантом передачи является использование коллективной функции. Для вычисления решений СЛАУ каждому процессу необходимо иметь все значений исходной матрицы, поэтому вызовем коллективную функцию широковещательной рассылки данных целиком int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm). Эта операция предполагает рассылку данных из буфера buffer, содержащего count элементов типа datatype с процесса, имеющего номер root, всем процессам, входящим в коммуникатор comm [7].

Дале каждый процесс обрабатывает свою порцию данных и формирует свой вектор значений solution, после чего все значения заносятся в вектор basicSolution. Это происходит посредством вызова коллективной функции сбора данных int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm). Каждый процесс в коммуникаторе comm передает sendcount элементов типа sendtype из буфера sendbuf на процесс с рангом root. Этот "ведущий" процесс осуществляет склейку поступающих от каждого процесса recvcounts элементов типа recvtype в буфере recbuf с учетом соответствующего смещения из displs. Склейка данных осуществляется линейно, положение пришедшего фрагмента данных определяется рангом процесса, его приславшего [7].

С помощью функции MPI_Barrier(MPI_COMM_WORLD) ожидаем выхода из цикла по numOfOperations всех процессов коммуникатора comm, считаем и выводим на консоль время работы алгоритма. Для измерения времени работы алгоритма в начале и конце используется функция double MPI_Wtime().

Реализация данного алгоритма представлена в приложении В.

3.4 Результаты работы

Сравнение результатов последовательного алгоритма и параллельного с технологией OpenMP представлены в таблице 1. В таблице 2 представлены измерения параллельного алгоритма с технологией MPI.

На рисунках 1 и 2 приведено сравнение времени последовательного алгоритма с параллельными с помощью OpenMP и MPI соответственно.

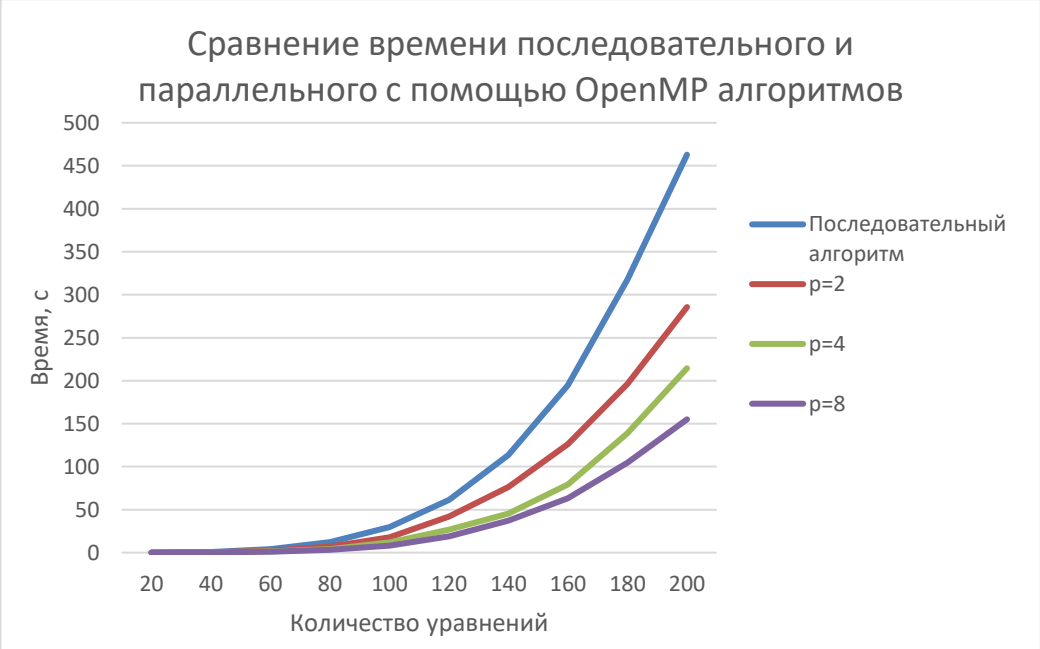

Рисунок 1 – Сравнение времени последовательного алгоритма с параллельным с помощью OpenMP

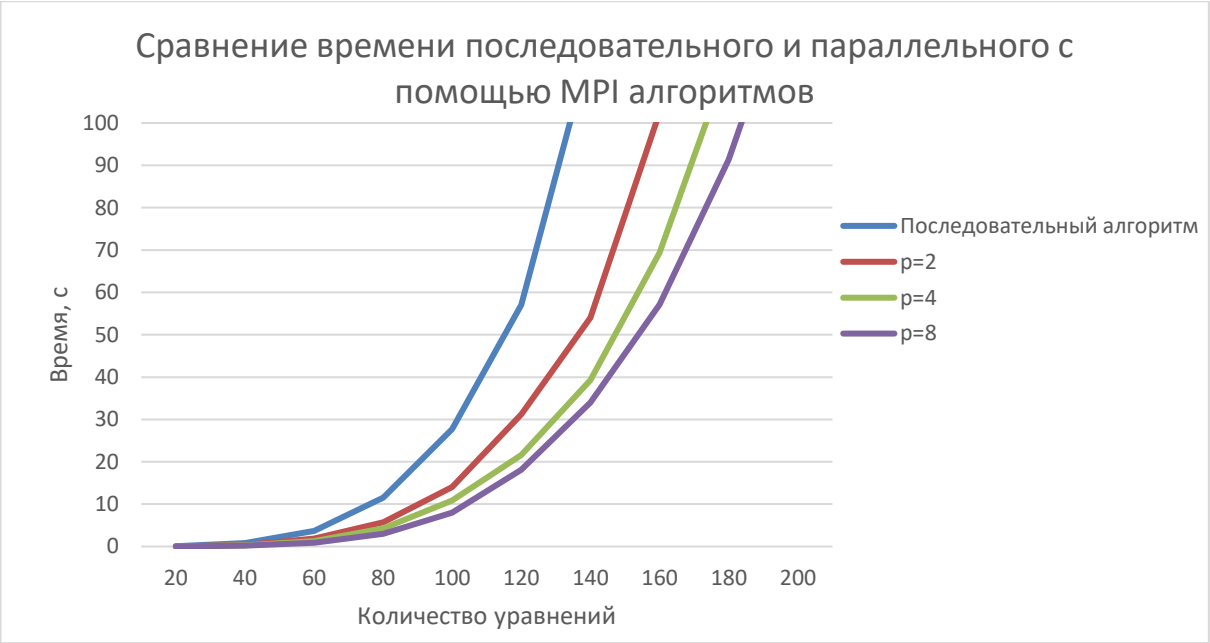

Рисунок 2 – Сравнение времени последовательного алгоритма с параллельным с помощью MPI

Таблица 1 – Сравнение результатов последовательного алгоритма и параллельного с технологией OpenMP

Количество итераций, n	20	40	60	80	100	120	140	160	180	200
Время работы последовательного алгоритма t,c	0,0649923	0,814896	3,70945	11,5147	27,6964	57,0367	118,098	178,252	285,002	433,217
Время работы алгоритма с помощью OpenMP p=2 t,c	0,0376304	0,501128	2,10022	6,18404	14,7811	31,9711	61,6425	101,936	161,13	244,348
Ускорение S	1,72712	1,62612	1,76622	1,86200	1,87377	1,78401	1,91585	1,74867	1,76877	1,77295
Эффективность E	0,86356	0,81306	0,88311	0,93100	0,93689	0,89200	0,95793	0,87433	0,88439	0,88648
Время работы алгоритма с помощью OpenMP p=4 t,c	0,0296133	0,333254	1,45799	4,23667	9,81469	23,5753	45,2179	74,9698	119,654	179,357
Ускорение S	2,19470	2,44527	2,54422	2,71787	2,82193	2,41934	2,61175	2,37765	2,38188	2,41539
Эффективность E	0,54867	0,61132	0,63606	0,67947	0,70548	0,60484	0,65294	0,59441	0,59547	0,60385
Время работы алгоритма с помощью OpenMP p=8 t,c	0,0224332	0,278357	1,07672	3,18463	7,01399	19,0925	36,6138	60,6142	95,9621	143,964
Ускорение S	2,89715	2,92752	3,44514	3,61571	3,94874	2,98739	3,22551	2,94076	2,96994	3,00920
Эффективность E	0,36214	0,36594	0,43064	0,45196	0,49359	0,37342	0,40319	0,36760	0,37124	0,37615

Таблица 2 – Сравнение результатов последовательного алгоритма и параллельного с технологией MPI

Количество итераций, n	20	40	60	80	100	120	140	160	180	200
Время работы последовательного алгоритма t,c	0,0649923	0,814896	3,70945	11,5147	27,6964	57,0367	118,098	178,252	285,002	433,217
Время работы алгоритма с помощью MPI p=2 t,c	0,0293611	0,388341	1,85255	5,70409	14,0276	31,2083	54,0564	102,459	146,13	224,555
Ускорение S	2,21355	2,09840	2,00235	2,01867	1,97442	1,82761	2,18472	1,73974	1,95033	1,92922
Эффективность E	1,10678	1,04920	1,00117	1,00934	0,98721	0,91381	1,09236	0,86987	0,97517	0,96461
Время работы алгоритма с помощью MPI p=4 t,c	0,0198551	0,25491	1,29461	4,31764	10,8975	21,5713	39,3111	69,3535	114,936	159,476
Ускорение S	3,27333	3,19680	2,86530	2,66690	2,54154	2,64410	3,00419	2,57019	2,47966	2,71650
Эффективность E	0,81833	0,79920	0,71633	0,66672	0,63538	0,66103	0,75105	0,64255	0,61991	0,67913
Время работы алгоритма с помощью MPI p=8 t,c	0,0169353	0,192666	0,912794	3,00829	8,02093	18,1563	33,9807	57,1178	91,3893	138,506
Ускорение S	3,83768	4,22958	4,06384	3,82766	3,45302	3,14143	3,47544	3,12078	3,11855	3,12779
Эффективность E	0,47971	0,52870	0,50798	0,47846	0,43163	0,39268	0,43443	0,39010	0,38982	0,39097

Сравнение ускорений параллельных алгоритмах при p=2, p=4, p=6 показано на рисунках 3, 4 и 5 соответственно.

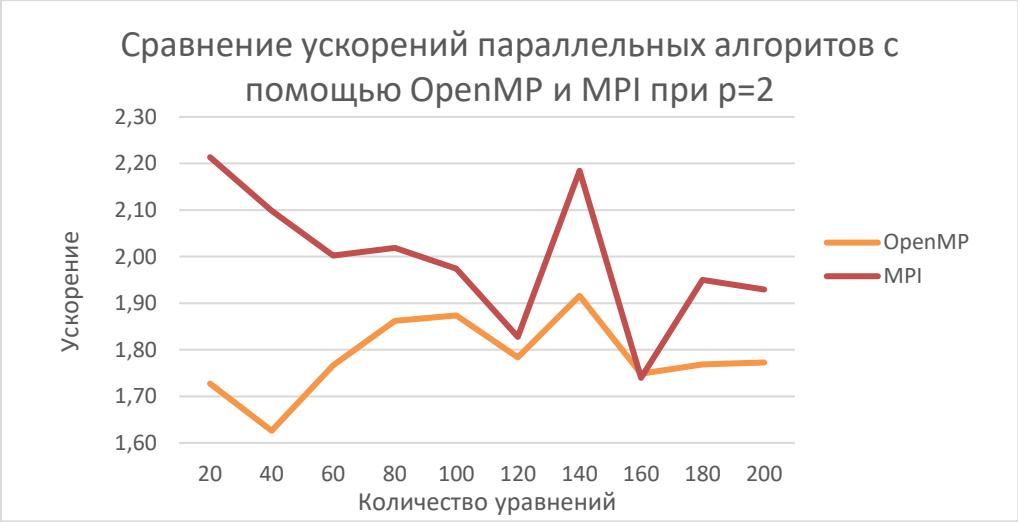

Рисунок 3 – Сравнение ускорений параллельных алгоритмов при p=2

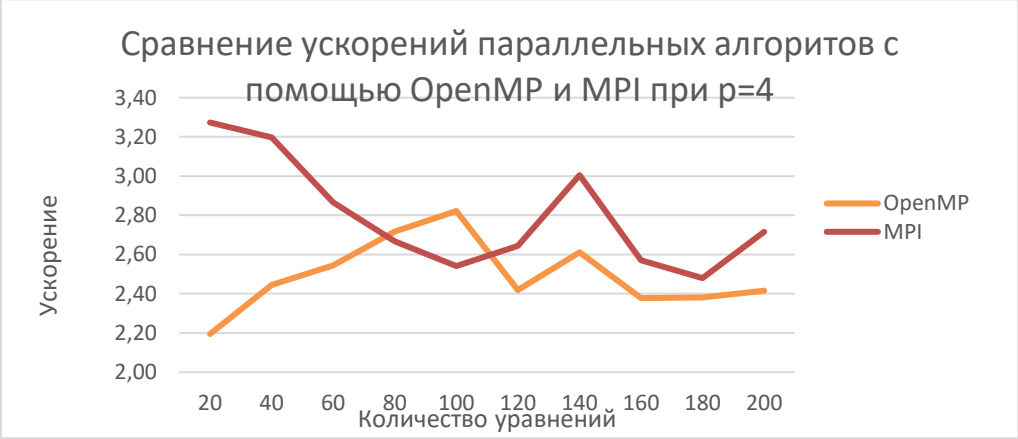

Рисунок 4 – Сравнение ускорений параллельных алгоритмов при p=4

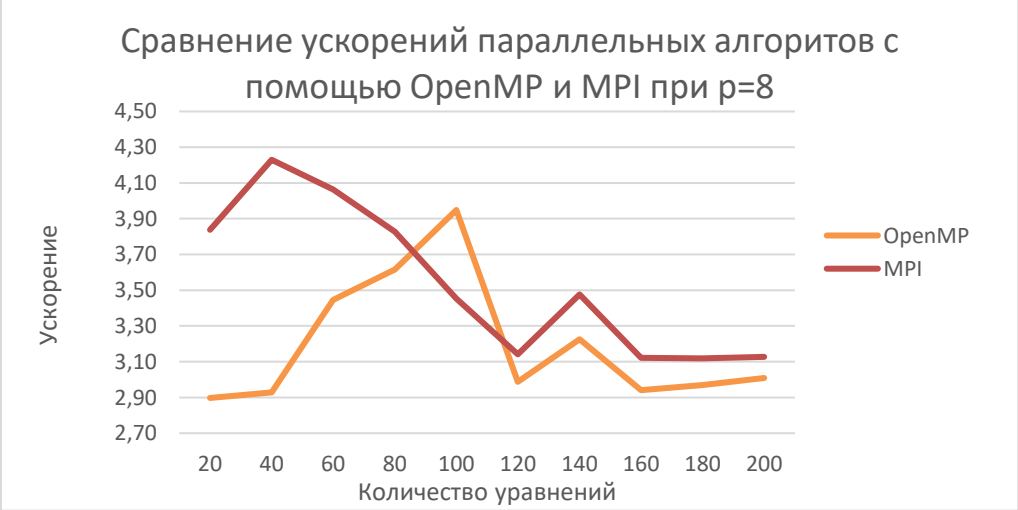

Рисунок 5 – Сравнение ускорений параллельных алгоритмов при p=8

Исходя из графиков, мы видим, что при любых значениях числа потоков p параллельный алгоритм на основе MPI в среднем дает лучшее ускорение S, чем параллельный алгоритм на основе OpenMP. Причем с увеличением p увеличивается среднее значение ускорения S.

На рисунках 6, 7 и 8 приведены сравнения эффективности параллельных алгоритмов при p=2, p=4 и p=6 соответственно.

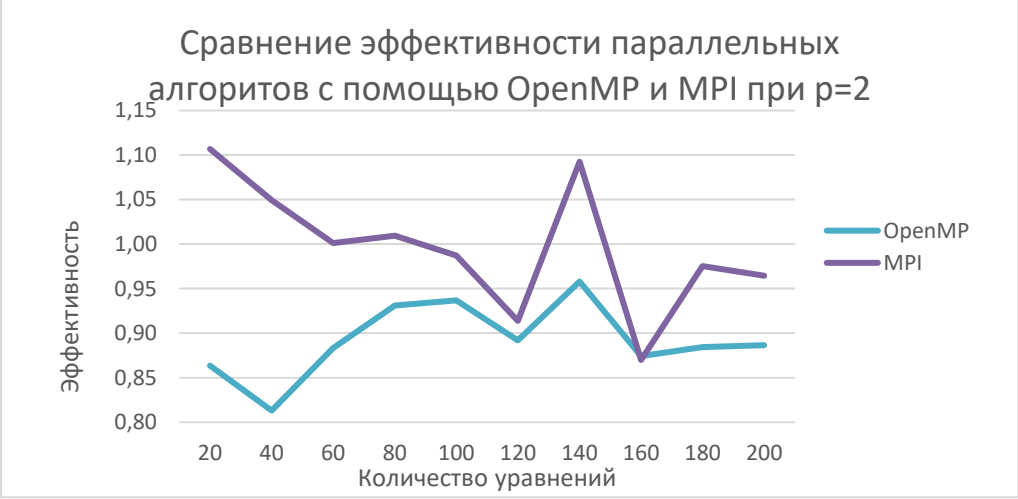

Рисунок 6 – Сравнение эффективности параллельных алгоритмов при p=2

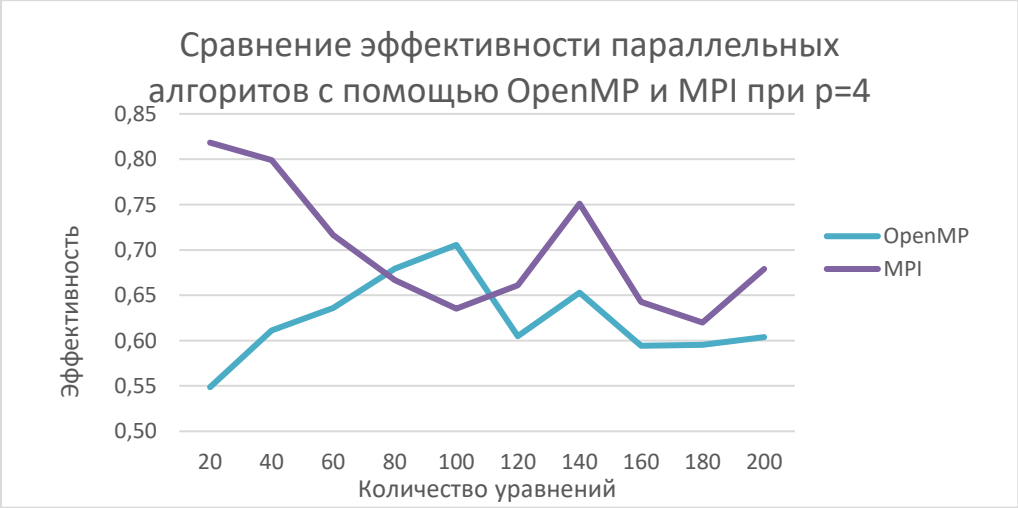

Рисунок 7 – Сравнение эффективности параллельных алгоритмов при p=4

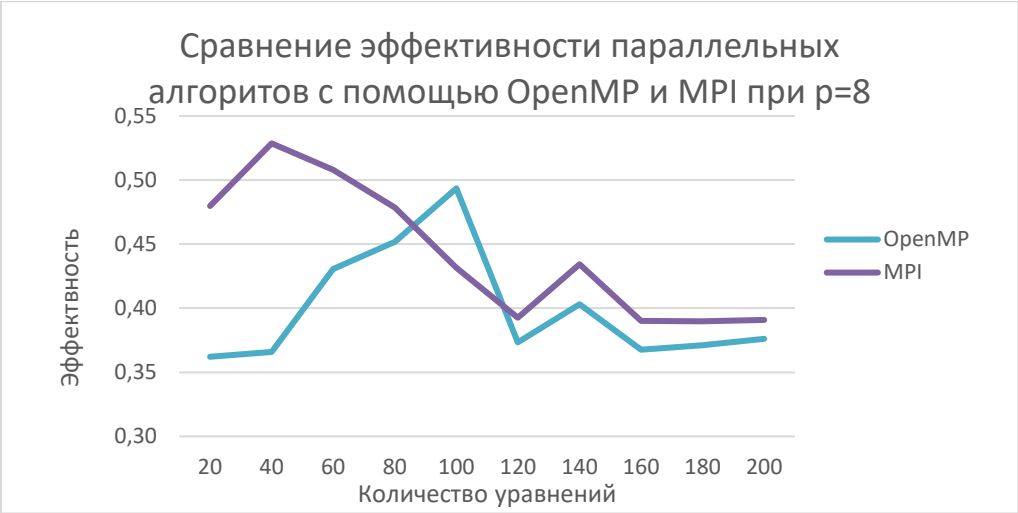

Рисунок 8 – Сравнение эффективности параллельных алгоритмов при p=8

Исходя из графиков, мы видим, что при любых значениях числа потоков p параллельный алгоритм на основе MPI в среднем имеет лучшую эффективность E, чем параллельный алгоритм на основе OpenMP. Причем с увеличением p уменьшается среднее значение эффективности E.

ЗАКЛЮЧЕНИЕ

В процессе выполнения курсового проекта был разработан последовательный и параллельный алгоритмы (с помощью технологии OpenMP и библиотеки MPI) решения СЛАУ методом Крамера в виде программы на языке C++, проведены вычислительные эксперименты для различных параметров задачи и получены следующие выводы и закономерности:

1. При любых параметрах задачи последовательный алгоритм работает медленнее, чем любой из параллельных алгоритмов, а алгоритм на основе OpenMP в среднем – медленнее, чем на основе MPI;
2. С увеличением числа потоков p время работы параллельных алгоритмов уменьшается, соответственно увеличивается ускорение S;
3. С увеличением числа потоков p эффективность E параллельных алгоритмов в среднем уменьшается. Снижение эффективности связано с увеличением затрат на коммуникацию при увеличении количества потоков. Как правило, при одинаковых входных данных, по мере увеличения числа процессоров делить задачу между ними поровну становится все труднее, загрузка потоков и переключение данных между ними занимает массу времени. В результате синхронность счета нарушается, время вычислений увеличивается и, следовательно, эффективность снижается.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Лекция 1: Введение [Электронный ресурс] // Лекция Интернет-Университет Суперкомпьютерных Технологий. Основы параллельных вычислений: [сайт]. URL: https://intuit.ru/studies/courses/1091/293/lecture/7337 (дата обращения: 21.11.2021).

2 Применение параллельных методов решения систем линейных уравнений ленточных матриц [Электронный ресурс] // Библиофонд [сайт]. URL: https://www.bibliofond.ru/view.aspx?id=663300 (дата обращения: 21.11.2021).

3 Методы решения больших СЛАУ на разреженных матрицах общего вида [Электронный ресурс] // ДонНТУ. Портал магистров [сайт]. URL: https://masters.donntu.org/2010/fknt/dyachenko/library/article7.htm (дата обращения: 21.11.2021).

4 Метод Крамера [Электронный ресурс] // Справочник [сайт]. URL: https://spravochnick.ru/matematika/metod_kramera/ (дата обращения: 21.11.2021).

5 Параллельное программирование на OpenMP [Электронный ресурс] // Сервер компьютерных классов Факультета информационных технологий НГУ [сайт]. URL: http://ccfit.nsu.ru/arom/data/openmp.pdf/ (дата обращения: 26.11.2021).

6 Технологии параллельного программирования. Message Passing Interface (MPI) [Электронный ресурс] // Лаборатория Параллельных информационных технологий Научно-исследовательского вычислительного центра Московского государственного университета имени М.В.Ломоносова [сайт]. URL: https://parallel.ru/vvv/mpi.html (дата обращения: 08.12.2021).

7 Коллективные операции [Электронный ресурс] // Параллельное программирование в системах с общей памятью. Инструментальная поддержка (MPI) [сайт]. URL: http://www.unn.ru/pages/e-library/aids/2007/85.pdf (дата обращения: 10.12.2021).

ПРИЛОЖЕНИЕ А

Реализация последовательного алгоритма

```
#include <iostream>
#include <vector>
#include <ctime>
#include "omp.h"

using namespace std;

int findMaxInColumn(const vector<vector<double>>& matrix, int col, int n) {
	int max = abs(matrix[col][col]);
	int maxPos = col;
	for (int i = col + 1; i < n; ++i) {
		int element = abs(matrix[i][col]);
		if (element > max) {
			max = element;
			maxPos = i;
		}
	}
	return maxPos;
}

int toTriangularMatrix(vector<vector<double>>& matrix, int n) {
	unsigned int swapCount = 0;
	for (int i = 0; i < n - 1; ++i) {
		unsigned int imax = findMaxInColumn(matrix, i, n);
		if (i != imax) {
			swap(matrix[i], matrix[imax]);
			++swapCount;
		}
		for (int j = i + 1; j < n; ++j) {
			if (matrix[i][i] == 0)
				exit(1);
			double mul = -matrix[j][i] / matrix[i][i];
			for (int k = i; k < n; ++k)
				matrix[j][k] += matrix[i][k] * mul;
		}
	}
	return swapCount;
}

void swapCoefficientsWithConsts(vector<vector<double>> & matrix, vector<double> &
constantTermsSLAU, int i, int n) {
	double temp;
	for (int j = 0; j < n; j++) {
		temp = matrix[j][i];
		matrix[j][i] = constantTermsSLAU[j];
		constantTermsSLAU[j] = temp;
	}
}

double findDetByGauss(vector<vector<double>> & matrix, int n) {
	unsigned int swapCount = toTriangularMatrix(matrix, n);
	double det;
	if (swapCount % 2 == 1)
		det = -1;
	else
		det = 1;
	for (int i = 0; i < n; ++i)
		det *= matrix[i][i];
	return det;
}
```

```
vector<double> metodCramer(vector<vector<double>> & coefficientsSLAU, vector<double> &
constantTermsSLAU, int n) {
	vector<double> solution(n);
	vector<vector<double>> basicMatrix(coefficientsSLAU);
	vector<vector<double>> tempMatrix(coefficientsSLAU);
	double mainDet = findDetByGauss(basicMatrix, n);
	for (int i = 0; i < n; ++i) {
		swapCoefficientsWithConsts(tempMatrix, constantTermsSLAU, i, n);
		vector<vector<double>> tempTriangleMatrix = tempMatrix;
		solution[i] = findDetByGauss(tempTriangleMatrix, n) / mainDet;
		swapCoefficientsWithConsts(tempMatrix, constantTermsSLAU, i, n);
	}
	return solution;
}

int random(int a, int b)
{
	srand(time(NULL));
	if (a > 0)
		return a + rand() % (b - a);
	else
		return a + rand() % (abs(a) + b);
}

int main() {
	setlocale(LC_ALL, "Russian");
	cout << "\n Курсовая работа по предмету \"Параллельное программирование\"" <<
endl;
	cout << "\n\n Последовательный алгоритм решения СЛАУ методом Крамера\n" << endl;
	int n = 20;
	double timeStart, timeFinish;
	vector<vector<double>> coefficientsSLAU;
	vector<double> constantTermsSLAU;
	do {
		coefficientsSLAU.resize(n);
		constantTermsSLAU.resize(n);
		for (int i = 0; i < n; i++) {
			coefficientsSLAU[i].resize(n);
			for (int j = 0; j < n; j++)
				coefficientsSLAU[i][j] = random(-30, 30);
			constantTermsSLAU[i] = random(-100, 100);
		}
		timeStart = omp_get_wtime();
		vector<double> solution = metodCramer(coefficientsSLAU, constantTermsSLAU,
n);
		timeFinish = omp_get_wtime();
		cout << "  n = " << n << ", t = " << timeFinish - timeStart << endl;
		n += 20;
	} while (n <= 200);

	cout << "\n\n Выполнили: Гижевская В.Д и Сусликова М.С." << endl;
	system("pause");
}
```

ПРИЛОЖЕНИЕ Б

Реализация параллельного алгоритма с помощью OpenMP

```
#include <stdio.h>
#include <omp.h>
#include <cstdlib>
#include <iostream>
#include <vector>
#include <ctime>

using namespace std;

int findMaxInColumn(const vector<vector<double>>& matrix, int col, int n) {
	int max = abs(matrix[col][col]);
	int maxPos = col;
	for (int i = col + 1; i < n; ++i) {
		int element = abs(matrix[i][col]);
		if (element > max) {
			max = element;
			maxPos = i;
		}
	}
	return maxPos;
}

int toTriangularMatrix(vector<vector<double>>& matrix, int n) {
	unsigned int swapCount = 0;
	for (int i = 0; i < n - 1; ++i) {
		unsigned int imax = findMaxInColumn(matrix, i, n);
		if (i != imax) {
			swap(matrix[i], matrix[imax]);
			++swapCount;
		}
#pragma omp parallel for
		for (int j = i + 1; j < n; ++j) {
			if (matrix[i][i] == 0)
				exit(1);
			double mul = -matrix[j][i] / matrix[i][i];
			for (int k = i; k < n; ++k)
				matrix[j][k] += matrix[i][k] * mul;
		}
	}
	return swapCount;
}

void swapCoefficientsWithConsts(vector<vector<double>> & matrix, vector<double> &
constantTermsSLAU, int i, int n) {
	double temp;
	for (int j = 0; j < n; j++) {
		temp = matrix[j][i];
		matrix[j][i] = constantTermsSLAU[j];
		constantTermsSLAU[j] = temp;
	}
}

double findDetByGauss(vector<vector<double>> & matrix, int n) {
	unsigned int swapCount = toTriangularMatrix(matrix, n);
	double det;
	if (swapCount % 2 == 1)
		det = -1;
	else
		det = 1;
	for (int i = 0; i < n; ++i)
```

```
			det *= matrix[i][i];
		return det;
}

vector<double> metodCramer(vector<vector<double>> & coefficientsSLAU, vector<double> &
constantTermsSLAU, int n) {
		vector<double> solution(n);
		vector<vector<double>> basicMatrix(coefficientsSLAU);
		vector<vector<double>> tempMatrix(coefficientsSLAU);
		double mainDet = findDetByGauss(basicMatrix, n);
#pragma omp parallel for
		for (int i = 0; i < n; ++i) {
			swapCoefficientsWithConsts(tempMatrix, constantTermsSLAU, i, n);
			vector<vector<double>> tempTriangleMatrix = tempMatrix;
			solution[i] = findDetByGauss(tempTriangleMatrix, n) / mainDet;
			swapCoefficientsWithConsts(tempMatrix, constantTermsSLAU, i, n);
		}
		return solution;
}

int random(int a, int b)
{
		srand(time(NULL));
		if (a > 0)
			return a + rand() % (b - a);
		else
			return a + rand() % (abs(a) + b);
}

int main() {
		setlocale(LC_ALL, "Russian");

		omp_set_num_threads(8);

		cout << "\n Курсовая работа по предмету \"Параллельное программирование\"" <<
endl;
		cout << "\n\n Параллельный алгоритм решения СЛАУ методом Крамера" << endl;
		cout << " с помощью OpenMP (p=8)\n" << endl;
		int n = 20;
		double timeStart, timeFinish;
		vector<vector<double>> coefficientsSLAU;
		vector<double> constantTermsSLAU;
		do {
			coefficientsSLAU.resize(n);
			constantTermsSLAU.resize(n);
			for (int i = 0; i < n; i++) {
				coefficientsSLAU[i].resize(n);
				for (int j = 0; j < n; j++)
					coefficientsSLAU[i][j] = random(-30, 30);
				constantTermsSLAU[i] = random(-100, 100);
			}
			timeStart = omp_get_wtime();
			vector<double> solution = metodCramer(coefficientsSLAU, constantTermsSLAU,
n);
			timeFinish = omp_get_wtime();
			cout << "  n = " << n << ", t = " << timeFinish - timeStart << endl;
			n += 20;
		} while (n <= 200);

		cout << "\n\n Выполнили: Гижевская В.Д и Сусликова М.С." << endl;
		system("pause");
}
```

ПРИЛОЖЕНИЕ В

Реализация параллельного алгоритма с помощью MPI

```
#include <iostream>
#include <vector>
#include <ctime>
#include <mpi.h>
#include "omp.h"

using namespace std;

int findMaxInColumn(const vector<vector<double>>& matrix, int col, int n) {
	int max = abs(matrix[col][col]);
	int maxPos = col;
	for (int i = col + 1; i < n; i++) {
		int element = abs(matrix[i][col]);
		if (element > max) {
			max = element;
			maxPos = i;
		}
	}
	return maxPos;
}

int toTriangularMatrix(vector<vector<double>>& matrix, int n) {
	unsigned int swapCount = 0;
	for (int i = 0; i < n - 1; ++i) {
		unsigned int imax = findMaxInColumn(matrix, i, n);
		if (i != imax) {
			swap(matrix[i], matrix[imax]);
			++swapCount;
		}
		for (int j = i + 1; j < n; ++j) {
			if (matrix[i][i] == 0)
				exit(1);
			double mul = -matrix[j][i] / matrix[i][i];
			for (int k = i; k < n; ++k)
				matrix[j][k] += matrix[i][k] * mul;
		}
	}
	return swapCount;
}

void swapCoefficientsWithConsts(vector<vector<double>>& matrix, vector<double>&
constantTermsSLAU, int i, int n) {
	double temp;
	for (int j = 0; j < n; j++) {
		temp = matrix[j][i];
		matrix[j][i] = constantTermsSLAU[j];
		constantTermsSLAU[j] = temp;
	}
}

double findDetByGauss(vector<vector<double>>& matrix, int n) {
	unsigned int swapCount = toTriangularMatrix(matrix, n);
	double det;
	if (swapCount % 2 == 1)
		det = -1;
	else
		det = 1;
	for (int i = 0; i < n; ++i)
		det *= matrix[i][i];
	return det;
```

```
}

void parallelMetodCramer(vector<vector<double>>& coefficientsSLAU, vector<double>&
constantTermsSLAU, int n, int numOfOperations, int rank, int size) {
	MPI_Status status;
	vector<vector<double>> basicMatrix, tempMatrix;
	vector<double> basicSolution, solution;
	vector<int> localCounts(size), offsets(size);
	int remainder = n % size, sum = 0;
	double mainDet;
	for (int i = 0; i < size; i++) {
		localCounts[i] = n / size;
		if (remainder > 0) {
			localCounts[i] += 1;
			remainder--;
		}
		offsets[i] = sum;
		sum += localCounts[i];
	}
	solution.resize(localCounts[rank]);
	for (int i = 0; i < numOfOperations; i++) {
		if (rank == 0) {
			basicSolution.resize(n);
			basicMatrix = coefficientsSLAU;
			tempMatrix = coefficientsSLAU;
			mainDet = findDetByGauss(basicMatrix, n);
		}
		else {
			tempMatrix.resize(n);
			for (int j = 0; j < n; j++)
				tempMatrix[j].resize(n);
		}
		for (int j = 0; j < n; j++)
			MPI_Bcast(tempMatrix[j].data(), n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
		MPI_Bcast(&mainDet, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
		for (int j = offsets[rank]; j < offsets[rank] + localCounts[rank]; j++) {
			swapCoefficientsWithConsts(tempMatrix, constantTermsSLAU, j, n);
			vector<vector<double>> tempTriangleMatrix = tempMatrix;
			solution[j - offsets[rank]] = findDetByGauss(tempTriangleMatrix, n) /
mainDet;
			swapCoefficientsWithConsts(tempMatrix, constantTermsSLAU, j, n);
		}
		MPI_Gatherv(solution.data(), localCounts[rank], MPI_DOUBLE,
basicSolution.data(), localCounts.data(), offsets.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
	}
	MPI_Barrier(MPI_COMM_WORLD);
}

int random(int a, int b)
{
	if (a > 0)
		return a + rand() % (b - a);
	else
		return a + rand() % (abs(a) + b);
}

int main(int argc, char** argv) {
	setlocale(LC_ALL, "Russian");
	int n = 20, rank, size, numOfOperations = 5;
	double timeStart = 0, timeFinish = 0;
	vector<vector<double>> coefficientsSLAU;
	vector<double> constantTermsSLAU;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
	srand(time(NULL));
	do {
		coefficientsSLAU.resize(n);
		constantTermsSLAU.resize(n);
		for (int i = 0; i < n; i++) {
			coefficientsSLAU[i].resize(n);
			for (int j = 0; j < n; j++)
				coefficientsSLAU[i][j] = random(-30, 30);
			constantTermsSLAU[i] = random(-100, 100);
		}
		if (rank == 0)
			timeStart = MPI_Wtime();
		parallelMetodCramer(coefficientsSLAU, constantTermsSLAU, n,
numOfOperations, rank, size);
		if (rank == 0) {
		timeFinish = MPI_Wtime();
		cout << "n = " << n << ", MPI t = " << (timeFinish - timeStart) /
numOfOperations << endl;
	}
		n += 20;
	} while (n <= 200);
	MPI_Finalize();
	return 0;
}
```