МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Южно-Уральский государственный университет»
(национальный исследовательский университет)
Высшая школа электроники и компьютерных наук
Кафедра системного программирования

РАБОТА ПРОВЕРЕНА

Рецензент
к.т.н., доцент

_____ Т.Ю. Оленчикова

“___”_____ 2017 г.

ДОПУСТИТЬ К ЗАЩИТЕ

Заведующий кафедрой,
д.ф.-м.н., профессор

_____ Л.Б. Соколинский

“___”_____ 2017 г.

«Q-ЭФФЕКТИВНАЯ РЕАЛИЗАЦИЯ АЛГОРИТМОВ УМНОЖЕНИЯ МАТРИЦ НА СУПЕРКОМПЬЮТЕРЕ «ТОРНАДО ЮУРГУ»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
ЮУрГУ – 02.03.02.2016.13-004-1382.ВКР

Научный руководитель
к.ф.-м.н., доцент
_____В.Н. Алеева

Автор работы,
студент группы КЭ-401
_____Н.В. Валькевич

Ученый секретарь
(нормоконтролер)

_____ О.Н. Иванова

“___”_____ 2017 г.

Челябинск-2017

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время многоядерная архитектура является основной для подавляющего большинства ЭВМ. Преимущество данной архитектуры заключается в возможности исполнения параллельных алгоритмов.

На сегодняшний день большое внимание уделяется программированию и выполнению таких алгоритмов, разработано множество подходов и технологий [1, 15], но также необходимо совершенствовать теорию построения параллельных алгоритмов. Одной из таких теорий является концепция *Q*-детерминанта [6], согласно которой можно представить любой алгоритм в виде множества *Q*-термов.

Актуальность использования концепции *Q*-детерминанта очевидна: на данный момент не существует ни одной модели, позволяющей полноценно описать параллельный алгоритм. В то же время, единая форма представления параллельного алгоритма позволит уменьшить количество ошибок в параллельной программе.

На данный момент опубликовано несколько работ о *Q*-детерминанте и работе с ним [1, 14-15].

Популярными в настоящее время являются технологии CUDA [11], MPI [12], OpenMP [13], которые позволяют осуществлять выполнение параллельных алгоритмов на процессорах компаний Intel и NVIDIA.

Цель и задачи исследования

Целью данной работы является *Q*-эффективная реализация алгоритмов умножения плотных и разреженных матриц на суперкомпьютере «Торнадо ЮУрГУ», анализ ускорения и эффективности выполнения полученных реализаций как на общей, так и на распределенной памяти.

Для достижения поставленной цели необходимо решить следующие *задачи:*

1) изучение подхода к построению Q-эффективной реализации, основанной на представлении алгоритмов в форме Q-детерминанта;
2) изучение архитектуры суперкомпьютера «Торнадо ЮУрГУ»;
3) изучение программных средств для выполнения параллельных реализаций алгоритмов на суперкомпьютере «Торнадо ЮУрГУ»;
4) программирование *Q*-эффективных реализаций алгоритмов умно-

жения плотных и разреженных матриц на суперкомпьютере «Торнадо ЮУрГУ» с использованием различных программных средств на общей и распределенной памяти;

5) проведение вычислительных экспериментов на различных вычислительных мощностях и объёмах данных;

6) анализ ускорения разработанных параллельных программ в сравнении с последовательными;

7) анализ эффективности выполнения разработанных параллельных программ.

Структура и объем работы

Выпускная квалификационная работа состоит из введения, четырех разделов, заключения, библиографии. Объем работы составляет 33 страницы, объем библиографии – 17 наименований.

Содержание работы

Первый раздел «Анализ предметной области» содержит описание концепции *Q*-детерминанта и технологий распараллеливания алгоритмов, а также характеристики суперкомпьютера «Торнадо ЮУрГУ».

Второй раздел «Проектирование» содержит диаграмму вариантов использования и описание модульной структуры.

В третьем разделе «Реализация» описаны Q-эффективные программы умножения плотных и разреженных матриц на общей и распределенной памяти.

В четвертом разделе «Вычислительные эксперименты» представлены результаты вычислительных экспериментов и их анализ.

В заключении подводятся итоги проделанной работы.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Концепция *Q*-детерминанта

Пусть *A* - некоторый алгоритм для решения алгоритмической проблемы, *B* - множество входных данных (счетное множество переменных). Пусть *Q* - множество операций, используемых алгоритмом *A*. Будем допускать, что все операции из *Q* нульместны (константы), одноместны, или двуместны. Примером множества *Q* является множество арифметических, логических операций и операций сравнения. Над множествами *B* и *Q* можно строить выражения, которые имеют уровни вложенности.

Под вычислением *Q-терма f* при интерпретации следует понимать вычисление его выражения *w* при некотором множестве параметров *N*. Если алгоритм *A* состоит в том, что для выполнения требуется вычислить Q-терм *f*, то множество *Q*-термов *F* называется *Q-детерминантом* алгоритма *A*. Представление алгоритма *A* в виде множества *Q*-термов называется представлением алгоритма в форме *Q*-детерминанта.

Если в *Q*-детерминанте все операции *w* вычисляются одновременно (параллельно) и при вычислении каждого из выражений операции выполняются по мере их готовности, то реализация называется *Q-эффективной*. *Q*-эффективная реализация алгоритма с формальной точки зрения является максимально быстрой.

Реализация алгоритма *A* называется *выполнимой*, если одновременно необходимо выполнять конечное число операций. Существуют алгоритмы, для которых *Q*-эффективная реализация не является выполнимой.

1.2. Метод конструирования параллельных программ на основе концепции *Q*-детерминанта

Подход к разработке программы, выполняющей *Q*-эффективную реализацию алгоритма, основан на следующих утверждениях:

- *Q*-детерминант можно построить для любого численного алгоритма;
- *Q*-детерминант позволяет описать *Q*-эффективную реализацию алгоритма;
- если *Q*-эффективная реализация алгоритма является выполнимой, то можно разработать программу для ее выполнения.

Процесс разработки программы состоит из следующих этапов:

- построение *Q*-детерминанта алгоритма;
- описание *Q*-эффективной реализация алгоритма;
- если *Q*-эффективная реализация выполнима, то для нее разрабатывается программа.

Разработанную программу будем называть *Q*-эффективной, а процесс ее разработки *Q*-эффективным программированием. *Q*-эффективная программа полностью использует ресурс параллелизма алгоритма, т.к. выполняет его *Q*-эффективную реализацию. В связи с этим она не допускает дальнейшего распараллеливания.

Q-эффективная программа – программа, выполняющая *Q*-эффективную реализацию алгоритма на определённой вычислительной архитектуре. В данной работе для рассматриваются следующие особенности архитектуры: наличие общей либо распределенной памяти, а также количество независимых вычислительных узлов.

1.3. Распараллеливание вычислений

Идея распараллеливания вычислений основана на том, что большинство задач может быть разделено на набор меньших задач, которые могут быть решены одновременно.

Существует три варианта распараллеливания программы:

- параллелизм команд (инструкций) – команды программы разбиваются на группы, вычисляемые параллельно;
- параллелизм данных – данные разделяются на независимые части, к каждой части параллельно применяются установленные команды;
- параллелизм задач – алгоритм разбивается на независимые задачи, затем каждая выполняется параллельно.

1.3.1. OpenMP

OpenMP [5, 13] – стандарт распараллеливания программ на языках C, C++ и Fortran. OpenMP может применяться для распараллеливания программ с большими циклами независимых итераций. Сам механизм разложения программы подразумевает добавление OpenMP-директив в исходный код. Это дает стандарту гибкость, предоставляющую большие возможности контроля над поведением параллельного приложения.

Под параллельной программой в рамках OpenMP понимается программа, для которой в специально указываемых при помощи директив местах – параллельных фрагментах – исполняемый программный код может быть разделен на несколько раздельных командных потоков (threads). Важно отметить, что разделение вычислений между потоками осуществляется под управлением соответствующих директив. Равномерное распределение вычислительной нагрузки – балансировка (load balancing) – имеет принципиальное значение для получения максимально возможного ускорения выполнения параллельной программы.

Потоки могут выполняться на разных процессорах (процессорных ядрах) либо могут группироваться для исполнения на одном вычислительном элементе (в этом случае их исполнение осуществляется в режиме разделения времени). В предельном случае для выполнения параллельной программы может использоваться один процессор – как правило, такой способ применяется для начальной проверки правильности параллельной программы.

При выполнении программ код разделяется на потоки, или нити (англ. threads), каждая из которых выполняется отдельно. В конце результат объединяется.

На рисунке 1 представлена модель распараллеливания программы. Главный поток (Master Thread) состоит из трех заданий (Parallell Task I – III), которые могут разбиты на три, четыре и два подзадания соответственно.

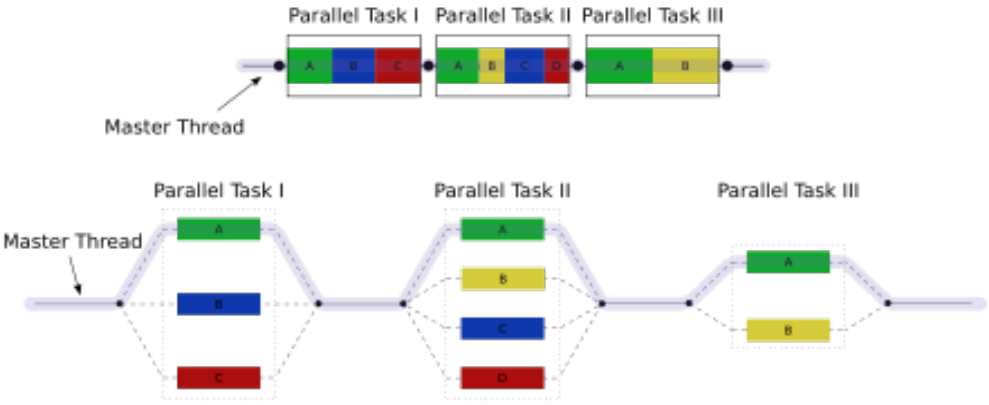

Рис. 1. Разбиение на потоки в OpenMP.

1.3.2. MPI

MPI (Message Parsing Interface) [12] – программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. MPI является наиболее распространенным стандартом интерфейса обмена данными для вычислительныз систем с распределенной памятью, существуют его реализации для большого числа компьютерных платформ. Используется при разработке программ для кластеров и суперкомпьютеров.

Базовым механизмом связи между MPI процессами является передача и прием сообщений. Сообщение несет в себе передаваемые данные и информацию, позволяющую принимающей стороне осуществлять их выборочный прием:

- отправитель – ранг (номер в группе) отправителя сообщения;
- получатель – ранг получателя;
- признак – может использоваться для разделения различных видов сообщений;
- коммуникатор - код группы процессов.

Операции приема и передачи могут быть блокирующимися и неблокирующимися. Для неблокирующихся операций определены функции проверки готовности и ожидания выполнения операции.

Другим способом связи является удаленный доступ к памяти (RMA), позволяющий читать и изменять область памяти удаленного процесса. Локальный процесс может переносить область памяти удаленного процесса (внутри указанного процессами окна) в свою память и обратно, а также комбинировать данные, передаваемые в удаленный процесс с имеющимися в его памяти данными (например, путем суммирования). Все операции удаленного доступа к памяти не блокирующиеся, однако, до и после их выполнения необходимо вызывать блокирующиеся функции синхронизации.

1.3.3. CUDA

CUDA (англ. Compute Unified Device Architecture) [11] – программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы Nvidia. Архитектура CU-

DA дает разработчику возможность по своему усмотрению организовывать доступ к набору инструкций графического ускорителя и управлять его памятью.

По сравнению с традиционным подходом к организации вычислений общего назначения посредством возможностей графических API, у архитектуры CUDA отмечают следующие преимущества [7] в этой области:

- интерфейс программирования приложений CUDA (CUDA API) основан на стандартном языке программирования Си с некоторыми ограничениями;
- более эффективные транзакции между памятью центрального процессора и видеопамятью;
- полная аппаратная поддержка целочисленных и побитовых операций;
- поддержка компиляции GPU кода средствами открытого LLVM;
- возможность использования видеокарт персональных компьютеров для вычислений.

1.4. Формат CSR

Разреженная матрица – это матрица с преимущественно нулевыми элементами. В случае, если большая часть элементов матрицы ненулевые, матрица считается плотной. На данный момент нет единого критерия определения количества нулевых элементов в разреженной матрице.

Для хранения разреженных матриц используется формат CSR (Compressed Sparse Rows) [5]. Для представления матрицы в памяти используются три массива. Первый массив хранит значения элементов построчно (строки рассматриваются по порядку сверху вниз), второй – номера столбцов для каждого элемента, а в третьем – индекс начала каждой строки (рис. 2). Отметим, что количество элементов массива *RowIndex* равно N+1. При этом элементы строки i в массиве *Value* находятся по индексам от *RowIndex*[i] до *RowIndex*[i + 1] – 1 включительно (i-ый элемент массива *RowIndex* указывает на начало i-ой строки). Исходя из этого обрабатывается случай пустых строк, а также добавляется «лишний» элемент в массив *RowIndex* – устраняется особенность при доступе к элементам последней строки. Данный количество ненулевых элементов массива.

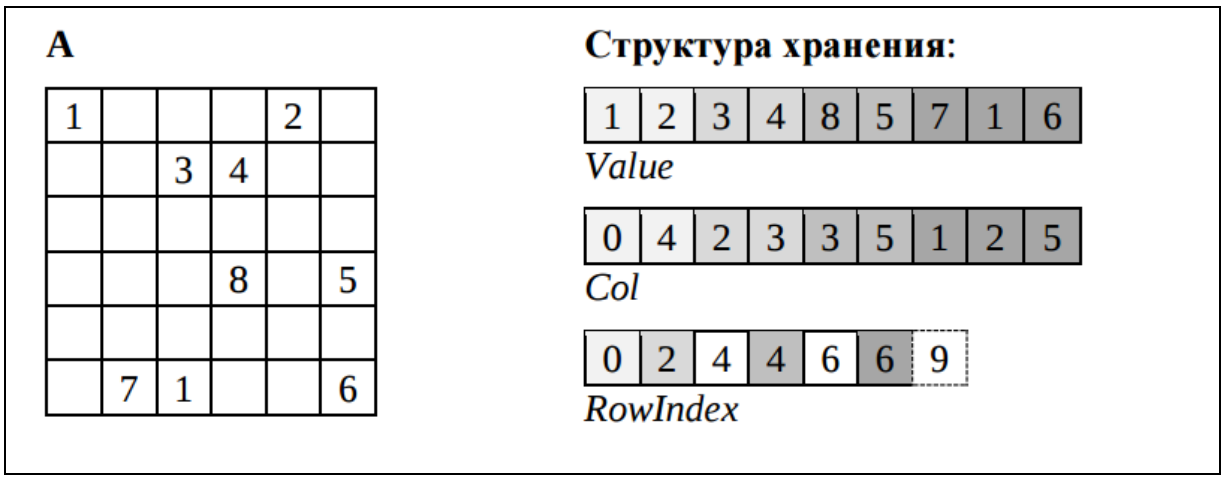

Рис. 2. Представление матрицы *A* в формате CSR

1.5. Суперкомпьютер «Торнадо ЮУрГУ»

Для вычислений был выбран суперкомпьютер «Торнадо ЮУрГУ», входящий в рейтинги Green500 и СНГ ТОР50.

В таблице 1 представлены характеристики используемого суперкомпьютера.

Табл. 1. Характеристики суперкомпьютера «Торнадо ЮУрГУ»

Характеристики	Процессор
Число вычислительных узлов/процессорных ядер:	480/29184
Тип процессора:	Intel Xeon X5680 (Gulftown, 6 ядер по 3.33 GHz) — 960 шт.
Оперативная память:	16.9 TB
Дисковая память:	300 TB, параллельная СХД Panasas ActiveStore 11
Тип управляющей сети:	Gigabit Ethernet
Пиковая производительность комплекса:	473.6 TFlops
Производительность комплекса на тесте LINPACK:	288.2 TFlops

В таблице 2 представлены вычислительные характеристики узлов

суперкомьютера

Табл. 2. Характеристики вычислительного узла суперкомпьютера «Торнадо ЮУрГУ»

Характеристики	Процессор
Модель	Intel Xeon X5680
Количество ядер	6
Характеристики	Процессор
Частота ядер, GHz	3.33
Количество потоков на ядро	2
Производительность, Тфлопс	0.371

Вывод

После анализа программных и аппаратных средств суперкомпьютера «Торнадо ЮУрГУ» было принято решение использовать для выполнения ВКР технологии OpenMP и MPI.

2. ПРОЕКТИРОВАНИЕ

2.1 Варианты использования

Реализуемые Q-эффективные программы представляются в виде консольного приложения, позволяющего осуществлять умножение плотных и разреженных матриц на общей и распределенной памяти. Единственным актером данной системы является *Исследователь* - пользователь системы, выполняющий вычислительные эксперименты.

Исследователь может:

- задавать размерность исходных матриц;
- выполнять умножение полных и разреженных матриц на общей либо распределенной памяти;
- получать результат выполнения алгоритма;
- осуществлять проверку результирующих матриц.

Диаграмма вариантов использования представлена на рис. 3.

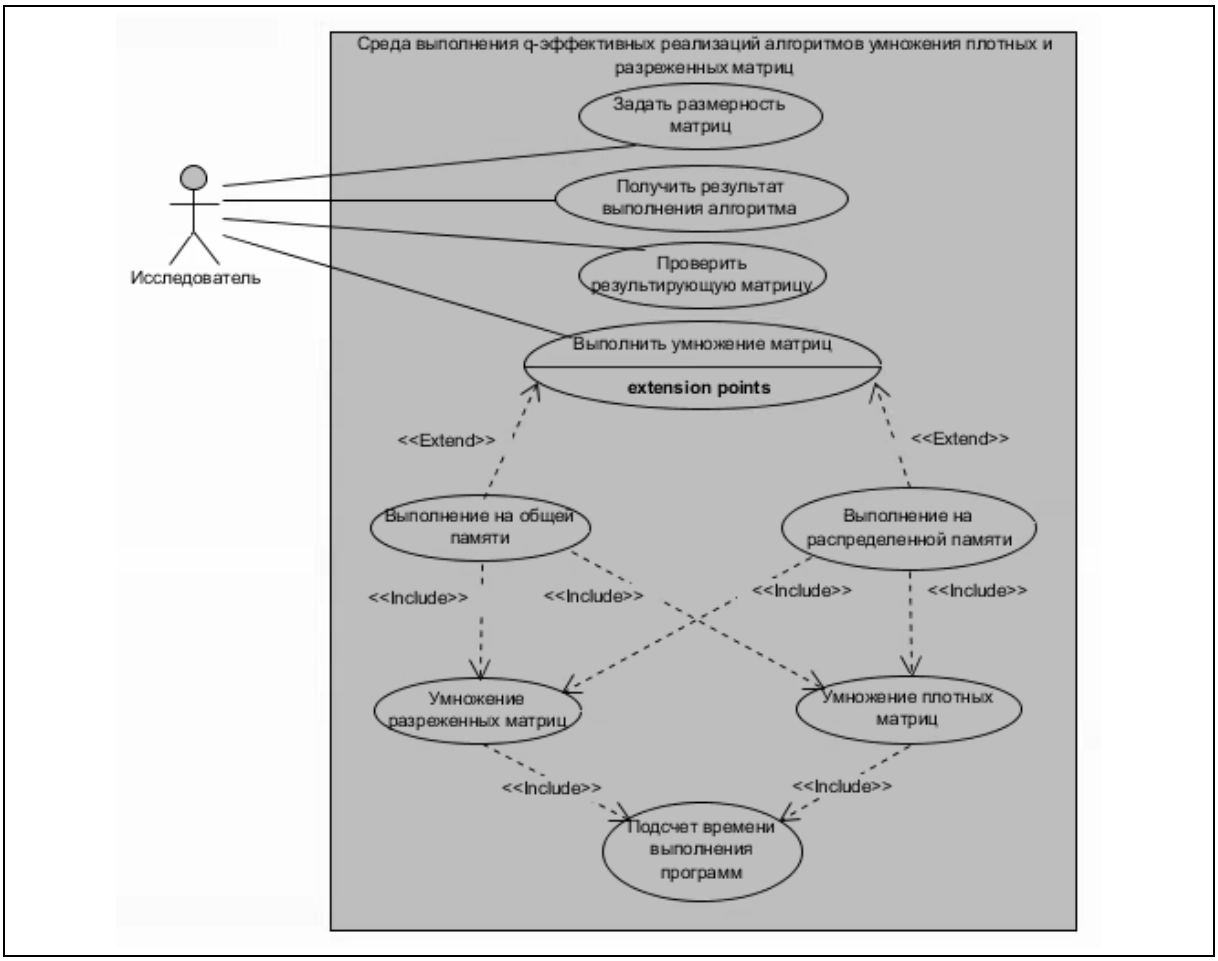

Рис. 3. Диаграмма вариантов использования

2.2. Модульная структура

В ходе проектирования система была разбита на модули. Модульная структура, отражающая архитектуру системы, представлена на рис. 4.

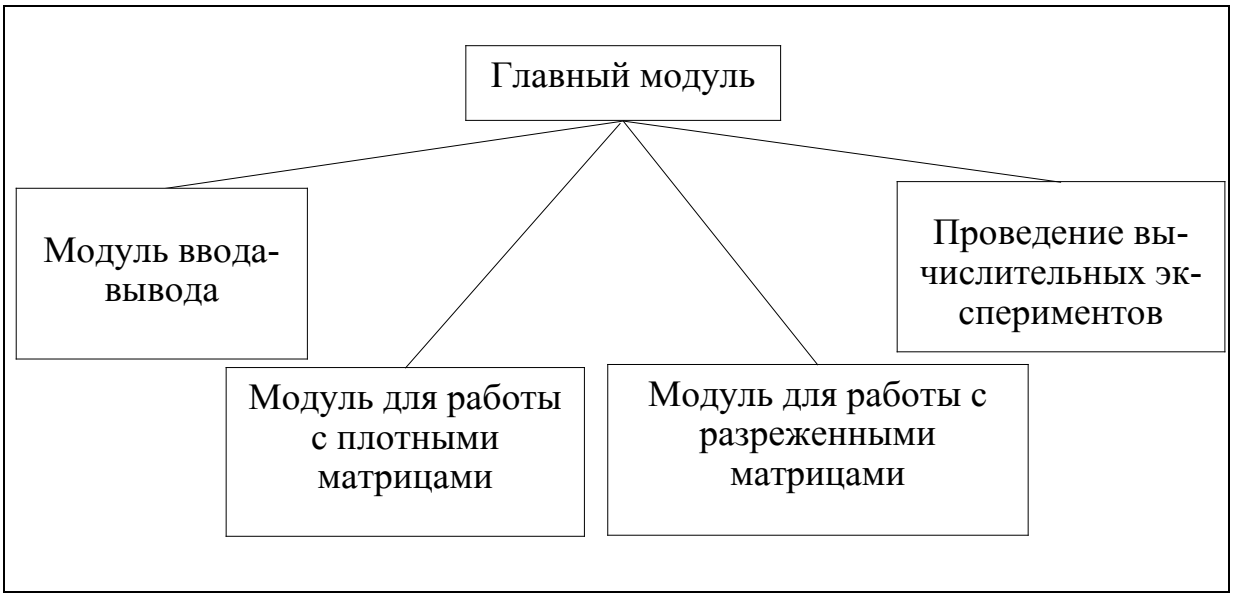

Рис. 4. Модульная структура системы

2.2.1. Главный модуль

Данный модуль отвечает за работу программы в целом. Опираясь на возможности описанных ниже модулей, он обеспечивает их совместную работу.

2.2.2. Модуль ввода-вывода

Ввод-вывод данных и взаимодействие с пользователем являются важными функциями любого ПО. Данный модуль реализует эти функции для разработанной системы и предоставляет к ней функциональный интерфейс. Поскольку особенности поставленной задачи не предполагают интерактивного взаимодействия с пользователем, единственной реализованной формой взаимодействия с ним является вывод информационных сообщений.

2.2.3. Модуль для работы с разреженными матрицами

Данный модуль генерирует, сравнивает, транспонирует, умножает матрицы, представленные в формате CSR.

2.2.4. Модуль для работы с плотными матрицами

Модуль осуществляет генерацию, сравнение, умножение плотных матриц.

2.2.5. Модуль для проведения вычислительных экспериментов

Модуль использует результаты, полученные в ходе вычислений для анализа и получения информации об эффективности алгоритмов, а также об ускорении при их выполнении.

3. РЕАЛИЗАЦИЯ

Чтобы реализовать *Q*-эффективную программу для вычисления произведения матриц, необходимо было сначала описать последовательный алгоритм, а затем разработать *Q*-эффективную реализацию алгоритма на основе представления в форме *Q*-детерминанта.

В ходе написания работы было разработано четыре *Q*-эффективных программы для двух *Q*-эффективных реализаций. Две, описанные в разделах 3.1.1 и 3.1.2, являются параллельными реализациями алгоритма умножения плотных матриц. Две, описанные в разделах 3.2.1 и 3.2.2, являются аналогичными реализациями алгоритма для умножения матриц, представленных в формате CSR.

В общем случае, умножение матриц проходит в несколько этапов:

1) получение куба промежуточных значений из всех возможных перемножений элементов исходных матриц;

2) попарное сложение по схеме сдваивания элементов куба, полученного на первом этапе.

3.1. Умножение плотных матриц

Произведение матриц *A* и *B* состоит из всех возможных комбинаций скалярных произведений вектор-строк матрицы *A* и вектор-столбцов матрицы *B*. Элемент матрицы *AB* с индексами *i, j* есть скалярное произведение *i*-ой вектор-строки матрицы *A* и *j*-го вектор-столбца матрицы *B*.

Последовательный алгоритм умножения матриц в ходе выполнения данной выпускной квалификационной работы будет использоваться в качестве эталонного при проверке корректности работы параллельных алгоритмов. На рис. 5 представлен листинг последовательной программы для умножения двух матриц на общей памяти

```
void Matrix::LinearMult()
{
    for (int i=0;i<size;i++)
        for (int j=0;j<size;j++)
            for (int k=0;k<size;k++)
                linear_result[i][j]+=A[i][k]*B[k][j];
}
```

Рис. 5. Листинг последовательной программы для умножения матриц

Для получения *Q*-детерминанта последовательный алгоритм был проанализирован на наличие атомарных (неделимых) операций, которые были разделены на независимые (операции умножения) и зависимые (операции сложения). Исходя из этого, для каждого элемента матрицы C_{ij} *Q*-терм должен иметь следующий вид.

1. Попарное перемножение элементов строки *i* матрицы *A* с соответствующими элементами столбца *j* матрицы *B* (шаг 1).

2. Сложение полученных произведений по схеме сдваивания (шаг 2, шаг 3).

Поскольку алгоритм вычисления одинаков для каждого элемента результирующей матрицы, Q-эффективная реализация будет состоять из i*j одинаковых Q-термов. На рис. 6 представлена Q-эффективная реализация Q-терма для вычисления элемента плотной матрицы C_{11}.

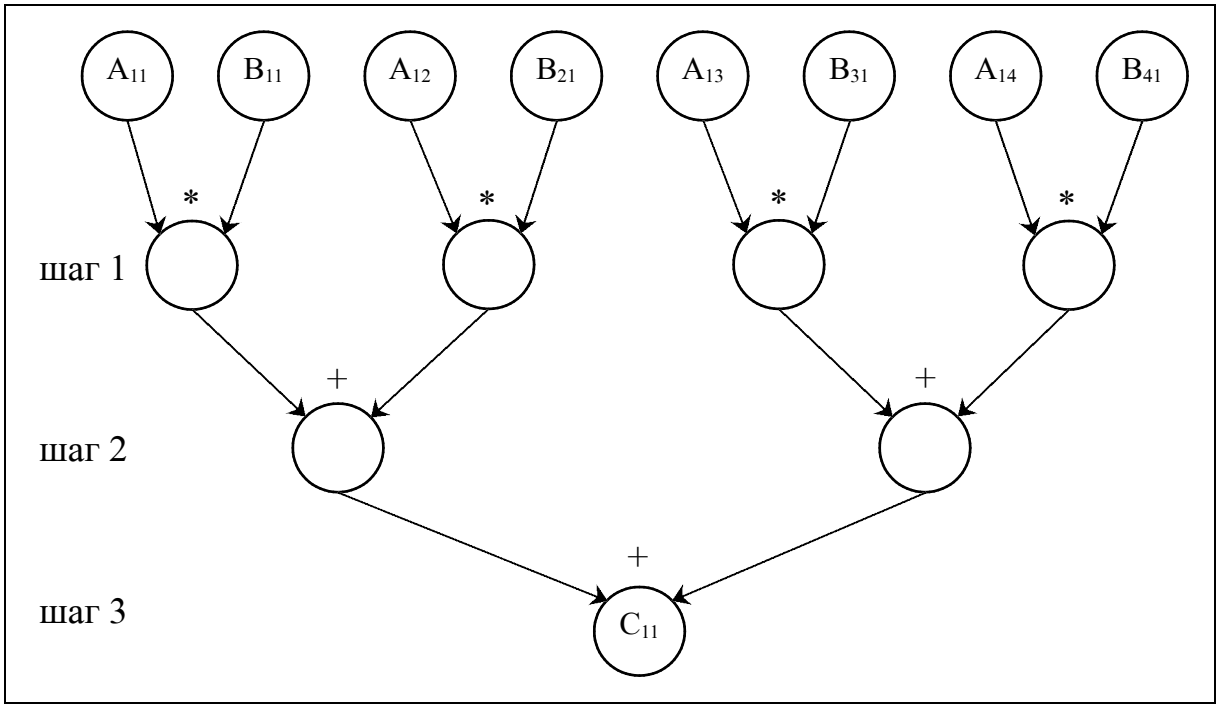

Рис. 6. *Q*-эффективная реализация *Q*-терма для вычисления элемента плотной матрицы C_{11}

3.1.1. *Q*-эффективная программа для архитектуры с общей памятью

Для работы с общей памятью использовалась технология OpenMP [8-9], весь процесс делился на 2 этапа:

1) получение куба промежуточных значений, в котором содержатся

все возможные произведения пар элементов исходных матриц;

2) сложение полученных произведений по схеме сдваивания.

В данном случае, *Q*-эффективная реализация накладывает ограничение на доступ к чтению промежуточных значений результирующей матрицы до тех пор, пока не будут выполнены все операции перемножения (сложения) предыдущих этапов. Это позволит гарантировать выполнение операций каждого *Q*-терма по мере готовности. Листинг кода Q-эффективной программы умножения матриц для архитектуры с общей памятью представлен на рис. 7.

```
#pragma omp parallel for private(i, j, k)
for (int i = 0; i < Size; i++)
    for (int j = 0; j < Size; j++)
        for (int k = 0; k < Size; k++)
            C[i][j][k] = A[i][k] * B[k][j];

int iterations = Size;
for (int g = 1; g < Size; g *= 2)
{
    iterations /= 2;
#pragma omp parallel for private (i, j, k, cell)
    for (int i = 0; i < Size; i++)
        for (int j = 0; j < Size; j++)
            for (int k = 0; k < iterations; k++)
            {
                int cell = g * k;
                C[i][j][cell] += C[i][j][cell + g];
                C[i][j][cell + g] = 0;
            }
}
```

Рис. 7. Листинг кода *Q*-эффективной программы для архитектуры с общей памятью

3.1.2. Q-эффективная программа для архитектуры с распределенной памятью

При выполнении на *N* вычислительных узлах матрицы *A* и *B* необходимо разделить. Поскольку вычислительная мощность узлов суперкомпьютера «Торнадо ЮУрГУ» одинакова, матрица будет разделена на *N* одинаковых полос. На рис. 8 представлен листинг метода, отвечающего за параллельное умножение матриц при распределенной памяти.

Из-за наличия распределенной памяти появляется необходимость обеспечения обмена данными между вычислительными узлами класте-

ра [4, 10]. Для этого мной была использована технология MPI. Таким образом, изменилась *Q*-эффективная программа, при этом *Q*-эффективная реализация алгоритма осталась прежней. Листинг кода Q-эффективной программы представлен на рис. 8.

```
double *tmpC = new double[Size * bufSize]
MPI_Scatter(A, bufSize, MPI_DOUBLE, bufA, bufSize, MPI_DOUBLE, 0,
                                                        MPI_COMM_WORLD);
MPI_Scatter(B, bufSize, MPI_DOUBLE, bufB, bufSize, MPI_DOUBLE, 0,
                                                        MPI_COMM_WORLD);

#pragma omp parallel for private(j,k)
for (int i = 0; i < part; i++)
    for (int j = 0; j < part; j++)
        for (int k = 0; k < Size; k++)
           tmpC[(i * Size + j) * Size + k] = bufA[i * Size + k] *
                                                    bufB[k * Size + j];
int iterations = Size;

for (int pwr = 1; pwr < Size; pwr *= 2)
{
    iterations /= 2;
#pragma omp parallel for private(j,k,cell)
    for (int i = 0; i < part; i++)
        for (int j = 0; j < part; j++)
            for (int k = 0; k < iterations; k++) {
                int cell = pwr * k;
                tmpC[(i * Size + j) * Size + cell] +=
                                    bufC[(i * Size + j) * Size + cell + pwr];
                if (iterations < 2)
                    bufC[i * Size + j] = tmpC[(i * Size + j) * Size];
                }
MPI_Gather(bufC, bufSize, MPI_DOUBLE, C, bufSize, MPI_DOUBLE, 0,
                                                        MPI_COMM_WORLD);
```

Рис. 8. Листинг кода Q-эффективной программы умножения плотных матриц для распределенной памяти

3.2. Умножение разреженных матриц в формате CSR

Для получения *Q*-детерминанта последовательный алгоритм был проанализирован на наличие атомарных (неделимых) операций, которые были разделены на независимые (операции умножения) и зависимые (операции сложения). Важно учесть, что из-за особенностей представления формата CSR, необходимо предварительно транспонировать матрицу *B*, иначе объем вычислений неоправданно возрастает.

Исходя из этого, для каждого элемента матрицы C_{ij} *Q*-терм должен выполняться следующим образом.

1) Получение списка *m* из перемножений всех возможных комбина-

ций элементов вектор-строки *a* матрицы *A* и вектор-столбца *b* матрицы *B*.

2) Получение списка значений *e*, в котором содержатся 1, если индексы элементов совпадают и 0 в противном случае.

3) Обновление списка промежуточных значений m, путем поэлементного перемножения со списком e.

4) Сложение элементов списка *m* по схеме сдваивания.

Поскольку результат выполнения п. 2 не зависит от результатов п. 1, целесообразно выполнить пункты 1 и 2 параллельно.

На рис. 9 схематично представлена Q-эффективная реализация Q-терма для вычисления элемента результирующей матрицы C. (При этом используются следующие обозначения: *a* - строка матрицы *A*, *b* - столбец матрицы *B*, C_{ab} - элемент результирующей матрицы *C*, являющийся произведением *a* и *b*.

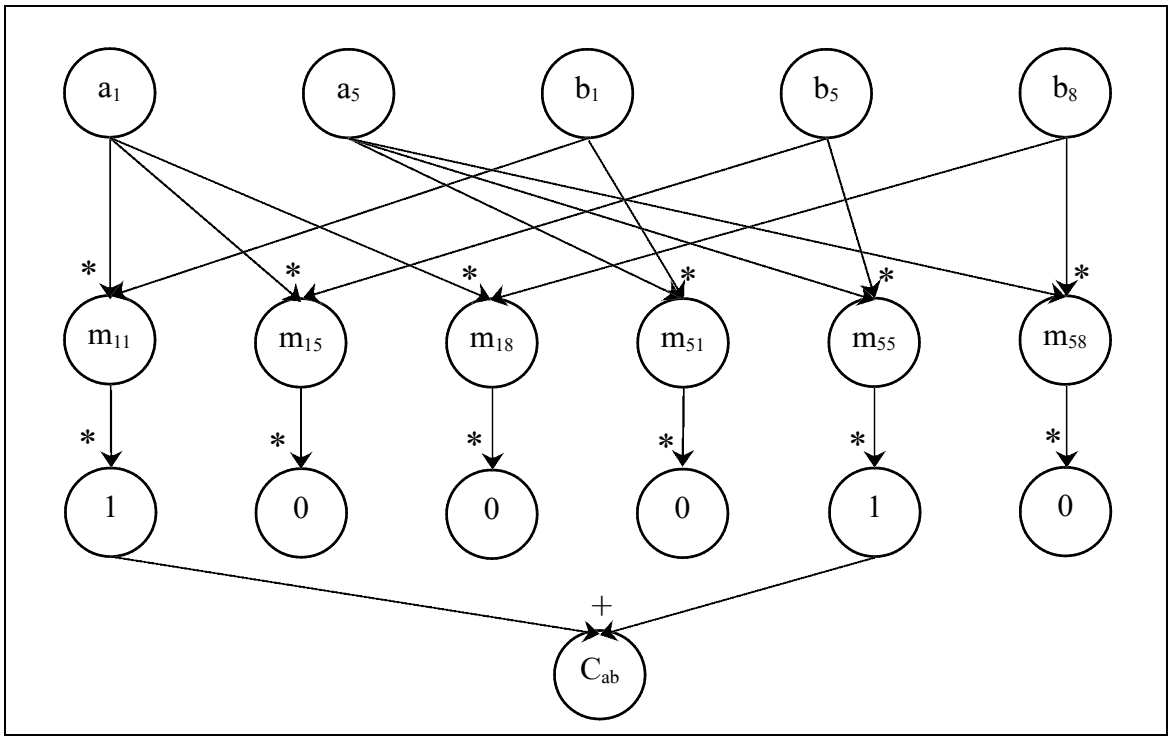

Рис. 9. Q-эффективная реализация Q-терма для вычисления одного элемента матрицы формата CSR

3.2.1. *Q*-эффективная программа для архитектуры с общей памятью

В данном случае, архитектура накладывает ограничение на доступ к чтению промежуточных значений результирующей матрицы до тех пор, пока не будут выполнены все операции предыдущих этапов. Это позволит

гарантировать выполнение всех зависимостей каждого *Q*-терма. На рис. 10 представлен листинг кода *Q*-эффективной программы умножения матриц формате CSR для общей памяти.

```
#pragma omp parallel for private (i, j, k)
for (int i = 0; i < Size-1; i++)
{
    int lower_bound_A = A[2][i];
    int upper_bound_A = A[2][i+1];
    int lower_bound_B = B[2][i];
    int upper_bound_B = B[2][i+1];
    if ((upper_bound_A-lower_bound_A != 0) & (upper_bound_B-
                                                lower_bound_B !=0))
        for (int j = lower_bound_A; j < upper_bound_A; j++)
            for (int k = lower_bound_B; k < upper_bound_B; k++)
                if (A[1][lower_bound_A+j] == B[1][lower_bound_B+k])
                    M[i][j].push_back(A[0][lower_bound_A+j]*
                                                B[0][lower_bound_B+k]);
}

for (int g = 1; g < Size; g *= 2)
{
#pragma omp parallel for private (i, j, k, cell)
    for (int i = 0; i < Size; i++)
        for (int j = 0; j < Size; j++)
            for (int k = 0; k < M[i][j].size(); k++)
            {
                int cell = g * k;
                M[i][j][cell] += M[i][j][cell + g];
                M[i][j][cell + g] = 0;
            }
}

for (int i = 0; i < Size; i++)
{
    for (int j = 0;  j < Size; j++)
        if (M[i][j] != 0) {
            C[1].push_back(j);
            C[0].push_back(M[i][j]);
        }
        C[2].push_back(Size);
}
```

Рис. 10. Листинг кода *Q*-эффективной программы умножения матриц в формате CSR для общей памяти

3.2.2. *Q*-эффективная программа для архитектуры с распределенной памятью

При использовании архитектуры с распределенной памятью налагаются ограничения, аналогичные п. 3.1.2. Листинг кода Q-эффективной программы представлен на рис. 11.

```
MPI_Bcast(A, Size , MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(B, Size , MPI_DOUBLE, 0, MPI_COMM_WORLD);

int lower_bound = rank * ( Size / num_worker);
int upper_bound = ((rank + 1) * ( Size / num_worker))-1;

#pragma omp parallel for private (i, j, k)
for (int i = lower_bound; i < upper_bound; i++)
{
    int lower_bound_A = A[2][i];
    int upper_bound_A = A[2][i+1];
    int lower_bound_B = B[2][i];
    int upper_bound_B = B[2][i+1];
    if ((upper_bound_A-lower_bound_A != 0) & (upper_bound_B-
                                                lower_bound_B !=0))
        for (int j = lower_bound_A; j < upper_bound_A; j++)
            for (int k = lower_bound_B; k < upper_bound_B; k++)
                if (A[1][lower_bound_A+j] == B[1][lower_bound_B+k])
                    M[i][j].push_back(A[0][lower_bound_A+j]*
                                                B[0][lower_bound_B+k]);
}

for (int g = 1; g < Size; g *= 2)
{
#pragma omp parallel for private (i, j, k, cell)
    for (int i = 0; i < Size; i++)
        for (int j = 0; j < Size; j++)
            for (int k = 0; k < M[i][j].size(); k++)
            {
                int cell = g * k;
                M[i][j][cell] += M[i][j][cell + g];
                M[i][j][cell + g] = 0;
            }
}

for (int i = 0; i < Size; i++)
{
    for (int j = 0;  j < Size; j++)
        if (M[i][j] != 0) {
            C[1].push_back(j);
            C[0].push_back(M[i][j]);
        }
        C[2].push_back(Size);
}

       MPI_Gather(C, Size, MPI_DOUBLE,
           result_matrix, Size,  MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Рис. 11. Листинг кода Q-эффективной программы для умножения разреженных матриц в формате CSR на распределенной памяти

3.3. Тестирование

Для тестирования программы был реализован метод, который вычисляет результирующую матрицу эталонным последовательным алгоритмом и поэлементно сравнивает полученную матрицу C с результатами Q-эффективных программм с точностью 10^{-6}. Метод возвращает только количество несовпадающих элементов, так как при больших размерностях матриц возможно засорение потока вывода ненужными данными.

4. ВЫЧИСЛИТЕЛЬНЫЕ ЭКСПЕРИМЕНТЫ

Вычислительные эксперименты были проведены на суперкомпьютере «Торнадо ЮУрГУ» на ресурсах до 30 вычислительных узлов включительно. Оценивалось время выполнения *Q*-эффективных программ, их ускорение и эффективность [2, 3]. Для каждого вычислительного эксперимента генерировались матрицы *A* и *B*, заполненные случайными числами. Доля ненулевых элементов в разреженных матрицах (плотность) была принята за 20 %.

Время выполнения последовательной версии алгоритма обозначим как T_1, время выполнения параллельного алгоритма на p одинаковых процессорах - T_p. Ускорением параллельного алгоритма называют отношение времени выполнения последовательного алгоритма к времени выполнения параллельного алгоритма:

$$S = \frac{T_1}{T_p}. \quad (1)$$

Для оценки масштабируемости параллельного алгоритма используется понятие эффективности:

$$E = \frac{S}{p}. \quad (2)$$

Параллельный алгоритм называется масштабируемым, если при росте числа процессоров он обеспечивает увеличение ускорения при сохранении постоянного уровня эффективности использования процессоров [8].

4.1. Проведение экспериментов

Выполнение программы на суперкомпьютере «Торнадо ЮУрГУ» начинается с постановки задачи в очередь. Для каждой поставленной задачи указано количество узлов, на которых она выполняется [16].

Для упрощения проведения вычислительных экспериментов были написаны скрипты на языке Bash [17], позволяющие автоматизировать запуск задач и сбор результатов на суперкомпьютере. Результатом работы скриптов являются текстовые файлы, содержащие форматированный вывод программ. На рис. 12 представлен пример скрипта для выполнения экспериментов для архитектуры с разреженной памятью и сохранения ре-

зультатов в файл с датой и временем запуска задачи.

```
parameters=(1000 2000 3000 5000 10000 12000 15000)
nodes=(2 5 10 20 30)

mpicc –openmp mpi_multiplication.cpp –o mpi.exec

filename=‘result_mpi-‘date +%d-%m-%Y\ %H:%M‘‘
echo ‘Nodes \\t Size \\t Time \\t Acceleration \\t Efficiency \\t Errors‘
                                                                    > $filename

for parameter in $parameters;
do
  for node in $nodes;
    do

      sbatch –N $node –p quick mpirun mpi.exec $parameter >> $filename;
    done;
done
```

Рис. 12. Листинг скрипта, отвечающего за проведение вычислительных экспериментов и сбор результатов.

На рис. 13 представлен пример вывода результатов экспериментов для размерностей 500*500 и 1000*1000 на 2 и 5 узлах.

```
Nodes   Size    Time    Acceleration    Efficiency    Errors
2       500     0.03002 1.8487675       92.43837      0
2       1000    0.22398 1.8742745       93.71372      0
5       500     0.01860 3.0833333       61.66667      0
5       1000    0.13655 3.0743317       61.8663       0
```

Рис. 13. Пример вывода результатов экспериментов.

Далее представлены графики, отражающие зависимости ускорения от объема вычислений. а также эффективности работы алгоритмов в зависимости от количества вычислителей.

4.2. Результаты экспериментов

На рис. 14 и 15 представлены графики ускорения параллельных алгоритмов для плотных и разреженных матриц на общей и распределенной памяти. Видна линейная зависимость ускорения от количества ядер на обоих алгоритмах. Ускорение *Q*-эффективной программы умножения разреженных матриц растет медленнее из-за наличия дополнительного шага сравнения индексов элементов, описанного в п. 3.2, а также из-за формирования результирующей матрицы в формате CSR. Оба действия требуют до-

полнительного времени работы.

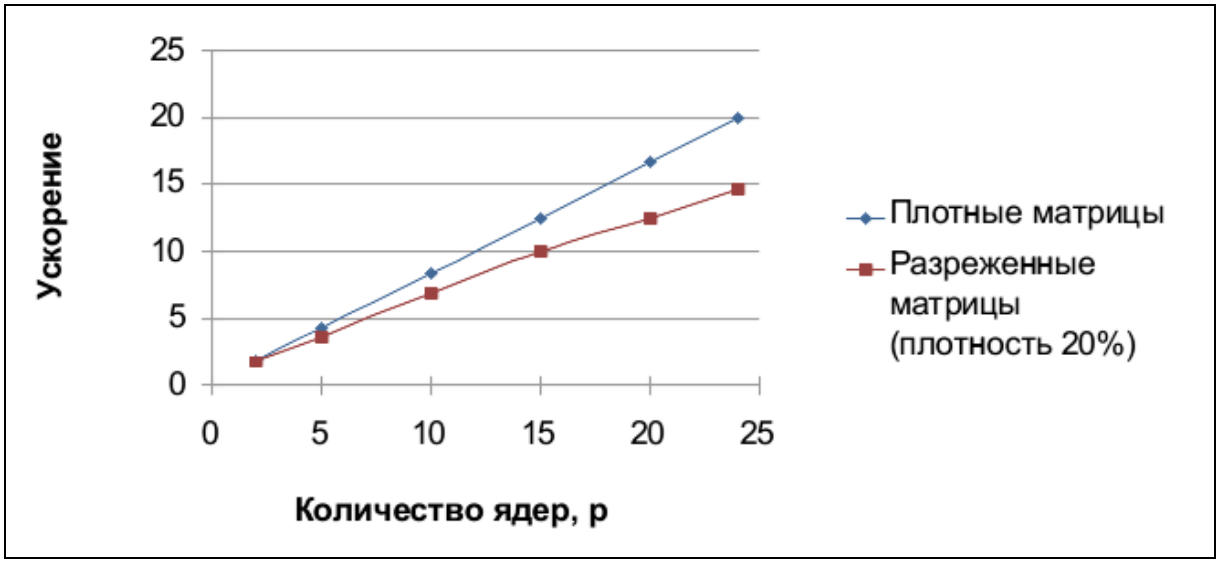

Рис. 14. Ускорение *Q*-эффективных программ умножения матриц для общей памяти

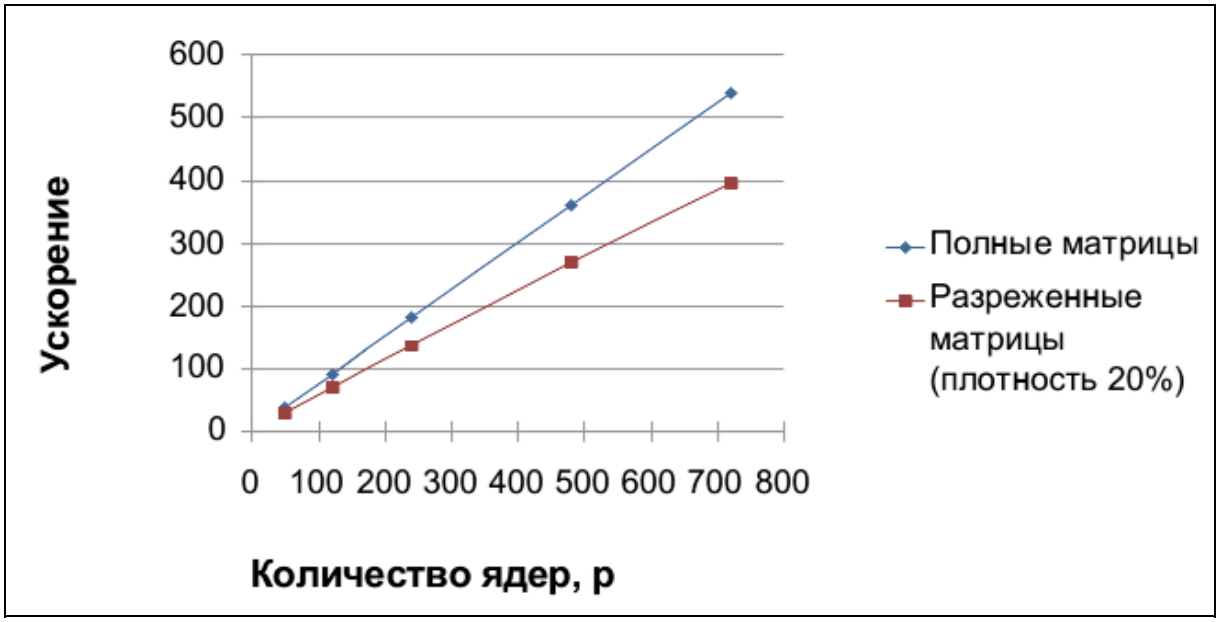

Рис. 15. Ускорение *Q*-эффективных программ умножения матриц на распределенной памяти

На рис. 16 и 17 представлена эффективность параллельных алгоритмов для общей и распределенной памяти. Графики показывают, что эффективность ассимпотически стремится к значениям в 75 % для алгоритма умножения плотных матриц и 55 % для алгоритма умножения разреженных матриц.

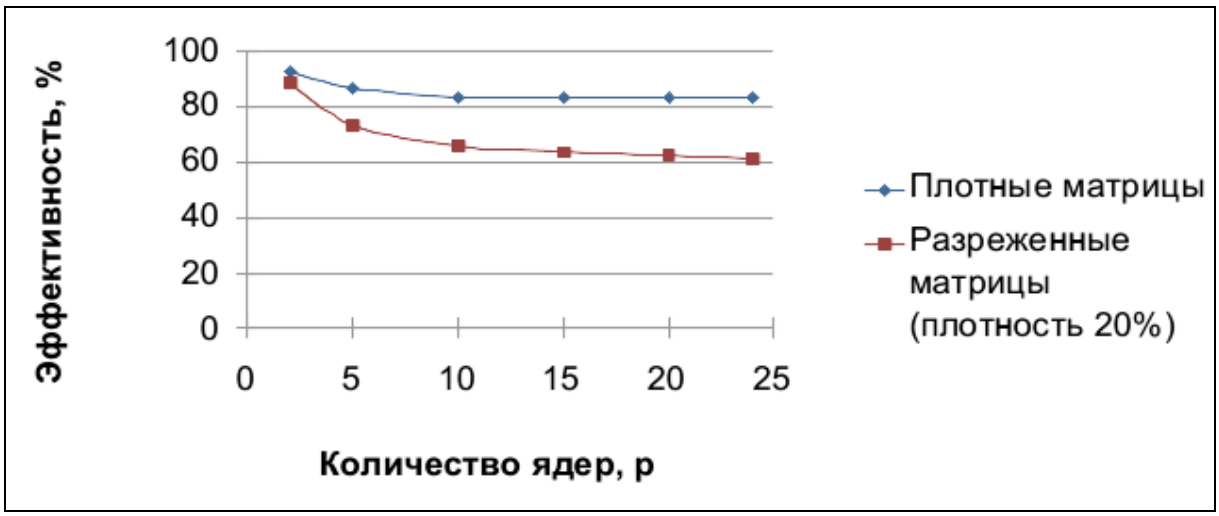

Рис. 16. Эффективность *Q*-эффективных программ умножения матриц на общей памяти

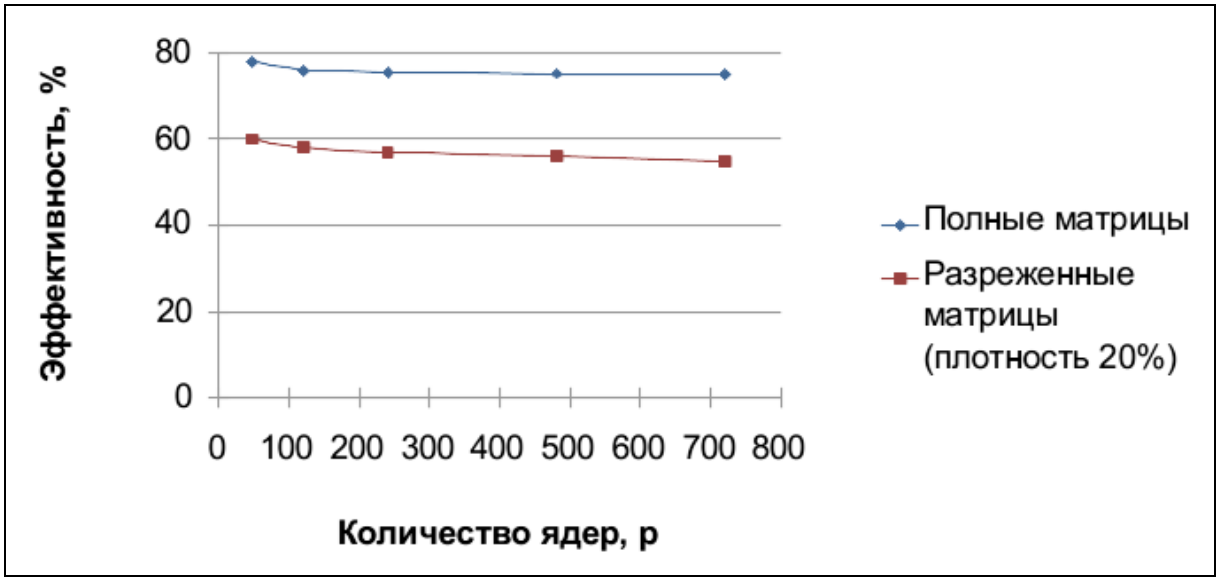

Рис. 17. Эффективность *Q*-эффективных программ умножения матриц на распределенной памяти

Исходя из графиков, можно сделать следующие выводы:

1) реализованные Q-эффективные программы являются масштабируемыми, т.к. при увеличении количества вычислительных ядер график эффективности стремится к постоянному значению, а ускорение растет;

2) для увеличения ускорения и эффективности реализованных программ необходимо учитывать больше параметров архитектуры.

ЗАКЛЮЧЕНИЕ

В ходе проделанной работы были решены следующие задачи.

1. Изучен подход к построению *Q*-эффективной реализации, основанной на представлении алгоритмов в форме *Q*-детерминанта.
2. Изучена архитектуры суперкомпьютера «Торнадо ЮУрГУ»
3. Изучены программные средства для выполнения параллельных реализаций алгоритмов на суперкомпьютере «Торнадо ЮУрГУ».
4. Выполнено программирование *Q*-эффективных реализаций алгоритмов умножения плотных и разреженных матриц на суперкомпьютере «Торнадо ЮУрГУ» с использованием различных программных средств на общей и распределенной памяти.
5. Проведены вычислительные экспериментов на различных вычислительных мощностях и объемах данных.
6. Произведен анализ ускорения быстродействия параллельных алгоритмов в сравнении с последовательными.
7. Произведен анализ эффективности выполнения *Q*-эффективных реализаций.

Дальнейшим направлением развития будет увеличение эффективности и ускорения работы параллельных алгоритмов.

Данная работа выполнена при поддержке Российского Фонда Фундаментальных Исследований, грант № 17-07-00865.

В рамках работы были опубликованы 2 стендовых доклада на международной конференции.

1. Алеева В.Н., Валькевич Н.В., Лаптева Ю.С., Тарасов Д.Е. Разработка принципов выполнения Q-эффективных реализаций численных алгоритмов на параллельных вычислительных системах // Параллельные вычислительные технологии – XI международная конференция, ПаВТ’2017, г. Казань, 3–7 апреля 2017 г. Короткие статьи и описания плакатов. Челябинск: Издательский центр ЮУрГУ, 2017. С. 503.
2. Алеева В.Н., Багаутдинов А.Р., Валькевич Н.В. Исследование ресурса параллелизма численных алгоритмов на основе концепции Q-детер-

минанта // Параллельные вычислительные технологии – XI международная конференция, ПаВТ’2017, г. Казань, 3–7 апреля 2017 г. Короткие статьи и описания плакатов. Челябинск: Издательский центр ЮУрГУ, 2017. С. 504.

ЛИТЕРАТУРА

1. Aleeva V.N., Sharabura I.S., Suleymanov D.E. Software System for Maximal Parallelization of Algorithms on the Base of the Conception of Q-determinant. // 13th International Conference on Parallel Computing Technologies (PaCT-2015), Petrozavodsk, Russian Federation, 31 August - 4 September 2015, Proceedings. Lecture Notes in Computer Science, Vol. 9251. Springer, 2015 – P. 3-9.

2. Coppersmith D. Rectangular matrix multiplication revisited. // Journal of Complexity, 1997. – P. 14:42-49.

3. Huang X. Fast rectangular matrix multiplications and applications. // Journal of Complexity, 1998. - P. 25:257-299.

4. Hempel R., Walker D.W. The emergence of the MPI message passing standard forparallel computing. // Computer Standards & Interfaces, 25 May 1999. – V. 21. – Issue 1. - P. 11:25-31.

5. Barrett R.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. / Berry M., Chan T.F., Demmel J., Donato J.M., Dongarra J., Eijkhout V., Pozo R., Romine C., Van der Vorst H. // Society for Industrial and Applied Mathematics, 2005. – P. 18:64 – 194

6. Алеева В.Н. Анализ параллельных численных алгоритмов / В.Н. Алеева. – Новосибирск: Препринт ВЦ СО АН СССР, 1985. – № 590. – 23 с.

7. Боресков А.В., Харламов А.А. Параллельные вычисления на GPU. Архитектура и программная модель CUDA. Учебное пособие. – Издательство Московского университета, 2012 – 54 с.

8. Гергель В.П. Теория и практика параллельных вычислений. Учебное пособие. – ИНТУИТ, Бином. Лаборатория знаний, 2007. – 40 с.

9. Левин М.П. Параллельное программирование с использование OpenMP. Учебное пособие. – М.: БИНОМ. Лаборатория знаний, Интернет-Университет Информационных Технологий (ИНТУИТ), 2008, – 120 с.

10. Мееров И.Б., Сысоев А.В. Разреженное матричное умножение. Учебное пособие. – М.: Нижний Новгород, 2011. – 81 с.

11. Официальный сайт проекта CUDA. [Электронный ресурс] URL: http://www.nvidia.com/object/cuda_home_new.html (дата обращения: 03.04.2017).

12. Официальный сайт проекта MPI. [Электронный ресурс] URL: https://www.open-mpi.org/ (дата обращения: 18.03.2017).

13. Официальный сайт проекта OpenMP. [Электронный ресурс] URL: http://www.openmp.org (дата обращения: 15.03.2017).

14. Свирихин Д.И., Алеева В.Н. Определение максимально эффективной реализации алгоритма на основе концепции Q-детерминанта. // Параллельные вычислительные технологии (ПаВТ'2013): труды международной научной конференции (1–5 апреля 2013 г., г. Челябинск). Челябинск: Издательский центр ЮУрГУ, 2013. – С. 617.

15. Сулейманов Д.Э., Алеева В.Н., Шарабура И.С. Разработка программной системы QStudio для получения максимально быстрой реализации алгоритма. // Параллельные вычислительные технологии (ПаВТ'2015): труды международной научной конференции (31 марта - 2 апреля 2015 г., г. Екатеринбург). Челябинск: Издательский центр ЮУрГУ, 2015. – С. 523.

16. Пользовательская документация суперкомпьютера “Торнадо ЮУрГУ”. [Электронный ресурс] URL: http://supercomputer.susu.ru/users/support/ (дата обращения 6.03.2017)

17. Техническая документация к языку Bash. [Электронный ресурс] URL: http://www.gnu.org/software/bash/manual/ (дата обращения 5.03.2017)