МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ 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

ДОПУСТИТЬ К ЗАЩИТЕ

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

_____ Л.Б. Соколинский

«___»_____ 2024 г.

Разработка Q-эффективных программ для реализации алгоритма сортировки Шелла на общей и распределенной памяти кластерной вычислительной системы

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
ЮУрГУ – 02.03.02.2024.308-597.ВКР

Научный руководитель,
доцент кафедры СП, к.ф.-м.н.
_____ В.Н. Алеева

Автор работы,
студент группы КЭ-401
_____ Е.К. Кузнецов

Ученый секретарь
(нормоконтролер)
_____ И.Д. Володченко
«___»_____ 2024 г.

Челябинск, 2024 г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ 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

УТВЕРЖДАЮ
Зав. кафедрой СП
_____ Л.Б. Соколинский
29.01.2024 г.

ЗАДАНИЕ
на выполнение выпускной квалификационной работы бакалавра
студенту группы КЭ-401
Кузнецову Егору Константиновичу,
обучающемуся по направлению
02.03.02 «Фундаментальная информатика и информационные технологии»

1. **Тема работы** (утверждена приказом ректора от 24.04.2024 г. № 764-13/12)
Разработка Q-эффективных программ для реализации алгоритма сортировки Шелла на общей и распределенной памяти кластерной вычислительной системы.
2. **Срок сдачи студентом законченной работы:** 03.06.2024 г.
3. **Исходные данные к работе**

3.1. Алеева В.Н., Шатов М.Б. Применение концепции Q-детерминанта для эффективной реализации численных алгоритмов на примере метода сопряженных градиентов для решения систем линейных уравнений. // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика, 2021. – Т. 10, № 3. – С. 56–71.
3.2. Уильямс Э. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. – М.: ДМК Пресс, 2012. – 672 с.
3.3. Антонов А.С. Технологии параллельного программирования MPI и OpenMP: учеб. пособие. – М.: Издательство Московского университета, 2012. – 344 с.
3.4. Кнут Д. Искусство программирования. // Сортировка и поиск, 2-е изд: Пер. с англ. – Т. 3. – М.: ООО «И. Д. Вильямс», 2018. – 832 с.

4. Перечень подлежащих разработке вопросов

4.1. Провести обзор научной литературы, выполнить анализ предметной области.

4.2. Разработать Q-эффективные программы для реализации алгоритма сортировки Шелла на общей и распределенной памяти кластерной вычислительной системы.

4.3. Провести тестирование разработанных программ.

4.4. Оценить динамические характеристики разработанных программ.

5. Дата выдачи задания: 29.01.2024 г.

Научный руководитель,
доцент кафедры СП, к.ф.-м.н. В.Н. Алеева

Задание принял к исполнению Е.К. Кузнецов

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность

Сортировка является одной из самых распространенных задач в программировании. Для ее ускорения и оптимизации применяются различные алгоритмы и их реализации. Главной целью этой работы является разработка программ, выполняющих Q-эффективную реализацию алгоритма сортировки Шелла на общей и распределенной памяти кластерной вычислительной системы для дальнейшей оценки динамических характеристик реализованных в программах алгоритмов сортировки. Q-эффективная реализация алгоритма сортировки должна быть самой быстродейственной в сравнении с остальными реализациями того же алгоритма. Q-эффективная программа полностью использует ресурс параллелизма алгоритма, поскольку выполняет Q-эффективную реализацию. Другими словами, Q-эффективная программа одновременно выполняет наибольшее количество операций среди программ, реализующих этот алгоритм. Данная работа позволит определить целесообразность применения метода проектирования Q-эффективных программ для алгоритмов сортировки [1].

Постановка задачи

Целью выпускной квалификационной работы является разработка Q-эффективных программ для реализации алгоритма сортировки Шелла на общей и распределенной памяти, проведение вычислительных экспериментов с разработанными программами, оценка их динамических характеристик. Для достижения поставленной цели необходимо решить следующие задачи.

1. Разработать Q-эффективную программу для реализации алгоритма сортировки Шелла на общей памяти.
2. Разработать Q-эффективную программу для реализации алгоритма сортировки Шелла на распределенной памяти.
3. Провести функциональное тестирование разработанных программ.
4. Провести вычислительные эксперименты.

5. Оценить динамические характеристики разработанных программ.

Структура и содержание работы

Работа состоит из введения, четырех глав, заключения и списка литературы. Объем работы составляет 41 страницу, объем списка литературы – 17 источников.

В первой главе проводится анализ предметной области.

Вторая глава посвящена проектированию разрабатываемой программы. Выявляются основные требования к разрабатываемым Q-эффективным программам.

В третьей главе были описаны программные средства разработки программы для разработки программ, реализующих Q-эффективную реализацию на общей и распределенной памяти алгоритма сортировки Шелла, а также приведены детали разработки и листинги.

Четвертая глава посвящена функциональному тестированию программы, вычислительным экспериментам и оценке динамических характеристик разработанных программ.

В приложении А содержится Q-детерминант алгоритма сортировки Шелла для массива из четырех элементов.

В приложении Б содержатся результаты функционального тестирования программ.

В приложении В содержатся результаты вычислительных экспериментов.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Описание предметной области

Основной задачей является разработка программ для Q-эффективной реализации алгоритма сортировки Шелла на общей и распределенной памяти и анализ их динамических характеристик. Q-эффективная реализация алгоритма должна быть выполнима, то есть одновременно должно выполняться конечное множество операций, она предполагает полное использование ресурсов параллелизма алгоритма [4].

Для реализации поставленной задачи будет использован метод проектирования Q-эффективных программ [2]. Он заключается в том, что можно разработать параллельную программу, используя Q-детерминант алгоритма.

Для данного метода используются следующие аргументы.

1. Q-детерминант можно построить для любого численного алгоритма.

2. Используя Q-детерминант алгоритма, можно описать его Q-эффективную реализацию.

3. Если Q-эффективная реализация алгоритма выполнима, то можно разработать программный код для ее выполнения.

Сам метод состоит из трех этапов:

1) построение Q-детерминанта алгоритма;

2) описание Q-эффективной реализации алгоритма;

3) разработка параллельной программы выполнимой Q-эффективной реализации алгоритма.

Программа называется Q-эффективной, если она разработана с помощью этого метода. Кроме того, процесс разработки Q-эффективной программы будет называться Q-эффективным программированием [3].

Q-детерминант любого алгоритма сортировки обладает следующей особенностью: его логические Q-термы очень комплексные и содержат в себе все возможные сравнения элементов сортируемого массива, а стоящие

с ними в паре Q-термы содержат только один элемент – перестановка элементов массива. В приложении А приведен Q-детерминант алгоритма сортировки Шелла для массива из четырех элементов, используемые в рамках курсовой работы, выполненной в 2023 году, на тему: «Применение метода проектирования Q-эффективных программ для алгоритма сортировки Шелла». По этой причине Q-детерминант для Q-эффективных программ, реализующих алгоритм сортировки Шелла, не строился, но программы были разработаны таким образом, чтобы все операции, выполняемые во время сортировки, выполнялись сразу по мере их готовности. Такая программа также может называться Q-эффективной [5].

Q-эффективная программа полностью использует ресурс параллелизма алгоритма, поскольку выполняет Q-эффективную реализацию алгоритма. Таким образом, у Q-эффективной программы самый высокий параллелизм среди программ, реализующих алгоритм. Другими словами, Q-эффективная программа одновременно выполняет наибольшее количество операций среди программ, реализующих этот алгоритм. Это означает, что Q-эффективная программа использует больше вычислителей вычислительной системы (например, ядер, процессоров), чем программы, выполняющие другие реализации алгоритма.

Для сравнительной оценки динамических характеристик Q-эффективной реализации алгоритма сортировки Шелла на распределенной памяти будет использована Q-эффективная реализация алгоритма сортировки Шелла на общей памяти, а также последовательная реализация алгоритма сортировки Шелла.

Сортировка Шелла – алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода заключается в сравнении разделенных на группы элементов последовательности, находящихся друг от друга на некотором расстоянии. Изначально это расстояние

равно d или $\frac{N}{2}$, где N – общее число элементов последовательности. На первом шаге каждая группа содержит два элемента, расположенных друг от друга на расстоянии $\frac{N}{2}$, они сравниваются между собой и, в случае необходимости, меняются местами. На последующих шагах также происходят проверка и обмен, но расстояние d сокращается на $\frac{d}{2}$, и количество групп, соответственно, уменьшается. Постепенно расстояние между элементами уменьшается, и на $d = 1$ проход по последовательности происходит в последний раз.

Среднее время работы алгоритма зависит от длин промежутков d, на которых будут находиться сортируемые элементы исходной последовательности длины N на каждом шаге алгоритма. Существуют следующие подходы к выбору этих значений [10].

1. Последовательность Шелла (формула 1):

$$d_1 = \frac{N}{2}, d_i = \frac{d_{i-1}}{2}, d_k = 1. \quad (1)$$

В худшем случае сложность алгоритма составит ($O(N^2)$).

2. Последовательность Хиббарда (формула 2):

$$2^i - 1 \leq N, i \in N. \quad (2)$$

Сложность алгоритма составит ($O(N^{\frac{3}{2}})$).

3. Последовательность Седжвика (формула 3):

$$d_i = 9 \times 2^i - 9 \times 2^{\frac{2}{i}} + 1, \textit{если i четное,} \quad (3)$$

$$\textit{и } d_i = 8 \times 2^i - 6 \times 2^{\frac{i+1}{2}} + 1, \textit{если i нечетное.}$$

Средняя сложность алгоритма составит ($O(N^{\frac{7}{6}})$), худшая – ($O(N^{\frac{4}{3}})$).

4. Последовательность Пратта (формула 4):

$$2^i \times 3^j \leq \frac{N}{2}, i, j \in N. \quad (4)$$

Сложность алгоритма составит $O(N(\log N)^2)$.

5. Эмпирическая последовательность Марцина Циура: A102549 в OEIS. Считается лучшей последовательностью для сортировки Шелла.

1.2. Сравнительный анализ аналогов

На текущий момент программ для Q-эффективной реализации на общей или распределенной памяти кластерной вычислительной системы любого из алгоритмов сортировки не было реализовано. Однако в рамках, выполненной в 2023 году, курсовой работы на тему: «Применение метода проектирования Q-эффективных программ для алгоритма сортировки Шелла» была разработана программа, реализующая параллельную Q-эффективную реализацию алгоритма сортировки Шелла на общей памяти персональной вычислительной машины. Для реализованной программы было зафиксировано среднее время выполнения алгоритма сортировки и вычислено ускорение реализации в сравнении с последовательной сортировкой алгоритмом сортировки Шелла. Результаты оценки временных характеристик приведены в таблицах 1–3.

Таблица 1 – Время работы и ускорение реализаций алгоритма сортировки для случайных массивов

Реализация	**Длина массива**					
	10000	100000	300000	500000	700000	1000000
Параллельная	0,00039	0,0056	0,02487	0,04292	0,07461	0,10115
Последовательная	0,00043	0,00659	0,02566	0,0566	0,09858	0,13471
Ускорение	1,1025641	1,1767857	1,0317651	1,3187325	1,3212706	1,3317844

Исходя из данных таблицы 1, ускорение параллельного алгоритма для обратных массивов длиной от 10000 до 1000000 увеличивается от ~10% до ~33%.

Таблица 2 – Время работы и ускорение реализаций алгоритма сортировки для обратных массивов

Реализация	**Длина массива**					
	10000	100000	300000	500000	700000	1000000
Параллельная	0,00137	0,01555	0,05379	0,08397	0,12821	0,19998
Последовательная	0,00137	0,01628	0,05731	0,0968	0,16461	0,22893
Ускорение	1	1,046945338	1,065439673	1,152792664	1,283909211	1,14476447

Исходя из данных таблицы 2, ускорение параллельного алгоритма для частично упорядоченных массивов длиной от 10000 до 1000000 колеблется между 0% и ~28%.

Таблица 3 – Время работы и ускорение реализаций алгоритма сортировки для частично упорядоченных массивов

Реализация	**Длина массива**					
	10000	100000	300000	500000	700000	1000000
Параллельная	0,00039	0,0056	0,02487	0,04292	0,07461	0,10115
Последовательная	0,00043	0,00659	0,02566	0,0566	0,09858	0,13471
Ускорение	1,1025641	1,1767857	1,0317651	1,3187325	1,3212706	1,3317844

Исходя из данных таблицы 3, ускорение параллельного алгоритма для обратных массивов длиной от 10000 до 1000000 увеличивается от ~10% до ~33%.

1.3. Обзор инструментов для реализации

Разработка программ для Q-эффективной реализации алгоритма сортировки Шелла будет осуществляться с использованием языка программирования C++. Для реализации на распределенной памяти кластерной вычислительной системы были применены технологии MPI и OpenMP, а для общей памяти кластерной вычислительной системы – только технология OpenMP.

MPI – программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. MPI разработан Уильямом Гроуппом, Эвином Ласком и другими.

MPI является наиболее распространенным стандартом интерфейса обмена данными в параллельном программировании, существуют его реализации для большого числа компьютерных платформ. Используется при разработке программ для кластеров и суперкомпьютеров. Основным средством

коммуникации между процессами в MPI является передача сообщений друг другу [14].

Стандартизацией MPI занимается MPI Forum. В стандарте MPI описан интерфейс передачи сообщений, который должен поддерживаться как на платформе, так и в приложениях пользователя [13]. В настоящее время существует большое количество бесплатных и коммерческих реализаций MPI [15]. Существуют реализации для языков Фортран 77/90, Java, C и C++.

OpenMP – открытый стандарт для распараллеливания программ на языках C, C++ и Fortran [16]. Он дает описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью. В OpenMP используется модель параллельного выполнения «ветвление-слияние». Программа OpenMP начинается как единственный поток выполнения, называемый начальным потоком. Когда поток встречает параллельную конструкцию, он создает новую группу потоков, состоящую из себя и некоторого числа дополнительных потоков, и становится главным в новой группе. Все члены новой группы (включая главный) выполняют код внутри параллельной конструкции. В конце параллельной конструкции имеется неявный барьер. После параллельной конструкции выполнение пользовательского кода продолжает только главный поток. В параллельный регион могут быть вложены другие параллельные регионы, в которых каждый поток первоначального региона становится основным для своей группы потоков. Вложенные регионы могут в свою очередь включать регионы более глубокого уровня вложенности.

Выводы по первой главе

В этой главе были рассмотрены предметная область и инструменты реализации проекта. Целью работы является разработка программ для такой реализации на общей и распределенной памяти кластерной системы, а также оценка временных характеристик таких программ.

2. ПРОЕКТИРОВАНИЕ

2.1. Требования к разрабатываемым Q-эффективным программам

Разрабатываемые программы должны отвечать следующим функциональным требованиям.

1. Программы должны сгенерировать три различных последовательности, элементы которых: расположены в случайном порядке, в обратном порядке, частично отсортированы.

2. Программы должны провести параллельную сортировку элементов последовательностей с помощью технологии OpenMP на общей памяти кластерной вычислительной системы и гибрида MPI и OpenMP на распределенной памяти кластерной вычислительной системы.

3. Программы должны зафиксировать время выполнения всех алгоритмов со всеми последовательностями.

4. Программы должны вывести время выполнения алгоритмов на консоль и сохранить их в файле.

Разрабатываемые программы должны отвечать следующим нефункциональным требованиям.

1. Программы должна быть стабильной и надежной, а количество ошибок и сбоев должно быть минимальным.

2. Время исполнения должно быть достаточно маленьким, чтобы обеспечить быстродействие на практике.

3. Программы должна быть способна обрабатывать большие объемы данных без потери производительности и стабильности.

2.2. Варианты использования

Для проектирования Q-эффективных программ был использован язык графического описания для объектно-ориентированного программирования UML. Была построена модель взаимодействия внешнего актера с программами в виде диаграммы вариантов использования. В ходе анализа требова-

ний к разрабатываемым программам были выявлены варианты использования, показанные на рисунке 1. Варианты использования совпадают для программы, реализованной для общей памяти и программы, реализованной для распределенной памяти.

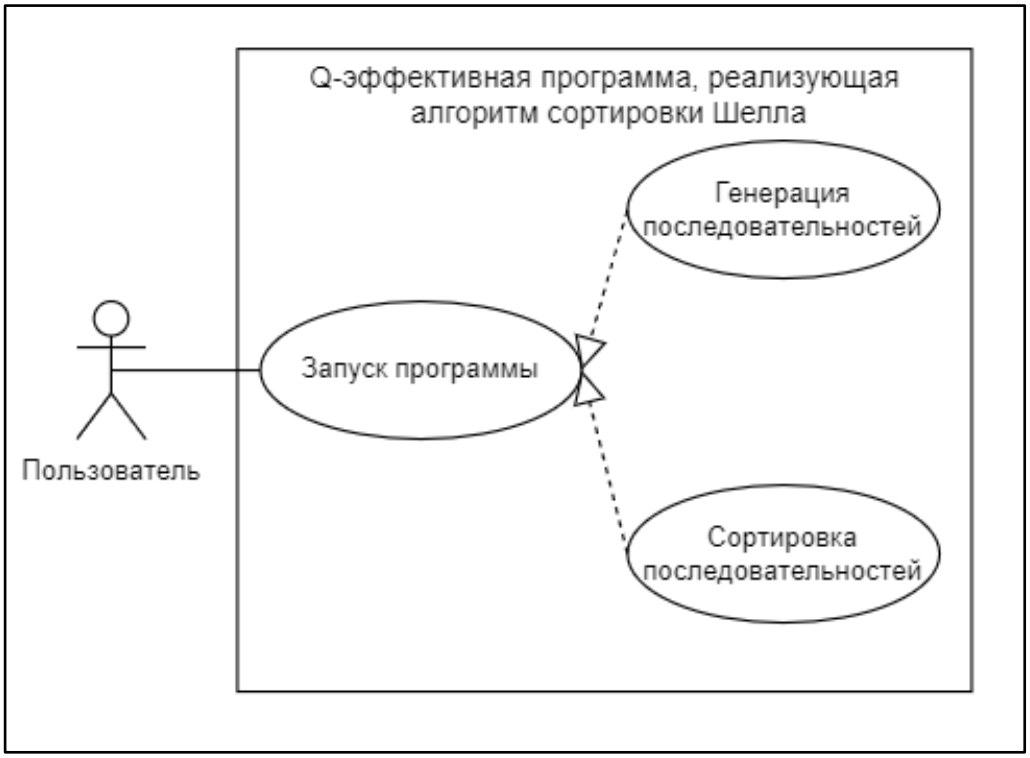

Рисунок 1 – Варианты использования Q-эффективных программ

Главным и единственным актером является пользователь.

«Генерация последовательности» включает в себя генерацию трех различных последовательностей заданной длины, элементы которой: расположены в случайном порядке, отсортированы в обратном порядке и частично упорядочены.

«Сортировка последовательности» содержит в себе реализацию алгоритма сортировки Шелла для общей и распределенной памяти кластерной вычислительной системы, включая все операции над массивами, необходимые для их сортировки, как, например, распределение и сбор данных в программе для распределенной памяти. Также данный прецедент отвечает за подсчет времени работы алгоритмов.

Спецификации вариантов использования представлены в таблицах 4 и 5.

Таблица 4 – Спецификация для прецедента «Генерация последовательности»

UseCase: Генерация последовательности.
ID: UC-01
Аннотация: Задание пользователем параметров для генерации.
Главные актеры: Пользователь.
Второстепенные актеры: Нет.
Предусловия: Программа запущена и ожидает ввода параметров генерации последовательности.
Основной поток: 1. Прецедент начинается, когда пользователь запускает программу. 2. Программа генерирует три различных последовательности заданной длины N.
Постусловия: Программа ожидает выбора типа сортировки последовательности.
Альтернативные потоки: Нет.

Таблица 5 – Спецификация для прецедента «Сортировка последовательности»

UseCase: Сортировка последовательности.
ID: UC-02
Аннотация: Выбор пользователем типа сортировки последовательности.
Главные актеры: Пользователь.
Второстепенные актеры: Нет.
Предусловия: Программа ожидает выбора реализации сортировки последовательности.
Основной поток: 1. Прецедент начинается после генерации последовательностей. 2. Выполняется поочередная сортировка всех сгенерированных последовательностей. 3. Программа фиксирует время выполнения сортировки. 4. Программа отображает время сортировки в консоли и записывает его в файле.
Постусловия: Программа завершает свое выполнение.
Альтернативные потоки: Нет.

2.3. Общее описание архитектуры Q-эффективных программ

Программы, реализующие алгоритм сортировки Шелла на общей и распределенной памяти кластерной вычислительной системы, включают в себя по одному классу: ShellSortOpenMP (рисунок 2) и ShellSortMPI (рисунок 3) соответственно.

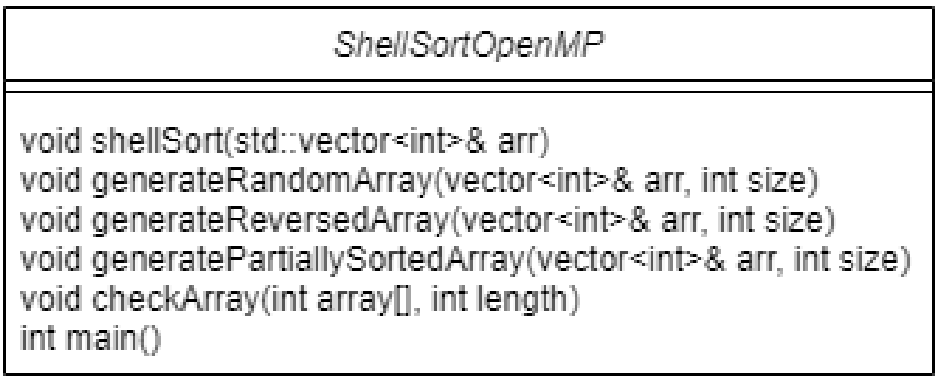

Рисунок 2 – Класс `ShellSortOpenMP`

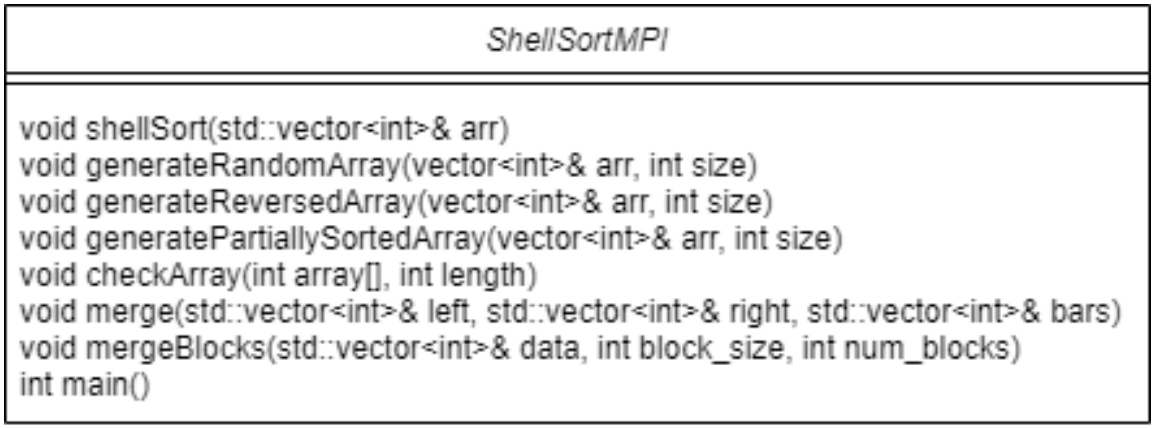

Рисунок 3 – Класс `ShellSortMPI`

В данных классах за выполнение сортировки методом Шелла определенного объема данных отвечает функция `ShellSort`, за генерацию трех различных массивов заданной длины – функции `generateRandomArray`, `generateReversedArray` и `generatePartiallySortedArray`, проверка корректности сортировки массивов осуществляется функцией `checkArray`. В классе ShellSortMPI также реализованы методы, отвечающие за слияние подмассивов. Вызовы функций генерации, сортировки и проверки корректности сортировки массивов, а также подсчет времени выполнения алгоритма осуществляются в функции `main`.

На диаграмме конечных автоматов (рисунок 4) представлены состояния, в которых находится Q-эффективная программа во время выполнения,

реализующая алгоритм сортировки Шелла на общей памяти кластерной вычислительной системы. В начале своей работы программа генерирует три массива заданной длины, элементы которых: расположены в случайном порядке, частично упорядочены, отсортированы в обратном порядке. После чего программа поочередно сортирует каждый из массивов алгоритмом сортировки Шелла, реализованным в программе параллельно с применением технологии OpenMP, фиксируя время работы алгоритма. После проверки расположения элементов каждого из массивов в порядке неубывания программа завершает свою работу. Все операции, во время выполнения сортировки массивов осуществляются по мере готовности.

Рисунок 4 – Диаграмма конечных автоматов программы для общей памяти

На диаграмме конечных автоматов (рисунок 5) представлены состояния, в которых находится программа, реализующая алгоритм сортировки Шелла на распределенной памяти кластерной вычислительной системы, во время своего выполнения.

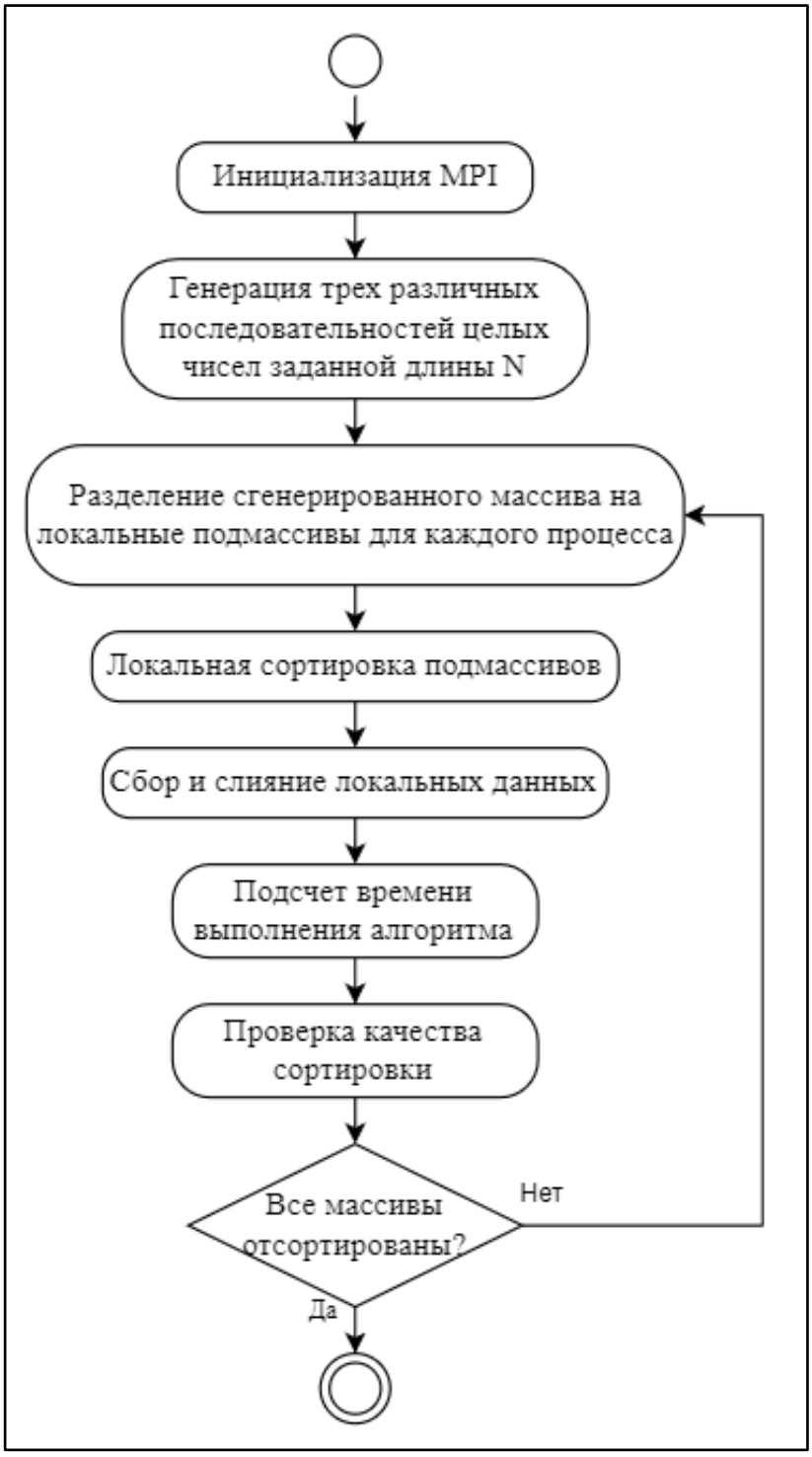

Рисунок 5 – Диаграмма конечных автоматов программы для распределенной памяти

После запуска программа инициализирует MPI, затем главный процесс генерирует массивы для дальнейшей их сортировки. После чего главный процесс разделяет данные между всеми процессами узлов, на которых была запущена программа. Дальше осуществляется локальная сортировка подмассивов отдельными процессами с применением OpenMP. После локальной сортировки подмассивы собираются главным процессом и последовательно объединяются в один таким образом, чтобы после слияния отсортированных подмассивов получился отсортированный результат, после чего программа фиксирует время сортировки и проверяет расположение элементов массива в порядке неубывания и выводит время сортировки и результат проверки массива. Затем программа осуществляет данный алгоритм поочередно для оставшихся неотсортированных массивов, если все массивы были отсортированы и проверены, программа завершает свое выполнение. Управление блоками данных, разделение их между процессами и последующий сбор осуществляется с применением технологии MPI. Все операции, во время выполнения сортировки массивов осуществляются по мере готовности.

Выводы по второй главе

В данном разделе были определены функциональные и нефункциональные требования проектируемых программ. На их основе была определена архитектура программ, а также смоделировано их поведение с помощью описания состояний, в которых могут находится программы во время их выполнения.

3. РЕАЛИЗАЦИЯ

Для реализации Q-эффективных программ был выбран язык программирования C++. Для реализации алгоритма сортировки Шелла на общей памяти кластерной вычислительной системы была применена технология OpenMP для управления потоками программы, а для реализации на распределенной памяти системы – гибрид технологий MPI для управления процессами программы и OpenMP для управления потоками программы.

3.1. Реализация Q-эффективной программы для общей памяти кластерной вычислительной системы

Q-эффективная программа ShellSortOpenMP реализует алгоритм сортировки Шелла на общей памяти. Данная программа в начале своего выполнения генерирует случайный, обратный и частично отсортированный массивы, затем выполняет их сортировку с применением алгоритма Шелла, после чего фиксирует время выполнения алгоритма для каждого из массивов и сохраняет эти данные в отдельный файл с названием вида: «slurm-<уникальный_идентификатор_задачи>.out». Все операции над массивами выполняются сразу по мере их готовности.

Для сборки программы для общей применяется скрипт «compile.sh», содержащий строку `g++ -std=c++0x -fopenmp ShellSortOpenMP.cpp -o ShellSortOpenMP_XX`, в которой выполняется команда `g++ -std=c++0x -fopenmp`, осуществляющая сборку OpenMP-программы компилятором g++ со стандартом `-std=c++0x`. После сборки, в директории, в которой находятся скрипт и компилируемый cpp-файл создается исполняемый файл с названием «ShellSortOpenMP_XX», где XX – идентификатор длины массивов, которые генерируются программой. В дальнейшем созданный исполняемый файл будет использоваться для постановки задачи в очередь.

Для постановки задачи в очередь используется «run_task.sh», содержащий строку `sbatch --cpus-per-task=X -p quick ./task.sh`, в которой

выполняется команда очереди задач и запускается скрипт «task.sh» (листинг 1), используя $2 \leq X \leq 12$ процессорных ядер узла кластерной системы.

Листинг 1 – Скрипт task.sh для OpenMP программы

```
#!/bin/bash

#SBATCH

mpirun ./ShellSortOpenMP_XX
```

Указанный скрипт осуществляет запуск исполняемого файла OpenMP программы «ShellSortOpenMP_XX».

В программе ShellSortOpenMP алгоритм сортировки Шелла реализован в функции `shellSort`, приведенной в листинге 2.

Листинг 2 – Метод `shellSort`

```
void shellSort(std::vector<int>& arr) {
    int n = arr.size();

    #pragma omp parallel
    {
        for (int gap = n / 2; gap > 0; gap /= 2) {
            #pragma omp for
            for (int i = gap; i < n; i++) {
                int temp = arr[i];
                int j;
                #pragma omp simd reduction(+:j)
                for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                    arr[j] = arr[j - gap];
                }
                arr[j] = temp;
            }
        }
    }
}
```

Принимая в качестве входных параметров заранее сгенерированный массив целых чисел или часть массива, в случае выполнения программы для распределенной памяти, данная функция осуществляет сортировку блока данных в соответствии с методом Шелла, параллельно выполняя итерации циклов с помощью технологии OpenMP, разделяя работу между потоками процесса.

Генерацию случайного, обратного и частично отсортированного массивов заданной длины осуществляют функции `generateRandomArray` (листинг 3), `generateReversedArray` (листинг 4), `generatePartiallySortedArray` (листинг 5) соответственно.

Листинг 3 – Генерация случайного массива

```
void generateRandomArray(vector<int>& arr, int size) {
    arr.resize(size);
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % 1000;
    }
}
```

Данная функция заполняет случайными числами от 0 до 999 пустой массив длины `size`, выступающий в качестве входных параметров функции.

Листинг 4 – Генерация обратного массива

```
void generateReversedArray(vector<int>& arr, int size) {
    arr.resize(size);
    for (int i = 0; i < size; ++i) {
        arr[i] = size - i;
    }
}
```

Данная функция заполняет пустой массив длины `size`, выступающий в качестве входных параметров функции, числами от `size` до 0.

Листинг 5 – Генерация частично упорядоченного массива

```
void generatePartiallySortedArray(vector<int>& arr, int size) {
    arr.resize(size);
    for (int i = 0; i < size; ++i) {
        if (i < size / 2) {
            arr[i] = i;
        } else {
            arr[i] = rand() % 1000;
        }
    }
}
```

Данная функция заполняет первую половину пустого массива длины `size`, выступающего в качестве входных параметров функции, числами от 0 до `size/2`, вторая половина массива заполняется случайными числами от 0 до 999.

3.2. Реализация Q-эффективной программы для распределенной памяти кластерной вычислительной системы

Q-эффективная программа ShellSortMPI реализует алгоритм сортировки Шелла на распределенной памяти кластерной вычислительной системы. Данная программа в начале своего выполнения инициализирует MPI, после определяет ранг текущего процесса и общее число процессов, затем главный процесс генерирует случайный, обратный и частично отсортированный массивы. После чего сгенерированные массивы разделяются на подмассивы равной длины для каждого процесса программы. Затем осуществляется локальная сортировка подмассивов каждым из процессов с применением алгоритма Шелла. После локальной сортировки главный процесс осуществляет сбор и слияние подмассивов в глобальный отсортированный массив. После чего фиксирует время выполнения алгоритма для каждого из массивов и сохраняет эти данные в отдельный файл с названием вида: «slurm-<уникальный_идентификатор_задачи>.out». Все операции над массивами выполняются сразу по мере их готовности.

Для сборки программы применяется скрипт «compile.sh», содержащий строку `mpicc ShellSortMPI.cpp -o ShellSortMPI_XX`, в которой выполняется команда `mpicc`, осуществляющая сборку MPI/OpenMP-программы, автоматически подключающую все необходимые заголовочные файлы и библиотеки. После сборки, в директории, в которой находятся скрипт и компилируемый cpp-файл создается исполняемый файл с названием «ShellSortMPI_XX», где XX – идентификатор длины массивов, которые генерируются программой. В дальнейшем созданный исполняемый файл будет использоваться для постановки задачи в очередь.

Постановка задачи в очередь осуществляется с помощью скрипта «run_task.sh», содержащий строку `sbatch -N X -p quick ./task.sh`, выполняющую команду очереди задач и запускающую скрипт «task.sh» (листинг 6), используя $2 \leq X \leq 32$ узлов кластерной системы и все процессорные ядра задействованных узлов.

Листинг 6 – Скрипт task.sh для MPI программы

```
#!/bin/bash

#SBATCH

mpiexec.hydra ./ShellSortMPI_XX
```

Указанный скрипт осуществляет запуск исполняемого файла MPI программы ShellSortMPI_XX.

Функции генерации и сортировки массивов в гибридной MPI/OpenMP программе не отличаются от функций, выполняющих эти задачи в OpenMP программе, которые были приведены в листингах 2–5.

В листинге 7 приведена инициализация MPI в реализованной программе.

Листинг 7 – Инициализация MPI

```
MPI_Init(&argc, &argv);

int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Функция `MPI_Init(&argc, &argv)` непосредственно инициализирует MPI. Функция `MPI_Comm_rank(MPI_COMM_WORLD, &rank)` возвращает ранг текущего процесса внутри коммуникатора `MPI_COMM_WORLD` [6]. Функция `MPI_Comm_size(MPI_COMM_WORLD, &size)` возвращает общее количество процессов в коммуникаторе `MPI_COMM_WORLD` для дальнейшего разделения данных .

Разделение данных между процессами осуществляется с помощью функции `MPI_Scatter` (листинг 8). Эта функция принимает в качестве аргументов исходный массив, количество элементов, которые должны быть отправлены каждому процессу, тип этих элементов, приемный буфер, количество элементов, ранг процесса-источника и коммуникатор, которые должны быть приняты каждым процессом [9].

Листинг 8 – Функция `MPI_Scatter`

```
MPI_Scatter(global_arrays[i].data(), array_size / size, MPI_INT,
local_array.data(), array_size / size, MPI_INT, 0, MPI_COMM_WORLD);
```

Объединение массивов в реализованной программе выполняется с помощью функции `mergeBlocks` (листинг 9). Эта функция принимает в качестве аргументов исходный массив данных, размер блока и количество блоков. Она создает временные массивы `left` и `right` для хранения двух блоков, которые будут объединены, и массив `merged_data` для хранения результатов объединения.

Листинг 9 – Функция `mergeBlocks`

```
void mergeBlocks(std::vector<int>& data, int block_size, int num_blocks) {
    std::vector<int> merged_data(data.size());
    std::vector<int> left(block_size);
    std::vector<int> right(block_size);

    for (int i = 0; i < num_blocks - 1; i++) {
        std::copy(data.begin() + i * block_size, data.begin() + (i + 1) *
block_size, left.begin());
        std::copy(data.begin() + (i + 1) * block_size, data.begin() + (i +
2) * block_size, right.begin());

        merge(left, right, merged_data);

        std::copy(merged_data.begin(), merged_data.begin() + (i + 2) *
block_size, data.begin());
    }
}
```

В цикле для каждой пары блоков выполняются следующие шаги.

1. Копирование блоков: с помощью функции `std::copy` копируются два блока данных из исходного массива в массивы `left` и `right`.

2. Объединение блоков: функция `merge` (листинг 10) объединяет два блока в один отсортированный блок. Она сравнивает первые элементы обоих блоков и перемещает меньший элемент в массив `merged_data`. Этот процесс повторяется, пока один из блоков не опустеет. Затем оставшиеся элементы из непустого блока добавляются в `merged_data`.

3. Копирование объединенных данных обратно в исходный массив: объединенные данные копируются обратно в исходный массив с помощью функции `std::copy`.

Этот процесс повторяется для каждой пары блоков в исходном массиве. В результате получается один отсортированный массив.

Важно отметить, что объединение блоков выполняется только в процессе с рангом 0 (главный процесс), так как только этот процесс имеет доступ ко всему глобальному массиву после выполнения функции `MPI_Gather` [17].

Листинг 10 – Функция `merge`

```
void merge(std::vector<int>& left, std::vector<int>& right, std::vec-
tor<int>& bars) {
    int nL = left.size();
    int nR = right.size();
    int i = 0, j = 0, k = 0;

    while (j < nL && k < nR) {
        if (left[j] < right[k]) {
            bars[i] = left[j];
            j++;
        }
        else {
            bars[i] = right[k];
            k++;
        }
        i++;
    }
    while (j < nL) {
        bars[i] = left[j];
        j++; i++;
    }
    while (k < nR) {
        bars[i] = right[k];
        k++; i++;
    }
}
```

Выводы по третьей главе

В рамках данной главы были разработаны программы, реализующие алгоритм сортировки Шелла для общей и распределенной памяти. Приведена программная реализация всех функциональных требований.

4. ТЕСТИРОВАНИЕ

4.1. Функциональное тестирование разработанных программ

Исследования выполнены с использованием суперкомпьютерных ресурсов ФГАОУ ВО «ЮУрГУ (НИУ)» [7, 12].

Функциональное тестирование проверяет соответствие требованиям разрабатываемых Q-эффективных программ. Для проверки корректности работы алгоритма сортировки в разработанных программах используется функция `checkArray`, приведенная в листинге 11.

Листинг 11 – Функция `checkArray`

```
void checkArray(int array[], int length)
{
      bool check;
      for (int i = 0; i < length; ++i)
      {
           if (array[i] > array[i + 1])
           {
               check = true;
           }
           else
           {
               check = false;
               break;
           }
      }
      if (check) cout << "\n Array successfully sorted";
      else cout << "\n Array not sorted";
}
```

Данная функция проверяет расположение элементов массива по неубыванию и выводит в консоль результат проверки. Ожидается, что все типы генерируемых массивов будут успешно отсортированы программой при ее запуске на любом количестве процессорных ядер. В приложении Б приведены результаты функционального тестирования разработанных программ. Во всех проводимых тестах программа выводила подтверждение корректной работы алгоритма. Таким образом, функциональное тестирование программ было успешно пройдено. Сортировка массивов происходит корректным образом.

4.2. Оценка динамических характеристик

Вместе с функциональным тестированием программ проводились вычислительные эксперименты на суперкомпьютере «Торнадо ЮУрГУ», в которых определялись следующие динамические характеристики [11].

1. Ускорение – отношение времени выполнения последовательной программы к времени выполнения параллельной программы.

2. Эффективность распараллеливания – отношение полученного ускорения к числу используемых процессорных ядер.

Под последовательной программой понимается параллельная программа, выполняемая на одном ядре одного вычислительного узла. В остальных случаях во время экспериментов в программе для общей памяти количество процессорных ядер варьировалось от 2 до 12, а в программе для распределенной памяти на каждом вычислительном узле используются все 12 физических ядер и максимальное возможное количество одновременно выполняющихся нитей, равное 24. В приложении В представлены результаты экспериментов для обеих разработанных программ.

Результаты экспериментов можно охарактеризовать следующим образом. Во время сортировки случайных и частично упорядоченных массивов на общей памяти наблюдается умеренный рост ускорения и уменьшение эффективности распараллеливания по мере увеличения числа процессорных ядер, на которых запускалась программа.

Программа же для распределенной памяти обладает значительно большим ускорением, видимый рост которого наблюдается до запуска программы на 204 процессорных ядрах системы, после чего начинает использоваться весь ресурс параллелизма программы и ускорение перестает возрастать, эффективность программ достигает своего пика на 24 процессорных ядрах, на большем их количестве постепенно снижается.

Данные закономерности справедливы для всех размерностей массивов, однако также наблюдается, что программы обладают наименьшими ускорением и эффективностью для массивов длиной 50 миллионов, при

этом ускорение и эффективность возрастают при изменении длины массива в большую и меньшую стороны. Во время сортировки обратных массивов резкий рост ускорения и эффективности наблюдается во время запуска программы для общей памяти на четырех процессорных ядрах системы. Программа для распределенной памяти во время сортировки обратных массивов обладает меньшим ускорением, чем программа для общей памяти, ресурс параллелизма программы для распределенной памяти при этом полностью используется при запуске программы на 84 процессорных ядрах. Для обратных массивов зависимость ускорения и эффективности от длины массива соответствует таковой для случайных и частично упорядоченных, то есть достигают своего минимума при размерности 50 миллионов, увеличиваясь при иных размерностях.

При этом, при полном использовании ресурса параллелизма алгоритма сортировки Шелла наибольшие ускорение и эффективность наблюдаются во время сортировки случайных массивов, а наименьшее – во время сортировки обратных массивов.

Выводы по четвертой главе

В рамках данной главы было проведено функциональное тестирование программ, результаты которого показали корректное выполнение сортировки сгенерированных массивов программами при любых изменяемых параметрах запуска, а также проведены вычислительные эксперименты и выявлены зависимости ускорения и эффективности программ от размерностей сортируемых массивов и количества процессорных ядер, на которых запускались программы.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы были разработаны Q-эффективные программы для реализации алгоритма сортировки Шелла на общей и распределенной памяти, проведены вычислительные эксперименты с разработанными программами и оценены их динамические характеристики. Для достижения данной цели были решены следующие задачи:

1) разработана Q-эффективная программа для реализации алгоритма сортировки Шелла на общей памяти;
2) разработана Q-эффективная программа для реализации алгоритма сортировки Шелла на распределенной памяти;
3) проведено функциональное тестирование разработанных программ;
4) проведен сбор данных о работе программ;
5) оценены динамические характеристики разработанных программ.

Опираясь на полученные результаты вычислительных экспериментов, можно сделать вывод о том, что Q-эффективные программы, реализующие алгоритм сортировки Шелла, обладают значительно большей производительностью в сравнении с последовательной сортировкой. При этом наилучшие результаты показывает программа для распределенной памяти, сортирующая случайный массив. Однако для выявления дальнейших закономерностей и возможного достижения лучших динамических характеристик для Q-эффективных программ, реализующих алгоритмы сортировки необходимо проводить дополнительные исследования с другими алгоритмами сортировки, другими типами и размерностями сортируемых массивов.

ЛИТЕРАТУРА

1. Алеева В.Н. Q-эффективная реализация численных алгоритмов. // Челябинск: Издательский центр ЮУрГУ, 2017. – С. 346–353.

2. Алеева В.Н. Основные положения технологии Q-эффективного программирования. // Челябинск: Издательский центр ЮУрГУ, 2019. – С. 334–342.

3. Алеева В.Н., Алеев Р.Ж. Применение Q-детерминанта численных алгоритмов для параллельных вычислений. // Челябинск: Издательский центр ЮУрГУ, 2019. – С. 133–145.

4. Алеева В.Н., Шатов М.Б. Применение концепции Q-детерминанта для эффективной реализации численных алгоритмов на примере метода сопряженных градиентов для решения систем линейных уравнений. // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика, 2021. – Т. 10, № 3. – С. 56–71.

5. Aleeva V. Designing a parallel programs on the base of the conception of Q-Determinant. // Supercomputing: 4th Russian Supercomputing Days, RuSCDays 2018, Moscow, Russia, September 24–25, 2018, Revised Selected Papers 4. – Springer International Publishing, 2019. – 565–577 pp.

6. Антонов А.С. Технологии параллельного программирования MPI и OpenMP: Учеб. пособие. // М.: Издательство Московского университета, 2012. – 344 с.

7. Биленко Р.В., Долганина Н.Ю., Иванова Е.В., Рекачинский А.И. Высокопроизводительные вычислительные ресурсы ЮжноУральского государственного университета. // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2022. – Т. 11, № 1. – С. 15–30.

8. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. // СПб.: БХВ-Петербург, 2002. – 608 с.

9. Гримм Р. Параллельное программирование на современном языке C++. // пер. с англ. В. Ю. Винника. // – М.: ДМК Пресс, 2022. – 618 с.

10. Кнут Д. Искусство программирования. Том 3. Сортировка и поиск, 2-е изд. // – М.: ООО «И. Д. Вильямс», 2018. – 832 с.

11. Открытая энциклопедия параллельных алгоритмических функций. [Электронный ресурс] URL: http://algowikiproject.org/en/Open_Encyclopedia_of_Parallel_Algorithmic_Features (дата обращения: 24.05.2024 г.).

12. Программно-аппаратный вычислительный комплекс «СКИФАврора ЮУрГУ». Суперкомпьютер «Торнадо ЮУрГУ». Руководство пользователя // 88746109.007.ЮУрГУ.И3.2. Москва: ЗАО «РСК Технологии», 2012. – С. 10–12.

13. MPI Forum Documents. [Электронный ресурс] URL: https://www.mpi-forum.org/docs (дата обращения: 22.05.2024 г.).

14. Microsoft MPI – Microsoft Docs. [Электронный ресурс] URL: https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi?redirectedfrom=MSDN#community-resources (дата обращения: 22.10.2023 г.)

15. Open MPI Documentation. [Электронный ресурс] URL: https://www.open-mpi.org/doc/ (дата обращения: 22.05.2024 г.).

16. The OpenMP API specification for parallel programming. [Электронный ресурс] URL: https://www.openmp.org/specifications (дата обращения: 10.02.2023 г.).

17. Уильямс Э. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. // М.: ДМК Пресс, 2012. – 672 с.

ПРИЛОЖЕНИЯ

Приложение А. Q-детерминант алгоритма сортировки Шелла для массива из 4 элементов

В рамках курсовой работы, выполненной в 2023 году, на тему: «Применение метода проектирования Q-эффективных программ для алгоритма сортировки Шелла» был построен Q-детерминант для алгоритма сортировки Шелла для массива из 4 элементов, приведенный в таблице 1.

Таблица 1 – Q-детерминант алгоритма сортировки Шелла для массива из 4 элементов

A(1)A(2)A(3)A(4) (A(1)<=A(3))&(A(2)<=A(4))&(A(1)<=A(2))&(A(2)<=A(3))&(A(3)<=A(4))
A(1)A(3)A(2)A(4) (A(1)<=A(3))&(A(2)<=A(4))&(A(1)<=A(2))&(A(2)>A(3))&(A(1)<=A(3))&(A(2)<=A(4))
A(1)A(4)A(3)A(2) (A(1)<=A(3))&(A(2)>A(4))&(A(1)<=A(4))&(A(4)<=A(3))&(A(3)<=A(2))
A(1)A(3)A(4)A(2) (A(1)<=A(3))&(A(2)>A(4))&(A(1)<=A(4))&(A(4)>A(3))&(A(1)<=A(3))&(A(4)<=A(2))
A(4)A(1)A(3)A(2) (A(1)<=A(3))&(A(2)>A(4))&(A(1)>A(4))&(A(1)<=A(3))&(A(3)<=A(2))
A(3)A(2)A(1)A(4) (A(1)>A(3))&(A(2)<=A(4))&(A(3)<=A(2))&(A(2)<=A(1))&(A(1)<=A(4))
A(3)A(1)A(2)A(4) (A(1)>A(3))&(A(2)<=A(4))&(A(3)<=A(2))&(A(2)>A(1))&(A(3)<=A(1))&(A(2)<=A(4))
A(2)A(3)A(1)A(4) (A(1)>A(3))&(A(2)<=A(4))&(A(3)>A(2))&(A(3)<=A(1))&(A(1)<=A(4))
A(3)A(4)A(1)A(2) (A(1)>A(3))&(A(2)>A(4))&(A(3) <=A(4))&(A(4)<=A(1))&(A(1)<=A(2))
A(2)A(3)A(1)A(4) (A(1) >A(3))& (A(2) <=A(4))&(A(3) >A(2))&(A(3) <=A(1))&(A(1)<=A(4))
A(3)A(1)A(4)A(2) (A(1)>A(3))&(A(2)>A(4))&(A(3)<=A(4))&(A(4)>A(1))&(A(3)<=A(1))&(A(4)<=A(2))
A(4)A(3)A(1)A(2) (A(1)>A(3))&(A(2)>A(4))&(A(3)>A(4))&(A(3)<=A(1))&(A(1)<=A(2)
A(1)A(2)A(4)A(3) (A(1)<=A(3))&(A(2)<=A(4))&(A(1)<=A(2))&(A(2)<=A(3))&(A(3)>A(4))&(A(2)<=A(4))
A(2)A(1)A(4)A(3) (A(1)<=A(3))&(A(2)<=A(4))&(A(1)>A(2))&(A(1)<=A(3))&(A(3)>A(4))&(A(1)<=A(4))
A(1)A(4)A(2)A(3) (A(1)<=A(3))&(A(2)>A(4))&(A(1)<=A(4))&(A(4)<=A(3))&(A(3)>A(2))&(A(4)<=A(2))
A(4)A(1)A(2)A(3) (A(1)<=A(3))&(A(2)>A(4))&(A(1)>A(4))&(A(1)<=A(3))&(A(3)>A(2))&(A(1)<=A(2))
A(3)A(2)A(4)A(1) (A(1)>A(3))&(A(2)<=A(4))&(A(3)<=A(2))&(A(2)<=A(1))&(A(1)>A(4))&(A(2)<=A(4))

A(2)A(3)A(4)A(1) (A(1)>A(3))&(A(2)<=A(4))&(A(3)>A(2))&(A(3)<=A(1))&(A(1)>A(4))&(A(3)<=A(4))
A(3)A(4)A(2)A(1) (A(1)>A(3))&(A(2)>A(4))&(A(3)<=A(4))&(A(4)<=A(1))&(A(1)>A(2))&(A(4)<=A(2))
A(4)A(3)A(2)A(1) (A(1)>A(3))&(A(2)>A(4))&(A(3)>A(4))&(A(3)<=A(1))&(A(1)>A(2))&(A(3)<=A(2))
A(2)A(4)A(1)A(3) (A(1)<=A(3))&(A(2)<=A(4))&(A(1)>A(2))&(A(1)<=A(3))&(A(3)>A(4))&(A(1)>A(4))&(A(2)<=A(4))
A(4)A(2)A(1)A(3) (A(1)<=A(3))&(A(2)>A(4))&(A(1)>A(4))&(A(1)<=A(3))&(A(3)>A(2))&(A(1)>A(2))&(A(4)<=A(2))
A(2)A(4)A(3)A(1) (A(1)>A(3))&(A(2)<=A(4))&(A(3)>A(2))&(A(3)<=A(1))&(A(1)>A(4))&(A(3)>A(4))&(A(2)<=A(4))
A(4)A(2)A(3)A(1) (A(1)>A(3))&(A(2)>A(4))&(A(3)>A(4))&(A(3)<=A(1))&(A(1)>A(2))&(A(3)>A(2))&(A(4)<=A(2))

Приложение Б. Результаты функционального тестирования программ

Результаты функциональных тестов программ приведены в таблице 2.

Таблица 2 – Результаты функционального тестирования

Количество процессорных ядер	**Тип массивов**		
	Случайный	**Обратный**	**Частично упорядоченный**
	Тест пройден?		
2	Да	Да	Да
3	Да	Да	Да
4	Да	Да	Да
5	Да	Да	Да
6	Да	Да	Да
7	Да	Да	Да
8	Да	Да	Да
9	Да	Да	Да
10	Да	Да	Да
11	Да	Да	Да
12	Да	Да	Да
24	Да	Да	Да
36	Да	Да	Да
48	Да	Да	Да
60	Да	Да	Да
72	Да	Да	Да
84	Да	Да	Да
96	Да	Да	Да
120	Да	Да	Да
132	Да	Да	Да
156	Да	Да	Да
168	Да	Да	Да
180	Да	Да	Да
192	Да	Да	Да
204	Да	Да	Да
216	Да	Да	Да
228	Да	Да	Да
240	Да	Да	Да
252	Да	Да	Да
264	Да	Да	Да
276	Да	Да	Да
288	Да	Да	Да
312	Да	Да	Да
324	Да	Да	Да
336	Да	Да	Да
348	Да	Да	Да
360	Да	Да	Да
372	Да	Да	Да
384	Да	Да	Да

Приложение В. Результаты экспериментов

На рисунках 1–6 приведены результаты вычислительных экспериментов.

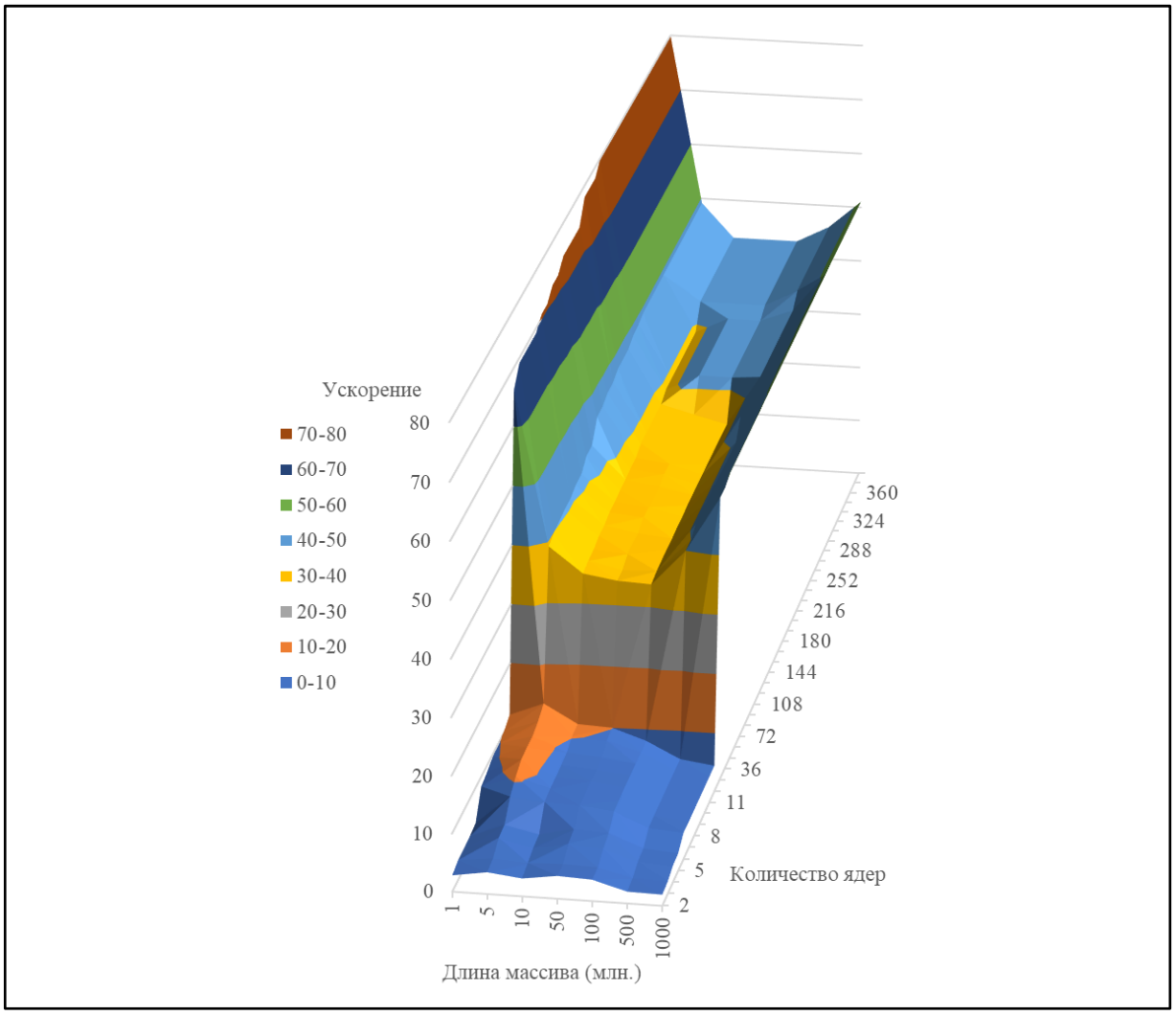

Рисунок 1 – Ускорение Q-эффективных программ для случайных массивов

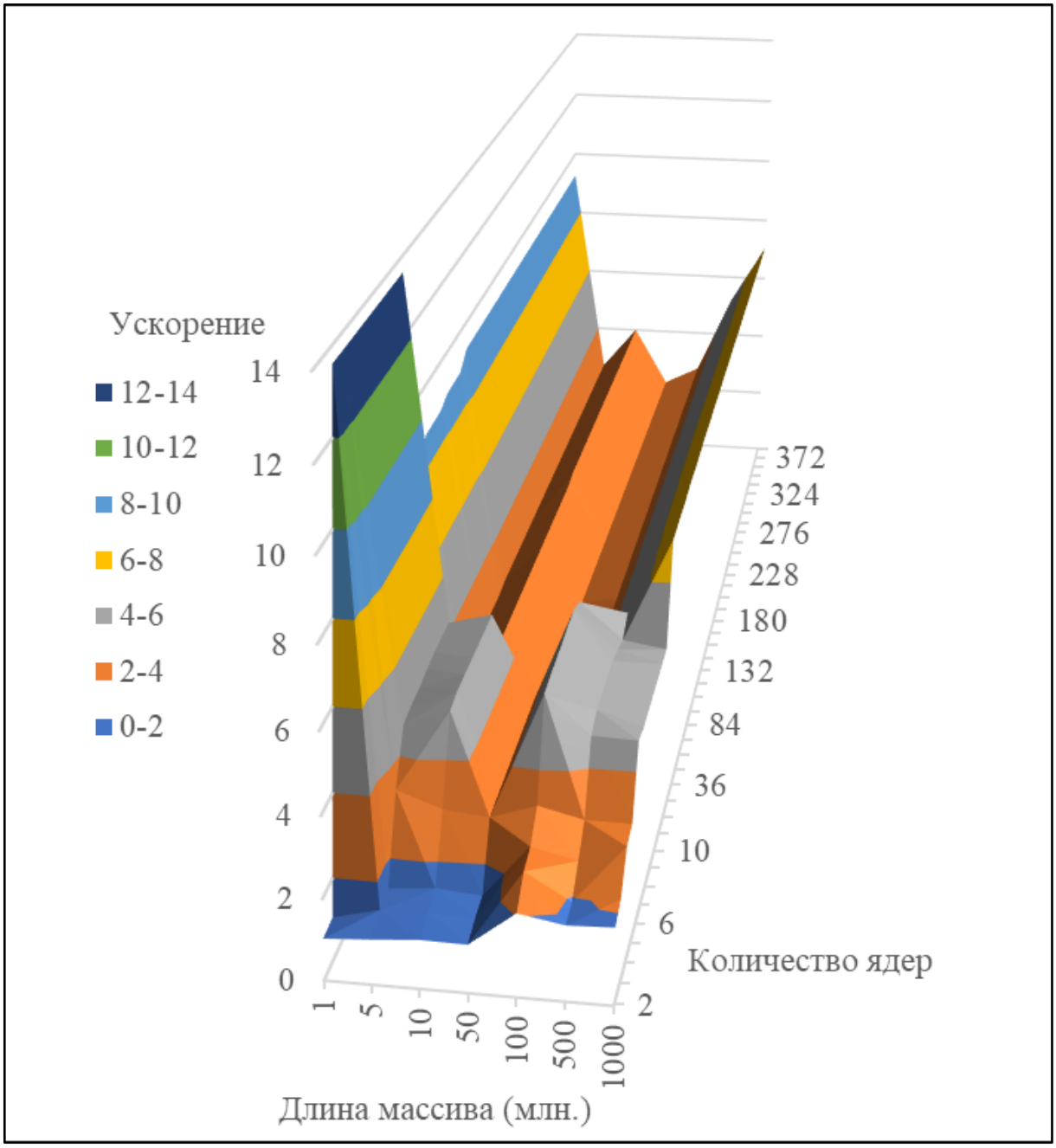

Рисунок 2 – Ускорение Q-эффективных программ для обратных массивов

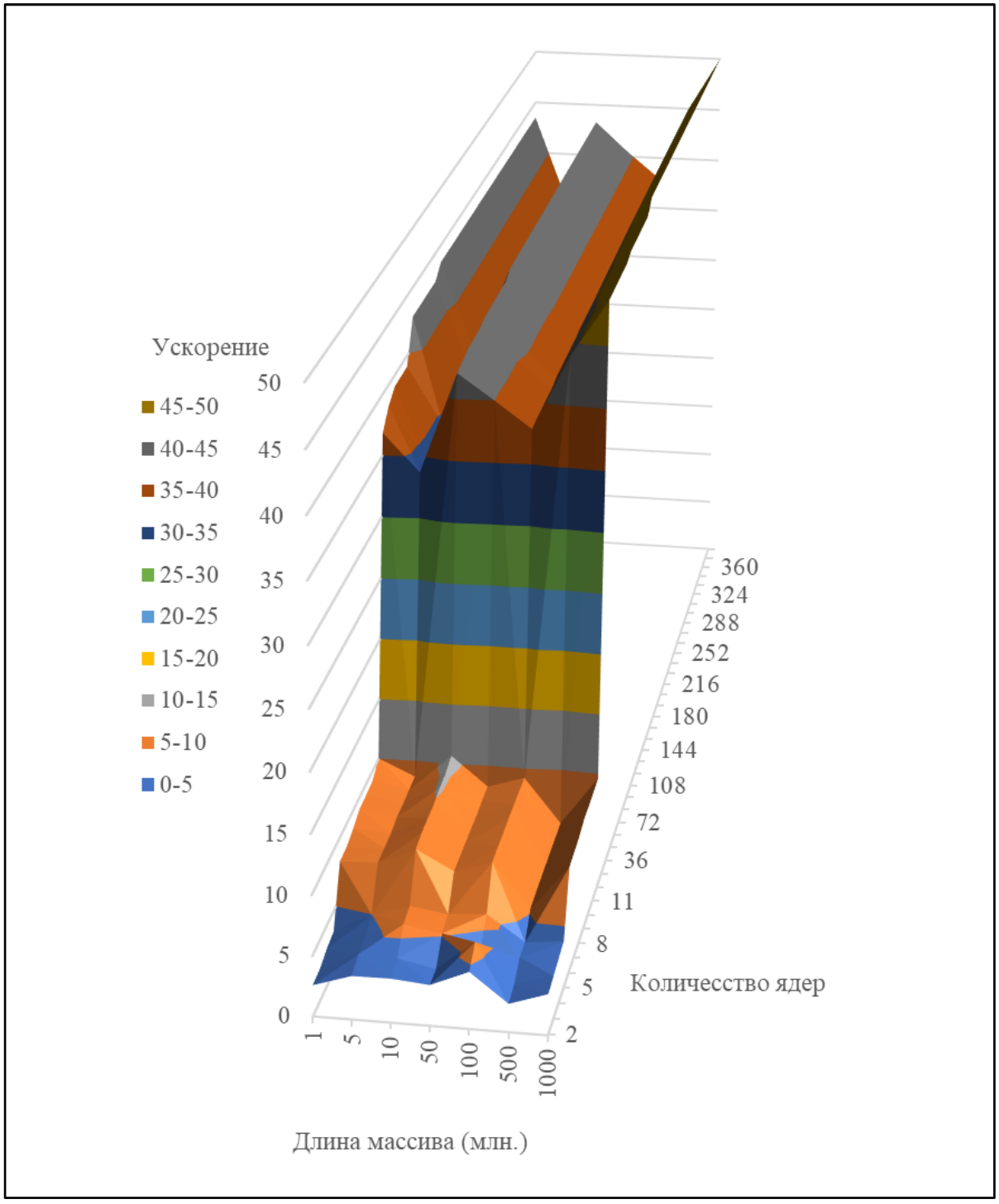

Рисунок 3 – Ускорение Q-эффективных программ для частично упорядоченных массивов

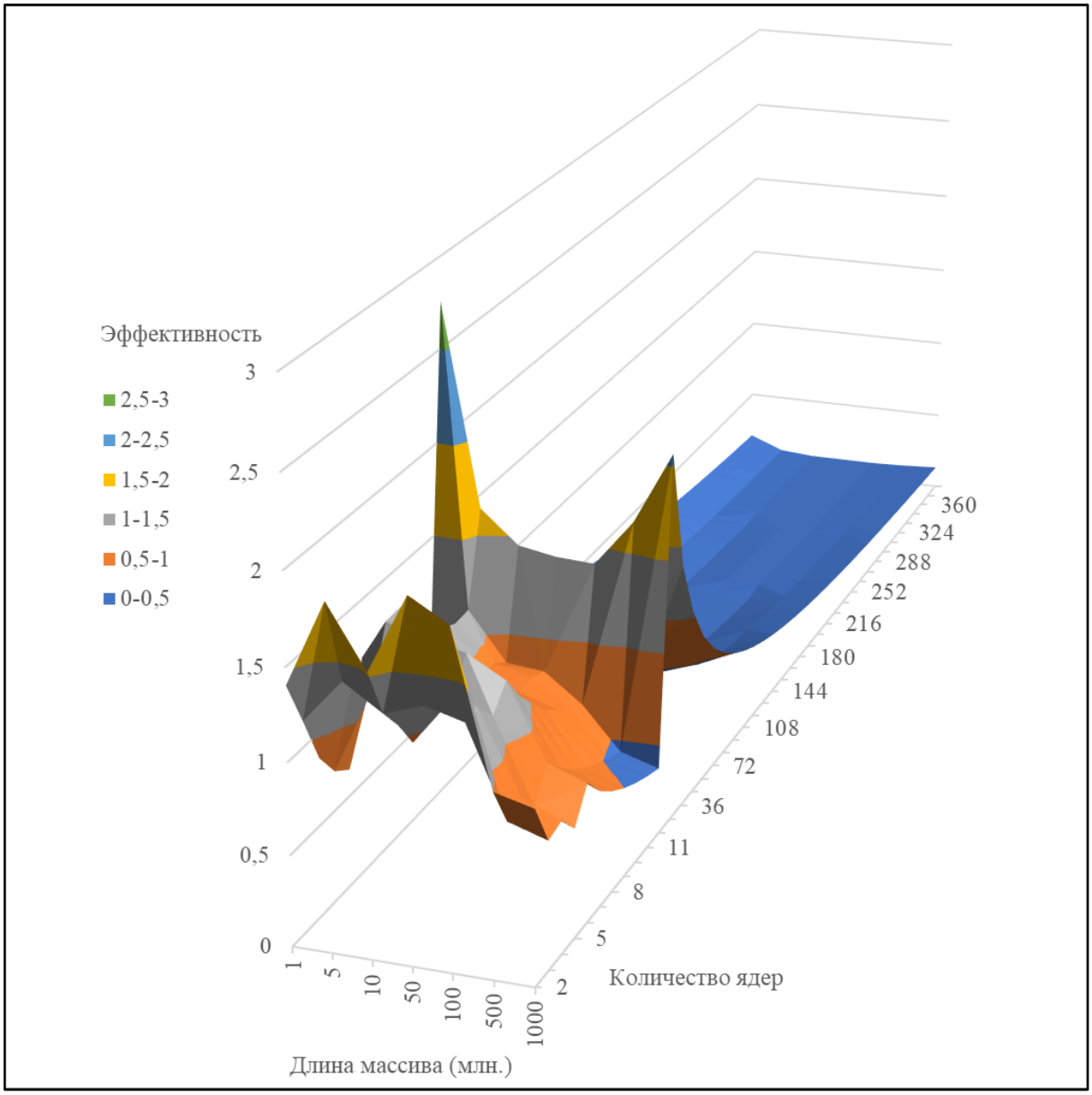

Рисунок 4 – Эффективность Q-эффективных программ для случайных массивов

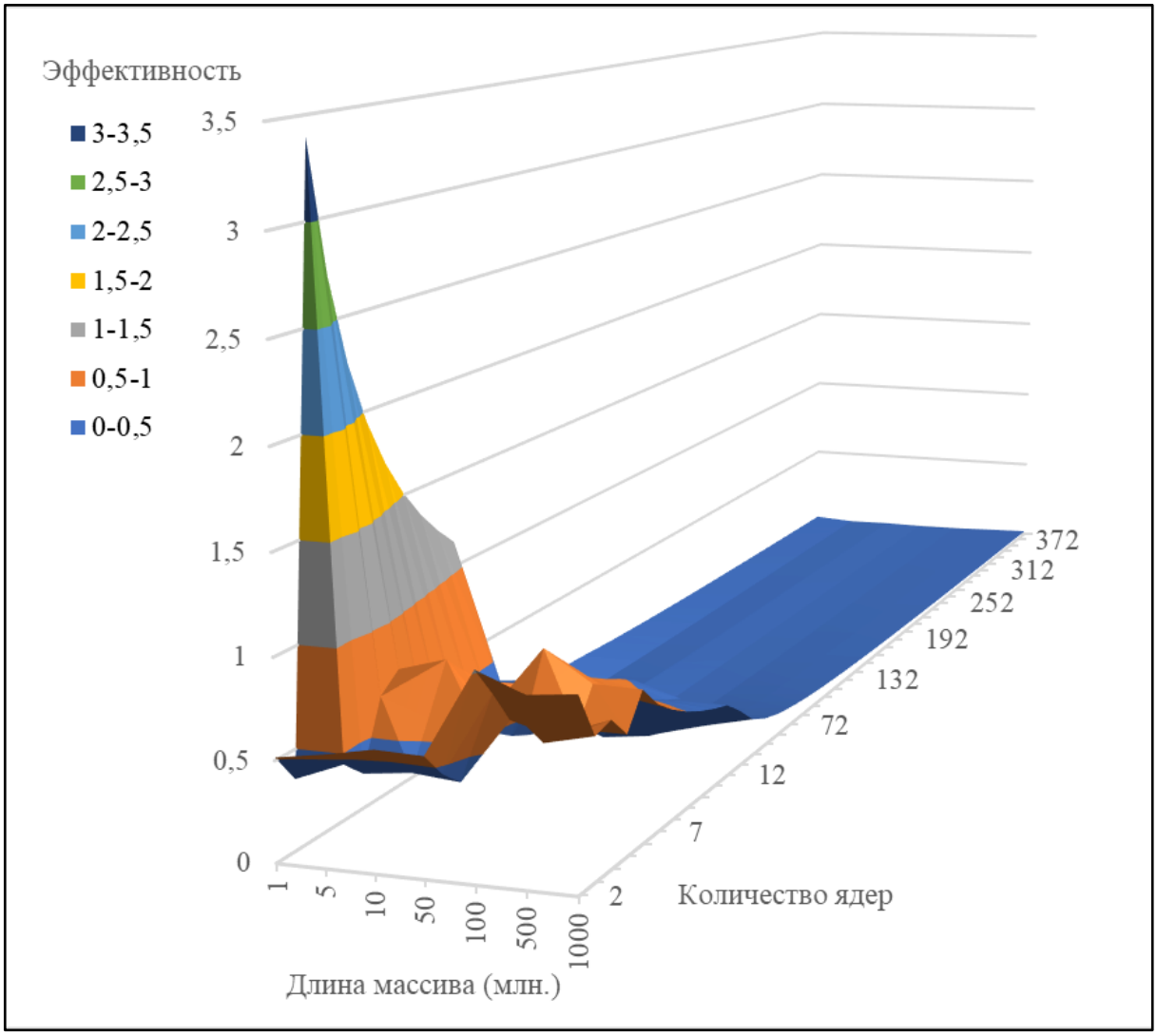

Рисунок 5 – Эффективность Q-эффективных программ для обратных массивов

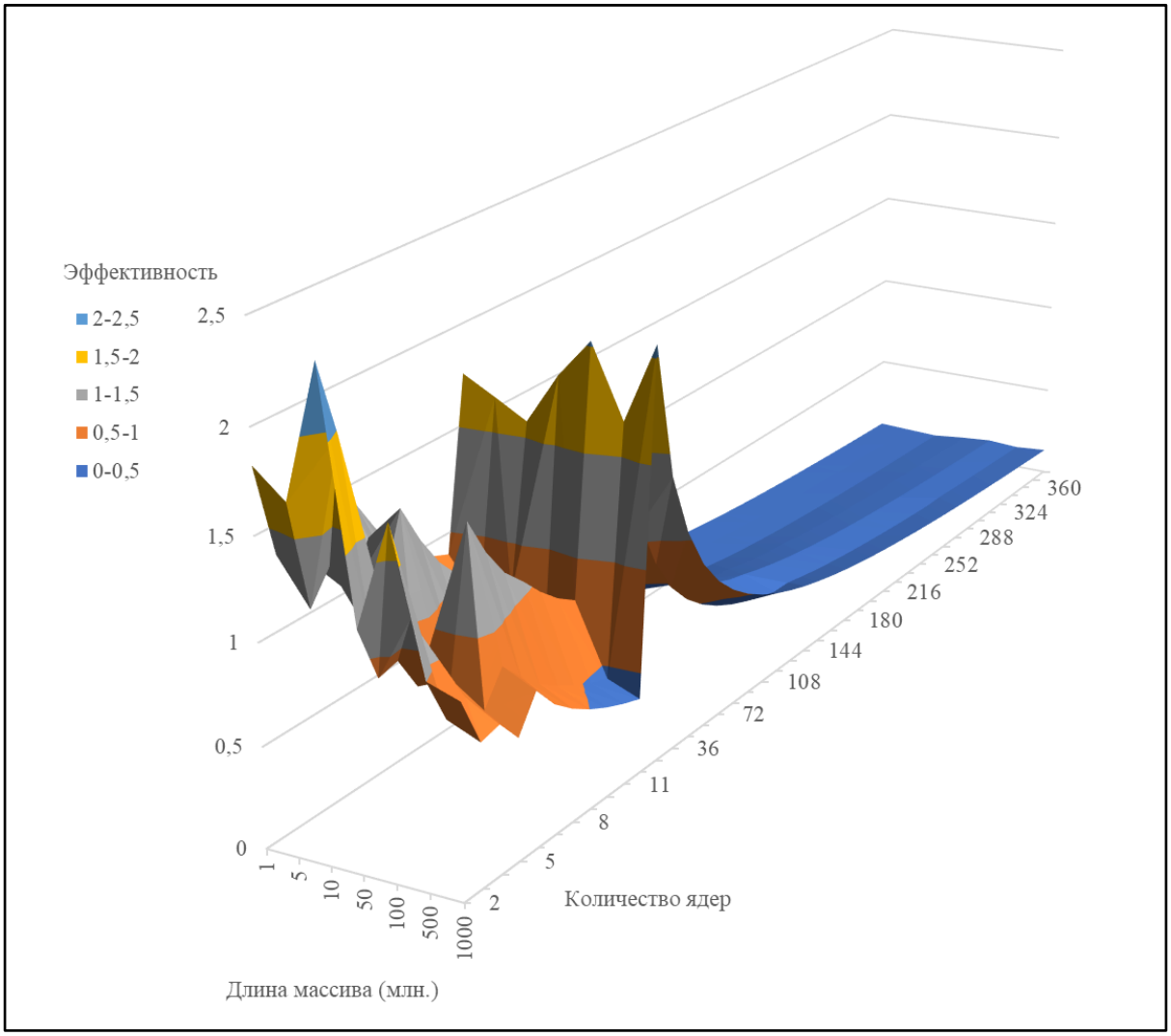

Рисунок 6 – Эффективность Q-эффективных программ для частично упорядоченных массивов