МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра параллельных вычислений

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Баранова Ильи Николаевича

Тема работы:

РАЗРАБОТКА ЭКСПЕРИМЕНТАЛЬНОЙ БИБЛИОТЕКИ РАБОТЫ С РАСПРЕДЕЛЕННЫМ МАССИВОМ ДЛЯ ПОДДЕРЖКИ АВТОМАТИЧЕСКОГО КОНСТРУИРОВАНИЯ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

«К защите допущена»
Заведующий кафедрой,
д.т.н., профессор
Малышкин В.Э. /……
(ФИО) / (подпись)
«31» мая 2022 г.

Руководитель ВКР
к.т.н., доцент
доцент каф. ПВ ФИТ НГУ
Маркова В.П./……...
(ФИО) / (подпись)
«20» мая 2022 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Киреев С.Е. /……..
(ФИО) / (подпись)
«20» мая 2022 г.

Новосибирск, 2022

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий

Кафедра параллельных вычислений

Направление подготовки 09.03.01 Информатика и вычислительная техника Направленность (профиль): Программная инженерия и компьютерные науки

УТВЕРЖДАЮ
Зав. кафедрой Малышкин В.Э.
(фамилия, И., О.)
…….
(подпись)
«22» января 2022 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Баранову Илье Николаевичу, группы 18201
(фамилия, имя, отчество, номер группы)

Тема: Разработка экспериментальной библиотеки работы с распределенным массивом для поддержки автоматического конструирования параллельных программ
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 20.01.2022 № 0012

Срок сдачи студентом готовой работы 20 мая 2022 г.

Исходные данные (или цель работы):

Разработка параллельной библиотеки работы с распределенным массивом, а также средства для автоматизации процесса подбора ее параметров на основе знаний эксперта.

Структурные части работы:

Обзор предметной области, постановка задачи, формулировка требований к программе, реализация программы, тестирование.

Руководитель ВКР
к.т.н., доцент
доцент каф. ПВ ФИТ НГУ
Маркова В.П./……...
(ФИО) / (подпись)
«21» января 2022 г.

Задание принял к исполнению
Баранов И.Н./……...
(ФИО студента) / (подпись)
«21» января 2022 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Киреев С.Е. /……..
(ФИО) / (подпись)
«21» января 2022 г.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Параметры реализации программы — это такие параметры, которые влияют на способ получение итогового результата, но не на его значение, то есть от них, например, может зависеть время выполнения программы или потребляемые ею ресурсы.

Эвристический алгоритм — алгоритм решения задачи, включающий практический метод, не являющийся гарантированно точным или оптимальным, но достаточный для решения поставленной задачи.

Функция приспособленности — вещественная или целочисленная функция одной или нескольких переменных, подлежащая оптимизации в результате работы генетического алгоритма, направляет эволюцию в сторону оптимального решения.

Градиент — вектор, своим направлением указывающий направление наибольшего возрастания некоторой скалярной величины (значение которой меняется от одной точки пространства к другой, образуя скалярное поле), а по величине (модулю) равный скорости роста этой величины в этом направлении.

Априорное распределение вероятностей неопределенной величины — распределение вероятностей, которое выражает предположения о величине до учёта экспериментальных данных. Например, если величина — доля избирателей, готовых голосовать за определенного кандидата, то априорным распределением будет предположение о величине до учета результатов опросов или выборов. Противопоставляется апостериорной вероятности.

Апостериорная вероятность — условная вероятность случайного события при условии того, что известны апостериорные данные, то есть полученные после опыта.

ВВЕДЕНИЕ

В настоящее время программисты, занимающиеся созданием параллельных программ, сталкиваются с различными сложностями их написания. В основном возникают сложности с коммуникациями между процессами: дедлоки, синхронизация, состояние гонки. Также есть трудности с фрагментацией задачи, с отображением фрагментов на ресурсы, с выбором схемы взаимодействия параллельных процессов и другие. Одним из способов решить эти проблемы является синтез параллельных программ. Таким синтезом занимается, например, система автоматического конструирования параллельных программ LuNA [1].

В задаче синтеза параллельных программ в системе LuNA есть проблема подбора параметров, которые влияют не на результат работы программы, а на её нефункциональные характеристики, такие как время работы, потребление памяти и другие. Такие параметры будем называть параметрами реализации. Подобрать приемлемые значения параметров не всегда простая задача в связи с большим количеством различных комбинаций этих параметров. Зачастую с ней может хорошо справиться только эксперт, и обычный пользователь не в состоянии подобрать параметры таким образом, чтобы добиться достаточной эффективности программы.

За решение данной проблемы брались большие компании, однако ни одно из решений не является универсальным, так как данная проблема полностью решается только с помощью полного перебора всех комбинаций параметров, а сложность такого способа экспоненциально растет с ростом количества параметров. Таким образом, данная проблема до сих пор актуальна и требует современных подходов к ее решению. Эта задача может быть упрощена в случае, когда есть знания о влиянии параметров на эффективность работы программы.

Поскольку такие знания для каждого класса задач различны, ограничимся одним из них. Для данной работы был выбран класс задач пространственной динамики на регулярных сетках. Такие задачи относятся к классу задач с

параллелизмом по данным [2]. Их параллельная реализация может быть поддержана с помощью такого средства, как распределенный массив. Параметры реализации в этих задачах и их зависимости между собой известны.

При синтезе параллельных программ частного вида, нацеленных на решение задач из некоторого класса, могут использоваться специализированные параллельные библиотеки, скрывающие детали реализации высокоуровневых конструкций. Создание параллельной параметризованной библиотеки, реализующей распределенный массив, позволит проработать подход к подбору параметров на основе знаний эксперта и автоматизировать его. Такую библиотеку можно внедрять в системы автоматического конструирования параллельных программ. Благодаря автоматизированному подбору значений параметров реализации, библиотека будет подстраиваться под различные условия платформы, на которой ее будут использовать, и показывать максимально высокую производительность.

Цель работы - разработка параллельной библиотеки работы с распределенным массивом, а также средства для автоматизации процесса подбора ее параметров на основе знаний эксперта.

Задачи:

1. Разработать библиотеку параллельного программирования для работы с распределенными массивами. Выделить параметры реализации библиотеки. Написать программу, решающую прикладную задачу с помощью разработанной библиотеки.
2. Разработать средство описания и применения знаний эксперта для подбора параметров.
3. С помощью разработанного средства провести тестирование автоматического подбора параметров написанной параллельной программы.

Глава 1. Разработка библиотеки параллельного программирования для работы с распределенными массивами

1.1 Обзор существующих средств параллельного программирования

Для того, чтобы разработать собственную библиотеку параллельного программирования, необходимо ознакомиться с уже существующими средствами и разобраться, каким требованиям должна будущая библиотека соответствовать и стоит ли заниматься ее разработкой.

1.1.1 DVM (Distribute virtual memory)

DVM-система, являющаяся российской разработкой, представляет из себя расширение языка C и Fortran, позволяющая эффективно синтезировать параллельные программы. Система включает в себя не только реализацию распределенных массивов, но и позволяет делать распараллеливание итераций цикла, организовывать обмен данных между процессами, выполнять операции редукции, а также работать с ускорителями.

Еще одной особенностью DVM является эффективная работа на различных архитектурах. По описанию разработчиков система умеет автоматически подстраиваться под возможности производительности вычислительных узлов и эффективно балансировать нагрузку между ними [3].

Данная система способна решать задачи пространственной динамики на регулярных сетках.

1.1.2 Язык параллельного программирования HPF (High Performance Fortran)

High Performance Fortran (HPF) — это расширение языка Fortran для работы с параллельными вычислениями. В код, написанный на обычном Fortran, в виде комментариев вставляются директивы для компилятора, которые содержат указания, как данные и операции над ними должны быть распределены по оперативной памяти и процессорам. Для того, чтобы применить указания из директив, необходимо компилировать программу специальным HPF компилятором.

HPF программы очень напоминают OpenMP программы из-за подхода добавления директив компилятора в последовательную программу, но OpenMP реализует распределенные вычисления, а HPF - распределенные данные [4].

Однако, HPF в наше время практически не используется в силу проблем с производительностью и устаревшей технологией компиляции, а также недостаточной гибкостью распределения данных [5, 6].

1.1.3 Coarray Fortran

На смену HPF пришел Coarray Fortran. Он является расширением Fortran, частью стандарта Fortran 2008 [7]. Особенностью данного расширения является явное распределение массивов по параллельным процессам, что увеличивает возможности программиста при написании параллельных программ, но снижает их уровень по сравнению с HPF.

1.1.4 LuNA

LuNA (Language for Numerical Algorithms) — это система автоматического конструирования параллельных программ для мультикомпьютеров, поддерживающая технологию фрагментированного программирования [8]. Язык системы является языком программирования с однократным присваиванием, который потом транслируется в C++ [1]. LuNA покрывает большое количество различных классов вычислительных задач, а производительность программ на ней можно сравнивать с оптимизированными вручную программами, написанными на чистом C++ и MPI [9].

Система не реализует концепцию распределенного массива, но покрывает решение задач пространственной динамики на регулярных сетках.

1.1.5 Язык программирования Норма

Язык Норма, также как и DVM, является российской разработкой и призван упростить создание параллельных программ, однако он является именно полноценным непроцедурным языком программирования, а не расширением к какому-то другому языку.

В качестве отличительной особенности Нормы можно выделить предназначение этого языка для математика, который в привычных ему терминах может описать задачу и выбрать метод ее решения, а организацию вычислений на конкретной машине транслятор Нормы берет на себя.

Нельзя не отметить, что язык Норма, также как и язык системы LuNA, является языком с однократным присваиванием, то есть действует правило: одна переменная - одно значение. Такой подход позволяет трансляторам осуществлять эффективное распараллеливание вычислений, а также избавляет язык от побочных эффектов [10].

1.1.6 Regent

Regent - это высокопроизводительный язык программирования для высокопроизводительных вычислений с логическими областями.

Пользователи Regent составляют программы с задачами (функции, допускающие параллельное выполнение) и логическими областями (иерархические коллекции структурированных объектов). Программы Regent создают иллюзию последовательного исполнения, не требуют явной синхронизации, и защищены от дедлоков. Типовая система Regent улавливает много распространенных классов ошибок и гарантирует, что программа с правильным последовательным выполнением дает идентичные результаты на параллельных и распределенных машинах.

Компилятор для Regent способен переводить программы Regent в эффективные реализации для Legion — асинхронной модели, основанной на задачах (an asynchronous task-based model). Regent обеспечивает производительность сравнимую с настроенным вручную Legion.

В языке Regent присутствуют две главные абстракции: tasks и logical regions. Программы Regent выглядят как обычные последовательные программы с вызовами задач, которые являются функциями (tasks), выполняющиеся параллельно. Зависимости между задачами разрешаются автоматически внутри системы, освобождая пользователя от необходимости явно синхронизировать или управлять перемещением данных по машине.

Программы Regent безопасны с точки зрения взаимных блокировок и избегают ряд классов ошибок, возможных в распределенном программировании нижнего уровня [11].

1.1.7 Выводы из обзора

Из обзора существующих решений видно, что универсальной библиотеки, решающей задачу написания параллельной программы по произвольному последовательному алгоритму, нет. Каждое средство направлено на какие-то конкретные классы задач и по-разному производит распараллеливание.

Многие из решений можно использовать для наших целей, но за разработку собственной реализации распределенного массива есть следующие аргументы:

1. Для такого простого класса задач не сложнее разработать свою библиотеку, чем изучать готовое решение.
2. Важна предсказуемость влияния параметров реализации на нефункциональные характеристики программы. Однако знаний о внутреннем устройстве готовых библиотек у нас нет и там могут быть подводные камни, ведущие к непредсказуемым результатам. В собственном решении мы сами гарантируем эту предсказуемость.
3. Собственное средство можно развивать, добавлять различные способы реализации и оптимизации.

1.2 Проектирование библиотеки

При проектировании библиотеки важно выделить специфику, требования, а также компоненты, которые будут в неё входить. Для наших целей важно заложить приемлемую производительность, масштабируемость и гибкость в настройке библиотеки.

1.2.1 Специфика библиотеки

Специфическими особенностями библиотеки является:

1. Использование языка C++. Данный язык был выбран, потому что он является одним из самых производительных языков программирования, а

также программы, написанные на нем, работают на любой операционной системе, и их можно запускать на кластере.

2. Использование интерфейса взаимодействия между процессами MPI для реализации многопроцессорности.
3. Использование стандартных нитей (std::thread) языка C++.

1.2.2 Требования к разрабатываемой библиотеке

Исходя из рассмотренных средств, а также современных тенденций развития параллельных вычислений были сформулированы следующие требования к библиотеке:

1. Возможность написание полноценной параллельной программы с использованием библиотеки без каких-либо знаний о MPI и работы с потоками в C++.
2. Возможность гибкой настройки параметров библиотеки под разные задачи, в которых используются распределенные массивы.
3. Сравнимая с вручную оптимизированной программой производительность библиотеки.
4. Наличие распределенного массива и средств для работы с его элементами в глобальных координатах.

1.2.3 Компоненты библиотеки

Поскольку в библиотеке предполагается использовать технологию MPI, должен быть компонент, отвечающий за вызов функции MPI_Init — инициализатор библиотеки. При завершении работы с библиотекой необходимо вызвать функцию MPI_Finalize, за это также будет отвечать компонент-инициализатор.

Основным элементом библиотеки является распределенный массив. Его реализация должна зависеть от ряда параметров, определяющих его размеры, тип данных, размерность, тип декомпозиции и другие свойства. Это позволит гибко подстраивать массив под разные задачи. В этом же компоненте можно заложить выделение памяти под теневые грани и возможность их передачи

между процессами как синхронно, так и асинхронно. Теневые грани необходимы, чтобы получать доступ к элементам, которые не входят в локальную часть распределенного массива, но они необходимы для решения задач пространственной динамики. В этих задачах значение элемента в каждой ячейке зависит от значений некоторых соседних ячеек, а “соседи” могут не попасть в локальную часть массива, поэтому их нужно передавать.

В условиях распределенной памяти стоит задача работы с массивом в глобальных координатах. Решение этой задачи напрямую сильно ухудшит производительность программы по сравнению с последовательной — необходимо будет проверять элемент на принадлежность его к локальной части массива при каждом обращении. Эту проблему решает вспомогательный компонент Range, который автоматически определяет, с какими локальными индексами нужно работать. С его помощью вместо работы с отдельными элементами массива мы работаем с множеством элементов в виде n-мерного прямоугольного параллелепипеда, что позволяет выполнять проверки на принадлежность локальному подмассиву не для каждого элемента, а для всего множества сразу, и таким образом снизить накладные расходы.

Работу внутри локальной части массива в каждом процессе можно распараллелить с помощью потоков и использования общей памяти. Для этого в каждом процессе можно создать пул потоков, которые будут синхронизироваться после прохода очередной итерации, чтобы не создать состояния гонки. Таким образом можно достичь максимальной производительности на процессорах с гипертредингом.

Возможность применения операций редукции также можно вынести в отдельный компонент, поскольку они не зависят от других частей библиотеки. Редукция в задачах пространственной динамики нужна, в основном, чтобы сравнивать значения в ячейках текущей и предыдущей итерации для определения конца итерационного алгоритма.

1.3 Реализация библиотеки

На основе проекта с учетом требований и специфики была реализована библиотека параллельного программирования для работы с распределенными массивами, реализующая спроектированные компоненты.

1.3.1 Инициализация библиотеки. Класс ParaHelper

В начале программы, написанной с помощью разрабатываемой библиотеки, необходимо создать объект класса ParaHelper, который инициализирует библиотеку. Данный класс в конструкторе скрывает вызов функции MPI_Init, а в деструкторе MPI_Finalize, без этих функций не обходится ни одна MPI программа. Интерфейс класса ParaHelper представлен на рисунке 1.

```
class ParaHelper {
public:
    ParaHelper(int argc, char** argv);
    ~ParaHelper();
    int getNumberOfProcceses(MPI_Comm communicator);
    int getCurrentProccesNumber(MPI_Comm communicator);
    int getNumberOfProcceses();
    int getCurrentProccesNumber();
    int getParametr(std::string paramName, int defaultValue);
    void writeStreamToFile(std::stringbuf * visualStream, std::string dirName);
private:
    char** find(char** first, char** last, const std::string& value);
    char* getCmdOption(const std::string& option);
    char** argv;
    int argc;
};
```

Рисунок 1 — Интерфейс класса ParaHelper

Данный класс также содержит необходимые для отладки программы функции, такие как получение количества процессов, а также номер текущего процесса. Для отладки также может понадобится визуализация процесса расчетов, для этого добавлена функция writeStreamToFile. С её помощью можно записать данные из строкового буфера в файл с учетом номера записывающего процесса.

1.3.2 Распределенный массив. Класс Grid

Grid - класс, включающий в себя распределенный массив данных и методы работы с ним. Класс является шаблонным, в качестве параметров шаблона выступают размерность массива, а также тип данных массива. Создать объект класса Grid можно разными способами: передав размерности в виде массива или списком инициализации, а также передав объект специального класса DecompositionType, отвечающего за способ декомпозиции массива. По умолчанию используется линейная декомпозиция по оси X. Надо отметить, что реальный размер массива напрямую зависит от способа декомпозиции, так как выделяется дополнительная память под теневые грани.

К локальным элементам массива можно обращаться с помощью перегруженного оператора (), который в общем случае принимает массив индексов. Интерфейс класса Grid можно видеть на рисунке 2.

```
template<int dimention, typename T>
class Grid {
public:
    ~Grid();
    Grid(int* sizes, DecompositionType decompositionType);
    Grid(std::initializer_list<int> sizes, DecompositionType decompositionType);
    Grid(int* sizes);
    Grid(std::initializer_list<int> sizes);

    T& operator()(int i, int j);
    const T operator()(int i, int j) const;
    T& operator()(int i, int j, int k);
    const T operator()(int i, int j, int k) const;
    T& operator()(const int indexes[dimention]);
    const T operator()(const int indexes[dimention]) const;
    void print();
    void fullPrint();
    std::stringstream visualPrint(int iterationNumber);
    std::stringstream visualPrintWithBorders(int iterationNumber);
    int length();
    int getSize(int coordinateNumber);
    std::pair<int,int> startPoint;
    XYPair getShift();
    int shift();
    Waiter send_grid_bound_async();
    void send_grid_bound();

    std::array<int, dimention> globalSizes;
    DecompositionType decomposition_type;

    XYPair processCoord;
    int* processDimSizes;

    void swap(Grid& anotherGrid);
```

Рисунок 2 — интерфейс класса Grid

Для обмена теневыми гранями предусмотрено два метода send_grid_bound_async и send_grid_bound. Они служат для асинхронного и синхронного обмена соответственно.

1.3.3 Способ декомпозиции. Класс DecompositionType

Распределенный массив Grid может разными способами распределяться по процессам, за эти способы отвечает класс DecompositionType. Он поддерживает как линейную, так и двухмерную декомпозицию с возможностью выбора топологии процессов. Интерфейс данного класса представлен на рисунке 3.

```
class DecompositionType
{
public:
    DecompositionType();
    DecompositionType(std::initializer_list<bool> dim_params,
    std::initializer_list<int> dims_create_params);
    DecompositionType(MPI_Comm communicator,
    std::initializer_list<bool> dim_params,
    std::initializer_list<int> dims_create_params);

    int * get_grid_sizes(int nnodes, int grid_dim);
    void init_grid_sizes(int * grid_sizes, int grid_dim, int* process_grid);
    bool getDimParam(int index);
    bool getDimParam(int index) const;
    int shift();
    std::pair<int,int> getStartPoint(Grid<2,double> grid);
    XYPair getProcessCoord(int rank, int * sizes);
    MPI_Comm communicator;
    int rank;
    int commsize;

    int getProcessRankByCoord(XYPair coord, int * sizes);

private:
    std::vector<bool> dim_params; // for 3D {true, false, true}
    std::vector<int> dims_create_params; // for 2D (8x8) {0,0} или {2,0}
};
```

Рисунок 3 — Интерфейс класса DecompositionType

В качестве первого параметра конструктора данного класса нужно указать в виде списка инициализации те оси, по которым нужно делать декомпозицию: true — делать декомпозицию по оси, false — не делать. В качестве второго параметра передается список инициализации, определяющий количество

процессов по каждой координате решетки процессов. Можно опционально передать MPI коммуникатор, по умолчанию MPI_COMM_WORLD.

Например, на рисунке 4 создается объект класса DecompositionType, определяющий, что массив будет декомпозирован по оси X линейно, топология процессов будет выбрана автоматически.

```
DecompositionType decType({ true,false }, { 0,0 });
```

Рисунок 4 — Пример создания объекта класса DecompositionType

1.3.4 Обмен границами. Класс Waiter

Как можно было заметить, метод send_grid_bound_async класса Grid возвращает объект класса Waiter. Данный класс имеет метод wait(), который отвечает за ожидание завершения операции обмена границами. Метод возвращает значение типа void, а если функция возвращается к вызываемой программе, это означает, что процесс обмена границами завершился, и можно приступить к работе с полученными данными. Таким образом, в библиотеке реализован асинхронный обмен границами. Для асинхронного обмена используется MPI_Request как видно из рисунка 5.

```
class Waiter {
public:
    Waiter(int count);
    Waiter(const Waiter &waiter);
    ~Waiter();
    void wait();

    MPI_Request* getRequest(int n);

private:
    MPI_Request* requests;
    int requestCount;
};
```

Рисунок 5 — Интерфейс класса Waiter

Примеры работы с синхронным и асинхронным обменом границами можно увидеть на рисунках 6 и 7. Использование асинхронного обмена

является более эффективным подходом к написанию параллельной программы, так как можно выполнять вычисления не дожидаясь завершения обмена, например, вычисляя значения центральных ячеек массива.

```
Grid<2,double> multiGrid({ nx,ny });

do {
    multiGrid.send_grid_bound();

    //some work

} while (condition)
```

Рисунок 6 — Пример работы с синхронным обменом границ

```
Grid<2,double> multiGrid({ nx,ny });

do {
    Waiter waiter = multiGrid.send_grid_bound_async();
    //some work in centre
    waiter.wait();
    //some work in bounds
} while (condition)
```

Рисунок 7 — Пример работы с асинхронным обменом границ

1.3.5 Работа с областями массива. Класс Range

Одним из требований к библиотеке была возможность удобной и эффективной работы с распределенным массивов в глобальных координатах, за реализацию такой возможности и отвечает класс Range. Интерфейс данного класса показан на рисунке 8.

```
template<int dimention>
class Range {
public:
    Range();
    Range(std::initializer_list<int> list);
    Range(std::initializer_list<int> startIndexesList, std::initializer_list<int> lastIndexesList);

    void for_each(Grid<2, double> * grid, std::function<void(const int[dimention])> function);
    void for_each(Grid<2, double> * grid, ThreadPool* pool, std::function<void(const int[dimention])> function);
    void for_each(Grid<2, double> * grid, std::initializer_list<int> order, std::function<void(const int[dimention])> function);
    void for_each(Grid<2, double> * grid, std::initializer_list<int> order, ThreadPool* pool, std::function<void(const int[dimention])> function);

    int getStartIndex(int coordinate);
    int getLastIndex(int coordinate);
private:
    std::array<int, dimention> startIndexes;
    std::array<int, dimention> lastIndexes;

    StartLastXY getStartLastXY(Grid<2, double> * grid);
    StartLastXY getStartLastXYDouble(Grid<2, double> * grid);

    void for_each_reverse(Grid<2, double>* grid, std::function<void(const int[dimention])> function);
    void for_each_reverse(Grid<2, double>* grid, ThreadPool* pool, std::function<void(const int[dimention])> function);
};
```

Рисунок 8 — Интерфейс класса Range

Данный класс является шаблонным и принимает в качестве параметра шаблона размерность области. Создавая объект класса Range, необходимо указать координаты начальной и конечной точек. На рисунке 9 длина K = f1 - s1, L = f2 - s2. Range может быть произвольных размеров и не зависит от массивов, которые нужно будет обходить.

```
Range<2> range({s1,s2},{f1,f2});
```

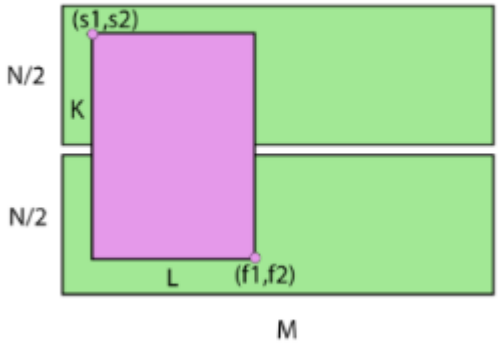

Рисунок 9 — Пример создания области Range

Основным методом данного класса является метод for_each, с помощью которого можно выполнить обход области Range. Он принимает в качестве аргументов “опорный” распределенный массив типа Grid, содержащий важные знания о типе декомпозиции, а также лямбда-функцию, задающую операцию

над элементами индексного пространства. В примере на рисунке 10 показано, как с помощью метода for_each можно заполнить область Range единицами.

```
Grid<2,double> multiGrid({N,M});
Range<2> range({s1,s2},{f1,f2});
range.for_each(&multiGrid, [&multiGrid](const int indexes[2]) {
    multiGrid(indexes) = 1;
});
```

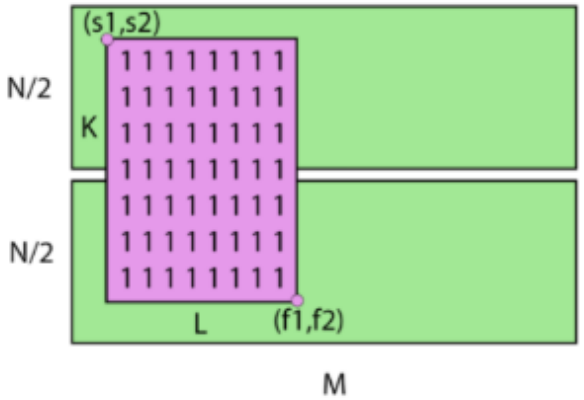

Рисунок 10 — Пример работы метода for_each

Опционально в метод for_each можно передать объект класса ThreadPool, являющийся пулом потоков. В таком случае обход области будет автоматически распараллелен по потокам, что в ряде случаев может ускорить время выполнения вычислений. Порядок циклов обхода также может быть изменен, для этого в метод передается список инициализации с указанием порядка циклов. Например, список {2,0,1} будет означать, что ось Z будет внешней, далее будет идти ось X и самой внутренней будет ось Y. По умолчанию порядок будет {0,1,2}.

1.3.6 Операции редукции. Класс Reduce

Ни одну библиотеку параллельного программирования высокого уровня невозможно представить без коллективных операций. Одной из них является редукция, за неё отвечает класс Reduce. Объект этого класса создавать не нужно, его основные методы являются статическими. Шаблонный метод reduce принимает в качестве параметров объект класса Range, “опорный” массив Grid, лямбда-функцию и, опционально, MPI коммуникатор. Параметрами шаблона

метода является размерность, тип данных и enum класс Operation. Поддерживаемые операции редукции: сумма (SUM), максимальное значение (MAX), минимальное значение (MIN). Интерфейс класса Reduce показан на рисунке 11.

По умолчанию в качестве MPI коммуникатора используется MPI_COMM_WORLD.

```
class Reduce
{
public:

    template<int dim, typename T, Operation Op>
    static T reduce( Range<dim>& rng, Grid<2, double> * grid, std::function<T(const int[dim])> f, MPI_Comm comm);

    template<int dim, typename T, Operation Op>
    static T reduce( Range<dim>& rng, Grid<2, double> * grid, std::function<T(const int[dim])> f);

    template<Operation operation>
    class get_mpi_op
    {
    public:
        static MPI_Op value;
    };

    template<Datatype datatype>
    class get_mpi_type {
    public:
        static MPI_Datatype value;
    };

    template<typename T, Operation operation>
    class initial_value
    {
    public:
        static T value;
    };

    template<typename T, Operation operation>
    class perform_op
    {
    public:
        T operator()(T value, T func_result);
        static T perform(T value, T func_resutl);
    };

};
```

Рисунок 11 — Интерфейс класса Reduce

1.3.7 Визуализация работы библиотеки

Для отладки программы можно собрать данные о значениях в каждой ячейке распределенного массива и визуализировать итерационный процесс целиком. Специально для этих целей был написан визуализатор на языке C# с использованием библиотеки Windows Forms.

В ходе работы программы можно сохранить текстовые файлы с результатами выполнения программы, в том числе данные о номерах процессов, номере итерации и значении элементов массива. По этим данным легко восстанавливается весь итерационный процесс. На рисунках 12 и 13 разным цветом показаны части массива, принадлежащие разным процессам, в зависимости от выбранной декомпозиции. Можно заметить, что значения в ячейках массива на каждой итерации не отличаются при выборе разной декомпозиции.

	0	1	2	3	4	5	6	7	8	9	10	11
0	2	1,66942	1,40496	1,20661	1,07438	1,00826	1,00826	1,07438	1,20661	1,40496	1,66942	2
1	1,66942	1,33882	1,07436	0,876012	0,743781	0,677665	0,677665	0,743781	0,876012	1,07436	1,33882	1,66942
2	1,40496	1,07436	0,809896	0,611549	0,479318	0,413203	0,413203	0,479318	0,611549	0,809896	1,07436	1,40496
3	1,20661	0,876012	0,611549	0,413202	0,280971	0,214856	0,214856	0,280971	0,413202	0,611549	0,876012	1,20661
4	1,07438	0,743781	0,479318	0,280971	0,14874	0,0826244	0,0826244	0,14874	0,280971	0,479318	0,743781	1,07438
5	1,00826	0,677665	0,413203	0,214856	0,0826244	0,0165088	0,0165088	0,0826244	0,214856	0,413203	0,677665	1,00826
6	1,00826	0,677665	0,413203	0,214856	0,0826244	0,0165088	0,0165088	0,0826244	0,214856	0,413203	0,677665	1,00826
7	1,07438	0,743781	0,479318	0,280971	0,14874	0,0826244	0,0826244	0,14874	0,280971	0,479318	0,743781	1,07438
8	1,20661	0,876012	0,611549	0,413202	0,280971	0,214856	0,214856	0,280971	0,413202	0,611549	0,876012	1,20661
9	1,40496	1,07436	0,809896	0,611549	0,479318	0,413203	0,413203	0,479318	0,611549	0,809896	1,07436	1,40496
10	1,66942	1,33882	1,07436	0,876012	0,743781	0,677665	0,677665	0,743781	0,876012	1,07436	1,33882	1,66942
11	2	1,66942	1,40496	1,20661	1,07438	1,00826	1,00826	1,07438	1,20661	1,40496	1,66942	2

Рисунок 12 — Визуализация распределенного массива с линейной декомпозицией по оси Y

	0	1	2	3	4	5	6	7	8	9	10	11
0	2	1,66942	1,40496	1,20661	1,07438	1,00826	1,00826	1,07438	1,20661	1,40496	1,66942	2
1	1,66942	1,33882	1,07436	0,876012	0,743781	0,677665	0,677665	0,743781	0,876012	1,07436	1,33882	1,66942
2	1,40496	1,07436	0,809896	0,611549	0,479318	0,413203	0,413203	0,479318	0,611549	0,809896	1,07436	1,40496
3	1,20661	0,876012	0,611549	0,413202	0,280971	0,214856	0,214856	0,280971	0,413202	0,611549	0,876012	1,20661
4	1,07438	0,743781	0,479318	0,280971	0,14874	0,0826244	0,0826244	0,14874	0,280971	0,479318	0,743781	1,07438
5	1,00826	0,677665	0,413203	0,214856	0,0826244	0,0165088	0,0165088	0,0826244	0,214856	0,413203	0,677665	1,00826
6	1,00826	0,677665	0,413203	0,214856	0,0826244	0,0165088	0,0165088	0,0826244	0,214856	0,413203	0,677665	1,00826
7	1,07438	0,743781	0,479318	0,280971	0,14874	0,0826244	0,0826244	0,14874	0,280971	0,479318	0,743781	1,07438
8	1,20661	0,876012	0,611549	0,413202	0,280971	0,214856	0,214856	0,280971	0,413202	0,611549	0,876012	1,20661
9	1,40496	1,07436	0,809896	0,611549	0,479318	0,413203	0,413203	0,479318	0,611549	0,809896	1,07436	1,40496
10	1,66942	1,33882	1,07436	0,876012	0,743781	0,677665	0,677665	0,743781	0,876012	1,07436	1,33882	1,66942
11	2	1,66942	1,40496	1,20661	1,07438	1,00826	1,00826	1,07438	1,20661	1,40496	1,66942	2

Рисунок 13 — Визуализация распределенного массива с двумерной декомпозицией и топологией процессов 3x4

1.4 Тестирование библиотеки на прикладной задаче

1.4.1 Описание прикладной задачи

В качестве решаемой с помощью библиотеки задачи была выбрана реализация решения двумерного уравнения Гельмгольца методом Якоби. Требуется решить уравнение (то есть найти функцию φ)

$$\frac{\partial^2\varphi}{\partial^2 x^2} + \frac{\partial^2\varphi}{\partial^2 y^2} - a\varphi = \rho,\ a \geq 0 \quad (1)$$

$$\varphi = \varphi(x,y),\ \rho = \rho(x,y)$$

в прямоугольной области Ω с краевыми условиями 1-го рода (т.е. на границе G известны значения искомой функции φ):

$$\varphi|_G = F(x,y)$$

Для численного решения задачи необходимо перейти к ее дискретному аналогу. Для этого введем в области Ω равномерную прямоугольную сетку размером Nx×Ny узлов. Шаги сетки (расстояния между соседними узлами) по осям X, Y будут равны:

$$h_x = \frac{D_x}{N_x - 1} \qquad h_y = \frac{D_y}{N_y - 1}$$

Тогда координаты узла с произвольными индексами i, j вычисляются следующим образом:

$$x_i = x_0 + i \cdot h_x, \qquad y_j = y_j + j \cdot h_y$$

где x_0, y_0 – начальные координаты области Ω, i = 0,…,N_x, j = 0,…,N_y. Вместо непрерывных функций φ(x,y,z), ρ(x,y,z), F(x,y,z) в области Ω вводятся сеточные функции $\varphi_{i,j}$, $\rho_{i,j}$,$\rho_{i,j}$, заданные в узлах сетки:

$$\varphi_{i,j} = \varphi(x_i, y_j) \qquad \rho_{i,j} = \rho(x_i, y_j) \qquad F_{i,j} = F(x_i, y_j)$$

На выбранной сетке запишем разностную схему – дискретный аналог уравнения (1):

$$\frac{\varphi_{i+1,j} - 2\varphi_{i,j} + \varphi_{i-1,j}}{h_x^2} + \frac{\varphi_{i,j+1} - 2\varphi_{i,j} + \varphi_{i,j-1}}{h_y^2} - a\varphi_{i,j} = \rho_{i,j}$$

Полученная разностная схема связывает значения искомой функции в узлах сетки в некоторой локальной окрестности. Если выразить из нее φi,j, можно получить формулу для итерационного процесса метода Якоби (здесь m – номер итерации):

$$\varphi_{i,j}^{m+1} = \frac{1}{\frac{2}{h_x^2}+\frac{2}{h_y^2}}\left(\frac{\varphi_{i+1,j}^{m}+\varphi_{i-1,j}^{m}}{h_x^2} + \frac{\varphi_{i,j+1}^{m}+\varphi_{i,j-1}^{m}}{h_y^2} - \rho_{i,j}\right)$$

Если, начиная с некоторого начального приближения, многократно вычислять значения $\varphi_{i,j}$ по формуле (2), то они будут постепенно приближаться к искомому решению. Условие завершения итерационного процесса по достижению некоторого порога сходимости:

$$max_{i,j}\left|\varphi_{i,j}^{m+1} - \varphi_{i,j}^{m}\right| < \varepsilon$$

1.4.2 Реализация прикладной задачи с помощью разработанной библиотеки

Для решения задачи написана программа с помощью разработанной библиотеки. Использовалась линейная декомпозиция по оси X и синхронный обмен границами, распараллеливание по потокам не производилось.

В качестве порога сходимости ε было взято число 10^-6, параметр уравнения A равнялся 10^5. Не считая инициализации, основная часть программы показана на рисунке 14.

```
std::function<void(const int[2])> main_function = [&u1, &u2,nx,ny,HX,HY](const int indexes[2]) -> void {
    int i = indexes[0];
    int j = indexes[1];
    int r1 = i + u2.getShift().X;
    int r2 = j + u2.getShift().Y;

    u2(i, j) = (1.0 / (2.0 / (HX * HX) + 2.0 / (HY * HY) + A)) *
        ((u1(i + 1, j) + u1(i - 1, j)) / (HX * HX) +
            (u1(i, j + 1) + u1(i, j - 1)) / (HY * HY) -
            ro(r1, r2, nx, ny));
};

do {
    u1.send_grid_bound();

    r_in.for_each(&u1, {xOrder, yOrder}, &pool, main_function);
    diff = Reduce::reduce<2, double, Operation::MAX>(r, &u2, diff_function);
    u2.swap(u1);

    if (progressParam) {
        if (iterations % 100 == 0) {
            printProgress(iterations, diff, parahelper);
        }
    }

    iterations++;
} while (diff >= E );
```

Рисунок 14 — Реализация прикладной задачи с помощью библиотеки

По результатам тестирования, показанным в таблице 1, убедились, что в зависимости от количества процессов производительность возрастает.

Таблица 1 — соотношения количества процессов и времени выполнения на сетке 240x2400

Количество процессов	Время выполнения, с	Ускорение
1	21.27	-
2	12.46	1,7
4	6.66	3,19
8	3.55	5,99

При тестировании программы на разных размерах сетки определенная заданной ε ошибка была незначительна (< 10^-6) и совпадала при различных параметрах реализации.

1.5 Выводы

Разработанная библиотека показала свою работоспособность на тестовой задаче. В библиотеке можно выделить следующие параметры реализации:

- количество процессов;
- количество потоков на процесс;
- способ декомпозиции массива;
- параметры, отвечающие за задание топологии процессов при декомпозиции;
- параметры, отвечающие за порядок циклов при обходе индексного пространства локальных частей распределенного массива.

Значения выделенных параметров влияет на нефункциональные свойства программы и требуют подбора для достижения приемлемой производительность программы.

Глава 2. Разработка средства подбора оптимальных параметров

Убедившись в правильности работы библиотеки, можно приступить к разработке средств для ее оптимизации под конкретную задачу. В этой главе будет приведен обзор современных подходов к подбору параметров и средств подбора параметров. На основе обзора будет описан алгоритм их подбора с применением знаний эксперта.

2.1 Обзор современных подходов к подбору оптимальных параметров

В некоторых областях программирования, таких как машинной обучение, разработка нейронных сетей, параллельные вычисления, часто требуется подобрать оптимальные параметры: веса нейронов, параметры модели машинного обучения, соотношение количества потоков и процессов в параллельной программе и другие. Для решения этой задачи существует множество подходов, в этом разделе будут рассмотрены самые популярные из них.

2.1.1 Ручной подбор параметров

Данный метод заключается в подборе параметров на основании интуиции, опыте или догадках. Подбор завершается, когда заканчивается терпение или найденная комбинация параметров становится достаточно удовлетворительной [12]. Из плюсов можно выделить простоту применения этого метода и отсутствие необходимости использовать сторонние средства или реализовывать свои. Минус очевиден: данный метод не гарантирует получение приемлемого результата за обозримое время.

2.1.2 Поиск по сетке

Сеточный поиск предполагает построение n-мерной сетки, в узлах которой находятся комбинации значений параметров, после чего данные комбинации перебираются с некоторым шагом по каждому из параметров. Если шаг для каждого из параметров равняется минимально возможной длине шага, то перебираться будут все возможные комбинации параметров. Плюсами данного подхода являются достаточно простая реализация алгоритма подбора,

возможность полного перебора параметров, тривиальность распараллеливания. К основным недостаткам можно отнести низкую скорость поиска, проверка заведомо неудачных комбинаций параметров, а если увеличивать шаг обхода, то велика вероятность пропустить оптимальную комбинацию. Поиск по сетке часто применяется в низкоразмерных пространствах: одномерном или двумерном [12].

2.1.3 Случайный поиск

Данный метод заключается в выборе комбинаций параметров случайным образом из выборки с заданным распределением. В большинстве случаев случайный поиск оказывается эффективнее поиска по сетке [12, 13], однако существует большой риск перебора заведомо неудачных комбинаций. Обычно, случайный поиск ограничивается либо по времени, либо по достижению достаточно удовлетворительного результата.

2.1.4 Генетический метод

Эвристические алгоритмы поиска параметров, использующие случайный поиск, комбинирование параметров, а далее отбор наиболее оптимальных комбинаций, известны как генетические алгоритмы. Сначала случайным образом выбирается популяция — некоторый набор комбинаций параметров, затем каждая комбинация оценивается с помощью функции приспособленности, которая определяет насколько та или иная комбинация хороша. Затем, на основании значения функции приспособленности, выбираются наилучшие комбинации, этот процесс называется отбором. Далее отобранные комбинации “скрещивают” с другими, получая новый набор и снова оценивают его с помощью функции приспособленности. Способ скрещивания комбинаций зависит от задачи. Для преодоления недостатков популяции применяются случайные мутации. Процедура повторяется, пока не будет получена наиболее оптимальная комбинация. Генетические алгоритмы хорошо оптимизируются, за счет оптимизации функции приспособленности, и

распараллеливаются, за счет возможности скрещивать различные комбинации и вычислять их функции приспособленности параллельно [14].

2.1.5 Градиентный спуск

Метод градиентного спуска используется для минимизации некоторой функции путем итерационного спуска к минимальному значению, определяя направление спуска с помощью отрицательного значения градиента. Алгоритм заканчивает работу, когда невозможно спускаться ниже, то есть достигается локальный минимум. Проблема метода состоит в том, что локальный минимум может оказаться не самой наилучшей точкой пространства параметров, определяющей искомую комбинацию, однако такой подход, в большинстве случаев, отлично справится с задачей, в которой функция имеет единственный минимум. Еще одной проблемой является необходимость вычисления градиента искомой функции, поскольку это может быть ресурсозатратной процедурой [15].

2.1.6 Байесовская оптимизация

Метод байесовской оптимизации способен работать в том числе со стохастическими, вогнутыми и даже разрывными функциями и он не требует вычисление градиента [16].

Ничего не зная про функцию, описывающую пространство параметров, метод байесовской оптимизации принимает ее за случайную функцию и размещает априорное распределение вероятностей над ней. Далее алгоритм фиксирует знания о предполагаемом распределении значений целевой функции. После сбора оценок функции, которые обрабатываются как данные, априорное значение обновляется для формирования апостериорного распределения по целевой функции. Апостериорное распределение, в свою очередь, используется для построения функции сбора данных (часто также называемой критерием выборки заполнения), которая определяет следующую точку запроса. Таким образом, метод выбирает область поиска на основе опыта предыдущих

экспериментов, уменьшая их количество, что делает этот метод эффективным по сравнению с другими в общем случае.

2.2 Обзор современных средств для подбора оптимальных параметров на основе знаний

Рассмотренные методы подбора значений параметров хороши в случае, когда про задачу практически ничего не известно, а пользователь знает только допустимые значения. А что, если пользователем будет эксперт в задаче, которую он решает, и он будет знать свойства подбираемых параметров? В таком случае любой из методов можно оптимизировать под конкретную задачу и он будет эффективнее находить оптимальные параметры.

2.2.1 Active Harmony

Active Harmony — это фреймворк для языка C, позволяющий подбирать параметры для функций языка C. Active Harmony делит задачу поиска оптимальных точек на две ключевые абстракции: стратегию поиска и слои обработки.

Стратегия поиска определяет, как новые точки выбираются из пространства поиска. Active Harmony реализует несколько стратегий поиска, каждая из которых обладает различными свойствами для поддержки широкого спектра клиентских приложений. Стратегии поиска работают исключительно на числовом уровне, сопоставляя точки пространства поиска с значениями производительности. Они не имеют представления о том, как точка будет использоваться клиентом. В системе есть возможность использования трех стратегий поиска: полный перебор, случайный поиск и метод Нелдера — Мида [17].

Слои обработки обрабатывают любые дополнительные задачи, которые должны выполняться либо до, либо после генерации точки. Например, есть слой, записывающий пары “точка - производительность” в файл после генерации точки.

Active Harmony обеспечивает гибкость за счет реализации стратегий поиска и обработки слоев в виде подключаемых модулей, загружаемых во время сеанса настройки. Такая структура позволяет Active Harmony удовлетворять потребности любого приложения автоматической настройки с минимальными усилиями. Специализированный автотюнер может быть эффективно собран по частям [18].

2.2.2 Intel MPI tuning

Intel MPI tuning — это утилита, позволяющая оптимизировать работу программ, написанных с помощью библиотеки Intel MPI Library. Настройка очень сильно зависит от технических характеристик конкретной платформы. Автоматический тюнер ищет наилучшую возможную реализацию коллективных MPI операций во время выполнения приложения. Каждая коллективная операция имеет свои собственные предварительные настройки, которые состоят из алгоритма и его параметров. Утилита оценивает производительность каждой из них. После того, как автонастройка оценила пространство поиска, она выбирает самую быструю реализацию и использует ее для до завершения выполнения приложения, что повышает производительность программы [19].

Таким образом, данная утилита подбирает параметры реализации MPI процедур на основе знаний о характеристиках платформы исполнения программы, а также о параметрах, которые подбираются.

2.2.3 ATLAS

ATLAS (Automatically Tuned Linear Algebra Software) — это программная библиотека для линейной алгебры. Она предоставляет реализацию API-интерфейсов BLAS с открытым исходным кодом для C и Fortran 77. С помощью этой библиотеки можно генерировать оптимизированную реализацию BLAS на новых операционных системах и платформах. Генерация осуществляется основываясь на параметрах платформы, например, размер кэша. Также, в библиотеки реализовано несколько различных вариантов

функций, среди которых выбирается наиболее производительная для данной платформы. Используется и автоматическая генерация исходного кода на основе знаний о системе [20].

2.2.4 FFTW

FFTW (Fastest Fourier Transform in the West) представляет собой программную библиотеку для вычисления дискретных преобразований Фурье. Она поддерживает множество алгоритмов для вычисления и выбирает наиболее оптимальный из них в конкретных условиях работы.

Для достаточно большого числа повторяющихся преобразований выгодно измерять производительность некоторых или всех поддерживаемых алгоритмов на заданном размере массива и платформе. Эти измерения, которые авторы называют "мудростью", могут быть сохранены в файле или строке для последующего использования [21].

2.3 Проектирование средства для автоматизированного подбора параметров

Все рассмотренные средства каким-то образом используют знания о подбираемых параметрах, а не только их допустимые значения. Однако, каждое из них имеет свои ограничения. В этой работе данный подход обобщается таким образом, чтобы его можно было применить на любой готовой программе.

Чтобы дать возможность эксперту описать свой собственный алгоритм подбора параметров нужно предоставить ему инструмент. Создавать фреймворк для какого-то существующего языка программирования, как это сделано в Active Harmony, не очень хорошая идея, поскольку с таким подходом приходится модифицировать существующую программу.

В качестве инструмента можно использовать специальный скриптовый интерпретируемый язык программирования, интерпретатор которого будет работать на большинстве операционных систем. С помощью него описываются возможные значения подбираемых параметров, а также правила, по которым

они будут подбираться. Далее по описанному в скрипте сценарию будет подобрана наилучшая комбинация параметров.

Исходя из обзора современных методов подбора оптимальных параметров были сформулированы требования для разрабатываемого средства:

1. Возможность добавлять и убирать конкретные комбинации параметров в итоговую выборку для перебора. Это позволяет имитировать процесс подбора параметров вручную, но его можно описать в виде скрипта.
2. Возможность использовать перебор параметров по сетке с любым шагом по любому параметру.
3. Возможность использовать различные методы поиска.
4. Возможность установления правил подбора с помощью условий или вычисляемых параметров.
5. Возможность работы средства в условиях ограниченного доступа к прямому запуску программы, например в распределенных системах с собственным планировщиком задач.
6. Возможность работы с программой как с черным ящиком, не прибегая к её модификации.

2.4 Реализация средства подбора параметров

Было разработано средство подбора параметров, соответствующее сформулированным требованиям. см. приложение А. Синтаксис скриптового языка программирования для описания алгоритма подбора параметров должен обладать простотой, чтобы пользователь без труда написал свою первую программу и быстро начал пользоваться разработанным средством. В этом разделе будут описаны синтаксические конструкции языка.

2.4.1 Описание параметров программы

Язык предполагает описание параметров с помощью конструкции **ADD PARAM**. Причем с помощью этой конструкции описываются не только подбираемые параметры, но и неизменяемые, например, исходные параметры задачи, такие как размерность сетки.

Эта конструкция также используется для описания имени исполняемого файла программы, используя ключевое слово **PROGNAME** вместо имени параметра. Таким же образом определяется метрика, значение которой оптимизируется, для нее используется ключевое слово **METRIC** в качестве имени параметра.

Допустимые значения описываются двумя способами: с помощью перечисления (оператор **ENUM**) и с помощью задания диапазона (оператор **RANGE**). Можно использовать любую комбинацию этих описаний, например, если нам нужно перебрать параметры в двух непересекающихся диапазонах. На рисунке 15 представлен вариант описания параметра *-test*, значения которого лежат в диапазоне [0;10] с шагом 1, [20;30] с шагом 2, а также конкретные значения 50,60,70.

```
ADD PARAM : -test RANGE(0->10|1) RANGE(20->30|2) ENUM(50,60,70)
```

Рисунок 15 — Пример описания допустимых значений параметра

2.4.2 Метод определения выборки

Особенностью подхода к получению итоговой выборки комбинаций разработанного средства является выбор конкретных комбинаций по правилам из начальной выборки. Эту начальную выборку можно получить либо с помощью метода поиска по сетке, либо выбрав некоторое количество комбинаций случайным образом или используя другой скрипт.

Оператор **METHOD** фактически является точкой входа программы, все параметры должны быть описаны заранее, а несохраненные до него вычисления забываются. Для поиска по сетке используется конструкция **METHOD: GRID**. Использовать его можно двумя вариантами: либо указать один общий шаг, тогда он будет применен ко всем параметрам, либо указать с каким шагом обходить какой параметр. Например, на рисунке 16 в первом варианте каждый параметр будет обходиться с шагом 1, то есть начальная выборка будет сформирована из всех возможных комбинаций параметров, если они положительные и

целочисленные. Во втором случае параметр с именем *-a* будет обходиться с шагом 1, параметр *-b* с шагом 2, а остальные с шагом 1.

```
METHOD : GRID -> 1
METHOD : GRID -> (-a,1) (-b,2)
```

Рисунок 16 — Пример описания метода поиска по сетке для создания начальной выборки

Есть также возможность в качестве начальной выборки взять случайные N комбинаций в таком случае нужно использовать оператор **RANDOM** вместо **GRID**. В примере, представленном на рисунке 17 будет выбрано 10 случайных комбинаций в качестве начальной выборки.

```
METHOD : RANDOM -> 10
```

Рисунок 17 — Пример описания метода случайного поиска для создания начальной выборки

Поскольку **METHOD** является точкой входа программы, его можно использовать не только для создания начальной выборки, но и для вызова подпрограммы. Чтобы импортировать написанный скрипт, используется ключевое слово **SCRIPT**. Таким образом, можно в один скрипт собрать вызовы нескольких скриптов, например, для автоматизации подбора параметров для нескольких программ. В качестве параметра данного оператора передается абсолютный путь или относительный относительно программы-интерпретатора к нужному скрипту. Пример использования другого скрипта в качестве метода можно видеть на рисунке 18.

```
METHOD: SCRIPT -> scripts/optimalScript.txt
```

Рисунок 18 — Пример использования скрипта в качестве метода

2.4.3 Условный оператор

Когда начальная выборка создана, можно приступать к описанию правил, по которым будут отбираться комбинации параметров для проведения экспериментов.

Первым вариантом описания правил отбора является конструкция **IF THEN**. С помощью нее можно наложить условие на значение одного параметра в зависимости от другого. Например, в примере на рисунке 19 описаны три условия: если параметр *-a* равен 2, то параметр *-b* не должен быть равен 12, если параметр *-a* равен 3, то параметр *-b* не должен быть равен ни 6, ни 12, если параметр *-a* равен 4, то параметр *-b* должен быть равен 5.

Оператор работает исключительно на уменьшение размера начальной выборки, поэтому последнее условие не изменит значение параметра *-b* в комбинации на 5, а удалит все такие комбинации, в которых параметр *-a* равен 4, а параметр *-b* не равен 5.

```
IF -a = 2 THEN -b != 12
IF -a = 3 THEN -b != 6,12
IF -a = 4 THEN -b = 5
```

Рисунок 19 — Пример использования условного оператора

2.4.4 Вычисляемые параметры

Более масштабной в плане потенциала использования является возможность создания вычисляемых параметров. Для этой цели используется конструкция **CALC FROM**. Суть данной конструкции заключается в том, что один параметр будет вычисляться на основании значений других параметров.

Есть два варианта использования этой конструкции, в первом варианте параметр полностью зависит от других, а во втором частично.

Рассмотрим программу на рисунке 20. Допустим у нас есть 2 параметра: *-n* и *-threadnum* и мы хотим , чтобы значение параметра -threadnum зависело от значения параметра *-n* и константы numOfProccess. В данном примере во всех комбинациях параметров значение *-threadnum* будет заменено по правилу:

numOfProccess разделить на *-n*, причем не важно какое значение было у параметра *-threadnum* в комбинации до этого. Таким образом параметр *-threadnum* полностью зависит от значений других параметров. Выражение после двоеточия может быть любым, однако допустимыми операциями являются сложение (+), вычитание (-), умножение (*), деление (/), остаток от деления (%).

```
ADD PARAM : -n RANGE(1->3|1) ENUM(6,12)
ADD PARAM : -threadnum ENUM(1,2,3,6,12)
ADD PARAM : numOfProccess ENUM(12)

METHOD : GRID -> 1

CALC -threadnum FROM -n, numOfProccess : numOfProccess / -n
```

Рисунок 20 — Пример описание вычисляемого параметра, полностью зависящего от других

Теперь рассмотрим частичную зависимость. Допустим, нам нужно, чтобы комбинации оставались в выборке при каком-то условии, но условного оператора **IF THEN** не хватает для описания этого условия. Тогда можно использовать конструкцию **CALC FROM : COND**.

Рассмотрим пример на рисунке 21. Пусть у нас есть параметры *-a*,*-b*,*-c*,*-d* и мы хотим, чтобы *-a* в случае, когда *-d* = 0 делилось на *-b*, а когда *-d* = 1 делилось на *-c*. Для этого можно составить два простых логических выражения как показано на рисунке 21. Если условие в скобках после ключевого слова **COND** выполняется, комбинация остается, если нет - удаляется. Видно, что значение параметра -a не полностью зависит от других параметров, а только при срабатывании условия, поэтому *-a* зависит от других параметров частично. Логическое выражение может быть любой сложности, основными операторами являются равенство (=), неравенство (!=), больше (>), меньше (<), больше или равно (>=), меньше или равно (<=), логическое И (AND), логическое ИЛИ (OR) и другие [22].

```
ADD PARAM : -a ENUM(1,2,3,4,5)
ADD PARAM : -b ENUM(100,300)
ADD PARAM : -c ENUM(300,100)
ADD PARAM : -d ENUM(0,1)

METHOD : GRID -> 1

CALC -a FROM -b, -d : COND((-b % -a) = 0 OR -d = 1)
CALC -a FROM -c, -d : COND((-c % -n) = 0 OR -d = 0)
```

Рисунок 21 — Пример описание вычисляемого параметра, частично зависящего от других

2.4.5 Повторные запуски

В языке есть возможность повторных запусков программы с лучшими комбинациями. За эту функцию отвечает оператор **SELECT**, суть его работы заключается в том, что он отбирает N лучших комбинаций и инициирует повторный запуск программы с этими комбинациями. Лучшими могут быть как максимальные значения метрики, так и минимальные. Например, на рисунке 22 представлен вариант использования данного оператора, в этом случае будет отобрано 3 комбинации с минимальной метрикой и эти комбинации будут проверены повторно.

```
// Create some combinations

SELECT 3 MIN
```

Рисунок 22 — Пример использования оператора SELECT

Ясно, что поскольку запуск программы является дорогой операцией, использование данного оператора может быть нецелесообразно. Однако если функция значений метрики имеет несколько экстремумов, можно проверить, какой их них является наиболее подходящим.

2.4.6 Добавление и удаление конкретных комбинаций

Если необходимо проверять какую-то конкретную комбинацию или всегда убирать её из выборки, а придумывать для этого правила отбора сложно или трудоемко, в языке предусмотрена возможность это сделать. Для добавления комбинации используется конструкция **ADD COMB,** а для удаления **REMOVE COMB**.

Например, в примере на рисунке 23 в итоговую выборку будет добавлена комбинация, где все параметры равны 1, и не важно, что при определении допустимых значений этих параметров некоторые из них 1 не принимают. При использовании конструкции **ADD COMB** необходимо полностью прописать все параметры комбинации, ведь она будет добавлена в выборку в неизменном виде.

Таких строгих правил не накладывается на конструкцию **REMOVE COMB -** не обязательно прописывать значения всех параметров. В примере на рисунке 23 будут убраны все комбинации, в которых *-a* равно 2, а *-b* 3. Удаление комбинаций можно использовать как ещё один вариант фильтрации итоговой выборки.

```
ADD PARAM : -a ENUM(1,2,3,4,5)
ADD PARAM : -b ENUM(2,3,4,5,6)
ADD PARAM : -c ENUM(3,4,5,6,7)
ADD PARAM : -d ENUM(4,5,6,7,8)

ADD COMB : PROGNAME program.bat METRIC metric -a 1 -b 1 -c 1 -d 1

REMOVE COMB : -a 2 -b 3
```

Рисунок 23 — пример добавления и удаления конкретных комбинаций

Оператор **REMOVE** может быть использован вместе с ключевым словом **ALL**, тогда количество комбинаций остается неизменным, однако из строки запуска убирается некоторый параметр. Таким образом можно убираться временные переменные из итоговых комбинаций.

2.4.7 Выбор некоторого количества комбинаций

Если появляется необходимость выбора некоторого количества комбинаций из сформированной выборки для проверки, например, взять N случайных комбинаций из выборки и проверить их, в языке предусмотрен оператор **TAKE**.

На рисунке 24 показан вариант использования оператора **TAKE** для взятия 10 случайных комбинаций из начальной выборки. Всего в начальной выборке 5 в 4 степени различных комбинаций, а запуск программы дорогая операция, данный оператор можно использовать как реализацию случайного поиска с ограничением в количестве запусков.

```
ADD PARAM : -a ENUM(1,2,3,4,5)
ADD PARAM : -b ENUM(2,3,4,5,6)
ADD PARAM : -c ENUM(3,4,5,6,7)
ADD PARAM : -d ENUM(4,5,6,7,8)

METHOD : GRID -> 1

TAKE 10 RANDOM
```

Рисунок 24 — Пример использования оператора TAKE

Вместо **RANDOM** можно использовать **FIRST** и **LAST**, тогда комбинации будут взяты в порядке запусков, это может быть полезно в случаях, когда график значения метрики монотонный, и можно взять наибольшие или наименьшие значения в итоговую выборку при последующей проверке.

2.4.8 Запуск программы

Для запуска программы с разными комбинациями значений параметров предусмотрен оператор **RUN**.

Данный оператор отвечает не только за запуск программ, но и за выходные данные скрипта. В качестве выхода можно вывести максимальное значение метрики и комбинацию, при которой это значение было достигнуто. Можно выводить максимальное значение (**MAX**), минимальное значение (**MIN**), среднее (**AVG**), а также построить график зависимости значений

метрики от номера запуска (**GRAPH**). Параметры пишутся через пробел после оператора **RUN**. На рисунке 25 можно увидеть пример использования оператора **RUN** с выводом всех возможных выходных данных и построением графика.

```
RUN MAX MIN AVG GRAPH
```

Рисунок 25 — Пример использования оператора RUN

Бывают случаи, когда программу необходимо запускать не напрямую, а ставить ее в очередь. Разработанное средство поддерживает автоматическую постановку задач в очередь при работе с планировщиком задач Altair PBS Pro [23], именно его использует информационно-вычислительный центр Новосибирского государственного университета [24], доступ к которому был предоставлен для тестирования программы. Для того, чтобы программа запускалась не напрямую, а ставилась в очередь после ключевого слова **RUN** необходимо использовать ключевое слово **PBS**. В скобках первым аргументом указывается путь к директории, в которой будут созданы скрипты для постановки в очередь, а следующими аргументами являются параметры скрипта. Например на рисунке 26 скрипты будут сгенерированы в папке scripts. Пример сгенерированного скрипта можно видеть на рисунке 27. Запрашиваемые ресурсы могут зависеть от входных параметров программы. Например, если подбирается оптимальное количество процессов, язык позволяет создавать нужную строку запроса ресурсов в зависимости от числа процессов с помощью оператора **CALC**.

```
ADD PARAM : -l ENUM(select=1:ncpus=12:mpiprocs=12:mem=2000m,walltime=00:01:00)
ADD PARAM : -o ENUM(/mnt/storage/home/inbaranov/out/output_0.txt)
ADD PARAM : -e ENUM(mnt/storage/home/inbaranov/out/errorput_0.txt)

// some combinations

RUN PBS(/mnt/storage/home/inbaranov/scripts,-l,-o,-e)
```

Рисунок 26 — Пример постановки задач в очередь

```
#!/bin/bash
#PBS -l select=1:ncpus=12:mpiprocs=1:mem=2000m,walltime=00:01:00
#PBS -o /mnt/storage/home/inbaranov/out/output_0.txt
#PBS -e /mnt/storage/home/inbaranov/out/errorput_0.txt
run_script -np 1 -threadnum 12 -dec 0 -reverse 0 -nx 240 -ny 2000
```

Рисунок 27 — Пример сгенерированного PBS скрипта

Выходные данные после завершения работы всех задач, поставленных в очередь, будут содержаться в файлах, а не выведены в консоль, поэтому эти данные нужно обрабатывать отдельно. Для этого существует оператор **ANALIZE**. Через пробел после вызова этого оператора нужно указать путь до директории, в которой лежат файлы с выходными данными. По заданной метрике файлы будут проанализированы и в консоль будет выведена наилучшая комбинация. Пример использования этого оператора представлен на рисунке 28.

```
ADD PARAM : METRIC ENUM(Time)

METHOD : GRID -> 1

ANALIZE someDirectory MIN
```

Рисунок 28 — Пример использования оператора ANALIZE

2.4.9 Дополнительные функции

В языке есть некоторые дополнительные функции, без которых можно обойтись в общем случае, однако они могут быть полезны в некоторых ситуациях. Оператор **COUNT** позволяет хранить в параметре номер запуска. Это может быть удобно, например, чтобы ассоциировать файлы вывода с номером запуска как на рисунке 29.

```
ADD PARAM : -o ENUM(output)
ADD PARAM : -e ENUM(errorput)
ADD PARAM : fileNumber ENUM(0)

COUNT -> fileNumber

CALC -o FROM fileNumber -> /mnt/storage/home/inbaranov/out/output_fileNumber.txt
CALC -e FROM fileNumber -> /mnt/storage/home/inbaranov/out/errorput_fileNumber.txt
```

Рисунок 29 — Пример использования оператора **COUNT**

Если параметр не принимает никаких значений, а является флагом, то вместо **ENUM** или **RANGE** можно использовать ключевое слово **FLAG,** тогда параметр будет без всяких значений. Например, такая функция может понадобиться при подборе ключей компилятора.

Иногда появляется необходимость контролировать порядок запуска комбинаций. Это можно сделать с помощью сортировки комбинации по какому-то из параметров. Для этого предусмотрена конструкция **ORDER_BY**.

2.4.10 Подсветка синтаксиса

Несмотря на простоту разработанного языка, писать программы на нем без подсветки синтаксиса достаточно некомфортно, поэтому было разработано расширение для Visual Studio Code, которое решает эту проблему. На всех рисунках с программами расширение уже используется. Расширение можно установить по названию “Tetrayder ultra language”.

2.5 Тестирование

Разобравшись с синтаксисом языка, можно приступить к тестированию. Для начала напишем простой скрипт для простой программы, чтобы проверить правильность работы интерпретатора.

2.5.1 Простая программа

В качестве простейшей программы, для которой нужно подобрать параметры возьмем программу, которая перемножает 2 матрицы. Параметром, который мы будем подбирать, является количество OpenMP нитей, оптимизирующих перемножение матриц с помощью распараллеливания. На

рисунке 30 представлен фрагмент кода с OpenMP директивой.

```
    Matrix c(a.rowSize, b.colSize);

    int i, j, k;

#pragma omp parallel for shared(a, b, c) private(i, j, k)
    for (i = 0; i < a.rowSize; i++)
    {
        for (j = 0; j < b.colSize; j++)
        {
            c(i, j) = 0;
            for (k = 0; k < a.colSize; k++)
            {
                c(i, j) += a(i, k) * b(k, j);
            }
        }
    }
```

Рисунок 30 — Часть кода, отвечающая за перемножение матриц

На вход программа принимает количество потоков, указанных в параметре *-t*. Имя скомпилированной программы a.out. Напишем скрипт для полного перебора количества потоков от одного до восьми. В качестве метрики будет время работы, его программа выводит в секундах после слова Time. Так как нам нужно минимальное время, указываем, что нужно вывести минимум и график. Код скрипта можно видеть рисунке 31.

```
ADD PARAM : PROGNAME ENUM(a.out)
ADD PARAM : METRIC ENUM(Time)

ADD PARAM : -t RANGE(1->8|1)

METHOD : GRID -> 1

RUN MIN GRAPH
```

Рисунок 31 — Скрипт для подбора оптимального количества потоков

Процессор имеет 8 физических ядер, поэтому ожидаем, что наилучший результат будет при 8 потоках. На рисунке 32 можно видеть график работы

программы. Как и ожидалось наилучшим оказалось количество потоков равное 8.

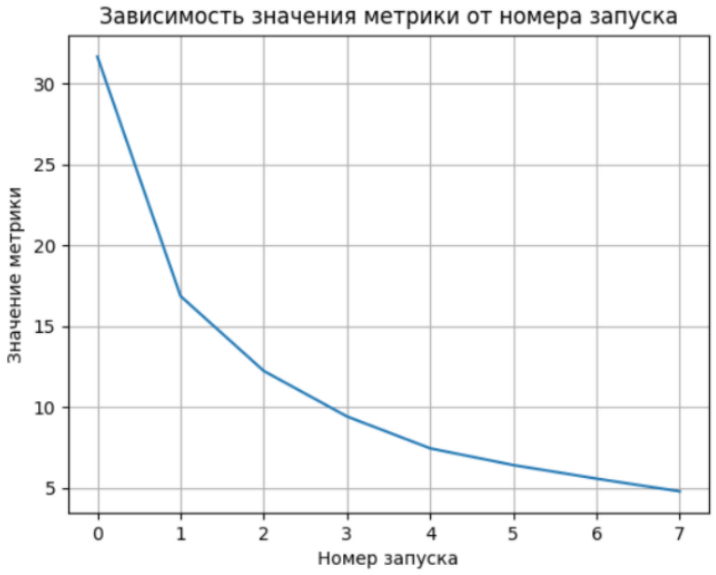

Рисунок 32 — График зависимости значения метрики от номера запуска для теста перемножения матриц

2.5.2 Программа, написанная с помощью библиотеки

Если в предыдущем случае нам удавалось сделать полный перебор и найти оптимальное значение параметра, то для более сложной программы с большим количеством параметров так сделать не получается. Опишем параметры нашей программы в случае 16-ти ядерного процессора.

Параметры, которые будут подбираться, а также их допустимые значения, представлены в таблице 2.

Таблица 2 — параметры реализации программы, написанной с помощью библиотеки

Параметр	Название параметра	Допустимые значения
Количество процессов	-n	Любое положительное целочисленное число
Количество потоков	-treadnum	Любое положительное целочисленное число

Способ декомпозиции	-dec	0 - линейная по оси X, 1 - линейная по оси Y, 2 - двумерная декомпозиция
Порядок цикла обхода индексного пространства по оси X	-forOrderX	0 - внешний, 1 - внутренний
Порядок цикла обхода индексного пространства по оси Y	-forOrderY	0 - внешний, 1 - внутренний
Количество процессов в топологии процессов по оси X	-dimX	Целочисленное положительное число > 0 и < -n
Количество процессов в топологии процессов по оси Y	-dimY	Целочисленное положительное число > 0 и -n

Очевидным ограничением на параметры является зависимость между количеством процессов и количеством потоков. Их произведение не должно превышать количество физических ядер. Запрограммируем это условие с помощью оператора **CALC**. Для этого введем константу numOfProccess, отвечающую за количество физических ядер. Количество потоков будем получать из количества процессов делением количества физических ядер на количество процессов. Поскольку количество потоков целое, приведем его к целочисленному типу. Таким образом, если количество процессов равно, например, 15, а 16 нацело на 15 не делиться, то количество потоков после приведения к целочисленному значению будет равно 1. В данном случае будем приводить к типу System.Int32.

В случае линейных декомпозиций явно зададим порядок циклов при обходе индексного пространства. Если декомпозиция выполняется по оси X, то

для достижения большей эффективности внешним должен быть цикл по оси X (параметр -forOrderX = 0). Соответственно, для случая с декомпозицией по Y, внешним будет цикл по оси Y (параметр -forOrderX = 1).

Зависимость между параметрами -forOrderX и -forOrderY очевидна — один из них должен быть 1, а другой 0.

Для двумерной декомпозиции поставим ограничение на количество процессов равное 2,4,6,8, так как эти значения нацело делятся на размеры сетки, и локальные области массива после декомпозиции будут равны. В этом случае ограничения на параметры -forOrderX и -forOrderY не ставим — они будут зависеть от размеров сетки. Получившийся скрипт представлен на рисунке 33.

```
ADD PARAM : PROGNAME ENUM(run)

ADD PARAM : METRIC ENUM(Time)

ADD PARAM : -n RANGE(1->8|1)
ADD PARAM : -threadnum RANGE(1->16|1)
ADD PARAM : -dec RANGE(0->2|1)
ADD PARAM : -forOrderX ENUM(0,1)
ADD PARAM : -forOrderY ENUM(0,1)
ADD PARAM : -dimX ENUM(0)
ADD PARAM : -dimY ENUM(0)

ADD PARAM : -nx ENUM(2400)
ADD PARAM : -ny ENUM(240)

ADD PARAM : numOfProccess ENUM(16)

METHOD : GRID -> 1

IF -dec = 2 THEN -n = 2,4,6,8

IF -dec = 0 THEN -forOrderX = 0
IF -dec = 1 THEN -forOrderX = 1

IF -forOrderX = 1 THEN -forOrderY = 0
IF -forOrderY = 0 THEN -forOrderX = 1

CALC -threadnum FROM -n, numOfProccess : Convert(numOfProccess / -n,'System.Int32')

REMOVE ALL : numOfProccess

ORDER_BY : -dec DESC

RUN MIN
```

Рисунок 33 — скрипт подбора параметров для программы, написанной с помощью библиотеки

Благодаря наложенным ограничениям из 1536 возможных комбинаций будем перебирать всего 24. Для размеров 240x2400 получилось, что в наилучшей комбинации применяется линейная декомпозиция по оси X, на втором месте двумерная декомпозиция с параметрами -forOrderX = 0 и -forOrderY = 1, то есть цикл по оси X был внешний. Во всех тестах выборка комбинаций была отсортирована по типу декомпозиции, поэтому графики можно визуально разделить на 3 части: слева двумерная декомпозиция, посредине декомпозиция по оси Y, справа по оси X. Результаты теста можно видеть на рисунках 34 и 35.

Launch number	Combination	Metric
23	-n 8 -threadnum 2 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 240 -ny 2400	3.765464
6	-n 8 -threadnum 2 -dec 2 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 240 -ny 2400	4.580295
22	-n 7 -threadnum 2 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 240 -ny 2400	4.843114
21	-n 6 -threadnum 3 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 240 -ny 2400	5.192511
20	-n 5 -threadnum 3 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 240 -ny 2400	5.521638
19	-n 4 -threadnum 4 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 240 -ny 2400	5.750832
14	-n 7 -threadnum 2 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 240 -ny 2400	6.372892
15	-n 8 -threadnum 2 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 240 -ny 2400	6.460194
18	-n 3 -threadnum 5 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 240 -ny 2400	6.939652

Рисунок 34 — Результаты работы программ с разными входными параметрами для сетки с размерами 240x2400

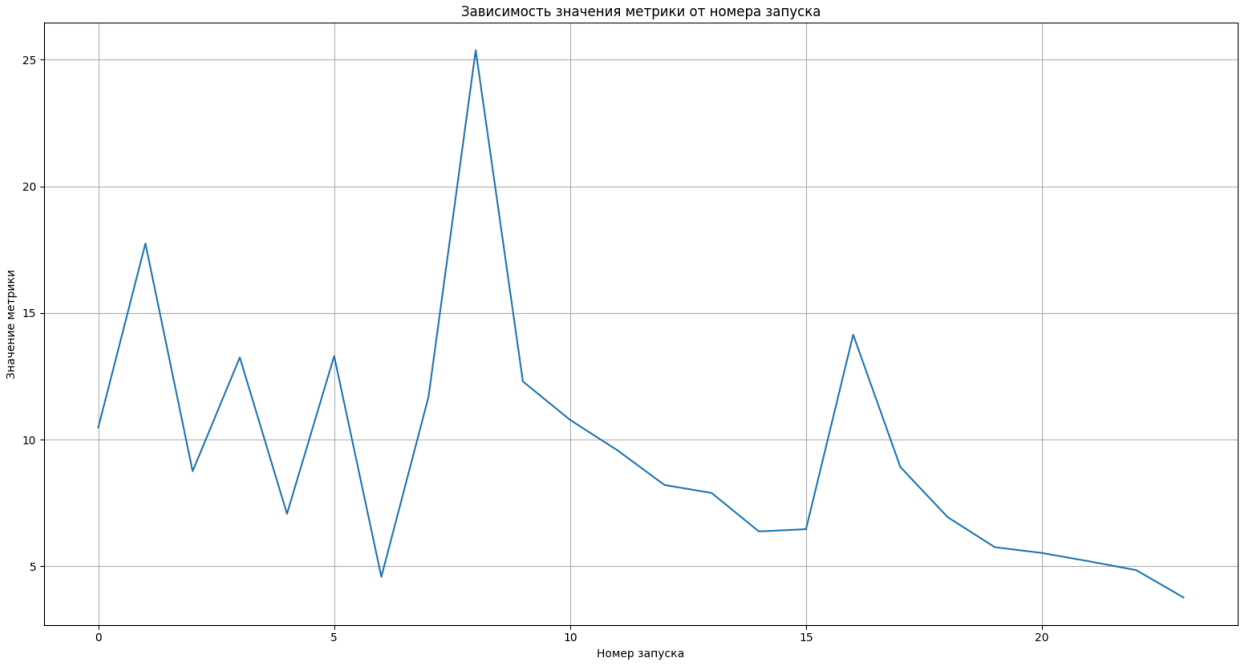

Рисунок 35 — График зависимости номера запуска от значения метрики для сетки с размерами 240x2400

Проведем тест симметричных размеров сетки, чтобы увидеть зависимость значений параметров от них. Возьмем размер сетки 2400x240.

Как видно из рисунков 36 и 37, для задачи с размерами сетки 2400x240 наоборот лучше себя показали двумерная декомпозиция с внешним циклом по оси Y и линейная декомпозиция по оси Y. Отсюда можно увидеть зависимость между размерами задачи и выбранными значениями параметров. Если размер по X больше размера по Y выгоднее использовать декомпозицию по оси X и наоборот.

Launch number	Combination	Metric
7	-n 8 -threadnum 2 -dec 2 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 2400 -ny 240	3.885485
15	-n 8 -threadnum 2 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 2400 -ny 240	3.942279
14	-n 7 -threadnum 2 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 2400 -ny 240	3.963047
13	-n 6 -threadnum 3 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 2400 -ny 240	4.829727
12	-n 5 -threadnum 3 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 2400 -ny 240	5.15769
5	-n 6 -threadnum 3 -dec 2 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 2400 -ny 240	5.577526
11	-n 4 -threadnum 4 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 2400 -ny 240	5.839442
3	-n 4 -threadnum 4 -dec 2 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 2400 -ny 240	6.429258
23	-n 8 -threadnum 2 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 2400 -ny	6.618447

Рисунок 36 — Результаты работы программ с разными входными параметрами для сетки с размерами 2400x240

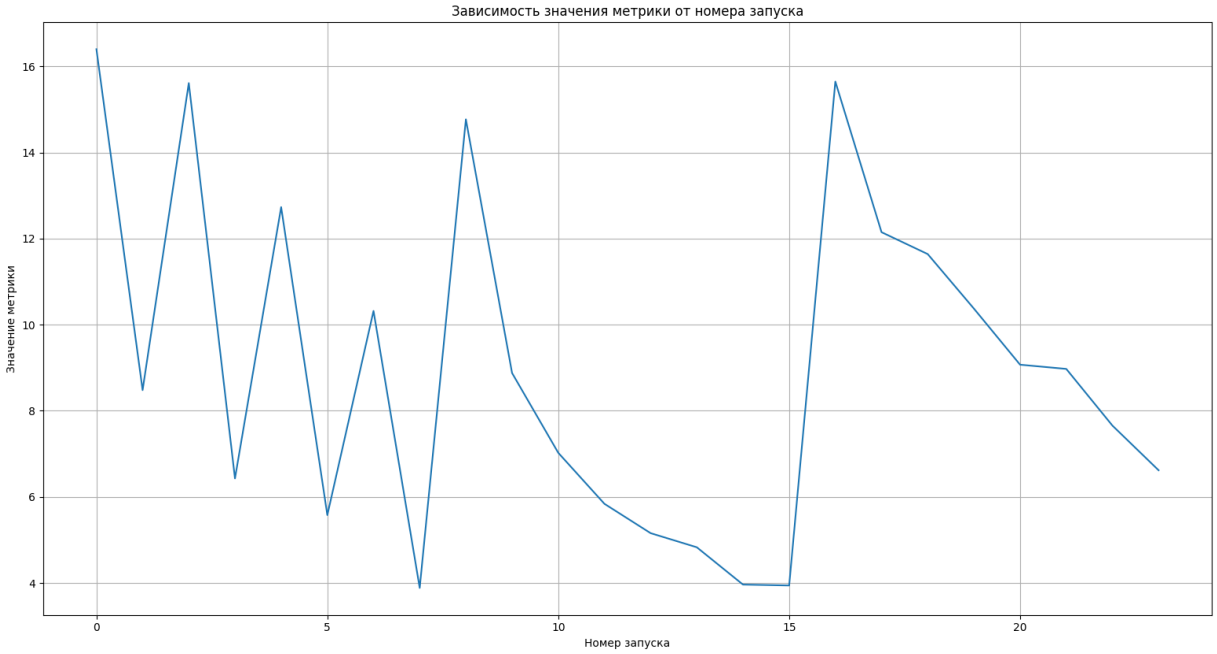

Рисунок 37 — График зависимости номера запуска от значения метрики для сетки с размерами 2400x240

В обоих случаях комбинация процессов и потоков равные 8 и 2 соответственно показали наилучший результат, это было ожидаемо, поскольку использовался 8-ми ядерный процессор с гипертредингом.

Далее можно выполнить тесты на больших вычислительных мощностях, например на кластере НГУ.

2.5.3 Тестирование на кластере НГУ

Для тестирования на кластере применим полученные зависимости между параметрами для уменьшения выборки. Во-первых, изменим шаг по количеству процессов с 1 на 2 и минимальное количество процессов зафиксируем на 8, потому что наилучшие результаты получались на четном количестве процессов. Во-вторых, в случае двумерной декомпозиции будем менять порядок циклов обхода индексного пространства в зависимости от размеров сетки. Новые фильтры можно видеть на рисунке 38.

```
IF -dec = 2 THEN -np = 8,12,24

CALC isNxBiggerThanNy FROM -nx,-ny : -nx > -ny
CALC -forOrderX FROM isNxBiggerThanNy, -forOrderX, -dec :
    COND ((-dec = 2 AND isNxBiggerThanNy = -forOrderX) OR -dec = 1 OR -dec = 0)

IF -dec = 0 THEN -forOrderX = 0
IF -dec = 1 THEN -forOrderX = 1

IF -forOrderX = 1 THEN -forOrderY = 0
IF -forOrderY = 0 THEN -forOrderX = 1

CALC -threadnum FROM -np, numOfProccess : Convert(numOfProccess / -np,'System.Int32')
CALC -threadnum FROM -threadnum : Convert((-threadnum = 0), 'System.Int32') + -threadnum

CALC nAndThreads FROM -np, -threadnum : -np * -threadnum
```

Рисунок 38 — Измененные под запуск на кластере фильтры для параметров

Проведем несколько тестов. Возьмем размеры сетки равные 240x2400, 2400x240 и равномерную — 840x840. В последнем случае проверим двумерную декомпозицию с разными вариантами обхода индексного пространства. Результаты тестов представлены на рисунках 39, 40, 41, 42, 43 и 44.

Launch number	Combination	Metric
2	-np 24 -threadnum 1 -dec 2 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 2400 -ny 240 mpiProcs 12 numOfNodes 2	1.515999
11	-np 24 -threadnum 1 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 2400 -ny 240 mpiProcs 12 numOfNodes 2	1.549563
20	-np 24 -threadnum 1 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 2400 -ny 240 mpiProcs 12 numOfNodes 2	1.677397
19	-np 22 -threadnum 1 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 2400 -ny 240 mpiProcs 12 numOfNodes 2	1.781141
9	-np 20 -threadnum 1 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 2400 -ny 240 mpiProcs 10 numOfNodes 2	1.826607
7	-np 16 -threadnum 2 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 2400 -ny 240 mpiProcs 6 numOfNodes 3	1.912921
18	-np 20 -threadnum 1 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 2400 -ny 240 mpiProcs 10 numOfNodes 2	1.953894
16	-np 16 -threadnum 2 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 2400 -ny 240 mpiProcs 6 numOfNodes 3	2.156465
17	-np 18 -threadnum 1 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 2400 -ny 240 mpiProcs 10 numOfNodes 2	2.227428

Рисунок 39 — Результаты работы программы с разными входными параметрами для сетки с размерами 2400x240

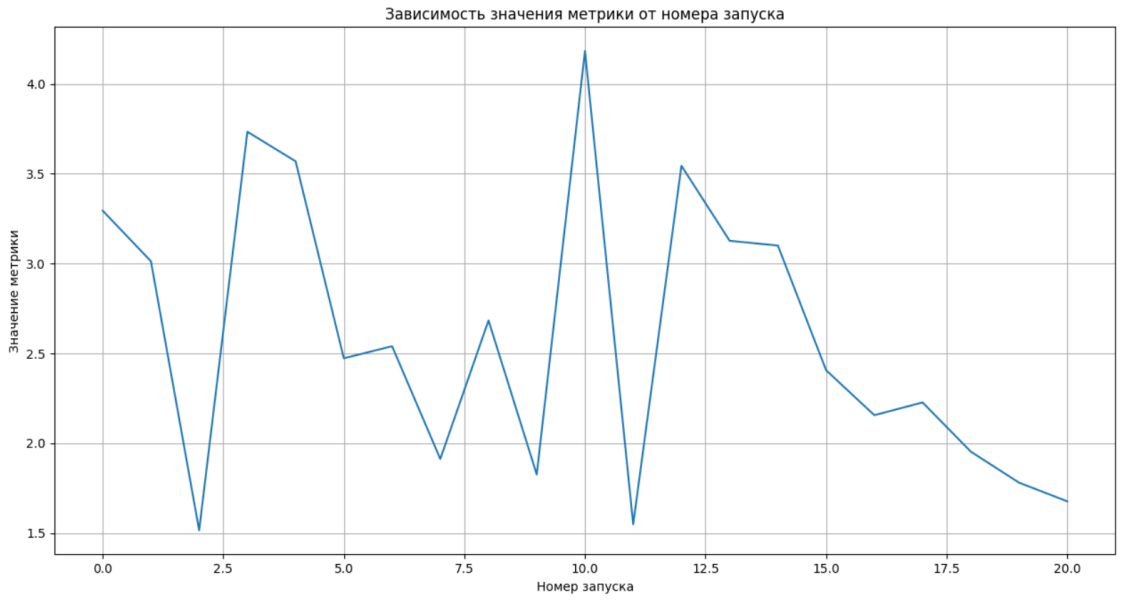

Рисунок 40 — График зависимости номера запуска от значения метрики для сетки с размерами 2400x240

Launch number	Combination	Metric
2	-np 24 -threadnum 1 -dec 2 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 240 -ny 2400 mpiProcs 12 numOfNodes 2	1.546928
11	-np 24 -threadnum 1 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 240 -ny 2400 mpiProcs 12 numOfNodes 2	1.552492
10	-np 22 -threadnum 1 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 240 -ny 2400 mpiProcs 12 numOfNodes 2	1.648504
20	-np 24 -threadnum 1 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 240 -ny 2400 mpiProcs 12 numOfNodes 2	1.655698
9	-np 20 -threadnum 1 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 240 -ny 2400 mpiProcs 10 numOfNodes 2	1.853479
18	-np 20 -threadnum 1 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 240 -ny 2400 mpiProcs 10 numOfNodes 2	2.038489
8	-np 18 -threadnum 1 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 240 -ny 2400 mpiProcs 10 numOfNodes 2	2.065092
7	-np 16 -threadnum 2 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 240 -ny 2400 mpiProcs 6 numOfNodes 3	2.227793
16	-np 16 -threadnum 2 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 240 -ny 2400 mpiProcs 6 numOfNodes 3	2.273796

Рисунок 41 — Результаты работы программы с разными входными параметрами для сетки с размерами 240x2400

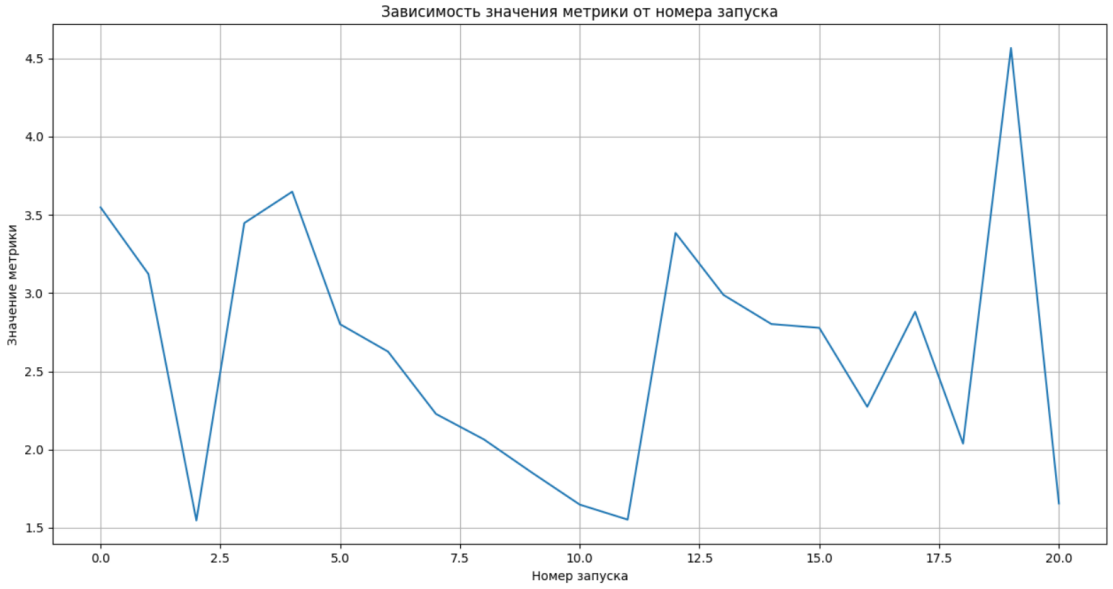

Рисунок 42 — График зависимости номера запуска от значения метрики для сетки с размерами 240x2400

Launch number	Combination	Metric
14	-np 24 -threadnum 1 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 840 -ny 840 mpiProcs 12 numOfNodes 2	0.530187
4	-np 24 -threadnum 1 -dec 2 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 840 -ny 840 mpiProcs 12 numOfNodes 2	0.543477
5	-np 24 -threadnum 1 -dec 2 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 840 -ny 840 mpiProcs 12 numOfNodes 2	0.548078
23	-np 24 -threadnum 1 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 840 -ny 840 mpiProcs 12 numOfNodes 2	0.573081
13	-np 22 -threadnum 1 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 840 -ny 840 mpiProcs 12 numOfNodes 2	0.6272
12	-np 20 -threadnum 1 -dec 1 -forOrderX 1 -forOrderY 0 -dimX 0 -dimY 0 -nx 840 -ny 840 mpiProcs 10 numOfNodes 2	0.639367
22	-np 22 -threadnum 1 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 840 -ny 840 mpiProcs 12 numOfNodes 2	0.683696
21	-np 20 -threadnum 1 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 840 -ny 840 mpiProcs 10 numOfNodes 2	0.7029
19	-np 16 -threadnum 2 -dec 0 -forOrderX 0 -forOrderY 1 -dimX 0 -dimY 0 -nx 840 -ny 840 mpiProcs 6 numOfNodes 3	0.742821

Рисунок 43 — Результаты работы программы с разными входными параметрами для сетки с размерами 840x840

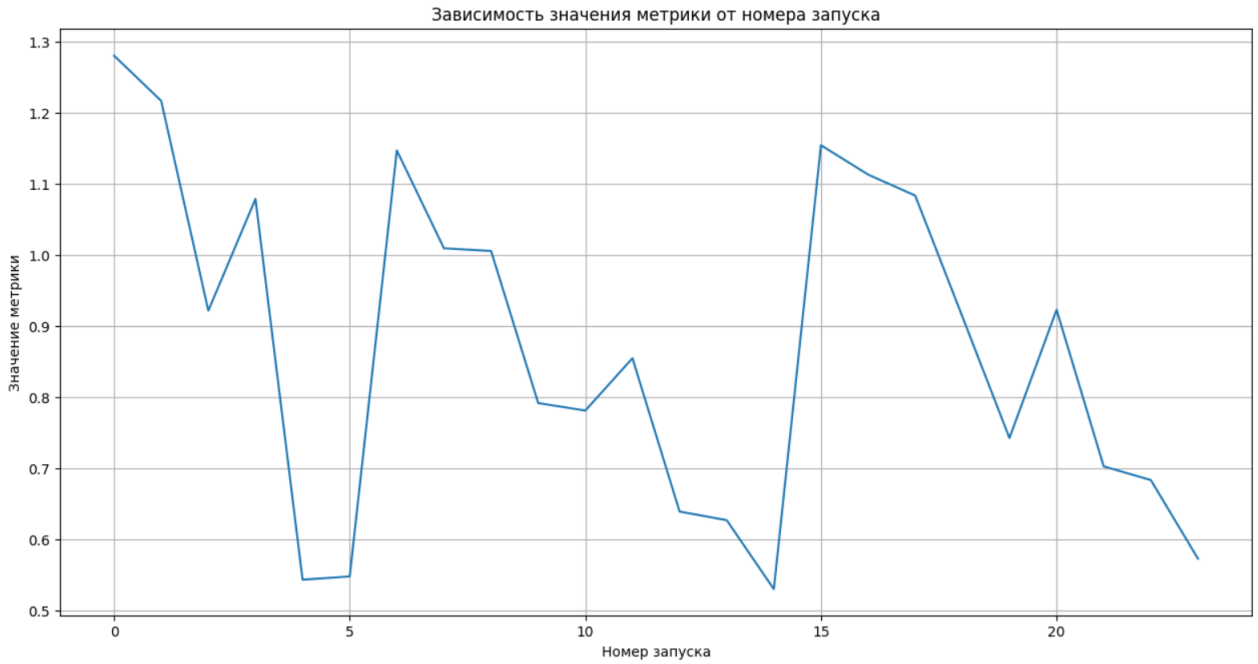

Рисунок 44 — График зависимости номера запуска от значения метрики для сетки с размерами 840x840

Из проведенных тестов видно, что двумерная декомпозиция самая стабильная, на 24 процессах она показывает один из наилучших результатов во всех тестах. Если сравнивать графики на рисунках 40 и 42, видно, что в первом случае наблюдается плавный спуск с увеличением количества процессов при декомпозиции по оси Y, а во втором случае по оси X. В случае равномерной сетки на рисунке 44 видно, что части графика при декомпозиции по оси X и Y очень похожи, а двумерная декомпозиции показывает примерно один и тот же результат при разных вариантах обхода индексного пространства. Это происходит из-за того, что размеры сетки по оси X и Y равны. Таким образом, влияние параметров реализации на время работы программы предсказуемо, и подбор оптимальных параметров был оптимизирован за счет знания о параметрах.

ЗАКЛЮЧЕНИЕ

Была разработана библиотека параллельного программирования для работы с распределенными массивами, специализирующаяся на решении задач пространственной динамики на регулярных сетках. В библиотеки были выделены параметры реализации, которые влияют на нефункциональные характеристики программы. Эти параметры необходимо подбирать для достижения наилучшей производительности программы. Работоспособность библиотеки показана с помощью тестовой программы, решающей уравнение Гельмгольца методом Якоби.

Было разработано программное средство подбора параметров на основе знаний эксперта. Для описания знаний был разработан скриптовый язык программирования и соответствующий интерпретатор для него.

Для программы, написанной с помощью библиотеки, был разработан скрипт подбора параметров на основе знаний о них. Параметры были успешно подобраны для различных вариантов входных данных, и средство показало свою эффективность.

В дальнейшем разработанные средства могут быть расширены дополнительной функциональностью: библиотека — различными способами реализации и оптимизации, а средство подбора параметров — способами поиска, а также расширением возможностей скриптового языка программирования. Эти средства могут быть применены при синтезе параллельных программ в системе LuNA.

Выпускная квалификационная работа выполнена мной самостоятельно и с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного

цитирования является основанием для не допуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

Баранов Илья Николаевич _____

ФИО студента *Подпись студента*

«_____»_____ 20 __г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Описание языка LuNA [Электронный ресурс] . URL: https://gitlab.ssd.sscc.ru/luna/luna5/wikis/luna_lang_v01

2. Hillis W. D., Steele Jr G. L. Data parallel algorithms // Communications of the ACM. – 1986. – Т. 29. – №. 12. – С. 1170-1183.

3. DVM-система разработки параллельных программ [Электронный ресурс]. URL: http://dvm-system.org/ru/about/ (дата обращениея 17.05.2022)

4. Язык параллельного программирования HPF [Электронный ресурс]. URL: http://hpc.sfedu.ru/tutor/high_performance_computing/chapter1/page15.html (дата обращениея 17.05.2022)

5. Ken Kennedy, Charles Koelbel, Hans Zima, The Rise and Fall of High Performance Fortran: An Historical Object Lesson, Proceedings of the third ACM SIGPLAN conference on History of programming languages 2007, San Diego, California, 2007

6. Восход и закат High Performance Fortran: наглядный урок истории [Электронный ресурс]. URL: http://citforum.ru/programming/digest/hpf/ (дата обращениея 17.05.2022)

7. Coarray Fortran [Электронный ресурс]. URL: http://caf.rice.edu (дата обращения 17.05.2022)

8. Малышкин В. Э. Технология фрагментированного программирования // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. – 2012. – №. 46 (305). – С. 45-55.

9. Маркова В. П., Остапкевич М. Б. Сравнение возможностей MPI и LuNA на примере реализации модели клеточно-автоматной интерференции волн // Проблемы информатики. – 2017. – №. 2 (35). – С. 53-64.

10. Технологии параллельного программирования: НОРМА [Электронный ресурс]. URL: https: //parallel.ru/tech/norma (дата обращениея 17.05.2022)
11. Slaughter E. et al. Regent: A high-productivity programming language for HPC with logical regions // Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. – 2015. – С. 1-12.
12. Automated Model Tuning [Электронный ресурс]. URL: https://www.kaggle.com/willkoehrsen/automated-model-tuning (дата обращениея 17.05.2022)
13. Bergstra J., Bengio Y. Random search for hyper-parameter optimization // Journal of machine learning research. – 2012. – Т. 13. – №. 2.
14. Genetic Algorithms [Электронный ресурс]. URL: https://www.sciencedirect.com/topics/engineering/genetic-algorithm (дата обращениея 17.05.2022)
15. Ruder S. An overview of gradient descent optimization algorithms // arXiv preprint arXiv:1609.04747. – 2016.
16. Dewancker I., McCourt M., Clark S. Bayesian optimization for machine learning: A practical guidebook // arXiv preprint arXiv:1612.04858. – 2016.
17. Nelder J. A., Mead R. A simplex method for function minimization // The computer journal. – 1965. – Т. 7. – №. 4. – С. 308-313.
18. Active Harmony User's Guide [Электронный ресурс]. URL: https://www.dyninst.org/sites/default/files/manuals/harmony/v4.6.0/index.html (дата обращения 17.05.2022)
19. Intel® MPI Library Developer Guide for Windows* OS [Электронный ресурс]. URL: https://www.intel.com/content/www/us/en/develop/documentation/mpi-developer-guide-windows/top/analysis-and-tuning/mpi-tuning.html (дата обращения 17.05.2022)

20. User contribution to ATLAS [Электронный ресурс]. URL: http://math-atlas.sourceforge.net/devel/atlas_contrib/ (дата обращения 17.05.2022)

21. FFTW Home Page [Электронный ресурс]. URL: https://fftw.org (дата обращения 17.05.2022)

22. DataColumn.Expression Property [Электронный ресурс]. URL: https://docs.microsoft.com/en-us/dotnet/api/system.data.datacolumn.expression?view=net-5.0 (дата обращения 17.05.2022)

23. PBS Professional 19.2 User's Guide [Электронный ресурс]. URL: http://nusc.nsu.ru/wiki/lib/exe/fetch.php/doc/pbs/pbsuserguide19.2.1.pdf (дата обращения 17.05.2022)

24. Информационно-вычислительный центр Новосибирского государственного университета [Электронный ресурс]. URL: http://nusc.nsu.ru/wiki/doku.php/doc/index (дата обращения 17.05.2022)

ПРИЛОЖЕНИЕ А

Средство подбора параметров на основе знаний эксперта

Руководство оператора

Листов 7

Новосибирск 2022

АННОТАЦИЯ

В данном программном документе приведено руководство оператора по применению и эксплуатации средства подбора параметров на основе знаний эксперта.

В данном программном документе, в разделе «Назначение программы» указаны сведения о назначении программы и информация, достаточная для понимания функций программы и ее эксплуатации.

В разделе «Условия выполнения программы» указаны требования, необходимые для выполнения программы.

В разделе «Выполнение программы» указана последовательность действий оператора, обеспечивающих загрузку, запуск, выполнение и завершение программы.

Оформление программного документа «Руководство оператора» произведено по требованиям ЕСПД : 19.101-77, 19.105-78, ГОСТ 19.505-79.

СОДЕРЖАНИЕ

1 Назначение программы

1.1 Функциональное назначение программы

Разработанная программа предназначена для подбора оптимальных параметров для программы на основе знаний эксперта. Пользователь в виде скрипта описывает знания о параметрах, запускает скрипт и получает наилучшую комбинацию параметров.

1.2 Эксплуатационное назначение программы

Средство подбора параметров используется для нахождения оптимальных параметров для программы, что позволяет достигать максимальной ее производительности.

1.3 Состав функций

Программа обеспечивает возможность выполнения перечисленных ниже функций:

- Описание параметров (Диапазоны, перечисления, их комбинации)
- Выбор метода перебора (Перебор по сетке, случайный, с помощью другого скрипта)
- Фильтраций параметров (логические выражения, удаление комбинаций по строке, оставление N комбинаций)
- Вычисляемые параметры
- Генерацию скриптов для системы очередей PBS для работы с кластером
- Визуализация результатов

2 Условия выполнения программы

2.1 Минимальный состав аппаратных средств

Программа предназначена для использования на персональном компьютере или кластере. Устройство, на котором запускается программа, должно иметь одну из следующих операционных систем: Windows, Linux, Mac OS, а также иметь графический дисплей и средства ввода такие, как клавиатура и компьютерная мышь.

2.2 Требование к персоналу

Конечный пользователь программы (оператор) должен обладать практическими навыками работы с консольным интерфейсом операционной системы.

3 Выполнение программы

3.1 Загрузка и запуск программы

Для использования средства пользователю необходимо загрузить интерпретатор. Его можно скачать по следующей ссылке: https://gitlab.com/i.baranov/tetrayder-ultra-interpreter. Для удобства, путь до интерпретатора можно добавить в переменную окружения PATH.

Интерпретатор принимает на вход единственный аргумент — путь до скрипта, который нужно исполнить. Скрипт представляет из себя текстовый файл на специальном языке программирования. Пример такого скрипта можно увидеть на рисунке 1.

```
ADD PARAM : PROGNAME ENUM(a.out)
ADD PARAM : METRIC ENUM(Time)

ADD PARAM : -t RANGE(1->8|1)

METHOD : GRID -> 1

RUN MIN GRAPH
```

Рисунок 1 — пример скрипта

Для запуска скрипта необходимо вызвать из командной строки интерпретатор и передать ему в качестве аргумента путь до написанного скрипта. Если скрипт называется testscript.tdr, то для запуска необходимо вызвать команду как на рисунке 2.

```
baranov@ssdcomputer:~/optimizer/publish$ ./ParametrsOptimizerConsole testscript.tdr
```

Рисунок 2 — Запуск скрипта с помощью интерпретатора

3.2 Выполнение программы

После запуска интерпретатор будет исполнять строчки кода из скрипта по очереди. В данном случае он запустит программу a.out 8 раз и выведет комбинацию параметров при которой программа a.out выдает наименьшее

значение метрики. В данном случае такой комбинацией будет “-t 8” как показано на рисунке 3.

```
baranov@ssdcomputer:~/optimizer/publish$ ./ParametrsOptimizerConsole testscript.tdr
Initial number of combinations: 8
Number of combinations after filter: 8
Номер запуска 0: a.out -t 1

1 threads...
Time: 10.6826
Номер запуска 1: a.out -t 2

2 threads...
Time: 5.43845
Номер запуска 2: a.out -t 3

3 threads...
Time: 3.94512
Номер запуска 3: a.out -t 4

4 threads...
Time: 2.95736
Номер запуска 4: a.out -t 5

5 threads...
Time: 2.36732
Номер запуска 5: a.out -t 6

6 threads...
Time: 2.04377
Номер запуска 6: a.out -t 7

7 threads...
Time: 1.6964
Номер запуска 7: a.out -t 8

8 threads...
Time: 1.47315
Минимум: -t 8 = 1.47315
```

Рисунок 3 — Пример вывода программы

В качестве результата также будет сгенерирован файл combinations.txt, содержащий информацию о запусках в формате номер запуска — комбинация. Данный файл генерируется до первого запуска программы a.out. Пример файла combinations.txt показан на рисунке 4.

После того как все запуски отработают будет сгенерирован ещё один файл visual.txt, содержащий не только номер запуска и комбинацию параметров, но ещё и значение метрики. С помощью него можно построить график зависимости значения метрики от номера запуска. Пример такого файла показан на рисунке 5.

optimizer > publish > combinations.txt

```
launchNumber;Combination
0; -t 1
1; -t 2
2; -t 3
3; -t 4
4; -t 5
5; -t 6
6; -t 7
7; -t 8
```

Рисунок 4 — Пример файла combinations.txt

optimizer > publish > visual.txt

```
launchNumber;Combination;Metric
0;-t 1;10.6826
1;-t 2;5.43845
2;-t 3;3.94512
3;-t 4;2.95736
4;-t 5;2.36732
5;-t 6;2.04377
6;-t 7;1.6964
7;-t 8;1.47315
```

Рисунок 5 — Пример файла visual.txt

3.3 Завершение работы программы

Программа завершается автоматически после проверки всех комбинаций параметров. Программу можно прервать в любой момент послав сигнал SIGINT. Для этого в операционной системе Windows используется комбинация клавиш “Ctrl-z”, в Linux “Ctrl-c”.

Если был использован оператор GRAPH, то будет вызвана программа-визуализатор, написанная на языке программирования python. Она использует файл visual.txt для построения графика и таблицы с результатами. Пример такого графика можно видеть на рисунке 6.

Для построения графика используется библиотека matplotlib, а для таблицы plotly.

Таблица открывается в браузере в формате html. Её можно сортировать по каждому из столбцов с помощью выпадающего списка. Сортировка осуществляется по возрастанию. Пример таких таблиц с сортировкой по разным столбцам можно видеть на рисунках 7 и 8.

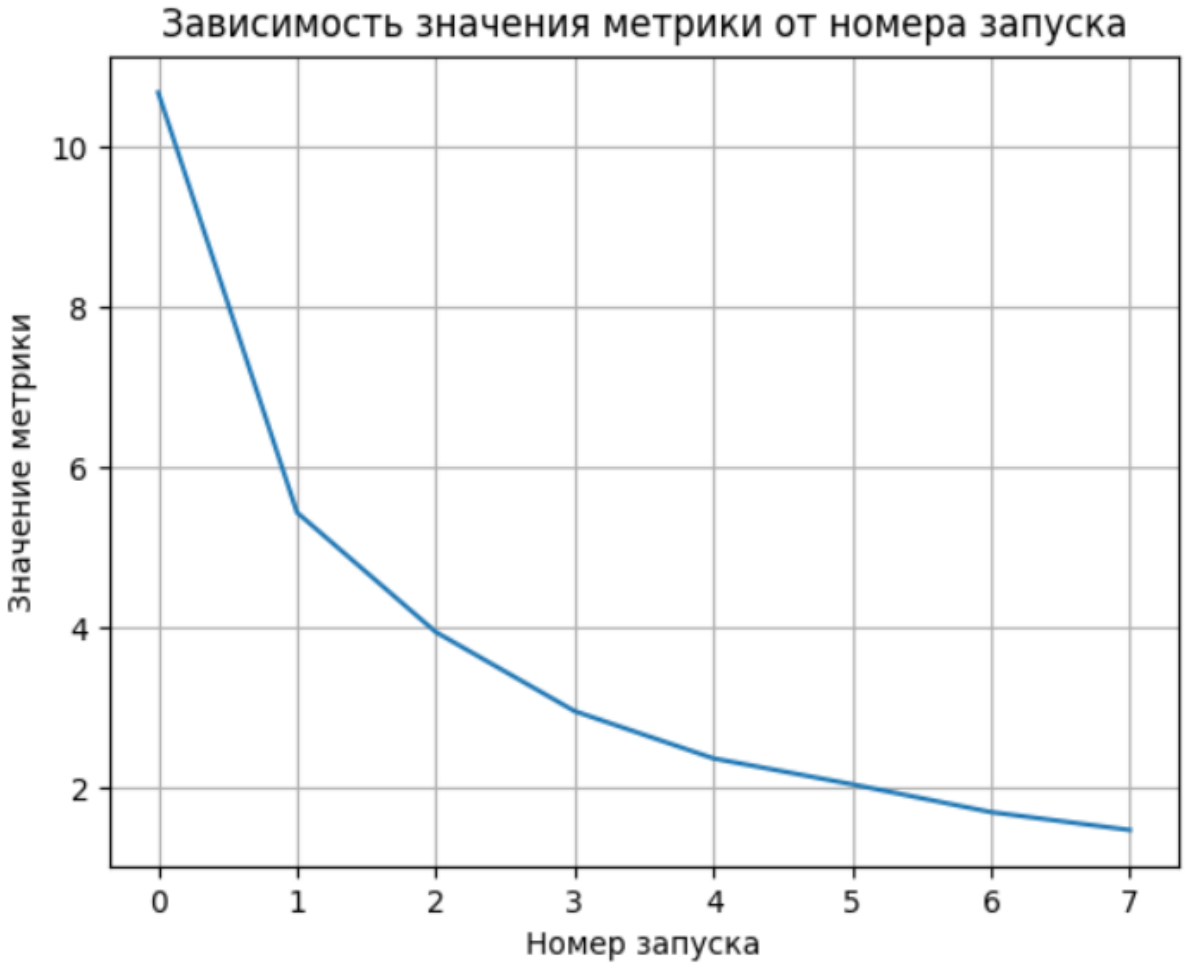

Рисунок 6 — Пример графика, полученного из файла visual.txt

Launch number ▼

Launch number	Combination	Metric
0	-t 1	10.6826
1	-t 2	5.43845
2	-t 3	3.94512
3	-t 4	2.95736
4	-t 5	2.36732
5	-t 6	2.04377
6	-t 7	1.6964
7	-t 8	1.47315

Рисунок 7 — Пример таблицы с результатами работы программы, отсортированными по номеру запуска

Metric ▼

Launch number	Combination	Metric
7	-t 8	1.47315
6	-t 7	1.6964
5	-t 6	2.04377
4	-t 5	2.36732
3	-t 4	2.95736
2	-t 3	3.94512
1	-t 2	5.43845
0	-t 1	10.6826

Рисунок 8 — Пример таблицы с результатами работы программы, отсортированными по значению метрики