Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики

Отчет

по курсу
“Суперкомпьютеры и параллельная обработка данных”

Тема:

Разработка и анализ программ для задачи релаксации с использованием OpenMP и MPI

Выполнил: студент 321 группы
Кирнев Юрий

Лектор: доцент, к.ф.-м.н. Бахтин Владимир Александрович

Москва, 2024 г.

Содержание

1 Постановка задачи

Для предложенного алгоритма реализовать несколько версий параллельных программ с использованием технологии OpenMP. а) Вариант параллельной программы с распределением витков циклов при помощи директивы for. б) Вариант параллельной программы с использованием механизма задач (директива task).

Исследовать эффективность полученных параллельных программ на суперкомпьютере Polus.

Исследовать масштабируемость полученной параллельной программы: построить графики зависимости времени выполнения параллельной программы от числа используемых ядер для различного объёма входных данных.

2 Описание изначальной программы

Исходная программа состоит из нескольких частей:

- Инициализация массива A размером $N \times N$ с коэффициентами, зависящими от индексов i и j.
- Реализация метода релаксации для обновления значений в массиве B.
- Расчет разности между массивами A и B и вычисление ошибки.
- Верификация результата.

3 Изменения для параллелизации

Для улучшения производительности программы были внесены изменения, заключающиеся в использовании библиотеки OpenMP для распараллеливания вычислений.

3.1 Основные изменения

1. Добавлена директива `#include <omp.h>` для подключения библиотеки OpenMP. 2. В функции `init()` добавлена директива `#pragma omp parallel for` для параллельной инициализации массива A. 3. В функции `relax()` добавлена директива `#pragma omp parallel for` для параллельного выполнения циклов обновления массива B. 4. В функции `resid()` добавлена директива `#pragma omp parallel for reduction(max:eps)` для параллельного вычисления максимальной ошибки и обновления массива A.

В результате программа использует многопоточность, что позволяет значительно ускорить выполнение на многопроцессорных системах.

3.2 Графики

Для анализа производительности программы с различными параметрами, такими как количество потоков, были построены графики времени выполнения. На графиках представлены результаты работы программы в зависимости от числа потоков.

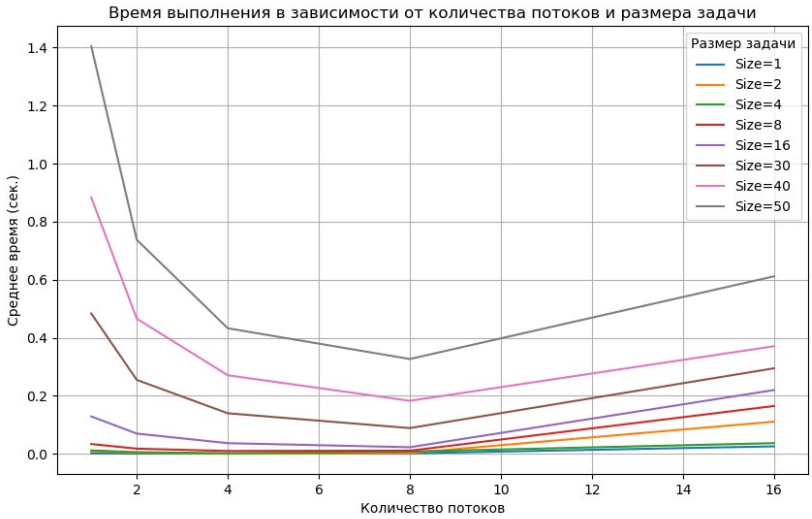

Рис. 1: График времени выполнения программы в зависимости от числа потоков.

3.3 Таблица с результатами

Таблица ниже демонстрирует среднее время выполнения программы в зависимости от размера задачи (Size) и количества потоков. Время выполнения измеряется в секундах.

Size	**1**	**2**	**4**	**8**	**16**	**30**	**40**	**50**
1	0.002	0.009	0.012	0.034	0.129	0.484	0.884	1.405
2	0.001	0.004	0.006	0.018	0.070	0.255	0.466	0.738
4	0.001	0.001	0.002	0.010	0.037	0.140	0.271	0.433
8	0.002	0.003	0.008	0.011	0.023	0.089	0.183	0.327
16	0.026	0.111	0.037	0.165	0.220	0.295	0.371	0.612

Таблица 1: Среднее время выполнения программы в зависимости от размера задачи и количества потоков (в секундах).

4 Распараллеливание с использованием OpenMP с задачами

В этой части работы была реализована версия программы с использованием параллельных задач OpenMP для выполнения вычислений на многопроцессорных системах. Основные изменения по сравнению с предыдущей версией включают использование динамических массивов и директивы ‘pragma omp task‘, что позволяет распараллелить отдельные участки кода.

4.1 Изменения в коде

В предыдущей версии программы использовалась простая параллелизация с помощью директивы ‘pragma omp parallel for‘. В новой версии были внесены следующие изменения:

1. Параллельные задачи: Каждая итерация внутренних циклов была распараллелена с использованием ‘pragma omp task‘. Это позволяет разделить выполнение

вычислений на несколько потоков, улучшив производительность на многопроцессорных системах.

2. Синхронизация задач: Для того, чтобы гарантировать правильную последовательность выполнения, используется директива ‘pragma omp taskwait‘, которая заставляет главный поток ждать завершения всех задач перед продолжением выполнения программы.

3. Обновление переменной eps: Для предотвращения гонки данных при обновлении переменной ‘eps‘ используется блок ‘pragma omp critical‘, который обеспечивает синхронизацию доступа к переменной.

4.2 Графики

Для анализа производительности программы с различными параметрами, такими как количество потоков, были построены графики времени выполнения. На графиках представлены результаты работы программы в зависимости от числа потоков.

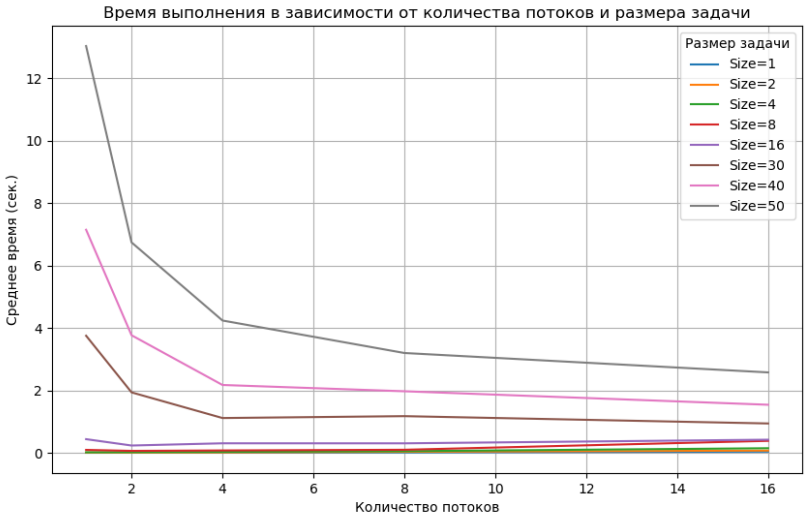

Рис. 2: График времени выполнения программы в зависимости от числа потоков.

4.3 Таблица с результатами

Таблица ниже демонстрирует среднее время выполнения программы в зависимости от размера задачи (Size) и количества потоков. Время выполнения измеряется в секундах.

Size	**1**	**2**	**4**	**8**	**16**	**30**	**40**	**50**
1	0.007	0.0015	0.023	0.099	0.442	3.754	7.147	13.029
2	0.009	0.014	0.026	0.067	0.239	1.940	3.769	6.742
4	0.010	0.017	0.036	0.081	0.310	1.119	2.177	4.241
8	0.020	0.038	0.058	0.103	0.309	1.179	1.976	3.201
16	0.044	0.076	0.152	0.388	0.425	0.945	1.544	2.580

Таблица 2: Среднее время выполнения программы в зависимости от размера задачи и количества потоков (в секундах).

5 График сравнения For и Task

Ниже представлен график, который иллюстрирует среднее время выполнения для каждого метода при различных размерах задачи. Он позволяет более наглядно сравнить производительность методов.

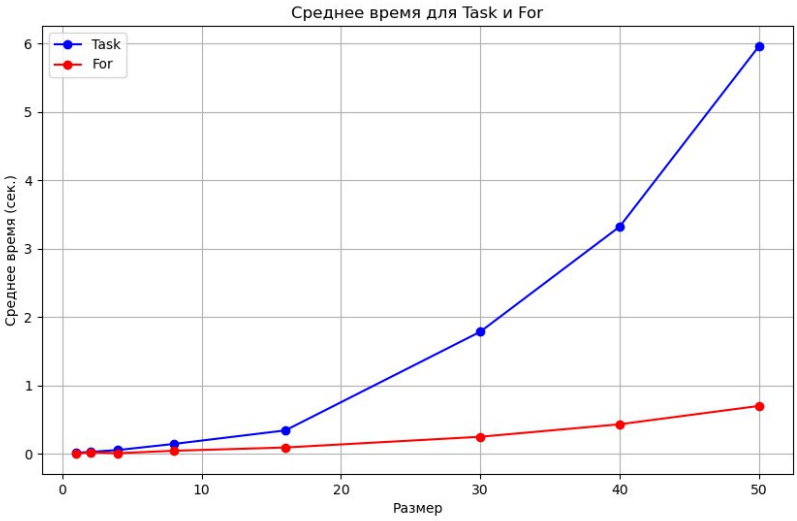

Рис. 3: Сравнение времени выполнения для методов `task` и `for`

Из полученных данных видно, что метод с использованием `task` показывает значительно более высокие времена выполнения по сравнению с методом `for`, особенно при увеличении размера задачи. На графике видно, что для маленьких размеров задач разница в производительности между методами не столь велика, однако с ростом размера задачи метод `task` показывает значительное ухудшение производительности.

Основной причиной этого может быть более высокая накладная нагрузка при использовании `task` в OpenMP, которая увеличивается с числом потоков и размером задачи. Метод `for`, напротив, показывает более стабильное поведение и лучше масштабируется при увеличении числа потоков.

6 Изменения для параллелизаци 2

6.1 Изменения

Для улучшения производительности программы были внесены изменения, заключающиеся в использовании библиотеки MPI для распараллеливания вычислений.

- Использование MPI для обмена данными между процессами.
- Распараллеливание вычислений: каждый процесс отвечает за часть сетки.
- Добавление функции `exchange_rows`, которая отвечает за обмен строками между соседними процессами.
- Использование MPI функций для синхронизации и сбора результатов (например, `MPI_Allreduce`, `MPI_Reduce`).

6.2 Эксперименты и результаты

В ходе экспериментов была проведена проверка производительности распараллеленной версии алгоритма. Мы измеряли время выполнения для разных размеров задачи и количества процессов.

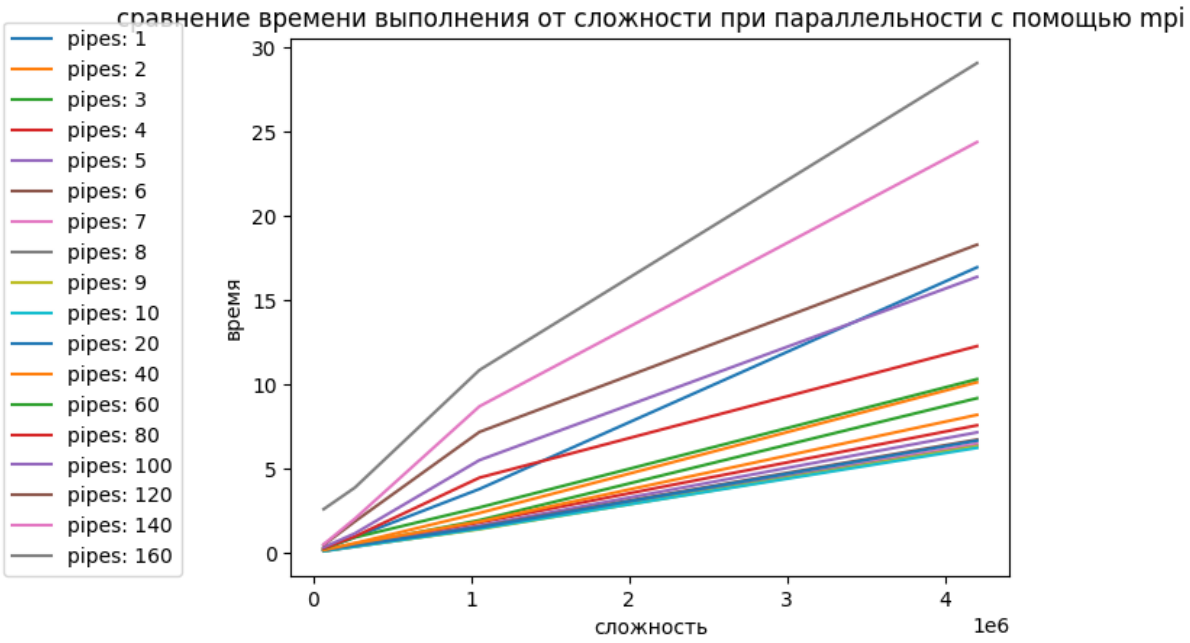

Рис. 4: Зависимость времени выполнения от числа потоков.

7 Заключение

В ходе работы была успешно распараллелена программа с использованием OpenMP. Параллелизация позволила значительно ускорить выполнение программы на многопроцессорных системах. Основные изменения включают добавление директив OpenMP для параллельного выполнения циклов в функциях `init()`, `relax()` и `resid()`. Результаты работы показали значительное улучшение производительности при увеличении числа потоков. В результате работы была также реализована параллельная версия алгоритма с использованием MPI. Эксперименты показали значительное ускорение работы алгоритма при увеличении числа процессов, что подтверждает эффективность использования параллелизма для данной задачи. Но наибольшее преимущество было именно благодаря MPI Это можно увидеть на общем графике всех 4 кодов:

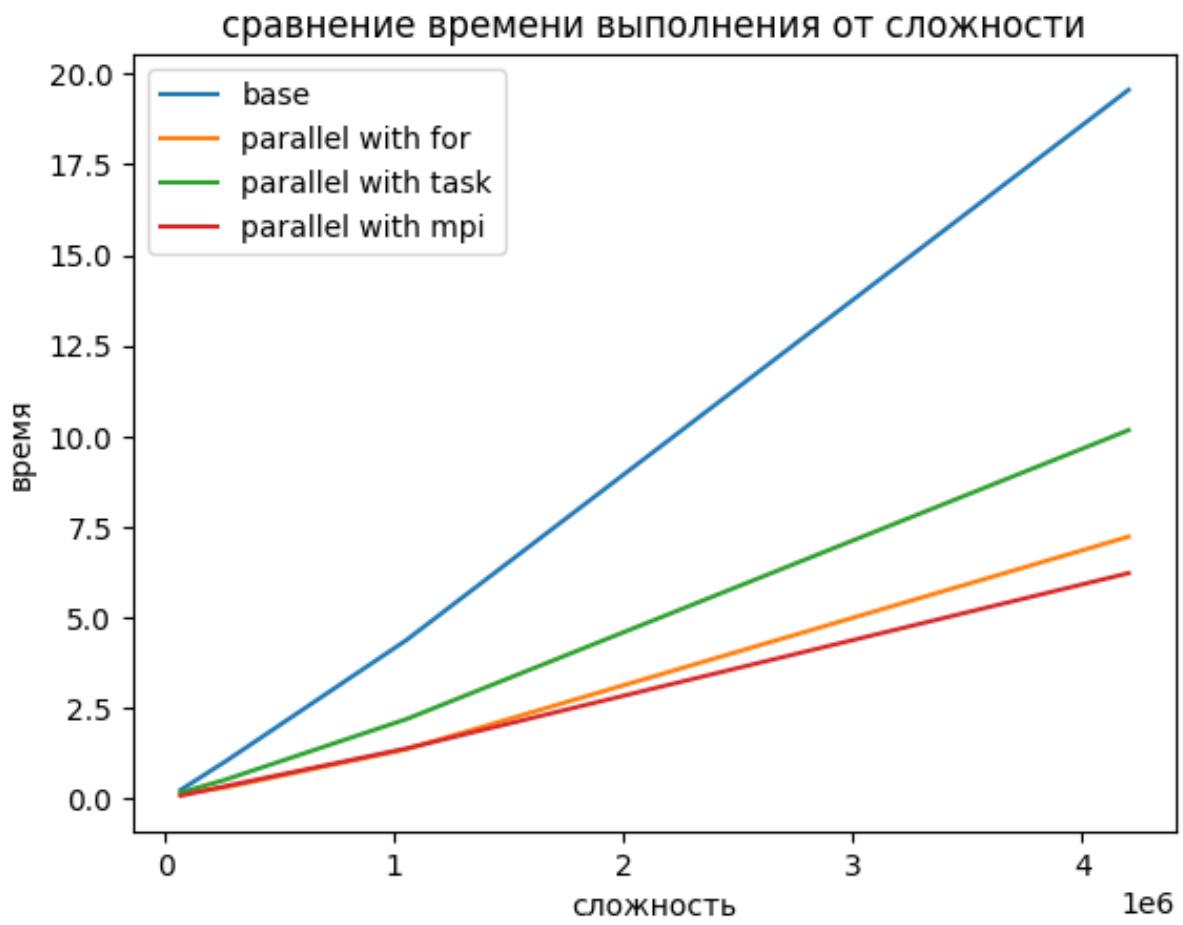

Рис. 5: График сравнения всех 4х реализаций