



Отчет по курсу
«Суперкомпьютерное моделирование и технологии»

Студент: Пикуров Даниил
Группа: № 617

Задача Дирихле для уравнения Пуассона в криволинейной области
Задание 5: Трапеция с вершинами $A(-3, 0)$, $B(3, 0)$, $C(2, 3)$, $D(-2, 3)$

1 Математическая постановка задачи

Требуется решить двумерную задачу Дирихле для уравнения Пуассона в криволинейной области (трапеции):

$$-\Delta u = f(x, y), \quad (x, y) \in D \quad (1)$$

$$u(x, y) = 0, \quad (x, y) \in \gamma \quad (2)$$

где:

- D - трапеция с вершинами $A(-3, 0), B(3, 0), C(2, 3), D(-2, 3)$
- γ - граница области D
- $f(x, y) = 1$ - правая часть уравнения

2 Численный метод решения

2.1 Метод фиктивных областей

Для решения задачи в криволинейной области применяется метод фиктивных областей. Исходная область D вкладывается в прямоугольник $\Pi = \{(x, y) : -3 \leq x \leq 3, 0 \leq y \leq 3\}$.

Вводится фиктивная область $\hat{D} = \Pi \setminus D$ и решается задача:

$$-\frac{\partial}{\partial x} \left(k(x, y) \frac{\partial v}{\partial x} \right) - \frac{\partial}{\partial y} \left(k(x, y) \frac{\partial v}{\partial y} \right) = F(x, y) \quad (3)$$

$$v(x, y) = 0, \quad (x, y) \in \Gamma \quad (4)$$

где:

- $k(x, y) = \begin{cases} 1, & (x, y) \in D \\ 1/\varepsilon, & (x, y) \in \hat{D} \end{cases}$
- $F(x, y) = \begin{cases} f(x, y), & (x, y) \in D \\ 0, & (x, y) \in \hat{D} \end{cases}$
- $\varepsilon = h^2, h = \max(h_x, h_y)$

2.2 Разностная схема

На прямоугольной сетке $\bar{\omega}_h = \bar{\omega}_1 \times \bar{\omega}_2$ строится разностная схема:

$$-\frac{1}{h_1} \left(a_{i+1,j} \frac{w_{i+1,j} - w_{ij}}{h_1} - a_{ij} \frac{w_{ij} - w_{i-1,j}}{h_1} \right) - \frac{1}{h_2} \left(b_{ij+1} \frac{w_{ij+1} - w_{ij}}{h_2} - b_{ij} \frac{w_{ij} - w_{ij-1}}{h_2} \right) = F_{ij} \quad (5)$$

Коэффициенты вычисляются по формулам:

$$a_{ij} = \frac{1}{h_2} \int_{y_{j-1/2}}^{y_{j+1/2}} k(x_{i-1/2}, t) dt, \quad b_{ij} = \frac{1}{h_1} \int_{x_{i-1/2}}^{x_{i+1/2}} k(t, y_{j-1/2}) dt \quad (6)$$

2.3 Метод сопряженных градиентов

Для решения СЛАУ применяется метод сопряженных градиентов с диагональным предобусловливанием. Алгоритм:

1. Начальное приближение: $w^{(0)} = 0$
2. Невязка: $r^{(0)} = B - Aw^{(0)}$
3. Решение системы предобусловливания: $Dz^{(0)} = r^{(0)}$
4. Направление спуска: $p^{(1)} = z^{(0)}$

5. Итерационный процесс:

$$\begin{aligned}\alpha_k &= \frac{(z^{(k-1)}, r^{(k-1)})}{(Ap^{(k)}, p^{(k)})} \\ w^{(k)} &= w^{(k-1)} + \alpha_k p^{(k)} \\ r^{(k)} &= r^{(k-1)} - \alpha_k Ap^{(k)}\end{aligned}$$

$$\begin{aligned}Dz^{(k)} &= r^{(k)} \\ \beta_k &= \frac{(z^{(k)}, r^{(k)})}{(z^{(k-1)}, r^{(k-1)})} \\ p^{(k+1)} &= z^{(k)} + \beta_k p^{(k)}\end{aligned}$$

3 Параллельная реализация

3.1 OpenMP реализация

Для параллелизации последовательного кода использованы директивы OpenMP:

- `#pragma omp parallel for collapse(2)` - для вложенных циклов по сетке
- `#pragma omp parallel for reduction(+:result)` - для скалярных произведений
- `omp_set_num_threads(num_threads)` - установка числа потоков
- `omp_get_wtime()` - для измерения времени

Основные параллелизуемые участки:

- Инициализация сеток и масок
- Вычисление коэффициентов a_{ij}, b_{ij}, F_{ij}
- Применение оператора A
- Решение системы с предобусловливателем D
- Скалярные произведения и обновления векторов

3.2 MPI реализация

Для распределенной параллелизации использована библиотека MPI. Реализован алгоритм двумерного разбиения прямоугольной области Π на домены (подобласти) с соблюдением следующих условий:

1. Отношение количества узлов по переменным x и y в каждом домене принадлежит диапазону $[1/2, 2]$
2. Количество узлов по переменным x и y любых двух доменов отличается не более, чем на единицу

Основные компоненты MPI-реализации:

- `MPI_Init, MPI_Finalize` - инициализация и завершение работы с MPI
- `MPI_Comm_rank, MPI_Comm_size` - определение ранга и размера коммуникатора
- `MPI_Sendrecv` - обмен граничными данными между соседними процессами в четырех направлениях (верх, низ, лево, право)
- `MPI_Allreduce` - глобальная редукция для скалярных произведений и вычисления нормы невязки (обеспечивает синхронизацию процессов)
- `MPI_Reduce` - сбор максимального времени выполнения среди всех процессов

- MPI_Gatherv - сбор локальных решений на процесс 0
- MPI_Wtime - измерение времени выполнения

Особенности реализации:

- Двумерная декомпозиция области с использованием структуры ProcessTopology2D, хранящей информацию о размерах локального домена, координатах процесса в сетке и границах с ghost cells
- Каждый процесс хранит локальную часть сетки с граничными ячейками (ghost cells) для обмена данными с соседними процессами
- Использование структуры ExchangeBuffers для предвыделенных буферов обмена граничными данными
- Скалярные произведения и нормы вычисляются локально на каждом процессе, затем суммируются через MPI_Allreduce для получения глобального значения.
- Финальное решение собирается на процессе 0 с использованием MPI_Gatherv для сохранения результатов

3.3 Гибридная MPI+OpenMP реализация

Для повышения производительности разработана гибридная реализация, объединяющая преимущества MPI и OpenMP. В MPI-код добавлены директивы OpenMP для параллелизации вычислений внутри каждого процесса.

Основные особенности гибридной реализации:

- Двумерное разбиение области между процессами MPI (как в чистой MPI реализации)
- Параллелизация вычислений внутри каждого процесса с помощью OpenMP
- Использование директив #pragma omp parallel for collapse(2) для вложенных циклов по сетке
- Использование #pragma omp parallel for collapse(2) reduction(+:local_result) для скалярных произведений и подсчета неизвестных
- Параллелизация всех основных операций: инициализация сеток, вычисление коэффициентов a_{ij} , b_{ij} , F_{ij} , применение оператора A , решение системы с предобусловливателем D , обновления векторов в методе сопряженных градиентов
- Параллелизация обмена граничными данными (копирование в/из предвыделенных буферов)
- Каждый процесс может использовать несколько потоков OpenMP для обработки своего локального домена

3.4 MPI+CUDA реализация

Для ускорения вычислений разработана реализация с использованием MPI и CUDA, которая переносит основные вычисления на графические процессоры (GPU).

Основные особенности MPI+CUDA реализации:

- Двумерное разбиение области между процессами MPI (как в чистой MPI реализации)
- Основные вычисления выполняются на GPU с помощью CUDA ядер:
 - apply_A_kernel - применение оператора A к вектору
 - solve_D_kernel - решение системы с предобусловливателем D
 - dot_product_kernel - вычисление скалярных произведений
 - update_vectors_kernel - обновление векторов в методе сопряженных градиентов
 - update_p_kernel - обновление направления спуска
 - init_residual_kernel - инициализация невязки

- Редукция выполняется с помощью библиотеки Thrust на GPU (без использования разделяемой памяти и atomic операций)
- Каждый MPI процесс использует свой GPU (распределение через `rank % device_count`)
- Данные копируются между CPU и GPU только при необходимости (обмен границами, сбор результатов)
- Измеряется детальное время выполнения всех операций: копирование данных на GPU, вычисления на GPU, коммуникации MPI, сбор результатов

Технические детали реализации:

- Использование `cudaMalloc` для выделения памяти на GPU
- `cudaMemcpy` для копирования данных между CPU и GPU
- CUDA события (`cudaEvent_t`) для точного измерения времени выполнения операций на GPU
- Ограничение compute capability до 3.5 (sm_35) или 6.0 (sm_60) в соответствии с требованиями задания
- Компиляция через Makefile с переменными `ARCH` и `HOST_COMP`

4 Результаты расчетов

4.1 Последовательная программа

Таблица 1: Результаты последовательных расчетов (малые сетки)

Сетка ($M \times N$)	Число итераций	Время (с)
10×10	33	0.00022
20×20	61	0.00116
40×40	119	0.00726

4.2 OpenMP программа для малых сеток

Таблица 2: Результаты OpenMP расчетов (сетка 40×40)

Число потоков	Число итераций	Время (с)	Ускорение
1	119	0.00961	0.76
4	119	0.00679	1.07
16	119	0.01708	0.43

4.3 OpenMP программа для больших сеток

Таблица 3: Таблица с результатами расчетов на ПВС IBM Polus (OpenMP код)

Количество OpenMP-нитей	Число точек сетки	Число итераций	Время решения (с)	Ускорение
2	400×600	1782	9.849	1.00
4	400×600	1782	5.912	1.67
8	400×600	1782	3.849	2.56
16	400×600	1782	3.165	3.11
4	800×1200	3596	43.907	1.00
8	800×1200	3596	28.979	1.52
16	800×1200	3596	23.4077	1.88
32	800×1200	3596	27.563	1.59

4.4 MPI программа

Таблица 4: Результаты MPI расчетов (сетка 40×40)

Количество процессов MPI	Число точек сетки	Число итераций	Время (с)	Ускорение
1	119	0.00680205	1.07	
2	119	0.0043437	1.67	
4	119	0.00482718	1.50	

Таблица 5: Таблица 2: Результаты расчетов MPI-программы на ПВС IBM Polus

Количество процессов MPI	Число точек сетки	Число итераций	Время решения (с)	Ускорение
2	400×600	1782	6.50059	1.00
4	400×600	1782	3.52396	1.85
8	400×600	1782	1.8521	3.51
16	400×600	1782	1.30841	4.97
4	800×1200	3596	26.9512	1.00
8	800×1200	3596	14.113	1.91
16	800×1200	3596	9.07348	2.97
20	800×1200	3596	8.33331	3.23

4.5 Гибридная MPI+OpenMP реализация

Таблица 6: Таблица 3: Результаты расчетов гибридной MPI+OpenMP программы (сетка 40×40)

Количество процессов	Потоков на процесс	Число итераций	Время (с)	Ускорение
1	4	119	0.00748	0.97
2	4	119	0.00611	1.19

Таблица 7: Таблица 3: Результаты расчетов гибридной MPI+OpenMP программы на ПВС IBM Polus

Количество процессов MPI	Количество OpenMP-нитей в процессе	Число точек сетки	Число итераций	Время решения (с)	Ускорение
2	1	400×600	1782	9.40471	1.00
2	2	400×600	1782	5.54675	1.70
2	4	400×600	1782	3.37292	2.79
2	8	400×600	1782	2.53256	3.71
4	1	800×1200	3596	40.213	1.00
4	2	800×1200	3596	23.1807	1.73
4	4	800×1200	3596	15.0471	2.67
4	8	800×1200	3596	10.8915	3.69

4.6 MPI+CUDA программа

Таблица 8: Результаты MPI+CUDA расчетов (сетка 2400×3600)

Количество процессов MPI	Число точек сетки	Число итераций	Общее время (с)	Время CG (с)	Ускорение
1	2400×3600	10242	822.745	779.571	3.57
2	2400×3600	10242	419.607	372.516	7.01

Таблица 9: Детальное время выполнения операций MPI+CUDA (сетка 2400×3600)

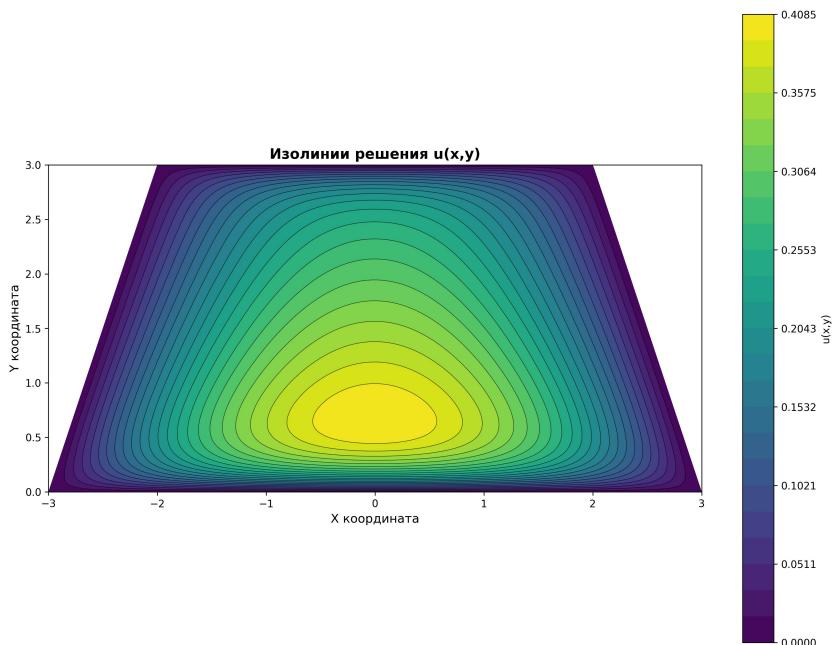
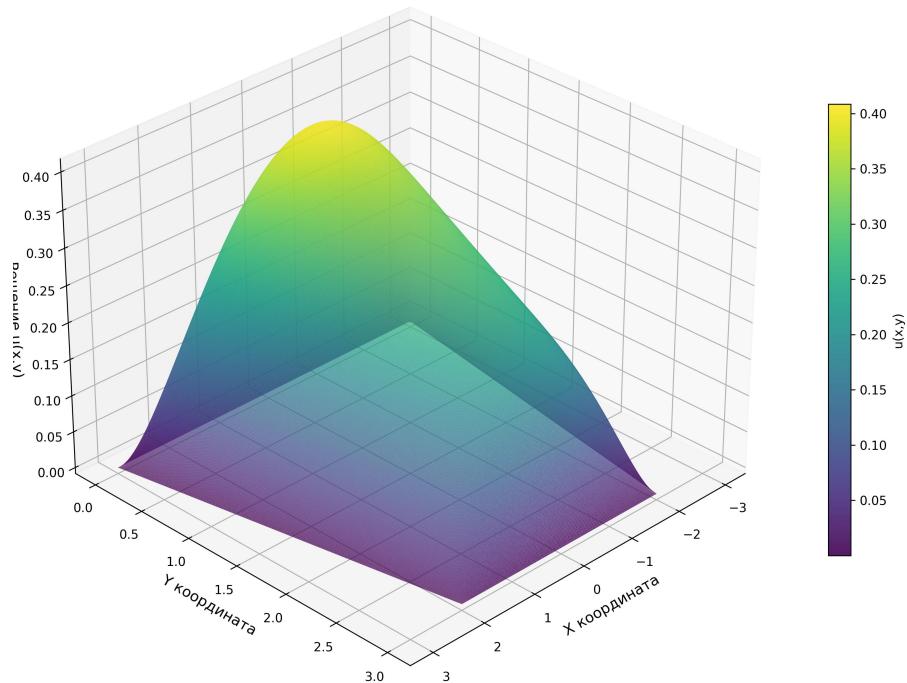
Операция	1 процесс (с)	2 процессы (с)
Инициализация и завершение		
Инициализация (Init)	19.229	7.322
Настройка (Setup)	17.813	6.934
Сбор результатов (Gather)	0.131	0.159
Метод сопряженных градиентов (CG)		
	779.571	372.516
Копирование данных CPU \leftrightarrow GPU		
Копирование на GPU	342.660	153.141
Копирование с GPU	378.378	182.834
<i>Всего копирование</i>	721.038	335.975
Вычисления на GPU		
Всего вычислений на GPU	58.358	59.096
Применение оператора A	6.405	3.528
Решение с предобусловливателем D	5.878	3.288
Скалярные произведения	35.646	47.951
Обновление векторов	6.801	3.805
Обновление направления p	3.613	2.039
Коммуникации MPI		
	0.015	24.284
Общее время		
	822.745	419.607

Таблица 10: Сравнение всех реализаций для сетки 2400×3600

Реализация	Конфигурация	Время (с)	Ускорение
Последовательная	1 процесс	2939.49	1.00
OpenMP	20 потоков	431.988	6.81
MPI+OpenMP	20 процессов, 8 потоков	255.755	11.49
MPI+CUDA	1 процесс	822.745	3.57
MPI+CUDA	2 процессы	419.607	7.01

4.7 Визуализация решения

3D визуализация решения уравнения Пуассона



5 Анализ результатов

5.1 Сходимость метода

- Метод сопряженных градиентов демонстрирует устойчивую сходимость на всех сетках
- Число итераций растет с увеличением размера сетки

5.2 Эффективность параллелизации OpenMP для больших сеток

- Для сетки 400×600 наблюдается хорошее ускорение:
 - 4 потока: ускорение 1.67
 - 8 потоков: ускорение 2.56
 - 16 потоков: ускорение 3.11
- Для сетки 800×1200 ускорение не столь большое:
 - 8 потоков: ускорение 1.52
 - 16 потоков: ускорение 1.88
 - 32 потока: ускорение 1.59

5.3 Эффективность параллелизации MPI для больших сеток

- Для сетки 400×600 :
 - 4 процесса: ускорение 1.85
 - 8 процессов: ускорение 3.51
 - 16 процессов: ускорение 4.97
- Для сетки 800×1200 :
 - 8 процессов: ускорение 1.91
 - 16 процессов: ускорение 2.97
 - 20 процессов: ускорение 3.23

5.4 Анализ масштабируемости

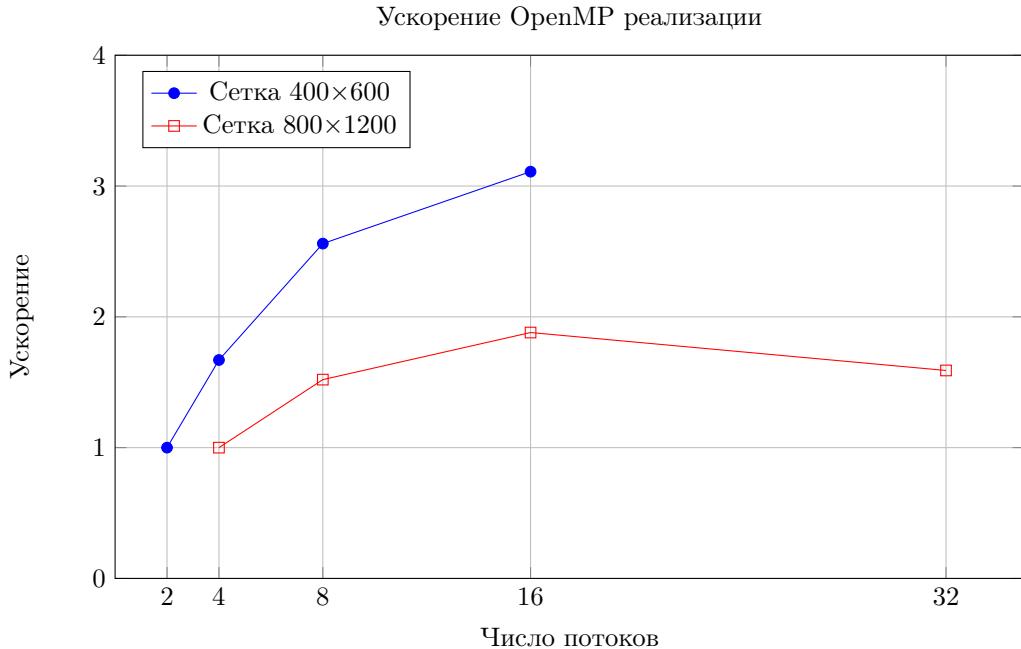


Рис. 1: Сравнение ускорения OpenMP для разных размеров сеток

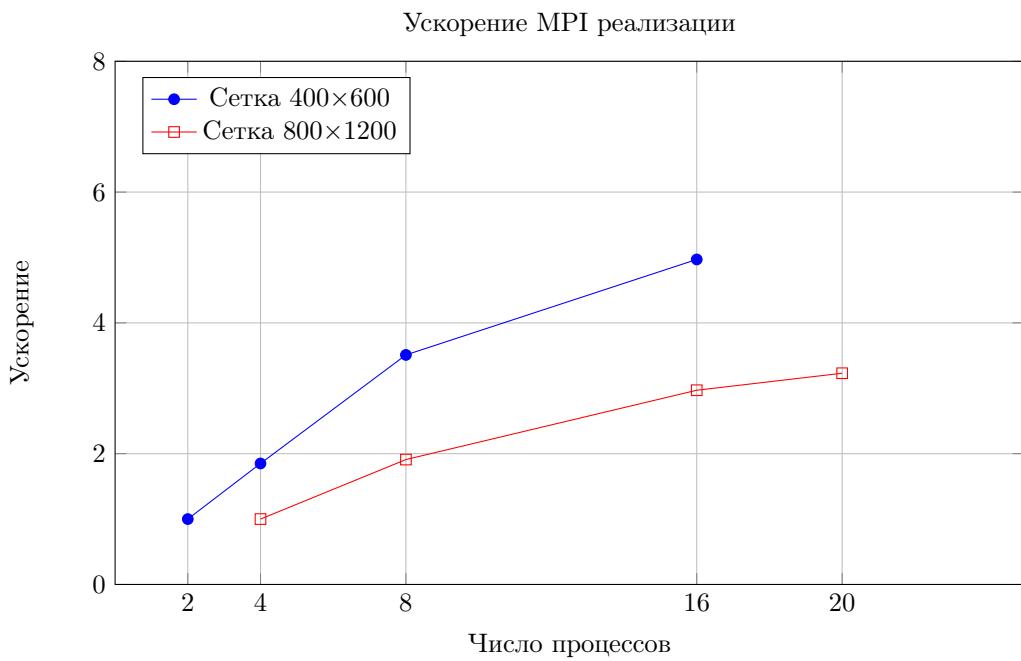


Рис. 2: Сравнение ускорения MPI для разных размеров сеток

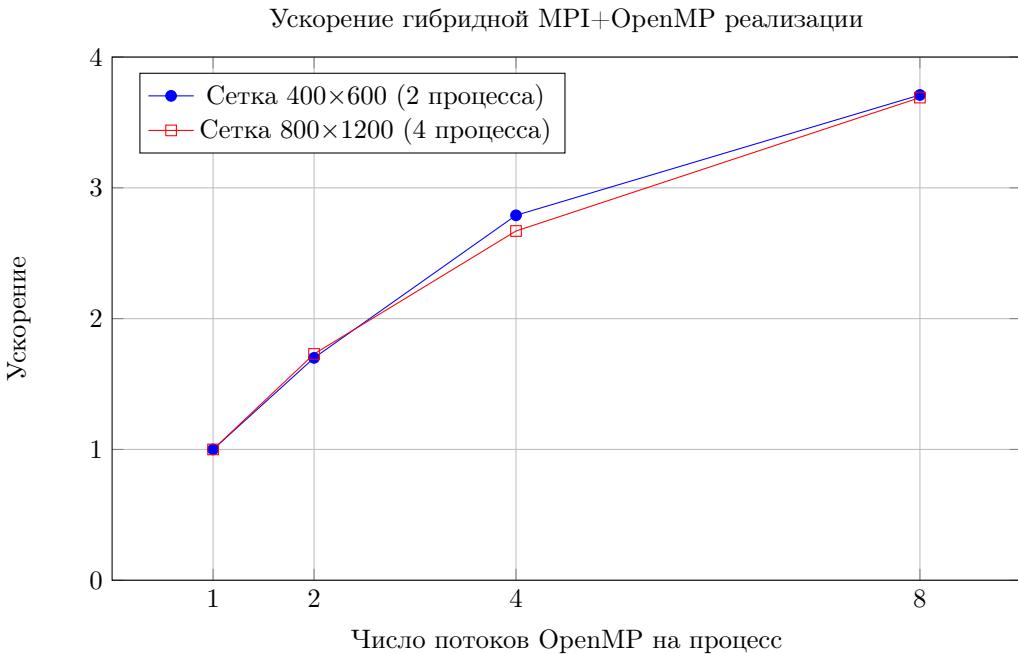


Рис. 3: Сравнение ускорения гибридной MPI+OpenMP для разных размеров сеток

1. Для малых сеток (40×40):

- **OpenMP:** Накладные расходы превышают выигрыш от параллелизма (ускорение 0.76 на 1 потоке относительно последовательной программы, 1.07 на 4 потоках)
- **MPI:** Наблюдается ускорение относительно последовательной программы: 1.07 на 1 процессе, 1.67 на 2 процессах, 1.50 на 4 процессах
- **Гибридная MPI+OpenMP:** Для малых сеток гибридный подход показывает результаты, близкие к чистой MPI реализации (0.97 на 1 процессе с 4 потоками, 1.19 на 2 процессах с 4 потоками), что объясняется преобладанием накладных расходов над выигрышем от параллелизма

2. Для средних сеток (400×600):

- **OpenMP:** Наблюдается хорошее ускорение до 16 потоков (ускорение 3.11) благодаря достаточному объему вычислений на поток, хорошему балансу между вычислениями и накладными расходами, эффективному использованию кэша
- **MPI:** Отличная масштабируемость до 16 процессов (ускорение 4.97 относительно 2 процессов), что значительно лучше OpenMP (ускорение 3.11)
- **Гибридная MPI+OpenMP:** Демонстрирует эффективное масштабирование по потокам внутри процессов: ускорение от 1.00 (1 поток на процесс) до 3.71 (8 потоков на процесс) при использовании 2 процессов.

3. Для больших сеток (800×1200):

- **OpenMP:** Ускорение ограничено из-за того, что на сервере до 20 ядер в сумме и 32 потока не выполняются независимо, из-за этого возникают накладные расходы, которые замедляют программу
- **MPI:** Показывает хорошую масштабируемость до 20 процессов (ускорение 3.23 относительно 4 процессов), что более чем в 2 раза превосходит OpenMP.
- **Гибридная MPI+OpenMP:** Демонстрирует практически идентичное ускорение для сеток 400×600 и 800×1200 при одинаковом числе потоков на процесс, что подтверждает его эффективность для больших задач.

4. Для очень больших сеток (2400×3600):

- **Последовательная программа:** Время выполнения составляет 2939.49 секунд, что демонстрирует необходимость параллелизации для таких задач

- **OpenMP (20 потоков):** Ускорение 6.81 относительно последовательной программы (431.988 секунд), что показывает хорошую эффективность для задач с большим объемом вычислений
- **MPI+OpenMP (20 процессов, 8 потоков):** Наилучший результат - ускорение 11.49 (255.755 секунд), демонстрируя эффективность гибридного подхода для очень больших задач
- **MPI+CUDA (1 процесс):** Ускорение 3.57 (822.745 секунд) относительно последовательной программы. Основное время тратится на копирование данных между CPU и GPU (342.66 с на копирование на GPU, 378.378 с на копирование с GPU), что составляет около 88% от общего времени CG метода. Время вычислений на GPU составляет только 58.358 секунд, что показывает высокую эффективность GPU для вычислений
- **MPI+CUDA (2 процесса):** Ускорение 7.01 (419.607 секунд) относительно последовательной программы и 1.96 относительно 1 процесса. Время копирования данных уменьшилось пропорционально размеру локальной области (153.141 с на копирование на GPU, 182.834 с на копирование с GPU). Время коммуникаций MPI составляет 24.284 секунды, что является приемлемым для такого размера задачи

Анализ производительности MPI+CUDA:

- **Инициализация и завершение:** Время инициализации (19.2 с для 1 процесса, 7.3 с для 2 процессов) включает создание топологии процессов, выделение памяти на GPU и инициализацию CUDA. Время настройки (17.8 с для 1 процесса, 6.9 с для 2 процессов) включает вычисление коэффициентов a_{ij} , b_{ij} , F_{ij} и масок. Время сбора результатов минимально (0.13-0.16 с), так как выполняется только один раз в конце программы.
- **Копирование данных CPU ↔ GPU:** Основное узкое место реализации. Для 1 процесса копирование занимает 721.0 секунд (88% от времени CG метода), из которых 342.7 с - копирование на GPU и 378.4 с - копирование с GPU. Для 2 процессов время копирования уменьшается до 336.0 секунд (90% от времени CG метода), что объясняется уменьшением размера локальной области каждого процесса. Копирование выполняется синхронно перед и после каждой итерации метода сопряженных градиентов для обмена граничными данными.
- **Вычисления на GPU:** Выполняются очень эффективно - всего 58.4 с для 1 процесса и 59.1 с для 2 процессов. Основные операции:
 - Применение оператора A : 6.4 с (1 процесс) и 3.5 с (2 процесса) - уменьшение времени объясняется меньшим размером локальной области
 - Решение с предобусловливателем D : 5.9 с (1 процесс) и 3.3 с (2 процесса)
 - Скалярные произведения: 35.6 с (1 процесс) и 48.0 с (2 процесса) - увеличение для 2 процессов связано с необходимостью выполнения редукции через Thrust на каждом процессе и последующей глобальной редукции через MPI
 - Обновление векторов: 6.8 с (1 процесс) и 3.8 с (2 процесса)
 - Обновление направления p : 3.6 с (1 процесс) и 2.0 с (2 процесса)
- **Коммуникации MPI:** Для 1 процесса коммуникации практически отсутствуют (0.015 с), так как нет обмена данными между процессами. Для 2 процессов время коммуникаций составляет 24.3 секунды (6.5% от времени CG метода), что включает обмен граничными данными между соседними процессами. Коммуникации выполняются синхронно через MPI_Sendrecv, поэтому время коммуникаций измеряется точно и не перекрывается с вычислениями.
- **Масштабирование:** При переходе от 1 к 2 процессам общее время уменьшается с 822.7 с до 419.6 с (ускорение 1.96), время CG метода - с 779.6 с до 372.5 с (ускорение 2.09). Время копирования данных уменьшается пропорционально размеру локальной области, время вычислений на GPU остается примерно постоянным, а время коммуникаций увеличивается из-за необходимости обмена данными между процессами.

6 Исходный код и репозиторий

Весь исходный код проекта, включая последовательную и параллельные реализации, а также данный отчет, размещены в Git-репозитории:

<https://github.com/Pikudan/SKModel.git>

Структура репозитория:

- `main_seq.cpp` - последовательная реализация (ветка `sequential`)
- `main_openmp.cpp` - реализация с использованием OpenMP (ветка `openmp`)
- `main_mpi.cpp` - реализация с использованием MPI (ветка `mpi`)
- `main_hybrid.cpp` - гибридная MPI+OpenMP реализация (ветка `hybrid`)
- `main_mpi_cuda.cpp` - реализация MPI+CUDA (ветка `mpi-cuda`)
- `Makefile` - файл сборки для MPI+CUDA версии
- `*.lsf` - скрипты для запуска на кластере IBM Polus
- `main.tex`, `main.pdf` - исходный код и скомпилированный отчет
- `README.md` - описание проекта и инструкции по сборке и запуску

В репозитории сохранена полная история коммитов, отражающая процесс разработки и распараллеливания задачи:

- Начальная последовательная реализация
- Добавление OpenMP директив
- Реализация MPI версии с двумерной декомпозицией
- Разработка гибридной MPI+OpenMP версии
- Реализация MPI+CUDA версии с использованием GPU
- Оптимизация и улучшение производительности на каждом этапе

Каждая параллельная реализация находится в отдельной ветке Git, что позволяет легко переключаться между версиями и сравнивать результаты. Финальная версия со всеми реализациями находится в ветке `main`.

7 Выводы

1. Разработаны параллельные реализации с использованием OpenMP и MPI для решения задачи Дирихле для уравнения Пуассона в криволинейной области:
 - OpenMP реализация демонстрирует хорошую эффективность для средних сеток (ускорение до 3.11 на 16 потоках)
 - MPI реализация показывает отличную масштабируемость для больших сеток, превосходя OpenMP (ускорение 4.97 на 16 процессах относительно 2 процессов против 3.11 на 16 потоках OpenMP для сетки 400×600)
 - Гибридная MPI+OpenMP реализация демонстрирует отличную эффективность, для сетки 400×600 с 2 процессами и сетки 800×1200 с 4 процессами ускорение совпадает, что показывает эффективность гибридного подхода для больших задач.
2. Наблюдается **закон Амдала**: с ростом числа потоков/процессов эффективность параллелизации снижается, но для MPI этот эффект выражен слабее.
3. Число итераций и численное решение остается постоянным для фиксированной сетки при разном числе потоков/процессов, что подтверждает детерминированность алгоритма
4. На сервере было доступно только две видеокарты, с учетом линейного ускорения на 4 гпу должно было быть больше чем на 160 потоках MPI+OpenMP