

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ**

**ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**(ДГТУ)**

Факультет Отдел магистратуры

Кафедра Математика и информатика

Предмет Параллельное программирование

**ОТЧЕТ ПО ЛАБОРАТОРНЫМ РАБОТАМ**

**Тема «Технологии параллельного программирования»**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Автор |  | | |  | Дроботов С.В. | | | |
|  | (подпись, дата) | | |  |  | | | |
| Обозначение | ОМ.25 0000.000 | | Группа | | | | МСК11 |
| Направление подготовки | | 09.04.02 Информационные системы и технологии | | | | | |
| Профиль | Искусственный интеллект, математическое моделирование и суперкомпьютерные технологии в разработке информационных систем | | | | | | |
| Проверил |  | | |  | | Литвинов В.Н. | |
|  | (подпись, дата) | | |  | |  | |

Ростов-на-Дону

2025г.

Задание 14. Создание проекта в среде MS Visual Studio с поддержкой MPI

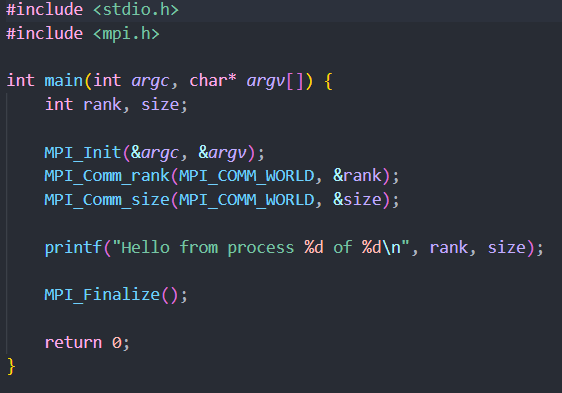
Цель

Создайте проект в среде Visual Studio 2010 с поддержкой MPI.

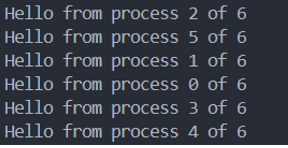
Выполнение

1. **Инициализация MPI**:
   * MPI\_Init(&argc, &argv) — запускает среду выполнения MPI.
   * Аргументы argc и argv передаются для корректной обработки параметров командной строки.
2. **Получение информации о процессе**:
   * MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank) — возвращает уникальный идентификатор (ранг) текущего процесса в коммуникаторе MPI\_COMM\_WORLD (от 0 до size-1).
   * MPI\_Comm\_size(MPI\_COMM\_WORLD, &size) — возвращает общее количество процессов в коммуникаторе.
3. **Вывод сообщения**:
   * Каждый процесс печатает строку вида: Hello from process X of Y, где X — его ранг, Y — общее количество процессов.
   * **Важно**: вывод может быть "перемешан", так как процессы работают параллельно и выводят данные независимо.
4. **Завершение работы**:
   * MPI\_Finalize() — корректно завершает работу с MPI. Все MPI-функции должны быть вызваны до этого.

Пример кода:



Пример вывода:



Задание 15. Программа «I am!»

Цель

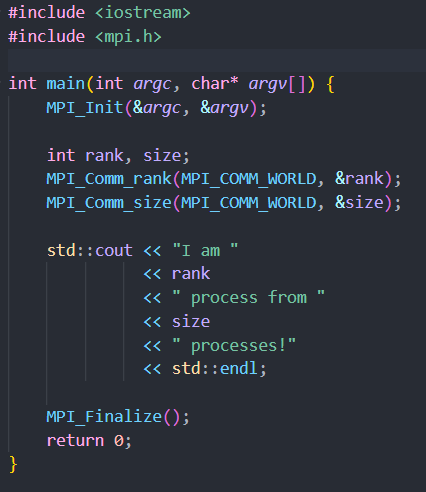
Напишите программу, в которой каждый процесс выводит на экран свой номер и общее количество процессов в приложении в формате: I am process from processes!

Выполнение

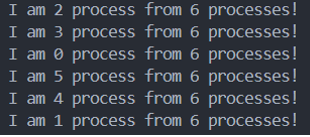
1. **Инициализация MPI**:
   * MPI\_Init(&argc, &argv) — запускает среду выполнения MPI. Корректно обрабатывает аргументы командной строки.
2. **Получение информации о процессе**:
   * MPI\_Comm\_rank — возвращает ранг процесса (уникальный ID в диапазоне [0, size-1]).
   * MPI\_Comm\_size — возвращает общее количество процессов в коммуникаторе MPI\_COMM\_WORLD.
3. **Вывод сообщения**:
   * Используется std::cout для вывода строки вида: I am X process from Y processes!, где X — ранг процесса, Y — общее число процессов.
   * **Особенность**: В C++ std::cout может буферизовать вывод, что иногда приводит к "перемешанным" строкам из-за параллельной работы процессов.
4. **Завершение работы:**

* MPI\_Finalize() — корректно завершает работу с MPI. Все MPI-функции вызываются до этой точки.

Пример кода:



Пример вывода:



Задание 16. Программа «На первый-второй рассчитайся!»

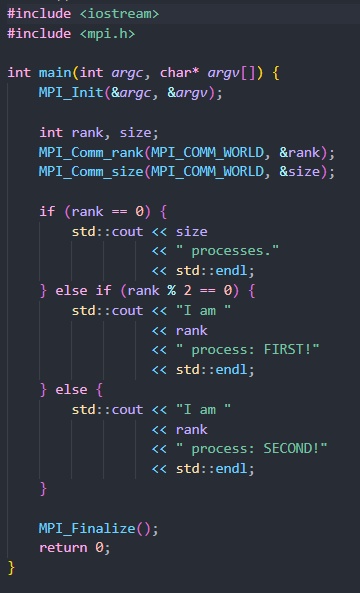
Цель

Напишите программу, в которой каждый процесс с четным номером выводит на экран строку «I am : FIRST!», а каждый процесс с нечетным номером – «I am : SECOND!». Процесс с номером 0 должен вывести на экран общее количество процессов в приложении в формате « processes.».

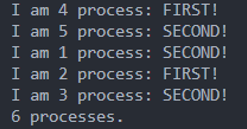
Выполнение

1. **Инициализация MPI**:
   * MPI\_Init инициализирует среду выполнения MPI.
   * MPI\_Comm\_rank и MPI\_Comm\_size получают ранг процесса и общее число процессов.
2. **Логика вывода**:
   * Процесс 0 выводит общее количество процессов (size).
   * Процессы с **чётными рангами** (2, 4, 6...) выводят "FIRST!".
   * Процессы с **нечётными рангами** (1, 3, 5...) выводят "SECOND!".

Пример кода:



Пример вывода:



Задание 17. Коммуникации «точка-точка»: простые блокирующие обмены

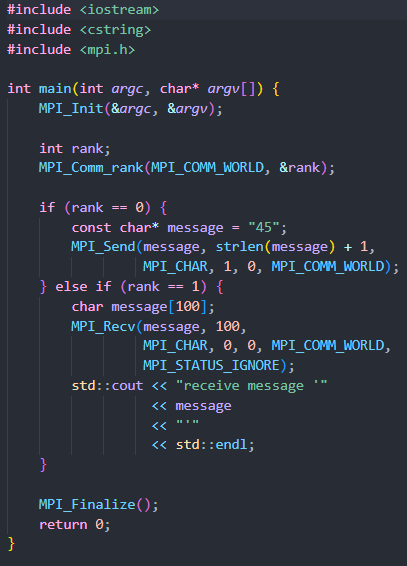
Цель

Изучите основные MPI-функции блокирующей передачи сообщений точка-точка MPI\_Send и MPI\_Recv. Напишите MPI-программу, в которой с помощью данных функций процесс с номером 0 отправляет сообщение процессу с номером 1. Процесс 1 выводит полученное сообщение на экран.

Выполнение

1. **Инициализация MPI**:
   * MPI\_Init инициализирует среду выполнения MPI.
   * MPI\_Comm\_rank определяет ранг текущего процесса.
2. **Логика работы**:
   * **Процесс 0**:
     + Создаёт сообщение "45" (включая нуль-терминатор \0).
     + Отправляет его процессу 1 с помощью MPI\_Send.
     + Параметры:
       - strlen(message) + 1 — длина сообщения + 1 байт для \0.
       - MPI\_CHAR — тип данных (символы).
       - 1 — получатель (ранг процесса).
       - 0 — тег сообщения.
   * **Процесс 1**:
     + Принимает сообщение в буфер message размером 100 символов.
     + Выводит полученное сообщение.
3. **Корректность**:
   * Размер буфера (100) достаточен для сообщения "45" (3 байта: '4', '5', \0).
   * Использование MPI\_STATUS\_IGNORE допустимо, если не требуется проверять статус приёма.
   * Блокирующие функции гарантируют, что процесс 0 не завершится раньше передачи данных, а процесс 1 дождётся сообщения.

Пример кода:



Пример вывода:



Задание 18. Коммуникации «точка-точка»: схема «эстафетная палочка»

Цель

Напишите MPI-программу, реализующую при помощи блокирующих функций посылки сообщений типа точка-точка схему коммуникации процессов «эстафетная палочка», в которой каждый процесс дожидается сообщения от предыдущего и потом посылает следующему (см. рис. 1). В качестве передаваемого сообщения используйте на процессе 0 его номер, на остальных процессах – инкрементированное полученное сообщение.

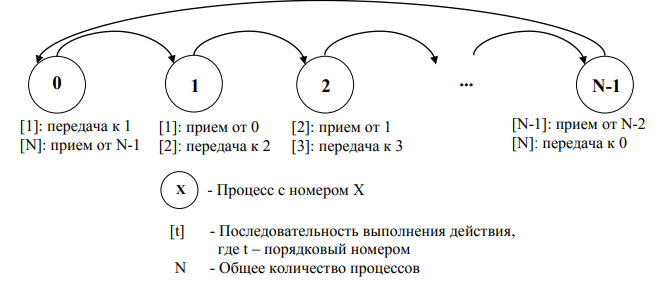
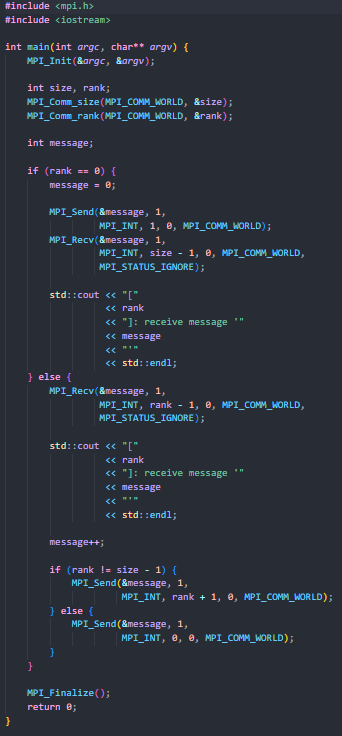


Рис. 1. Схема коммуникации процессов «эстафетная палочка»

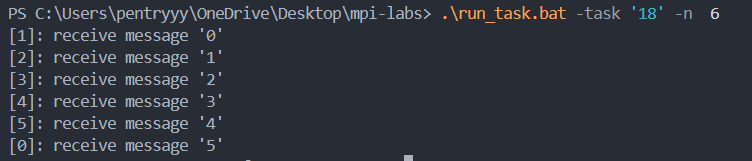
Выполнение

1. **Логика работы**:
   * **Процесс 0**:
     + Инициирует передачу, отправляя message = 0 процессу 1.
     + Принимает финальное сообщение от последнего процесса (size - 1).
   * **Процессы 1, 2, ..., size-1**:
     + Принимают сообщение от предыдущего процесса (rank - 1).
     + Увеличивают значение сообщения на 1.
     + Отправляют его следующему процессу (для rank = size-1 следующий — процесс 0).
2. **Корректность**:
   * **Блокирующие операции**: MPI\_Send и MPI\_Recv синхронизируют передачу, исключая гонки данных.
   * **Кольцевая топология**: Последний процесс передаёт сообщение обратно процессу 0.

Пример кода:



Пример вывода:



Задание 19. Коммуникации «точка-точка»: схема «мастер-рабочие»

Цель

Напишите MPI-программу, реализующую при помощи блокирующих функций посылки сообщений типа точка-точка схему коммуникации процессов «master-slave», в которой один процесс, называемый master, принимает сообщение от остальных процессов, называемых slave (см. рис. 2). В качестве передаваемого сообщения используйте номер процесса. Master-процесс должен вывести на экран все полученные сообщения.

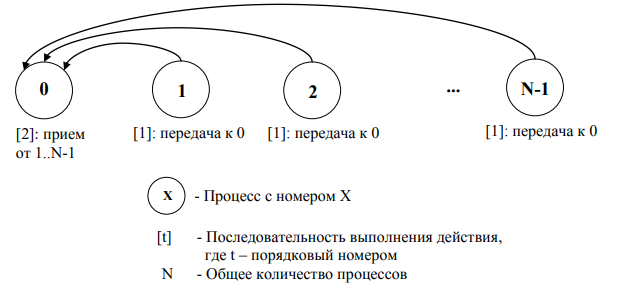
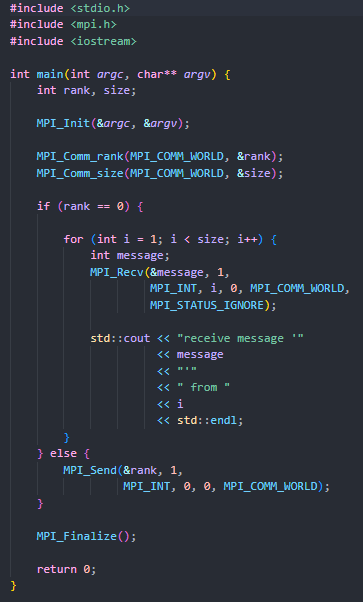


Рис. 2. Схема коммуникации процессов «master-slave»

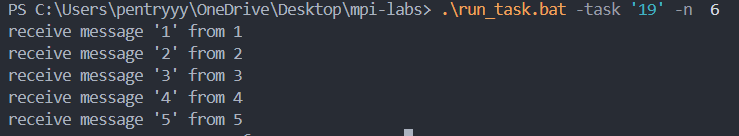
Выполнение

1. **Логика работы**:
   * **Мастер (ранг 0)**:
     + В цикле принимает сообщения от всех рабочих процессов (1, 2, ..., size-1).
     + Выводит полученное сообщение (ранг отправителя) в формате:  
       receive message 'X' from Y, где X — значение ранга, Y — номер процесса.
   * **Рабочие (ранг > 0)**:
     + Отправляют свой ранг (rank) мастеру с помощью MPI\_Send.
2. **Корректность**:
   * **Блокирующие операции**: Мастер последовательно принимает сообщения от каждого рабочего, что гарантирует порядок вывода.
   * **Отсутствие гонок данных**: Каждый рабочий отправляет уникальное сообщение (свой ранг), конфликтов нет.

Пример кода:



Пример вывода:



Задание 20. Коммуникации «точка-точка»: простые неблокирующие обмены

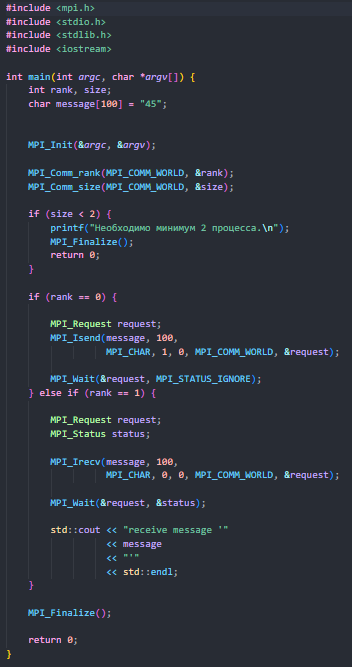
Цель

Изучите основные MPI-функции неблокирующей передачи сообщений точка-точка MPI\_Isend, MPI\_Irecv, MPI\_Wait. Напишите MPI-программу, в которой с помощью данных функций процесс с номером 0 отправляет сообщение процессу с номером 1. Процесс 1 выводит полученное сообщение на экран.

Выполнение

1. **Инициализация MPI**:
   * MPI\_Init инициализирует среду выполнения MPI.
   * MPI\_Comm\_rank и MPI\_Comm\_size получают ранг процесса и общее количество процессов.
2. **Логика работы**:
   * **Процесс 0**:
     + Инициализирует сообщение "45" (3 символа: '4', '5', \0).
     + Отправляет его процессу 1 через MPI\_Isend (неблокирующая операция).
     + MPI\_Wait блокирует выполнение до завершения отправки.
   * **Процесс 1**:
     + Принимает сообщение через MPI\_Irecv (неблокирующий приём).
     + MPI\_Wait ожидает завершения приёма.
     + Выводит сообщение.
3. **Корректность**:
   * Размер сообщения (100) избыточен, но безопасен.
   * Использование MPI\_Wait гарантирует завершение операций до продолжения работы.
   * Проверка if (size < 2) предотвращает запуск с одним процессом.

Пример кода:



Пример вывода:



Задание 21. Коммуникации «точка-точка»: схема «сдвиг по кольцу»

Цель

Напишите MPI-программу, реализующую при помощи блокирующих 0 1 2 N-1 Х - Процесс с номером Х [t] - Последовательность выполнения действия, где t – порядковый номером N - Общее количество процессов [2]: прием от 1..N-1 ... [1]: передача к 0 [1]: передача к 0 [1]: передача к 0 15 функций посылки сообщений типа точка-точка схему коммуникации процессов «сдвиг по кольцу», в которой осуществляются одновременные посылка и прием сообщений всеми процессами (см. рис. 3). В качестве передаваемого сообщения используйте номер процесса.

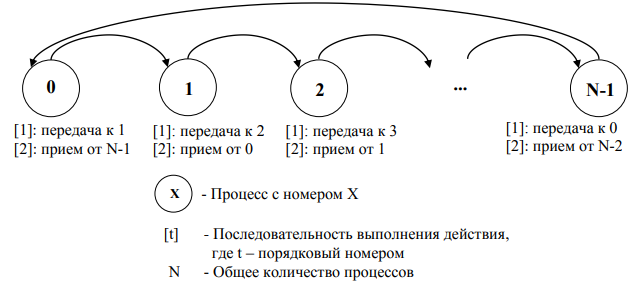
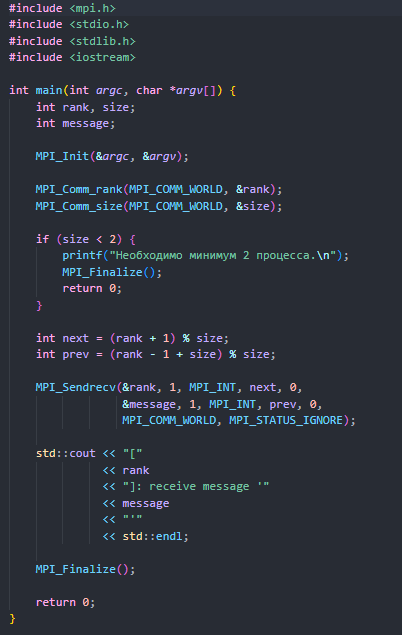


Рис. 3. Схема коммуникации процессов «сдвиг по кольцу»

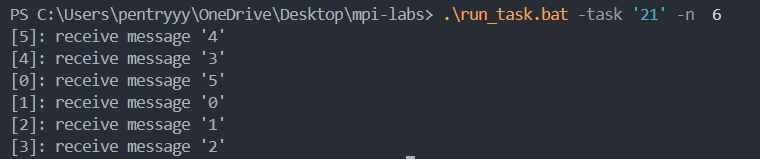
Выполнение

1. **Инициализация MPI**:
   * MPI\_Init, MPI\_Comm\_rank, MPI\_Comm\_size — стандартная инициализация.
2. **Кольцевая топология**:
   * next — следующий процесс в кольце ((rank + 1) % size).
   * prev — предыдущий процесс ((rank - 1 + size) % size).  
     Например, для 4 процессов:
   * Процесс 0: next = 1, prev = 3.
   * Процесс 3: next = 0, prev = 2.
3. **Функция**MPI\_Sendrecv:
   * **Отправка**: Передаёт rank следующему процессу (next).
   * **Приём**: Получает message от предыдущего процесса (prev).
   * **Особенность**: MPI\_Sendrecv объединяет отправку и приём в одну атомарную операцию, избегая deadlock.
4. **Пример работы для 3 процессов**:
   * Процесс 0 отправляет 0 → процесс 1, принимает от процесса 2.
   * Процесс 1 отправляет 1 → процесс 2, принимает от процесса 0.
   * Процесс 2 отправляет 2 → процесс 0, принимает от процесса 1.

Пример кода:



Пример вывода:



Задание 22. Коммуникации «точка-точка»: схема «каждый каждому»

Цель

Напишите MPI-программу, реализующую при помощи блокирующих функций посылки сообщений типа точка-точка схему коммуникации процессов «каждый каждому», в которой осуществляется пересылка сообщения от каждого процесса каждому (см. рис. 4). В качестве передаваемого сообщения используйте номер процесса. Каждый процесс должен вывести на экран все полученные сообщения.

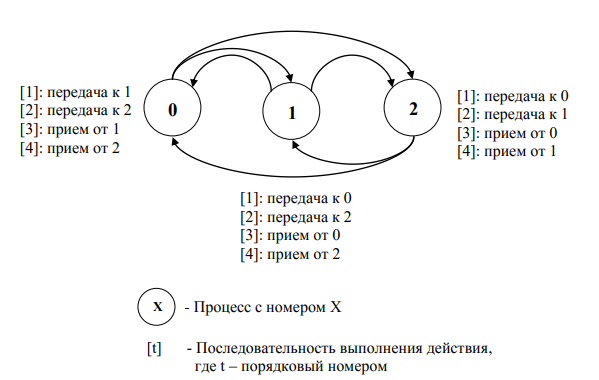
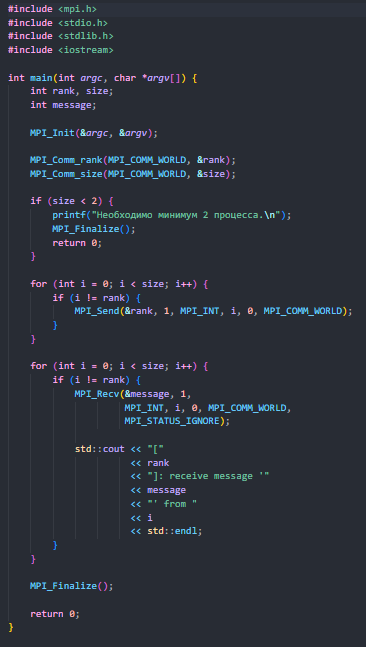


Рис. 4. Схема коммуникации процессов «каждый каждому» на примере 3-х процессов

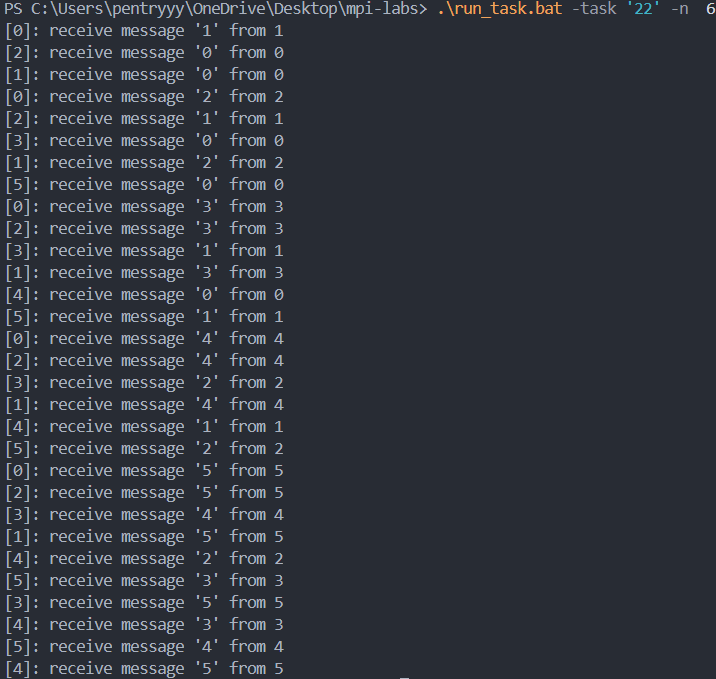
Выполнение

1. **Инициализация MPI**:
   * MPI\_Init, MPI\_Comm\_rank, MPI\_Comm\_size — стандартные функции для старта MPI, получения ранга и числа процессов.
2. **Логика работы**:
   * **Отправка**: Каждый процесс отправляет свой rank всем другим процессам (кроме себя) через цикл for.
   * **Приём**: Каждый процесс принимает rank от всех других процессов (кроме себя) через второй цикл for.
3. **Пример работы для 3 процессов**:
   * Процесс 0 получает: 1 от процесса 1, 2 от процесса 2.
   * Процесс 1 получает: 0 от процесса 0, 2 от процесса 2.
   * Процесс 2 получает: 0 от процесса 0, 1 от процесса 1.

Пример кода:



Пример вывода:



Задание 23. Коллективные коммуникации: широковещательная рассылка данных

Цель

1. Изучите MPI-функцию широковещательной рассылки данных MPI\_Bcast. Напишите MPI-программу, которая в строке длины n определяет количество вхождений символов. Ввод данных должен осуществляться процессом с номером 0. Для рассылки строки поиска и ее длины по процессам используйте функцию MPI\_Bcast.
2. Перепишите программу, используя вместо функции MPI\_Bcast функции коммуникации «точка-точка». Сравните эффективность выполнения программ с коллективными и точечными обменами.

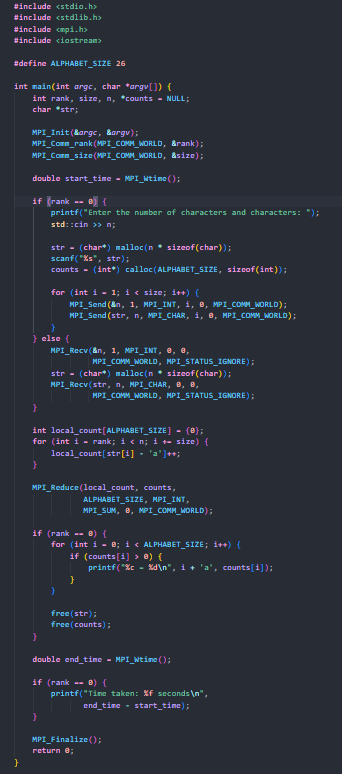
Выполнение

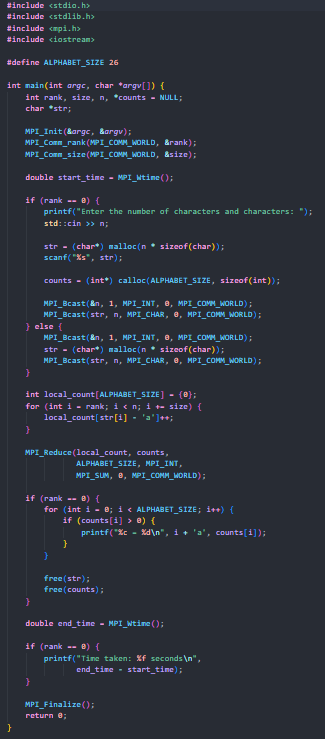
**Ключевые различия**

1. **MPI\_Bcast vs MPI\_Send/Recv**:
   * MPI\_Bcast — коллективная операция, которая эффективно рассылает данные всем процессам за *O*(log*N*) шагов (например, через двоичное дерево).
   * MPI\_Send/Recv — точечные операции, требующие O(N) шагов для рассылки.
2. **Выделение памяти**:
   * В первом коде процессы (кроме 0) получают n через MPI\_Recv, затем выделяют память.
   * Во втором коде все процессы получают n через MPI\_Bcast, что позволяет им выделить память одновременно.
3. **Производительность**:
   * MPI\_Bcast уменьшает задержки за счёт оптимизированных алгоритмов рассылки.
   * MPI\_Send/Recv создаёт нагрузку на процесс 0, так как он отправляет данные последовательно.

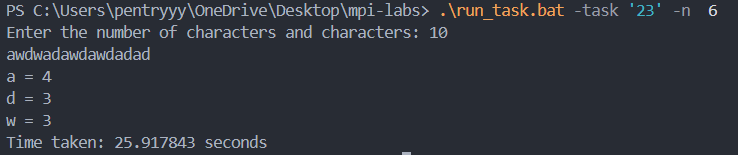
Код с MPI\_Bcast более эффективен и читаем. Коллективные операции (Bcast, Reduce) — стандартный подход для задач с однородными данными.

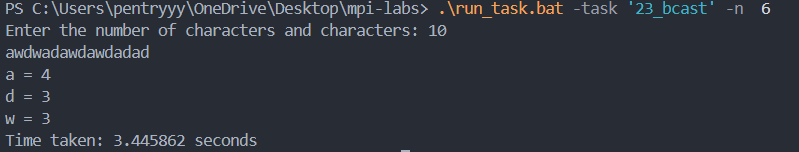
Пример кода для обычной реализации и bcast:





Пример вывода для обычной реализации и bcast:





Задание 24. Коллективные коммуникации: операции редукции

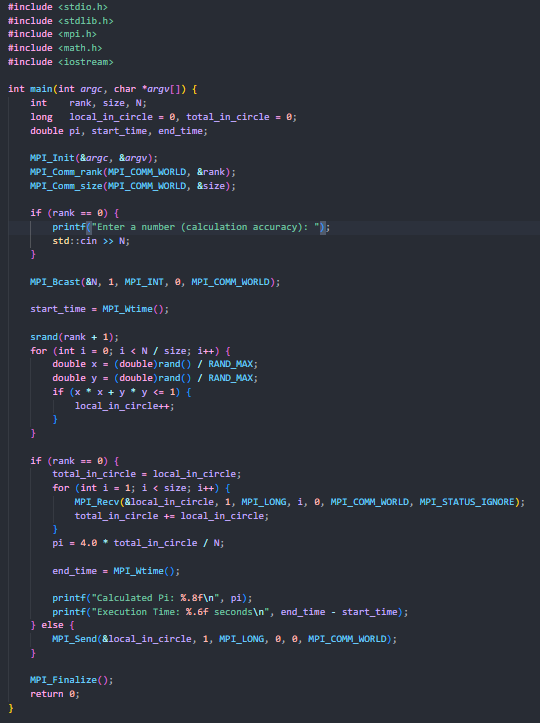
Цель

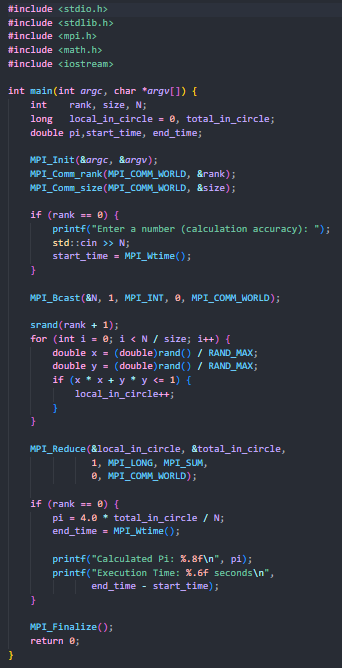
1. Изучите MPI-функцию для выполнения операций редукции над данными, расположенными в адресных пространствах различных процессов, MPI\_Reduce. Реализуйте программу вычисления числа 𝜋 (см. задание 8), используйте функцию MPI\_Reduce для суммирования результатов, вычисленных каждым процессом.
2. Перепишите программу, используя вместо функции MPI\_Reduce функции коммуникации «точка-точка». Сравните эффективность выполнения программ с коллективными и точечными обменами.

Выполнение

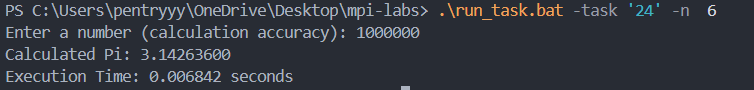
1. **MPI\_Reduce vs Ручной сбор**:
   * MPI\_Reduce выполняет глобальную операцию редукции (например, суммирование) за O(logN) шагов.
   * Ручной сбор через MPI\_Send/MPI\_Recv требует O(N) операций, что менее эффективно.
2. **Распределение нагрузки**:
   * В обоих подходах нагрузка распределяется равномерно: каждый процесс обрабатывает N/size точек.
   * Формула для π≈.
3. **Генерация случайных чисел**:
   * srand(rank + 1) инициализирует генератор уникальным seed для каждого процесса.
   * **Проблема**: rand() не является потокобезопасным, но в MPI каждый процесс работает изолированно.

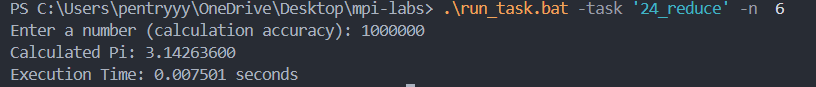
Пример кода для обычной реализации и reduce:





Пример вывода для обычной реализации и reduce:





Задание 25. Коллективные коммуникации: функции распределения и сбора данных

Цель

1. Изучите MPI-функции распределения и сбора блоков данных по процессам MPI\_Scatter и MPI\_Gather. Напишите программу, которая вычисляет произведение двух квадратных матриц 𝐴 × 𝐵 = С размера 𝑛 × 𝑛. Используйте формулу, приведенную в задании 9. Ввод данных и вывод результата должны осуществляться процессом с номером 0. Для распределения матриц A и B и сбора матрицы С используйте функций MPI\_Scatter и MPI\_Gather. 18
2. Перепишите программу, используя вместо функций MPI\_Scatter и MPI\_Gather функции коммуникации «точка-точка». Сравните эффективность выполнения программ с коллективными и точечными обменами.

Выполнение

1. **Функция**matrix\_multiply:
   * **Распределение строк**: Каждый процесс обрабатывает строки матрицы A с шагом size (например, для size=2 процесс 0 вычисляет строки 0, 2, 4..., процесс 1 — 1, 3, 5...).
   * **Локальные вычисления**: Каждая строка C[i] вычисляется как сумма произведений элементов строки A[i] на столбцы B.
   * **Синхронизация**: MPI\_Allreduce объединяет частичные результаты со всех процессов, сохраняя полную матрицу C на каждом процессе.
2. **Основная логика (**main**)**:
   * **Ввод данных**: Процесс 0 считывает размер матрицы n и элементы матриц A и B.
   * **Рассылка данных**: MPI\_Bcast передаёт n, A и B всем процессам.
   * **Вывод результата**: Только процесс 0 выводит матрицу C.

**Распределение вычислений**

* Каждый процесс вычисляет свой набор строк матрицы C. Например, для n=4 и size=2:
  + Процесс 0: строки 0, 2.
  + Процесс 1: строки 1, 3.
* После локальных вычислений MPI\_Allreduce суммирует все частичные результаты, так как каждый процесс пишет в свою часть C. Это необходимо, если следующие этапы программы требуют полной матрицы C на всех процессах.

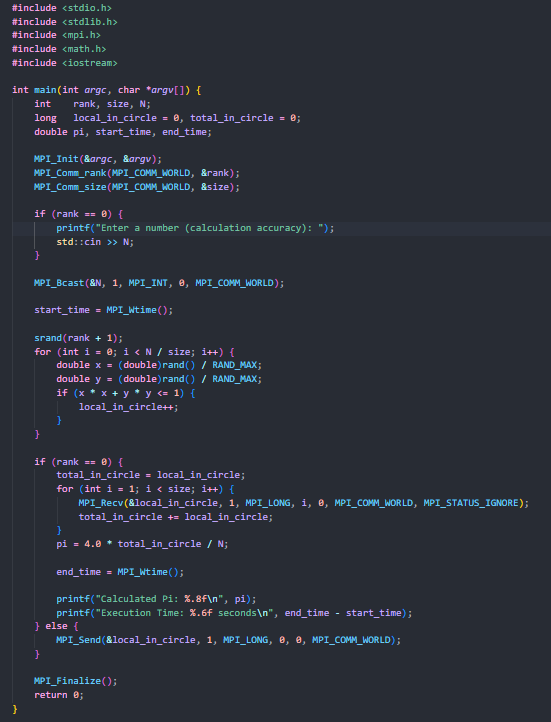
**Особенности**MPI\_Allreduce

* **Параметр**MPI\_IN\_PLACE: Указывает, что входной и выходной буферы совпадают. Данные каждого процесса объединяются (в данном случае — суммируются) и сохраняются в исходном массиве C.
* **Результат**: Все процессы получают полную матрицу C, что удобно для дальнейших параллельных вычислений, но избыточно, если только процесс 0 выводит результат.

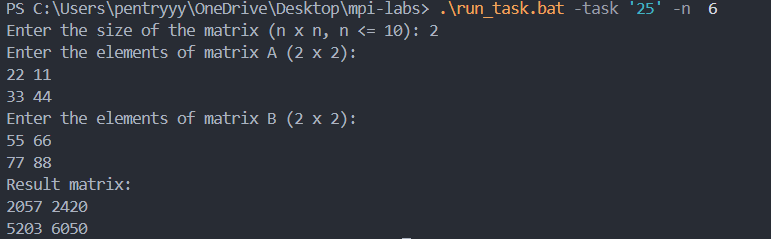
**Ограничения**

* **Максимальный размер матрицы**: Жёстко задан как MAX\_N=10. Для больших матриц нужно динамическое выделение памяти.
* **Эффективность**:
  + Распределение строк может быть неравномерным, если n не кратно size.
  + MPI\_Allreduce передаёт n\*n элементов, что затратно для больших n.

Пример кода:



Пример вывода:



Задание 26. Группы и коммуникаторы

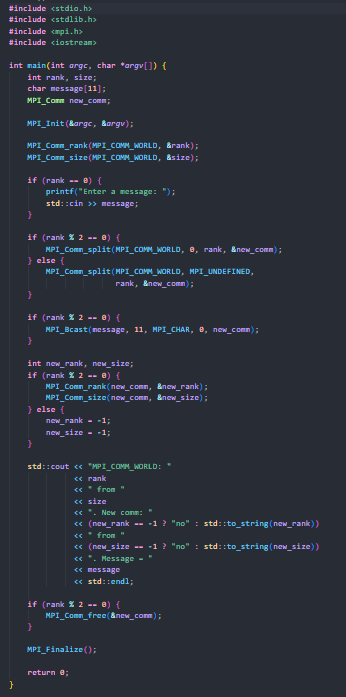
Цель

Напишите программу, в которой производится широковещательная рассылка сообщения с помощью функции MPI\_Bcast, но только по процессам с четным номером. Для рассылки сообщения создайте новый коммуникатор. Каждый процесс приложения должен выводить на экран «MPI\_COMM\_WORLD: from . New comm: from . Message = »

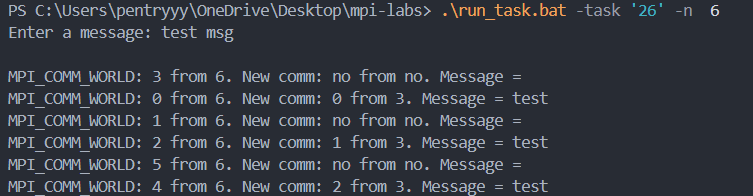
Выполнение

1. **Создание нового коммуникатора** (MPI\_Comm\_split):
   * **Чётные процессы**:
     + color = 0 — попадают в новый коммуникатор.
     + key = rank — порядок процессов в новом коммуникаторе сохраняется.
   * **Нечётные процессы**:
     + color = MPI\_UNDEFINED — исключаются из нового коммуникатора.
     + new\_comm = MPI\_COMM\_NULL — недействительный коммуникатор.
2. **Широковещательная рассылка** (MPI\_Bcast):
   * Процесс 0 (в новом коммуникаторе) рассылает сообщение message всем процессам в new\_comm.
   * **Важно**: Только чётные процессы участвуют в операции.
3. **Получение информации о новом коммуникаторе**:
   * MPI\_Comm\_rank и MPI\_Comm\_size вызываются только для процессов в new\_comm.
   * Для нечётных процессов new\_rank и new\_size остаются -1.

Пример кода:



Пример вывода:



Задание 27. MPI-2: динамическое создание процессов

Цель

Напишите программу, в которой нулевым процессом динамически запускается еще n процессов. Каждый процесс в программе выводит сообще- 19 ние в формате «I am process from processes! My parent is ».

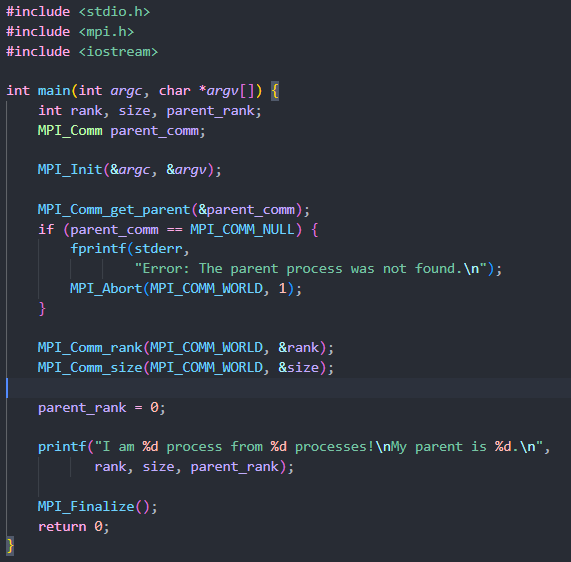
Выполнение

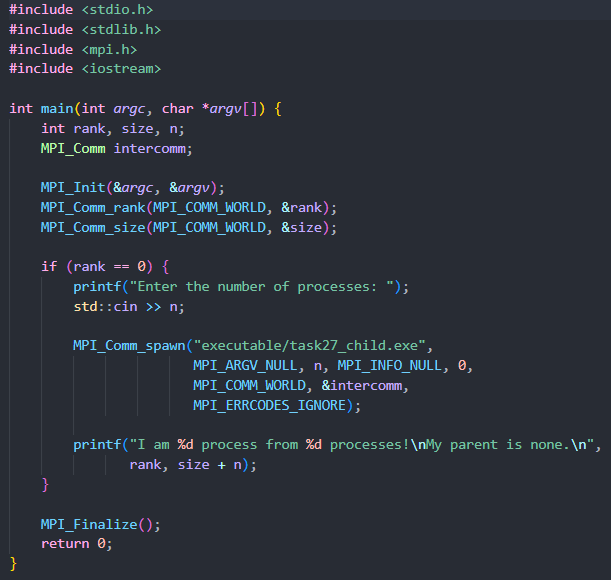
* **Получение родительского коммуникатора**:  
  MPI\_Comm\_get\_parent возвращает коммуникатор, связывающий дочерний процесс с родителем. Если родитель не найден (MPI\_COMM\_NULL), программа завершается с ошибкой.
* **Информация о процессе**:  
  Дочерний процесс получает свой ранг (rank) и общее количество дочерних процессов (size).
* **Динамическое создание процессов**:  
  MPI\_Comm\_spawn запускает n копий программы task27\_child.exe.
  + intercomm — межкоммуникатор для связи с дочерними процессами.

**Взаимодействие родителя и дочерних процессов**

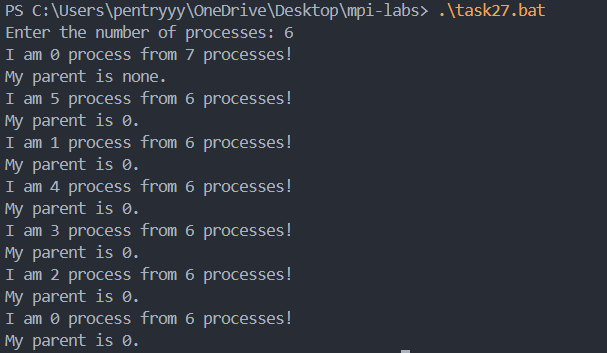
1. **Создание дочерних процессов**:
   * Родитель (ранг 0) вызывает MPI\_Comm\_spawn, создавая n дочерних процессов.
   * Дочерние процессы образуют свой коммуникатор (MPI\_COMM\_WORLD).
2. **Межкоммуникатор**:
   * intercomm позволяет родителю и дочерним процессам обмениваться сообщениями.
   * Родитель имеет ранг 0 в intercomm, дочерние процессы — ранги от 0 до n-1.

Пример кода для child и parent:





Пример вывода:



Задание 28. MPI-2: односторонние коммуникации

Цель

Реализуйте программу вычисления числа 𝜋 (см. задание 24), используйте функции односторонней коммуникации для обмена данными между процессами.

Выполнение

1. **Распределение точек**:
   * Каждый процесс обрабатывает N / size точек.
   * **Проблема**: Если N не кратно size, часть точек не учитывается.
   * **Решение**: Распределить остаток между процессами:

int local\_N = N / size + (rank < N % size ? 1 : 0);

1. **Генерация случайных чисел**:
   * rand() не потокобезопасен и генерирует одинаковые последовательности во всех процессах.
   * **Решение**: Инициализировать генератор уникальным seed:

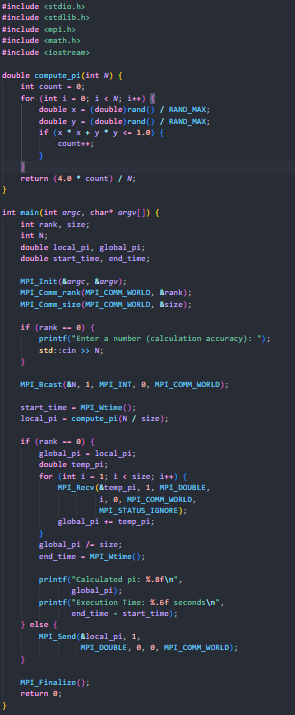
srand(rank + time(NULL));

1. **Сбор результатов**:
   * Используется цикл MPI\_Recv на процессе 0, что неэффективно для большого числа процессов.
   * **Оптимизация**: Заменить на MPI\_Reduce:

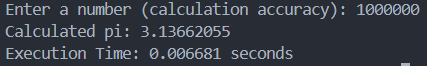
MPI\_Reduce(&local\_pi, &global\_pi, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

global\_pi /= size;

Пример кода:



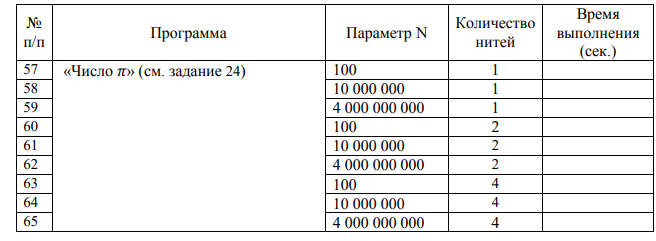
Пример вывода:

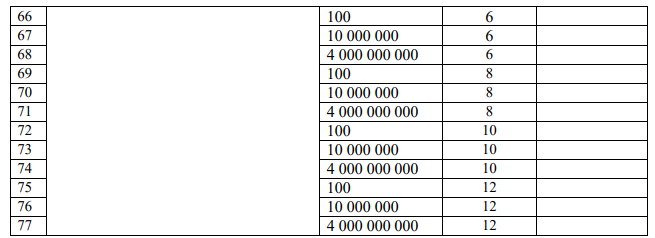


Задание 29. Исследование масштабируемости MPI-программ

Цель

Проведите серию экспериментов на суперкомпьютере по исследованию масштабируемости OpenMP-программ. Заполните следующую таблицу:





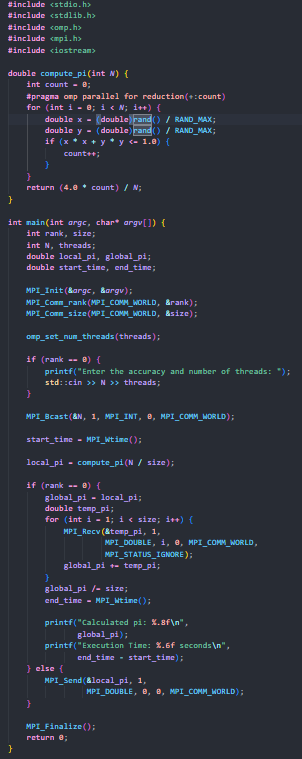
Выполнение

**Этапы работы**:

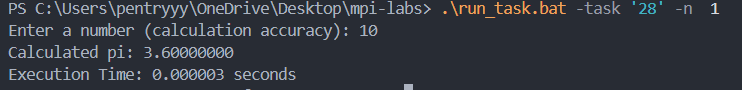
1. **Инициализация MPI**:
   * MPI\_Init инициализирует среду MPI.
   * MPI\_Comm\_rank и MPI\_Comm\_size получают ранг процесса и общее количество процессов.
2. **Настройка OpenMP**:
   * omp\_set\_num\_threads(threads) устанавливает количество потоков для OpenMP.
   * **Ошибка**: Переменная threads инициализируется только на процессе 0, остальные процессы используют мусорное значение.
3. **Ввод данных**:
   * Процесс 0 запрашивает у пользователя:
     + N — общее количество точек.
     + threads — количество потоков OpenMP на процесс.
4. **Рассылка данных**:
   * MPI\_Bcast рассылает N всем процессам.
   * **Проблема**: threads не рассылается, что приводит к некорректной работе OpenMP.
5. **Вычисления**:
   * Каждый процесс вычисляет свою часть π, обрабатывая N / size точек.
   * **Проблема**: Если N не кратно size, часть точек теряется.
   * Функция compute\_pi использует OpenMP для распараллеливания внутри процесса.
6. **Сбор результатов**:
   * Процесс 0 собирает локальные результаты через цикл MPI\_Recv.
   * Результаты суммируются и усредняются:

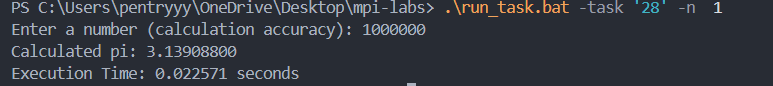
global\_pi = (local\_pi\_0 + local\_pi\_1 + ... + local\_pi\_{size-1}) / size

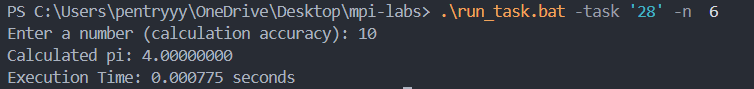
Пример кода:

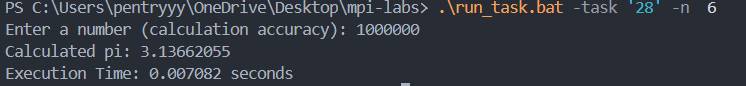


Примеры вывода времени для разных параметров N:









Задание 30. Проект в среде Visual Studio 2010 с поддержкой MPI и OpenMP

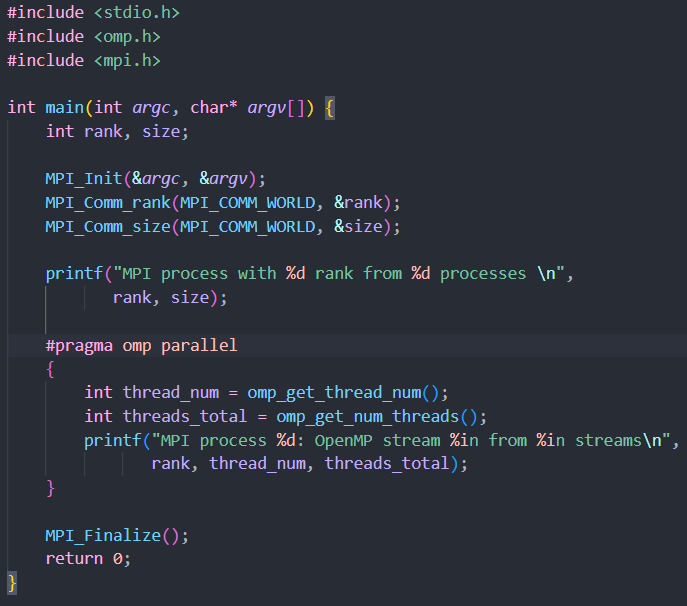
Цель

Создайте проект в среде Visual Studio 2010 с минимальным кодом с поддержкой MPI и OpenMP.

Выполнение

1. **Инициализация MPI**:
   * MPI\_Init инициализирует среду выполнения MPI.
   * MPI\_Comm\_rank возвращает уникальный идентификатор (ранг) текущего процесса.
   * MPI\_Comm\_size возвращает общее количество MPI-процессов.
2. **Параллельная секция OpenMP**:
   * Директива #pragma omp parallel создаёт параллельную область, где каждый MPI-процесс порождает несколько потоков OpenMP.
   * omp\_get\_thread\_num() возвращает номер текущего потока OpenMP.
   * omp\_get\_num\_threads() возвращает общее количество потоков OpenMP в текущей параллельной секции.
3. **Вывод информации**:
   * Каждый MPI-процесс выводит свой ранг и общее количество процессов.
   * Каждый поток OpenMP выводит свой номер и общее количество потоков внутри своего MPI-процесса.

Пример кода:



Пример вывода:



Задание 31. Программа «I am»

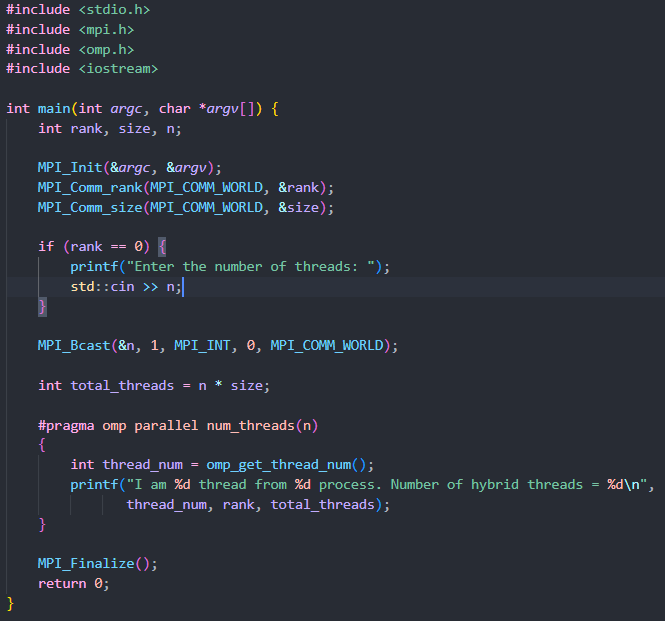
Цель

Напишите программу, в которой в каждом процессе создается n нитей. Каждая нить должна выводить на экран свой номер, номер процессародителя и общее количество нитей во всех процессах в следующем формате: I am thread from process. Number of hybrid threads = .

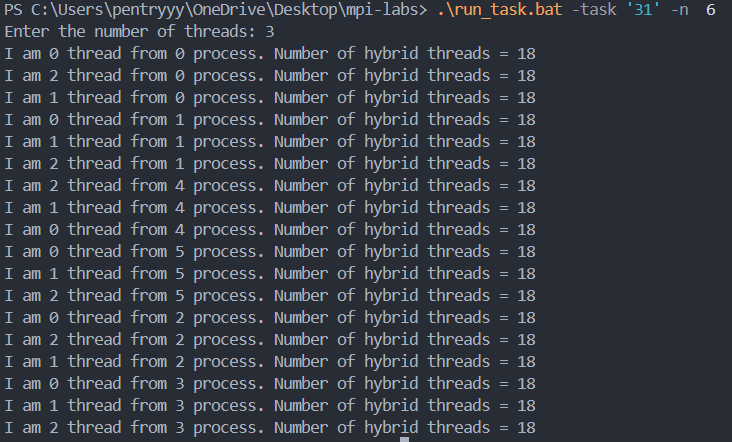
Выполнение

1. **Инициализация MPI**:
   * MPI\_Init инициализирует среду выполнения MPI.
   * MPI\_Comm\_rank возвращает ранг текущего процесса.
   * MPI\_Comm\_size возвращает общее количество MPI-процессов.
2. **Ввод данных**:
   * Только процесс с рангом 0 (rank == 0) запрашивает у пользователя количество потоков OpenMP через std::cin.
   * Значение n рассылается всем процессам с помощью MPI\_Bcast.
3. **Параллельная секция OpenMP**:
   * Директива #pragma omp parallel num\_threads(n) создаёт n потоков OpenMP в каждом MPI-процессе.
   * Каждый поток выводит свой номер (omp\_get\_thread\_num()), ранг MPI-процесса и общее число гибридных потоков (total\_threads = n \* size).

Пример кода:



Пример вывода:



Задание 32. Программа «Число 𝜋»

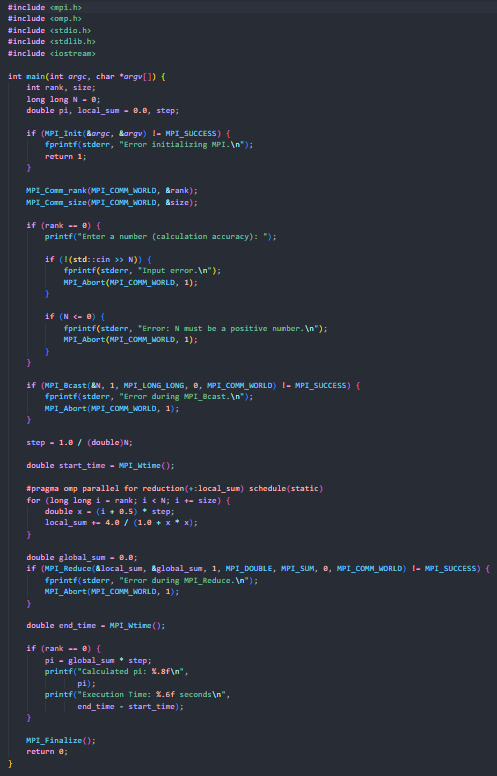
Цель

Реализуйте программу вычисления числа 𝜋 (см. задание 8) с использованием MPI+OpenMP.

Выполнение

1. **Гибридный параллелизм**:
   * MPI распределяет вычисления между узлами кластера.
   * OpenMP ускоряет вычисления внутри каждого узла, используя многоядерность.
2. **Обработка ошибок**:
   * Проверка статусов MPI-функций (MPI\_Init, MPI\_Bcast, MPI\_Reduce).
   * Корректное завершение работы через MPI\_Abort при ошибках.
3. **Оптимизации**:
   * Статическое распределение итераций (schedule(static)) минимизирует накладные расходы.
   * Использование MPI\_Wtime для профилирования.

Пример кода:



Пример вывода:

