**Отчёт: Использование MPI для параллельного построения множества Мандельброта**

**Введение**

MPI (Message Passing Interface) — это стандарт взаимодействия между процессами в параллельных вычислительных системах. Он позволяет эффективно распределять задачи между множеством независимых процессов, которые могут выполняться как на одной машине, так и на кластере. MPI используется в научных и инженерных задачах, где важна масштабируемость и скорость вычислений.

В данной работе демонстрируется использование MPI для параллельного построения множества Мандельброта — классического фрактала, визуализируемого в виде сложного двумерного изображения. Поскольку каждая точка изображения вычисляется независимо, задача идеально подходит для распараллеливания.

**1.1 Классическое решение древа Мандельброта**

Последовательная реализация алгоритма генерации множества Мандельброта состоит из следующих шагов:

1. Устанавливаются размеры изображения (например, 3840×2160) и максимальное количество итераций (max\_iter), определяющее детализацию.
2. Каждая точка на изображении (пиксель) интерпретируется как комплексное число c.
3. Вычисляется последовательность zₙ₊₁ = zₙ² + c, начиная с z₀ = 0, пока модуль z не превысит 2 или не будет достигнут предел итераций.
4. Количество итераций до выхода z за пределы используется как основа для цветовой заливки пикселя.
5. Итоговое изображение сохраняется в файл.

Поскольку каждый пиксель не зависит от других, вычисления производятся в двойном цикле for по координатам x и y. При этом процессор последовательно обходит всю область изображения, загружая лишь одно ядро на 100%.

**1.2 Использование MPI в древе Мандельброта**

Для повышения производительности используется модель параллельных вычислений с помощью MPI. Программа разбивается на несколько процессов, каждый из которых обрабатывает определённую часть изображения.

Распределение нагрузки осуществляется следующим образом:

1. Все процессы инициализируются через mpi::initialize() и получают доступ к объекту world, представляющему коммуникатор всех процессов.
2. Каждый процесс определяет свой rank (уникальный номер) и общее число процессов size.
3. Высота изображения делится на size частей. Каждый процесс получает свой диапазон строк, которые он обрабатывает.
4. После завершения локальных вычислений, данные (массив RGB значений) передаются главному процессу (rank == 0) с помощью gather\_into.
5. Главный процесс объединяет полученные фрагменты и сохраняет итоговое изображение.

Пример кода, где используется MPI:

let universe = mpi::initialize().unwrap();

let world = universe.world();

let rank = world.rank();

let size = world.size();

let rows\_per\_process = height / size as u32;

let start\_row = rank as u32 \* rows\_per\_process;

let end\_row = if rank == size - 1 { height } else { start\_row + rows\_per\_process };

// Вычисления фрагмента изображения...

if rank == 0 {

world.process\_at\_rank(0).gather\_into\_root(&local\_data[..], &mut full\_data);

} else {

world.process\_at\_rank(0).gather\_into(&local\_data[..]);

}

Таким образом, основная работа по генерации выполняется одновременно в нескольких процессах, а главный процесс отвечает за сбор результатов и финальную сборку изображения.

**1.3 Сравнение результатов**

Для демонстрации эффективности MPI-реализации был проведён эксперимент с использованием следующих параметров:

let width = 3840;

let height = 2160;

let max\_iter = 3000;

Измерения проводились с помощью /usr/bin/time -v.

**Сравнительная таблица метрик**

| **Метрика** | **Последовательная версия** | **MPI-версия (4 процесса)** |
| --- | --- | --- |
| Время выполнения (wall clock) | 16.88 сек | 7.55 сек |
| User time | 16.75 сек | 28.71 сек |
| Загрузка CPU | 99% | 382% |
| Максимальное использование RAM | 28.5 MB | 72.0 MB |
| Speedup | 1.0× | 2.23× |
| Efficiency | — | 55.8% |

**Анализ**

MPI позволил ускорить выполнение более чем в два раза при использовании четырёх процессов. Хотя суммарное время CPU (User time) увеличилось, это связано с тем, что несколько процессов выполняются параллельно. Загрузка процессора близка к теоретическому максимуму (4×100%).

Увеличение потребления оперативной памяти объясняется необходимостью хранения промежуточных фрагментов изображения в каждом процессе. Это допустимый компромисс ради выигрыша по времени.

**Выводы**

Использование MPI в задаче генерации множества Мандельброта позволяет добиться значительного прироста производительности за счёт распределения вычислений между процессами. Простота структуры задачи (отсутствие зависимости между пикселями) делает её идеальной для параллельного исполнения.

Полученные результаты показывают, что даже без сложной балансировки нагрузки можно достичь эффективности около 56%, что делает подход практичным для задач визуализации высокой точности. Применение логарифмического масштабирования цвета позволило сохранить яркость и детализацию изображения при высоком разрешении и относительно низком max\_iter.

В дальнейшем возможны расширения: динамическое распределение строк, использование гибридных моделей (MPI + многопоточность), или генерация фрактала в реальном времени.