Решение задач по параллельному программированию

29 апреля 2025 г.

Решение варианта 10

1. Что такое статическая балансировка? В каких случаях её использование целесообразно? Примеры алгоритмов.

Ответ:
Статическая балансировка — это метод распределения нагрузки между процессами или потоками, при котором нагрузка распределяется заранее, до выполнения программы, и не изменяется в процессе работы.

Целесообразность использования:

- Когда нагрузка на систему предсказуема и равномерна.
- В системах с низкими накладными расходами на перераспределение задач.
- Для простых параллельных алгоритмов, где время выполнения задач известно заранее.

Примеры алгоритмов:

- Циклическое распределение (Round-robin).
- Блочное распределение (Block distribution).
- Распределение с использованием заранее заданных весов задач.

2. Что такое DMV-система? Какие её отличительные свойства?

Ответ:
DMV-система (Distributed Memory Virtual machine) — это виртуальная машина, предназначенная для выполнения программ в распределённой памяти.

Отличительные свойства:

- Поддержка распределённой памяти (каждый процессор имеет свою локальную память).
- Использование механизмов передачи сообщений (например, MPI) для обмена данными.
- Масштабируемость за счёт отсутствия общей памяти.
- Высокая производительность для задач с чёткими границами данных.

3. К чему приводит недетерминированность параллельных программ в процессе разработки алгоритма и в процессе отладки.

Ответ:
Недетерминированность параллельных программ проявляется в том, что результат выполнения может зависеть от порядка выполнения потоков или процессов, что усложняет разработку и отладку.

Влияние на разработку:

- Трудно гарантировать корректность алгоритма из-за возможных состояний гонки (race conditions).
- Необходимость использования синхронизации (мьютексы, семафоры), что может снизить производительность.

Влияние на отладку:

- Ошибки могут проявляться нерегулярно, что затрудняет их воспроизведение.
- Требуются специальные инструменты (например, детекторы гонок) для выявления проблем.

4. Основные свойства сортировки слиянием. Положительные и отрицательные свойства, оценка асимптотической сложности.

Ответ:

Основные свойства:

- Алгоритм устойчив (сохраняет порядок равных элементов).
- Использует принцип «разделяй и властвуй».
- Требует дополнительной памяти для слияния.

Положительные свойства:

- Гарантированная сложность $O(n \log n)$ в худшем случае.
- Хорошо работает с большими объемами данных.

Отрицательные свойства:

- Требует дополнительной памяти $O(n)$.
- Может быть менее эффективен для небольших массивов по сравнению с сортировкой вставками.

Асимптотическая сложность:

- Лучший случай: $O(n \log n)$.
- Средний случай: $O(n \log n)$.
- Худший случай: $O(n \log n)$.

5. Требования к генераторам псевдослучайных чисел для многопроцессорных систем.

Ответ:
Требования:

- Независимость последовательностей для разных процессов (отсутствие корреляции).
- Высокая скорость генерации.
- Повторяемость (для воспроизводимости результатов).
- Длинный период повторения последовательности.
- Равномерное распределение генерируемых чисел.

6. Процесс создания и запуска программы на C с использованием MPI. Определение числа исполнителей. Изменение количества процессов.

Ответ:

Процесс создания и запуска:

1. Написание программы на C с использованием функций MPI (например, `MPI_Init`, `MPI_Send`, `MPI_Recv`).
2. Компиляция программы с помощью MPI-компилятора (например, `mpicc`).
3. Запуск программы через MPI-рантайм (например, `mpiexec -n 4 ./program`).

Определение числа исполнителей:

- Количество процессов задаётся при запуске (с использованием аргумента `-n` в `mpiexec`).
- Внутри программы можно узнать число процессов с помощью функции `MPI_Comm_size(MPI_COMM_W &size)`.

Изменение количества процессов:

- Стандартный MPI (MPI-1, MPI-2) не поддерживает динамическое изменение числа процессов во время выполнения.
- В MPI-2 существует ограниченная поддержка динамического управления процессами (например, с использованием функции `MPI_Comm_spawn`), однако это используется редко на практике.

Решение варианта 30

1. Закон Амдала. Пример. Максимальное ускорение при 10% последовательных вычислений.

Ответ:
Закон Амдала описывает максимальное теоретическое ускорение программы при увеличении количества вычислительных ресурсов:

$$S = \frac{1}{(1-P) + \frac{P}{N}},$$

где:

- S — ускорение;
- P — доля параллелизуемой части программы;
- N — число процессоров.

Пример:
Если $P = 0{,}7$, $1 - P = 0{,}3$, $N = 4$, то:

$$S = \frac{1}{0{,}3 + \frac{0{,}7}{4}} \approx 2{,}1.$$

Максимальное ускорение при 10% последовательных вычислений:
Если $1 - P = 0{,}1$, при $N \to \infty$:

$$S_{\max} = \frac{1}{0{,}1} = 10.$$

2. Методы динамической балансировки. Описание двух из них.

Методы с динамической балансировкой:

- Метод диффузной балансировки.
- Метод конвейерного параллелизма.

Диффузная балансировка:

- Каждый процессор проверяет загрузку соседей.
- При дисбалансе задачи передаются менее загруженным.
- Пример — алгоритмы в сетях с топологией «решётка».

Конвейерный параллелизм:

- Задачи разбиваются на этапы.
- Этапы последовательно выполняются разными процессорами.
- Пример — обработка изображений (фильтрация, сжатие и т.д.).

3. Иерархическая декомпозиция графов на p доменов.

Этапы:

1. **Предварительная обработка:** граф разбивается на крупные подграфы.
2. **Рекурсивное разбиение:** каждый подграф делится на более мелкие части до получения p доменов.
3. **Балансировка нагрузки:** проверяется равномерность, выполняется перераспределение вершин.
4. **Финальная оптимизация:** минимизация числа междоменных рёбер (например, методом Кернигана–Лина).

Пример: использование библиотеки METIS для декомпозиции графа социальной сети.

4. Параллельная сортировка при нехватке памяти.

Этапы:

1. **Распределение данных:** массив делится между процессорами.
2. **Локальная сортировка:** каждый процессор сортирует свой блок (например, сортировкой слиянием).
3. **Глобальное слияние:**
   - Выбор разделителей из отсортированных блоков.
   - Перераспределение данных по разделителям.
   - Финальное слияние внутри процессоров.

Алгоритмы:

– `External Merge Sort` — при хранении данных на диске.

– `Parallel Quick Sort` с динамической балансировкой.

5. Период генератора $u_{i+1} = (45 \cdot u_i + 17) \bmod 512$.

Ответ:
Период равен модулю (512), если выполнены:

– $\gcd(45, 512) = 1$;

– $\gcd(17, 512) = 1$;

– $45 - 1 = 44$ делится на все простые делители 512 (но $512 = 2^9$, а $44 \not\equiv 0 \mod 2^2$).

Вывод: условия не выполняются, период меньше 512. Для точного значения требуется анализ последовательности.

6. Передача данных точка-точка в MPI. Пример со структурами.

Тип: точка-точка — обмен между двумя процессами.

Блокирующие функции:

– `MPI_Send` — отправка:

  – аргументы: `buf`, `count`, `datatype`, `dest`, `tag`, `comm`.

– `MPI_Recv` — приём:

  – аргументы: `buf`, `count`, `datatype`, `source`, `tag`, `comm`, `status`.

Пример передачи структуры на C:

```
typedef struct {
    int a;
    double b;
} MyStruct;

MPI_Datatype mytype;
int blocks[2] = {1, 1};
MPI_Aint offsets[2];
```



```
offsets[0] = offsetof(MyStruct, a);
offsets[1] = offsetof(MyStruct, b);
MPI_Datatype types[2] = {MPI_INT, MPI_DOUBLE};

MPI_Type_create_struct(2, blocks, offsets, types, &mytype);
MPI_Type_commit(&mytype);

//                              :
MPI_Send(&data, 1, mytype, dest, tag, MPI_COMM_WORLD);
MPI_Recv(&data, 1, mytype, source, tag, MPI_COMM_WORLD, &status);
```

Листинг 1: Передача пользовательской структуры с помощью MPI

Решение варианта 45

1. Ускорение и эффективность параллельного алгоритма

Ускорение (Speedup, S) определяется как отношение времени выполнения последовательного алгоритма T_1 к времени выполнения параллельного алгоритма T_p на p процессорах:

$$S = \frac{T_1}{T_p}$$

Диапазон значений: $1 \leq S \leq p$ (идеальное ускорение — $S = p$).

Эффективность (Efficiency, E) — это отношение ускорения к количеству процессоров:

$$E = \frac{S}{p}$$

Диапазон: $0 < E \leq 1$ в большинстве случаев.

Суперлинейное ускорение ($E > 1$) возможно, если:

- параллельный алгоритм эффективнее использует кэш-память;
- в последовательной версии имеются дополнительные накладные расходы.

2. Балансировка нагрузки для слабосвязанных задач с непредсказуемым временем выполнения

Рекомендуется использовать **динамическую балансировку нагрузки**, например:

- **Work stealing (кража задач)** — когда один процессор, оставшись без работы, "крадёт"задачи у других.
- **Диффузная балансировка** — перераспределение нагрузки между соседями в реальном времени.

Обоснование: Предсказать время выполнения невозможно, а слабая связанность позволяет эффективно перераспределять задачи.

3. Сложность разбиения планарного графа методом рекурсивной геометрической бисекции

Для графа с n вершинами и ограниченной степенью:

$$\text{Число шагов} = O(\log p)$$

$$\text{На каждом шаге:} \quad O(n) + O(n) = O(n)$$

$$\Rightarrow \text{Итоговая сложность} = O(n \log p)$$

Пример: для $p = 8$, $n = 10^6$: $10^6 \cdot \log_2(8) = 3 \cdot 10^6$ операций.

4. Число тактов сети чётно-нечётной перестановки (Odd-Even Transposition Sort)

Для p элементов ($p = 2^k$), сортировка выполняется за p тактов.

Пример: $p = 8 \Rightarrow$ требуется 8 тактов.

5. Пошаговая сортировка массива с помощью алгоритма Кеннета Бэтчера (Bitonic Sort)

Исходный массив: [5, 2, 12, 24, 3, 7, 23, 9]

1. Попарная сортировка: [2,5], [12,24], [3,7], [9,23]
2. Объединение в битонические последовательности и сортировка:

$$[2, 5, 24, 12] \rightarrow [2, 5, 12, 24], \quad [3, 7, 23, 9] \rightarrow [3, 7, 9, 23]$$

3. Финальная битоническая последовательность:

$$[2, 5, 12, 24, 23, 9, 7, 3] \rightarrow [2, 3, 5, 7, 9, 12, 23, 24]$$

Результат: [2, 3, 5, 7, 9, 12, 23, 24]

6. Групповые операции в MPI и пользовательские операции

Стандартные операции:

- `MPI_Allgather` — сбор данных от всех и распространение результата.
- `MPI_Reduce` — редукция (суммирование, максимум и др.) с сохранением результата на одном процессе.
- `MPI_Allreduce` — как Reduce, но результат доступен всем.

Пользовательские операции: Создаются с помощью `MPI_Op_create`. Операция должна быть ассоциативной, желательно коммутативной.

Пример кода:

```
MPI_Op my_op;

void my_function(void* invec, void* inoutvec, int* len, MPI_Datatype*
    datatype) {
     //
}

MPI_Op_create(my_function, 1, &my_op);  // 1

MPI_Allreduce(data, result, count, my_type, my_op, MPI_COMM_WORLD);
```

Решение задачи 7

Условие задачи:

Программа использует функцию `MPI_Gather` для сбора данных от 4 процессов. Необходимо определить, что будет выведено на процессе с рангом 0 после выполнения операции сбора.

Анализ:

- Каждый процесс инициализирует массив `sbuf[2]`:

`sbuf[0] = rank,  sbuf[1] = rank + 6`

- Процесс 0 дополнительно инициализирует массив `rbuf[8]` для приема данных.
- Функция `MPI_Gather` собирает по 2 элемента `MPI_INT` от каждого процесса и записывает их в `rbuf` на процессе 0.
- Заполнение происходит в порядке возрастания рангов: сначала данные от процесса 0, затем от 1, и так далее.

Данные каждого процесса:

Процесс 0: `sbuf = [0, 6]`
Процесс 1: `sbuf = [1, 7]`
Процесс 2: `sbuf = [2, 8]`
Процесс 3: `sbuf = [3, 9]`

После вызова `MPI_Gather` массив `rbuf` на процессе 0 будет содержать:

`rbuf = [0, 6, 1, 7, 2, 8, 3, 9]`

Вывод на процессе 0:

0 6 1 7 2 8 3 9

Заключение:

При запуске программы на 4 процессах, после выполнения `MPI_Gather`, процесс с рангом 0 выведет:

0 6 1 7 2 8 3 9

1.Вариант 51 Геометрический параллелизм

Определение: геометрический параллелизм — это метод параллельной обработки данных, при котором вычислительная область разбивается на подобласти (блоки), распределяемые между процессорами. Каждый процессор работает со своей частью данных, а взаимодействие происходит только на границах подобластей.

Вид балансировки: статическая, так как разбиение данных фиксируется до начала вычислений.

Рекомендуемые задачи:

- Задачи с регулярной структурой данных (например, сетки, матрицы);
- Численное моделирование (например, решение уравнений в частных производных).

Пример: моделирование теплопередачи в стержне:

- Стержень разбивается на отрезки, каждый процесс вычисляет температуру на своём участке;
- На границах отрезков осуществляется обмен данными о температуре.

2. Метод коллективного решения

Определение: метод коллективного решения — это подход, при котором все процессы участвуют в выполнении общей операции (например, обмене данными или синхронизации).

Основные свойства:

- Все процессы в коммуникаторе участвуют в операции;

- Операции бывают синхронными (например, `MPI_Barrier`) и асинхронными (например, `MPI_Allreduce`).

Рекомендуемые случаи использования:

- Сбор или рассылка данных (например, `MPI_Bcast`);
- Синхронизация процессов (`MPI_Barrier`);
- Выполнение редукционных операций (например, сумма, максимум).

Пример: использование `MPI_Allgather` для сбора данных от всех процессов и рассылки результатов обратно всем.

3. Алгоритм спектральной бисекции графа

Применение: используется для разделения графа на две части с минимальным числом пересечений между ними.

Основные случаи применения:

- Декомпозиция графов для распределённых вычислений;
- Балансировка нагрузки в параллельных системах;
- Разбиение сеток в задачах вычислительной гидродинамики.

Условия эффективности:

- Разреженность графа;
- Желательно наличие геометрической структуры.

4. Число тактов при выполнении барьера

Если количество процессов $p = 2^k$, то минимальное число тактов при реализации барьера на основе синхронных операций передачи данных:

$$\log_2 p$$

Обоснование:

- Алгоритм использует двоичное дерево, на каждом уровне процессы обмениваются данными попарно;
- Общее число уровней — $\log_2 p$.

Пример: при $p = 8$:

$$\log_2 8 = 3 \text{ такта}$$

5. Сортировка слиянием массива [17, 6, 8, 21, 5, 10, 14, 0]

Шаги сортировки:

- Разбиваем массив:
  $[17, 6, 8, 21, 5, 10, 14, 0] \rightarrow$
  Левый: $[17, 6, 8, 21]$, Правый: $[5, 10, 14, 0]$;
- Далее: $[17, 6]$, $[8, 21]$, $[5, 10]$, $[14, 0]$;
- Ещё: $[17], [6], [8], [21], [5], [10], [14], [0]$.

Слияние:

- $[17] + [6] \rightarrow [6, 17]$;
- $[8] + [21] \rightarrow [8, 21]$;
- $[5] + [10] \rightarrow [5, 10]$;
- $[14] + [0] \rightarrow [0, 14]$;
- $[6, 17] + [8, 21] \rightarrow [6, 8, 17, 21]$;
- $[5, 10] + [0, 14] \rightarrow [0, 5, 10, 14]$;
- Финальное слияние:
  $[6, 8, 17, 21] + [0, 5, 10, 14] \rightarrow [0, 5, 6, 8, 10, 14, 17, 21]$.

Итог: $[0, 5, 6, 8, 10, 14, 17, 21]$

6. Поведение функции `MPI_Send`

Может ли `MPI_Send` завершиться до начала приёма? Да, возможно.

Условия:

- При использовании `MPI_Bsend` (буферизованная передача): данные копируются во внутренний буфер;
- При использовании `MPI_Send` (стандартная передача): если буфер доступен, завершение происходит немедленно;
- В случае `MPI_Ssend` (синхронная передача): завершение только после начала приёма.

Действия с буфером:

- При `MPI_Bsend`: исходный буфер можно переиспользовать сразу;
- При `MPI_Send` без буферизации: изменение буфера до завершения передачи может привести к неопределённому поведению.

Рекомендация: использовать `MPI_Bsend` или `MPI_Isend` для безопасного параллелизма и немедленного освобождения буфера.

Решение задачи 7

Условие задачи:

Программа на языке C использует MPI и функцию `sleep()`. Необходимо определить время выполнения программы на 5 исполнителях (ранги 0–4), учитывая только время, затраченное на выполнение `sleep()`.

Анализ программы:

1. **Инициализация MPI:**
   - Каждый процесс получает свой ранг (`rank`).
   - Всего 5 процессов (`size = 5`).
2. **Логика работы:**
   - **Процесс с рангом 0 (`rank == 0`):**
     - Получает сообщения от процессов с рангами 1, 2, 3, 4.
     - Для каждого полученного сообщения вызывает `sleep(buf + 4)`, где `buf` — значение ранга отправителя.
   - **Процессы с рангами 1, 2, 3, 4:**
     - Отправляют свой ранг процессу 0 (`MPI_Send`).
     - Ожидают завершения операции `MPI_Barrier`.
3. **Выполнение `sleep()`:**
   - Процесс 0 получает значения `buf` от процессов 1, 2, 3, 4 (значения 1, 2, 3, 4 соответственно).
   - Для каждого значения `buf` вызывается `sleep(buf + 4)`:
     - `sleep(5)` (для `buf = 1`),
     - `sleep(6)` (для `buf = 2`),
     - `sleep(7)` (для `buf = 3`),
     - `sleep(8)` (для `buf = 4`).
4. **Последовательность выполнения:**
   - Процесс 0 выполняет `sleep()` последовательно для каждого полученного сообщения:
     - `sleep(5)` $\rightarrow$ 5 секунд,
     - `sleep(6)` $\rightarrow$ 6 секунд,
     - `sleep(7)` $\rightarrow$ 7 секунд,
     - `sleep(8)` $\rightarrow$ 8 секунд.
   - Общее время `sleep()` на процессе 0: $5 + 6 + 7 + 8 = 26$ секунд.
   - Остальные процессы (`rank = 1-4`) завершают работу после отправки сообщения и ожидания `MPI_Barrier`.
5. **Барьерная синхронизация (`MPI_Barrier`):**
   - Все процессы ожидают завершения операции `MPI_Barrier`.
   - Последним завершит работу процесс 0 после выполнения всех `sleep()` (26 секунд).
   - Остальные процессы будут заблокированы на `MPI_Barrier` до этого момента.

Итоговое время выполнения:

Программа завершит работу через 26 секунд, так как это время определяется процессом 0, который выполняет все `sleep()` последовательно.

Ответ:

Время выполнения программы на 5 исполнителях составит $\boxed{26}$ секунд.

1 Вариант 57

1.1 Области применения:

- Моделирование климата — прогнозирование изменений климата с учетом множества факторов
- Квантовая химия — расчет молекулярных структур и взаимодействий
- Астрофизика — симуляция галактик и черных дыр
- Генетика — анализ ДНК и белковых структур
- Машинное обучение — обучение сложных нейросетевых моделей

1.2 Наиболее требовательные задачи:

- Задачи с высокой степенью параллелизма и большими объемами данных:
  - Моделирование ядерных реакций
  - Гидродинамические расчеты (например, аэродинамика самолетов)
  - Обработка данных с Большого адронного коллайдера

1.3 "Задачи большого вызова"(Grand Challenge Problems):

- Сложные научные проблемы, требующие экзафлопсных вычислений
- Примеры:
  - Моделирование человеческого мозга
  - Создание искусственного интеллекта общего назначения

2 Эффективность параллельного алгоритма: теоретические и практические аспекты

2.1 Теоретические значения эффективности:

- Диапазон: $0 < E \leq 1$
- Формула: $E = \frac{S}{p}$, где S — ускорение, p — число процессоров
- Идеальная эффективность $E = 1$ достигается при линейном ускорении ($S = p$)

2.2 Влияние архитектурных особенностей:

- Кэш-память: Лучшее использование кэша может привести к сверхлинейному ускорению ($E > 1$)
- Накладные расходы: Задержки передачи данных снижают эффективность

2.3 Примеры сверхлинейного ускорения:

- Алгоритмы поиска в больших базах данных
- Некоторые рекурсивные алгоритмы (например, быстрая сортировка)

3 Критерии декомпозиции расчетных сеток для систем с общей памятью

3.1 Критерии:

- Балансировка нагрузки
- Минимизация обмена данными
- Локализация данных
- Масштабируемость

3.2 Пример:

- Декомпозиция сетки для решения уравнения теплопроводности
- Разбиение на блоки с минимальным обменом данными на границах

4 Оценка числа операций для сортировки методом четно-нечетного слияния Бэтчера

4.1 Условия:

- Массив из n элементов
- p процессоров ($n \gg p$)

4.2 Оценка сложности:

- Число этапов: $\log_2 p$
- Операций на этап: $O\left(\frac{n}{p}\right)$ (слияние подмассивов)
- Общая сложность: $O\left(\frac{n}{p} \log_2 p\right)$

4.3 Пример:

Для $n = 10^6$, $p = 8$:

$$O\left(\frac{10^6}{8} \times 3\right) = O(375\,000)$$

5 Рекомендации для линейных конгруэнтных генераторов (ЛКГ)

5.1 Применимость:

- Не криптографические задачи:
  - Моделирование (метод Монте-Карло)
  - Генерация тестовых данных
- Условия:
  - Высокая скорость генерации
  - Не требуется строгая случайность

5.2 Ограничения:

- Не подходят для криптографии
- Могут иметь короткий период

6 Система очередей на суперкомпьютерах

6.1 Назначение:

- Управление задачами множества пользователей
- Оптимизация загрузки ресурсов

6.2 Примеры использования:

- Пакетная обработка задач (SLURM, PBS)
- Планирование долгих вычислений

6.3 Интерактивные программы:

- Неудобны в системе очередей
- Альтернатива: выделение интерактивных сессий (например, `salloc` в SLURM)

7 Вариант 79.Динамическая балансировка: применение и примеры алгоритмов

Определение:
Динамическая балансировка — это метод распределения вычислительной нагрузки между процессами/потоками **в реальном времени**, в зависимости от текущего состояния системы.

Когда рекомендована:

- При **неравномерной** или **непредсказуемой** нагрузке (например, задачи с ветвлением)
- В системах с **разнородными** вычислительными ресурсами

- Для задач, где время выполнения частей **сильно варьируется**

Примеры алгоритмов:

- **Work stealing:**
  - Процессы с пустыми очередями "крадут"задачи у перегруженных соседей
  - *Пример:* Cilk, TBB
- **Диффузная балансировка:**
  - Нагрузка перераспределяется между соседними узлами по графу топологии
  - *Пример:* Алгоритмы для сеток с регулярной структурой
- **Динамическое планирование (chunking):**
  - Задачи разбиваются на "порции которые динамически назначаются процессорам

8 Ограничение ускорения в методе конвейерного параллелизма для стены Фокса

Ускорение ограничено:

- Длиной конвейера (числом этапов)
- Временем самого медленного этапа (по закону Амдала)

Для стены Фокса (матричное умножение):

- Максимальное ускорение: $S_{\max} = \frac{T_{\text{посл}}}{T_{\text{паралл}}}$, где $T_{\text{паралл}}$ включает накладные расходы на передачу данных между этапами
- При большом числе процессоров ускорение стремится к $\frac{N}{\text{latency}}$, где N — размер матрицы, а latency — задержка передачи

Пример:
Для матрицы 1000×1000 и 100 этапов: $S_{\max} \approx 50 - 100$.

9 Адаптивный алгоритм параллельного интегрирования

Алгоритм:

1. **Рекурсивное разбиение** интервала интегрирования на подынтервалы
2. **Оценка погрешности** для каждого подынтервала (например, методом Гаусса)
3. **Балансировка нагрузки:**
   - Подынтервалы с большой погрешностью делятся дальше
   - Распределение подынтервалов между процессорами

Преимущества:

- Автоматическая адаптация к сложности подынтегральной функции
- Высокая точность за счет локального уточнения

Недостатки:

- Накладные расходы на рекурсивное разбиение
- Сложность балансировки для "негладких"функций

Пример:
Интегрирование $\int_0^1 \sin(x^2)dx$ с точностью 10^{-6}.

10 Параллельная сортировка Бэтчера (Bitonic Sort)

Принцип действия:

1. Построение **битонической последовательности** (возрастание + убывание)
2. Рекурсивное **сравнение и обмен** элементов на расстоянии 2^k

Свойства:

- **Сложность:** $O(\log^2 n)$ для n элементов на n процессорах
- **Достоинства:**
  - Хорошо распараллеливается
  - Стабильная работа для степеней двойки
- **Недостатки:**
  - Неэффективен для небольших массивов
  - Требует $O(n \log n)$ сравнений

Пример:
Сортировка массива [8, 14, 2, 3, 7, 18, 16, 15] требует 6 этапов сравнений.

11 Пирамидальная сортировка (Heapsort) для массива [8, 14, 2, 3, 7, 18, 16, 15]

Шаги:

1. **Построение кучи (max-heap):**
   - Исходная куча: [18, 15, 16, 14, 7, 2, 8, 3]
2. **Извлечение максимума и перестроение кучи:**
   - $18 \rightarrow [16, 15, 8, 14, 7, 2, 3] \rightarrow 16 \rightarrow [15, 14, 8, 3, 7, 2] \rightarrow ...$
3. **Результат:** |2, 3, 7, 8, 14, 15, 16, 18|

Визуализация:

```
Исходный массив: [8, 14, 2, 3, 7, 18, 16, 15]
Построение кучи:
      18
     /  \
    15   16
   / \   / \
  14 7  2  8
 /
3
```

12 Измерение времени в MPI

Способы:

- **MPI_Wtime():**
  - Возвращает время в **секундах** (double) с высокой точностью (до наносекунд)

```
double start = MPI_Wtime();
//
double end = MPI_Wtime();
printf("Time: %f sec\n", end - start);
```

- **MPI_Barrier + MPI_Reduce:**
  - Для синхронизации измерений на всех процессах

Точность:
Зависит от ОС и аппаратуры, обычно микросекунды.
Пример:
Измерение времени пересылки данных:

```
double t1 = MPI_Wtime();
MPI_Send(buf, count, MPI_INT, dest, tag, comm);
double t2 = MPI_Wtime();
printf("Send time: %e sec\n", t2 - t1);
```

Условие задачи:

Программа на языке C использует MPI-функцию `MPI_Reduce` с операцией `MPI_BXOR` (побитовое исключающее ИЛИ). Необходимо определить, что будет выведено при выполнении программы на 5 исполнителях (ранги 0-4).

Анализ программы:

1. Инициализация MPI:

- Каждый процесс получает свой ранг (`rank` от 0 до 4)
- Всего 5 процессов (`size = 5`)

2. Работа с буферами:

- Каждый процесс инициализирует массив `sbuf[2]`:
  - `sbuf[0] = rank` (значения: 0, 1, 2, 3, 4)
  - `sbuf[1] = rank + 512` (значения: 512, 513, 514, 515, 516)
- На процессе 0 выделен массив `rbuf[2]` для результата

3. Операция `MPI_Reduce`:

- Операция `MPI_BXOR` применяется к элементам `sbuf` всех процессов
- Результат сохраняется в `rbuf` на процессе 0

Вычисление результата:

1. Для `rbuf[0]` (операция XOR над `rank`):

$$0 \oplus 1 = 1$$
$$1 \oplus 2 = 3$$
$$3 \oplus 3 = 0$$
$$0 \oplus 4 = \boxed{4}$$

2. Для `rbuf[1]` (операция XOR над `rank + 512`):

$$512 \oplus 513 = 1$$
$$1 \oplus 514 = 515$$
$$515 \oplus 515 = 0$$
$$0 \oplus 516 = \boxed{516}$$

Вывод на процессе 0:

Программа выведет значения `rbuf[0]` и `rbuf[1]`:

```
4 516
```

Проверка корректности:

- Побитовое XOR ассоциативно и коммутативно
- Прямое вычисление:
  - `rbuf[0]` $= 0 \oplus 1 \oplus 2 \oplus 3 \oplus 4 = 4$
  - `rbuf[1]` $= 512 \oplus 513 \oplus 514 \oplus 515 \oplus 516 = 516$

Итог:

При выполнении программы на 5 исполнителях процесс с рангом 0 выведет:

$$\boxed{4\ 516}$$