РАЗДЕЛ 1.
ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1. Быстрая сортировка. Последовательный алгоритм.

Алгоритм быстрой сортировки относится к числу эффективных методов упорядочивания данных и широко используется в практических приложениях.

Общая идея алгоритма состоит в следующем:

- Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива или же число, вычисленное на основе значений элементов; от выбора этого числа сильно зависит эффективность алгоритма.
- Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: «меньшие опорного», «равные» и «большие».
- Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

На практике массив обычно делят не на три, а на две части: например, «меньшие опорного» и «равные и большие»; такой подход в общем случае эффективнее, так как упрощает алгоритм разделения. [1]

Проиллюстрируем на примере описанный алгоритм. Пусть дана исходная последовательность чисел вида:

55	88	22	99	44	11	66	77	33

Решение задачи сортировки данных при помощи алгоритма быстрой сортировки представлено на рисунке 1.1

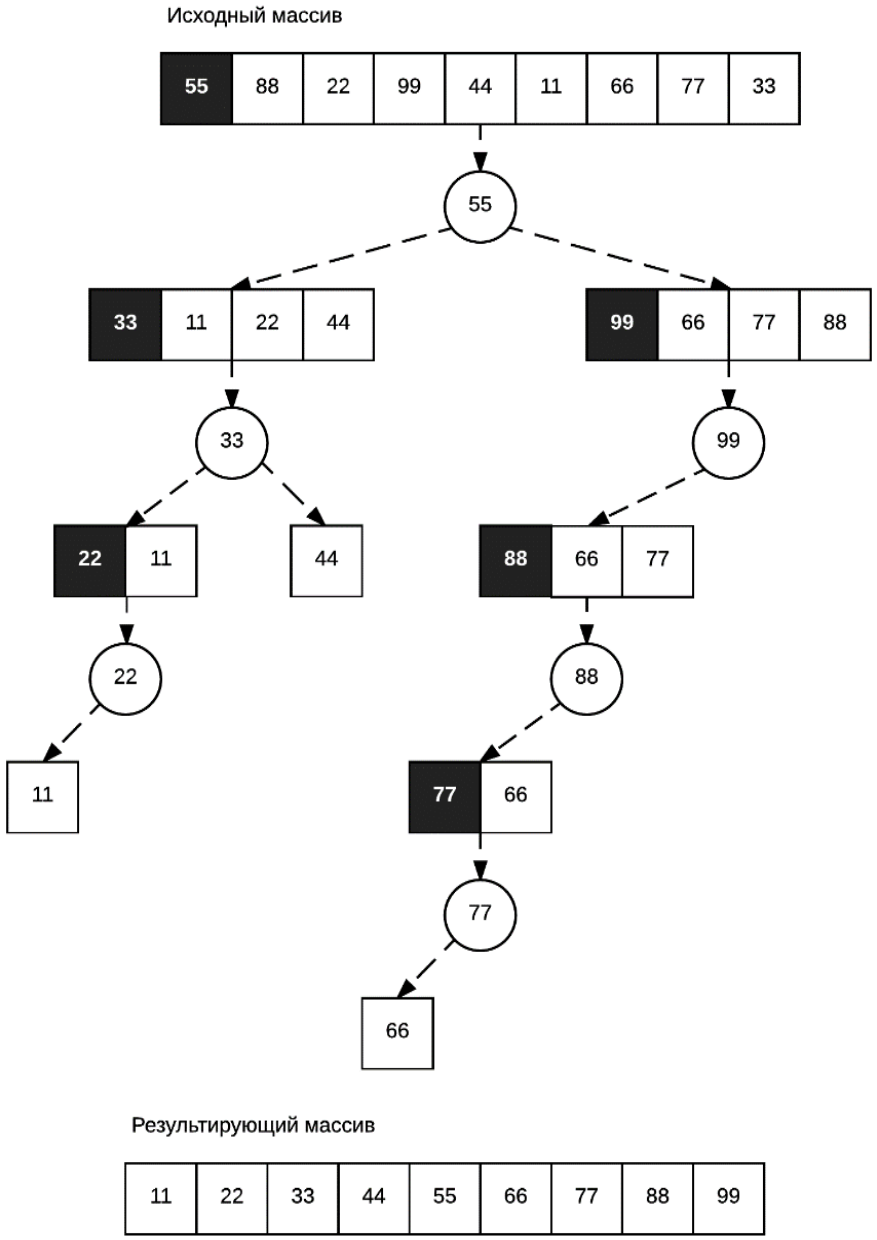

Рисунок 1.1. – Иллюстрация алгоритма быстрой сортировки для девяти чисел

Дадим пояснения к рисунку 1.1. В закрашенных квадратах отмечены числа, которые выбираются в качестве опорных элементов для исходной части массива на каждом из рекурсивных этапов алгоритма. Выбранные опорные элементы выносятся в промежуточные этапы на графе. Также видно, что после выбора опорного элемента происходит разделение

исходного массива данных на две части: элементы больше опорного и элементы меньше опорного. После разделения алгоритм рекурсивно повторяется для каждой из получившихся частей. Рекурсия останавливается на этапе, когда в подмассиве данных остается один элемент.

Приведем псевдокод, реализующий изложенный метод:

```
Quicksort(A as array, low as int, high as int){
    if (low < high){
        pivot_location = Partition(A,low,high)
        Quicksort(A,low, pivot_location)
        Quicksort(A, pivot_location + 1, high)
    }
}

Partition(A as array, low as int, high as int){
     pivot = A[low]
     leftwall = low

     for i = low + 1 to high{
         if (A[i] < pivot) then {
             swap(A[i], A[leftwall])
             leftwall = leftwall + 1
         }
     }
     swap(pivot,A[leftwall])

    return (leftwall)
}
```

Оценка сложности алгоритма

Очевидно, что операция разделения массива на две части относительно опорного элемента занимает время $O(n)$. Поскольку все операциии разделения, проделываемые на одной глубине рекурсии, обрабатывают разные части исходного массива, суммарно на каждом уровне рекурсии потребуется также $O(n)$ операций. Отсюда следует, что общая сложность алгоритма определяется лишь количеством разделений, то есть глубиной рекурсии. Глубина рекурсии в свою очередь зависит от сочетания входных данных и способа выбора опорного элемента.

В лучшем случае при каждой операции разделения массив делится на две почти одинаковые части, следовательно, максимальная глубина рекурсии, при которой размеры обрабатываемых подмассивов достигнут 1, составит $log_2(n)$, что дает общую сложность алгоритма равную $O(n * log_2 n)$

В самом несбалансированном варианте каждое разделение дает два подмассива размерами 1 и n – 1 элементов, то есть при каждом рекурсивном вызове больший массив будет на 1 короче, чем в предыдущий раз. Такое может произойти, если в качестве опорного элемента на каждом этапе будет выбран элемент либо наименьший, либо наибольший из всех обрабатываемых. В этом случае потребуется n – 1 операций разделения, а общее время работы составит $\sum_{i=0}^{n}(n-i) = O(n^2)$. Помимо квадратичного времени выполнения, также возникает проблема возможности переполнения стека. Так как глубина рекурсии в худшем варианте достигнет n, что будет означать n-кратное сохранение адреса возврата и локальных переменных процедуры разделения массивов.

1.2. Быстрая сортировка. Параллельный алгоритм.

Параллельное обобщение алгоритма быстрой сортировки наиболее простым способом может быть получено для случая, когда потоки параллельной программы могут быть организованы в виде N-мерного гиперкуба.

Количество необходимых процессоров для реализации данного алгоритма можно вычислить следующим образом:

$P = 2^N$, где N – размерность гиперкуба

Также примем во внимание тот факт, что над гиперкубом размерности N можно провести операцию декомпозиции на два суб-гиперкуба размерности N – 1, соответствующие вершины которых соединены, это позволит рекурсивно применить операцию «сравнить и разделить» для каждого из суб-гиперкубов, в результате которой после log(p) рекурсий каждый из процессов будет иметь локальный список неотсортированных данных, при этом наибольшее значение на i-ом процессоре будет меньше, чем наименьшее значение на i + 1 процессоре.

Представим алгоритм в виде следующих этапов:

1. Рассылка данных. Корневой процесс делит исходный массив данных на P фрагментов размера N / P. После чего выполняется рассылка i-го фрагмента массива соответствующему процессору;
2. Цикл от $i = N$ до $i = 1$
   2.1. Выбор опорного элемента.
   Каждый из процессоров с диапазоном рангов равному $j * 2^i, 0 \leq j < 2^{N-1}$ выбирает опорный элемент по алгоритму медианы по трем элементам (первому, среднему и последнему);
   2.2. Рассылка опорного элемента.
   Процессоры, на которых был выбран опорный элемент на этапе 2.1. являются ведущими для своего суб-гиперкуба. Они выполняют широковещательную рассылку опорного элемента на остальные $2^i - 1$ процессоров
   2.3. Разделение массива данных
   Каждый процесс выполняет разделение собственного фрагмента исходного массива данных на два подмассива: значения больше или равные опорному элементу (high-part) и значения меньшие опорного элемента (low-part)
   2.4. Обмен частями массива
   Представим ранг каждого процессора в виде двоичного числа. Таким образом для процессора, ранг которого в i-ом бите равен 0 (1 соответственно) отправляет соседнему процессу (с противоположным значением бита в ранге на соответствующей позиции) подмассив с числами большими опорного (меньше опорного соответственно).
   2.5. Переупорядочивание частей массива
   Каждый из процессоров выполняет слияние и переупорядочивание полученных частей массива.

3. Корневой процесс собирает локальный массивы данных с каждого процесса в новый массив. При этом локальные фрагменты данных, полученные от каждого из процессов располагаются в порядке возрастания ранга процессора. Таким образом в новом массиве все элементы будут упорядочены.

Для наглядности проиллюстрируем основные шаги алгоритма. Для примера возьмем топологию гиперкуба размерности 3. Тогда количество процессоров будет равно 8. Исходный вид гиперкуба представлен на рисунке 1.2.

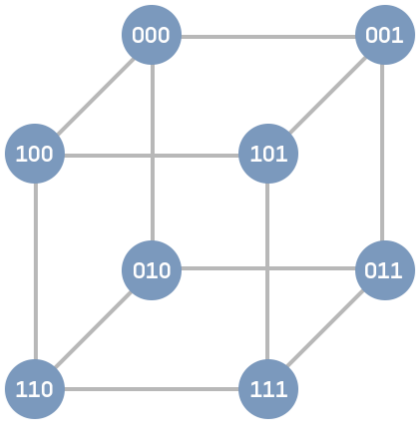

Рисунок 1.2. – Топология типа «гиперкуб» размерности 3

Проиллюстрируем процесс рассылки опорного элемента между узлами вычислительного кластера. Как видно из рисунка 1.3. на каждой из итераций гиперкуб размерности N делится на два гиперкуба размерности N – 1. В каждом из суб-гиперкубов также выбирается ведущий узел, который выполняет рассылку опорного элемента. На рисунке стрелками отмечены направления пересылки опорного элемента.

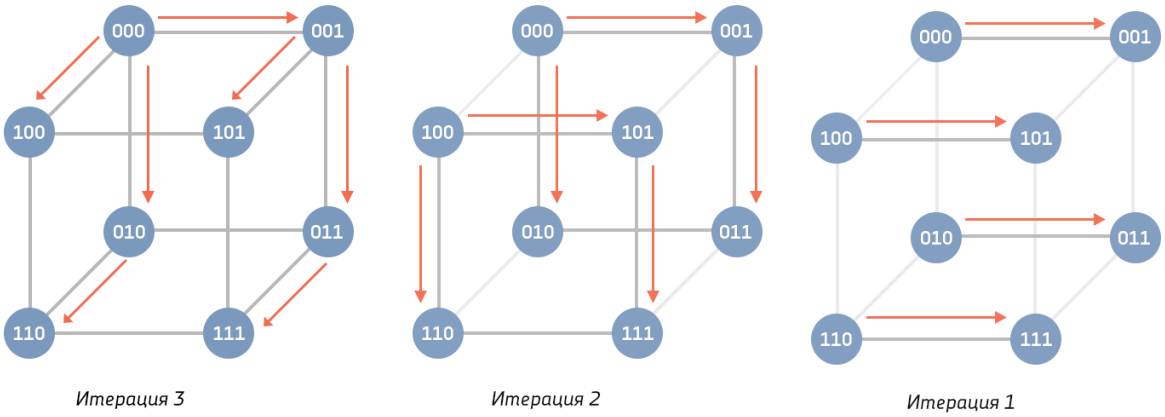

Рисунок 1.3. – Процесс пересылки опорного элемента для каждой из итераций алгоритма

Также проиллюстрируем процесс обмена частями массива между блоками (рисунок 1.4). Как было описано выше, обмен происходит только между узлами вычислительного кластера, в которых биты на позициях соответствующих текущей итерации различаются. При этом обратный цикл позволяет применить очень быстрые битовые операции для вычисления соседних элементов, позволяя обойтись без дорогостоящей операции создания программной топологии.

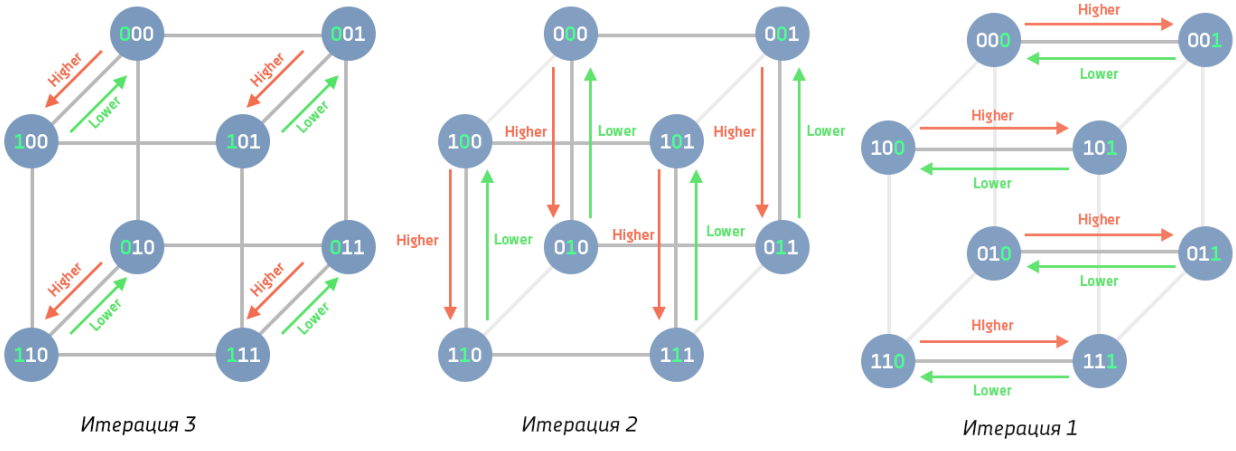

Рисунок 1.4. – Процесс обмена частями массива между соседними узлами кластера

На рисунке 1.4. стрелками показаны направления обмена, а надписи над стрелками соответствуют частям пересылаемого фрагмента данных. Части массива можно представить в виде (1.1)

$$\begin{cases} a_i \geq pivot, \ \forall a_i \in Higher \\ b_i < pivot, \ \forall b_i \in Lower \end{cases} \quad (1.1)$$

Вычисление соседнего элемента происходит при помощи битовых операций, что позволяет значительно повысить производительность программы и избежать дорогостоящей операции создания топологии с помощью инструментов MPI. Вычисление ранга соседнего для обмена элемента происходит по формуле (1.2)

$$Neighbour_{i,k} = rank(P_i)\ XOR\ \big(1 \ll (k-1)\big), \quad (1.2)$$

где i – номер текущего процесса, k – номер итерации

Для иллюстрации рассчитаем по формуле (1.2) соседние элементы для процесса с нулевым рангом на каждой итерации. Получим следующее:

$$\begin{cases} Neigbour_{03} = 000\ \hat{}\ (001 \ll (3-1)) = 000\ \hat{}\ 100 = 100 \\ Neigbour_{02} = 000\ \hat{}\ (001 \ll (2-1)) = 000\ \hat{}\ 010 = 010 \\ Neigbour_{01} = 000\ \hat{}\ (001 \ll (1-1)) = 000\ \hat{}\ 001 = 001 \end{cases}$$

Таким образом результаты расчетов подтверждают возможность замены программной топологии на битовые операции.

Пример параллельной быстрой сортировки

Пусть дан исходный массив данных следующего вида:

24	84	81	55	40	149	97	86	128	67	120	95	183	153	29	0

Этапы алгоритма для гиперкуба размерности 3:

1. Разбиение исходного массива на 2^3 фрагментов и рассылка соответствующих фрагментов на каждый из процесс. Также на каждом из процессов происходит локальная сортировка полученного блока, для упрощения выбора опорного элемента. Тогда локальные массивы примут следующий вид:

Ранг	000		001		010		011		100		101		110		111	
Данные	24	84	55	81	40	149	86	97	67	128	95	120	153	183	0	29

2. Основной этап алгоритма

2.1. Итерация 3

1) Ведущий узел (000) выбирает опорный элемент из своего локального массива и рассылает его остальным узлам вычислительной системы;
2) Каждый из узлов разделяет свой локальный массив данных относительно полученного опорного элемента на две части: значения больше или равные опорному и значения меньше опорного. Таким образом данные на каждом из процессов примут следующий вид (рисунок 1.5)
3) Обмен частями массива с соседним узлом (рисунок 1.6)
4) Переупорядочивание нового локального массива данных. Таким образом данные примут следующий вид:

Ранг	000		001		010	011		100		101		110			111	
Данные	24	67	55	81	40	0	29	84	128	95	120	149	153	183	86	97

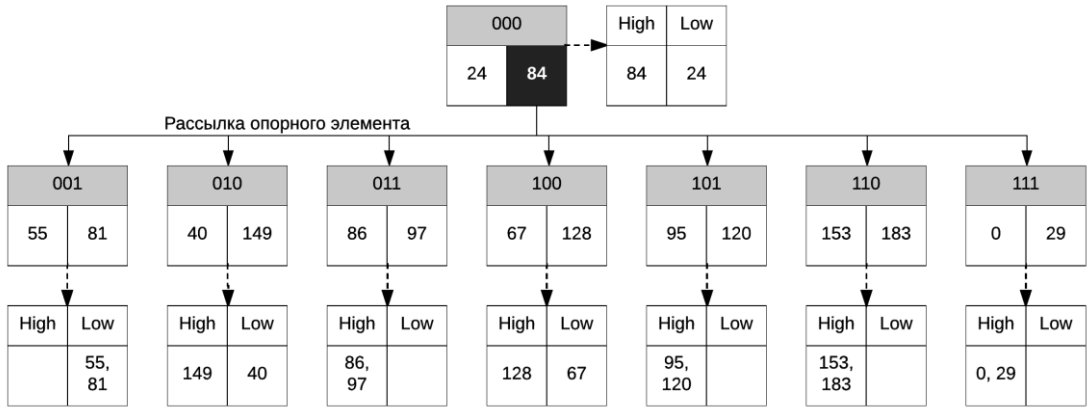

Рисунок 1.5. – Иллюстрация рассылки опорного элемента между процессами, а также этапа деления исходного массива по опорному элементу для 3й итерации

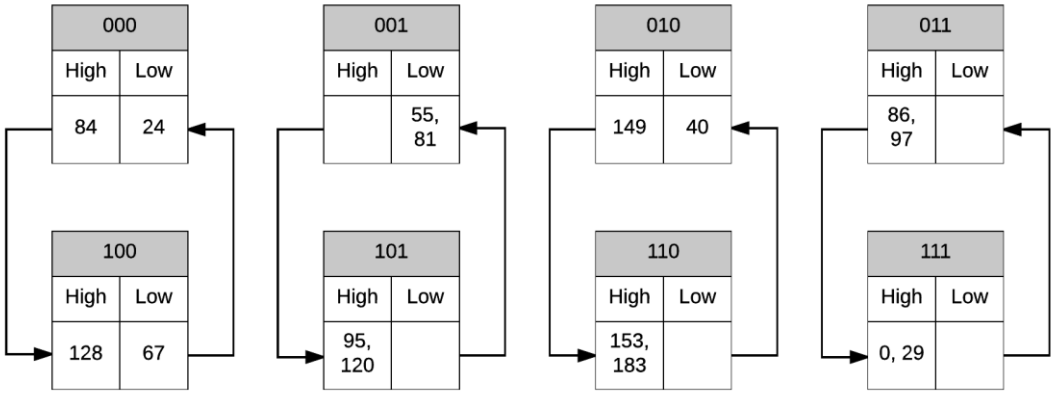

Рисунок 1.6. – Иллюстрация процесса обмена блоками данных между процессами для 3й итерации

2.2. Итерация 2

1) Исходный гиперкуб делится на два субгиперкуба размерностью 2. На каждом из таких суб-гиперкубов выбирается ведущий узел, который выбирает новый опорный элемент и рассылает его процессам внутри суб-гиперкуба. Таким образом на данной итерации ведущими являются узлы (000) и (100)
2) Аналогично предыдущей итерации происходит деление исходных массивов на каждом узле относительно полученного опорного элемента (рисунок 1.7)
3) Обмен частями массива с соседним узлом в каждом гиперкубе (рисунок 1.8)
4) Переупорядочивание нового локального массива данных. Таким образом данные примут следующий вид:

Ранг	000	001	010	011	100	101	110	111
Данные	24, 40	0, 29, 55	67	81	84	86, 95, 97, 120	128, 149, 153, 183	

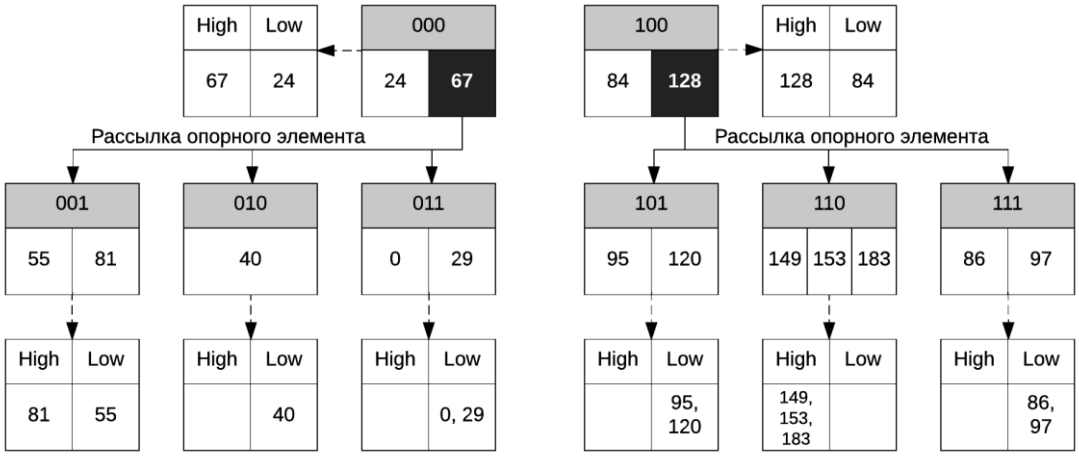

Рисунок 1.7. – Иллюстрация рассылки опорного элемента между процессами, а также этапа деления исходного массива по опорному элементу для 2й итерации

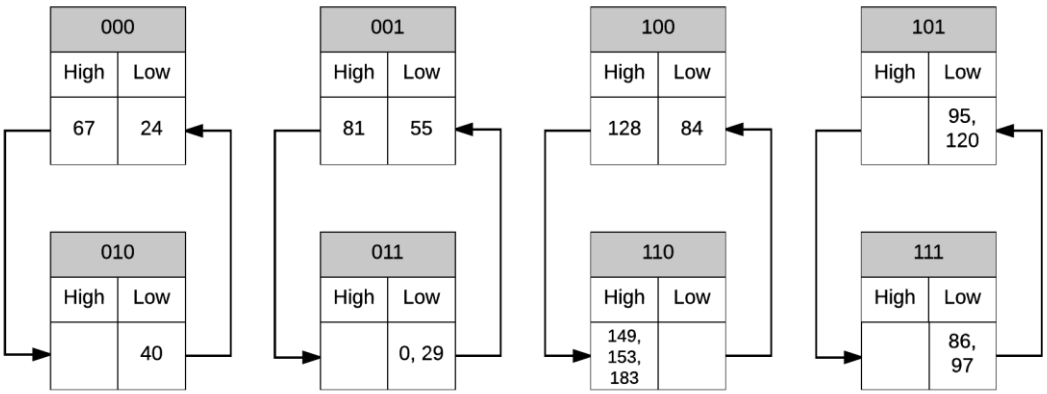

Рисунок 1.8. – Иллюстрация процесса обмена блоками данных между процессами для 2й итерации

2.3. Итерация 1

1) Теперь каждый из двух суб-гиперкубов, которые были получены на итерации 2, также делится ещё на два суб-гиперкуба. Теперь ведущими в своем суб-гиперкубе являются узлы: 000, 010, 100, 110. Данные узлы выполняют поиск и рассылку нового опорного элемента.
2) Аналогично предыдущим итерациям происходит деление данных относительно полученного опорного элемента на каждом из процессов (рисунок 1.9)

3) Обмен частями массива с соседним узлом в каждом гиперкубе (рисунок 1.10)
4) Переупорядочивание нового локального массива данных. Таким образом данные примут следующий вид:

Ранг	000	001	010	011	100	101	110	111
Данные	0, 24, 29	40, 55		67, 81		84, 86, 95, 97, 120	128, 149	153, 183

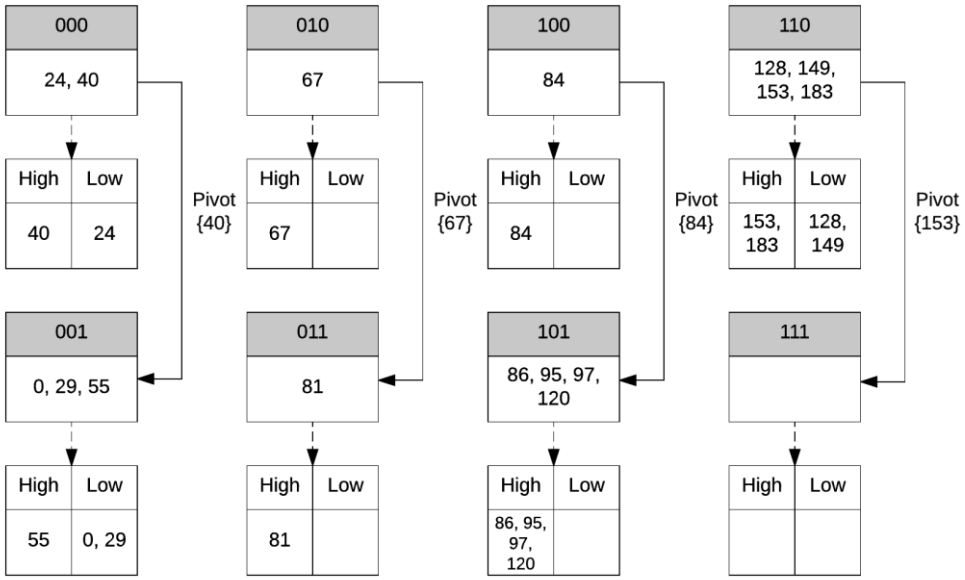

Рисунок 1.9. – Иллюстрация рассылки опорного элемента между процессами, а также этапа деления исходного массива по опорному элементу для 1й итерации

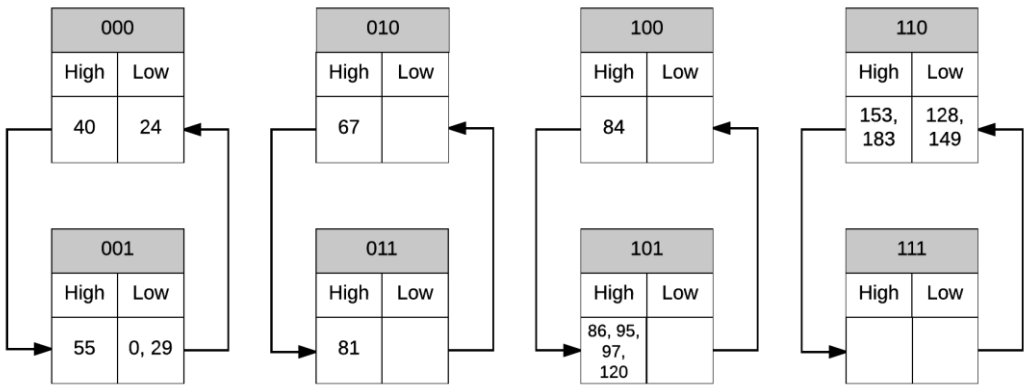

Рисунок 1.10. – Иллюстрация процесса обмена блоками данных между процессами для 1й итерации

3. Завершающий этап

На данном этапе корневой узел исходного гиперкуба выполняет сбор всех локальных блоков данных в новый массив, который представляет из себя отсортированную оригинальную последовательность. Таким образом результирующая последовательность примет вид:

0	24	29	40	55	67	81	84	86	95	97	120	128	149	153	183