Бельская Е. Э. 11-101

Отчет: последовательная и параллельная быстрая сортировка

Последовательная

```
void quicksort(int *arr, int low, int high) {
    if (low < high) {
        int pivot = arr[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                std::swap(arr[i], arr[j]);
            }
        }

        std::swap(arr[i + 1], arr[high]);
        pivot = i + 1;

        quicksort(arr, low, pivot - 1);
        quicksort(arr, pivot + 1, high);
    }
}
```

Краткое объяснение шагов:

1. **Условие остановки:** Если индекс начального элемента (`low`) меньше индекса конечного элемента (`high`), то массив содержит более одного элемента и сортировка нужна.
2. **Выбор опорного элемента (`pivot`):** Выбирается элемент из массива в качестве опорного (в данном случае, последний элемент массива).
3. **Разделение массива:** Массив разделяется на две части так, чтобы элементы, меньшие или равные опорному, находились слева, а элементы, большие опорного, - справа. Это делается с использованием переменной `i`, которая указывает на последний элемент меньший или равный опорному.
4. **Перемещение опорного элемента:** Опорный элемент меняется местами с элементом, следующим за последним меньшим или равным опорному.
5. **Рекурсивная сортировка подмассивов:** Рекурсивно вызывается функция `quicksort` для левого подмассива (от начального индекса до индекса опорного элемента минус один) и для правого подмассива (от индекса опорного элемента плюс один до конечного индекса).

Параллельная

```
void init_sendcounts(int* sendcounts, int size) {
    for (int i = 0; i < size; i++) {
```

```
        sendcounts[i] = n / size;
    }

    if (n % size != 0) {
        int mod = n % size;
        int i = 0;
        while (mod != 0) {
            sendcounts[i % size]++;
            i++;
            mod--;
        }
    }
}

void init_displs(int* displs, int* sendcounts, int size) {
    displs[0] = 0;
    for (int i = 1; i < size; i++) {
        displs[i] = displs[i - 1] + sendcounts[i - 1];
    }
}

int* merge(int size, int *sendcounts, int *displs, int *arr) {
    int *merged = new int[n];
    int pos = 0;

    int *loc_pos = new int[size]{0};
    int *rest = new int[size];
    for (int i = 0; i < size; i++) {
        rest[i] = sendcounts[i];
    }

    while (pos != n) {
        int min;
        int rank;
        for (int i = 0; i < size; i++) {
            if (rest[i] != 0) {
                min = arr[displs[i] + loc_pos[i]];
                rank = i;
                break;
            }
        }

        for (int i = rank+1; i < size; i++) {
            if (rest[i] != 0 && min > arr[displs[i] + loc_pos[i]]) {
                min = arr[displs[i] + loc_pos[i]];
                rank = i;
            }
        }

        merged[pos] = min;
        pos++;

        loc_pos[rank]++;
        rest[rank]--;
    }

    return merged;
}
```

Этот код связан с реализацией операции слияния (merge) для параллельной сортировки массива на нескольких процессах. Код предполагает, что массив `arr` уже разделен между процессами, и каждый процесс сортирует свой подмассив.

Разберем функции поочередно:

1. `init_sendcounts`: Эта функция инициализирует массив `sendcounts`, который содержит количество элементов, которые процесс 0 отправляет другим процессам для слияния. Она равномерно распределяет элементы, а если размер массива `n` не делится равномерно на количество процессов `size`, то остаток распределяется по процессам.
2. `init_displs`: Эта функция инициализирует массив `displs`, который содержит смещения для каждого процесса в общем массиве после слияния. Она вычисляет смещения, начиная с 0 и добавляя к ним соответствующие значения из массива `sendcounts`.

Функция `merge` выполняет слияние (merge) отсортированных подмассивов, представленных различными процессами, в один упорядоченный массив.

Внешний цикл: Цикл выполняется, пока не все элементы слиты в результирующий массив (`pos` не равен `n`).

Внутренний цикл: первый цикл (`for i`) ищет первый доступный (оставшийся) элемент в каждом из подмассивов. Как только находит, сохраняет его значение в `min` и соответствующий индекс процесса в `rank`. Второй цикл (`for i` с начальным значением `rank + 1`) проверяет оставшиеся подмассивы, чтобы найти более минимальный элемент. Если находит, обновляет `min` и `rank` соответственно.

Добавление минимального элемента в результирующий массив:

элемент с минимальным значением (`min`) добавляется в результирующий массив `merged` на позицию `pos`. Увеличивается значение `pos`, указывая на следующую позицию в `merged`.

Обновление позиций и количества оставшихся элементов: `loc_pos[rank]` увеличивается, чтобы указать на следующий элемент в подмассиве, из которого был взят минимальный элемент. `rest[rank]` уменьшается, так как был использован один элемент из подмассива.

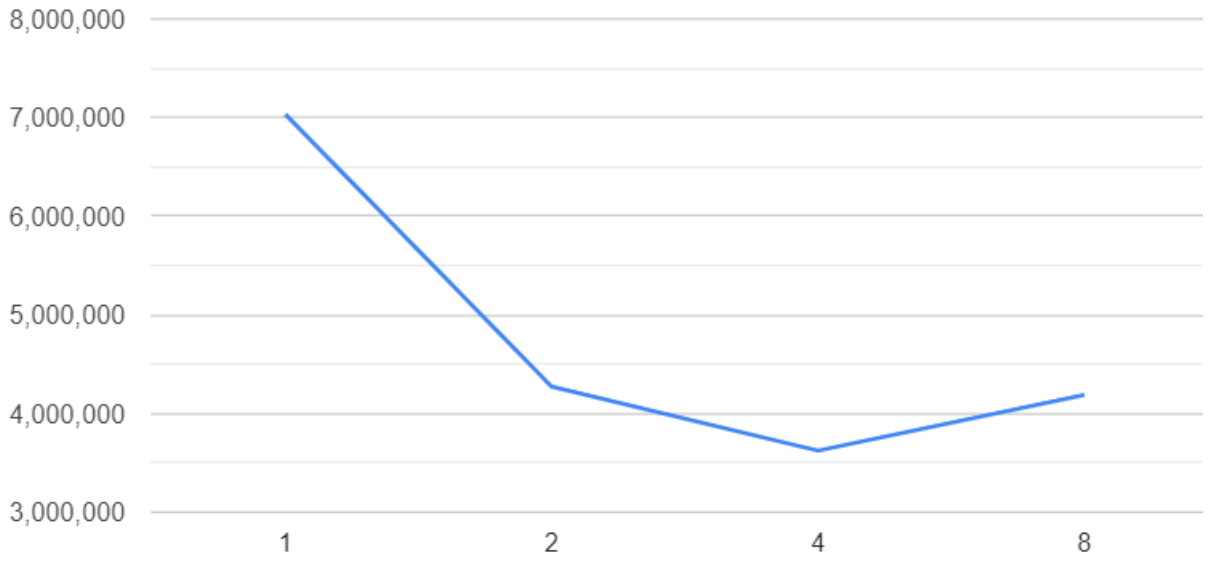
```
MPI_Scatterv((void*)arr, sendcounts, displs, MPI_INT, (void*)loc_arr, sendcounts[rank], MPI_INT, 0,
MPI_COMM_WORLD);

quicksort(loc_arr, 0, sendcounts[rank] - 1);

MPI_Gatherv((void*)loc_arr, sendcounts[rank], MPI_INT, (void*)arr, sendcounts, displs, MPI_INT, 0,
MPI_COMM_WORLD);
```

Этот код использует MPI для параллельной сортировки массива. `MPI_Scatterv` разбивает массив, `quicksort` сортирует локальные части, затем `MPI_Gatherv` собирает их. Процесс с рангом 0 выполняет дополнительное слияние с использованием функции `merge`.

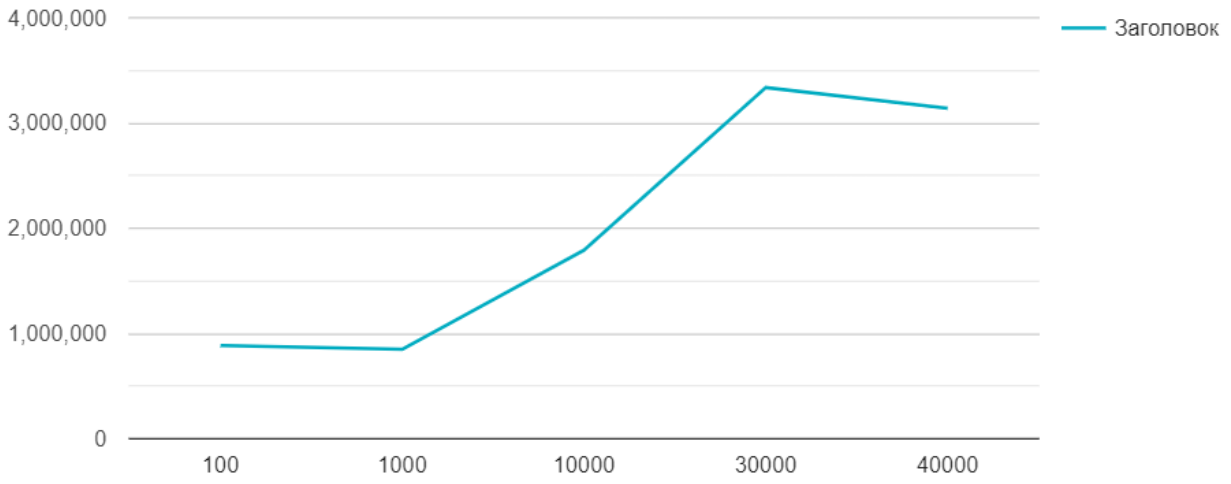
Графики

Соотношение количества процессов и времени выполнения в наносекундах при размере массива n = 30000

.

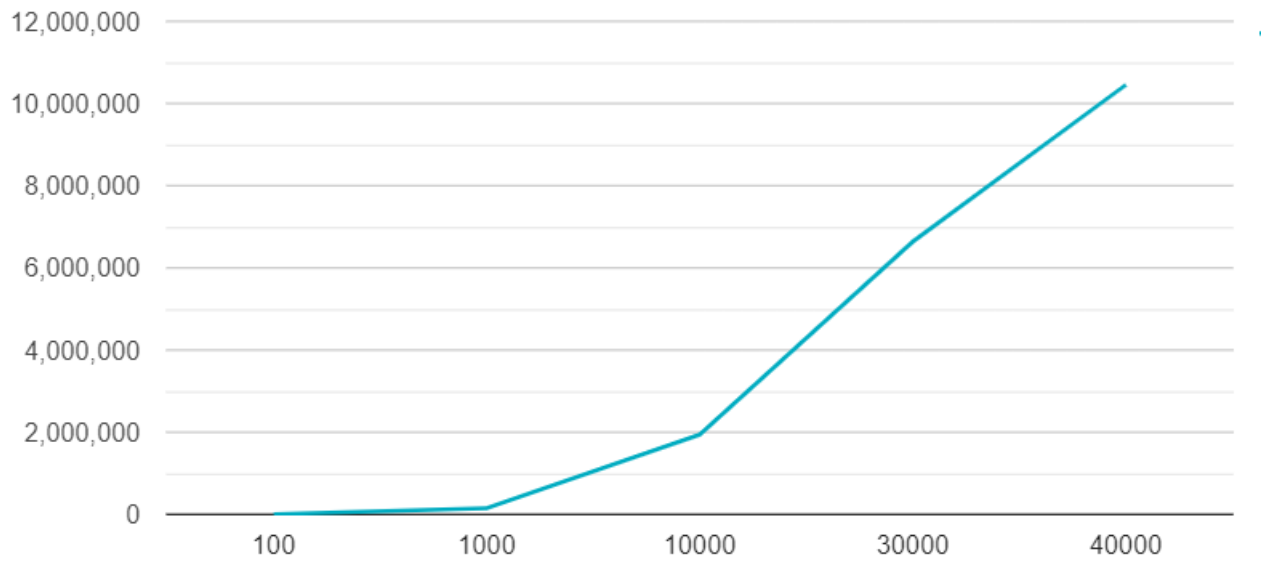


Соотношение количества элементов в массиве и времени выполнения в наносекундах при 4 процессах для параллельного выполнения



100	886400
1000	849300
10000	1790600
30000	3335900
40000	3138700

Соотношение количества элементов в массиве и времени выполнения в наносекундах для последовательного выполнения



100	9400
1000	154200
10000	1944400
30000	6640300
40000	10455900

Показатели ускорения и эффективности алгоритмов в последовательном и параллельном случае.

Расчет ускорения и эффективности при n = 30000:

S = 1.99; E = 0.49
Параллельная версия выполняется в два раза быстрее, только половина ресурсов процессоров используется с максимальной эффективностью.

Расчет ускорения и эффективности при n = 10000:
S = 1.086; E = 0.27
Ускорение чуть больше 1 указывает на некоторую выгоду от параллельной реализации, но она не является значительной.

Вывод:

Параллельная реализация алгоритма быстрой сортировки проявляет высокую эффективность при обработке обширных объемов данных. Однако на небольших объемах данных ее производительность снижается из-за накладных расходов - распределения данных между процессами, синхронизации и координации работ процессов.