Parallel Quick Sort

Algorithms Sorting Algorithms

LINKED LIST PROBLEMS
Algorithms that make you a National Programmer
Practice Problems Included.
BE A NATIONAL PROGRAMMER
Learn ALGORITHMS 100X with practice.
It take years to arrive at these techniques. This book serves you decades of knowledge.
TIME COMPLEXITY ANALYSIS
DYNAMIC PROGRAMMING ON TREES
PROBLEMS ON ARRAY
Mastering QUICK SORT
BINARY TREE PROBLEMS
MUST READ
DECREASE & CONQUER ALGORITHMS
Algorithms that make you a National Programmer
Practice Problems Included.

In this post, we have discussed how to implement Quick Sort algorithm parallely using 5 different approaches including HyperQuickSort, Parallel quicksort by regular sampling and many more.

Table of contents:

1. Introduction to Parallel Programming
2. Sequential Quick Sort Algorithm
3. Approach 1: Naive Parallel Quick Sort
4. Approach 2: Optimized Parallel Quick Sort
5. Approach 3: using both sequential and parallel approaches

Up

6. Approach 4: Hyperquicksort
7. Approach 5: Parallel quicksort by regular sampling

Prerequisite: Quick Sort, OpenMP, CPU vs GPU

Let us get started with Parallel Quick Sort.

Introduction to Parallel Programming

Sorting is a very important building block in most useful algorithms. We need to sort large amounts of data so we can process it efficiently.

The normal way to implement quick sort on serialized processors is whereby steps are executed sequentially until the program terminates when the task is completed. One process is started in the CPU which executes the code line by line.

Parallel programming is whereby a program is broken down into concurrent programs which are executed concurrently on multiple threads on a processor. Here coordination is required.

Why parallel processing?

- Performant.
- Less time taken to complete tasks.
- Lower work loads per processor.

You can learn more on the link at the end of this post.

Generally, there are cases whereby ***performance is prioritized over costs*** and in these cases, parallel processing is implemented. We can see cases of it being used in cryptocurrency mining and video rendering.

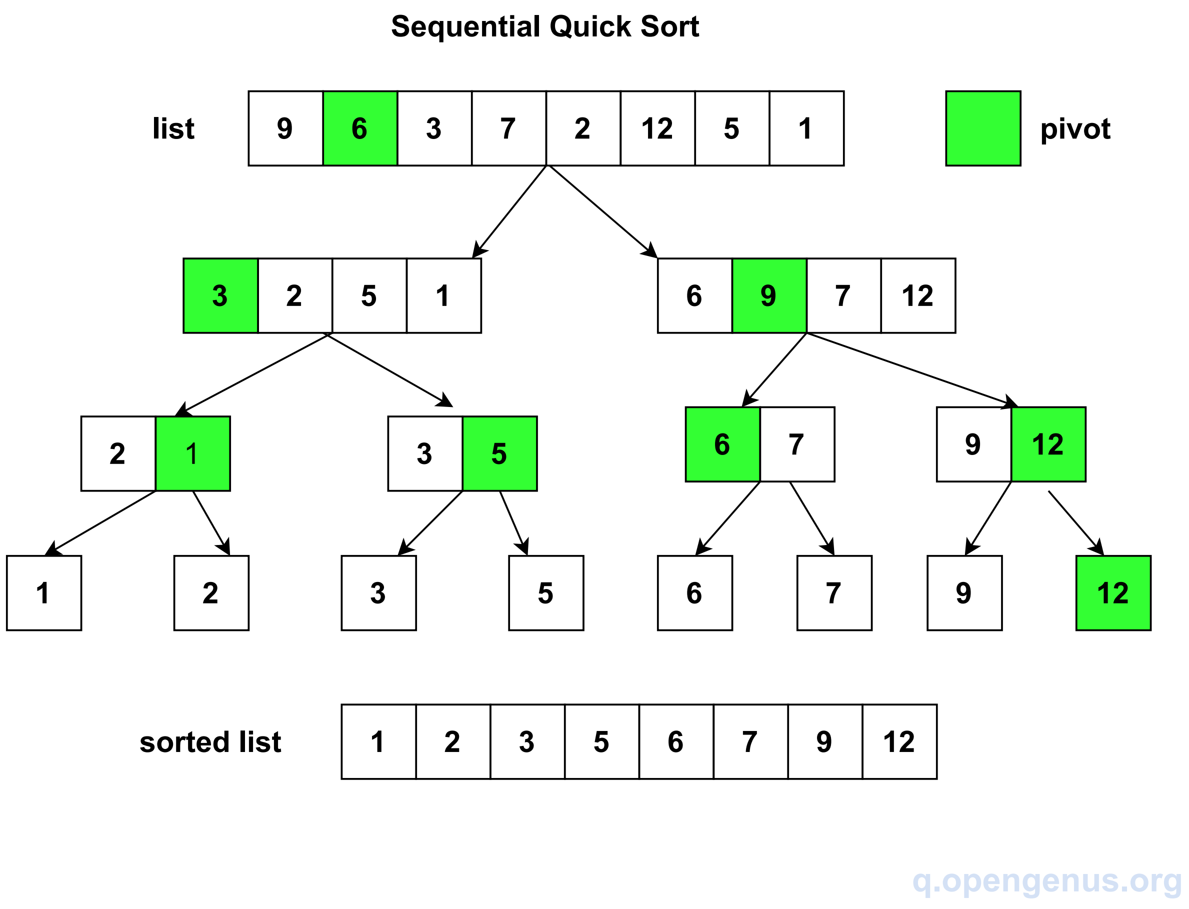
Why is this a problem?

While it is straightforward to implement most algorithms using serial processors, a need may arise whereby we need to implement these algorithms parallelly. Some of the commonly used algorithms especially those recursive in nature(quick sort) do not sit well with GPUs(parallel processors).

In serial CPUs, there is a stack to store recursive calls, while in GPUs there is no "stack" of stored values but an emulation of a large contiguous memory whereby the pointer to the top of the "stack" is tracked.

Sequential Quick Sort Algorithm

1. Find a random pivot p.
2. Partition the list in accordance with this pivot, elements less than pivot to the left of pivot, elements greater than pivot to the right of pivot and elements equal to pivot in the middle. <p =p >p.
   - That is initialize i to first element in list and j to last element.
   - Increment i until list[i] > pivot
   - Decrement j until list[j] < pivot
   - Repeat the above steps until i > j.
   - Replace pivot element with list[j]
3. Recurse on each partition.
4. When the list size is 1, it terminates. This acts as the base case. At this point the partitions are in sorted order so it merges them forming a complete sorted list.



The above algorithm runs on one process executing each step after another. One step has to finish before the next starts.

Sequential quick sort analysis:

The time complexity is O(nlogn) in the average case.
The space complexity is O(logn).

Approach 1: Naive Parallel Quick Sort

In this naive approach the algorithm starts a process at each step to concurrently process the partitions.

Steps.

1. Start a process to find pivot and partition the list into two, p < =p > p.
2. At each step, start p processes proportional to n partitions.
3. Each process finds the pivot element and divides the list based on the selected pivot.
4. Finally processes values are merged, a sorted list is returned.

Analysis:

The algorithm takes $\theta\left(n\right)$ to choose pivot and rearrange list. There are n processes in each step.
Total time complexity is $\theta\left(n^2\right)$.

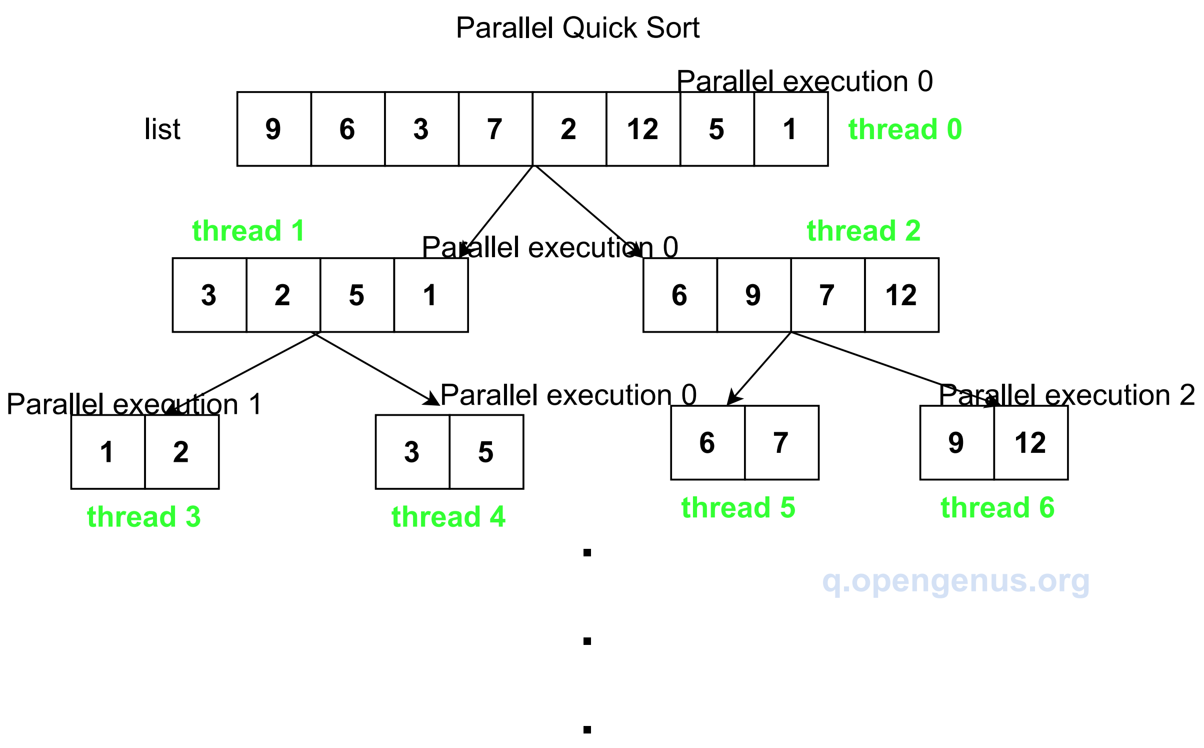
Approach 2: Optimized Parallel Quick Sort

In this approach we change a small detail in the number of processes used at each step. Instead of doubling the number of processes at each step, this approach uses n number of processes throughout the whole algorithm to find pivot element and rearrange the list. All these processes run concurrently at each step sorting the lists.

Steps.

1. Start n processes which will partition the list and sort it using selected pivot element.
2. n processes will work on all partitions from the start of the algorithm till the list is sorted.
3. Each processes finds a pivot and partitions the list based on selected pivot.
4. Finally the list is merged forming a sorted list.

Here is the algorithm pictorially,



Code

```
#include<iostream>
#include<omp.h>

using std::cout;
using std::endl;

class ParallelQuickSort{
    //keep count of threads
    int k = 0;

    private:
        //partitioning procedure
        int partition(int arr[], int l, int r){
            int i = l + 1;
            int j = r;
            int key = arr[l];
            int temp;
            while(true){
                while(i < r && key >= arr[i])
                    i++;
                while(key < arr[j])
                    j--;
                if(i < j){
                    temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }else{
                    temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                    return j;
                }
            }
        }

    public:
        void quickSort(int arr[], int l, int r){
            if(l < r){
                int p = partition(arr, l, r);
                cout << "pivot " << p << " found by thread no. " << k << endl

                #pragma omp parallel sections
                {
                    #pragma omp section
                    {
                        k = k + 1;
                        quickSort(arr, l, p-1);
                    }
                    #pragma omp section
                    {
                        k = k + 1;
                        quickSort(arr, p+1, r);
                    }
                }
            }
        }
       //prints array
        void printArr(int arr[], int n){
            for(int i = 0; i < n; i++)
                cout << arr[i] << " ";
            cout << endl;
        }
        //run the whole procedure
        void run(){
            int arr[] = {9, 6, 3, 7, 2, 12, 5, 1};
            int n = sizeof(arr) / sizeof(arr[0]);
            quickSort(arr, 0, n-1);
            printArr(arr, n);
        }
};

int main(){
    ParallelQuickSort pqs;
    pqs.run();
    return 0;
```

```
}
```

Code Explanation

#include<omp.h> is a header file for openmp so we can be able to use its functions.
We use ***openmp(open multiprocessing)*** , an open source library for multi-threading which will enable use to implement the algorithm concurrently.
You can learn more about it on the link at the end of this post.

#pragma omp parallel sections defines a parallel region containing the code that we will execute using multiple threads in parallel. This code will be divided among all threads.

Variable ***k*** stores the thread number which we print out just for understanding what is happening under the hood.

Compile and run code using ***g++ -fopenmp fileName.cpp -o outputFile && ./outputFile***.

Parallel quick sort analysis

At each step n processes process log(n) lists in constant time O(1).
The parallel execution time is O(logn) and there are n processes. Total time complxity is $\theta(nlogn)$.
This complexity did not change from the sequential one but we have a achieved an algorithm that can run on parallel processors, meaning it will execute much faster at a larger scale.

Space complexity is O(logn).

Approach 3: using both sequential and parallel approaches

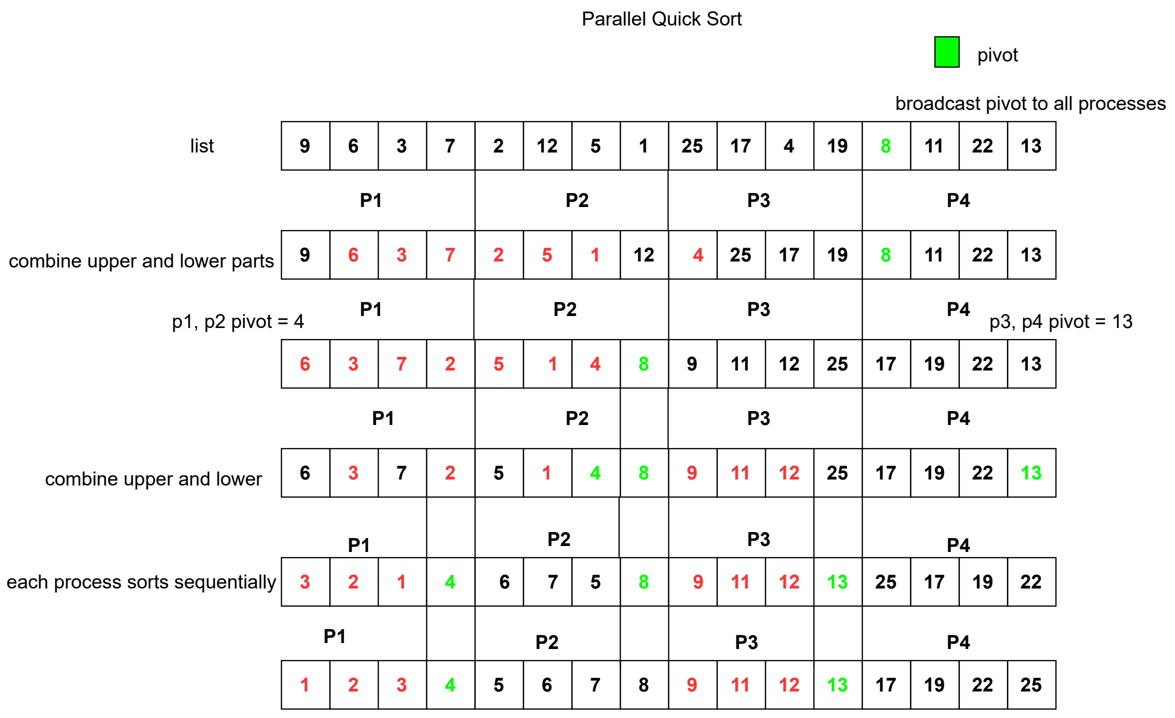
In this approach, the initial list is divided into n smaller sublists that are explicitly dispatched to p remote processors for parallel execution and once the execution is finished at a distributed processor, the sorted sublist is sent back to the central processing node which merges the results from the processes giving a fully sorted list.

Steps:

1. Divide a list of size n to create a number of sublists compatible with the number of available processors p.
2. Create p threads according to the number of available processors.
3. Assign the sublist to each of the p threads so that each has n/p consecutive elements from the original list.

4. Randomly select a pivot element and broadcast it to all partner processes.
5. Each process will partition its elements and divide them into two groups according to the selected pivot. group1 <= pivot <= group2. This happens parallelly across all processes concurrently.
6. Each process in upper half of the process list sends its "low list" to a partner process in the lower half of the process list and it receives the "high list" in return.
7. The lower half will have values less than pivot and upper half elements will have a values greater than pivot.
8. After logP recursions, each process has an unsorted sublist disjoint from values of other processes.
    - The largest value of process i will be less the the smallest value of process i+1.
    - At this point the sublist is small enough, each process sorts its values sequentially and the main process combines the sorted results for each process.

Here is the procedure pictorially,



Analysis:

In above approach, p number of processes work on n number of elements of a list, each p process works on n/p sub-list elements for(logn) steps.
The total time complexity is O(N/P * logN).
Space complexity is O(logn).

Note:

- ***Pivot***; This algorithm does a poor job of load balancing, that is, we need to choose a suitable median value as a pivot element for the algorithm so as to divide the list into at-least equal partitions and maintain balance.
    - Finding a median value is an expensive operation on a parallel processor.
    - The solution for the above problem is to find a median value that is close to the true median.
- ***Combining Processes***; We can concatenate each block in process order, that is find where each block starts and ends and join its end to the start of the next process.

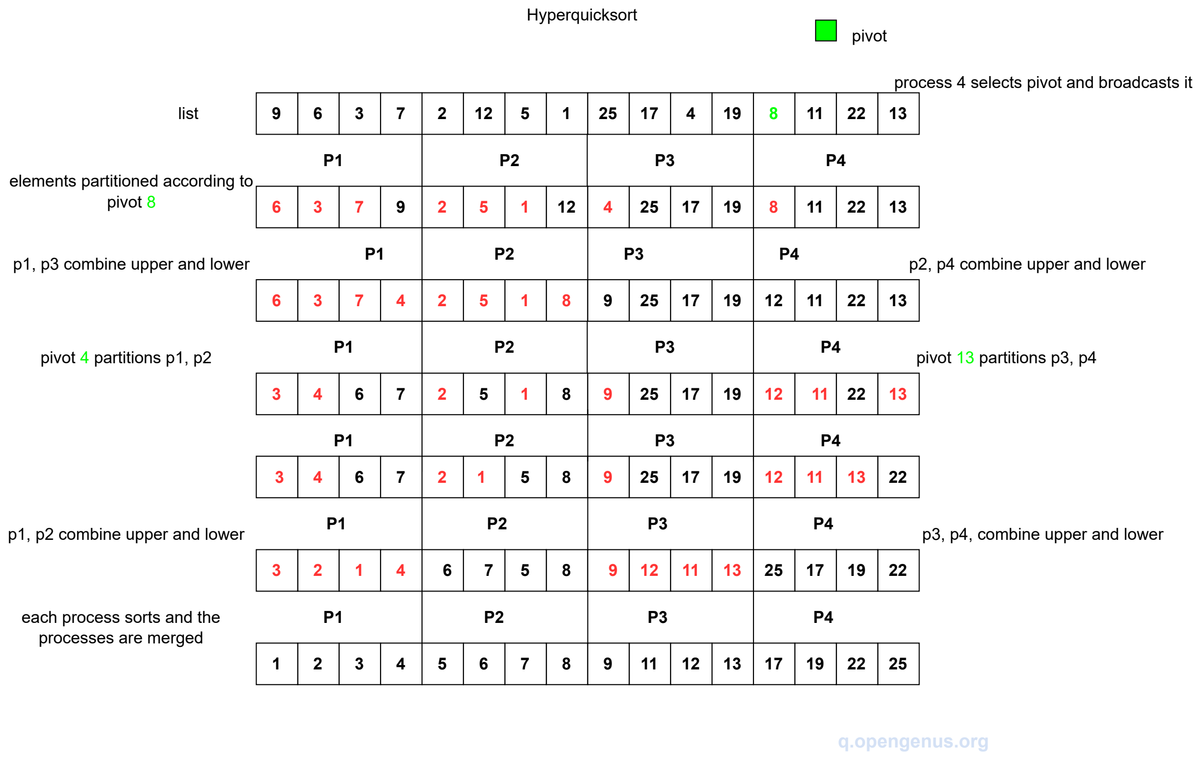
Assuming a machine has 64 threads, a list of size n can be distributed among all the threads n/64 and processing can happen in parallel until the last step where the sublist is sorted sequentially and combined, as n increases the size of list each thread handles increases.
Each threads processes a list in constant time O(1) and there are logn steps.
Assuming an unlimited threads can run in parallel the time and space complexity is O(logn).

Approach 4: Hyperquicksort

This approach is an improvement of the previous approach. Previously we had a problem of load balancing.
This process improves the chances of finding a true median by sorting the sublists sequentially using one pivot that is broadcasted to all processes at the beginning of the algorithm.

Steps:

1. A list of size n is divided among n processes. Assume list of size 16 and 4 processes, each process will handle 4 elements.
2. A process among the four responsible for finding the pivot element, finds a pivot and broadcasts it to all processes which sort their sublists sequentially using the broadcasted pivot element. This step will improve chances of finding pivots close to the true median.
3. We repeat steps 4-6 from the previous approach,
    - Pivot selection and broadcasting to partner processes.
    - Sublist partitioning of low and high values.
    - Swapping of values between partner processes.
4. The remaining top half from one partner process and the received top half from the other partner process are merged into local sublist for each process.
5. Recurse the upper half and lower half of each subprocess to achieve a sorted list.
6. Finally merge the processes in order to get a fully sorted list.



Note:

- There is a communication overhead in that values are passed between partner processes.
- Load imbalance may still occur but the algorithm is better as compared to the previous approach which is much worse at load balancing.

Analysis:

There are logn steps and n processes, the total time complexity is $\theta(nlogn)$.
Space complexity is O(logn).

Approach 5: Parallel quicksort by regular sampling

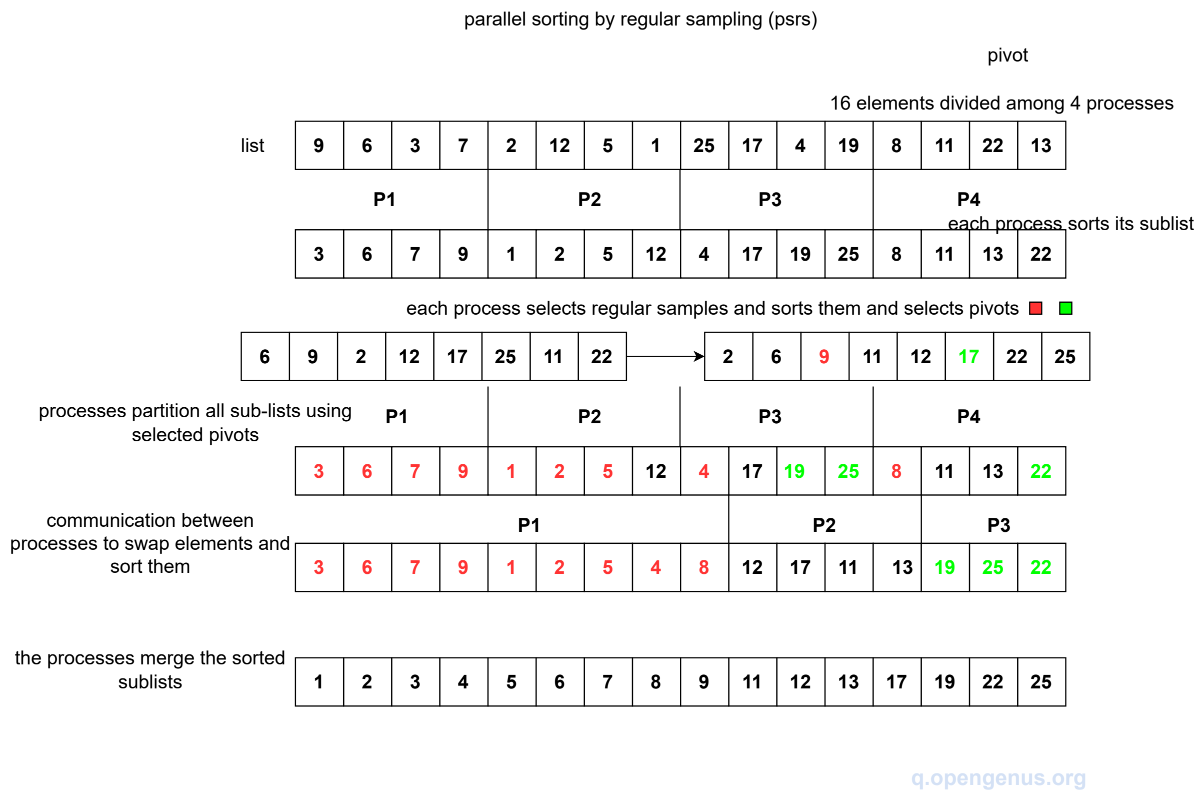
In this approach the algorithm sorts the list sequentially at the beginning and then selects a range of samples which will be used for further partitioning and swapping of elements in subsequent processes.

Steps:

1. Original list is divided among n processes.
2. Each process sorts its sublist using sequential quicksort.
3. Each process selects regular samples from its sorted sublist.
4. A single process gathers the samples, sorts them and broadcasts selected pivots to other processes.

5. All processes use selected pivots to divide their sublists into partitions according to selected pivots concurrently.
6. Processes communicate to swap sorted values with other partner processes.
7. The sorted processes values are merged to return a fully sorted list.

Here is the procedure pictorially,


parallel sorting by regular sampling (psrs)
pivot
16 elements divided among 4 processes
list
9 6 3 7 2 12 5 1 25 17 4 19 8 11 22 13
P1 P2 P3 P4
each process sorts its sublist
3 6 7 9 1 2 5 12 4 17 19 25 8 11 13 22
each process selects regular samples and sorts them and selects pivots 🟥 🟩
6 9 2 12 17 25 11 22
2 6 9 11 12 17 22 25
processes partition all sub-lists using selected pivots
P1 P2 P3 P4
3 6 7 9 1 2 5 12 4 17 19 25 8 11 13 22
communication between processes to swap elements and sort them
P1 P2 P3
3 6 7 9 1 2 5 4 8 12 17 11 13 19 25 22
the processes merge the sorted sublists
1 2 3 4 5 6 7 8 9 11 12 13 17 19 22 25
q.opengenus.org

Advantages of PSRS

- Better load balancing is achieved but not perfect.
- Repeated swappings of same values are avoided, no overhead.
- The number of processes don't have to be a power of 2, during the algorithm some processes can be freed up depending on the order of elements and pivots selected.

Analysis.

Initial quicksort is $\theta\left(\frac{n}{n}\log\ \frac{n}{p}\right)$.
Sorting samples is $\theta\left(p^2\ \log\ \ p\right)$.
Merging sub-arrays is $\theta\left(\frac{n}{p}\log\ \ p\right)$.
Total time complexity is O(nlogn).
Space complexity is O(logn)

Summary.

We discussed 3 main parallel quicksort algorithms.

- Parallel quicksort, ***poor***.
- Hyperquicksort, ***better***.
- PRSR algorithm, ***best***.

Questions.

In some cases we may not be able to store the data we are processing to memory in this case we need an efficient external sorting algorithm, which is this algorithm, can you parallelize it?

With this article at OpenGenus, you must have the complete idea of Parallel Quick Sort.

Erick Lumunge

Erick is a passionate programmer with a computer science background who loves to learn about and use code to impact lives positively.

Read More

Improved & Reviewed by: **OpenGenus Foundation** **Ned Nedialkov**

— OpenGenus IQ: Computing Expertise & Legacy —

ALGORITHMS

Jump Game II: Minimum number of jumps to last

Algorithms

Space Time Complexity [Computer Science]

Mario less and Mario more - CS50 Exercise

Linear Search in Java [both Array + Linked List]

See all 912 posts →

index

In this post, we will explore various ways to solve the minimum number of jumps to last index (Jump Game II) problem. In this, we will use ideas of Dynamic Programming and Greedy Algorithm.

ERICK LUMUNGE

C++

Dot Product of Two Vectors in C++

In this article, we have presented two different ways to do Dot Product of Two Vectors in C++. This involves the use of inner_product method in C++ STL (Standard Template Library).

MAINAK DEBNATH