Concurrent quick sort

normal quick sort

checks if the size of an array segment to be sorted is less than 5 , if it is insertion sort is called .

if not , the partition function is called

once the array is partitioned into two parts and the randomly chosen pivot arrives at the correct location , recursively the same function is called again for the two parts

partition functtion

```
int partition(int * arr , int start , int end)
{
    int pivind = start+ rand() % (end-start)  ;
    int pivot = arr[pivind];
    //    printf("\npivot %d pivind %d \n",pivot,pivind);
    arr[pivind]=arr[end-1];
    arr[end-1]=pivot;
    int smallind = start ;
    for (int i=start;i<end-1;i++)
    {
        if (arr[i]<pivot)
        {
            int temp = arr[i];
            arr[i]=arr[smallind];
            arr[smallind]=temp;
            smallind++;
        }
    }
    arr[end-1]=arr[smallind];
    arr[smallind]=pivot;
    return smallind;
  }
```

randomly choses an element of the array as the pivot and divides the array into two parts , all the elements smaller than the pivot come to the left of the pivot whereas the elements bigger than the pivot come to the right of the pivot

the final index of the pivot is also returned

concurrent quicksort

same as the normal quicksort but after partitioning separate processes are created for the left part and for the right part

```
void concqsort (int * arr,int start,int end)
{
    if(end-start<5 && start < end )
    {
        insertionSort(start,arr,end);
    }
    if (start < end )
    {
        int partind = partition(arr,start,end);
        pid_t leftarr ;
        leftarr=fork();
        int statusl;
        int statusr;
        if (leftarr==0)
        {
            concqsort(arr , start , partind);
            exit(0);
        }
        else
        {
            pid_t rightarr= fork();
            if (rightarr==0)
            {
                concqsort(arr , partind+1 , end );
                exit(0);
            }
            else
```

```
            {
               waitpid(rightarr,&statusr,0);
               waitpid(leftarr,&statusl,0);
            }
         }
      }
  }
```

forking is done twice so as to create two different child processes

Threaded quicksort

after every partition a different thread is created for both the parts pf the array and the same process recursively keeps on going until the entire array is sorted

```
void* threadedqsort(void * arrstruct)
{
    struct thrdarg * args = (struct thrdarg *) arrstruct;
    if ( args->start >= args->end )
        return NULL;

    if(args->end-args->start<5)
    {
        insertionSort(args->start,args->arr,args->end);
    }



    else
    {
        int pivind=partition(args->arr,args->start,args->end);
        struct thrdarg  leftarg  ;
        leftarg.start=args->start;
        leftarg.end=pivind;
        leftarg.arr=args->arr;
        pthread_t leftid;
        pthread_create(&leftid,NULL,threadedqsort,&leftarg);
```

```
        struct thrdarg  rightarg;
        rightarg.start=pivind +1 ;
        rightarg.end=args->end;
        rightarg.arr=args->arr;
        pthread_t rightid;
        pthread_create(&rightid,NULL,threadedqsort,&rightarg);

        pthread_join(leftid,NULL);
        pthread_join(rightid,NULL);
    }
    return NULL;

}
```

Results

- for smaller values of array length **normal quicksort** was the Most efficient since the overhead of creating various processes or threads was more than the time saved due to multiprocessing and multithreading respectively
- as the input array size grew larger the time for the three started to converge
- threaded quickdort could not be tested for very large inputs because of the limitations of the computer
- for very large values of n (>100000) either the time was highly similar or concurrent (multi processing) quicksort proved to be the fastest