QUESTION 1 : Concurrent Quicksort

AIM

Implement a Concurrent version of Quicksort algorithm

IMPLEMENTATION

- When the number of elements in the array for a process is less than 5, perform an insertion sort to sort the elements of that array.
- Otherwise, partition the array around the pivot such that all the elements with a value less than the pivot are positioned before it, while all the elements with a value greater than the pivot are positioned after. In case of equality, they can go on either side of the partition.
- Recurse for the low and high subarray.

Concurrent quick sort using processes

Here, the recursion part is done using processes. Create a child process (using fork) that sorts the higher subarray by calling the same function again. The parent process would create another child process (using fork) that sorts the lower subarray parallely by calling the same function again. The parent process waits till both the children are done with their sorting process.

```
void quicksort(int *arr, int l, int r){
    if(l>r) _exit(1);

    //insertion sort
    int x;
    if(r-l+1<=5){
        int a[5], mi=INT_MAX, mid=-1;
        for(int i=l;i<r;i++)
        {
            int j=i+1;
            for(;j<=r;j++)
                if(arr[j]<arr[i] && j<=r)
                {
                    int temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
        }
        return;
    }
    else
    {


        x=partition(arr,l,r);

    }



    int pid1 = fork();
    int pid2;
    if(pid1==0){
        //sort left half array
        quicksort(arr, l, x-1);
        _exit(1);
    }
    else{
        pid2 = fork();
        if(pid2==0){
            //sort right half array
            quicksort(arr,x+1,r);
            _exit(1);
        }
        else{
            //wait for the right and the left half to get sorted
            int status;
            waitpid(pid1, &status, 0);
            waitpid(pid2, &status, 0);
        }
    }
    return;
}
```

Concurrent quick sort using threads

Here, the recursion part is done using threads. Create two threads that sorts two subarrays separately and join them using the pthread_join function.

```
void *threaded_mergesort(void* a){
    //note that we are passing a struct to the threads for simplicity.
    int x;
    struct arg *args = (struct arg*) a;

    int l = args->l;
    int r = args->r;
    int *arr = args->arr;
    if(l>r) return NULL;

    //insertion sort
    if(r-l+1<=5){
        int a[5], mi=INT_MAX, mid=-1;
        for(int i=l;i<r;i++)
        {
            int j=i+1;
            for(;j<=r;j++)
                if(arr[j]<arr[i] && j<=r)
                {
                    int temp = arr[i];

                    arr[i] = arr[j];
                    arr[j] = temp;
                }
        }
        return NULL;
    }
    else
    {


        x=partition(arr,l,r);

    }



    //sort left half array
    struct arg a1;
    a1.l = l;
    a1.r = x-1;
    a1.arr = arr;
    pthread_t tid1;
    pthread_create(&tid1, NULL, threaded_mergesort, &a1);

    //sort right half array
    struct arg a2;
    a2.l = x+1;
    a2.r = r;
    a2.arr = arr;
    pthread_t tid2;
    pthread_create(&tid2, NULL, threaded_mergesort, &a2);

    //wait for the two halves to get sorted
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);



}
```

Normal quick sort

Here, the recursion part is done using functions. Recursively call the function to sort the upper and lower parts.

COMPARISON OF THE PERFORMANCES

For comparing the performances, calculate the time taken for sorting the same array in the three methods and divide them to get a permformance ratio

To run the files run

> *gcc -pthread Q1.c; ./a.out*

SAMPLE RUN

Running concurrent_mergesort for n = 15 `time = 0.016333`

`1 1 1 1 1 2 2 2 2 2 2 4 5 5 55`

Running Threaded_quicksort for n = 15 `time = 0.001599`

`1 1 1 1 1 2 2 2 2 2 2 4 5 5 55`

Running normal_quicksort for n = 15

`time = 0.000003`

`1 1 1 1 1 2 2 2 2 2 2 4 5 5 55`

```
normal_mergesort ran:
        [ 5268.674399 ] times faster than concurrent_mergesort
        [ 515.741640 ] times faster than threaded_concurrent_mergesort
```