QUICKER QUICKSORT

Jivitesh Jain, 2018101092, UG-2, CSE

Overview

The quicksort sorting algorithm works by dividing the task of sorting the array into two subtasks of half the size - a classic case of divide and conquer. After partitioning the array such that all elements greater than a randomly chosen pivot are to the left of it and others are to its right, the algorithm works separately on the left and right subarrays. This is an opportunity to parallelize the algorithm, as the calls on the left and right subarrays run independent of each other and hence can be carried out parallely on separate processing cores.

The program achieves this by making the recursive function calls on the left and right subarrays in different processes (or threads) - allowing the scheduler to schedule them on separate cores.

However, this does *not* guarantee a tangible performance improvement over serialized quicksort. This is because the overheads of creating child processes (or threads) can easily overshadow the benefits, and the multiple processes (or threads) created may not even be scheduled on different processing cores by the scheduler, which nullifies the benefits and, coupled with the overheads, can make the parallelized version much slower than the usual serial quicksort.

Implementation

Insertion Sort

To avoid creating too many child processes (or threads) the program runs an insertion sort on arrays of length atmost 5. The code for which looks like:

```
if (end - start + 1 <= 5) {
    for (int i = start + 1; i <= end; i++) {
        int temp = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > temp) {
            arr[j + 1] = arr[j];
            j--;
        }
```

```
        arr[j + 1] = temp;
    }
    return;
}
```

All three variants use this code snippet verbatim.

The Partition Function

For longer arrays, where quicksort is actually required, the partition function uses a random pivot and segregates the array such that all elements to the left of the pivot are smaller than it and the other way.

```
int pivot = get_pivot_index(start, end);
swap(arr, pivot, end);
pivot = end;

int i = start - 1;
for (int j = start; j < end; j++) {
    if (arr[j] < arr[pivot]) {
        i++;
        swap(arr, i, j);
    }
}
swap(arr, i + 1, end);
pivot = i+1;
```

Where `get_pivot_index()` generates a random number between start and end.

The partition function is also used verbatim in all three approaches.

Serial Quicksort

After partitioning, the function simply calls itself on both the halves.

```
void normal_quick_sort(int arr[], int start, int end) {
    if (start >= end)
```

```
        // BASE CASE
        return;

    if (end - start + 1 <= 5) {
        // INSERTION SORT
    }

    // PARTITION

    normal_quick_sort(arr, start, pivot - 1);
    normal_quick_sort(arr, pivot + 1, end);
}
```

Multiprocess Quicksort

After partitioning, the function forks into two child processes (three processes in all), and makes the recursive calls in those proccesses. The parent process simply creates the child processes and wait for them to finish before returning control.

```
void multiproc_quick_sort(int arr[], int start, int end) {
    if (start >= end)
        // BASE CASE
        return;

    if (end - start + 1 <= 5) {
        // INSERTION SORT
    }

    // PARTITION

    pid_t left_pid = fork();
    if (left_pid < 0) {
        // IN PARENT, CHILD NOT CREATED
        perror("Could not fork");
        return;
    } else if (left_pid == 0) {
```

```
        // IN LEFT CHILD, SORT LEFT HALF
        multiproc_quick_sort(arr, start, pivot - 1);
        exit(0);
    } else {
    // IN PARENT, CHILD CREATED

        pid_t right_pid = fork();
        if (right_pid < 0) {
            // IN PARENT, CHILD NOT CREATED
            perror("Could not fork");
            return;
        } else if (right_pid == 0) {
            // IN RIGHT CHILD, SORT RIGHT HALF
            multiproc_quick_sort(arr, pivot + 1, end);
            exit(0);
        } else {
            // IN PARENT, CHILD CREATED
            int w_st_left, w_st_right;
            waitpid(left_pid, &w_st_left, 0);
            waitpid(right_pid, &w_st_right, 0);
        }
    }
}
```

Multithreaded Quicksort

After partitioning, the function creates two threads - one to sort each half. The recursive calls are made in these threads, while the parent waits for them to finish by calling `pthread_join()` on them.

```
void* multithread_quick_sort(void* inp) {
    args* input = (args*)inp;
    int start = input->start;
    int end = input->end;
    int* arr = input->arr;

    if (start >= end)
```

```
        // BASE CASE
        return NULL;

    if (end - start + 1 <= 5) {
        // INSERTION SORT
    }

    // PARTITION

    args left_input;
    left_input.start = start;
    left_input.end = pivot - 1;
    left_input.arr = arr;

    pthread_t left_tid;
    pthread_create(&left_tid, NULL, multithread_quick_sort, (void*)&left_inpu
t);

    args right_input;
    right_input.start = pivot + 1;
    right_input.end = end;
    right_input.arr = arr;

    pthread_t right_tid;
    pthread_create(&right_tid, NULL, multithread_quick_sort, (void*)&right_inpu
t);

    pthread_join(left_tid, NULL);
    pthread_join(right_tid, NULL);

    return NULL;
}
```

Shared Memory

For multiple processes to share data (the array to be sorted), a shared memory region has to be created, which is achieved by the following code snippet.

```
key_t mem_key = IPC_PRIVATE;
int shm_id = shmget(mem_key, sizeof(int) * n, IPC_CREAT | 0666);
int* b =  (int*)shmat(shm_id, NULL, 0);

for (int i = 0; i < n; i++) {
    b[i] = arr[i];
}
```

This memory is later detached and deleted using the following code snippet:

```
if (shmdt(b) == -1) {
    perror("shmdt");
    exit(1);
}
if (shmctl(shm_id, IPC_RMID, NULL) == -1) {
    perror("shmctl");
    exit(1);
}
```

Results

As explained above, serialized quicksort usually runs the fastest on small input sizes because of the lack of overheads of process/thread creation (which still does not guarantee that the processes/threads be scheduled on different processing cores).

Multithreaded quicksort usually runs faster than multiprocess quicksort because creating threads is faster than creating processes as threads don't have a separate PCB and share several memory regions.

```
Enter the number of elements: 20
Enter the elements: -3 -3 4 10 2 -11 -100 1 1 1 0 100 -100 100 200 20 -20 -3 -3
04 40
Running normal quicksort.
-304 -100 -100 -20 -11 -3 -3 -3 0 1 1 1 2 4 10 20 40 100 100 200
This took 0.000013 seconds.
```

```
Running multiprocessing quicksort.
-304 -100 -100 -20 -11 -3 -3 -3 0 1 1 1 2 4 10 20 40 100 100 200
This took 0.004138 seconds.

Running multithreaded quicksort.
-304 -100 -100 -20 -11 -3 -3 -3 0 1 1 1 2 4 10 20 40 100 100 200
This took 0.000336 seconds.

For n = 20:

Normal quicksort was:
324.476010 times faster than multiprocess quicksort.
26.332262 times faster than multithread quicksort.

Multithreaded quicksort was 12.322375 times faster than multiprocess quicksort.
```

The speed-up as a result of parallelization can be noticed on large inputs with the threshold for insertion sorting increased to 500 to avoid performing small computations in separate processes/threads (decreasing the associated overhead):

```
Enter the number of elements: 100000
Enter the elements:
...

Running normal quicksort.
...
This took 0.012211 seconds.

Running multiprocessing quicksort.
...
This took 0.011994 seconds.

Running multithreaded quicksort.
...
This took 0.003653 seconds.
```

```
...
```