Process Synchronization

Concurrent Quicksort

Problem statement

Given the number of elements (N) in the array, and the elements of the array. You are required to sort the numbers using a Concurrent and threaded version of quick sort.

Quick sort algorithm

Quicksort is a Divide and Conquer sorting algorithm. It is one of the most efficient sorting algorithms and is based on the idea of splitting an array into smaller ones. The algorithm first picks an element called a pivot and partitions the array into a low subarray , elements smaller than the pivot and a high subarray , elements greater than the pivot. This step is called the partitioning step. Quicksort algorithm recursively applies the above step to the sub-arrays, resulting in sorting the array.

Requirements

1. **sys/ipc.h** for inter-process communication
2. **sys/shm.h** fr shared memory
3. **sys/wait.h** for wait() function for waiting for processes to complete
4. **time.h** for obtaining times at required time in runtime
5. **pthread.h** for threads

Compiling the code

gcc -g <filename> -lpthread

Input format

N (number of array elements)

arr[N] (array with N elements)

Implementation

Concurrent quick sort

As concurrent quick sort algorithm runs many processes concurrently, forking the processes is required. As it is a divide and conquer algorithm, it divides the given array based on pivot and runs the two subarrays as two separate arrays.

As the array is accessed by more than one processes, we need to create a shared memory region.

If the size of the array is less than or equal to 5 (five), the code executes insertion sort on the given subarray.

If the size of array is greater than 5, then a pivot is selected randomly from the array and the array is sorted into lower subarray and higher subarray w.r.t the pivot. The chosen pivot is swapped with last element of the array for easier filtering of the array into lower and higher subparts.

If the size of array is n, and start index is s and end index is e, then a random pivot is chosen and is swapped with arr[e]. Later we run a for loop from index s to e-1 keeping two index pointers (one for iterating through the array and other for keeping track of partition index). After completing this step out array is filtered, then we swap back pivot to its place (which is the partition index pointer which we maintained).

Now we run lower and higher subarrays as two different processes by passing arguments (indices and array). We have to wait for both processes to exit using wait() function. After both child processes exit, the parent process also exits which gives us the sorted array.

Threaded quick sort

As it is a divide and conquer algorithm, it divides the given array based on pivot and runs the two subarrays as two separate arrays.

If the size of the array is less than or equal to 5 (five), the code executes insertion sort on the given subarray.

If the size of array is greater than 5, then a pivot is selected randomly from the array and the array is sorted into lower subarray and higher subarray w.r.t the pivot. The chosen pivot is swapped with last element of the array for easier filtering of the array into lower and higher subparts.

If the size of array is n, and start index is s and end index is e, then a random pivot is chosen and is swapped with arr[e]. Later we run a for loop from index s to e-1 keeping two index pointers (one for iterating through the array and other for keeping track of partition index). After completing this step out array is filtered, then we swap back pivot to its place (which is the partition index pointer which we maintained).

Now we run lower and higher subarrays as two different threads by passing arguments (indices and array) as void* as thread function's argument and return value has to be of void* type. We have to wait for both threads to finish executing using thread_join() function. After both both threads exit, the current thread also exits which gives us the sorted array.

Comparison of runtimes

To compare runtimes of different kinds of quick sorts, we need to find out the runtime of each kind which is provided by clock() function.

After finding different runtimes, the code gives the ratios of different combinations of algorithms to compare runtimes of normal quicksort, concurrent quick sort using processes and threaded quick sort using threads.

For small inputs

*Normal quicksort is faster than concurrent quicksort. Threaded quicksort is slowest***.**

As concurrent quicksort creates processes, there is a lot of overhead created for it, hence it is slower than normal quicksort. Threaded quicksort needs to be synchronized which takes longer time than normal and concurrent quicksort.

For large inputs

Concurrent quicksort is faster than normal quicksort. Threaded quicksort is slowest.

Concurrent has overhead of creating processes, but due to its concurrency, it overcomes its overhead and becomes faster than normal quicksort. Threaded quicksort needs to be synchronized which takes longer time than normal and concurrent quicksort.

NOTE : Due to some memory-related issues and constraints, the code works for relatively smaller inputs.