Akshita Mittel
CS13B1040
13 September 2015

Multithreaded and MultiProcess Sorting

OS ASSIGNMENT 6

The Sorting algorithms in both processes were the same. Since both sorts were implemented iteratively, no additional functions were required.

Algorithm 1 : Insertion Sort.

Algorithm 2 : Bubble Sort.

MULTITHREADING:

Three arrays were created globally, a unsorted array, sorted array, and temporary array. In the main of the parent, the array length was inputted by the user, the array length is again a global variable; accessible by all the threads . After which random numbers between 0 - 100, were stored sequentially in both the sorted and unsorted array.

Three threads were created using, pthread_create, namely sortOne, sortTwo, and merge. No arguments were sent and no return type was expected from any thread, so the corresponding fields have been marked NULL. In sortOne, the first half, i.e. indexes 0 to half of the length is sorted using insertion sort. The basic iterative algorithm for insertion sort (taking an element iteratively and placing it in its correct place, corresponding to the other numbers) was used.

Bubble sort was used to sort the other half of the array in sortTwo. Again, this is an iterative sorting algorithm based on two iterative loops, where the numbers are swapped over and over till all the numbers are in ascending order.

In thread three, merge was carried out. Both halves of the array were traversed element by element and the lower element from each half was placed next in the merged array. After

all the elements were merged, the temporary (merged) array, was copied back into the sorted array.

In the main program the sorted and unsorted arrays are then printed.

Also, a clock was started before calling the threads, after printing the output the clock was stopped and the difference was divided by the number of clock cycles per second to get the time taken for executing the entire process.

MULTIPROCESSING:

Shared memory was used in multiprocessing. The memory was shared between both the children processes and the parent process using shm and mmap, which creates a map of specific size and returns a pointer to that location. The size is allocated using the ftruncanate function. In both ftruncanate and mmap, the shared memory is recognized by the shm_fd, which was created using shm_open. The memory was created and opened in both read and write mode, since the child needs to both write to and read from the memory. The argument MAP_SHARED in mmap, specifies that the memory is a shared memory. The memory was then allocated as follows:

0	**1**	**…**	**num**	**num +1**	**…**	**2* num**	**2num+1**	**…**	**3* num**
num	Sorted Start	…	Sorted end	Unsorted start	…	Unsorted end	Temp start	…	Temp end

Index 0: The number of entries in the array

Index 1 - num: Sorted array

Index num+1 - 2*num: Unsorted array

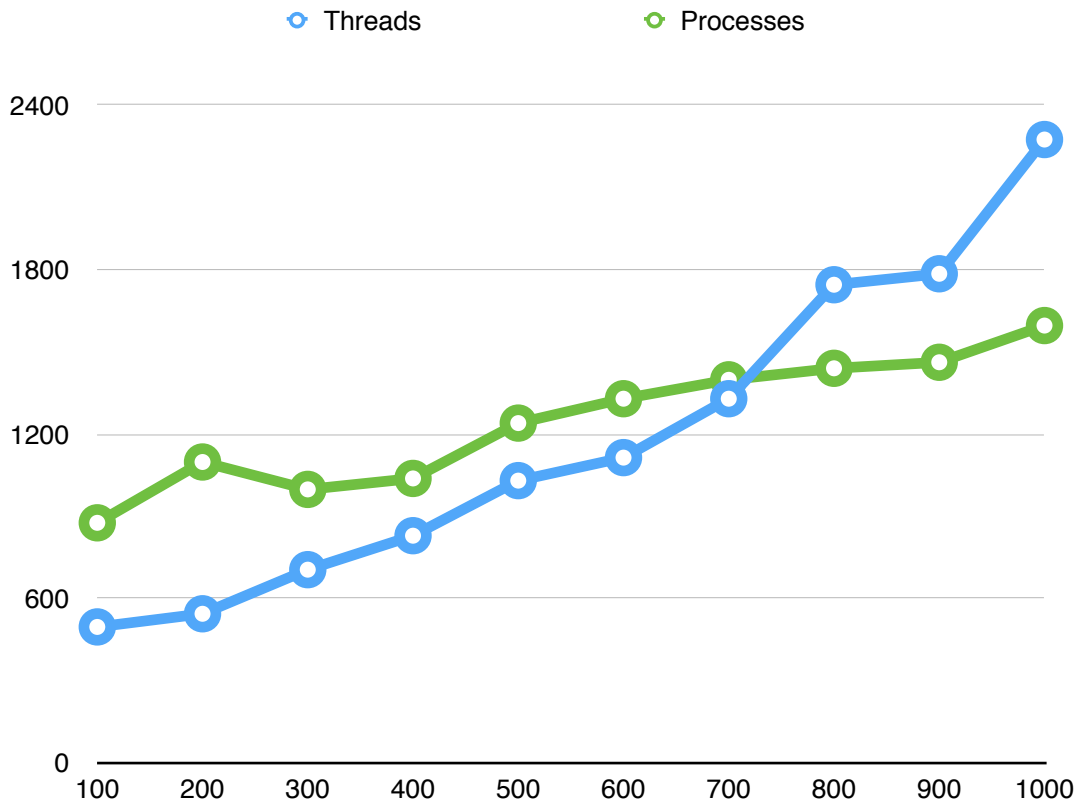
Index 2*num+1 - 3*num: Temporary array

Like in threads, the parent process stores random numbers in the sorted and unsorted arrays in the shared memory. The the sorting processes are forked and the same sorting algorithms are carried out like before. The parent process waits for the sorting processes to terminate, before it forks again and merges the two halves of the array. Again in the same manner as the threading program. The output is then printed and the shared memory is unlinked.

The same clock cycle mechanism is used to calculate the time required to carry out the entire process.

TIME COMPARISON:

The time measurement process has been described in the program description. The time outputted was shown in second, which were converted into microseconds to yield a reasonable graph as follows:



The x-axis show the number of entries in the array, whereas the y-axis is the time taken for the process to execute in microseconds.

From the graph above we can deduce that there is not much difference once a thread or a process is running. However, threads tend to be slightly more efficient initially as they are executed in the same memory space, unlike processes. However, as the load increases, processes are more likely to be able to distribute the load more efficiently.