Assignment 2

Shared memory parallelization using Pthreads

Anton Sjöberg, Alessandro Piccolo.

1. Problem description

The goal of this assignment was to sort an array of double precision numbers using quick sort in parallel.

2. Implementation

First, a serial quicksort algorithm was implemented. The serial Quicksort algorithm can be described as follows:

1. Select a pivot element.
2. Divide the data into two sets according to the pivot(smaller and larger).
3. Sort each list using Quicksort recursively.

2.1 Divide and conquer parallelization

The divide and conquer parallelization is a very standard parallelization implementation. Every time the array is split using the pivot, a new thread is created by the current thread. The old thread takes the right part of the split array and the new thread takes the left part. This stops when we reach our predefined max level. Every thread then uses a serial quicksort algorithm to sort the small arrays. When the threads have sorted their respective smaller arrays of numbers, they all merge in order.

2.2 Peer parallelization

When using peer parallelization, also know as bucket sort, n threads are created at the beginning of the sorting. Then the array is divided into n intervals, and given to the respecting thread of the interval it belongs to. After that each thread uses the serial quicksort on the elements in the thread. When the threads have sorted their respective smaller arrays of numbers, they all merge in order.

3. Results

In figure 1 and 2 shows the speedup of the quicksort algorithm with bucket respectively divide and conquer strategy. Each dot represents three runs and the plotted value is the mean value of those three runs.

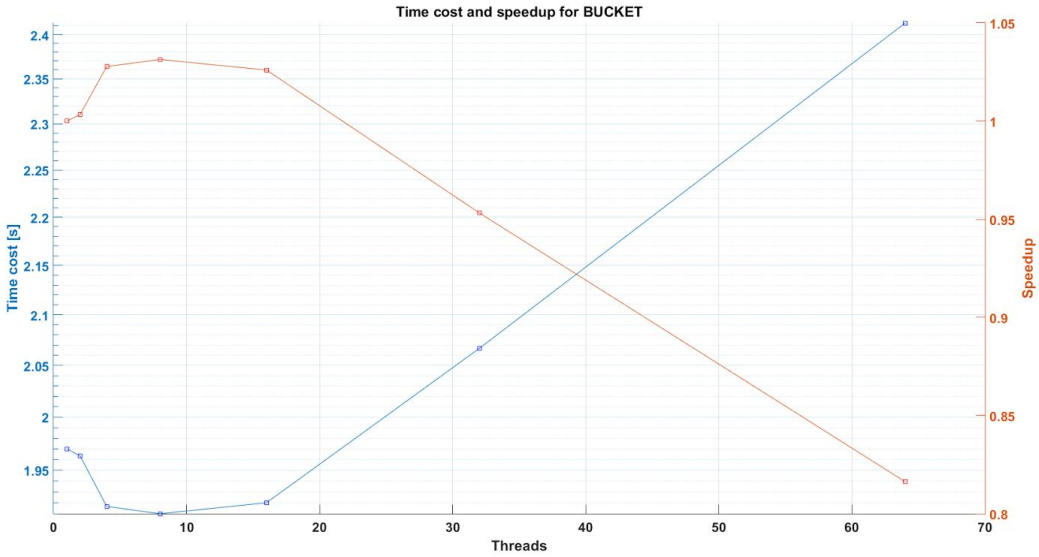

Figure 1. Time and speedup when using Peer Parallelization.

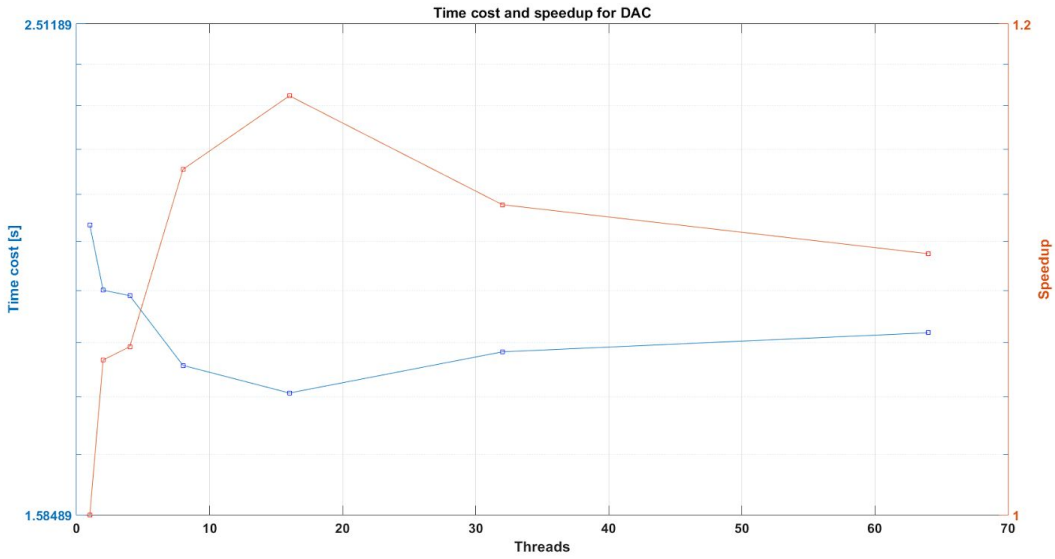

Figure 2. Time and speedup when using the Divide And Conquer Parallelization.

4. Conclusion

Initially when increasing the numbers of threads, the speedup goes up. But at around 16 threads, an increase in threads will lower the speedup. This pattern can be seen in both *Figure 1* and *Figure 2.* At around 16 threads the overhead of creating new threads exceeds the benefits from dividing the vector further. *T*he bucket algorithm performance depends heavily on the load balance. The function drand48() was used for randomizing the values to be sorted and getting a good spread for the load balance. One big performance problem when using divide and conquer is the choice of the pivot element. If the spread is not

uniform, a bad choice of pivot can lead to a poor performance. An alternative strategy could have been choosing the pivot element as the mean value of the vector.

4. Appendix

Attached files

- quick_sort.c
- BUCKET_quick_sort.c
- DAC_quick_sort.c
- speedupVStime.m