Uppsala University
Department of Information Technology
Scientific Computing

Programming of Parallel Computers
2016-02-02

Compulsory Assignment 2

Shared memory parallelization using Pthreads

1 Quick-Sort

The serial Quick-sort algorithm is described numerously in the literature and briefly the algorithm follows as:

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

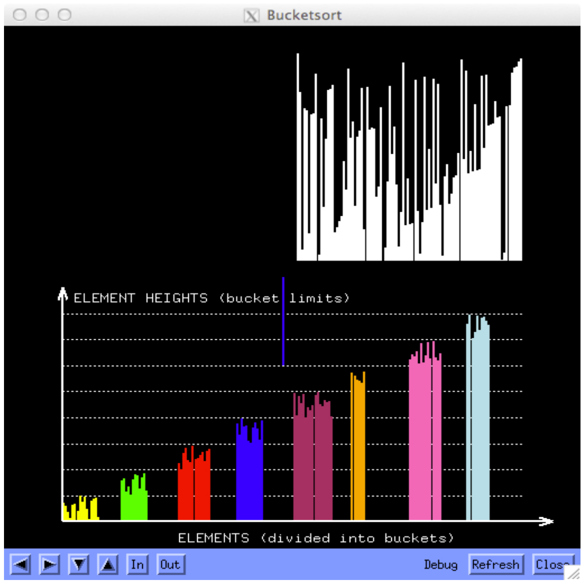
Start by implementing a serial version of this algorithm following the notes, e.g., in Wikipedia [**http://en.wikipedia.org/wiki/Quicksort**] using the in-place version. The algorithm can then be parallelized in many ways but we will only study and implement two inherently different ways to do this, a *divide-and-conquer algorithm* and a *peer algorithm*.

1.1 Divide and conquer parallelization

A straightforward parallelization of Quick-Sort is to acquire a new thread for each recursion step and sort the two lists in parallel on the two threads. One problem is then to not acquire to many threads (or too few threads). After a number of recursion levels one can proceed without requiring new threads using the serial Quick-sort algorithm on the respective threads. How does the number of threads affect the performance and in what way? What are the performance obstacles?

1.2 Peer parallelization

Here the elements are first filtered into buckets (i.e. elements within range $[a_i \; b_i]$ belong to bucket i). Then the buckets can be sorted concurrently and independently in different threads, e.g., using the quicksort algorithm. See Figure 1 below.


Bucketsort
ELEMENT HEIGHTS (bucket limits)
ELEMENTS (divided into buckets)
In
Out
Debug
Refresh
Close

Figur 1: Peer parallelization

The algorithm is straightforward to parallelize as the sorting of the buckets is trivially parallel. The problem becomes to get a good load balance among the threads. Think of different strategies to load balance the computations and implement one suitable strategy.

2 Experiments

Sort double precision random number sequences up to 100 000 000 elements and measure the speedup using your parallel implementation. To get a good random number sequence use the function drand48(). Measure speedup in your implementations, compare the algorithms, and analyze/explain your results discussing the advantages and disadvantages of the respective algorithm.

3 Writing a report on the results

No formal report is required but present your experiments with relevant figures, tables, reflections and explaining text, i.e., submit a short informal report including your source codes in **one PDF file**. Submit also the source codes as c-files so that we can compile and run them if necessary.

The last day to hand in the report is **February 29, 2016** but try to do it as soon as possible.