Parallel Programming 2016, Assignment 3: Parallel Sorting Algorithm

Deadline: Wednesday, May 25 before 23:59 hours.

This assignment concerns the implementation of a parallel sorting algorithm. The (starting) sequential sorting algorithm which should be used is not specified and can be chosen to be any sorting algorithm you can think of. The final implementation of the parallelized sorting algorithm should be run on the DAS-4 cluster in Leiden. The data set to be sorted will be generated at run-time on the separate compute nodes.

The deadline for this assignment is May 25. The assignment has to be completed individually. We expect you to submit a tarball containing your source code and a report (PDF format) which describes your parallel sorting algorithm and the benchmark results for each combination of the data sets and the nodes. The assignments can be handed in by e-mail to *krietvel (at) liacs (dot) nl*, please ensure the e-mail subject equals *Parallel Programming Assignment 3*.

1 Implementation

Your algorithm needs to be implemented in the C/C++ language, using MPI for interprocess communication. The target platform is the DAS-4 cluster. Also for this assignment, parallelization is *only* to be done by distributing the work over different nodes. So, we assume that only a single process, without threading, is executed on each node of the DAS-4 cluster.

The data to be sorted consists of 32-bit integers, which will be generated at run-time on the separate compute nodes. By using a pseudo-random number generator and a fixed seed, we can ensure that the same data is generated for subsequent runs, which facilitates debugging and makes for a fair performance comparison. The following code fragment sets a random seed and fills an array with pseudo-random numbers:

```
#include <stdlib.h>

#define N 200000UL
#define BASE_SEED 0x1234abcd

...

  int *my_array = malloc(sizeof(int) * N);
  int rank = 0;

  /* Initialize the random number generator for the given BASE_SEED
   * plus an offset for the MPI rank of the node, such that on every
   * node different numbers are generated.
   */
  srand(BASE_SEED+rank);

  /* Generate N pseudo-random integers in the interval [0, RAND_MAX] */
  for (size_t i = 0; i < N; i++)
    my_array[i] = rand();

...

  free(my_array);
```

A quick experiment has shown that a DAS-4 node can generate between 70 and 80 million 32-bit integers per second (single-threaded). 37 GB of numbers can be generated in little over 2 minutes.

The result of the sorting algorithm, the ordered array, should not be printed in its entirety. Instead, only print the numbers at subscripts $0, 10000, 20000, 30000, \ldots$ to the standard output. Next to that, print the time it took the algorithm to perform the sorting (*important:* exclude the time it took to generate the arrays with random numbers).

2 Benchmarking

You should perform experiments with 1, 2, 4, 8 and 16 nodes on the DAS-4 cluster located in Leiden. Each experiment should be repeated five times, with five different data sets of the same size. The run times that are presented in the report should be averages of these five runs.

The five different data sets will be generated by choosing five different (base) seeds to initialize the random number generator. The following seeds should be used:

$$0x1234abcd, 0x10203040, 0x40e8c724, 0x79cbba1d, 0xac7bd459$$

The total sizes of the data sets should be:

$$
\begin{aligned}
200.000 \quad & \text{32-bit integers} \\
1.600.000 \quad & \text{32-bit integers} \\
80.000.000 \quad & \text{32-bit integers} \\
16.000.000.000 \quad & \text{32-bit integers}
\end{aligned}
$$

resulting in a data set size per node (N) with P (1, 2, 4, 8 or 16) nodes of:

$$
\begin{aligned}
200.000/P \text{ 32-bit integers} : & \ (200.000, 100.000, 50.000, 25.000, 12.500) \\
1.600.000/P \text{ 32-bit integers} : & \ (1.600.000, 800.000, 400.000, 200.000, 100.000) \\
80.000.000/P \text{ 32-bit integers} : & \ (80.000.000, 40.000.000, 20.000.000, 10.000.000, 5.000.000) \\
16.000.000.000/P \text{ 32-bit integers} : & \ (16.000.000.000, 8.000.000.000, 4.000.000.000, \\
& \ 2.000.000.000, 1.000.000.000)
\end{aligned}
$$

Note that with the largest data set, performing experiments on a single node is really pushing the available swap space. Depending on the efficiency of your implementation, the largest data set might not run on a single node. In that case, for the largest data set you should start the benchmarks on 2 nodes, followed by 4, 8 and 16 nodes.