Parallel Programming 2020, Assignment 3
Parallel Sorting

Deadline: Friday, December 11 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. Two variations have to be designed, one in which sequential sorting runs on the CPU and one in which it runs on the GPU. The final implementations of the parallelized sorting algorithm should be run on the DAS-5 cluster located in the VU on the TitanX nodes. The data set to be sorted will be generated at run-time on the separate compute nodes.

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 your GPU variation in additional detail) and the benchmark results for each combination of the data sets, the nodes and the two implementation variations. The assignments can be handed in by e-mail to *hpc1-2020 (at) dvdzwaan (dot) com*.

1 Implementation

Your algorithm needs to be implemented in the C/C++ language, using MPI for interprocess communication. The target platform is the DAS-5 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-5 cluster.

A variation should also be designed which runs sorting workloads on each node using the GTX Titan X GPU available on each node, instead of using the CPU to do these local sorting workloads. You are expected to implement a tailored GPU sorting implementation yourself using CUDA.

The data to be sorted consists of 32-bit unsigned 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>
#include <random>

...
  /* N, base_seed will be passed as command line arguments */

  uint32_t* my_array = malloc(sizeof(uint32_t) * N);

  /* 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.
   */
  std::mt19937 generator(base_seed + mpi_rank);

  /* Generate N pseudo-random uint32_t's */
  std::uniform_int_distribution<uint32_t> distribution(0, std::numeric_limits<uint32_t>::max());
  for (size_t i = 0; i < N; i++)
    my_array[i] = distribution(generator);

...

  free(my_array);
```

It is suggested (but not required) to design your implementation to *either* first (re)distribute the generated numbers between all processes so that the process with rank x (out of P total processes) has all numbers from $\frac{x}{P} \times (\texttt{UINT32\_MAX} + 1)$ up to $\frac{x+1}{P} \times (\texttt{UINT32\_MAX} + 1)$, then locally sorting this range of assigned numbers on the node itself. *Or*, first sort the locally generated numbers on the node itself, then merge the sorted numbers of all nodes. This allows easily switching between the CPU and GPU implementations to locally sort the numbers in one of the stages.

2 Benchmarking

You should perform experiments with 1, 2, 4, 8 and 16 nodes on the DAS-5 cluster located in the VU. 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. All experiments should be performed for both the CPU and the GPU variation. Experiments should only be performed on the TitanX nodes (by means of `-C TitanX --gres=gpu:1` or `-native '-C TitanX --gres=gpu:1'` when using prun).

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:

$$\texttt{305441741}, \texttt{270544960}, \texttt{1088997156}, \texttt{2043394589}, \texttt{2893796441}$$

The total sizes of the data sets should be:

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

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

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

3 Input & output

Your submission is expected to include a *Makefile* which will build and output an executable called "`sort`" in the *same directory* as the Makefile.

In order to easily benchmark your application, the executable is expected to take exactly three arguments: first the (base) seed, then the number of integers in the data set and finally a number indicating if the sequential sorting should be accelerated on the GPU (0 or 1).

Your program is expected to output various numbers on stdout. First, the time taken to sort the integers in seconds (excluding the time to generate the random numbers):

```
fprintf(stdout, "%.20f\n", elapsed_time.count());
```

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:

```
...
  fprintf(stdout, "%u\n", ...);
...
```

Nothing else should be written to stdout. Any output on *stderr* is ignored.

It is recommended to use *prun* to easily test your application: `prun -np 2 -script $PRUN_ETC/prun-openmpi -native '-C TitanX --gres=gpu:1' `pwd`/sort 305441741 200000 0`. This will also be used to verify your submitted program works correctly. If you use a custom job script, mention this clearly in an additional *README* file.

4 Links

- DAS-5 website with information on job submission, starting MPI programs and using CUDA: `http://www.cs.vu.nl/das5/jobs.shtml`.
- CUDA introduction: `https://devblogs.nvidia.com/even-easier-introduction-cuda/`.