Parallel Programming 2017, Assignment 3

Deadline: Saturday, December 30 before 23:59 hours.

For the completion of assignment 3 a choice is given to either design a parallel version of the first assignment, i.e. Sparse LU factorization with Partial Pivoting or to develop a Parallel Sorting algorithm.

1 Parallel Implementation of Sparse LU Factorization

This task concerns the implementation of Parallel Sparse LU Factorization. After completing and testing your implementation, you will perform a thorough experimental evaluation of your implementation.

The assignment has to be completed individually. You are expected to hand in a tarball containing your source code and an extensive report in PDF format that describes your implementation and contains a thorough experimental evaluation. The assignments can be handed in by e-mail to *pp2017 (at) handin (dot) liacs (dot) nl*.

1.1 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.

You need to implement a Parallel Sparse LU Factorization kernel that applies *partial* pivoting. Parallelization must be carried out using the MPI framework. You can assume that only one process will be running per node, so within your parallelized program you do not have to account for multiple processes that execute within the same address space (threading). The program must be written in C/C++.

Describe in your report how you have parallelized the LU Factorization kernel and how you solved the problems you encountered.

1.2 Experimental Evaluation

The final part of the assignment is to perform a thorough experimental evaluation of your implementation. The target machine will be the DAS-4 cluster at Leiden University. To be benchmarked are the execution time required to factorize the system, the total time to compute all five solution vectors, and the relative errors $\frac{||\tilde{x}-x||}{||x||}$ for each solution. $||x||$ denotes the Euclidean Norm: $\sqrt{\sum x_i^2}$ and x is the actual solution and $\tilde{x}$ is the solution computed by your implementation of LU factorization. Think about *what* you want to show and *how* you need to show this: so, what experiments do you need to carry out. Examples are: performance / execution time, and amount of communication between nodes.

As test data, you can again use matrices from the University of Florida Sparse Matrix Collection (`http://www.cise.ufl.edu/research/sparse/matrices/`). We selected the following test matrices:

Simon/raefsky6
Garon/garon1
Grund/poli4
Schenk IBMNA/c-25
Bai/cryg10000

For the initialization of your program you need to construct five b vectors which are obtained by multiplying a given matrix A with an actual solution vector x (so, $b = Ax$). Take for x the following vectors:

- vector x consisting of all ones.
- vector x consisting of 0.1 for every element.
- vector x consisting of alternating $+1$, -1 values: $+1, -1, +1, -1, +1, -1, \ldots$.
- vector x consisting of alternating $+5$, -5 values.
- vector x consisting of alternating $+100$, -100 values.

Note that the initialization is not part of the actual computation, so does not need to be benchmarked. A standard matrix multiplication or an external tool like Python with Scipy may be used to compute the b vectors.

2 Parallel Sorting

This task 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 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 *pp2017 (at) handin (dot) liacs (dot) nl*.

2.1 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).

2.2 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:

200.000	32-bit integers
1.600.000	32-bit integers
80.000.000	32-bit integers
16.000.000.000	32-bit integers

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

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

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.