CSE 6220-A Introduction to High Performance Computing
Spring 2016
Programming Assignment 2
Due March 30 2016

Overview

In this programming assignment, you will implement parallel, distributed radix sort for sorting generic data types by an integer key. In the process, you will use and learn about MPI collective operations, reductions, and custom data types.

Radix sort

Least-significant-digit (LSD) radix sort is a linear time integer sorting algorithm, which sorts integer keys digit-by-digit, or byte-by-byte. Given a sequence of integer keys, we start with the least-significant digit and stably sort all keys by one digit at a time. Each iteration of sorting by a digit has to be stable, which means that those keys which share the same current digit, remain in the same order relative to one another. Figure 1 shows an example for sorting small 3-digit integers.

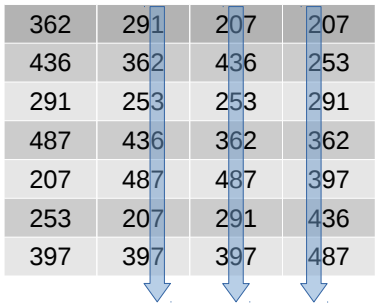

Figure 1: LSD radix sort example: sort first by last digit, then by middle digit, and finally by the first digit.

In this assignment, you will develop and implement parallel radix sort for sorting arbitrary types by an integer key. Instead of sorting by a single bit, we will sort by k bits at a time. For example, we could sort the keys one byte at a time, i.e. $k = 8$. We will denote each k bit block as a "digit". To simplify the assignment, we will assume that keys are of type `unsigned int`, a 32 bit integer, and assume that k is a power of 2.

The n input elements are distributed across all processors with $\frac{n}{p}$ elements on each processor. For this assignment, we will assume that n is a multiple of p.

In every iteration, each processor will first scan the current digit and create a histogram which counts the number of occurrences of each unique digit. Since each digit contains k bits, the histogram will have 2^k bins. Next we sort the local elements in a stable fashion by their current digit.

Note that since we know how many keys for each digit there are, we can simply compute the sorted position for each element via a local prefix sum over the histogram.

So far all operations are local to each processor. Next we have to achieve a global sorting by the current digit over all elements on all processors. Our goal is to use a single call to `MPI_Alltoallv` to communicate all elements to their target processors. To do so, we need to compute a `send_counts` array, which contains the number of elements to send to each processor. Since our local elements are already sorted, all elements going to one processor are contiguous. In order to compute how many elements to send to each processor, we will again scan the locally sorted sequence. For each local element e_i, we can determine its exact position in the global sorted order by three numbers:

1. $G(e_i)$: The overall total number of elements with their current digit smaller than e_i's current digit
2. $P(e_i)$: The total number of elements with the same digit but on processors with smaller rank
3. $L(e_i)$: The number of elements with the same digit on the same processor and left of this element

The global target index for e_i is then simply: $T(e_i) = G(e_i) + P(e_i) + L(e_i)$. Note that $G(e_i)$ and $P(e_i)$ depend on values from other processors, whereas we can simply keep a counter while iterating the local elements to determine $L(e_i)$. $G(e_i)$ and $P(e_i)$ can be computed via `MPI_Exscan` and `MPI_Reduce` over the histograms we computed before.

Knowing the global target index for an element allows you to determine which processor to send each element to, thus computing the required `send_counts`. Once the `MPI_Alltoallv` communication is completed, we need to locally sort by the current digit again to finalize the global sorting by this digit.

After a total of $\frac{32}{k}$ iterations, all elements will be sorted.

Tasks

Code Framework

You are required to implement your solution within the provided framework hosted on Georgia Tech's Enterprise GitHub page at: github.gatech.edu/pflick3/sp16-cse6220-prog2. In the event that we have to update the framework, we will publish the updates in this repository and notify the class via T-Square. The framework comes with some pre-implemented unit tests. You can run `make && make test` to execute these automated test cases. This will help you identify problems with your implementation. These tests are implemented in the file `mpi_tests.cpp`. The test cases we provide only cover some possible inputs. You are encouraged to add more test cases in this file to make sure that your implementation of radix sort works as expected.

Implement Custom Datatypes

The files `mystruct.h` and `mystruct.cpp` contain a declaration of a structure `MyStruct`. We want to sort these structures via our radix sort implementation and need to communicate elements of the type `struct MyStruct` via MPI. For this you need to create a custom data type in MPI. Implement the function `mystruct_get_mpi_type()` inside `mystruct.cpp` to return the `MPI_Datatype` for `MyStruct`.

Implement radix sort

Implement the radix sort function inside the header file `radix_sort.h`. This function takes the input range as two pointers: `begin` and `end`. The `begin` pointer points to the first element of the range, and the `end` pointer points to the element after the last element of the range, such that the overall number of elements is equal to `n = end - begin`. Furthermore, the `radix_sort` function takes a function pointer as input using `key_func` parameter. This function returns the `unsigned int` key for each element. Next, the `radix_sort` function takes two MPI parameters: the `MPI_Datatype` corresponding to the input type and the `MPI_Comm` communicator, which has to be used for all MPI communication within the `radix_sort` function. The final parameter `k` gives the number of bits to sort by in each iteration. For a more detailed API description, refer to the documentation inside `radix_sort.h`.

Experiments

Evaluate the performance of your program. For instance, fix a problem size, vary the number of processors, and plot the runtime as a function of the number of processors. Evaluate the influence of the value for k on the runtime both theoretically and experimentally. Fix the number of processors and see what happens as the problem size is varied. Try to come up with some important observations on your program. For example, what is the minimum problem size per processor to get good parallel efficiency? We are not asking that specific questions be answered but you are expected to think on your own, experiment, and show the conclusions along with the evidence that led you to reach the conclusions.

Optimize the performance

This task goes hand in hand with the experiments, as you will run experiments and change your implementation based on the runtime behavior you observe. A copy of the files `mystruct.h` and `mystruct.cpp` are provided inside the framework: `mystruct_opt.h` and `mystruct_opt.cpp`. Investigate how you can change the struct in `mystruct_opt.h` to achieve a better MPI performance. You can use the two separate executables `sort-mystruct` and `sort-mystruct-opt` to directly compare the performance between the original struct implementation and your optimized version in `mystruct_opt.h/cpp`.

Submission Policies

You are expected to work in teams of two. No matter how you decide to share the work between yourselves, each student is expected to have full knowledge of the submitted program. One member from each team should submit a zip file (file name : assign_2.zip) containing the following on T-Square

1. A text file containing the names of all team members along with their contribution to the assignment (file name : names.txt)

2. Source files (`radix_sort.h`, `mystruct.cpp`, `mpi_tests.cpp`, `mystruct_opt.h/cpp`). You should use good programming style and comment your program so that it is easy to read and understand. Note that we will test your implementation by copying your `radix_sort.h` and `mystruct.cpp` into an otherwise clean framework, and execute our own test cases. Make sure that your implementation does not rely on any extra files or functions implemented outside of these files.

3. A PDF report containing the following (file name : report.pdf)

   - Short design description of your algorithm(s)
   - Evaluation of the performance of your program. Show the results of your experiments, and the conclusions along with the evidence that led you to reach the conclusions.
   - Explain any ideas or optimizations of your own that you might have applied and how they might have affected the performance. Explain how you optimized the structure in `mystruct_opt` and how it affected the overall performance.

The program will be graded on correctness and use of the appropriate parallel programming features.

Notes

We will enforce the specified polices strictly. You will risk losing points if you do not adhere to them. We will check all submissions for plagiarism. Any cases of plagiarism will be taken seriously, and dealt in accordance with the GT academic honor code (honor.gatech.edu).