TU WIEN Informatics Parallel Computing

Prof. Dr. Jesper Larsson Träff
Assoc.Prof. Dr. Sascha Hunold
TU Wien, Faculty of Informatics
Research Group for Parallel Computing

High Performance Computing

2025S
Assignment 1: Allgather-Merge

Issue date: 2025-04-08
Due date: 2024-06-03 (23:59)

Hand-in via TUWEL. No deadline extensions will be granted. The project can be done in groups of at most three (3) participants. One hand-in per group suffices. Group registration in TUWEL required. Mark clearly if, where, and how AI tools like ChatGPT were used (at best not at all!). State all other sources used, books, script, papers, etc., and cite correctly.

1. Introduction

This HPC project is an exercise in HPC algorithm engineering. The challenge is to gradually improve over a simple baseline that defines the problem to be solved, both by improving the implementation and the algorithm(s) used. The achieved improvements are to be documented experimentally by comparison to the baseline.

The problem is to implement the so called allgather-merge collective operation. In this problem, each processor contributes a buffer of elements in increasing order (say, integers, with the standard $\leq$ order), and computes a buffer with all the elements from all the processes in increasing order. Preferably, no duplicate (same key) elimination is to be performed, *but* no duplicates of input elements are allowed to be created. The allgather-merge was, for instance, used as building block in certain sorting algorithms [1].

The obvious, baseline implementation is to use the MPI allgather collective (`MPI_Allgather`) to collect all ordered blocks at all processes, and then sequentially, per process, merge the p ordered blocks together using a good p-way merge implementation. This will not create any duplicates, but the p-way merge can be implemented to eliminate duplicates, if so desired (not here!). Here, p is the number of processes participating in the operation. The challenge of the project is to improve over this baseline. Anything goes: The algorithms and implementations for the sequential two-way or p-way merge operations can be improved, the number of process local copy operations can be reduced, SIMD-vectorization can be employed, the algorithm for gathering blocks can be improved and combined with the merge steps, etc., etc. You may or may not use in-place merging to be space efficient.

The implementations are to be benchmarked with given inputs. For a fixed number of MPI processes, the size of the ordered blocks per process are increased. Different types of inputs are to be used for testing and benchmarking (disjoint inputs, same inputs). The inputs are generated by the template code supplied with the project (in TUWEL), and this framework has to be used.

2. The Problem

The integer allgather-merge problem is the following. Given p input blocks, one for each of the p processes, each with the same number m/p of integer elements, each block in sorted order, for a total of m input elements. The output for each process is a single block of the m input elements but in sorted order. Duplicates should not be removed, and no duplicates introduced.

Algorithm 1 Algorithm for the allgather operation for processor $r \in C_p^s$ on the circulant graph C_p^s with straight doubling skips $s_k = 2^k, 0 \leq k < q$ and $s_q = p$, where $q = \lceil \log_2 p \rceil$. The result data blocks at each processor are indexed as $R[i], i = 0, 1, \ldots, r, \ldots p-1$. The block that processor r has initially is $R[r]$. An auxiliary array B is used to gradually build up a sequence of resulting blocks.

procedure ALLGATHER($R[p], r \in C_p^s$)
  $q \leftarrow \lceil \log_2 p \rceil$
  $B[0] \leftarrow R[r]$
  for $k = 0, \ldots, q-1$ **do**
    $t, f \leftarrow (r - s_k + p) \bmod p, (r + s_k) \bmod p$ ▷ To- and from-processors
    $\mathsf{Send}(B[0 \ldots s_{k+1} - s_k - 1], t, C_p^s) \parallel \mathsf{Recv}(B[s_k \ldots s_{k+1} - 1], f, C_p^s)$
  end for
  for all $1 \leq i < p$ **do** $R[(r+i) \bmod p] \leftarrow B[i]$
  end for
end procedure

Algorithm 1 and Algorithm 2 recall two useful algorithms from the lecture that can be used as inspiration. In Algorithm 2, $\oplus$ is an associative, commutative vector operator which you may somehow replace with a merge-operation.

3. The Concrete Tasks

The project has three concrete implementation tasks and a benchmarking task. The implemented allgather-merge operations must all have signature

```
int HPC_AllgatherMergeX(int *sendbuf, const int sendcount, MPI_Datatype sendtype,
                        int *recvbuf, const int recvcount, MPI_Datatype recvtype,
                        MPI_Comm comm);
```

similar to the `MPI_Allgather` operation, where `X` can be used to designate the particular variant, e.g., the baseline (`Base`), first optimization (`Bruck`), second optimization (`Circulant`), etc. Note that the signature takes also datatype arguments; however, the functions are supposed to work for the type `MPI_INT` only. The `sendcount` and `recvcount` arguments denote the number of integers in one block, that is m/p for the total problem size m and p MPI processes.

Exercise 1 (10 points) [Baseline Implementation]

Implement the baseline for solving the problem consisting in an `MPI_Allgather` call followed by sequential, p-way merging by each process. Do this in the best possible way you can think of by for instance using the `MPI_IN_PLACE` argument to the `MPI_Allgather` call (which may or may not

Algorithm 2 Algorithm for the allreduce operation for processor $r \in C_p^s$ on the circulant graph C_p^s with adjusted roughly halving skips $s_k, k = 0, 1, \ldots, q-1$ and $s_q = p$ and $s_k = \lceil s_{k+1}/2 \rceil$, where $q = \lceil \log_2 p \rceil$. The input vector for each processor is V and each processor computes the reduction of all input vectors into W. The associative and commutative operator is $\oplus$. The invariant is that after round k, $W = \oplus_{i=1}^{s_{k+1}} V_{(r+i) \bmod p}$ for each processor r. The algorithm exploits the commutativity of $\oplus$.

procedure ALLREDUCE($V, W, \oplus, r \in C_p^s$)
  if $p = 1$ **then** ▷ For good measure, handle $p = 1$ case
    $W \leftarrow V$ **return**
  end if
  for $k = 0, \ldots, q-1$ **do**
    $\varepsilon \leftarrow s_{k+1} \wedge \texttt{0x1}$ ▷ Adjustment term $\varepsilon = 1$ if s_{k+1} is odd, $\varepsilon = 0$ if s_{k+1} is even
    $t, f \leftarrow (r - s_k + \varepsilon + p) \bmod p, (r + s_k - \varepsilon) \bmod p$ ▷ To- and from-processors, adjusted
    if $\varepsilon = 1$ **then** ▷ s_{k+1} odd
      $\mathsf{Send}(W, t, C_p^s) \parallel \mathsf{Recv}(T, f, C_p^s)$
      $W \leftarrow W \oplus T$
    else ▷ s_{k+1} even
      if $k = 0$ **then**
        $\mathsf{Send}(V, t, C_p^s) \parallel \mathsf{Recv}(W, f, C_p^s)$
      else
        $W' \leftarrow V \oplus W$
        $\mathsf{Send}(W', t, C_p^s) \parallel \mathsf{Recv}(T, f, C_p^s)$
        $W \leftarrow W \oplus T$
      end if
    end if
  end for
  $W \leftarrow V \oplus W$
end procedure

make a difference) and in particular by doing the p-way merge well. You may implement your own p-way merge or pick a library implementation from somewhere (state where in this case).

1. Explain briefly your implementation and the choices you made.
2. Explain and state the asymptotic complexity as $O(f(m,p))$ in terms of m, the total size of the output per process, and p, the number of processes, of your implementation, assuming that `MPI_Allgather` can (as explained in the lectures) be done efficiently in $O(m + \log p)$ operations with small constants on the number of send and receive operations.
3. You may do a refined analysis in terms of the number of memory copies of arrays, incurred two-way merge operations, or other significant parameters.

Exercise 2 (20 points) [Implementation with a Standard Allgather Algorithm]

Improve over this baseline by doing the merging in a distributed way as part of a modified allgather-like algorithm. The idea is to try to use the Bruck straight doubling circulant graph algorithm shown as Algorithm 1 (this will be described in the lectures) as blueprint. This idea has a certain drawback that you are asked to explain.

Concretely, try to modify Algorithm 1 with the necessary merge steps after each communication round. Instead of the array R storing the blocks, maintain an array M of merged elements received so far. In each round, except the last, $s_{k+1} - s_k = 2^k$ elements (in order) are sent and received. The difficulty lies in getting the last round correct (if p is not a power of two). Also, be careful with not doing too many process local copies back and forth of merged array blocks. Be careful also with allocated extra space, which should be in $O(m)$. The task is to explore whether (and under what restrictions) such a solution is possible.

1. Explain briefly your implementation and the choices you made, and give the modified algorithm in pseudo-code. You may use a three-argument merge-subroutine defined to your liking (two ordered input arrays, one to be ordered output array). Show clearly where process local copying is needed (copying one array into another). Describe obstacles and possible solutions. Some creativity is required for this task.
2. Explain and state the asymptotic complexity as $O(f(m,p))$ in terms of m, the total size of the output per process, and p, the number of processes, of your implementation, assuming that send and receive operations with m data can be accounted for as $\alpha + \beta m$. Does it differ from the baseline solution? In which respects?
3. You may do a refined analysis in terms of the number of memory copies of arrays, incurred two-way merge operations, total amount of data merged, number of communication rounds, or other significant parameters.

Exercise 3 (20 points) [Implementation with a Specialized Allreduce Algorithm]

The final approach is based on a slightly modified algorithm for the `MPI_Allreduce` operation (will also be explained in the lecture, see Algorithm 2), which alleviates the drawback of the allgather based solution (Algorithm 2 can be written even more compactly, this may be of help).

Table 1: Performance of algorithms (execution time in microseconds). Configuration with $p = N \times n$ MPI processes, where N is the number of nodes and n the number of processes (tasks) per node.

Algorithm	**Type**	**Message size (number of elements)**					
		1	**10**	**100**	**1000**	**10000**	**100000**
Baseline	0	2.00	1.40	5.60	3.82×10^1	3.61×10^2	3.67×10^3
	1	1.00	2.70	5.00	3.69×10^1	2.92×10^2	2.67×10^3
	2	9.00×10^{-1}	1.40	6.40	2.86×10^1	2.79×10^2	2.70×10^3
Algorithm 1 based	0	1.80	1.40	5.00	4.10×10^1	3.61×10^2	3.61×10^3
	1	1.00	1.30	4.70	2.89×10^1	2.49×10^2	2.63×10^3
	2	2.90	1.00	4.40	2.70×10^1	2.47×10^2	2.62×10^3
Algorithm 2 based	0	1.40	1.20	4.40	1.81×10^1	1.49×10^2	1.81×10^3
	1	1.00	1.20	4.80	2.54×10^1	2.83×10^2	2.54×10^3
	2	9.00×10^{-1}	3.00	3.90	2.45×10^1	3.77×10^2	2.57×10^3

Concretely, modify Algorithm 2 by replacing the $\oplus$ operations with two-way merge operations into an array M of merged elements received so far. Give the modified algorithm in pseudo-code. You may use any three-argument merge-subroutine defined to your liking. Show clearly where process local copying is needed (copying one array into another). Explain and state the asymptotic complexity and account for the number of merge steps and array copies in your explanation.

1. Explain briefly your implementation and the choices you made, and give the modified algorithm in pseudo-code.

2. Explain and state the asymptotic complexity as $O(f(m,p))$ in terms of m, the total size of the output per process, and p, the number of processes, of your implementation, assuming that send and receive operations with m data can be accounted for as $\alpha + \beta m$. In which respects does your new algorithm improve on the baseline and the other algorithm?

3. You may do a refined analysis in terms of the number of memory copies of arrays, incurred two-way merge operations, total amount of data merged, number of communication rounds, or other significant parameters.

Exercise 4 (20 points) [Benchmark results]

Benchmark your three implementations with input sizes (block size per process, `recvcount` = `sendcount`) $m/p = 1, 10, 100, 1000, 10000, 100000, 1000000$ (7 inputs), three problem types for each input size, and MPI process configurations as listed below. Fill in Table 1 by running the benchmark for each of your implementations. This part of the report will have 9 such tables, one for each configuration, with a column for each input size, and run times for each of your three implementations with the three input configurations. Comment on the results, in particular expected (or unexpected) improvements. You may, at your own discretion, provide additional plots or otherwise analyze the data.

4. Programming

The three versions should be incorporated in the boiler plate code supplied. You can program with pure `C` or mild `C++` (which may be convenient). This boiler plate code also has a simple (but expensive) result correctness checker based on `MPI_Allgather` and sorting. The boiler plate code

contains a (simplistic) benchmarking framework (which you should use). It can also generate inputs of the three different types that you are asked to benchmark with. You may find it helpful to work with other input types during debugging and testing.

5. Testing and Benchmarking

All algorithms should be benchmarked on the `hydra` system.

The following input and MPI process configurations and have to be used. The run times should be averages of at least 10 repetitions. Further statistical analysis is optional. These experiments must be conducted on the `hydra` system!

1. Single node process configurations $p = 1 \times 1$, $p = 1 \times 10$, $p = 1 \times 32$. Multi-node configurations $p = 10 \times 1$, $p = 20 \times 1$, $p = 10 \times 10$, $p = 20 \times 10$, and $p = 10 \times 32$, $p = 20 \times 32$. The notation $p = N \times n$ denotes p processes distributed over N nodes each with n processes (tasks).
2. Weak scaling only: Inputs $m/p = 1, 10, 100, 1000, 10000, 100000, 1000000$.
3. - Input type 1: Process r has only r elements, $r, r, r, \ldots$.
   - Input type 2: Process r has elements $0, 1, 2, \ldots, m/p - 1$ (independent of r).
   - Input type 2: Process r has $r, p + r, 2p + r, 3p + r, \ldots$, elements.

Requirements:

What to hand in?

Your hand-in will consist of your report and the code, including cmake file needed to compile and run the code. This has to be packed as a `zip`-file, and uploaded via TUWEL.

- A short report, explaining briefly algorithms and implementations, especially what you did on your own (improvements, reduction of memory copies, etc.). Explain and state asymptotic complexities.
- Experimental results in tabular form as explained, optional, additional plots, as part of the report. A summary comparison between the different implementations (baseline vs. the distributed merging improvements) and discussion is expected (part of the report).
- Perform at least 10 repeated measurements and report the average running time; further analysis is optional.
- C/C++ code. cmake file. Code is uploaded separately to TUWEL.

A. How to Run on hydra

1. First, you will have to log into hydra.

```
ssh hydra
```

2. Once you are on hydra, you will have to load the MPI library

```
spack load openmpi@4.1.5
```

3. In order to run your application, you have two options: 1) use `sbatch` to submit a job to the queue or 2) use `srun` to run the application directly if enough compute nodes are free.

B. A Step-by-step Example for Running an MPI program on hydra

You may have an MPI program like this on you local machine in a file called `mpi1.c`:

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[]){
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Hello␣from␣process␣%d␣of␣%d\n", rank, size);

    MPI_Finalize();

    return 0;
}
```

We now copy the file to hydra using `scp` or `sftp` (`hydra` is the name of the machine used in your `.ssh/config` file):

```
scp mpi1.c hydra:~/
```

Then, we log on to hydra:

```
ssh hydra
```

Load the MPI library and compile program:

```
spack load openmpi@4.1.5
mpicc -o mpi1 -O3 mpi1.c
```

Now, run using srun:

```
stester@hydra-head:~$ srun -t 5 -p q_student -N 4 --ntasks-per-node=3 ./mpi1
Hello from process 0 of 12
Hello from process 2 of 12
Hello from process 6 of 12
Hello from process 7 of 12
Hello from process 10 of 12
```

```
Hello from process 11 of 12
Hello from process 3 of 12
Hello from process 1 of 12
Hello from process 4 of 12
Hello from process 5 of 12
Hello from process 8 of 12
Hello from process 9 of 12
```

The parameter -N denote the number of compute nodes requested, where on each compute node –ntasks-per-node processes will be created. Thus, in total, we create 12 processes.

Often, it is better to submit a job to the queuing system and wait for the result. Once the jobs has been submitted, you can log out and come back at a later point in time to collect the results. To do so, you will have to write a batch script (I give the file the name `job1.sh`):

```
#! /bin/bash

#SBATCH -p q_student
#SBATCH -N 2   # how many compute node
#SBATCH --ntasks-per-node=4 # create 4 processes per compute node
#SBATCH --cpu-freq=High
#SBATCH --time=5:00  # max time a job can run before it is killed

srun ./mpi1
```

You can now submit this job and check whether it is running:

```
stester@hydra-head:~$ sbatch job1.sh
Submitted batch job 4330052
stester@hydra-head:~$ squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
4330052 q_student  job1.sh  stester  R       0:01      2 hydra[01-02]
```

With squeue, you will see the status of your job(s). In this case, the job is already running and has gotten ID 4330052. After the job has completed, there will be an output file with the result.

```
stester@hydra-head:~$ cat slurm-4330052.out
Hello from process 5 of 8
Hello from process 1 of 8
Hello from process 2 of 8
Hello from process 3 of 8
Hello from process 0 of 8
Hello from process 4 of 8
Hello from process 6 of 8
Hello from process 7 of 8
```

References

[1] Michael Axtmann and Peter Sanders. Robust massively parallel sorting. In *19th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 83–97, 2017.