МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

Направление подготовки: «Программная инженерия»

**ОТЧЁТ**

по лабораторной работе

**РЕАЛИЗАЦИЯ АЛГОРИТМА БЫСТРОЙ СОРТИРОВКИ С ПРОСТЫМ СЛИЯНИЕМ**

**Выполнила:** студентка группы

382008-1

\_\_\_А.Д. Коробейникова \_\_\_\_\_\_\_

Подпись

**Проверил:** м.н.с.

\_\_\_А.Ю. Нестеров\_\_\_\_\_\_\_\_

Подпись

Нижний Новгород  
2022 г.

## Содержание

[1. Введение 3](#_Toc27873363)

[2. Постановка задачи](#_Toc27873364) 4

[3. Описание алгоритмА](#_Toc27873365) 5

[4. Описание СХЕМЫ РАСПАРАЛЛЕЛИВАНИЯ](#_Toc27873374) 7

[5. Описание MPI-версии](#_Toc27873374) 9

[6. Результаты ЭКСПЕРИМЕНТОВ](#_Toc27873375) 12

[7. ВЫВОДЫ ИЗ РЕЗУЛЬТАТОВ](#_Toc27873374) 13

[8. Заключение 1](#_Toc27873376)4

[9. Литература](#_Toc27873377) 15

[10. Приложение](#_Toc27873378) 16

1. Введение

Message Passing Interface (MPI, интерфейс передачи сообщений) — программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. MPI является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании. Основным средством коммуникации между процессами в MPI является передача сообщений друг другу. В этой работе используется реализация MPI для языка программирования C++.

## **Постановка задачи**

Необходимо реализовать алгоритм быстрой сортировки с простым слиянием на языке C++, используя MPI. Данные для сортировки хранятся в одномерном массиве. Результатом работы является программа, которая должна корректно параллельно выполняться на нескольких процессах, а также набор тестов (не менее пяти), проверяющих работу этой программы. В процессе выполнения лабораторной работы требуется использовать систему контроля версий [Git][git] и фрэймворк для разработки автоматических тестов [Google Test][gtest].

Выполнение работы предполагает решение следующих задач:

* реализация алгоритма сортировки, который будут выполнять процессы,
  + реализация алгоритма планирования для распределения данных между процессами,
* разработка тестов для проверки работоспособности алгоритма,
* грамотная работа с системой контроля версий, выполненная по инструкции из репозитория. Результатом должны быть четыре файла: CMakeLists.txt, main.cpp, quick\_merge\_sort.h и quick\_merge\_sort.cpp.

## **Описание алгоритма**

Алгоритм работы программы:

1. Формирование данных для сортировки.
2. Распределение данных между всеми процессами (для этого - реализовать алгоритм планирования).
3. Работа каждого отдельного процесса:
   1. сортировка данных алгоритмом быстрой сортировки, полученных при распределении между процессами;
   2. отправка данных процессу, который занимается приёмом и последующим слиянием данных.
4. Выполнение сортировки данных на одном процессе.
5. Сравнение результатов работы параллельной сортировки данных с результатом работы последовательной сортировки.

Алгоритм планирования занимается подсчётом размера данных (количество элементов массива), которые достаются каждому отдельному процессу. Пусть имеется n элементов и m процессов, среди которых нужно распределить эти элементы, а количество этих элементов будет записываться в одномерный массив с названием elements\_per\_proc размером m, каждый номер ячейки которого совпадает с номером процесса. Алгоритм на псевдокоде:

remained\_elements = n mod m;

цикл от proc\_number := 0 до m [шаг 1]

elements\_per\_proc[proc\_number] := n div m;

если (remained\_elements > 0)

то

elements\_per\_proc[proc\_number] := elements\_per\_proc[proc\_number] + 1;

remained\_elements := remained\_elements - 1;

всё-если

всё-цикл

В результате работы этого алгоритма массив elements\_per\_proc будет содержать количество элементов, которое нужно будет отсортировать процессу с номером таким же, что и номер ячейки этого массива.

Описание алгоритма быстрой сортировки здесь приведено не будет, так же, как и алгоритма слияния, так как я воспользовалась стандартной реализацией std::sort() из STL и std::merge() из <algorithm>. std::sort() реализует быструю сортировку, а std::merge() - выполняет слияние двух отсортированных массивов.

1. **Описание схемы распараллеливания**

Рисунок 1 содержит условную схему распараллеливания. Эта схема описывает работу функции parallelSort(), которая принимает на вход данные для сортировки и возвращает их копию, отсортированную в порядке возрастания. Стоит упомянуть, что каждый процесс, кроме процесса с номером 0, возвращает лишь часть данных, которая досталась ему в результате распределения. Процесс с номером 0 возвращает полный объём данных, переданных в функцию (так как именно этот процесс занимается распределением и сбором данных). Следовательно, в вызывающем коде решением задачи будет значение parallelSort(), который вернётся в результате выполнения этой функции на нулевом процессе.

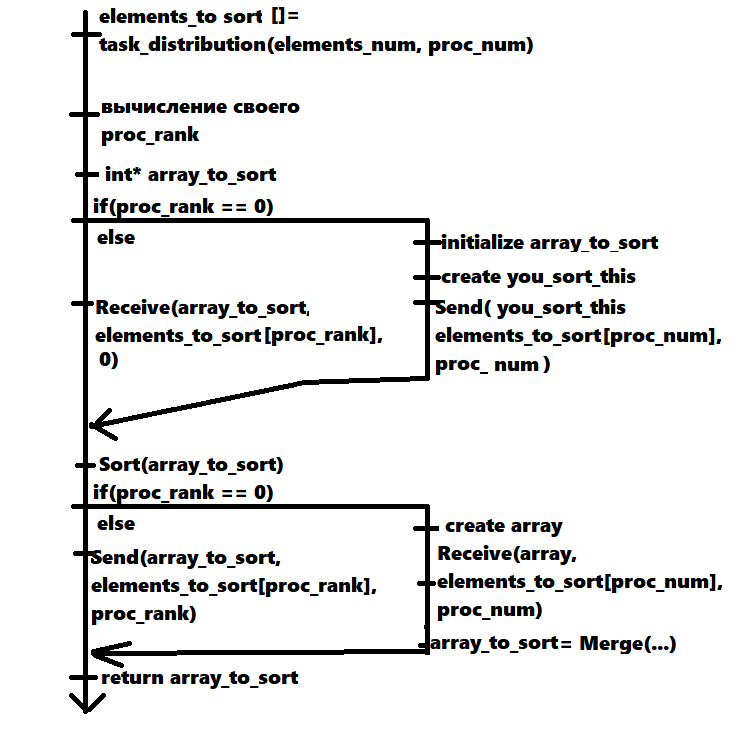


Рис. 1

Шаг 1: заполнение массива elements\_to\_sort с количеством элемента для каждого процесса.

Шаг 2: вызов функции, определяющей номер текущего процесса.

Шаг 3: создание пустого массива array\_to\_sort, который позже будет заполнен.

Шаг 4 (если текущий процесс имеет номер 0): процесс инициализирует свой массив для сортировки array\_to\_sort из исходного (берёт столько элементов, сколько написано в elements\_to\_sort[0]); затем в цикле формирует для каждого процесса proc\_num (кроме 0) массив, который нужно сортировать ему, и отправляет каждому процессу его массив.

Шаг 4 (если текущий процесс имеет номер не 0): процесс принимает от нулевого процесса данные в array\_to\_sort данные размером elements\_to\_sort[proc\_rank].

Шаг 5: сортировка данных.

Шаг 6 (если текущий процесс имеет номер 0): процесс в цикле создаёт временный буфер для принимаемых данных; принимает в этот буфер данные от каждого процесса; сливает все пришедшие и собственные отсортированные данные в один массив; присваивает новое значение array\_to\_sort.

Шаг 6 (если текущий процесс имеет номер не 0): процесс отправляет свои отсортированные данные процессу 0.

Шаг 7: функция возвращает значение array\_to\_sort.

1. **Описание MPI-версии**

Получение количества процессов: int MPI\_Comm\_size(MPI\_Comm comm, int \*size), где comm - коммуникатор, size - переменная, куда запишется количество процессов. Я использую глобальный коммуникатор MPI\_COMM\_WORLD.

Получение номера процесса: int MPI\_Comm\_rank(MPI\_Comm comm, int \*rank), где comm - коммуникатор, rank - переменная, куда запишется номера процесса.

Отправка данных с помощью MPI может выполняться функцией MPI\_Send(). Это блокирующая операция отправки данных одному конкретному процессу. Описание функции: int MPI\_Send(const void \*buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm), где

* buf - адрес буфера отправки
* count - количество элементов из буфера отправки для передачи
* datatype - тип посылаемых элементов (специальный тип MPI\_Datatype)
* dest - номер процесса-получателя
* tag - тэг сообщения (если отсутствует, равен 0)
* comm - коммуникатор, в котором находится процесс получатель

Отправка данных на шаге 4 из нулевого процесса осуществляется в цикле по всем номерам процессов (начиная с 1) таким вызовом MPI\_Send:

MPI\_Send(&global\_array[elems\_to\_skip], elements\_to\_sort[proc\_num], MPI\_INT, proc\_num, 0, MPI\_COMM\_WORLD);

* global\_array - массив данных, пришедших на вход функции parallelSort()
* elems\_to\_skip - сдвиг по массиву, который осуществляется после посылки предыдущему процессу. Это значение вычисляется на каждой итерации цикла по формуле:
* elements\_to\_sort[proc\_num] - ячейка массива elements\_to\_sort, содержащая количество элементов, который должен отсортировать proc\_num
* proc\_num - переменная, по которой итерируется цикл

Отправка данных на шаге 6 из ненулевого процесса: MPI\_Send(array\_to\_sort, elements\_to\_sort[proc\_rank], MPI\_INT, 0, 0, MPI\_COMM\_WORLD).

Получение данных в MPI осуществляется через функцию MPI\_Recv(). Это блокирующая операция приёма данных от одного процесса. Описание функции: int MPI\_Recv(void \*buf, int count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm comm, MPI\_Status \*status), где

* buf - адрес буфера, в который будут записаны полученные данные
* count - количество элементов, которое должно быть принято
* datatype - тип получаемых элементов (специальный тип MPI\_Datatype)
* source - номер процесса-отправителя
* tag - тэг сообщения (если отсутствует, равен 0)
* comm - коммуникатор, в котором находится процесс отправитель
* status - переменная специального типа, в которую будет записана информация о прохождении вызова (успех операции и др.)

Получение данных на шаге 4 в ненулевом процессе: MPI\_Recv(array\_to\_sort, elements\_to\_sort[rank], MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &status).

Получение данных на шаге 6 в нулевом процессе осуществляется в цикле по всем номерам процессов (начиная с 1): MPI\_Recv(ptr\_queue.back()->data(), elements\_to\_sort[proc\_num], MPI\_INT, proc, 0, MPI\_COMM\_WORLD, &status), где

* ptr\_queue - очередь std::queue, содержащая указатели на массивы. В эти массивы записываются пришедшие данные
* proc\_num - переменная, по которой итерируется цикл

1. **Результаты экспериментов**

С помощью MPI\_Wtime() можно засечь время выполнения.

Я тестирую работу своей программы на следующих данных: количество процессов меняется от 1 до 8; процессы исполняют одни и те же пять тестов: сортируют 17, 30, 149, 991 и 100000 элементов. Таблица ниже содержит результаты выполнения этих тестов.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Кол-во процессов  Кол-во  элементов | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 17 | 5.58001e-05 | 0.000546 | 0.0005821 | 0.0006664 | 0.0008424 | 0.0009198 | 0.0017931 | 0.0018604 |
| 30 | 3.39e-05 | 0.0001224 | 8.09e-05 | 0.0001157 | 0.0001483 | 0.000211 | 0.0001384 | 0.0002085 |
| 149 | 5.69e-05 | 0.0001067 | 0.0001434 | 0.0001289 | 0.0001455 | 0.0001853 | 0.0001579 | 0.0001928 |
| 991 | 0.0002998 | 0.0003639 | 0.0002996 | 0.0002323 | 0.0003774 | 0.0003755 | 0.0003766 | 0.0004862 |
| 100000 | 0.0247257 | 0.0167032 | 0.0179571 | 0.0140793 | 0.0148861 | 0.0167483 | 0.0136261 | 0.0150003 |

На пересечении строки и столбца находится время сортировки в секундах данного количества элементов данным количеством процессов.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Кол-во элементов | 17 | 30 | 149 | 991 | 100000 |
| Время | 7.79994e-06 | 1.05001e-05 | 5.42001e-05 | 0.0002747 | 0.0238914 |

Время сортировки, выполненной на одном процессе с помощью std::sort().

1. **Выводы из результатов**

Параллельная программа работает гораздо медленнее последовательной при небольших данных (меньше 100000, как видно из таблиц) и быстрее последовательной (примерно в 2 раза) при больших данных. Это происходит из-за накладных расходов на переключение между процессами и передачи данных между процессами.

1. **Заключение**

В ходе данной лабораторной работы мною были описаны следующие алгоритмы: распараллеливание быстрой сортировки слиянием одномерного массива и планирование количества обрабатываемых данных для каждого массива. Реализована эта работа на языке C++ с использованием MPI.

В заключении стоит, наверное, отметить, что данная работа позволила самостоятельно разобраться, как использовать параллелизм на уровне процессов, а также рассмотреть реализацию MPI для C++.

1. **Литература**
2. Википедия: сайт. — URL: <https://ru.wikipedia.org/wiki/Message_Passing_Interface> (дата обращения: 24.12.2022). — Текст: электронный.
3. MPICH.org: сайт — URL: <https://www.mpich.org/static/docs/latest/www3/MPI_Comm_size.html> (дата обращения: 24.12.2022). — Текст: электронный.
4. MPICH.org: сайт — URL: <https://www.mpich.org/static/docs/latest/www3/MPI_Comm_rank.html> (дата обращения: 24.12.2022). — Текст: электронный.
5. MPICH.org: сайт — URL: <https://www.mpich.org/static/docs/latest/www3/MPI_Send.html> (дата обращения: 24.12.2022). — Текст: электронный.
6. MPICH.org: сайт — URL: <https://www.mpich.org/static/docs/latest/www3/MPI_Recv.html> (дата обращения: 24.12.2022). — Текст: электронный.
7. MPICH.org: сайт — URL: <https://www.mpich.org/static/docs/latest/www3/MPI_Wtime.html>(дата обращения: 24.12.2022). — Текст: электронный.
8. **Приложение**

Файл CMakeLists.txt:

|  |  |
| --- | --- |
|  | get\_filename\_component(ProjectId ${CMAKE\_CURRENT\_SOURCE\_DIR} NAME) |
|  | enable\_testing() |
|  |  |
|  | if( USE\_MPI ) |
|  | if( UNIX ) |
|  | set(CMAKE\_C\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wno-uninitialized") |
|  | set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wno-uninitialized") |
|  | endif( UNIX ) |
|  |  |
|  | set(ProjectId "${ProjectId}\_mpi") |
|  | project( ${ProjectId} ) |
|  | message( STATUS "-- " ${ProjectId} ) |
|  |  |
|  | file(GLOB\_RECURSE ALL\_SOURCE\_FILES \*.cpp \*.h) |
|  |  |
|  | set(PACK\_LIB "${ProjectId}\_lib") |
|  | add\_library(${PACK\_LIB} STATIC ${ALL\_SOURCE\_FILES} ) |
|  |  |
|  | add\_executable( ${ProjectId} ${ALL\_SOURCE\_FILES} ) |
|  |  |
|  | target\_link\_libraries(${ProjectId} ${PACK\_LIB}) |
|  | if( MPI\_COMPILE\_FLAGS ) |
|  | set\_target\_properties( ${ProjectId} PROPERTIES COMPILE\_FLAGS "${MPI\_COMPILE\_FLAGS}" ) |
|  | endif( MPI\_COMPILE\_FLAGS ) |
|  |  |
|  | if( MPI\_LINK\_FLAGS ) |
|  | set\_target\_properties( ${ProjectId} PROPERTIES LINK\_FLAGS "${MPI\_LINK\_FLAGS}" ) |
|  | endif( MPI\_LINK\_FLAGS ) |
|  | target\_link\_libraries( ${ProjectId} ${MPI\_LIBRARIES} ) |
|  | target\_link\_libraries(${ProjectId} gtest gtest\_main) |
|  |  |
|  | enable\_testing() |
|  | add\_test(NAME ${ProjectId} COMMAND ${ProjectId}) |
|  |  |
|  | if( UNIX ) |
|  | foreach (SOURCE\_FILE ${ALL\_SOURCE\_FILES}) |
|  | string(FIND ${SOURCE\_FILE} ${PROJECT\_BINARY\_DIR} PROJECT\_TRDPARTY\_DIR\_FOUND) |
|  | if (NOT ${PROJECT\_TRDPARTY\_DIR\_FOUND} EQUAL -1) |
|  | list(REMOVE\_ITEM ALL\_SOURCE\_FILES ${SOURCE\_FILE}) |
|  | endif () |
|  | endforeach () |
|  |  |
|  | find\_program(CPPCHECK cppcheck) |
|  | add\_custom\_target( |
|  | "${ProjectId}\_cppcheck" ALL |
|  | COMMAND ${CPPCHECK} |
|  | --enable=warning,performance,portability,information,missingInclude |
|  | --language=c++ |
|  | --std=c++11 |
|  | --error-exitcode=1 |
|  | --template="[{severity}][{id}] {message} {callstack} \(On {file}:{line}\)" |
|  | --verbose |
|  | --quiet |
|  | ${ALL\_SOURCE\_FILES} |
|  | ) |
|  | endif( UNIX ) |
|  |  |
|  | SET(ARGS\_FOR\_CHECK\_COUNT\_TESTS "") |
|  | foreach (FILE\_ELEM ${ALL\_SOURCE\_FILES}) |
|  | set(ARGS\_FOR\_CHECK\_COUNT\_TESTS "${ARGS\_FOR\_CHECK\_COUNT\_TESTS} ${FILE\_ELEM}") |
|  | endforeach () |
|  |  |
|  | add\_custom\_target("${ProjectId}\_check\_count\_tests" ALL |
|  | COMMAND "${Python3\_EXECUTABLE}" |
|  | ${CMAKE\_SOURCE\_DIR}/scripts/check\_count\_tests.py |
|  | ${ProjectId} |
|  | ${ARGS\_FOR\_CHECK\_COUNT\_TESTS} |
|  | ) |
|  | else( USE\_MPI ) |
|  | message( STATUS "-- ${ProjectId} - NOT BUILD!" ) |
|  | endif( USE\_MPI ) |

Файл main.cpp:

1. // Copyright 2022 Korobeynikova Alisa
3. #include <gtest/gtest.h>
4. #include <gtest-mpi-listener.hpp>
5. #include "./quick\_merge\_sort.h"
6. std::vector<int> seqSolution(const std::vector<int>& m) {
7. std::vector<int> seq\_sort\_res(m);
8. std::sort(seq\_sort\_res.begin(), seq\_sort\_res.end());
9. return seq\_sort\_res;
10. }
11. void setRandomValues(std::vector<int> \*vec) {
12. std::random\_device dev;
13. std::mt19937 gen(dev());
14. for (int i = 0; i < vec->size(); ++i) {
15. vec->at(i) = gen() % 100;
16. }
17. }
18. TEST(Parallel\_Operations\_MPI, Test\_Sort) {
19. int rank;
20. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);
21. const int size = 27;
22. std::vector<int> global\_vec;
23. if (rank == 0) {
24. global\_vec = IntVector(size);
25. setRandomValues(&global\_vec);
26. }
27. std::vector<int> ps = parallelSort(global\_vec, size);
28. if (rank == 0) {
29. std::vector<int> ss = seqSolution(global\_vec);
30. ASSERT\_EQ(ps, ss);
31. }
32. }
33. TEST(Parallel\_Operations\_MPI, Test\_Sort\_2) {
34. int rank;
35. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);
36. const int size = 30;
37. std::vector<int> global\_vec;
38. if (rank == 0) {
39. global\_vec = IntVector(size);
40. setRandomValues(&global\_vec);
41. }
42. std::vector<int> ps = parallelSort(global\_vec, size);
43. if (rank == 0) {
44. std::vector<int> ss = seqSolution(global\_vec);
45. ASSERT\_EQ(ps, ss);
46. }
47. }
48. TEST(Parallel\_Operations\_MPI, Test\_Sort\_3) {
49. int rank;
50. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);
51. const int size = 17;
52. std::vector<int> global\_vec;
53. if (rank == 0) {
54. global\_vec = IntVector(size);
55. setRandomValues(&global\_vec);
56. }
57. std::vector<int> ps = parallelSort(global\_vec, size);
58. if (rank == 0) {
59. std::vector<int> ss = seqSolution(global\_vec);
60. ASSERT\_EQ(ps, ss);
61. }
62. }
63. TEST(Parallel\_Operations\_MPI, Test\_Sort\_4) {
64. int rank;
65. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);
66. const int size = 7;
67. std::vector<int> global\_vec;
68. if (rank == 0) {
69. global\_vec = IntVector(size);
70. setRandomValues(&global\_vec);
71. }
72. std::vector<int> ps = parallelSort(global\_vec, size);
73. if (rank == 0) {
74. std::vector<int> ss = seqSolution(global\_vec);
75. ASSERT\_EQ(ps, ss);
76. }
77. }
78. TEST(Parallel\_Operations\_MPI, Test\_Sort\_5) {
79. int rank;
80. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);
81. const int size = 50;
82. std::vector<int> global\_vec;
83. if (rank == 0) {
84. global\_vec = IntVector(size);
85. setRandomValues(&global\_vec);
86. }
87. std::vector<int> ps = parallelSort(global\_vec, size);
88. if (rank == 0) {
89. std::vector<int> ss = seqSolution(global\_vec);
90. ASSERT\_EQ(ps, ss);
91. }
92. }
93. int main(int argc, char\*\* argv) {
94. ::testing::InitGoogleTest(&argc, argv);
95. MPI\_Init(&argc, &argv);
96. ::testing::AddGlobalTestEnvironment(new GTestMPIListener::MPIEnvironment);
97. ::testing::TestEventListeners& listeners =
98. ::testing::UnitTest::GetInstance()->listeners();
99. listeners.Release(listeners.default\_result\_printer());
100. listeners.Release(listeners.default\_xml\_generator());
101. listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);
102. return RUN\_ALL\_TESTS();
103. }

Файл quick\_merge\_sort.h:

1. #pragma once
2. // Copyright 2022 Korobeynikova Alisa
3. #ifndef MODULES\_TEST\_TASKS\_TEST\_MPI\_OPS\_MPI\_H\_
4. #define MODULES\_TEST\_TASKS\_TEST\_MPI\_OPS\_MPI\_H\_
5. #include <random>
6. #include <vector>
7. using IntVector = std::vector<int>;
8. using IntVectorPtr = std::shared\_ptr<IntVector>;
9. std::vector<int> taskDistrib(const int proc\_num, const int task\_num);
10. std::vector<int> parallelSort(const std::vector<int>& global\_vec, const int elems\_num);
11. #endif // MODULES\_TEST\_TASKS\_TEST\_MPI\_OPS\_MPI\_H\_

Файл quick\_merge\_sort.cpp:

1. // Copyright 2022 Korobeynikova Alisa
2. #include "../../../modules/task\_3/korobeynikova\_a\_quick\_merge\_sort/quick\_merge\_sort.h"
3. #include <mpi.h>
4. #include <algorithm>
5. #include <queue>
6. #include <memory>
7. #include <vector>
8. IntVector taskDistrib(const int proc\_num, const int task\_num) {
9. IntVector task\_per\_proc(proc\_num);
10. int remained\_rows = task\_num % proc\_num;
11. for (int proc = 0; proc < proc\_num; ++proc) {
12. task\_per\_proc.at(proc) = task\_num / proc\_num + (remained\_rows > 0);
13. --remained\_rows;
14. }
15. return task\_per\_proc;
16. }
17. IntVector parallelSort(const IntVector &global\_vec, const int elems\_num) {
18. int size, rank;
19. MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);
20. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);
21. if (elems\_num == 0) {
22. return IntVector();
23. } else if (elems\_num == 1) {
24. return IntVector({global\_vec.at(0)});
25. }
26. IntVector elems\_per\_process = taskDistrib(size, elems\_num);
27. if (elems\_per\_process.at(rank) == 0) {
28. return IntVector{};
29. }
30. IntVector sorted\_vec(elems\_per\_process[rank]);
31. if (rank == 0) {
32. int elems\_to\_skip = elems\_per\_process.at(0);
33. for (int proc = 1; proc < size; ++proc) {
34. if (elems\_per\_process.at(proc) != 0) {
35. MPI\_Send(const\_cast<int \*>(&global\_vec.at(elems\_to\_skip)),
36. elems\_per\_process.at(proc), MPI\_INT, proc, 0, MPI\_COMM\_WORLD);
37. }
38. elems\_to\_skip += elems\_per\_process.at(proc);
39. }
40. }
41. IntVector local\_vec(elems\_per\_process.at(rank));
42. if (rank == 0) {
43. local\_vec = IntVector(global\_vec.begin(),
44. global\_vec.begin() + elems\_per\_process.at(rank));
45. } else {
46. MPI\_Status status;
47. MPI\_Recv(local\_vec.data(), elems\_per\_process.at(rank), MPI\_INT, 0, 0,
48. MPI\_COMM\_WORLD, &status);
49. }
51. std::sort(local\_vec.begin(), local\_vec.end());
52. if (rank == 0) {
53. MPI\_Status status;
54. std::queue<IntVectorPtr> ptr\_queue;
55. ptr\_queue.push(std::make\_shared<IntVector>(local\_vec));
56. for (int proc = 1; proc < size; ++proc) {
57. if (elems\_per\_process.at(proc) != 0) {
58. ptr\_queue.push(std::make\_shared<IntVector>(elems\_per\_process.at(proc)));
59. MPI\_Recv(ptr\_queue.back()->data(), ptr\_queue.back()->size(), MPI\_INT,
60. proc, 0, MPI\_COMM\_WORLD, &status);
61. }
62. }
63. while (ptr\_queue.size() != 1) {
64. IntVectorPtr f\_v\_ptr = ptr\_queue.front();
65. ptr\_queue.pop();
66. IntVectorPtr s\_v\_ptr = ptr\_queue.front();
67. ptr\_queue.pop();
68. auto temp =
69. std::make\_shared<IntVector>(f\_v\_ptr->size() + s\_v\_ptr->size());
70. std::merge(f\_v\_ptr->begin(), f\_v\_ptr->end(), s\_v\_ptr->begin(),
71. s\_v\_ptr->end(), temp->begin());
72. ptr\_queue.push(temp);
73. }
74. return \*ptr\_queue.back();
75. } else {
76. MPI\_Send(local\_vec.data(), local\_vec.size(), MPI\_INT, 0, 0, MPI\_COMM\_WORLD);
77. }
78. return local\_vec;
79. }