Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное

образовательное учреждение высшего образования

«Южно-Уральский государственный университет

(национальный исследовательский университет)»

Высшая школы электроники и компьютерных наук

Кафедра системного программирования

ОТЧЕТ  
о лабораторной работе №6  
по дисциплине «Технологии параллельного программирования»

Выполнил:   
студент группы КЭ-220   
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Голенищев А. Б.   
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2024 г.   
   
Отчет принял:   
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Жулев А. Э.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2024 г.

***Задание 17. Коллективные коммуникации: широковещательная рассылка данных***

Разработали программу обмена для определения количества повторений каждого символа в строке фиксированной длины с использованием программы подсчета повторений символов в строке с MPI\_Bcast, листнинг 1. Представлен результат ее работы, рисунок 1.

#include <mpi.h>

#include <stdio.h>

#include <string.h>

#include <ctype.h>

// Golenishchev Artem, KE-220 Task 17 (1)

#define MAX\_LEN 100

#define CHAR\_RANGE 128 // ASCII

int main(int argc, char\*\* argv) {

int rank, size, n = 0, counts[CHAR\_RANGE] = {0}, global\_counts[CHAR\_RANGE] = {0};

char buf[MAX\_LEN] = {0};

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (rank == 0) {

printf("Enter the string (max length %d): \n", MAX\_LEN);

fgets(buf, MAX\_LEN, stdin);

if ((n = strlen(buf)) > 0 && buf[n - 1] == '\n') buf[--n] = '\0';

for (int i = 0; i < n; i++) buf[i] = tolower(buf[i]);

}

MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(buf, n, MPI\_CHAR, 0, MPI\_COMM\_WORLD);

for (int i = rank; i < CHAR\_RANGE; i += size)

for (int j = 0; j < n; j++)

if (buf[j] == (char)i) counts[i]++;

MPI\_Reduce(counts, global\_counts, CHAR\_RANGE, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0)

for (int i = 0; i < CHAR\_RANGE; i++)

if (global\_counts[i] > 0 && isprint(i))

printf("%c = %d\n", (char)i, global\_counts[i]);

MPI\_Finalize();

return 0;

}

Листнинг 1. Код программы подсчета повторений символов в строке с MPI\_Bcast

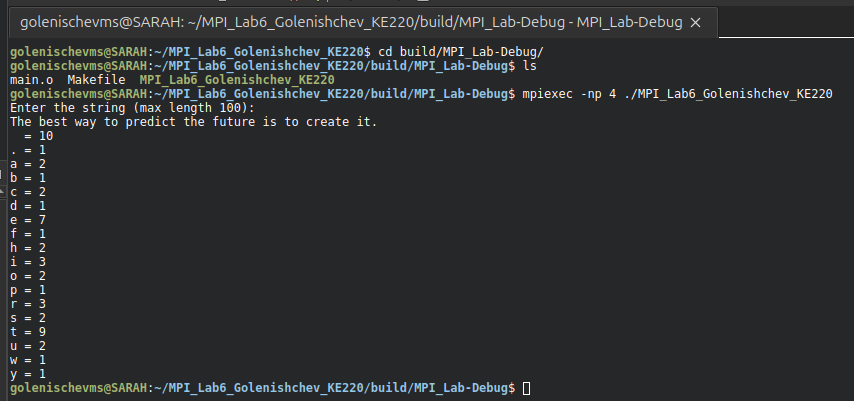


Рисунок 1. Результат работы программы подсчета повторений   
символов в строке с MPI\_Bcast

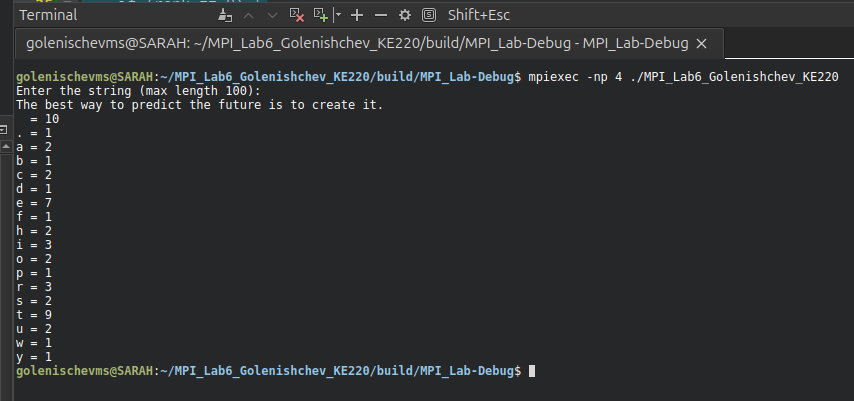
Разработали программу обмена для определения количества повторений каждого символа в строке фиксированной длины с использованием программы подсчета повторений символов в строке с MPI\_Send и MPI\_Recv, листнинг 2. Представлен результат ее работы, рисунок 2.

Рисунок 2. Результат работы программы подсчета повторений

символов в строке с MPI\_Send и MPI\_Recv

Листнинг 2. Код программы подсчета повторений символов с MPI\_Send и MPI\_Recv

#include <mpi.h>

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX\_LEN 100

#define CHAR\_RANGE 128

// Golenishchev Artem, KE-220 Task 17 (2)

int main(int argc, char\*\* argv) {

int rank, size, n = 0, counts[CHAR\_RANGE] = {0}, global\_counts[CHAR\_RANGE] = {0};

char buf[MAX\_LEN] = {0};

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (rank == 0) {

printf("Enter the string (max length %d): \n", MAX\_LEN);

fgets(buf, MAX\_LEN, stdin);

if ((n = strlen(buf)) > 0 && buf[n - 1] == '\n') buf[--n] = '\0';

for (int i = 0; i < n; i++) buf[i] = tolower(buf[i]);

for (int dest = 1; dest < size; dest++) {

MPI\_Send(&n, 1, MPI\_INT, dest, 0, MPI\_COMM\_WORLD);

MPI\_Send(buf, n, MPI\_CHAR, dest, 0, MPI\_COMM\_WORLD);}

} else {

MPI\_Recv(&n, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

MPI\_Recv(buf, n, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

for (int i = rank; i < CHAR\_RANGE; i += size)

for (int j = 0; j < n; j++)

if (buf[j] == (char)i) counts[i]++;

if (rank == 0) {

memcpy(global\_counts, counts, sizeof(counts));

for (int src = 1; src < size; src++) {

MPI\_Recv(counts, CHAR\_RANGE, MPI\_INT, src, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

for (int i = 0; i < CHAR\_RANGE; i++) global\_counts[i] += counts[i];

}

for (int i = 0; i < CHAR\_RANGE; i++)

if (global\_counts[i] > 0 && isprint(i))

printf("%c = %d\n", (char)i, global\_counts[i]);

} else {

MPI\_Send(counts, CHAR\_RANGE, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

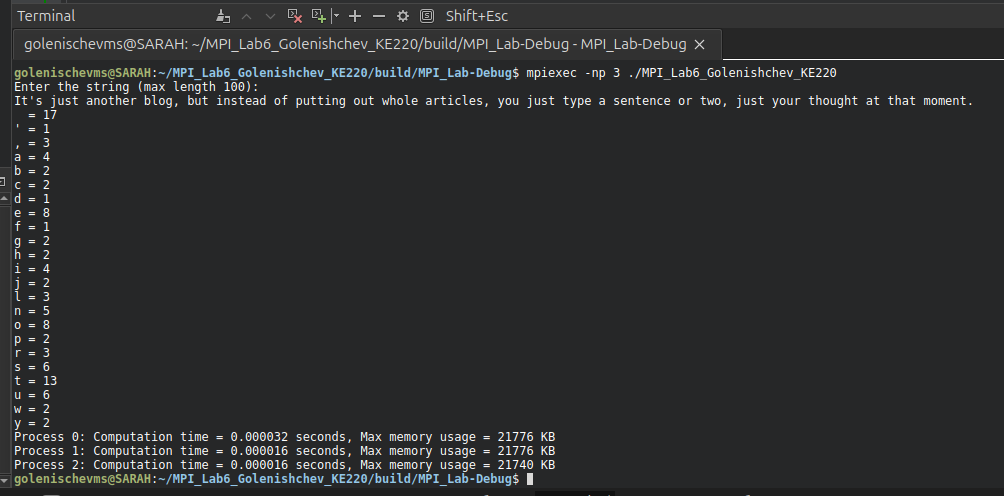
}

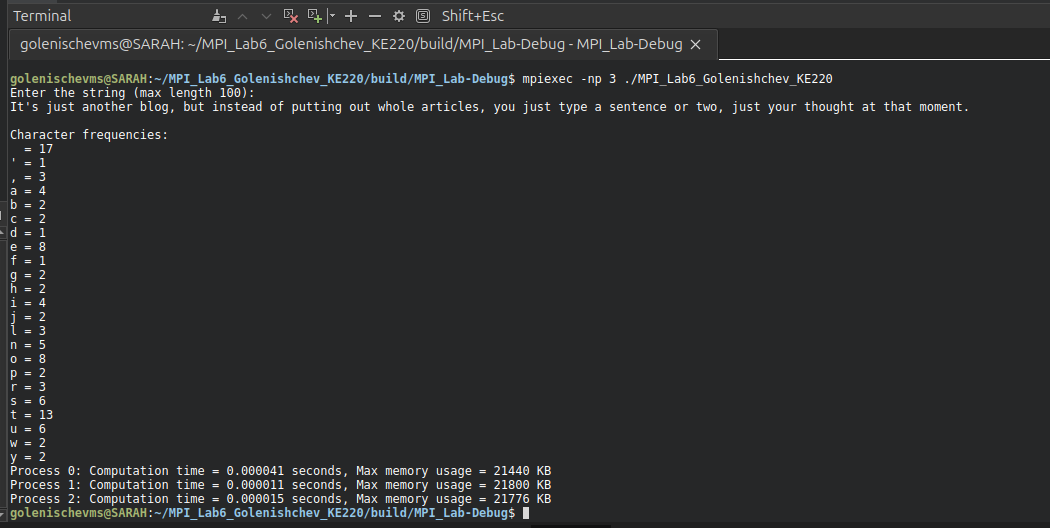
MPI\_Finalize();

return 0;

}

Разработали программу подсчета повторения символов в строке с MPI\_Bcast, код представлен в отдельном файле, приложенном к отчету. Представлен результат ее работы, рисунок 3. Повторили результат с MPI\_Send и MPI\_Recv, рисунок 4.

Рисунок 3. Результат работы программы подсчета повторений   
символов в строке с MPI\_Bcast с вычислением затраченных ресурсов

Рисунок 4. Результат работы программы подсчета повторений   
символов в строке с MPI\_Send и MPI\_Recv с вычислением затраченных ресурсов

В коде измерение времени выполнения выполняется с помощью функции MPI\_Wtime(), которая фиксирует начальное и конечное время выполнения в секундах в главном процессе (rank == 0), а разница между ними выводится как общее время выполнения. Для замера памяти используется функция getrusage() из <sys/resource.h>, которая возвращает максимальный объем использованной памяти процессом через поле ru\_maxrss, преобразованный в мегабайты в функции get\_memory\_usage\_MB().

Первая программа, использующая MPI\_Reduce для агрегации данных, работает быстрее и эффективнее с точки зрения производительности, так как минимизирует количество коммуникаций между процессами, что снижает сетевые задержки и нагрузку на память. Вторая программа, использующая точечные передачи данных с помощью MPI\_Send и MPI\_Recv, страдает от большего времени задержки и большего объема обмена данными, что увеличивает время вычислений и использует больше памяти. Поэтому первая программа будет более оптимальной с точки зрения как времени выполнения, так и использования ресурсов.

***Задание 18. Коллективные коммуникации: операции редукции***

Разработали программу вычисления числа π с использованием MPI\_Reduce, листнинг 3. Представлен результат ее работы, рисунок 5.

#include <stdio.h>

#include <mpi.h>

// Golenishchev Artem, KE-220 Task 18 (1)

double compute\_pi(int start, int end) {

double sum = 0.0;

for (int i = start; i < end; i++) {

sum += 1.0 / (1.0 + ((i + 0.5) / 1000000000.0) \* ((i + 0.5) / 1000000000.0));

}

return sum;

}

int main(int argc, char\*\* argv) {

int rank, size;

int N;

double local\_sum = 0.0, global\_sum = 0.0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (rank == 0) {

printf("Enter N: ");

scanf("%d", &N);

}

// Рассылка значения N всем процессам

MPI\_Bcast(&N, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Разбиение работы между процессами

int chunk\_size = N / size;

int start = rank \* chunk\_size;

int end = (rank == size - 1) ? N : start + chunk\_size;

// Вычисление локальной суммы

local\_sum = compute\_pi(start, end);

// Суммирование локальных сумм в глобальную с использованием MPI\_Reduce

MPI\_Reduce(&local\_sum, &global\_sum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

// Вывод результата на процессе 0

if (rank == 0) {

double pi = global\_sum \* 4.0 / N;

printf("Computed pi: %.8f\n", pi);

}

MPI\_Finalize();

return 0;

}

Листнинг 3. Код программы вычисления числа π с MPI\_Reduce

Разработали программу вычисления числа π с использованием MPI\_Reduce, листнинг 4. Представлен результат ее работы, рисунок 6

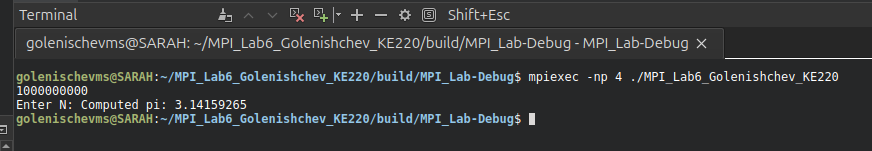
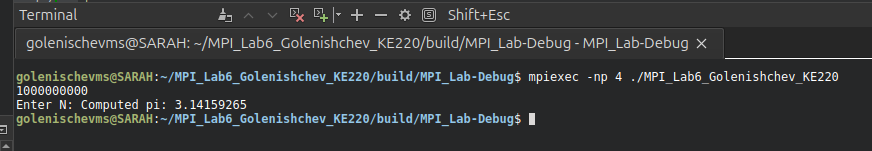


Рисунок 5. Результат выполнения программы вычисления числа π с MPI\_Reduce

Рисунок 6. Результат выполнения программы вычисления числа π с   
MPI\_Send и MPI\_Recv

Первая программа, использующая MPI\_Reduce для агрегации локальных сумм в глобальную, является более эффективной с точки зрения производительности, так как она минимизирует количество коммуникаций между процессами, что снижает затраты на передачу данных и время ожидания. Вторая программа, использующая точечные коммуникации через MPI\_Send и MPI\_Recv, требует дополнительных операций обмена сообщениями между процессами, что приводит к увеличению времени вычислений и сетевых задержек. Таким образом, первая программа будет быстрее и эффективнее, так как выполняет суммирование данных более оптимально, с меньшими затратами на память и время.

Листнинг 4. Код программы вычисления числа π с MPI\_Send и MPI\_Recv

// Golenishchev Artem, KE-220 Task 18 (2)

int main(int argc, char\*\* argv) {

int rank, size;

int N;

double local\_sum = 0.0, global\_sum = 0.0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (rank == 0) {

printf("Enter N: ");

scanf("%d", &N);

}

// Рассылка значения N всем процессам

MPI\_Bcast(&N, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Разбиение работы между процессами

int chunk\_size = N / size;

int start = rank \* chunk\_size;

int end = (rank == size - 1) ? N : start + chunk\_size;

// Вычисление локальной суммы

local\_sum = compute\_pi(start, end);

// Использование точечных коммуникаций для сбора данных

if (rank == 0) {

global\_sum = local\_sum;

for (int i = 1; i < size; i++) {

double temp\_sum;

MPI\_Recv(&temp\_sum, 1, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

global\_sum += temp\_sum;

}

double pi = global\_sum \* 4.0 / N;

printf("Computed pi: %.8f\n", pi);

} else {

MPI\_Send(&local\_sum, 1, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);

}

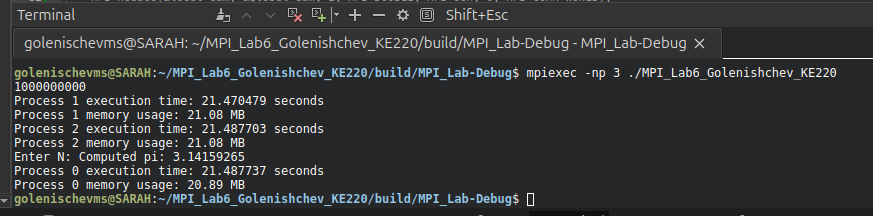
MPI\_Finalize();

return 0;

}

Представлен результат замера времени вычислений числа π с MPI\_Reduce, рисунок 7.

Представлен результат замера времени вычислений числа π с MPI\_Send и MPI\_Recv, рисунок. 8.

Рисунок 7. Код выполнения программы расчета числа π с MPI\_Reduce и замерами затраченного времени, памяти процессов

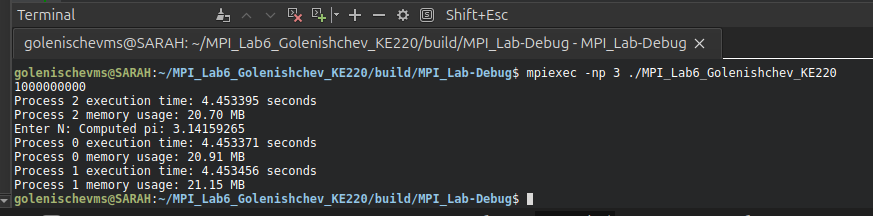


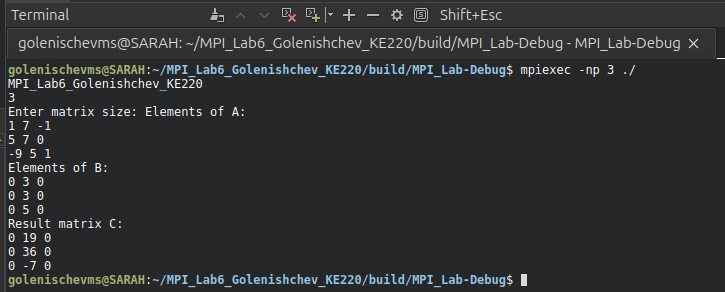
Рисунок 8. Код выполнения программы расчета числа π с MPI\_Send и MPI\_Recv   
и замерами затраченного времени, памяти процессов

Первая программа использует MPI\_Reduce для агрегации результатов подсчета частоты символов, что эффективно минимизирует количество коммуникаций между процессами и снижает нагрузку на сеть. Вторая программа, использующая точечные коммуникации (MPI\_Send и MPI\_Recv), требует дополнительного обмена сообщениями между процессами, что может увеличить время вычислений и сетевых задержек, особенно при большом числе процессов. Таким образом, первая программа будет более производительной в плане времени выполнения и сетевой нагрузки, так как она использует более оптимизированный механизм агрегации данных с помощью MPI\_Reduce. В то время как вторая программа имеет больше коммуникационных шагов, что может привести к большему времени ожидания.

***Задание 19. Коллективные коммуникации: функции распределения и***

***сбора данных***

Разработали программу вычисления двух квадратных матриц фиксированного размера с использованием MPI\_Scatter и MPI\_Gather, листнинг 5. Представлен результат ее работы, рисунок 9. Также представлен альтернативный вариант программы, листнинг 6. Показана его работа, рисунок 10.

Рисунок 9. Результат программы вычисления произведения матриц с   
MPI\_Scatter и MPI\_Gather

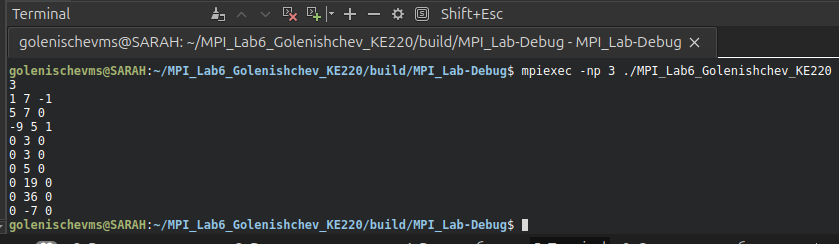


Рисунок 10. Результат программы вычисления произведения матрицы «точка-точка»

Листнинг 5. Код программы умножения матриц с MPI\_Scatter и MPI\_Gather

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

// Golenishchev Artem, KE-220 Task 19 (1)

int main(int argc, char\*\* argv) {

MPI\_Init(&argc, &argv);

int rank, size, n, \*A = NULL, \*B = NULL, \*C = NULL;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (rank == 0) {

printf("Enter matrix size: ");

scanf("%d", &n);

if (n % size != 0) {

printf("Matrix size must be divisible by the number of processes.\n");

MPI\_Abort(MPI\_COMM\_WORLD, 1); }

A = (int\*)malloc(n \* n \* sizeof(int));

B = (int\*)malloc(n \* n \* sizeof(int));

printf("Elements of A:\n");

for (int i = 0; i < n \* n; i++) scanf("%d", &A[i]);

printf("Elements of B:\n");

for (int i = 0; i < n \* n; i++) scanf("%d", &B[i]); }

MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (rank != 0) B = (int\*)malloc(n \* n \* sizeof(int));

MPI\_Bcast(B, n \* n, MPI\_INT, 0, MPI\_COMM\_WORLD);

int rows\_proc = n / size, \*local\_A = (int\*)malloc(rows\_proc \* n \* sizeof(int));

int\* local\_C = (int\*)calloc(rows\_proc \* n, sizeof(int));

MPI\_Scatter(A, rows\_proc \* n, MPI\_INT, local\_A, rows\_proc \* n, MPI\_INT, 0, MPI\_COMM\_WORLD);

for (int i = 0; i < rows\_proc; i++)

for (int j = 0; j < n; j++)

for (int k = 0; k < n; k++)

local\_C[i \* n + j] += local\_A[i \* n + k] \* B[k \* n + j];

if (rank == 0) C = (int\*)malloc(n \* n \* sizeof(int));

MPI\_Gather(local\_C, rows\_proc \* n, MPI\_INT, C, rows\_proc \* n, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

printf("Result matrix C:\n");

for (int i = 0; i < n \* n; i++) {

printf("%d ", C[i]);

if ((i + 1) % n == 0) printf("\n"); }

free(A); free(B); free(C);

} else {

free(B); }

free(local\_A); free(local\_C);

MPI\_Finalize();

return 0;

}

Листнинг 6. Код программы умножения матриц «точка-точка»

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

// Golenishchev Artem, KE-220 Task 19 (2)

int main(int argc, char\*\* argv) {

MPI\_Init(&argc, &argv);

int rank, size, n, \*A = NULL, \*B = NULL, \*C = NULL;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (rank == 0) {

scanf("%d", &n); // Размер матрицы

if (n % size != 0) {

fprintf(stderr, "Matrix size must be divisible by the number of processes.\n");

MPI\_Abort(MPI\_COMM\_WORLD, 1);}

A = (int\*)malloc(n \* n \* sizeof(int));

B = (int\*)malloc(n \* n \* sizeof(int));

for (int i = 0; i < n \* n; i++) scanf("%d", &A[i]);

for (int i = 0; i < n \* n; i++) scanf("%d", &B[i]);}

MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (rank != 0) B = (int\*)malloc(n \* n \* sizeof(int));

MPI\_Bcast(B, n \* n, MPI\_INT, 0, MPI\_COMM\_WORLD);

int rows\_proc = n / size; // Количество строк на процесс

int\* local\_A = (int\*)malloc(rows\_proc \* n \* sizeof(int));

int\* local\_C = (int\*)calloc(rows\_proc \* n, sizeof(int));

if (rank == 0) {

for (int i = 1; i < size; i++) {

MPI\_Send(A + i \* rows\_proc \* n, rows\_proc \* n, MPI\_INT, i, 0, MPI\_COMM\_WORLD);}

for (int i = 0; i < rows\_proc \* n; i++) local\_A[i] = A[i];

} else {

MPI\_Recv(local\_A, rows\_proc \* n, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE); }

for (int i = 0; i < rows\_proc; i++) {

for (int j = 0; j < n; j++) {

for (int k = 0; k < n; k++) {

local\_C[i \* n + j] += local\_A[i \* n + k] \* B[k \* n + j];}}}

if (rank == 0) {

C = (int\*)malloc(n \* n \* sizeof(int));

for (int i = 0; i < rows\_proc \* n; i++) C[i] = local\_C[i];

for (int i = 1; i < size; i++) {

MPI\_Recv(C + i \* rows\_proc \* n, rows\_proc \* n, MPI\_INT, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE); }

for (int i = 0; i < n \* n; i++) {

printf("%d%s", C[i], (i + 1) % n ? " " : "\n"); }

free(A); free(B); free(C);

} else {

MPI\_Send(local\_C, rows\_proc \* n, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

free(B); }

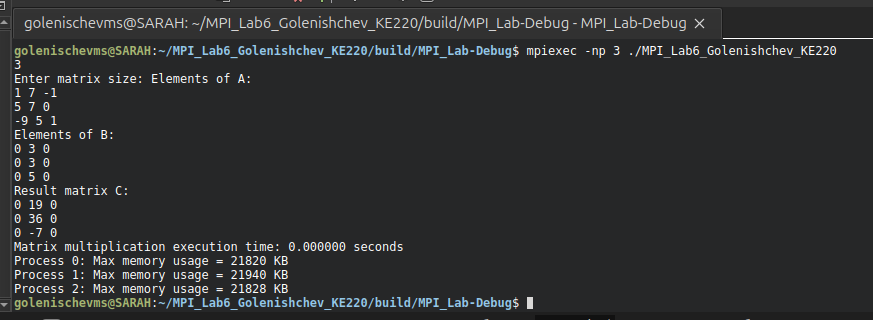
free(local\_A);

free(local\_C);

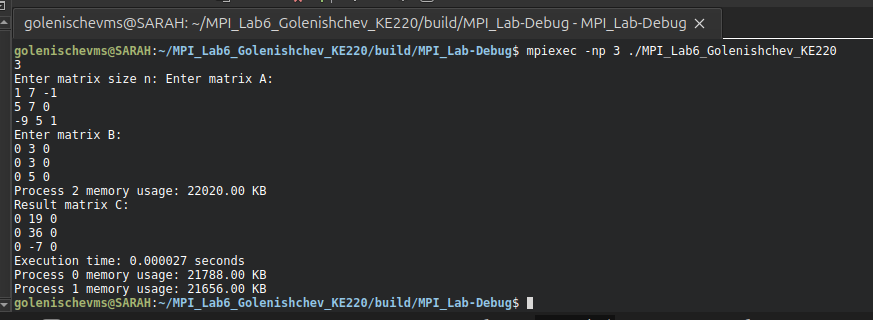
MPI\_Finalize();

return 0; }

Представлен результат замера используемой памяти и временных затрат на выполнение программы умножения матриц с MPI\_Scatter и MPI\_Gather, рисунок 11. Полный код программы доступен в отдельном прикрепленном с отчетом файле.

Рисунок 11. Результат программы вычисления произведения матриц с   
MPI\_Scatter и MPI\_Gather с оценкой затрат времени ее выполнения и выводом памяти процессов

Представлен результат замера используемой памяти и временных затрат на выполнение программы умножения матриц «точка-точка», рисунок 12. Полный код программы доступен в отдельном прикрепленном с отчетом файле.

Рисунок 12. Результат программы вычисления произведения матриц   
«точка-точка» с оценкой затрат времени ее выполнения и выводом памяти процессов

Методы MPI\_Scatter и MPI\_Gather в сравнении с "точка-точка" коммуникациями (например, MPI\_Send и MPI\_Recv) обычно демонстрируют лучшую производительность в задачах с большими объемами данных, требующих коллективного взаимодействия. Эти коллективные операции оптимизированы для распределения и сбора данных между процессами и используют эффективные алгоритмы передачи, минимизируя количество коммуникаций. В то же время, "точка-точка" подход может быть гибче для нерегулярных или асимметричных задач, но он требует ручного управления передачами, что увеличивает сложность кода и может приводить к большему времени выполнения из-за последовательного обмена данными. Таким образом, MPI\_Scatter и MPI\_Gather предпочтительны для сбалансированных задач, а "точка-точка" более подходит для кастомизированных сценариев.

***Ответы на вопросы:***

1. В чем отличие широковещательной коммуникации от коммуникации "точка-точка"?

Широковещательная коммуникация (MPI\_Bcast) передает данные от одного процесса (корневого) всем остальным процессам в коммуникаторе. Коммуникация "точка-точка" (например, MPI\_Send и MPI\_Recv) предполагает обмен данными между двумя процессами, где один отправляет данные, а другой получает. Широковещание происходит между несколькими процессами, а "точка-точка" — между двумя.

1. Опишите назначение и параметры функции MPI\_Bcast.

MPI\_Bcast используется для широковещательной передачи данных от одного процесса (корневого) ко всем другим в коммуникаторе. Параметры: адрес данных, размер данных, тип данных, ранг корневого процесса и коммуникатор. Эта функция обеспечивает эффективную передачу данных в иерархичных структурах.

1. Опишите назначение и параметры функции MPI\_Reduce. Какие агрегирующие операции могут использоваться в функции MPI\_Reduce? В чем отличие работы этой функции в корневом процессе и в остальных процессах коммуникатора?

MPI\_Reduce выполняет агрегирующую операцию (например, суммирование или нахождение максимума) на данных, собранных с разных процессов, и передает результат в корневой процесс. Параметры: данные для редукции, результат, количество элементов, тип данных, операция, ранг корневого процесса и коммуникатор. Агрегирующие операции включают SUM, MAX, MIN и другие. В корневом процессе результат редукции сохраняется, в других процессах — данные не изменяются.

1. Опишите назначение и параметры функций MPI\_Scatter и MPI\_Gather.

MPI\_Scatter делит данные на части и распределяет их между процессами, каждый из которых получает свою часть данных. MPI\_Gather собирает данные с разных процессов и объединяет их в одном процессе. Параметры обеих функций: исходные и целевые массивы, количество элементов, тип данных, и коммуникатор.

1. Можно ли с помощью этих функций рассылать/собирать динамически выделенные двумерные массивы (матрицы)? Можно ли использовать эти функции, если количество данных не кратно количеству нитей?

Да, с помощью MPI\_Scatter и MPI\_Gather можно рассылать и собирать динамически выделенные двумерные массивы, но необходимо использовать одномерные массивы для передачи данных, адаптируя индексы. Если количество данных не кратно количеству нитей, то процессам, получающим данные, нужно быть готовыми к обработке неполных блоков, что требует дополнительной логики в программе.

***Выводы:***

Изучили основные механизмы параллельных вычислений с использованием библиотеки MPI, включая коллективные операции, такие как MPI\_Bcast для широковещательной рассылки, MPI\_Scatter и MPI\_Gather для распределения и сбора данных, а также MPI\_Reduce для агрегирования данных с выполнением операций (например, суммы или максимума). Мы разобрались в отличиях коллективных операций от точка-точка коммуникаций (MPI\_Send, MPI\_Recv) и научились применять их для задач, таких как умножение матриц. Также мы оценили производительность вычислений, замерив время выполнения операций и объем потребляемой памяти, и разобрались с ограничениями при работе с некратным количеством данных и динамическими структурами.