

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Практикум по учебному курсу

"Распределенные системы"

**Разработка надежной программы для перемножения
матриц с использованием алгоритма Кэннона с
использованием технологии MPI**

**Разработка древовидного маркерного алгоритма для
прохождения всеми процессами критических секций**

Отчет

студента 420 группы

факультета ВМК МГУ

Михельсон Герман Владимирович

2022 год

Задание 1

Постановка задачи

Все 25 процессов, находящихся на разных ЭВМ сети, одновременно выдали запрос на вход в критическую секцию. Реализовать программу, использующую древовидный маркерный алгоритм для прохождения всеми процессами критических секций.

Критическая секция:

```
<проверка наличия файла "critical.txt">;
if (<файл "critical.txt" существует>)
{
    <сообщение об ошибке>;
    <завершение работы программы>;
}
else
{
    <создание файла "critical.txt">;
    Sleep (<случайное время>);
    <уничтожение файла "critical.txt">;
}
```

Для передачи маркера использовать средства MPI.

Получить временную оценку работы алгоритма. Оценить сколько времени потребуется, если маркером владеет нулевой процесс. Время старта (время «разгона» после получения доступа к шине для передачи сообщения) равно 100, время передачи байта равно 1 ($T_s=100, T_b=1$). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

Решение

Будем считать, что корнем двоичного дерева является 0-ой процесс, у которого находится маркер в начальный момент времени. Проведём эмуляцию одновременного входа в критическую секцию при помощи функции *MPI_Barrier*. Инициализация общей памяти проводится для определения сколько ещё процессов не прошло критическую секцию. Когда все процессы прошли её, работа программы завершается.

Для оценки времени составляется сбалансированное дерево. Где бы не находился маркер, нам необходимо $N-1$ посылок запросов для инициализации (по каждому ребру либо в одну, либо в другую сторону). Начинаем обход с вершины, где есть маркер, и обходим его в глубину, начиная сначала с правых поддеревьев, переходя к левым. Тогда

по каждому ребру придётся пройти 2 раза (вперёд и назад), кроме последней самой левой цепочки. Ещё стоит учесть M запросов после того, как маркер покидает вершину, в направлении ушедшего маркера (они выполняются, если очередь запросов в узле не пуста). Эту оценку можно улучшить, посылая запрос сразу же вместе с маркером в одном сообщении.

Итого:

$(25 + \sim 46) \cdot (Ts + 1 \cdot Tb) + M \cdot Tb = 71 \cdot 101 + \sim 21 = 7192$ для 25 процессов.

Если маркер в начальный момент времени будет находиться не в 0-ом процессе, то нам потребуется меньше операций, и следовательно время уменьшится.

Древовидный маркерный алгоритм:

- Вход в критическую секцию:

Если есть маркер, то процесс выполняет критическую секцию, если нет маркера, то помещает свой запрос в FIFO-очередь запросов и посылает сообщение «ЗАПРОС» в направлении владельца маркера и ждёт сообщений.

- Поведение процесса при приёме сообщений:

Процесс, не находящийся внутри критической секции, должен реагировать на сообщения двух видов – «МАРКЕР» (-1) и «ЗАПРОС» (номер узла, от 0 до $N-1$).

А) Если пришло сообщение «МАРКЕР»

- Взять 1-ый запрос из очереди. Если это свой запрос, то исполнить вход в критическую секцию, изъять его из очереди и вернуться к пункту *a*.
Иначе послать маркер его автору.
- Поменять значение указателя в сторону маркера. (*marker_pointer*)
- Исключить запрос из очереди.
- Если в очереди остались запросы, то послать сообщение «ЗАПРОС» в сторону маркера.

В) Если пришло сообщение «ЗАПРОС»

Поместить запрос в очередь. Если нет маркера, то послать сообщение «ЗАПРОС» в сторону маркера, иначе (если есть маркер) – перейти на пункт *a*.

- Если очередь запросов пуста, то при выходе ничего не выполняется, иначе выполняется пункт *a*.

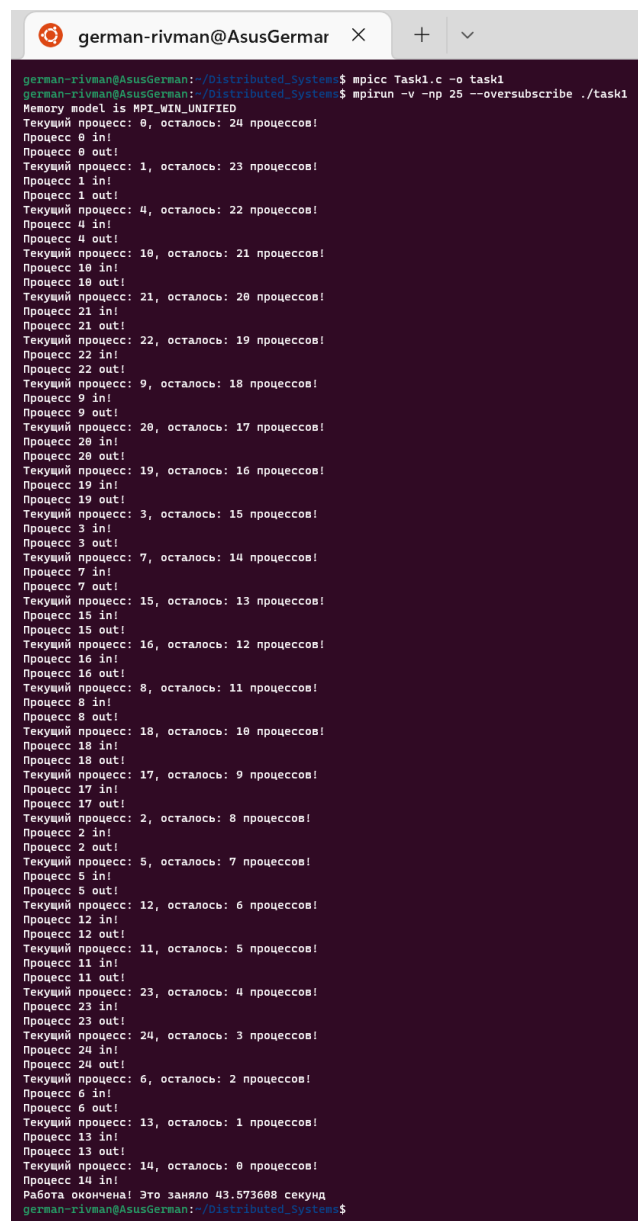
Начало работы – запрос входа в критическую секцию осуществляется отдельным условием в *main*, ожидание маркера (*wait_marker*). При получении маркера – возврат к работе, прохождение критической секции. При завершении критической секции

процессы продолжают «слушать» (*check_query*), отвечая на запросы от соседних процессов, пока не наступит период, за который не будет принято ни одного сообщения (проверка состояния осуществляется при помощи *MPI_Test* через равные промежутки времени).

Программу рекомендуется запускать в UNIX подобной системе:

1. Компиляция программы: *mpicc Task1.c -o task1*
2. Запуск программы: *mpirun -v -np 25 --oversubscribe ./task1*

(*) Все измерения проводились с помощью *--oversubscribe* из-за нехватки мощности ноутбука



```
german-rivman@AsusGerman: ~$ mpicc Task1.c -o task1
german-rivman@AsusGerman: ~$ mpirun -v -np 25 --oversubscribe ./task1
Memory model is MPI_WIN_UNIFIED
Текущий процесс: 0, осталось: 24 процессов!
Процесс 0 in!
Процесс 0 out!
Текущий процесс: 1, осталось: 23 процессов!
Процесс 1 in!
Процесс 1 out!
Текущий процесс: 4, осталось: 22 процессов!
Процесс 4 in!
Процесс 4 out!
Текущий процесс: 10, осталось: 21 процессов!
Процесс 10 in!
Процесс 10 out!
Текущий процесс: 21, осталось: 20 процессов!
Процесс 21 in!
Процесс 21 out!
Текущий процесс: 22, осталось: 19 процессов!
Процесс 22 in!
Процесс 22 out!
Текущий процесс: 9, осталось: 18 процессов!
Процесс 9 in!
Процесс 9 out!
Текущий процесс: 20, осталось: 17 процессов!
Процесс 20 in!
Процесс 20 out!
Текущий процесс: 19, осталось: 16 процессов!
Процесс 19 in!
Процесс 19 out!
Текущий процесс: 3, осталось: 15 процессов!
Процесс 3 in!
Процесс 3 out!
Текущий процесс: 7, осталось: 14 процессов!
Процесс 7 in!
Процесс 7 out!
Текущий процесс: 15, осталось: 13 процессов!
Процесс 15 in!
Процесс 15 out!
Текущий процесс: 16, осталось: 12 процессов!
Процесс 16 in!
Процесс 16 out!
Текущий процесс: 8, осталось: 11 процессов!
Процесс 8 in!
Процесс 8 out!
Текущий процесс: 18, осталось: 10 процессов!
Процесс 18 in!
Процесс 18 out!
Текущий процесс: 17, осталось: 9 процессов!
Процесс 17 in!
Процесс 17 out!
Текущий процесс: 2, осталось: 8 процессов!
Процесс 2 in!
Процесс 2 out!
Текущий процесс: 5, осталось: 7 процессов!
Процесс 5 in!
Процесс 5 out!
Текущий процесс: 12, осталось: 6 процессов!
Процесс 12 in!
Процесс 12 out!
Текущий процесс: 11, осталось: 5 процессов!
Процесс 11 in!
Процесс 11 out!
Текущий процесс: 23, осталось: 4 процессов!
Процесс 23 in!
Процесс 23 out!
Текущий процесс: 24, осталось: 3 процессов!
Процесс 24 in!
Процесс 24 out!
Текущий процесс: 6, осталось: 2 процессов!
Процесс 6 in!
Процесс 6 out!
Текущий процесс: 13, осталось: 1 процессов!
Процесс 13 in!
Процесс 13 out!
Текущий процесс: 14, осталось: 0 процессов!
Процесс 14 in!
Работа окончена! Это заняло 43.573688 секунд
german-rivman@AsusGerman: ~$
```

Код программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <unistd.h>

#define MPI_TAG 25

int world_rank = 0;
int *query, *query_head, *query_tail;
int marker_rank;

int flag;
int *processesRemaining;

MPI_Win wintable;

MPI_Aint winsize;
int windisp;

double start_time, end_time;

char *critical_file = "critical.txt";

void move_query()
{
    query_head++;
    if (query_head > query_tail)
    {
        query_head = query;
        query_tail = query;
        *query = -1;
    }
}
```

```

void add_query(int rank)
{
    *query_tail = rank;
    query_tail++;
}

void check_out()
{
    if (processesRemaining[0] == 0)
    {
        remove(critical_file);
        end_time = MPI_Wtime();
        printf("Работа окончена! Это заняло %f секунд\n", end_time-start_time);
        exit(0);
    }
}

void critical_section()
{
    MPI_Win_shared_query(wintable, 0, &winsize, &windisp, &processesRemaining);

    processesRemaining[0] = processesRemaining[0] - 1;

    printf("Текущий процесс: %d, осталось: %d процессов!\n", world_rank,
processesRemaining[0]);
    FILE* fp;
    if (fp = fopen(critical_file, "r"))
    {
        fclose(fp);
        printf("Ошибка!: критический файл уже существует! Вызвана процессом:
%d\n", world_rank);
        exit(1);
    }
    else
    {
        if (fp = fopen(critical_file, "w"))

```

```

    {
        fclose(fp);
        printf("Процесс %d in!\n", world_rank);
        check_out();
        printf("Процесс %d out!\n", world_rank);
        remove(critical_file);
        return;
    }
    else
    {
        printf("Ошибка! Не удалось открыть файл на запись!\n");
        exit(1);
    }
}
}

void send_marker(int whither)
{
    int temp = -1;
    MPI_Request request;
    MPI_Isend(&temp, 1, MPI_INT, whither, MPI_TAG, MPI_COMM_WORLD, &request);
    MPI_Request_free(&request);
}

void send_request(int marker_rank)
{
    MPI_Request request;
    MPI_Isend(&world_rank, 1, MPI_INT, marker_rank, MPI_TAG, MPI_COMM_WORLD,
&request);
    MPI_Request_free(&request);
}

void accept_marker()
{
    marker_rank = world_rank;
    if (*query_head == world_rank)
    {

```

```

        move_query();
        return;
    }
    if (*query_head == -1)
    {
        return;
    }
    marker_rank = *query_head;
    send_marker(*query_head);
    move_query();
    if (*query_head != -1)
    {
        send_request(marker_rank);
    }
}

void accept_request(int rank)
{
    add_query(rank);

    if (marker_rank == world_rank)
    {
        marker_rank = *query_head;
        send_marker(*query_head);
        move_query();
        if (*query_head != -1)
        {
            send_request(marker_rank);
        }
    }
    else
    {
        send_request(marker_rank);
        // отправили свой ранк процессу с маркером
    }
}

```



```

void wait_marker()
{
    int buffer = 0;
    MPI_Status status;
    MPI_Request request;
    MPI_Irecv(&buffer, 1, MPI_INT, MPI_ANY_SOURCE, MPI_TAG,
MPI_COMM_WORLD, &request);
    MPI_Wait(&request, &status);
    if (buffer== -1)
    {
        accept_marker();
        return;
    }
    accept_request(buffer);

    wait_marker();
}

```

```

void check_query()
{
    int buffer = 0;
    MPI_Status status;
    MPI_Request request;
    MPI_Irecv(&buffer, 1, MPI_INT, MPI_ANY_SOURCE, MPI_TAG,
MPI_COMM_WORLD, &request);

    int flag = 0, counter = 1024;
    MPI_Test(&request, &flag, &status);
    while((!flag) && (counter>0))
    {
        sleep(1);
        MPI_Test(&request, &flag, &status);
        counter--;
    }
    if (counter == 0)
    {
        return;
    }
}

```

```

    }

    if (buffer == -1)
    {
        accept_marker();
    }
    else
    {
        accept_request(buffer);
    }

    check_query();
}

int main(int argc, char ** argv){
    //Инициализируем MPI, получаем необходимые переменные, инициализируем
очередь
    int status = MPI_Init(&argc, &argv);
    if (status != MPI_SUCCESS)
    {
        return status;
    }
    start_time = MPI_Wtime();
    int world_size = 1;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    query = (int*)malloc(world_size*sizeof(int)*1024);
    for (int i=0; i<world_size; i++)
    {
        query[i] = -1;
    }
    query_head = query;
    query_tail = query;

    //Настраиваем общую память
    int tablesize = 1;
    MPI_Comm nodecomm;

```

```

    MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED,
world_rank, MPI_INFO_NULL, &nodecomm);
    MPI_Comm_size(nodecomm, &world_size);
    MPI_Comm_rank(nodecomm, &world_rank);

    int localtablesize = 0;
    if (world_rank == 0)
    {
        localtablesize = tablesize;
    }

    int *model;
    int *localtable;
    MPI_Win_allocate_shared(localtablesize*sizeof(int), sizeof(int), MPI_INFO_NULL,
nodecomm, &localtable, &wintable);
    MPI_Win_get_attr(wintable, MPI_WIN_MODEL, &model, &flag);
    if (flag != 1)
    {
        printf("Attribute MPI_WIN_MODEL not defined\n");
    }
    else
    {
        if (MPI_WIN_UNIFIED == *model)
        {
            if (world_rank == 0) printf("Memory model is MPI_WIN_UNIFIED\n");
        }
        else
        {
            if (world_rank == 0) printf("Memory model is *not* MPI_WIN_UNIFIED\n");
            MPI_Finalize();
            return -1;
        }
    }

    processesRemaining = localtable;

```

```
if (world_rank != 0)
{
    MPI_Win_shared_query(wintable, 0, &winsize, &windisp, &processesRemaining);
}

MPI_Win_fence(0, wintable);

if (world_rank == 0)
{
    processesRemaining[0] = world_size;
}

MPI_Win_fence(0, wintable);

marker_rank = (world_rank == 0) ? 0 : (world_rank - 1) / 2;

//Используем барьерную синхронизацию
MPI_Barrier(MPI_COMM_WORLD);

if (world_rank != marker_rank)
{
    accept_request(world_rank);
    wait_marker();
}

critical_section();

check_query();

MPI_Finalize();
return 0;
}
```

Задание 2

Постановка задачи

Доработать MPI-программу, реализованную в рамках курса “Суперкомпьютеры и параллельная обработка данных”.

Добавить контрольные точки для продолжения работы программы в случае сбоя.

Реализовать один из 3-х сценариев работы после сбоя:

- продолжить работу программы только на “исправных” процессах;
- вместо процессов, вышедших из строя, создать новые MPI-процессы, которые необходимо использовать для продолжения расчетов;
- при запуске программы на счет сразу запустить некоторое дополнительное количество MPI-процессов, которые использовать в случае сбоя.

Решение

В случае сбоя программы выбран сценарий, когда при запуске программы на счет сразу запустить дополнительное количество MPI-процессов, которые будут использованы в случае сбоя.

В старую программу было добавлено:

- функция-обработчик ошибок (*verbose_errhandler*), в которой описано, как реагировать процессу в случае возникновения сбоя. В случае смерти одного из процессов во всех процессах управление переходит в *verbose_errhandler*. По итогу сбоя все процессы должны удалить из своей рабочей группы мертвые процессы с помощью функции *MPIX_Comm_shrink*;
- в качестве дополнительного процесса используется один;

- функции чтения и сохранения данных: *data_save* и *data_load*.

Алгоритм работы:

- Основная работа начинается с цикла количество проходов, которого равно числу процессов, однако в данном случае количество проходов на 1 меньше общего числа процесса. Также важный момент, что необходимо при запуске программы подавать количество процессов на 1 больше квадрата числа;
- каждую итерацию для каждого процесса перемножаются матрицы *A* и *B*, а результат записывается в матрицу *C*. В файл *data* сохраняются все матрицы последнего процесса в конечном итоге;
- после сохранения, мы проверяем, на какой итерации все процессы находятся. Если номер процесса совпадает с заранее вложенным(в данном случае взяли половину от общего числа, но можно и любое другое), то убиваем последний из процессов в процессорной решётке;
- после этого, когда остальные (оставшиеся в живых) процессы доходят до барьера, они замечают потерю и вызывается функцию *verbose_errhandler*. А в нем уже находится номер упавшего процесса, делается *MPIX_Comm_shrink*, чтобы получить новый коммуникатор без мертвых процессов, и на его основе пересоздается *Cart* с помощью функции *MPI_Cart_create*;
- для нового *Cart* мы переопределяем номера процессов(так как упал у нас последний, новый просто займёт его место, получив его ранк). Потом в функции *load_data* этот новый процесс получает сохраненные данные от упавшего. И теперь новый процесс обладает и ранком, и данными упавшего процесса и может выполнять его задачи.

Результаты замеров времени выполнения

Работа задачи рассмотрена на ноутбуке с процессором 12th Gen Intel(R) Core(TM) i7-12700H с различным числом нитей (от 5 до 26) и различными размерами матрицы (от 500 до 2000).

Каждое измерение проводилось 3 раза. В таблице записаны усредненные результаты времени выполнения.

(*) Все измерения проводились с помощью *--oversubscribe* из-за нехватки мощности ноутбука.

Таблица с результатами

Узлы/размер матрицы	500	1000	1500	2000

5	0.2093s	1.3845s	4.5512s	14.6779s
10	0.1074s	0.8587s	3.0666s	8.3872s
17	0.0972s	0.7369s	2.2731s	6.8588s
26	0.0588s	0.4699s	1.2818s	5.9865s

Программу рекомендуется запускать в UNIX подобной системе:

3. Компиляция программы: *mpicc Cannon.c -o cannon*
4. Запуск программы, где после названия исполняемого файла необходимо указать размер матриц для перемножения: *mpirun -v -np 5 --enable-recovery --with-ft=ulfm --oversubscribe ./cannon 500*

```

german-rivman@AsusGerman: ~/Distributed_Systems/Task1$ mpicc Cannon.c -o cannon
german-rivman@AsusGerman:~/Distributed_Systems/Task1$ mpirun -v -np 5 --enable-recovery --with-ft=ulfm --oversubscribe ./cannon 500

WARNING: The selected 'topo' module 'basic' is not tested for post-failure
operation, yet you have requested support for fault tolerance.
When using this component, normal failure free operation is expected;
However, failures may cause the application to abort, crash or deadlock.

In this framework, the following components are tested to operate under
failure scenarios: {}

DEAD
Rank 4 / 5: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 1 found dead: { 3 }
Rank 2 / 5: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 1 found dead: { 3 }
Rank 0 / 5: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 1 found dead: { 3 }
Rank 1 / 5: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 1 found dead: { 3 }
Time: 0.2238s
german-rivman@AsusGerman:~/Distributed_Systems/Task1$

```

Код программы:

```

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <mpi.h>

#include <signal.h>

#include <mpi-ext.h>

```

```
MPI_Comm ACTIVE_COMM = MPI_COMM_WORLD, cannon_comm;
```

```
int NI;
```

```
double *A, *B, *C;
```

```
int global_rank, global_size;
```

```
int coords[2];
```

```
int left, right, up, down;
```

```
int ndims = 2;
```

```
int dims[2];
```

```
int periods[2];
```

```
static void data_save()
```

```
{
```

```
    if (global_rank == global_size - 2) {
```

```
        FILE* f = fopen("data", "wb");
```

```
        fwrite(&A[0], sizeof(double), NI * NI, f);
```

```
        fwrite(&B[0], sizeof(double), NI * NI, f);
```

```
        fwrite(&C[0], sizeof(double), NI * NI, f);
```

```
        fclose(f);
```

```
    }
```

```
    MPI_Barrier(ACTIVE_COMM);
```

```
}
```

```
static void data_load()
```

```
{
```



```

if (global_rank == global_size - 1) {

    FILE* f = fopen("data", "rb");

    fread(&A[0], sizeof(double), Nl * Nl, f);

    fread(&B[0], sizeof(double), Nl * Nl, f);

    fread(&C[0], sizeof(double), Nl * Nl, f);

    fclose(f);

    printf("Proc %d\n", global_rank);

}

MPI_Barrier(ACTIVE_COMM);
}

static void verbose_errhandler(MPI_Comm* pcomm, int* perr, ...) {

    MPI_Comm comm = *pcomm;

    int err = *perr;

    char errstr[MPI_MAX_ERROR_STRING];

    int i, rank, size, nf, len, eclass;

    MPI_Group group_c, group_f;

    int *ranks_gc, *ranks_gf;

    MPI_Error_class(err, &eclass);

    if( MPIX_ERR_PROC_FAILED != eclass ) {

        MPI_Abort(comm, err);

    }
}

```

```
MPI_Comm_rank(comm, &rank);

MPI_Comm_size(comm, &size);

MPIX_Comm_failure_ack(comm);

MPIX_Comm_failure_get_acked(comm, &group_f);

MPI_Group_size(group_f, &nf);

MPI_Error_string(err, errstr, &len);

printf("Rank %d / %d: Notified of error %s. %d found dead: { ", rank, size, errstr, nf);

ranks_gf = (int*)malloc(nf * sizeof(int));

ranks_gc = (int*)malloc(nf * sizeof(int));

MPI_Comm_group(comm, &group_c);

for(i = 0; i < nf; i++) {

    ranks_gf[i] = i;

}

MPI_Group_translate_ranks(group_f, nf, ranks_gf,

group_c, ranks_gc);

for(i = 0; i < nf; i++) {

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

}

printf("}\n");


MPIX_Comm_shrink(comm, &ACTIVE_COMM);

MPI_Comm_rank(ACTIVE_COMM, &global_rank);

MPI_Cart_create(ACTIVE_COMM, ndims, dims, periods, 0, &cannon_comm);
```

```

MPI_Cart_coords(cannon_comm, global_rank, 2, coords);

MPI_Cart_shift(cannon_comm, 1, 1, &left, &right);

MPI_Cart_shift(cannon_comm, 0, 1, &up, &down);

data_load();
}

int main(int argc, char **argv)
{
    if (argc != 2)
    {
        printf("ERROR: Please enter NODE_NUMBERA: \n./run NODE_NUMBERS\n");

        return -1;
    }

    double start, end;

    double *buf, *tmp;

    long long int N = atoi(argv[1]);

    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &global_rank);

    MPI_Comm_size(MPI_COMM_WORLD, &global_size);

    MPI_Errhandler errh;

    MPI_Comm_create_errhandler(verbose_errhandler, &errh);

```

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD, errh);

int new_rank = global_rank, new_size = global_size;

dims[0] = 0;

dims[1] = 0;

periods[0] = 1;

periods[1] = 1;

MPI_Dims_create(global_size-1, 2, dims);

if(dims[0] != dims[1])
{
    if(global_rank == 0)
    {
        printf("The number of processors must be a square.\n");
    }

    MPI_Finalize();

    return 0;
}

Nl = N / dims[0];

A = (double*)malloc(Nl * Nl * sizeof(double));

B = (double*)malloc(Nl * Nl * sizeof(double));

buf = (double*)malloc(Nl * Nl * sizeof(double));

C = (double*)calloc(Nl * Nl, sizeof(double));

for(int i=0; i<Nl; i++)
{
```

```

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

{

    A[i*Nl+j] = global_rank;//5 - (int)( 10.0 * rand() / ( RAND_MAX + 1.0 ) );

    B[i*Nl+j] = global_rank;//5 - (int)( 10.0 * rand() / ( RAND_MAX + 1.0 ) );

    C[i*Nl+j] = 0.0;

}

}

MPI_Comm new_comm;

MPI_Comm_split(MPI_COMM_WORLD, global_rank != global_size-1, global_rank,
&new_comm);

if (global_rank != global_size-1)

{

    MPI_Cart_create(new_comm, ndims, dims, periods, 0, &cannon_comm);

    MPI_Cart_coords(cannon_comm, global_rank, 2, coords);

    MPI_Cart_shift(cannon_comm, 1, coords[0], &left, &right);

    MPI_Cart_shift(cannon_comm, 0, coords[1], &up, &down);

    if (coords[0]) {

        MPI_Sendrecv(A, Nl * Nl, MPI_DOUBLE, left, 1, buf, Nl * Nl, MPI_DOUBLE,
right, 1, cannon_comm, &status);

        tmp = buf;

        buf = A;

        A = tmp;

    }

```

```

if (coords[1]) {

    MPI_Sendrecv(B, Nl * Nl, MPI_DOUBLE, up, 2, buf, Nl * Nl, MPI_DOUBLE,
down, 2, cannon_comm, &status);

    tmp = buf;

    buf = B;

    B = tmp;

}

MPI_Cart_shift(cannon_comm, 1, 1, &left, &right);

MPI_Cart_shift(cannon_comm, 0, 1, &up, &down);

}

start = MPI_Wtime();

for(int shift=0; shift < dims[0]; shift++)

{

    for(int i=0; i<Nl; i++)

    {

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

        {

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

            {

                C[i*Nl+j] += A[i*Nl+k]*B[k*Nl+j];

            }

        }

    }

}

```

```

    }

    data_save();

    if(shift == dims[0] / 2)

    {

        if (global_rank == global_size - 2) {

            printf("DEAD\n");

            raise(SIGKILL);

        }

    }

    MPI_Barrier(ACTIVE_COMM);

    if(shift == dims[0] - 1)

    {

        break;

    }

    if (global_rank != global_size-1)

    {

        MPI_Sendrecv(A, NI * NI, MPI_DOUBLE, left, 1, buf, NI * NI, MPI_DOUBLE,
right, 1, cannon_comm, &status);

        tmp = buf;

        buf = A;

        A = tmp;

        MPI_Sendrecv(B, NI * NI, MPI_DOUBLE, up, 2, buf, NI * NI, MPI_DOUBLE,
down, 2, cannon_comm, &status);

        tmp = buf;

```

```
        buf = B;

        B = tmp;

    }

}

MPI_Barrier(ACTIVE_COMM);

end = MPI_Wtime();

if(new_rank == 0)

{

    printf("Time: %.4fs\n", end-start);

}

free(A);

free(B);

free(buf);

free(C);

MPI_Finalize();

return 0;

}
```