Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики

Компьютерный практикум по курсу

“Распределенные системы”

Задания

Алгоритм “Задиры” для выбора координатора.

Доработка MPI-программы, реализованной в рамках курса *“Суперкомпьютеры и параллельная обработка данных”*. Добавление контрольных точек для продолжения работы программы в случае сбоя.

Отчет

о выполненном задании
студента 420 группы факультета ВМК МГУ
Шарипова Саита Равильевича

Лекторы:
профессор, д.ф.-м.н. Крюков Виктор Алексеевич
доцент, к.ф.-м.н. Бахтин Владимир Александрович

Москва, 2021

Содержание

1 Задание №1

1.1 Постановка задачи

Реализовать программу для выбора координатора среди *25* процессов, находящихся на разных ЭВМ сети, использующую алгоритм **«Задиры»**.

Все необходимые межпроцессорные взаимодействия реализовать при помощи *пересылок MPI типа точка-точка*.

Получить временную оценку работы алгоритма. Оценить сколько времени потребуют выборы координатора, если «Задира» имеет уникальный номер *0*. Время старта (время «разгона» после получения доступа к шине для передачи сообщения) равно *100*, время передачи байта равно *1* (Ts = *100*, Tb = *1*). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

1.2 Алгоритмы выбора координатора

Многие распределенные алгоритмы требуют, чтобы один из процессов был координатором, инициатором или выполнял другую специальную роль. Выбор такого специального процесса называется ***выбором координатора***. При этом очень часто бывает неважно, какой именно процесс будет выбран. Обычно выбирается процесс с самым большим уникальным номером. Алгоритмы выбора координатора различаются способами поиска такого процесса. Предполагается, что в начале работы алгоритма процессы не знают, какие из них в настоящее время работают, а какие нет. Алгоритм выбора координатора должен гарантировать, что если процедура выборов началась, то она закончится согласием всех процессов относительно нового координатора.

1.3 Описание алгоритма “Задиры”

Одним из алгоритмов выбора координатора является алгоритм **“Задиры”**. Когда один из процессов замечает, что координатор больше не отвечает на запросы, он инициирует голосование за выбор нового координатора. Процесс ***P*** проводит выборы следующим образом:

1. ***P*** посылает сообщение *“ВЫБОРЫ”* всем процессам с большими номерами, чем у него.
2. Далее возможно два варианта развития событий:

- Если процессу ***P*** не отвечает ни один процесс, то он выигрывает голосование и становится новым координатором.
- Если процессу ***P*** отвечает один из процессов с большим номером, он берет на себя проведение выборов (снова инициирует голосование), а участие процесса ***P*** в выборах заканчивается.

В любой момент процесс может получить сообщение *“ВЫБОРЫ”* от одного из процессов с меньшим номером. В этом случае он посылает ответ *“OK”*, чтобы сообщить, что он работает и берет проведение выборов на себя, а затем инициирует голосование за выборы координатора (если к этому моменту он его не вел). Таким образом, в итоге все процессы кроме одного прекратят выборы, этот последний и будет новым координатором. Он извещает все процессы о своей победе и вступлении в должность координатора сообщением *“КООРДИНАТОР”*.

Если процесс, который находился в нерабочем состоянии начинает работать, то он инициирует голосование.

1.4 Пример работы алгоритма “Задиры”

Предположим, что группа состоит из 8 процессов, пронумерованных от 0 до 7. Ранее координатором был процесс №7, но он завис.

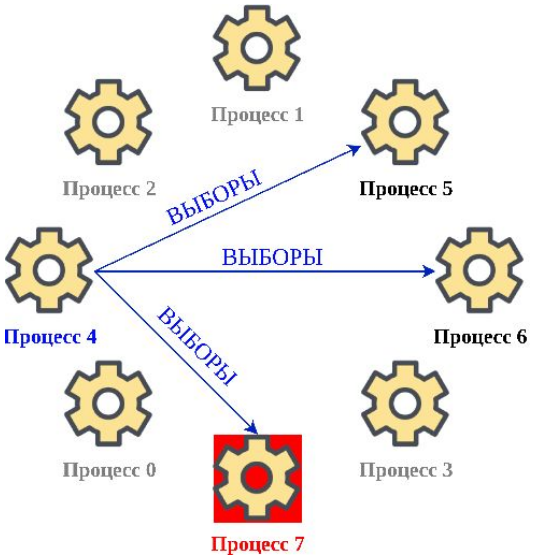
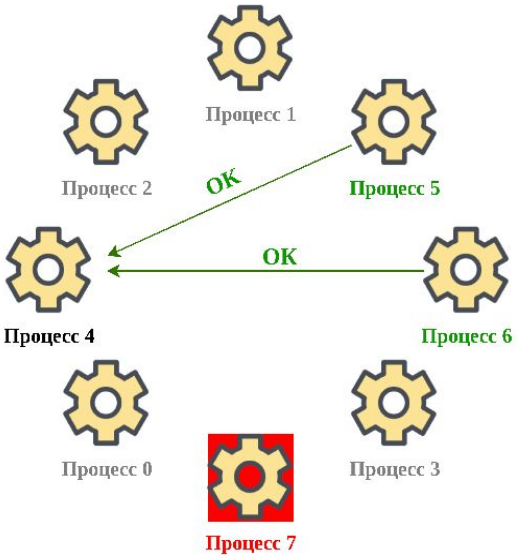


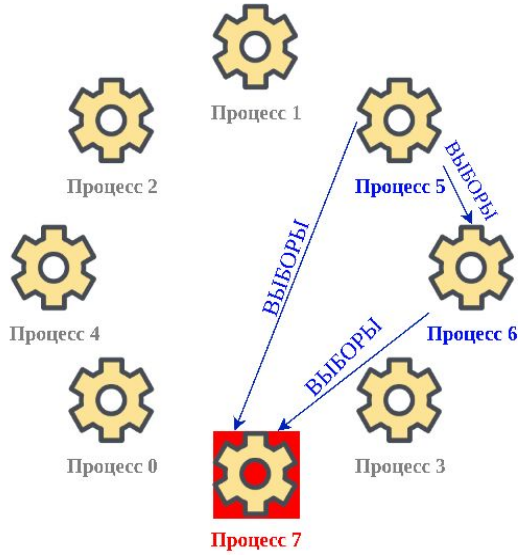
Рисунок 1.

Процесс №4 первым замечает это и посылает сообщение “ВЫБОРЫ” всем процессам с номерами больше, чем у него, то есть процессам №5, №6, №7 как показано на рисунке 1.



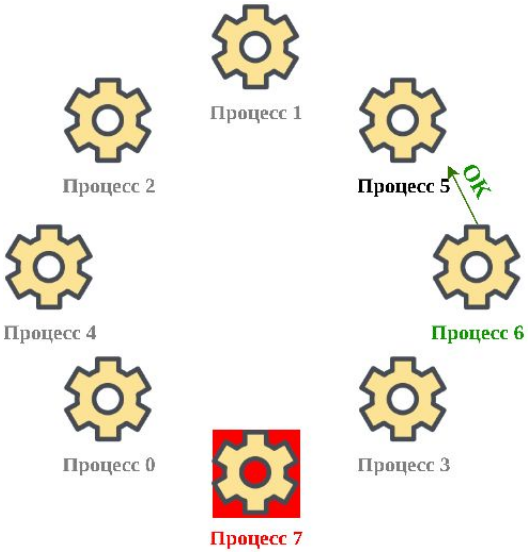
Процессы №5 и №6 работают, поэтому отвечают сообщением “ОК” как показано на рисунке 2. После получения первого из этих ответов процесс №4 прекращает свое участие в выборах.

Рисунок 2.



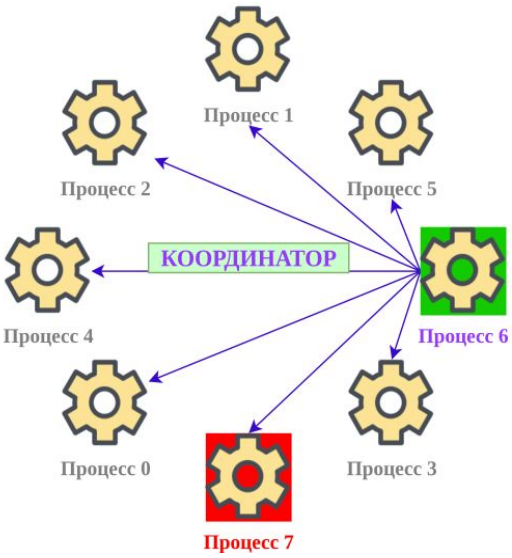
На рисунке 3 процессы №5 и №6 инициируют свои голосования. Каждый посылает сообщения только тем процессам, номера у которых больше их собственных.

Рисунок 3.



Процессы №6 работает, поэтому отвечают сообщением “ОК” процессу №5 как показано на рисунке 4. После получения этого ответа процесс №5 прекращает свое участие в выборах.

Рисунок 4.



В итоге процесс №6 понял, что процесс №7 мертв, а значит победитель в выборах координатора – он сам. На рисунке 5 процесс №6 уведомляет все процессы о том, что он теперь новый координатор.

Рисунок 5.

Когда процесс №4 получит от процесса №6 сообщение “КООРДИНАТОР”, он продолжит работу с той операции, которую пытался выполнить, когда обнаружил, что процесс №7 мертв, использую теперь в качестве координатора процесс №6. Таким образом, после сбоя работы процесса №7 работа продолжилась.

1.5 Реализация алгоритма “Задиры”

Введем следующие обозначения:

- `size` – общее число процессов;
- `start_bully` – номер изначального задиры;
- `current_cordinator` – номер координатора (в работе изначального координатора произошел сбой);

При запуске программы можно задать параметры – {`current_cordinator`, `start_bully`}. В случае, если эти параметры не задаются, то `current_cordinator = size - 1`, `start_bully = 0`.

Далее случайным образом, с вероятностью ½ (вероятность можно задать в программе), некоторые процессы признаются неработающими, при этом изначальный `current_cordinator` признается неработающим, а `start_bully` работающим с вероятностью 1. Все процессы, которые были признаны неработающими завершают работу.
Далее пока идут выборы (цикл `while (election_go)`) происходит следующее:

- Задира отправляет сообщение с тегом **ELECTION**, в котором содержится его номер, всем старшим процессам с помощью `MPI_Isend`. Это сообщение – приглашение на участие в выборах.

- Все работающие процессы пытаются принять сообщение – приглашение на участие в выборах с тегом **ELECTION** от всех процессов с меньшими номерами с помощью `MPI_Irecv`. В случае, если это удается, процесс отправляет сообщение с тегом **OK** в ответ с помощью `MPI_Isend`, причем это обязательно сделать на каждое сообщение – приглашение на участие в выборах, а не только на какое-то одно полученное, иначе может возникнуть ситуация, что какой-нибудь процесс не получил ни одного **OK**, и ошибочно признал себя координатором. Реализация данного пункта представлена на рисунке 6.

```
// Принимаем сообщение для приглашения на выборы
int election_msg;
for (int i = 0; i < rank; i++) { // пытаемся принять приглашение на голосование от всех младших
    MPI_Irecv(&election_msg, count: 1, MPI_INT, i, ELECTION, MPI_COMM_WORLD, &request);
    int flag = false;
    double start_wait = MPI_Wtime();
    while (!flag) {  // пока сообщение "OK" не получено, flag == true -> получили сообщение
        MPI_Test(&request, &flag, &status); // Проверка завершенности асинхронной процедуры MPI_Irecv
        if (MPI_Wtime() - start_wait >= 1) {  // ждем 1 секунду
            // Ничего не получили, поэтому отменяем неблокирующий Recieve
            MPI_Cancel(&request);
            MPI_Request_free(&request);
            break;
        }
    }
    if (flag == true) { // если действительно получили сообщение-приглашение на выборы
        i_am_bully = true; // теперь этот процесс тоже станет задирой
        // Отправляем сообщение "OK" о готовности в выборах задире обратно
        MPI_Isend(&rank, count: 1, MPI_INT, status.MPI_SOURCE, OK, MPI_COMM_WORLD, &request);
        printf( format: "Процесс %d получил приглашение на выборы от процесса %d и отправил ему OK.\n", rank,
               status.MPI_SOURCE);
    }
}
```

Рисунок 6.

- Если процесс не получил ни одного сообщения – приглашения на участие в выборах, то значит его номер меньше, чем у изначального задиры, и он никогда в будущем не получит такого сообщения, поэтому он выходит из цикла.

- Каждый процесс, который когда-либо инициировал голосование ждет сообщение с тегом **OK**, от какого либо процесса (**MPI_ANY_SOURCE**) с помощью `MPI_Irecv`. Если удается принять такое сообщение, то процесс прекращает участие в выборах и выходит из цикла. Иначе процесс объявляет себя координатором и отправляет всем

процессам сообщение с тегом **COORDINATOR**, в котором содержится номер его номер, с помощью MPI_Isend, и также выходит из цикла, завершая свою работу.

- После цикла выборов, каждый работающий процесс, который не стал координатором ждет сообщение с тегом **COORDINATOR**, с номером нового координатора от любого процесса (**MPI_ANY_SOURCE**) с помощью MPI_Recv, после получения этого сообщения процесс завершает свою работу.

1.6 Пример вывода реализованного алгоритма “Задиры”

На рисунке 7 приведен пример вывода реализованного алгоритма “Задиры” для выбора координатора среди *25* процессов, в случае, если изначальным координатором, в работе которого произошел сбой, был процесс *№24*, а задирой является процесс *№17*.

```
Нерабочие процессы перед голосованием: 14 10 16 8 5 9 24 0 2 7 13 19 23

В голосовании участвуют процессы: 18 20 22 15 17 21 4 6 12 1 3 11

Процесс 17 - задира и начал голосование.
Процесс 22 получил приглашение на выборы от процесса 17 и отправил ему OK.
Процесс 20 получил приглашение на выборы от процесса 17 и отправил ему OK.
Процесс 21 получил приглашение на выборы от процесса 17 и отправил ему OK.
Процесс 17 получил сообщение OK от процесса 20 и заканчивает участие в выборах.
Процесс 18 получил приглашение на выборы от процесса 17 и отправил ему OK.
Процесс 18 - задира и начал голосование.
Процесс 21 получил приглашение на выборы от процесса 18 и отправил ему OK.
Процесс 20 получил приглашение на выборы от процесса 18 и отправил ему OK.
Процесс 22 получил приглашение на выборы от процесса 18 и отправил ему OK.
Процесс 22 получил приглашение на выборы от процесса 20 и отправил ему OK.
Процесс 22 получил приглашение на выборы от процесса 21 и отправил ему OK.
Процесс 22 - задира и начал голосование.
Процесс 21 получил приглашение на выборы от процесса 20 и отправил ему OK.
Процесс 21 - задира и начал голосование.
Процесс 20 - задира и начал голосование.
Процесс 18 получил сообщение OK от процесса 20 и заканчивает участие в выборах.
Процесс 20 получил сообщение OK от процесса 21 и заканчивает участие в выборах.
Процесс 21 получил сообщение OK от процесса 22 и заканчивает участие в выборах.
Процесс 1 получил сообщение - новый координатор процесс 22
Процесс 6 получил сообщение - новый координатор процесс 22
Процесс 22 победил в выборах и стал новым координатором.
Процесс 11 получил сообщение - новый координатор процесс 22
Процесс 12 получил сообщение - новый координатор процесс 22
Процесс 17 получил сообщение - новый координатор процесс 22
Процесс 20 получил сообщение - новый координатор процесс 22
Процесс 18 получил сообщение - новый координатор процесс 22
Процесс 15 получил сообщение - новый координатор процесс 22
Процесс 4 получил сообщение - новый координатор процесс 22
Процесс 3 получил сообщение - новый координатор процесс 22
Процесс 21 получил сообщение - новый координатор процесс 22
```

Рисунок 7.

1.7 Инструкция для запуска реализованной программы

Код реализованной программы доступен во вложении к данному решению или по ссылке github, в файле bully.c.

Для запуска программы с использованием *25* процессов требуется выполнить следующие команды:

- Без задания параметров:
  - *mpicc bully.c -o bully*
  - *mpirun -np 25 --oversubscribe ./bully*
- С заданием параметров {`current_cordinator`, `start_bully`}:
  - *mpicc bully.c -o bully*
  - *mpirun -np 25 --oversubscribe ./bully 24 17*

1.8 Временная оценка работы алгоритма “Задиры”

Оценим сколько времени потребуют выборы координатора, если «Задира» имеет уникальный номер *0* (считая, что все процессы работают). Время старта (время «разгона» после получения доступа к шине для передачи сообщения) равно *100*, время передачи байта равно *1* (Ts = *100*, Tb = *1*). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

Введем обозначения:

- $L_{выборы}$ – длина сообщения “ВЫБОРЫ” в байтах.
- $L_{ок}$ – длина сообщения “ОК” в байтах.
- $L_{координатор}$ – длина сообщения “КООРДИНАТОР” в байтах.

Тогда:

- $Ts + Tb \cdot L_{выборы}$ – время передачи одного сообщения “ВЫБОРЫ”.
- $Ts + Tb \cdot L_{ок}$ – время передачи одного сообщения “ОК”.
- $Ts + Tb \cdot L_{координатор}$ – время передачи одного сообщения “КООРДИНАТОР”.

Заметим, что если задира имеет уникальный номер *0*, то каждый процесс номер i отправит сообщение “ВЫБОРЫ” каждому процессу с номером j где $j > i$. Следовательно всего будет ровно *(1 + 2 + … + 23 + 24)* сообщений “ВЫБОРЫ”.

Также каждый процесс номер *i* отправит сообщение “ОК” каждому процессу с номером *j* где $j < i$. Следовательно всего будет ровно $(1 + 2 + ... + 23 + 24)$ сообщений “ОК”.

В конце координатором станет процесс номер *24*, и он отправит всем остальным процессам сообщение “КООРДИНАТОР”. Следовательно всего будет ровно *24* сообщения “КООРДИНАТОР”.

В результате время работы алгоритма “Задиры” T в данном случае будет рассчитываться по формуле:

$$T = (1 + 2 + ... + 23 + 24) \cdot (Ts + Tb \cdot L_{выборы}) +$$
$$(1 + 2 + ... + 23 + 24) \cdot (Ts + Tb \cdot L_{ок}) +$$
$$24 \cdot (Ts + Tb \cdot L_{координатор}).$$

Учитывая, что в приведенной реализации $L_{выборы} = L_{ок} = L_{координатор} = 4$ байта, а по условию $Ts = 100, Tb = 1$, получаем время работы алгоритма “Задиры”:

$$T = (1 + 2 + ... + 23 + 24) \cdot (100 + 1 \cdot 4) +$$
$$(1 + 2 + ... + 23 + 24) \cdot (100 + 1 \cdot 4) +$$
$$24 \cdot (100 + 1 \cdot 4) = \mathbf{64\ 896}.$$

2 Задание №2

2.1 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:

a) продолжить работу программы только на “исправных” процессах;

b) вместо процессов, вышедших из строя, создать новые MPI-процессы, которые необходимо использовать для продолжения расчетов;

c) при запуске программы на счет сразу запустить некоторое дополнительное количество MPI-процессов, которые использовать в случае сбоя.

2.2 Описание последовательной версии программы

```
// Основная функция (последовательных) вычислений над массивами A и B
static void kernel_heat_3d(int t_steps, int n, double A[n][n][n], double B[n][n][n]) {

    for (int t = 1; t <= t_steps; t++) { // t_steps раз вычисляем

        // Массив B вычисляется через массив A
        for (int i = 1; i < n - 1; i++) {
            for (int j = 1; j < n - 1; j++) {
                for (int k = 1; k < n - 1; k++) {
                    B[i][j][k] = 0.125 * (A[i + 1][j][k] - 2.0 * A[i][j][k] + A[i - 1][j][k])
                                + 0.125 * (A[i][j + 1][k] - 2.0 * A[i][j][k] + A[i][j - 1][k])
                                + 0.125 * (A[i][j][k + 1] - 2.0 * A[i][j][k] + A[i][j][k - 1])
                                + A[i][j][k] + 1.0; // добавлено +1.0, иначе программа не изменяет значений массивов
                }
            }
        }

        // Массив A вычисляется через массив B
        for (int i = 1; i < n - 1; i++) {
            for (int j = 1; j < n - 1; j++) {
                for (int k = 1; k < n - 1; k++) {
                    A[i][j][k] = 0.125 * (B[i + 1][j][k] - 2.0 * B[i][j][k] + B[i - 1][j][k])
                                + 0.125 * (B[i][j + 1][k] - 2.0 * B[i][j][k] + B[i][j - 1][k])
                                + 0.125 * (B[i][j][k + 1] - 2.0 * B[i][j][k] + B[i][j][k - 1])
                                + B[i][j][k] + 2.0; // добавлено +2.0, иначе программа не изменяет значений массивов
                }
            }
        }

    }
}
```

Рисунок 8.

Данная программа (файл heat3d_basic.c) производит вычисления для задачи *"Уравнение теплопроводности в трехмерном пространстве"* (*"Heat equation over 3D data domain"*).

Изначально происходит инициализация массивов *A* и *B* с помощью функции `init_array (int n, double A[n][n][n], double B[n][n][n])`.

Основной вычисляющей функцией программы, код которой представлен на рисунке 8, является `kernel_heat_3d(int t_steps, int n, double A[n][n][n], double B[n][n][n])`. В данной функции *t* раз выполняются два тройных цикла. В первом тройном цикле обновляется массив *B*, полагаясь на значения в массиве *A*. Во втором тройном цикле обновляется массив *A*, полагаясь на обновленные в предыдущем тройном цикле значения в массиве *B*.

Функция `print_array(int n, double A[n][n][n], FILE *file_for_arrays)` записывает вычисленные массивы *A* и *B* в файл.

2.3 Описание параллельной версии программы

Параллельная версия программы (файл heat3d_MPI.c) функционирует следующим образом:

- Изначально были упрощены выражения для вычислений массивов *A* и *B*, чтобы уменьшить количество вычислений.
- Процесс *№0* является ***процессом-мастером***, не участвующим в вычислениях массивов *A* и *B*, а занимающийся сборкой итоговых результатов от рабочих процессов по тегу **`RESULTS`** с помощью `MPI_Recv` и выводом информации об этапах выполнения программы в консоль (функция `receive_results(...)`).
- Остальные процессы являются ***рабочими***, вычисляя части массивов *A* и *B*. После окончания вычислений они отправляют свои результаты процессу-мастеру по тегу **`RESULTS`** с помощью `MPI_Send`. (функция `send_results(...)`)
- Трехмерные массивы `A_all[n][n][n]` и `B_all[n][n][n]` равномерно разделяются по первому измерению на части по `n_plane` плоскостей (рисунок 9) так, чтобы каждый рабочий процесс производил вычисления над их частями `A[n_plane + 2][n][n]` и `B[n_plane + 2][n][n]`. В первом измерении `n_plane + 2` плоскостей для того,

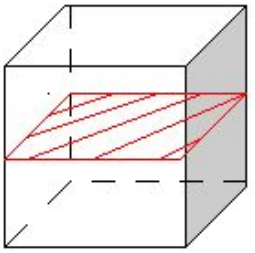


Рисунок 9.

чтобы рабочие процессы могли принимать необходимые плоскости от предыдущего и следующего процесса при вычислениях.

- Отправка и прием необходимых плоскостей для вычисления осуществляется при помощи `MPI_Isend` и `MPI_Recv` соответственно с тегом **`MSG_PLANE`**.

- После сборки итоговых результатов процесс-мастер вызывает функцию `print_array(int n, double A[n][n][n], FILE *file_for_arrays)` для записи вычисленных массивов *A* и *B* в файл.

2.4 Описание устойчивой к сбоям параллельной версии программы

Параллельная версия программы, устойчивая к сбоям (файл heat3d_MPI_FT.c) функционирует следующим образом:

- Для того, чтобы при сбое одного из процессов программа не завершалась с ошибкой, а продолжала своё выполнение, ипользовалась функция из стандарта MPI `MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);`

- Был реализовать сценарий работы после сбоя, в котором при запуске программы сразу запускается один резервный MPI-процесс, который будет использоваться в случае сбоя. В качестве резервного используется последний процесс.

- Вся программа разделена на вычисляющие блоки, в которые можно попасть с помощью оператора `goto`.

- Номер процесса, в работе которого должен произойти сбой – `rank_to_damage`, блок, в котором должен произойти сбой – `block_to_damage` и шаг вычислений на котором должен произойти сбой – `t_step_to_damage` задаются случайным образом в начале работы программы. Номер процесса, в котором должен произойти сбой, не может равняться нулевому процессу (процесс-мастер) и последнему (резервный процесс).

- Сохранение данных осуществляется в контрольных точках в конце каждого блока (функция `save_control_point(...)`).

- Для того, чтобы при подключении к вычислениям резервный процесс смог загрузить последнюю контрольную точку, название файлов для сохранения вычислений рабочих процессов имеет шаблон `"control_point_[rank]_[t_step]_[block].txt"`.

- В начале каждого блока вызывается функция `try_to_suicide(int rank, int curr_block, int curr_t_step)`, которая сравнивает текущие значения номера процесса, номера блока и номера шага вычислений с теми, при достижении которых в работе процесса должен произойти сбой.

- В случае совпадения значений, описанных в предыдущем пункте, в работе процесса происходит сбой, реализованный с помощью `raise(SIGTERM)`.

- По коду возвратов функций `MPI_Recv`, которые использовались в изначальной параллельной версии программы, процессы могут понять, что в работе процесса, от которого ожидается получение данных, произошла ошибка. В этом случае процесс, который заметил сбой, отправляет сообщение с номером процесса, который следует заменить, и тегом **`RECOVERY`** резервному процессу с помощью `MPI_Isend`. После отправки процесс прыгает на начало блока, в котором ему не удалось получить данные с помощью оператора `goto`.

- Так как поломку процесса мог заметить любой из остальных процессов, то резервному процессу требуется ждать сообщение с номером процесса, в работе которого произошел сбой, от любого процесса. Однако `MPI_Recv` с получением сообщения от любого процесса с помощью **`MPI_ANY_SOURCE`** не работает ожидаемым образом (в качестве сообщения приходит непредсказуемое значение от непредсказуемого отправителя). Поэтому перед принятием сообщения о поломке процесса требовалось выяснить номер процесса, от которого следует его ждать. Для достижения этой цели использовался цикл, в котором вызывалась функция `MPI_Irecv` с тегом **`TEST_DEAD`** от всех процессов по очереди и функция `MPI_Waitany`, после цикла. В результате по индексу, полученному в `MPI_Waitany`, устанавливался поломанный процесс (так как `MPI_Irecv` для этого процесса завершается с ошибкой), а по нему уже без труда устанавливался процесс, от которого требуется ждать сообщения об умершем процессе – так как каждый рабочий процесс в параллельной программе взаимодействовал только с соседними процессами, то можно ждать сообщения от левого соседа умершего процесса.

- Узнав номер процесса, от которого следует ожидать сообщение о умершем процессе, резервный процесс выполняет от него `MPI_Irecv` с тегом **`RECOVERY`**.

- Для того, чтобы заменить процесс, в работе которого произошел сбой, резервный процесс загружает данные из его последней контрольной точки (функция

`load_control_point(...)`), подменяет свой `rank` номером умершего процесса и с помощью оператора `goto` прыгает в соответствующую точку вычислений, в которой произошел сбой в работе умершего процесса.

- После сборки итоговых результатов процесс-мастер вызывает функцию `print_array(int n, double A[n][n][n], FILE *file_for_arrays)` для записи вычисленных массивов A и B в файл. Если сбой в работе какого-либо процесса произошел на этапе сборки результатов, резервный процесс также способен заменить умерший процесс и сборка завершится успешно.

2.5 Инструкция для запуска реализованной программы

Код описанных программ доступен во вложении к данному решению или по ссылке github. Описанные программы находятся в следующих файлах:

- heat3d_basic.c – последовательная версия программы.
  Для запуска программы требуется выполнить следующие команды:
  - Без задания параметров:
    - *mpicc heat3d_basic.c*
    - *mpirun --oversubscribe -n 1 a.out*
  - С заданием параметров {`t_steps`, `n`}, где `t_steps` – число шагов вычислений, а `n` – размер измерения 3х-мерной кубической матрицы:
    - *mpicc heat3d_basic.c*
    - *mpirun --oversubscribe -n 1 a.out 10 24*
- heat3d_MPI.c – параллельная версия программы.
  Для запуска программы с использованием *7* процессов (*1* процесс-мастер и *6* рабочих процессов) требуется выполнить следующие команды:
  - Без задания параметров:
    - *mpicc heat3d_MPI.c*
    - *mpirun --oversubscribe -n 1 a.out*
  - С заданием параметров {`t_steps`, `n`}, где `t_steps` – число шагов вычислений, а `n` – размер измерения 3х-мерной кубической матрицы:
    - *mpicc heat3d_MPI.c*
    - *mpirun --oversubscribe -n 7 a.out 10 24*

- heat3d_MPI_FT.c – параллельная версия программы, устойчивая к сбоям.

  При работе программы создаются файлы контрольных точек, которые необходимо удалять перед каждым новым запуском.

  Для запуска программы с использованием *8* процессов (*1* процесс-мастер, *6* рабочих процессов и 1 резервный процесс) требуется выполнить следующие команды:

  - Без задания параметров:
    - *source dockervars.sh*
    - *mpicc heat3d_MPI_FT.c*
    - *mpirun --oversubscribe -n 8 a.out*
  - С заданием параметров {`t_steps`, `n`}, где `t_steps` – число шагов вычислений, а `n` – размер измерения 3х-мерной кубической матрицы:
    - *source dockervars.sh*
    - *mpicc heat3d_MPI_FT.c*
    - *mpirun --oversubscribe -n 8 a.out 10 24*

Для проверки работы параллельной программы и параллельной программы, устойчивой к сбоям, можно сравнить выходные файлы, в которых записаны вычисленные массивы *A* и *B*. Для этого в командной строке можно выполнить команды:

- *diff output_calc_basic.txt output_calc_MPI.txt*
- *diff output_calc_basic.txt output_calc_MPI_FT.txt*

Где:

- *output_calc_basic.txt* – выходной файл с вычислениями последовательной версии программы;
- *output_calc_MPI.txt* – выходной файл с вычислениями параллельной версии программы;
- *output_calc_MPI_FT.txt* – выходной файл с вычислениями параллельной версии программы, устойчивой к сбоям;

3 Заключение

В результате данной практической работы были выполнены задачи:

- Алгоритм “Задиры” для выбора координатора.
- Доработка MPI-программы, реализованной в рамках курса “Суперкомпьютеры и параллельная обработка данных”. Добавление контрольных точек для продолжения работы программы в случае сбоя.

При реализации одного из алгоритмов выбора координатора – алгоритма “Задиры”, на практике был изучен способ выбора нового координатора в случае, если в работе прежнего координатора произошел сбой. Также была получена временная оценка алгоритма “Задиры”.

При доработке MPI-программы, реализованной в рамках курса “Суперкомпьютеры и параллельная обработка данных”, таким образом, чтобы она была устойчивая к сбоям, был изучен подход, позволяющий продолжить и успешно завершить вычисления в том случае, если в работе одного из рабочих процессов произошел сбой.