Отчёт

1. Реализация MPI_Scan

1.1. Задача

Реализовать программу, моделирующую выполнение операции MPI_Scan для транспьютерной матрицы размером 4*4 при помощи пересылок MPI типа точка-точка. В каждом узле транспьютерной матрицы запущен один MPI-процесс. Каждый процесс имеет свое число. В результате выполнения операции MPI_Scan каждый i-ый процесс должен получить сумму чисел, которые находятся у процессов с номерами 0, ... i включительно. Оценить сколько времени потребуется для выполнения операции MPI_Scan, если все процессы выдали эту операцию редукции одновременно. Время старта 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. Алгоритм

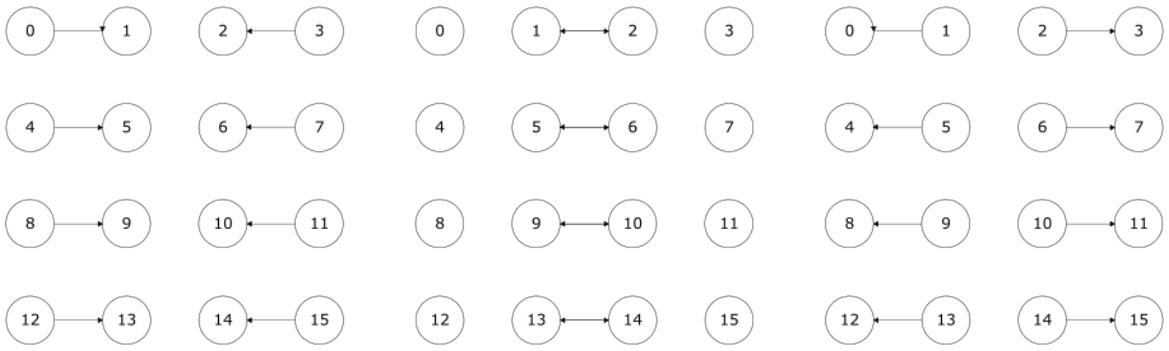
Заметим, что

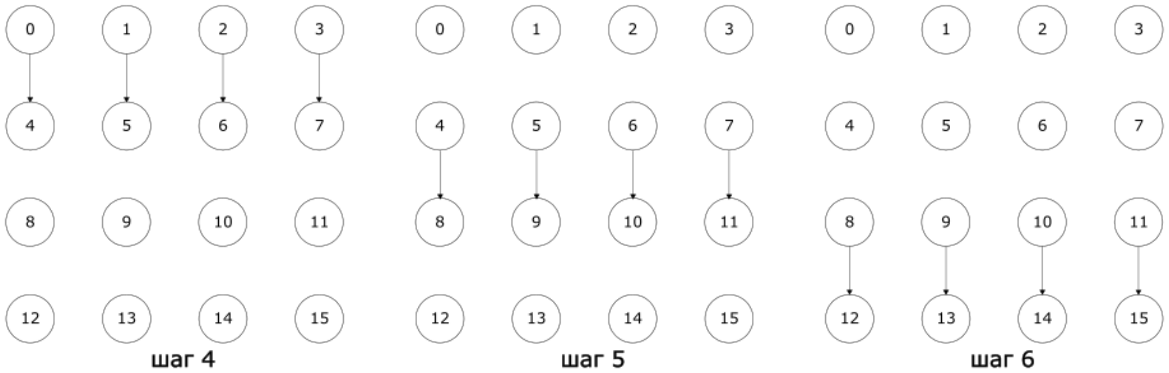
$\sum_{j=0}^{i} a_j = \sum_{k=0}^{p-1} \sum_{j=0}^{3} a_{4 \cdot k+j} + \sum_{j=0}^{q} a_j$, где $i = 4 \cdot p + q$, т.е. сумма – это сумма всех вышележащих строк транспьютерной матрицы (1) и суммы всех элеметов (2), стоящих слева (включая данный элемент), таким образом приходим к следующей стратегии:

1) Сначала считаем в каждом элементе строки считаем сумму всей строки и сумму (2)

2) Передаём значения сумм строк нижестоящим элементам

Получаются следующие шаги алгоритма:





Все передачи на одном шаге производятся одновременно.

1.3. Оценка времени

Время для матрицы $N \times M$:
$T(N, M) = (M - 1 + 2 \cdot \left[\frac{N-1}{2}\right] + 1 - N\%2) \cdot (Ts + B \cdot Tb)$, где B – максимальный размер чисел (байтовая длина), а $[\ ]$ – округление вниз.
$T(4, 4) = (2 \cdot 3 + 4 - 1) \cdot (100 + 4 \cdot 1) = 936$

1.4. Исходный код

Реализацию алгоритма можно найти тут:
https://github.com/hkctkuy/Parallel-programming/blob/main/3-mpi-scan/scan.c

1.5. Результаты

```
mpicc scan.c -o scan
mpiexec -np 16 --oversubscribe ./scan
```

```
[Process 4]: has local num 4
[Process 3]: has local num 3
[Process 2]: has local num 2
[Process 9]: has local num 9
[Process 6]: has local num 6
[Process 7]: has local num 7
[Process 5]: has local num 5
[Process 10]: has local num 10
[Process 1]: has local num 1
[Process 12]: has local num 12
[Process 0]: has local num 0
[Process 11]: has local num 11
[Process 15]: has local num 15
[Process 8]: has local num 8
[Process 14]: has local num 14
[Process 13]: has local num 13
[Process 2]: has received sum: 3
[Process 1]: has received sum: 1
[Process 3]: has received sum: 6
[Process 5]: has received sum: 15
[Process 7]: has received sum: 28
[Process 0]: has received sum: 0
[Process 9]: has received sum: 45
[Process 13]: has received sum: 91
[Process 4]: has received sum: 10
[Process 6]: has received sum: 21
[Process 10]: has received sum: 55
[Process 8]: has received sum: 36
[Process 11]: has received sum: 66
[Process 15]: has received sum: 120
[Process 12]: has received sum: 78
[Process 14]: has received sum: 105
```

2. Улучшение MPI программы

2.1. Задача

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

2.2. Алгоритм

MPI-программа, разработанная в рамках курса “Суперкомпьютеры и параллельная обработка данных”, реализовывала параллельной версии программы для умножения матрицы на вектор.

Отчёт по данной задаче можно найти по следующей ссылке:
https://github.com/hkctkuy/Parallel-programming/blob/main/report-matrix-vector.pdf
А саму реализацию здесь:
https://github.com/hkctkuy/Parallel-programming/blob/main/2-matrix-vector-mpi/mpi.c

Сначала создадим новый слой: всю логику, реализующую параллельный слой вынесем в отдельную функцию. Теперь создадим обработчик MPI ошибок, который будет вызывать функцию MPIX_Comm_shrink(), создающей новый коммутатор для оставшихся процессов, и передать управление с помощью longjmp() назад к вызову основной функции, начиная вычисления заново:

```
static void errhandler(MPI_Comm *comm, int *err, ...) {
    // Do not kill anyone else
    TO_KILL = -1;

    int len;
    char errstr[MPI_MAX_ERROR_STRING];

    MPI_Error_string(*err, errstr, &len);
    errstr[len] = 0;

    printf("Captured error: %s\n", errstr);

    MPIX_Comm_shrink(*comm, &MPI_COMM_CUSTOM);
    MPI_Barrier(MPI_COMM_CUSTOM);

    longjmp(jbuf, 0);
}
```

Далее нужно задать нужную функцию в качестве обработчика:

```
    // Initialize MPI
    MPI_Init(&argc, &argv);
    MPI_COMM_CUSTOM = MPI_COMM_WORLD;
    // Add error handler
    MPI_Errhandler errh;
    MPI_Comm_create_errhandler(errhandler, &errh);
    MPI_Comm_set_errhandler(MPI_COMM_CUSTOM, errh);
```

2.3. Исходный код

Реализацию алгоритма можно найти тут:
https://github.com/hkctkuy/Parallel-programming/blob/main/4-matrix-vector-mpi-fault-tolerance/mpi.c

2.4. Результаты

```
SIZE=2 NP=2 ./mpi_run.sh
```

```
[Process 1]: has been killed
Captured error: MPI_ERR_PROC_FAILED: Process Failure
Time: 0.000001
Result:
1.000000
2.000000
```