МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Практикум по учебному курсу

"Распределенные системы"

Алгоритм MPI_GATHER для транспьютерной матрицы

Раработка отказоустойчивой параллельной версии программы для задачи Gauss

Отчет

студента 421 группы

факультета ВМК МГУ

Шапошникова Владимира Александровича

Москва 2020

Оглавление

1 Постановка задачи

Требуется разработать и реализовать:

1. Реализовать программу, моделирующую выполнение операции MPI_GATHERV на транспьютерной матрице при помощи пересылок MPI типа точка-точка. Получить временную оценку работы алгоритма. Оценить сколько времени потребуется для выполнения операции MPI_GATHERV, если все процессы выдали ее 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.
2. 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 Структура проекта

2.1 FirstTask

Директория FirstTask содержит наивную и быструю реализацию программы (файлы naive.cpp main.cpp), моделирующий MPI_GATHERV для транспьютерной матрицы 4x4. Директория cmake-build-debug содержит Makefile. Для запуска требуется проделать следующие команды:

- cd cmake-build-debug
- make
- mpirun -np 16 –oversubscribe ./Normal (oversubscribe параметр в случае недостаточного количества ядер).

2.2 SecondTask

Директория SecondTask содержит реализацию отказоустойчивой программы Gauss (main.c), Makefile, а также два скрипта, реализующие сборку и запуск программы. Обязательно требуется проделать следующую команду:

- docker pull abouteiller/mpi-ft-ulfm

  Также для удобства можно проделать следующие команды:

- alias make='docker run -v $PWD:/sandbox:Z abouteiller/mpi-ft-ulfm make'
- alias mpirun='docker run -v $PWD:/sandbox:Z abouteiller/mpi-ft-ulfm mpirun --oversubscribe -mca btl tcp,self'

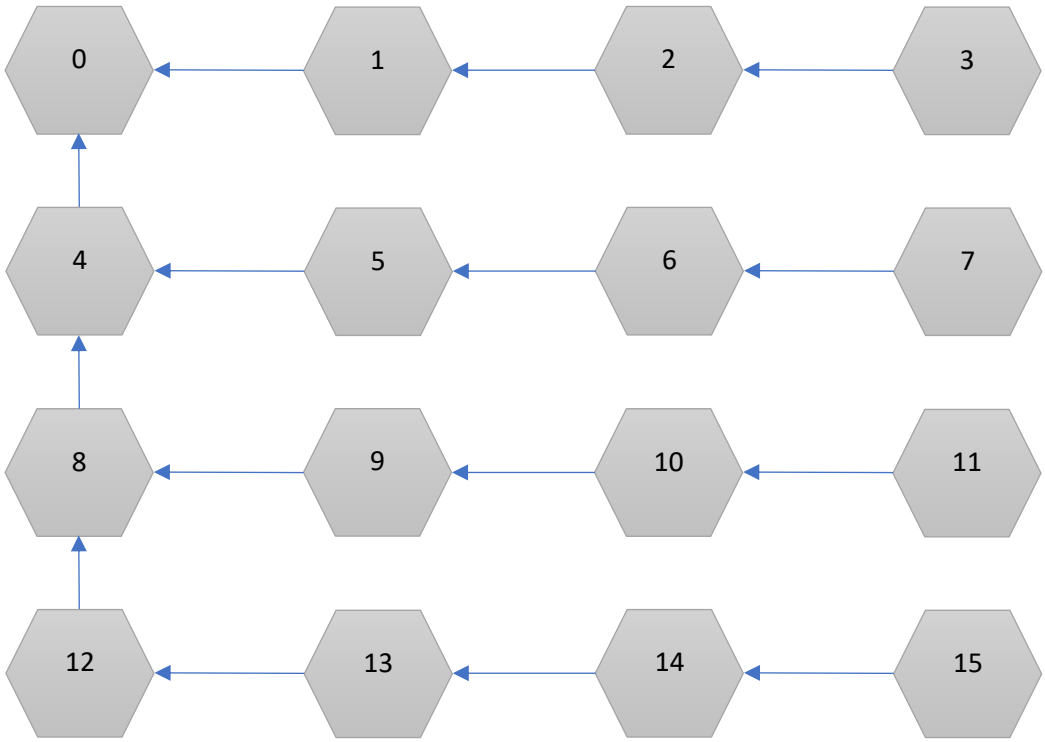
В результате их выполнение make и mpirun будут выполняться на докерах (без использования скрипта).

3 MPI_GATHERV для транспьютерной матрицы 4x4

3.1 Постановка задачи

Реализовать программу, моделирующую выполнение операции MPI_GATHERV на транспьютерной матрице при помощи пересылок MPI типа точка-точка. Получить временную оценку работы алгоритма.

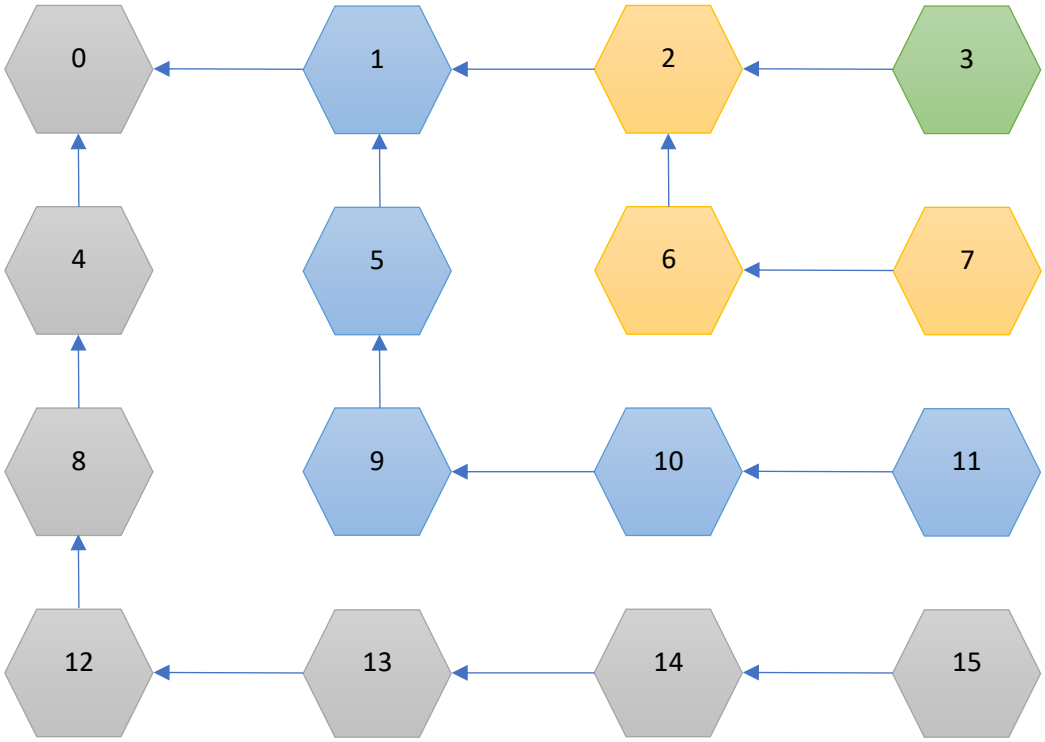
3.2 Наивный алгоритм



Существует наивный алгоритм, который подразумевает последовательную пересылку сообщений. Количество входных канало – 2, передающих узлов – 15, значит минимальное количество тактов передачи – 8. Следовательно, временная оценка будет следующей:

$$T_{naive}(4,4) = 8 * (T_{st} + B * T_b) = 8 * (100 + 4 * 1) = 832$$

3.3 Улучшенный алгоритм



Улучшенный алгоритм подразумевает получение значений в собирающих узлах для матриц $T_{3,3}$, $T_{2,2}, T_{1,1}$ в узлах 1, 2, 3 соответственно. Чтобы рассчитать время требуется найти максимум между временной оценки для $T_{3,3}$ и T^4_{snake} (пересылка с 15->14->13->12->8->4->0 для матрицы $T_{4,4}$). В данном случае (2 * N – 2) – количество передающих узлов в змейке.

$$T_{N,N} = \max\left(T^N_{snake}, T_{N-1,\ N-1}\right), \qquad N \geq 1$$

$$T^N_{snake} = \ (2 * N - 2) \ * (T_{st} + B * T_b)$$

$$T_{1,1} = \ (T_{st} + B * T_b) \ = 100 + 4 * 1 = 104$$

В таком случае временная оценка:

$$T_{1,1} = \ (T_{st} + B * T_b) \ = 100 + 4 * 1 = 104$$

$$T^2_{snake} = \ \ 2 \ * (100 + 4 * 1) = 208$$

$$T_{2,2} = \ \max\left(T^2_{snake}, T_{1,1}\right) = 208$$

$$T^3_{snake} = \ \ 4 \ * (100 + 4 * 1) = 416$$

$$T_{3,3} = \ \max\left(T^3_{snake}, T_{2,2}\right) = 416$$

$$T^4_{snake} = \ \ 6 \ * (100 + 4 * 1) = 624$$

$$T_{4,4} = \max\left(T^4_{snake}, T_{3,3}\right) = 624 < T_{naive}(4,4) = 832$$

4 Реализация отказоустойчивой программы Gauss

Для реализации отказоустойчивой версии был выбран метод, перераспределяющий нагрузку на ‘исправных’ процессах в случае сбоя.

Для этого рассмотрим основные функции

4.1 data_save

Данная функция отвечает за запись данных в бинарные файлы. Так как все процессы имеют одинаковые данные, то записью занимается процесс с myrank = 0. Остальные же процессы ждут завершения процесса записи с помощью MPI_Barrier.

4.2 data_load

В случае сбоя, данные загружаются из бинарных файлов в память каждого процесса. Это сделано для того, чтобы никакой процесс не имел на каком-то шаге наполовину обновленную матрицу. Матрицы должны быть сброшены к предыдущему шагу. После загрузки процессы ждут друг друга с помощью MPI_Barrier.

4.3 verbose_errhandler

Данная функция описывает реакцию процессов в случае сбоя. В результате сбоя каждый процесс должен обновить свою рабочую группу (удалить из неё мёртвый процесс с помощью `MPIX_Comm_shrink`) и после этого сделать data_load, чтобы иметь правильные данные на момент начала чекпоинта.

4.4 Организация чекпоинта

```
1.  first_itter = true;
2.  while (err_happens || first_itter) {
3.      err_happens = false;
4.      …….
5.      first_itter = false;
6.      if (!err_happens)
7.          MPI_Barrier(main_comm);
8.  }
9.  data_save();
10. MPI_Barrier(main_comm);
```

В случае если в данном блоке возникает ошибка, то флаг err_happens будет равен true. В результате данный блок будет выполнен ещё раз с данными, которые были до цикла (в силу data_load). Если блок будет завершён успешно, то дожидается завершения операций во всём блоке, после чего идёт обновление бинарного файла.

5 Основные ссылки

1. https://github.com/Volodimirich/SkiPod2 - репозиторий на github.
2. https://fault-tolerance.org/2018/11/08/ulfm-2-1a1-docker-package/ - docker ulfm.