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

Копмьютерный практикум по учебному курсу

«РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ»

ЗАДАНИЯ

Круговой алгоритм выбора координатора

Доработка MPI программы, реализованной в рамках курса ”Суперкомпьютеры и параллельная обработка данных”с целью улучшить надежность

ОТЧЕТ

о выполненном задании

студента 421 учебной группы факультета ВМК МГУ

Никифорова Н.И.

Москва
2021

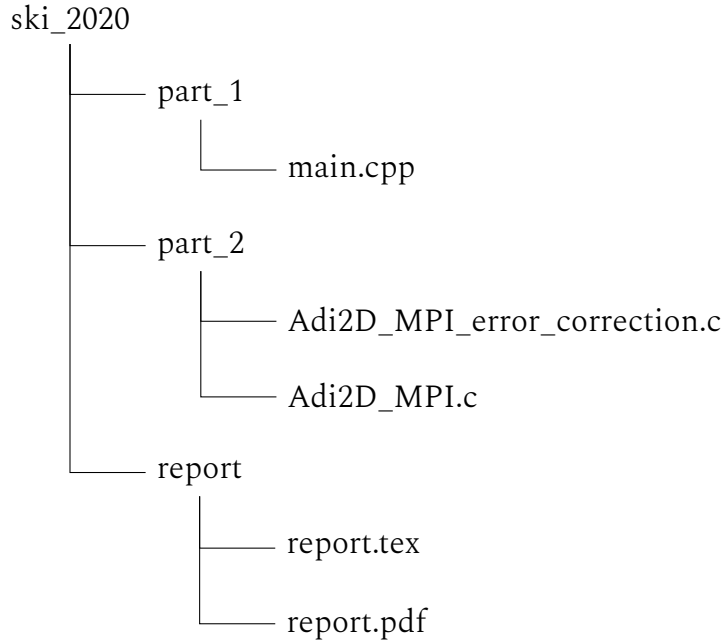
Постановка задачи

Необходимо было выполнить следующие два пункта:

- В транспьютерной матрице размером 5*5, в каждом узле которой находится один процесс, необходимо переслать очень длинное сообщение (длиной L байт) из узла с координатами (0,0) в узел с координатами (4,4). Реализовать программу, моделирующую выполнение такой пересылки на транспьютерной матрице с использованием блокирующих операций MPI. Получить временную оценку работы алгоритма, если 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.
- 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.

Структура проекта

Код проекта доступен в открытом репозитории по ссылке **https://github.com/Rav263/ski_2020**, код предыдущего задания находится по ссылке **https://github.com/Rav263/SKI_adi2d**. Проект состоит из двух частей, для каждого пункта задачи соответственно.



Сборка проекта

Первый пункт (блокирующий алгоритм передачи сообщения)

```
cd ./part_1
mpic++ main.cpp
mpirun -n 25 --oversubscribe ./a.out
```

Второй пункт (Улучшение существующего алгоритма Adi2D)

Для сборки второй части программы требуется установленный пакет ulfm (Можно найти на сайте **fault-tolerance.org**). Сборка ulfm (флаги для **configure** взяты из системного mpi:

```
git clone https://bitbucket.org/icldistcomp/ulfm2.git
cd ulfm2
git submodule update --init --recursive
./autogen.pl
./configure '--prefix=/'usr '--sysconfdir=/etc/'openmpi\'
            --enable-mpi-fortran='all\'
            --libdir=/usr/lib/'openmpi\'
            --enable-builtin-'atomics '--enable-mpi-'cxx\'
            --with-'valgrind '--enable-'memchecker\'
            --enable-pretty-print-'stacktrace\'
            --without-'slurm '--with-hwloc=/'usr\'
            --with-libltdl=/'usr 'FC=/usr/bin/'gfortran
make all
sudo make install
```

Сборка алгоритма:

```
cd ./part_2
mpicc Adi2D\_MPI\_error\_correction.c
mpirun -n 6 --oversubscribe ./a.out
```

Тестовый стенд

- Процессор — AMD Ryzen 2700X 8/16 (ядер/потоков) 4.2 Ghz на одно ядро, 4.0 Ghz на все ядра.
- Память — 32Gb DDDR4 3200Mhz (Пропускная способность 44GB/s).

Алгоритм блокирующей пересылки сообщения на транспьютерной матрице

На рисунке 1 изображена транспьютерная матрица, желтым цветом обозначены стартовая и конечная вершина, красным и зелёным соответственно первый и второй пути передачи сообщения. В начале программы происходит инициализация процессов в клетках транспьютерной матрице. Затем первый процесс начинает рассылку сообщения. Время старта равно 100, время передачи одного байта равно 1. $Ts = 100, Tb = 1$. Введём некоторые обозначения:

- Длина сообщения — L (байт);
- Количество кусков, на которое делится сообщение — K;
- Количество путей — $P = 2$ так как пути не должны пересекаться;
- Размер одного сообщения — $N = \frac{L}{P*K}$ (байт);

Теперь рассчитаем время передачи одного сообщения. Из процесса $(0, 0$ до процесса $(4, 4)$, происходит 8 передач. Соответственно на передачу одного куска сообщения потребуется $8 *$

$(Ts + Tb * \frac{L}{P*K}) = 8 * (Ts + Tt * N)$. (На инициализацию канала передачи) А время передачи остальных кусков сообщения $(K-1) * (Ts + Tb * N)$. Тогда общее время передачи будет равно:

$$T_{all} = 8 * (Ts + Tb * \frac{L}{P*K}) + (K-1) * (Ts + Tb * \frac{L}{P*K})$$
$$T_{all} = 8 * (Ts + Tb * N) + (K-1) * (Ts + Tb * N)$$

Но так как в условии говорится об *очень длинном сообщении* можно пренебречь временем старта передачи сообщения, длиной маршрута и временем разгона конвейера. Таким образом у нас остаётся только Tb:

$$T_{all} = L * Tb/P$$

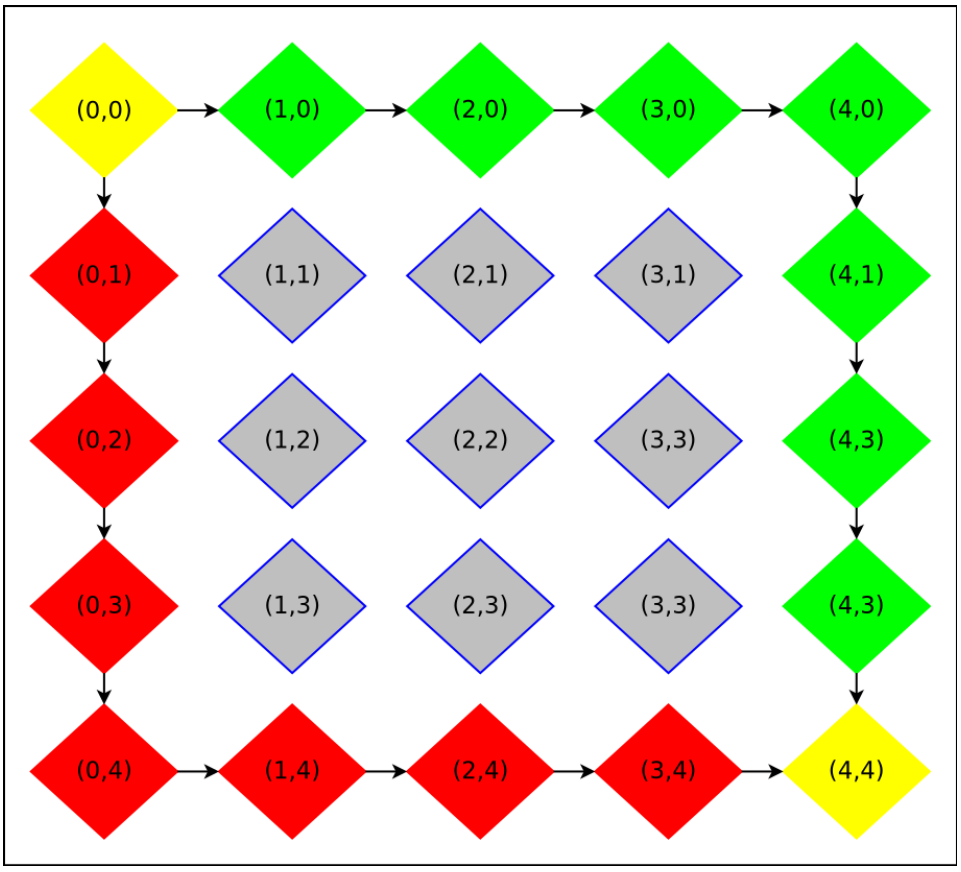


Рис. 1: Транспьютерная матрица

Улучшение алгоритма Adi2D

Необходимо было улучшить алгоритм `Adi2D` с целью улучшения надёжности. Был выбран вариант с созданием дополнительных процессов при старте программы, которые будут использоваться в случае сбоя. Также каждые 5 итераций каждый процесс сохраняет свою часть матрицы в файл — контрольные точки.

Описание внесённых изменений:

1. Переменная **additional_procs** задаёт количество дополнительных процессов, который создаются на старте.

2. Функция **verbose_errhandler** отвечает за обработку ошибок, возникших во время работы программы. В начале выполнения данной функции вызывается `MPIX_Comm_revoke`, чтобы прервать все текущие операции общения и все оставшиеся процессы попали в данную функцию. В конце данной функции происходит перераспределение рангов, очищается выделенная память и происходит выделение новой — соответствующей по размеру

рангу процесса. Затем выставляется флаг ошибки, который позволяет пропустить, после выхода из функции обработки ошибок, все дальнейшие операции вычисления до новой итерации. В начале итерации, если стоит флаг ошибки происходит загрузка данных с последней контрольной точки, и флаг ошибки сбрасывается.

3. Функция **save_matrix**() сохраняет матрицу в файл `matrix_`*rank*, где *rank* — ранг процесса, выполняющего запись.

4. Функция **load_matrix**() загружает матрицу из файла `matrix_`*rank*, где *rank* — ранг процесса, выполняющего загрузку.

5. Переменная **last_save_it** хранит номер последний итерации на которой была сделана контрольная точка.

Экспериментальное исследование

Было проведено небольшое экспериментальное исследование, целью которого было выяснить процентное падение производительности относительно не модифицированной версии программы. А также выяснить время, которое занимает восстановление программы. Исходя из прошлого исследования был выбран размер матрицы 8192, как самый оптимальный для подсчёта на 8 потоках.

Как видно на рисунке 3 с увеличением количества потоков и увеличивается разрыв во времени выполнения старой и улучшенной версий, это связано с количеством записей в файлы, так как чем больше число потоков, тем больше времени занимает запись. В процентном соотношении разрыв составил 13%...90%.

Рассмотрим теперь время затрачиваемое на восстановления после возникновения ошибки. Как видно из рисунка **??** чем больше процессов используется, тем больше времени занимает восстановление, при одинаковом количестве ошибок.

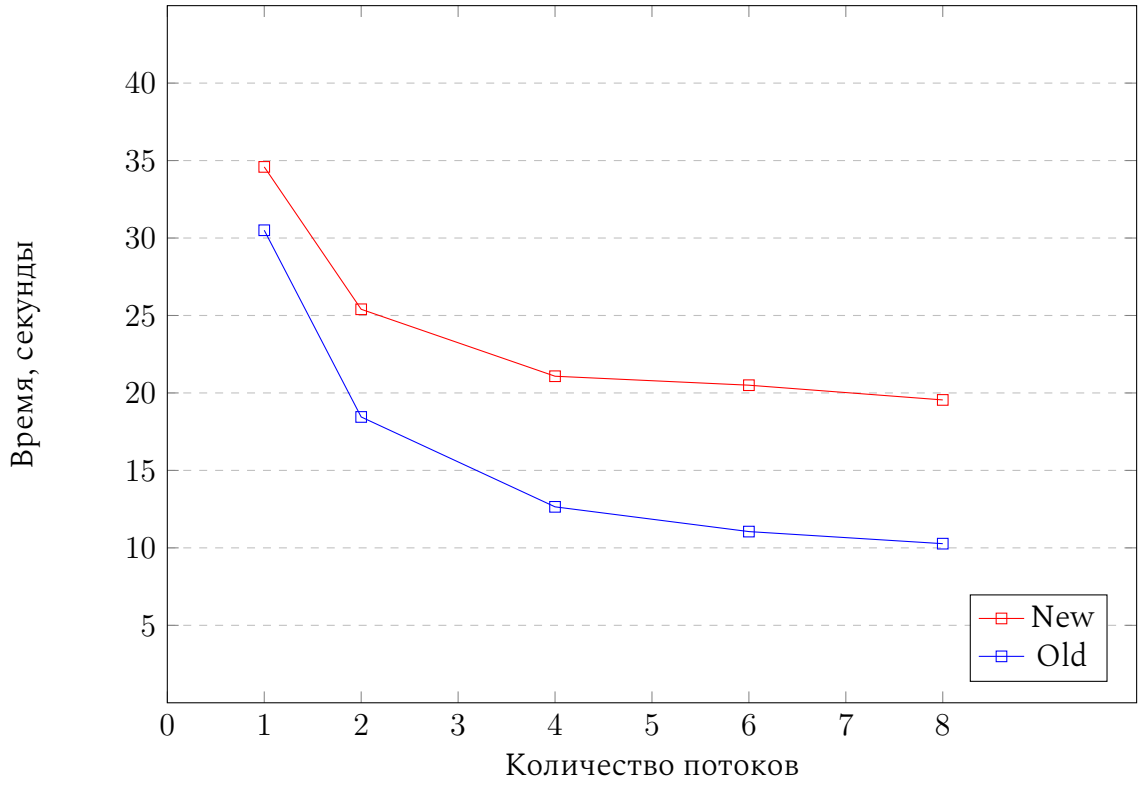


Рис. 2: Зависимость времени исполнения от количества потоков

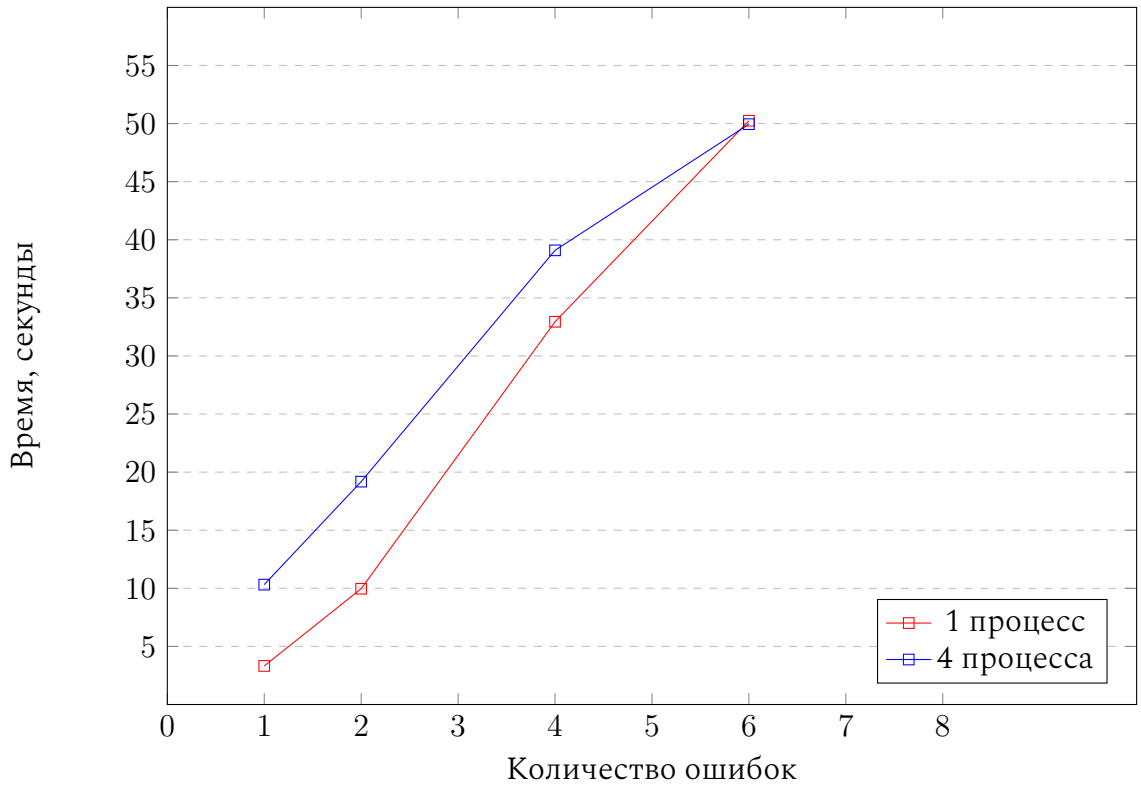


Рис. 3: Зависимость времени восстановления от количества ошибок