МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Практикум по курсу

"Суперкомпьютеры и параллельная обработка данных"

Разработка параллельной версии программы для транспонирования матриц

ОТЧЕТ

о выполненном задании

студентки 328 учебной группы факультета ВМК МГУ

Шелепнёвой Дарьи Дмитриевны

Москва, 2021 г.

Оглавление

1 Постановка задачи

Ставится задача транспонирования матрицы.

Дана матрица $A \in R^{n \times m}$, $n, m \in N$, требуется получить матрицу $B \in R^{m \times n}$, где $B = A_{ij}^T = A_{ji}$.

Требуется:

1. Реализовать параллельный алгоритм транспонирования матрицы с помощью технологий параллельного программирования OpenMP и MPI.
2. Сравнить их эффективность.
3. Исследовать масштабируемость полученных программ и построить графики зависимости времени выполнения программ от числа используемых потоков и объёма входных данных.

2 Описание алгоритма транспонирования матрицы

2.1 Основа: последовательный алгоритм

Простейший алгоритм транспонирования матрицы имеет следующий вид:

```
double **
transpose(double **a, int n, int m)
{
    double **b = malloc(sizeof(double *) * m);
    int i, j;
    for (i = 0; i < m; i++)
    {
        b[i] = malloc(sizeof(double) * n);
        for (j = 0; j < n; j++)
        {
            b[i][j] = a[j][i];
        }
    }
    return b;
}
```

Этот алгоритм имеет сложность $O(nm)$.

2.2 Параллельный алгоритм

Разбиваем задачу вычисления конечной матрицы на подзадачи по вычислению строк и распределяем их по потокам. Разбиение на вычисление отдельных полей не производим, т.к. размеры матриц при вычислениях и так будут на порядки превышать число потоков/процессов.

В OpenMP модификация кода сводится к добавлению ***omp parallel for.***

```
double **
transpose(double **a, int n, int m, int nThreads)
{
    double **b = malloc(sizeof(double *) * m);
    int i, j;
#pragma omp parallel for private(i, j) shared(a, b) num_threads(nThreads)
    for (i = 0; i < m; i++)
    {
        b[i] = malloc(sizeof(double) * n);
        for (j = 0; j < n; j++)
        {
            b[i][j] = a[j][i];
        }
    }
    return b;
}
```

В MPI-версии производится широковещательная рассылка заполненной матрицы ***a***, каждый процесс изменяет соответствующие строки и отправляет их процессу-мастеру (имеющий rank=0) с помощью команд ***MPI_Isend*** и ***MPI_Irecv***. Для синхронизации используются команды ***MPI_Barrier.***

```
void
transpose(int n, double a[n], double resMpi[n][n], int rank, int nProc)
{
    size_t i = 0, j, idx;
    double rowMpi[n];
    MPI_Request request;
    while (rank + i < n) {
        for (j = 0; j < n; j++)
        {
            idx = j * n + rank + i;
            if (idx < n * n) {
                if (!rank) {
                    resMpi[i][j] = a[idx];

                } else {
                    rowMpi[j] = a[idx];
                }
            }
        }
        if (rank) {
            MPI_Isend(rowMpi, n, MPI_DOUBLE, 0, i, MPI_COMM_WORLD, &request);
        }
        i += nProc;
    }
}
```

Коды программ можно посмотреть в репозитории https://github.com/DariaShel/skipod в соответствующих папках.

3 Результаты замеров времени выполнения

Ниже приведены результаты замеров времени выполнения программ на суперкомпьютере Polus в табличной форме и наглядно на графиках.

Программы запускались со следующими параметрами:

- Матрица $A \in R^{n \times n}, n \in \{1000,\ 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000\}$
- Количество потоков (процессов) $nThread$ $(nProc) \in \{1, 2, 4, 8, 16, 32, 64\}$

Было взято среднее значение времени за 5 запусков для каждой конфигурации.

3.1 Таблицы

size	n_thread	average_time
1000	1	0.003237
1000	2	0.000920
1000	4	0.000769
1000	8	0.000771
1000	16	0.000791
1000	32	0.000968
1000	64	0.001563

size	n_thread	average_time
2000	1	0.017415
2000	2	0.011337
2000	4	0.007021
2000	8	0.005574
2000	16	0.004837
2000	32	0.004030
2000	64	0.003520

size	n_thread	average_time
3000	1	0.044771
3000	2	0.026812
3000	4	0.016373
3000	8	0.014286
3000	16	0.011223
3000	32	0.009348
3000	64	0.008856

size	n_thread	average_time
4000	1	0.105689
4000	2	0.063245
4000	4	0.037896
4000	8	0.034011
4000	16	0.023616
4000	32	0.018923
4000	64	0.019144

size	n_thread	average_time
5000	1	0.413145
5000	2	0.230878
5000	4	0.115746
5000	8	0.058552
5000	16	0.036033
5000	32	0.030078
5000	64	0.035809

size	n_thread	average_time
6000	1	0.683482
6000	2	0.374073
6000	4	0.194368
6000	8	0.100035
6000	16	0.056551
6000	32	0.047744
6000	64	0.064573

size	n_thread	average_time
7000	1	1.037804
7000	2	0.571479
7000	4	0.292600
7000	8	0.149152
7000	16	0.081613
7000	32	0.072746
7000	64	0.122549

size	n_thread	average_time
8000	1	1.525624
8000	2	0.829646
8000	4	0.428459
8000	8	0.219062
8000	16	0.117176
8000	32	0.112184
8000	64	0.204893

size	n_thread	average_time
9000	1	1.967304
9000	2	1.062822
9000	4	0.552159
9000	8	0.285058
9000	16	0.163063
9000	32	0.173082
9000	64	0.293878

size	n_thread	average_time
10000	1	2.478779
10000	2	1.344648
10000	4	0.691871
10000	8	0.354458
10000	16	0.228351
10000	32	0.255214
10000	64	0.380011

Результаты алгоритма OpenMP

size	n_proc	average_time
1000	1	0.006682
1000	2	0.009562
1000	4	0.005900
1000	8	0.005751
1000	16	0.007685
1000	32	0.010985
1000	64	0.008205

size	n_proc	average_time
2000	1	0.028534
2000	2	0.039763
2000	4	0.025600
2000	8	0.025357
2000	16	0.038586
2000	32	0.046880
2000	64	0.026394

size	n_proc	average_time
3000	1	0.066866
3000	2	0.104470
3000	4	0.057236
3000	8	0.051360
3000	16	0.081035
3000	32	0.107907
3000	64	0.061528

size	n_proc	average_time
4000	1	0.135502
4000	2	0.160250
4000	4	0.133958
4000	8	0.080882
4000	16	0.143760
4000	32	0.167827
4000	64	0.101201

size	n_proc	average_time
5000	1	0.301439
5000	2	0.349909
5000	4	0.232931
5000	8	0.150804
5000	16	0.223041
5000	32	0.263956
5000	64	0.169383

size	n_proc	average_time
6000	1	0.475649
6000	2	0.540449
6000	4	0.361787
6000	8	0.229945
6000	16	0.310338
6000	32	0.389934
6000	64	0.239652

size	n_proc	average_time
7000	1	0.793677
7000	2	0.837309
7000	4	0.610371
7000	8	0.336368
7000	16	0.464161
7000	32	0.533313
7000	64	0.313629

size	n_proc	average_time
8000	1	1.263932
8000	2	1.379071
8000	4	1.043464
8000	8	0.441304
8000	16	0.657626
8000	32	0.701206
8000	64	0.415631

size	n_proc	average_time
9000	1	1.608660
9000	2	1.701407
9000	4	1.233092
9000	8	0.717673
9000	16	0.826475
9000	32	0.933972
9000	64	0.586568

size	n_proc	average_time
10000	1	2.003993
10000	2	1.923126
10000	4	1.495551
10000	8	1.011924
10000	16	1.047196
10000	32	1.164294
10000	64	0.820514

Результаты алгоритма MPI

3.2 Графики

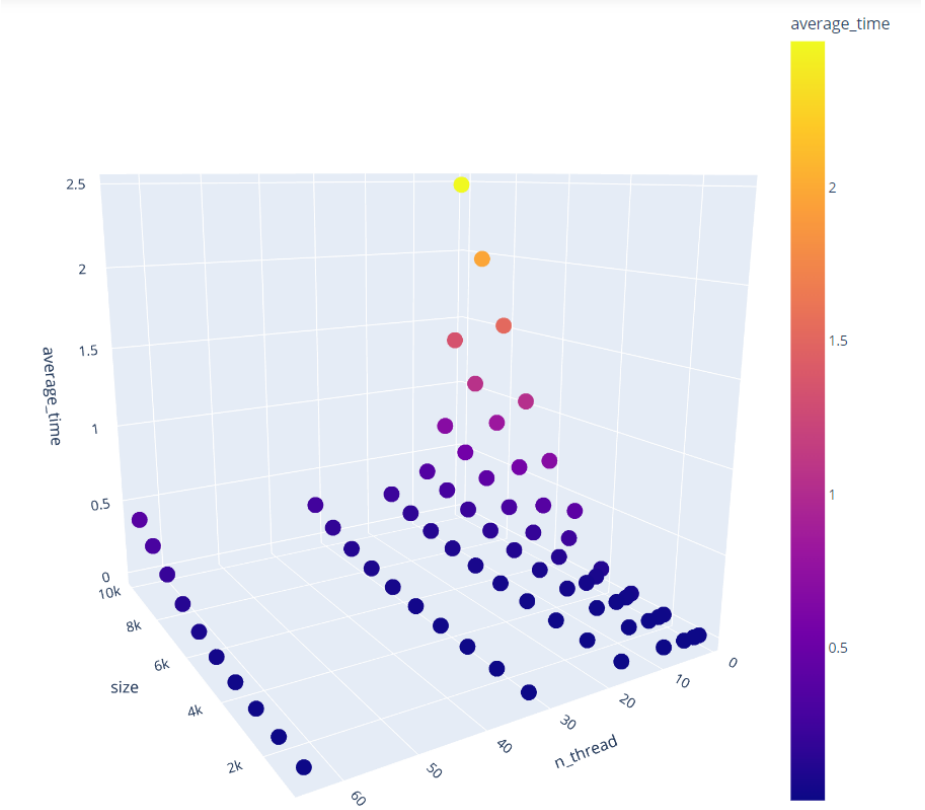


График1 (OpenMP)

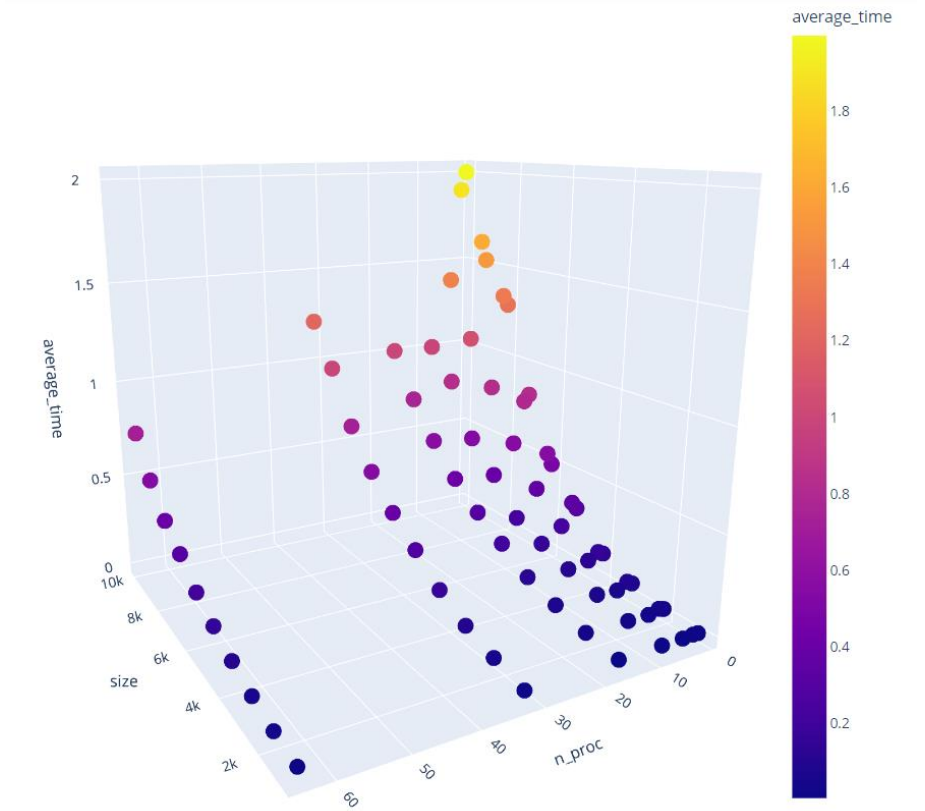


График1 (MPI)

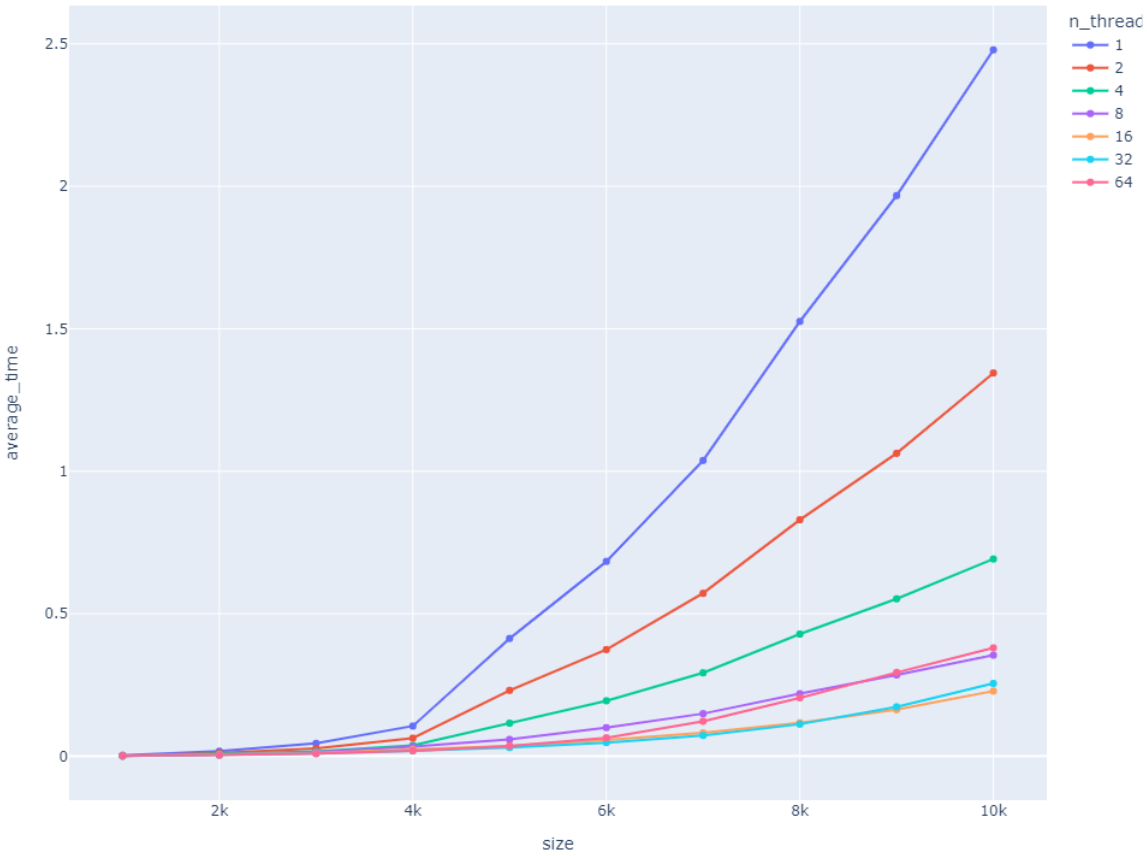


График2 (OpenMP)

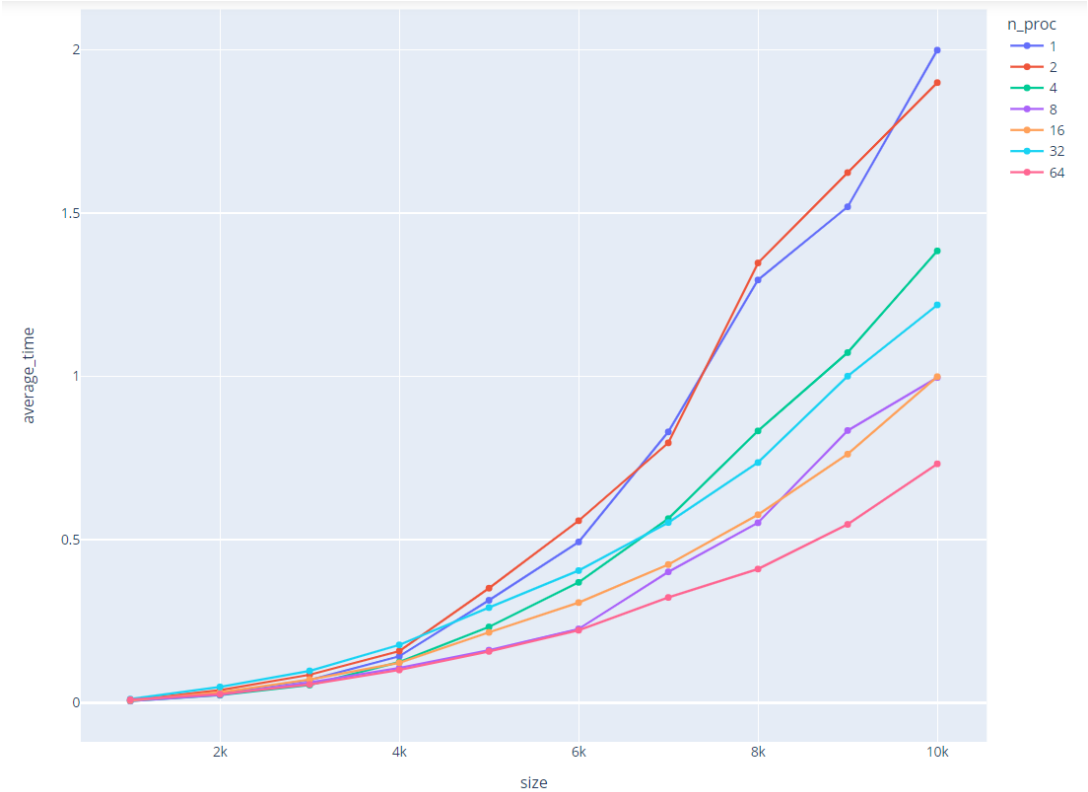
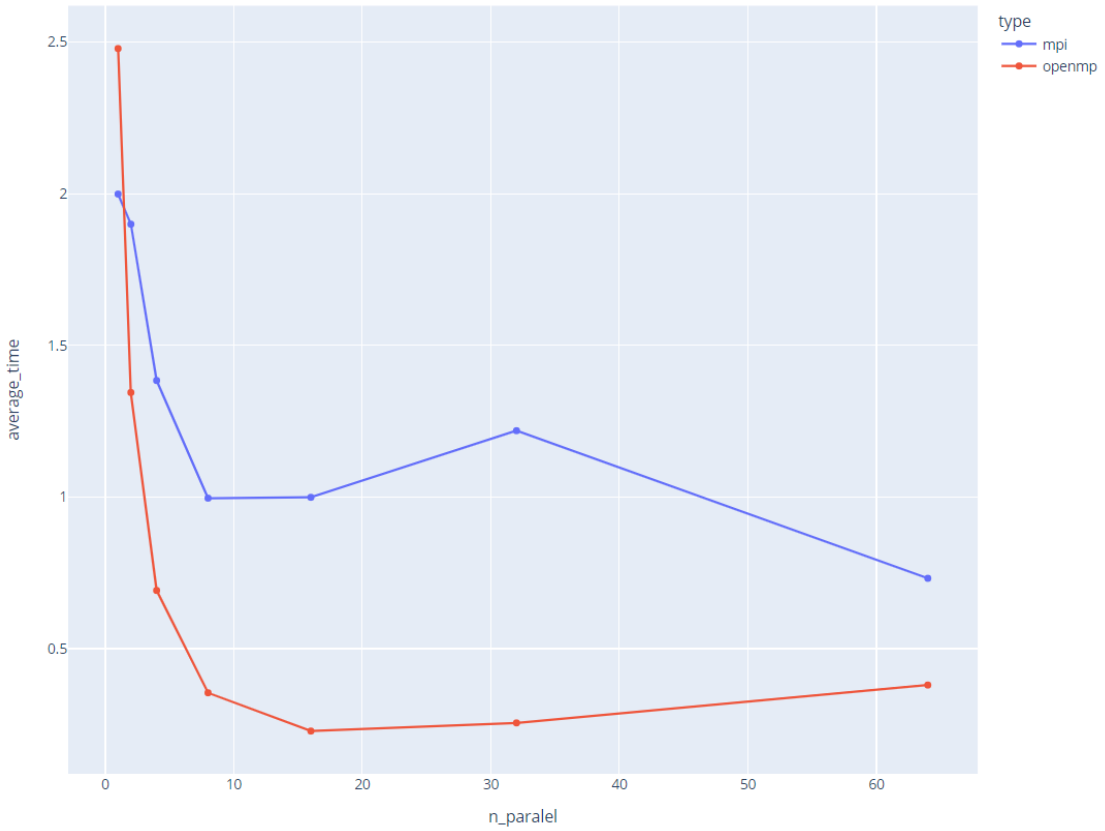


График2 (MPI)

4 Анализ результатов

На одинаковых конфигурациях OpenMP показал результаты лучшие, чем MPI. В алгоритме MPI тратится дополнительное время на пересылку данных между процессорами. Для наглядности был построен график зависимости скорости выполнения алгоритма от количества потоков/процессов для матрицы размера 10000×10000.

size	n_paralel	average_time	type
10000	1	1.999050	mpi
10000	1	2.478779	openmp
10000	2	1.344648	openmp
10000	2	1.899659	mpi
10000	4	0.691871	openmp
10000	4	1.384143	mpi
10000	8	0.354458	openmp
10000	8	0.996295	mpi
10000	16	0.228351	openmp
10000	16	0.999043	mpi
10000	32	0.255214	openmp
10000	32	1.218787	mpi
10000	64	0.380011	openmp
10000	64	0.731948	mpi



Сравнение алгоритмов OpenMP и MPI

Как видно из графика, OpenMP справляется с поставленной задачей гораздо быстрее, чем MPI.

5 Выводы

Выполнена работа по разработке параллельной версии алгоритма транспонирования матриц. Изучена технология написания параллельных алгоритмов OpenMP и MPI. Проанализировано время выполнения алгоритмов на вычислительной системе Polus.

Технология OpenMP достаточно удобна в использовании и даёт значительный прирост производительности на рассчитанных на многопоточные вычисления системах.

MPI можно назвать более низкоуровневой технологией: разработка MPI-программы знакомит с основами взаимодействия вычислительных узлов суперкомпьютера.