Суперкомпьютеры и параллельная обработка данных
Практическое задание

Вариант 44
Лебедев Андрей, группа 324

Цель работы

Научиться использовать технологии MPI, реализовать параллельную версию предложенного алгоритма.

Постановка задачи

1) Для метода релаксации для решения двумерного уравнения Пуассона реализовать параллельную версию программы с использованием технологии MPI.

2) Убедиться в корректности разработанных версий программ.

3) Исследовать эффективность полученных параллельных программ на суперкомпьютере Polus. Исследовать масштабируемость полученной параллельной программы: построить графики зависимости времени выполнения параллельной программы от числа используемых ядер для различного объема входных данных.
   Каждый прогон программы с новыми параметрами выполнять несколько раз с последующим усреднением результата для избавления от случайных выбросов.

4) Определить основные причины недостаточной масштабируемости программы при максимальном числе используемых ядер/процессоров.

Решение задачи

1. Реализация программ

Приведенная программа представляет собой параллельную реализацию метода Якоби для решения уравнения Пуассона с использованием библиотеки MPI для распределенных вычислений. Методы MPI, которые используются в коде:

1. MPI_Init(&an, &as) и MPI_Finalize() используются для инициализации и завершения MPI-приложения соответственно.

2. MPI_Comm_size(MPI_COMM_WORLD, &num_procs) возвращает общее количество процессов в коммуникаторе MPI_COMM_WORLD.

3. MPI_Comm_rank(MPI_COMM_WORLD, &rank) возвращает ранг текущего процесса в коммуникаторе MPI_COMM_WORLD.

4. MPI_Barrier(MPI_COMM_WORLD) используется для синхронизации всех процессов, чтобы они дождались друг друга перед продолжением выполнения.

5. MPI_Irecv и MPI_Isend используются для асинхронного приема и отправки данных между процессами. В данном случае, они используются для обмена строками между процессами.

6. MPI_Waitall ожидает завершения нескольких асинхронных операций, запущенных MPI_Irecv и MPI_Isend.

7. MPI_Allreduce используется для выполнения операции редукции (в данном случае, нахождение максимального значения local_eps) и обмена результатами между всеми процессами.

8. MPI_Wtime() используется для измерения времени выполнения программы.

В программе также реализовано разделение области данных между процессами, чтобы каждый процесс обрабатывал свой сегмент матрицы. Обмен данных между процессами происходит в каждой итерации метода Якоби для обновления граничных строк.

2. Корректность разработанных версий

Программа выдает одинаковое значение S при одном и том же размере N матрицы A. Таким образом, программа работает корректно.

3. Исследование эффективности на Polus

В данном разделе приведено исследование масштабируемости полученной параллельной программы и построены графики зависимости времени выполнения параллельной программы от числа используемых нитей для различного объема входных данных.

Для каждого из размеров (N) матрицы A проведем 4 запуска на количестве нитей от 1 до 160. Измерим время на каждом запуске, а далее найдем среднее по каждому количеству нитей. Таким образом, анализ нескольких запусков поможет избежать выбросы.
Ниже приведены таблицы с соответствующими результатами, где строка соответствует конкретному запуску, а столбец количеству нитей.

1. N = 258

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	0.215487	0.107774	0.078367	0.059103	0.039815	0.033311	0.029156	0.025307	0.028632
2	0.129802	0.065205	0.054088	0.040994	0.027642	0.028451	0.024929	0.018292	0.016852
3	0.158612	0.065215	0.054163	0.041048	0.027623	0.028431	0.024929	0.018292	0.016852
4	0.129779	0.079703	0.054131	0.041014	0.033766	0.028488	0.024915	0.018274	0.016791
Среднее	0.176307	0.088178	0.064119	0.059103	0.039815	0.033311	0.035636	0.030931	0.023426

	10	**20**	**40**	**60**	**80**	**100**	**120**	**140**	**160**
1	0.020937	0.018720	0.019422	0.021320	0.026853	0.130485	0.121033	0.148358	0.08614
2	0.129802	0.018716	0.017150	0.018448	0.021687	0.026855	0.167525	0.174605	0.122124
3	0.158612	0.018643	0.017143	0.017697	0.019995	0.026312	0.394358	0.042490	0.172833
4	0.129779	0.018704	0.017055	0.017955	0.023502	0.025716	0.105932	0.161419	0.089387
Среднее	0.176307	0.020533	0.017517	0.018381	0.021626	0.026490	0.199325	0.124886	0.133175

2. N = 1026

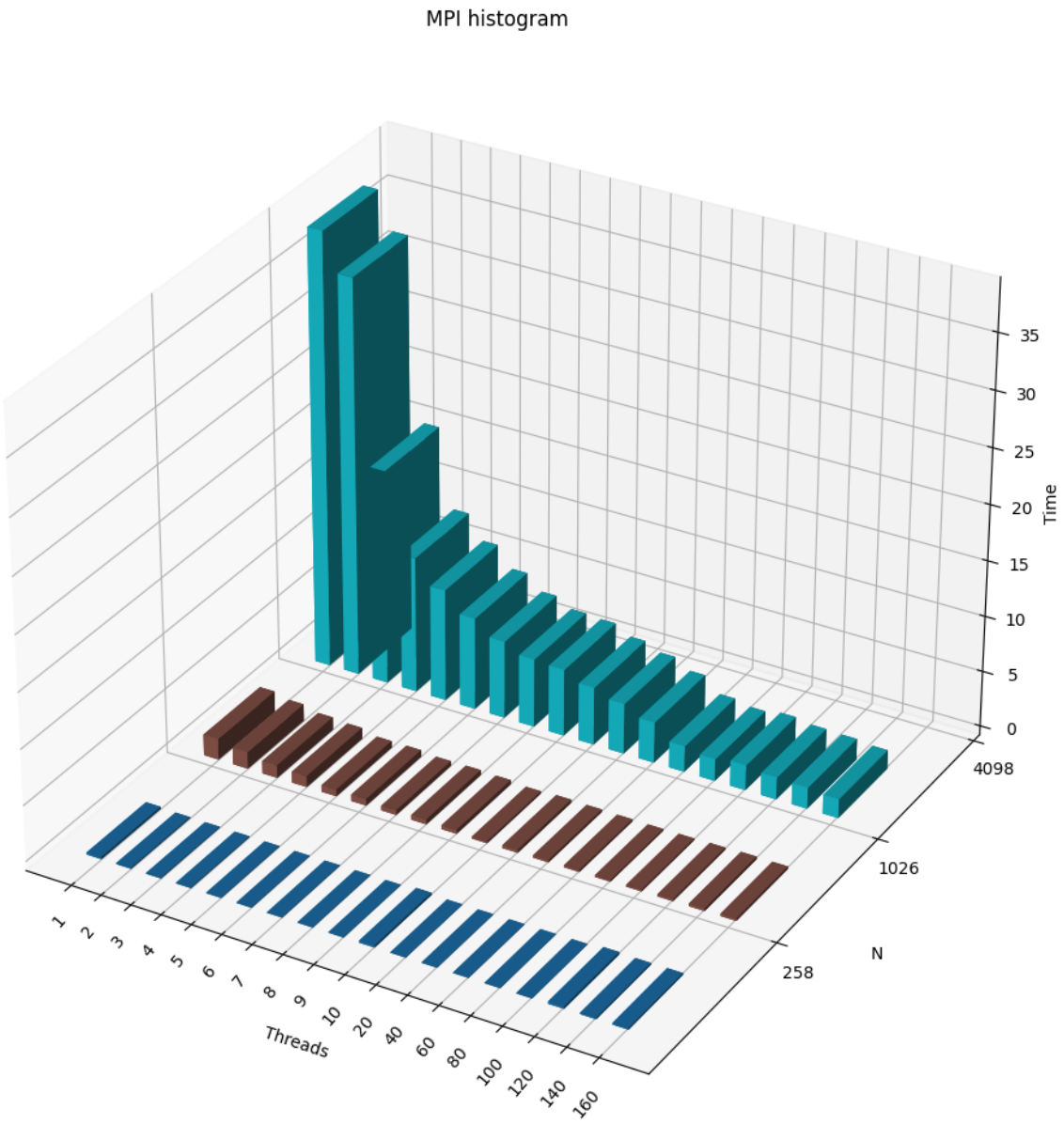
	1	2	3	4	5	6	7	8	9
1	2.277059	1.139599	1.161508	0.876591	0.665136	0.726987	0.624524	0.496801	0.489828
2	0.129802	2.509366	1.348236	0.845287	0.656997	0.541786	0.423903	0.330413	0.299177
3	0.158612	2.509067	1.174606	0.807014	0.586758	0.511581	0.349779	0.315818	0.298967
4	0.129779	2.509300	1.146123	0.741803	0.635861	0.525954	0.426810	0.360973	0.325448
Среднее	0.176307	2.509042	1.122128	0.975735	0.591768	0.585687	0.411938	0.429501	0.352412

	10	20	40	60	80	100	120	140	160
1	0.444939	0.213386	0.198945	0.158758	0.186717	0.195230	0.158258	0.214732	0.189044
2	0.129802	0.215035	0.174310	0.161081	0.184352	0.174952	0.188808	0.278647	0.334694
3	0.158612	0.220792	0.246766	0.152109	0.159582	0.186142	0.146152	0.183737	0.173085
4	0.129779	0.216227	0.175100	0.171619	0.177079	0.163009	0.282388	0.204275	0.181904
Среднее	0.176307	0.259902	0.214113	0.176984	0.150451	0.152074	0.185099	0.237610	0.211015

3. N = 4098

	1	2	3	4	5	6	7	8	9
1	36.53537	23.15791	11.88531	11.12648	8.10186	7.36197	6.59628	6.88197	4.76711
2	38.26322	36.55183	19.22590	15.03707	11.94325	8.15496	8.38367	7.13425	6.92016
3	38.58953	42.72410	20.70067	14.67667	9.70114	7.49812	6.99554	8.11737	5.88231
4	41.56736	38.16063	17.45317	11.24639	8.30141	7.32921	6.48128	5.41819	5.50728
Среднее	38.90930	35.57737	19.26107	12.17635	10.05848	8.25617	6.94470	8.13231	6.99176

	10	20	40	60	80	100	120	140	160
1	4.006419	3.603098	2.554991	2.289861	1.985696	2.088327	1.707752	1.518623	1.617394
2	0.129802	4.758148	3.689513	2.309270	1.999530	2.050973	1.889283	1.793150	1.693124
3	0.158612	6.312507	3.509047	2.481544	2.132821	1.992641	1.677786	1.550885	1.644357
4	0.129779	3.722778	3.289118	2.348042	2.196150	2.165724	1.744570	1.646972	1.502310
Среднее	1.176307	4.515066	3.639304	2.304845	1.959112	2.240238	1.967246	1.887725	1.684327



4. Причины недостаточной масштабируемости

Недостаточная масштабируемость программы при максимальном числе используемых ядер/процессоров может быть вызвана различными причинами:

1. Избыточная синхронизация. Программы часто требуют синхронизации между потоками/процессами для корректного выполнения. Избыточная синхронизация может привести к ожиданию доступа к общим ресурсам и уменьшению эффективности параллельных вычислений.

2. Неравномерное распределение нагрузки. Некоторые потоки или процессы могут выполняться дольше или требовать больше ресурсов, что приводит к неравномерному распределению нагрузки между ядрами/процессорами.

Код программы

Директива for

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

#define Max(a, b) ((a) > (b) ? (a) : (b))

#define N (4096 + 2)
double maxeps = 0.1e-7;
int itmax = 100;
int i, j, k;
double eps;
double A[N][N], B[N][N];

int num_procs, rank, step, start_index, end_index;
MPI_Request requests[4];

void relax();
void init();
void verify();

int main(int an, char **as)
{
    MPI_Init(&an, &as);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    start_index = (rank) * N / num_procs;
    end_index = (rank + 1) * N / num_procs;
    step = end_index - start_index;

    double start_time = MPI_Wtime();

    int it;
```

```
    init();
    for (it = 1; it <= itmax; it++)
    {
        eps = 0.;
        relax();
        if (eps < maxeps)
            break;
    }
    verify();

    MPI_Barrier(MPI_COMM_WORLD);

    double end_time = MPI_Wtime();
    double execution_time = end_time - start_time;

    if (rank == 0)
    {
        printf("Processes: %i\nTime: %f seconds\n", num_procs,
execution_time);
    }
    MPI_Finalize();
    return 0;
}


void init()
{
    int offset_l = 1, offset_r = 1;
    if (rank == 0)
        offset_l = 0;

    if (rank == num_procs - 1)
        offset_r = 0;

    for (i = start_index - offset_l; i < end_index + offset_r; i++)
        for (j = 0; j <= N - 1; j++)
        {
            if (i == 0 || i == N - 1 || j == 0 || j == N - 1)
                A[i][j] = 0.;
            else
                A[i][j] = (4. + i + j);
        }

}

void share_end_rows()
```

```
{
    if (rank != 0)
        MPI_Irecv(A[start_index - 1], 1 * N, MPI_DOUBLE, rank - 1, 1,
MPI_COMM_WORLD, requests);

    if (rank != num_procs - 1)
        MPI_Isend(A[end_index - 1], 1 * N, MPI_DOUBLE, rank + 1, 1,
MPI_COMM_WORLD, requests + 2);
}

void share_start_rows()
{
    if (rank != num_procs - 1)
        MPI_Irecv(A[end_index], 1 * N, MPI_DOUBLE, rank + 1, 2,
MPI_COMM_WORLD, requests + 3);

    if (rank != 0)
        MPI_Isend(A[start_index], 1 * N, MPI_DOUBLE, rank - 1, 2,
MPI_COMM_WORLD, requests + 1);
}

void waitall()
{
    int count = 4, shift = 0;

    if (rank == 0)
    {
        count = 2;
        shift = 2;
    }
    if (rank == num_procs - 1)
    {
        count = 2;
    }

    MPI_Waitall(count, requests + shift, MPI_STATUSES_IGNORE);
}

void relax()
{
    int offset_l = 0, offset_r = 0;
    if (rank == 0)
        offset_l = 1;

    if (rank == num_procs - 1)
        offset_r = 1;
```

```
    for (i = start_index + offset_l; i < end_index - offset_r; i++)
        for (j = 1; j <= N - 2; j++)
        {
            B[i][j] = (A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4.;
        }

    double local_eps = eps;
    for (i = start_index + offset_l; i < end_index - offset_r; i++)
        for (j = 1; j <= N - 2; j++)
        {
            double e;
            e = fabs(A[i][j] - B[i][j]);
            A[i][j] = B[i][j];
            local_eps = Max(local_eps, e);
        }

    if (num_procs != 1)
    {
        share_end_rows();
        share_start_rows();
        waitall();
    }

    MPI_Allreduce(&local_eps, &eps, 1, MPI_DOUBLE, MPI_MAX,
MPI_COMM_WORLD);
}

void verify()
{
    double s = 0.;
    double local_s = 0.;
    for (i = start_index; i < end_index; i++)
        for (j = 0; j <= N - 1; j++)
        {
            local_s = local_s + A[i][j]* (i + 1) * (j + 1) / (N * N);
        }

    MPI_Reduce(&local_s, &s, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0)
    {
        printf("  S = %f\n", s);
    }
}
```