МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики

Задание №2:
«Разработка параллельной версии программы RedBlack_2D с использованием технологии MPI»

325 группа ВМК МГУ
Бежанян Регина

2020 год

Содержание

Постановка задачи

Разработать параллельную версию программы для задачи RedBlack_2D с использованием технологии MPI (Message passing interface), а затем исследовать масштабируемость полученной программы, построить графики зависимости времени её выполнения от числа используемых процессов и объёма входных данных. Сделать выводы с учетом полученных зависимостей. А также сравнить результаты работы параллельных версий программ для задачи RedBlack_2D с использованием технологии OpenMP и MPI.

Описание алгоритма

Алгоритм обрабатывает матрицу размера N*N, где N – объем данных, который задается до начала работы программы. Сначала происходит инициализация матрицы изначально заданным способом в функции init(). Дальше происходит основная часть работы алгоритма – релаксация матрицы relax(). Это итерационный метод решения систем линейных алгебраических уравнений. Последним шагом *verify()* вычисляется ответ.

Реализация параллельной версии программы с использованием технологии MPI

Для использования механизмов MPI необходимо подключить библиотеку “mpi.h”.

Основные вызовы библиотеки “mpi.h”, которые использовались в программе:

MPI_Comm_size(MPI_COMM_WORLD, int *size) - указывает число процессов в коммуникаторе | вычисляет размер

MPI_Comm_rank(MPI_COMM_WORLD, int *rank) - указывает номер вызывающего процесса, который располагается в диапазоне от 0 до size-1

MPI_Barrier(MPI_COMM_WORLD) – синхронизация. Узел, вызывающий его, будет блокирован, пока все узлы в пределах группы не вызвали его.

MPI_Send(void* message, int count,
MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm) – функция отправки собщений (блокирующая)

MPI_Isend(void* message, int count,
MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm) – функция отправки собщений (неблокирующая)

MPI_Recv(void* message, int count,
MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Status* status) – функция приема сообщений (блокирующая)

MPI_Irecv(void* message, int count,
MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Status* status) – функция приема сообщений (неблокирующая)

В данной задаче происходит пересылка строк матрицы A

MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

- функция, которая применяет коллективную операцию (в данной задаче – это суммирование eps) к локальным переменным каждого процесса и рассылает результат всем процессам в группе.

Для замера времени использовался вызов функции MPI_Wtime().

Объем входных данных изменялся в самой программе, а количество используемых процессов задается опцией -n.

Тестирование программы происходило в системе Blue Gene:

scp -i edu-cmc-skpod20-325-02 redblack_2d_mpi.c edu-cmc-skpod20-325-02@blugene.hpc.cs.msu.ru:

ssh -i edu-cmc-skpod20-325-02 edu-cmc-skpod20-325-02@blugene.hpc.cs.msu.ru

Компиляция программы проводилась с помощью команды:

mpixlc redblack_2d_mpi.c -o run

Дальше программа подавалась в очередь с помощью команды:

mpisubmit.bg -n 1 -w 00:10:00 -m vn run

где флагом -n указывается запрашиваемое число процессов (в данной задаче 1,…,256), а -w – максимальное время выполнения программы

Информация о времени работы программы при разном количестве процессов извлекалась из файлов run.%J.out.

Код программы:

```
1. #include <math.h>
2. #include <stdio.h>
3. #include "mpi.h"
4. #define Max(a, b) ((a)>(b)?(a):(b))
5.
6. #define N 64
7.
8. double max_eps = 0.1e-7;
```

```
9. int itmax = 100;
10. double w = 0.5;
11. double eps;
12. int i, j;
13. int num_procs, rank;
14. int min_row = 0, max_row = N - 1;
15.
16. MPI_Status status;
17. MPI_Request request;
18.
19. double A[N][N];
20.
21. void init();
22. void relax();
23. void verify();
24.
25. int main(int argc, char **argv) {
26.     MPI_Init(&argc, &argv);
27.     double time_start, time_end;
28.     init();
29.
30.     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
31.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
32.     MPI_Barrier(MPI_COMM_WORLD);
33.
34.     if (rank == 0) {
35.         time_start = MPI_Wtime();
36.     }
37.
38.     min_row = rank == 0 ? 0 : (N / num_procs) * rank - 1;
39.     max_row = rank == num_procs - 1 ? N - 1 : (N / num_procs) *
   (rank + 1);
40.
41.     for (int it = 1; it <= itmax; ++it) {
42.         eps = 0.;
43.         relax();
44.         if (eps < max_eps) {
45.             break;
46.         }
47.     }
```

```
48.
49.     if (rank != 0) {
50.         MPI_Send(A[min_row + 1], (max_row - min_row - 1) * N,
    MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
51.     } else {
52.         for (i = 1; i < num_procs; ++i) {
53.             int tmp_min_row = i == 0 ? 0 : N / num_procs * i -
    1;
54.             int tmp_max_row = i == num_procs - 1 ? N - 1 : (N /
    num_procs) * (i + 1);
55.             MPI_Recv(A[tmp_min_row + 1], (tmp_max_row -
    tmp_min_row - 1) * N, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);
56.         }
57.     }
58.
59.     MPI_Barrier(MPI_COMM_WORLD);
60.
61.     if (rank == 0) {
62.         verify();
63.         time_end = MPI_Wtime();
64.         printf("time  = %f\n", time_end - time_start);
65.     }
66.
67.     MPI_Finalize();
68.     return 0;
69. }
70.
71. void init() { //инициализация матрицы
72.     for(j=0; j<=N-1; j++)
73.     for(i=0; i<=N-1; i++)
74.     {
75.         if(i==0 || i==N-1 || j==0 || j==N-1) A[i][j]= 0.;
76.         else A[i][j]= ( 1. + i + j) ;
77.     }
78. }
79.
80. void relax() { //релаксация матрицы
81.
82.     int nrank = rank == num_procs - 1 ? MPI_PROC_NULL : rank +
    1;
```

```
83.     int prank = rank == 0 ? MPI_PROC_NULL : rank - 1;
84.     double local_eps = eps;
85.
86.     for (i = min_row + 1; i <= max_row - 1; i++) {
87.         for ( j = 1 + (i % 2); j <= N - 2; j += 2) {
88.             double b = w * ((A[i - 1][j] + A[i + 1][j] + A[i][j
    - 1] + A[i][j + 1]) / 4. - A[i][j]);
89.             A[i][j] = A[i][j] + b;
90.             local_eps = Max(fabs(b), local_eps);
91.         }
92.     }
93.
94.     MPI_Isend(A[max_row - 1], N, MPI_DOUBLE, nrank, 0,
    MPI_COMM_WORLD, &request);
95.     MPI_Recv(A[max_row], N, MPI_DOUBLE, nrank, 0,
    MPI_COMM_WORLD, &status);
96.
97.     MPI_Isend(A[min_row + 1], N, MPI_DOUBLE, prank, 0,
    MPI_COMM_WORLD, &request);
98.     MPI_Recv(A[min_row], N, MPI_DOUBLE, prank, 0,
    MPI_COMM_WORLD, &status);
99.
100.    for ( i = min_row + 1; i <= max_row - 1; ++i) {
101.        for (j = 1 + ((i + 1) % 2); j <= N - 2; j += 2) {
102.            double b = w * ((A[i - 1][j] + A[i + 1][j] + A[i][j
    - 1] + A[i][j + 1]) / 4. - A[i][j]);
103.            A[i][j] = A[i][j] + b;
104.        }
105.    }
106.
107.    MPI_Isend(A[max_row - 1], N, MPI_DOUBLE, nrank, 0,
    MPI_COMM_WORLD, &request);
108.    MPI_Recv(A[max_row], N, MPI_DOUBLE, nrank, 0,
    MPI_COMM_WORLD, &status);
109.
110.    MPI_Isend(A[min_row + 1], N, MPI_DOUBLE, prank, 0,
    MPI_COMM_WORLD, &request);
111.    MPI_Recv(A[min_row], N, MPI_DOUBLE, prank, 0,
    MPI_COMM_WORLD, &status);
112.
```

```
113.     MPI_Allreduce(&local_eps, &eps, 1, MPI_DOUBLE, MPI_MAX,
    MPI_COMM_WORLD);
114. }
115.
116.
117. void verify() { //подсчет ответа
118.     double s;
119.     s = 0.;
120.     for (i = 0; i <= N - 1; i++) {
121.         for (j = 0; j <= N - 1; j++) {
122.             s = s + A[i][j] * (i + 1) * (j + 1) / (N * N);
123.         }
124.     }
125.     printf("S = %f\n", s);
126. }
```

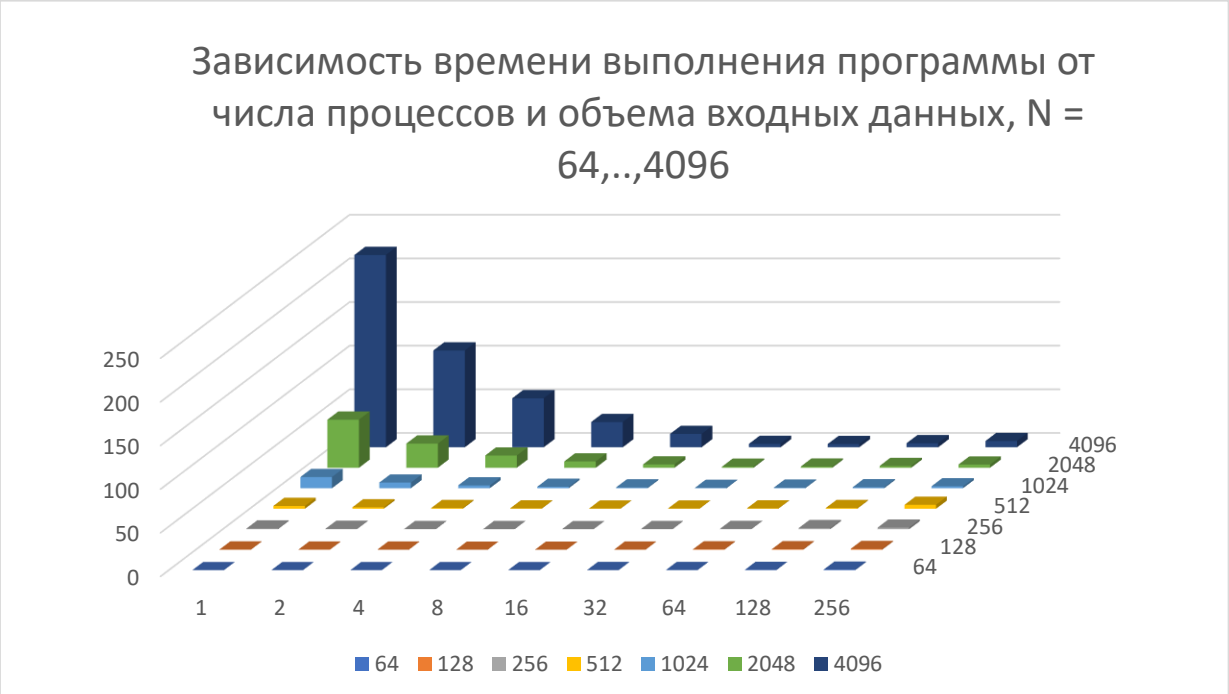
Результаты

Для подсчета времени выполнения производилось несколько запусков на одинаковых данных, после чего было сделано усреднение результата для каждого числа процессов. Это производится для избавления от случайных выбросов.

Время выполнения

Число процессов, num_procs	Размер массива, N						
	64	128	256	512	1024	2048	4096
1	0,049128	0,197324	0,796794	3,200212	12,831651	55,099683	220,36845
2	0,02578	0,10275	0,406084	1,615452	6,463902	27,721398	110,91698
4	0,017513	0,058172	0,215177	0,829047	3,288232	14,04473	56,147093
8	0,013119	0,033313	0,118914	0,43681	1,705645	7,239592	28,882191
16	0,014745	0,026375	0,079697	0,250918	0,927854	3,855928	15,284398
32	0,118252	0,08693	0,171465	0,246743	0,458822	1,211702	3,9253
64	0,187771	0,357954	0,327993	0,42063	0,670088	1,393104	3,73904
128	0,326838	0,519725	1,319189	0,790168	1,171697	2,076962	4,654556
256	0,617387	0,852149	1,859458	4,545415	2,206872	3,646617	7,15751

Для наглядности построим график:



Выводы

Можно заметить, что при любом N сначала время выполнения программы линейно зависит от числа процессов и уменьшается, а при числе процессов 64,..,256 стабильно получается ухудшение по времени работы. (Однако при N=4096 результат при num_procs = 64 немного лучше, чем при 32, что логично, так как объем данных достаточно большой). Это связано с накладными расходами.

Если сравнивать результаты работы параллельной версии программы MPI с параллельной версии программы OpenMP (результаты работы программы представлены в отчет по разработке параллельной версии программы RedBlack_2D OpenMP), то можно сделать вывод о том, что при всех объемах данных задача RedBlack_2D решается быстрее при использовании технологии OpenMP. Для больших N различие во времени время работы программ становится значительным. Предполагаю, что это связано с особенностями технологий. OpenMP работает на устройствах с общей памятью, а MPI с распределенной памятью. Из-за этого происходят постоянные пересылки строк матрицы (с помощью функций, описанных в пункте 3) для взаимодействия процессов, что сильно сказывается на времени работы программы.

В целом параллельный подход к реализации программы RedBlack_2D дает хороший выигрыш по времени по сравнению с обычной программой. При этом для данной задачи проще и эффективнее использовать технологию OpenMP.