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**

1. Постановка задачи
2. Описание алгоритма
3. Реализация параллельной версии программы с использованием технологии MPI
4. Результаты
5. Выводы

**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 А

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](mailto: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](mailto: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))
6. #define N 64
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;
16. MPI\_Status status;
17. MPI\_Request request;
19. **double** A[N][N];
21. **void** init();
22. **void** relax();
23. **void** verify();
25. **int** main(**int** argc, **char** \*\*argv) {
26. MPI\_Init(&argc, &argv);
27. **double** time\_start, time\_end;
28. init();
30. MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_procs);
31. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);
32. MPI\_Barrier(MPI\_COMM\_WORLD);
34. **if** (rank == 0) {
35. time\_start = MPI\_Wtime();
36. }
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);
41. **for** (**int** it = 1; it <= itmax; ++it) {
42. eps = 0.;
43. relax();
44. **if** (eps < max\_eps) {
45. **break**;
46. }
47. }
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. }
59. MPI\_Barrier(MPI\_COMM\_WORLD);
61. **if** (rank == 0) {
62. verify();
63. time\_end = MPI\_Wtime();
64. printf("time  = %f\n", time\_end - time\_start);
65. }
67. MPI\_Finalize();
68. **return** 0;
69. }
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. }
80. **void** relax() { //релаксация матрицы
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;
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. }
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);
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);
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. }
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);
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);
113. MPI\_Allreduce(&local\_eps, &eps, 1, MPI\_DOUBLE, MPI\_MAX, MPI\_COMM\_WORLD);
114. }

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](http://dvmh.keldysh.ru/attachments/download/2878/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0%20%D0%BF%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D0%B9%20%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B8%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D1%8B%20RedBlack_2D%20OpenMP.pdf)), 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.