МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Практикум по учебному курсу

"Суперкомпьютеры и параллельная обработка данных"

**Задание №2:**

**Разработка параллельной версии программы 2 Matrix Multiplications с использованием технологии MPI.**

Отчет

студента 325 группы

факультета ВМК МГУ

Хайбулаева Глеба Сергеевича

2019 год

***Постановка задачи***

Распараллелить предложенную реализацию программы 2 Matrix Multiplications с использованием технологии MPI, а затем исследовать масштабируемость полученной программы, построить графики зависимости времени её выполнения от числа используемых ядер и объёма входных данных. Программа получает на вход 4 числа – размеры четырех матриц A, B, C и D и на основе этих данных вычисляет выражение 1.5\*A\*B\*C + 1.2\*D.

*Код программы*

1. *#include <stdio.h>*
2. *#include <unistd.h>*
3. *#include <math.h>*
4. *#include <stdlib.h>*
5. *#include <mpi.h>*
6. *static*
7. *void init\_array(int ni, int nj, int nk, int nl,*
8. *double \*alpha,*
9. *double \*beta,*
10. *double A[ ni][nk],*
11. *double B[ nk][nj],*
12. *double C[ nj][nl],*
13. *double D[ ni][nl],*
14. *double tmp[ ni][nj])*
15. *{*
16. *int i, j;*
17. *\*alpha = 1.5;*
18. *\*beta = 1.2;*
19. *for (i = 0; i < ni; i++)*
20. *for (j = 0; j < nk; j++)*
21. *A[i][j] = (double) ((i\*j+1) % ni) / ni;*
22. *for (i = 0; i < nk; i++)*
23. *for (j = 0; j < nj; j++)*
24. *B[i][j] = (double) (i\*(j+1) % nj) / nj;*
25. *for (i = 0; i < nj; i++)*
26. *for (j = 0; j < nl; j++)*
27. *C[i][j] = (double) ((i\*(j+3)+1) % nl) / nl;*
28. *for (i = 0; i < ni; i++)*
29. *for (j = 0; j < nl; j++)*
30. *D[i][j] = (double) (i\*(j+2) % nk) / nk;*
31. *for (i = 0; i < ni; i++)*
32. *for (j = 0; j < nj; j++)*
33. *tmp[i][j] = 0.0;*
34. *}*
35. *static*
36. *void print\_array(int ni, int nl,*
37. *double D[ ni][nl]) {*
38. *int i, j;*
39. *fprintf(stderr, "==BEGIN DUMP\_ARRAYS==\n");*
40. *fprintf(stderr, "begin dump: %s\n", "D");*
41. *for (i = 0; i < ni; i++) {*
42. *for (j = 0; j < nl; j++) {*
43. *fprintf(stderr, "%0.2lf ", D[i][j]);*
44. *}*
45. *fprintf(stderr, "\n");*
46. *}*
47. *fprintf(stderr, "\nend dump: %s\n", "D");*
48. *fprintf(stderr, "==END DUMP\_ARRAYS==\n");*
49. *}*
50. *static*
51. *void kernel\_2mm(int ni, int nj, int nk, int nl,*
52. *double alpha,*
53. *double beta,*
54. *int rank,*
55. *int max\_row,*
56. *double tmp[ ni][nj],*
57. *double A[ ni][nk],*
58. *double B[ nk][nj],*
59. *double C[ nj][nl],*
60. *double D[ ni][nl])*
61. *{*
62. *int i, j, k;*
63. *for (i = rank \* max\_row; i < max\_row + rank \* max\_row; i ++) {*
64. *for (j = 0; j < nj; j++) {*
65. *tmp[i][j] = 0.0;*
66. *for (k = 0; k < nk; ++k) {*
67. *tmp[i][j] += alpha \* A[i][k] \* B[k][j];*
68. *}*
69. *}*
70. *}*
71. *MPI\_Gather(&tmp[rank\*max\_row], max\_row\*nj, MPI\_DOUBLE, tmp, max\_row\*nj, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);*
72. *MPI\_Barrier(MPI\_COMM\_WORLD);*
73. *for (i = rank \* max\_row; i < max\_row + rank \* max\_row; i ++) {*
74. *for (j = 0; j < nl; j++) {*
75. *D[i][j] \*= beta;*
76. *for (k = 0; k < nj; ++k)*
77. *D[i][j] += tmp[i][k] \* C[k][j];*
78. *}*
79. *}*
80. *MPI\_Gather(&D[rank\*max\_row], max\_row\*nl, MPI\_DOUBLE, D, max\_row\*nl, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);*
81. *MPI\_Barrier(MPI\_COMM\_WORLD);*
82. *}*
83. *int main(int argc, char\*\* argv)*
84. *{*
85. *int err = MPI\_Init(&argc, &argv);*
86. *if (err != MPI\_SUCCESS)*
87. *{*
88. *fprintf(stderr, "Error while starting! \n");*
89. *MPI\_Abort(MPI\_COMM\_WORLD, err);*
90. *}*
91. *int size, rank = 0;*
92. *MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);*
93. *MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);*
94. *if (!rank) {*
95. *printf("Number of threads: %d\n", size);*
96. *}*
97. *int nis[5] = {16, 40, 180, 800, 1600};*
98. *int njs[5] = {18, 50, 190, 900, 1800};*
99. *int nks[5] = {22, 70, 210, 1100, 2200};*
100. *int nls[5] = {24, 80, 220, 1200, 2400};*
101. *char \*names[5] = {"MINI", "SMALL", "MEDIUM", "LARGE", "EXTRALARGE"};*
102. *int ni;*
103. *int nj;*
104. *int nk;*
105. *int nl;*
106. *MPI\_Barrier(MPI\_COMM\_WORLD);*
107. *for (int i = 0; i < 5; i++) {*
108. *ni = nis[i];*
109. *nk = nks[i];*
110. *nj = njs[i];*
111. *nl = nls[i];*
112. *double alpha;*
113. *double beta;*
114. *double (\*tmp)[ni][nj] = NULL;*
115. *double (\*A)[ni][nk] = NULL;*
116. *double (\*B)[nk][nj] = NULL;*
117. *double (\*C)[nj][nl] = NULL;*
118. *double (\*D)[ni][nl] = NULL;*
119. *tmp = (double (\*)[ni][nj]) malloc((ni) \* (nj) \* sizeof(double));*
120. *A = (double (\*)[ni][nk]) malloc((ni) \* (nk) \* sizeof(double));*
121. *B = (double (\*)[nk][nj]) malloc((nk) \* (nj) \* sizeof(double));*
122. *C = (double (\*)[nj][nl]) malloc((nj) \* (nl) \* sizeof(double));*
123. *D = (double (\*)[ni][nl]) malloc((ni) \* (nl) \* sizeof(double));*
124. *init\_array(ni, nj, nk, nl, &alpha, &beta,*
125. *\*A,*
126. *\*B,*
127. *\*C,*
128. *\*D,*
129. *\*tmp);*
130. *double start = MPI\_Wtime();*
131. *int max\_row = ni/size;*
132. *kernel\_2mm(ni, nj, nk, nl,*
133. *alpha, beta,*
134. *rank, max\_row,*
135. *\*tmp,*
136. *\*A,*
137. *\*B,*
138. *\*C,*
139. *\*D);*
140. *double end = MPI\_Wtime();*
141. *if (rank == 0){*
142. *printf("Dataset %s:\nTime = %fs\n", names[i], end-start);*
143. *fflush(stdin);*
144. *}*
145. *if (argc > 42 && !strcmp(argv[0], "")) print\_array(ni, nl, \*D);*
146. *MPI\_Barrier(MPI\_COMM\_WORLD);*
147. *if (tmp != NULL)*
148. *free((void \*) tmp);*
149. *tmp = NULL;*
150. *if (A != NULL)*
151. *free((void \*) A);*
152. *A = NULL;*
153. *if (B != NULL)*
154. *free((void \*) B);*
155. *B = NULL;*
156. *if (C != NULL)*
157. *free((void \*) C);*
158. *C = NULL;*
159. *if (D != NULL)*
160. *free((void \*) D);*
161. *D = NULL;*
162. *}*
163. *MPI\_Finalize();*
164. *return 0;*
165. *}*

Распараллелена программа классическими приёмами, рассмотренными на лекции.

***Результаты замеров времени выполнения***

Работа задачи рассмотрена на суперкомпьютере Polus с различным числом нитей (1 - 64) и различными наборами данных, предоставленных вместе с исходным кодом (см ниже). Каждое измерение проводилось 6 раз. В таблице и на графиках записаны усредненные результаты времени выполнения.  
Наборы данных:  
Mini – 16, 18, 22, 24  
Small – 40, 50, 70, 80  
Medium – 180, 190, 210, 220  
Large – 800, 900, 1100, 1200  
Extralarge – 1600, 1800, 2200, 2400

***Таблица с результатами***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Нити/размеры матриц | Mini | Small | Medium | Large | Extralarge |
| 1 | 0.000151 | 0.002874 | 0.166226 | 18.561344 | 139.938766 |
| 2 | 0.000095 | 0.001511 | 0.069315 | 8.515160 | 71.453593 |
| 4 | 0.000085 | 0.001124 | 0.053768 | 4.528845 | 35.470738 |
| 8 | 0.000068 | 0.000660 | 0.027161 | 3.303941 | 18.185954 |
| 16 | 0.000049 | 0.000285 | 0.014227 | 1.193156 | 9.835589 |
| 32 | 0.000059 | 0.000482 | 0.018878 | 0.822930 | 6.493348 |
| 64 | 0.000084 | 0.000203 | 0.010056 | 0.525736 | 2.835041 |

***Графики: время выполнения программы в зависимости от размеров матриц и количества потоков***

В виде линий:

В виде поверхности:

В виде линий в трехмерном пространстве:

***Вывод:***

Распараллеливание программы дало выигрыш около 10 по времени раз на массивах небольшого размера (Mini, Small, Medium) и до 50 раз на массивах большого размера (Large, Extralarge).

Как показали эксперименты, в отличие от технологии OpenMP, MPI не создает больших накладных расходов при увеличении числа нитей, что позволяет получать улучшенный результат даже при малых наборах данных. Вместе с этим при увеличении числа нитей существенно возрастает производительность, позволяя быстро выполнять программу даже при больших наборах данных.

Отсюда следует вывод, что MPI дает прирост производительности даже при небольших наборах данных и большом числе нитей. Исходя из этого можно сказать, что, в отличие от OpenMP, использование MPI оправдано при любых наборах данных, даже при большом числе нитей.