****

并行计算实验报告

|  |  |
| --- | --- |
| 学生姓名： |  |
| 指导教师： | 雷向东 |
| 学 院： | 计算机学院 |
| 专业班级： |  |

本科生院制

目录

目录

[目录 2](#_Toc6244053)

[1 实验步骤 2](#_Toc6244054)

[1.1 环境准备 2](#_Toc6244055)

[2 实验一 MPI实现Cannon算法 3](#_Toc6244056)

[2.1 原理 3](#_Toc6244057)

[2.2 源代码 4](#_Toc6244058)

[3 实验二 MPI实现Fox算法 4](#_Toc6244059)

[3.1 原理 4](#_Toc6244060)

[3.2 源代码 4](#_Toc6244061)

[4 实验三 并行正则采样排序（PSRS） 4](#_Toc6244062)

[4.1 原理 4](#_Toc6244063)

[4.2 源代码 5](#_Toc6244064)

[5 OMP实验雅各比迭代算法 10](#_Toc6244065)

[5.1 原理 10](#_Toc6244066)

[5.2 代码 10](#_Toc6244067)

[6 遇到的问题 10](#_Toc6244068)

[7 实验运行结果 11](#_Toc6244069)

[7.1 Cannon算法 11](#_Toc6244070)

[7.2 Fox算法 11](#_Toc6244071)

[7.3 PSRS算法 11](#_Toc6244072)

[7.4 雅各比迭代算法 11](#_Toc6244073)

[8 结束语 11](#_Toc6244074)

[参考文献 11](#_Toc6244075)

# 实验步骤

## 环境准备

* 下载安装包

1. wget http://www.mpich.org/static/downloads/3.3/mpich-3.3.tar.gz

* 解压

1. tar -zxvf mpich-3.3.tar.gz

* 配置与检查

进入到解压的文件夹mpich-3.3下，并运行configure文件检查环境并配置环境。其中-prefix=后的参数为安装目录。--disable-fortran表示不安装fortran的环境。

1. ./configure -prefix=/usr/local/mpich --disable-fortran

运行configure文件检查环境，发现c++编译器有问题。



问题的原因是没有安装g++编译器。安装后即可进入下一步

* 编译与安装

还是在解压文件夹mpich-3.3的目录下运行以下指令。编译的时间比较长。

1. make
2. make install

* 配置环境变量

进入到root目录下。打开.bashrc文件

1. vim .bashrc

在该文件尾部添加：

1. export PATH=/usr/local/mpich/bin:$PATH

保存并退出 :wq

* 检查是否安装成功

通过使用 mpicc mpirun来查看是否有该指令，判断是否安装成功。

# 实验一 MPI实现Cannon算法

## 原理

Cannon算法是一种存储有效的算法。为了使两矩阵下标满足相乘的要求，它和上一节的并行分块乘法不同，不是仅仅让B矩阵的各列块循环移动，而是有目的地让A的各行块以及B的各列块皆施行循环移位，从而实现对C的子块的计算。将矩阵A和B分成p个方块Aij和Bij，(0)，每块大小为，并将它们分配给p个 处理器。

开始时处理器P­ij存放块Aij和Bij，并负责计算块Cij，然后算法开始执行：

* 将块Aij向左循环移动i步；将块Bij向上循环移动j步；
* Pij执行乘加运算后将块Aij向左循环移动1步，块Bij向上循环移动1步；
* 重复第⑵步，总共执行次乘加运算和次块Aij和Bij的循环单步移位。

伪代码

**Cannon乘法算法**

1. **输入：***An*×*n*，*Bn*×*n*
2. **输出**：*Cn*×*n*
3. **Begin**
4. 对所有处理器my\_rank(my\_rank=0,…,p-1)同时执行如下的算法:
5. 计算子块的行号 *i*=my\_rank/sqrt(*p*)
6. 计算子块的列号 *j*=my\_rank mod sqrt(*p*)
7. **for** *k*=0 **to** -1 **do**
8. **if** (*i*>*k*) **then** Leftmoveonestep(*a*) **end if** /\* *a*循环左移至同行相邻处理器中\*/
9. **if** (*j*>*k*) **then** Upmoveonestep(*b*) **end if**/\* b循环上移至同列相邻处理器中\*/
10. **end for**

## 源代码

1. #include<stdio.h>
2. #include<stdlib.h>
3. #include<mpi.h>
4. #include<math.h>
5. #include<time.h>
7. **int** allocMatrix(**int**\*\*\* mat, **int** rows, **int** cols) {
8. // Allocate rows\*cols contiguous items
9. **int**\* p = (**int**\*)malloc(**sizeof**(**int**\*) \* rows \* cols);
10. **if** (!p) {
11. **return** -1;
12. }
13. // Allocate row pointers
14. \*mat = (**int**\*\*)malloc(rows \* **sizeof**(**int**\*));
15. **if** (!mat) {
16. free(p);
17. **return** -1;
18. }
20. // Set up the pointers into the contiguous memory
21. **for** (**int** i = 0; i < rows; i++) {
22. (\*mat)[i] = &(p[i \* cols]);
23. }
24. **return** 0;
25. }
27. **int** freeMatrix(**int** \*\*\*mat) {
28. free(&((\*mat)[0][0]));
29. free(\*mat);
30. **return** 0;
31. }
33. **void** matrixMultiply(**int** \*\*a, **int** \*\*b, **int** rows, **int** cols, **int** \*\*\*c) {
34. **for** (**int** i = 0; i < rows; i++) {
35. **for** (**int** j = 0; j < cols; j++) {
36. **int** val = 0;
37. **for** (**int** k = 0; k < rows; k++) {
38. val += a[i][k] \* b[k][j];
39. }
40. (\*c)[i][j] = val;
41. }
42. }
43. }
45. **void** printMatrix(**int** \*\*mat, **int** size) {
46. **for** (**int** i = 0; i < size; i++) {
47. **for** (**int** j = 0; j < size; j++) {
48. printf("%d ", mat[i][j]);
49. }
50. printf("\n");
51. }
52. }
54. **void** printMatrixFile(**int** \*\*mat, **int** size, **FILE** \*fp) {
55. **for** (**int** i = 0; i < size; i++) {
56. **for** (**int** j = 0; j < size; j++) {
57. fprintf(fp,"%d ", mat[i][j]);
58. }
59. fprintf(fp,"\n");
60. }
61. }
63. **int** main(**int** argc, **char**\* argv[]) {
64. MPI\_Comm cartComm;
65. **int** dim[2], period[2], reorder;
66. **int** coord[2], id;
67. **FILE** \*fp;
68. **int** \*\*A = NULL, \*\*B = NULL, \*\*C = NULL;
69. **int** \*\*localA = NULL, \*\*localB = NULL, \*\*localC = NULL;
70. **int** \*\*localARec = NULL, \*\*localBRec = NULL;
71. **int** rows = 0;
72. **int** columns;
73. **int** count = 0;
74. **int** worldSize;
75. **int** procDim;
76. **int** blockDim;
77. **int** left, right, up, down;
78. **int** bCastData[4];
79. **clock\_t** start,end;
81. // Initialize the MPI environment
82. MPI\_Init(&argc, &argv);
83. // World size
84. MPI\_Comm\_size(MPI\_COMM\_WORLD, &worldSize);
86. // Get the rank of the process
87. **int** rank;
88. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);
89. **if** (rank == 0) {
90. **int** n;
91. **char** ch;
93. // Determine matrix dimensions
94. fp = fopen("A.txt", "r");
95. **if** (fp == NULL) {
96. MPI\_Abort(MPI\_COMM\_WORLD, 1);
97. }
98. **while** (fscanf(fp, "%d", &n) != EOF) {
99. ch = fgetc(fp);
100. **if** (ch == '\n') {
101. rows = rows + 1;
102. }
103. count++;
104. }
105. printf("count:%d rows:%d",count,rows);
106. columns = count / rows;
108. // Check matrix and world size
109. **if** (columns != rows) {
110. printf("[ERROR] Matrix must be square!\n");
111. MPI\_Abort(MPI\_COMM\_WORLD, 2);
112. }
113. **double** sqroot = sqrt(worldSize);
114. **if** ((sqroot - floor(sqroot)) != 0) {
115. printf("[ERROR] Number of processes must be a perfect square!\n");
116. MPI\_Abort(MPI\_COMM\_WORLD, 2);
117. }
118. **int** intRoot = (**int**)sqroot;
119. **if** (columns%intRoot != 0 || rows%intRoot != 0) {
120. printf("[ERROR] Number of rows/columns not divisible by %d!\n", intRoot);
121. MPI\_Abort(MPI\_COMM\_WORLD, 3);
122. }
123. procDim = intRoot;
124. blockDim = columns / intRoot;
126. fseek(fp, 0, SEEK\_SET);
128. **if** (allocMatrix(&A, rows, columns) != 0) {
129. printf("[ERROR] Matrix alloc for A failed!\n");
130. MPI\_Abort(MPI\_COMM\_WORLD, 4);
131. }
132. **if** (allocMatrix(&B, rows, columns) != 0) {
133. printf("[ERROR] Matrix alloc for B failed!\n");
134. MPI\_Abort(MPI\_COMM\_WORLD, 5);
135. }
137. // Read matrix A
138. **for** (**int** i = 0; i < rows; i++) {
139. **for** (**int** j = 0; j < columns; j++) {
140. fscanf(fp, "%d", &n);
141. A[i][j] = n;
142. }
143. }
144. printf("A matrix:\n");
145. printMatrix(A, rows);
146. fclose(fp);
148. // Read matrix B
149. fp = fopen("B.txt", "r");
150. **if** (fp == NULL) {
151. **return** 1;
152. }
153. **for** (**int** i = 0; i < rows; i++) {
154. **for** (**int** j = 0; j < columns; j++) {
155. fscanf(fp, "%d", &n);
156. B[i][j] = n;
157. }
158. }
159. printf("B matrix:\n");
160. printMatrix(B, rows);
161. fclose(fp);
163. **if** (allocMatrix(&C, rows, columns) != 0) {
164. printf("[ERROR] Matrix alloc for C failed!\n");
165. MPI\_Abort(MPI\_COMM\_WORLD, 6);
166. }
168. bCastData[0] = procDim;
169. bCastData[1] = blockDim;
170. bCastData[2] = rows;
171. bCastData[3] = columns;
172. }
174. start=clock();
175. // Create 2D Cartesian grid of processes
176. MPI\_Bcast(&bCastData, 4, MPI\_INT, 0, MPI\_COMM\_WORLD);
177. procDim = bCastData[0];
178. blockDim = bCastData[1];
179. rows = bCastData[2];
180. columns = bCastData[3];
182. dim[0] = procDim; dim[1] = procDim;
183. period[0] = 1; period[1] = 1;
184. reorder = 1;
185. MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, dim, period, reorder, &cartComm);
187. // Allocate local blocks for A and B
188. allocMatrix(&localA, blockDim, blockDim);
189. allocMatrix(&localB, blockDim, blockDim);
191. // Create datatype to describe the subarrays of the global array
192. **int** globalSize[2] = { rows, columns };
193. **int** localSize[2] = { blockDim, blockDim };
194. **int** starts[2] = { 0,0 };
195. MPI\_Datatype type, subarrtype;
196. MPI\_Type\_create\_subarray(2, globalSize, localSize, starts, MPI\_ORDER\_C, MPI\_INT, &type);
197. MPI\_Type\_create\_resized(type, 0, blockDim \* **sizeof**(**int**), &subarrtype);
198. MPI\_Type\_commit(&subarrtype);
200. **int** \*globalptrA = NULL;
201. **int** \*globalptrB = NULL;
202. **int** \*globalptrC = NULL;
203. **if** (rank == 0) {
204. globalptrA = &(A[0][0]);
205. globalptrB = &(B[0][0]);
206. globalptrC = &(C[0][0]);
207. }
209. // Scatter the array to all processors
210. **int**\* sendCounts = (**int**\*)malloc(**sizeof**(**int**) \* worldSize);
211. **int**\* displacements = (**int**\*)malloc(**sizeof**(**int**) \* worldSize);
213. **if** (rank == 0) {
214. **for** (**int** i = 0; i < worldSize; i++) {
215. sendCounts[i] = 1;
216. }
217. **int** disp = 0;
218. **for** (**int** i = 0; i < procDim; i++) {
219. **for** (**int** j = 0; j < procDim; j++) {
220. displacements[i \* procDim + j] = disp;
221. disp += 1;
222. }
223. disp += (blockDim - 1)\* procDim;
224. }
225. }
227. MPI\_Scatterv(globalptrA, sendCounts, displacements, subarrtype, &(localA[0][0]),
228. rows \* columns / (worldSize), MPI\_INT,
229. 0, MPI\_COMM\_WORLD);
230. MPI\_Scatterv(globalptrB, sendCounts, displacements, subarrtype, &(localB[0][0]),
231. rows \* columns / (worldSize), MPI\_INT,
232. 0, MPI\_COMM\_WORLD);
234. **if** (allocMatrix(&localC, blockDim, blockDim) != 0) {
235. printf("[ERROR] Matrix alloc for localC in rank %d failed!\n", rank);
236. MPI\_Abort(MPI\_COMM\_WORLD, 7);
237. }
239. // Initial skew
240. MPI\_Cart\_coords(cartComm, rank, 2, coord);
241. MPI\_Cart\_shift(cartComm, 1, coord[0], &left, &right);
242. MPI\_Sendrecv\_replace(&(localA[0][0]), blockDim \* blockDim, MPI\_INT, left, 1, right, 1, cartComm, MPI\_STATUS\_IGNORE);
243. MPI\_Cart\_shift(cartComm, 0, coord[1], &up, &down);
244. MPI\_Sendrecv\_replace(&(localB[0][0]), blockDim \* blockDim, MPI\_INT, up, 1, down, 1, cartComm, MPI\_STATUS\_IGNORE);
246. // Init C
247. **for** (**int** i = 0; i < blockDim; i++) {
248. **for** (**int** j = 0; j < blockDim; j++) {
249. localC[i][j] = 0;
250. }
251. }
253. **int**\*\* multiplyRes = NULL;
254. **if** (allocMatrix(&multiplyRes, blockDim, blockDim) != 0) {
255. printf("[ERROR] Matrix alloc for multiplyRes in rank %d failed!\n", rank);
256. MPI\_Abort(MPI\_COMM\_WORLD, 8);
257. }
258. **for** (**int** k = 0; k < procDim; k++) {
259. matrixMultiply(localA, localB, blockDim, blockDim, &multiplyRes);
261. **for** (**int** i = 0; i < blockDim; i++) {
262. **for** (**int** j = 0; j < blockDim; j++) {
263. localC[i][j] += multiplyRes[i][j];
264. }
265. }
266. // Shift A once (left) and B once (up)
267. MPI\_Cart\_shift(cartComm, 1, 1, &left, &right);
268. MPI\_Cart\_shift(cartComm, 0, 1, &up, &down);
269. MPI\_Sendrecv\_replace(&(localA[0][0]), blockDim \* blockDim, MPI\_INT, left, 1, right, 1, cartComm, MPI\_STATUS\_IGNORE);
270. MPI\_Sendrecv\_replace(&(localB[0][0]), blockDim \* blockDim, MPI\_INT, up, 1, down, 1, cartComm, MPI\_STATUS\_IGNORE);
271. }
273. // Gather results
274. MPI\_Gatherv(&(localC[0][0]), rows \* columns / worldSize, MPI\_INT,
275. globalptrC, sendCounts, displacements, subarrtype,
276. 0, MPI\_COMM\_WORLD);
278. freeMatrix(&localC);
279. freeMatrix(&multiplyRes);
281. end=clock();
282. // Finalize the MPI environment
284. **if** (rank == 0) {
285. **clock\_t** sstart,send;
286. sstart=clock();
287. **int** i, j, k;
288. printf("C is:\n");
289. printMatrix(C, rows);
290. **for** (i = 0; i<columns; i++)
291. **for** (j = 0; j<rows; j++)
292. **for** (k = 0; k<columns; k++)
293. C[i][j] += A[i][k] \* B[k][j];
294. send=clock();
295. printf("\nSpeed up:%lf\n",(**double**)(send-start)/(end-start));
296. }
297. MPI\_Finalize();
299. **return** 0;
300. }

# 实验二 MPI实现Fox算法

## 原理

分块: 同Cannon分块算法

算法原理：

(1)Ai,i向所在行的其他处理器进行一到多播送；

(2)处理器将收到的A块与原有的B块进行乘-加运算;

(3) B块向上循环移动一步;

(4)如果Ai,j是上次第i行播送的块，本次选择向所在行的其他处理器进行一到多播送;

(5)转(2)执行次;

## 源代码

1. #include "mpi.h"
2. #include <algorithm>
3. #include <fstream>
4. #include <cmath>
5. #include<cstring>
7. **const** **int** root\_id = 0;
8. **const** **int** max\_procs\_size = 16;
10. **int** main(**int** argc,**char** \*argv[])
11. {
12. **double** start\_time, end\_time, time;
13. **int** procs\_id, procs\_size;
14. MPI\_Status status;
15. MPI\_Request reqSend, reqRecv;
16. MPI\_Init(&argc,&argv);
17. start\_time = MPI\_Wtime();
18. MPI\_Comm\_size(MPI\_COMM\_WORLD,&procs\_size);
19. MPI\_Comm\_rank(MPI\_COMM\_WORLD,&procs\_id);
21. // 参数检查
22. **int** N=4;
23. {
24. **for** (**int** i=1; i<argc; ++i ){
25. **char** \* pos =strstr(argv[i], "-N=");
26. **if** ( pos!=NULL){
27. sscanf(pos, "-N=%d", &N);
28. **break**;
29. }
30. }
31. }
33. **const** **int** procs\_size\_sqrt = floor(sqrt(**static\_cast**<**double**>(procs\_size)));
34. **const** **int** n = N / procs\_size\_sqrt;
35. **const** **int** n\_sqr = n\*n;
37. **if** (procs\_size<4 || procs\_size> max\_procs\_size) {
38. printf("The fox algorithm requires at least 4 processors and at most %d processors. ",
39. max\_procs\_size);
40. MPI\_Finalize();
41. **return** 0;
42. }
43. **if** (procs\_size\_sqrt\*procs\_size\_sqrt != procs\_size ) {
44. printf("The number of process must be a square. ");
45. MPI\_Finalize();
46. **return** 0;
47. }
48. **if** (N % procs\_size\_sqrt !=0) {
49. printf("N mod procs\_size\_sqrt !=0  ");
50. MPI\_Finalize();
51. **return** 0;
52. }
54. //初始化矩阵
55. **int** \*A = **new** **int**[n\_sqr];
56. **int** \*B = **new** **int**[n\_sqr];
57. **int** \*C = **new** **int**[n\_sqr];
58. **int** \*T = **new** **int**[n\_sqr];
60. **for** (**int** i=0; i<n; ++i)
61. **for** (**int** j=0; j<n; ++j) {
62. A[i\*n+j] = (i+j)\*procs\_id;
63. B[i\*n+j] = (i-j)\*procs\_id;
64. C[i\*n+j] = 0;
65. }
67. //输出矩阵
68. printf("\nA on procs %d :  \n", procs\_id);
69. **for** (**int** i=0; i<n; ++i) {
70. **for** (**int** j=0; j<n; ++j) {
71. printf("%5d",A[i\*n+j]);
72. }
73. printf("\n");
74. }
76. printf("\nB on procs %d :\n", procs\_id);
77. **for** (**int** i=0; i<n; ++i) {
78. **for** (**int** j=0; j<n; ++j) {
79. printf("%5d",B[i\*n+j]);
80. }
81. printf("\n");
82. }
84. // 划分组, 建立子通信空间
85. MPI\_Comm cart\_all, cart\_row, cart\_col;
86. **int** dims[2], periods[2];
87. **int** procs\_cart\_rank, procs\_coords[2];
88. dims[0] = dims[1] = procs\_size\_sqrt;
89. periods[0] = periods[1] = **true**;
90. MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, dims, periods, **false**, &cart\_all);
91. MPI\_Comm\_rank(cart\_all, &procs\_cart\_rank);
92. MPI\_Cart\_coords(cart\_all, procs\_cart\_rank,  2, procs\_coords);
93. MPI\_Comm\_split(cart\_all, procs\_coords[0], procs\_coords[1], &cart\_row);
94. MPI\_Comm\_split(cart\_all, procs\_coords[1], procs\_coords[0], &cart\_col);
95. **int** rank\_cart\_row, rank\_cart\_col;
96. MPI\_Comm\_rank(cart\_row, & rank\_cart\_row);
97. MPI\_Comm\_rank(cart\_col, & rank\_cart\_col);
99. // 计算并传递
100. **for** (**int** round = 0; round < procs\_size\_sqrt; ++ round) {
102. MPI\_Isend(B, n\_sqr, MPI\_INT, (procs\_coords[0] - 1 + procs\_size\_sqrt) % procs\_size\_sqrt,
103. 1, cart\_col, &reqSend);
105. **int** broader = (round + procs\_coords[0]) % procs\_size\_sqrt;
106. **if** (broader == procs\_coords[1]) std::copy(A,A+n\_sqr,T);
108. MPI\_Bcast(T, n\_sqr, MPI\_INT, broader , cart\_row);
110. **for** (**int** row=0; row<n; ++row)
111. **for** (**int** col=0; col<n; ++col)
112. **for** (**int** k=0; k<n; ++k) {
113. C[row\*n+col] += T[row\*n+k]\*B[k\*n+col];
114. }
116. MPI\_Wait(&reqSend, &status);
117. MPI\_Recv(T, n\_sqr,  MPI\_INT, (procs\_coords[0] + 1) % procs\_size\_sqrt
118. , 1, cart\_col, &status);
119. std::copy(T,T+n\_sqr,B);
121. }
123. //输出结果
124. printf("\nC on procs %d :\n", procs\_id);
125. **for** (**int** i=0; i<n; ++i) {
126. **for** (**int** j=0; j<n; ++j) {
127. printf("%5d",C[i\*n+j]);
128. }
129. printf("\n");
130. }


134. // 释放
135. MPI\_Comm\_free(&cart\_col);
136. MPI\_Comm\_free(&cart\_row);
137. MPI\_Comm\_free(&cart\_all);
138. **delete** []A;
139. **delete** []B;
140. **delete** []C;
141. **delete** []T;
142. end\_time = MPI\_Wtime();
143. MPI\_Finalize();
144. printf("task %d consumed %lf seconds ", procs\_id, end\_time-start\_time);
145. **return** 0;
146. }

# 实验三 并行正则采样排序（PSRS）

## 原理

并行正则采样排序，简称PSRS(Parallel Sorting by Regular Sampling)它是一种基于均匀划分（Uniform Partition）原理的负载均衡的并行排序算法。假定待排序的元素有n个，系统中有p个处理器，那么系统首先将n个元素均匀地分割成p段，每段含有n/p个元素，每段指派一个处理器，然后各个处理器同时施行局部排序。为了使各段中诸局部有序的元素在整个序列中也能占据正确的位置，那么就首先从各段中抽取几个代表元素，再从他们产生出p-1个主元，然后按这些主元与原各局部有序中的元素之间的偏序关系，将各个局部有序段划分成p段，接着通过全局交换将各个段中的对应部分集合在一起，最后将这些集合在一起的各部分采用多路归并的方法进行排序，这些有序段汇合起来就自然成为全局有序序列了。

## 源代码

1. # define NDEBUG
2. # include <stdio.h>
3. # include <stdlib.h>
4. # include <assert.h>
5. # include <time.h>
6. # include <mpi.h>
8. //================================================================================
9. //
10. // Function name : cmp
11. //
12. // Function : input parameter for qsort (sort integer)
13. //
14. //================================================================================
15. **int** cmp (**const** **void** \*a, **const** **void** \*b)
16. {
17. **return** (\*(**int** \*)a - \*(**int** \*)b);
18. }
20. **int** main (**int** argc, **char** \*argv[])
21. {
22. **if** (argc != 2)
23. {
24. printf ("\n Error : the number of input parameters is wrong ! \n");
25. exit (EXIT\_FAILURE);
26. }
28. **long** n = atol (argv[1]);                            // size of array
29. **long** \*a\_all = NULL;                                 // array a
31. **long** i, j, k;
32. **int** size, rank;
33. **double** begin, end, t;
35. // initialize MPI environment :
36. MPI\_Init (&argc, &argv);
37. MPI\_Comm\_size (MPI\_COMM\_WORLD, &size);
38. MPI\_Comm\_rank (MPI\_COMM\_WORLD, &rank);
40. // phase 1 : Initialization
41. **if** (rank == 0)
42. {
43. a\_all = (**long** \*)calloc (n, **sizeof** (**long**));
44. assert (a\_all != NULL);
46. srand (time (NULL));
47. **for** (i = 0; i < n; i++){
48. a\_all[i] =1+(**int**)(50.0\*rand()/(RAND\_MAX+1.0));
49. printf("%ld ",a\_all[i]);
50. }
51. //        a\_all[0] = 15; a\_all[1] = 46; a\_all[2] = 48; a\_all[3] = 93; a\_all[4] = 39; a\_all[5] = 6; a\_all[6] = 72; a\_all[7] = 91; a\_all[8] = 14;
52. //        a\_all[9] = 36; a\_all[10] = 69; a\_all[11] = 40; a\_all[12] = 89; a\_all[13] = 61; a\_all[14] = 97; a\_all[15] = 12; a\_all[16] = 21; a\_all[17] = 54;
53. //        a\_all[18] = 53; a\_all[19] = 97; a\_all[20] = 84; a\_all[21] = 58; a\_all[22] = 32; a\_all[23] = 27; a\_all[24] = 33; a\_all[25] = 72; a\_all[26] = 20;
54. }
56. // begin clock :
57. begin = MPI\_Wtime ();
59. // phase 2 : Scatter data, local sort and regular samples collecte
60. **long** n\_per = n / size;
61. **long** \*a = (**long** \*)calloc (n\_per, **sizeof** (**long**));
62. assert (a != NULL);
64. MPI\_Scatter (a\_all, n\_per, MPI\_LONG, a, n\_per, MPI\_LONG, 0, MPI\_COMM\_WORLD);
65. qsort (a, n\_per, **sizeof** (**long**), cmp);
67. **long** \*samples = (**long** \*)calloc (size, **sizeof** (**long**));
68. assert (samples != NULL);
69. **for** (i = 0; i < size; i++)
70. samples[i] = a[i \* size];
72. // phase 3 : Gather and merge samples, choose and broadcast (size - 1) pivots
73. **long** \*samples\_all;
74. **if** (rank == 0)
75. {
76. samples\_all = (**long** \*)calloc ((size \* size), **sizeof** (**long**));
77. assert (samples\_all != NULL);
78. }
79. MPI\_Gather (samples, size, MPI\_LONG, samples\_all, size, MPI\_LONG, 0, MPI\_COMM\_WORLD);
81. **long** \*pivots = (**long** \*)calloc ((size - 1), **sizeof** (**long**));
82. assert (pivots != NULL);
83. **if** (rank == 0)
84. {
85. qsort (samples\_all, (size \* size), **sizeof** (**long**), cmp);
86. **for** (i = 0; i < (size - 1); i++)
87. pivots[i] = samples\_all[(i + 1) \* size];
88. }
89. MPI\_Bcast (pivots, (size - 1), MPI\_LONG, 0, MPI\_COMM\_WORLD);
91. // phase 4 : Local data is partitioned
92. **int** index = 0;
93. **int** \*partition\_size = (**int** \*)calloc (size, **sizeof** (**int**));
94. assert (partition\_size != NULL);
95. **for** (i = 0; i < n\_per; i++)
96. {
97. **if** (a[i] > pivots[index])
98. {
99. index += 1;
100. }
101. **if** (index == (size - 1))
102. {
103. partition\_size[index] = n\_per - i;
104. **break**;
105. }
106. partition\_size[index]++;
107. }
109. // phase 5 : All ith classes are gathered and merged
110. **int** \*new\_partition\_size = (**int** \*)calloc (size, **sizeof** (**int**));
111. assert (new\_partition\_size != NULL);
112. MPI\_Alltoall (partition\_size, 1, MPI\_INT, new\_partition\_size, 1, MPI\_INT, MPI\_COMM\_WORLD);
114. **int** totalsize = 0;
115. **for** (i = 0; i < size; i++)
116. totalsize += new\_partition\_size[i];
117. **long** \*new\_partitions = (**long** \*)calloc (totalsize, **sizeof** (**long**));
118. assert (new\_partitions != NULL);
120. **int** \*send\_dis = (**int** \*)calloc (size, **sizeof** (**int**));
121. assert (send\_dis != NULL);
122. **int** \*recv\_dis = (**int** \*)calloc (size, **sizeof** (**int**));
123. assert (recv\_dis != NULL);
124. send\_dis[0] = 0;
125. recv\_dis[0] = 0;
126. **for** (i = 1; i < size; i++)
127. {
128. send\_dis[i] = send\_dis[i - 1] + partition\_size[i - 1];
129. recv\_dis[i] = recv\_dis[i - 1] + new\_partition\_size[i - 1];
130. }
131. MPI\_Alltoallv (a, partition\_size, send\_dis, MPI\_LONG, new\_partitions, new\_partition\_size, recv\_dis, MPI\_LONG, MPI\_COMM\_WORLD);
132. qsort (new\_partitions, totalsize, **sizeof** (**long**), cmp);
134. // phase 6 : Root processor collects all the data
135. **int** \*recv\_count;
136. **if** (rank == 0)
137. {
138. recv\_count = (**int** \*)calloc (size, **sizeof** (**int**));
139. assert (recv\_count != NULL);
140. }
141. MPI\_Gather (&totalsize, 1, MPI\_INT, recv\_count, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);
142. **if** (rank == 0)
143. {
144. recv\_dis[0] = 0;
145. **for** (i = 1; i < size; i++)
146. recv\_dis[i] = recv\_dis[i - 1] + recv\_count[i - 1];
147. }
148. MPI\_Gatherv (new\_partitions, totalsize, MPI\_LONG, a\_all, recv\_count, recv\_dis, MPI\_LONG, 0, MPI\_COMM\_WORLD);
150. // end clock :
151. end = MPI\_Wtime ();
152. t = (end - begin);
153. MPI\_Barrier (MPI\_COMM\_WORLD);
155. // find the maximum running time :
156. **double** \*t\_all;
157. **if** (rank == 0)
158. {
159. t\_all = (**double** \*)calloc (size, **sizeof** (**double**));
160. assert (t\_all != NULL);
161. }
162. MPI\_Gather (&t, 1, MPI\_DOUBLE, t\_all, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
163. **if** (rank == 0)
164. {
165. // output sorting result :
167. **for** (i = 0; i < n; i++)
168. printf ("%ld ", i, a\_all[i]);
169. printf("\n");
170. **double** t\_max = t\_all[0];
171. **for** (i = 1; i < size; i++)
172. {
173. **if** (t\_max < t\_all[i])
174. t\_max = t\_all[i];
175. }
176. printf ("\n t\_max = %lf \n", t\_max);
177. }
179. // free memory :
180. **if** (rank == 0)
181. {
182. free (a\_all);
183. free (samples\_all);
184. free (t\_all);
185. }
186. free (a);
187. free (samples);
188. free (pivots);
189. free (partition\_size);
190. free (new\_partition\_size);
191. free (new\_partitions);
192. free (send\_dis);
193. free (recv\_dis);
194. free (recv\_count);
196. **return** 0;
197. }

# OMP实验雅各比迭代算法

## 原理

考虑线性方程组Ax = b时，一般当A为低阶稠密矩阵时，用主元消去法解此方程组是有效方法。但是，对于由工程技术中产生的大型稀疏矩阵方程组（A的阶数很高，但零元素较多，例如求某些偏微分方程数值解所产生的线性方程组），利用迭代法求解此方程组就是合适的，在计算机内存和运算两方面，迭代法通常都可利用A中有大量零元素的特点。雅克比迭代法就是众多迭代法中比较早且较简单的一种，其命名也是为纪念普鲁士著名数学家雅可比。

迭代流程：

* 首先将方程组中的系数矩阵A分解成三部分，即：A = L+D+U，如图1所示，其中D为对角阵，L为下三角矩阵，U为上三角矩阵。
* 之后确定迭代格式，X^(k+1) = B\*X^(k) +f ，（这里^表示的是上标，括号内数字即迭代次数），如图2所示，其中B称为迭代矩阵，雅克比迭代法中一般记为J。（k = 0,1，......）
* 再选取初始迭代向量X^(0)，开始逐次迭代。

雅克比迭代法的优点明显，计算公式简单，每迭代一次只需计算一次矩阵和向量的乘法，且计算过程中原始矩阵**A**始终不变，比较容易并行计算。然而这种迭代方式收敛速度较慢，而且占据的存储空间较大，所以工程中一般不直接用雅克比迭代法，而用其改进方法。

## 代码

1. #include <omp.h>
2. #include<stdio.h>
3. #include <time.h>
4. **static** **long** num\_steps = 100000;
5. **double** step;
6. #include<Windows.h>
7. #include<iostream>
8. #include<math.h>
9. **using** **namespace** std;
11. #define epx 0.5 \* 1e-5
12. #define M 10000000 //最大迭代次数
13. #define N 5  //方程组的阶数
14. #define tread\_count 4
16. **int** main() {
17. **clock\_t** start=clock();
18. **double** a[N][N] = { { 0,0,0,0,0 },{ 0,8,7,0,0 },{ 0,6, 12, 5, 0 },{ 0,0, 4, 9, 3 },{ 0, 0, 0, 1, 2 } };
19. **double** b[N] = { 0, 0, -2, 8,6 };
20. **double** y[N]; //结果向量
21. **double** x[N];
22. **double** norm;
23. memset(y, 0, **sizeof**(y));
24. **int** i, j;
25. **int** flag;
26. **double** result;
28. **for**(**int** k=0;k<M;k++)
29. {
30. flag = 0;
31. //# pragma omp parallel for
32. **for** (i = 0; i < N; i++) {
33. x[i] = y[i];
34. }
36. //#pragma omp parallel for
37. **for** (i = 1; i < N; i++) {
38. result = 0;
39. **for** (j = 1; j < N; j++) {
40. **if** (i != j) {
41. result += a[i][j] \* y[j];
42. }
43. }
44. y[i] = (b[i] - result) / a[i][i]; //求出解空间
45. }
47. //求y -x 的范数
48. norm = 0.0;
49. //#pragma omp parallel for
50. **for** (i = 1; i < N; i++) {
51. **if** (fabs(x[i] - y[i]) > norm) {
52. norm = fabs(x[i] - y[i]);
53. }
54. }
56. **if** (norm < epx) {
57. **break**;
58. }
59. }
60. **clock\_t** finish = clock();
61. **double** duration = (**double**)(finish - start) / CLOCKS\_PER\_SEC;
62. //打印结果s
63. **for** (i = 1; i < N; i++) {
64. printf("%-4lf\t", y[i]);
65. }
66. printf("\nTime used:%lf s\n", duration);
68. **return** 0;
69. }

# 实验结果与性能分析

## Cannon算法

图7.1.1为cannon算法的运行结果。其中串行算法使用的是普通的矩阵乘法。代码如下所示：

     for (i = 0; i<columns; i++)

         for (j = 0; j<rows; j++)

             for (k = 0; k<columns; k++)

                 C[i][j] += A[i][k] \* B[k][j];

对于4x4的矩阵而言，使用cannon算法的加速比约为1.041812，此加速比并不是很高，一个主要原因是矩阵的维度太小，使用并行算法的开销比计算的开销要大，因而会出现加速较少或者甚至速度变慢的情况。

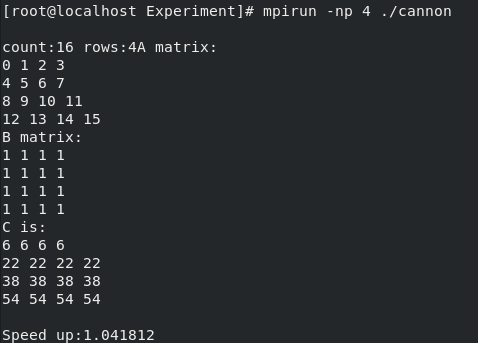


图7.1.1 cannon算法运行结果

## PSRS算法

图7.2.1-2分别为PSRS算法对100个数以及10个数排序的运行结果以及加速比，可以看到在对100个数进行排序时，其加速比约为0.017，而对10个数进行排序的加速比约为：0.0015，将近相差了10倍。出现加速比如此低的一个原因，也是因为排序的数字太少，问题的规模较小，计算开销小于使用并行计算的开销。

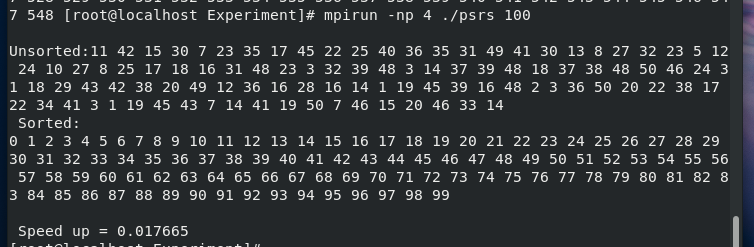


图7.2.1 100个数的PSRS

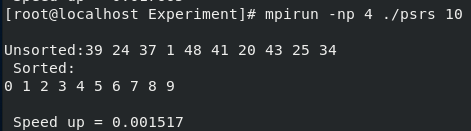


图7.2.2 10个数的PSRS

## 雅各比迭代算法

图7.4.1 为雅可比迭代算法的运行结果，第一行为计算的方程的根，其方程为：

double a[N][N] = { { 0,8,5,2,1 },{ 0,8,7,0,0 },{ 0,6, 12, 5, 0 },{ 0,0, 4, 9, 3 },{ 3, 2, 0, 1, 2 } };

double b[N] = { 0, 0, -2, 8,6 };

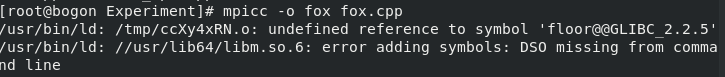
经验算，结果正确。其加速比约为7.5%，这意味着使用omp平行计算后速度并没有变快，而是变慢了，这也是由于计算的矩阵维度较小的原因，计算的开销远没有达到使用并行算法的开销，因而效率不增反降。



图7.4.1 雅可比迭代算法并行计算运行结果

# 遇到的问题

在编译mpi程序时遇到如下错误：



经过查询，发现其实是因为头文件的问题。当引用了数学函数时，需要链接数学的库

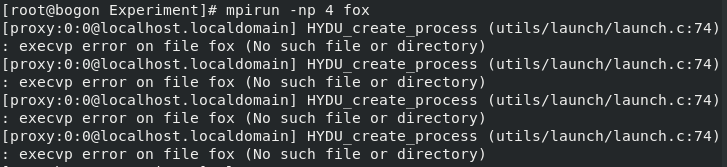
If your code includes mathematical functions (like exp, cos, etc.), you need to link to the mathematics library libm.so. This is done, just like for serial compiling, by adding -lm to the end of your compile command, that is,

gcc -o test test.cpp -lstdc++ -lm

因此在编译时需使用如下的代码:

Mpicc -o fox fox.cpp -lstdc++ -lm

另外需要注意的是，运行时目标文件需制定路径，否则会出现如下的错误：



正确的指令为：

mpirun -np 4 ./fox

# 实验心得体会

此次实验主要了解了并行程序的编写，了解了MPI以及OpenMP程序的结构，以及他们特有的语言规范。并行计算在大型工程中十分实用，可以大大加快很多计算方法，让复杂方程的求解成为现实。掌握并行程序的编写可以一定程度上提高程序的效率。

本次实验中，有几个算法并行时间效率反而变差，这与解决问题的计算量有关。当然也与，编写的程序有关。虽然本次实验，简单的实现了一些MPI以及OpenMP的程序，但要掌握两种并行规范的精髓，还是需要花很多时间研究的。

# 参考文献

1. <https://blog.csdn.net/reborn_lee/article/details/80959509>
2. <https://baike.baidu.com/item/%E9%9B%85%E5%85%8B%E6%AF%94%E8%BF%AD%E4%BB%A3%E6%B3%95/8736410>