## Cannon算法MPI实现

### cannon算法实现思路

1. 进程0将数组A[100][150],B[150][250]从文件中读入内存；
2. 进程0将数组A,B划分成25个小块，按照进程编号，分别将A,B的小块发送给各个进程；(为了保证接下来的逻辑处理合理，我们特意将0号进程加入计算过程中)
3. 各个进程接收数组A[20][30],B[30][50]，其中0号进程不需要使用MPI\_Send，直接使用memcpy即可。
4. 在开始cannon算法前需要坐标对齐，矩阵块必须定位对齐；以5×5的矩阵块举例：

对于矩阵A，第i行所有的矩阵块需要左移i次，如下图所示：



同理可得，对于矩阵B，第j列所有的矩阵块需要上移j次，如下图所示：



1. 记我们所用的进程数目为：nprocs。root=sqrt(nprocs)，矩阵被划分成root\*root个小块。

对于每个进程：

* 先计算每个小块A[20][30]×B[30][50]的值，结果存入数组C[20][50]；
* 再将矩阵A向左平移，矩阵B向上平移，平移的策略按照上体所示的结构给出；
* 对于进程号p，我们需要计算出它所在的行号row和列号column。
* 对于位于[row][column]的矩阵A，需要平移至[row][column-1]，使用MPI\_Sendrecv指令;
* 对于位于[row][column]的矩阵B，需要平移至[row-1][column] 使用MPI\_Sendrecv指令;
* 上诉过程需要循环root次，循环结束后，计算结束了，将结果数组C发送给进程0。进程0 根据回收的数组来自的进程编号，拼接还原结果数组结果C[100][250]。

1. 将数组C保存至文件c.txt，并打印cannon算法耗费的时间。

### cannon算法流程图[1]



### cannon算法复杂度分析[2]

矩阵A,B和C分成的方块，大小均为，p个处理器编号为存放；

1. 所有块向左循环移动i步；所有向上循环移动j步；
2. 所有处理器做执行乘-加法运算；
3. A的每个块上做循环移动一步；B的每个块向上循环移动一步；
4. 转b)执行次

|  |  |
| --- | --- |
| 运算步 | 执行时间 |
| a)步 |  |
| b)步 |  |
| c)步 |  |
|  |  |

### 分析算法用到的设计方法和模式

1. **任务划分(Partitioning)**

数据分解方法有以下几种：本次实验采用2D的block，block方法。



功能分解：

采用分布式计算模型，放弃主从模型(master-slave)，各个节点地位相同，计算方式相同。

1. **通讯分析(Communication)**

通讯方式有：

局部通讯：通讯限制在一个邻域内



全局通讯：



结构化通讯：



非结构化通讯：



本次实验采用的通讯模型[2]：



1. **任务组合(Agglomeration)**

根据表面-容积效应：通讯量与任务子集的表面积（尺寸）成正比，计算量与任务子集的体积（尺寸平方）成正比。

为了减少通讯成本，我们需要增加计算粒度。对于矩阵A[100][150],B[150][250]，我们将其划分为A[20][30],B[30][50].

1. **处理器映射(Mapping)**

考虑到负载平衡和任务调度，我们采用*2-D mesh*拓扑结构



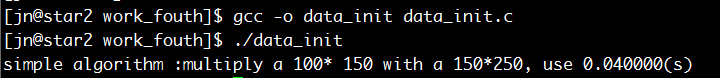
映射方法：由于本次实验中，每个进程与一个矩阵快一一对应，故不需要采用映射。

任务调度算法：本次实验采用非集中模式，降低通讯耗时，减少部分节点计算量过大的情况。

### 实验结果

下面的时间结果均只统计核心算法耗费时间，不考虑数据预处理时间和结果整理时间。

1. 不采用cannon算法，只用简单的循环计算结果耗时：0.04秒

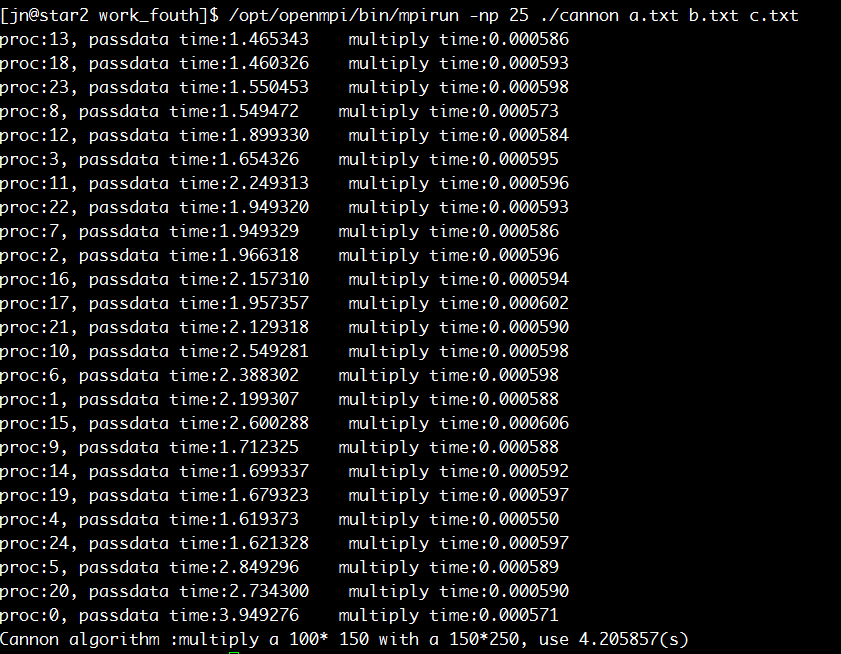


1. 采用cannon算法，耗时：



这个结果令人难以置信，为什么使用优化后的并行算法时间反而变长了100倍，

为此专门统计了每个线程花费的矩阵计算时间和通信花费的时间：



可以看到，我们统计了每个线程计算完成cannon花费时间，用于矩阵乘法时间确实比普通时间节省很多，但是传递大数组花费了大量时间，这样做未免得不偿失。

1. 使用上次小实验的pthread矩阵算法稍作修改，也统计了耗时：



相比于cannon算法，我们的实验效果已经蛮好了，但是在这里我统计的时间从进程0开始发数据开始，到进程0 回收结果结果结束。中间经历了大量pthread初始化的时间耗费，假若我们忽略这些初始化耗费的时间占用，我们还可以得到更优秀的结果：



这个结果已经比简单的算法快了一个数量级，效果很好

### 编写程序实现

|  |
| --- |
| #include "mpi.h"  #include <math.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  void scatter\_matrix(int\* fstream,int n1,int n2,int\*Q,int root,int tag){  /\*每个矩阵块的大小\*/  int rows=(n1+root-1)/root;  int cols=(n2+root-1)/root;  int\* tmp\_matrix=(int\*)malloc(rows\*cols\*sizeof(int));    int i,j;  memset(Q,0,rows\*cols\*sizeof(int));  for(i=0;i<root;i++){  for(j=0;j<root;j++){  int p=0,q=0;  int imin=i\*rows\*n2;  int jmin=j\*cols;  memset(tmp\_matrix,0,sizeof(tmp\_matrix));  /\*在划分矩阵时，由于地空间不连续，需要另开辟一个数组连续的保存起来，以便于调用MPI\_Send\*/  for(p=0;p<rows;p++,imin+=n2){  for(q=0;q<cols;q++){  tmp\_matrix[p\*cols+q]=fstream[imin+jmin+q];  }  }  if(i==0&&j==0){  /\*进程0 不需要使用MPI\_Send将数据发送给自己，直接使用memcpy将结果拷贝即可\*/  memcpy(Q,tmp\_matrix,rows\*cols\*sizeof(int));  }else{  /\*将分块发送给位于i行，j列的进程\*/  MPI\_Send(tmp\_matrix,rows\*cols,MPI\_INT,i\*root+j,tag,MPI\_COMM\_WORLD);  }  }  }  }  /\*  \*@row:矩阵所在的行  \*@col:矩阵所在的列  \*@sp:sp=root=sqrt(nprocs)  \*@return 根据行列号计算进程实际编号  \*/  int get\_index(int row,int col,int sp){  int tmp=((row+sp)%sp)\*sp+(col+sp)%sp;  return tmp;  }  /\*计算矩阵乘法，将结果存入C中\*/  void matrix\_multi(int\* A,int \*B,int \*C,int n1,int n2,int n3,int myid){  int i=0,j=0,k=0;  int\* tmp\_C=(int\*)malloc(n1\*n3\*sizeof(int));  memset(tmp\_C,0,sizeof(int)\*n1\*n3);    for(i=0;i<n1;i++){  for(j=0;j<n3;j++){  for(k=0;k<n2;k++){  tmp\_C[i\*n3+j]+=A[i\*n2+k]\*B[k\*n3+j];  }  C[i\*n3+j]+=tmp\_C[i\*n3+j];  }    }    }  /\*用于矩阵下标定位对齐\*/  void shuffle(int\*A,int\*buf\_A,int buf\_A\_size,int \*B,int\*buf\_B,int buf\_B\_size,int root,int myid){  int i,j;  MPI\_Status status;  int cur\_col=0;  int cur\_row=0;  /\*通过进程编号计算获得当前进程所在的行号和列号\*/  cur\_row=myid/root;  cur\_col=myid-cur\_row\*root;  /\*对于矩阵A，第i行的矩阵需要向左平移i次\*/  for(i=0;i<cur\_row;i++){  /\*接收来自右边的数据，并将当前矩阵发送给左边的进程\*/  MPI\_Sendrecv(A,buf\_A\_size,MPI\_INT,get\_index(cur\_row,cur\_col-1,root),102,  buf\_A,buf\_A\_size,MPI\_INT,get\_index(cur\_row,cur\_col+1,root),102,MPI\_COMM\_WORLD,&status);  memcpy(A,buf\_A,buf\_A\_size\*sizeof(int));/\*buf\_A用于通信时缓存矩阵\*/  memset(buf\_A,0,buf\_A\_size\*sizeof(int));  }  /\*对于矩阵B，第j列的矩阵需要向上平移j次\*/  for(j=0;j<cur\_col;j++){  /\*接收来自下边的数据，并将当前矩阵发送给上边的进程\*/  MPI\_Sendrecv(B,buf\_B\_size,MPI\_INT,get\_index(cur\_row-1,cur\_col,root),103,  buf\_B,buf\_B\_size,MPI\_INT,get\_index(cur\_row+1,cur\_col,root),103,MPI\_COMM\_WORLD,&status);  memcpy(B,buf\_B,buf\_B\_size\*sizeof(int));/\*buf\_B用于通信时缓存矩阵\*/  memset(buf\_B,0,buf\_B\_size\*sizeof(int));  }  /\*printf("I have shuffled!\n");\*/  }  void cannon(int\*A,int\*buf\_A,int buf\_A\_size,int \*B,int\*buf\_B,int buf\_B\_size,  int \*C,int buf\_C\_size,int row\_a,int col\_a,int col\_b,int root,int myid){  MPI\_Status status;  double elapsed\_time,multiply\_time=0,passdata\_time=0;  int i,j;  memset(C,0,sizeof(int)\*buf\_C\_size);  int cur\_col=0;  int cur\_row=0;  /\*通过进程编号计算获得当前进程所在的行号和列号\*/  cur\_row=myid/root;  cur\_col=myid-cur\_row\*root;    for(i=0;i<root;i++){/\*一共需要循环root次，root=sqrt(nprocs)\*/  elapsed\_time=MPI\_Wtime();  matrix\_multi(A,B,C,row\_a,col\_a,col\_b,myid);/\*计算矩阵乘法\*/  elapsed\_time=MPI\_Wtime()-elapsed\_time;  multiply\_time+=elapsed\_time;  /\*elapsed\_time=MPI\_Wtime(); \*/  /\*接收来自右边(row,col+1)的数据，并将当前矩阵发送给左边(row,col-1)的进程\*/  MPI\_Sendrecv(A,buf\_A\_size,MPI\_INT,get\_index(cur\_row,cur\_col-1,root),102,  buf\_A,buf\_A\_size,MPI\_INT,get\_index(cur\_row,cur\_col+1,root),102,MPI\_COMM\_WORLD,&status);  /\*接收来自下边(row+1,col)的数据，并将当前矩阵发送给上边(row-1,col)的进程\*/  MPI\_Sendrecv(B,buf\_B\_size,MPI\_INT,get\_index(cur\_row-1,cur\_col,root),103,  buf\_B,buf\_B\_size,MPI\_INT,get\_index(cur\_row+1,cur\_col,root),103,MPI\_COMM\_WORLD,&status);  /\*elapsed\_time=MPI\_Wtime()-elapsed\_time;  passdata\_time+=elapsed\_time;\*/  memcpy(B,buf\_B,buf\_B\_size\*sizeof(int));/\*将buf\_B中的数据拷贝至B中\*/  memcpy(A,buf\_A,buf\_A\_size\*sizeof(int));/\*将buf\_A中的数据拷贝至A中\*/    }  /\*将计算结果发送给数组C\*/  MPI\_Send(C,row\_a\*col\_b,MPI\_INT,0,104,MPI\_COMM\_WORLD);    printf("proc:%d, passdata time:%lf multiply time:%lf\n",myid,passdata\_time,multiply\_time);  }    void gather\_matrix(int \*fstream,int n1,int n3,int\*C,int root,FILE\*fhc){  MPI\_Status status;  int rows=(n1+root-1)/root;  int cols=(n3+root-1)/root;  int\* tmp\_matrix=(int\*)malloc(rows\*cols\*sizeof(int));  int i,j;  for(i=0;i<root;i++){  for(j=0;j<root;j++){  int p,q;  int imin=i\*rows\*n3;  int jmin=j\*cols;  memset(tmp\_matrix,0,sizeof(tmp\_matrix));  /\*接收来自各个进程的数据\*/  MPI\_Recv(tmp\_matrix,rows\*cols,MPI\_INT,i\*root+j,104,MPI\_COMM\_WORLD,&status);  /\*printf("I am passed proc:%d \n",i\*root+j);\*/  /\*将接收的矩阵tmp拼接到矩阵C中去，需要按照合理顺序拼接，否则结果会出错\*/  for(p=0;p<rows;p++,imin+=n3){  for(q=0;q<cols;q++){  fstream[imin+jmin+q]=tmp\_matrix[p\*cols+q];  /\*printf("%d ",((int\*)fstream)[imin+jmin+q]);\*/  }  /\*printf("\n");\*/  }  }  }  /\*将结果打印到文件中\*/  for(i=0;i<n1;i++){  for(j=0;j<n3;j++){  fprintf(fhc,"%d ",fstream[i\*n3+j]);  }  fprintf(fhc,"\n");  }  }  int main(int argc,char\*\*argv){  int myid,numprocs;  int i,j;  MPI\_Status status;  int root=0;  int dim[3];  double elapsed\_time=0;  int max\_rows\_a,max\_cols\_a,max\_rows\_b,max\_cols\_b;  int buf\_A\_size,buf\_B\_size,buf\_C\_size;  FILE\* fhc;  /\*suppose A:n1\*n2 ,B:n2\*n3;n1,n2,n3 are read from input file\*/  int n1,n2,n3;  /\*buffer for matrix A,B,C will be shifted ,so they each have two buffer\*/  int \*A,\*B,\*C,\*buf\_A,\*buf\_B;  /\*on proc0,buffers to cache matrix files of A,B and C\*/  int \*fstream\_a=NULL,\*fstream\_b=NULL,\*fstream\_c=NULL;  MPI\_Init(&argc,&argv);/\*初始化\*/  MPI\_Comm\_rank(MPI\_COMM\_WORLD,&myid);/\*获取当前进程ID\*/  MPI\_Comm\_size(MPI\_COMM\_WORLD,&numprocs);/\*获取全部进程数量\*/  root=sqrt(numprocs);  if(numprocs!=root\*root){  /\*如果进程总数不是平方数，则结束程序\*/  printf("process number must be a squre!\n");  exit(-1);  }  /\*on proc0,preprocess the command line,read in file  for A,B and put their sizes in dim[]\*/  if(myid==0){  FILE \*file\_a,\*file\_b,\*file\_c;  int n1,n2,n3;  int i,j;  file\_a=fopen(argv[1],"r");/\*打开文件a，文件名从运行时给的参数中获得\*/  file\_b=fopen(argv[2],"r");/\*打开文件b，文件名从运行时给的参数中获得\*/  fscanf(file\_a,"%d %d",&n1,&n2);/\*从文件a中读取矩阵A的行数，列数\*/  fscanf(file\_b,"%d %d",&n2,&n3);/\*从文件b中读取矩阵B的行数，列数\*/    dim[0]=n1,dim[1]=n2,dim[2]=n3;  fstream\_a=(int\*)malloc(n1\*n2\*sizeof(int));/\*分配一块内存，用于将矩阵A读入内存\*/  fstream\_b=(int\*)malloc(n2\*n3\*sizeof(int));/\*分配一块内存，用于将矩阵B读入内存\*/  /\*printf("Yeah! I got n1=%d,n2=%d,n3=%d\n",n1,n2,n3);\*/  /\*读入矩阵A，保存在fstream\_a中\*/  for(i=0;i<n1;i++)  for(j=0;j<n2;j++)  fscanf(file\_a,"%d",&((int\*)fstream\_a)[i\*n2+j]);  /\*读入矩阵B，保存在fstream\_b中\*/  for(i=0;i<n2;i++)  for(j=0;j<n3;j++)  fscanf(file\_b,"%d",&((int\*)fstream\_b)[i\*n3+j]);  }  /\*将矩阵的行数，列数通过Bcast广播给所有进程\*/  MPI\_Bcast(dim,3,MPI\_INT,0,MPI\_COMM\_WORLD);  n1=dim[0],n2=dim[1],n3=dim[2];  /\*begin new version\*/  max\_rows\_a=(n1+root-1)/root;/\*子矩阵块A的行数\*/  max\_cols\_a=(n2+root-1)/root;/\*子矩阵块A的列数\*/  max\_rows\_b=max\_cols\_a; /\*子矩阵块B的行数\*/  max\_cols\_b=(n3+root-1)/root;/\*子矩阵块B的列数\*/  buf\_A\_size=max\_rows\_a\*max\_cols\_a;/\*子矩阵块A的大小\*/  buf\_B\_size=max\_rows\_b\*max\_cols\_b;/\*子矩阵块B的大小\*/  buf\_C\_size=max\_rows\_a\*max\_cols\_b;/\*子矩阵块C的大小\*/  /\*给A,，buf\_A,buf\_B,B,C分配内存空间，其中buf\_A,buf\_B用于通讯中的缓存\*/  A=(int\*)malloc(sizeof(int)\*buf\_A\_size);  buf\_A=(int\*)malloc(sizeof(int)\*buf\_A\_size);  B=(int\*)malloc(sizeof(int)\*buf\_B\_size);  buf\_B=(int\*)malloc(sizeof(int)\*buf\_B\_size);  C=(int\*)malloc(sizeof(int)\*buf\_C\_size);  if(A==NULL||buf\_A==NULL||B==NULL||buf\_B==NULL||C==NULL)  {  /\*如果内存申请失败，就退出\*/  printf("Memory allocation failed!\n");  exit(-1);  }  /\*proc 0 scatter A,B to other procs in a 2D block distribution fashion\*/  if(myid==0){  /\*printf("max\_rows\_a:%d\n",max\_rows\_a);  printf("max\_cols\_a:%d\n",max\_cols\_a);  printf("max\_rows\_b:%d\n",max\_rows\_b);  printf("max\_cols\_b:%d\n",max\_cols\_b);\*/  /\*进程0 将矩阵A，B划分成小块，分发给其他进程\*/  scatter\_matrix((int\*)fstream\_a,n1,n2,A,root,100);  /\*printf("I am debuging!\n");\*/  scatter\_matrix((int\*)fstream\_b,n2,n3,B,root,101);  /\*printf("I am finding fault!\n");\*/  }else{  /\*其他进程接收来自进程0 发送的矩阵A,B\*/  MPI\_Recv(A,max\_rows\_a\*max\_cols\_a,MPI\_INT,0,100,MPI\_COMM\_WORLD,&status);  MPI\_Recv(B,max\_rows\_b\*max\_cols\_b,MPI\_INT,0,101,MPI\_COMM\_WORLD,&status);  }  MPI\_Barrier(MPI\_COMM\_WORLD);/\*等待全部进程完成数据接收工作。\*/  /\*printf("I am proc %d\n",myid);  for(i=0;i<max\_rows\_a;i++){  printf("%d: ",myid);  for(j=0;j<max\_cols\_a;j++){  printf("%d ",A[i\*max\_cols\_a+j]);  }  printf("\n");  }  printf("I am proc %d\n",myid);  for(i=0;i<max\_rows\_b;i++){  printf("%d: ",myid);  for(j=0;j<max\_cols\_b;j++){  printf("%d ",B[i\*max\_cols\_b+j]);  }  printf("\n");  }\*/    /\*compute C=A\*B by Cannon algorithm\*/  /\*矩阵块必须定位对齐，先做预处理\*/  shuffle(A,buf\_A,buf\_A\_size,B,buf\_B,buf\_B\_size,root,myid);  elapsed\_time=MPI\_Wtime();  /\*包含cannon全部内容\*/  cannon(A,buf\_A,buf\_A\_size,B,buf\_B,buf\_B\_size,  C,buf\_C\_size,max\_rows\_a,max\_cols\_a,max\_cols\_b,root,myid);  MPI\_Barrier(MPI\_COMM\_WORLD);  elapsed\_time=MPI\_Wtime()-elapsed\_time;/\*统计cannon算法实际耗时\*/    MPI\_Barrier(MPI\_COMM\_WORLD);/\*等待所有进程完成cannon算法，将结果发送给进程0\*/  int fsize\_c=sizeof(int)\*n1\*n3;  if(myid==0){  /\*进程0创建文件写句柄，准备将计算结果写入文件中\*/  if(!(fhc=fopen(argv[3],"w"))){  printf("Cant't open file %s\n",argv[3]);  MPI\_Finalize();  }  fstream\_c=(int\*)malloc(fsize\_c);  /\*进程0 接收来自各个进程的结果矩阵，拼接成一个完整的结果，写入文件，持久化数据结果\*/  gather\_matrix(fstream\_c,n1,n3,C,root,fhc);  }    MPI\_Barrier(MPI\_COMM\_WORLD); /\*make sure proc 0 read all it needs\*/  if(myid==0){  int i,j;  printf("Cannon algorithm :multiply a %d\* %d with a %d\*%d, use %lf(s)\n",  n1,n2,n2,n3,elapsed\_time);  /\*printf("I have finished!\n");\*/  fclose(fhc);/\*关闭文件读写句柄\*/  /\*释放申请的内存空间\*/  free(fstream\_a);  free(fstream\_b);  free(fstream\_c);  }  /\*释放申请的内存空间\*/  free(A);free(buf\_A);  free(B);free(buf\_B);  free(C);  MPI\_Finalize();  return 0;  } |

### 辅助程序

* 生成矩阵程序

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <time.h>  #define n1 100  #define n2 150  #define n3 250  #define max 1000  int main(){  int i,j,k;  FILE \*fa,\*fb,\*fCheck;  int A[n1][n2],B[n2][n3],CCheck[n1][n3];  clock\_t start, finish;  double duration;  fa=fopen("a.txt","w");  fb=fopen("b.txt","w");  fCheck=fopen("c\_check.txt","w");  if(fa==NULL ||fb==NULL ||fCheck==NULL){  printf("Can't open file!\n");  exit(-1);  }  srand((unsigned)time(NULL));  fprintf(fa,"%d %d\n",n1,n2);  for(i=0;i<n1;i++){  for(j=0;j<n2;j++){  A[i][j]=rand()%max;  fprintf(fa," %d",A[i][j]);  }  fprintf(fa,"\n");  }  fprintf(fb,"%d %d\n",n2,n3);  for(i=0;i<n2;i++){  for(j=0;j<n3;j++){  B[i][j]=rand()%max;  fprintf(fb," %d",B[i][j]);  }  fprintf(fb,"\n");  }  /\*generate matrix C using simple method\*/  start=clock();  memset(CCheck,0,sizeof(CCheck));  for(i=0;i<n1;i++){  for(j=0;j<n3;j++){  for(k=0;k<n2;k++){  CCheck[i][j]+=A[i][k]\*B[k][j];  }  fprintf(fCheck," %d",CCheck[i][j]);  }  fprintf(fCheck,"\n");  }  finish=clock();  duration = (double)(finish - start) / CLOCKS\_PER\_SEC;  printf("simple algorithm :multiply a %d\* %d with a %d\*%d, use %lf(s)\n",  n1,n2,n2,n3,duration);  fclose(fa);  fclose(fb);  fclose(fCheck);  } |

* 打印矩阵程序

|  |
| --- |
| void gather\_matrix(int \*fstream,int n1,int n3,int\*C,int root,FILE\*fhc){  MPI\_Status status;  int rows=(n1+root-1)/root;  int cols=(n3+root-1)/root;  int\* tmp\_matrix=(int\*)malloc(rows\*cols\*sizeof(int));  int i,j;  for(i=0;i<root;i++){  for(j=0;j<root;j++){  int p,q;  int imin=i\*rows\*n3;  int jmin=j\*cols;  memset(tmp\_matrix,0,sizeof(tmp\_matrix));  /\*接收来自各个进程的数据\*/  MPI\_Recv(tmp\_matrix,rows\*cols,MPI\_INT,i\*root+j,104,MPI\_COMM\_WORLD,&status);  /\*printf("I am passed proc:%d \n",i\*root+j);\*/  /\*将接收的矩阵tmp拼接到矩阵C中去，需要按照合理顺序拼接，否则结果会出错\*/  for(p=0;p<rows;p++,imin+=n3){  for(q=0;q<cols;q++){  fstream[imin+jmin+q]=tmp\_matrix[p\*cols+q];  /\*printf("%d ",((int\*)fstream)[imin+jmin+q]);\*/  }  /\*printf("\n");\*/  }  }  }  /\*将结果打印到文件中\*/  for(i=0;i<n1;i++){  for(j=0;j<n3;j++){  fprintf(fhc,"%d ",fstream[i\*n3+j]);  }  fprintf(fhc,"\n");  }  } |

* 比较矩阵程序



|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <math.h>  #define n1 100  #define n3 250  int main(){  int i,j,k;  int flag=0;  FILE \*fc,\*fCheck;  int C[n1][n3],CCheck[n1][n3];  fc=fopen("c.txt","r");  fCheck=fopen("c\_check.txt","r");  if(fc==NULL||fCheck==NULL){  printf("Can't open file!\n");  exit(-1);  }  for(i=0;i<n1;i++){  for(j=0;j<n3;j++){  fscanf(fc,"%d ",&C[i][j]);  fscanf(fCheck,"%d ",&CCheck[i][j]);  if(C[i][j]!=CCheck[i][j]){  printf("%d %d\n",C[i][j],CCheck[i][j]);  flag=1;  }  }  }  if(flag==0){  printf("Designed Cannon Algorithm is right!\n");  }else{  printf("Designed Cannon Algorithm is wrong!\n");  }  fclose(fc);  fclose(fCheck);  } |

* **附加程序，用pthread实现的矩阵乘法：**

|  |
| --- |
| #include"mpi.h"  #include"pthread.h"  #include<stdio.h>  #include<stdlib.h>  #include <sys/sysinfo.h>  #define M 100  #define N 150  #define P 250  /\*创建struct 便于管理信息\*/  struct threadArg{  int tid;  int (\*B)[P]; /\*二维矩阵，用指针数组保存每一行的第一个数的地址\*/  int \*A\_row; /\*矩阵A的一行\*/  int \*C\_row; /\*结果矩阵C的一行\*/  int numthread;  };  void \*worker(void\*arg){  int i,j;  struct threadArg \*myarg=(struct threadArg\*)arg;  /\*平均分配D的所有列\*/  for(i=myarg->tid;i<P;i+=myarg->numthread){  myarg->C\_row[i]=0.0;  for(j=0;j<N;j++){  /\*B的一列与A的一行相乘，存入C对应位置中\*/  myarg->C\_row[i]+=myarg->A\_row[j]\*myarg->B[j][i];  }  /\*printf("C[%d]:%f\n",i,myarg->C\_row[i]);\*/  }  return NULL;  }  int main(int argc,char\*argv[]){  double start\_time=0,end\_time=0;  int i,j;  int myid,numprocs;  MPI\_Status status;  int sender;  int A[M][N],B[N][P],C[M][P];  MPI\_Init(&argc,&argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD,&myid);  MPI\_Comm\_size(MPI\_COMM\_WORLD,&numprocs);  if(myid==0){  int n1,n2,n3;    FILE \*file\_a,\*file\_b;  file\_a=fopen("a.txt","r");  file\_b=fopen("b.txt","r");  fscanf(file\_a,"%d %d",&n1,&n2);  fscanf(file\_b,"%d %d",&n2,&n3);  for(int i=0;i<n1;i++)  for(int j=0;j<n2;j++)  fscanf(file\_a,"%d",&A[i][j]);    for(int i=0;i<n2;i++)  for(int j=0;j<n3;j++)  fscanf(file\_b,"%d",&B[i][j]);  fclose(file\_a);  fclose(file\_b);  }  /\*将矩阵B广播给所有其他计算节点\*/  MPI\_Bcast(B[0],N\*P,MPI\_INT,0,MPI\_COMM\_WORLD);  MPI\_Barrier(MPI\_COMM\_WORLD);  if(myid==0){ /\*分配任务，回收结果\*/  int i,j,numsend;  /\*比较进程数量和A的行维数大小，选取二者之中的较小者\*/    j=((numprocs-1)<M?(numprocs-1):M);  for(i=1;i<=j;i++){  /\*将A的每一行分发给从进程\*/  MPI\_Send(A[i-1],N,MPI\_INT,i,99,MPI\_COMM\_WORLD);  }  numsend=j;  for(i=1;i<=M;i++){  sender=(i-1)%(numprocs-1)+1;  if(numsend<M){/\*若数组A还没有发送完数据，则继续发送\*/  MPI\_Send(A[i-1],N,MPI\_INT,sender,99,MPI\_COMM\_WORLD);  numsend++;  }else{  /\*终止sender进程的运行\*/  MPI\_Send(&j,0,MPI\_INT,sender,0,MPI\_COMM\_WORLD);  }  /\*回收来自从进程sender的计算结果\*/  MPI\_Recv(C[i-1],P,MPI\_INT,sender,100,MPI\_COMM\_WORLD,&status);      }    /\*打印结果\*/  FILE \*file\_C=fopen("d.txt","w");  for(i=0;i<M;i++){  for(j=0;j<P;j++){  fprintf(file\_C,"%d ",C[i][j]);  }  fprintf(file\_C,"\n");  }  fclose(file\_C);    }else{  int i,j;  int numthread=10;/\*获取当前计算节点的CPU数量\*/    /\*printf("cpu number:%d.\n",numthread);get cpu number\*/  /\*创建一组线程，保证每个CPU上运行一个线程\*/  pthread\_t \*tids=(pthread\_t\*)malloc(numthread\*sizeof(pthread\_t));  /\*A矩阵的一行\*/  int \*A\_row=(int\*)malloc(N\*sizeof(int));  /\*C矩阵的一行\*/  int \*C\_row=(int\*)malloc(P\*sizeof(int));  /\*用于统一管理每个线程的A，B，C矩阵信息\*/  struct threadArg \*targs=(struct threadArg\*)malloc(numthread\*sizeof(struct threadArg));  for(i=0;i<numthread;i++){  targs[i].tid=i; /\*编号\*/  targs[i].B=B;  targs[i].A\_row=A\_row;  targs[i].C\_row=C\_row;  targs[i].numthread=numthread;  }  start\_time=MPI\_Wtime();    while(1){  int i,j;  /\*接收主进程发来的一行,MPI\_ANY\_TAG表示任意标记的数据都要接收\*/  MPI\_Recv(A\_row,N,MPI\_INT,0,MPI\_ANY\_TAG,MPI\_COMM\_WORLD,&status);  /\*for(i=0;i<N;i++)  printf("A\_row[%d]: %.0f \n",i,A\_row[i]);\*/  /\*检查接受到的tag标记，若为0，则退出\*/  if(status.MPI\_TAG==0) break;  for(i=0;i<numthread;i++){  /\*创建一组线程  第一个参数为指向线程ID。  第二个参数用来设置线程属性。  第三个参数是线程运行函数的起始地址。  最后一个参数是运行函数的参数\*/  pthread\_create(&tids[i],NULL,worker,&targs[i]);  }  for(i=0;i<numthread;i++){  /\*以阻塞的方式等待thread指定的线程结束,等待全部线程计算完成\*/  pthread\_join(tids[i],NULL);  }  /\*从进程返回计算结果给0进程\*/  MPI\_Send(C\_row,P,MPI\_INT,0,100,MPI\_COMM\_WORLD);  end\_time=MPI\_Wtime()-start\_time;  printf("pthread algorithm:multiply a %d\* %d with a %d\*%d, use %f(s)\n",  M,N,N,P,end\_time);  /\*printf("I put an flag\n");  for(j=0;j<P;j++){  printf("%d ",(int)C\_row[j]);  }  printf("I put an flag again\n");\*/  }  }  MPI\_Finalize();  return 0;  } |

### 实验结论

采用MPI实现的cannon算法，纯矩阵计算时间很短，但是通信花费大量时间，而且是不可避免的，时间延时是不可忍受的，或许存在更优化的通信方法，本实验采用的是基于阻塞机制的MPI\_Sendrecv，或许还可以采用其他的函数，例如MPI\_Cart\_Shift，性能可能会更好。

本次编程中，一律采用local variable，而不是global variable，主要是老师在课上用几组例子，向我们解说了global varible不容易编译优化，产生的二进制文件效率远远低于本地变量，调用函数时传递参数也使用指针传递地址，这样减少堆栈在runtime时的消耗。最后，为了简便起见，本次实验均采用Int型数据，而不是老师在参考模板里给出的double型数据，同时为了撰写函数方便，我还对老师给的模板做了一些修改，但是大致相同，还望见谅。

本次实验，花费了大量时间在验证数据据算的正确性，每次都采用printf的方法，逐一检查每个节点的计算结果是否正确，数据通信是否正确，最郁闷的是在不知道计算cannon之前 需要矩阵定位对齐，只怪自己ppt没有参透，仔细阅读，花费大量时间检查每一组数据，用matlab辅助计算中间结果，最后才意识到矩阵下标需要定位对齐，啊多么痛的领悟！

### 参考文献：

[1] 陈鹏，樊小超. 几种矩阵并行算法的对比分析. 新疆师范大学学报. 2012，31（3）：5-10

[2] 杨尚琴，罗省贤. 基于多层次并行模型的矩阵乘算法的实现与分析. 2008年全国高性能计算年会