程序性能优化理论与方法

韩林　高伟　著

先进编译实验室
Advanced Compiler

目录

上篇

目录

下篇

第十一章 MPI程序优化

11.1.1 MPI是什么

- MPI是Message Passing Interface的缩写，是一组用于编写并行程序的多节点数据通信标准。
- 通常基于单核处理器上编写的程序无法直接利用多核处理器，因此可以手动将串行程序改写为并行程序以使程序充分利用多核处理器更快的运行程序。
- MPI作为一种服务于进程间通信的消息传递编程模型，可运行在不同的机器或平台上，具有很好的可移植性。
- 可以将MPI理解为一种协议或接口，而OpenMPI及MPICH是这一接口的常用实现。

- MPI的库函数有很多，大体上分为以下五类：
  - 基本函数
  - 阻塞型点对点传递函数
  - 非阻塞型点对点传递函数
  - 组消息传递函数
  - MPI自定义数据类型函数

- **基本函数**主要用于MPI环境的初始化工作、资源释放工作、获取MPI程序的信息和执行时间等，常用基本函数及其参数如下。
  - int MPI_Init(int *argc, char **argv[]); //初始化MPI环境
  - int MPI_Finalize(void); //终止MPI执行环境
  - int MPI_Comm_rank(MPI_Comm comm, int *rank); //获得当前进程标识
  - int MPI_Comm_size(MPI_Comm comm, int *size); //获取通信域包含的进程总数
  - int MPI_Get_processor_name(char *name, int *resultlen); //获得本进程的机器名
  - double MPI_Wtime(void); //以秒为单位返回从过去某点开始的执行时间

- **阻塞型点对点传递函数**需要等待指定操作的实际完成，或至少所涉及的数据已被MPI系统安全地备份后才返回。常用阻塞型点对点传递函数及其参数如下。

  - int MPI_Send(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm); //消息发送函数：将发送缓冲区buf中count个datatype数据类型的数据发送到标识号为dest的目的进程，本次发送的消息标识是tag

  - int MPI_Recv(void *buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Status *status); //消息接收函数：将从标识号为source的目的进程接收count个datatype数据类型的数据到缓冲//区buf中，tag与消息发送时指定的tag号一致

- **非阻塞型点对点传递函数**的调用总是立即返回，而实际操作则由MPI系统在后台进行，使用非阻塞会带来性能提升，但是提高了编写程序的难度。常用非阻塞型点对点传递函数及其参数如下。

  - int MPI_Isend(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request); //比阻塞操作只多一个参数MPI_Request *request，随后必须调用其它函数，如函数MPI_Wait 和MPI_Test，来等待操作完成或查询操作的完成情况
  - int MPI_Irecv(void *buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Request *request); //比阻塞操作只多一个参数MPI_Request *request，随后必须调用其它函数，如函数MPI_Wait 和MPI_Test，来等待操作完成或查询操作的完成情况
  - int MPI_Wait(MPI_Request *request,MPI_Status *status); //等待MPI发送或接收结束然后返回

- **非阻塞型点对点传递函数**的调用总是立即返回，而实际操作则由MPI系统在后台进行，使用非阻塞会带来性能提升，但是提高了编写程序的难度。常用非阻塞型点对点传递函数及其参数如下。

  - int MPI_Test(MPI_Request *request,int *flag,MPI_Status *status); //若flag为true，则如同执行了MPI_Wait调用；若flag为false，则如同执行了一个空操作
  - int MPI_Probe(int source,int tag,MPI_Comm comm,MPI_Status *status); //阻塞式检查
  - int MPI_Iprobe(int source,int tag,MPI_Comm comm,int *flag,MPI_Status *status); //非阻塞式检查

- **组消息传递相关函数**也可称为集合通信函数，集合通信调用可以和点对点通信共用一个通信域，MPI保证由集合通信调用产生的消息不会与点对点通信调用产生的消息相混淆。常用集合通信函数及其参数如下。
  - int MPI_Barrier(MPI_Comm comm); //障碍同步，阻塞通信体中所有进程，直到所有的进程组成员都调用了它。仅当进程组所有的成员都进入了这个调用后，各个进程中这个调用才可以返回
  - int MPI_Bcast(void *buf,int count,MPI_Datatype datatype,int root,MPI_Comm comm); //是从一个序号为root的进程将一条消息广播发送到进程组内的所有进程
  - int MPI_Gather(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype,int root,MPI_Comm comm); //每个进程将其发送缓冲区中的内容发送到进程，进程根据发送这些数据的进程序列号将它们依次存放到自己的消息缓冲区中

- **组消息传递相关函数**也可称为集合通信函数，集合通信调用可以和点对点通信共用一个通信域，MPI保证由集合通信调用产生的消息不会与点对点通信调用产生的消息相混淆。常用集合通信函数及其参数如下。

  - int MPI_Scatter(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype,int root,MPI_Comm comm); //从根进程部分地散播缓冲区中的值到进程组

  - int MPI_Reduce(void *sendbuf,void *recvbuf,int count,MPI_Datatype recvtype,MPI_OP op,int root,MPI_Comm comm); //将组内每个进程输入缓冲区中的数据按op操作组合起来，并将其结果返回到序号为root的进//程的输出缓冲区中

- **MPI自定义数据类型函数**可以有效减少消息传递次数，增大通信力度，同时可以避免或减少消息传递时数据在内存中的拷贝。常用自定义数据类型函数及其参数如下。

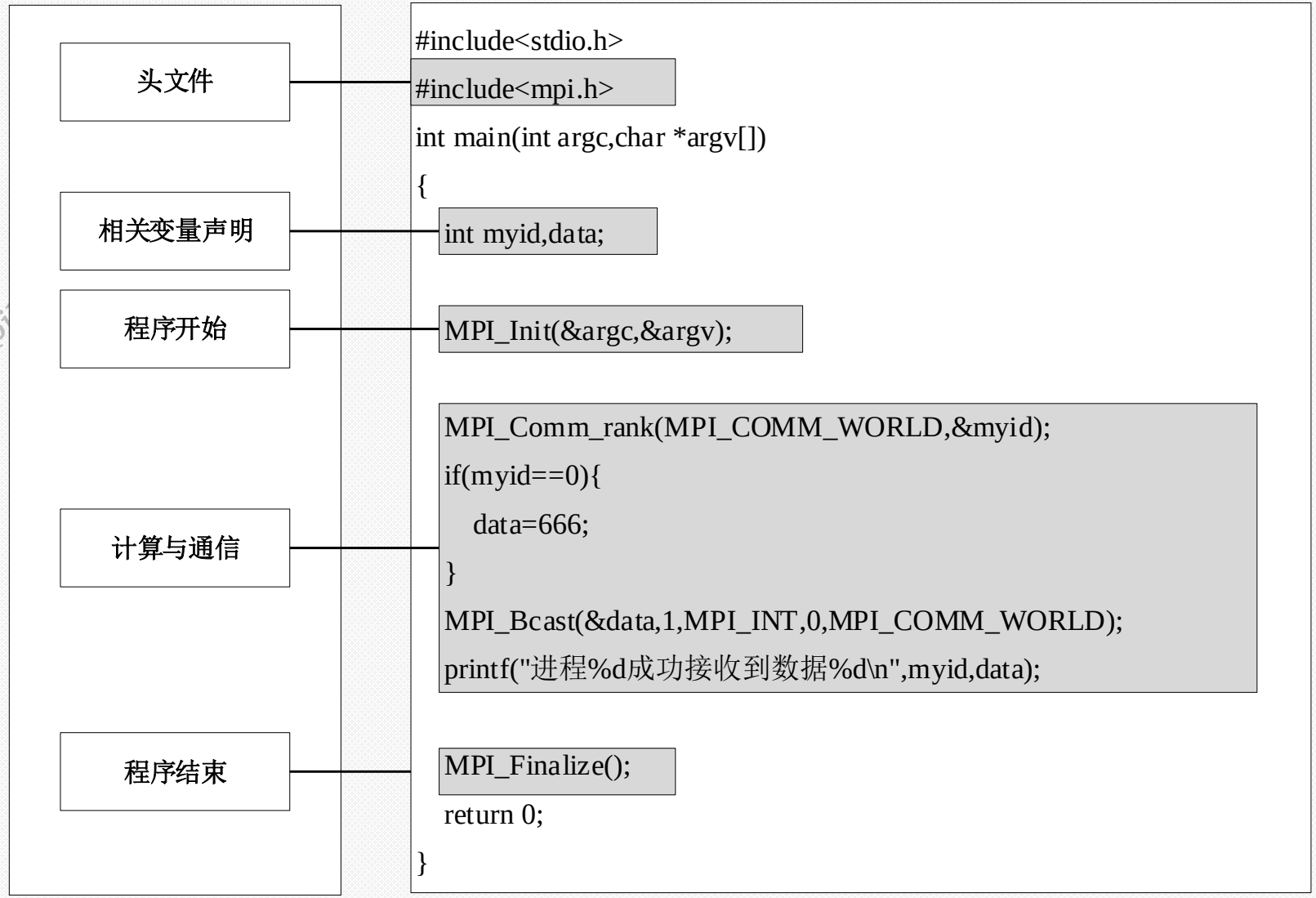
  - int MPI_Type_contiguous(int count,MPI_Datatype oldtype,MPI_Datatype *newtype); //连续数据类型生成
  - int MPI_Type_vector(int count,int blocklength,int stride,MPI_Datatype oldtype,MPI_Datatype *newtype); //向量数据类型的生成
  - int MPI_Type_indexed(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype); //索引数据类型的生成

- **MPI自定义数据类型函数**可以有效减少消息传递次数，增大通信力度，同时可以避免或减少消息传递时数据在内存中的拷贝。常用自定义数据类型函数及其参数如下。

  - int MPI_Type_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype array_of_types, MPI_Datatype *newtype); //结构数据类型的生成
  - int MPI_Type_commit(MPI_Datatype *datatype); //数据类型的注册
  - int MPI_Type_free(MPI_Datatype *datatype); //数据类型的释放

- MPI程序的基本框架

主要由头文件、相关变量声明、程序开始、计算与通信和程序结束五部分组成。



11.1.3 MPI程序编写

- 入门的MPI程序

此代码的目的是使用0号进程发送一个整型数据，1号进程接收这个数据。

```
1#include<stdio.h>
2#include<mpi.h>
3int main(int argc,char *argv[])
4{
5    int world_rank,world_size,send,recv;
6    MPI_Status status;
7    MPI_Init(&argc,&argv);
8    MPI_Comm_rank(MPI_COMM_WORLD,&world_rank);
9    MPI_Comm_size(MPI_COMM_WORLD,&world_size);
10   if(world_rank==0){
11       send=666;
12       MPI_Send(&send,1,MPI_INT,1,0,MPI_COMM_WORLD);
13       printf("共%d个进程，其中进程%d成功发送数据
%d\n",world_size,world_rank,send);
14   }
15   if(world_rank==1){
16       MPI_Recv(&recv,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
17       printf("共%d个进程，其中进程%d成功接收数据%d\n ", world_size ,
world_rank,recv);
18   }
19   MPI_Finalize();
20   return 0;
21}
```

- 将该程序命名为ex.c后，分别使用命令mpicc -o ex ex.c和mpirun -np 2 ex对其进行编译运行，得到如下结果：

```
共2个进程，其中进程0成功发送数据666
共2个进程，其中进程1成功接收数据666
```

- 串行矩阵乘法

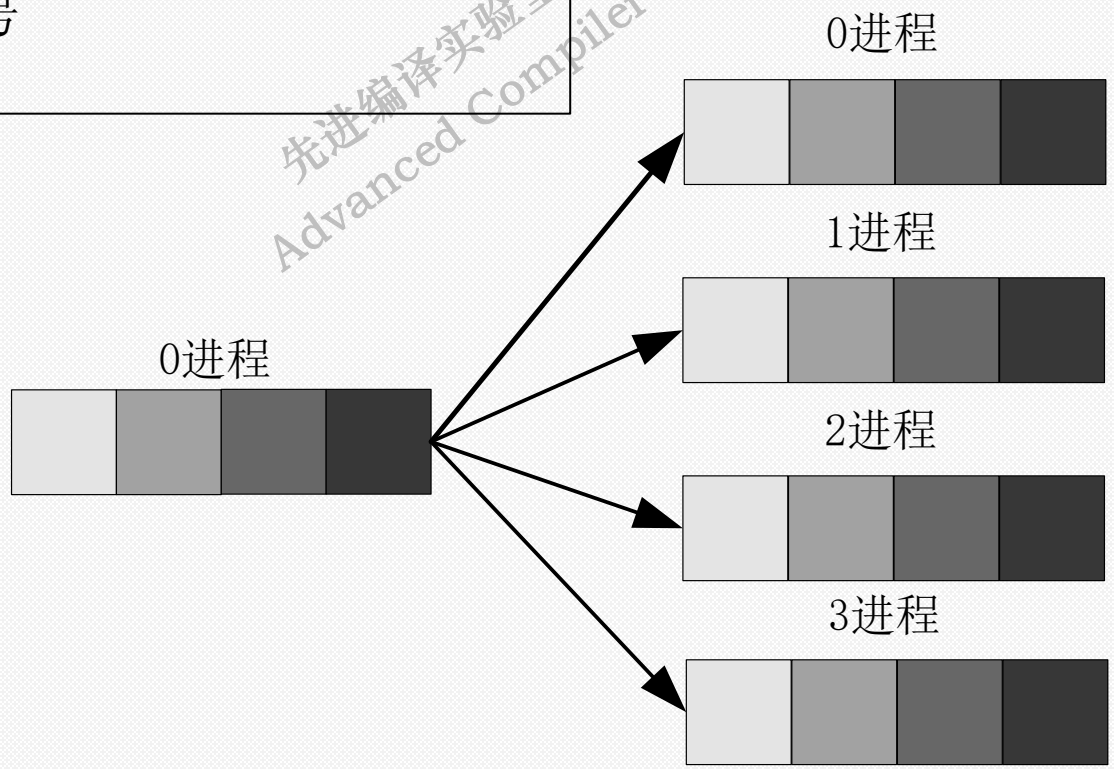
```
1#include<stdio.h>
2#include<mpi.h>
3#include<time.h>
4#include"mympi.h"
5#define DIMS 1000
6int main(int argc,char *argv[]){
7    data_t *A,*B,*C,i;
8    double start_time,end_time;
9    A=(data_t*)malloc(sizeof(data_t)*DIMS*DIMS);
10    B=(data_t*)malloc(sizeof(data_t)*DIMS*DIMS);
11    C=(data_t*)malloc(sizeof(data_t)*DIMS*DIMS);
12    //初始化A和B矩阵
13    //初始化函数的参数意味随机生成0/1矩阵，传
入1代表生成0矩阵
14    Init_Matrix(A,DIMS*DIMS,2);
15    Init_Matrix(B,DIMS*DIMS,2);
16    Init_Matrix(C,DIMS*DIMS,1);
17    start_time=(double)clock();
18    //矩阵A与B相乘，结果存于矩阵C
19    Mul_Matrix(A,B,C,DIMS,DIMS,DIMS);
20    end_time=(double)clock();
21    printf("进程的执行时为:%.2lf\n",(end_time-
start_time)/1e3);
22    free(A);
23    free(B);
24    free(C);
25    return 0;
26}
```

11.1.4 MPI版矩阵乘

- MPI_Bcast：函数可以使用一个进程将消息广播发送给通信域中所有其它进程，同时包括它本身在内

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

buffer	所要广播/接收的数据的起始地址
count	所要广播/接收的数据的个数
datatype	数据类型
root	要发送广播消息的进程的进程号
comm	通信域



- 基础并行矩阵乘法

```
1#include<stdio.h>
2#include<mpi.h>
3#include"mympi.h"
4#define DIMS 1000
5int main(int argc, char *argv[]){
6     data_t *A,*B,*C;
7     int world_rank, world_size,lens,i;
8     double start_time, end_time;
9     MPI_Init(&argc, &argv);
10    MPI_Comm_rank(MPI_COMM_WORLD,
&world_rank);
11    MPI_Comm_size(MPI_COMM_WORLD,
&world_size);
12    if(DIMS%world_size!=0){
13        printf("总进程数world_size应整除矩阵维数
DIMS!!!\n");
14        MPI_Finalize();
15        return 0;
16    }
17    //为所有进程创建A、B、C的空间并初始化C
18    //在0进程初始化A、B进程
19    A=malloc(sizeof(data_t)*DIMS*DIMS);
20    B=malloc(sizeof(data_t)*DIMS*DIMS);
21    C=malloc(sizeof(data_t)*DIMS*DIMS);
22    Init_Matrix(C,DIMS*DIMS,1);
23    if(world_rank==0){
24        Init_Matrix(A,DIMS*DIMS,2);
25        Init_Matrix(B,DIMS*DIMS,2);
26    }
27    start_time=MPI_Wtime();
```

- 基础并行矩阵乘法

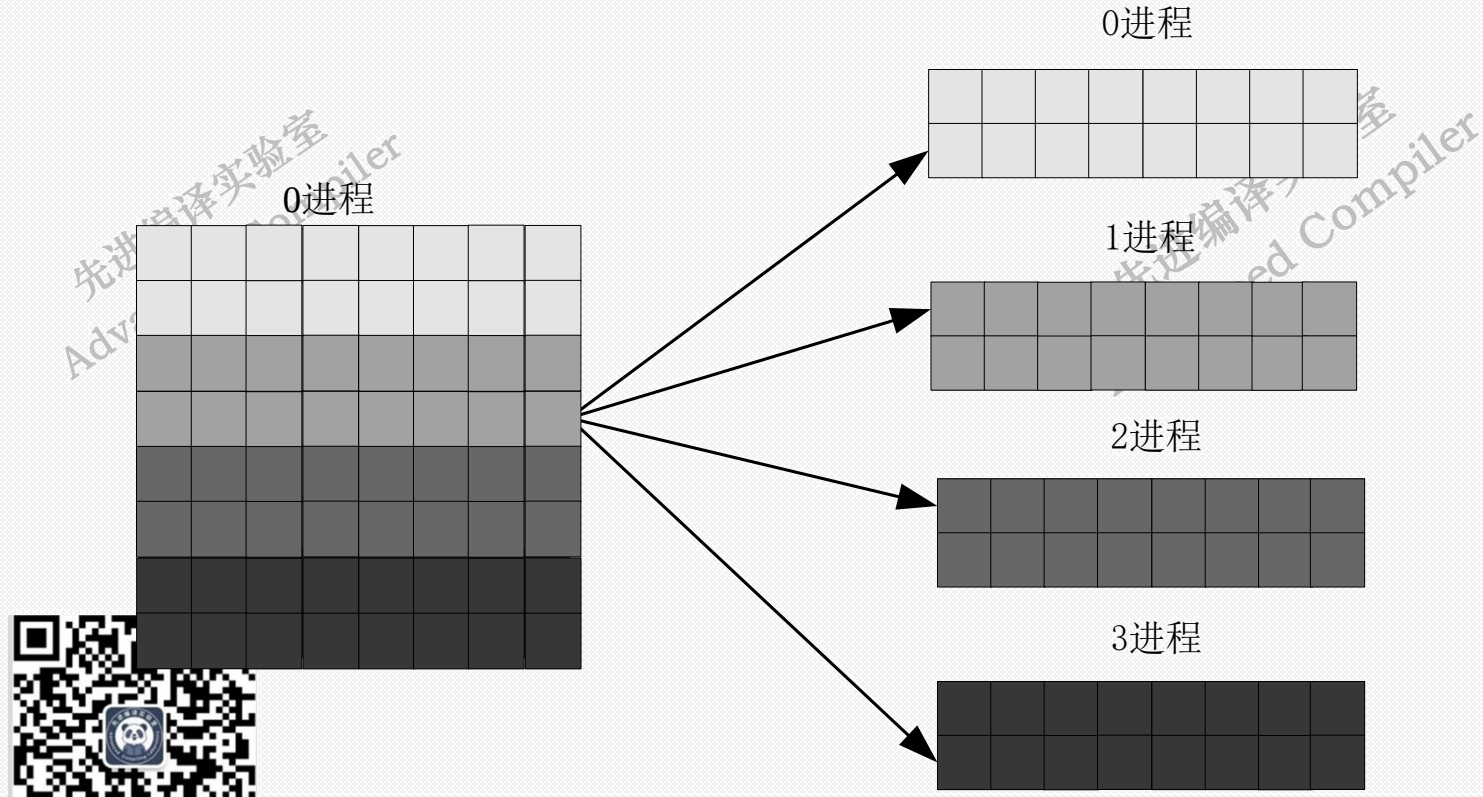
```
28   //广播矩阵A、B到其它所有进程
29
MPI_Bcast(A,DIMS*DIMS,MPI_FLOAT,0,MPI_COM
M_WORLD);
30
MPI_Bcast(B,DIMS*DIMS,MPI_FLOAT,0,MPI_COM
M_WORLD);
31   //每个矩阵要处理的A的行数
32   lens = DIMS/world_size;
33   //将A对应行与B相乘，结果存于C对应行
34
Mul_Matrix(A+lens*DIMS*world_rank,B,C+lens*DIM
S*world_rank,lens,DIMS,DIMS);
35   //各进程将自己的一块C广播到其它进程，组合
成完整的C
36   for(i=0;i<world_size;i++){
37
MPI_Bcast(C+i*lens*DIMS,lens*DIMS,MPI_FLOAT,i,
MPI_COMM_WORLD);
38   }
39   end_time=MPI_Wtime();
40   printf("进程%d的运行时间
为:%lf\n",world_rank,(end_time-start_time));
41   free(A);
42   free(B);
43   free(C);
44   MPI_Finalize();
45   return 0;
46}
```

11.2 数据划分优化

- 数据划分通常对规模较大的数据进行划分，将分解后的数据块聚集或映射到多个处理器上，实现在多个进程上同时执行以加快程序运行速度。
- 在保证结果正确的前提下要使数据划分后程序的性能较好就需要使负载尽可能保持均衡。
- 以矩阵乘算法为例，基础的并行算法是使用0号进程将生成的矩阵完整的广播到各个进程，这样可确保结果的正确性但效率不高，所以可以采用数据划分方法让不同的进程去执行矩阵A某个分块和矩阵B某个分块的乘法计算得到结果C的不同部分，最后将C的不同部分聚合以得到完整的结果C，这里常用的矩阵划分方法有三种，分别为按行、按列以及棋盘式划分方法。

11.2.1 按行分解

- 由于在计算矩阵C的第i行时只需要用到矩阵A的第i行以及完整的矩阵B，因此每个进程上存储A中多余的行会增加很多不必要的通信，可以使用按行划分解的方式，每个进程负责处理矩阵A的若干行与矩阵B相乘，得到结果C中的若干行，最后合并结果。



11.2.1 按行分解

- 采用按行分解的矩阵乘并行算法的实现流程如下：

① 由0进程生成矩阵A和B。

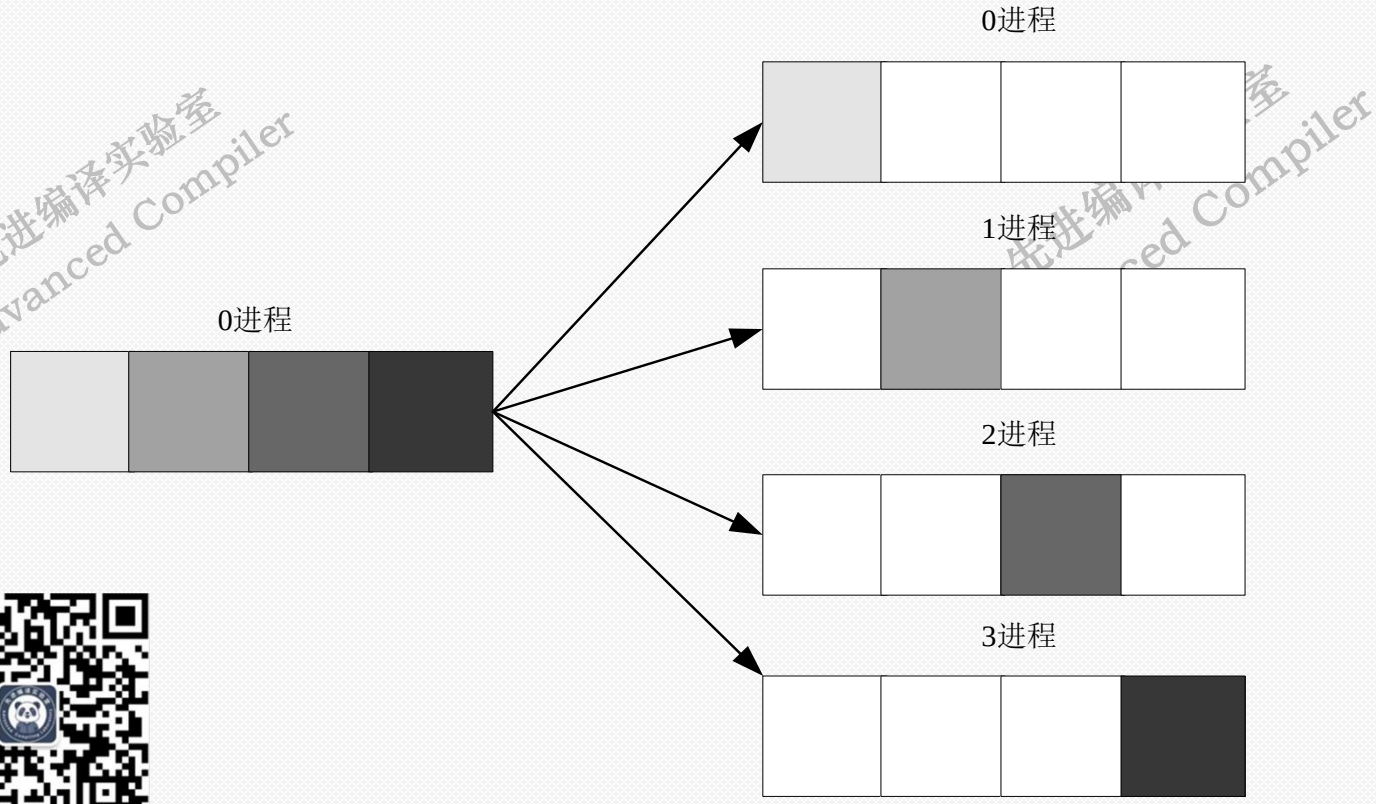
② 0号进程将矩阵B发送到所有进程。

③ 0号进程依据总进程数与矩阵维数的关系划分任务，分别将A的对应若干行发送给不同的进程。

④ 各个进程完成矩阵C部分行的计算。

⑤ 将结果聚合到0号进程。

- 与MPI_Bcast类似，MPI_Scatter也是一个一对多的通信函数，但是与MPI_Bcast的不同之处在于，MPI_Scatter的0号进程向每个进程发送的数据可以是不同的，0号进程将连续的4个不同的数据按照进程号大小的顺序依次发送给通信域中的所有进程。

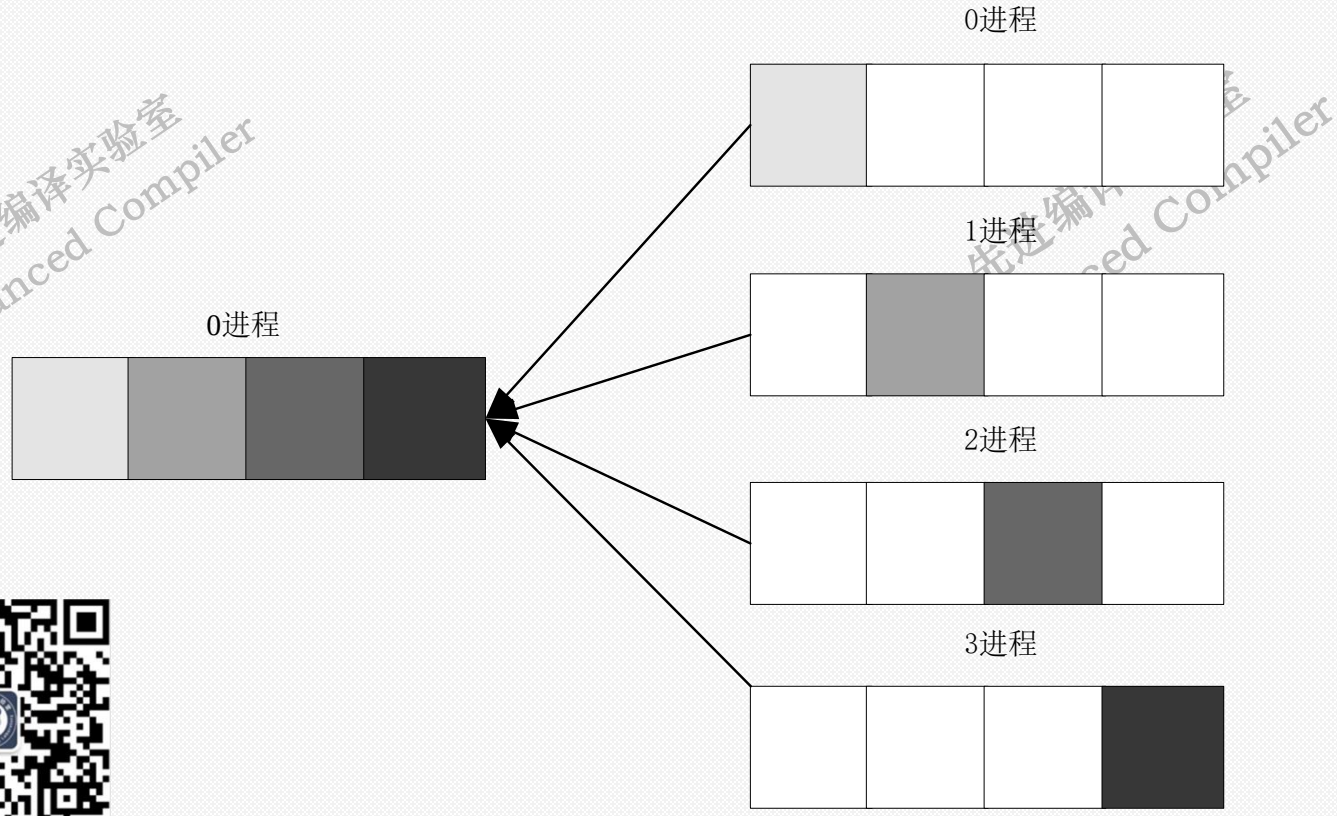


- 此函数的原型如下，其所有参数对根进程来说都是有意义的，而对于子进程来说只需考虑recvbuf、recvcount、recvtype、root和comm，参数root和comm在所有参与计算的进程中都必须是一致的。

int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

sendbuf	发送消息缓冲区的起始地址
sendcount	发送给的数据个数
sendtype	发送的数据类型
recvbuf	接收缓冲区的起始地址
recvcount	待接收的元素个数
recvtype	接收类型
root	数据发送进程的序列号
comm	通信域

- 和MPI_Scatter相反，MPI_Gather是一个典型的用于多对一通信的函数。每个进程都会将一个相同大小的数据块发送给根进程，这些数据到达根进程后，会按照进程号的大小排序存储到接收缓冲区，因此根进程需要开辟出一块足以容纳所有进程发送数据的空间。



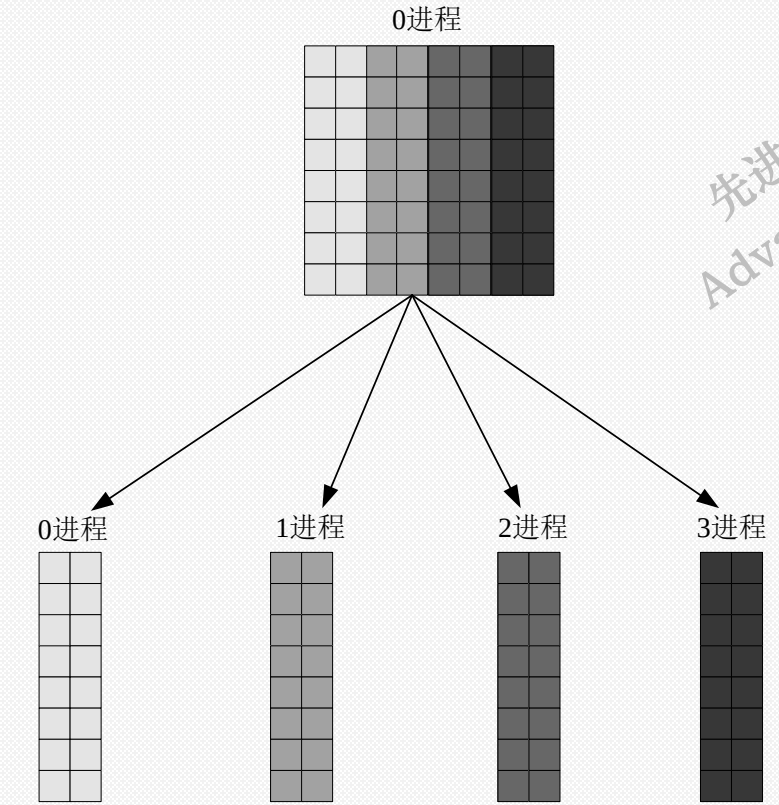
- 此函数的原型如下，参数recvcount是指根进程接收每个进程发来的数据大小，此调用中的所有参数对根进程来说都是有意义的，而对于其它子进程只需考虑sendbuf、sendcount、sendtype、root和comm，其它的参数虽然没有意义但是却不能省略，root和comm在所有进程中都必须是一致的。

int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype,int root, MPI_Comm comm)

sendbuf	发送缓冲区的起始地址
sendcount	每个进程发送的数据个数
sendtype	发送的数据类型
recvbuf	接收缓冲区的起始地址
recvcount	从每个进程接收到的数据个数
recvtype	接收的数据类型
root	接收进程的进程号
comm	通信域

11.2.2 按列分解

- 按行分解矩阵乘原理是通过降低矩阵A在每个进程中的存储空间降低了通信消耗以及内存的使用，但没有对B矩阵进行处理，使用按列分解方法可以同时降低B矩阵内存开销，即将A矩阵和B矩阵按照行分解的方法进行划分，每个进程负责处理矩阵A的若干列与矩阵B的若干行相乘，以得到结果C中的一部分，最后将各个进程的计算结果进行归约操作得到完整的结果矩阵C。



11.2.2 按列分解

- 采用按列分解的矩阵乘并行算法的实现流程如下：

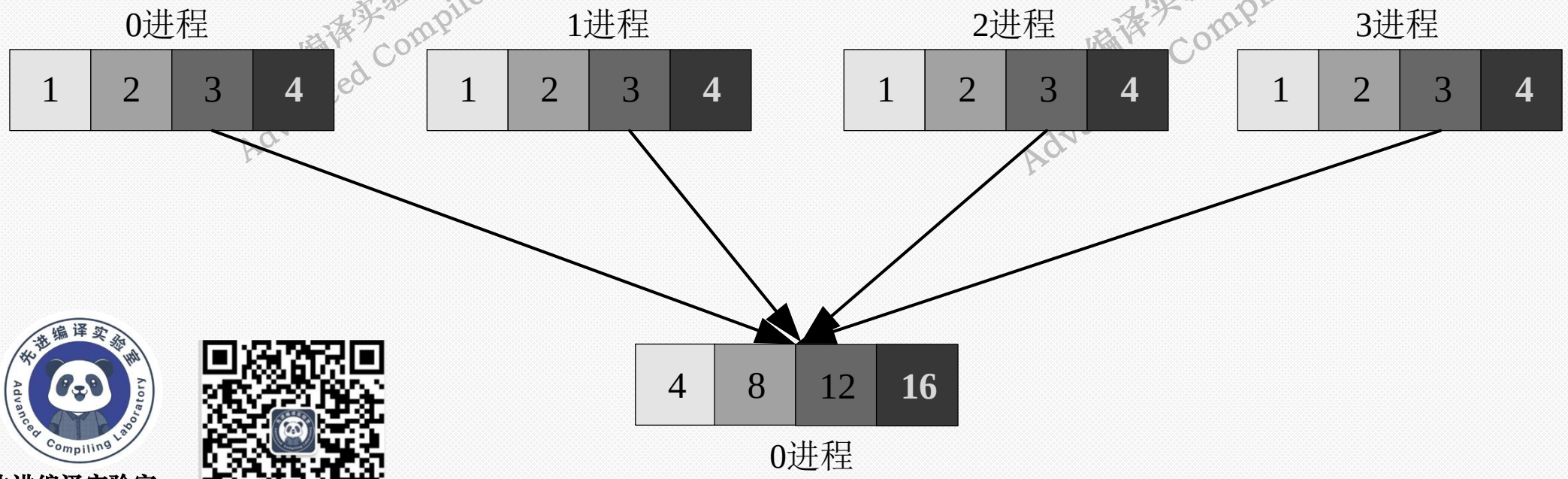
① 由0进程生成矩阵A和B。

② 0号进程依据总进程数与矩阵维数的关系划分任务，分别将A的对应若干列和B对应的若干行发送给不同的进程。

③ 各个进程完成部分矩阵C的计算。

④ 将各个进程的矩阵C汇聚到0号进程，并对各个C矩阵的对应位置进行归约求和操作。

- MPI_Reduce将组内每个进程输入缓冲区中的数据在相应的位置按给定的操作进行运算，并将其结果返回到0号进程。当进程数为4时使用MPI_Reduce进行求和归约时，会将所有进程对应位置的数据进行求和，最后将结果汇聚到0号进程。



- MPI_Reduce其函数原型如下，对于所有进程来说，都会有count个以sendbuf为起始地址的datatype类型的数据，不同进程中的所有对应位置的数据都互相进行归约操作，待操作完成后将结果存储于0号进程中的recvbuf位置处。

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,MPI_Datatype datatype, MPI_Op op, int root,
MPI_Comm comm)
```

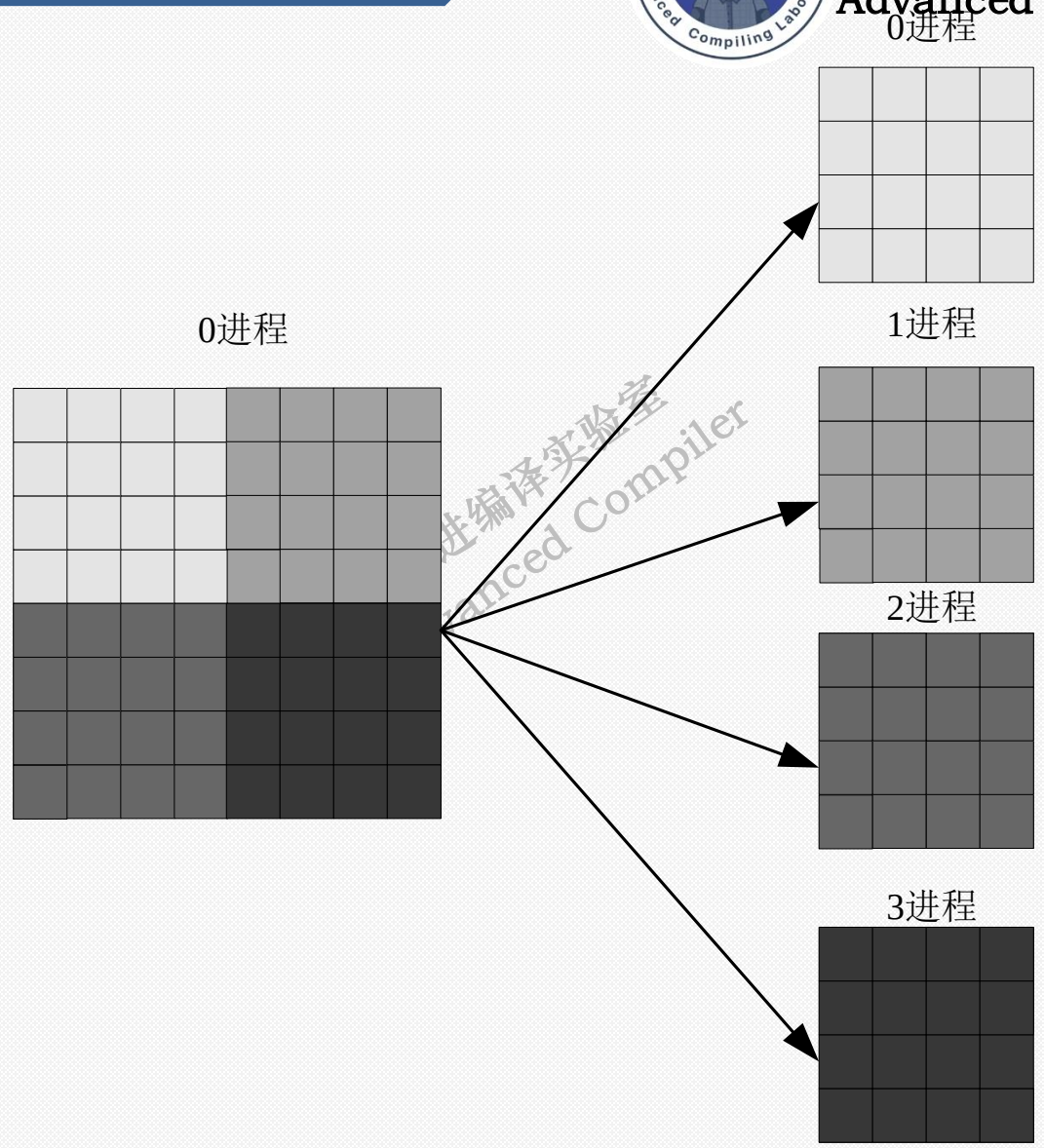
sendbuf	要进行归约操作的元素的起始地址
recvbuf	存放归约结果的起始地址
count	sendbuf中的数据个数
datatype	sendbuf的元素类型
op	归约操作符
root	根进程的进程号
comm	通信域

11.2.3 棋盘式分解

- 前面实现了按行分解以及按列分解的并行矩阵乘法运算，但是这两种分解方法都存在着一个问题，即随着问题规模的增加通信量和存储量也会急剧增加，导致缓存命中率下降影响程序性能，而对矩阵进行棋盘式分解在进行运算时能够极大的提高缓存的命中率。
- 在棋盘式分解中，所有进程构成一个虚拟网格，并且所要处理的A矩阵和B矩阵也要按照这个网格进行数据划分，每个进程只负责一个块内的矩阵乘法，进程间的关系类似一种基于二维网格的虚拟拓扑结构。

11.2.3 棋盘式分解

- 这种分解方法的虚拟网格数和进程数是一一对应的。
- 进一步节省了存储量以及通信总量，具有较高的可扩展性。



11.2.3 棋盘式分解

- 在基于棋盘式分解的并行矩阵乘算法中，最突出的代表是Cannon算法。Cannon算法是一种存储有效的算法。为了使两矩阵的下标满足相乘的要求，它不是将矩阵完整的行或列进行多播传送，而是有目的地在各行和各列上实施循环位移，降低处理器的总存储要求，解决了使用按行分解算法时由于矩阵维数增加而带来的内存快速消耗问题。算法实现的流程如下：

① 将矩阵A和B分成√P*√P个分块，每个分块负责n/√P*n/√P的数据。其中P为进程总数，n为矩阵维数。并将每块矩阵按照行优先的顺序映射到P个处理器上。

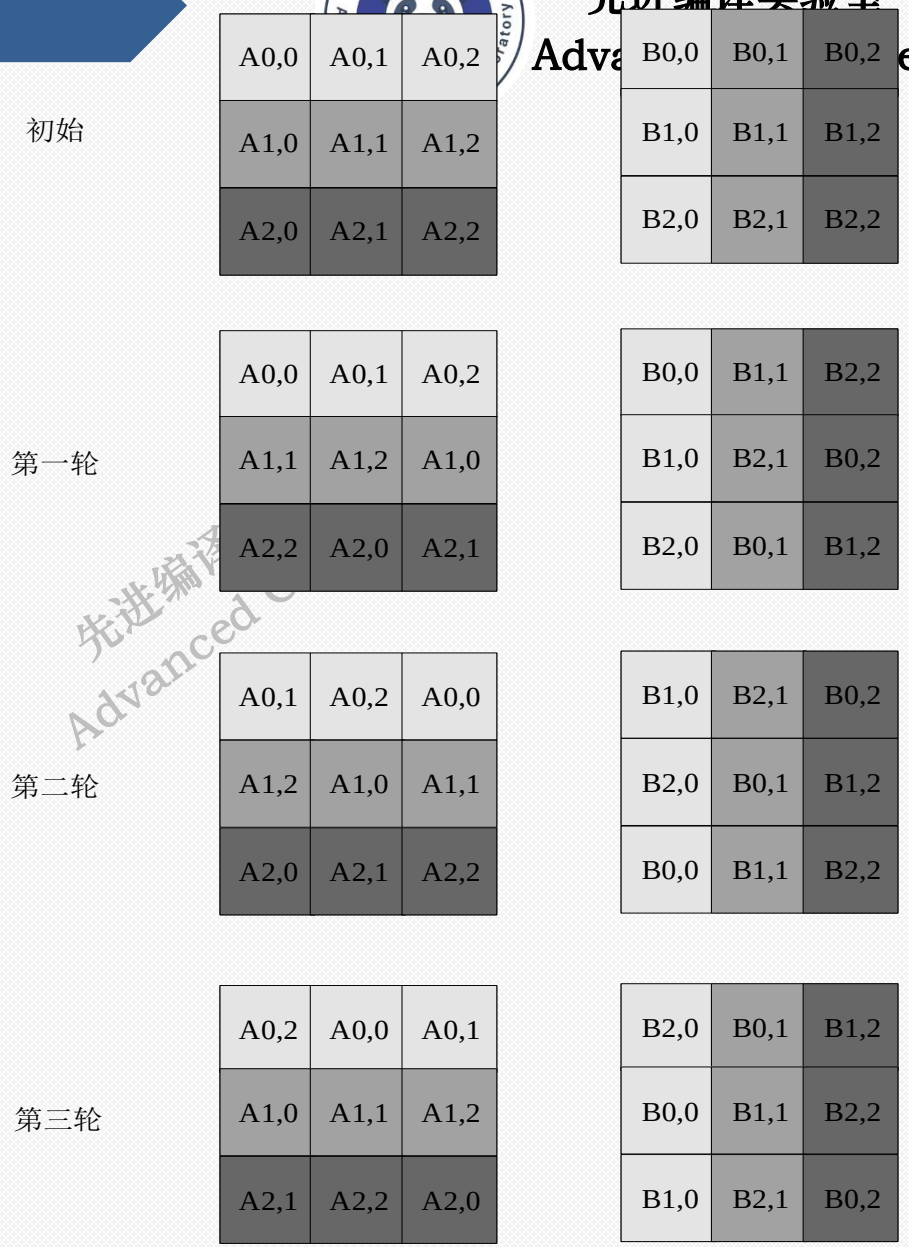
② 将块A_(i,j) (0<=i,j<√P)向左循环移动i步，将块B_(i,j) (0<=i,j<√P)向上循环移动j步。

③ P_(i,j)执行乘法和加法运算，将块A_(i,j) (0<=i,j<√P)向左循环移动1步；将块B_(i,j) (0<=i,j<√P)向上循环移动1步。

④ 重复第3步，在P_(i,j)中共执行√P次乘法和加法运算和√P次块A_(i,j)和B_(i,j)的循环单步移动。

11.2.3 棋盘式分解

- 例如：在开启9进程时，矩阵A和矩阵B使用Cannon算法的矩阵位移示意如右图所示，其中每个分块对应着一个进程，每个进程仅计算结果C的部分数据。



- 在实现Cannon算法时会使用到以下MPI函数
  - MPI_Cart_create
  - MPI_Cart_rank
  - MPI_Cart_cords
  - MPI_Cart_shift
  - MPI_Sendrecv
  - MPI_Sendrecv_replace

- MPI_Cart_create函数用于描述任意维的笛卡尔结构，其函数原型如下。

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,int reorder, MPI_Comm *comm_cart)
comm_old        输入通信域
ndims           笛卡尔网格的维数
dims            大小为ndims的整数数组，定义每一维的进程数。对于二维，就是指每行和每列各多少进程
periods         大小为ndims的逻辑数组定义在一维上网格的周期性。即数组越界后能否正确循环指定进程号
reorder         标识数是否可以重排序
comm_cart       带有新的笛卡尔拓扑的通信域
```

- MPI_Cart_rank函数用于笛卡尔通信域comm中的坐标coors映射为进程号rank，其函数原型如下。

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
comm            带有笛卡尔结构的通信域
coords          坐标，是一个整数数组
rank            坐标对应的一维线性坐标，是一个整数
```

- MPI_Cart_coords函数将comm通信域中的一维线性坐标映射为maxdims维的笛卡尔坐标coords，其函数原型如下。

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
comm                带有笛卡尔结构的通信域
rank                一维线性坐标，是一个整数
maxdims             维数
coords              返回一维线性坐标对应的坐标
```

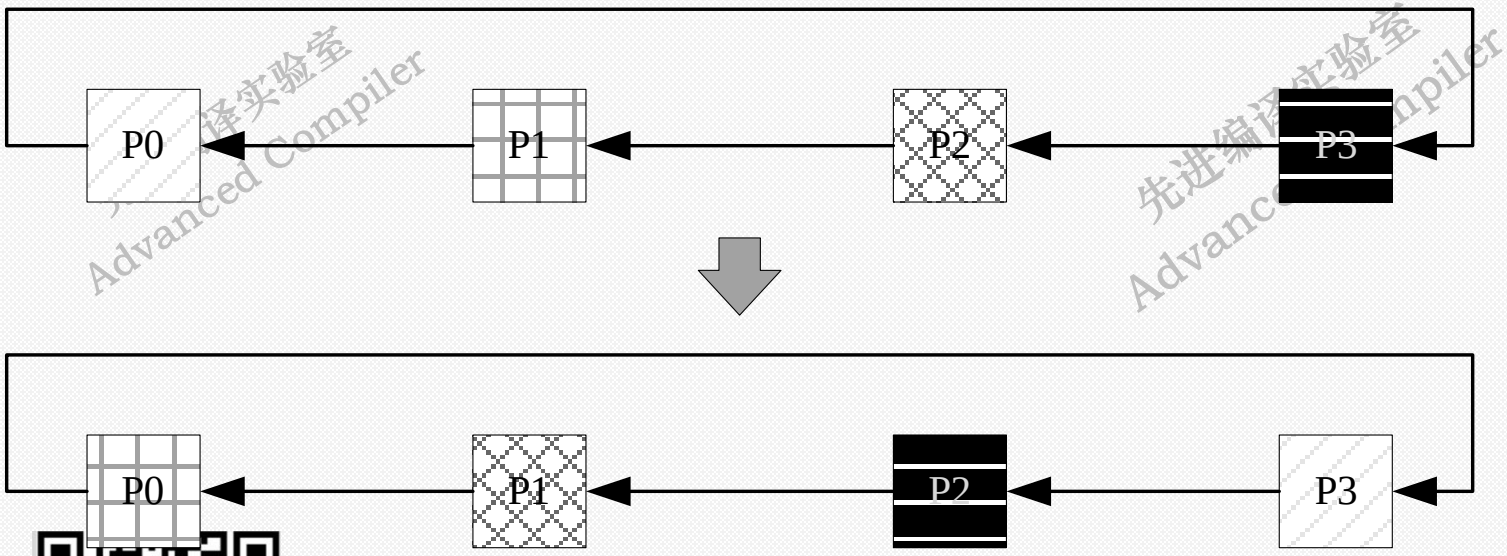
- MPI_Cart_shift用于获取本进程在笛卡尔网格的direction维度上距离为disp的进程编号信息，其函数原型如下。

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)
comm                带有笛卡尔结构的通信域
direction           需要平移的坐标维数
disp                偏移量
rank_source         本进程在direction维disp正方向距离的进程号
rank_dest           本进程在direction维disp反方向距离的进程号
```

- MPI_Sendrecv_replace函数用于在同一标识的起始地址处阻塞地交换数据，其函数原型如下，所交换的数据的个数以及数据类型也都应该相同。

```
int MPI_Sendrecv_replace(void * buf, int count, MPI_Datatype datatype,int dest, int sendtag, int source, int recvtag,
MPI_Comm comm, MPI_Status *status);
buf                 发送和接收数据的起始地址
count               发送和接收数据的个数
datatype            数据类型
dest                目的进程号
sendtag             发送数据的标识
source              源进程号
recvtag             接收数据的标识
comm                源和目的的通信域
status              发送和接收的状态
```


- MPI_Sendrecv_replace函数用于在同一标识的起始地址处阻塞地交换数据，其函数原型如下，所交换的数据的个数以及数据类型也都应该相同。

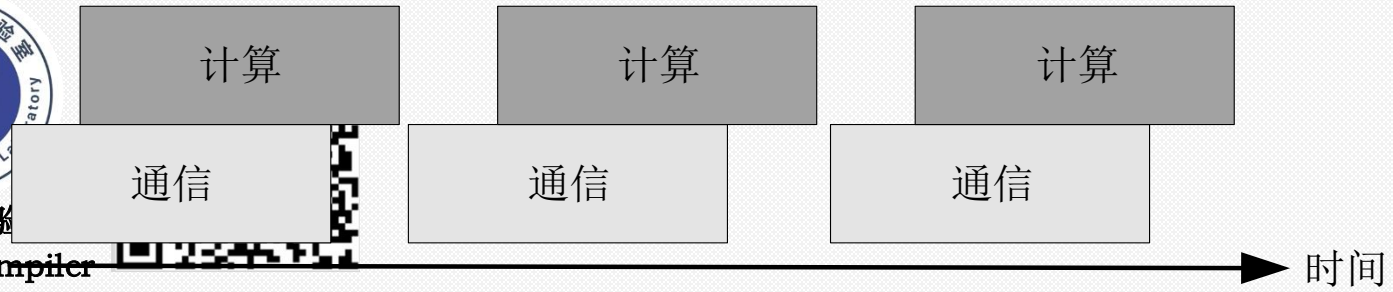


11.3 重叠通信和计算

- 程序串行执行时需要先完成通信再进行计算，每部分的通信和计算都是串行执行的。

- 串行执行的方式容易理解，但是执行速度较慢，因此在编程时应尽可能地把对性能影响较大的部分并行化。

- 通信与计算进行重叠可以提高并行程序的运算速度、避免程序隐式串行化以及使进程间通信的竞争达到最小化。每部分的通信和计算都有一定程度的并行。



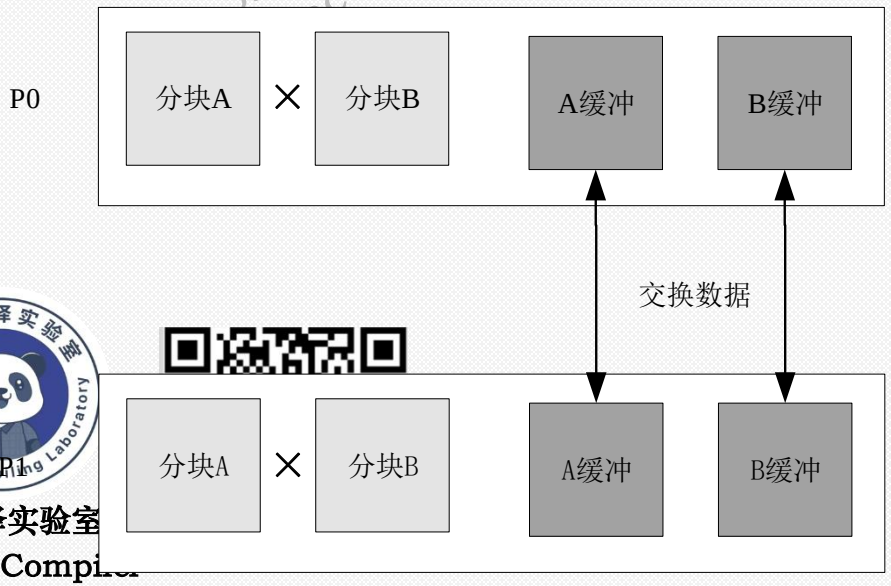
11.3 重叠通信和计算

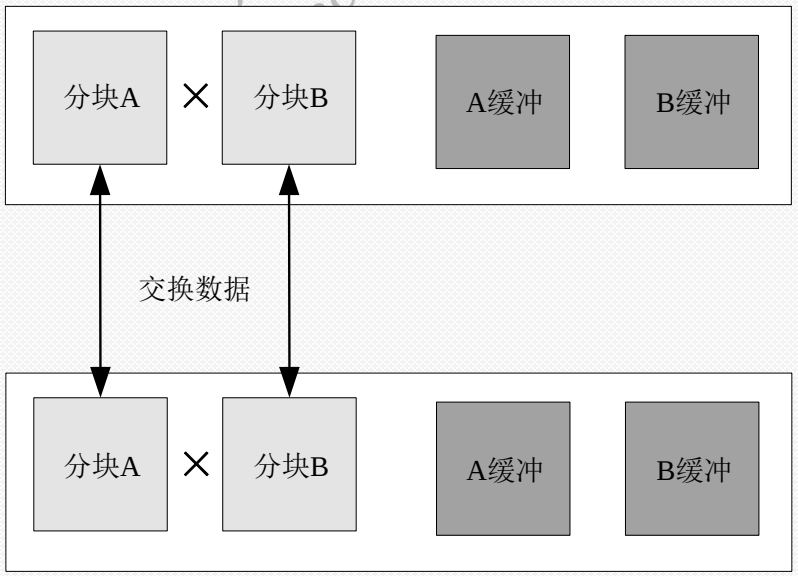
- 在方法上，可分为单进程的通信与计算重叠以及多进程间的通信与计算重叠。
- 在单个进程上，一种方法是使用多线程技术，让主线程在执行与通信无关的代码时开始一个辅助线程用于传输数据；另一种方法是在收发数据时使用非阻塞的通信函数去传输数据。
- 对于实现的棋盘式分解的矩阵乘代码来说，可以将其中通信替换为非阻塞式通信函数，再利用通信和计算重叠的方法优化程序中的通信部分。

11.3 重叠通信和计算

- 具体来说，需要先开辟两个缓冲区，在非阻塞通信的情况下使得矩阵A分块和矩阵B分块相乘和数据的收发工作分别在两个缓冲区同时进行。

- 要在两块缓冲区之间来回的切换，数据的安全与正确性是必须保证的，所以当进程完成一块缓冲区上的计算后，需要先确认第二块缓冲区上的数据是否完成了通信，只有在完成通信的情况下才能继续对第二块缓冲区上的数据进行计算，并且让第一块缓冲区去交换新的数据。





11.3 重叠通信和计算

- 在实现非阻塞通信算法时需要用到MPI的函数有MPI_Isend、MPI_Irecv、MPI_Wait、MPI_Test，下面分别进行介绍。

- 其中MPI_Isend用于非阻塞式发送信息，使用方法与MPI_Send一致，MPI_Request对象用于检测通信状态。

int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,MPI_Comm comm, MPI_Request *request)

buf	发送缓冲区的起始地址
count	发送数据的个数
datatype	发送数据的数据类型
dest	目的进程号
tag	消息标志
comm	通信域
request	返回的非阻塞通信对象

11.3 重叠通信和计算

- MPI_Irecv用于非阻塞式接收其它进程发送过来的消息，和MPI_Recv不同之处在于，不需要等到接收完所有数据就可以返回函数调用。

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag,MPI_Comm comm, MPI_Request *request)

buf	接收缓冲区的起始地址
count	接收数据的最大个数
datatype	每个数据的数据类型
source	源进程标识
tag	消息标志
comm	通信域
request	非阻塞通信对象

11.3 重叠通信和计算

- MPI_Wait函数用于等待某个通信的完成，其需要检查是否完成非阻塞点对点通信的发送或者接收，如未完成则说明程序阻塞要完成通信后才继续运行。

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
request                    非阻塞通信对象
status                     返回的状态
```

- 函数MPI_Test与MPI_Wait具有类似的功能，但并不支持阻塞，只是返回一个是否完成通信的信息。当flag为0时代表未完成，为1时代表完成。

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
request                    非阻塞通信对象
flag                       操作是否完成标志
status                     返回的状态
```


11.4.1串行算法

- Eratosthenes筛法是由古希腊数学家提出的一种素数求解算法用来找出一定范围内所有的素数，下面对该算法的执行步骤进行简单描述。

① 假设有一整数列表，0,1,2,3,……，n，其中的数都未被标记。

② 令k等于列表中下一个未被标记的数，将其所有倍数标记。

③ 重复第二步直到k^2>n。

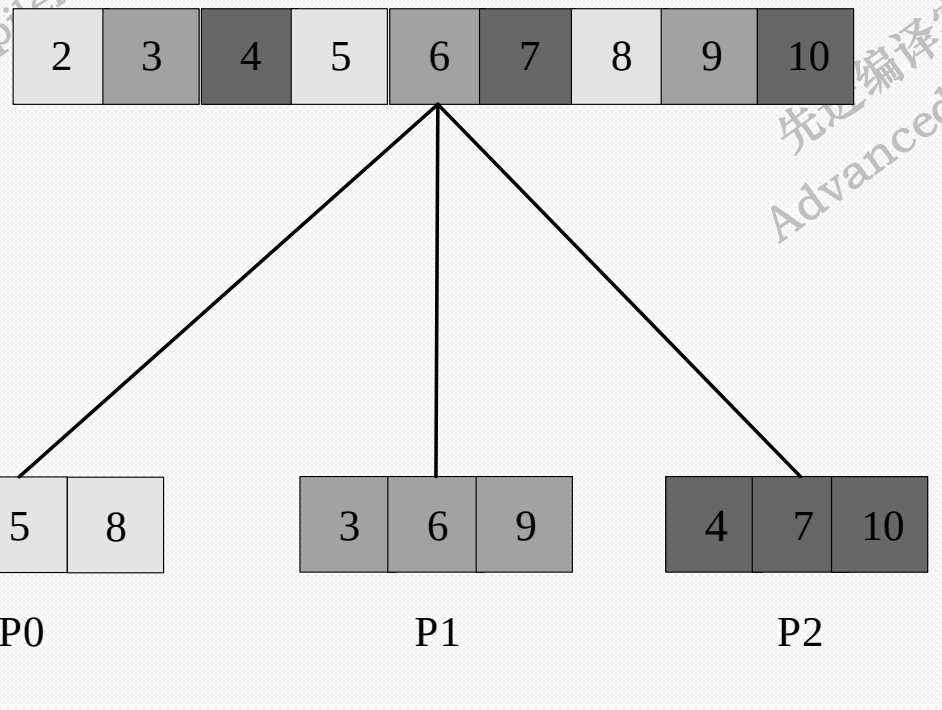
④ 列表中未被标记数的即为该列表中所有的素数。

11.4.1串行算法

右图展示列表范围为2-34时使用Eratosthenes算法筛选素数的过程，其中未被标记的即为素数。

11.4.2交叉分解

- Eratosthenes筛法在进行并行化之前需要对数据进行分解，本节采用的数据分解方法为交叉数据分解，也就是每个进程按照进程号的大小依次进行数据划分。交叉分解对于一个给定的数组下标，很容易确定负责该数据的计算的进程号，为对数字2~10使用交叉分解如下所示。



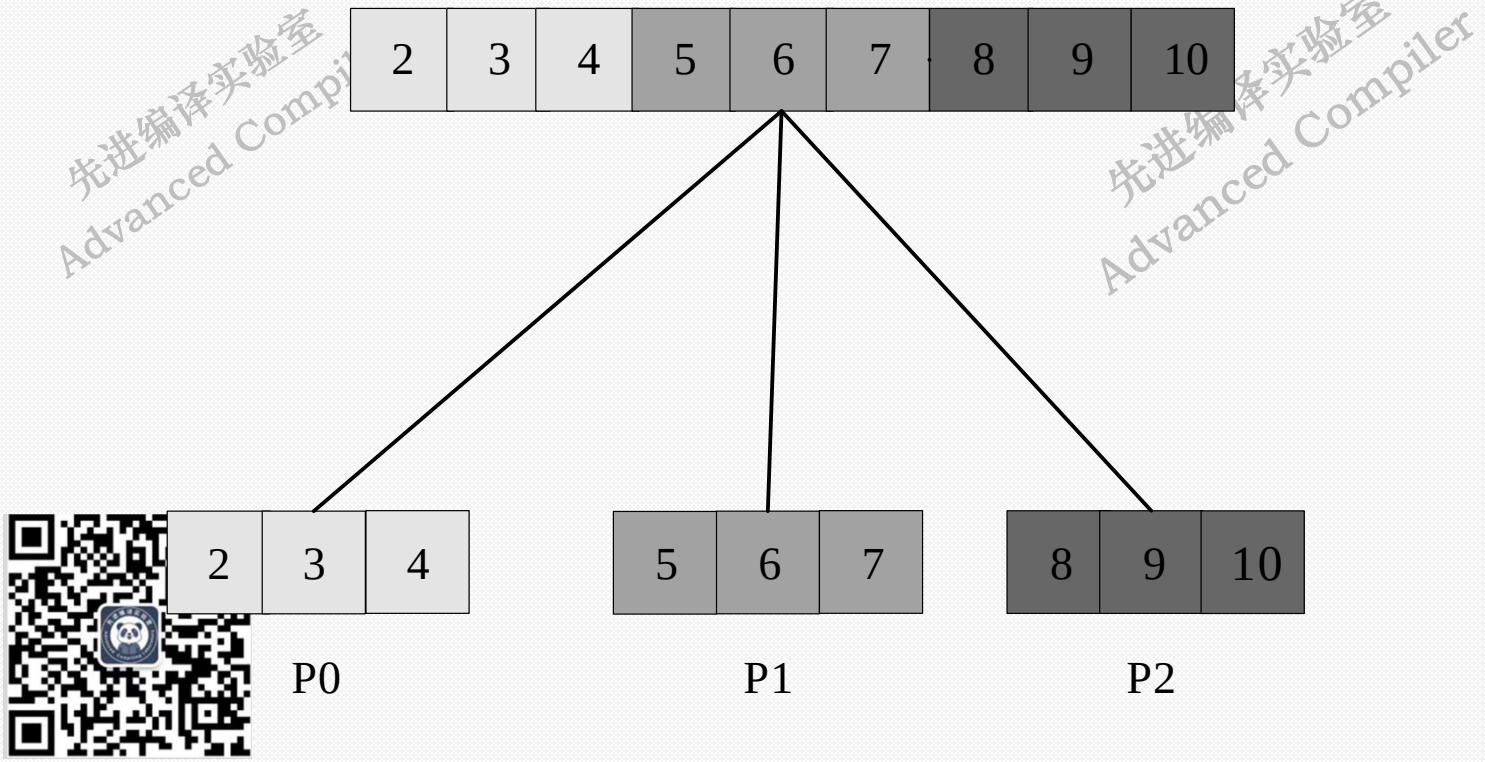
- 经测试交叉数据分解的素数筛法相比串行算法程序性能反而下降了。经分析此程序运行效率差的主要原因有以下三点：

1. 在标记某个数的倍数时，需要重复地计算当前下标所对应的数是什么；
2. 由于数据分布不均匀，在某些时刻可能只有一个进程在工作，其它所有进程都在等待接收下一个未被标记数；
3. 每次选择下一个未被标记数时都需要进行一次归约操作和广播操作。

- 总的来说在严重的负载不均衡情况下，使用多个处理器进行数据处理的效率可能还远远不及使用单处理器进行数据处理。所以在使用MPI实现程序并行时，需要注意优化数据划分方法使进程负载尽可能均衡，进而提高整个程序的执行速度。

11.4.3 按块分解

- 对于p个进程来说，按块分解就是将原始任务依次划分成p个块，每块的大小由处理器数是否能整除任务量n决定，若能整除则每个块的大小完全相等，不能整除时前n%p个进程处理[n/p]个数据，剩余进程处理[n/p]个数据，之后每个进程并行地进行筛选。



11.5 冗余计算减少通信

- 按块分解的并行Eratosthenes筛法的通信耗时较长，本节将分析如何使用冗余计算的方式提高按块数据分解算法的性能。
- MPI程序中进程之间通信时，都会产生创建发送或接收消息的开销，因此为提高使用MPI的应用程序的性能，尽量减少进程之间交换的消息数量是有必要的。
- 当通信时间较长时，进程间相互通信获取数据的时间可能会比进程直接计算的时间更长，会导致性能下降，所以为了减少算法中的通信耗时可以尝试利用冗余计算的方式优化。

11.6 小结

MPI编程简介

对MPI的基本概念、MPI的函数库和MPI并行程序的编写进行介绍，然后详细描述从串行版本矩阵乘法到MPI版本矩阵乘法的改写过程。

冗余计算减少通信

提高MPI应用程序的性能时，尽量减少进程之间交换的消息数量是有必要的。当通信时间较长时，进程间相互通信获取数据的时间可能会比进程直接计算的时间更长，此时为了减少算法中的通信耗时可以尝试利用冗余计算的方式优化。

MPI程序优化

数据划分优化

从数据的划分方式入手，使用按行分解、按列分解和棋盘式分解的数据划分方法对MPI版矩阵乘法程序进行优化。

负载均衡优化

以Eratosthenes筛法为例，分别使用交叉分解和按块分解来证明在MPI程序优化时，进程间负载均衡的重要性。

重叠通信和计算

通信与计算进行重叠可以提高并行程序的运算速度、避免程序隐式串行化以及使进程间通信的竞争达到最小化。