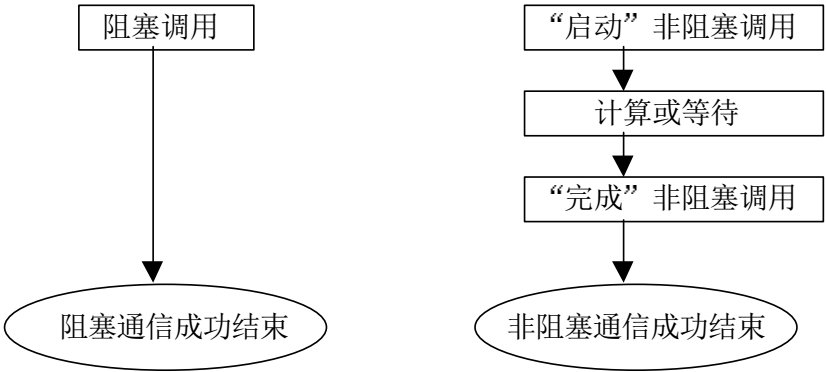
**MPI---进阶编程：非阻塞通信MPI程序设计（实现计算与通信的重叠）**

所有阻塞通信的形式都有相应的非阻塞通信的形式。非阻塞通信:不必等到通信操作完全完成便可以返回,实现了计算与通信的重叠 ,大大提高程序执行的效率。

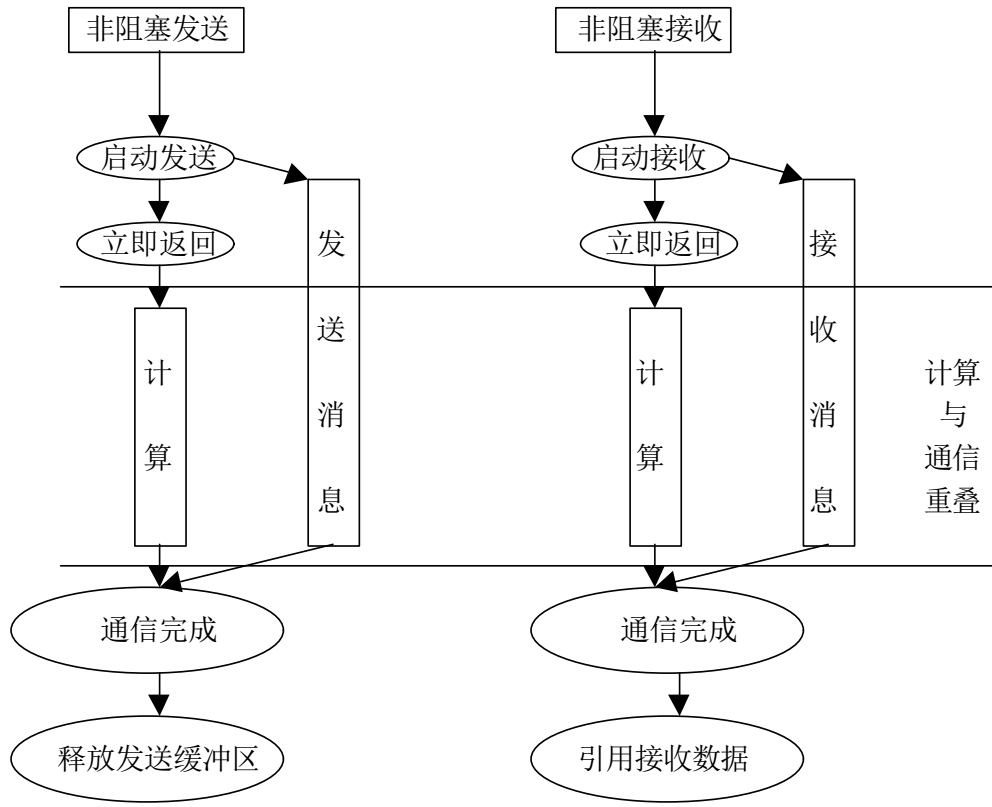
由于当非阻塞通信调用返回时一般该通信操作还没有完成，因此对于非阻塞的发送操作，发送缓冲区必须等到发送完成后才能释放，这样便需要引入新的手段（非阻塞通信完成对象），让程序员知道什么时候该消息已成功发送；同样，对于非阻塞的接收操作，该调用返回后并不意味着接收消息已全部到达，必须等到消息到达后才可以引用接收到的消息数据。

阻塞通信 只需要一个调用函数即可以完成 但是对于非阻塞通信一般需要两个调用函数 首先是非阻塞通信的 启动 但启动并不意味着该通信过程的完成 因此 为了保证通信的完成 还必须调用与该通信相联系的通信 完成 调用接口 通信完成调用才真正将非阻塞通信完成

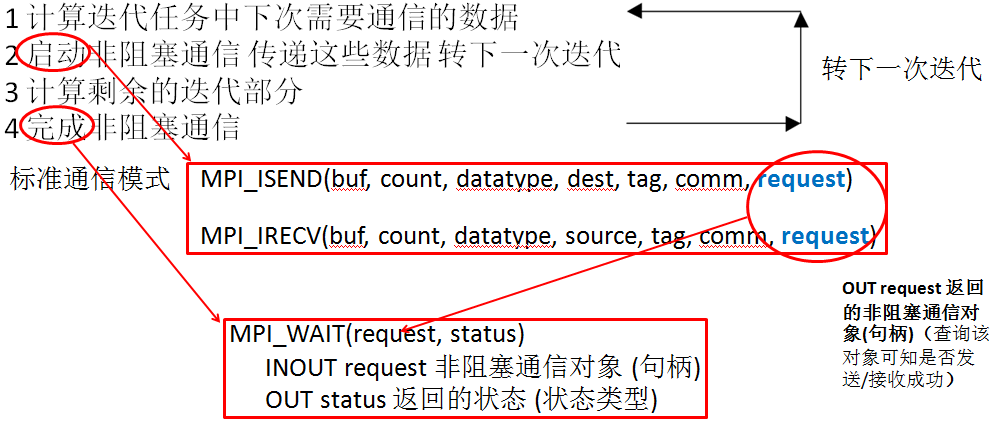


**阻塞与非阻塞调用的对比**

为了实现计算与通信的最大重叠 一个通用的原则就是“尽早开始通信，尽晚完成通信”在开始通信和完成通信之间进行计算，这样通信启动得越早，完成得越晚，就有可能有更多的计算任务可以和通信重叠，也使通信可以在计算任务执行期间完成，而不需要专门的等待时间 。



**标准非阻塞消息发送和接收**



**非阻塞迭代过程**

**例9 非阻塞通信MPI程序设计**

**#include "mpi.h"**

**#include <stdio.h>**

**#include <stdlib.h>**

**#define N 8**

**#define SIZE N/4**

**#define T 2**

**void print\_matrix(int myid, float myRows[][N]);**

**int main(int argc, char \*argv[])**

**{**

**float matrix1[SIZE+2][N], matrix2[SIZE+2][N];**

**int myid;**

**MPI\_Status status[4];**

**MPI\_Request request[4];**

**MPI\_Init(&argc, &argv);**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myid);**

**// 初始化**

**int i,j;**

**for(i=0; i<SIZE+2; i++)**

**{**

**for(j=0; j<N; j++)**

**{**

**matrix1[i][j] = matrix2[i][j] = 0;**

**}**

**}**

**if(0==myid) // 按行划分 上面第一分块矩阵 上边界**

**{**

**for(j=0; j<N; j++) matrix1[1][j] = matrix2[1][j] = N;**

**}**

**if (3==myid)**

**{ // 按行划分 最下面一分块矩阵 下边界**

**for(j=0; j<N; j++) matrix1[SIZE][j] = matrix2[SIZE][j] = N;**

**}**

**for(i=1; i<SIZE+1; i++) // 每个矩阵的两侧边界**

**{**

**matrix1[i][0] = matrix1[i][N-1] = matrix2[i][0] = matrix2[i][N-1] = N;**

**}**

**// 引入虚拟进程 并计算每个进程上下相邻进程**

**int up\_proc\_id = myid==0 ? MPI\_PROC\_NULL : myid-1;**

**int down\_proc\_id = myid==3 ? MPI\_PROC\_NULL : myid+1;**

**// jacobi迭代过程**

**int t,row,col;**

**for(t=0; t<T; t++)**

**{**

**// 1 计算边界数据**

**if(0==myid) // 最上的矩阵块**

**{**

**for (col=1; col<N-1; col++)**

**{**

**matrix2[SIZE][col] = (matrix1[SIZE][col-1]+matrix1[SIZE][col+1]+**

**matrix1[SIZE+1][col]+matrix1[SIZE-1][col])\*0.25;**

**} }**

**}**

**else if (3==myid)**

**{ // 最下的矩阵块**

**for (col=1; col<N-1; col++)**

**{**

**matrix2[1][col] = (matrix1[1][col-1]+matrix1[1][col+1]+**

**matrix1[2][col]+matrix1[0][col])\*0.25;**

**}**

**}**

**else**

**{**

**for(col=1; col<N-1; col++) // 中间的矩阵块**

**{**

**matrix2[SIZE][col] = (matrix1[SIZE][col-1]+matrix1[SIZE][col+1]+**

**matrix1[SIZE+1][col]+matrix1[SIZE-1][col])\*0.25;**

**matrix2[1][col] = (matrix1[1][col-1]+matrix1[1][col+1]+matrix1[2][col]+matrix1[0][col])\*0.25;**

**}**

**}**

**// 2 利用非阻塞函数传递边界数据 为下一次计算做准备**

**int tag1 = 1, tag2 = 2;**

**MPI\_Isend(&matrix2[1][0], N, MPI\_FLOAT, up\_proc\_id, tag1, MPI\_COMM\_WORLD, &request[0]);**

**MPI\_Isend(&matrix2[SIZE][0],N,MPI\_FLOAT,down\_proc\_id,tag2,MPI\_COMM\_WORLD, &request[1]);**

**MPI\_Irecv(&matrix1[SIZE+1][0], N, MPI\_FLOAT, down\_proc\_id, tag1, MPI\_COMM\_WORLD, &request[2]);**

**MPI\_Irecv(&matrix1[0][0], N, MPI\_FLOAT, up\_proc\_id, tag2, MPI\_COMM\_WORLD, &request[3]);**

**// 3 计算中间数据**

**int begin\_row = 0==myid ? 2 : 1;**

**int end\_row = 3==myid ? (SIZE-1) : SIZE;**

**for (row=begin\_row; row<end\_row; row++)**

**{**

**for (col=1; col<N-1; col++)**

**{**

**matrix2[row][col] = (matrix1[row][col-1]+matrix1[row][col+1]+**

**matrix1[row+1][col]+matrix1[row-1][col])\*0.25;**

**}**

**}**

**// 4 更新矩阵 并等待各个进程间数据传递完毕**

**for (row=begin\_row; row<=end\_row; row++)**

**{**

**for (col=1; col<N-1; col++)**

**{**

**matrix1[row][col] = matrix2[row][col];**

**}**

**}**

**MPI\_Waitall(4, &request[0], &status[0]);**

**//MPI\_Waitall:Waits for all given MPI Requests to complete Synopsis**

**//int MPI\_Waitall(int count, MPI\_Request array\_of\_requests[], MPI\_Status array\_of\_statuses[])**

**}**

**MPI\_Barrier(MPI\_COMM\_WORLD);**

**print\_matrix(myid, matrix1);**

**MPI\_Finalize();**

**}**

**void print\_matrix(int myid, float myRows[][N])**

**{**

**int i,j;**

**int buf[1];**

**MPI\_Status status;**

**buf[0] = 1;**

**if ( myid>0 )**

**{**

**MPI\_Recv(buf, 1, MPI\_INT, myid-1, 0, MPI\_COMM\_WORLD, &status);**

**}**

**printf("Result in process %d:\n", myid);**

**for ( i = 0; i<SIZE+2; i++)**

**{**

**for ( j = 0; j<N; j++)**

**printf("%1.3f\t", myRows[i][j]);**

**printf("\n");**

**}**

**if ( myid<3 )**

**{**

**MPI\_Send(buf, 1, MPI\_INT, myid+1, 0, MPI\_COMM\_WORLD);**

**}**

**MPI\_Barrier(MPI\_COMM\_WORLD);**

**}**

**归约MPI\_REDUCE：将组内每个进程输入缓冲区中的数据按给定的操作op进行运算**

MPI\_REDUCE(sendbuf,recvbuf,count,datatype,op,root,comm)

IN sendbuf 发送消息缓冲区的起始地址(可选数据类型)

OUT recvbuf 接收消息缓冲区中的地址(可选数据类型)

IN count 发送消息缓冲区中的数据个数(整型)

IN datatype 发送消息缓冲区的元素类型(句柄)

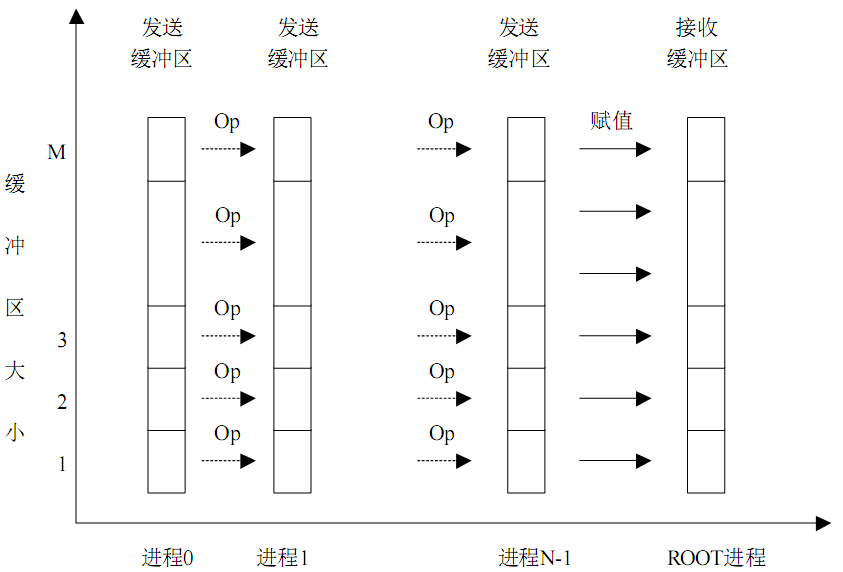
IN op 归约操作符(句柄)

IN root 根进程序列号(整型)

IN comm 通信域(句柄)

**op 归约操作符(句柄)**





**MPI归约操作图示**

**例11 归约MPI\_Reduce**

**/\***

**组通信MPI程序**

**通过积分（和式）求PI的近似值，利用了广播MPI\_Bcast和归约MPI\_Reduce。**

**root进程负责输入矩形的个数N并将它广播给各个进程，**

**各个进程计算部分和mypi后将结果(归约)累加到root进程的pi，**

**root进程打印pi。**

**\*/**

**#include "mpi.h"**

**#include <stdio.h>**

**#include <math.h>**

**/\* 定义被积函数f(x) \*/**

**double f(double x)**

**{**

**return 4.0/(1.0+x\*x);**

**}**

**void main(int argc, char \*argv[])**

**{**

**int root=0,size,myid;**

**int N=0; /\* [0,1]划分为N个矩形 \*/**

**double mypi,pi,h,sum,x;**

**double PI25DT=3.141592653589793238462643; /\* 先给出已知的较为准确的PI值 \*/**

**double startwtime=0.0,endwtime;**

**int namelen;**

**char processor\_name[MPI\_MAX\_PROCESSOR\_NAME];**

**int i;**

**MPI\_Init(&argc,&argv);**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD,&myid);**

**MPI\_Comm\_size(MPI\_COMM\_WORLD,&size);**

**MPI\_Get\_processor\_name(processor\_name,&namelen);**

**fprintf(stderr,"Process %d of %d on %s\n",myid,size,processor\_name);**

**MPI\_Barrier(MPI\_COMM\_WORLD); /\* 同步 \*/**

**if(myid==root)**

**{**

**printf("Please give N=");**

**scanf("%d",&N);**

**startwtime=MPI\_Wtime(); /\* 时间计时开始 \*/**

**}**

**MPI\_Bcast(&N,1,MPI\_INT,root,MPI\_COMM\_WORLD); /\* 将N值广播出去 \*/**

**h=1.0/(double)N; /\* 得到矩形的宽度，所有的矩形的宽度都相同 \*/**

**sum=0.0; /\* 给矩形的面积赋初值 \*/**

**for(i=myid+1;i<=N;i+=size)**

**{**

**/\* 每一个进程计算一部分矩形的面积，若总进程数为size=4,**

**将[0,1]划分为100个矩形，则各个进程分别计算矩形块：**

**0进程 1, 5, 9, 13, ..., 97**

**1进程 2, 6, 10, 14, ..., 98**

**2进程 3, 7, 11, 15, ..., 99**

**3进程 4, 8, 12, 16, ..., 100**

**\*/**

**x=h\*((double)i-0.5);**

**sum+=f(x);**

**}**

**mypi=h\*sum; /\* 各个进程计算得到的部分和 \*/**

**/\* 归约：各进程将其计算的部分和mypi累加到root进程的pi \*/**

**MPI\_Reduce(&mypi,&pi,1,MPI\_DOUBLE,MPI\_SUM,root,MPI\_COMM\_WORLD);**

**if(myid==root) /\* 执行累加的root进程打印pi \*/**

**{**

**fprintf(stderr,"pi is appproximately %0.16f, Error is %.16f\n",**

**pi,fabs(pi-PI25DT)); /\* 打印pi和及其与PI25DT的误差 \*/**

**endwtime=MPI\_Wtime(); /\* 墙上时间计时结束 \*/**

**/\* 打印并行计算pi值所花费的时间 \*/**

**fprintf(stderr,"wall clock time=%f\n",endwtime-startwtime);**

**}**

**MPI\_Finalize();**

**}**