装订线 答题时不要超过此线

中 国 科 学 技 术 大 学

2009－2010 学年第一学期考试试卷

参考解答

考试科目: 并行程序设计　　得分:_____

学生所在系:_____　姓名:_____　学号:_____

一、给出用基本的 **MPI_Send** 和 **MPI_Recv** 函数实现 **MPI_Allgather** 函数功能的伪代码。这三个函数的调用格式如下：（20 分）

（1） **MPI_Send(buf, count, datatype, dest, tag, comm)**

（2） **MPI_Recv(buf, count, datatype, source, tag, comm, status)**

（3） **MPI_Allgather**(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

参考解答：

```
for( i=0; i<GroupSize; i++ ){
    MPI_Send(sendbuf, sendcount, sendtype, i, myrank, MPI_COMM_WORLD);
}
for( i=0; i<GroupSize; i++ ){
    MPI_Recv(recvbuf+i*recvcount, recvcount,
                        recvtype, i, i, MPI_COMM_WORLD,status);
}
```

二、以下是一个用 C 语言描述的计算 π 的串行程序。（30 分）

```
static long num_steps = 100000;
double step;
void main ()
{       int i; double x, pi, sum = 0.0;
        step = 1.0/(double) num_steps;
        for (i=1;i<= num_steps; i++){
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

请给出**四种不同的 OpenMP 程序**实现来并行计算 π。

参考解答：

使用并行域并行化的程序

```
//OpenMP PI Program:
//Parallel Region example (SPMD Program)
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{       int i;
        double x, pi, sum[NUM_THREADS];
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS)
#pragma omp parallel
{       double x;
        int id;
        id = omp_get_thread_num();
        for (i=id, sum[id]=0.0;i< num_steps; i=i+NUM_THREADS){
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
}
        for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

使用共享任务结构并行化的程序

```
//OpenMP PI Program:
//Work sharing construct
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{       int i;
        double x, pi, sum[NUM_THREADS];
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS)
#pragma omp parallel
{       double x;
        int id;
```

```
        id = omp_get_thraead_num();
        sum[id] = 0;
#pragma omp for
        for (i=0;i< num_steps; i++){
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
}
        for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

使用 private 子句和 critical 部分并行化的程序

```
//OpenMP PI Program:
//private clause and a critical section
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{       int i;
        double x, sum, pi=0.0;
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS)
#pragma omp parallel private (x, sum)
{
        id = omp_get_thread_num();
        for (i=id,sum=0.0;i< num_steps;i=i+NUM_THREADS){
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
#pragma omp critical
        pi += sum
}
}
```

使用并行归约得出的并行程序

```
//OpenMP PI Program :
//Parallel for with a reduction
#include <omp.h>
static long num_steps = 100000;
```

```
double step;
#define NUM_THREADS 2
void main ()
{       int i;
        double x, pi, sum = 0.0;
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS)
#pragma omp parallel for reduction(+:sum) private(x)
        for (i=1;i<= num_steps; i++){
              x = (i-0.5)*step;
              sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

三、以下程序是快速排序的一种 OpenMP 并行递归实现。请补充实线框内的程序代码。(20 分)

```
void qs(int *v, int first, int last) {
  int start[2], end[2], pivot, i, temp;
  // 以下为数组划分的代码
  if (first < last) {
     start[1] = first;
     end[0] = last;
     pivot = v[(first + last) / 2];
     while (start[1] <= end[0]) {
       while (v[start[1]] < pivot) start[1]++;
       while (pivot < v[end[0]])   end[0]--;
       if (start[1] <= end[0]) {
         temp = v[start[1]];
         v[start[1]] = v[end[0]];
         v[end[0]] = temp;
         start[1]++;
         end[0]--;
       }
     }
     start[0] = first;
     end[1]   = last;
  // 以下实线框内为并行部分的代码，请补充！
```

参考解答如下面文本框所示：

```
#pragma omp parallel
{
#pragma omp for nowait
        for(i = 0; i <= 1; i++) {
           qs(v, start[i], end[i]);
        }
}
```

```
  } // end-if
} // end-of-qs
```

四、请给出矩阵分块乘法的并行 MPI 实现代码（30 分）

矩阵乘法也可以用分块的思想实现并行，即分块矩阵乘法(Block Matrix Multiplication)，将矩阵 A（mXn）按行划分为 p 块(p 为处理器个数)，设 $u=\lceil m/p \rceil$，每块含有连续的 u 行向量，这些行块依次记为 $A_0,A_1,\cdots,A_{p-1}$，分别存放在标号为 0,1,…, p-1 的处理器中。对矩阵 B（nXk）按列划分为 p 块，记 $v=\lceil k/p \rceil$，每块含有连续的 v 列向量，这些列块依次记为 $B_0,B_1,\cdots,B_{p-1}$，分别存放在标号 0,1, …, p-1 为的处理器中。将结果矩阵 C 也相应地同时进行行、列划分，得到 $p\times p$ 个大小为 $u\times v$ 的子矩阵，记第 i 行第 j 列的子矩阵为 C_{ij}，显然有 $C_{ij}=A_i\times B_j$，其中，A_i 大小为 $u\times n$，B_j 大小为 $n\times v$。

开始，各处理器的存储内容如图(a)所示。此时各处理器并行计算 $C_{ii}=A_i\times B_i$ 其中 i=0,1,…,p-1，此后第 i 号处理器将其所存储的 B 的列块送至第 i-1 号处理器（第 0 号处理器将 B 的列块送至第 p-1 号处理器中，形成循环传送），各处理器中的存储内容如图(b)所示。它们再次并行计算 $C_{ij}=A_i\times B_j$，这里 $j=(i+1) \bmod p$。B 的列块在各处理器中以这样的方式循环传送 p-1 次并做 p 次子矩阵相乘运算，就生成了矩阵 C 的所有子矩阵。最终，编号为 i 的处理器的内部存储器存有 C 子矩阵 $C_{i0},C_{i1},\ldots,C_{i(p-1)}$。

处理器编号	存储内容	
0	A_0	B_0
1	A_1	B_1
2	A_2	B_2
•••	•••	•••
P-1	A_{p-1}	B_{p-1}

(a)

处理器编号	存储内容	
0	A_0	B_1
1	A_1	B_2
2	A_2	B_3
•••	•••	•••
P-1	A_{p-1}	B_0

(b)

参考解答如下：

矩阵并行分块乘法算法

输入：$A_{m\times n}$, $B_{n\times k}$,

输出：$C_{m\times k}$

Begin

对所有处理器 my_rank(my_rank=0,…,p-1)同时执行如下的算法:

(1)目前计算 C 的子块号 $l=(i+\text{my\_rank}) \bmod p$

(2)**for** z=0 **to** u-1 **do**

for j=0 **to** v-1 **do**

$c[l,z,j]$=0

for s=0 **to** n-1 **do**

$c[l,z,j]=c[l,z,j]+a[z,s]*b[s,j]$

end for

end for

end for

(3)计算左邻处理器的标号 $mm1=(p+\text{my\_rank}-1) \bmod p$

计算右邻处理器的标号 *mp*1=(my_rank+1) *mod p*

(4)**if** (*i*≠*p*-1) **then**

(4.1)**if** (my_rank mod 2= 0) **then** /*编号为偶数的处理器*/

(i)将所存的 B 的子块发送到其左邻处理器中

(ii)接收其右邻处理器中发来的 B 的子块

end if

(4.2)**if** (my_rank mod 2≠ 0) **then** /*编号为奇数的处理器*/

(i)将所存的 B 子块在缓冲区 buffer 中做备份

(ii)接收其右邻处理器中发来的 B 的子块

(iii)将 buffer 中所存的 B 的子块发送到其左邻处理器中

end if

end if

End

具体 MPI 程序如下：

```
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"

#define intsize sizeof(int)
#define floatsize sizeof(float)
#define charsize sizeof(char)
#define A(x,y) A[x*K+y]
#define B(x,y) B[x*N+y]
#define C(x,y) C[x*N+y]
#define a(x,y) a[x*K+y]
#define b(x,y) b[x*n+y]
#define buffer(x,y) buffer[x*n+y]
/* 此宏用来简化对标号为奇数的处理器内的缓冲空间的访问 */
#define c(l,x,y) c[x*N+y+l*n]

float *a,*b,*c,*buffer;
int s;
float *A,*B,*C;
/* A[M,K],B[P,N].正确的情况下 K 应该等于 P,否则无法进行矩阵相乘 */
int M,N,K,P ;
int m,n;
int myid;
int p;                          /* 保存工作站集群中处理器数目，也即通信子大小 */
FILE *dataFile;                 /* 用于读取输入文件内容和将计算结果输出到结果文件的临时
文件指针 */
MPI_Status status;
double time1;
double starttime,endtime;

/*
 * 函数名: readData
 * 功能：  此函数被 rankID 为 0 的进程调用，负责从 dataIn.txt 文件中读入
 *         A[M,K],B[P,N]两个相乘矩阵的数据，并为结果矩阵 C[M,N]分配空间。
 *         其中 C[N,N]=A[M,K]*B[P,N]
 * 输入：  无
 * 返回值：无
```

```
 */
void readData()
{
    int i,j;
    starttime = MPI_Wtime();

    dataFile=fopen("dataIn.txt","r");
    fscanf(dataFile,"%d%d", &M, &K);              /* 读取矩阵 A 的行，列数 M,K */
    A=(float *)malloc(floatsize*M*K);             /* 为矩阵 A 分配空间 */
    for(i = 0; i < M; i++)                        /* 读入矩阵 A 的各元素 */
    {
        for(j = 0; j < K; j++)
        {
            fscanf(dataFile,"%f", A+i*K+j);
        }
    }

    fscanf(dataFile,"%d%d", &P, &N);              /* 读取矩阵 B 的行，列数 P,N */
    if (K!=P)                                     /* K 应该等于 P,否则矩阵无法相乘
*/
    {
        printf("the input is wrong\n");
        exit(1);
    }
    B=(float *)malloc(floatsize*K*N);             /* 为矩阵 B 分配空间 */
    for(i = 0; i < K; i++)                        /* 从文件中读入矩阵 B 的各元素 */
    {
        for(j = 0; j < N; j++)
        {
            fscanf(dataFile,"%f", B+i*N+j);
        }
    }
    fclose(dataFile);

    printf("Input of file \"dataIn.txt\"\n");
    printf("%d\t %d\n",M, K);                     /* 输出 A 矩阵的维数 */
    for(i=0;i<M;i++)                              /* 输出 A 矩阵的数据 */
    {
        for(j=0;j<K;j++) printf("%f\t",A(i,j));
        printf("\n");
    }
    printf("%d\t %d\n",K, N);                     /* 输出 B 矩阵的维数 */
    for(i=0;i<K;i++)                              /* 输出 B 矩阵的数据 */
    {
        for(j=0;j<N;j++) printf("%f\t",B(i,j));
        printf("\n");
    }

    C=(float *)malloc(floatsize*M*N);             /* 为结果矩阵 C[M,N]分配空间 */

}

/*
 * 函数名: gcd
 * 功能：   此函数用来返回两个整数的不大于 group_size 的最大公因子
 * 输入：   M,N:要求最大公因数的两个整数
```

```
 *              group_size 所求公因子必须小于此参数，此参数代表用户指定的通信子大小
 * 返回值：M 和 N 的不大于 group_size 的最大公因子
 */
int gcd(int M,int N,int group_size)
{
    int i;
    for(i=M; i>0; i--)
    {
        if((M%i==0)&&(N%i==0)&&(i<=group_size))
            return i;

    }
    return 1;
}


/*
 * 函数名: printResult
 * 功能：此函数被 rankID 为 0 的进程调用，用来将 A,B,C 矩阵打印输出给用户，
 *          并输出用于分发数据和并行计算的时间
 * 输入：无
 * 返回值：无
 */
void printResult()
{
    int i,j;
    printf("\nOutput of Matrix C = AB\n");
    for(i=0;i<M;i++)                                    /* 输出 C 矩阵的结果数据 */
    {
        for(j=0;j<N;j++) printf("%f\t",C(i,j));
        printf("\n");
    }

    endtime=MPI_Wtime();
    printf("\n");

    printf("Whole running time       = %f seconds\n",endtime-starttime);
    printf("Distribute data time    = %f seconds\n",time1-starttime);
    printf("Parallel compute time = %f seconds\n",endtime-time1);

}

/*
 * 函数名: main
 * 功能：程序的主函数
 * 输入：argc 为命令行参数个数；
 *          argv 为每个命令行参数组成的字符串数组。
 * 输出：返回 0 代表程序正常结束；其它值表明程序出错。
 */
int main(int argc, char **argv)
{
    int i,j,k,l,group_size,mp1,mm1;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&group_size);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

```
    p=group_size;

//下面一段程序负责从 dataIn.txt 文件中读入 A[M,K],B[P,N]两个相乘矩阵的数据，
//并为结果矩阵 C[M,N]分配空间。C[N,N]=A[M,K]*B[P,N]
//注意这段程序只有编号为 0 的处理器才执行此步操作
    if(myid==0)
    {
        readData();
    }

if (myid==0)
/*  由编号为 0 的进程将 A,B 两矩阵的行列维数 M,K,N 发送给所有其他进程  */
        for(i=1;i<p;i++)
    {
        MPI_Send(&M,1,MPI_INT,i,i,MPI_COMM_WORLD);
        MPI_Send(&K,1,MPI_INT,i,i,MPI_COMM_WORLD);
        MPI_Send(&N,1,MPI_INT,i,i,MPI_COMM_WORLD);
    }
else
/*  编号非 0 的进程负责接收 A,B 两矩阵的行列维数 M,K,N */
    {
        MPI_Recv(&M,1,MPI_INT,0,myid,MPI_COMM_WORLD,&status);
        MPI_Recv(&K,1,MPI_INT,0,myid,MPI_COMM_WORLD,&status);
        MPI_Recv(&N,1,MPI_INT,0,myid,MPI_COMM_WORLD,&status);
    }

    p=gcd(M,N,group_size);
m=M/p;
 /* m 代表将矩阵按行分块后每块的行数  */
n=N/p;
 /* m 代表将矩阵按列分块后每块的列数  */

    if(myid<p)
    {
        a=(float *)malloc(floatsize*m*K);
 /* a[m,K]用来存储本处理器拥有的矩阵 A 的行块  */
        b=(float *)malloc(floatsize*K*n);
/* b[K,n]用来存储此时处理器拥有的矩阵 B 的列块  */
        c=(float *)malloc(floatsize*m*N);
 /* c[m,N]用来存储本处理器计算 p-1 次得到所有结果  */

        if (myid%2!=0)
 /*  为标号为奇数的处理器分配发送缓冲空间  */
            buffer=(float *)malloc(K*n*floatsize);

        if (a==NULL||b==NULL||c==NULL)
/*  如果分配空间出错，则打印出错信息  */
            printf("Allocate space for a,b or c fail!");

        if (myid==0)
 /*  标号为 0 的处理器将应该它拥有的矩阵 A,B 的元素读入自己的 a,b 中  */
        {
            for (i=0;i<m;i++)
                for (j=0;j<K;j++)
                    a(i,j)=A(i,j);
```

```
            for (i=0;i<K;i++)
                for (j=0;j<n;j++)
                    b(i,j)=B(i,j);
        }

        if (myid==0)
 /*  标号为 0 的处理器将其他处理器的初始数据分别发给各处理器  */
        {
            for (i=1;i<p;i++)
            {
                MPI_Send(&A(m*i,0),K*m,MPI_FLOAT,i,i,MPI_COMM_WORLD);

                for (j=0;j<K;j++)
                    MPI_Send(&B(j,n*i),n,MPI_FLOAT,i,i,MPI_COMM_WORLD);
            }
            free(A);
            free(B);
/*  至此，A,B 两矩阵的数据已经完全被分散到各处理器。释放 A,B 所占空间  */
        }
        else
/*  标号非 0 的处理器从 0 处理器接受各自的初始矩阵数据  */
        {
            MPI_Recv(a,K*m,MPI_FLOAT,0,myid,MPI_COMM_WORLD,&status);

            for (j=0;j<K;j++)

MPI_Recv(&b(j,0),n,MPI_FLOAT,0,myid,MPI_COMM_WORLD,&status);
        }

        if (myid==0)
            time1=MPI_Wtime();
 /*  标号为 0 的处理器记录开始矩阵相乘计算的时间  */

        for (i=0;i<p;i++)                                   /*  一共进行 p 轮计算  */
        {
            l=(i+myid)%p;

            for (k=0;k<m;k++)
                for (j=0;j<n;j++)
                    for (c(l,k,j)=0,s=0;s<K;s++)
                        c(l,k,j)+=a(k,s)*b(s,j);

            mm1=(p+myid-1)%p;
/*  计算本进程的前一个进程的标号  */
            mp1=(myid+1)%p;
/*  计算本进程的后一个进程的标号  */

            if (i!=p-1)
            {
                if(myid%2==0)
 /*  偶数号处理器先发送后接收  */
                {
                    MPI_Send(b,K*n,MPI_FLOAT,mm1,mm1,MPI_COMM_WORLD);

MPI_Recv(b,K*n,MPI_FLOAT,mp1,myid,MPI_COMM_WORLD,&status);
                }
```

```
                    else
/*奇数号处理器先将 B 的列块存于缓冲区 buffer 中，然后接收编号在其后面的
  处理器所发送的 B 的列块，最后再将缓冲区中原矩阵 B 的列块发送给编号
  在其前面的处理器
*/
                    {
                        for(k=0;k<K;k++)
                            for(j=0;j<n;j++)
                                buffer(k,j)=b(k,j);

MPI_Recv(b,K*n,MPI_FLOAT,mp1,myid,MPI_COMM_WORLD,&status);

MPI_Send(buffer,K*n,MPI_FLOAT,mm1,mm1,MPI_COMM_WORLD);
                    }
                }
            }
            if (myid==0)
  /*  标号为 0 的进程直接将计算结果保存到结果矩阵 C 中  */
                for(i=0;i<m;i++)
                    for(j=0;j<N;j++)
                        C(i,j)=*(c+i*N+j);

            if (myid!=0)
/*  标号非 0 的进程则要把计算结果发送到标号为 0 的处理器中去  */
                MPI_Send(c,m*N,MPI_FLOAT,0,myid,MPI_COMM_WORLD);
            else
/*  标号为 0 的进程负责接收其他进程的计算结果并保存到结果矩阵 C 中  */
            {
                for(k=1;k<p;k++)
                {
                    MPI_Recv(c,m*N,MPI_FLOAT,k,k,MPI_COMM_WORLD,&status);

                    for(i=0;i<m;i++)
                        for(j=0;j<N;j++)
                            C((k*m+i),j)=*(c+i*N+j);
                }
            }
            if(myid==0)
  /* 0 号处理器负责将 A,B,C 矩阵打印输出给用户，并输出用于分发数据和并行计算的时间  */
                printResult();
        }
        MPI_Finalize();
        if(myid<p)                  /*  释放所有临时分配空间  */
        {
            free(a);
            free(b);
            free(c);
            if(myid==0)             /*  只有 0 号进程要释放 C */
                free(C);
            if(myid%2!=0)           /*  只有奇数号进程要释放 buffer */
                free(buffer);
        }
        return (0);
}
```