

- 并行程序设计 上机练习一
  - 实验一
    - 二叉树求和
    - 蝶式求和
  - 实验二
    - 题目
    - 作业1.3.1
    - 作业3.3.2
    - 作业3.5.1
    - 作业3.5.2
  - 实验三
    - 题面
    - 代码实现
    - 运行截图:
  - 实验四
    - 题面
    - 代码实现:
    - 运行截图
- 并行程序设计 上机练习二
  - 实验一(LU分解):
    - 题目
    - 代码实现
    - 运行截图:
  - 实验二(QR分解)
    - 题目
    - 代码实现
    - 运行截图
  - 实验三(summa)
    - 题目:
    - 代码实现:
    - 运行截图
  - 实验四(自选命题-随机算法并行实现)
    - 题目
    - 代码实现
    - 运行截图

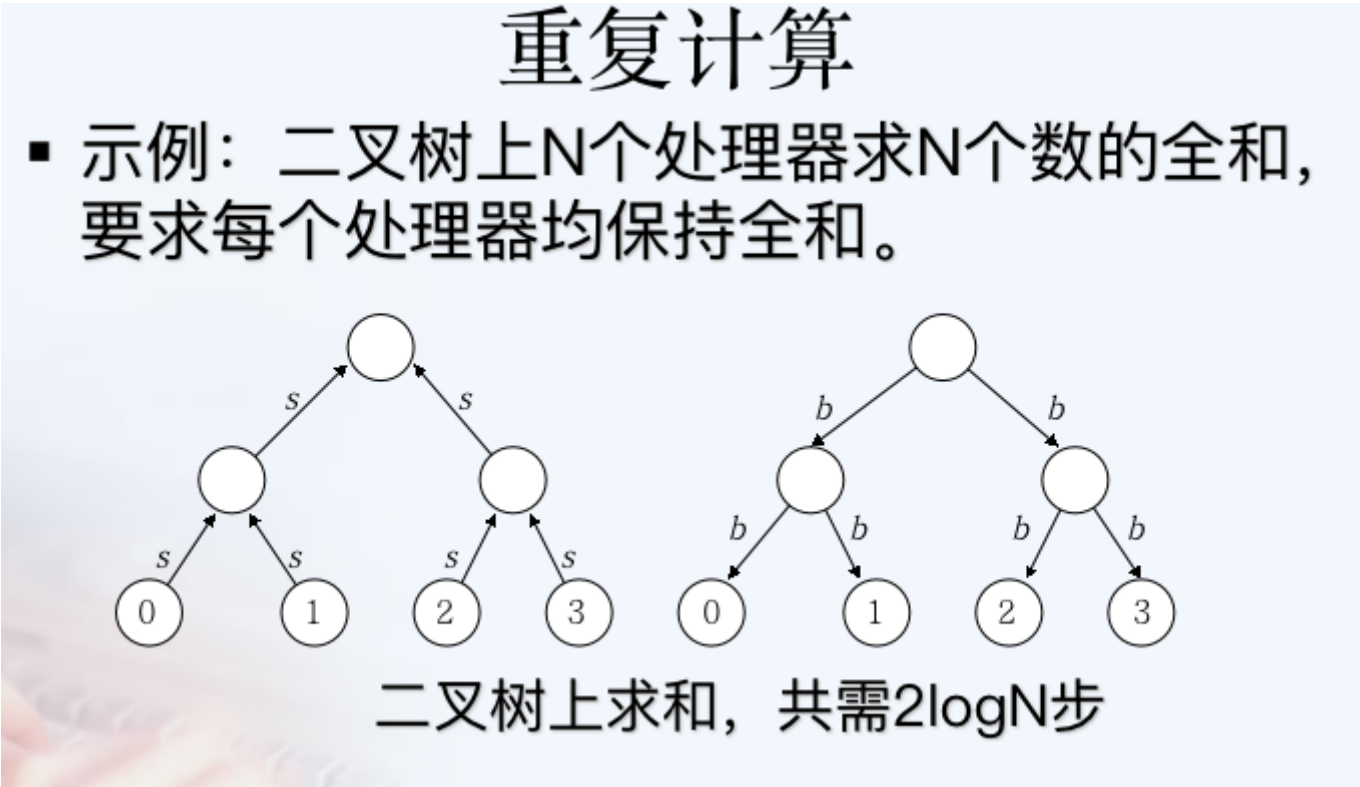
# 并行程序设计 上机练习一

## 实验一

**题目:** 课程主页lec2\_PP.ppt的P35-36两种重复计算方式的OpenMP和MPI实现。（任务数可约定为2的幂次方）

### 二叉树求和

算法流程图:



#### PCAM分析

**划分:** 域分解:每个节点均包含一个当前节点权值,均为一个域 功能分解:将任务分成三种,叶子节点,中间节点,根节点 **通信:** 使用静态的结构化异步通信 对于叶子节点,完成两次通信,第一次向其父节点发送该叶子节点的值,第二次等待父亲节点传回sum 对于中间节点,完成三次通信,第一次所有接受子节点的和,第二次等待其父节点回传sum,第三次通信发送收到的sum值给所有儿子节点 对于根节点完成两次通信,第一次通信获得所有子节点的和,第二次将计算的sum发送给所有儿子节点 **组合:** 我们将任务分成三种,因此我们对与每个节点均以自己作为一个组合(不划分)

- 对于叶子节点在第二次通信时将节点自身值赋为sum
- 对于根节点,在第二次通信前计算所有子节点传来的数值的和,并加上本身节点值
- 对于中间节点,在第一次通信以及第二次通信前计算所有子节点传来的数值的和,并加上本身节点值

**映射:** 将对应的任务分别发送给叶子节点/中间节点/根节点

代码实现:

```

#include <stdio.h>
#include <string.h>
#include <bits/stdc++.h>
#include "mpi.h"
#define GHH(...) printf(__VA_ARGS__)
// #define GHH(...)
// using namespace std;

int main(int argc, char* argv[])
{
    clock_t begin = clock();
    int numprocs, myid, source;
    MPI_Status status;
    int data[10];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    data[0] = myid;
    GHH("thread %d begin!\n", myid);
    if(myid == 0);
    else if(myid == 1){ //根节点
        MPI_Recv(data+1, 1, MPI_INT, myid*2, myid*2, //阶段1,收集来自左右子树的和
            MPI_COMM_WORLD, &status);
        GHH("root %d get sum %d from left son %d\n", myid, data[1], myid*2);

        MPI_Recv(data+2, 1, MPI_INT, myid*2+1, myid*2+1,
            MPI_COMM_WORLD, &status);
        GHH("root %d get sum %d from right son %d\n", myid, data[2], myid*2+1);

        data[0] += data[1] + data[2];

        GHH("root %d send sum %d to left son %d\n", myid, data[0], myid*2);
        MPI_Send(data, 1, MPI_INT, myid*2, myid, //阶段2,向下广播求和的值
            MPI_COMM_WORLD);

        GHH("root %d send sum %d to right son %d\n", myid, data[0], myid*2+1);
        MPI_Send(data, 1, MPI_INT, myid*2+1, myid, //阶段2,向下广播求和的值
            MPI_COMM_WORLD);
    } else if (myid >= (numprocs+1)/2){ //叶子结点
        MPI_Send(data, 1, MPI_INT, myid/2, myid, //阶段1,向上传递自身节点值
            MPI_COMM_WORLD);
        GHH("leaf %d send itself to parent %d\n", myid, myid/2);

        MPI_Recv(data, 1, MPI_INT, myid/2, myid/2, //阶段2,收到来自根节点的全局和
            MPI_COMM_WORLD, &status);
        GHH("leaf %d get sum %d from parent %d\n", myid, data[0], myid/2);
    } else { //中间节点
        MPI_Recv(data+1, 1, MPI_INT, myid*2, myid*2, //阶段1,收集来自左右子树的和
            MPI_COMM_WORLD, &status);
        GHH("midnode %d get sum %d from left son %d\n", myid, data[1], myid*2);
    }
}

```

```

MPI_Recv(data+2, 1, MPI_INT, myid*2+1, myid*2+1,
          MPI_COMM_WORLD, &status);
GHH("midnode %d get sum %d from right son %d\n",myid,data[2],myid*2+1);

data[0]+=data[1]+data[2];

MPI_Send(data, 1, MPI_INT, myid/2, myid,      //向其父亲节点发送该节点左右子树
以及自己的和
          MPI_COMM_WORLD);
GHH("midnode %d send itself to parent %d\n",myid,myid/2);

MPI_Recv(data, 1, MPI_INT, myid/2, myid/2,      //阶段2,向下广播自根节点的全局
和
          MPI_COMM_WORLD, &status);
GHH("midnode %d get sum %d from parent %d\n",myid,data[0],myid/2);

MPI_Send(data, 1, MPI_INT, myid*2, myid,
          MPI_COMM_WORLD);
GHH("midnode %d send sum %d to left son %d\n",myid,data[0],myid*2);

MPI_Send(data, 1, MPI_INT, myid*2+1, myid,
          MPI_COMM_WORLD);
GHH("midnode %d send sum %d to right son %d\n",myid,data[0],myid*2+1);

}
MPI_Finalize();

clock_t end = clock();
printf("thread %d sum is %d. Runtime :%lf(ms)\n",myid,data[0],1000.0*(end-
begin)/CLOCKS_PER_SEC);
} /* end main */

// mpic++ -o fun test.cpp -fopenmp > compile.log
// mpirun -n 8 -genv OMP_NUM_THREADS 1 ./fun > out.log

```

## 运行结果展示

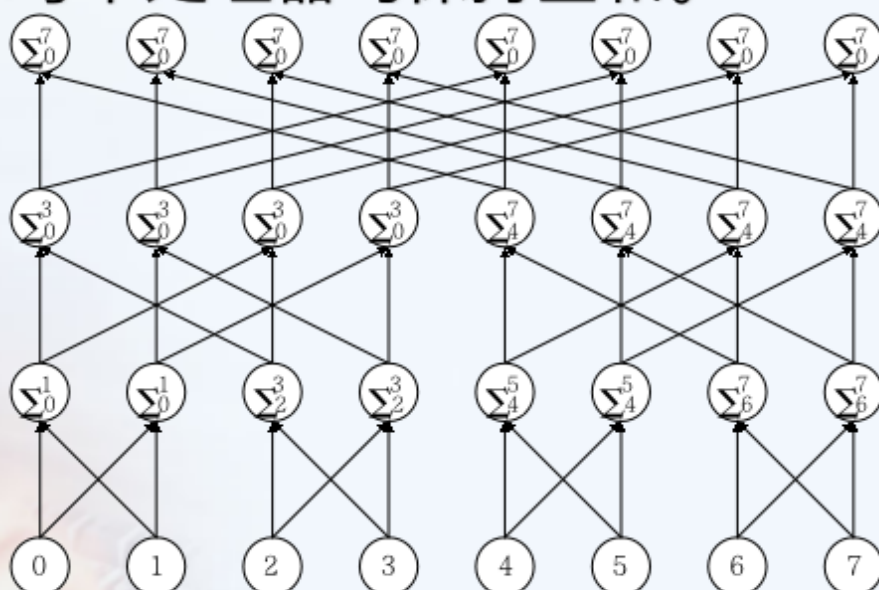
```
1 thread 0 begin!
2 thread 2 begin!
3 thread 3 begin!
4 thread 7 begin!
5 leaf 7 send itself to parent 3
6 thread 1 begin!
7 root 1 get sum 11 from left son 2
8 root 1 get sum 16 from right son 3
9 midnode 2 get sum 4 from left son 4
10 midnode 2 get sum 5 from right son 5
11 midnode 2 send itself to parent 1
12 midnode 2 get sum 28 from parent 1
13 midnode 2 send sum 28 to left son 4
14 midnode 2 send sum 28 to right son 5
15 midnode 3 get sum 6 from left son 6
16 midnode 3 get sum 7 from right son 7
17 midnode 3 send itself to parent 1
18 midnode 3 get sum 28 from parent 1
19 midnode 3 send sum 28 to left son 6
20 midnode 3 send sum 28 to right son 7
21 thread 4 begin!
22 leaf 4 send itself to parent 2
23 leaf 4 get sum 28 from parent 2
24 thread 5 begin!
25 leaf 5 send itself to parent 2
26 leaf 5 get sum 28 from parent 2
27 thread 6 begin!
28 leaf 6 send itself to parent 3
29 leaf 6 get sum 28 from parent 3
30 root 1 send sum 28 to left son 2
31 root 1 send sum 28 to right son 3
32 leaf 7 get sum 28 from parent 3
33 thread 0 sum is 0. Runtime :70.020000(ms)
34 thread 7 sum is 28. Runtime :77.778000(ms)
35 thread 2 sum is 28. Runtime :63.924000(ms)
36 thread 3 sum is 28. Runtime :67.978000(ms)
37 thread 4 sum is 28. Runtime :75.195000(ms)
38 thread 5 sum is 28. Runtime :70.540000(ms)
39 thread 6 sum is 28. Runtime :59.505000(ms)
40 thread 1 sum is 28. Runtime :68.156000(ms)
```

## 蝶式求和

### 算法流程图

# 重复计算

- 示例：二叉树上N个处理器求N个数的全和，要求每个处理器均保持全和。



蝶式结构求和，使用了重复计算，共需 $\log N$ 步

## PCAM分析

**划分:** 域分解:每个计算节点均等价,因此每个计算节点本身就是一个划分 功能分解:每个计算节点均等价,因此每个计算节点功能一致,仅有一个划分 **通信:** 使用动态的非结构化同步通信 使用同步的方式,记录当前节点id为 $i$ (从0到 $n-1$ ),第 $k$ 轮将节点 $i$ 给id为 $i \oplus 2^k$  ( $\oplus$ 表示异或)的节点发送当前节点sum **组合:** 由于任务只有一种,因此我们仅需要叙述每个节点对应的步骤:

- 首先使用同步的通信方式,所有节点按轮完成其对应的数据传输任务(保证数据一致性)
- 对于第 $k$ 轮将节点 $i$ 给id为 $i \oplus 2^k$  ( $\oplus$ 表示异或)的节点发送当前节点sum
- 保证每个节点均接受到该轮传输的对应 $sum_{old}$ 之后,更新本身节点值 $sum = sum + sum_{old}$

**映射:** 将上述组合后的任务分发至所有节点即可

## 代码实现

```
#include <stdio.h>
#include <string.h>
#include <bits/stdc++.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
    clock_t begin = clock();
    int numprocs, myid, source;
    MPI_Status status;
    int data[10];
    MPI_Init(&argc, &argv);
```

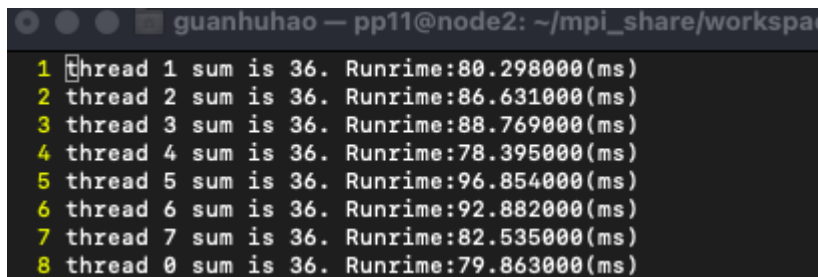
```

MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
data[0]=myid+1;
for(int i = 1; i < numprocs ; i<=1){
    MPI_Send(data,1,MPI_INT,myid^i,myid^i,MPI_COMM_WORLD);
    MPI_Recv(data+1,1,MPI_INT,myid^i,myid,MPI_COMM_WORLD,&status);
    data[0] += data[1];
}
MPI_Finalize();
clock_t end = clock();
printf("thread %d sum is %d. Runrime:%lf(ms)\n",myid,data[0],1000.0*(end-
begin)/CLOCKS_PER_SEC);
} /* end main */

// mpic++ -o fun test.cpp -fopenmp > compile.log
// mpirun -n 8 -genv OMP_NUM_THREADS 1 ./fun > out.log

```

### 运行结果展示



```

1 thread 1 sum is 36. Runrime:80.298000(ms)
2 thread 2 sum is 36. Runrime:86.631000(ms)
3 thread 3 sum is 36. Runrime:88.769000(ms)
4 thread 4 sum is 36. Runrime:78.395000(ms)
5 thread 5 sum is 36. Runrime:96.854000(ms)
6 thread 6 sum is 36. Runrime:92.882000(ms)
7 thread 7 sum is 36. Runrime:82.535000(ms)
8 thread 0 sum is 36. Runrime:79.863000(ms)

```

## 实验二

### 题目

前期练习作业题目中的相关程序实现（ex-21-1/2/3 中有要求向量化/并行化的程序实习）。

### 作业1.3.1

题面:

### 3. 向量化以下循环。如果不能，请说明原因。

```

(1) for I = 1 to N do
    S:A(I) = B(I) + C(I+1);
    T:C(I) = A(I)* D(I);
end for

```

```

S:A(1,N) = B(1:N) + C(2:N+1);
T:C(1:N) = A(1:N) * D(1:N);

```

```
(2) for I = 1 to N do
    S:A(I) = A(I-1) + 1
end for
```

存在依赖 $S\delta^f S$ 方向向量为(1)因此不能并行化

代码实现:

```
#include <bits/stdc++.h>
#include "mpi.h"
using namespace std;
const int n = 1000000;
int a[n+10],b[n+10],c[n+10],d[n+10];
int aa[n+10],bb[n+10],cc[n+10],dd[n+10];
void chuan(){
    clock_t begin = clock();
    for(int i =1;i<=n;i++){
        aa[i]=bb[i]+cc[i+1];
    }
    for(int i=1;i<=n;i++){
        cc[i]=aa[i]*dd[i];
    }
    clock_t end = clock();
    printf("串行用时:%5lf(ms)\n",1000.0*(end-begin)/CLOCKS_PER_SEC);
}
void check(){
    for(int i=1;i<=n;i++){
        if(aa[i]!=a[i]||cc[i]!=c[i]) cout<<"error!"<<endl;
    }
}
int main(int argc, char* argv[])
{
    for(int i=0;i<n+5;i++){
        a[i]=aa[i]=rand()%100;
        b[i]=bb[i]=rand()%100;
        c[i]=cc[i]=rand()%100;
        d[i]=dd[i]=rand()%100;
    }
    chuan();

    clock_t begin = clock();
    #pragma omp parallel for
    for(int i =1;i<=n;i++){
        a[i]=b[i]+c[i+1];
    }
    #pragma omp parallel for
    for(int i=1;i<=n;i++){
        c[i]=a[i]*d[i];
    }
}
```



```

    }
    clock_t end = clock();
    printf("并行用时:%51f(ms)\n", 1000.0*(end-begin)/CLOCKS_PER_SEC);

    check();
} /* end main */

```

## 运行结果展示

当OMP\_NUM\_THREADS 设置为2时运行截图如下:

```

guanhuhao — pp11@node2: ~/mpi_
1 串行程序与并行程序结果一致!
2 并行用时:5.511000(ms)
3 串行用时:5.652000(ms)
4 加速比为:1.025585

```

当

OMP\_NUM\_THREADS 设置为4时运行截图如下:

```

guanhuhao — pp11@node2: ~/mpi_shar
1 串行程序与并行程序结果一致!
2 并行用时:1.917000(ms)
3 串行用时:7.097000(ms)
4 加速比为:3.702139

```

当

OMP\_NUM\_THREADS 设置为8时运行截图如下:

```

guanhuhao — pp11@node2: ~/mp
1 串行程序与并行程序结果一致!
2 并行用时:7.220000(ms)
3 串行用时:5.568000(ms)
4 加速比为:0.771191

```

因此我们可以得出,需要设置合适的OMP\_NUM\_THREADS才能发挥并行的优势

## 作业3.3.2

### 题面

(2) 尝试向量化/并行化此循环。

```

for i = 1 to 100 do // 循环 2 N 是常量
    for j = 1 to 100 do
        S2:B[j] = A[j, N]; // 语句 S2
        doall k = 1 to 100 do
            S3:A[j+1, k] = B[j] + C[j, k]; // 语句 S3
        enddoall // loop-k
        S4:Y[i+j] = A[j+1, N]; // 语句 S4
    endfor // loop-j
endfor // loop-i

doall i = 1 to 100 do
    S1:X[i] = Y[i] + 10; // 语句 S1
enddoall

```

### 代码实现

```

#include <bits/stdc++.h>
#include "mpi.h"
using namespace std;
const int n = 100;
int a[n+10][n+10], b[n+10], c[n+10][n+10], d[n+10];
int aa[n+10][n+10], bb[n+10], cc[n+10][n+10], dd[n+10];
int y[n*2+10], x[n*2+10];
int yy[n*2+10], xx[n*2+10];
double chuan(){ //串行程序计算
    clock_t begin = clock();
    for(int i = 1; i <= n; i++){
        xx[i] = yy[i] + 10;
        for(int j = 1; j <= n; j++){
            bb[j] = aa[j][n];
            for(int k = 1; k <= n; k++){
                aa[j+1][k] = bb[j] + cc[j][k];
            }
            yy[i+j] = aa[j+1][n];
        }
    }
    clock_t end = clock();
    // printf("串行用时:%5lf(ms)\n", 1000.0*(end-begin)/CLOCKS_PER_SEC);
    return 1000.0*(end-begin)/CLOCKS_PER_SEC;
}
bool check(){ //检查是否与串行程序相同
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= n; j++){
            if(aa[i][j] != a[i][j] || cc[i][j] != c[i][j]) {
                cout << "error!" << endl;
                return false;
            }
        }
        if(b[i] != bb[i] || d[i] != dd[i] || x[i] != xx[i] || y[i] != yy[i]) {
            cout << "error" << endl;
            return false;
        }
    }
    return true;
}
void randData(){ //生成随机数据
    for(int i = 0; i < n+5; i++){
        for(int j = 0; j < n+5; j++) {
            a[i][j] = aa[i][j] = rand()%100;
            c[i][j] = cc[i][j] = rand()%100;
        }
        b[i] = bb[i] = rand()%100;
        d[i] = dd[i] = rand()%100;
        x[i] = xx[i] = rand()%100;
        y[i] = yy[i] = rand()%100;
    }
}
int main(int argc, char* argv[])
{
    randData(); //生成随机数据

```

```

double time_seq = chuan();

clock_t begin = clock();
for(int i =1;i<=n;i++){
    for(int j = 1;j<=n;j++){
        b[j]=a[j][n];
        // #pragma omp parallel for
        for(int k =1;k<=n;k++){
            a[j+1][k]=b[j]+c[j][k];
        }
        y[i+j] = a[j+1][n];
    }
}


#pragma omp parallel for
for(int i=1;i<=n;i++){
    x[i]=y[i]+10;
}
clock_t end = clock();

if(check()) printf("串程序与并程序结果一致!\n");
double time_pra = 1000.0*(end-begin)/CLOCKS_PER_SEC;
printf("并行用时:%51f(ms)\n",time_pra);
printf("串行用时:%51f(ms)\n",time_seq);
printf("加速比为:%1f\n",time_seq/time_pra);

} /* end main */

```

### 运行结果展示



```

guanhuhao — pp11@node2: ~/mpi_
1 串程序与并程序结果一致！
2 并行用时:5.358000(ms)
3 串行用时:6.580000(ms)
4 加速比为:1.228070

```

### 作业3.5.1

#### 题面

五、分析以下 3 个循环中存在的依赖关系；分别通过循环交换、分布和逆转 等多种方法来尝试向量化和/或并行化变换：

```

for i = 1 to 100 do //循环 1
    S:A[i] = A[i] + B[i-1];
    T:B[i] = C[i-1] * 2 ;
    U:C[i] = 1 / B[i] ;
    V:D[i] = C[i] * C[i] ;
endfor

```

## 代码实现:

```
#include <bits/stdc++.h>
#include "mpi.h"
using namespace std;
const int n = 1000000;
const int mod = 1e4;
int a[n+10],b[n+10],c[n+10],d[n+10];
int aa[n+10],bb[n+10],cc[n+10],dd[n+10];
double chuan(){ //串行程序计算
    clock_t begin = clock();
    for(int i=1;i<=n;i++){
        aa[i]=aa[i]+bb[i-1];
        bb[i]=cc[i-1]*2%mod+1;
        cc[i]=int(1/bb[i]);
        dd[i]=cc[i]*cc[i]%mod+1;
    }
    clock_t end = clock();
    // printf("串行用时:%5lf(ms)\n",1000.0*(end-begin)/CLOCKS_PER_SEC);
    return 1000.0*(end-begin)/CLOCKS_PER_SEC;
}
bool check(){
    for(int i=1;i<=n;i++){
        if(aa[i]!=a[i]||cc[i]!=c[i]||bb[i]!=b[i]||dd[i]!=d[i]) {
            cout<<"error!"<<endl;
            return false;
        }
    }
    return true;
}
void randData(){ //生成随机数据
    for(int i=0;i<n+5;i++){
        a[i]=aa[i]=rand()%100;
        b[i]=bb[i]=rand()%100;
        c[i]=cc[i]=rand()%100;
        d[i]=dd[i]=rand()%100;
    }
}
int main(int argc, char* argv[])
{
    randData(); //生成随机数据

    double time_seq = chuan(); //串行程序计算

    clock_t begin = clock();
    for(int i =1;i<=n;i++){
        b[i]=c[i-1]*2%mod+1;
        c[i]=int(1/b[i]);
    }
    #pragma omp parallel for
    for(int i=1;i<=n;i++){
```

```


        a[i]=a[i]+b[i-1];
        d[i]=c[i]*c[i]%mod+1;
    }
    clock_t end = clock();

    if(check()) printf("串行程序与并行程序结果一致!\n");
    double time_pra = 1000.0*(end-begin)/CLOCKS_PER_SEC;
    printf("并行用时:%51f(ms)\n",time_pra);
    printf("串行用时:%51f(ms)\n",time_seq);
    printf("加速比为:%1f\n",time_seq/time_pra);

} /* end main */

```

### 运行结果展示



```

guanhuhao — pp11@node2: ~/mpi_share
1 串行程序与并行程序结果一致！
2 并行用时:2955.580000(ms)
3 串行用时:3374.459000(ms)
4 加速比为:1.141725

```

由于该算法比较简单,在数据规模较小时无论怎么设置

### 作业3.5.2

#### 题面

```

for i = 1 to 999 do // 循环 2
    S:A[i] = B[i] + C[i];
    T:D[i] = ( A[i] + A[ 999-i+1 ] ) / 2 ;
endfor

```

#### 代码实现

```

#include <bits/stdc++.h>
#include "mpi.h"
using namespace std;
const int n = 99999;
const int mod = 1e4;
int a[n+10],b[n+10],c[n+10],d[n+10];
int aa[n+10],bb[n+10],cc[n+10],dd[n+10];
double chuan(){
    clock_t begin = clock();
    for(int i=1;i<=n;i++){
        aa[i]=bb[i]+cc[i];
        dd[i]=(aa[i]+aa[n-i+1])/2;
    }
    clock_t end = clock();
}

```

```
// printf("串行用时:%5lf(ms)\n",1000.0*(end-begin)/CLOCKS_PER_SEC);
return 1000.0*(end-begin)/CLOCKS_PER_SEC;
}
bool check(){
    for(int i=1;i<=n;i++){
        if(aa[i]!=a[i]||cc[i]!=c[i]||bb[i]!=b[i]||dd[i]!=d[i]) {
            cout<<i<<" "<<"error!"<<endl;
            return false;
        }
    }
    return true;
}
int main(int argc, char* argv[])
{

    for(int i=0;i<n+5;i++){
        a[i]=aa[i]=rand()%100;
        b[i]=bb[i]=rand()%100;
        c[i]=cc[i]=rand()%100;
        d[i]=dd[i]=rand()%100;
    }

    double time_seq = chuan();

    clock_t begin = clock();
    #pragma omp parallel for
    for(int i =1;i<=(n+1)/2;i++){
        a[i]=b[i]+c[i];
        // d[i]=(a[i]+a[n-i+1])/2;
    }
    #pragma omp parallel for
    for(int i =1;i<=(n+1)/2;i++){
        d[i]=(a[i]+a[n-i+1])/2;
    }
    #pragma omp parallel for
    for(int i=(n+1)/2+1;i<=n;i++){
        a[i]=b[i]+c[i];
        // d[i]=(a[i]+a[n-i+1])/2;
    }
    #pragma omp parallel for
    for(int i=(n+1)/2+1;i<=n;i++){
        d[i]=(a[i]+a[n-i+1])/2;
    }
    clock_t end = clock();

    if(check()) printf("串程序与并程序结果一致!\n");
    double time_pra = 1000.0*(end-begin)/CLOCKS_PER_SEC;
    printf("并行用时:%5lf(ms)\n",time_pra);
    printf("串行用时:%5lf(ms)\n",time_seq);
    printf("加速比为:%1f\n",time_seq/time_pra);
} /* end main */
```

## 运行结果展示



```
guanhuhao — pp11@node2: ~/mpi_s
1 串行程序与并行程序结果一致!
2 并行用时:0.341000(ms)
3 串行用时:0.582000(ms)
4 加速比为:1.706745
```

## 实验三

### 题面

新的广播MyBcastMPI实现。基本思路：（1）将MPI进程按所在节点划分子通讯域N；（2）可以将各子通讯域的首进程（编号为0）再组成一个子通讯域H；（3）由广播的root进程将消息发给原来最大通讯域中的0号进程h，再由h在H通讯域中广播（MPI\_Bcast），各首进程然后在各自子通讯域N中再行广播（MPI\_Bcast）。

### 代码实现

```
// 本实验通信域大小为4,其中所有myid%4=0的构成通信域H
// root节点为id最大的,因此要求numprocs应该为4k+1
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int data[10];
int main(int argc, char* argv[])
{
    int numprocs, myid, source;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

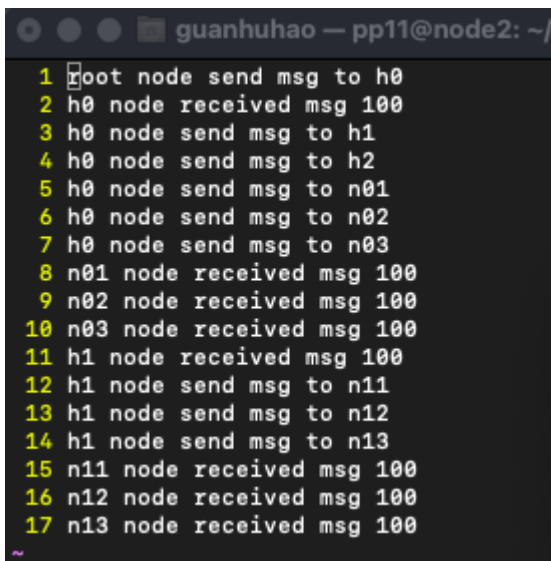
    if(myid == numprocs-1){ //为root节点
        data[0]=100;
        printf("root node send msg to h0\n");
        MPI_Send(data, 1, MPI_INT, 0, 0,MPI_COMM_WORLD);
    }else if(myid%4 == 0){ //设定myid%4==0的节点构成子通信域N
        if(myid==0){ //myid=0的为0号进程
            MPI_Recv(data, 1, MPI_INT, numprocs-1, myid,
                MPI_COMM_WORLD, &status);
            printf("h0 node received msg %d\n",data[0]);
            for(int i = myid+4; i <= numprocs-1 ; i+=4){
                printf("h0 node send msg to h%d\n",i/4);
                MPI_Send(data, 1, MPI_INT, i, i,MPI_COMM_WORLD);
            }
        }else{
            MPI_Recv(data, 1, MPI_INT, 0, myid,
                MPI_COMM_WORLD, &status);
            printf("h%d node received msg %d\n",myid/4,data[0]);
        }
        for(int i = 1; i <= 3; i++){
```

```

        printf("h%d node send msg to n%d%d\n",myid/4,myid/4,i);
        MPI_Send(data, 1, MPI_INT, myid+i, myid+i,
                 MPI_COMM_WORLD);
    }
} else {           //通信域N收到消息
    MPI_Recv(data, 1, MPI_INT, myid-myid%4, myid,
             MPI_COMM_WORLD, &status);
    printf("n%d%d node received msg %d\n",myid/4,myid%4,data[0]);
}
MPI_Finalize();
} /* end main */

```

### 运行截图:



```

guanhuhao — pp11@node2: ~/
1 root node send msg to h0
2 h0 node received msg 100
3 h0 node send msg to h1
4 h0 node send msg to h2
5 h0 node send msg to n01
6 h0 node send msg to n02
7 h0 node send msg to n03
8 n01 node received msg 100
9 n02 node received msg 100
10 n03 node received msg 100
11 h1 node received msg 100
12 h1 node send msg to n11
13 h1 node send msg to n12
14 h1 node send msg to n13
15 n11 node received msg 100
16 n12 node received msg 100
17 n13 node received msg 100

```

## 实验四

### 题面

用MPI\_Send和MPI\_Recv来模拟实现诸如MPI\_Alltoall, MPI\_Allgather功能并与标准MPI实现做简要性能对比

### 代码实现:

```

// 本实验通信域大小为4,其中所有myid%4=0的构成通信域H
// root节点为id最大的,因此要求numprocs应该为4k+1
#include <stdio.h>
#include <string.h>
#include <bits/stdc++.h>
#include "mpi.h"
using namespace std;
int data[1050];
int recv[1050];
void my_alltoall(const void *sendbuf, const int sendcount, MPI_Datatype sendtype,
                void *recvbuf, const int recvcount, MPI_Datatype recvtype, MPI_Comm comm, int
numprocs, int myid){
    int offset = 0;

```



```

    MPI_Status status;
    // printf("numprocs: %d\n",numprocs);
    for(int i=0;i<numprocs;i++) {
        if(i==myid) continue;
        MPI_Send((int*)(sendbuf)+i*sendcount, sendcount, sendtype, i, 1,
comm);
        // printf("thread %d send msg to thread %d\n",myid,i);
    }
    for(int i=0;i<numprocs;i++) {
        if(i==myid) continue;
        MPI_Recv((int*)(recvbuf)+i*recvcount, recvcount, recvtype, i, 1, comm,
&status);
    }

    // printf("thread %d recived data :\n",myid);
    // for(int i=0;i<numprocs*recvcount;i++){
    //     printf("%d ",*((int*)(recvbuf)+i*recvcount));
    // }
    // printf("\n");
}

void my_allgather(const void *sendbuf, const int sendcount, MPI_Datatype sendtype,
void *recvbuf,const int recvcount, MPI_Datatype recvtype, MPI_Comm comm,int
numprocs,int myid){
    int offset = 0;
    MPI_Status status;
    for(int i=0;i<numprocs;i++) {
        if(i==myid) continue;
        MPI_Send((int*)sendbuf, sendcount, sendtype, i, 1, comm);
    }
    for(int i=0;i<numprocs;i++) {
        if(i==myid) continue;
        MPI_Recv((int*)(recvbuf)+i*recvcount, recvcount, recvtype, i, 1, comm,
&status);
    }

    // printf("thread %d recived data :\n",myid);
    // for(int i=0;i<numprocs*recvcount;i++){
    //     printf("%d ",*((int*)(recvbuf)+i*recvcount));
    // }
    // printf("\n");
}

int main(int argc, char* argv[])
{
    int numprocs, myid, source;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    for(int i=0;i<10;i++) data[i]=myid*10+i;

    clock_t begin = clock();
    MPI_Alltoall(data,1,MPI_INT,recv,1,MPI_INT,MPI_COMM_WORLD);
    clock_t end = clock();
    if(myid == 0) printf("MPI_Alltoall 用时%.3lf(ms)\n",1000.0*(end-

```

```

begin)/CLOCKS_PER_SEC);

    begin = clock();
    my_alltoall(data,1,MPI_INT,recv,1,MPI_INT,MPI_COMM_WORLD,numprocs,myid);
    end = clock();
    if(myid == 0) printf("my_alltoall 用时%.3lf(ms)\n",1000.0*(end-
begin)/CLOCKS_PER_SEC);

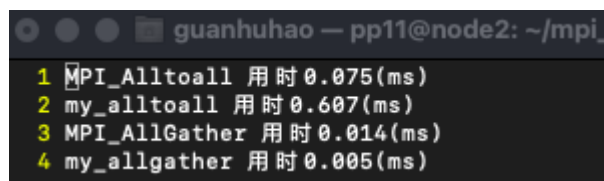
    begin = clock();
    MPI_Allgather(data,1,MPI_INT,recv,1,MPI_INT,MPI_COMM_WORLD);
    end = clock();
    if(myid == 0) printf("MPI_AllGather 用时%.3lf(ms)\n",1000.0*(end-
begin)/CLOCKS_PER_SEC);

    begin = clock();
    my_allgather(data,1,MPI_INT,recv,1,MPI_INT,MPI_COMM_WORLD,numprocs,myid);
    end = clock();
    if(myid == 0) printf("my_allgather 用时%.3lf(ms)\n",1000.0*(end-
begin)/CLOCKS_PER_SEC);

    MPI_Finalize();
} /* end main */

```

## 运行截图



```

guanhuhao — pp11@node2: ~/mpi_
1 MPI_Alltoall 用时0.075(ms)
2 my_alltoall 用时0.607(ms)
3 MPI_AllGather 用时0.014(ms)
4 my_allgather 用时0.005(ms)

```

虽然我们写的alltoall以及allgather运行速度比标准mpi的快,但是主要考虑到我们没有做必要的类型检查以及类似的程序鲁棒性保障,对于死锁等问题也仅采用简单控制,因此总体上来说并没有标准mpi完备.

## 并行程序设计 上机练习二

### 实验一(LU分解):

#### 题目

在教材中 18.5 节 LU 分解的并行 MPI 实现基础上, 给出 MPI+OpenMP混合实现

#### 代码实现

由于源代码太长了,因此此处并不放源代码仅对重要步骤进行说明: 由于在教材给出的LU分解,因此对于原始算法并没有太大的改动,主要是用OpenMPI进行局部并行化,这里我们主要观察下面代码:

```

for(i=1;i<p;i++){
    for(j=0;j<m;j++){
        MPI_Recv(&a(j,0),M,MPI_FLOAT,i,j,MPI_COMM_WORLD,&status);
    }
}

```

```

        for(k=0;k<M;k++)
            A((j*p+i),k)=a(j,k);
    }
}

```

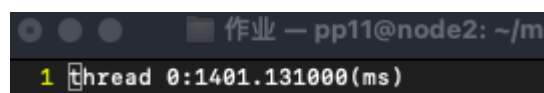
这里我们发现对于最内存循环k,没有存在依赖关系,具有并行基础,因此此处我们可以使用openMPI进行并行化,因此类似这处地方我们采用下面的方式进行优化:

```

for(i=1;i<p;i++){
    for(j=0;j<m;j++){
        MPI_Recv(&a(j,0),M,MPI_FLOAT,i,j,MPI_COMM_WORLD,&status);
        #pragma omp parallel for private(i,j,k) shared(A,a,p)
        for(k=0;k<M;k++)
            A((j*p+i),k)=a(j,k);
    }
}

```

运行截图:

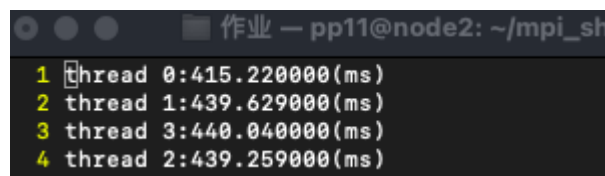


```

1 thread 0:1401.131000(ms)

```

上图是算法在单核单线程的情况下运行的结果

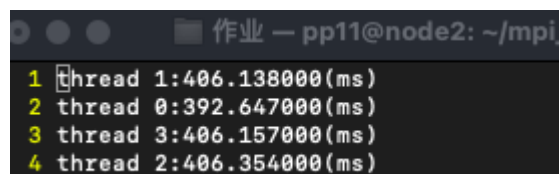


```

1 thread 0:415.220000(ms)
2 thread 1:439.629000(ms)
3 thread 3:440.040000(ms)
4 thread 2:439.259000(ms)

```

上图是算法在4核单线程的情况下运行的结果



```

1 thread 1:406.138000(ms)
2 thread 0:392.647000(ms)
3 thread 3:406.157000(ms)
4 thread 2:406.354000(ms)

```

上图是算法在4核4线程的情况下运行的结果

我们可以得出使用多核进行计算时,可以极大提升计算性能,在此基础上使用omp可以进一步小幅优化,但是幅度并不明显(实际实验中本人更换参数,发现即使使用不同参数omp起到的作用都不明显)

## 实验二(QR分解)

### 题目

针对教材中 18.6 节 QR 分解, 给出纯 OpenMP 实现

### 代码实现

整体代码同样不短,因此这里我们仅对重要部分进行摘取:

```

#pragma omp parallel shared(Q, A, condition) private(my_thread, total_thread,
base, block)

```

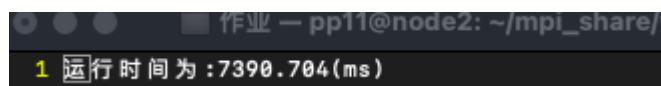
```

{
    my_thread = omp_get_thread_num();
    total_thread = omp_get_num_threads();
    block = N / total_thread;
    base = my_thread * block;
    for (int j = 0; j < base + block; ++j)
    {
        if (j < base)
        {
            wait(j, base, condition);
            transform(j, base, base + block);
        }
        else
        {
            transform(j, j + 1, base + block);
        }
        condition[j] = base + block;
    }
}

```

上面代码为已经使用omp优化后的代码,主要的思路在于内循环j没有依赖关系,可以使用omp进行并行执行分块的求解

### 运行截图

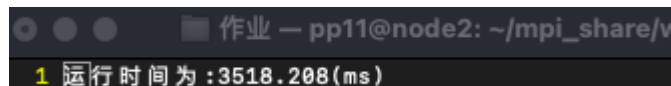


```

1 运行时间为 :7390.704(ms)

```

上述为单核单线程的运行截图

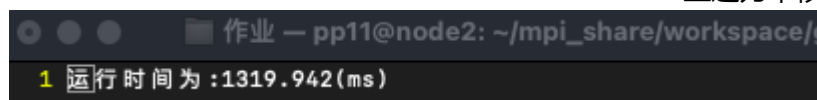


```

1 运行时间为 :3518.208(ms)

```

上述为单核8线程的运行截图



```

1 运行时间为 :1319.942(ms)

```

上述为单核16线程的运行截图

因此我们不难看出随着开辟的线程数变多,性能也有一定的提升,尽管提升并不是线性的

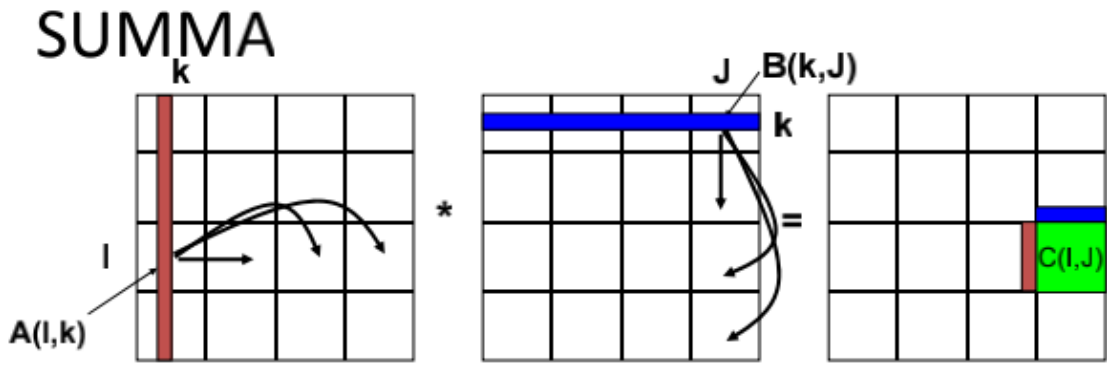
### 实验三(summa)

#### 题目:

SUMMA 并行矩阵乘法的 MPI 实现。参看文件 summa\_2010.pdf 和原始文章 lawn96.pdf。

#### 代码实现:

同样由于代码过长,因此简单简述主要算法思想(主要借用ppt)



- 假设处理器拓扑为 $P_r \times P_c$ 的二维mesh网格, (图例中:  $p_r = p_c = 4$ ), 无需是平方数。这样, 有 $P_r$ 个处理器行,  $P_c$ 个处理器列; 每个处理器有编号  $(pos\_I, pos\_J)$ 。
- 矩阵A和B, 以及结果矩阵C, 均按照 $P_r \times P_c$ 划分为行列块。 (简单起见, 假设矩阵行列数可被 $P_r$ 或 $P_c$ 整除。)
- $I, J$  表示为分块子矩阵的行列分块号; 则, 分块子矩阵 $C(I,J)$ 定义为  

$$C(I,J) = C(I,J) + \sum_k A(I,k) * B(k,J)$$

因此其可以表示为下面伪代码的形式

```

作业 — pp11@node2: ~/mpi_share/workspace/gSA21011162 — ssh pp11@202.38.86.23
1 rank: 0, sendA time:9.28187, sendB time:12.1403, calC time:8044.55, recvC time:9.46617 ms
    
```

```

对所有进程(pos_I,pos_J), 执行:
for( k = 0; k<L, k++ ) {
    所有的处理器行I : A(I,k)的所有者在该处理器行内广播A子块中的该列;
    所有的处理器列J : B(k,J)的所有者在该处理器列内广播B子块中的该行;
    接收来自处理器行pos_I的A(pos_I,k)到Acol;
    接收来自处理器列pos_J的B(k,pos_J)到Brow;
    C(pos_I,pos_J) = C(pos_I,pos_J) + Acol * Brow;
}
    
```

### 运行截图

```

作业 — pp11@node2: ~/mpi_share/workspace/gSA21011162 — ssh pp11@202.38.86.23
1 rank: 0, sendA time:9.28187, sendB time:12.1403, calC time:8044.55, recvC time:9.46617 ms
    
```

上图为使用单核情况下程序对于1024\*1024问题求解的运行截图

```

作业 — pp11@node2: ~/mpi_share/workspace/gSA21011162 — ssh pp11@202.38.86.23
1 rank: 0, sendA time:5.05638, sendB time:5.73134, calC time:2018.68, recvC time:8.77213 ms
2 rank: 2, sendA time:15.8782, sendB time:6.32668, calC time:2018.68, recvC time:8.77333 ms
3 rank: 3, sendA time:16.6879, sendB time:5.51128, calC time:2018.68, recvC time:8.7688 ms
4 rank: 1, sendA time:15.0909, sendB time:7.13706, calC time:2018.68, recvC time:8.77309 ms
    
```

上图为使用4核情况下对相同规模任务求的运行时间

```

作业 — pp11@node2: ~/mpi_share/workspace/gSA21011162 — ssh pp11@202.38.86.233
1 rank: 4, sendA time:90.1885, sendB time:107.276, calc time:950.506, recvC time:33.8299 ms
2 rank: 6, sendA time:109.731, sendB time:103.599, calc time:950.533, recvC time:33.8023 ms
3 rank: 8, sendA time:107.334, sendB time:90.3277, calc time:966.102, recvC time:33.833 ms
4 rank: 10, sendA time:107.967, sendB time:89.7627, calc time:966.111, recvC time:33.8233 ms
5 rank: 13, sendA time:110.273, sendB time:103.03, calc time:950.508, recvC time:33.8297 ms
6 rank: 15, sendA time:149.302, sendB time:63.6737, calc time:950.535, recvC time:33.8027 ms
7 rank: 0, sendA time:121.513, sendB time:64.3504, calc time:966.114, recvC time:33.8311 ms
8 rank: 2, sendA time:105.326, sendB time:92.0231, calc time:966.11, recvC time:33.8335 ms
9 rank: 3, sendA time:105.925, sendB time:91.8021, calc time:979.984, recvC time:19.9599 ms
10 rank: 5, sendA time:106.211, sendB time:119.139, calc time:938.538, recvC time:33.7934 ms
11 rank: 7, sendA time:109.551, sendB time:103.77, calc time:950.506, recvC time:33.8292 ms
12 rank: 9, sendA time:107.643, sendB time:90.0655, calc time:985.173, recvC time:14.7765 ms
13 rank: 11, sendA time:117.322, sendB time:80.055, calc time:968.748, recvC time:31.1868 ms
14 rank: 14, sendA time:110.625, sendB time:102.72, calc time:950.505, recvC time:33.8316 ms
15 rank: 1, sendA time:93.9007, sendB time:91.9671, calc time:992.598, recvC time:7.34615 ms
16 rank: 12, sendA time:94.1706, sendB time:103.297, calc time:950.507, recvC time:33.8302 ms

```

上图为使用16核对相同问题求解的运行时间

我们可以发现随着显成熟的提升,传送数据所需时间也会上升,相对应的计算C的时间会下降,因此需要在其中找到一个平衡,找到合适的线程数使得总体时间(传输时间+计算时间)最小化

## 实验四(自选命题-随机算法并行实现)

### 题目

本次自定义实验本人选择了最近学到的概率算法中的一个题目作为选题,算法的大致流程如下: 使用随机撒点的方式计算pi的近似值,原理如下:

- 随机生成点坐标(x,y),随机算法保证x,y为(0,1)的均匀分布
- 判断点落入圆内/圆外
- 重复多次后,记录总重复次数为sum,落入圆内次数为hit,则pi可以表示为:

$$\pi = \frac{4 * hit}{sum}$$

首先一个显然的性质就是,重复撒点的次数越多,则算出来的pi越接近真实值,而每次撒点均是独立无关的因此很自然的我们可以开多线程,并行的进行撒点试验,之后将各自的结果送回到rank为0的节点计算最后的结果

### 代码实现

```

#include <bits/stdc++.h>
#include <random>
#include <time.h>
#include <mpi.h>
using namespace std;
const int maxn = 1e6;
int A[100];
int main(int argc, char **argv)
{
    int rank;
    int size;
    MPI_Status status;
    MPI_Init(&argc, &argv); //初始化进程
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

```

```

MPI_Comm_size(MPI_COMM_WORLD, &size);

clock_t begin = clock();
default_random_engine random(time(NULL)); //定义随机数生成器
uniform_real_distribution<double> uniform(0.0,1.0);

for(int i=0;i<maxn;i++){ //随机撒点实验
    double x = uniform(random);
    double y = uniform(random);
    if(x*x+y*y<=1) A[rank]++;
}

if(rank==0){ //rank为0的节点负责搜集信息,并计算最后pi值并输出
    for(int i=1;i<size;i++)
        MPI_Recv(A+i,1,MPI_INT,i,0,MPI_COMM_WORLD,&status);
    int sum = 0;
    for(int i=0;i<size;i++) sum += A[i];

    printf("tot:%d hit:%d pi:%lf 相对误差:%lf\n",maxn*size,sum,4.0*sum/maxn/size,abs(4.0*sum/maxn/size-3.1415926535)/3.1415926535*100);

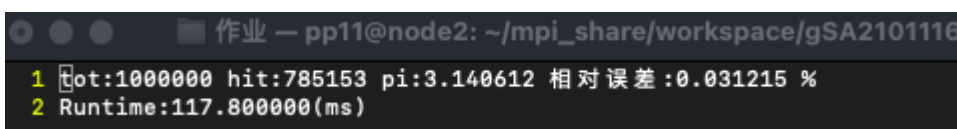
}else { //其他节点将自己的实验结果送给rank为0的节点
    MPI_Send(A+rank,1,MPI_INT,0,0,MPI_COMM_WORLD);
}

MPI_Finalize();
clock_t end = clock();
if (rank == 0){
    printf("Runtime:%lf(ms)\n",1000.0*(end-begin)/CLOCKS_PER_SEC);
}

return 0;
}

```

## 运行截图

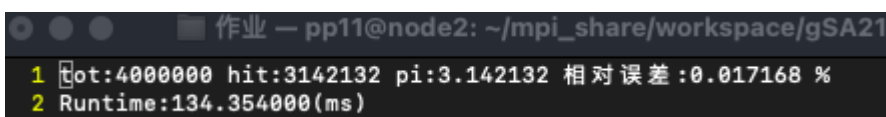


```

作业 — pp11@node2: ~/mpi_share/workspace/gSA2101116
1 tot:1000000 hit:785153 pi:3.140612 相对误差:0.031215 %
2 Runtime:117.800000(ms)

```

上图为单核运行的结果



```

作业 — pp11@node2: ~/mpi_share/workspace/gSA2101116
1 tot:4000000 hit:3142132 pi:3.142132 相对误差:0.017168 %
2 Runtime:134.354000(ms)

```

上图为使用4核计算的结果,在时间代

价没有显著增加的情况下提高了重复次数,进而提高了精度