并行计算 上机报告

> 上机题目：
>
> 1. 用MPI实现π值的计算
> 2. 用MPI实现PSRS排序
>
> 姓名：张劲暾
>
> 学号：PB16111485
>
> 日期：2019年5月11日
>
> 实验环境：
>
> CPU：Intel® Core™ i7-6500U CPU @ 2.50GHz × 4
>
> 内存：7.7 GiB
>
> 操作系统：Ubuntu 18.10 64bit
>
> 软件平台：gcc (Ubuntu 8.2.0-7ubuntu1) 8.2.0

算法设计与分析

题目一

用MPI编程实现π值的计算：

设计：

求π的积分方法：使用公式arctan(1)=π/4以及(arctan(x))'=1/(1+x^2).

在求解arctan(1)时使用矩形法求解：

求解arctan(1)是取a=0，b=1.

$$\int_b^a f(x)dx = y_0\Delta x + y_1\Delta x + \cdots + y_{n-1}\Delta x \tag{1}$$
$$\Delta x = (b-a)/n$$
$$y = f(x)$$
$$y_i = f(a + i*(b-a)/n) \qquad i = 0, 1, 2, \ldots, n$$

```c++
#include <stdio.h>
#include <mpi.h>

long    n;          // 积分区间数
double  sum,        // 进程局部和
        pi,         // 全局pi值
        mypi,       // 局部pi值
        h;          // 积分步长
int     groupSize,  // 通信进程数
        myRank;     // 局部进程号
```

```

int main(int argc, char* argv[])
{
    // 进入MPI环境并完成所有的初始化工作
    MPI_Init(&argc, &argv);
    // 获取当前进程号
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    // 获取通信域进程数
    MPI_Comm_size(MPI_COMM_WORLD, &groupSize);

    n = 2000;

    // 把积分区间数广播到整个通信域
    MPI_Bcast(  &n,             /*发啥*/
                1,              /*发几个*/
                MPI_LONG,       /*发的是啥*/
                0,              /*谁发*/
                MPI_COMM_WORLD  /*给谁发*/
                );

    h = 1.0 / (double)n;
    sum = 0.0;
    for(long i = myRank; i < n; i += groupSize)
    {
        double x = h * (i + 0.5);
        sum += 4.0 / (1.0 + x * x);
    }
    mypi = h * sum;
    // 整个通信域规约
    MPI_Reduce( &mypi,          /*从哪发*/
                &pi,            /*发到哪*/
                1,              /*发几个*/
                MPI_DOUBLE,     /*发的啥*/
                MPI_SUM,        /*收到了怎么操作*/
                0,              /*谁收*/
                MPI_COMM_WORLD  /*收谁*/
                );

    if(myRank == 0)
    {
        printf("%.16lf\n", pi);
    }

    // 从MPI环境中退出
    MPI_Finalize();
    return 0;
}
```

结果：结果正确

```shell
$ mpic++ pi.cpp -o pi
$ mpiexec -n 4 ./pi
3.1415926744231277
$
```

题目二

用MPI实现PSRS排序

设计：

PSRS (Parallel Sorting by Regular Sampling) 排序算法：

- STEP1 均匀划分：将n个元素A[1,...,n]均匀划分为p段，每个pi处理A[(i-1)n/p+1,...,in/p]
- STEP2 局部排序：pi调用串行排序算法对A[(i-1)n/p+1,...,in/p]排序
- STEP3 正则采样：pi从有序子序列A[(i-1)n/p+1,...,in/p]中选取p个样本元素
- STEP4 采样排序：用一台处理器对p^2个样本元素进行串行排序
- STEP5 选择主元：用一台处理器从排好序的样本序列中选取p-1个主元，并传播给其他pi
- STEP6 主元划分：pi按主元将有序段A[(i-1)n/p+1,...,in/p]划分成p段
- STEP7 全局交换：各处理器将其有序段按段号交换到对应的处理器中
- STEP8 局部排序：各处理器对接收到的元素进行局部排序

```c++
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <unistd.h>
#include "quickSort.h"
#include "mergeSort.h"

#define ALLTOONE_TYPE   100
#define MULTI_TYPE      300
#define MULTI_LEN       600

long    arrayLen;
int*    array;
int*    tempArray;
int     localArrayLen;
int*    sample;         // 采样，主元，段长
int*    pivotIndex;

void PSRSSort()
{
    int localID, groupSize;

    MPI_Comm_rank(MPI_COMM_WORLD, &localID);
    MPI_Comm_size(MPI_COMM_WORLD, &groupSize);

    MPI_Status status[groupSize];
    MPI_Request request[groupSize];

    array = (int*)malloc(arrayLen * sizeof(int));
    tempArray = (int*)malloc(arrayLen * sizeof(int));
    if(groupSize > 1)
    {
        sample = (int*)malloc(groupSize * (groupSize - 1) * sizeof(int));
        pivotIndex = (int*)malloc(groupSize * 2 * sizeof(int));
    }
//=========================================================================
    // 均匀划分
    MPI_Barrier(MPI_COMM_WORLD);
    localArrayLen = arrayLen / groupSize;
    srand((unsigned int)time(NULL) + localID);
    usleep(5 * localID * 1000);
    printf("On Process %d the input data is:\n", localID);
    for(int i = 0; i < localArrayLen; i++)
```

```
    {
        array[i] = Myrandom();
        printf("%d\t",array[i]);
    }
    printf("\n");
//=========================================================================
    // 局部排序
    MPI_Barrier(MPI_COMM_WORLD);
    quickSort(array, 0, localArrayLen - 1);
//=========================================================================
    MPI_Barrier(MPI_COMM_WORLD);
    if(groupSize > 1)
    {
        MPI_Barrier(MPI_COMM_WORLD);
        // 正则采样
        int step = (int)(localArrayLen / groupSize);
        for(int i = 0; i < groupSize - 1; i++)
        {
            sample[i] = array[(i + 1) * step - 1];
        }
//=========================================================================
        MPI_Barrier(MPI_COMM_WORLD);
        if(localID == 0)    // 主进程收集采样
        {
            for(int i = 1, j = 0; i < groupSize; i++, j++)
            {   // Begins a nonblocking receive
                MPI_Irecv(  &sample[i * (groupSize - 1)],
                          // initial address of receive buffer (choice)
                            sizeof(int) * (groupSize - 1),
                          // number of elements in receive buffer (integer)
                            MPI_CHAR,
                          // datatype of each receive buffer element (handle)
                            i,
                          // rank of source (integer)
                            ALLTOONE_TYPE + i,
                          // message tag (integer)
                            MPI_COMM_WORLD,
                          // communicator (handle)
                            &request[j]
                          // communication request (handle)
                            );
            }
            // Waits for all given MPI Requests to complete
            MPI_Waitall (   groupSize - 1,
                        // list length (integer)
                            request,
                        // array of request handles (array of handles)
                            status
                        // array of status objects (array of Statuses). May be MPI_STATUSES_IGNORE.
                        );
//=========================================================================
            MPI_Barrier(MPI_COMM_WORLD);
            // 采样排序
            quickSort(sample, 0, groupSize * (groupSize - 1) - 1);
            MPI_Barrier(MPI_COMM_WORLD);
//=========================================================================
            for(int i = 1; i < groupSize; i++)
            {
                sample[i] = sample[i * (groupSize - 1) - 1];
            }
```

```
104             // 主元广播
105             // Broadcasts a message from the process with rank "root"
106             // to all other processes of the communicator
107             MPI_Bcast(  sample,                     // starting address of buffer (choice)
108                         groupSize * sizeof(int),    // number of entries in buffer (integer)
109                         MPI_CHAR,                   // data type of buffer (handle)
110                         0,                          // rank of broadcast root (integer)
111                         MPI_COMM_WORLD              // communicator (handle)
112                         );
113             MPI_Barrier(MPI_COMM_WORLD);
114 //==========================================================================
115         }
116         else
117         {   // 局部采样结果发出
118             MPI_Send(   sample,
119                      // initial address of send buffer (choice)
120                         sizeof(int) * (groupSize - 1),
121                      // number of elements in send buffer (nonnegative integer)
122                         MPI_CHAR,
123                      // datatype of each send buffer element (handle)
124                         0,
125                      // rank of destination (integer)
126                         ALLTOONE_TYPE + localID,
127                      // message tag (integer)
128                         MPI_COMM_WORLD
129                      // communicator (handle)
130                         );
131 //==========================================================================
132             MPI_Barrier(MPI_COMM_WORLD);
133             // 采样排序
134             // quickSort(sample, 0, groupSize * (groupSize - 1) - 1);
135             MPI_Barrier(MPI_COMM_WORLD);
136 //==========================================================================
137             // 接收主元
138             MPI_Bcast(  sample,                     // starting address of buffer (choice)
139                         groupSize * sizeof(int),    // number of entries in buffer (integer)
140                         MPI_CHAR,                   // data type of buffer (handle)
141                         0,                          // rank of broadcast root (integer)
142                         MPI_COMM_WORLD              // communicator (handle)
143                         );
144             MPI_Barrier(MPI_COMM_WORLD);
145 //==========================================================================
146         }
147         // 主元划分
148         int m = 1;  /*主元指针*/
149         pivotIndex[0] = 0;
150         for(int i = 0; i < localArrayLen && m < groupSize;)
151         {
152             if(array[i] > sample[m])
153             {
154                 pivotIndex[2 * m    ] = i;
155                 pivotIndex[2 * m - 1] = i;
156                 m++;
157             }
158             else
159             {
160                 i++;
161             }
162 
163         }
```

```
164         while(m != groupSize){
165             pivotIndex[2 * m    ] = localArrayLen;
166             pivotIndex[2 * m - 1] = localArrayLen;
167             m++;
168         }
169         pivotIndex[2 * m - 1] = localArrayLen;
170 //=========================================================================
171         MPI_Barrier(MPI_COMM_WORLD);
172         // 全局交换
173         for(int i = 0, j = 0; i < groupSize; i++)
174         {
175             if(i == localID)
176             {   // 划分段长度，先发射出去，就知道下一步真正传数据要传多少
177                 sample[i] = pivotIndex[2 * i + 1] - pivotIndex[2 * i];
178                 for(int m = 0, n; m < groupSize; m++)
179                 {
180                     if(m != localID)
181                     {
182                         n = pivotIndex[2 * m + 1] - pivotIndex[2 * m];
183                         MPI_Send(   &n,
184                                     // initial address of send buffer (choice)
185                                     sizeof(int),
186                                     // number of elements in send buffer (nonnegative integer)
187                                     MPI_CHAR,
188                                     // datatype of each send buffer element (handle)
189                                     m,
190                                     // rank of destination (integer)
191                                     MULTI_LEN + localID,
192                                     // message tag (integer)
193                                     MPI_COMM_WORLD
194                                     // communicator (handle)
195                                     );
196                     }
197                 }
198             }
199             else
200             {   // Blocking receive for a message
201                 MPI_Recv(   &sample[i],
202                             // initial address of receive buffer (choice)
203                             sizeof(int),
204                             // maximum number of elements in receive buffer (integer)
205                             MPI_CHAR,
206                             //
207                             i,
208                             // rank of source (integer)
209                             MULTI_LEN + i,
210                             // message tag (integer)
211                             MPI_COMM_WORLD,
212                             //
213                             &status[j++]
214                             // status object (Status)
215                             );
216             }
217         }
218 //=========================================================================
219         MPI_Barrier(MPI_COMM_WORLD);
220         int localPointer = 0;
221         for(int i = 0, j = 0; i < groupSize; i++)
222         {
223             MPI_Barrier(MPI_COMM_WORLD);
```

```
            if(i == localID)
            {
                for(int n = pivotIndex[2 * i]; n < pivotIndex[2 * i + 1]; n++)
                {
                    tempArray[localPointer++] = array[n];
                }
            }
            MPI_Barrier(MPI_COMM_WORLD);
            if(i == localID)
            {
                for(int m = 0, n = 0; m < groupSize; m++)
                {
                    if(m != localID)
                    {
                        MPI_Send(   &array[pivotIndex[2 * m]],
                                 // initial address of send buffer (choice)
                                    sizeof(int) * (pivotIndex[2 * m + 1] - pivotIndex[2 * m]),
                                 // number of elements in send buffer (nonnegative integer)
                                    MPI_CHAR,
                                 // datatype of each send buffer element (handle)
                                    m,                      // rank of destination (integer)
                                    MULTI_TYPE + localID,   // message tag (integer)
                                    MPI_COMM_WORLD          // communicator (handle)
                                    );
                    }
                }
            }
            else
            {
                MPI_Recv(   &tempArray[localPointer],
                         // initial address of receive buffer (choice)
                            sizeof(int) * sample[i],
                         // maximum number of elements in receive buffer (integer)
                            MPI_CHAR,       //
                            i,              // rank of source (integer)
                            MULTI_TYPE + i, // message tag (integer)
                            MPI_COMM_WORLD, //
                            &status[j++]    // status object (Status)
                            );
                localPointer += sample[i];
            }
            MPI_Barrier(MPI_COMM_WORLD);
        }
        localArrayLen = localPointer;
        MPI_Barrier(MPI_COMM_WORLD);
        // 归并排序
        multiMergeSort(tempArray, sample, array, groupSize);
        MPI_Barrier(MPI_COMM_WORLD);
    }
    //*********************************************************************************
    usleep(5 * localID * 1000);
    if(localID == 0)
        printf("\n=========================================================\n\n");
    printf("Process %d's sorted data:\n",localID);
    for(int i = 0; i < localArrayLen; i++)
    {
        printf("%d\t",array[i]);
    }
    printf("\n");
    //*********************************************************************************
```

```
}

int main(int argc, char* argv[])
{
    int localPID;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &localPID);
    arrayLen = 64;

    PSRSSort();
    MPI_Finalize();
    return 0;
}
```

结果：

```shell
zjt@zjt-HP-Pavilion-Notebook:~/ParallelComputingAlgorithm/MPI$ mpic++ psrs.cpp -o psrs
zjt@zjt-HP-Pavilion-Notebook:~/ParallelComputingAlgorithm/MPI$ mpiexec -n 4 ./psrs
On Process 0 the input data is:
-26 -22 12  -31 -25 36  9   35  -26 19  15  -8  47  35  -16 32
On Process 1 the input data is:
16  -2  25  42  -7  -35 -4  38  39  4   -26 15  21  5   -17 -12
On Process 2 the input data is:
27  24  44  33  33  11  -39 -3  -22 20  0   46  29  27  -26 -17
On Process 3 the input data is:
-44 32  -34 43  21  11  -48 -1  -47 -26 -39 -18 -8  42  -20 35

================================================================================================

Process 0's sorted data:
-48 -47 -44 -39 -39 -35 -34 -31 -26 -26 -26 -26 -26 -25 -22 -22 -20 -18
Process 1's sorted data:
-17 -17 -16 -12 -8  -8  -7  -4  -3  -2  -1  0   4
Process 2's sorted data:
5   9   11  11  12  15  15  16  19  20  21  21
Process 3's sorted data:
24  25  27  27  29  32  32  33  33  35  35  35  36  38  39  42  42  43  44  46  47
```

总结

通过算法实现锻炼了并行思维，熟悉了MPI并行库的使用。

附录

辅助头文件quickSort.h

```c++
#include <time.h>
#include <math.h>
#define RANDOM_LIMIT 50

double Myrandom(void)
```

```
{
    int Sign = rand() % 2;
    return (rand() % RANDOM_LIMIT) / pow(-1,Sign + 2);
}
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
int partition(int* array, int left, int right)
{
    int x = array[right];
    int i = left - 1;
    for(int j = left; j < right; j++)
    {
        if(array[j] <= x)
        {
            swap(&array[++i],&array[j]);
        }
    }
    swap(&array[i + 1],&array[right]);
    return i + 1;
}
void quickSort(int* array, int left, int right)
{
    if(left < right)
    {
        int q = partition(array, left, right);
        quickSort(array, left, q - 1);
        quickSort(array, q + 1, right);
    }
}
```

辅助头文件mergeSort.h

```c++
void merge(int* arraySource, int len1, int len2, int* arrayDest)
{
    int index1 = 0, index2 = len1;
    for(int i = 0; i < len1 + len2; i++)
    {
        if(index1 == len1)
        {
            arrayDest[i] = arraySource[index2++];
        }
        else
        {
            if(index2 == len1 + len2)
            {
                arrayDest[i] = arraySource[index1++];
            }
            else
            {
                if(arraySource[index1] > arraySource[index2])   arrayDest[i] = arraySource[index2++];
```

```
                else                                            arrayDest[i] = arraySource[index1++];
            }
        }
    }
}
void multiMergeSort(int* arraySource, int* div, int* arrayDest, int groupSize)
{
    int j = 0;
    for(int i = 0; i < groupSize; i++)
    {
        if(div[i] > 0)
        {
            div[j++] = div[i];
            if(j < i + 1) div[i] = 0;
        }
    }
    if(j > 1)
    {
        int n = 0;
        for(int i = 0; i + 1 < j; i++)
        {
            merge(&arraySource[n], div[i], div[i + 1], &arrayDest[n]);
            div[i] += div[i + 1];
            div[i + 1] = 0;
            n += div[i];
        }
        if(j % 2 == 1)
        {
            for(int i = 0; i < div[j - 1]; i++, n++)
            {
                arrayDest[n] = arraySource[n];
            }
        }
        multiMergeSort(arrayDest, div, arraySource, groupSize);
    }
}
```