《并行计算》上机报告

姓名：	李子旸	学号：	PB16110959	日期：	2019.5.18
上机题目：	MPI				

实验环境：
CPU:Intel 至强 E5-2650 v2; 内存:64GB; 操作系统:Ubuntu; 软件平台:vim

一、算法设计与分析：

题目：
用 MPI 编程实现 PI 的计算。
用 MPI 实现 PSRS 排序

算法设计：
pi 的计算：
串行：

```
for (int i = 1; i <= n; i++){
x = (i - 0.5) / n;
sum += 4 / (1 + x * x);
}
pi = sum / n;
```

使用 mpi 库进行并行，利用多核计算机。

PSRS 排序：
均匀划分->局部排序->选取样本->样本排序->选择主元->主元划分->全局交换->归并排序

二、核心代码：

计算 pi：

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD); //将 n 广播到所有进程中
h = 1.0 / (double)n; //每个矩形块的宽
sum = 0.0;
for (int i = myid + 1; i <= n; i += size)
{ //4 个进程
x = h * ((double)i - 0.5); //进程 0:1,5,9,... 进程 1:2,6,10,...
sum = sum + f(x); //进程 2:3,7,11,... 进程 3:4,8,12,...
}
```

```
sum = sum * h; //每个进程计算的矩形面积之和
MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD); //利用归约操作(MPI_SUM)将所有进程的 sum 累加到 root 进程(0)的 sum 当中得到结果
MPI_Barrier(MPI_COMM_WORLD);
```

PSRS 排序:

```
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &myId);
MPI_Get_processor_name(processorName, &nameLength);

//iprintf("Process %d is on %s\n", myId, processorName);
if (myId == 0){
    starttime = MPI_Wtime();
}

pivots = (int *)malloc(p * sizeof(int));
partitionSizes = (int *)malloc(p * sizeof(int));
newPartitionSizes = (int *)malloc(p * sizeof(int));
for (k = 0; k < p; k++){
partitionSizes[k] = 0;
}

// 获取起始位置和子数组大小
startIndex = myId * N / p;
if (p == (myId + 1)){
endIndex = N;
}
else{
endIndex = (myId + 1) * N / p;
}
subArraySize = endIndex - startIndex;

MPI_Barrier(MPI_COMM_WORLD);
//调用各阶段函数
phase1(array, N, startIndex, subArraySize, pivots, p);
if (p > 1){
phase2(array, startIndex, subArraySize, pivots, partitionSizes, p, myId);
phase3(array, startIndex, partitionSizes, &newPartitions, newPartitionSizes, p);
phase4(newPartitions, newPartitionSizes, p, myId, array);
}
```

三、结果与分析：

计算 pi 值：

n = 100000000

串行：1.723480s

mpi 并行：

4 核：0.746483s

8 核：0.396780s

12 核：0.26198s

16 核：0.185104s

24 核：0.127489s

32 核：0.099769s

PSRS 排序：

n = 10000000

串行归并排序：4.952611s

mpi 并行：

4 核：0.776521s

8 核：0.415834s

12 核：0.317133s

16 核：0.35468s

24 核：0.260217s

32 核：0.246681s

四、备注（* 可选）：

有可能影响结论的因素：

核与核之间的通信耗时,针对特定的环境可以采取特定的并行方法,对实验结果影响较大。

呈现出核越多，时间开销越小的趋势

总结：

使用 mpi 充分发挥多核计算机的特性，核越多，速度越快。

附录（源代码）	算法源代码（C/C++/JAVA 描述）
	
	mpi 计算 pi 值：
	#include <cstdio>
	#include <mpi.h>
	#include <iostream>
	using namespace std;
	const long long INF = 1000000000;
	
	inline double f(double x){
	return 4 / (1 + x * x);
	}
	
	int main(int argc, char *argv[])
	{
	double pi, h, sum, x, startime, endtime;
	int size, myid;
	long long n;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &myid);
	MPI_Comm_size(MPI_COMM_WORLD, &size);
	
	n = 0;
	if (0 == myid)
	{
	n = INF;
	startime = MPI_Wtime();
	}
	MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD); //将 n 广播到所有进程中
	h = 1.0 / (double)n; //每个矩形块的宽
	sum = 0.0;
	for (int i = myid + 1; i <= n; i += size)
	{ //4 个进程
	x = h * ((double)i - 0.5); //进程 0:1,5,9,... 进程 1:2,6,10,...
	sum = sum + f(x); //进程 2:3,7,11,... 进程 3:4,8,12,...
	}
	sum = sum * h; //每个进程计算的矩形面积之和
	MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD); //利用归约操作(MPI_SUM)将所有进程的 sum

```
累加到 root 进程(0)的 sum 当中得到结果
MPI_Barrier(MPI_COMM_WORLD);
if (myid == 0)
{
endtime = MPI_Wtime();
printf("用时:%f\n", endtime - startime);
printf("%0.15f\n", pi);
}
}
```

mpi PSRS 排序：

```
#include <cstdlib>
#include <cstdio>
#include <climits>
#include <cassert>
#include <sys/time.h>
#include <unistd.h>
#include <iostream>
#include <ctime>
#include "mpi.h"
using namespace std;

int i, j, k;
int N = 70000000;

int cmp(const void *a, const void *b){
if (*(int *)a < *(int *)b)
return -1;
if (*(int *)a > *(int *)b)
return 1;
else
return 0;
}

void phase1(int *array, int N, int startIndex, int subArraySize, int
*pivots, int p){
// 对子数组进行局部排序
qsort(array + startIndex, subArraySize, sizeof(array[0]), cmp);

// 正则采样
```

```
for (i = 0; i < p; i++){
pivots[i] = array[startIndex + (i * (N / (p * p)))];
}
return;
}

void phase2(int *array, int startIndex, int subArraySize, int *pivots,
int *partitionSizes, int p, int myId){
int *collectedPivots = (int *)malloc(p * p * sizeof(pivots[0]));
int *phase2Pivots = (int *)malloc((p - 1) * sizeof(pivots[0])); //主元
int index = 0;

//收集消息，根进程在它的接受缓冲区中包含所有进程的发送缓冲区的连接。
MPI_Gather(pivots, p, MPI_INT, collectedPivots, p, MPI_INT, 0,
MPI_COMM_WORLD);
if (myId == 0){
qsort(collectedPivots, p * p, sizeof(pivots[0]), cmp); //对正则采样的样本进行排序

// 采样排序后进行主元的选择
for (i = 0; i < (p - 1); i++){
phase2Pivots[i] = collectedPivots[(((i + 1) * p) + (p / 2)) - 1];
}
}
//发送广播
MPI_Bcast(phase2Pivots, p - 1, MPI_INT, 0, MPI_COMM_WORLD);
// 进行主元划分，并计算划分部分的大小
for (i = 0; i < subArraySize; i++){
if (array[startIndex + i] > phase2Pivots[index]){
//如果当前位置的数字大小超过主元位置，则进行下一个划分
index += 1;
}
if (index == p){
//最后一次划分，子数组总长减掉当前位置即可得到最后一个子数组划分的大小
partitionSizes[p - 1] = subArraySize - i + 1;
break;
}
partitionSizes[index]++; //划分大小自增
}
free(collectedPivots);
free(phase2Pivots);
```

```
return;
}

void phase3(int *array, int startIndex, int *partitionSizes, int **newPartitions, int *newPartitionSizes, int p){
int totalSize = 0;
int *sendDisp = (int *)malloc(p * sizeof(int));
int *recvDisp = (int *)malloc(p * sizeof(int));

// 全局到全局的发送，每个进程可以向每个接收者发送数目不同的数据.
MPI_Alltoall(partitionSizes, 1, MPI_INT, newPartitionSizes, 1, MPI_INT, MPI_COMM_WORLD);

// 计算划分的总大小，并给新划分分配空间
for (i = 0; i < p; i++){
totalSize += newPartitionSizes[i];
}
*newPartitions = (int *)malloc(totalSize * sizeof(int));

// 在发送划分之前计算相对于 sendbuf 的位移，此位移处存放着输出到进程的数据
sendDisp[0] = 0;
recvDisp[0] = 0; //计算相对于 recvbuf 的位移，此位移处存放着从进程接受到的数据
for (i = 1; i < p; i++){
sendDisp[i] = partitionSizes[i - 1] + sendDisp[i - 1];
recvDisp[i] = newPartitionSizes[i - 1] + recvDisp[i - 1];
}

//发送数据，实现 n 次点对点通信
MPI_Alltoallv(&(array[startIndex]), partitionSizes, sendDisp, MPI_INT, *newPartitions, newPartitionSizes, recvDisp, MPI_INT, MPI_COMM_WORLD);

free(sendDisp);
free(recvDisp);
return;
}

void phase4(int *partitions, int *partitionSizes, int p, int myId, int *array){
int *sortedSubList;
int *recvDisp, *indexes, *partitionEnds, *subListSizes, totalListSize;
```

```
indexes = (int *)malloc(p * sizeof(int));
partitionEnds = (int *)malloc(p * sizeof(int));
indexes[0] = 0;
totalListSize = partitionSizes[0];
for (i = 1; i < p; i++){
totalListSize += partitionSizes[i];
indexes[i] = indexes[i - 1] + partitionSizes[i - 1];
partitionEnds[i - 1] = indexes[i];
}
partitionEnds[p - 1] = totalListSize;

sortedSubList = (int *)malloc(totalListSize * sizeof(int));
subListSizes = (int *)malloc(p * sizeof(int));
recvDisp = (int *)malloc(p * sizeof(int));

// 归并排序
for (i = 0; i < totalListSize; i++){
int lowest = INT_MAX;
int ind = -1;
for (j = 0; j < p; j++){
if ((indexes[j] < partitionEnds[j]) && (partitions[indexes[j]] <
lowest)){
lowest = partitions[indexes[j]];
ind = j;
}
}
sortedSubList[i] = lowest;
indexes[ind] += 1;
}

// 发送各子列表的大小回根进程中
MPI_Gather(&totalListSize, 1, MPI_INT, subListSizes, 1, MPI_INT, 0,
MPI_COMM_WORLD);

// 计算根进程上的相对于 recvbuf 的偏移量
if (myId == 0){
recvDisp[0] = 0;
for (i = 1; i < p; i++){
recvDisp[i] = subListSizes[i - 1] + recvDisp[i - 1];
}
}
```

```
//发送各排好序的子列表回根进程中
MPI_Gatherv(sortedSubList, totalListSize, MPI_INT, array, subListSizes, recvDisp, MPI_INT, 0, MPI_COMM_WORLD);

free(partitionEnds);
free(sortedSubList);
free(indexes);
free(subListSizes);
free(recvDisp);
return;
}

//PSRS 排序函数，调用了 4 个过程函数
void psrs_mpi(int *array, int N){
double starttime, endtime;
int p, myId, *partitionSizes, *newPartitionSizes, nameLength;
int subArraySize, startIndex, endIndex, *pivots, *newPartitions;
char processorName[MPI_MAX_PROCESSOR_NAME];

MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &myId);
MPI_Get_processor_name(processorName, &nameLength);

//iprintf("Process %d is on %s\n", myId, processorName);
if (myId == 0){
    starttime = MPI_Wtime();
}

pivots = (int *)malloc(p * sizeof(int));
partitionSizes = (int *)malloc(p * sizeof(int));
newPartitionSizes = (int *)malloc(p * sizeof(int));
for (k = 0; k < p; k++){
partitionSizes[k] = 0;
}

// 获取起始位置和子数组大小
startIndex = myId * N / p;
if (p == (myId + 1)){
endIndex = N;
}
else{
endIndex = (myId + 1) * N / p;
}
```

```
subArraySize = endIndex - startIndex;

MPI_Barrier(MPI_COMM_WORLD);
//调用各阶段函数
phase1(array, N, startIndex, subArraySize, pivots, p);
if (p > 1){
phase2(array, startIndex, subArraySize, pivots, partitionSizes, p, myId);
phase3(array, startIndex, partitionSizes, &newPartitions, newPartitionSizes, p);
phase4(newPartitions, newPartitionSizes, p, myId, array);
}

if (myId == 0){
endtime = MPI_Wtime();
printf("time : %f\n", endtime - starttime);
}
if (p > 1){
free(newPartitions);
}
free(partitionSizes);
free(newPartitionSizes);
free(pivots);

free(array);
MPI_Finalize();
}

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

int *array;
array = (int *)malloc(N * sizeof(int));

srand(time(NULL));
for (k = 0; k < N; k++){
array[k] = rand() % N;
}
MPI_Init(&argc, &argv); //MPI 初始化
psrs_mpi(array, N); //调用 PSRS 算法进行并行排序

return 0;
}
```