《并行计算》上机报告

姓名：	王立峰	学号：	PB16111245	日期：	2019.5.4
上机题目：	OpenMP				

实验环境：gcc+OpenMP
CPU：虚拟 CPU; 内存: 虚拟内存; 操作系统：Ubuntu 18.10; 软件平台： vscode ;

一、算法设计与分析：

题目:
1、用 4 种不同并行方式的 OpenMP 实现 π 值的计算
2、用 OpenMP 实现 PSRS 排序

算法设计：
1、求 pi 的方法如下
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.

$$\int_a^b f(x)dx = y_0\Delta x + y_1\Delta x + \cdots + y_{n-1}\Delta x$$

$$\Delta x = (b-a)/n$$

$$y = f(x)$$

$$y_i = f(a + i * (b-a)/n) \quad i = 0,1,2,\ldots,n$$

然后利用这歌计算方法可以构造出四种不同的 OpenMP 并行化计算 pi 的值。

2、PSRS 算法如下
PSRS 排序可分为 8 个部分：
均匀划分： 将 n 个元素 A[1..n]均匀划分成 p 段，每个 pi 处理 A[(i-1)n/p+1..in/p]；
局部排序：pi 调用串行排序算法对 A[(i-1)n/p+1..in/p]排序；
正则采样：pi 从其有序子序列 A[(i-1)n/p+1..in/p]中选取 p 个样本元素；
采样排序：用一台处理器对 p2 个样本元素进行串行排序；
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，并播送给其他 pi；
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；
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；
归并排序：各处理器对接收到的元素进行归并排序。

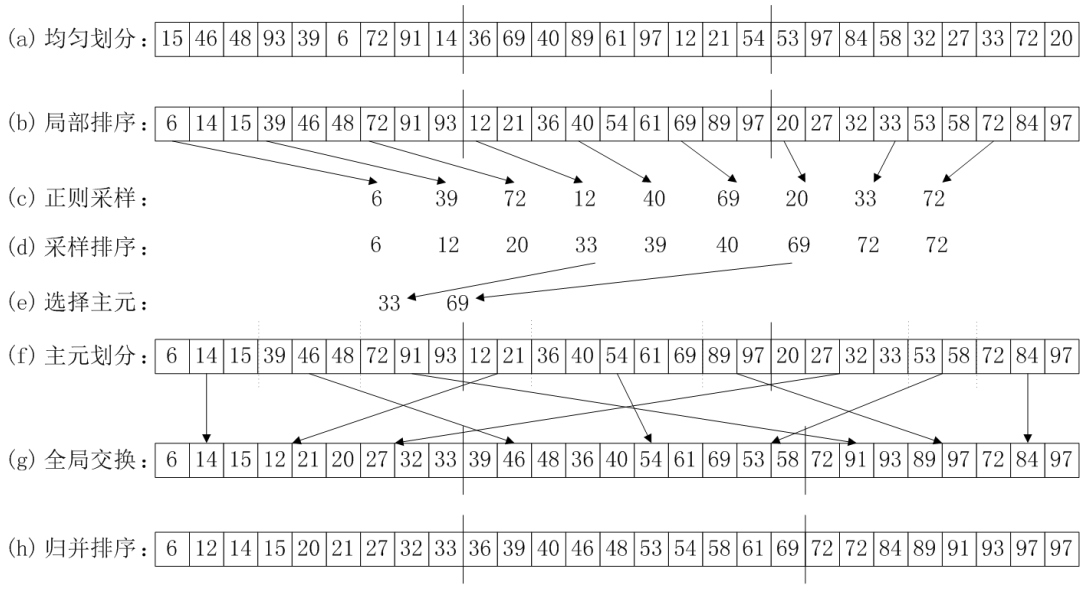


图6. 1

算法分析：

1、

使用并行域并行化的程序共 2 个线程参加计算，其中线程 0 进行迭代步 0,2,4,...线程 1 进行迭代步 1,3,5,....；

使用共享任务结构并行化的程序共 2 个线程参加计算，其中线程 0 进行迭代步 0~49999，线程 1 进行迭代步 50000~99999；

使用 private 子句和 critical 部分并行化的程序共 2 个线程参加计算，其中线程 0 进行迭代步 0,2,4,...线程 1 进行迭代步 1,3,5,....当被指定为 critical 的代码段　正在被 0 线程执行时，1 线程的执行也到达该代码段，则它将被阻塞知道 0 线程退出临界区；

使用并行规约的并行程序共 2 个线程参加计算，其中线程 0 进行迭代步 0~49999，线程 1 进行迭代步 50000~99999。

2、PSRS 算法分析

定义处理器处理段个数数组变量、划分段的大小、正则采样数 p、主元数组、处理器各部分排序的三维数组空间。

设置并行线程后，添加第一个并行域：

```
\#pragma omp parallel shared(base,array,n,i,pivot,count) private(id)
{
    每个处理器对所在的段进行局部串行归并排序；
    并行域嵌套
    #pragma omp critical
    {
        每个处理器选出 p 个样本；
```

```
        }
    设置路障#pragma omp barrier
    主线程并行域#pragma omp master
    {
            选出 num_threads-1 个主元
    }
    设置路障#pragma omp barrier
    {
            根据主元对每一个 cpu 数据进行划分
    }
}
第二个并行域：
＃pragma omp parallel shared(pivot_array,count)
{
    向各个线程发送数据，各个线程自己排序；
    打印输出数据；
}
主函数测试数组排序。
```

二、核心代码：

1、计算 pi 的代码助教已给出，不作说明了

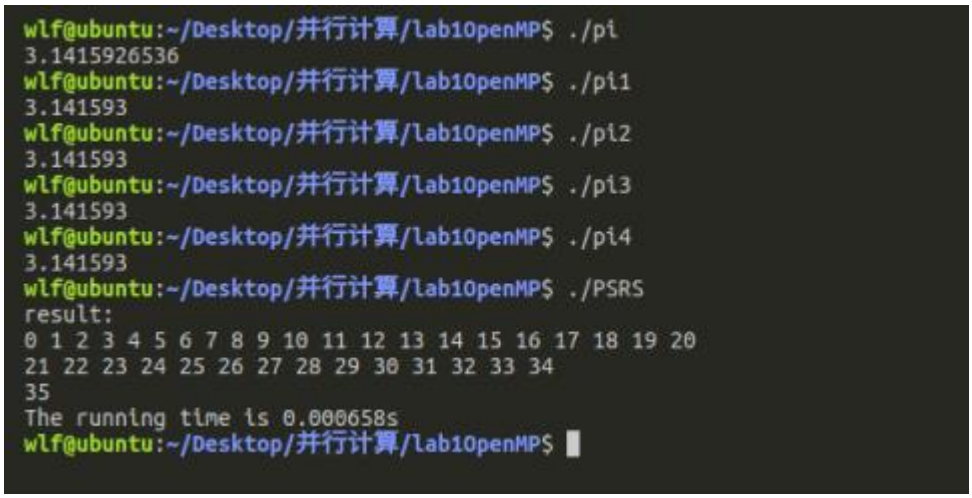
2、PSRS 核心代码如下：

```
void PSRS(int *array, int n)
{
    int id;
    int i=0;
    int count[num_threads][num_threads] = { 0 };      //每个处理器每段的个数
    int base = n / num_threads;        //划分的每段段数
    int p[num_threads*num_threads]; //正则采样数为 p
    int pivot[num_threads-1];          //主元
    int pivot_array[num_threads][num_threads][50]={0};   //处理器数组空间

    omp_set_num_threads(num_threads);
    #pragma omp parallel shared(base,array,n,i,p,pivot,count) private(id)
    {
        id = omp_get_thread_num();
        //每个处理器对所在的段进行局部串行归并排序
        MergeSort(array,id*base,(id+1)*base-1);
        #pragma omp critical
```

```
        //每个处理器选出 P 个样本，进行正则采样
        for(int k=0; k<num_threads; k++)
            p[i++] = array[(id-1)*base+(k+1)*base/(num_threads+1)];

        //设置路障，同步队列中的所有线程
        #pragma omp barrier

        //主线程对采样的 p 个样本进行排序
        #pragma omp master
        {
            MergeSort(p,0,i-1);
            //选出 num_threads-1 个主元
            for(int m=0; m<num_threads-1; m++)
                pivot[m] = p[(m+1)*num_threads];
        }

        #pragma omp barrier
        //根据主元对每一个 cpu 数据段进行划分
        for(int k=0; k<base; k++)
        {
            for(int m=0; m<num_threads; m++)
            {
                if(array[id*base+k] < pivot[m])
                {
                    pivot_array[id][m][count[id][m]++] = array[id*base+k];
                    break;
                }
                else if(m == num_threads-1) //最后一段
                {
                    pivot_array[id][m][count[id][m]++] = array[id*base+k];
                }
            }
        }
    }

    //向各个线程发送数据，各个线程自己排序
    #pragma omp parallel shared(pivot_array,count)
    {
        int id=omp_get_thread_num();
        for(int k=0; k<num_threads; k++)
        {
            if(k!=id)
            {
```

```
memcpy(pivot_array[id][id]+count[id][id],pivot_array[k][id],sizeof(int)*count[k][id])
;
                        count[id][id] += count[k][id];
                }
        }
        MergeSort(pivot_array[id][id],0,count[id][id]-1);
    }
    i = 0;
    printf("result:\n");
    for(int k=0; k<num_threads; k++)
    {
        for(int m=0; m<count[k][k]; m++)
        {
            printf("%d ",pivot_array[k][k][m]);
        }
        printf("\n");
    }
}
```

三、结果与分析：



通过结果可以看到四种计算 pi 的方案都可以准确快速地算出 pi 的值，而 PSRS 并行算法也可以很快地排好序。

四、备注（* 可选）：

有可能影响结论的因素：

总结：

通过本次实验，对 OpenMP 实现并行计算有了初步掌握，大致学会了几种并行的方法，加深了对并行计算的理解。

附录（源代码）	算法源代码（C/C++/JAVA 描述） 计算 pi 的源代码不贴出了，以下是 PSRS 的源代码 #include<stdio.h> #include<string.h> #include<stdlib.h>

```
#include<omp.h>

#define num_threads 3

int *L,*R;

//Merge 函数合并两个子数组形成单一的已排好序的字数组
//并代替当前的子数组 A[p..r]
void Merge(int *a, int p, int q, int r)
{
	int i,j,k;
	int n1 = q - p + 1;
	int n2 = r - q;
	L = (int*)malloc((n1+1)*sizeof(int));
	R = (int*)malloc((n2+1)*sizeof(int));
	for(i=0; i<n1; i++)
		L[i] = a[p+i];
	L[i] = 65536;
	for(j=0; j<n2; j++)
		R[j] = a[q+j+1];
	R[j] = 65536;
	i=0,j=0;
	for(k=p; k<=r; k++){
		if(L[i]<=R[j]){
			a[k] = L[i];
			i++;
		}
		else{
			a[k] = R[j];
			j++;
		}
	}
}

//归并排序
void MergeSort(int *a, int p, int r)
{
	if(p<r){
		int q = (p+r)/2;
		MergeSort(a,p,q);
		MergeSort(a,q+1,r);
		Merge(a,p,q,r);
	}
}
```

```
void PSRS(int *array, int n)
{
	int id;
	int i=0;
	int count[num_threads][num_threads] = { 0 };	//每个处理器每段的个数
	int base = n / num_threads;		//划分的每段段数
	int p[num_threads*num_threads]; //正则采样数为 p
	int pivot[num_threads-1];		//主元
	int  pivot_array[num_threads][num_threads][50]={0};	//处理器数组空间

	omp_set_num_threads(num_threads);
	#pragma omp parallel shared(base,array,n,i,p,pivot,count) private(id)
	{
		id = omp_get_thread_num();
		//每个处理器对所在的段进行局部串行归并排序
		MergeSort(array,id*base,(id+1)*base-1);
		#pragma omp critical

		//每个处理器选出 P 个样本，进行正则采样
		for(int k=0; k<num_threads; k++)
			p[i++] = array[(id-1)*base+(k+1)*base/(num_threads+1)];

		//设置路障，同步队列中的所有线程
		#pragma omp barrier

		//主线程对采样的 p 个样本进行排序
		#pragma omp master
		{
			MergeSort(p,0,i-1);
			//选出 num_threads-1 个主元
			for(int m=0; m<num_threads-1; m++)
				pivot[m] = p[(m+1)*num_threads];
		}

		#pragma omp barrier
		//根据主元对每一个 cpu 数据段进行划分
		for(int k=0; k<base; k++)
		{
			for(int m=0; m<num_threads; m++)
```

```
                {
                    if(array[id*base+k] < pivot[m])
                    {
                        pivot_array[id][m][count[id][m]++] = array[id*base+k];
                        break;
                    }
                    else if(m == num_threads-1) //最后一段
                    {
                        pivot_array[id][m][count[id][m]++] = array[id*base+k];
                    }
                }
            }
        }

        //向各个线程发送数据，各个线程自己排序
        #pragma omp parallel shared(pivot_array,count)
        {
            int id=omp_get_thread_num();
            for(int k=0; k<num_threads; k++)
            {
                if(k!=id)
                {

memcpy(pivot_array[id][id]+count[id][id],pivot_array[k][id],sizeof(int)*count[k][id]);
                    count[id][id] += count[k][id];
                }
            }
            MergeSort(pivot_array[id][id],0,count[id][id]-1);
        }
        i = 0;
        printf("result:\n");
        for(int k=0; k<num_threads; k++)
        {
            for(int m=0; m<count[k][k]; m++)
            {
                printf("%d ",pivot_array[k][k][m]);
            }
            printf("\n");
        }
}
```

```
int main()
{
	int array[36] = { 16,2,17,24,33,28,30,1,0,27,9,25,
34,23,19,18,11,7,21,13,8,35,12,29 , 6,3,4,14,22,15,32,10,26,31,20,5 };
	double begin,end,time;
	begin = omp_get_wtime();
	PSRS(array, 36);
/*
	MergeSort(list,0,35);
	for(int i=0; i<36; i++)
	{
		printf("%d ",list[i]);
	}
*/
	end = omp_get_wtime();
	time = end - begin;
	printf("The running time is %lfs\n",time);

	return 0;
}
```