**中山大学计算机学院本科生实验报告**

课程名称：超级计算机原理与操作 任课教师：吴迪&黄聃

|  |  |  |  |
| --- | --- | --- | --- |
| 年级 | **2019级** | 专业（方向） | **计算机科学与技术** |
| 学号 | **18329015** | 姓名 | **郝裕玮** |
| 开始日期 | **2021.4.19** | 完成日期 | **2021.4.20** |

1. **实验题目**

并行计数排序的实现

1. **实验内容**

补充部分代码（即Count\_sort\_parallel函数，并行计数排序）如下所示（绝大部分代码已分析包含在注释中）

void Count\_sort\_parallel(int a[], int n, int thread\_count) {

   int i,j,count;//i,j用于循环

   //count用于记录每次需排序元素的位置（通过记录有多少个比待排序元素小或相等且位置小于待排序元素的元素个数）

   int\* temp=(int \*)malloc(n\*sizeof(int));//生成n个元素的int型数组temp

#  pragma omp parallel num\_threads(thread\_count) default(none) \

      private(i, j, count) shared(n, a, temp)

      //pragma的解释放在代码外进行文字解释

   {

#     pragma omp for

      //对于omp for同样放在代码外进行文字解释

      for(i=0;i<=n-1;i++){//把每个元素都进行一次计数排序

         count=0;//每次排序都要将count清零

         //以便统计有多少个比待排序元素小或相等且位置小于待排序元素的元素个数

         for(j=0;j<=n-1;j++){//开始统计有多少个比待排序元素小或相等且位置小于待排序元素的元素个数

            if(a[j]<a[i]||(a[j]==a[i]&&j<i)){//需要计数的两种情况

            //比待排序元素小或相等且位置小于待排序元素

               count++;//计数+1

            }

         }

         temp[count]=a[i];//将待排序元素的正确位置通过count放置到temp数组中

      }

#     pragma omp for

      for(i=0;i<=n-1;i++){

         a[i]=temp[i];//将排好序的数组temp内容复制到数组a中

         //用并行（omp for）来对串行计数排序中的memcpy(a, temp, n\*sizeof(int))进行时间和效率上的优化

      }

   }

   free(temp);//释放temp数组申请的空间

}  /\* Count\_sort\_parallel \*/

对于

#  pragma omp parallel num\_threads(thread\_count) default(none) \

      private(i, j, count) shared(n, a, temp)

对于预处理指令#pragma：

（1）在系统中加入预处理器指令一般是用来允许不是基本C语言规范部分的行为。不支持pragma的编译器会忽略pragma指令提示的那些语句，这样就允许使用pragma的程序在不支持它们的平台上运行。

（2）与所有的预处理器指令一样，pragma的默认长度是一行，因此如果有一个pragma在一行中放不下，那么新行需要被“转义”，在前面加一个反斜杠“\”即可。

（3）# pragma omp parallel代表之后的结构化代码块应该被多个线程并行执行。

（4）num\_threads(thread\_count)：对num\_threads进行初始化。

（5）default(none)：编译器将要求我们明确在这个块中使用的每个变量和已经在块之外声明的变量的作用域。(在一个块中声明的变量都是私有的，因为它们会被分配给线程的栈。)

对于

#     pragma omp for

（1）用parallel\_指令在外部循环前创建thread\_count个线程的集合。然后，我们不在每次内部循环执行时创建一组新的线程，而是使用一个for指令，告诉OpenMP用已有的线程组来并行化for循环。

接下来是对其余已完成函数的解析:

宏定义（代码分析已全部包含在注释中）

#define GET\_TIME(now) { \

   struct timeval t; \

   gettimeofday(&t, NULL); \

   now = t.tv\_sec + t.tv\_usec/1000000.0; \

}

////最后一行将us转换为s，统一单位

//该结构体用于计算多线程和串行计算pi的运行时间

主函数（绝大部分代码分析已全部包含在注释中）

int main(int argc, char\* argv[]) {

   int n, thread\_count;//数组元素个数和线程数

   int \*a, \*copy;//对数组a进行排序，最终结果复制到copy中

   double start, stop;//用于计算运行时间

   /\* please choose terms 'n', and the threads 'thread\_count' here. \*/

   n = 10;

   thread\_count = 4;

   /\* You can also get number of threads from command line \*/

   //if (argc != 3) Usage(argv[0]);

   //Get\_args(argv, &thread\_count, &n);

   /\* Allocate storage and generate data for a \*/

   a = (int \*)malloc(n\*sizeof(int));//生成n个元素的int型数组a

   Gen\_data(a, n);//为a数组生成数据

   /\* Allocate storage for copy \*/

   copy = (int \*)malloc(n\*sizeof(int));//生成n个元素的int型数组copy

   /\* Serial count sort \*/

   memcpy(copy, a, n\*sizeof(int));//将排序好的数组a的结果复制到copy中

#  ifdef DEBUG

   Print\_data(copy, n, "Original: Serial sort a");

#  endif

   GET\_TIME(start);//得到串行计数排序运行的开始时间

   Count\_sort\_serial(copy, n);//串行计数排序

   GET\_TIME(stop);//得到串行计数排序运行的结束时间

#  ifdef DEBUG

   Print\_data(copy, n, "Sorted: Serial sort a");

#  endif

   if (!Check\_sort(copy, n))//检查排序是否成功

      printf("Serial sort failed\n");

   printf("Serial run time: %e\n\n", stop-start);//打印串行计数排序运行时间

   /\* Parallel count sort \*/

   memcpy(copy, a, n\*sizeof(int));//将排序好的数组a的结果复制到copy中

#  ifdef DEBUG

   Print\_data(copy, n, "Original: Parallel qsort a");

#  endif

   GET\_TIME(start);//得到并行计数排序运行的开始时间

   Count\_sort\_parallel(copy, n, thread\_count);//并行计数排序

   GET\_TIME(stop);//得到并行计数排序运行的结束时间

#  ifdef DEBUG

   Print\_data(copy, n, "Sorted: Parallel sort a");

#  endif

   if (!Check\_sort(copy, n))//检查排序是否成功

      printf("Parallel sort failed\n");

   printf("Parallel run time: %e\n\n", stop-start);//打印并行计数排序运行时间

   /\* qsort library \*/

   memcpy(copy, a, n\*sizeof(int));//将排序好的数组a的结果复制到copy中

#  ifdef DEBUG

   Print\_data(copy, n, "Original: Library qsort a");

#  endif

   GET\_TIME(start);//得到快排运行的开始时间

   Library\_qsort(copy, n);//快排

   GET\_TIME(stop);//得到快排运行的结束时间

#  ifdef DEBUG

   Print\_data(copy, n, "Sorted: Library qsort a");

#  endif

   if (!Check\_sort(copy, n))//检查排序是否成功

      printf("Library sort failed\n");

   printf("qsort run time: %e\n", stop-start);//打印快排运行时间

   free(a);//释放a数组

   free(copy);//释放copy数组

   return 0;

}  /\* main \*/

对于各种

#  ifdef DEBUG

#  endif

在工程设置里有一些设置会对该工程自动产生一系列的宏，用以控制程序的编译和运行。如果你把代码夹在#ifdef DEBUG和对应的#endif中间，那么这段代码只有在调试（DEBUG）下才会被编译。也就是说，如果你在RELEASE模式下，这些代码根本就不会存在于你的最终代码里头。

生成数据（代码分析已全部包含在注释中）

/\*---------------------------------------------------------------------

 \* Function:  Gen\_data

 \* Purpose:   Generate random ints in the range 1 to n

 \* In args:   n: number of elements

 \* Out arg:   a: array of elements

 \*/

void Gen\_data(int a[], int n) {//生成需要排序的数据

   int i;

   for (i = 0; i < n; i++)

      a[i] = random() % n + 1; // (double) RAND\_MAX;

      //随机生成n个数据

#  ifdef DEBUG

   Print\_data(a, n, "a");

#  endif

}  /\* Gen\_data \*/

串行计数排序（代码分析已全部包含在注释中）

/\*---------------------------------------------------------------------

 \* Function:     Count\_sort\_serial

 \* Purpose:      sort elements in an array using count sort

 \* In args:      n: number of elements

 \* In/out arg:   a: array of elements

 \*/

void Count\_sort\_serial(int a[], int n) {//与Count\_sort\_parallel内容几乎一致，不再解释。

   int i, j, count;

   int\* temp = (int \*)malloc(n\*sizeof(int));

   for (i = 0; i < n; i++) {

      count = 0;

      for (j = 0; j < n; j++)

         if (a[j] < a[i])

            count++;

         else if (a[j] == a[i] && j < i)

            count++;

      temp[count] = a[i];

   }

   memcpy(a, temp, n\*sizeof(int));

   free(temp);

}  /\* Count\_sort\_serial \*/

快速排序（代码分析已全部包含在注释中）

/\*---------------------------------------------------------------------

 \* Function:     Library\_qsort

 \* Purpose:      sort elements in an array using qsort library function

 \* In args:      n: number of elements

 \* In/out arg:   a: array of elements

 \*/

void Library\_qsort(int a[], int n) {

   qsort(a, n, sizeof(int), My\_compare);//快排函数

   //函数原型为：void qsort(void\* base,size\_t num,size\_t size,int (\*compar)(const void\*,const void\*));

   //void\* base：指向要排序的数组的第一个对象的指针，将其转换为void\*。

   //size\_t num：base指向的数组中元素的数量，size\_t 是无符号整数类型。

   //size\_t size：数组中每个元素的大小（以字节为单位），size\_t 是无符号整数类型。

   //int (\*compar)(const void\*,const void\*)：指向比较两个元素的函数的指针。

   //该函数被重复调用qsort比较两个元素。它应遵循以下原型：

   //int compar (const void\* p1, const void\* p2);

   //<0  p1放到p2之前

   //=0  p1,p2位置不变

   //>0  p1放到p2之后

}  /\* Library\_qsort \*/

快排第四个参数的函数（代码分析已全部包含在注释中）

/\*---------------------------------------------------------------------

 \* Function:     My\_compare

 \* Purpose:      compare integer elements for use with qsort function

 \* In args:      element a, element b

 \* Return val:   positive if a > b, negative if b > a, 0 if equal

 \*/

int My\_compare(const void\* a, const void\* b) {

   const int\* int\_a = (const int\*) a;

   const int\* int\_b = (const int\*) b;

   return (\*int\_a - \*int\_b);//具体含义可见Library\_qsort函数中的解释

}  /\* My\_compare \*/

Debug时打印数组信息（代码分析已全部包含在注释中）

/\*---------------------------------------------------------------------

 \* Function:  Print\_data

 \* Purpose:   print an array

 \* In args:   a: array of elements

 \*            n: number of elements

 \*            msg: name of array

 \*/

void Print\_data(int a[], int n, char msg[]) {//debug时打印数组相关信息

   int i;

   printf("%s = ", msg);//打印数组名

   for (i = 0; i < n; i++)

      printf("%d ", a[i]);//按顺序打印数组元素

   printf("\n");

}  /\* Print\_data \*/

检查排序结果（代码分析已全部包含在注释中）

/\*---------------------------------------------------------------------

 \* Function:  Check\_sort

 \* Purpose:   Determine whether an array is sorted

 \* In args:   a: array of elements

 \*            n: number of elements

 \* Ret val:   true if sorted, false if not sorted

 \*/

int  Check\_sort(int a[], int n) {//检查是否排序成功

   int i;

   for (i = 1; i < n; i++)

      if (a[i-1] > a[i]) return 0;//若a[i-1]>a[i]则证明升序排序失败

   return 1;

}  /\* Check\_sort \*/

Get\_args和Usage为命令行编译程序并输入数组元素个数和线程数，与本题关系不大，不再赘述（因为主函数中已有可以直接修改这两项值的语句）

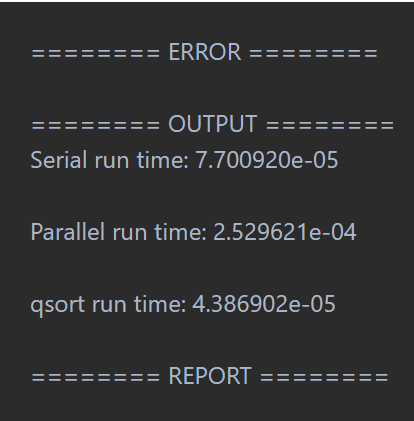
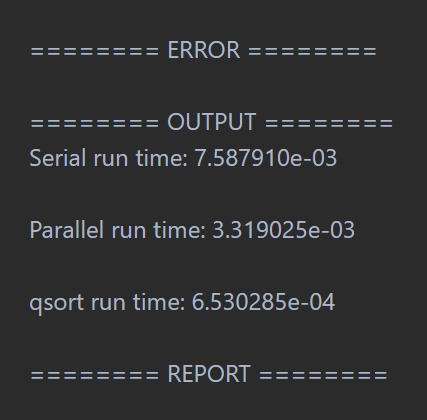
   /\* please choose terms 'n', and the threads 'thread\_count' here. \*/

   n = 10;

   thread\_count = 4;

1. **实验结果**

设置数组元素个数分别为100，1000，10000，线程数不变，均为4：

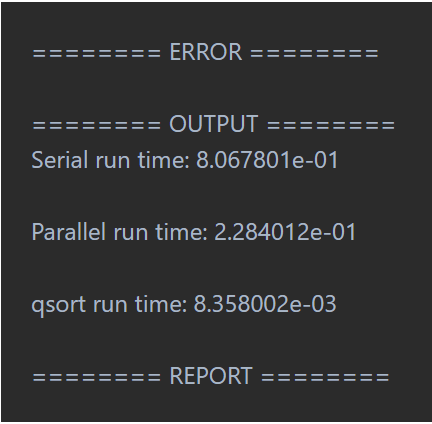
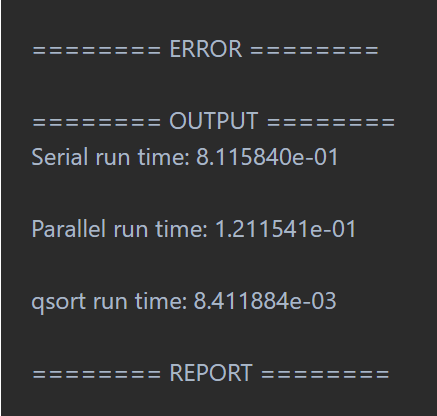
  

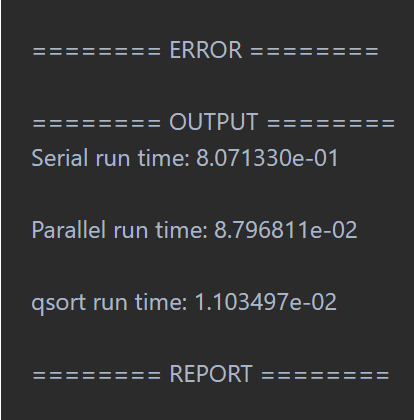
由上述结果可知，数组元素个数增加，串行，并行，快排三种排序的运行时间均会增加。

数组元素个数较小时，速率：快排>串行>并行（因为对于并行来说，OpenMP的parallel region结束时，线程之间需要同步：即主线程需要等待所有其他线程完成工作之后才能继续，这个过程可以称做barrier。该项时间在元素个数较小时对最终运行时间影响较大）

数组元素个数较大时，速率：快排>并行>串行

设置线程数分别为4，8，16，数组元素个数不变，均为10000：



由上述结果可知，线程数增加，并行的运行时间会减少。串行和快排的时间几乎不变。