《数组拼接》

--测评报告

洪瑶　1801111621

目录

一、并行算法描述

输入参数为：p（线程数量）,n（2^n 大小的数组）,m（2*2^m 拼接指示数组）;

数组拼接的数学模型为：

共有 M 个数组片段，每个片段由若干个 64 位的正整数组成，第 k 个片段中元素的最大值要小于第 k+1 个片段中元素的最小值。现将这些片段被存储数组 A 中；并用数组 B 记录了各片段在 A 中的分布，B 的长度为 2M，每个元素为 64 位的整数。B[2i]是数组片段的序号，B[2i+1]是第 B[2i]个的长度。数组 A 中依次存储第 B[0]号片段、第 B[2]号片段、第 B[4]号片段、.....、第 B[2M-2]号片段。

设计的并行算法为：

Step1： 对数组 B 进行分配，在每个线程上分别分配 2*2^m/p 个大小的数组（求余后若有剩余，优先分配给给靠前的线程多一组处理）。

Step2： 创建一个数组，将 B 重新保存为一个按偶数位依次递增的新数组 pp，这个过程每个线程分别分配 2*2^m/p 个大小规模的 B 的子数组进行求解得到 pp（求余后若有剩余，优先分配给给靠前的线程多一组处理）。因此 pp 格式为[0,x0,1,x1,2,x2…]。

Step3: 同 step2 中分配计算资源到各个线程，计算 qqa 数组和 qq 数组，其中 qqa 为　对 B 数组的偶数为进行累加求和，qq 为对 pp 数组的偶数位进行累加求和。

Step4：同 step2 中分配计算资源到各个线程，并构建一个结构体，

结构体中有三个成员，成员 1 为 B 的偶次位，成员 2 为 B 的奇次位，成员 3 为 qq,即每个片段应该在拼接完成后的起始位置。

Step5: 同 step2 中分配计算资源到各个线程，在每个线程中分别用 qsort 函数进行片段内的大小整理。

Step6: 将整理后的 A 按结构体函数中保存的起始位置进行拷贝，得到最终拼接完成的 C 数组。

二、改变线程数量和迭代空间规模，所编写并行程序的评测结果

（1）、控制 P=8，改变 N.

N=10:

```
***************value evaluation*****************
array-size = 2^10  segment_number=2^8
congratulations, result of your impl. is correct
************performance evaluation**************
reference serial impl.  : time_cost=5.0048e-04
reference parallel impl.: time_cost=4.7754e-04 speedup=1.048
your impl.              : time_cost=4.6027e-04 speedup=1.087
```

N=14:

```
***************value evaluation*****************
array-size = 2^14  segment_number=2^8
congratulations, result of your impl. is correct
************performance evaluation**************
reference serial impl.  : time_cost=2.9971e-03
reference parallel impl.: time_cost=1.4083e-03 speedup=2.128
your impl.              : time_cost=1.1878e-03 speedup=2.523
```

N=18:

```
***************value evaluation*****************
array-size = 2^18  segment_number=2^8
congratulations, result of your impl. is correct
************performance evaluation**************
reference serial impl.  : time_cost=4.5812e-02
reference parallel impl.: time_cost=1.5056e-02 speedup=3.043
your impl.              : time_cost=1.4610e-02 speedup=3.136
```

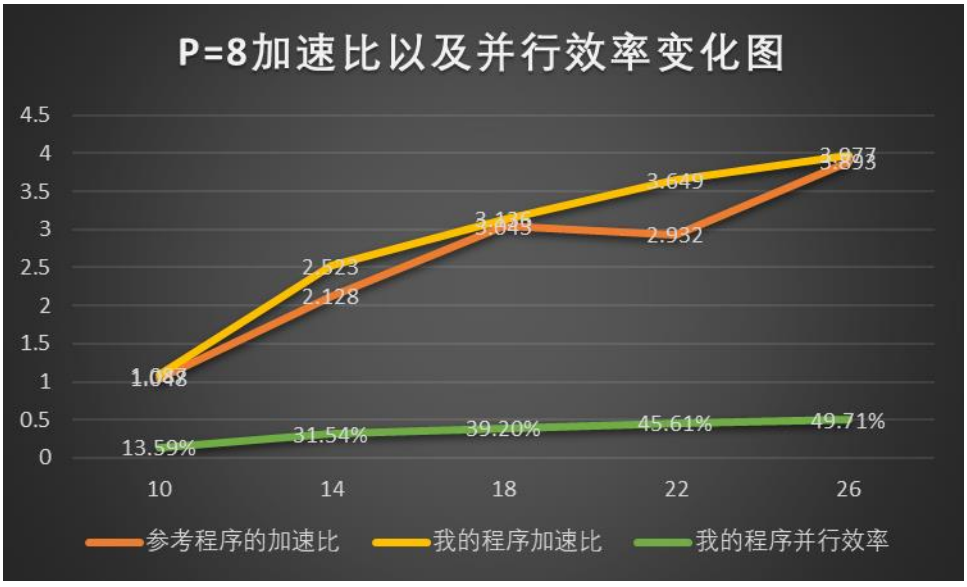
N=22：

```
***************value evaluation*****************
array-size = 2^22  segment_number=2^9
congratulations, result of your impl. is correct
************performance evaluation**************
reference serial impl.  : time_cost=5.0991e-01
reference parallel impl.: time_cost=1.7389e-01 speedup=2.932
your impl.              : time_cost=1.3973e-01 speedup=3.649
```

N=26：

```
***************value evaluation*****************
array-size = 2^26  segment_number=2^12
congratulations, result of your impl. is correct
************performance evaluation**************
reference serial impl.  : time_cost=7.8701e+00
reference parallel impl.: time_cost=2.0216e+00 speedup=3.893
your impl.              : time_cost=1.9791e+00 speedup=3.977
```

加速比变化图：



（2）、控制 N=20，改变 P

P=2：

```
***************value evaluation*****************
array-size = 2^20  segment_number=2^8
congratulations, result of your impl. is correct
************performance evaluation**************
reference serial impl.  : time_cost=1.4181e-01
reference parallel impl.: time_cost=9.7163e-02 speedup=1.459
your impl.              : time_cost=8.7660e-02 speedup=1.618
```

P=4：

```
***************value evaluation*****************
array-size = 2^20  segment_number=2^8
congratulations, result of your impl. is correct
************performance evaluation**************
reference serial impl.  : time_cost=1.6060e-01
reference parallel impl.: time_cost=5.3615e-02 speedup=2.996
your impl.              : time_cost=4.7615e-02 speedup=3.373
```

P=6：

```
 ***************value evaluation*****************
array-size = 2^20  segment_number=2^8
congratulations, result of your impl. is correct
************performance evaluation**************
reference serial impl.  : time_cost=1.3796e-01
reference parallel impl.: time_cost=5.6003e-02 speedup=2.463
your impl.              : time_cost=5.3457e-02 speedup=2.581
```

P=8：

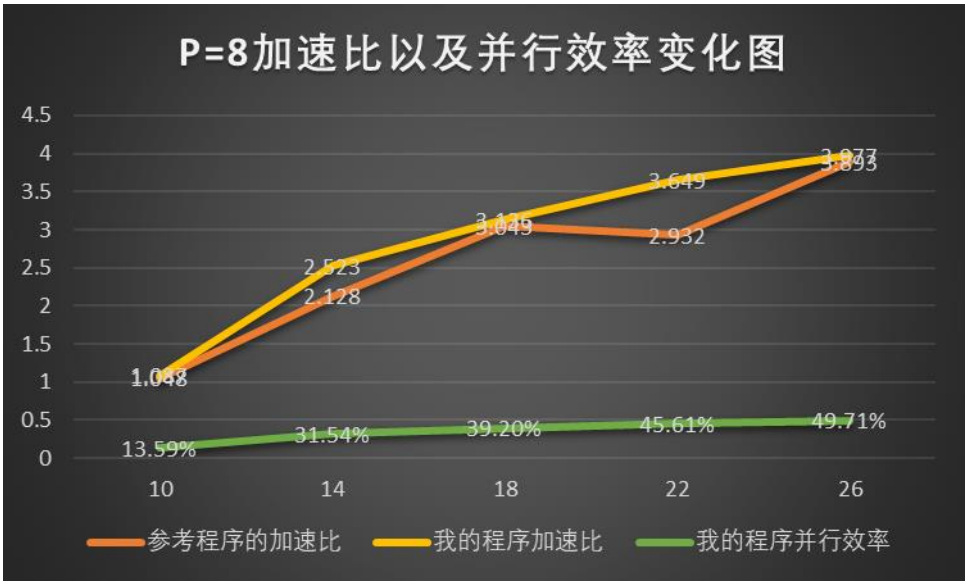
```
***************value evaluation*****************
array-size = 2^20  segment_number=2^8
congratulations, result of your impl. is correct
************performance evaluation**************
reference serial impl.  : time_cost=1.7480e-01
reference parallel impl.: time_cost=5.4652e-02 speedup=3.198
your impl.              : time_cost=4.4433e-02 speedup=3.934
```

P=10：

```
***************value evaluation*****************
array-size = 2^20  segment_number=2^8
congratulations, result of your impl. is correct
************performance evaluation**************
reference serial impl.  : time_cost=1.6031e-01
reference parallel impl.: time_cost=5.4862e-02 speedup=2.922
your impl.              : time_cost=4.5355e-02 speedup=3.535
```

加速比变化图：

三、评测结果分析



上图为 P=8 时加速比随着问题规模的变化图，我的程序随着计算规模的增大，加速比也越来越大，并行效率也越来越高。而参考程序的加速比随着并行规模的进行加速比也呈现上升趋势，但是不稳定，中间下降了一次。在程序中，计算规模越大，由于展开资源分配，锁操作等操作的时间占比就会相应减少，因此加速比和并行效率增加。

上图为控制 N=20，加速比随着线程数量的变化情况，可以看

到和加速比随着线程增加的变化趋势大概一致，在 p=6 时会比较小，着可能是因为 6 的倍数不是 2 的 n 次方的原因，导致计算资源分配不均匀，加速比相应减小。并行效率随着 P 的增加并没有增加，可见线程增加的同时，join 和 barrier 同步函数导致的等待时间等其他开销时间也相应增加，从而并不会导致加速比随着线程数量的增加而有明显的提升。

四、Pthread 程序

```
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <vector>
#include <math.h>
#include "fio.h"

int64_t n,n_num,m,m_num,sizeofA,sizeofB,sizeofC;
int32_t thread_num,threadid;
int64_t *A,*B,*C,*pp,*qq,*qqa;
pthread_barrier_t barrier;
struct SGROUP {int64_t pianduan;int64_t geshu;int64_t locA;};
//pianduan 为该片段，geshu 为该片段的个数，locA 为排序后该片段的首位在 A 中的位置
struct SGROUP *groupA;
//比较函数指针
int myCompar(const void *arg1,const void *arg2){
    int64_t *pa=(int64_t*)arg1,*pb=(int64_t*)arg2;
    return *pa>*pb;
}
//**********************//
//**********子**********//
//**********线**********//
//**********程**********//
//**********************//
void  *worker(void *arg) {
    int64_t i,j,lb,ub;
    int myID = __sync_fetch_and_add(&threadid, 1);
```

```
//    printf("xianchengyfhdshd s");
    int64_t loc_size = (m_num /2)/thread_num;
    int64_t rest = (m_num /2)%thread_num;
    //**********给线程分配计算资源**********
    if (myID < rest) {
        lb = loc_size * myID + myID;
        ub = lb + loc_size + 1;
    } else {
        lb = loc_size * myID + rest;
        ub = lb + loc_size;
    }
    //**********将 B 数组两位两位地保存在 pp 中**********
    for (i =lb; i < ub; i++) {
        for (j = 0; j < m_num /2; j++) {
            if (B[2*j] == i) {
                pp[2*i]=B[2*j];
                pp[2*i+1]=B[2*j+1];
            }
        }
    }
    pthread_barrier_wait(&barrier);
//**********开辟空间得到 qqa（排序前累积）以及 qq（排序后累积）
**********
    if(pthread_barrier_wait(&barrier)==
PTHREAD_BARRIER_SERIAL_THREAD) {
        for (i = 1; i < m_num / 2; i++) {
            qqa[i] = qqa[i - 1] + B[2*i - 1];
            qq[i] = qq[i - 1] + pp[2*i - 1];
        }
    }
    pthread_barrier_wait(&barrier);
//**********保存结构体的前两个数据**********
    for (i = lb; i <ub; i++) {
        groupA[i].pianduan = B[2*i];
        groupA[i].geshu = B[2*i+1];
    }
    pthread_barrier_wait(&barrier);
    //**********保存结构体的第三个数据**********
    for (i = lb; i <ub; i++) {
        groupA[i].locA = qq[groupA[i].pianduan];
    }
    pthread_barrier_wait(&barrier);
    //**********先对每个片段排序，再拷贝排序前 A 的片段到 C 的相应
的位置**********
```

```
     for (i = lb; i < ub; i++) {
         qsort(&A[qqa[i]], groupA[i].geshu, sizeof(int64_t),
myCompar);
         memcpy(&C[groupA[i].locA], &A[qqa[i]], sizeof(int64_t) *
groupA[i].geshu);
     }

    return (void *) 0;
}

int main(int argc, char *argv[ ] ){
    int i;
    thread_num=atoi(argv[1]);        //      设置 P
    n=atoll(argv[2]);                //      设置 N
    n_num=((int64_t) 1<<n);          //        设置 2^N 大小的数组
    m=atoll(argv[3]);                //      设置 M
    m_num=((int64_t) 1<<(m+1));      //      设置 2*2^M 大小的数
组
    threadid=0;
    pp= (int64_t *)malloc(sizeof(int64_t)*n_num);   //将 B 重新排序后
的数组
    qq= (int64_t *)malloc(sizeof(int64_t)*m_num/2);   //A 排序后 B 中
个数的累加数组
    qqa= (int64_t *)malloc(sizeof(int64_t)*m_num/2);   //A 排序前 B 中
个数的累加数组
    qq[0]=0;
    qqa[0]=0;
    sizeofA = (sizeof(int64_t)) *n_num;              //A 数组空间大小
    sizeofB = (sizeof(int64_t)) *m_num;              //B 数组空间大小
    sizeofC = (sizeof(int64_t)) *n_num;              //C 数组空间大小
    A= (int64_t *)malloc(sizeof(int64_t)*n_num);     //开辟 A 数组空间
    B= (int64_t *)malloc(sizeof(int64_t)*m_num);     //开辟 B 数组空间
    C= (int64_t *)malloc(sizeof(int64_t)*n_num);     //开辟输出数组 C
数的组空间
    groupA=(struct SGROUP*)malloc((m_num/2)*sizeof(SGROUP));
    input_data(A, sizeofA, B, sizeofB);        //导入 A、B。
    for (i = 0; i <8; i++) {
    pthread_barrier_init(&barrier,NULL,thread_num);    //设置栅樟
    pthread_t *threads = new pthread_t[thread_num];
    for(int i=0; i<thread_num; i++)
pthread_create(&(threads[i]),NULL,worker,&thread_num);
    for(int i=0; i<thread_num; i++)  pthread_join(threads[i],NULL);
    output_data(C,sizeofC ); //输出 C
    delete[ ] threads;
```

```
    pthread_barrier_destroy(&barrier);       //删除栅栏*/
    return EXIT_SUCCESS;
}



//
// Created by 洪垚 on 2018/10/29.
//
```