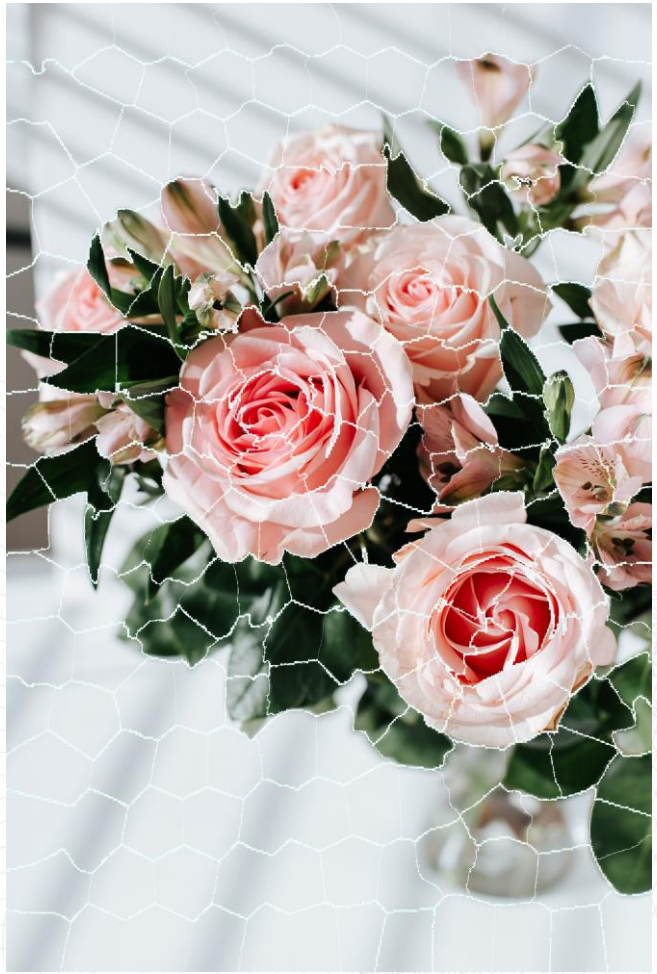
ACM中国－国际并行计算挑战赛

ACM-China International Parallel Computing Challenge

目录 CONTENTS



参赛队简介

参赛队伍编号：IPCC20216734　　参赛队伍名称：Result0

参赛队伍学校：武汉大学

指导老师：邓娟 武汉大学计算机学院 副教授

参赛队员：吴智琨 陈洲

应用程序运行的硬件环境和软件环境——硬件信息

CPU	Intel 10875H	AMD EPYC 7452	Intel Xeon Glod 8180
Core(s) per socket	8	32	28
Thread(s) per core	2	1	2
Sockets (numa)	1	2	2
Frequency	2.3 ~ 5.1(4.3 all) GHz	2.35 ~ 3.35 GHz	2.5 ~ 3,8 GHz
L1d/L1i cache	256KB/256KB	32KB/32KB	32KB/32KB
L2 cache	2MB	512KB	1MB
L3 cache	16MB	16MB	38MB
AVX2-GFLOPS	< 700	2406.4*2	2240*2
stream	24.1GB/s	244.9GB/s	137.4GB/s
Max bandwidth	45.8 GB/s	400 GB/s	250 GB/s
	开发/编译平台	运行平台	参考参数

开发/编译平台：INTEL i7-10875H
信息简述：8核16线程，开发及初步调优使用

运行平台：AMD EPYC 7452
信息简述：NUMA架构，32核x2，无超线程，支持fma, avx2，比赛实际运行平台

应用程序运行的硬件环境和软件环境——软件信息

ACM中国
国际并行计算挑战赛

```
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.2 LTS
Release:        20.04
Codename:       focal
```

```
LSB Version:    :core-4.1-amd64:core-4.1-noarch:cxx-4.1-amd64:cxx-4.1-noarch:desktop-4.1-amd64:desktop-4.1-noarch:languages-4.1-amd64:languages-4.1-noarch:printing-4.1-amd64:printing-4.1-noarch
Distributor ID: CentOS
Description:    CentOS Linux release 7.9.2009 (Core)
Release:        7.9.2009
Codename:       Core
```

开发/编译平台：
系统：WSL2 Ubuntu 20.04 LTS
Gcc：9.3.0
Glibc：2.31

运行平台：
系统：CentOS Linux 7.9
Gcc：4.8.5
Glibc：2.17(本机自带版本) 2.31(静态链接版本)

编译环境依赖于：
gcc>=9，glibc>=2.30

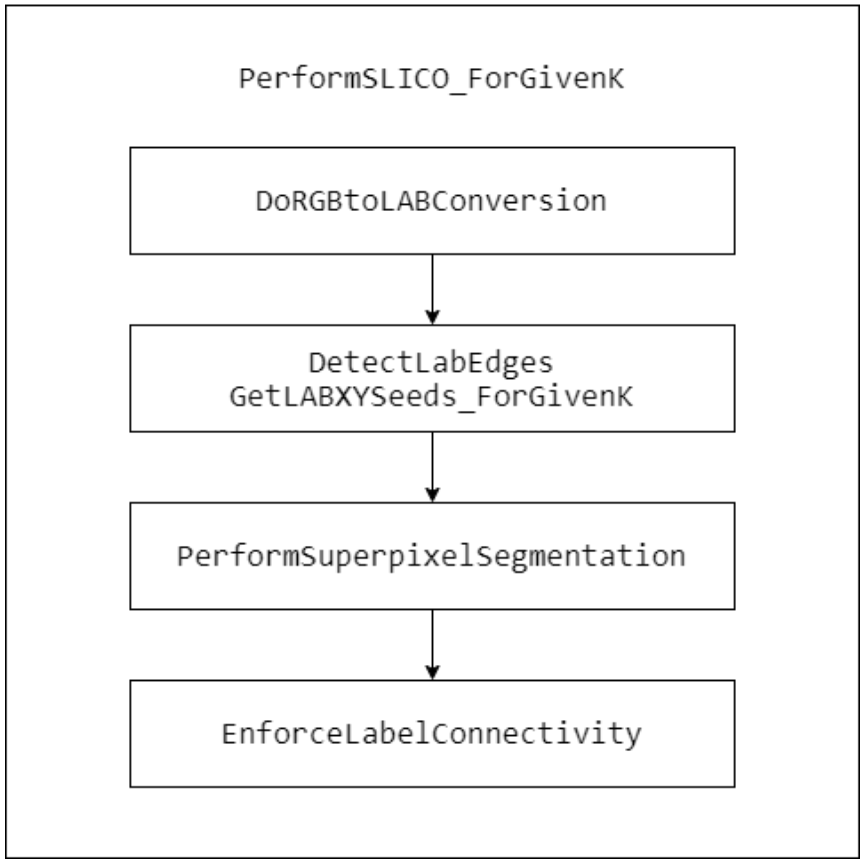
应用程序的代码结构

SLIC/	描述
SLIC.h	超像素算法头文件
SLIC.cpp	超像素算法主要实现
utils.h	辅助头文件
utils.cpp	辅助函数实现
input_image.ppm	输入的图像文件
check.ppm	校验文件
Makefile	make 编译脚本
run.sh	运行脚本
SLIC.out	可执行文件
case1(dir)	原始测试，含源码及可执行文件
case2(dir)	测试2，含源码及可执行文件
case3(dir)	测试3，含源码及可执行文件

SLIC (Simple Linear Iterative Clustering)
超像素算法并行优化

```
Main(){
    ... Loadppm ...
    计时开始
    PerformSLICO_ForGivenK(...)
    计时结束
    ... check ...
}
```

应用程序的代码结构

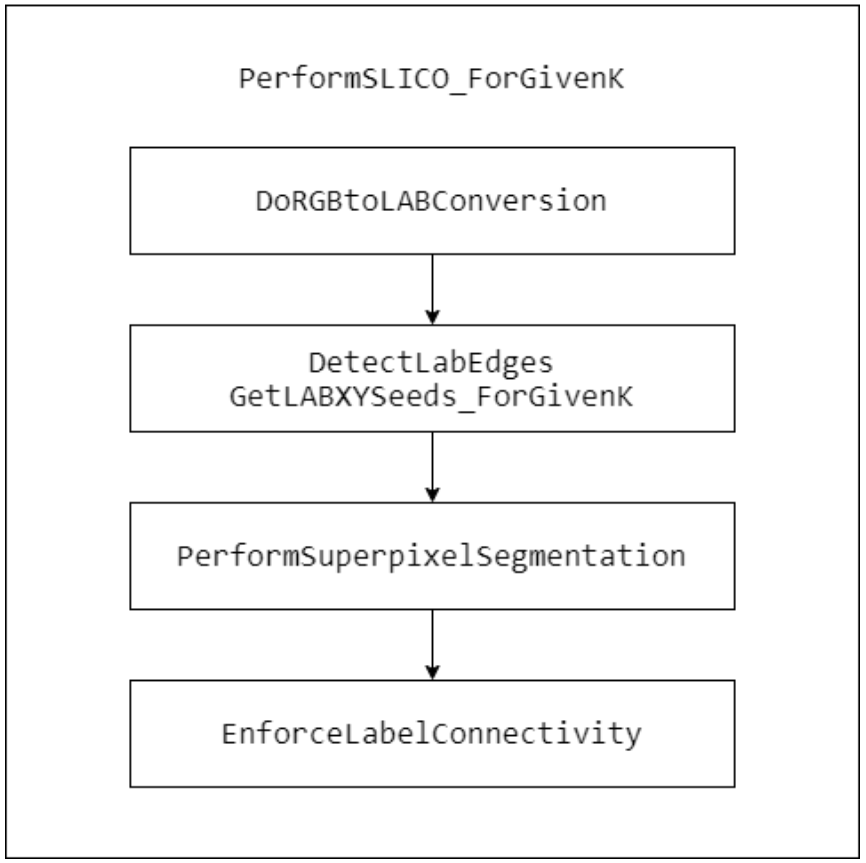


计时区仅有 SLIC::PerformSLICO_ForGivenK 一个函数

主要流程分为左图的四步：

1. 将 ubuff 中以 RGB 存储的像素转换为 LAB 格式
2. 在 LAB 空间做边缘检测，生成种子
3. 运行 K 聚类算法，迭代多次
4. 运用 BFS 算法遍历各连通聚类，重新标记。

优化方法



对四个主要流程进行优化，主要优化方法为：

1. 算法初步优化，提高串行运算速度
2. 向量化，利用 avx256 运算
3. 多线程并行化，利用 openMP 并行化 for 循环
4. 并行算法优化，提升数据局部性，同时大幅修改部分难以并行化的算法，添加前后辅助函数

注1：先更新编译器及标准库进行初步优化

注2：测试表明 MPI 通信时间高于优化后整体运算时间，故仅考虑单节点运算。

优化方法 基准建立

1. 以初赛赛题为基准，以平台默认方式为准，编译及运行方法如赛题所示，其中版本为 g++==4.8.5，glibc==2.17

程序使用方法

1）参考编译命令：g++ -std=c++11 SLIC.cpp -o SLIC

2） 运行命令：srun –p amd_256 –N 1 ./SLIC

```
[sca3207@ln121%bscc-a5 SLIC_BK]$ srun -p amd_256 -N 1 -t 10 ./SLIC.out
Computing time=25263 ms
There are 0 points' labels are different from original file.
```

2. 原代码开启-O3：
g++ -std=c++11 –O3 SLIC.cpp –o SLIC.out

```
[sca3207@ln121%bscc-a5 SLIC_BK]$ srun -p amd_256 -N 1 -t 10 ./SLIC.out
Computing time=8520 ms
There are 0 points' labels are different from original file.
```

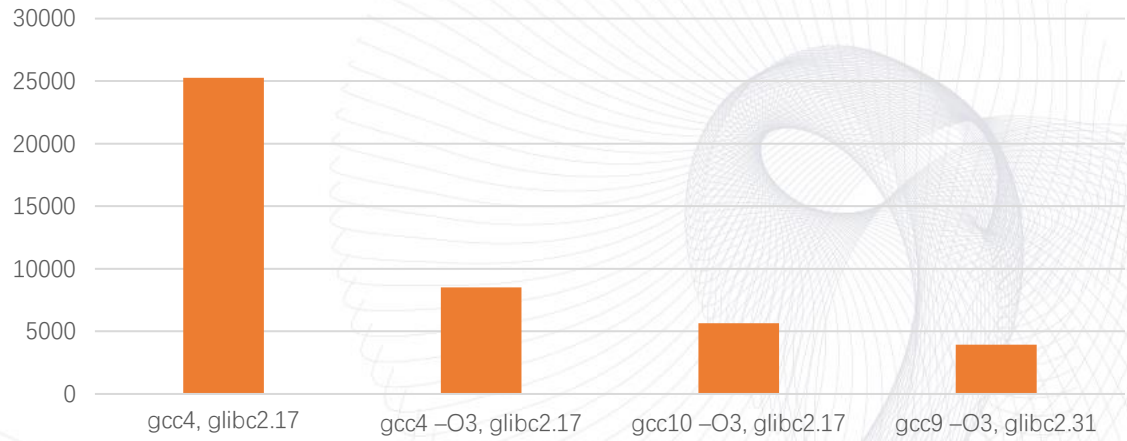
3. 原代码开启-O3，更换平台上的 gcc10：
g++ -std=c++11 –O3 SLIC.cpp –o SLIC.out

```
[sca3207@ln121%bscc-a5 SLIC_BK]$ srun -p amd_256 -N 1 -t 10 ./SLIC.out
Computing time=5635 ms
There are 0 points' labels are different from original file.
```

4. 原代码开启-O3，更换为本地的 gcc9，静态链接 glibc2.31：
g++ -std=c++11 –O3 SLIC.cpp –o SLIC.out

```
[sca3207@ln121%bscc-a5 SLIC_BK]$ srun -p amd_256 -N 1 -t 10 ./SLIC.out
Computing time=3920 ms
There are 0 points' labels are different from original file.
```

gcc4, glibc2.17	gcc4 –O3, glibc2.17	gcc10 –O3, glibc2.17	gcc9 –O3, glibc2.31
25263 ms	8520 ms	5635 ms	3920 ms



由于基准 1 性能较差，不利于优化指标参考，在优化方法章节，以上述基准4为优化基准作为对比。
之后，将以基准1为原始基准，基准4为优化基准。

优化方法

原流程简化节选

```
1. if(sR <= 10) r = sR/3294.6;
   else         r = pow(sR/269.025+0.055/1.055,2.4);

2. double xr = (r*0.4124564 + g*0.3575761 + b*0.1804375) / Xr;

3. if(xr > epsilon) fx = pow(xr, 1.0/3.0);
   else             fx = (kappa_m*xr + 16.0/116.0);

4. lvec[j] = 116.0*fy-16.0;
```

向量化流程简化节选

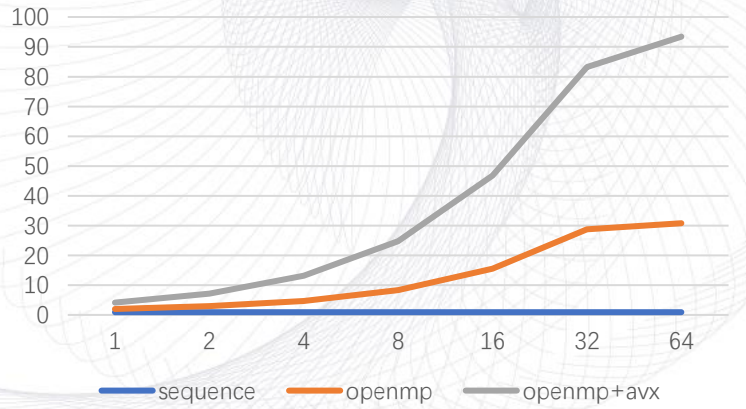
```
1. __m256d mm_r = _mm256_load_pd(block_r);

2. mm_r = _mm256_blendv_pd(_mm256_div_pd(mm_r, mm_rgb_xyz_cont[0]),
          _ZGVdN4vv_pow(_mm256_fmadd_pd(mm_r, mm_rgb_xyz_cont[1],
                                        mm_rgb_xyz_cont[2]), mm_rgb_xyz_cont[3]),
          _mm256_cmp_pd(mm_r, mm_rgb_xyz_cont[4], _CMP_GT_OQ));

3. auto mm_x = _mm256_mul_pd(mm_r, mm_rgb_xyz_mat[0]);
   mm_x = _mm256_fmadd_pd(mm_g, mm_rgb_xyz_mat[1], mm_x);
   mm_x = _mm256_fmadd_pd(mm_b, mm_rgb_xyz_mat[2], mm_x);

4. mm_x = _mm256_blendv_pd(_ZGVdN4vv_pow(mm_x, mm_xyz_lab_cont[5]),
          _mm256_fmadd_pd(mm_x, mm_xyz_lab_cont[4], mm_xyz_lab_cont[6]),
          _mm256_cmp_pd(mm_x, mm_xyz_lab_cont[3], _CMP_LE_OQ));
```

1. 在循环前加上 #pragma omp parallel for 来进行多线程并行化
2. 使用 avx256 进行向量化，一次同时操作4个 double 元素
3. 使用 _mm256_blendv_pd 三元运算符来实现 avx 下的 if 逻辑
4. 强制调用 glibc 中 mvec 的_ZGVdN4vv_pow 来实现向量化的乘方
5. 使用 fmadd 来替代 mul add 进行乘加运算，更好得发挥运算性能。
6. 使用 _mm256_stream_pd 绕过缓存直接写内存



优化方法 DetectLabEdges && GetLABXYSeeds


```
void SLIC::PerturbSeeds(...){
...
for( int i = 0; i < 8; i++ )
{
    int nx = ox+dx8[i];//new x
    int ny = oy+dy8[i];//new y
    if( nx >= 0 && nx < m_width && ny >= 0 && ny < m_height)
    {
        int nind = ny*m_width + nx;
        if( edges[nind] < edges[storeind])
        {
            storeind = nind;
        }
    }
}
...
}
```

替代上方运算

```
if( nx >= 0 && nx < m_width && ny >= 0 && ny < m_height)
{
    int nind = ny*m_width + nx;
    edges_f++;
    // printf("%d %d ", nind, storeind);
    // if( edges[nind] < edges[storeind])
    if(getEdge(ny, nx) < getEdge(storey, storex))
    {
        storeind = nind;
        storex = nx;
        storey = ny;
    }
}
```

去除此运算

```
void SLIC::DetectLabEdges(...)
{
    int sz = width*height;

    edges.resize(sz,0);
    for( int j = 1; j < height-1; j++ )
    {
        for( int k = 1; k < width-1; k++ )
        {
            int i = j*width+k;

            double dx = (lvec[i-1]-lvec[i+1])*(lvec[i-1]-lvec[i+1]) +
                        (avec[i-1]-avec[i+1])*(avec[i-1]-avec[i+1]) +
                        (bvec[i-1]-bvec[i+1])*(bvec[i-1]-bvec[i+1]);

            double dy = (lvec[i-width]-lvec[i+width])*(lvec[i-width]-lvec[i+width]) +
                        (avec[i-width]-avec[i+width])*(avec[i-width]-avec[i+width]) +
                        (bvec[i-width]-bvec[i+width])*(bvec[i-width]-bvec[i+width]);

            //edges[i] = (sqrt(dx) + sqrt(dy));
            edges[i] = (dx + dy);
        }
    }
}
```

观察生成初始种子的过程，并不需要全部的 edge 信息，因此可以将全局 edges 计算去除变为按需计算。

注：此处未列出加速比，整体耗时已经小于毫秒量级

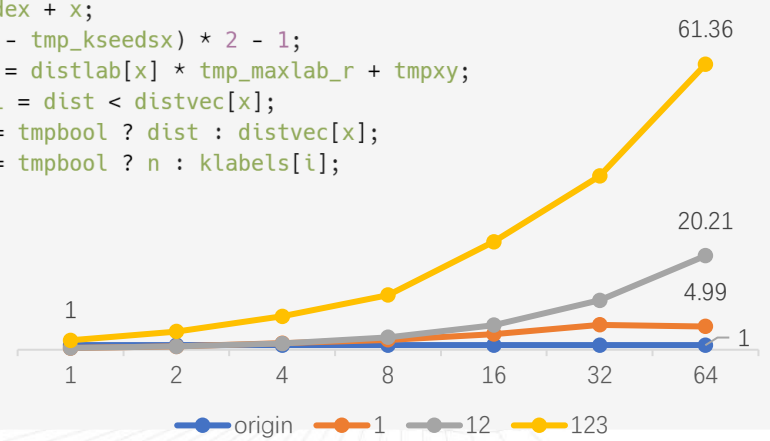
优化方法 PerformSuperpixelSegmentation

核心计算部分

```
for( int n = 0; n < numk; n++ ){
    int y1 = max(0,          (int)(kseedsy[n]-offset));
    int y2 = min(m_height,   (int)(kseedsy[n]+offset));
    int x1 = max(0,          (int)(kseedsx[n]-offset));
    int x2 = min(m_width,    (int)(kseedsx[n]+offset));
    for( int y = y1; y < y2; y++ ){
        for( int x = x1; x < x2; x++ ){
            int i = y*m_width + x;
            double l = m_lvec[i]; double a = m_avec[i]; double b = m_bvec[i];
            distlab[i] =    (l - kseedsl[n])*(l - kseedsl[n]) +
                            (a - kseedsa[n])*(a - kseedsa[n]) +
                            (b - kseedsb[n])*(b - kseedsb[n]);
            distxy[i] =     (x - kseedsx[n])*(x - kseedsx[n]) +
                            (y - kseedsy[n])*(y - kseedsy[n]);
            double dist = distlab[i]/maxlab[n] + distxy[i]*invxywt;
            if( dist < distvec[i] ){
                distvec[i] = dist;
                klabels[i]  = n;
            }
        }
    }
}
```

1. 在遍历纵坐标循环前加上 #pragma omp parallel for 来进行多线程并行化
2. 修改循环层次，增大并行粒度
3. 删去不必要的 distxy 数组存储
   利用迭代运算简化 distxy 对应的实际计算
4. 综合优化其他部分并微调代码结构

```
#pragma omp parallel for
for (int y = ccymin; y < ccymax; y++) {
    // initialization
    for (int n = 0; n < numk; n++) {
        if ((int)(kseedsy[n] - offset) <= y && y < (int)(kseedsy[n] + offset)) {
            const int x1 = max(0, (int)(kseedsx[n] - offset));
            const int x2 = min(m_width, (int)(kseedsx[n] + offset));
            const double tmp_kseedsl = kseedsl[n];
            const double tmp_kseedsa = kseedsa[n];
            const double tmp_kseedsb = kseedsb[n];
            const double tmp_kseedsx = kseedsx[n];
            const double tmp_kseedsy = kseedsy[n];
            const double tmp_maxlab_r = (STEP * STEP) / maxlab[n];
            const double powery = (y - tmp_kseedsy) * (y - tmp_kseedsy);
            double tmpxy = (x1 - tmp_kseedsx) * (x1 - tmp_kseedsx)
                         + powery - (x1 * 2 - tmp_kseedsx * 2 - 1);
            for (int x = x1; x < x2; x++) {
                int i = yindex + x;
                distlab[x] = (m_lvec[i] - tmp_kseedsl) * (m_lvec[i] - tmp_kseedsl) +
                             (m_avec[i] - tmp_kseedsa) * (m_avec[i] - tmp_kseedsa) +
                             (m_bvec[i] - tmp_kseedsb) * (m_bvec[i] - tmp_kseedsb);
            }
            for (int x = x1; x < x2; x++) {
                int i = yindex + x;
                tmpxy += (x - tmp_kseedsx) * 2 - 1;
                double dist = distlab[x] * tmp_maxlab_r + tmpxy;
                auto tmpbool = dist < distvec[x];
                distvec[x] = tmpbool ? dist : distvec[x];
                klabels[i] = tmpbool ? n : klabels[i];
            }
        }
    }
    // reduction
    // clear some vars
}
}
```



1718 ms 344 ms 85 ms 28 ms

优化方法 PerformSuperpixelSegmentation

无用运算

```
for (int i = 0; i < sz; i++) {
    if (maxlab[klabels[i]] < distlab[i])
        maxlab[klabels[i]] = distlab[i];
    if (maxxy[klabels[i]] < distxy[i])
        maxxy[klabels[i]] = distxy[i];
}

...    原来为单层遍历

for (int j = 0; j < sz; j++) {
    int temp = klabels[j];

    sigmal[klabels[j]] += m_lvec[j];
    sigmaa[klabels[j]] += m_avec[j];
    sigmab[klabels[j]] += m_bvec[j];
    sigmax[klabels[j]] += (j % m_width);
    sigmay[klabels[j]] += (j / m_width);

    clustersize[klabels[j]]++;
}
```

Mod 运算耗时

除法运算耗时

改为双层yx遍历

前一步中的核心运算在此

```
#pragma omp parallel for num_threads(THE_THREAD_NUMS_DIST)
for(int li = 0; li<THE_THREAD_NUMS_DIST; li++){
    ...
    for(int y = ccymin; y<ccymax; y++){
        ...
        for(int n=0; n<numk; n++){...}
        const int thread_num = omp_get_thread_num();
        for (int x = 0; x < m_width; x++) {
            const int index = (yindex + x);
            const int kl = klabels[index];
            if(maxlab_p[thread_num][kl] < distlab_y[x])
              maxlab_p[thread_num][kl] = distlab_y[x];

            sigmass[thread_num][0][kl] += m_lvec[index];
            sigmass[thread_num][1][kl] += m_avec[index];
            sigmass[thread_num][2][kl] += m_bvec[index];
            sigmass[thread_num][3][kl] += x;
            sigmass[thread_num][4][kl] += y;
            clustersizess[thread_num][kl]++;
        }
        ...
    }
}
```

减少了耗时运算

这段代码本质上即为归约，优化方法如下：

1. 通过两层循环消除耗时的 %、/ 运算
2. 合并循环，将不必要的 maxxy 计算去除
3. 以 openmp 作多线程并行化
4. 整体和前述过程合并，提高局部性和并行性

优化方法　EnforceLabelConnectivity

ACM中国 国际并行计算挑战赛

```
for (int j = 0; j < height; j++) {
  for (int k = 0; k < width; k++) {
    if (0 > nlabels[oindex]) {
      nlabels[oindex] = label;
      xvec[0] = k;
      yvec[0] = j;
      for (int n = 0; n < 4; n++) {
        int x = xvec[0] + dx4[n];
        int y = yvec[0] + dy4[n];
        if ((x >= 0 && x < width) && (y >= 0 && y < height)) {
          int nindex = y * width + x;
          if (nlabels[nindex] >= 0)
            adjlabel = nlabels[nindex];
        }
      }
      int count(1);
      for (int c = 0; c < count; c++) {
        for (int n = 0; n < 4; n++) {
          int x = xvec[c] + dx4[n];
          int y = yvec[c] + dy4[n];
          if ((x >= 0 && x < width) && (y >= 0 && y < height)) {
            int nindex = y * width + x;
            if (0 > nlabels[nindex] && labels[oindex] == labels[nindex]) {
              xvec[count] = x;
              yvec[count] = y;
              nlabels[nindex] = label;
              count++;
            }
          }
        }
      }
      if (count <= SUPSZ >> 2) {
        for (int c = 0; c < count; c++) {
          int ind = yvec[c] * width + xvec[c];
          nlabels[ind] = adjlabel;
        }
        label--;
      }
      label++;
    }
    oindex++;
  }
}
```

优化效果：
100ms -> 7ms，14x

左侧为原代码，核心思路为进行 BFS，按照每个连通聚类第一个点出现的顺序重新打上标签，小于临界值的连通聚类合并入相邻连通聚类。此算法较难直接并行，且局部性差，优化方法如下

1. 直接并行 BFS 对连通聚类进行打标签。
2. 使用并查集对 BFS 的结果进行合并。
3. 顺序遍历检查并查集每个集合的大小，进行合并操作，同时获得全局顺序标签。
4. 并行遍历以映射新的标签。

原子量确保标签唯一

```
atomic_int label(0);

#pragma omp parallel for num_threads(THE_THREAD_NUMS_ENFORCE) schedule(static, 1)
for(int nt=0; nt<THE_THREAD_NUMS_ENFORCE; nt++){
    int ymin = (height / THE_THREAD_NUMS_ENFORCE) * nt;
    int ymax = (height / THE_THREAD_NUMS_ENFORCE) * (nt+1);
    if(nt == THE_THREAD_NUMS_ENFORCE-1){
        pheight = ymax;
        ymax = height;
    }
    int* xvec = xvec_p[nt];
    int* yvec = yvec_p[nt];
    memset(nlabels1+(ymin*width), -1, sizeof(int) * (ymax - ymin) * m_width);
    for(int y = ymin; y<ymax; y++){
        const int yindex = y*width;
        for(int x=0; x<width; x++){
            int oindex = yindex + x;
            if(0 > nlabels1[oindex] ){

                ... BFS ...

                labels_size[tmp_label] = count;
                labels_ivec[tmp_label] = oindex;
                labels_xyvec[tmp_label] = mpair(x, y);
            }
        }
    }
}
```

仅遍历每个聚类第一个点，避免了对全部像素的计算

```
for(int i=0; i<label; i++){
    const int y = labels_xyvec[i].y;
    const int x = labels_xyvec[i].x;
    const int index = y*width + x;
    int local_label = labels_map[nlabels1[index]];
    if(labels_map2[local_label] == -1){
        if(labels_size[local_label] <= SUPSZ >> 2){
            for(int n=0; n<4; n++){
                int tx = x + dx4[n];
                int ty = y + dy4[n];
                if( (tx >= 0 && tx < width) && (ty >= 0 && ty < height) ){
                    int nindex = ty*width + tx;
                    if(labels_map2[labels_map[nlabels1[nindex]]] != -1)
                        adjlabel = labels_map[nlabels1[nindex]];
                }
            }
            if(labels_map2[adjlabel] == -1) {
                labels_map2[local_label] = 0;
            }
            else{
                labels_map2[local_label] = labels_map2[adjlabel];
            }
        }
        else{
            labels_map2[local_label] = clabel;
            clabel++;
        }
    }
}
```

检查大小

以两级映射方式有效变换标签

程序运行结果


```
[sca3207@ln121%bscc-a5 SLIC_BK]$ srun -p amd_256 -N 1 -t 10 ./SLIC.out
Initial time = 40 ms
Conversion time = 3250 ms
DeleteEdges and Get_Seeds time = 289 ms
STEP = 227
Segmentation time = 20864 ms
EnforceLabelConnectivity time = 388 ms
Computing time=24833 ms
There are 0 points' labels are different from original file.
```

原代码 官方题目基准 默认平台环境
仅增添计时打印语句： 24833 ms

```
[sca3207@ln121%bscc-a5 SLIC_BK]$ srun -p amd 256 -N 1 -t 10 ./SLIC.out
srun: job 694299 queued and waiting for resources
srun: job 694299 has been allocated resources
Initial time = 3 ms
Conversion time = 1359 ms
DeleteEdges and Get Seeds time = 30 ms
STEP = 227
Segmentation time = 2378 ms
EnforceLabelConnectivity time = 100 ms
Computing time=3874 ms
There are 0 points' labels are different from original file.
```

接上，更换gcc9 glibc2.31
开启-O3： 3874 ms

1x -> 6.4x -> 500x，仅代码部分就实现了约80倍加速，
最终总体达到约 500 倍加速，

```
[sca3207@ln121%bscc-a5 SLIC]$ chmod +x ./SLIC.out
[sca3207@ln121%bscc-a5 SLIC]$ export OMP_PLACES=cores
[sca3207@ln121%bscc-a5 SLIC]$ srun -p amd 256 -N 1 -t 10 ./SLIC.out
width = 2599, height = 3898
sz = 10130902
Initial time = 0 ms
Conversion time = 13 ms
DeleteEdges and Get Seeds time = 0 ms
numk = 196
Dist iter time=4.743(4) ms
Dist iter time=7.144(2) ms
Dist iter time=9.283(2) ms
Dist iter time=11.404(2) ms
Dist iter time=13.625(2) ms
Dist iter time=15.853(2) ms
Dist iter time=18.042(2) ms
Dist iter time=20.183(2) ms
Dist iter time=22.32(2) ms
Dist iter time=24.474(2) ms
Computing time=24 ms
STEP = 227
Segmentation time = 25 ms
EC1 time=0 ms
EC2 time=3 ms
EC3 time=0 ms
EC4 time=1 ms
EnforceLabelConnectivity time = 7 ms
Computing time=47 ms
There are 0 points' labels are different from original file.
```

250x

834x

55x

最终优化 47 (47~50) ms
各部分加速倍率：≈∞，250，≈∞，834，55

程序运行结果 测试 case2 和 case3


```
[sca3207@ln121%bscc-a5 case2]$ srun -p amd_256 -N 1 -t 10 ./SLIC.out
Initial time = 96 ms
Conversion time = 7460 ms
DeleteEdges and Get_Seeds time = 666 ms
STEP = 246
Segmentation time = 49757 ms
EnforceLabelConnectivity time = 886 ms
Computing time=58868 ms
There are 0 points' labels are different from original file.
```

58868ms

```
[sca3207@ln121%bscc-a5 case3]$ srun -p amd_256 -N 1 -t 10 ./SLIC.out
Initial time = 29 ms
Conversion time = 2167 ms
DeleteEdges and Get_Seeds time = 198 ms
STEP = 222
Segmentation time = 14832 ms
EnforceLabelConnectivity time = 270 ms
Computing time=17498 ms
There are 0 points' labels are different from original file.
```

17498ms

```
[sca3207@ln121%bscc-a5 case2]$ srun -p amd_256 -N 1 -t 10 ./SLIC.out
width = 4000, height = 6000
sz = 24000000
Initial time = 0 ms
Conversion time = 28 ms
DeleteEdges and Get_Seeds time = 0 ms
Segmentation time = 56 ms
EnforceLabelConnectivity time = 11 ms
Computing time=97 ms
There are 0 points' labels are different from original file.
```

97ms

```
[sca3207@ln121%bscc-a5 case3]$ srun -p amd_256 -N 1 -t 10 ./SLIC.out
width = 2419, height = 3024
sz = 7315056
Initial time = 0 ms
Conversion time = 10 ms
DeleteEdges and Get_Seeds time = 0 ms
Segmentation time = 19 ms
EnforceLabelConnectivity time = 7 ms
Computing time=38 ms
There are 0 points' labels are different from original file.
```

38ms

Case2: 图片大小 4000x6000=240000000
为初始案例的约 2.4 倍
运行时间从 58868ms 提高到 97ms
达到约 600 倍提升

Case3: 图片大小 2419x3024=7315056
为初始案例的约 0.7 倍
运行时间从 17498ms 提高到 38ms
达到约 450 倍提升

感谢指导

THANKS FOR GUIDANCE