南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计期末开题报告

三角形光栅化反走样算法的并行化

1811516 余樱童

年级：2018级

专业：计算机科学与技术

指导教师：王刚

摘 要

对于本次期末研究，本小组要研究的是三角形光栅化的反走样算法的并行化。本小组选择了两种反走样算法进行研究，分别是低通滤波和超采样。其中，低通滤波可以分为三个步骤：快速傅里叶变换、频域上的乘积和逆快速傅里叶变换。本人负责的是低通滤波方式中的快速傅里叶变换和超采样两个部分。所以，本人将低通滤波的函数作为一个大整体，令逆傅里叶变换保持原样，主要通过对傅里叶变换以及频域上的乘积进行并行化来进行优化。主要采取的优化方式是循环展开、多线程以及多节点的方式。

关键词： 光栅化，反走样算法，低通滤波，快速傅里叶变换，频域上的乘积，串行算法，并行化，循环展开，多线程，多节点

目录

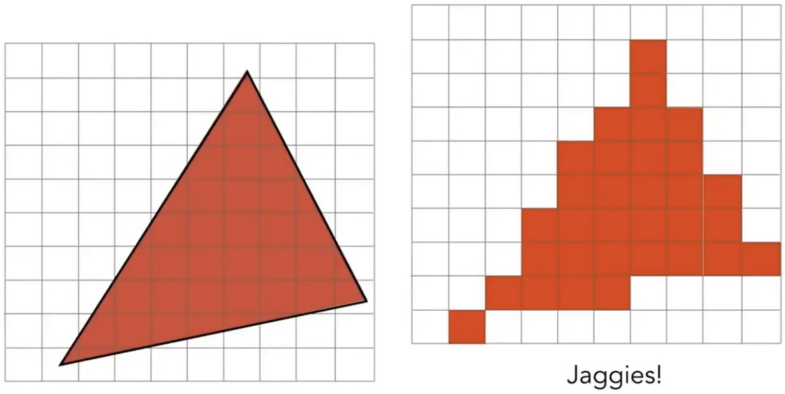
1 问题描述

1.1 走样及反走样

光栅化（Rasterization）是把顶点数据转换为片元的过程，具有将图转化为一个个栅格组成的图象的作用，特点是每个元素对应帧缓冲区中的一像素。 [6]

我们在光栅图形显示器上绘制非水平、非垂直的直线或多边形边界时，会呈现锯齿状外观。这是因为直线和多边形的边界是连续的，而光栅则是由离散的点组成。在光栅显示设备上表现直线、多边形等，必须在离散位置采样。由于采样不充分重建后造成的信息失真，就叫走样。

以下就是一个三角形光栅化前后的图片。（本文中的所有图片均来自 GAMES101 课程课件 [8]）。

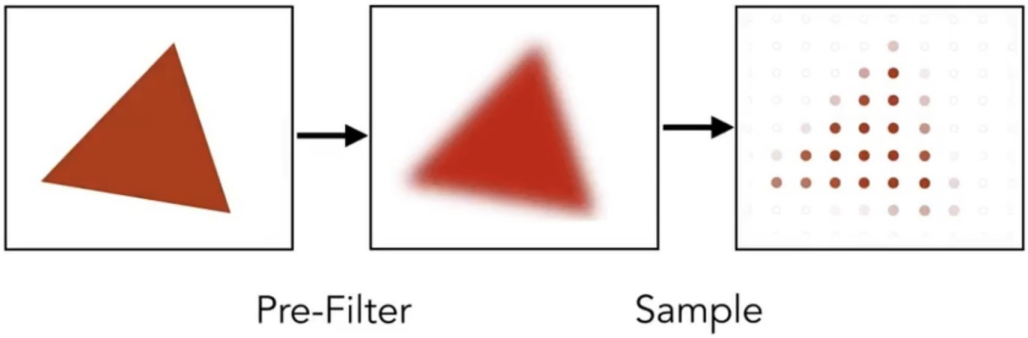


可以看到光栅化之后的图片有着非常严重的锯齿问题，这就是走样之一。

目前，主要有两种减弱走样错误的方法。第一种是提升采样的速率，为了提升采样的速率，我们可以提升频率之间的间隔，或者使用高分辨率的仪器。但是这类方法可能需要非常高昂的花费，为了达到一定的效果，可能需要非常高的分辨率。 [1]

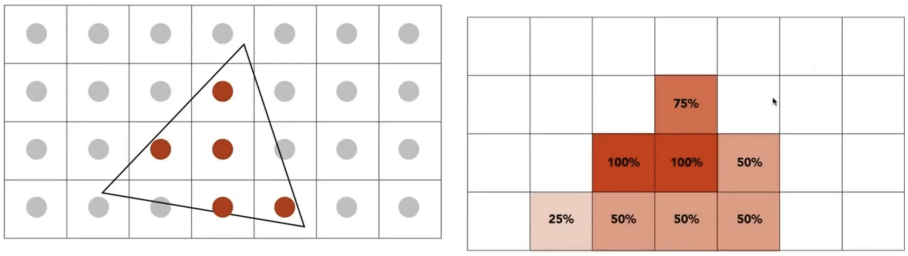
第二种方法就是反走样算法。低通滤波方式的的反走样算法就是先做模糊，之后再进行普通的采样。 [5]

如下例所示。



而 MSAA 的反走样算法就是通过将一个像素划分成很多个小像素来近似1-像素过滤的结果。

如下所示，左图将会转换为右图。因为这一部分不是我主要负责，之后只会具体介绍原理，不会过多讨论 MSAA 的并行化方法。



1.2 我们的研究

对本次期末研究，本小组打算研究如何对反走样算法进行并行化加速。本小组选择的需要进行并行化的算法大致可以分为如下2个部分：低通滤波（快速傅里叶变换、频域上的乘积、逆快速傅里叶变换）以及MSAA。

本小组成员为1811507文静静以及1811516余樱童，由余樱童完成快速傅里叶变换以及频域上的乘积部分，由文静静完成逆快速傅里叶变换以及MSAA的部分。

2 串行算法步骤及原理

2.1 低通滤波

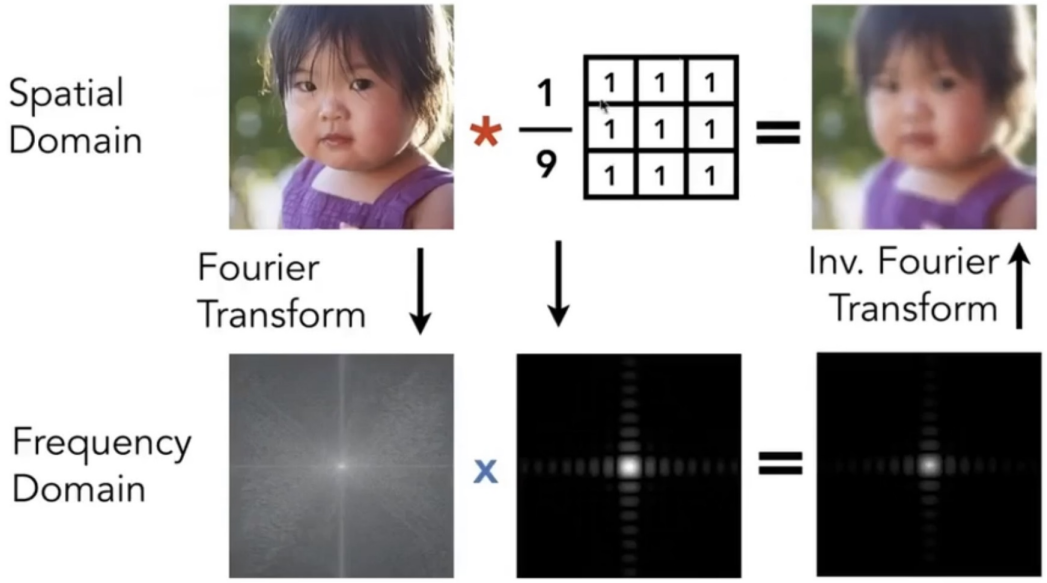
低通滤波(Low-pass filter) 是一种过滤方式，规则为低频信号能正常通过，而超过设定临界值的高频信号则被阻隔、减弱。从直观上，对图像作低通滤波之后，图像就会变为一个相比原图更加模糊的图像。如下所示。

这就实现了我们研究的反走样算法的第一步：模糊处理。而要具体实现这一步，有两个方法，一个是实现时域上的卷积，另一个是实现频域上的乘积，本小组选用的是第二种方法。于是就需要以下三个步骤：快速傅里叶变换、频域上的乘积和逆傅里叶变换。

我们提到，时域上的卷积就是频域上的乘积，而我们要做时域上的卷积，还需要一个卷积盒（或者叫滤波器、Box Filter）。卷积就相当于以某个像素为中心，所有该像素周围在卷积盒中的像素以及该像素本身的值做一个平均，作为该像素的新值。也就是说，做

完卷积之后的图像的每个像素，其实是周围像素的一个混合，可不就变模糊了吗！时域上的图片和这个卷积盒做卷积，就能得到模糊之后的图片。为了做频域上的乘积，我们在第一步也需要对这个卷积盒做傅里叶变换。

整个过程如下所示。

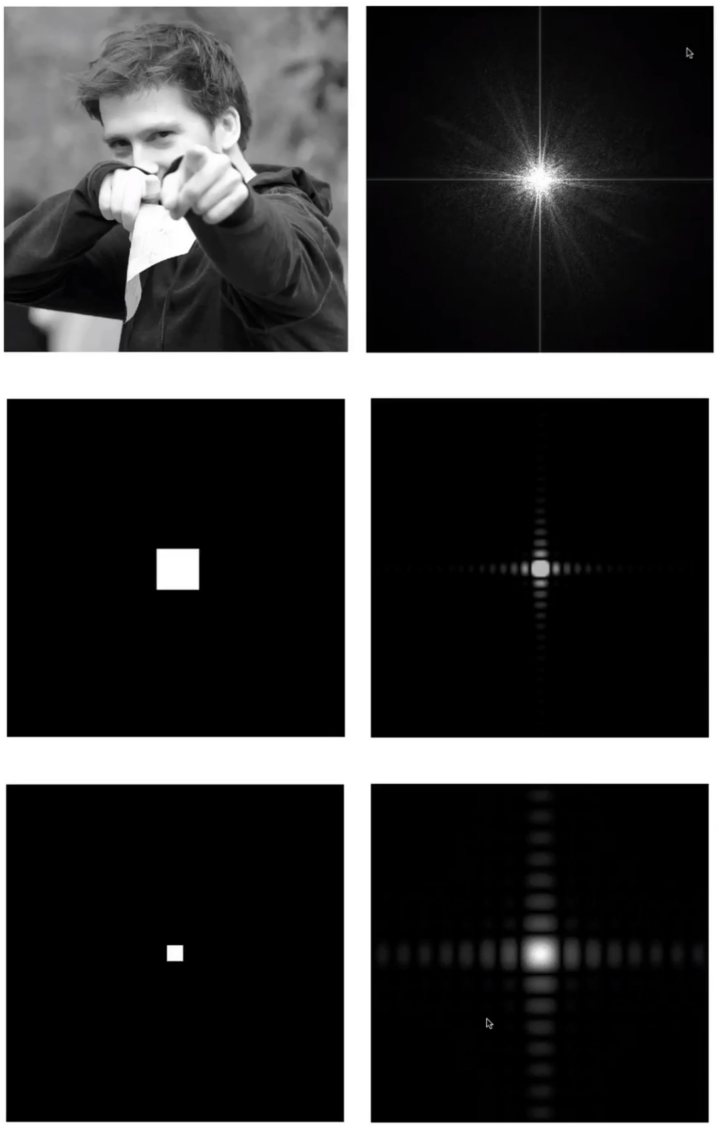


下面就根据这个流程图，来介绍每一步是做什么的。

2.1.1 傅里叶变换

这个步骤是将图像从时域转换到频域。

实现的效果如下所示。



左图就是时域上的图像，也是原图。右边是进行傅里叶变换之后的图像，是频域上的图像。越靠近原点频率越低，越远离原点频率越高，右图靠近原点的部分很亮，也可以看到照片中，是低频信息较多，而滤波器本来就表示低频信息，就只有中心的亮块。注：频域上的图片存在水平和竖直两条特别明亮的线，这两条线是转换过程导致的明亮，并不会体现原图的信息，所以直接忽略这两条线。

2.1.2 频域上的乘积

将两个傅里叶变换的结果做一个乘积，就能够得到模糊后的图在频域上的图像，这一步就是做的除去高频的信号。

这个乘积就类似于普通的点乘。

可以看到，在上面的图中，滤波器的频域图中只有中心的位置有亮块，高频区域是完全黑暗的，做乘积就意味着去除了高频的信号。

当滤波器缩小时，频域上的亮块反而变大，这是可以理解的。假设滤波器就是单个的像素，意味着原图中的所有像素就是本像素内的值的平均，对分辨率还行的设备，也就说明原图只发生了较小的模糊，那么该滤波器在频域上的图像应该就基本是全白的。所以，当滤波器缩小时，频域上的高频信息也会越多。

2.1.3 逆傅里叶变换

逆傅里叶变换基本上就是傅里叶变换的逆，将图像从频域转回时域。做完乘积之后的频域上的图再用逆傅里叶变换返回时域，就能得到一张模糊的图像，也就是我们期望得到的图像。

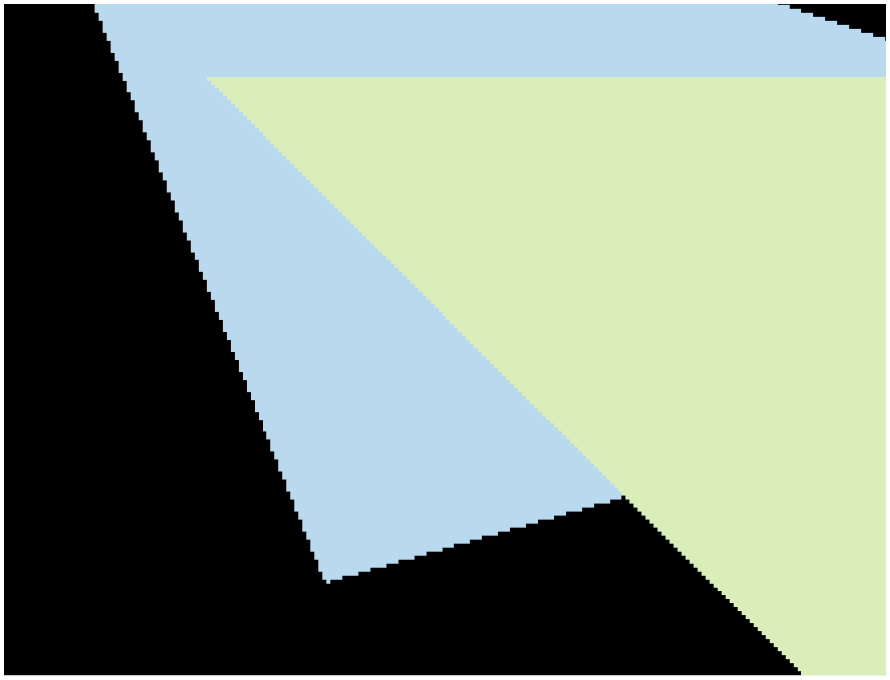
2.1.4 串行运行结果展示

我在 GAMES101 作业2的框架下，实现了三角形的光栅化以及低通滤波方式的反走样算法。完整的代码在提交的代码框架.zip中。因为补充的代码比较长，就不在报告中展示了，补充的代码主要在 rasterizer.cpp 文件的 rst::rasterizer::draw() 函数中。如果要运行该框架，可以在文件目录下使用以下的命令。（环境中需要有eigen这个库的存在）

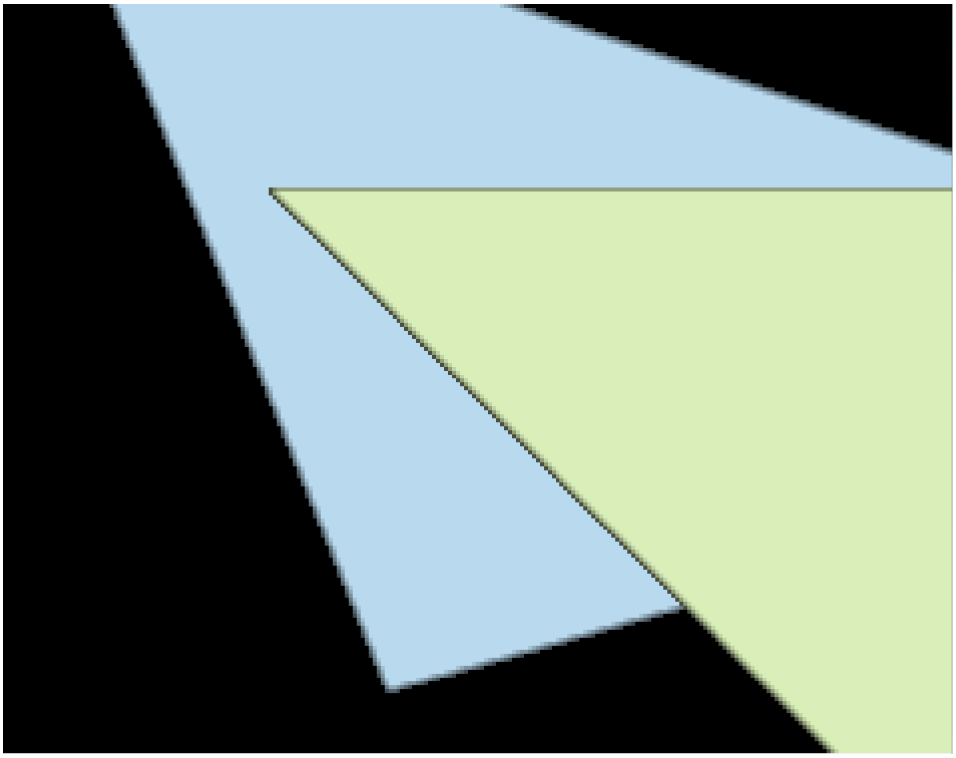
```
mkdir build
```

```
cd build
cmake ..
make -j4
./Rasterizer
```

未使用反走样算法的运行结果如下所示。

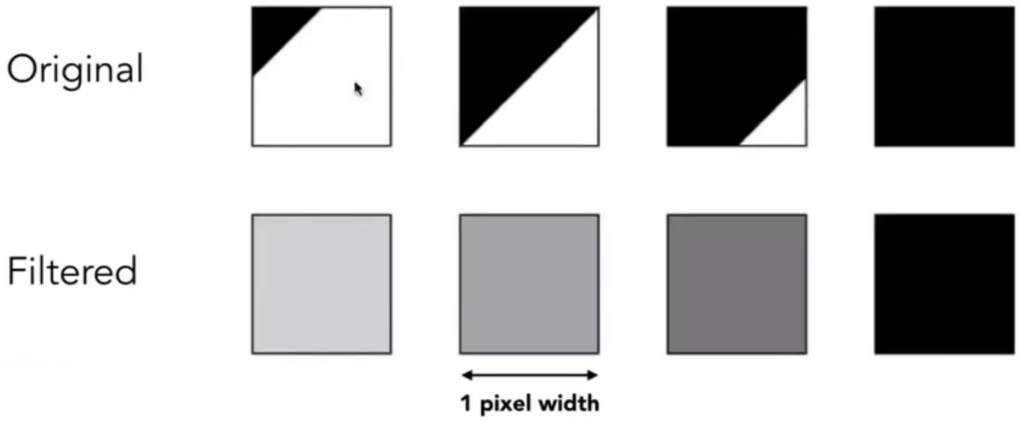


使用了低通滤波的运行结果如下所示。可以看到确实实现了模糊后的光栅化，边缘是有一个渐进消失的效果。尤其对于蓝色的三角形，锯齿问题减轻了很多。



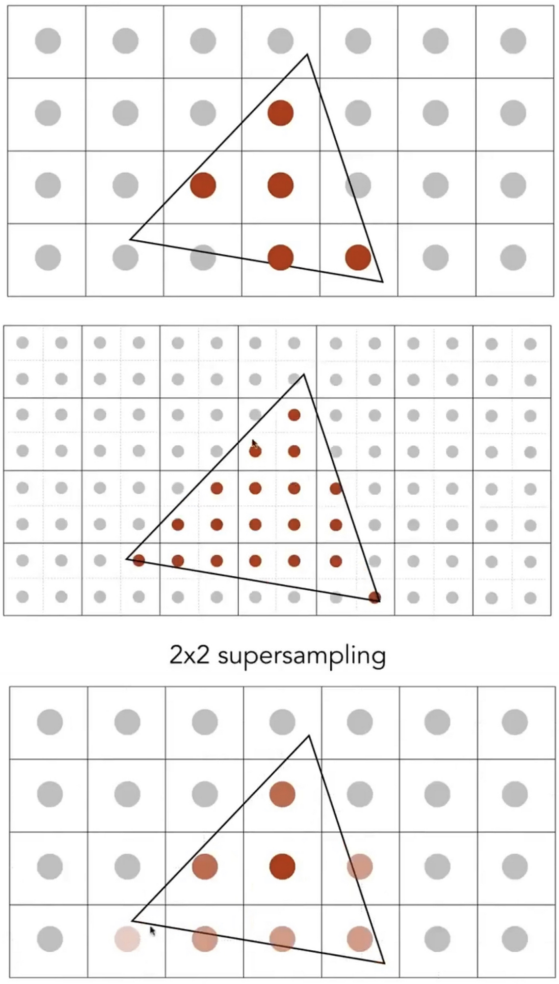
2.2 MSAA

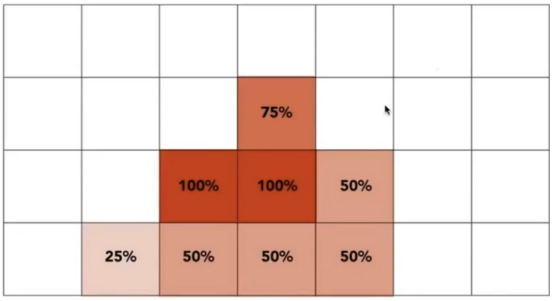
上文已经提到，MSAA其实就是通过将一个像素划分成很多个小像素来近似1-像素过滤的结果。根据前面介绍的卷积的原理，1-像素过滤就如下所示。



但是显然，在每一个像素中计算图像占据的面积，从而计算出该像素的平均值是很繁杂且相当花费时间的（上图给出的例子还是规则图像，对不规则图像，是很难计算的），所以，就出现了MSAA。

我们假设将单个像素划分为了4个小像素，对每个小像素，我们将小像素的值就定义为像素中心的值。得到所有小像素的值之后，我们再用小像素的值来定义包含它的大像素的值，大像素的值就等于其小像素值的平均。大致过程如下所示。





最后就得到了一个逼近1-像素过滤的结果。

3 相关文献

直接查找研究低通滤波和MSAA的并行优化，并没有找到相关的论文，于是，我们分问题进行调研，查找每个子问题优化的方法。

3.1 傅里叶变换

在向量化方面，论文《使用融合乘加加速快速傅里叶变换计算的向量化方法》提出可以在向量化的基础上，将FFT蝶形单元中的乘法和加法操作换为融合加乘的操作，以减少FFT计算的浮点操作指令次数。 [2]这个方法在向量化的基础上还使乘法和加法融合进行，进一步减少了指令数。

在使用多线程方面，大概有3篇相关的比较有代表性的论文。

《基于集群计算机的图像并行处理》介绍了基于集群计算机的图像处理技术。对于FFT算法，论文中提出，由主节点进行总体的控制，对任务进行分块以及分配到各个从节点上，从节点根据主节点传来的参数进行具体的计算，最后再将结果传回给主节点。 [3]

《MPI并行计算在图像处理方面的应用》介绍了使用MPI对一些图像处理算法的处理。在该论文中，作者首先说明了图像处理算法并行化的大致步骤。 [4]

（1）对图像处理问题进行抽象，建立算法串行模型；

（2）对算法串行模型进行分析，找出算法模型中需要并行处理的部分，确定算法并行实现方法，建立算法并行模型的描述；

（3）用并行计算语言实现并行算法；

（4）在并行集群计算系统上运行，调试并行算法。

而对于FFT，论文提出，二维傅里叶变换就是对图像进行两次

一维FFT:首先对图像进行行变换, 然后再对行变换的结果进行列变换。

并行实现快速傅里叶变换算法的思路就是把行变换和列变换的计算任务分配到各个节点并行完成。并行傅里叶变换的过程为:各节点进行行变换→各节点把行变换的结果发送给根进程→根进程对行变换结果进行转置运算→根进程把转置结果发送到所有计算节点→各计算节点再进行行变换→各节点把变换结果发送到根进程→根进程进行转置。

《多核微机基于OpenMP的并行计算》介绍了基于 OpenMP 对 FFT 进行并行优化的方法，而上面几篇论文提出的方法大同小异。基本思路也是将一个二维傅里叶变换的运算分解成水平和垂直方向上的一维傅里叶变换运算。 [7]

因为在我们要研究的问题中，傅里叶变换是一个被频繁调用的函数，因此难以实现用集群计算机进行并行处理。所以，对于傅里叶变换，还是考虑使用多线程的思想对其循环进行优化。

3.2 乘积

这个部分算是低通滤波算法的核心，该函数只会被调用一次，而在此函数中会有两层循环并且频繁调用傅里叶变换以及逆傅里叶变换函数，因此，该函数将会是我们重点优化的对象。对此函数，除了使用多线程外，也可以使用集群计算机进行多节点优化。

4 并行方式分析

4.1 代码分析

为了方便运行以及测试，另外写了一个只包含光栅化步骤的 cpp 文件（也就是文件夹下的 ord.cpp 文件），将光栅化的部分单独抽离出来进行并行化。文件中包含3个主要函数，分别是傅里叶变换函数 ft() 、进行频域上的乘积的函数 convolute() 、还有逆傅里叶变换变换的函数 ift() ，我只需要进行 ft() 以及 convolute() 的并行优化。

两个函数的串行代码如下所示。

ft()函数。其中传入的长高参数width以及height一般来说就是卷积盒的大小。

```
void ft(int width, int height, Vector3f** fxRealTwo,
   Vector3f** fxImagTwo, Vector3f** RealTwo)
{
  // 结果数据初始化
  for (int v = 0; v < width; v++)
  {
    for (int u = 0; u < height; u++)
    {
      fxRealTwo[v][u] = {0, 0, 0};
      fxImagTwo[v][u] = {0, 0, 0};
    }
  }

  // 使用4层循环进行运算
  for (int v = 0; v < width; v++)
  {
    for (int u = 0; u < height; u++)
```

```
17      {
18        for (int j = 0; j < width; j++)
19        {
20          for (int i = 0; i < height; i++)
21          {
22            fxRealTwo[v][u] += RealTwo[j][i] * cos(2 * MY_PI
                 * u * i / height + 2 * MY_PI * v * j / width);
23            fxImagTwo[v][u] -= RealTwo[j][i] * sin(2 * MY_PI
                 * u * i / height + 2 * MY_PI * v * j / width);
24          }
25        }
26      }
27    }
28  }
```

convolute()函数。该函数核心操作就是，遍历原图中的每个位置(x, y)，将以(x, y)为中心的一片区域（这片区域的大小由卷积盒的大小决定）进行傅里叶变换，然后该区域的每个位置与对应的卷积盒中的位置的数据相乘，最后将(x, y)位置的乘积结果进行逆傅里叶变换，就得到了该位置卷积的结果。遍历所有位置之后，就得到了整个图像卷积之后的结果，也就是低通滤波之后的结果。

```
1  void ord_convolution(Vector3f **ResultReal, Vector3f
     **original_color, Vector3f **FilterReal, Vector3f
     **FilterImage, int len_x, int len_y, int argc, char
     *argv[])
2  {
3    // 初始化临时变量
4    Vector3f** tempColor = new Vector3f*[filter_box_size];
5    for(int i = 0; i < filter_box_size; i++) tempColor[i] =
       new Vector3f[filter_box_size];
```

```
6
7   Vector3f** tempReal = new Vector3f*[filter_box_size];
8   for(int i = 0; i < filter_box_size; i++) tempReal[i] =
       new Vector3f[filter_box_size];
9
10  Vector3f** tempImage = new Vector3f*[filter_box_size];
11  for(int i = 0; i < filter_box_size; i++) tempImage[i] =
       new Vector3f[filter_box_size];
12
13  Vector3f** tempResultReal = new
       Vector3f*[filter_box_size];
14  for(int i = 0; i < filter_box_size; i++)
       tempResultReal[i] = new Vector3f[filter_box_size];
15
16  Vector3f** tempResultImage = new
       Vector3f*[filter_box_size];
17  for(int i = 0; i < filter_box_size; i++)
       tempResultImage[i] = new Vector3f[filter_box_size];
18
19  // 循环对每个区域都计算卷积
20  for(int i = filter_box_size / 2; i < len_x -
       filter_box_size / 2; i++)
21  {
22    for(int j = filter_box_size / 2; j < len_y -
         filter_box_size / 2; j++)
23    {
24      // 获得当前区域的原始颜色
25      for(int x = 0; x < filter_box_size; x++)
26        for(int y = 0; y < filter_box_size; y++)
27          tempColor[x][y] = original_color[i -
              filter_box_size / 2 + x][j - filter_box_size
              / 2 + y];
```

```
28          // 对原始区域进行卷积
29          // 进行傅里叶变换
30          ft(filter_box_size, filter_box_size, tempReal,
                tempImage, tempColor);
31          // 循环进行乘积的计算
32          int p = filter_box_size / 2;
33          for(int x = 0; x < filter_box_size; x++)
34          {
35            if(i + x - filter_box_size/2 < 0 || i + x -
                 filter_box_size/2 >= len_x) continue;
36            for(int y = 0; y < filter_box_size; y++)
37            {
38              if(j + y - filter_box_size/2 < 0 || j + y -
                   filter_box_size/2 >= len_y) continue;
39              tempResultReal[x][y] = tempReal[x][y] *
                   FilterReal[x][y] - tempImage[x][y] *
                   FilterImage[x][y];
40              tempResultImage[x][y] = tempReal[x][y] *
                   FilterImage[x][y] + FilterReal[x][y] *
                   tempImage[x][y];
41            }
42          }
43          // 逆傅里叶变换
44          ift(filter_box_size, filter_box_size,
                tempResultReal, tempResultImage, tempReal,
                tempImage);
45          // 保存运算结果
46          ResultReal[i][j] = tempReal[p][p];
47        }
48      }
```

另外，原来的代码中引入了 eigen 库来对向量进行运算，但是远程节点上并没有配置这个库，所以自己重现实现了一个数据结构 Vector3f 并进行运算符重载，来实现向量的运算。

下面，针对这个问题，我们分别从体系结构、SIMD、多线程以及多节点的角度进行分析。

4.2 体系结构

在这一部分，考虑对两个函数的内层循环进行循环展开，展开3 4层循环。

展开的结果如下所示。

ft()函数。因为卷积盒的长宽一般是奇数，因此对于 ft() 函数，只展开最内层的3层循环。

```
void ft(int width, int height, Vector3f** fxRealTwo,
    Vector3f** fxImagTwo, Vector3f** RealTwo)
{
  // 结果数据初始化
  for (int v = 0; v < width; v++)
  {
    for (int u = 0; u < height; u++)
    {
      fxRealTwo[v][u] = {0, 0, 0};
      fxImagTwo[v][u] = {0, 0, 0};
    }
  }

  // 具体运算
  for (int v = 0; v < width; v++)
  {
    for (int u = 0; u < height; u++)
```

```
17      {
18        for (int j = 0; j < width; j++)
19        {
20          int i;
21          // 循环展开
22          for(i = 0; i + 2 < height; i += 3)
23          {
24            int p = 2 * MY_PI * u * i / height + 2 * MY_PI *
                  v * j / width;
25            fxRealTwo[v][u] += RealTwo[j][i] * cos(p);
26            fxImagTwo[v][u] -= RealTwo[j][i] * sin(p);
27            p = 2 * MY_PI * u * (i + 1) / height + 2 * MY_PI
                  * v * j / width;
28            fxRealTwo[v][u] += RealTwo[j][i + 1] * cos(p);
29            fxImagTwo[v][u] -= RealTwo[j][i + 1] * sin(p);
30            p = 2 * MY_PI * u * (i + 2) / height + 2 * MY_PI
                  * v * j / width;
31            fxRealTwo[v][u] += RealTwo[j][i + 2] * cos(p);
32            fxImagTwo[v][u] -= RealTwo[j][i + 2] * sin(p);
33          }
34          for(; i < height; i++)
35          {
36            fxRealTwo[v][u] += RealTwo[j][i] * cos(2 * MY_PI
                  * u * i / height + 2 * MY_PI * v * j / width);
37            fxImagTwo[v][u] -= RealTwo[j][i] * sin(2 * MY_PI
                  * u * i / height + 2 * MY_PI * v * j / width);
38          }
39        }
40      }
41    }
42  }
```

convolute()函数。对于卷积函数，我们展开第二层循环。

```
void ord_convolution(Vector3f ResultReal[n][n], Vector3f
    **original_color, Vector3f **FilterReal, Vector3f
    **FilterImage, int len_x, int len_y, int argc, char
    *argv[])
{
  // 初始化临时变量
  Vector3f** tempColor = new Vector3f*[filter_box_size];
  for(int i = 0; i < filter_box_size; i++) tempColor[i] =
     new Vector3f[filter_box_size];

  Vector3f** tempReal = new Vector3f*[filter_box_size];
  for(int i = 0; i < filter_box_size; i++) tempReal[i] =
     new Vector3f[filter_box_size];

  Vector3f** tempImage = new Vector3f*[filter_box_size];
  for(int i = 0; i < filter_box_size; i++) tempImage[i] =
     new Vector3f[filter_box_size];

  Vector3f** tempResultReal = new
     Vector3f*[filter_box_size];
  for(int i = 0; i < filter_box_size; i++)
     tempResultReal[i] = new Vector3f[filter_box_size];

  Vector3f** tempResultImage = new
     Vector3f*[filter_box_size];
  for(int i = 0; i < filter_box_size; i++)
     tempResultImage[i] = new Vector3f[filter_box_size];

  // 循环对每个区域都计算卷积
  for(int i = filter_box_size / 2; i < len_x -
     filter_box_size / 2; i++)
```

```
21  {
22    int j = filter_box_size / 2;
23    // 循环展开
24    for(; j + 3 < len_y - filter_box_size / 2; j += 4)
25    {
26      // j = j
27      // 获得当前区域的原始颜色
28      for(int x = 0; x < filter_box_size; x++)
29        for(int y = 0; y < filter_box_size; y++)
30          tempColor[x][y] = original_color[i -
              filter_box_size / 2 + x][j - filter_box_size
              / 2 + y];
31      // 对原始区域进行卷积
32      // 进行傅里叶变换
33      ft(filter_box_size, filter_box_size, tempReal,
          tempImage, tempColor);
34      // 循环进行乘积的计算
35      int p = filter_box_size / 2;
36      for(int x = 0; x < filter_box_size; x++)
37      {
38        if(i + x - filter_box_size/2 < 0 || i + x -
            filter_box_size/2 >= len_x) continue;
39        for(int y = 0; y < filter_box_size; y++)
40        {
41          if(j + y - filter_box_size/2 < 0 || j + y -
              filter_box_size/2 >= len_y) continue;
42          tempResultReal[x][y] = tempReal[x][y] *
              FilterReal[x][y] - tempImage[x][y] *
              FilterImage[x][y];
43          tempResultImage[x][y] = tempReal[x][y] *
              FilterImage[x][y] + FilterReal[x][y] *
              tempImage[x][y];
```

```
44          }
45        }
46        // 逆傅里叶变换
47        ift(filter_box_size, filter_box_size,
              tempResultReal, tempResultImage, tempReal,
              tempImage);
48        // 保存运算结果
49        ResultReal[i][j] = tempReal[p][p];
50
51        // j = j + 1
52        j++;
53        // 获得当前区域的原始颜色
54        for(int x = 0; x < filter_box_size; x++)
55          for(int y = 0; y < filter_box_size; y++)
56            tempColor[x][y] = original_color[i -
                  filter_box_size / 2 + x][j - filter_box_size
                  / 2 + y];
57        // 对原始区域进行卷积
58        // 进行傅里叶变换
59        ft(filter_box_size, filter_box_size, tempReal,
              tempImage, tempColor);
60        // 循环进行乘积的计算
61        for(int x = 0; x < filter_box_size; x++)
62        {
63          if(i + x - filter_box_size/2 < 0 || i + x -
                filter_box_size/2 >= len_x) continue;
64          for(int y = 0; y < filter_box_size; y++)
65          {
66            if(j + y - filter_box_size/2 < 0 || j + y -
                  filter_box_size/2 >= len_y) continue;
67            tempResultReal[x][y] = tempReal[x][y] *
                  FilterReal[x][y] - tempImage[x][y] *
```

```
            FilterImage[x][y];
68          tempResultImage[x][y] = tempReal[x][y] *
            FilterImage[x][y] + FilterReal[x][y] *
            tempImage[x][y];
69        }
70      }
71      // 逆傅里叶变换
72      ift(filter_box_size, filter_box_size,
          tempResultReal, tempResultImage, tempReal,
          tempImage);
73      // 保存运算结果
74      ResultReal[i][j] = tempReal[p][p];
75
76      // j = j + 2
77      j++;
78      // 获得当前区域的原始颜色
79      for(int x = 0; x < filter_box_size; x++)
80        for(int y = 0; y < filter_box_size; y++)
81          tempColor[x][y] = original_color[i -
              filter_box_size / 2 + x][j - filter_box_size
              / 2 + y];
82      // 对原始区域进行卷积
83      // 进行傅里叶变换
84      ft(filter_box_size, filter_box_size, tempReal,
          tempImage, tempColor);
85      // 循环进行乘积的计算
86      for(int x = 0; x < filter_box_size; x++)
87      {
88        if(i + x - filter_box_size/2 < 0 || i + x -
            filter_box_size/2 >= len_x) continue;
89        for(int y = 0; y < filter_box_size; y++)
90        {
```

```
91            if(j + y - filter_box_size/2 < 0 || j + y -
                  filter_box_size/2 >= len_y) continue;
92            tempResultReal[x][y] = tempReal[x][y] *
                  FilterReal[x][y] - tempImage[x][y] *
                  FilterImage[x][y];
93            tempResultImage[x][y] = tempReal[x][y] *
                  FilterImage[x][y] + FilterReal[x][y] *
                  tempImage[x][y];
94          }
95        }
96        // 逆傅里叶变换
97        ift(filter_box_size, filter_box_size,
              tempResultReal, tempResultImage, tempReal,
              tempImage);
98        // 保存运算结果
99        ResultReal[i][j] = tempReal[p][p];
100
101       // j = j + 3
102       j++;
103       // 获得当前区域的原始颜色
104       for(int x = 0; x < filter_box_size; x++)
105         for(int y = 0; y < filter_box_size; y++)
106           tempColor[x][y] = original_color[i -
                  filter_box_size / 2 + x][j - filter_box_size
                  / 2 + y];
107       // 对原始区域进行卷积
108       // 进行傅里叶变换
109       ft(filter_box_size, filter_box_size, tempReal,
              tempImage, tempColor);
110       // 循环进行乘积的计算
111       for(int x = 0; x < filter_box_size; x++)
112       {
```

```
113             if(i + x - filter_box_size/2 < 0 || i + x -
                    filter_box_size/2 >= len_x) continue;
114             for(int y = 0; y < filter_box_size; y++)
115             {
116               if(j + y - filter_box_size/2 < 0 || j + y -
                      filter_box_size/2 >= len_y) continue;
117               tempResultReal[x][y] = tempReal[x][y] *
                      FilterReal[x][y] - tempImage[x][y] *
                      FilterImage[x][y];
118               tempResultImage[x][y] = tempReal[x][y] *
                      FilterImage[x][y] + FilterReal[x][y] *
                      tempImage[x][y];
119             }
120           }
121           // 逆傅里叶变换
122           ift(filter_box_size, filter_box_size,
                  tempResultReal, tempResultImage, tempReal,
                  tempImage);
123           // 保存运算结果
124           ResultReal[i][j] = tempReal[p][p];
125
126           j -= 3;
127         }
128         for(; j < len_y - filter_box_size / 2; j++)
129         {
130           // 获得当前区域的原始颜色
131           for(int x = 0; x < filter_box_size; x++)
132             for(int y = 0; y < filter_box_size; y++)
133               tempColor[x][y] = original_color[i -
                      filter_box_size / 2 + x][j - filter_box_size
                      / 2 + y];
134           // 对原始区域进行卷积
```

```
135         // 进行傅里叶变换
136         ft(filter_box_size, filter_box_size, tempReal,
                tempImage, tempColor);
137         // 循环进行乘积的计算
138         int p = filter_box_size / 2;
139         for(int x = 0; x < filter_box_size; x++)
140         {
141           if(i + x - filter_box_size/2 < 0 || i + x -
                filter_box_size/2 >= len_x) continue;
142           for(int y = 0; y < filter_box_size; y++)
143           {
144             if(j + y - filter_box_size/2 < 0 || j + y -
                  filter_box_size/2 >= len_y) continue;
145             tempResultReal[x][y] = tempReal[x][y] *
                  FilterReal[x][y] - tempImage[x][y] *
                  FilterImage[x][y];
146             tempResultImage[x][y] = tempReal[x][y] *
                  FilterImage[x][y] + FilterReal[x][y] *
                  tempImage[x][y];
147           }
148         }
149         // 逆傅里叶变换
150         ift(filter_box_size, filter_box_size,
                tempResultReal, tempResultImage, tempReal,
                tempImage);
151         // 保存运算结果
152         ResultReal[i][j] = tempReal[p][p];
153       }
154     }
155 }
```

4.3 SIMD

光栅化部分，我们的数据结构是一个3维的向量，代表着像素的RGB。因为原来计算的数据本身已经是个向量了，没有很好的办法对这个数据结构再次进行向量化，所以，虽然对每个数据执行的操作是一样的，在当前的大框架下，这部分代码并不能使用SSE/AVX 进行计算的向量化。

4.4 OpenMP

在多线程部分，我选择了使用 OpenMP 来对快速傅里叶变换以及频域上的乘积进行多线程计算。

快速傅里叶变换中有多层循环，可以将循环分配给不同的线程进行计算。

类似的，频域上的乘积也有多层循环，可以利用 OpenMP 将循环分配给不同的线程进行计算。

实现的核心代码如下所示。

在ft()中。

```
void ft(int width, int height, Vector3f** fxRealTwo,
   Vector3f** fxImagTwo, Vector3f** RealTwo)
{
  for (int v = 0; v < width; v++)
  {
    for (int u = 0; u < height; u++)
    {
      fxRealTwo[v][u] = {0, 0, 0};
      fxImagTwo[v][u] = {0, 0, 0};
    }
  }

```

```
12    // OpenMP
13    # pragma omp parallel for num_threads(THREAD_NUM)
14    for (int v = 0; v < width; v++)
15    {
16      for (int u = 0; u < height; u++)
17      {
18        for (int j = 0; j < width; j++)
19        {
20          for (int i = 0; i < height; i++)
21          {
22            fxRealTwo[v][u] += RealTwo[j][i] * cos(2 * MY_PI
                  * u * i / height + 2 * MY_PI * v * j / width);
23            fxImagTwo[v][u] -= RealTwo[j][i] * sin(2 * MY_PI
                  * u * i / height + 2 * MY_PI * v * j / width);
24          }
25        }
26      }
27    }
28  }
```

在convolute()中。

```
1   void ord_convolution(Vector3f **ResultReal, Vector3f
      **original_color, Vector3f **FilterReal, Vector3f
      **FilterImage, int len_x, int len_y, int argc, char
      *argv[])
2   {
3     // 初始化临时变量
4     Vector3f** tempColor = new Vector3f*[filter_box_size];
5     for(int i = 0; i < filter_box_size; i++) tempColor[i] =
        new Vector3f[filter_box_size];
6
```

```
7   Vector3f** tempReal = new Vector3f*[filter_box_size];
8   for(int i = 0; i < filter_box_size; i++) tempReal[i] =
        new Vector3f[filter_box_size];
9
10  Vector3f** tempImage = new Vector3f*[filter_box_size];
11  for(int i = 0; i < filter_box_size; i++) tempImage[i] =
        new Vector3f[filter_box_size];
12
13  Vector3f** tempResultReal = new
        Vector3f*[filter_box_size];
14  for(int i = 0; i < filter_box_size; i++)
        tempResultReal[i] = new Vector3f[filter_box_size];
15
16  Vector3f** tempResultImage = new
        Vector3f*[filter_box_size];
17  for(int i = 0; i < filter_box_size; i++)
        tempResultImage[i] = new Vector3f[filter_box_size];
18
19  // 循环对每个区域都计算卷积
20  // OpenMP
21  # pragma omp parallel for num_threads(THREAD_NUM)
22  for(int i = filter_box_size / 2; i < len_x -
        filter_box_size / 2; i++)
23  {
24    for(int j = filter_box_size / 2; j < len_y -
          filter_box_size / 2; j++)
25    {
26      // 获得当前区域的原始颜色
27      for(int x = 0; x < filter_box_size; x++)
28        for(int y = 0; y < filter_box_size; y++)
29          tempColor[x][y] = original_color[i -
              filter_box_size / 2 + x][j - filter_box_size
```

```
                / 2 + y];
30          // 对原始区域进行卷积
31          // 进行傅里叶变换
32          ft(filter_box_size, filter_box_size, tempReal,
               tempImage, tempColor);
33          // 循环进行乘积的计算
34          int p = filter_box_size / 2;
35          for(int x = 0; x < filter_box_size; x++)
36          {
37            if(i + x - filter_box_size/2 < 0 || i + x -
                 filter_box_size/2 >= len_x) continue;
38            for(int y = 0; y < filter_box_size; y++)
39            {
40              if(j + y - filter_box_size/2 < 0 || j + y -
                   filter_box_size/2 >= len_y) continue;
41              tempResultReal[x][y] = tempReal[x][y] *
                   FilterReal[x][y] - tempImage[x][y] *
                   FilterImage[x][y];
42              tempResultImage[x][y] = tempReal[x][y] *
                   FilterImage[x][y] + FilterReal[x][y] *
                   tempImage[x][y];
43            }
44          }
45          // 逆傅里叶变换
46          ift(filter_box_size, filter_box_size,
               tempResultReal, tempResultImage, tempReal,
               tempImage);
47          // 保存运算结果
48          ResultReal[i][j] = tempReal[p][p];
49        }
50      }
51
```

```
52    // 收回分配出去的空间
53    for(int i = 0; i < filter_box_size; i++) delete[]
          tempColor[i];
54    delete[] tempColor;
55
56    for(int i = 0; i < filter_box_size; i++) delete[]
          tempReal[i];
57    delete[] tempReal;
58
59    for(int i = 0; i < filter_box_size; i++) delete[]
          tempImage[i];
60    delete[] tempImage;
61
62    for(int i = 0; i < filter_box_size; i++) delete[]
          tempResultReal[i];
63    delete[] tempResultReal;
64
65    for(int i = 0; i < filter_box_size; i++) delete[]
          tempResultImage[i];
66    delete[] tempResultImage;
67  }
```

4.5 MPI

因为 ft() 函数主要是在 convolute() 函数中被调用，光栅化的主要部分是 convolute()，所以主要考虑利用 MPI 对 convolute() 进行并行优化。

使用主从结构，由一个主节点进行 convolute() 任务的分配，将循环中的计算任务分配给其他的计算节点，计算节点完成计算后

就将计算结果传递给主节点之后退出。主节点接收到所有的计算结果后退出函数，继续执行之后的内容。

实现后的covolute()代码如下所示。计算过程都是一样的，只是增加了任务分配和结果回收的过程。

```
// 卷积函数
void convolution(Vector3f ResultReal[n][n], Vector3f
    **original_color, Vector3f **FilterReal, Vector3f
    **FilterImage, int len_x, int len_y, int argc, char
    *argv[])
{
  int my_id, size;
  MPI_Status status;
  // 初始化mpi
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // 计算节点的个数
  int compute_nodes = size - 1;

  // 主节点获得运算数据
  if(my_id == 0)
  {
    cout << "n: " << n << endl;
    // 交叉获得每行的元素
    for(int i = filter_box_size / 2; i < len_x -
        filter_box_size / 2; i++)
    {
      // 主节点直接获得运算结果
      MPI_Recv(ResultReal[i], (len_y - (filter_box_size /
          2) * 2) * 3, MPI_FLOAT, (i - filter_box_size / 2)
```

```
            % (size - 1) + 1, i, MPI_COMM_WORLD, &status);
        }
    }
    // 从节点进行计算
    else
    {
        // 初始化临时变量
        Vector3f** tempColor = new Vector3f*[filter_box_size];
        for(int i = 0; i < filter_box_size; i++) tempColor[i]
            = new Vector3f[filter_box_size];

        Vector3f** tempReal = new Vector3f*[filter_box_size];
        for(int i = 0; i < filter_box_size; i++) tempReal[i] =
            new Vector3f[filter_box_size];

        Vector3f** tempImage = new Vector3f*[filter_box_size];
        for(int i = 0; i < filter_box_size; i++) tempImage[i]
            = new Vector3f[filter_box_size];

        Vector3f** tempResultReal = new
            Vector3f*[filter_box_size];
        for(int i = 0; i < filter_box_size; i++)
            tempResultReal[i] = new Vector3f[filter_box_size];

        Vector3f** tempResultImage = new
            Vector3f*[filter_box_size];
        for(int i = 0; i < filter_box_size; i++)
            tempResultImage[i] = new Vector3f[filter_box_size];

        // 从节点遍历自己的任务进行计算
        for(int i = filter_box_size / 2 + my_id - 1; i < len_x
            - filter_box_size / 2; i += compute_nodes)
```

```
46      {
47        for(int j = filter_box_size / 2; j < len_y -
              filter_box_size / 2; j++)
48        {
49          // 获得当前区域的原始颜色
50          for(int x = 0; x < filter_box_size; x++)
51            for(int y = 0; y < filter_box_size; y++)
52              tempColor[x][y] = original_color[i -
                    filter_box_size / 2 + x][j -
                    filter_box_size / 2 + y];
53          // 对原始区域进行卷积
54          ft(filter_box_size, filter_box_size, tempReal,
                tempImage, tempColor);
55          int p = filter_box_size / 2;
56          for(int x = 0; x < filter_box_size; x++)
57          {
58            if(i + x - filter_box_size/2 < 0 || i + x -
                  filter_box_size/2 >= len_x) continue;
59            for(int y = 0; y < filter_box_size; y++)
60            {
61              if(j + y - filter_box_size/2 < 0 || j + y -
                    filter_box_size/2 >= len_y) continue;
62              tempResultReal[x][y] = tempReal[x][y] *
                    FilterReal[x][y] - tempImage[x][y] *
                    FilterImage[x][y];
63              tempResultImage[x][y] = tempReal[x][y] *
                    FilterImage[x][y] + FilterReal[x][y] *
                    tempImage[x][y];
64            }
65          }
66          ift(filter_box_size, filter_box_size,
                tempResultReal, tempResultImage, tempReal,
```

```
            tempImage);
67          ResultReal[i][j] = tempReal[p][p];
68        }
69      }
70      for(int i = filter_box_size / 2 + my_id - 1; i < len_x
          - filter_box_size / 2; i += compute_nodes)
71      MPI_Send(ResultReal[i], (len_y - (filter_box_size / 2)
          * 2) * 3, MPI_FLOAT, 0, i, MPI_COMM_WORLD);
72
73      // 收回分配出去的空间
74      for(int i = 0; i < filter_box_size; i++) delete[]
          tempColor[i];
75      delete[] tempColor;
76
77      for(int i = 0; i < filter_box_size; i++) delete[]
          tempReal[i];
78      delete[] tempReal;
79
80      for(int i = 0; i < filter_box_size; i++) delete[]
          tempImage[i];
81      delete[] tempImage;
82
83      for(int i = 0; i < filter_box_size; i++) delete[]
          tempResultReal[i];
84      delete[] tempResultReal;
85
86      for(int i = 0; i < filter_box_size; i++) delete[]
          tempResultImage[i];
87      delete[] tempResultImage;
88    }
89
90    MPI_Finalize();
```

```
91
92    // 从节点直接结束程序
93    if(my_id != 0) exit(0);
94 }
```

同时，我们也可以把 OpenMP 多线程算法融入到 MPI 中。

5 实验平台

5.1 研究线程影响时

在研究多线程的影响时，使用的是本机的codeblocks进行实验。对应的硬件设置以及编译设置如下所示。

1.语言和编译器：Have g++ follow the C++17 GNU C++ language standard (ISO C++ plus GNU extensions)

2.目标机器：Target x86_64 (64bit)

3.优化选项：Optimize more (for speed)

4.CPU结构：Intel Core i7 CPU (MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AES, PCLMUL, 64-bit extensions)

5.编译选项添加：-fopenmp

6.链接选项添加：-fopenmp -lgomp

5.2 其他情况

其他情况下使用的都是基于金山云搭建的虚拟机集群。

虚拟机集群包括 1 个 master 节点和 32 个计算节点，每个计算节点 2 核，共 64 核可供计算。

具体架构如下所示。

Archeitecture: x86_64

CPU op-mode(s): 32-bit, 64-bit

CPU(s): 2

On-line CPU(s) list: 0, 1

Thread(s) per core: 1

Core(s) per socket: 2

Socket(s): 1

L1d cache: 32K
L1i cache: 32K
L2 cache: 1024K
L3 cache: 25344K

6 实验过程、结果以及分析

根据以上的分析，我们可以将算法分成以下几类：串行算法、循环展开算法、OpenMP算法、MPI算法、OpenMP + MPI算法。

针对这几种算法，我们不断更改问题规模，也就是要处理的图像的大小，查看各个算法的优化表现。同时，我们也可以在固定其他参数的情况下，分别更改线程数量、节点数量、卷积盒大小，探究其对算法效果的影响。

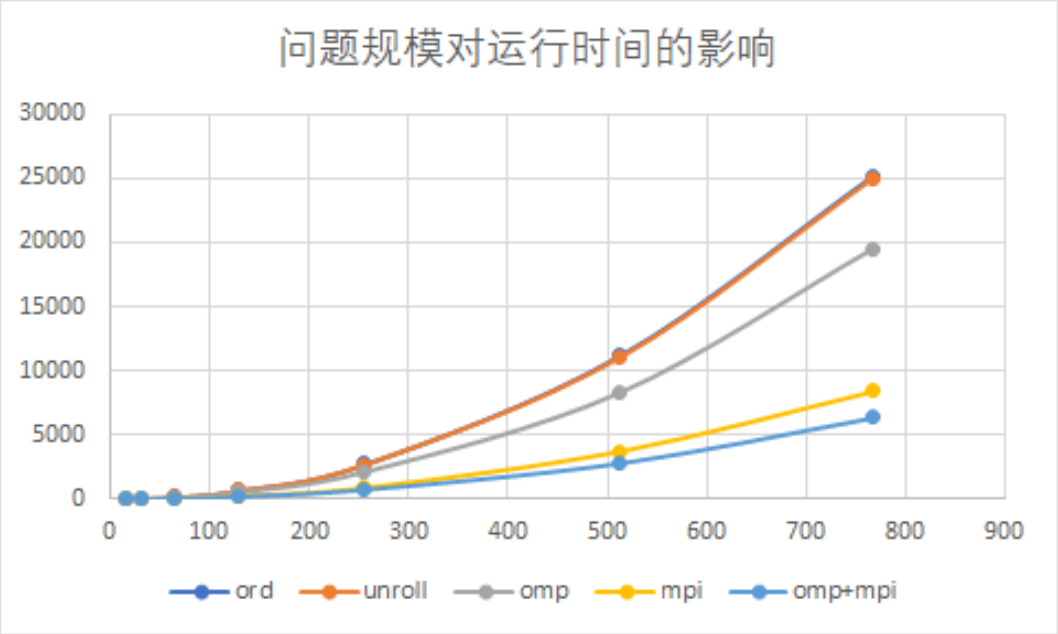
6.1 时间复杂度分析

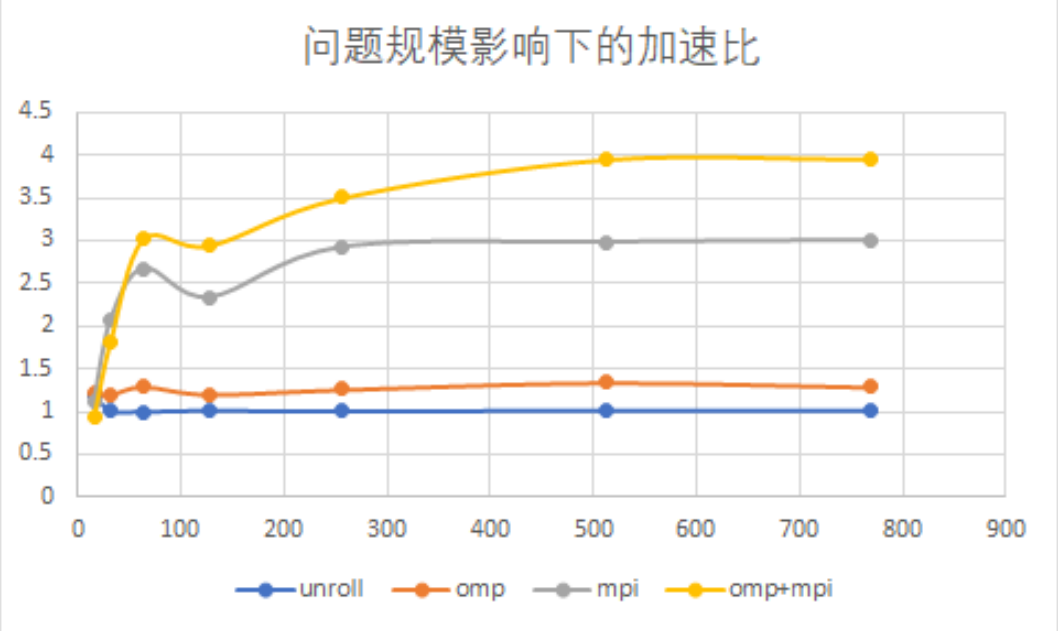
当卷积盒的大小远小于要处理的图片的大小时，算法的时间复杂度为 $O(n^2)$。但是，当卷积盒的大小逐渐增大时，算法的时间复杂度可能会升级为 $O(n^2m^4)$，其中m代表卷积盒的长（或宽），但是显然，m还是小于n的。

6.2 更改问题规模

在固定线程数量为2、节点数量为4、卷积盒大小为5×5的情况下，我们不断更改问题的规模（要处理的图像的长/宽），记录运行的时间以及各自的加速比，得到的结果如下所示。

n	16	32	64	128	256	512	768
ord	6.267	33.9244	154.699	663.954	2719.24	11137.2	25190.7
unroll	5.244	33.6583	155.168	658.021	2717.45	11023.7	24990.3
omp	5.143	28.2714	119.465	552.253	2159.51	8313.65	19553.5
mpi	5.589	16.425	57.864	284.332	931.016	3736.64	8392.95
omp+mpi	6.728	18.708	51.195	225.402	776.471	2818.32	6366.56





因为卷积盒的大小是不变的，因此，随着问题规模n的增大，运行时间应该大致呈平方增长，与实验结果相吻合。

可以看到循环展开算法的优化效果其实非常不明显，和并行算法的运行时间基本是相同的。当问题规模较大时，优化表现最好的是 OpenMP + MPI 的算法，第2的是 MPI 算法，纯OpenMP算法也有一定的优化效果。

因为我们令 OpenMP 使用的线程只有2个，而 MPI 可以使用4个节点，因此 MPI 的优化效果理应好于OpenMP。而 OpenMP + MPI 既使用了4个节点，每个节点也使用了多线程，因此优化效果是最好的。

对于加速比，可以看到，无论哪种优化算法，最后的加速比都趋于一个常数。

对于 OpenMP 算法，最终的结果在1 1.5之间，因为在 ft() 以及 convolute() 函数中，进行 OpenMP 多线程化的只是一部分循环，并不是整个函数都交给两个线程来并行，因此加速比达不到2。

对于 MPI 算法，最后的加速比大约为3。我们虽然分配了4个节点，但是真正实行运算的只有3个节点，还有一个主节点负责回收运算结果，因此加速比确实应该为3。

对于 OpenMP+MPI 算法，最后的加速比大约为4。就是在 MPI 3个运算节点的基础上，每个节点又有循环使用了2个线程来进行计算，因此加速比在3的基础上又有了一定的提升。

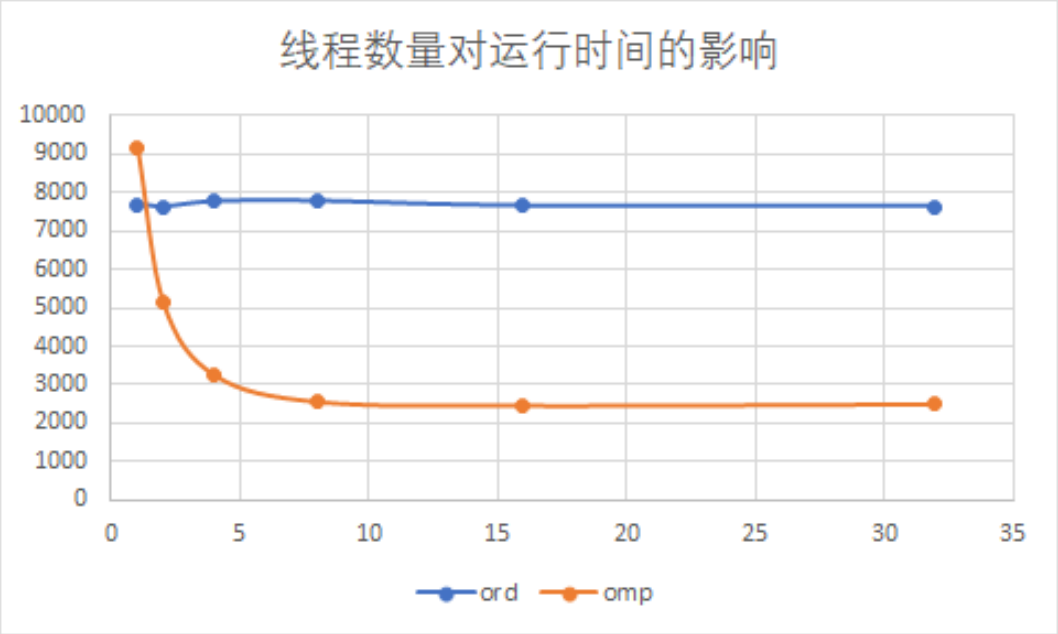
6.3 更改线程数量

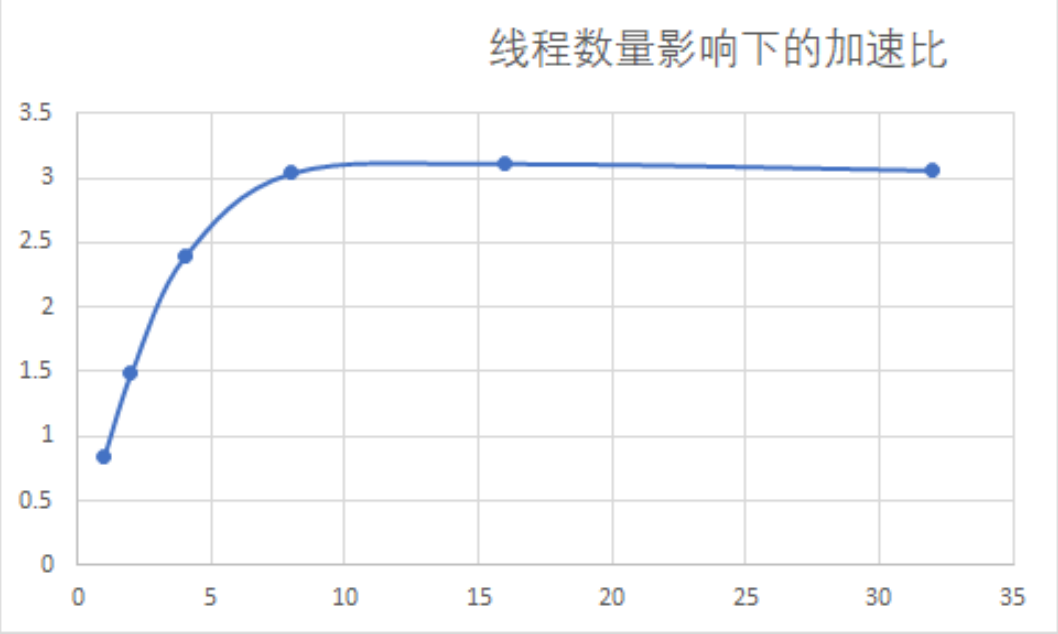
因为远程节点上，每个节点只支持2个线程，因此，我们直接在本机上对 OpenMP 算法的线程数量影响进行研究。

在固定问题规模为256、卷积盒大小为5×5的情况下，我们不

断更改 OpenMP 线程的数量，记录运行的时间以及各自的加速比，得到的结果如下所示。

线程数量	1	2	4	8	16	32
ord	7692.19	7617.28	7756.26	7768.39	7637.22	7630.83
omp	9138.03	5118.18	3242.41	2555.52	2453.2	2491.34





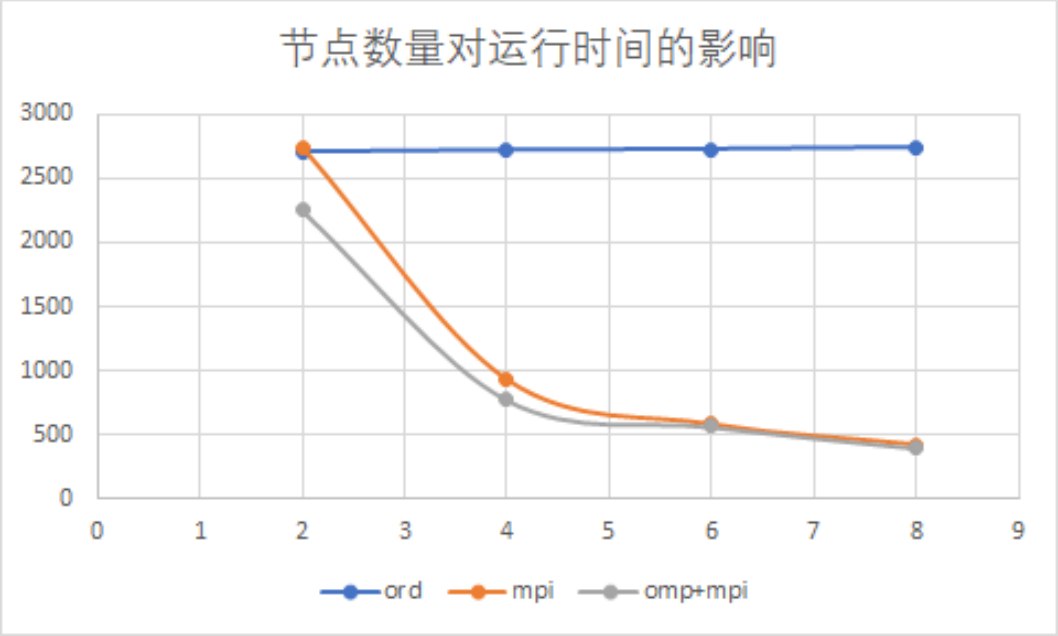
可以看到，随着线程数量的增加，串行算法的时间基本是相同的，而OpenMP的加速比先是不断增加，之后趋于一个常数。

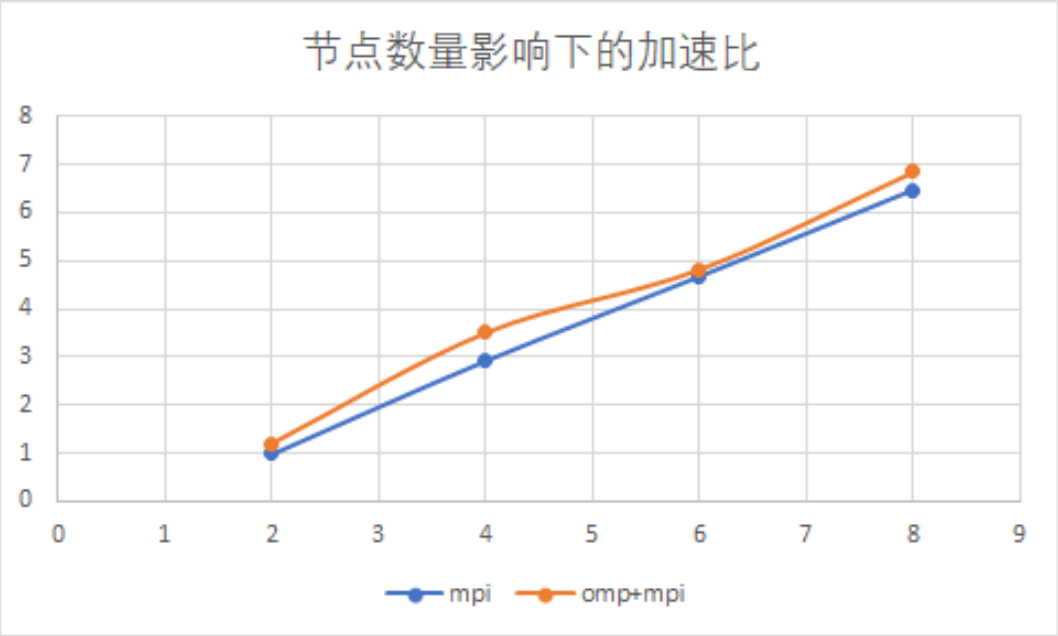
当线程数量较小时，多线程引入的多余时间比优化减少的时间多，因此速度比串行还慢。但是当线程数量达到一定程度后，优化减少的时间就大大增加，总运行时间的就大大减小了。后面，虽然可以设置线程数量不断增加，但硬件实际上无法支撑我们设置的这么多线程，运行时间就趋于相同了。

6.4 更改节点数量

在固定问题规模为256、线程数量为2、卷积盒大小为5×5的情况下，我们不断更改使用的节点的数量，记录运行的时间以及各自的加速比，得到的结果如下所示。

节点数量	2	4	6	8
ord	2703.64	2719.24	2727.81	2744.12
mpi	2743.11	931.016	584.045	424.635
omp+mpi	2254.5	776.471	565.863	400.586





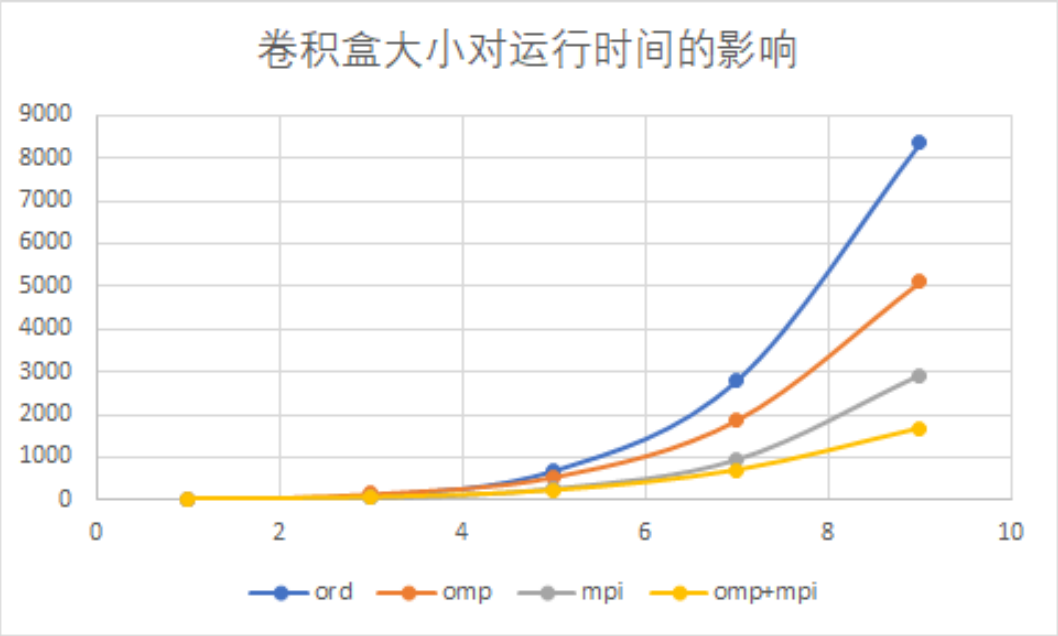
同样的，节点数量对串行算法的运行时间并没有什么影响，但是随着节点数量的增加， MPI 算法和 OpenMP + MPI 算法的加速比基本是按线性增长。

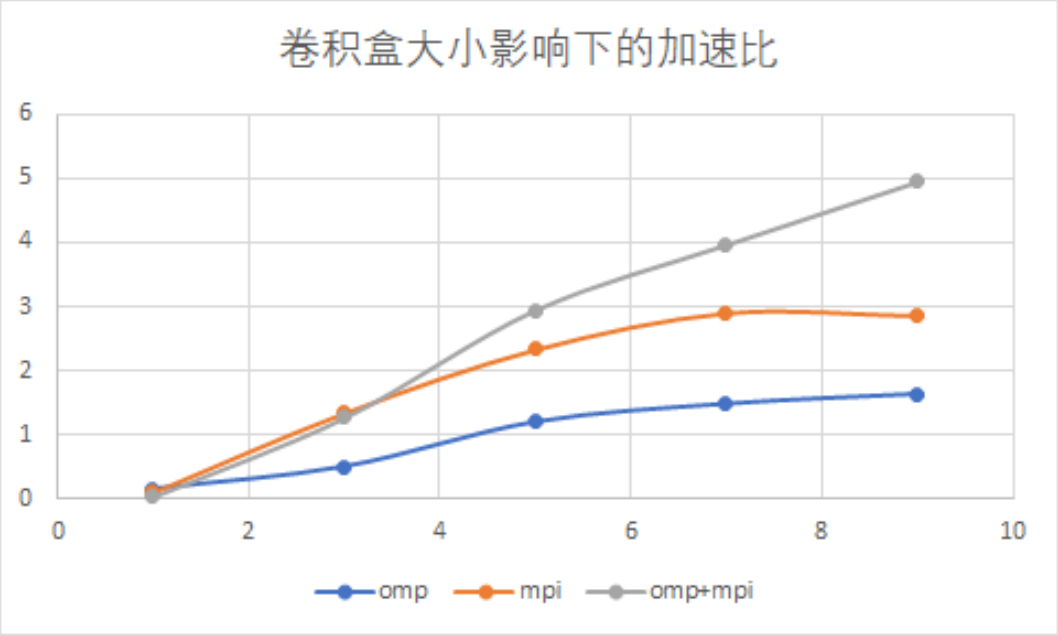
随着节点数量的增加，计算节点的数量也会增加，虽然通信的代价有所增加，但是每个计算节点所分配到的任务量也减少了，而且分配到的任务量与计算节点的数量是成反比的，如此，随着计算节点的增加，运行时间就会相应减少，并且加速比应该是随着节点数量增加呈线性增长的。

6.5 卷积盒大小

在固定问题规模为128、节点数量为4、线程数量为2的情况下，我们不断更改卷积盒的大小，记录运行的时间以及各自的加速比，得到的结果如下所示。

卷积盒大小	1	3	5	7	9
ord	0.994	75.733	663.954	2789.02	8351.08
omp	6.608	150.813	552.253	1877.86	5097.39
mpi	9.727	56.397	284.332	959.137	2915.39
omp+mpi	27.435	59.745	225.402	703.729	1689.62





随着卷积盒大小的增加，串行算法的运行时间迅速增加，另外3种算法的运行时间也在不断增加。当卷积盒较小时，优化效果最好的时 MPI 算法，其次是 OpenMP+MPI 算法，最后是 OpenMP 算法。当卷积盒增大到一定程度时，优化效果最好的变为 OpenMP + MPI 算法，其次是 MPI 算法，最后是 OpenMP 算法。

根据时间复杂度的分析，卷积盒达到一定大小时，时间复杂度就是 $O(n^2m^4)$，因此随着卷积盒大小的增加，运行时间大概是呈4次方增长的，与实验结果相匹配。

当卷积盒较小时，卷积盒对运行时间的影响并不是很大，MPI 算法的优化效果和 OpenMP + MPI 算法的优化效果差不多。当卷积盒增大到一定程度时，并行算法的优化效果才有了一定的体现。和之前时间复杂度的分析相匹配。

7 总结

对于图像光栅化过程中产生的走样问题，我们可以采用反走样算法来减轻产生的走样。

对于反走样算法中的低通滤波算法，要处理的图像的大小以及使用的卷积盒的大小对运行时间有较大的影响。

在当前使用的代码框架下，可以使用 OpenMP 进行多线程优化，也可以使用 MPI 进行多节点优化。受硬件的限制，随着线程数量增加，加速比会不断增大最终稳定在一个常数值。随着节点数量增加（节点数量保证每个节点都有一定任务），加速比也会按比例线性增长。

综合以上研究，我们可以使用傅里叶变换 + 乘积 + 逆傅里叶变换的方式实现低通滤波算法，来减轻光栅化过程中产生的锯齿问题。可以使用 OpenMP 以及 MPI 来加速过程，而且较小的卷积盒大小，不仅效果较好，而且运行速度也较快。

参考文献

[1] huffscan. 反走样技术. https://blog.csdn.net/qq_19272431/article/details/78021570, 2017.

[2] 刘仲，陈海燕，向宏卫. 使用融合乘加加速快速傅里叶变换计算的向量化方法. 国防科技大学学报, 37(02):72–78, 2015.

[3] 史凤丽. 基于集群计算机的图像并行处理. 硕士, 西安科技大学, 2010.

[4] 吕捷，张天序，张必银. Mpi并行计算在图像处理方面的应用. 红外与激光工程, 2004(05):496–499, 2004.

[5] 小楼札记. 数字图像处理-频域滤波-高通/低通滤波. https://www.cnblogs.com/laumians-notes/p/8592968.html, 2018.

[6] 白痴毛. 图形学光栅化详解（rasterization）. https://www.jianshu.com/p/54fe91a946e2, 2017.

[7] 蔡佳佳，李名世，郑锋. 多核微机基于openmp的并行计算. 计算机技术与发展, 2007(10):87–91, 2007.

[8] 闫令琪. Games101. https://sites.cs.ucsb.edu/~lingqi/teaching/games101.html, 2020.