北京化工大学

Beijing University of Chemical Technology

并行程序设计课程大作业报告

并行方法优化图像边缘化处理——

以 Celebrity Face Image Dataset 数据集为例

目录

摘要

边缘检测作为图像处理中的基础且关键步骤，对于图像分析、特征提取和图像识别等应用至关重要。传统的边缘检测算法（如 Sobel 算法）在处理大规模图像数据时，由于计算量大、耗时长，难以满足实时性要求。因此，利用并行计算技术（如 MPI）来加速 Sobel 边缘检测算法，成为了提高处理效率的有效途径。

我们组分别将 Sobel 算法应用于串行处理和并行处理中，得出不同的运行时间，计算出在不同数量的核下，其对性的加速比和效率。在串行方法中，Sobel 边缘检测算法按照逐像素的方式进行处理，我们采用在 0 号进程上完成整个串行方法，利用变量 serial_duration 统计串行运行时间。在并行方法中，首先主进程读取图像并转换为灰度图，然后广播图像尺寸给所有进程，接着将图像分块并分发给各个进程，随后各进程独立进行 Sobel 边缘检测，最后收集各进程的结果并合并为完整的图像并输出。

我们将串行方法与并行方法所得的时间分别记录并制表，得出加速比与效率，通过表格数据总结得出核心数量与运行时间的关系、核心数量与加速比、效率的关系。

关键词：并行计算;图像边缘检测;Sobel 算法;性能分析

第1章　边缘检测算法

1.1 图像边缘检测技术概述

图像边缘检测是一种在图像处理中用于识别对象轮廓的重要技术。其主要目的是检测出图像中强度变化最显著的部分，即边缘。边缘检测可以帮助我们理解图像中的结构和形状，并在许多应用中起到关键作用。

边缘检测通过识别图像中灰度值变化剧烈的区域来找到边缘。这些区域通常表示物体的边界、纹理的变化或其他显著特征。

图像的边缘指的是图像中像素灰度值突然发生变化的区域，如果将图像的每一行像素和每一列像素都描述成一个关于灰度值的函数，那么图像的边缘对应在灰度值函数中是函数值突然变大的区域。函数值的变化趋势可以用函数的导数描述。当函数值突然变大时，导数也必然会变大，而函数值变化较为平缓区域，导数值也比较小，因此可以通过寻找导数值较大的区域去寻找函数中突然变化的区域，进而确定图像中的边缘位置。

1.2 图像边缘检测的应用领域

1. 计算机视觉

边缘检测是计算机视觉中许多高级任务的基础，如对象识别、图像分割、立体视觉和运动检测。

2. 医学成像

在医学成像中，边缘检测用于识别和分析解剖结构，如检测 X 射线图像中的骨折、CT 和 MRI 图像中的肿瘤边界等。

3. 遥感图像处理

在遥感领域，边缘检测用于从卫星图像中提取地物信息，如道路、建筑物和植被区域的边界。

4. 机器人导航边缘检测

帮助机器人理解环境，进行路径规划和障碍物避让。

5. 质量检测

在工业领域，用于检测产品表面的缺陷和不规则性，如在制造过程中检测裂缝、划痕和瑕疵。

6. 人脸识别

在生物识别技术中，边缘检测用于定位和识别人脸特征，如眼睛、鼻子和嘴巴的边界。

7. 自动驾驶

用于检测道路标志、车道线和其他车辆，以辅助自动驾驶系统的决策。

1.3 Sobel 算法原理与实现

1.1.1 Sobel 算法原理

Sobel 算法基于图像梯度的概念来检测边缘。在图像中，边缘通常表现为灰度值的快速变化，即梯度较大的区域。因此，通过计算图像的梯度，我们可以有效地检测出边缘。

Sobel 算法使用两个 3x3 的卷积核，分别对应水平和垂直方向的梯度计算。这两个卷积核通常被称为 Sobel 算子，它们的数学表达式如下：

水平方向 Sobel 算子（Gx）：

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

垂直方向 Sobel 算子（Gy）：

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

这两个算子分别用于计算图像在水平和垂直方向上的梯度值。通过将算子与图像进行卷积操作，我们可以得到每个像素点在水平和垂直方向上的梯度分量。

假设输入图像为 I，则在每个像素点 (x，y)处，水平方向的梯度 G_x和垂直方向的梯度 G_y可以通过与上面的核进行卷积运算得到：

$$\mathrm{Gx} = \sum_{i=-1}^{1}\sum_{j=-1}^{1} \boldsymbol{I}(\boldsymbol{x}+\boldsymbol{i},\boldsymbol{y}+\boldsymbol{j}) * \boldsymbol{Gx}[\boldsymbol{i}+\boldsymbol{1}][\boldsymbol{j}+\boldsymbol{1}]$$

$$\mathrm{Gy} = \sum_{i=-1}^{1}\sum_{j=-1}^{1} \boldsymbol{I}(\boldsymbol{x}+\boldsymbol{i},\boldsymbol{y}+\boldsymbol{j}) * \boldsymbol{Gy}[\boldsymbol{i}+\boldsymbol{1}][\boldsymbol{j}+\boldsymbol{1}]$$

其中，I(x，y)表示图像在(x, y)位置的像素值。边缘强度的计算公式为 $G=\sqrt{Gx^2 + Gy^2}$

1.1.2 Sobel 算法实现

代码部分：

```
/*sobel 算法对图像卷积操作*/
void sobel_edge_detection(Mat &input, Mat &output) {
    int gx[3][3] = {
        {-1, 0, 1},
        {-2, 0, 2},
        {-1, 0, 1}
    };
    int gy[3][3] = {
        {-1, -2, -1},
        { 0,  0,  0},
        { 1,  2,  1}
    };

    for (int y = 1; y < input.rows - 1; y++) {
        for (int x = 1; x < input.cols - 1; x++) {
            int sumX = 0, sumY = 0;

            for (int i = -1; i <= 1; i++) {
                for (int j = -1; j <= 1; j++) {
                    int pixel = input.at<uchar>(y + i, x + j);
                    sumX += pixel * gx[i + 1][j + 1];
                    sumY += pixel * gy[i + 1][j + 1];
                }
            }
            int magnitude = sqrt(sumX * sumX + sumY * sumY);
            output.at<uchar>(y, x) = (magnitude > 255) ? 255 : magnitude;
        }
    }
}
```

第2章　选定数据集的介绍

2.1 数据集的基本信息

数据集名称：Celebrity Face Image Dataset

数据集来源：Kaggle

2.2 数据集的标签、大小和特点

2.2.1 数据集的标签

Arts and Entertainment、Image、Computer Vision、Deep Learning、Celebrities

2.2.2 数据集的大小

大小：53.MB

2.2.3 数据集的特点

1. 多样性

这些图像通常依然保持了一定的多样性，包括性别、种族和年龄等多种面部特征，以便进行广泛的实验和分析。

2. 标注信息

包含面部属性的标注信息。这样能够继续支持面部特征识别和分类任务。

3. 快速处理

更易于处理和管理，适合用作教学、模型快速迭代或概念验证。

4. 实验灵活性

研究人员可以在这个小规模数据集上快速测试各种算法和模型，然后将最佳的结果应用于更大规模的数据集进行进一步验证。

5. 样本均衡性

考虑各类属性的均衡性，以便模型在训练时能够学习到各种特征。

2.3 准备工作

代码部分：

环境配置

1. 安装 OpenMPI

```
sudo apt-get update
sudo apt-get install openmpi-bin openmpi-common libopenmpi-dev
```

2. 配置图像处理库

```
sudo apt-get install libopencv-dev
```

3. 编译命令

```
mpic++ -o version2_huge version2_huge.c `pkg-config --cflags --libs opencv4`
```

4. 运行命令

```
mpiexec -n 4 ./version2_huge //以 4 个进程运行
```

5. 数据集文件

需要在此文件同等目录下建立一个 out 文件夹用于存储处理后的图像

需要一个名为 dataset 的文件夹存储数据集中的图片

需要一个名为 image_list.txt 的文件存储数据集中每一张图片的路径用于后续对每一张图片进行分析

```
ls/home/sanshi/PROJECT/dataset/train/*.jpg >
/home/sanshi/PROJECT/output/test1/image_list.txt
```

第3章　串行实现方法

3.1 串行算法的设计与代码实现

3.1.1 串行算法的设计

串行处理函数在代码中用于按顺序处理每张图像，应用 Sobel 边缘检测，并将结果保存到输出文件中。功能：

(1) 逐个读取图像文件。

(2) 将每张图像转换为灰度图像。

(3) 对灰度图像应用 Sobel 边缘检测。

(4) 将处理后的图像保存到输出文件中。

我们在 0 号进程上完成整个串行方法，利用变量 serial_duration 统计串行运行时间。

3.1.2 串行算法的代码实现

串行处理函数 ‘serial_process’：

```
1.  void serial_process(const vector<string>& image_files) {
2.      for (const string& file : image_files) {
3.          Mat input = imread(file, IMREAD_COLOR);
4.          if (input.empty()) {
5.              printf("Could not open or find the image: %s\n", file.c_s
    tr());
6.              continue;
7.          }
8.
9.          Mat gray;
10.         cvtColor(input, gray, COLOR_BGR2GRAY);
11.         Mat output(gray.rows, gray.cols, CV_8UC1);
```

```
12.
13.         /*图像处理算法*/
14.         sobel_edge_detection(gray, output);
15.
16.         // string output_file = "output/test_small/small_serial/" + f
    ile.substr(file.find_last_of("/\\") + 1);
17.         string output_file = "output/test_huge/huge_serial/" + file.s
    ubstr(file.find_last_of("/\\") + 1);
18.         imwrite(output_file, output);
19.     }
20. }
```

0 号进程完成串行：

```
1.  /*串行方式并记录运行时间*/
2.     if (rank == 0) {
3.         int64 serial_start = getTickCount();
4.
5.         serial_process(image_files);
6.
7.         int64 serial_end = getTickCount();
8.         serial_duration = (serial_end - serial_start) / getTickFrequen
    cy();
9.         printf("串行条件下，数据集运行时
    间： %f seconds\n", serial_duration);
10.    }
```

3.2 代码分析

1. 图像读取：

使用 OpenCV 的 imread 函数读取每个图像文件为彩色图像。IMREAD_COLOR 参数表示将图像读取为三通道 BGR 图像。

如果图像读取失败（即图像为空），打印错误信息并继续处理下一个文件。

2. 颜色转换：

使用 cvtColor 函数将彩色图像转换为灰度图像。Sobel 算法通常应用于单通道图像（即灰度图像），以简化处理和计算。

3. 图像创建：

创建一个与灰度图像尺寸相同的空白输出图像，准备用于存储 Sobel 边缘检测的结果。

4. 边缘检测：

调用 sobel_edge_detection 函数，对灰度图像进行边缘检测，将结果存储在输出图像中。

5. 结果保存：

生成输出文件的名称，使用输入文件路径的最后部分（文件名）来生成输出文件路径，并在前面加上 out/output_ 前缀。

使用 imwrite 函数将处理后的输出图像保存到指定的输出文件中。

第4章 并行实现方法

4.1 并行算法的设计与代码实现

4.1.1 并行算法的设计

1. 主进程读取图像并转换为灰度图。
2. 广播图像尺寸给所有进程。
3. 将图像分块并分发给各个进程。
4. 各进程独立进行 Sobel 边缘检测。
5. 收集各进程的结果并合并为完整的图像并输出。

4.1.2 并行算法的代码实现

并行处理函数‘parallel_process’：

```
/*并行多进程的方式处理图像*/
void parallel_process(int rank, int size, const vector<string>& image_files) {
    for (const string& file : image_files) {
        Mat input, gray;
        if (rank == 0) {
            input = imread(file, IMREAD_COLOR);
            if (input.empty()) {
                printf("Could not open or find the image: %s\n", file.c_str());
                MPI_Abort(MPI_COMM_WORLD, 1);
            }
            cvtColor(input, gray, COLOR_BGR2GRAY);
        }

        int rows, cols;
        if (rank == 0) {
            rows = gray.rows;
```

```
17.             cols = gray.cols;
18.         }
19.
20.         /*广播图像信息*/
21.         MPI_Bcast(&rows, 1, MPI_INT, 0, MPI_COMM_WORLD);
22.         MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
23.
24.         int local_rows = rows / size;
25.         Mat local_gray(local_rows, cols, CV_8UC1);
26.         Mat local_output(local_rows, cols, CV_8UC1);
27.
28.         /*分散图像块*/
29.         MPI_Scatter(gray.data, local_rows * cols, MPI_UNSIGNED_CHAR,
    local_gray.data, local_rows * cols, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WO
    RLD);
30.
31.         /*算法处理*/
32.         sobel_edge_detection(local_gray, local_output);
33.
34.         Mat output;
35.         if (rank == 0) {
36.             output = Mat(rows, cols, CV_8UC1);
37.         }
38.
39.         /*广集图像块*/
40.         MPI_Gather(local_output.data, local_rows * cols, MPI_UNSIGNED
    _CHAR, output.data, local_rows * cols, MPI_UNSIGNED_CHAR, 0, MPI_COMM
    _WORLD);
41.
42.         if (rank == 0) {
43.
44.             // string output_file = "output/test_small/small_parallel
    /" + file.substr(file.find_last_of("/\\") + 1);
45.             string output_file = "output/test_huge/huge_parallel/" +
    file.substr(file.find_last_of("/\\") + 1);
46.             imwrite(output_file, output);
47.         }
48.     }
49. }
```

4.2 代码分析

1. 图像读取和转换 (Rank 0):

图像读取: 主进程 (`rank == 0`) 使用 `imread` 函数读取指定路径的图像文件。若图像读取失败，输出错误信息并终止程序。

图像转换: 主进程将彩色图像转换为灰度图像，这样可以减少后续处理的复杂度。转换后的灰度图像会被广播给所有进程。

2. 图像尺寸广播:

使用 `MPI_Bcast` 函数将图像的行数和列数广播到所有进程。这确保了所有进程都知道图像的尺寸，从而可以正确处理分配给它们的数据块。

3. 数据分发:

计算局部区域: 每个进程处理图像的一个分块。图像的总行数被均分为若干块，每个进程负责一个块。

数据散布: 使用 `MPI_Scatter` 函数将灰度图像的像素数据分发到各个进程。每个进程接收到图像的一部分数据，用于局部处理。

4. 并行计算:

Sobel 边缘检测: 每个进程在其分配的局部图像块上应用 Sobel 边缘检测算法。局部结果保存在 `local_output` 中。

5. 结果收集:

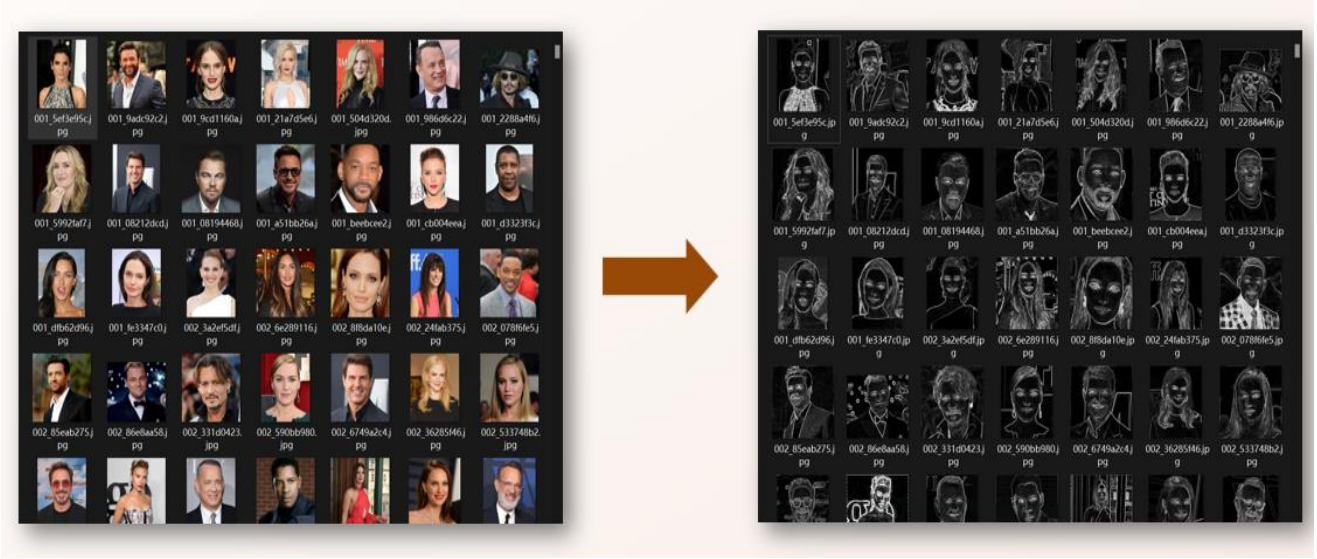
结果收集: 主进程使用 `MPI_Gather` 函数从各个进程收集局部处理结果。所有进程的局部输出被聚合到主进程中，形成完整的图像。

6. 结果保存 (Rank 0):

生成输出文件名: 主进程为每张处理后的图像生成唯一的文件名，并将其保存到指定的输出目录。

保存图像: 使用 `imwrite` 函数将最终处理后的图像保存到文件中。

第5章　算法实现成果



第6章　数据分析

6.1 串行与并行方法的比较

1. 运行时间

串行运行时间：单核（串行）模式下执行整个图像处理任务的时间。它是所有计算在单个核心上进行的时间。

并行运行时间：多核模式下执行整个图像处理任务的时间。计算被分配到多个核心上，理论上，运行时间应该减少。

2. 加速比 (Speedup)

加速比是串行运行时间与并行运行时间的比值，用于衡量并行计算的加速效果。计算公式如下：

Speedup = 串行运行时间/并行运行时间

结论：

高加速比表示并行化效果好，程序在多核下运行比单核运行显著加快。

加速比接近于核心数量 表示程序性能接近理论最佳状态。

3. 效率 (Efficiency)

效率衡量了并行计算的资源利用率。计算公式如下：

Efficiency = Speedup/核心数量

结论：

高效率表示计算资源（核心）被有效利用，性能接近理论最佳值。

效率下降可能是因为并行化开销、负载不均匀、通信延迟等原因。

由表 6.1 和 6.2 看出：

加速比：随着核心数量的增加，加速比增加，但增长幅度逐渐减小 。

效率：随着核心数量的增加，效率逐渐降低，核心利用率在减少。

表 6.1 半个数据集下不同核心数量下程序并行和串行运行时间、加速比和效率的对比情况

半个数据集				
核心	并行时间	串行时间	加速比	效率
2	6.171959	12.1984986	1.976438696	0.988219348
4	4.797252		2.542809634	0.635702408
6	4.709691		2.5900847	0.431680783
8	4.550054		2.680956885	0.335119611
10	4.369755		2.791574951	0.279157495

表 6.2 半个数据集下不同核心数量下程序并行和串行运行时间、加速比和效率的对比情况

整个数据集				
核心	并行时间	串行时间	加速比	效率
2	13.730034	26.7737612	1.950014195	0.975007098
4	10.662122		2.511110002	0.627777501
6	10.469258		2.55736951	0.426228252
8	10.081453		2.655744286	0.331968036
10	8.802764		3.041517551	0.304151755

6.2 并行方法在不同核数下的性能分析

由图 6.1 和 6.2 来看核心数量与运行时间的关系，可以看出增加核心数量可以减少并行运行时间，提高程序的执行速度。但我们可以观察到增加核心数量的收益会逐渐减少。

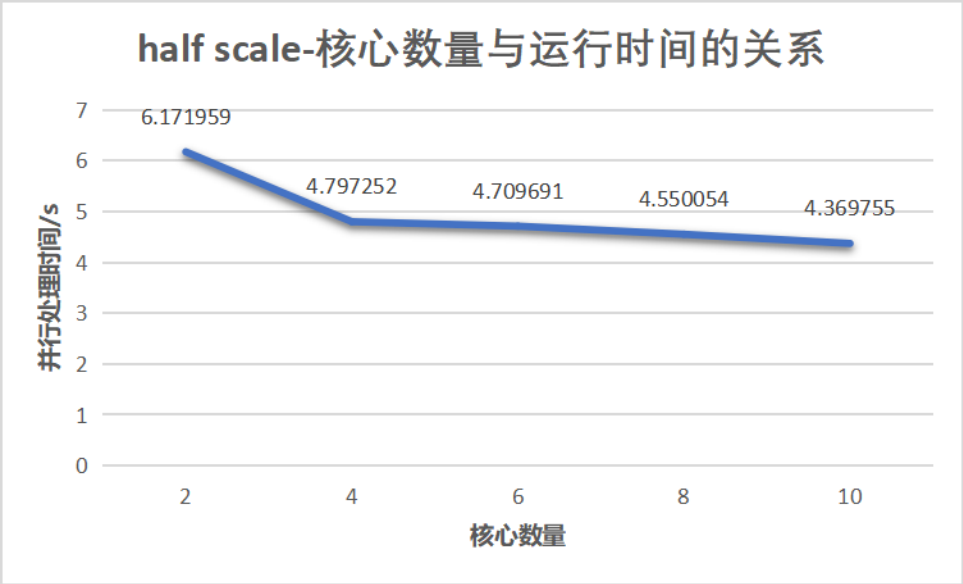


图 6.1 half scale-核心数量与运行时间的关系

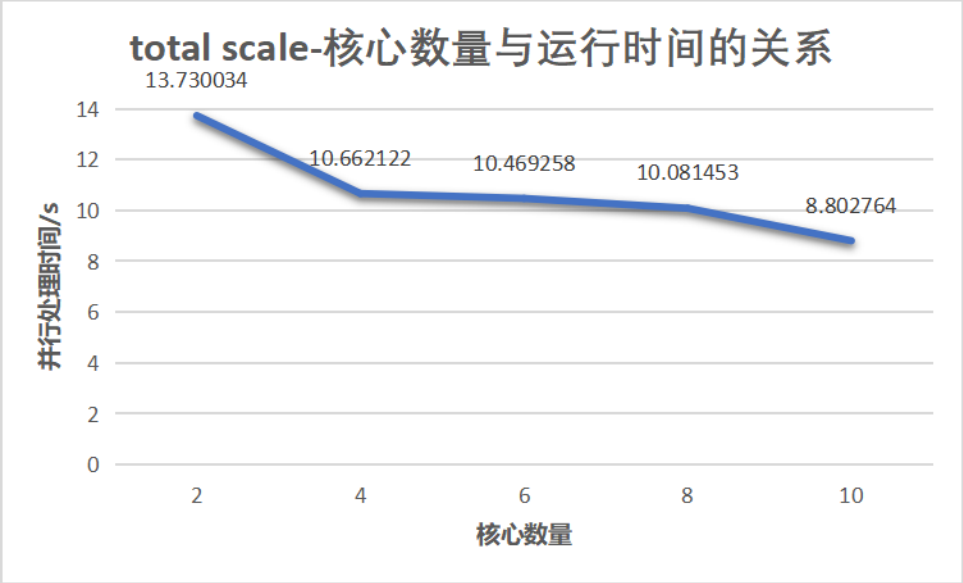


图 6.2 total scale-核心数量与运行时间的关系

由图 6.3 和 6.4 再看核心数量与加速比、效率的关系。虽然增加核心数量可以提升并行计算的性能，但这种提升并不是线性因此在实际应用中。我们需要在核心数量和实际效益之间找到一个平衡点。

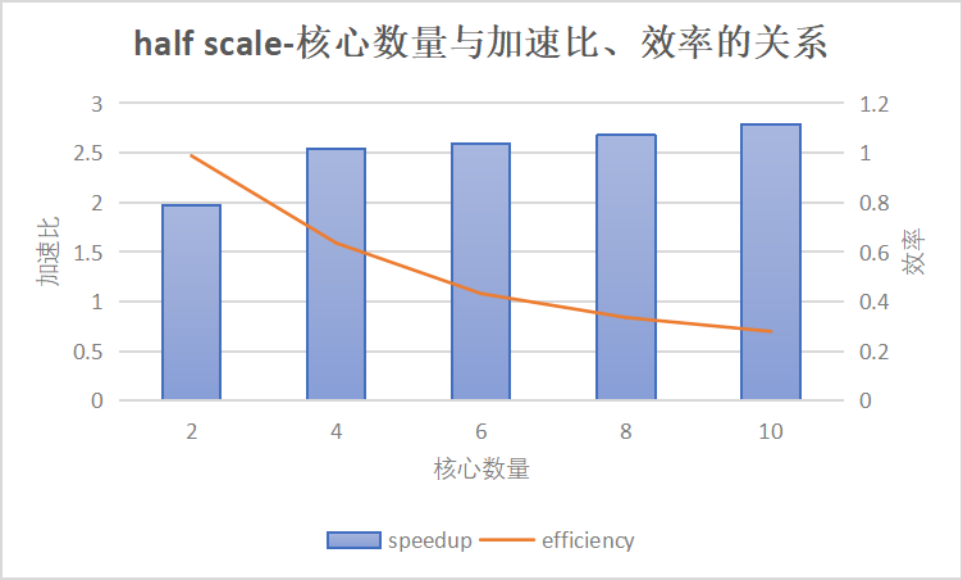


图 6.3 half scale-核心数量与加速比、效率的关系

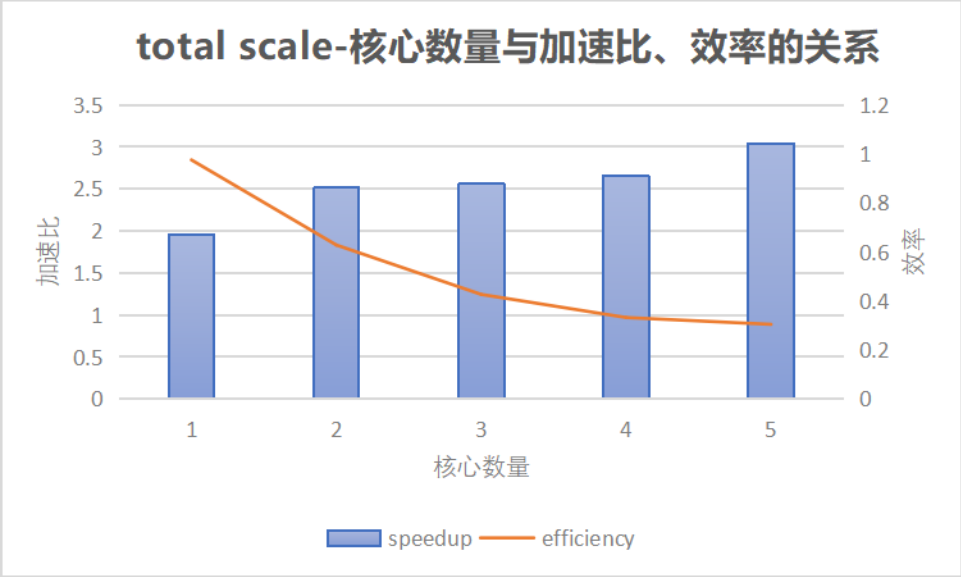


图 6.4total scale-核心数量与加速比、效率的关系

第7章　结论

7.1 针对数据分析结果总结

1、不同核心数量下程序并行和串行运行时间、加速比和效率的对比结论

(1) 加速比：随着核心数量的增加，加速比增加，但增长幅度逐渐减小。

(2) 效率：随着核心数量的增加，效率逐渐降低，核心利用率在减少。

2、核心数量与运行时间的关系结论

(1) 增加核心数量可以减少并行运行时间，提高程序的执行速度。

(2) 随着核心数量的增加，运行时间的减少幅度逐渐减小，增加核心数量的收益会逐渐减少。

3、核心数量与加速比、效率的关系结论

(1) 增加核心数量可以提升程序的加速比，但加速比的增长幅度会逐渐减小。

(2) 随着核心数量的增加，效率会逐渐降低，这反映了并行计算中增加核心数量并不能线性地提升性能