NCUSCC选拔考核-C语言项目实验报告

管嘉豪

考核要求

1.安装虚拟机：

- 在虚拟机中安装 Ubuntu 22.04 LTS 操作系统。
- 配置虚拟机的网络连接，确保可以正常联网。

2.安装 C 语言编译器：

- 安装最新版本的 gcc（可通过 PPA 安装最新稳定版）。
- 验证编译器安装成功，并确保其正常工作。

3.实现排序算法：

- 使用 C 语言手动实现以下算法（不调用任何库函数）：
  - 快速排序（递归 + 非递归版本）：基础排序算法，但需考虑 pivot 选择（如随机 pivot、三数取中）对性能的影响。
  - 归并排序（并行化版本）：基于 OpenMP 实现并行归并排序（利用#pragma omp parallel等指令，将大数组分块后多线程处理）。
- 运行测试代码，确认各排序算法的正确性。

4.生成测试数据：

- 编写代码或脚本自动生成测试数据到单独的数据文档，程序运行的时候需体现从文档读取数据的过程（随机生成浮点数或整数）。
- 测试数据应覆盖不同规模的数据集，其中必须包含至少 100 000 条数据的排序任务。

5.编译与性能测试：

- 使用不同等级的 gcc 编译优化选项（如 -O0, -O1, -O2, -O3, -Ofast 等）对快速排序和归并排序代码进行编译。
- 记录各优化等级下的排序算法性能表现（如执行时间和资源占用）。

6.数据记录与可视化：

- 编写脚本收集每个编译等级的运行结果和性能数据。
- 分析算法的时间复杂度，并将其与实验数据进行对比。
- 将数据记录在 CSV 或其他格式文件中。
- 使用 Python、MATLAB 等工具绘制矢量图，展示实验结论。

7.撰写实验报告：

- 撰写一份详细的实验报告，内容应包括：
- 实验环境的搭建过程（虚拟机安装、网络配置、gcc 安装等）。两种排序算法的实现细节。测试数据的生成方法，以及收集实验数据的过程。不同编译优化等级下的性能对比结果。数据可视化部分（附图表）。实验过程中遇到的问题及解决方案。
- 报告必须采用 LaTeX 或 Markdown 格式撰写。

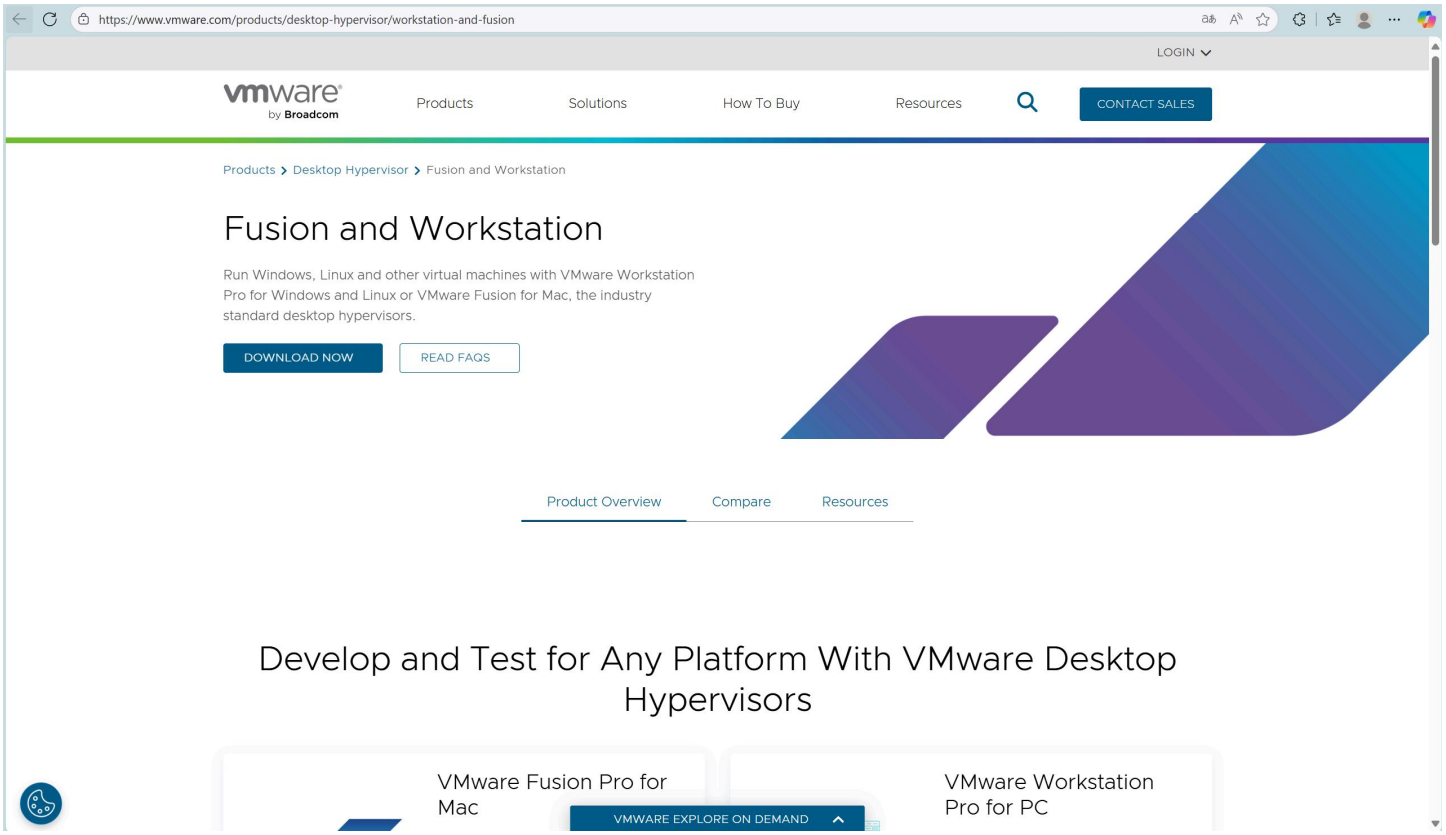
提交要求

- 将完整的实验报告和源代码上传至个人 GitHub 仓库。
- 提交报告的 PDF 文件及仓库链接。

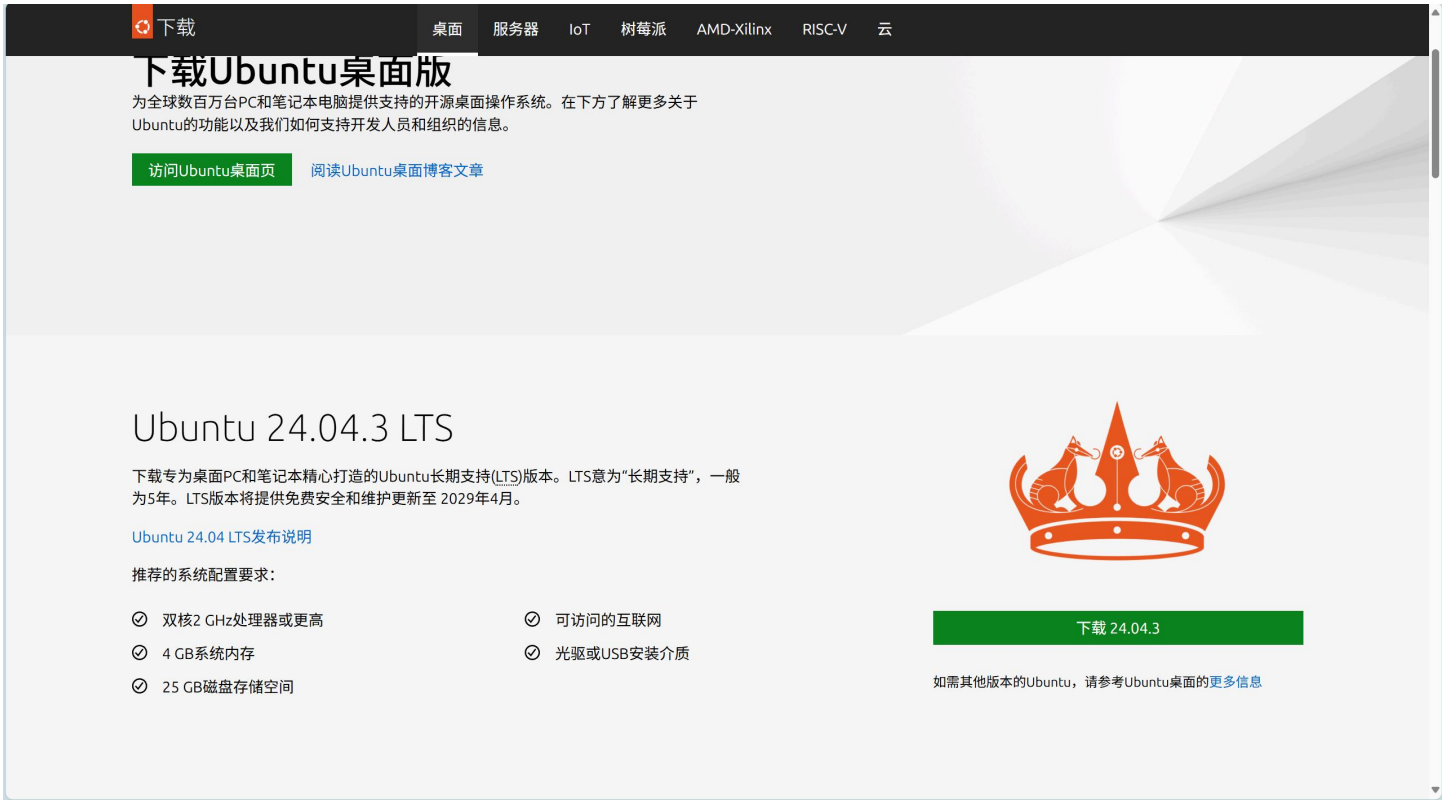
正式考核部分

一.搭建实验环境

实验目的：了解如何搭建虚拟机环境，同时熟悉Ubuntu系统的一些基本操作，学会如何使用终端。
实验流程1：使用Vmware Workstation创建虚拟机，并在虚拟机上安装Ubuntu24.04操作系统(由于我提前装了虚拟机，所以版本不太对😰)
(1)前往Vmware Workstation官网下载搭建虚拟机环境的软件



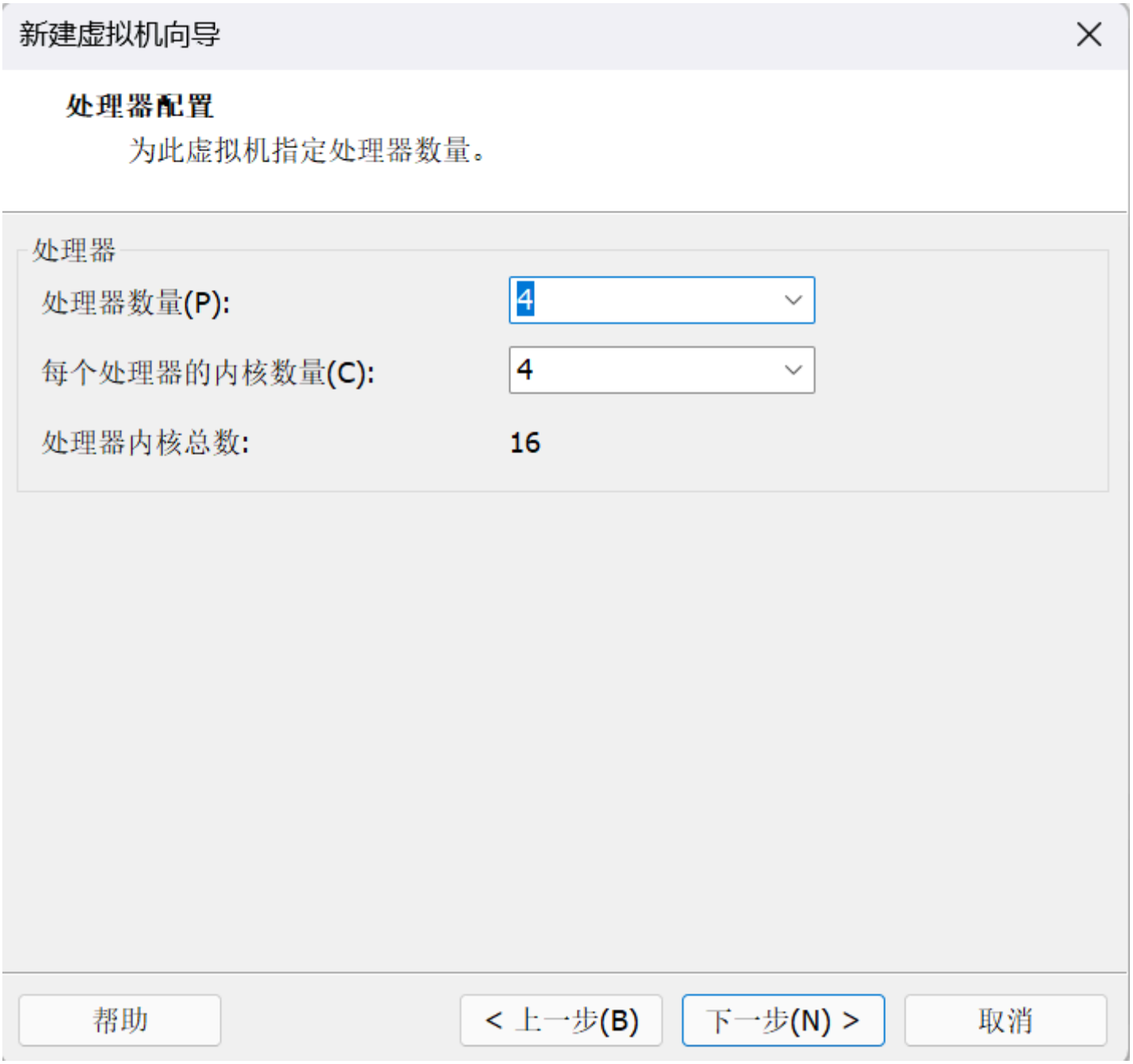
(2)前往Ubuntu官网获取Ubuntu24.04的ISO映像文件



(3)新建虚拟机，并分配内存，处理器和磁盘。


新建虚拟机向导
vmware
WORKSTATION
PRO™
17
欢迎使用新建虚拟机向导
您希望使用什么类型的配置?
典型(推荐)(T)
通过几个简单的步骤创建 Workstation 17.5 or later 虚拟机。
自定义(高级)(C)
创建带有 SCSI 控制器类型、虚拟磁盘类型以及与旧版 VMware 产品兼容性等高级选项的虚拟机。
帮助
< 上一步(B)
下一步(N) >
取消


新建虚拟机向导
安装客户机操作系统
虚拟机如同物理机，需要操作系统。您将如何安装客户机操作系统?
安装来源:
安装程序光盘(D):
无可用驱动器
安装程序光盘映像文件(iso)(M):
ubuntu-24.04.3-desktop-amd64.iso
浏览(R)...
无法读取此文件。
指定其他文件或选择其他选项继续。
稍后安装操作系统(S)。
创建的虚拟机将包含一个空白硬盘。
帮助
< 上一步(B)
下一步(N) >
取消


新建虚拟机向导
处理器配置
为此虚拟机指定处理器数量。
处理器
处理器数量(P):
4
每个处理器的内核数量(C):
4
处理器内核总数:
16
帮助
< 上一步(B)
下一步(N) >
取消


新建虚拟机向导
此虚拟机的内存
您要为此虚拟机使用多少内存?
指定分配给此虚拟机的内存量。内存大小必须为 4 MB 的倍数。
128 GB
64 GB
32 GB
16 GB
8 GB
4 GB
2 GB
1 GB
512 MB
256 MB
128 MB
64 MB
32 MB
16 MB
8 MB
4 MB
此虚拟机的内存(M):
4096
MB
最大推荐内存:
27.7 GB
推荐内存:
4 GB
客户机操作系统最低推荐内存:
2 GB
帮助
< 上一步(B)
下一步(N) >
取消

(4)进入系统后按提示安装系统，重启电脑。

实验流程2：网络配置

(1)在配置虚拟机时选择NAT连接

实验流程3：安装gcc编译器以及其他所需的软件

(1)在终端输入下列命令安装gcc编译器

```
#更新软件包列表
sudo apt update
#安装gcc编译器
sudo apt install gcc
sudo apt install build-essential
```

(2)安装 OpenMP 支持

```
sudo apt install libomp-dev
```

(3)安装python相关内容

```
#安装 Python 用于数据可视化
sudo apt install python3 python3-pip
#安装 Python 数据处理库
pip3 install matplotlib pandas numpy
```

问题发现

(1) 配置网络时选择NAT连接有时会出现连不上网的情况，有时重启后也没法连上网，这里参考了b站的视频成功解决了。

(2)在安装Python数据处理库时会出现网络不稳定的问题，这里可以选择清华的镜像网站下载。

```
pip install -i https://pypi.tuna.tsinghua.edu.cn/simple matplotlib pandas numpy
```

二.快速排序递归和非递归版本以及并行归并排序的实现

实验目的:考察代码能力，理解排序的实现，了解OpenMp的使用。
注：通过指针调用来实现整形与浮点型的泛用，并先完成交换函数与比较函数

```
//交换函数
void swap(void* a,void* b,size_t size)
{
   char* p=(char*)a;
   char* q=(char*)b;
   char temp;
   for(size_t i=0;i<size;i++){
       temp=p[i];
       p[i]=q[i];
       q[i]=temp;
   }
}

//整数比较函数
int compare_int(const void* a, const void* b)
{
   return (*(int*)a-*(int*)b);
}

//浮点数比较函数
int compare_float(const void* a,const void* b)
{
   float c=*(float*)a-*(float*)b;
   if(c>0) return 1;
   if(c<0) return -1;
   return 0;
}
```

(1)递归快速排序

```
//三数取中选择pivot
void* median_of_three(void* base, int low,int high,size_t size,int(*compare)(const void*,const
{
    int mid=low+(high-low)/2;
    char* arr=(char*)base;

    char* low_p=arr+low*size;
    char* mid_p=arr+mid*size;
    char* high_p=arr+high*size;

    if(compare(low_p,mid_p)>0){
        swap(low_p,mid_p,size);
    }
    if(compare(low_p,high_p)>0){
        swap(low_p,high_p,size);
    }
    if(compare(mid_p,high_p)>0){
        swap(mid_p,high_p,size);
    }
    return mid_p;
}

//分区函数
int partition(void* base,int low,int high,size_t size,int(*compare)(const void*,const void*))
{
    char* pivot_p= median_of_three(base, low, high, size, compare);
    char* arr=(char*)base;

    swap(pivot_p,arr+high*size,size);
    void* pivot=arr+high*size;

    int i=low-1;
    for(int j=low;j<high;j++){
        if(compare(arr+j*size,pivot)<=0){
            i++;
            swap(arr+i*size,arr+j*size,size);

        }
    }
    swap(arr+(i+1)*size,pivot,size);
    return i+1;
}

//递归快速排序
void quick_sort_recursive(void* base,int low,int high,size_t size,int(*compare)(const void*,cor
{
```

```
    if(low<high){
        int pi=partition(base,low,high,size,compare);
        quick_sort_recursive(base,low,pi-1,size,compare);
        quick_sort_recursive(base,pi+1,high,size,compare);
    }
}
```

代码实现：先通过三数取中法选定基准数，再将基准数与数组最后的数交换以实现将基准数放于数组尾端的操作，然后再利用i和j两个指针将其余数据以类似冒泡排序的方式排好序，此时i所指向的恰为最后一个比基准数小的数，通过与第(i+1)个数交换即可实现分区操作，最后通过不断调用分区函数即可按序将每个数排序好。

(2)非递归快速排序

```
//定义栈
typedef struct
{
    int* data;
    int capacity;
    int top;
}Stack;

Stack* create_stack(int capacity)
{
    Stack* stack=(Stack*)malloc(sizeof(Stack));
    if(stack==NULL) return NULL;

    stack->data=(int*)malloc(sizeof(int));
    if(stack->data==NULL){
        free(stack);
        return NULL;
    }
    stack->capacity = capacity;
    stack->top = -1;
    return stack;
}

void free_stack(Stack* stack){
    if(stack!=NULL){
        free(stack->data);
        free(stack);
    }
}

int is_empty(Stack* stack){
    return stack->top==-1;
}

void push(Stack* stack,int value){
    if(stack->top<stack->capacity-1){
        stack->data[++stack->top]=value;
    }
}

int pop(Stack* stack){
    if(!is_empty(stack)){
        return stack->data[stack->top--];
    }
```

```
        return -1;
    }
```

鉴于无法使用递归，所以选择用栈来实现，故先定义与栈相关的一些函数。

```
//非递归快速排序
void quick_sort_iterative(void* base,size_t num,size_t size,int(*compare)(const void*,const vo
{
    if(num<=1)return;

    Stack* stack=create_stack((int)(num*2));
    if(stack==NULL){
        printf("错误，无法创建栈\n");
        return;
    }
    push(stack,0);
    push(stack, (int)(num-1));

    char*arr=(char*)base;

    while(!is_empty(stack)){
        int high=pop(stack);
        int low=pop(stack);

        if(low<high){

            char* pivot_p= median_of_three(base, low, high, size, compare);

            swap(pivot_p,arr+high*size,size);
            void* pivot=arr+high*size;

            int i=low-1;
            for(int j=low;j<high;j++){
                if(compare(arr+j*size,pivot)<=0){
                    i++;
                    swap(arr+i*size,arr+j*size,size);

                }
            }
            swap(arr+(i+1)*size,pivot,size);
            int pi=i+1;

//将分区后的左右数组压入栈
            if(pi-1>low){
                push(stack,low);
                push(stack,pi-1);
            }
            if(pi+1<high){
                push(stack,pi+1);
                push(stack,high);
```

```
                }
            }
        }
        free_stack(stack);
    }
```

代码实现：前面与递归版本相同的部分就不过多赘述，主要区别在于非递归版本的排序是通过栈的压入和弹出和while循环来实现，利用栈先入后出的特点不断缩小排序的数组的范围，直到每个子数组都只含一个数，最后实现每个部分都按序排列。

(3)并行归并排序

```
//内存复制函数
void copy_memory(void*dest, const void* src,size_t n){
    char* char_dest=(char*)dest;
    const char* char_src=(const char*)src;

    for(size_t i=0;i<n;i++){
        char_dest[i]=char_src[i];
    }
}

//串行归并函数
void merge(void*array,void* left,void* right,size_t left_size,size_t right_size,size_t element_
{
    char* char_array=(char*)array;
    char* char_left=(char*)left;
    char* char_right=(char*)right;

    size_t i=0,j=0,k=0;

    while(i<left_size&&j<right_size){
        if(compare(char_left+i*element_size,char_right+j*element_size)<=0){
            copy_memory(char_array+k*element_size,char_left+i*element_size,element_size);
            i++;
        }else{
            copy_memory(char_array+k*element_size,char_right+j*element_size,element_size);
            j++;
        }
        k++;
    }

//复制剩余元素
    while(i<left_size){
        copy_memory(char_array+k*element_size,char_left+i*element_size,element_size);
        i++;
        k++;
    }
    while(j<right_size){
        copy_memory(char_array+k*element_size,char_right+j*element_size,element_size);
        j++;
        k++;
    }
}

//串行归并排序函数
void merge_sort_serial(void* array,size_t size,size_t element_size,int(*compare)(const void*,c
{
```

```
    if(size<=1){
        return;
    }

    size_t mid=size/2;
    char* char_array=(char*) array;

    //分配左右数组内存
    void* left=malloc(mid*element_size);
    void* right=malloc((size-mid)*element_size);

    if(!left||!right){
        free(left);
        free(right);
        return;
    }

    //复制数据到左右数组
    copy_memory(left,char_array,mid*element_size);
    copy_memory(right,char_array+mid*element_size,(size-mid)*element_size);

    //递归排序
    merge_sort_serial(left,mid,element_size,compare);
    merge_sort_serial(right,size-mid,element_size,compare);

    //合并
    merge(array,left,right,mid,size-mid,element_size,compare);

    free(left);
    free(right);
}

//并行归并排序
void merge_sort_parallel(void* array,size_t size,size_t element_size,int(*compare)(const void*,
{
    if(size<=1){
        return;
    }
    //调整并行深度阈值
    if(max_depth<=0||size<1000){
        merge_sort_serial(array,size,element_size,compare);
        return;
    }

    size_t mid=size/2;
    char* char_array=(char*)array;
```

```
    //分配左右数组内存
    void* left=malloc(mid*element_size);
    void* right=malloc((size-mid)*element_size);

    if(!left||!right){
        free(left);
        free(right);
        return;
    }

    //复制数据到左右数组
    copy_memory(left,char_array,mid*element_size);
    copy_memory(right,char_array+mid*element_size,(size-mid)*element_size);

    //并行处理左右数组
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            merge_sort_parallel(left,mid,element_size,compare,max_depth-1);
        }
        #pragma omp section
        {
            merge_sort_parallel(right,size-mid,element_size,compare,max_depth-1);
        }
    }

    //合并结果
    merge(array,left,right,mid,size-mid,element_size,compare);

    free(left);
    free(right);
}
```

代码实现：首先将乱序的数组分为左右两部分通过递归来实现两个有序的数组，然后利用i,j,k三个指针来遍历左右数组和合并后的数组，依次将更小的数复制到合并后的数组里，但上述是串行实现，只能一次次进行。而下文的并行排序就是通过利用OpenMP的并行 sections 在递归的每一层同时处理左右两个子数组，以此实现加速完成的效果。

三.测试数据的生成与读取

实验目的:学会使用代码生成随机数据，并读取以实现程序的测试

(1)随机数据的生成

```
// 生成随机整数测试数据
void generate_int_test_data(const char* filename, int size) {
    FILE* file = fopen(filename, "w");
    if (file == NULL) {
        printf("Error opening file for writing!\n");
        return;
    }

    srand(time(NULL));
    fprintf(file, "%d\n", size);

    for (int i = 0; i < size; i++) {
        // 生成 [0, 10000) 范围内的随机整数
        int value = rand() % 10000;
        fprintf(file, "%d\n", value);
    }

    fclose(file);
    printf("Generated integer test data: %s with %d elements\n", filename, size);
}

// 生成随机浮点数测试数据
void generate_double_test_data(const char* filename, int size) {
    FILE* file = fopen(filename, "w");
    if (file == NULL) {
        printf("Error opening file for writing!\n");
        return;
    }

    srand(time(NULL));
    fprintf(file, "%d\n", size);

    for (int i = 0; i < size; i++) {
        // 生成 [0, 10000) 范围内的随机浮点数
        double value = (double)rand() / RAND_MAX * 10000.0;
        fprintf(file, "%.6f\n", value);
    }

    fclose(file);
    printf("Generated double test data: %s with %d elements\n", filename, size);
}
```

该代码通过系统时间time(NULL)作为种子以生成随机的数据用于测试

(2)文件数据的读取

```
// 从文件读取整数数据
int* read_int_data_from_file(const char* filename, int* size) {
    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        printf("Error opening file for reading: %s\n", filename);
        return NULL;
    }

    fscanf(file, "%d", size);
    int* data = (int*)malloc(*size * sizeof(int));

    for (int i = 0; i < *size; i++) {
        fscanf(file, "%d", &data[i]);
    }

    fclose(file);
    return data;
}

// 从文件读取浮点数数据
double* read_double_data_from_file(const char* filename, int* size) {
    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        printf("Error opening file for reading: %s\n", filename);
        return NULL;
    }

    fscanf(file, "%d", size);
    double* data = (double*)malloc(*size * sizeof(double));

    for (int i = 0; i < *size; i++) {
        fscanf(file, "%lf", &data[i]);
    }

    fclose(file);
    return data;
}
```

这里通过两个函数将生成的数据从文件中读取出来并用于后续代码的排序测试，但具体怎么实现文件的读取我还不是很了解

四.不同编译等级下的性能优化对比

实验目的：了解不同的编译优化等级，体会其对不同排序的优化区别

(1)不同的编译优化等级

- O0（默认不优化）编译速度最快，调试信息非常准确，但生成的代码执行效率最低。
- O1 (基本优化) 编译时间相对较短，调试体验仍然较好，但优化程度有限，性能提升不明显。
- O2 (中级优化) 在代码大小和执行速度之间取得平衡，通常推荐使用，但编译时间较长，调试体验变差。
- O3 (高级优化) 包含所有-O2的优化，并进一步优化，但编译时间更长，生成的代码体积可能更大。
- Ofast（激进优化）在-O3的基础上，允许违反一些严格的标准（如IEEE浮点标准），以追求更快的速度，但可能引发不符合标准的行为。

(2)性能对比结果

并行归并排序的优化程度相对于快速排序较小，我觉得应该是出于OpenMP的并行区域有固定开销。在同优化等级下若数据较少，则快速排序更快，但如果是数据较多时，则归并排序更快。

五.数据可视化

实验目的：直观感受不同优化等级下的数据。

(1)*在完成python相关内容的安装后搭建虚拟环境执行命令*

```
#创建虚拟环境
python3 -m venv myenv
#激活虚拟环境
source myenv/bin/activate
#退出虚拟环境
deactivate
```

(2)*脚本实现*

附：

> 在整个任务完成中我仍有许多问题没解决，比如shell脚本的编写，python如何生成图像，以及如何实现跨文件的读取。这些只能有待我后续的继续学习了。