南開大學
Nankai University

计算机学院

并行程序设计实验报告

MPI 编程实验报告

马浩祎

学号：2213559

专业：计算机科学与技术

2024 年 6 月 3 日

目录

1 摘要

Abstract

本次实验是 MPI 多进程编程实验。首先实现了两种多进程优化策略：查询内和查询间；然后进行了 Windows 和 Linux 双平台下的查询间实验对比设计；创新地，我还设计了另一种基于存储格式的转换策略，并以此为基础设计了层进式结合的算法系列，逐步结合 **MPI 多进程，OpenMP 多线程，SIMD 向量化**的并行优化方法，最终对其层进式的性能差异进行了分析。特别地，为了深究 MPI 通信机制的差异，我重点探究了 **MPI 阻塞和非阻塞通信方式**，并给出了性能的差异及分析。而且值得一提的是，在实验中特别设置了单进程组，重点验证了与串行算法的性能差异，证实了 MPI 的通信环境等一系列开销的存在。
关键词： 多进程编程，MPI，倒排索引，列表求交

2 实验相关介绍

2.1 问题描述

2.1.1 倒排索引

在当代的数据库里，索引是检索数据最有效率的方式。但考虑搜索引擎的如下特点：

- 搜索引擎面对的是海量数据。像 Google，百度这样大型的商业搜索引擎索引都是亿级甚至百亿级的网页数量
- 搜索引擎使用的数据操作简单。一般而言，只需要增、删、改、查几个功能，而且数据都有特定的格式，可以针对这些应用设计出简单高效的应用程序。
- 搜索引擎面临大量的用户检索需求。这要求搜索引擎在检索程序的设计上要分秒必争，尽可能的将大运算量的工作在索引建立时完成，使检索运算尽量的少。

我们就不能去构建简单的索引了，早在 1958 年，IBM 就在一次会议上展示了一台“自动索引机器”。在当今的搜索引擎里也经常能看到它的身影，它就是——倒排索引。倒排索引又叫反向索引，它是一种逆向思维运算，是现代信息检索领域里面最有效的一种索引结构。

为满足用户需求，顺应信息时代快速获取信息的趋势，开发者们在进行搜索引擎开发时对这些信息数据进行了逆向运算，开发出“关键词——文档”形式的映射结构，实现了通过了物品属性信息对物品进行映射，可以帮助用户快速定位到目标信息，极大地降低了信息获取难度。

2.1.2 索引求交

有了倒排索引后，搜索引擎在查询处理过程中，需要对这些索引集合进行求交操作，这个也是运算时间占比非常大的部分。假定用户提交一个 k 个词的查询，查询词分别是 $t_1, t_2, \ldots, t_k$，这 k 个关键词对应的倒排列表为 $\ell_{t_1}, \ell_{t_2}, \ldots, \ell_{t_k}$ 求交算法返回 $\cap_{1 \le i \le k} l(t_i)$。

首先求交会按照倒排列表的长度对列表进行升序排序，使得：

$$|l(t_1)| \le |l(t_2)| \le \cdots \le |l(t_k)|$$

然后求交操作返回 k 个倒排列表的公共元素 $\cap_{1 \le i \le k} l(t_i)$。

2.2 MPI 简介

MPI（Message Passing Interface）是一种广泛使用的并行编程模型和通信协议，主要用于在分布式计算环境中实现多进程通信。MPI 提供了一套标准化的接口，使程序能够在不同计算节点之间传递消息，从而实现并行计算，如图2.1。可以说，它是一种多进程编程的并行编程工具。MPI 编程的风格和特点如下：

- 显式并行性。开发者需要显式地管理并行任务和进程间通信，清晰地定义各进程的任务和数据流。
- 灵活的通信模型。提供点对点通信和集体通信，适用于各种并行计算需求。
- 高效性。通过直接内存复制实现高效数据传输，适合高性能计算。
- 可扩展性。能够支持大规模并行计算，可在成千上万个进程上运行。
- 跨平台性。MPI 是一个标准，支持多种操作系统和硬件架构。

基于其高效方便的并行处理能力，MPI 目前被广泛应用于科学计算、工程模拟、数据分析等领域。在气象模拟、大规模线性代数运算、计算流体力学、结构分析、大数据处理、机器学习任务的分布式训练等领域均有它的身影。

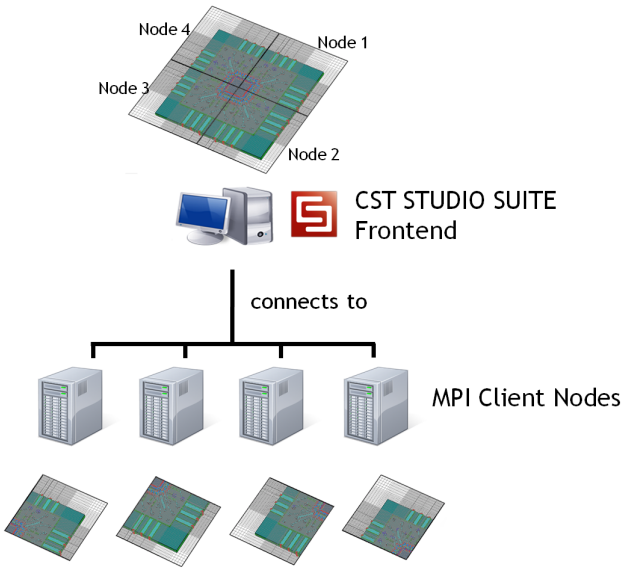


图 2.1: MPI 模拟流程

但是当程序员使用其编程时，也需要注意一些问题：

- 初始化和终止：在程序开始时使用 MPI_Init 初始化 MPI 环境，结束时使用 MPI_Finalize 进行清理。确保所有进程在 MPI_Init 之后和 MPI_Finalize 之前的代码是有效的 MPI 代码。
- 进程同步：注意进程间的同步问题，避免由于进程不同步导致的数据不一致或死锁。使用适当的同步机制，如 MPI_Barrier，确保进程在关键点同步。
- 数据传输：确保数据传输的一致性，发送和接收的数据类型、数量要匹配。使用非阻塞通信时，确保在数据有效之前进行同步或等待。
- 内存管理：注意内存管理，确保在发送和接收数据时避免内存泄漏和非法访问。

- 通信性能：尽量减少通信开销，优化通信模式，如合并多次通信、减少不必要的同步。可使用集体通信函数（如 MPI_Bcast, MPI_Reduce）代替多个点对点通信，提升效率, 也可以对比一下 MPI 特有的阻塞通信，非阻塞通信，单边通信，双边通信的性能差异。
- 负载均衡：确保各进程的计算任务均衡分配，避免出现某些进程过载或空闲的情况。根据任务特点进行合理的任务划分和数据分区。

2.3 实验目的

先前，我进行了 SIMD 并行化和多线程编程的实验，提出了适合 SIMD 并行化的存储形式和算法设计，以及适合不同多线程编程库的算法设计。

在本次 MPI 实验中，我将在 MSMPI 上编程实现适合 MPI 的任务分配算法，掌握 MPI_Init 等六个主要函数的编程使用。考虑查询间和查询内两种并行方式。同时对比不同节点数下的算法性能。随后我还将实现另一种编程策略：位图存储方式的优化，并给出 MPI 并行算法的设计。此外，我还考虑结合先前所进行的多线程优化的算法和 SIMD 下的 SSE 优化算法。最后，我拟探讨不同 MPI 通信阻塞方法的差异，以及多平台之间的性能分析。总之，我将进行以下的实验对比工作并给出分析：

- Windows 平台节点数的对比
- 查询间和查询内两种优化方式的对比
- 位图存储算法和原始列表算法的并行化对比
- SIMD、多线程与 MPI 结合的优化对比
- MPI 不同通信策略的对比
- Windows 和 Linux 平台的差异对比

2.4 实验环境

Windows 平台下，硬件：CPU：Intel(R) Core(TM)i9-14900HX,24 核，基准速度 2.2GHz，内存容量 16GB，Cache L1: 2.1MB;L2: 32.0MB;L3: 36.0MB。软件：编程 IDE 为 Visual Studio，编译器内置，MPI 环境为 VS 下的 MSMPI。

Linux 平台下是课程统一配置的鲲鹏服务器。

2.5 有关数据/性能

2.5.1 数据集

本问题研究所采用的数据集是给定好的 1000 条查询及 1756 条倒排索引列表，经过初步的数据分析，这些列表包含 DocID 最多 30000 个，最大 DocID 达 25205174，具体详见 github 仓库 src 文件夹下生成的各种数据信息。

2.5.2 性能测试方法

本问题研究的性能测试主要以时间为主，单位均为秒，Windows 下时间测试方法如下（Linux 时间测试代码会在后面修改内容部分介绍）：

```
LARGE_INTEGER freq, start, end0;
QueryPerformanceFrequency(&freq);
...Initial...
QueryPerformanceCounter(&start);
...Function...
QueryPerformanceCounter(&end0);
double elapsedSeconds = static_cast<double>(end0.QuadPart - start.QuadPart) / freq.QuadPart;
```

关于 baseline，我遵循串行最优性能原则，选择按元素求交算法的时间性能，具体见 SIMD 实验报告。

2.5.3 实验正确性

由于并行优化的前提是保证结果的正确性，所以我在问题研究中的相关实验都保证结果是正确的。同时我在 github 仓库中公开了所有本问题研究过程中用到的代码项目，可下载验证结果。而正确结果的生成是由 stl 库 set_intersection 函数生成的标准结果，并保存在 github 仓库 src 文件夹下。

2.5.4 GitHub 仓库

仓库中包含实验涉及的所有程序，所有查询的信息和性能数据，以及整理出来的数据表。
https://github.com/Mhy166/parallel-programming.git

3 MPI 算法设计

3.1 平凡串行算法

列表求交主要有两种方式: 按表求交，按元素求交。在先前的实验中，我经过性能比较，发现按元素求交算法远胜于按表求交，因此考虑按元素求交的多线程化。同时我根据基础算法设计出更优秀的串行算法，设计特点如下:

- 自适应重排序。算法在每轮迭代后都将针对列表大小进行重排序，选择最短的列表作为基准列表。这样可以在算法进行中动态调整策略，选择较快速的方式。
- 空表提前停止。利用列表升序特点，算法在进行中会逐一删除元素，并且不断检查列表为空的情况，一旦发现则可以直接结束算法，提前停止有利于加速算法执行时间。
- 时间复杂度大致为 $O(ln^2)$, 其中 l 为列表数，n 为列表规模。该复杂度仅作为规模的参考，事实上在实际算法中应用的上述优势可以有效提升实际性能，在结果部分我们会做分析。

下面是本算法的伪代码:

Algorithm 1 元素求交串行算法

Input: $\boldsymbol{l}(\boldsymbol{t}_1), \boldsymbol{l}(\boldsymbol{t}_2), \ldots, \boldsymbol{l}(\boldsymbol{t}_k)$, Sorted by $|\boldsymbol{l}(\boldsymbol{t}_1)| \leq |\boldsymbol{l}(\boldsymbol{t}_2)| \leq \cdots \leq |\boldsymbol{l}(\boldsymbol{t}_k)|$

Output: $\cap_{1 \leq i \leq k} l(t_i)$

1: **function** FUNCTION($\boldsymbol{l}(\boldsymbol{t}_1), \boldsymbol{l}(\boldsymbol{t}_2), \ldots, \boldsymbol{l}(\boldsymbol{t}_k)$)

2: **Input:** 多个列表 `lists`

Output: 交集元素集合 res
while lists[0] 不是空的 **do**
  $tmp \leftarrow lists[0].front()$
  $lists[0].pop\_front()$
  $cnt \leftarrow 1$
  $flag \leftarrow$ false
  for $i \leftarrow 1$ to lists.size() $- 1$ **do**
    逐一排查其他列表，从表头开始，如果元素小于等于 tmp 则删去，直到大于 tmp
    如果有列表为空，$flag \leftarrow$ true
    if $flag$ **then**
      直接停止算法
    end if
  end for
  if $cnt == lists.size()$ **then**
    $res.insert(tmp)$
  end if
  $lists$ 重排序，始终将最短列表置于第一个列表 $list[0]$
end while
end function

3.2 MPI 优化算法——查询内

本算法旨在通过 MPI 并行化技术来加速查询操作。算法的核心思想是将第一个列表进行分割，并将其他列表保持不变，分配给子进程进行并行处理。其设计思想如下：

- 数据读取和分发：主进程（rank 0）负责读取输入数据文件 ExpIndex 和查询文件 ExpQuery，并将数据分发给其他子进程进行处理。
- 并行查询处理：每个查询由主进程分解为多个部分，分配给不同的子进程处理。每个子进程处理其分配的查询部分，并将结果返回给主进程。
- 结果收集与合并：主进程从所有子进程收集部分结果，并将其合并为最终结果。
- 时间复杂度：仍为 $O(ln^2)$, 其中 l 为列表数，n 为列表规模。

限于篇幅原因，这里只给出关键的 MPI 代码和必要的通信流程的注释如下：

```
// 主进程：查询分发
int segmentSize = lists[0].size() / (size - 1); // 计算每个子进程分配到的查询片段大小
int startIdx = 0;
for (int i = 1; i < size; i++) {
    int endIdx = (i == size - 1) ? lists[0].size() : startIdx + segmentSize;
    // 计算每个子进程的结束索引
    MPI_Send(&startIdx, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
    // 发送开始索引到子进程
```

```
        MPI_Send(&endIdx, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        // 发送结束索引到子进程
        int querySize = queryIndices.size();
        MPI_Send(&querySize, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        // 发送查询索引数组大小到子进程
        MPI_Send(queryIndices.data(), querySize, MPI_UINT32_T, i, 0, MPI_COMM_WORLD);
        // 发送查询索引数组数据到子进程
        for (uint32_t j = 0; j < lists.size(); j++) {
            int listSize = lists[j].size();
            MPI_Send(&listSize, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            // 发送当前列表大小到子进程
            vector<uint32_t> listData(lists[j].begin(), lists[j].end());
            MPI_Send(listData.data(), listSize, MPI_UINT32_T, i, 0, MPI_COMM_WORLD);
            // 发送当前列表数据到子进程
        }
        startIdx = endIdx; // 更新开始索引以供下一个子进程使用
    }
    // 主进程：结果收集合并
    set<uint32_t> resultSet;
    for (int i = 1; i < size; i++) {
        int resultSize;
        MPI_Recv(&resultSize, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // 从子进程接收结果集
        vector<uint32_t> localRes(resultSize);
        MPI_Recv(localRes.data(), resultSize, MPI_UINT32_T, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // 从子进程接收结果集数据
        resultSet.insert(localRes.begin(), localRes.end());
        // 将子进程的结果集数据合并到主进程的结果集中
    }
    // 子进程：接收分配的查询数据
    while (true) {
        int startIdx, endIdx, querySize;
        MPI_Recv(&startIdx, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // 接收开始索引
        MPI_Recv(&endIdx, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // 接收结束索引
        MPI_Recv(&querySize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // 接收查询索引数组大小
        vector<uint32_t> queryIndices(querySize);
        MPI_Recv(queryIndices.data(), querySize, MPI_UINT32_T, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // 接收查询索引数组数据
        vector<list<uint32_t>> lists(querySize);
        for (uint32_t i = 0; i < querySize; i++) {
            int listSize;
            MPI_Recv(&listSize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
        // 接收当前列表大小
        vector<uint32_t> listData(listSize);
        MPI_Recv(listData.data(), listSize, MPI_UINT32_T, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // 接收当前列表数据
        lists[i] = list<uint32_t>(listData.begin(), listData.end());
        // 将接收到的数据转换为列表
    }
    set<uint32_t> localRes;
    processQueryPart(startIdx, endIdx, queryIndices, lists, localRes);
    // 处理查询数据并得到局部结果集
    int resultSize = localRes.size();
    vector<uint32_t> localResVec(localRes.begin(), localRes.end());
    MPI_Send(&resultSize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    // 发送局部结果集大小到主进程
    MPI_Send(localResVec.data(), resultSize, MPI_UINT32_T, 0, 0,MPI_COMM_WORLD);
    // 发送局部结果集数据到主进程
}
```

3.3 MPI 优化算法——查询间

3.3.1 Windows 平台算法设计

接下来我设计了查询间的 MPI 并行化策略。该算法较于 3.2 节的查询内算法来说比较简单，通信过程不用那么复杂。查询间优化算法的设计思想如下：

- 数据读取与广播：主进程负责读取输入数据文件 ExpIndex 和查询文件 ExpQuery，并将查询任务分发给各个子进程进行处理。
- 并行处理查询：每个查询任务由主进程分配给不同的子进程。每个子进程接收到查询任务后处理其分配的查询，并将结果返回给主进程。
- 结果收集与合并：主进程从所有子进程收集处理结果，并将结果合并和保存到输出文件中。
- 时间复杂度：仍为 $O(ln^2)$, 其中 l 为列表数，n 为列表规模。

我们同样看一下关键的 MPI 代码（必要的注释已经给出）：

```
// 主进程：发送查询任务
int dest = queryID % (size - 1) + 1;
int querySize = queryIndices.size();
MPI_Send(&queryID, 1, MPI_INT, dest, 0, MPI_COMM_WORLD); // 发送查询 ID
MPI_Send(&querySize, 1, MPI_INT, dest, 0, MPI_COMM_WORLD); // 发送查询索引大小
MPI_Send(queryIndices.data(), querySize, MPI_UINT32_T, dest, 0, MPI_COMM_WORLD);
//主进程：发送结束信号
int endSignal = -1;
```

```
    for (int i = 1; i < size; i++) {
        MPI_Send(&endSignal, 1, MPI_INT, i, 0, MPI_COMM_WORLD); // 发送结束信号
    }
    //主进程：接收处理结果
    for (int i = 0; i < queryCount; i++) {
        int id;
        double elapsedSeconds;
        int resultSize;
        MPI_Recv(&id, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // 接收查询 ID
        MPI_Recv(&elapsedSeconds, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // 接收查询时间
        MPI_Recv(&resultSize, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // 接收结果集大小
        results[id] = { id, elapsedSeconds };
        resultSizes[id] = resultSize;
    }
    // 子进程：接收查询总数 + 接收处理查询任务
    MPI_Recv(&queryCount, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    while (true) {
        int queryID;
        MPI_Recv(&queryID, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // 接收查询 ID
        if (queryID == -1) break; // 检查结束信号
        int querySize;
        MPI_Recv(&querySize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // 接收查询索引大小
        vector<uint32_t> queryIndices(querySize);
        MPI_Recv(queryIndices.data(), querySize, MPI_UINT32_T, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // 接收查询索引数组
        // 处理查询……
        int resultSize = res.size();
        MPI_Send(&queryID, 1, MPI_INT, 0, 0, MPI_COMM_WORLD); // 发送查询 ID
        MPI_Send(&elapsedSeconds, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD); // 发送查询时间
        MPI_Send(&resultSize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD); // 发送结果集大小
    }
```

3.3.2 Linux 平台算法设计

值得注意的是上述 3.3.1 节的代码是在 Windows 下进行测试的，我们可以很容易经过修改计时部分以适应鲲鹏 Linux 服务器的运行，因为只修改了少量内容，故不再赘述代码（有兴趣的读者可以自行查看源代码，双平台上是基本相似的），直接介绍修改内容：

- 移除 <bits/stdc++.h> 和 <Windows.h> 头文件，改用 Linux 平台下的标准库头文件。
- 添加 <sys/time.h> 头文件，替换 Windows 特定的计时方法，使用 gettimeofday() 获取起始和

结束时间。

- 其他部分均可保留，MPI 的库在 Linux 和 Windows 上是跨平台兼容的。

具体的性能对比分析，我会在下一大节中给出！

3.4 128 位存储串行算法设计

基于 OpenMP 和 SIMD 结合的并行思路特殊性——多在 for 循环处和数据存储结构处进行并行化。我的串行程序需要有一定修改，对于先前的串行算法，查询间并行仍可沿用，查询内的任务划分则不适合循环并行的处理，因为前述的串行算法求交的代码前后依赖性很强。故考虑 128 位位向量优化，从而构造出适合循环并行和 SIMD 优化的算法逻辑，下面是新的串行算法的设计特点：

- 数据结构的使用：vector<bitset<25205248» bitmaps：存储每个数组的位图表示。这种位图数据结构的使用可以加速按位与的速度，同时大大减少空间的占用。
- 查询处理：每次读取查询文件中的一行，并将其转换为查询索引的向量 queryIndices。对 queryIndices 进行排序，确保较小的集合在前，之后均以其为计算标准，其没出现的更大位为 0，按位与仍然是 0，可以省去不必要的计算，有助于优化后续的位图与操作。
- 128 位分块：将每个位图按 128 位进行分块存储，以便后续进行循环并行操作。同时这样规划还可以为以后的 SIMD 结合研究做好框架基础。

该算法时间复杂度为 $O(ln)$, 其中 l 为列表数，n 为位图规模。我们看一下关键代码：

```
vector<bitset<25205248>>bitmaps;//数组存储格式转换为位图存储格式
for (uint32_t i = 0; i < arrays.size(); i++) {
    for (uint32_t j = 0; j < arrays[i].size(); j++) {
        bitmaps[i][arrays[i][j]] = 1;
    }
}
//按位向量长短排序，即列表最后一个元素的数值，此处考虑篇幅，不再赘述
sort(queryIndices.begin(), queryIndices.end(), compareBySize);
vector<vector<bitset<128>>> rebit(queryIndices.size());//保存副本
uint32_t id = queryIndices[0];  //最短列表 id
//128 位副本形式保存——第 1 个列表
for (uint32_t i = 0; i < arrays[id][arrays[id].size() - 1]; i += 128) {
    bitset<128> tmp;
    for (uint32_t j = 0; j < 128; j++) {
        tmp[j] = bitmaps[queryIndices[0]][j + i];
    }
    rebit[0].push_back(tmp);
}
for (uint32_t i = 1; i < queryIndices.size(); i++) {
    //其他列表同样进行存储。不再赘述
}
```

```
    //查询执行，即按位与操作，第二层循环可以在未来无依赖式并行
    for (uint32_t i = 1; i < queryIndices.size(); i++) {
        for (uint32_t j = 0; j < rebit[i].size(); j++) {
            rebit[0][j] &= rebit[i][j];
        }
    }
```

3.5 层进式优化算法

接下来按照 3.4 节的新查询策略算法，我进行了一系列的并行优化尝试，这包含 MPI,OpenMP 和 SIMD 的层进式优化！

3.5.1 MPI 优化

这里我采用查询间的优化思路，主进程和子进程的工作和 3.3 节的基本一样，这里不再赘述。而且主进程分配任务的方式和接收结果的方式也近乎一样，这里也不再赘述，有兴趣的读者可以查看源代码了解细节。我们直接看子进程部分的差别较大的 MPI 代码（必要的注释已经给出）：

```
//接收并处理查询任务
//接收查询大小，查询索引，代码不再赘述
int resultSize;
double elapsedSeconds;
processQuery(queryID, queryIndices, resultSize, elapsedSeconds); // 处理查询
//发送查询 ID，发送查询时间，发送结果集大小，代码不再赘述
//子进程的查询处理函数（关键部分）
void processQuery(int queryID, const vector<uint32_t>& queryIndices,\
int& resultSize, double& elapsedSeconds) {
// ... 构建 rebit 数据结构，同 128 位存储串行算法 ...

// ... 进行查询处理，同 128 位存储串行算法 ...
//构造结果集大小（位图中 1 的个数）
    resultSize = 0;
    for (uint32_t i = 0; i < rebit[0].size(); i++) {
        resultSize += rebit[0][i].count();
    }
}
```

3.5.2 OpenMP 优化

下面我们就会看到，3.4 节的算法是多么适用于 OpenMP 的优化！查询执行部分是两个 for 循环进行位与运算，因此我们直接用 OpenMP 进行优化即可！简单看一下 OMP 命令代码：

```
//子进程处理查询仅需要一行即可实现优化！其余部分基本一样。
for (int i = 1; i < static_cast<int>(queryIndices.size()); i++) {
ragma omp parallel for num_threads(4)
    for (int j = 0; j < static_cast<int>(rebit[i].size()); j++) {
        rebit[0][j] &= rebit[i][j];
    }
}
```

3.5.3 SSE 优化

我们仍然可以看到，3.4 节的 128 位存储设计也是多么适用于 SIMD 的优化！我们只需要把涉及到 128 位的位图结构转换为 SSE 下的 128 向量结构即可！这里简单看一下 SIMD 优化的代码：

```
//仅介绍查询处理部分的改动代码
vector<vector<__m128i>> rebit(queryIndices.size());//SSE 向量存储
for (uint32_t i = 0; i < arrays[id][arrays[id].size() - 1]; i += 128) {
    bitset<128> tmp;
    const uint32_t* bitmapPtr = reinterpret_cast<const uint32_t*>(&tmp);
    __m128i bitmap = _mm_load_si128((__m128i*)bitmapPtr);//存入 SSE 向量
    rebit[0].push_back(bitmap);
}
//其他列表处理过程一样，不再赘述
//查询循环 OMP 部分也不再赘述，直接看 SSE 优化处
rebit[0][j] = _mm_and_si128(rebit[0][j], rebit[i][j]);//SSE 向量的位与运算
```

3.6 MPI 通信机制的探究算法设计

此部分实验仅为进一步探索 MPI 通信机制而设计，因此我将通信方式改为非阻塞通信。理论上说，非阻塞通信可以让程序在等待通信完成时继续执行其他操作，提高并行效率。主要改动如下（代码就不再赘述，除了改动基本一样）：

- 将所有的 MPI_Send 和 MPI_Recv(阻塞通信函数) 替换为 MPI_Isend 和 MPI_Irecv(非阻塞通信函数)。
- 使用 MPI_Wait 或 MPI_Waitall 等待非阻塞通信完成。
- 主进程在发送任务时使用 MPI_Isend 发送任务信息，并使用 MPI_Waitall 确保所有任务信息发送完成。子进程在接收任务时使用 MPI_Irecv 接收任务信息，并使用 MPI_Wait 确保任务信息接收完成。

4 实验结果

关于数据表：所有实验结果均均保留 7 位小数，以 ms 为单位，且仅列举具有规模代表性的部分查询，并给出查询 ID，以确保数据真实性和严谨性。关于数据图，一般来说，我将以两种形式来呈现，折线图体现性能随规模变化的趋势（为方便观察，制图采用平滑处理，平滑区间为 5）；柱状图体现规模区间内平均性能的对比，这里我按照 5000 为单位分组，一定程度上减少了实验误差的影响。全部相关数据和程序均在 Github 仓库内，读者可以查阅相关程序和查询数据。

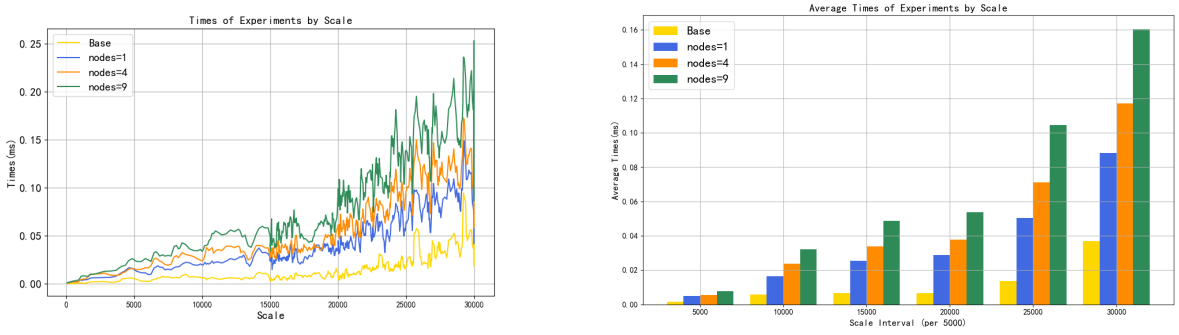
本大节将分为三部分分析结果，具体涉及实验内容将在各小节中体现。

4.1 查询内-综合结果

本小节包含 3.1 节平凡串行算法，3.2 节 MPI 查询内优化算法的实验结果分析。旨在分析节点数对于响应时间和整体性能的影响。我以列表平均长度为规模，测量了查询响应时间和全部查询耗时。由于是查询内并行，预估响应时间会有变化。下面我们看相应数据表1和图4.2.

查询 ID	列表平均长度	Base	MPI-N-1	MPI-N-4	MPI-N-9
648	17	0.0000069	0.0002181	0.0008646	0.0008162
217	595	0.0001349	0.0026581	0.0014214	0.0041624
268	1103	0.0004650	0.0033231	0.0025036	0.0052762
52	5813	0.0011444	0.0042645	0.0059894	0.0083589
504	10049	0.0028962	0.0165925	0.0194124	0.0358837
940	15003	0.0011770	0.0232088	0.0262674	0.0396166
403	20269	0.0071338	0.0409096	0.0695232	0.1195890
312	25096	0.0320208	0.0514300	0.0850174	0.0868809
39	30000	0.0248090	0.0377846	0.0552229	0.0750059
732	30000	0.0414919	0.1456320	0.2048010	0.3060330
Total	:	43.2737000	81.9943000	100.2830000	136.0940000

表 1: MPI 查询内数据表



(a) MPI 节点数折线图

(b) MPI 节点数平均柱状图

图 4.2: MPI 查询内实验图表

我根据实验性能数据作出如下解释和分析：

- 一般来说，查询内的并行会影响单个查询的响应时间，从表1的总时间也可以看出，确实影响了，但是……好像是负向影响，我经过所学，认为 MPI 的通信时间耗费造成了这一点，由于我们分割的是 L1 列表，因此每个进程接收自己任务，做部分查询，再由主进程接收，再进行结果的合

并，这一连串时间耗费其实远大于多进程并行带来的积极影响。而且我们可以看到随着节点数增加，这种通信耗费会呈正比增长，这确实验证了我前面的分析。

- 这里增加了单进程和串行的对比，结果表明，单进程也会较慢。这是因为 MPI 程序在开始时需要进行 MPI 环境的初始化，在结束时需要进行清理。这两个操作都会增加一定的开销，即使只有一个进程在运行。而且，即使是单进程，MPI 环境也会初始化通信子系统，这些通信子系统会引入额外的延迟和开销。这些开销在串行程序中是不存在的，因为串行程序不需要进行任何通信操作。此外，MPI 进程管理的开销也会影响单进程的性能，MPI 需要维护进程间的通信、同步等信息，这些信息在单进程运行时也是需要维护的。因此我们可以看到单进程会较慢，虽然我们之前可能会认为和串行差不多。

4.2 查询间-综合结果

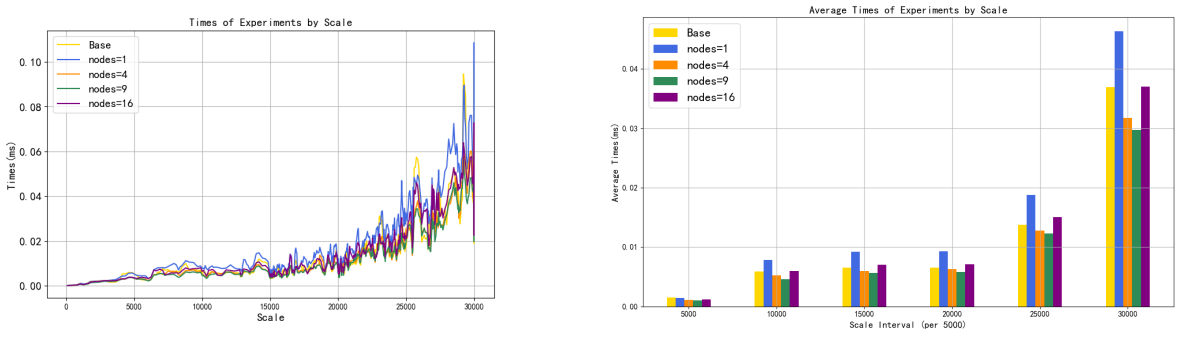
本小节包含 3.1 节平凡串行算法，3.3 节 MPI 查询间优化算法，3.6 节通信机制探究组的实验结果分析（都是查询间并行化框架下的）。旨在分析节点数对于响应时间和整体性能的影响，多平台的性能差异以及通信机制的探究差异。由于响应时间差异不会很明显，同时为了节省篇幅，本节的表只给出 5 条查询。（完整表见仓库）

4.2.1 节点规模实验组结果

下面我们先看查询间优化算法在节点数差异下的性能比较，见相应数据表2和图4.3!

查询 ID	列表平均长度	Base	MPI-J-1	MPI-J-4	MPI-J-9	MPI-J-16
648	17	0.0000069	0.0000109	0.0000072	0.0000070	0.0000139
52	5813	0.0011444	0.0012189	0.0012906	0.0011339	0.0018423
504	10049	0.0028962	0.0048006	0.0037030	0.0032294	0.0058013
403	20269	0.0071338	0.0059816	0.0037657	0.0034068	0.0059280
732	30000	0.0414919	0.0852369	0.0443478	0.0514054	0.0455788
Total	:	43.2737000	61.1511000	11.4144000	4.9437100	3.9448200

表 2: MPI 查询间数据表



(a) MPI 节点数折线图

(b) MPI 节点数平均柱状图

图 4.3: MPI 查询间实验图表

我根据实验性能数据作出如下解释和分析：

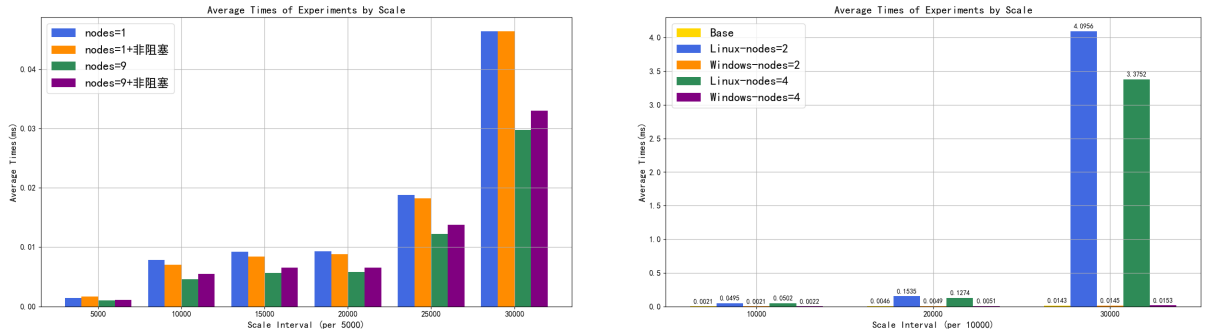
- 查询间的并行有效加速了我们的算法。从表2的 Total 可以看出，4 节点并行的加速比可以达到 3.77，9 线程加速比则 8.74。可以看到这并不是完全的 4 和 9，甚至 16 进程加速比为 10.96，差距还比较大，这应该是由于 MPI 通信的传输会造成开销，还会有额外的延迟，MPI 的进程管理的开销也会影响算法的性能，因为 MPI 需要维护进程间的通信、同步等信息。在大量进程的时候，MPI 的通信环境也会随之变大，通信和管理进程开销也会因此增多。因此导致了上述的加速比结果。但是我们也该注意到 9 线程是比较好的配置，因为其加速比并没有像 16 进程恶化的那么明显。

- 这里增加了单进程和串行的对比，结果表明，单进程性能会略微慢，这应该是因为使用了 MPI 环境，就会有环境创建，环境维护，通信管理等等开销。这些在串行程序中是没有的，因此单进程 Total 会比 Base 慢。

- 此外我给出了响应时间的图表数据，以供分析。我们居然发现查询间的并行居然优化了单条查询的响应时间，这很不可思议，因此值得我好好分析：在 MPI 多进程化的环境下，可以更好地利用多核 CPU 和系统资源，串行可能无法充分利用多核 CPU 的能力；当多个进程并行执行时，可能会使得数据更频繁地留在缓存中，从而减少了内存访问时间；在多进程环境中，一个进程在执行 I/O 操作时，其他进程可以继续执行计算任务，从而实现 I/O 重叠，提高了效率。然而进程数太大，对响应时间也会有反向影响，这可能是因为系统资源不再够分，需要进行限制，从而每个进程内的利用率反而下降。

4.2.2 非阻塞通信 +Linux 平台实验组结果

需要说明的是，这是两组不同的实验，不是叠加性质的。下面我给出非阻塞通信探究的性能差异和 Linux 平台，Windows 双平台下的性能差异图4.4：



(a) 非阻塞通信平均柱状图 (b) 双平台对比平均柱状图

图 4.4: 非阻塞通信 + 双平台实验图表

我对此实验结果，有如下探究分析：

- 针对非阻塞通信实验组：我们不难看出单进程非阻塞有一定优化，而多进程非阻塞性能则有所下降。这是因为单进程下，非阻塞通信使得进程能够更高效地利用 CPU 和其他资源，因为它不必等待通信完成，可以在通信过程中进行其他计算任务。而在多进程下，非阻塞通信需要主进程和子进程间进行更多的同步和协调，以确保数据的一致性和正确性，具体表现为 3.6 节的算法提到了 MPI_Wait 等一些等待函数，这种额外的同步和协调开销导致了性能的下降。

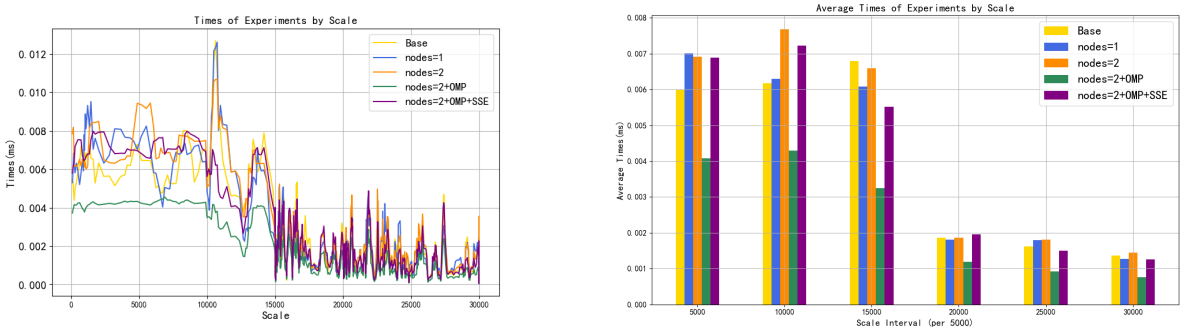
- 针对 Linux 平台对比结果：我们不难看出 Linux 平台处理相对 Windows 慢很多，这可能是因为鲲鹏服务器负载资源分配的一些问题所致。但是可以看到随着节点数增加（这里为了服务器的安全，我最多测到 4 个节点），响应时间仍然会变少，这和我们先前 4.2.1 节分析的结果是一样的！Linux 下的 MPI 多进程化的环境也可以更好地利用多核 CPU 和系统资源，其缓存也能在多进程执行时，使得数据更频繁地留在缓存中，从而减少了内存访问时间。这意味着了 MPI 的环境特点是普遍性的！

4.3 层进式优化-综合结果

本小节包含 3.4 节 128 位存储串行算法，3.5 节层进式优化结合算法的实验结果分析。旨在分析新存储策略下利用所学不断并行优化的性能差异，涉及到多进程，多线程，向量化的层进优化实验组。数据表3仅展示五个查询性能数据，直观性能对比见图4.5。

查询 ID	列表平均长度	Bit-Base	Nodes-1	Nodes-2	OMP-2	OMP+SSE-2
648	17	0.0038940	0.0026905	0.0045630	0.0028437	0.0044521
52	5813	0.0054555	0.0078504	0.0059615	0.0039087	0.0060906
504	10049	0.0094966	0.0065514	0.0062851	0.0043813	0.0079959
403	20269	0.0004349	0.0001540	0.0002321	0.0000958	0.0001441
732	30000	0.0008387	0.0009860	0.0015021	0.0004437	0.0008570
Total	:	632.7260000	609.9810000	366.5040000	265.1520000	364.3640000

表 3: MPI 层进式数据表



(a) 层进式优化折线图

(b) 层进式优化平均柱状图

图 4.5: MPI 层进式实验图表

针对此实验结果，我做出如下解释和分析：

- 可以看到，单条查询的响应时间和总体性能随着 OpenMP 的加入得到了非常有效的改善！结合所学和前面的实验理解，我认为 OpenMP 通过多线程并行执行，提高了 CPU 的利用率，使得查询能够在多个核上并行处理，从而减少了执行时间；OpenMP 还提供了高效的线程同步机制，能够减少线程间的等待时间，从而提高了并行执行的效率。

- 但是 SSE 加入后性能出现了下降，这也值得我思考分析，我认为，SSE 通过向量化处理来提高计算效率，但在某些情况下，向量化处理的开销可能超过其带来的性能提升。特别是当数据访问模式不适合向量化时，性能可能下降；而且 SSE 要求数据对齐以实现高效的向量化操作。如果数据未对齐或需要频繁访问非连续内存区域，SSE 的性能优势可能被抵消，甚至带来额外的开销。

- 同时我们应当注意到这种新存储算法下，虽然说总时间考虑了 128 位分割等存储操作而很久，但是查询部分的性能是很快的，同时这种新存储算法对 OpenMP 和 SSE 的兼容性也很好，未来期末研究我还可以尝试 AVX256 位优化，再看 SIMD 的结合效果!
- 注意到这种算法的性能变化和规模不再成正比，这是因为位存储的查询过程涉及到很多因素，包含列表个数，也包含列表最大位 DocID，未来的期末研究也可以结合这些因素进一步考虑规模的更准确定义。但是双进程的性能仍然有优化，加速比仍然是存在的，为 1.72，这意味着这种算法也是较好的适用于多进程化的，而 4.1 节的结果其实意味着那种任务划分策略不适用多进程化，需要考虑更多，从而设计出行之有效的多进程算法!

5 总结

本次实验是倒排索引问题的 MPI 优化实验，也作为期末研究的一个子实验，主要有以下收获：

- 实现了两种优化方式：查询内和查询间。增强了我的 MPI 风格下的程序撰写能力。
- 实现了 Windows 和 Linux 双平台下的查询间实验结果的对比，并以结果的趋势相一致验证了 MPI 的一些共性特征，如更好利用系统资源和缓存。
- 进行存储格式的转换，实现适于 OpenMP+SIMD 的位向量查询算法，并设计了层进式结合的算法系列，最终对其层进式的性能差异进行了分析。
- 重点探究了 MPI 阻塞和非阻塞通信方式的差异，并给出了性能的差异及分析。
- 探究了进程（节点）数所带来的管理调度的开销影响，特别是设置了单进程组，重点验证了与串行算法的性能差异，证实了 MPI 的通信环境等一系列开销的存在。

总之，这次实验帮我打好了期末研究的基础，比如 AVX 的结合以及更高效 MPI 算法的设计。让我掌握了 MPI 通信库的细节实现，理解了通信机制的差异，增强了我的多进程并行编程能力。