南開大學
Nankai University

计算机学院

并行程序设计实验报告

倒排索引求交 MPI 实验

姓名：唐明昊 朱世豪

学号：2113927 2113713

专业：计算机科学与技术

2023 年 6 月 4 日

目录

1 选题介绍

1.1 选题问题描述

倒排索引（inverted index），又名反向索引、置入文档等，多使用在全文搜索下，是一种通过映射来表示某个单词在一个文档或者一组文档中的存储位置的索引方法。在各种文档检索系统中，它是最常用的数据结构之一。

对于一个有 U 个网页或文档（Document）的数据集，若想将其整理成一个可索引的数据集，则可以为数据集中的每篇文档选取一个文档编号（DocID），使其范围在 [1,U] 中。其中的每一篇文档，都可以看做是一组词（Term）的序列。则对于文档中出现的任意一个词，都会有一个对应的文档序列集合，该集合通常按文档编号升序排列为一个升序列表，即称为倒排列表（Posting List）。所有词项的倒排列表组合起来就构成了整个数据集的倒排索引。

倒排列表求交（List Intersection）也称表求交或者集合求交，当用户提交了一个 k 个词的查询，查询词分别是 t_1，t_2，...，t_k，表求交算法返回 $\cap_{1\leq i\leq k} l(t_i)$。

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：

$$|l(t_1)| \leq |l(t_2)| \leq \ldots \leq |l(t_k)|$$

例如查询 “2014 NBA Final”，搜索引擎首先在索引中找到 “2014”，“NBA”，“Final” 对应的倒排列表，并按照列表长度进行排序：

$$l(2014) = (13,\ 16,\ 17,\ 40,\ 50)$$

$$l(NBA) = (4,\ 8,\ 11, 13, 14,\ 16, 17, 39, 40, 42, 50)$$

$$l(Final) = (1,\ 2,\ 3,\ 5,\ 9,\ 10,\ 13,\ 16,\ 18,\ 20,\ 40,\ 50)$$

求交操作返回三个倒排列表的公共元素，即：

$$l(2014) \cap l(NBA) \cap l(Final) = (13,\ 16,\ 40,\ 50)$$

链表求交有两种方式：按表求交（SVS 算法）和按元素求交（ADP 算法）。另外，我们还找到拉链法、跳表法、Bitmap 位图法、Hash 优化法、递归法等方法，针对 SIMD 编程的情景，我们选择了 Bitmap 位图法和 Hash 优化法作为研究的重点，同时综合两种基础算法：SVS 和 ADP 进行并行化研究。

1.2 数据集描述

给定的数据集是一个截取自 GOV2 数据集的子集，格式如下：

1) ExpIndex 是二进制倒排索引文件，所有数据均为四字节无符号整数（小端）。格式为：[数组 1] 长度，[数组 1]，[数组 2] 长度，[数组 2]....

2) ExpQuery 是文本文件，文件内每一行为一条查询记录；行中的每个数字对应索引文件的数组下标（term 编号）。

通过对数据集分析得到，ExpIndex 中每个倒排链表平均长度为 19899.4，数据最大值为 25205174，这个数据在**构建 hash 表和 BitMap 位图**时，对参数的确定至关重要。查询数据集 ExpQuery 一共 1000 条查询，单次查询最多输入 5 个列表，可以据此设计测试函数。

另外，在给定的数据集以外，我们还自行设置了小数据集（见 GitHub 仓库）：每个倒排链表的长度在 50 至 100，生成 1000 个倒排链表，总计 300 次查询，每次查询 5 个单词。该数据集用于验证算法的正确性以及测试不同算法在小规模输入下的性能表现。

1.3 实验环境

本实验对 ARM，x86 两种架构均做了实验测试，均采用 g++ 编译器。

- ARM 测试在鲲鹏平台运行，配置如下：

  aarch64 架构, L1 cache 64kB, L2 cache 512kB, L3 cache 49152kB

- x86 测试在本地电脑运行，配置如下：

  AMD Ryzen 5 5600H with Radeon Graphics, RAM 16.0 GB 22H2

  L1 cache 384kB, L2 cache 3.0MB, L3 cache 16.0MB

1.4 实验分工

本小组由唐明昊和朱世豪组成。对于 MPI 的 Query 内实现，唐明昊同学负责 SVS 算法和 Hash 算法的并行化实现，朱世豪同学负责 ADP 算法和 BitMap 算法的并行化实现。MPI 的 Query 间算法实现主要由唐明昊同学完成。ARM 平台的实现主要由朱世豪同学完成。另外，针对通信机制的优化，唐明昊提出减少通信，非阻塞通信等方法，朱世豪提出精确负载，组通信等方法。附唐明昊和朱世豪二人 GitHub 仓库链接。

1.5 实验概述

本次 MPI 实验分别从 MPI 的 Query 间并行化与 MPI 的 Query 内并行化两个角度入手。

Query 间即是在最外层循环进行循环划分，每个进程独自处理一部分 Query。由于 Query 与 Query 间并没有很强的依赖性，所以**较为适合并行化**，并行化效果好。

Query 内是对一个 Query 求交过程中进行并行化。普通算法 SVS 与 ADP 在是实现过程中用到了拉链法的思想，具有**提前结束**的特性，而分段并行处理后，各个进程对于倒排链表的前半部分元素做的求交工作无法像串行算法一样为后面的元素服务，从而会导致负载不均，即处理后面部分的进程工作量较大，导致最终算法执行效率低下。为了解决此问题，我们进一步提出了四种优化的思路：

- 算法改进：对于通信时的算法进行改进，以更好的适应于 MPI 的通信机制。
- 精确负载：对于任务分配时的机制进行改进，以更好的平衡各个进程的负载。
- 集合通信：对于各进程求交结果汇总时的通信使用 MPI 中的 gather 函数进行优化，以减少空闲等待进程的个数。
- 非阻塞通信：对于阻塞等待时间进行优化，使用非阻塞通信机制。

另外，为更全面的评价各个 MPI 算法的优化效果，我们还进行了

- MPI 在 x86 与 ARM 平台的对比研究
- 将 MPI 结合 OMP，SIMD 等并行化方法的测试
- 进程数不同时 MPI 算法效率的研究

最后我们探究了本次实验中所遇到的问题，并对其发生的原因进行了解释。

2 Query 间并行

2.1 并行实现

Query 间并行即是在最外层循环，**粗粒度**地为每个进程划分工作任务，分配一定量的查询任务，每个进程调用相应的倒排索引求交算法。进程间依赖较少，并行化效果较好。

Query 间 MPI 并行

```
MPI_Init(&argc, &argv);
int rank, size;// 初始化
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf("-----process%d start-----\n", rank);
//------求交------
long long head, tail, freq;
if (rank == 0) {// 一个进程输出
    QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
    QueryPerformanceCounter((LARGE_INTEGER*)&head);// Start Time
}
int length = floor(count / size);// 每个进程处理的length
int start = length * rank;
int end = rank!=size-1?length*(rank+1):count;// 考虑末尾
//-----根据任务范围求交-----
Some Codes Here...
MPI_Barrier(MPI_COMM_WORLD);// 进程同步
printf("-----process%d finished-----\n", rank);
```

此外，Query 间 MPI 并行很适合结合 OMP 与 SIMD 等并行化手段，MPI 进行粗粒度并行，后者细粒度并行，做到全过程并行化，这将在章节5详细介绍。

2.2 实验结果

四个算法的加速效果如图2.1所示。

注意，此时由于 Bitmap 算法与 Hash 求交算法开辟空间较大，电脑在四个进程下不得不调整相应的参数，放弃一部分空间换时间的效果，使得算法能够正常运行，为此此时在与串行算法比较时，也进行控制变量，将相应参数调整至与原来相同。**这一点使得串行算法的运行速度比上实验中慢一些**。

由图可知在使用了 Query 间的 MPI 并行化加速后，每个算法都取得了大约 2 至 3 倍的加速比，Query 间并行由于其良好的性质，任务划分相对均匀，故算法相对执行高效。

Query 间并行加速并未达到理想的三倍以上，其原因可能是：各个进程请求的任务个数相同，而 Query 内的工作量的大小并不相同。例如一个 Query 的倒排链表个数可能不同，根据数据集的特点，我们所用的数据集中的单个 Query 内部倒排链表数在 1 至 5 个之间不等，这可能会导致任务分配不均匀。对于 Query 间的 MPI 算法，我们无法提前预知每个 Query 内的任务量，因此很难对这一点进行优化。而在 Query 内的 MPI 算法，由于我们可以**利用二分查找算法确定自己的工作范围**，因此更容易达到均衡，对这点我们将在3.2.1节中进行讨论优化。

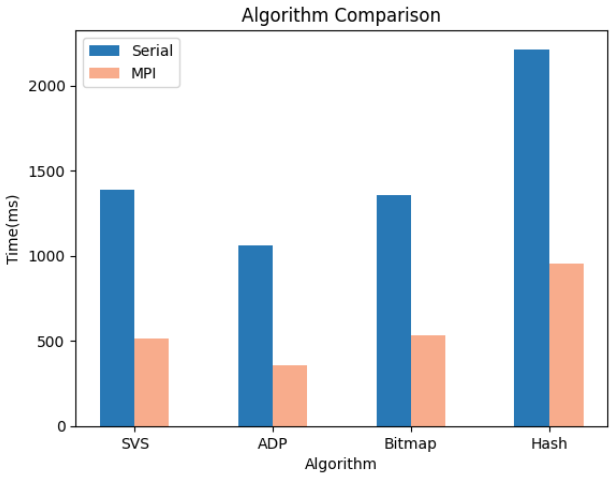

图 2.1: Query 间串行与 MPI 并行对比

2.3 x86 与 ARM 对比

四个算法在两平台上表现效果如下。

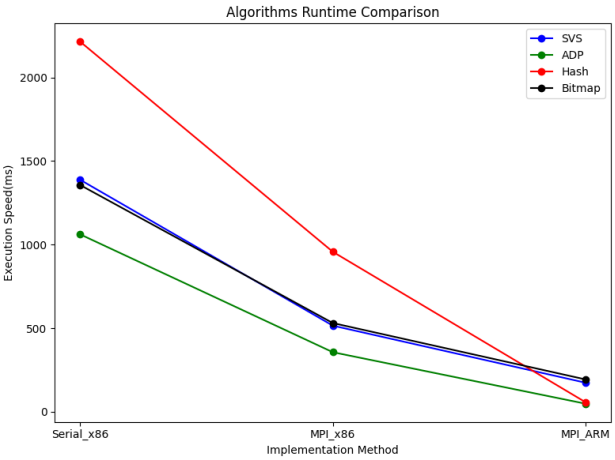

图 2.2: Query 间优化 x86 与 ARM 对比

可以从图中发现，在使用了 Query 间的 MPI 并行化加速后，鲲鹏服务器上的 MPI 算法都比 x86 上要更快，速度大约为 x86 上的 1/10。实际上这是由于 ARM 上串行算法的速度远快于 x86 上的串行算法导致的，**实际上两者的加速比大致相同**。详情见6.2节对于该异常问题的分析。

3 Query 内并行

3.1 SVS 算法

SVS 算法先使用两个表进行求交，得到中间结果再和第三个表求交，依次类推直到求交结束。这样求交的好处是，每一轮求交之后的结果都将变少，因此在接下来的求交中，计算量也将更少。

3.1.1 初步尝试

由于 SVS 算法的思想是每次求交后结果变少，加速后续求交。而现在由于 Query 内 MPI 并行是分布式内存，如果每个进程与一个链表完成一次求交就通信以减少元素会大幅降低算法效率；并且在实现上不能像共享内存那样，把链表中符合的元素全部移动到链表前端，最后做一次删除，这在 MPI 实现起来显然是过于复杂的。

于是调整思路，每个进程在函数开始阶段即划分好工作范围，不再进行修改；在求交过程中也不再删除链表中元素，而是将对应位置为极大值。经实测，这并不会过多影响算法效率。

在求交结束以后，非 0 进程遍历自己的工作范围，找出求交成功的元素，放进 int 数组里，而后实验 MPI_Send 发送给 0 号进程，0 号进程收集汇总求交结果。

在发送数据时，为了避免缓冲区超限，采取两次发送，第一次发送求交成功元素的个数，而后再发送具体的数据。这个通信机制导致 0 号进程 MPI_Recv 时不得不按顺序，分 tag 的接收，否则可能出现长度与数据不匹配等问题。对通信机制的优化尝试将在章节4介绍。

SVS_MPI 通信部分

```
if (rank == 0) {// 用0号进程做收集
    // 把该部分求交有效的拿出来->为了通信
    for (int i = begin; i < end; i++)
        if (s.docIdList[i] != maxIndex)
            result.docIdList.push_back(s.docIdList[i]);
    // 接收其他进程的数据
    for (int i = 1; i < size; i++) {
        int count;// 接收长度
        MPI_Recv(&count, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        unsigned int* temp = new unsigned int[count];// 接收数据
        MPI_Recv(temp, count, MPI_UNSIGNED, i, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // 放进结果向量
        for (int j = 0; j < count; j++)
            result.docIdList.push_back(temp[j]);
        delete []temp;
    }
}
else {// 非0号线程则把vector发送过去
    unsigned int* temp = new unsigned[end-begin];// 转成int数组发送
    int count = 0;
    for (int i = begin; i < end; i++)
        ...
    MPI_Send(&count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);// 先发一个长度过去
    MPI_Send(temp, count, MPI_UNSIGNED, 0, 1, MPI_COMM_WORLD);// 再把数据发过去
    delete []temp;
}
```

3.1.2 算法改进

为了减少通信阶段构造数组的复杂度；优化求交过程，保留 SVS 算法求交中减少长度的算法优势，对算法进行改进。在开始阶段，每个进程取出连续的 1/4 到倒排链表 s 中，求交过程按照原始算法进

行（同样可以加速大概率事件）。求交结束，非 0 进程将 s 剩余元素发送。虽然同样需要将元素拿出 s 构造数组，但由于保留了 SVS 算法的优势，最终求交成功的元素连续存放。于是在复制元素时，可以使用 **STL 库中的 copy 函数**，效率比 for 循环逐个遍历高很多。

SVS_MPI 改进

```
// 开始阶段取最短的列表
int initialLength = index[QueryList[0]].length;
InvertedIndex s;// 取工作范围1/4
int start = (initialLength / size) * rank,
    end = rank != size - 1 ? (initialLength / size) * (rank + 1) : initialLength;
s.docIdList.assign(index[QueryList[0]].docIdList.begin() + start,
     index[QueryList[0]].docIdList.begin() + end);
...
// 发送阶段使用copy函数加速复制
copy(s.docIdList.begin(), s.docIdList.end(), temp);
MPI_Send(&s.length, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);// 先发一个长度过去
MPI_Send(temp, s.length, MPI_UNSIGNED, 0, 1, MPI_COMM_WORLD);// 再把数据发过去
```

3.1.3 精确负载

在 OpenMP 实验中也提到过，SVS 算法为了提高表现，引入了拉链法，即链表前面的元素会帮助后面的元素移动求交链表中的指针，算法会**提前结束**。而在并行化分段以后，位于后面段的元素失去了前面段的信息，需要从头往后寻找匹配。又因为链表都是升序排列的，故在本链表中位于后半部分的元素，在另一链表中大概率也会位于后面。这将导致部分线程遍历链表的长度很大，也即工作量不均等。

之前采用的是**加速大概率事件**进行优化，确实起到了一定的效果，但并未完全消除其影响。朱世豪同学在优化 ADP 算法时参考 **Jacobi 迭代**提出了**精确负载**的思想，以最小化负载不均。详细介绍见3.2.1。

在 SVS 算法中，同样可以该思想使用来优化算法表现。由表1可以看出，在优化前，各个进程负载明显不均。进程 1 和进程 2 在发送后直接结束运行，而进程 3 由于负载不均明显较慢，进程 0 则需要阻塞等待其余线程。

	SVS_MPI2	SVS_Precise
进程 0	1115.96	833.69
进程 1	642.89	827.70
进程 2	953.69	833.59
进程 3	1115.89	827.83

表 1: 算法执行时间对比

在优化后，各个进程负载明显均衡，算法总运行时间得到有效提升。另外从表1还可以看出，**进程间通信导致的时间差异并不大**，在平衡负载后，各个进程运行时间差仅在个位数内。

3.2 ADP 算法

ADP 算法在各个列表中寻找当前文档，当在所有列表中寻找完某个文档之后，查看每条列表的剩余的未扫描文档数量，只要其中一条列表走到尽头，则本次求交结束。

由上面的分析可知，ADP 算法使用 MPI 进行优化时每个进程所求得的结果仍然是多个 docID，需要进行汇总。因此，优化方式与 MPI 实现方式大致与3.1.1节中 SVS 算法的实现方式大致相同。

3.2.1 精确负载

在之前的 pthread 实验中，已经发现了各个线程之间负载不均的问题，也提出了相应的方法进行优化，但是优化效果并不理想，并没有从根本上解决这一问题，导致算法运行效率低下。于是在本次实验中，尝试加入更好的机制，使得各个进程之间能够达到负载均衡。

从倒排索引求交的本质入手，其根本任务是找到多个链表中相同的 docId，而当用于参考的首链表被划分给 4 个进程，每个进程在剩余倒排链表中依旧只能整体扫描一遍，而没有在对应能够得到有效结果的区域内进行扫描，这就会导致时间上的浪费。于是可以在正式开做求交算法之前先把 1 到 num-1 倒排链表遍历一遍，划分出来每个线程要做的部分。随后每个进程在这些部分中扫描求交，这样会使得负载均衡的程度大大提高，因为每个进程不再做“无用功”，也就是在自己不可能扫描到相同 docId 的位置进行求交比对。

对于 1 到 num-1 号倒排链表划分操作的具体实现上，可以使用二分查找的方法，根据每个进程第一个 docId 来进行二分查找，找出 1 到 num-1 号倒排链表中第一个大于等于该 docId 的位置。该位置即为该进程的求交开始位置。而对于求交的结束位置，对于第 i 号进程，由于它的**求交结束位置恰好为第 i+1 号进程的求交开始位置**，由此可以使用进程间的通信将该索引进行发送，从而节省第 i 号进程再次使用二分索引查找确定求交结束点的时间。这也是运用了 Jacobi 迭代中相邻进程之间进行通信传输所需数据的方法，来对倒排索引问题进行的一个特异性的优化。

精确负载机制

```
//每个进程的第一个倒排链表的开始位置和结束位置
...some code here...
//找到剩余倒排链表的起始位置和结束位置
for (int i = 1; i < num; i++)
   //二分查找，找到自己的开始位置start[i-1]
   ...some code here...
if (rank != proNum - 1){//最后一个进程不需要知道自己何时结束，因为它直接扫描到最后
   //从下一个进程接受该进程的起始位置
   MPI_Recv(end, num - 1, MPI_INT, rank + 1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   for (int i = 1; i < num; i++)
      list[i].end = end[i - 1];//下一个进程的开始正是这个进程的结束位置
}
else{
   for (int i = 1; i < num; i++)//最后一个进程做到底
      list[i].end = index[list[i].key].docIdList.size();
}
if (rank != 0) //第一个进程不用发送信息，其余发送给前一个进程
   MPI_Send(start, num - 1, MPI_INT, rank - 1, 1, MPI_COMM_WORLD);
```

从表2可以看出，在加入精确负载机制后，各个进程之间的负载不均问题明显得到改善，但是算法的运行时间上总体却总体没有没有得到显著的提升，这可能是由于算法引入二分查找和进程之间的通信所带来的额外时间开销抵消了部分负载均衡所带来的收益。

	ADP_MPI2	ADP_MPI_Precis
进程 0	952.68	855.10
进程 1	952.63	855.07
进程 2	869.13	849.90
进程 3	912.14	849.88

表 2: 算法执行时间对比

3.3 Hash 算法

Hash 算法采取**用空间换时间**的思想，优化了 SVS 算法：为每个表构造 hash 表，将该表的 DocId 依次映射到对应的 hash 桶里，同一桶内元素按升序排列。在查找比较时，可以直接定位 hash 桶，再从 hash 桶找回比较的位置，很少的步骤即可查找到，不需要从头进行遍历。

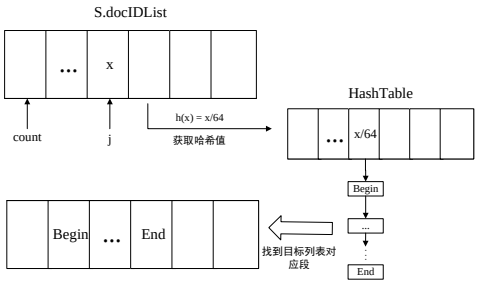

图 3.3: Hash 算法示意图

Hash 算法由于其良好的特性，不像 SVS 算法与 ADP 算法需要花较大精力平衡负载，每个进程都只需要映射 DodID 即可，段与段之间没有依赖关系。参照 SVS 算法划分任务后，每个进程使用 Hash 算法求交，最后使用长度与数据分开发送的通信机制进行数据汇总，具体的实现代码不再赘述。

3.4 实验结果

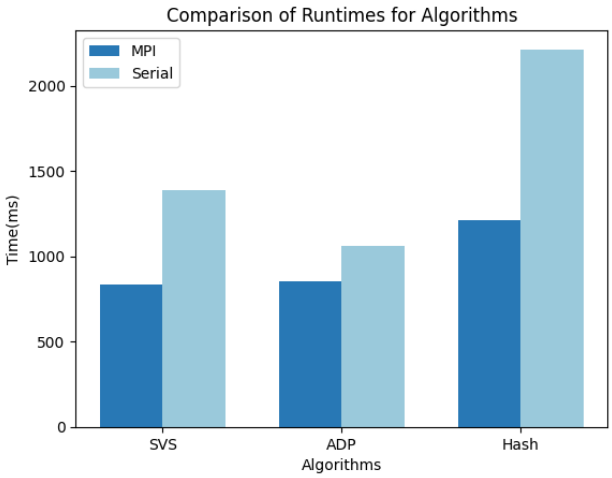

图 3.4: Query 内串行与 MPI 并行对比

如图3.4，使用本节中 Query 内 MPI 优化效果最好的精确负载算法与其相应的串行算法相比较。

而 Hash 由于多进程原因，串行同并行算法同使用较小空间进行测试。

可见对于每种算法而言，使用 MPI 都取得了一定的优化效果，其中 Hash 算法的优化效果最好，其并行化时间大约为串行算法的 1/2，这是由于 Hash 算法使用的哈希桶策略使得各个进程之间的依赖程度较小，从而运行效率较高。而对于 SVS 和 ADP 算法，虽然使用了精确负载的策略，但是因此又带了使用该策略的额外开销，故相对于 Hash 算法，两者的优化就并不显著了。

3.5 x86 与 ARM 对比

图3.5是 x86 平台与 ARM 平台 Query 内并行效果对比。

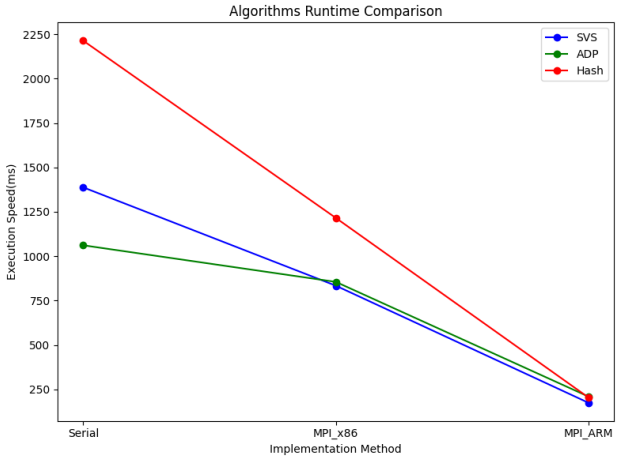

图 3.5: Query 内串行与 MPI 并行对比

可见就 Query 内的 MPI 优化来说，ARM 平台的执行速度明显快于 x86 平台，与 Query 间的并行化大致相同，这也是由于 ARM 平台串行算法速度过快导致的，具体的原因将在6.2节中进行分析，此处将不再展开赘述。

4 通信机制尝试

4.1 减少通信

之前采取的通信机制需要 0 进程使用 for 循环依次接收其余线程发送的消息，存在一种可能，某个进程已经做完工作，而 rank 比它小的进程未做完工作，则该进程需要一直阻塞等待；另外，之前的通信需要先发送长度，再发送数据，分两次进行发送，同样可能造成时间上的浪费。

考虑使用 **MPI_ANY_SOURECE**，0 进程每次循环任意接收数据；再将长度放到数据数组的 0 号位置。以上，减少通信等待与通信次数，探索通信对算法执行效率的影响。

减少通信

```
if (rank == 0) {// 用0号进程做收集
    ...
    for (int i = 1; i < size; i++) {// 接受数据
        MPI_Recv(temp, initialLength, MPI_UNSIGNED, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        int len = temp[0];// 拿出数据长度
        s.docIdList.insert(s.docIdList.end(), temp+1, temp +1+len);
```

```
    }
}
else {// 非0号线程则把vector发送过去
    ...
    temp[0] = s.length;
    copy(s.docIdList.begin(), s.docIdList.end(), temp+1);
    // 直接一起发过去
    MPI_Send(temp, s.length+1, MPI_UNSIGNED, 0, 1, MPI_COMM_WORLD);
}
```

但该方法还需要额外的同步机制维护算法的正确性。存在可能：某个非 0 进程 a 在发送完数据后进入下一个函数步内，再次发送数据给 0 进程，而某个进程 b 仍然在上一个函数步内未做完工作，于是进程 a 抢先进程 b 发送消息，这将导致算法执行错误。

于是，需要在发送消息前增加 **MPI_Barrier** 同步各个进程，保证各进程发送消息时都在同一函数步内。

	SVS_divide	SVS_packed
进程 0	905.04	1064.41
进程 1	724.57	1064.36
进程 2	487.41	1064.34
进程 3	905.03	1064.38

表 3: 执行时间对比

由表3中实验结果可知，该通信机制优化最终并没有提升算法的执行效率，可能的原因如下：

- 两次通信对算法影响并没有那么大。
- 由于前述提到的负载不均的问题，rank 大的进程相对来说工作量更大，导致 rank 较大进程长时间等待 rank 比它小的进程出现频率较小。
- 即使使用了 MPI_ANY_Source 让进程可以到下一函数步内提前做一些工作，但 barrier 同步机制也在一定程度上让进程阻塞等待。

4.2 非阻塞通信

非阻塞通信是在前述减少通信基础上的进一步尝试。为了进一步提高进程利用率，减少阻塞等待时间，使用非阻塞通信，非 0 进程在 Isend 数据后即进入到下一个函数步去，减少两个进程拥挤等待的时间。

非阻塞通信

```
if (rank == 0) {// 用0号进程做收集
    ...
    for (int i = 1; i < size; i++) {// 接受数据
        MPI_Irecv(temp, initialLength, MPI_UNSIGNED, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
            request);
        MPI_Wait(request, MPI_STATUS_IGNORE);// 阻塞下来，保证接收到才往后处理
        int len = temp[0];
        s.docIdList.insert(s.docIdList.end(), temp+1, temp +1+len);
```

```
    }
}
else {// 非0号线程则把vector发送过去
    ...
    // 直接一起发过去
    MPI_Isend(temp, s.length+1, MPI_UNSIGNED, 0, 1,MPI_COMM_WORLD,request);// 再把数据发过去
    ...
}
```

由下表可知，非阻塞通信有效的提升了减少通信的效率，基本与两次通信保持了一致。无法避免的 barrier 同步造成的时间消耗，与该通信机制对特殊情况阻塞优化获得的时间收益实现了对冲。

	Hash_divide	Hash_packed	Hash_isend
进程 0	1768.13	2066.91	1801.26
进程 1	1355.30	2045.47	1797.47
进程 2	1795.39	2053.40	1781.06
进程 3	973.91	2042.44	1798.89

表 4: 执行时间对比（hash 算法参数取 512）

4.3 集合通信

在进行 MPI 并行化的过程中，发现了各个进程求交后结果进程汇总的过程是一个线性的过程，也即 1、2、3 号进程均将自己的求交结果发送给 0 号进程。这使得进行接受的操作的 0 号进程必须等待所有进程发送完毕后才能离开，进入下一个函数步，形成了木桶效应，0 号进程成为了通信中的一块“短板”。

为此，我们使用 MPI 中的 Gather 函数进行优化，也即将汇总求交结果的过程变为 3 号进程传递给 1 号进程同时 2 号进程传递给 0 号进程，随后 1 号进程将自己的求交结果同 3 号进程的结果传递给 0 号进程。这样便将原来 0 号进程进行等待的时间进行均摊给 1 号进程，从而实现了更好的通信机制。

集合通信

```
vector<int> recvCounts(size); // 存储每个进程发送的数据数量
int totalCount = 0; // 所有进程发送的总数据数量
vector<int> displacements(size, 0); // 接收缓冲区中每个进程数据的位移量
int count = s.length; // 根据实际情况获取本进程的数据数量

// 收集每个进程的数据数量到根进程
MPI_Gather(&count, 1, MPI_INT, recvCounts.data(), 1, MPI_INT, 0, MPI_COMM_WORLD);

// 计算总数据数量
if (rank == 0) {
    // 计算位移量 displacements
    ...some codes here...
}
vector<unsigned int> receivedData(totalCount); // 存储接收到的所有数据

// 发送本进程的数据给进程0
```

```
MPI_Gatherv(s.docIdList.data(), count, MPI_UNSIGNED, receivedData.data(), recvCounts.data(),
    displacements.data(), MPI_UNSIGNED, 0, MPI_COMM_WORLD);
```

在表5中，列出了使用通信优化机制的 SVS 算法 SVS_Commun 以及使用通信优化机制及精准负载机制的算法 SVS_PreCom 同他们未使用 gather 通信机制时进行比较，可以看出使用通信机制后有两个进程比其他进程快出很多，这可能是由于使用 gather 后进程能够更快的进入下一个函数步的缘故，然而这一特点导致了负载不均衡，从而在最后一行所展示的各进程运行算法所需的最长时间上，使用集合通信这一机制的算法（SVS_Commun、SVS_PreCom）的表现反而不如他们各自的平凡算法（SVS_MPI2、SVS_Precise）。

	SVS_MPI2	SVS_Commun	SVS_Precise	SVS_PreCom
进程 0	642.89	929.34	827.70	871.28
进程 1	953.69	1175.59	833.59	871.32
进程 2	1115.89	445.59	827.83	243.53
进程 3	1115.96	1175.56	833.69	220.14
Max	1115.96	1175.60	833.69	871.32

表 5: 集合通信

5 多种并行方式结合

5.1 Query 间 MPI+Query 间 OMP

在 Query 间 MPI 的基础上，配合使用 Query 间 OpenMP。每个进程获得一定量的查询任务，进程再将查询任务细分给 NUM_THREADS 个线程，最大化加速算法。

MPI+OMP

```
int provided;// 多线程
MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
if (provided < MPI_THREAD_FUNNELED)// 提供的等级不够
    MPI_Abort(MPI_COMM_WORLD, 1);
int rank, size;// 初始化
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf("-----process%d start-----\n", rank);
...
#pragma omp parallel for num_threads(NUM_THREADS)
for (int i = start; i < end; i++) {// count个查询
    ...
}
MPI_Barrier(MPI_COMM_WORLD);// 进程同步
```

测试结果如下图所示。

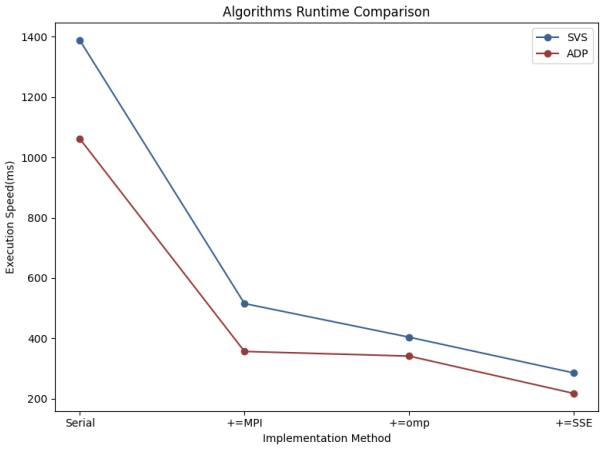

图 5.6: Query 间 OMP

可以看出，在增加了 OpenMP 多线程任务细化以后，算法的执行效率得到了进一步的提升。相较于单独 MPI 并行，增加了 OpenMP 以后，两个算法分别取得了 21.6% 和 4.3% 的性能提高。

5.2 Query 间 MPI+Query 内 OMP

为了探索 MPI 与多线程的结合，我们继续结合上一次实验的 Query 内 OpenMP。外层循环已经有了一次任务分配，我们对内层求交循环进行划分。具体来说，每个进程负责一部分查询，查询求交过程中进程内每个线程负责倒排索引链表的一部分，求交时进行共享内存的通信。

由于 SVS 和 ADP 提前结束的算法特性，段与段之间有着**隐式依赖**，query 内任务划分很容易造成负载不均，这在3.1.1中已经提到过。MPI 的 query 内并行可以使用精确负载的方法来提高表现。而 OpenMP 共享内存并没有使用该方法，采用了退而求其次的加速大概率优化，但效果如图5.7依然较差。

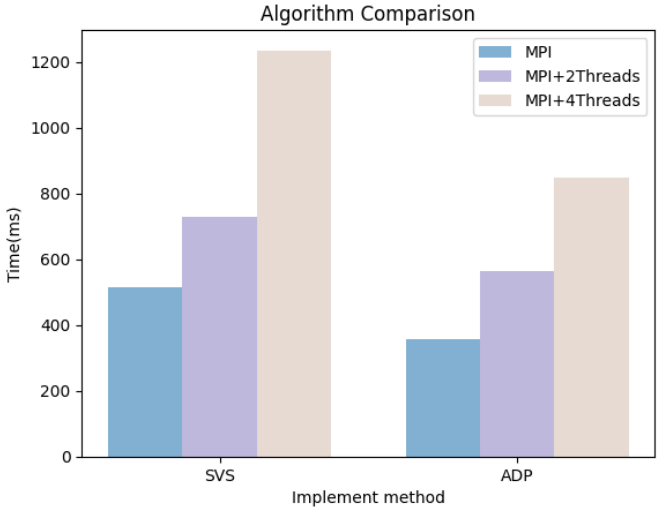

图 5.7: Query 内 OMP

5.3 MPI+OMP+SIMD

由前两节的探索可知，MPI 结合 Query 间 OpenMP 效果是更好的，所以在此基础上，我们再次叠加 SIMD。

外层循环进行了细致的任务划分，在内层求交时用 _mm_cmpeq_epi32 指令进行四位比较，再使用 _mm_movemask_epi8 指令生成掩码，加速求交。（四个元素至多只能有一个元素与 DocID 相匹配，当发生匹配时，对应位置置 1，则最终生成的掩码不为 0，一个判断条件即可完成。）

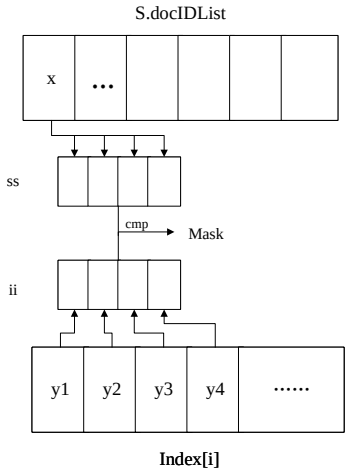

图 5.8: SIMD 示意图

实验结果已在前图5.6给出。在增加了 SSE 以后，比普通 MPI 并行性能分别提高了 44.6% 和 38.9%，相较于串行算法加速比达到 4.86 和 4.88。

6 问题探究

6.1 进程数影响

实验过程中为了探索 MPI 机制，测试了不同进程数的算法表现及加速比。

从表6中的加速比的计算结果可以看出，进程数增加，算法的加速比增大。从伸缩性角度来看，除 BitMap 算法以外，效率基本相当，算法中一些串行部分以及进程间的通信开销只让其略微下降，算法的**伸缩性**较好。

BitMap 算法在四进程的加速比提升相较于其他算法最差，这是由于 Bitmap 算法本身需要大量空间的特性导致的，事实上，如果多加入一个进程，就要多额外分配大约大约 3153920 字节的空间给新加入的进程，这一开销是 4 进程所难以接受的。在运行 Bitmap 算法的过程中，通过软件监控，**电脑的内存使用率几乎保持在 99%**，并且也多次因为这一空间实在过大而导致程序崩溃，无法正常运行。

可想而知，这种空间开销已经影响到了 MPI 程序的运行，巨大的内存开销所导致的 cache miss 已经超过了并行化带来的收益，由此 BitMap 算法在 4 进程时的运行效率反而更低。

	SVS	ADP	Hash-256	BitMap
2 进程加速比	1.29	1.58	1.38	1.26
效率	0.64	0.79	0.69	0.63
4 进程加速比	2.69	2.97	2.53	1.34
效率	0.67	0.74	0.63	0.33

表 6: 不同进程加速比

以上数据均是基于 x86 平台的数据，串行数据是**未使用 MPI** 正常运行得到的。但在实验过程中我们发现一个问题，鲲鹏平台并行加速异常的快，加速比已经超过进程数，这是超乎我们预料的。

6.2 加速比异常

根据以上提到的问题，我们对其进行了探索。我们发现，在鲲鹏服务器上，只要使用 MPI 程序进行编译，即使没有对进程进行任务划分，即每个进程串行地都执行完整的 1000 个 Query，**算法执行时间都是有明显下降的**。然而如果是在自己的机器（x86）上实行同样的操作，程序的运行结果反而比串行所消耗的时间（1387.76ms）更多。详细数据见表7。

	SVS_ARM	SVS_MPI_ARM	SVS_x86	SVS_MPI_x86
进程 0	151.08	32.056	1948.52	468.47
进程 1	154.39	36.719	1991.06	508.33
进程 2	157.55	37.098	2047.03	515.29
进程 3	158.03	40.072	2012.91	495.19
加速比	-	3.94	-	3.97

表 7: 串行执行（使用 MPI）与 MPI 并行执行

如果**以开了 MPI 编译选项的串行数据做为串行执行时间**，那么 arm 平台下，MPI 并行化的加速比回归正常，x86 的加速比也变大，二者几乎一致。

但最关键的问题是，为什么开启 MPI 编译选项后，在鲲鹏服务器上串行算法的执行效率会得到如此大的提升呢？我们怀疑是**不同架构下编译器及 CPU 对于 MPI 程序的优化方式**不尽相同，这也就导致了最终得到的程序效率不同。

在 x86 架构下，编译器和 CPU 通常更加注重向量化指令和多线程并行的优化。但在个人 PC 上，这一策略可能由于**资源限制**表现的不明显。

相比之下，在 ARM 架构下，编译器和 CPU 更加注重能效的优化。ARM 架构的设计目标是高能效，因此编译器可能会针对 MPI 程序进行代码优化，以减少功耗和内存访问次数。CPU 可能会采用一些功耗管理策略，如动态频率调节和核心睡眠等，以降低功耗和热量产生。从而保证更有效的内存访问与指令执行。

由于 x86 架构和 ARM 架构在体系结构和设计目标上存在差异，编译器和 CPU 在优化 MPI 程序方面的策略也会有所不同，导致最终的程序效率在两种架构上有明显的差异。在我们查阅到的资料中有提到 **“具体的差异倍数取决于具体的应用和程序，但通常情况下，这种差异可能会达到几倍甚至更多”**。这解释了在本实验的倒排索引求交算法中，两个不同平台上的求交效率差异巨大。

7 实验总结

在本次 MPI 实验中，我们使用 MPI 对 query 间和 query 内两个方面进行并行化。从实验结果来看，两种并行化均取得了一定的加速效果。此外，不局限于一种通信模式，我们尝试了 MPI 丰富的通信机制，包括非阻塞通信，集合通信等，对算法确有一定改进，过程中也学习了很多知识。最后，结合之前实验的 OpenMP 与 SIMD 并行手段，最大化了性能表现。

通过本次实验，我们熟练掌握了 MPI 的运用，熟悉了 MPI 编程范式，通信机制，多线程等各种操作。深刻体悟到了分布式内存的特点，以及多进程对于算法性能的提升。在往后的学习过程中，我们会更多地探索 MPI 的使用，挖掘其更多价值。