武汉大学 2022 —2023 学年 第一学期
《高性能地理计算》答题纸(论文）

学号：2020302131043 **姓名：吴斌文**

院系：遥感信息工程学院 **专业：遥感科学与技术**

总分	一	二	三	四	五					

基于 MPI 的 Eratosthenes 筛法的并行化设计

吴斌文

（武汉大学遥感信息工程学院　湖北　武汉　430000）

摘要：Eratosthenes 筛法，简称埃氏筛，是一种由古希腊数学家埃拉托斯特尼提出的一种简单检定素数的算法，虽然 Eratosthenes 筛法效率高，但是在传统串行化编程实现过程中，算法本身做了许多无用功，一个数会被筛选好几次，同时串行环境下算法执行速度也受限。因此，本文对传统的 Eratosthenes 算法进行分析，明确优化方向，并基于信息传递接口（message passing interface，MPI）实现算法并行并优化。结果表明，实验并行化设计相对于传统串行算法在执行效率上提升显著。

关键词：MPI 并行计算；Eratosthenes 筛法；算法优化

Parallelized design of the MPI-based Eratosthenes sieve method

WU Bingwen

(*School of Remote Sensing Information Engineering, Wuhan University　Hubei　Wuhan　430000)*

Abstract: Eratosthenes sieve method, referred to as Eratosthenes sieve, is an algorithm proposed by the ancient Greek mathematician Eratosthenes to simply check the prime numbers. Although Eratosthenes sieve method is efficient, the algorithm itself does a lot of useless work during the traditional serialized programming implementation, and a number will be sieved several times, and the execution speed of the algorithm is also limited in the serial environment. Therefore, this paper analyzes the traditional Eratosthenes algorithm, clarifies the optimization direction, and implements the algorithm in parallel and optimizes it based on message passing interface (MPI). The results show that the experimental parallelization design improves the execution efficiency significantly compared with the traditional serial algorithm.

Keywords: MPI parallel computing; Eratosthenes sieve method; algorithm optimization

0　引言

随着多核计算机的发展，串行执行程序的缺点暴露无遗，传统的 Eratosthenes 算法是串行算法，搜索过程易懂，程序设计简单，但大量内存空间与计算时间的耗费成为此算法的瓶颈，而并行计算的出现，为现有的许多的串行算法都提供了优化的方向。针对上述问题，本文提出了一种基于 MPI 并行计算的 Eratosthenes 筛法优化设计，利用 MPI 多线程之间的并行数据传输计算，在满足数据传输和运行时间的前提下，实现大数据量的 Eratosthenes 筛法实验。

1 MPI 简介

MPI（Message Passing Interface）为并发程序中一组进程指定通讯[1]，是一种消息传递标准库，其消息传递的并行化策略已经成为并行编程的一种高校且容易理解的典范。MPI 是一种跨语言的通讯协议，用于编写并行计算，其特点是高性能、大规模和可移植性，是一种基于信息传递的并行编程技术。

MPI 函数库中的 MPI_SEND 和 MPI_RECV 函数可以实现阻塞模式和非阻塞模式下的数据传输，同时可以用于实现数据的并行传输与计算。通过将整体的运算分为若干并行模块，将计算拆分为若干并行模块，最后将结果汇总，其基本框架如图 1 所示。

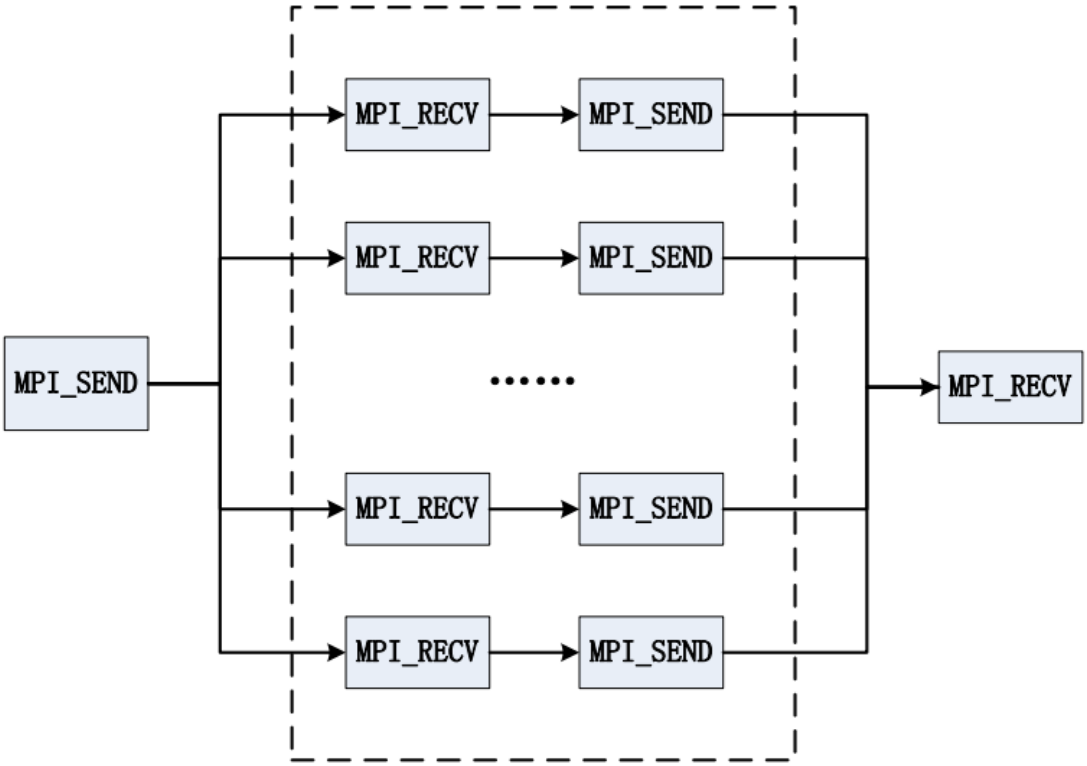

图 1. 基于 MPI 并行计算的数据传输框架

通过 MPI_SEND 函数实现数据并行传输，计算线程则通过 MPI_RECV 接收数据后开展并行计算，其运行的线程数可以根据实际需要调整，计算完成后将数据通过 MPI_RECV 进行汇总，图中虚线框内的流程是并行计算的核心，同时可以根据需要将并行计算进行扩展，其优势则是可以依靠当前 LINUX 系统支持的高性能服务器或者高性能集群，结合 MPI 并行计算，开展大规模的数据运算。[2]

2 基于 MPI 的 Eratosthenes 算法并行设计

2.1 Eratosthenes 算法简介

埃拉托斯特尼是一位古希腊数学家，他在寻找整数 N 以内的素数时，采用了一种与众不同的方法：先将 2~N 的各个数写在纸上，在 2 的上面画一个圆圈，然后划去 2 的其他倍数；第一个既未画圈又没有被划去的数是 3，将它画圈，再划去 3 的其他倍数；现在既未画圈又没有被划去的第一个数是 5，将它画圈，并划去 5 的其他倍数……依此类推，一直到所有小于或等于 N 的各数都画了圈或划去为止。这时，画了圈的以及未划去的那些数正好就是小于 N 的素数。

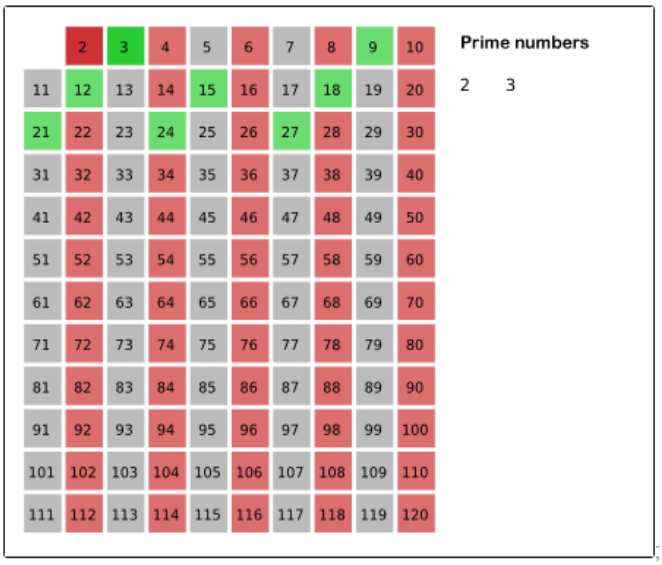

图 2. Eratosthenes 算法示意图

2.2 并行算法设计

2.2.1 Eratosthenes 算法步骤

①创建一个自然数 2，3，4，…，n 的列表，其中所有自然数都能被标记为 0；
②令 k=2，它是列表中第一个未被标记的自然数；
③重复下面步骤，直到$k^2 > n$为止；
（a）找出k^2和 n 之间的是 k 的倍数的数并标记为 1；
（b）找到比 k 大的未被标记的数中最小的那个，令 k 等于这个数；
④列表中未被标记的数就是质数；

2.2.2 MPI 并行设计

下面使用 MPI 并行计算加快这一算法，将数组分为 p 个连续的块，每个块的大小基本相等，为了平衡负载，要给每个进程分配 $[\frac{n}{p}]$ 个元素，我们考虑使用下面的实现方法：

①、首先计算$r = n \bmod p$,前 r 个进程分配$[\frac{n}{p}]$个元素，后面 p-r 个进程分配$[\frac{n}{p}]$个元素。

②、进程 i 控制的第一个元素是$i\left[\frac{i}{p}\right] + \min(i, r)$,最后一个元素是$(i+1)\left[\frac{n}{p}\right] + \min(i+1, r) - 1$,对特

定元素 j，控制它的进程是$\min\left(\left[\frac{j}{\left[\frac{n}{p}\right]+1}\right],\left[\frac{j-r}{\left[\frac{n}{p}\right]}\right]\right)$。

③、0 进程分到的数据块的大小$proc0_{size}=\left[\frac{n-1}{p}\right]$，我们用 0 进程来存储步骤③（b）中用于筛选的 k 值(即 2 到$\sqrt{n}$的质数)，所以程序运行的前提要求是$2+proc0_{size}\geq(int)sqrt((double)n)$

④、对每个进程都提供一个 marked[size]这样的数组，prime 保存当前用于筛选的质数，first 表示进程 id 中第一个要求被筛掉的数对应的 marked 数组的下标。index 用于步骤③（b）中找到的比 prime 大的未被标记的数中最小的那个数，index 为 0 进程专属。

2.2.3 时间复杂度

χ：执行二元操作所需要的时间。

λ：经由通道将一个整数传到另一个通道所需要的时间。

素数定理：$\pi(x)\sim\frac{x}{lnx}$，其中$\pi(x)$表示不超过 x 的素数的个数。

分析：2 到$\sqrt{n}$内，找到一个质数需要一个广播，所以通信的预期时间为$\left(\frac{\sqrt{n}}{\ln\sqrt{n}}\right)\lambda\log_2 p$，对 n 个质数，p 个进程，需要筛选$\frac{n}{p}$次，一共需要$\sum_p\frac{n}{p}$，又因为素数的倒数和为$\sum_p\frac{1}{p}\sim O(\ln\ln n)$，所以 p 个进程的筛选的预期时间粗略估计为$\chi(n\ln\ln n)/p$,总预期时间为$\sum_p\frac{1}{p}\sim O(\ln\ln n)+\left(\frac{\sqrt{n}}{\ln\sqrt{n}}\right)\lambda\log_2 p$。

3 算法优化

3.1 去偶数优化

利用已知除 2 以外的所有偶数都不是素数的常识，可以将待筛选数字总量减半，从而提高筛选效率。关键在于数组减半，找到新的索引映射，以及首个倍数（非素数）的位置。这样筛选的预期时间约为$\chi(n\ln\ln n)/(2p)$。

3.2 去广播优化

原理：初始的代码是通过进程 0 广播下一个筛选倍数的素数。进程之间需要通过 MPI_Bcast 函数进行通信。通信就一定会有开销，特别是在分布式计算机架构上，因此我们让每个进程都各自找出它们的前 sqrt(n) 个数中的素数，再通过这些素数筛选剩下的素数，这样一来进程之间就不需要每个循环广播素数了，只需要最后通信归总结果，这样只需要$\lambda[\log_2 p]$的通信时间。与去偶数优化结合，则一共只需要$\frac{\chi(n\ln\ln n)}{2p}+\frac{\chi(n\ln\ln n)}{p}+\lambda[\log_2 p]$。

3.3 分块筛选，提高 Cache 命中率

对于第三个有两个优化思路，第一个是基于 cache_linesize 的优化，另外一个是基于 cache_size 的优化。对于 windows，在任务管理器的性能中能看到相关信息，也可以通过软件 CPU-

Z 来查看。

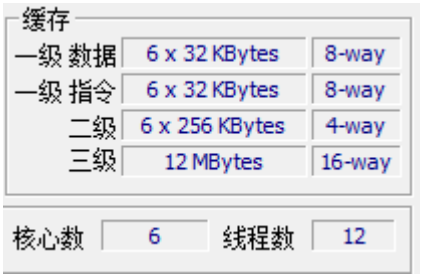

图 3. 实验环境 Cache 配置

对于大规模的数组来说 cache_line 的优化效果并不是很明显，所以下面只针对 cache 做优化，本优化方法需每个进程都各计算出前 sqrt(n) 个数中的素数，(对应去广播优化)。

由于从 cache 读取的速度远高于从内存中处理，所以 cache_size 优化的思路在于每次处理 cache 大小的数组，之前我们已经将 n 内分成大小约为 n/p 的块给每个进程处理，然后再在每个进程中将 n/p 大小块按照 cache_size 进行分块，在此之前我们需要对 cache 的大小从 byte 转化为 int，64 位系统即除以 8，以上图 windows 为例，L1，L2，L3 缓存分别可以存 48KB，0.1875MB，1.5MB 个 int，而单机（注意是单机，如果是分布式计算机理论上可以占满所有的 cache）中每个进程又将划分其中的 cache，如果对于 L3 而言如果分配 2 个进程则每个进程能够得到 0.75 MB 个 int 进程处理，而实际中由于计算机中有其他进程也会使用 cache，所以在实际中这个数还要小。另外根据测试的时候 n 大小进行选择 cache 的级别，比如我测试的是亿级的数据，远超过 cache 的大小，所以直接对 L3 级别的 cache 进行分块，当然选择 L3 并不一定是最优策略，需要多次实证才能知道。关键在于进程内分块。

4 软硬件环境与实现

4.1 硬件环境

实验硬件环境为个人电脑（Windows）的单机多核环境，计算机配置如下：

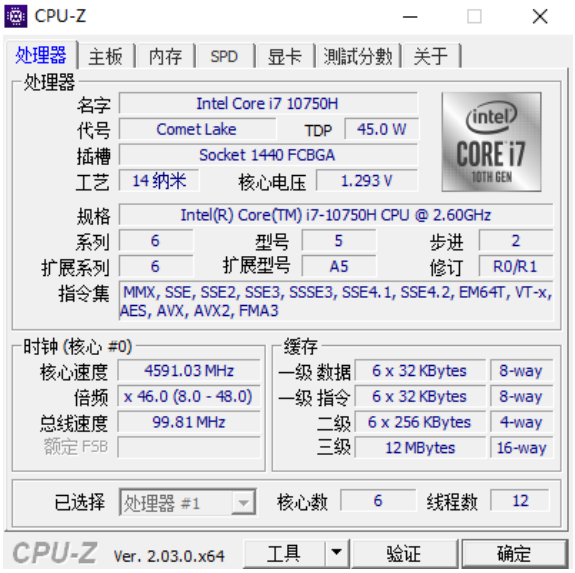

图 4. 硬件环境

4.2 软件环境

开发环境：Visual Studio 2019，MSMPI v10.0，MPI 环境配置（本次实验为 Windows 环境）：windows 下运行 mpi 首推微软的 msmp，因为比较简单，下载地址为：https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi，将两个安装包 msmpisdk.msi 和 msmpisetup.exe 分别下载然后安装完成后即可，下面是在 VS2019 中引入 MSMPI 的步骤：

①在 VS 中新建 C++控制台应用程序，将项目编译改为 X64

②去安装的 SDK 目录，找到 include 与 lib 文件夹右键项目 -- 属性 -- vc++ 目录中包含目录添加 include 文件夹路径，库目录中添加 lib 文件夹路径。

③C/C++ ->预处理器 -> 预处理器定义 -> 添加 MPICH_SKIP_MPICXX

④C/C++ -> 代码生成 -> 运行库 ->选择：多线程调试(/MTD)

⑤属性 -- 链接器 -- 输入 -- 附加依赖项中添加 msmpi.lib;

5 结果分析与讨论

5.1 实验组一（1,0000,0000）

5.1.1 运行时间

实验组一为在 windows 环境下，在规模 1,0000,0000 下从 4 到 32 个进程分别进行 5 次测试取运行时间平均值，得出的结果如下图：

Table 1 .实验组一运行时间汇总

进程数	MPI	优化 1	优化 2	优化 3
4	0.622856	0.276238	0.271789	0.105658
8	0.566249	0.220299	0.202315	0.062081
12	0.566094	0.232320	0.217845	0.056504
16	0.519578	0.179844	0.122979	0.062379
20	0.475152	0.135575	0.100967	0.054397
24	0.397196	0.103553	0.053613	0.044237
28	0.377199	0.087921	0.045886	0.046401
32	0.329006	0.077203	0.044334	0.051442

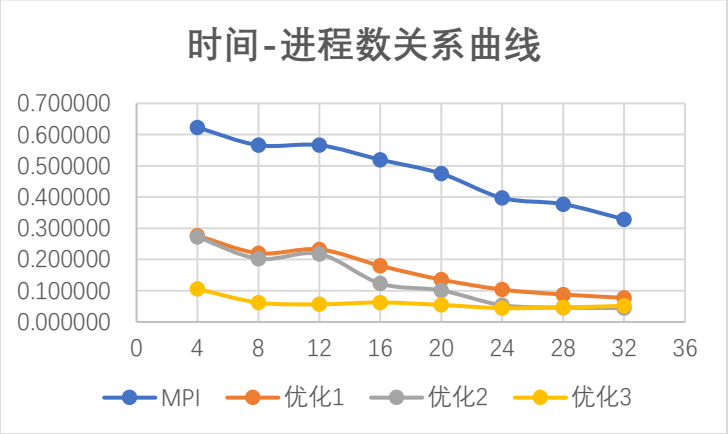

图 5. 实验组一运行时间

横向对比可以看出通过优化一去掉偶数可以使待处理数组减半，故而时间减少一半左右；通过优化二去掉通信在单机上进程数量较少的时候优化效果不明显，随着进程数量增加，优化效果有所提升；通过优化三增加 cache 命中率可以显著提升优化效果。

5.1.2 加速比

Table 2 .实验组一加速比汇总

进程数	MPI	优化 1	优化 2	优化 3
4	3.776151	8.514397	8.653772	22.260459
8	4.153648	10.676399	11.625459	37.886110
12	4.154787	10.123976	10.796648	41.625372
16	4.526752	13.078030	19.125217	37.705118
20	4.949999	17.348280	23.294694	43.237838
24	5.921516	22.712919	43.869793	53.168403
28	6.235442	26.751349	51.257911	50.688126
32	7.148814	30.465295	53.051834	45.721395

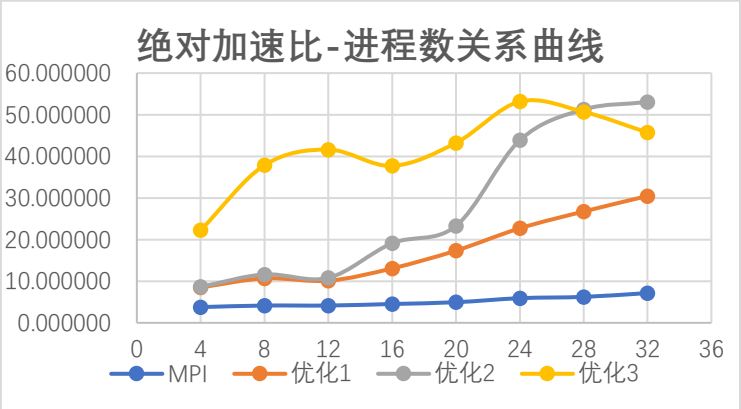

图 6. 实验组一加速比

1）可以发现通过优化 1 去除偶数，在进程数较少时，加速比表现为原始版本的两倍左右，在进程数大于 12 时，加速比继续稳定增长。

2）而通过优化 2 去除广播通信在进程数较少时优化效果不明显，但在进程较多的时候优化效果明显。

3）通过 Cache 优化对加速比提升明显，但在进程数不断增加时，加速比增长不够稳定，甚至出现了进程数增加，加速比下降的情况。

4）当 $S_p=p$ 时，S_p 便可以称为“线性加速比”，即当某一并行算法的加速比为理想加速比时，若将处理器数量加倍，执行速度也会加倍，但是在规模 1,0000,0000 的情况下除了 Cache 优化和去广播优化，其他两种优化并没有表现这种趋势，初步分析认为可能是 i/o 占据了程序的一部分。

5）对于超线性加速比的情况，通过维基百科得知，超线性加速比有几种可能的成因，如现代计算机的存储层次不同所带来的“高速缓存效应”；具体来说，较之顺序计算，在并行计算中，不仅参与计算的处理器数量更多，不同处理器的高速缓存也集合使用。而有鉴于此，集合的缓存便足以提供计算所需的存储量，算法执行时便不必使用速度较慢的内存，因而存储器读些时间便能大幅降低，这便对实际计算产生了额外的加速效果。

5.1.3 并行效率

Table 3 .实验组一并行效率汇总

进程数	MPI	优化 1	优化 2	优化 3
4	0.944038	2.128599	2.163443	5.565115
8	0.519206	1.334550	1.453182	4.735764
12	0.346232	0.843665	0.899721	3.468781
16	0.282922	0.817377	1.195326	2.356570
20	0.247500	0.867414	1.164735	2.161892
24	0.246730	0.946372	1.827908	2.215350
28	0.222694	0.955405	1.830640	1.810290
32	0.223400	0.952040	1.657870	1.428794

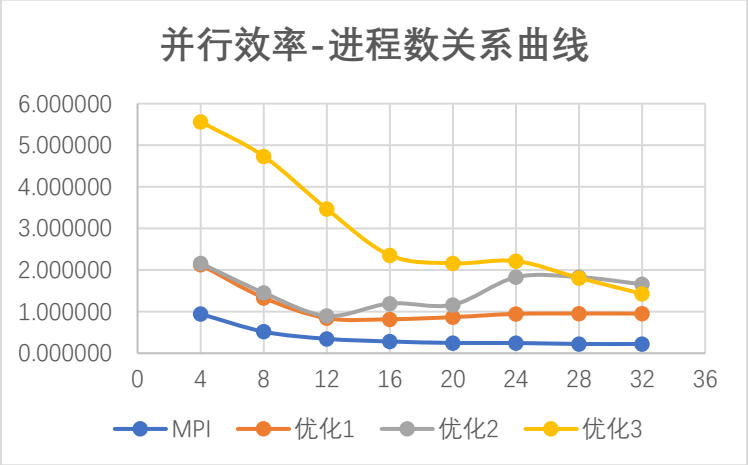

图 7. 实验组一并行效率

可以看到在没有使用 Cache 优化的时候，并行效率一般介于 0-2 之间，通过 Cache 优化在进程数较少时，并行效率提升明显，但随着进程数的增加，并行效率呈现急速下降并趋于平缓。

5.2 实验组二（10,0000,0000）

5.2.1 运行时间

Table 4.实验组二运行时间汇总

进程数	MPI	优化 1	优化 2	优化 3
4	7.396894	3.705501	3.678613	1.108405
8	6.956320	3.452338	3.443471	0.658759
12	6.913996	3.413446	3.419979	0.537246
16	6.879428	3.364131	3.374694	0.607570
20	6.783076	3.326852	3.285012	0.591393
24	6.773436	3.264225	3.280095	0.628525
28	6.743607	3.293699	3.190327	0.648143
32	6.713557	3.250400	3.188245	0.657364

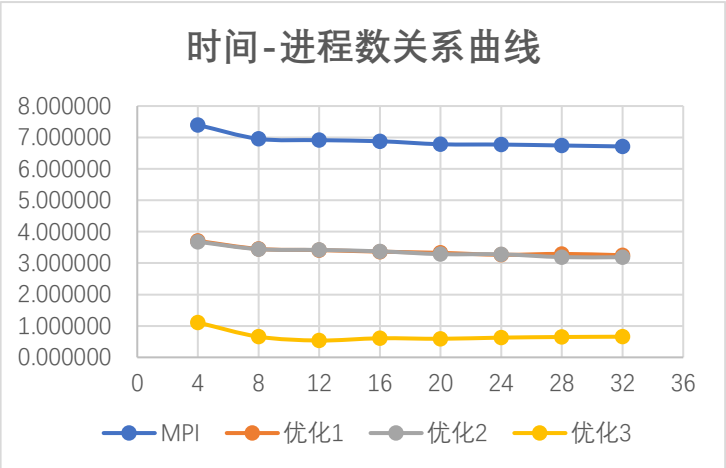

图 8. 实验组二运行时间

除优化 2 以外，其他优化方案均有明显的提升效果，但随着进程数的增加，各优化方案提升都不明显。

5.2.2 加速比

Table 5. 实验组二加速比汇总

进程数	MPI	优化 1	优化 2	优化 3
4	3.378580	6.744296	6.793593	22.546809
8	3.592560	7.238862	7.257503	37.936484
12	3.614552	7.321341	7.307354	46.516883
16	3.632715	7.428663	7.405412	41.132695
20	3.684317	7.511906	7.607583	42.257827
24	3.689560	7.656029	7.618987	39.761359
28	3.705880	7.587519	7.833367	38.557825
32	3.722468	7.688592	7.838482	38.017012

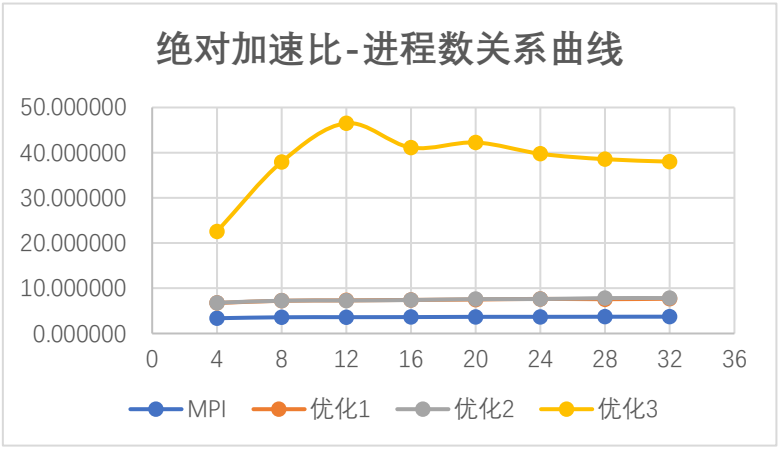

图 9. 实验组二加速比

在 10,0000,0000 数量级下，去除偶数优化对原始 MPI 小有提升，而去广播优化则几乎没有明显效果，前三种优化方案在进程数增加时，加速比增长都不明显。对比下，Cache 优化在前 12 核

加速比按照处理器核数线性增加，而后续则出现波动，并不在呈现线性趋势，因为处理器有限不能同时处理多个任务，这与之前查看 windows 系统 CPU 配置的 6 核 12 处理器相符。

5.2.3 并行效率

Table 6.实验组二并行效率汇总

进程数	MPI	优化 1	优化 2	优化 3
4	0.844645	1.686074	1.698398	5.636702
8	0.449070	0.904858	0.907188	4.742060
12	0.301213	0.610112	0.608946	3.876407
16	0.227045	0.464291	0.462838	2.570793
20	0.184216	0.375595	0.380379	2.112891
24	0.153732	0.319001	0.317458	1.656723
28	0.132353	0.270983	0.279763	1.377065
32	0.116327	0.240268	0.244953	1.188032

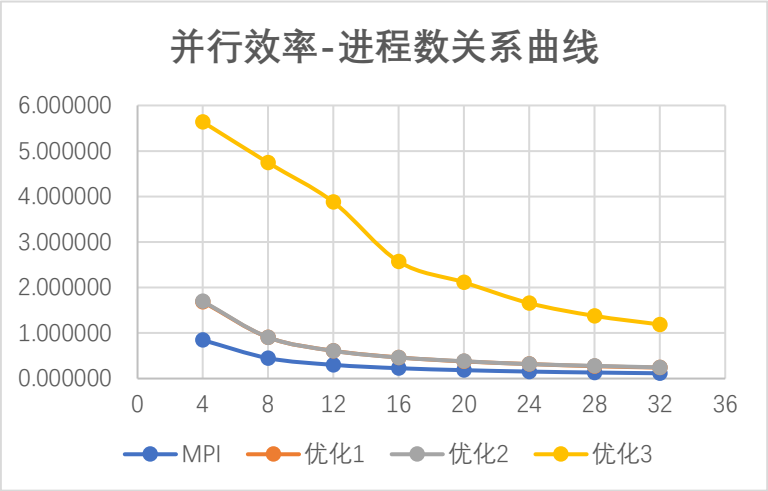

图 10. 实验组二并行效率

可以看到在没有使用 Cache 优化的时候，并行效率一般介于 0-2 之间，通过 Cache 优化在进程数较少时，并行效率提升明显，但随着进程数的增加，并行效率呈现下降趋势。

5.3 实验组三（1,000,000）

5.3.1 运行时间

Table 7. 实验组三运行时间汇总

进程数	MPI	优化 1	优化 2	优化 3
4	0.028340	0.012395	0.010752	0.010517
8	0.016458	0.007405	0.007090	0.006311
12	0.013770	0.007191	0.007968	0.007559
16	0.013583	0.006359	0.006231	0.006132
20	0.012564	0.006206	0.005640	0.005078
24	0.012364	0.007460	0.006857	0.005727
28	0.013863	0.008131	0.008076	0.007419
32	0.013006	0.009184	0.006090	0.007724

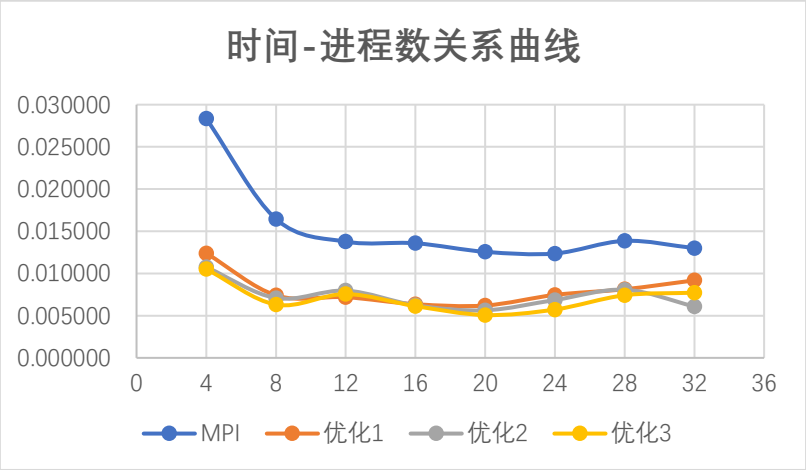

图 11. 实验组三运行时间

可以看到在规模较少的时候优化 1 和 2 对原始 MPI 的提升效果都不明显，同时 Cache 优化的效果也不及数量级大的情况。

5.3.2 加速比

Table 8. 实验组三加速比汇总

进程数	MPI	优化 1	优化 2	优化 3
4	6.746743	15.425077	17.782738	18.180435
8	11.617592	25.819694	26.966799	30.296308
12	13.885258	26.590271	23.994779	25.295020
16	14.076626	30.068566	30.683314	31.179675
20	15.218083	30.810880	33.898305	37.651136
24	15.463751	25.629340	27.882288	33.384551
28	13.792506	23.513786	23.675673	25.771667
32	14.700455	20.819269	31.394700	24.754013

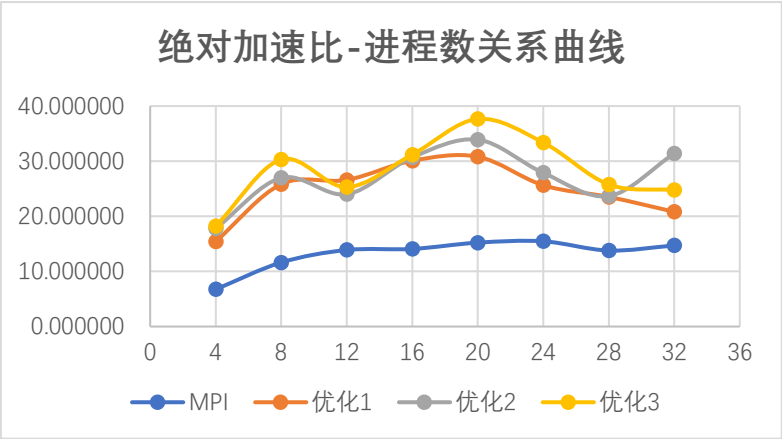

图 12.　实验组三加速比

在数量级较小时，优化 1、优化 2、优化 3 相较于原始 MPI 都有明显的提升，但之间相差不大，并随着进程数的增加，绝对加速比呈现波动趋势。同时，主要的优化提升都是优化 1（去偶数优化）所致，优化 2、3 的效果不明显。

5.3.3　并行效率

Table 9. 实验组三并行效率汇总

进程数	MPI	优化 1	优化 2	优化 3
4	1.686686	3.856269	4.445685	4.545109
8	1.452199	3.227462	3.370850	3.787039
12	1.157105	2.215856	1.999565	2.107918
16	0.879789	1.879285	1.917707	1.948730
20	0.760904	1.540544	1.694915	1.882557
24	0.644323	1.067889	1.161762	1.391023
28	0.492590	0.839778	0.845560	0.920417
32	0.459389	0.650602	0.981084	0.773563

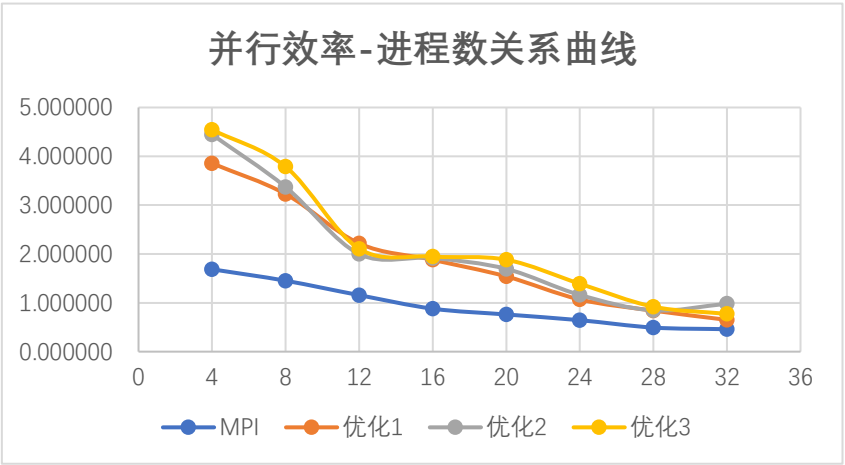

图 13.　实验组三并行效率

分析情况与绝对加速比分析一致。我认为导致的原因之一是数量级小，i/o 占程序的比重比计算

大，另一个原因可能是 Cache 分块太大导致的，本次实习采用的均为 12MB，即 12*1024*1024，默认使用的是 L3 的 cache。因此考虑将分块减小，再分析一下结果。

分块减小后的并行效率-进程数关系曲线：

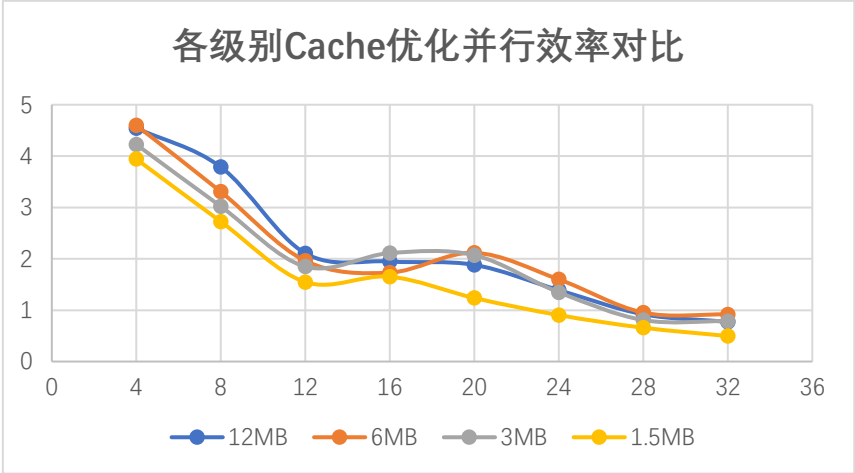

图 14. 各级别 Cache 并行效率

但是从结果上看，对 Cache 分块大小减小得到的并行效率并没有明显的改变。因此，我们继续在 Cache 分块大小为 12MB 不变的情况下，改变 cache_linesize 的大小进行实验。得到的并行效率-进程数关系曲线如下图：

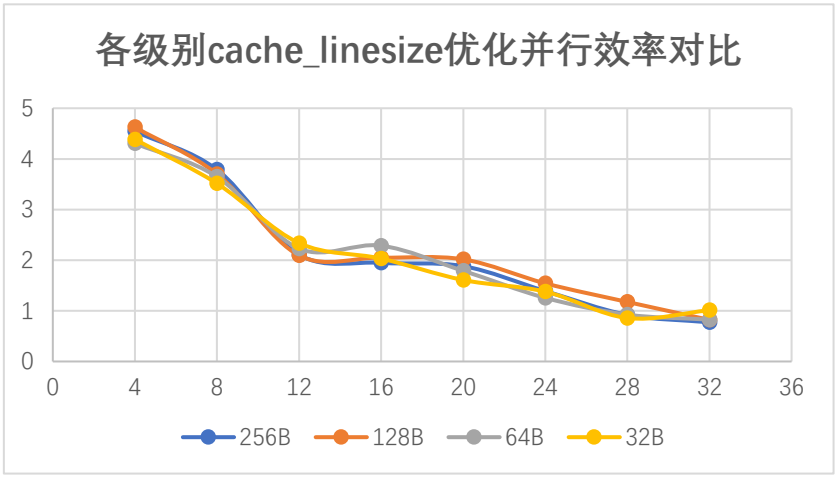

图 15. 各级别 cache_linesize 并行效率

从结果上看，在不同的 cache_linesize 大小下并行效率并没有呈现明显的差别，说明应该也不是 cache_linesize 的大小导致的。因此可以推测是由 i/o 占比增大导致。

6 Amdahl 定律评估

6.1 Amdahl 定律

Amdahl 定律的问题：没有考虑通讯时间 Wc 和并行化带来的开销 We，并行化带来的开销包括应用中单独具有的开销比如本实验中的数据分发，以及普遍具有的开销如进程管理，进程查询等，加入这两个参数后 Amdahl 模型变为

$$S_n = \frac{W_s + W_p}{W_s + {W_p}/{n} + W_c + W_e}$$

进一步变形，设置 $\alpha = \frac{W_s}{W_s+W_p}$，$\beta_c = \frac{W_c}{W_s+W_p}$，$\beta_e = \frac{W_e}{W_s+W_p}$

它们分别代表串行部分所占操作的比重、通信时间所占操作的比重、并行化额外操作所占操作的比重，得到：

$$S_n = \frac{1}{\alpha + \frac{(1-\alpha)}{n} + \beta_c + \beta_e}$$

6.2 加速比验证

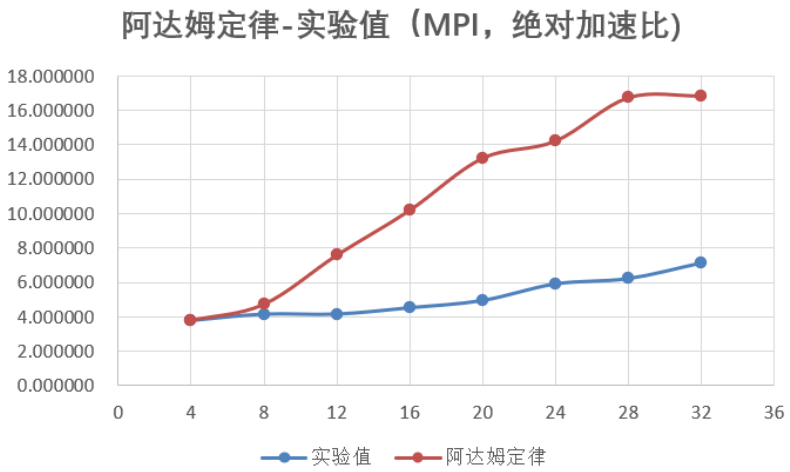

图 16. Amdal 定律计算与实验值对比

由此可见，随着核数的增加，通讯时间和额外开销的影响也越来越大

7 结论与改进思路

7.1 结论

(1).去偶数优化其实只是数量上线性地减少了问题规模（接近一半），通信和广播模式没有改变，所有优化效果是线性，接近 2 倍，并行效率也相似；

(2).核心数少时广播代价并不大，随着核心数增加广播代价增大，所以在核心数很大时去广播优化就明显优于去偶数。

(3).显然存在计算代价、通信代价（或者广播代价、计算代价、通信代价），而通信（通信和广播）会随着核心数增加增大，所以 Cache 优化和去广播优化在加速计算时，如果通信代价过大，计算加速带来的收益就下降，加速比曲线存在峰；而通信代价与直接计算代价的相对比例在不同问题规模不同，使不同问题规模下最大加速比对应核心数不同；

(4).通信代价存在，导致并行效率随核心数增大而下降；

(5).问题规模溢出本人计算机性能，出现大规模加速比下降，反阿姆达尔效应。

(6).从相对加速比上不难得出，在进程数较少时，各数据级情况的程序均呈现良好的可扩展性，而在进程数较大时，更大数据级情况的程序的可扩展性会更好。

7.2 改进思路

7.1 并行策略

本次实验中采用的是数据并行策略，因此可以考虑优化数据分配，通过资料查询，发现有另一种数据分配方法，其计算量较本次实验的方法更少。具体如下：

让进程 i 控制的第一个元素是$[\frac{in}{p}]$,最后一个元素是$\left[\frac{(i+1)n}{p}\right]-1$，对于特定元素$j$，控制它的进程是$[\frac{p(j+1)-1}{n}]$。

7.2 负载均衡

负载均衡是通过调整计算在各个处理器上的分配，以充分发挥系统内处理器的计算能力，通常意味着各个处理器近似同时结束计算。负载均衡法分为两种，分别是静态负载均衡和动态负载均衡：

1）静态负载均衡：指在程序运行前，开发人员已经将计算任务分割为多个部分并保证能够均匀地把各个部分计算分配给控制流运行[3]

2）动态负载均衡：指在程序运行过程中，重新调整任务的分布以达到负载均衡的目的

本次实验使用的是静态负载均衡，只在计算执行前进行了一次分配，因此可以考虑使用动态负载均衡优化程序。

7.3 并行 I/O

数据的串行 I/O 方式严重限制了并行计算性能的提升，这在我们的实验中也有体现。

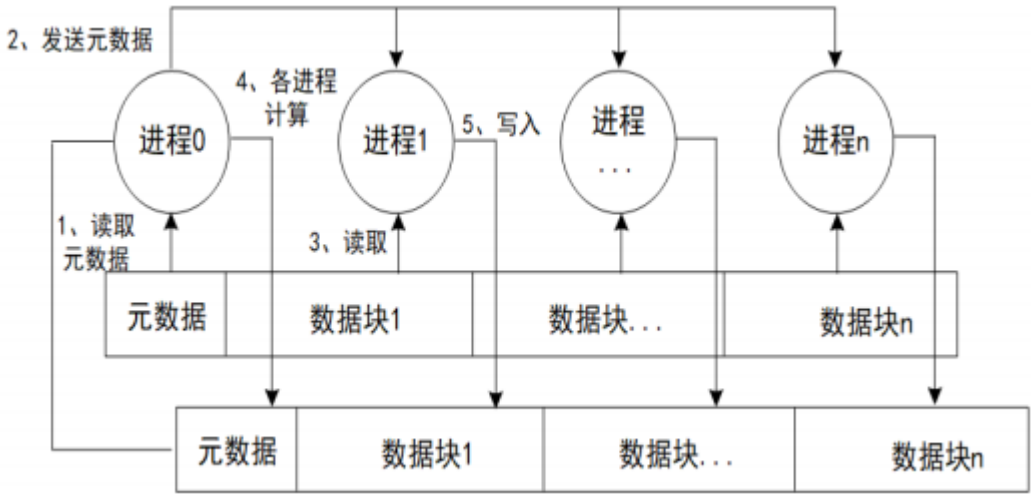

图 17. 串行 I/O 影响

解决方案：可以考虑使用 I/O 异步并行手段，即同时开展多个 I/O 操作(也被称为 overlapped I/O)。I/O 并行的美妙之处在于其伸缩性，在多核的环境下，如果可以充分利用计算资源，则通常会获得 2 倍甚至 8 倍的性能提高[4]。

7.4 通信

在本次实验的优化 2（去广播优化）中，我们采用的是通过使用串行算法先求出一部分质数，然

后最后再通信归总结果，这种做法避免了过多的广播通信，从结果上看，确实也是有效的。然而基于 Eratosthenes 算法本身的特点，我们实习中采用了 MPI_Barrier(MPI_Comm comm）函数。

这个函数就像一道障碍，在操作中，通信子 comm 中的所有进程相互同步，即它们相互等待，直到所有的进程都执行了它们各自的 MPI_Barrier 函数，然后再各自接着开始执行后续的代码，这么做是为了保证控制执行的顺序，但一定程度上降低了程序的效能。因此是否可以考虑使用异步通信方案，并采用诸如“树形结构通信”来进一步减小通信的开支，从而达到更佳的优化结果，这也是一个可以值得尝试的方向。

参考文献：

[1] 王亭亭，基于 OpenMP 和 MPI 的并行算法研究[D].吉林大学硕士学位论文.2011:28 页

[2] 武林平，景翠萍，等. MPI 并行程序中通信等待问题的诊断方法及其应用[J]. 国防科技大学学报，2020，42(2)：47-54

[3] 周伟明.多核计算与程序设计［M］.武汉：华中科技大学出版社，2009.

[4] 任小西,唐玲,张杰. 基于 MPI 多线程动态负载均衡技术研究［J］. 世界科技研究与发展,2008,30(3):281-285.