电子科技大学

计算机专业类课程

实验报告

课程名称：分布式并行计算

学　　院：计算机学院

专　　业：计算机科学与技术

学生姓名：汤佳睿

学　　号：2021080903012

指导教师：卢国明

日　　期：　　2024 年　3 月　30 日

电 子 科 技 大 学

实 验 报 告

实验一

一、实验名称：埃拉托斯特尼素数筛选算法并行及性能优化

二、实验学时：4

三、实验内容和目的：

实验目的：

（1）使用 MPI 编程实现埃拉托斯特尼筛法并行算法。

（2）对程序进行性能分析以及调优。

实验内容：

本实验要求安装部署 MPI 实验环境，并调试完成基准代码及其优化（包括消除偶数，消除广播，提高 cache 命中率等），并实测在不同进程规模（1，2，4，8，16）下的加速比，并进行相应的分析。

四、实验原理：

埃拉托斯特尼算法：

埃拉托斯特尼筛法（Sieve of Eratosthenes）是一种用来找出一定范围内所有素数的简单而有效的算法。它的基本原理如下：

初始化：首先，将 2 到 N（N 为要找素数的范围）的所有数标记为未筛选状态（假设为素数）。

从小到大筛选：从最小的素数开始，逐个筛选掉其倍数，因为这些数肯定不是素数。具体操作是，从 2 开始，将 2 的所有倍数标记为非素数；然后找到下一个未被标记为非素数的数，即下一个素数，再将其所有倍数标记为非素数；重复这个过程直到达到根号 N 为止。

输出素数：最后，所有未被标记为非素数的数即为素数。

这个算法的优点在于它的简单性和高效性。在一般情况下，它的时间复杂度为 O(n log log n)，其中 n 是要找素数的范围。

优化 1：去掉偶数

优化思想利用“大于 2 的质数都是奇数”这一知识，首先去掉所有偶数，偶数必然不是素数，这样相当于所需要筛选的数减少了一半，存储和计算性能都得到提高。

优化 2：消除广播

优化 1 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，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，在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，性能得到提高。

优化 3：Cache 优化

在上面代码基础上，可以根据所掌握的知识，探索重构循环，提高 cache 命中率等方式进一步优化程序性能，并给出结果和分析。

五、实验器材（设备、元器件）

华为云（西南-贵阳一区 弹性云服务器）
鲲鹏通用计算增强型| kc1.large.2 | 2vCPUs | 4GiB

六、实验步骤：

1）在华为云服务器上安装部署 MPI 实验环境，并调试完成基准代码，并实测在不同进程规模（1，2，4，8，16）加速比，并合理分析原因

首先在云服务器上运行“sudo apt update”指令以更新 apt 软件包列表，确保服务器上有最新版本的 mpich，然后运行“sudo apt install mpich”安装 mpich，执行“sudo apt install gdb”通过 apt 安装 gdb 调试工具。安装后运行“mpicc -v”结果如图 1-1 所示，可见安装成功。

```
tjr@ecs-e849:~/prime$ mpicc -v
mpicc for MPICH version 3.3a2
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/aarch64-linux-gnu/7/lto-wrapper
Target: aarch64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 7.5.0-3ubuntu1~18.04' --with-bugurl=file:///usr/share/doc
/gcc-7/README.Bugs --enable-languages=c,ada,c++,go,d,fortran,objc,obj-c++ --prefix=/usr --with-gcc-major-version-only --program
-suffix=-7 --program-prefix=aarch64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-include
d-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-bootstrap --enable-clocale=gnu --enable-libstdcxx-debu
g --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-libquadmath --disable-libqu
admath-support --enable-plugin --enable-default-pie --with-system-zlib --enable-multiarch --enable-fix-cortex-a53-843419 --disa
ble-werror --enable-checking=release --build=aarch64-linux-gnu --host=aarch64-linux-gnu --target=aarch64-linux-gnu
Thread model: posix
gcc version 7.5.0 (Ubuntu/Linaro 7.5.0-3ubuntu1~18.04)
tjr@ecs-e849:~/prime$
```

图 1 检查 mpich 安装信息

通过 vim 编辑器，编辑并调试基准代码 base.cpp，通过“mpic++ -o base base.cpp”编译为可执行文件 base，然后运行分别测试在不同进程规模的运行时间得到相应的加速比，运行结果如图 1-2 所示，图 1-3 显示各线程规模的加速比的结果。（由于时间较慢，所以这里只求 300000000 以内的素数）

```
tjr@ecs-e849:~/prime$ mpirun -np 1 ./base 3000000000
There are 144449537 primes less than or equal to 3000000000
SIEVE (1) 107.329695
tjr@ecs-e849:~/prime$ mpirun -np 2 ./base 3000000000
There are 144449537 primes less than or equal to 3000000000
SIEVE (2)  60.019229
tjr@ecs-e849:~/prime$ mpirun -np 4 ./base 3000000000
There are 144449537 primes less than or equal to 3000000000
SIEVE (4)  60.486930
tjr@ecs-e849:~/prime$ mpirun -np 8 ./base 3000000000
There are 144449537 primes less than or equal to 3000000000
SIEVE (8)  60.331992
tjr@ecs-e849:~/prime$ mpirun -np 16 ./base 3000000000
There are 144449537 primes less than or equal to 3000000000
SIEVE (16)  55.881360
tjr@ecs-e849:~/prime$
```

图 1-2 base 运行结果

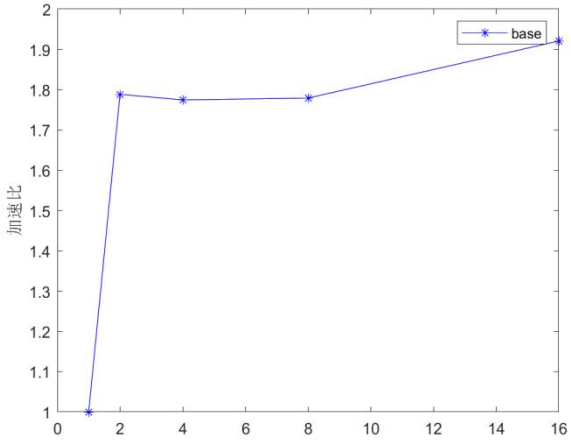


图 1-3 不同规模加速比可视化结果

分析：

通过测试可以发现，当进程规模大于二后加速比增长的速度就开始减缓，考虑到有可能是因为实验计算机为双核的原因，导致在双核计算机上，由于核心资源有限，当进程数量增加时，可能会增加核心

之间的资源竞争。这种竞争可能会导致一些进程等待处理器资源，从而影响整体性能。因此我又在之前配置的三台主机组成的集群上运行同样的代码，结果如图 1-4 所示。可见当进程数大于 2 以后运行速度仍然有较大提升且当进程数为 6 的时候运行速度最快。由此可以推断在单个计算机上当进程数大于二后加速比增速减缓的原因在一定程度上受资源竞争的影响。

```
tjr@ecs-86cd-0001:~/prime$ mpirun -np 6 -f /home/tjr/prime/config /home/tjr/prime/base 3000000000
There are 144449537 primes less than or equal to 3000000000
SIEVE (6)  20.076020
tjr@ecs-86cd-0001:~/prime$ mpirun -np 1 -f /home/tjr/prime/config /home/tjr/prime/base 3000000000
There are 144449537 primes less than or equal to 3000000000
SIEVE (1) 107.174674
tjr@ecs-86cd-0001:~/prime$ mpirun -np 2 -f /home/tjr/prime/config /home/tjr/prime/base 3000000000
There are 144449537 primes less than or equal to 3000000000
SIEVE (2)  59.793741
tjr@ecs-86cd-0001:~/prime$ mpirun -np 4 -f /home/tjr/prime/config /home/tjr/prime/base 3000000000
There are 144449537 primes less than or equal to 3000000000
SIEVE (4)  30.624042
tjr@ecs-86cd-0001:~/prime$ mpirun -np 8 -f /home/tjr/prime/config /home/tjr/prime/base 3000000000
There are 144449537 primes less than or equal to 3000000000
SIEVE (8)  29.818956
tjr@ecs-86cd-0001:~/prime$ mpirun -np 16 -f /home/tjr/prime/config /home/tjr/prime/base 3000000000
There are 144449537 primes less than or equal to 3000000000
SIEVE (16)  21.304933
tjr@ecs-86cd-0001:~/prime$
```

图 1-4 基准程序再三台双核计算机组成的集群的运行结果

2）完成优化 1，去除偶数优化，并实测在不同进程规模（1，2，4，8，16）加速比，并合理分析原因

通过 vim 编辑器，编辑并调试优化 1 的代码 optimize1.cpp，编译为可执行文件 optimize1，然后运行分别测试在不同进程规模的运行时间得到相应的加速比，运行结果如图 2-1 所示，图 2-2 显示各线程规模的加速比的结果。（从这里开始求 400000000 以内的素数）

```
tjr@ecs-e849:~/prime$ mpirun -np 1 ./optimize1 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (1)  70.780496
tjr@ecs-e849:~/prime$ mpirun -np 2 ./optimize1 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (2)  39.512091
tjr@ecs-e849:~/prime$ mpirun -np 4 ./optimize1 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (4)  39.718502
tjr@ecs-e849:~/prime$ mpirun -np 8 ./optimize1 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (8)  38.687988
tjr@ecs-e849:~/prime$ mpirun -np 16 ./optimize1 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (16)  38.616430
tjr@ecs-e849:~/prime$
```

图 2-1 optimize1 执行结果

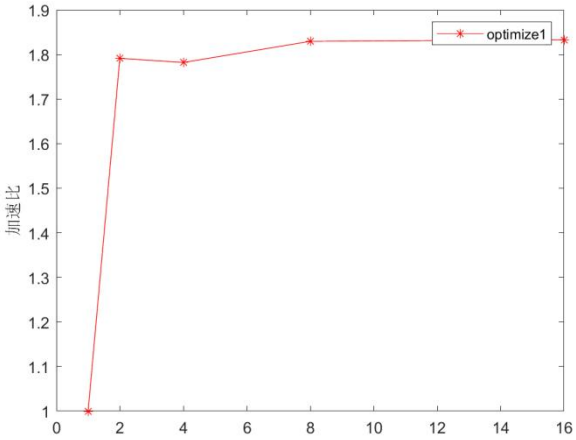


图 2-2 优化一不同进程规模下的加速比

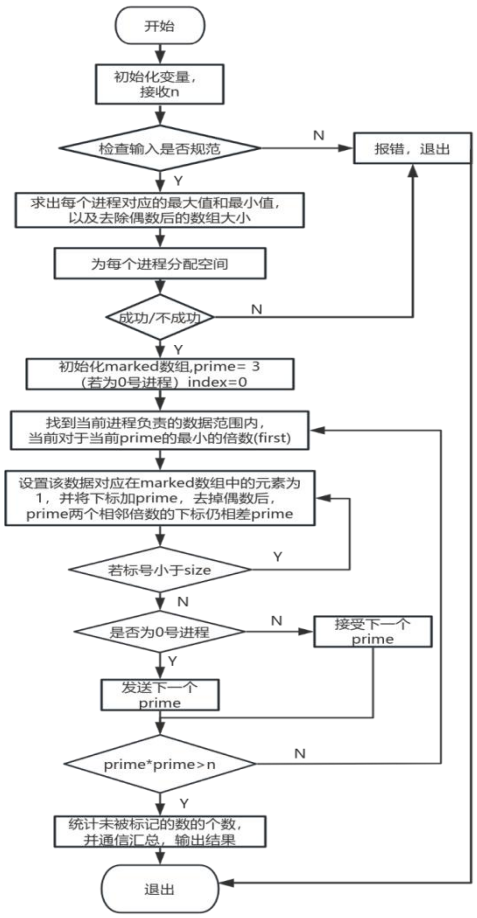
分析：

加速比的变化趋势与基准程序相同，可知有通信，资源竞争等因素影响代码性能。由于本此优化中只是消除了偶数，每一轮的通信开销仍然是影响性能的关键因素，这将在优化二中解决。本程序中的数据从 3 开始，经过数据推到可以得到 low_value = 3 + id * (n - 2) / p; high_value = 2 + (id + 1) * (n - 2) / p;size = (high_value - low_value) /2 + 1；由于当最小值与最大值的同为偶数时，size 变量的大小会多一，因此根据 low_value 是否为偶数，若为偶数则加一，（该偶数的前一个奇数有前一个进程处理），否则不变，即解决 size 大小的问题。

对于找到进程对应数组中的第一个某质数的倍数对应的下标 first,若 prime*prime>low_value 则 first 对应的数为 prime*prime，其对应的下标为（prime*prime - low_value）/2；否则需要判断 low_value 是否为 prime 的倍数若为 prime 的倍数则 first = 0，否则 first = prime - (low_value % prime)。通过举例可以发现

再去除偶数的情况下，两个相邻两个 prime 的倍数的下标相差 prime，因此将 first 赋值给遍历数组并标记所有 prime 的倍数。

程序框图：



3）完成优化 2，消除广播优化，并实测在不同进程规模（1，2，4，8，16）加速比，并合理分析原因

通过 vim 编辑器，编辑并调试优化 2 的代码 optimize2.cpp，编译为可执行文件 optimize2，然后运行分别测试在不同进程规模的运行时间得到相应的加速比，运行结果如图 3-1 所示，图 3-2 显示各线程规模的加速比的结果。（从这里开始求 400000000 以内的素数）

```
tjr@ecs-e849:~/prime$ mpirun -np 1 ./optimize2 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (1)  77.519536
tjr@ecs-e849:~/prime$ mpirun -np 2 ./optimize2 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (2)  40.747841
tjr@ecs-e849:~/prime$ mpirun -np 4 ./optimize2 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (4)  40.039991
tjr@ecs-e849:~/prime$ mpirun -np 8 ./optimize2 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (8)  37.823151
tjr@ecs-e849:~/prime$ mpirun -np 16 ./optimize2 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (16)  34.982329
tjr@ecs-e849:~/prime$
```

图 3-1 optimize2 执行结果

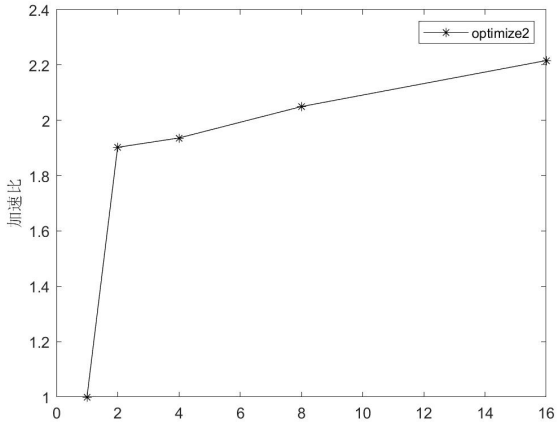
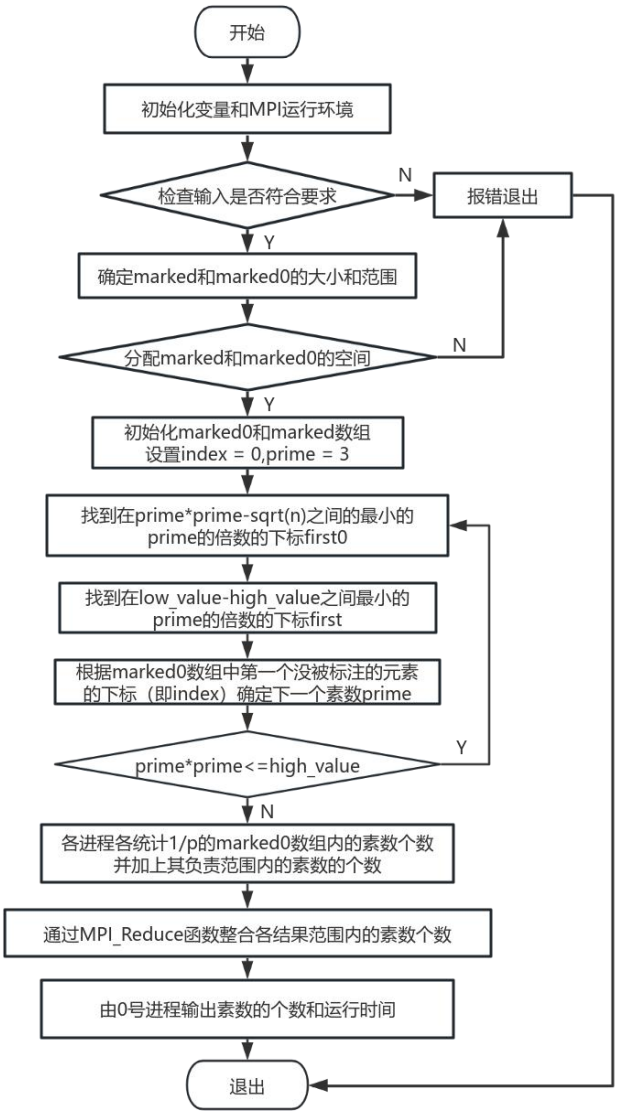


图 3-2 优化二不同进程规模下的加速比

分析：

由于优化一中通信的开销对性能的影响很大，所以本次在优化二的过程中每个进程都初始化两个数组，marked0 和 marked 分别用来求下一个素数和根据给定的素数在本进程负责的范围内删除该素数的倍数。每个进程分别统计 1/p 的 marked0 数组中素数的个数，并加上其负责范围内的素数个数，最终通过 MPI_Reduce 函数汇总结果。最后输出素数的个数和程序运行的时间。由于本程序中每个进程都会求前 sqrt(n)个数中素数，所以本程序中各进程负责的数据要从 sqrt(n)+1 开始，以减少遍历的数据量，因此不必讨论进程 0 是否大于 sqrt(n)。为了均衡负载，所以每个进程各统计 1/p 的 marked0 数组中的素数，加到 count 变量中。但是由于本代码中 marked0 和 marked 数组中标记合数的操作在一个循环中，因此在每一次更换下一个素数的时候都要切换 marked0 和 marked，导致 cache 命中率降低，增加了额外的访问内存的开销，因此优化二的性能提升较优化一并没有很明显。最好先标记所有 marked0 中的合数，再通过根据得到的素数，依次标记 marked 中的合数，将其分为两个循环，这将在优化三中进行改进。

程序框图：



4）完成优化 3，cache 优化，并实测在不同进程规模（1，2，4，8，16）加速比，并合理分析原因

通过 vim 编辑器，编辑并调试优化 3 的代码 optimize3.cpp，编译为可执行文件 optimize3，然后运行分别测试在不同进程规模的运行时间得到相应的加速比，运行结果如图 4-1 所示，图 4-2 显示各线程规模的加速比的结果。（从这里开始求 400000000 以内的素数）。可以发现在随着进程规模的增大加速比在

逐渐提高，但是增速减慢，与优化一和优化二的趋势相似

```
tjr@ecs-e849:~/prime$ mpirun -np 1 ./optimize3 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (1)  49.254714
tjr@ecs-e849:~/prime$ mpirun -np 2 ./optimize3 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (2)  22.872465
tjr@ecs-e849:~/prime$ mpirun -np 4 ./optimize3 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (4)  19.671076
tjr@ecs-e849:~/prime$ mpirun -np 8 ./optimize3 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (8)  18.259995
tjr@ecs-e849:~/prime$ mpirun -np 16 ./optimize3 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (16)  17.155988
tjr@ecs-e849:~/prime$
```

图 4-1 optimize3 执行结果

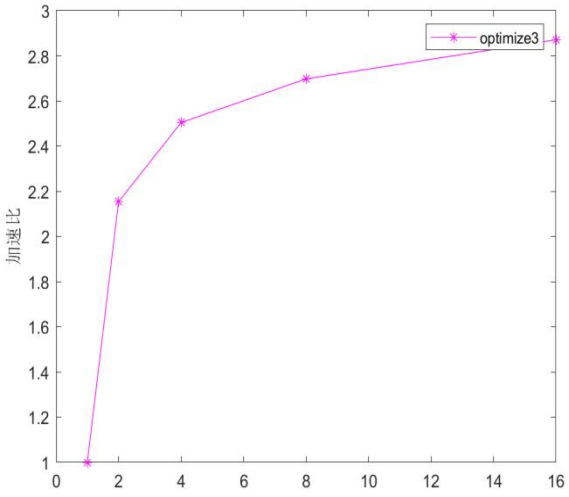


图 4-2 优化三不同进程规模下的加速比

分析：

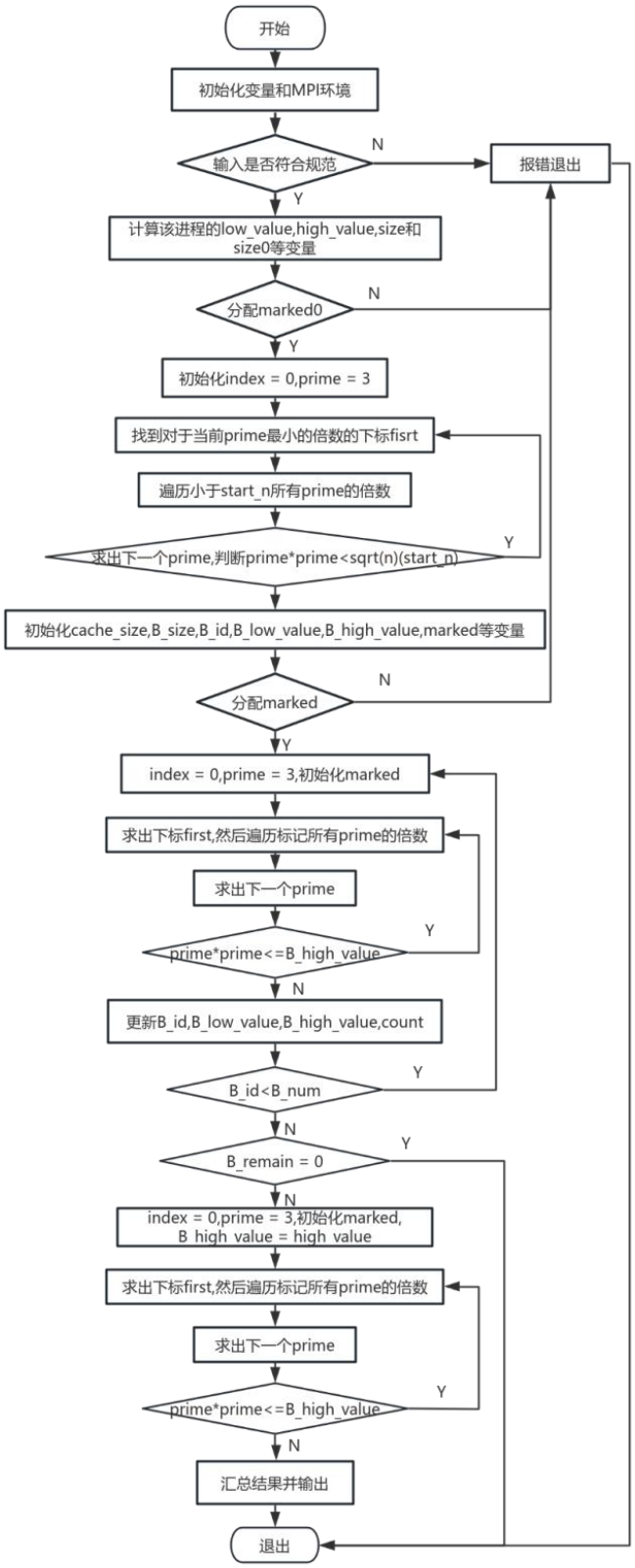
由于优化二中频繁切换 marked0 和 marked 数组对性能影响很大，所以在本次优化的过程中，将处理 marked0 和 marked 分为两个数组，首先求出标记后的 marked0 再依次根据得到素数处理 marked 数组。在第一个循环中得到对于给定素数的第一个大于其平方的倍数的下标 first = (prime*prime - 3)/2,再遍历并标记所有与该数相差 prime 的整数倍的数字，然后根据标记后的 marked0 求出下一个 prime，直到 prime*prime>sqrt(n)(在代码中用 start_n 保存)。

然后优化 marked 数组的标记方式，通过 lscpu 可以得到计算机的 cache 属性如图 4-3 所示。可以看到 L3cache 的大小为 32768KB = 33554432B，因此再代码中设置 cache_size = 33554432,由于每个 int 类型的变量要占 4 各字节所以其能保存 int 类型的数据为 cache_size/4 个，由于一共有 p 个进程所有进程共享 L3cache 所以 cache_int/p 为各进程可以存储的数组大小（用 B_size 表示），B_num = size/B_size 表示该进程需要处理 B_size 大小的数据的次数，B_remain = size%B_size 表示最后还有多少数据没有处理，若 B_remain 非零则需要再次标记 B_remain 大小的数组，每一个数组的标记方法与优化二的 marked 数组类似。

```
root@ecs-e849:~# lscpu
Architecture:        aarch64
Byte Order:          Little Endian
CPU(s):              2
On-line CPU(s) list: 0,1
Thread(s) per core:  1
Core(s) per socket:  2
Socket(s):           1
NUMA node(s):        1
Vendor ID:           0x48
Model:               0
Stepping:            0x1
BogoMIPS:            200.00
L1d cache:           64K
L1i cache:           64K
L2 cache:            512K
L3 cache:            32768K
NUMA node0 CPU(s):   0,1
Flags:               fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm jscvt fcma dcpop asimddp
```

图 4-3 cpu 的相关信息

程序框图：



5）完成最终优化，并得到运行结果：

受消除偶数的启发，在前面的很多数字比如 3，5，7 等，如果在一开始就可以去除其相对应的倍数，则可以大幅减少计算量，且计算量的减小幅度依次减小，因此在最终优化中本实验采取消除 2 和 3 的所有倍数已达到优化程序的目的。

由于不同于前几个优化版本，本次优化难以直接划分求解区间然后通过值来定位全局索引（由于边界值如 high_value，low_value 可能不在规定的数组中），因此本次优化先求解整体数组的长度（n/3）然

后再对数组进行划分，则数组中的每个位置即为对应的索引，然后通过公式（1）求解该索引对应的具体数值。其中 value 表示索引 i 对应位置的数值，i 为全局索引。

$$value_i = 3 * i + 5 - i\%2 \qquad (1)$$

通过举例可以发现去除 2 和 3 的倍数的自然数组成数列为：5,7,11,13...，是 6k-1 和 6k+1 的交替排列（这里的 k 是任意正整数，不是索引），因此我们可以将其分为两个数列，分别求出各素数在两个数列中的索引，然后通过两个循环分别标记相应位置的各素数的倍数。其数值与全局索引的对应关系如（2），（3）所示，在遍历数组标记各素数的倍数时通过该公式反向计算各倍数对应的全局坐标。

对于 $6k-1$ 的数列其通项公式：$a_i = 3 * i + 5$（其中 i 为全局索引） （2）

对于 $6k+1$ 的数列其通项公式：$a_i = 3 * i + 4$（其中 i 为全局索引） （3）

对于某素数的 first 求解，一致任意素数 p 都可以表示为 p = 3r + q（r 和 q 均为整数），r 为 p 除 3 的商，q 为 p 除 3 的余数（由于这里已消除 3 的倍数，因此 q 只能等于 1 或 2）。

首先讨论 q 等于 1 的情况：

由于 q = 1 则可知该素数位于 6k+1 的数列中，因此若 6k-1 的数组中有 p 的倍数则可以表示为 p+4+6k, 因为 2，3，4 的倍数都被去掉了，因此再该数组中最小的 p 的倍数为 5p，据此推导 k 的值的过程如（4）。据此得到全局的最小的 p 的倍数的值为 p+4+12r，然后逆向使用公式（2）得到其全局索引。再 6k+1 的数列中，由于 p 自身就位于其中，且该数列中 p 的倍数表示为 p+6k，显然 6 不能整除 p，因此 k 等于 p, 得到最小的 p 的倍数为 7p，再根据通项公式求得全局索引。

对于 q 等于 2 的情况：

由于 q = 2 则可知其位于 6k-1 的数列中，因此同理 q = 1 时的 6k+1 的情况，k 等于 p。在 6k+1 的数列中的最小的倍数表示为 p+2+6k，同理 q=1 中的相应情况最终得到 k = 2r+1，将 k 代入得到最小的倍数为 p+8+12r，再根据通项公式求其全局索引。

$p = 3r+1, 6k-1$: $\frac{p+4+6k}{p} = 5 \xrightarrow{p=3r+1} \frac{6k+4}{3r+1} = 4 \xrightarrow{移项化简} k = 2r$ （4）

$p = 3r+1, 6k+1$: $\frac{p+6k}{p} = m \xrightarrow{m为整数} k = p$ （5）

$p = 3r+2, 6k-1$: $\frac{p+6k}{p} = m \xrightarrow{m为整数} k = p$ （6）

$p = 3r+2, 6k+1$: $\frac{p+2+6k}{p} = 5 \xrightarrow{p=3r+2} \frac{6k+2}{3r+2} = 4 \xrightarrow{移项化简} k = 2r+1$ （7）

通过上述讨论我们可以求得对于任一素数 p 再全局范围内的最小倍数，但是由于每一块负责不同的数据范围，所以找到再该数据范围内的最小的 p 的倍数，若 B_low_value 是 p 的倍数则直接得到最小的倍数，否则 B_low_value 一定满足不等式（8），其中 k 为任意正整数，表示 B_low_value 一定在两个 p 的倍数之间，令变量 t 为 B_low_value 大于全局 first 的部分对于 6p 的余数，则大于 B_low_value 的最小的 p 的倍数一定是 6p-t，则 B_low_value+6p-t 是其对应范围内的最小的 p 的倍数，再根据通项公式求得全局索引。

$$first_{global} + 6kp < B\_low\_value < first_{global} + 6(k+1)p \qquad (8)$$

$$t = (B\_low\_value - first_{global}) \% 6p \qquad (9)$$

$$first_{global} + 6(k+1)p - B\_low\_value \xrightarrow{B\_low\_value = first_{global} + 6kp + t} 6p - t \qquad (10)$$

根据以上讨论可以得到相应最小倍数的全局索引，但是由于每个进程有不同的起始位置，每个进程内的每一块数据也有不同的起始位置，因此我使用 low_index 变量保存每一个进程负责范围内的起始全局索引，B_low_index 变量保存每一个块内的起始全局索引，因此对于求到的全局所以只要减去 B_low_index，就可以得到其在 marked 数组中的相对位置，然后再通过循环依次标记该范围内的 p 的倍

数。

除此之外，通过观察代码可以发现初始化的时候是通过全部相同的数字对标记数组进行赋值，然而如果能够再一开始初始化的过程中就将 5 的倍数进行标记，则在后续的过程中就可以直接从 7 的倍数开始标记，同样可以减少循环的次数，提高并行程序的性能。因此在该程序对于每一块 marked 初始化的过程中根据 B_low_index%10 的取值选择初始化的的循环，一保证标记所有 5 的倍数，然后设置 prime 从 7 开始，以标记剩余的素数的倍数。综上所有的改进添加金优化三的代码得到 optimize4t，其运行结果如图 5-1 所示，可以看到得到 4000000000 以内的素数有 189961812 个，与前几个优化的结果相同可见结果的正确性，其平均时间再 10.5s 左右，较优化三的 18.25s 有显著提升。

```
tjr@ecs-e849:~/prime$ vim optimize4.cpp
tjr@ecs-e849:~/prime$ mpic++ -o optimize4t optimize4.cpp
tjr@ecs-e849:~/prime$ mpirun -np 8 ./optimize4t 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (8)  10.511984
tjr@ecs-e849:~/prime$ mpirun -np 8 ./optimize4t 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (8)  10.588424
tjr@ecs-e849:~/prime$ mpic++ -o optimize4t optimize4.cpp
tjr@ecs-e849:~/prime$ mpirun -np 8 ./optimize4t 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (8)  10.491997
```

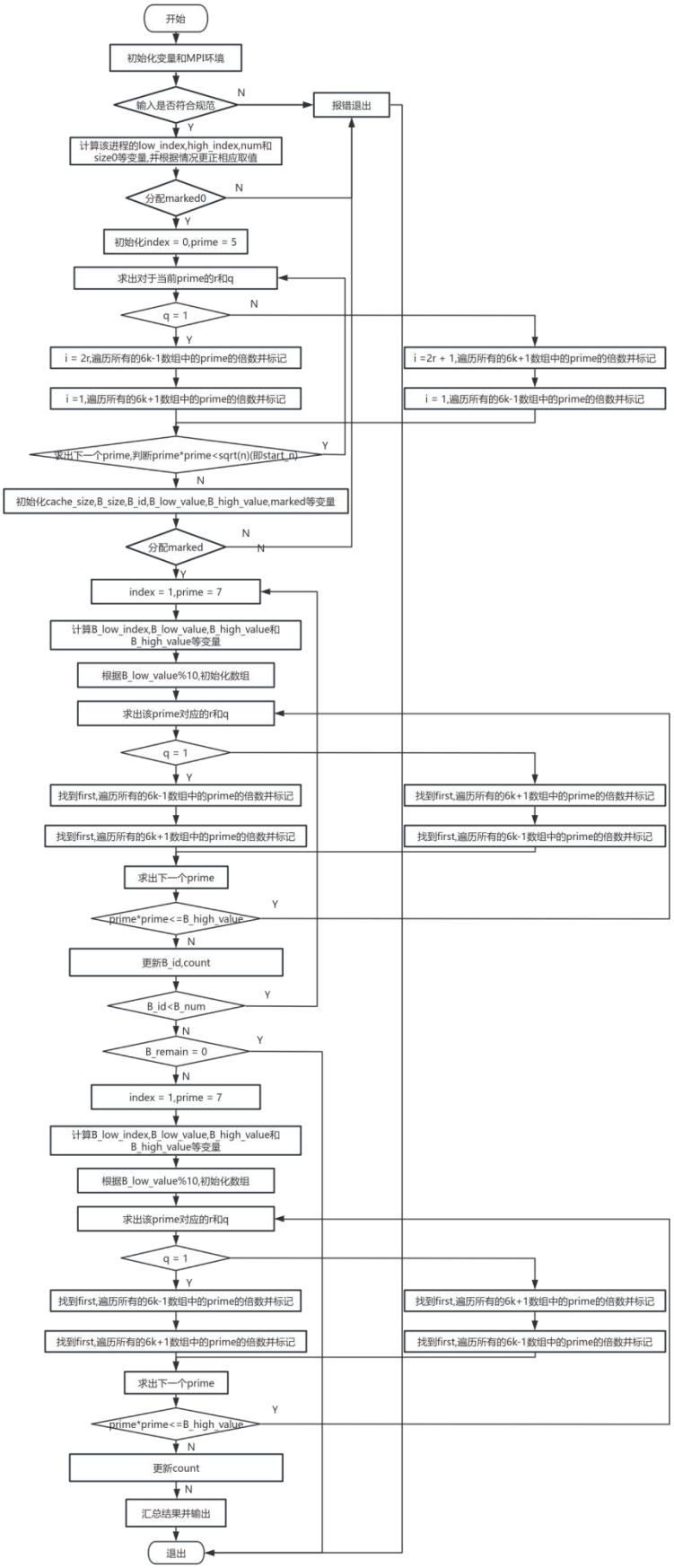
图 5-1 optimize4t 执行结果

进一步更改 optimize4t，使所有进程统计的数据范围由 sqrt(n)~n 变为 5~n,不统计 marked0 中的素数个数。发现虽然增大了要计算的范围，但是时间减少了 0.6s 左右，平均时间约为 9.9s。推测可能是因为 marked0 中的素数由 0 标记，合数由 1 标记，在统计 marked0 中的数据需要使用分支语句，大量的分支语句降低了性能，而再主体的循环过程中，用 1 标记素数，0 标记合数，所以只需要将数组求和就可以得到素数的个数，可以一定程度上节约运行时间提高性能。之所以 marked0 中使用 0 来标记素数，是为了在主体循环寻找下一个素数的过程中一旦 while 循环遇到素数就可以立即跳出避免了对 marked0 取非的操作（这会对性能的影响更大）。除此之外由于起始位置变为 5 然而在之前提及的优化的初始化循环中标记了所有的 5 的倍数，因此 5 虽然作为素数但是也被标记了，为了结果的正确性需要在得到的 global_count 加 3（分别代表 2，3，5）才得到最终的正确结果。图 5-2 为最终 optimize4 的运行情况。

```
tjr@ecs-e849:~/prime$ vim optimize41.cpp
tjr@ecs-e849:~/prime$ mpic++ -o optimize4 optimize41.cpp
tjr@ecs-e849:~/prime$ mpirun -np 8 ./optimize4 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (8)   9.869821
tjr@ecs-e849:~/prime$ mpirun -np 8 ./optimize4 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (8)   9.927995
tjr@ecs-e849:~/prime$ mpirun -np 8 ./optimize4 4000000000
There are 189961812 primes less than or equal to 4000000000
SIEVE (8)   9.955985
tjr@ecs-e849:~/prime$
```

图 5-2 最终 optimize4 执行结果

程序框图：


开始
初始化变量和MPI环境
输入是否符合规范
N
报错退出
Y
计算该进程的low_index,high_index,num和size0等变量,并根据情况更正相应取值
分配marked0
N
Y
初始化index = 0,prime = 5
求出对于当前prime的r和q
q = 1
N
Y
i = 2r,遍历所有的6k-1数组中的prime的倍数并标记
i =1,遍历所有的6k+1数组中的prime的倍数并标记
i =2r + 1,遍历所有的6k+1数组中的prime的倍数并标记
i = 1,遍历所有的6k-1数组中的prime的倍数并标记
求出下一个prime,判断prime*prime<sqrt(n)(即start_n)
Y
N
初始化cache_size,B_size,B_id,B_low_value,B_high_value,marked等变量
分配marked
N
N
Y
index = 1,prime = 7
计算B_low_index,B_low_value,B_high_value和B_high_value等变量
根据B_low_value%10,初始化数组
求出该prime对应的r和q
q = 1
Y
找到first,遍历所有的6k-1数组中的prime的倍数并标记
找到first,遍历所有的6k+1数组中的prime的倍数并标记
找到first,遍历所有的6k+1数组中的prime的倍数并标记
找到first,遍历所有的6k-1数组中的prime的倍数并标记
求出下一个prime
prime*prime<=B_high_value
Y
N
更新B_id,count
B_id<B_num
Y
N
B_remain = 0
Y
N
index = 1,prime = 7
计算B_low_index,B_low_value,B_high_value和B_high_value等变量
根据B_low_value%10,初始化数组
求出该prime对应的r和q
q = 1
Y
找到first,遍历所有的6k-1数组中的prime的倍数并标记
找到first,遍历所有的6k+1数组中的prime的倍数并标记
找到first,遍历所有的6k+1数组中的prime的倍数并标记
找到first,遍历所有的6k-1数组中的prime的倍数并标记
求出下一个prime
prime*prime<=B_high_value
Y
N
更新count
N
汇总结果并输出
退出

七、实验数据及结果分析：

总体来看，随着进程规模的增大，加速比在逐渐增长，且可以通过减少计算数据，减少通信，提高cache 命中率的方法来优化并行程序，且可以直观地感受到合理的并行程序相较串行程序设计可以大幅提高性能，提高工作效率。

综上各不同规模的进程对应的加速比变化可以看出，不论是 base 还是各优化程序在双核计算机上，当进程规模超过两个时，加速比的提升会减缓，经过分析得出这主要是由于以下几个原因：

1. 通信开销增加：随着进程数量的增加，通信开销也会增加。在 MPI 程序中，进程之间需要相互通信以交换数据或同步操作。随着进程数量增加，通信量增加，可能会导致更多的通信延迟，从而抵消了并行计算带来的性能提升。

2. 负载不平衡：随着进程数量增加，负载不平衡的可能性也增加。在某些情况下，一些进程可能会完成其工作并空闲，而其他进程可能会因为负载过重而被迫等待。这种情况会导致部分进程的效率降低，从而限制了整体加速比的提升。

3. 资源竞争：在双核计算机上，由于核心资源有限，当进程数量增加时，可能会增加核心之间的资源竞争。这种竞争可能会导致一些进程等待处理器资源，从而影响整体性能。

因此，在双核计算机上，当进程规模超过两个时，加速比的提升可能会受到这些因素的影响，导致加速比的增加不那么明显。要解决这些问题，可以采取一些优化措施，例如优化通信模式、实现负载均衡、减少资源竞争等。

八、实验结论、心得体会和改进建议：

通过本次实验我们了解了 MPI 通信和各进程运行的原理，熟悉了 MPI 编程的方法，并通过自己动手实现埃式筛算法的并行程序实现以及相关的改进让我们对并行计算对程序的优化效果有了明显的感受，理解了并行计算对于程序改进的重要意义。

电子签名：