**电 子 科 技 大 学**

**实 验 报 告**

|  |  |
| --- | --- |
| **学生姓名：李逢君** | **学 号：2016060601010** |
| **一、实验室名称：品学楼B302** | |
| **二、实验项目名称：基于MPI实现埃拉托斯特尼筛法及性能优化** | |
| **三、实验原理：**  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。  其伪代码如下所示：  Create list of unmarked natural numbers 2, 3, …, n  k  2  Repeat  Mark all multiples of k between k2 and n  k  smallest unmarked number > k  until k2 > n  The unmarked numbers are primes | |
| **四、实验目的：**   1. 掌握MPI环境搭建和MPI程序编译执行方法。 2. 使用MPI编程实现埃拉托斯特尼筛法。 3. 掌握并行程序性能分析以及优化的方法。 | |
| **五、实验内容：**   1. 根据附录1指示，完成MPI编译运行环境的配置。 2. 根据附录3给出的基础MPI版本埃拉托斯特尼筛法sieve1，实测加速比并绘制曲线。 3. 根据附录4给出的优化思路实现程序的并行优化。 | |
| 六、实验器材（环境配置）： Windows：  CPU: Intel® Core™ i7-6700HQ CPU @ 2.60GHz  Cache: L1:256KB L2:1.0MB,  L3:6.0MB  开发环境：Visual Studio 2013，MSMPI 10  Linux：  CPU: Intel(R) Xeon(R) CPU E5-2603 v2 @ 1.80GHz  Cache: L1:32KB  L2:256KB  L3:10.0MB  开发环境：CLION, MPICH | |
| **七、实验步骤及操作：**  **1. MPI环境配置**  **1.1 Linux-MPICH**  安装   * 1. sudo apt-get install mpich   CMakeLists.txt 下配置   1. cmake\_minimum\_required(VERSION 3.13) 2. project(MPI) 4. set(CMAKE\_CXX\_STANDARD 17)  7. find\_package(MPI REQUIRED) 9. include\_directories(${MPI\_INCLUDE\_PATH}) 11. set(CMAKE\_CXX\_COMPILER mpicxx) 12. set(CMAKE\_C\_COMPILER mpicc) 14. add\_executable(MPI main.cpp)   测试代码   1. #include "mpi\_hello.h" 2. #include "cstdio" 3. #include "cstring" 4. #include "mpi.h" 6. **const** **int** MAX\_STRING = 100; 8. **int** main() { 9. **char** greeting[MAX\_STRING]; 10. **int** comm\_sz; 11. **int** my\_rank; 13. MPI\_Init(NULL, NULL); 14. MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz); 15. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank); 17. **if** (my\_rank != 0) { 18. sprintf(greeting, "Greeting from process %d of %d!", my\_rank, comm\_sz); 19. MPI\_Send(greeting, strlen(greeting) + 1, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD); 20. } **else** { 21. printf("Greetings from process %d of %d!\n", my\_rank, comm\_sz); 22. **for** (**int** q = 1; q < comm\_sz; ++q) { 23. MPI\_Recv(greeting, MAX\_STRING, MPI\_CHAR, q, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE); 24. printf("%s\n", greeting); 25. } 26. } 27. MPI\_Finalize(); 28. **return** 0; 29. }   运行结果如下则证明配置成功：   1. /usr/bin/mpirun -np 4 ./mpi\_spmv 2. Greetings from process 0 of 4! 3. Greeting from process 1 of 4! 4. Greeting from process 2 of 4! 5. Greeting from process 3 of 4! 7. Process finished with exit code 0   **1.2 Windows-MSMPI**  windows 下运行mpi首推微软的msmp，因为其配置较mpich更加简单，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，下面是如何在VS2013中引入MSMPI的步骤：  1.在VS中新建C++win32空项目，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。    3. 属性 -- 链接器 -- 输入 -- 附加依赖项中添加msmpi.lib; msmpifec.lib; msmpifmc.lib;    4. 新建cpp文件，同样以上述代码进行测试，若输出正确则环境配置成功  **2. 初始版并行代码及说明**  定义一个数组marked, 每一个元素的下标对应一个整数，它的值表示这个整数是否为素数,值为1是素数，值为0不是素数。  先假定所有的数都是素数，将marked数组置0。  选定第一个整数2，从它对应的数组元素2\*2=4开始依次标记2的倍数，一直标记到最后一个数为止。  接下来选定下一个未标记的数，它一定是素数，在使用广播的形式通知各进程筛选出这个素数的倍数。  这样循环到最后，所有进程中未标记的数之和就是1-n中的所有素数了。  编译和运行初始版本素数筛选程序"sieve1"程序，代码如下：   1. #include "mpi.h" 2. #include "math.h" 3. #include "stdio.h" 4. #include "stdlib.h" 5. #include "string.h" 7. **using** **namespace** std; 9. #define MIN(a, b) ((a)<(b)?(a):(b)) 11. **int** main(**int** argc, **char** \*argv[]) { 12. **int** count;              /\* Local prime count \*/ 13. **double** elapsed\_time;    /\* Parallel execution time \*/ 14. **int** first;              /\* Index of first multiple \*/ 15. **int** global\_count;       /\* Global prime count \*/ 16. **int** high\_value;         /\* Highest value on this proc \*/ 17. **int** id;                 /\* Process ID number \*/ 18. **int** index;              /\* Index of current prime \*/ 19. **int** low\_value;          /\* Lowest value on this proc \*/ 20. **char** \*marked;           /\* Portion of 2,...,'n' \*/ 21. **int** n;                  /\* Sieving from 2, ..., 'n' \*/ 22. **int** p;                  /\* Number of processes \*/ 23. **int** proc0\_size;         /\* Size of proc 0's subarray \*/ 24. **int** prime;              /\* Current prime \*/ 25. **int** size;               /\* Elements in 'marked' \*/ 26. **int** low\_index;          /\* Lowest index on this proc \*/ 27. **int** high\_index;         /\* Highest index on this proc \*/ 29. // 初始化 30. // MPI程序启动时“自动”建立两个通信器： 31. // MPI\_COMM\_WORLD:包含程序中所有MPI进程 32. // MPI\_COMM\_SELF：有单个进程独自构成，仅包含自己 33. MPI\_Init(&argc, &argv); 35. // MPI\_COMM\_RANK 得到本进程的进程号，进程号取值范围为 0, …, np-1 36. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &id); 38. // MPI\_COMM\_SIZE 得到所有参加运算的进程的个数 39. MPI\_Comm\_size(MPI\_COMM\_WORLD, &p); 41. // MPI\_Barrier是MPI中的一个函数接口 42. // 表示阻止调用直到communicator中所有进程完成调用 43. MPI\_Barrier(MPI\_COMM\_WORLD); 45. // MPI\_WTIME返回一个用浮点数表示的秒数 46. // 它表示从过去某一时刻到调用时刻所经历的时间 48. elapsed\_time = -MPI\_Wtime(); 50. // 参数个数为2：文件名以及问题规模n 51. **if** (argc != 2) { 52. **if** (!id) printf("Command line: %s <m> \n", argv[0]); 53. // 结束MPI系统 54. MPI\_Finalize(); 55. exit(1); 56. } 58. // 表示找 <= n的素数 59. n = atoi(argv[1]); 61. **int** N = n - 1; 62. low\_index = id \* (N / p) + MIN(id, N % p); // 进程的第一个数的索引 63. high\_index = (id + 1) \* (N / p) + MIN(id + 1, N % p) - 1; // 进程的最后一个数的索引 64. low\_value = 2 + low\_index; //进程的第一个数 65. high\_value = 2 + high\_index;//进程的最后一个数 66. size = high\_value - low\_value + 1;    //进程处理的数组大小  69. // Bail out if all the primes used for sieving are not all held by process 0 70. proc0\_size = (n - 1) / p; 72. // 如果有太多进程 73. **if** ((2 + proc0\_size) < (**int**) sqrt((**double**) n)) { 74. **if** (!id) printf("Too many processes \n"); 75. MPI\_Finalize(); 76. exit(1); 77. } 79. // allocate this process 's share of the array 80. marked = (**char** \*) malloc(size); 81. **if** (marked == nullptr) { 82. printf("Cannot allocate enough memory \n"); 83. MPI\_Finalize(); 84. exit(1); 85. } 87. // 先假定所有的整数都是素数 88. **for** (**int** i = 0; i < size; i++) marked[i] = 0; 90. // 索引初始化为0 91. **if** (!id) index = 0; 93. // 从2开始搜寻 94. prime = 2; 95. **do** { 96. /\*确定该进程中素数的第一个倍数的下标 \*/ 97. // 如果该素数n\*n>low\_value，n\*(n-I)都被标记了 98. // 即n\*n为该进程中的第一个素数 99. // 其下标为n\*n-low\_value 100. **if** (prime \* prime > low\_value) { 101. first = prime \* prime - low\_value; 102. } **else** { 103. // 若最小值low\_value为该素数的倍数 104. // 则第一个倍数为low\_value，即其下标为0 105. **if** (!(low\_value % prime)) first = 0; 106. // 若最小值low\_value不是该素数的倍数 107. // 那么第一个倍数的下标为该素数减去余数的值 108. **else** first = prime - (low\_value % prime); 109. } 111. // 从第一个素数开始，标记该素数的倍数为非素数 112. **for** (**int** i = first; i < size; i += prime) marked[i] = 1; 114. // 只有id=0的进程才调用，用于找到下一素数的位置 115. **if** (!id) { 116. **while** (marked[++index]); // 先自加再执行 117. prime = index + 2; // 起始加偏移 118. } 120. // 只有id=0的进程才调用，用于将下一个素数广播出去 121. **if** (p > 1) { 122. MPI\_Bcast(&prime, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); 123. } 125. } **while** (prime \* prime <= n); 127. // 将标记结果发给0号进程 128. count = 0; 129. **for** (**int** i = 0; i < size; i++) 130. **if** (marked[i] == 0) { 131. count++; 132. } 133. MPI\_Reduce(&count, &global\_count, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD); 135. // stop the timer 136. elapsed\_time += MPI\_Wtime(); 138. // print the results 139. **if** (!id) { 140. printf("%d primes are less than or equal to %d \n", global\_count, n); 141. printf("Total elapsed time: %10.6f\n", elapsed\_time); 142. } 143. MPI\_Finalize(); 144. **return** 0; 145. }   **3. 优化及程序说明**  **3.1 优化一：去掉偶数**  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  **关键代码**：   1. **int** N = (n - 1) / 2; 2. low\_index = id \* (N / p) + MIN(id, N % p); // 进程的第一个数的索引 3. high\_index = (id + 1) \* (N / p) + MIN(id + 1, N % p) - 1; // 进程的最后一个数的索引 4. low\_value = low\_index \* 2 + 3; //进程的第一个数 5. high\_value = (high\_index + 1) \* 2 + 1;//进程的最后一个数 6. size = (high\_value - low\_value) / 2 + 1;    //进程处理的数组大小 8. **do** { 9. /\*确定该进程中素数的第一个倍数的下标 \*/ 10. // 如果该素数n\*n>low\_value，n\*(n-i)都被标记了 11. // 即n\*n为该进程中的第一个素数 12. // 其下标为n\*n-low\_value，并且由于数组大小减半所以除以2 13. **if** (prime \* prime > low\_value) { 14. first = (prime \* prime - low\_value) / 2; 15. } **else** { 16. // 若最小值low\_value为该素数的倍数 17. // 则第一个倍数为low\_value，即其下标为0 18. **if** (!(low\_value % prime)) first = 0; 19. // 若最小值low\_value不是该素数的倍数 20. // 但是其余数为偶数，那么第一个非素数的索引为该素数剪去求余除以2 21. **else** **if** (low\_value % prime % 2 == 0) first = prime - ((low\_value % prime) / 2); 22. // 若最小值low\_value不是该素数的倍数 23. // 那么第一个倍数的下标为该素数减去余数的值，并且由于数组大小减半所以除以2 24. **else** first = (prime - (low\_value % prime)) / 2; 25. } 26. // 从第一个素数开始，标记该素数的倍数为非素数 27. **for** (**int** i = first; i < size; i += prime) marked[i] = 1; 29. // 只有id=0的进程才调用，用于找到下一素数的位置 30. **if** (!id) { 31. **while** (marked[++index]); 32. prime = index \* 2 + 3; // 起始加偏移 33. } 34. // 只有id=0的进程才调用，用于将下一个素数广播出去 35. **if** (p > 1) { 36. MPI\_Bcast(&prime, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); 37. } 38. } **while** (prime \* prime <= n);   当 prime = 3，考虑序列 15 17 19 21 23，则满足low\_value % prime == 0，所以索引 0 即第一个数 15 就是 prime 的倍数，这个很好理解。  当 prime = 3，考虑序列 3 5 7 9 11 13，则满足prime \* prime > low\_value，那么first = (prime \* prime - low\_value) / 2 即第一个非素数索引为 3 即值为 9，这个和前面的一样，主要是减去一个偏移，不难理解。  比较难以理解的是后面两个判断：  当 prime = 3，考虑序列 17 19 21 23 25，则满足prime \* prime <= low\_value，并且满足low\_value % prime % 2 == 0，那么first = prime - ((low\_value % prime) / 2)，即第一个非素数的位置为 0，即值为15。  当 prime = 3，考虑序列 11 13 15 17 19，则满足prime \* prime <= low\_value，并且满足low\_value % prime % 2 != 0，那么 first = (prime - (low\_value % prime)) / 2，即第一个非素数索引为 2，即值为 15。  我理解了很久，我认为主要是由于每次遇到 prime 与 2 的公倍数的时候性质会发生改变，当 prime = 3，考虑原有序列 7 8 9 10 11 12 13 14 15 16 17，其中 12 为 2 和 3 的公倍数，排除除 12 以外的其他偶数，该序列为 7 9 11 12 13 15 17，再排除 3 的倍数，该序列为 7 11 12 13 17，问题的本质在于小于等于该数（7,11,13,17）最近的并且为 prime （3）和 2 的公倍数 A 是否为偶数。  对于 11，17 而言，即满足条件 3，A 为 9，15 为奇数，偏差 bias（求余）为 2，所以可以更进一步判断 bias 是否为偶数，由于其下一个 prime 倍数的位置只能取 [0, prime - 1]，故而 prime 下一个倍数所在的位置为 prime - bias / 2。  对于 7，13 而言，即满足条件 4，A 为 6，12 为偶数，偏差 bias（求余）为 1，同样由于其下一个 prime 倍数的位置只能取 [0, prime - 1]，故而 prime 下一个倍数所在的位置为 (prime - bias) / 2。  **3.2 优化二：去掉广播通信**  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，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。  **关键代码**：   1. /\* 小数组 \*/ 2. **char** \*sub\_marked;       /\* sub\_mark array \*/ 3. **int** sub\_low\_index;      /\* Lowest index on sub\_array \*/ 4. **int** sub\_low\_value;      /\* Lowest array on sub\_array \*/ 5. **int** sub\_high\_index;     /\* Highest index on sub\_array \*/ 6. **int** sub\_high\_value;     /\* Highest index on sub\_array \*/ 8. **int** sub\_n = (**int**) sqrt((**double**) n); 9. **int** sub\_N = (sub\_n - 1) / 2; 11. sub\_low\_index = 0 \* (sub\_N / p) + MIN(0, sub\_N % p); // 进程的第一个数的索引 12. sub\_high\_index = 1 \* (sub\_N / p) + MIN(1, sub\_N % p) - 1; // 进程的最后一个数的索引 13. sub\_low\_value = sub\_low\_index \* 2 + 3; //进程的第一个数 14. sub\_high\_value = (sub\_high\_index + 1) \* 2 + 1;//进程的最后一个数 16. sub\_marked = (**char** \*) malloc(sub\_n); 17. **if** (sub\_marked == nullptr) { 18. printf("Cannot allocate enough memory \n"); 19. MPI\_Finalize(); 20. exit(1); 21. } 23. // 先假定所有的整数都是素数 24. **for** (**int** i = 0; i < sub\_n; i++) sub\_marked[i] = 0; 26. // 索引初始化为0 27. index = 0; 29. prime = 3; 30. **do** { 31. // 从小数组开始标只会命中第一个条件 32. first = (prime \* prime - sub\_low\_value) / 2; 33. // 从第一个素数开始，标记该素数的倍数为非素数 34. **for** (**int** i = first; i < sub\_n; i += prime) { 35. sub\_marked[i] = 1; 36. } 37. **while** (sub\_marked[++index]); 38. prime = index \* 2 + 3; // 起始加偏移 39. } **while** (prime \* prime <= sub\_n);  42. /\* 大数组 \*/ 43. **do** { 45. ... 46. **while** (sub\_marked[++index]); 47. ... 49. // if (p > 1) { 50. //     MPI\_Bcast(&prime, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); 51. // } 52. } **while** (prime \* prime <= n);   **3.3 优化三：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，因此先需要获取 cache 的相关信息吧。  对于linux，只需要在命令行中输入getconf -a | grep CACHE，则可以看到不同级别的 cache 的相关信息，包括 cache 的大小 size，cache 每行的大小 linesize，cache 的链接形式 n 路组相连。    对于windows，在任务管理器的性能中也能看到相关信息，实验环境的处理器为4个核8处理器。    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，(对应优化思路 2)。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，然后在在每个进程中将 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，32 位系统即除以 4，以上图 windows 为例，L1，L2，L3 缓存分别可以存 64KB，0.25MB，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：  **关键代码**：   1. **int** LEVEL1\_CACHE\_size = 32768;      // default 32768 2. **int** LEVEL2\_CACHE\_size = 262144;     // default 262144 3. **int** LEVEL3\_CACHE\_size = 9485760;    // default 10485760 5. **int** LEVEL1\_CACHE\_int = LEVEL1\_CACHE\_size / 4; 6. **int** LEVEL2\_CACHE\_int = LEVEL2\_CACHE\_size / 4; 7. **int** LEVEL3\_CACHE\_int = LEVEL3\_CACHE\_size / 4; 9. **int** Block\_size = LEVEL3\_CACHE\_int / p ; 10. **int** Block\_num = size / Block\_size; 11. **int** Block\_remain = size % Block\_size; 13. **int** Block\_id = 0; 14. **int** Block\_N = Block\_size - 1; 15. **int** Block\_low\_index = Block\_id \* Block\_N + MIN(Block\_id, Block\_remain) + low\_index; 16. **int** Block\_high\_index = (Block\_id + 1) \* Block\_N + MIN(Block\_id + 1, Block\_remain) - 1 + low\_index; 17. **int** Block\_low\_value = Block\_low\_index \* 2 + 3; 18. **int** Block\_high\_value = (Block\_high\_index + 1) \* 2 + 1; 19. **int** Block\_count; 21. // allocate this process 's share of the array 22. marked = (**char** \*) malloc(Block\_size); 23. **if** (marked == nullptr) { 24. printf("Cannot allocate enough memory \n"); 25. MPI\_Finalize(); 26. exit(1); 27. } 29. // 总计数 30. count = 0; 32. **while** (Block\_id <= Block\_num) { 34. // 索引初始化为0 35. index = 0; 37. // 从3开始搜寻，first为第一个不是素数的位置 38. prime = 3; 40. // 块计数 41. Block\_count = 0; 43. // 先假定块中的整数都是素数 44. **for** (**int** i = 0; i < Block\_size; i++) marked[i] = 0; 46. // 在块内找素数 47. **do** { 48. /\*确定该进程中素数的第一个倍数的下标 \*/ 49. // 如果该素数n\*n>low\_value，n\*(n-i)都被标记了 50. // 即n\*n为该进程中的第一个素数 51. // 其下标为n\*n-low\_value，并且由于数组大小减半所以除以2 52. **if** (prime \* prime > Block\_low\_value) { 53. first = (prime \* prime - Block\_low\_value) / 2; 54. } **else** { 55. // 若最小值low\_value为该素数的倍数 56. // 则第一个倍数为low\_value，即其下标为0 57. **if** (!(Block\_low\_value % prime)) first = 0; 58. // 若最小值low\_value不是该素数的倍数 59. // 但是其余数为偶数，那么第一个非素数的索引为该素数剪去求余除以2 60. **else** **if** (Block\_low\_value % prime % 2 == 0) first = prime - ((Block\_low\_value % prime) / 2); 61. // 若最小值low\_value不是该素数的倍数 62. // 那么第一个倍数的下标为该素数减去余数的值，并且由于数组大小减半所以除以2 63. **else** first = (prime - (Block\_low\_value % prime)) / 2; 64. } 66. // 从第一个素数开始，标记该素数的倍数为非素数 67. **for** (**int** i = first; i < Block\_size; i += prime) { 68. marked[i] = 1; 69. } 71. // 用于找到下一素数的位置 72. **while** (sub\_marked[++index]); 74. prime = index \* 2 + 3; // 起始加偏移 76. } **while** (prime \* prime <= Block\_high\_value); 78. // 统计块内计数 79. **for** (**int** i = 0; i < Block\_size; i++) { 80. **if** (marked[i] == 0) { 81. Block\_count++; 82. } 83. } 85. // 汇总总体计数 86. count += Block\_count;  89. // 处理下一个块 90. Block\_id++; 91. Block\_low\_index = Block\_id \* Block\_N + MIN(Block\_id, Block\_remain) + low\_index; 92. Block\_high\_index = (Block\_id + 1) \* Block\_N + MIN(Block\_id + 1, Block\_remain) - 1 + low\_index; 93. Block\_low\_value = Block\_low\_index \* 2 + 3; 94. **if** (Block\_id == Block\_num) { 95. Block\_high\_value = high\_value; 96. Block\_high\_index = high\_index; 97. Block\_size = (Block\_high\_value - Block\_low\_value) / 2 + 1; 98. } **else** { 99. Block\_high\_value = (Block\_high\_index + 1) \* 2 + 1; 100. } 102. }   **4. 调试说明**  由于MPI的程序编译运行的时候不能打断点，所以调试的时候先判断进程id再在里面打印相应变量的值进行调试。  **5. 项目**  项目完整代码地址如下，包括windows和linux下的实现，篇幅有限不再贴出  https://github.com/543877815/uestc\_Internet\_plus\_course\_project/tree/master/parallel\_programming | |
| **八、实验数据及结果分析：**  **1.运行结果**  下面只贴出windows下的实验数据  使用命令mpiexec -n 8 main.exe 100000000运行生成的可执行文件   * main.exe 为原版代码 * main1.exe 为去掉偶数优化代码 * main2.exe 为去掉通信代码 * main3.exe 为提升 cache 命中率代码，需要额外传入cache大小的参数。 * 因为我windows的L3级cache大小为6MB，所以我设置输入为6000000(Bytes)   8个进程在1,0000,0000中寻找素数个数运行结果截图如下：    可以看到四个程序运行的结果相同，在1,0000,0000中找到的素数个数为5761454+1(去掉偶数之后没有加上2)=57614545个，结果是正确的。初始代码的运行时间为1.562787s，去掉偶数优化后运行时间为0.770396s，去掉广播优化后运行时间为0.759649s，cache优化后运行时间为0.103454s，可以看到伴随着不同的优化进行运行时间相比单处理器无优化有很大的提升。  下面在windows环境下再规模1,0000,0000进行从1到20个进程分别进行10次测试取运行时间平均值，得出的结果如下图：    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，故而时间减半；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，如果考虑分布式计算机，优化效果会更加明显；通过优化三增加cache命中率可以显著提升优化效果。  纵向对比可以发现随着处理器的增加使得运行时间有所减少。  **2.加速比**  加速比计算公式为  https://gss2.bdstatic.com/-fo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D54/sign=79e1e1d9b499a9013f355b321f958d83/2fdda3cc7cd98d10673831fc2d3fb80e7aec900a.jpg  其中：  P指CPU数量  T1指顺序执行[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95" \t "_blank)的执行时间，此处应该为去除MPI的埃拉托斯特尼素数算法运行时间  Tp指当有p个[处理器](https://baike.baidu.com/item/%E5%A4%84%E7%90%86%E5%99%A8" \t "_blank)时，[并行算法](https://baike.baidu.com/item/%E5%B9%B6%E8%A1%8C%E7%AE%97%E6%B3%95)的执行时间  规模为1,0000,0000的时候绘制的加速比 – 处理器曲线如下    可以发现通过优化一去除偶数使得运行时间减半，加速比表现为原始版本的两倍，而通过优化二去掉通信在单机上优化效果不太明显，但在进程较多的时候还是有所体现，因此，加速比比优化一还是有所提升。当Sp=p时，Sp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cache优化外其他三种代码并没有表现出这种趋势，初步分析认为这是由于i/o而不是计算占程序的主要部分导致的。  因此为了让计算占据更多程序运行的更大部分，我将规模增加为10,0000,0000后再进行对比：    可以发现情况依然如此，因此我姑且得出的结论是前三种代码并不是能够产生理想加速比的代码，当然这存在的着通信开销以及一些其他的系统运行开销的影响。  而通过cache缓冲后加速比，在前8核加速比按照处理器核数线性增加，达到8核以后有加速比有上升的趋势但是也有所波动并且不再呈现线性的趋势，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4核8处理器相符。另外有Sp>p，即产生了超线性加速比，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。  **3.并行效率**  并行效率是由加速比派生出的效率，是量度性能的指标，定义为：  https://gss2.bdstatic.com/-fo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D109/sign=09a4fb23b88f8c54e7d3c12f032b2dee/241f95cad1c8a7866fe277c36b09c93d71cf5036.jpg  绘制的并行效率与执行进程数的曲线如下：    可以看到没有使用cache优化的时候并行效率一般介于0~1之间，通过cache命中优化使得产生超线性加速比并且并行效率大于1，并且随着进程的增加，效率呈现下降的趋势。  **4.其他规模**  规模10,000,000的情况下      可以看到在规模较少的时候不同的时候cache的优化不是特别明显，我认为导致的原因之一是i/o占程序的比重比计算大，另一个是因为cache分块太大导致的，我写的是6,000,000，默认使用的是L3的cache，如果用L1或者L2效果可能更加稳定一点。  规模10,0000,0000的情况下      可以看到在规模较大的时候不同的时候加速比就比较稳定了，基本是在前八核呈现线性递增的趋势，之后比较稳定或者趋于平缓，此时的计算占程序的比重较i/o更大，和之前的分析相一致。 | |
| **九、实验结论：**  对原始代码通过多种优化最终使得代码运行时间得到很大的降低，并且通过cache命中优化极大地提高了并行的效率。 | |
| **十、总结及心得体会：**   1. 掌握了MPI环境搭建和MPI程序编译执行方法。 2. 学会了使用MPI编程实现埃拉托斯特尼筛法。   3、 掌握了并行程序性能分析以及优化的方法。  4、 感受到高性能计算的魅力。 | |
| **十一、对本实验过程及方法、手段的改进建议：**  希望老师可以提供更多实验参考资料，特别是针对cache优化部分。 | |
| **报告评分：**  **指导教师签字：** | |