**电 子 科 技 大 学**

**实 验 报 告**

|  |  |
| --- | --- |
| **学生姓名：** | **学 号：** |
| **一、实验室名称：** | |
| **二、实验项目名称：埃拉托斯特尼素数筛选算法并行及性能优化** | |
| **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。 | |
| **四、实验目的：**   1. 用MPI实现埃拉托斯特尼筛法 2. 从去除偶数、消除广播、cache优化三个方面进行性能优化 | |
| **五、实验内容：**   1. 安装部署MPI实验环境，调试完成基准代码，并实测在不同进程规模加速比，并合理分析原因 2. 完成优化1，去除偶数优化，并实测在不同进程规模加速比，并合理分析原因 3. 完成优化2，消除广播优化，并实测在不同进程规模加速比，并合理分析原因 4. 完成优化3，cache优化，并实测在不同进程规模加速比，并合理分析原因 | |
| **六、实验器材（设备、元器件）：**  浪潮 5280M4  CPU： E5-2660 v4 2颗  内存：64G  L1\_cache：32K  L2\_cache：256K  L3\_cache：35840K  本地测试机器CPU：AMD 5900X | |
| **七、实验步骤及操作：**   1. 根据MPICH官方文档在本地虚拟机上部署环境，用于本地测试代码 2. 调试完成基准代码，修改示例中计算每个进程所要处理数据块的代码既可。 3. int N = n - 1;*//按照块分配数据，先去除1，从2开始计算* 4. int least = N % p;*//剩下几个数不满足都分配到* 5. low\_id = (long long) id \* (N / p) + MIN(id, least); 6. high\_id = (long long) (id + 1) \* (N / p) + MIN(id + 1, least) - 1; 7. low\_value = 2 + low\_id; *//第一个数值* 8. high\_value = 2 + high\_id; 9. size = high\_id - low\_id + 1; 10. 去除偶数优化，因为所有偶数都不可能是素数，因此可以只筛选奇数。这样数组的大小少一倍，需要修改映射关系。最困难的部分在于难以找到第一个素数的倍数对应的下标。最后我通过找规律发现了根据余数为奇数或偶数来区分下标的计算规律。改动代码如下 11. int N = (n - 1) / 2;*//按照块分配数据，先去除1，从2开始计算* 12. low\_value = 3 + low\_id \* 2; *//第一个数值* 13. high\_value = 3 + high\_id \* 2; 14. prime = 3; 15. do { 16. if (prime \* prime > low\_value) 17. first = (prime \* prime - low\_value) / 2; 18. else { 19. *// 若low\_value为该素数的倍数，则第一个倍数为low\_value* 20. if (!(low\_value % prime)) first = 0; 21. else if (low\_value % prime % 2 == 0) first = prime - ((low\_value % prime) / 2); 22. *//以27 29 31 33 35为例，5为当前质数，找规律* 23. */\** 24. \* 数 余数 first 5-x  5-余数 25. \* 27 2  4  1  3 26. \* 29 4  3  2  1 27. \* 31 1  2  3  4 28. \* 33 3  1  4  2 29. \*/ 30. *//得到余数为2的倍数，则直接除2.不然用5-余数再除2.* 31. else first = (prime - (low\_value % prime)) / 2; 32. } 33. *// 标记该素数的倍数为非素数* 34. for (i = first; i < size; i += prime) marked[i] = 1; 35. *// 只有id=0的进程才调用，用于找到下一素数的位置* 36. if (!id) { 37. while (marked[++index]); 38. prime = index \* 2 + 3; 39. } 40. if (p > 1) MPI\_Bcast(&prime, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); 41. } while (prime \* prime <= n); 42. 消除广播优化，因为找到的素数都需要广播出去，而通信就会有开销。因此通过让每个进程计算前sqrt(n)个数的素数，然后根据这个进行筛选，就能消除通信。节约的时间为计算前sqrt(n)的素数所需时间-通信所需时间。只需要将原找素数的代码修改一下范围既可实现。要注意在最后的大循环中，下一个素数从before\_marked中找 43. *//选前根号n的素数* 44. int before\_n = (int) sqrt((double) n); 45. before\_marked = (char \*) malloc(before\_n); 46. if (before\_marked == nullptr) { 47. printf("Cannot allocate enough memory \n"); 48. MPI\_Finalize(); 49. exit(1); 50. } 51. memset(before\_marked, 0, before\_n); 52. index = 0; 53. prime = 3; 54. do { 55. *//最小值为3，一定满足prime \* prime > low\_value* 56. first = (prime \* prime - 3) / 2; 57. *//标记该素数的倍数为非素数* 58. for (int i = first; i < before\_n; i += prime) { 59. before\_marked[i] = 1; 60. } 61. while (before\_marked[++index]); 62. prime = index \* 2 + 3; 63. } while (prime \* prime <= before\_n); 64. while (before\_marked[++index]); 65. cache优化，cache优化可以基于cache line优化和基于cache size的优化。基本思路是将每个进程需要处理的数据再次划分成为大小为cache size或者cache line的块（或者几倍于cache size，cache line）。这样CPU每次取数据可以从高速缓冲中获取，而不用多次访问内存。基于cache line优化就是一次处理一个cache line的数据，这样从内存中取数据的次数最好，利用率最高，基于cache size优化就是将要处理的数据放满高速缓冲，利用率稍低。这样操作让内存的利用率提高了，但是需要更多的控制转移指令来实现这个功能，转移指令在流水线中的开销是比较大的，因此需要对于不同规模的数据量，选择不同的方案，不同的cache size。因为这次实验用的数据量为1e9级别的，因此采用cache size的优化方法，那么是选择L1\_cache，L2\_cache，L3\_cache还是其他的数据则需要通过实验来确定。优化代码如下 66. *//cache优化* 67. int block\_size = L1\_cache\_size;*//实测结果* 68. int block\_num = size / block\_size; 69. int Block\_least = size % block\_size; 70. int block\_id = 0; 71. int block\_low\_id = low\_id; 72. int block\_high\_id = block\_size + low\_id; 73. int block\_low\_value = block\_low\_id \* 2 + 3; 74. int block\_high\_value = block\_high\_id \* 2 + 3; 75. if (block\_num == 0) block\_size = size; 76. marked = (char \*) malloc(block\_size); 77. while (block\_id <= block\_num) { 78. index = 0; 79. prime = 3; 80. memset(marked, 0, block\_size); 81. do { 82. if (prime \* prime > block\_low\_value) 83. first = (prime \* prime - block\_low\_value) / 2; 84. else { 85. if (!(block\_low\_value % prime)) first = 0; 86. else if (block\_low\_value % prime % 2 == 0) first = 87. prime - ((block\_low\_value % prime) / 2); 88. else first = (prime - (block\_low\_value % prime)) / 2; 89. } 90. *//标记* 91. for (register int i = first; i < block\_size; i += prime) marked[i] = 1; 92. while (before\_marked[++index]); 93. prime = index \* 2 + 3; 94. } while (prime \* prime <= block\_high\_value); 95. register int block\_count = 0; 96. for (register int i = 0; i < block\_size; i++) { 97. if (!marked[i]) { 98. block\_count++; 99. } 100. } 101. count += block\_count; 102. *// 处理下一块* 103. block\_id++; 104. block\_low\_id = block\_id \* block\_size + low\_id; 105. block\_high\_id = (block\_id + 1) \* block\_size - 1 + low\_id; 106. block\_low\_value = block\_low\_id \* 2 + 3; 107. block\_high\_value = block\_high\_id \* 2 + 3; 108. if (block\_id == block\_num) { 109. block\_high\_value = high\_value; 110. block\_high\_id = high\_id; 111. block\_size = (block\_high\_value - block\_low\_value) / 2 + 1; 112. } 113. } 114. 其他优化，使用register标记最近频繁会使用的变量，可以将此变量放到寄存器中，可以将代码加速一倍。每次清空标记数组通过memset命令，可以加快速度。 115. 调试情况说明，我通过输出Log的方式调试MPI程序，在Log中标注进程的id以及其他必要数据可以进行方便的调试。 116. 遇到的问题及调试过程，在处理1e9的数据的时候，我发现以不同线程数运行程序的时候，输出的结果各不相同，而在1e8的数据的时候没有这个问题。因此我第一想到的是数据过大导致爆int，但是1e9的数据在int的范围内，而改成long long后又能正常运行。于是我对可能出现数据溢出的地方输出当前的值，最后发现在计算low\_id和high\_id的时候中间有数据溢出的情况，导致结果为负数。 | |
| **八、实验数据及结果分析：**   1. 使用shell脚本统计了对于不同进程数，不同数据大小，循环30次的花费时间。 2. 基准代码的加速比分析。通过对于1e3到1e9的规模，以及不同线程数的加速比分析，可以发现在数据量小于1e4的情况下，加速比反而是小于1的，并行程序反而不如串行程序。这说明了并行计算的开销大于计算量，在数据量只有1e4的情况下很合理。而当规模变大的时候，加速比提升不明显，效率也只有大约0.2-0.8。当线程数为16时，由于我的电脑仅有12个核心，所以会导致性能的下降，没有普遍参考价值。  |  |  |  |  |  | | --- | --- | --- | --- | --- | | **规模** | **线程数** | **30次总耗时(ns)** | **加速比** | **效率** | | 1000 | 1 | 164.255 | 1 | 1 | | 1000 | 2 | 3788.986 | 0.043351 | 0.021675 | | 1000 | 4 | 4590.16 | 0.035784 | 0.008946 | | 1000 | 8 | 29848.49 | 0.005503 | 0.000688 | | 1000 | 16 | 30054.513 | 0.005465 | 0.000342 | | 10000 | 1 | 1114.88 | 1 | 1 | | 10000 | 2 | 3444.094 | 0.323708 | 0.161854 | | 10000 | 4 | 4647.013 | 0.239913 | 0.059978 | | 10000 | 8 | 8411.262 | 0.132546 | 0.016568 | | 10000 | 16 | 27295.546 | 0.040845 | 0.002553 | | 100000 | 1 | 11495.433 | 1 | 1 | | 100000 | 2 | 9022.767 | 1.274047 | 0.637024 | | 100000 | 4 | 7180.168 | 1.600998 | 0.400249 | | 100000 | 8 | 10515.267 | 1.093214 | 0.136652 | | 100000 | 16 | 550340.234 | 0.020888 | 0.001305 | | 1000000 | 1 | 124623.529 | 1 | 1 | | 1000000 | 2 | 68432.22 | 1.821124 | 0.910562 | | 1000000 | 4 | 37867.6 | 3.291033 | 0.822758 | | 1000000 | 8 | 27408.011 | 4.546975 | 0.568372 | | 1000000 | 16 | 51413.922 | 2.423926 | 0.151495 | | 10000000 | 1 | 1303763.321 | 1 | 1 | | 10000000 | 2 | 717021.372 | 1.818305 | 0.909152 | | 10000000 | 4 | 387966.371 | 3.360506 | 0.840127 | | 10000000 | 8 | 252159.389 | 5.170394 | 0.646299 | | 10000000 | 16 | 200578.594 | 6.500012 | 0.406251 | | 100000000 | 1 | 22313481.59 | 1 | 1 | | 100000000 | 2 | 13768859.73 | 1.620576 | 0.810288 | | 100000000 | 4 | 9677465.159 | 2.305716 | 0.576429 | | 100000000 | 8 | 7131364.76 | 3.128922 | 0.391115 | | 100000000 | 16 | 6289365.829 | 3.547811 | 0.221738 | | 1000000000 | 1 | 266771025.1 | 1 | 1 | | 1000000000 | 2 | 172630418.9 | 1.54533 | 0.772665 | | 1000000000 | 4 | 127819281.7 | 2.087095 | 0.521774 | | 1000000000 | 8 | 116172106.5 | 2.296343 | 0.287043 | | 1000000000 | 16 | 119451196.1 | 2.233306 | 0.139582 |  1. 基于上述对于单一优化方式的分析，可以发现当数据量小时并行没有任何优势，且收到随机性影响较大，因此之后按照1e9的数据规模进行分析   对于进程数对于加速比和效率的影响，有如下图表。可以看出总体来说进程数的增加使得加速比上升和效率下降，优化的策略并没能达到超线性。但是cache优化对于性能的提升十分明显，大大提高了效率和加速比。这是因为对于内存的访问占用了大量的时间，cache优化就是针对这一部分。根据Amdahl定律，提升明显是显然的。    对于各个优化对于花费时间的影响有以下图表，其中op1和op2重合，说明在本地机器上通信开销基本没有影响。     1. 在本地机器上优化1，2的加速比和效率提升不明显，首先是因为本地机器速度快，运行开销占比小。而且因为不是分布式机器，通信开销低。对于使用register的优化，本地机器基本看不出任何区别，但是在学校服务器上有一倍的差距，可能与编译器或者CPU有关系。Memset的优化，服务器上加速比例明显大于本地机器，因为内存速度不同。 2. Cache size的确定，对于不同的size进行测试，从学校服务器上得出以下数据，可以得出在1e9的规模下使用L1\_cache是最佳选择。 3. 程序框图 | |
| **九、实验结论：**  对于原始的程序进行了多种优化，使得运行时间大幅度降低，并且通过优化cache极大提高了效率。  了解了对于不同运算速度的机器来说，各项优化的作用差距很大，而且加速比和效率也会受到不同机器的大幅影响。 | |
| **十、总结及心得体会：**  1、 学习了MPI环境搭建  2、 学习了用MPI实现埃拉托斯特尼筛法  3、 学习了并行程序性能分析和优化 | |
| **十一、对本实验过程及方法、手段的改进建议：**  希望有更多的优化方法 | |
| **报告评分：**  **指导教师签字：** | |