**电 子 科 技 大 学**

**实 验 报 告**

|  |  |
| --- | --- |
| **学生姓名： 巢玲威** | **学 号：2019081305001** |
| **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。  这里，我们把Ｎ取120来举例说明埃拉托斯特尼筛法思想：   1. 首先将２到120写出 2. 在２上面画一个圆圈，然后划去２的其它倍数，这时划去的是除了２以外的其它偶数 3. 从２往后一个数一个数地去找，找到第一个没有被划去的数3，将它画圈，再划去３的其它倍数（以斜线划去） 4. 再从３往后一个数一个数地去找，找到第一个没有被划去的数５，将它画圈，再划去５的倍数（以交叉斜线划去） 5. 再往后继续找，可以找到9、11、13、17、19、23、29、31、37、41、43、47…将它们分别画圈，并划去它们的倍数（可以看到，已经没有这样的数了 6. 这时，小于或者等于120的各数都画上了圈或者被划去，被画圈的就是素数了 | |
| **四、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。 | |
| **五、实验内容：**   1. 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（40分） 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（10分） 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（15分） 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（10分）。 5. 性能得分：在完成优化3的基础上，可以利用课内外知识，全面优化代码性能。根据全班优化3在目标机上实测性能，最高性能（最短执行时间）得分25分，最低性能得0分，其他按执行时间进行插值。（25分） | |
| **六、实验器材（设备、元器件）：**  1.服务器配置：  目标机器配置：  浪潮 5280M4  CPU： E5-2660 v4 2颗  内存：64G  2.本地PC配置：  操作系统：Windows 10 x64  CPU：Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40 GH  内核：4 | |
| **七、实验步骤及操作：**  1.配置环境：按照教程在本地PC端配置MPI环境、下载远程连接工具Xshell  2. 基准代码测试：  首先将实验指导书上给的基准代码保存为文件base.cpp，然后登录集群，运行基准代码，发现出现如图1错误：    图1 基准代码运行报错  根据错误提示，发现基准代码缺少头文件 “stdlib.h”、“string.h”，添加即可如图2：    图2 修改基准代码  测试基础代码：分别测试进程规模为1、2、4、8、16，数据为1000000时，结果如图3：    图3 基准代码测试1000000  根据图3所示，当进程规模为2、4、8、16时，都能得出正确结果78498，但p = 1时，程序输出错误，查看并分析基准代码，发现基准程序只处理了p>1的情况（图4）：即使用MPI的规约函数MPI\_Reduce，将通信子内各进程的记录素数个数的count变量相加，并将求和结果保存在global\_count中发送给0号进程，但忽略了p = 1的情况。    图4 基准代码处理p > 1情况  当p = 1时，即只有一个进程，进程0的count即为2-n之间的所有素数个数，直接将进程0的count的内容存放到global\_count里，修改如图5：    图5 添加p = 1的处理  再次按相同数据规模测试添加了p = 1的情况处理的基准程序，结果如图6：    图6修改之后基础代码测试结果  根据图6测试结果所示，经过修改之后，基准代码能够正确输出。下面考虑代码优化。  3. 优化一：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。  不考虑去掉偶数时，用于存放数据数组num[i]的下标i与值的对应关系为：value = i+ 2（i <= n - 2）;  考虑去掉偶数时，因为2是唯一是素数的偶数，因此我们从3开始筛选，只需要将最后的global\_count + 1即可得到2 – n之间的素数个数。因此用于存放数据的数据num里面存放的全部是奇数，下标i与值的映射关系为：value = 2 \* i + 3。在对数据进行分块时，可以先对数组下标进行分块，再利用下标与值的映射关系，求得当前进程的数据范围。  对数组下标进行分块：去掉偶数之后的数据总量为N = (n – 3 ) / 2 + 1;  对于进程i：  low\_index = id \* N / p;  high\_index = (id + 1) \* N / p – 1;  根据i与value的映射关系求得进程i的数据范围：  low\_value = 2 \* low\_index + 3;  high\_value = 2 \* high\_index + 3;  关键代码如图7、图8所示：（完整代码见文件optimizer1.cpp）    图7 去掉偶数优化分配数据模块    图8 去掉偶数优化筛选倍数模块  程序框图如图9：    图9 去掉偶数优化程序框图  4. 优化二：消除广播通信  基准代码base.cpp与去掉偶数优化optimizer1.cpp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。  消除广播通信的关键在于每个进程存放有下一个未被标记的最小的素数，因此在每个进程分配一个数组sub\_marked[ – 3) / 2+ 1]，用于存放3 里的素数。  关键代码如图10、图11所示：（完整代码见文件optimizer2.cpp）    图10 消除广播通信优化寻找3 素数代码    图11 消除广播通信筛选倍数代码  程序框图如图12：    图12 消除广播通信优化程序框图  5. 优化三：cache优化  每个进程根据机器Cache Block的大小，将待筛选数据进一步分块，在每个块内使用3中的素数进行标记筛选，从而提高cache命中率。  使用命令getconf -a | grep CACHE获取实验服务器的cache大小，cache信息如图13：    图13 服务器的cache信息  本地PC端的cache信息通过任务管理器->性能查看，cache信息如图14：    图14 本地PC端的cache信息  由图13、图14可得，服务器的L3级cache大小为36700160，每个进程分得的cahce块大小cache\_block\_size = cache\_size / p，因为一个int数据占四个字节，因此每个进程的cache块最多存放block\_num = cache\_block\_size / 4个数据，我们将当前进程的数据以block\_num为大小进行分块，在每个块里用3之间的素数进行标记，提高cache命中率。对于服务器，block\_num = (36700160 / p ) / 4 ， 对于本地PC端，block\_num = (8 \* 1024 \* 1024 / p ) / 4。  因为服务器和本地PC的cache大小不一样，所以在本地运行时设置的block\_size = 8388608 / p进行测试，而提交的完整代码里block\_size = 36700160 / p。  关键代码如图15、图16所示：（完整代码见optimizer3.cpp）    图15 cache优化求block\_num代码    图16 cache优化筛选倍数代码  程序框图如图17：（计算3~sqrt(n)之间的素数、块内素数筛选具体过程见优化1程序框图，此程序框图不再详细展开）    图17 cache优化程序框图 | |
| **八、实验数据及结果分析：**  1. 基准代码、三种优化代码测试  测试系统：windows系统  测试规模：1、2、4、8、16  测试数据：1e8（即100000000）    图18 p = 1时四个程序测试结果    图19 p = 2时四个程序测试结果    图20 p = 4时四个程序测试结果    图21 p = 8时四个程序测试结果    图22 p = 16时四个程序测试结果  从图18-22中可以看出，去掉偶数优化optimizer1的时间大约是基准代码base的一半，消除广播通信优化optimizer2的时间比optimizer1短，cache优化optimizer3的运行时间最短，且相对base、optimizer1、optimizer2时间减少程度大，优化性能明显提高。测试结果符合预期设想。（为了排除偶尔性，以上测试用相同测试数据在同一机器上按照相同测试方法均测试了不少于10组，本实验报告取其中一组测试结果分析，其余测试结论与本组实验基本一致）  测试系统：实验室环境 linux  测试规模：1、2、4、8、16  测试数据：1e8（即100000000）  测试结果如表1：（表中数据与windows系统下的测试结果有所偏差，考虑是服务器负载过大，同一时段执行实验的同学过多，但基本规律一致）  表1 服务器上的测试结果   |  |  |  |  |  | | --- | --- | --- | --- | --- | | p | base | optimizer1 | optimizer2 | optimizer3 | | 1 | 32.381755 | 14.713852 | 11.340639 | 7.683426 | | 2 | 22.147030 | 11.413209 | 9.724965 | 3.900512 | | 4 | 21.789988 | 10.483733 | 2.743995 | 1.967011 | | 8 | 21.575012 | 10.993988 | 1.573990 | 1.252995 | | 16 | 20.253019 | 10.807995 | 0.795998 | 0.625016 |   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  根据计算公式，绘制三种优化代码的加速比曲线如图23：  图23 加速比  从图23可以看出通过去掉偶数优化使得运行时间减半，加速比大约是2，消除广播通信优化的加速比与去掉偶数优化相差不大，大约都在3左右，而cache优化的加速比随着进程数的增加也在显著提高，初步表现出进程数加倍，执行速度加倍的趋势。  去掉偶数优化：  从图23可以看出，去掉偶数优化的加速比并没有随着进程数增加而有明显的增加，大概稳定在2左右，从图24中可以看出，进程数加倍，代码的执行速度并没有随之加倍，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，并且进程增加，进程间通信开销也随之增加。    图24 去掉偶数优化加速比测试  （当然由于是在本地实验环境测试的，实验负载会有一定影响）  消除广播通信优化：  从图23中可以看出，消除广播通信的加速比与去掉偶数优化的加速比相差不大，但随着进程数增多，其加速比也在小幅度增加，但依旧与去掉偶数优化一样，没有随着进程数的加倍而执行速度加倍。从图25可以看出，当进程数从2变为4时，4变为8时，执行时间并没有依次减半。分析可能是由于系统的其他运行开销或是程序在串行寻找素数的开销影响。    图25 消除广播通信优化加速比测试  Cache优化：  从图23中可以看出，cache优化的加速比曲线是比较符合理想加速比的情况，而从图26中可以看出，当进程数8增加到16时，其执行时间并没有大幅减少，猜测是由于本实验的测试数据为1e8，测试环境的L3级cache缓存为8MB，数据量较少所以导致加速不明显。    图26 cache优化加速比测试 | |
| **九、实验结论：**  通过对基准代码进行去掉偶数优化，使得执行时间减半，再进行消除广播通信优化，执行时间有小幅度减少，而通过cache优化使得程序执行时间有大幅度的减少，符合预期设想。 | |
| **十、总结及心得体会：**  这次实验对我来说是一次难度很大的实验，因为我之前完全没有接触过分布式并行编程知识。在实验过程中，遇到了很多问题，比如当时对于去掉偶数优化时思路不清楚，导致做了很多无用功，后来请教了身边同学，才彻底捋清优化思路，得以顺利完成优化一；优化二对我来说是比较顺利的，自己独立思考完成；在进行优化三时，我又去复习了一遍cache的知识，但却遇到了测试时间漫长、内存分配失败的问题，后来经过同学提醒，大概是因为测试数据过大（测试数据为1e9）程序在运行计算过程中溢出导致的，于是修改数据类型为long long，问题得以解决。通过这次实验，我初步了解和掌握了mpi编程的知识，拓宽了知识广度，编程能力也有所提高。 | |
| **十一、对本实验过程及方法、手段的改进建议：**  希望老师能提供更多关于cache优化的知识，实验指导书更加详细。 | |
| **报告评分：**  **指导教师签字：** | |