**电 子 科 技 大 学**

**实 验 报 告**

|  |  |
| --- | --- |
| **学生姓名：** | **学 号：** |
| **一、实验室名称：**主楼A2-412 | |
| **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：** 埃拉托斯特尼筛法及描述 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。  伪代码如下：   1. 创建一个自然数2，3，4，… ，n的列表，其中所有的自然数都没有被标记。 2. 令k = 2，它是列表中第一个未被标记的数。 3. 重复下面的步骤直到： 4. 被和n之间的是k倍数的数都标记出来。 5. 找出比k大的未被标记的数中最小的那个，令k等于这个数。 6. 列表中未被标记的数就是素数。  * 数据块分配方法   先分配每个进程需要计算的数值，需要知道每个进程的最小数值和最大数值，以及给定数值属于哪个进程。（n：数值个数，p：进程数）  进程i的第一个数值：i \* n / p  进程i的最后一个数值：(i + 1) \* n / p - 1  给定数值j属于哪个进程：(p \* (j + 1) - 1) / n  基准代码流程图如下图所示，其中串行代码即上述伪代码描述的部分：    图3-1 base.cpp流程图 程序的优化  * 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。  加上“去掉偶数”的优化后的流程图如下图所示：    图3-2 optimizer1.cpp流程图   * 1. 消除广播   2.1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。  加上“消除广播”的优化后的流程图如下图所示：    图3-3 optimizer2.cpp流程图   * 1. 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。  加上“cache优化”后的流程图如下图所示：    图3-4 optimizer3.cpp流程图 | |
| **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；  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；  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；  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；  5．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。 | |
| **六、实验器材（设备、元器件）：**  浪潮 5280M4  CPU： E5-2660 v4 2颗  内存：256G | |
| **七、实验步骤及操作：**  **1. 调试完成基准代码**  （1）将/data/base.cpp复制到/2020060902021/下，直接运行，指定n为1e10，发现返回Segment fault。  （2）将n改为1e9再次运行成功得到结果。  （3）经过代码分析后发现是在计算发现基准代码在计算块内素数的时候发生了溢出，将相关变量改为long long类型后成功运行。  （4）测试选取的数据规模为1e10，在不同进程规模下测试性能，选取的进程规模有1、2、4、8、16和32。为了避免偶然性，对于每组数据，均在相同环境下测试10次，并取平均值作为最终结果。测试结果见：八、实验数据及结果分析部分。  **2. 调试完成优化1——去掉偶数**  在基准代码base.cpp的基础上，利用“除2以外的所有偶数都不是素数”这一点，可以将待筛选数字总量减少一半，从而提高筛选效率。记经过第一次优化后的代码为optimizer1.cpp，主要修改部分如下图所示：    图7-1 optimizer1.cpp数据初始化部分  因为n个数中1与2不用考虑，因此只用考虑剩下n-2个数，则分块大小应为(n - 2) / p（其中为进程数），由于我们从3开始筛选，因此还应该加上3，由此便建立了块内下标和真实值之间的一一映射。由于每个块内都有一半的偶数，但考虑到整除问题，因此将size设置成(high\_value – low\_value) / 2 + 1。    图7-2 optimizer1.cpp主体循环过程  在base.cpp的基础上，在计算first时，我们要考虑偶数的影响：①如果low\_value与prime \* prime的距离为偶数（即low\_value % prime为偶数），则需要“追两圈”，first应设置成prime - （low\_value % prime）/ 2；②如果low\_value与prime \* prime的距离为奇数（即low\_value % prime为奇数），则只需要“追一圈”，first应设置成（prime - （low\_value % prime））/ 2。  测试选取的数据规模为1e10，在不同进程规模下测试性能，选取的进程规模有1、2、4、8、16和32。为了避免偶然性，对于每组数据，均在相同环境下测试10次，并取平均值作为最终结果。测试结果见：八、实验数据及结果分析部分。  **3. 调试完成优化2——消除广播**  基准代码是通过进程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,产生一定的开销。因此，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，减少进程间广播的通信，以提高性能。  基于第一次优化后的代码optimizer1.cpp，记经过第二次优化后的代码为optimizer2.cpp，主要增加部分如下图所示：    图7-3 optimizer2.cpp新增参数    图7-4 optimizer2.cpp维护前sqrt(n)个素数  因为我们要在每个进程都维护3到sqrt(n)范围内的素数，所以需要新增common\_marked数组进行标记。    图7-5 optimizer2.cpp答案统计  因为每个进程中都维护了前sqrt(n)个素数，所以省去了广播的开销。  测试选取的数据规模为1e10，在不同进程规模下测试性能，选取的进程规模有1、2、4、8、16和32。为了避免偶然性，对于每组数据，均在相同环境下测试10次，并取平均值作为最终结果。测试结果见：八、实验数据及结果分析部分。  **4. 调试完成优化3——cache优化**  基准代码是先用一个素数对剩余的所有数进行筛选，再用下一个素数对剩余的所有数进行筛选。由于需要找出该素数的所有倍数，cache命中率很低，进程会不停地访问不同的块，造成频繁地换入换出，效率很低。因此，在一个块中用不同的素数依次找出它们的倍数，再访问下一个块，这样可以提高cache命中率，进而提升性能。  在命令行输入getconf -a | grep CACHE查询服务器的Cache缓存信息，结果如下图所示：    图7-6 实验所用服务器的CACHE配置信息  从图中可以看出，实验所用服务器的L1数据CACHE和L2 CACHE大小总和为49152 B + 1310720 B = 1359872 B。再来分析性能瓶颈循环：    图7-7 optimizer3中性能瓶颈循环  此处使用了i、first、k、size、prime五个longlong变量，chunk一个int变量，和char数组marked，经计算可得分块大小设置为（1359872 B – 64 B）/ 1 B = 1359808最为合适。  基于第二次优化后的代码optimizer2.cpp，记经过第三次优化后的代码为optimizer3.cpp，主要修改部分如下图所示：    图7-8 optimizer3.cpp新增参数  优化3的主要思路是将数据进一步分块，在每次标记完整块内的素数再标记下一块。定义变量block\_low为分块内的最小值，chunk为分块大小（即每次循环步长）。    图7-9 optimizer3.cpp分块处理关键代码  主体循环如上图所示，我们在依据chunk所分块内先标记完所有合数，增加cache命中率，减少内存读写次数。  测试选取的数据规模为1e10，在不同进程规模下测试性能，选取的进程规模有1、2、4、8、16和32。为了避免偶然性，对于每组数据，均在相同环境下测试10次，并取平均值作为最终结果。测试结果见：八、实验数据及结果分析部分。  **5. 自选优化并调试**  在自选优化部分，我主要使用了以下6个优化思路：  ①压缩marked数组空间  因为实际上我们只关注对应数字的标记值0或1，所以其实只需要一个二进制位即可完成。在之前的代码中，我们使用一个字节来存储标记信息。所以可以考虑进行压缩，用一个字节来存储连续八个数字的信息，这样空间将减少到原来的1/8，有助于进一步优化cache，提高cache命中率，主要改动代码如下：    图7-10 optimizer4.cpp优化思路①关键代码1  如果想访问某一个数字i对应的数组下标，只需要访问i/8即可，想找到对应的位数，计算i%8即可，赋值和访问相关操作代码如下：    图7-11 optimizer4.cpp优化思路①关键代码2  ②预处理优化计数查询  在原来代码中的统计count的过程中，需要逐一求和，效率比较低。在进行了压缩后，我们只需要查询一个八位二进制串中1的个数即可。因此我们可以先预处理0 - 255中1的个数，预处理过程可以使用动态规划的思路，转移方程num[i] = (i & 1) + num[i >> 1]，关键代码如下：    图7-12 optimizer4.cpp优化思路②关键代码1    图7-13 optimizer4.cpp优化思路②关键代码2  ③改变内存申请方式  在原来的代码中，使用malloc动态申请空间，但是malloc返回的空间中的数据是不确定的，因此需要初始化（时间较长）。在优化中，我使用了calloc函数，calloc也是动态申请空间的一种方式，但calloc保证了返回的空间中的数据都是0，因此可以省去初始化的过程，相关代码如下：    图7-14 optimizer4.cpp优化思路③关键代码  ④减少数据冒险  在原来的代码中，统计count时，因为可能会连续对count进行写操作，可能构成数据冒险，我采用的优化思路是增加了count0、count1、count2，使得count不会连续写，减少了可能的数据冒险，相关代码如下：    图7-15 optimizer4.cpp优化思路④关键代码  ⑤循环展开  在原来的代码中，循环均未做出优化。但我们可以合理设置步长，来使得CPU的多发并行效率提高，最常用到的就是循环展开，两段修改的循环均展开了16级，相关代码如下：    图7-16 optimizer4.cpp优化思路⑤关键代码1    图7-17 optimizer4.cpp优化思路⑤关键代码2  ⑥分支优化  在原来的代码中，均未做分支优化，存在结构冒险，如果CPU的分支预测准确率很低，会影响时间效率。考虑到素数分布是稀疏的，因此可以考虑在判断是否为素数的分支预测为不是，减少结构冒险，相关代码如下：    图7-18 optimizer4.cpp优化思路⑥关键代码  测试选取的数据规模为1e10，在不同进程规模下测试性能，选取的进程规模有1、2、4、8、16和32。为了避免偶然性，对于每组数据，均在相同环境下测试10次，并取平均值作为最终结果。测试结果见：八、实验数据及结果分析部分。 | |
| **八、实验数据及结果分析：**  **1. 基准代码数据及分析**  如表8-1所示，平均运行时间随着进程数增多而减少，Sp/p<1。但随着进程数的增加，性能的提升幅度越来越小。这可能是因为系统创建进程、管理进程以及调度进程具有一定的开销。进程越多，系统主要的开销越会花在这些进程上，反而是不利于计算的。在进程数超过一定值后，性能的提升趋近于没有，而多进程带来的CPU占用率会提高，不利于同时段CPU的其它使用，因此，寻求时间效率与系统开销的折中是很重要的。  表8-1 base.cpp运行时间及加速比对比   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | base.cpp运行时间（单位：s） | | | | | | | 次数\进程数 | 1 | 2 | 4 | 8 | 16 | | 1 | 300.277181 | 144.996837 | 80.703660 | 45.744926 | 28.220607 | | 2 | 304.714271 | 154.699129 | 83.403386 | 44.641785 | 28.094008 | | 3 | 317.107308 | 165.724957 | 82.102221 | 44.986804 | 28.968335 | | 4 | 288.774775 | 149.859659 | 82.234427 | 46.524470 | 28.001650 | | 5 | 280.326367 | 145.008291 | 83.393955 | 45.570238 | 30.297180 | | 6 | 286.761389 | 170.473357 | 85.151335 | 47.617547 | 28.279524 | | 7 | 288.406886 | 148.783041 | 79.000847 | 45.494534 | 27.868304 | | 8 | 286.119802 | 151.546604 | 81.195134 | 44.135905 | 28.588903 | | 9 | 341.341422 | 144.326944 | 81.714109 | 44.396109 | 27.705766 | | 10 | 281.863600 | 145.368585 | 80.597736 | 44.600239 | 28.418534 | | 平均运  行时间 | 297.569300 | 152.078740 | 81.949681 | 45.371256 | 28.444281 | | 进程加速比 | 1.00 | 1.96 | 3.63 | 6.56 | 10.46 |   **2. 优化1——去掉偶数数据及分析**  如表8-2所示，平均运行时间随着进程数增多而减少，Sp/p<1。但随着进程数的增加，性能的提升幅度越来越小。这可能是因为系统创建进程、管理进程以及调度进程具有一定的开销。  再将优化1的平均运行时间与基准代码的平均运行时间进行比较，发现都加速了2倍以上。这个结果是符合预期的，这是因为优化1去除偶数数值，使得数据规模减少一半，也就是程序的计算量直接减少了一半，性能提升两倍。  表8-2 optimizer1.cpp运行时间及加速比对比   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | optimizer1.cpp运行时间（单位：s） | | | | | | | 次数\进程数 | 1 | 2 | 4 | 8 | 16 | | 1 | 141.748385 | 72.540448 | 40.960218 | 21.903297 | 14.199672 | | 2 | 138.026870 | 80.718025 | 42.710666 | 22.951769 | 14.103563 | | 3 | 138.281590 | 73.099093 | 42.740540 | 22.090401 | 14.275156 | | 4 | 137.489889 | 73.079048 | 43.541340 | 22.670970 | 14.340211 | | 5 | 138.484035 | 74.005605 | 41.363930 | 22.598108 | 14.516805 | | 6 | 138.534870 | 73.134448 | 41.205826 | 22.585971 | 14.490582 | | 7 | 137.917369 | 73.909972 | 42.341255 | 22.633663 | 14.241550 | | 8 | 138.057126 | 75.580131 | 40.150057 | 22.963148 | 14.618435 | | 9 | 143.718579 | 73.824226 | 41.427985 | 22.513223 | 14.703742 | | 10 | 149.710936 | 74.136748 | 43.603124 | 23.082119 | 14.342538 | | 平均运  行时间 | 140.196965 | 74.402774 | 42.004494 | 22.599267 | 14.383225 | | 进程加速比 | 1.00 | 1.88 | 3.34 | 6.20 | 9.75 |   **3. 优化2——消除广播数据及分析**  如表8-3所示，平均运行时间随着进程数增多而减少，Sp/p<1。但随着进程数的增加，性能的提升幅度越来越小。这可能是因为系统创建进程、管理进程以及调度进程具有一定的开销。  再将优化2的平均运行时间与优化1的平均运行时间进行比较，发现性能提升不是很明显。在进程数为1时，因为本身不存在广播开销，所以基本运行时间与优化1相同。在进程数为2及以上时，由于广播开销得以消除，基本相对优化1运行时间加快1-2秒。  这可能是因为进程间的广播通信时间与计算时间相比，占比较小，优化后减少的运行时间就不多。另外，如果测试时指定的进程数比较少，不能充分发挥“消除广播”的优化作用。当数据规模足够大并且进程数足够多时，优化2才能体现出较为明显的性能提升。  表8-3 optimizer2.cpp运行时间及加速比对比   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | optimizer2.cpp运行时间（单位：s） | | | | | | | 次数\进程数 | 1 | 2 | 4 | 8 | 16 | | 1 | 138.303674 | 72.717218 | 39.497027 | 21.921082 | 13.779253 | | 2 | 137.572590 | 73.308649 | 39.675876 | 21.838809 | 13.316547 | | 3 | 137.053308 | 70.971269 | 39.732650 | 21.810561 | 13.843979 | | 4 | 143.135001 | 69.913968 | 37.946583 | 21.900835 | 13.725600 | | 5 | 137.992517 | 69.855852 | 39.903025 | 21.871528 | 13.512712 | | 6 | 137.617200 | 72.882529 | 39.820703 | 21.882096 | 13.602820 | | 7 | 144.076775 | 72.937765 | 39.729085 | 22.102942 | 13.830752 | | 8 | 137.758831 | 72.691491 | 41.981424 | 21.894575 | 13.986094 | | 9 | 140.901256 | 73.920668 | 39.627623 | 21.856188 | 13.416315 | | 10 | 139.637679 | 72.876057 | 38.301317 | 21.844545 | 12.545870 | | 平均运  行时间 | 139.404883 | 72.207547 | 39.621531 | 21.892316 | 13.555994 | | 进程加速比 | 1.00 | 1.93 | 3.52 | 6.37 | 10.28 |   **4. 优化3——cache优化数据及分析**  如表8-4所示，平均运行时间随着进程数增多而减少，在进程数为2、4、8时，Sp/p>1，即出现了超线性加速比。但随着进程数的增加，性能的提升幅度越来越小。这可能是因为系统创建进程、管理进程以及调度进程具有一定的开销。  通过查阅资料，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。  再将优化3的平均运行时间与优化2的平均运行时间进行比较，发现性能提升很明显。并且，随着进程数的增多，提高cache命中率对性能提升的作用越来越大。在优化2的基础上，性能普遍提升两倍以上。  表8-4 optimizer3运行时间及加速比对比   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | optimizer3.cpp运行时间（单位：s） | | | | | | | 次数\进程数 | 1 | 2 | 4 | 8 | 16 | | 1 | 75.601991 | 36.618872 | 18.452214 | 9.333401 | 4.796708 | | 2 | 73.868904 | 36.562188 | 18.389426 | 9.240284 | 4.867348 | | 3 | 76.038426 | 40.475045 | 18.805101 | 9.273355 | 4.846367 | | 4 | 73.882308 | 37.039007 | 18.995278 | 9.493920 | 4.875669 | | 5 | 76.406499 | 36.661491 | 18.376134 | 9.226977 | 4.852733 | | 6 | 79.857435 | 36.612953 | 18.285204 | 9.320431 | 4.806278 | | 7 | 73.752766 | 36.629700 | 18.264166 | 9.273686 | 4.831994 | | 8 | 73.779233 | 36.967090 | 19.058516 | 9.324555 | 4.835125 | | 9 | 73.728570 | 36.616287 | 18.240472 | 9.208428 | 4.883429 | | 10 | 73.641637 | 37.477275 | 19.057896 | 9.446432 | 4.822695 | | 平均运  行时间 | 75.055777 | 37.165991 | 18.592441 | 9.314147 | 4.841835 | | 进程加速比 | 1.00 | 2.02 | 4.04 | 8.06 | 15.50 |   **5. 自选优化数据及分析**  如表8-5所示，平均运行时间随着进程数增多而减少，在优化3的基础上，在进程数为16时，也出现了超线性加速比。但随着进程数的增加，性能的提升幅度越来越小。这可能是因为系统创建进程、管理进程以及调度进程具有一定的开销。  再将自选优化的平均运行时间与优化3的平均运行时间进行比较，发现性能提升很明显，说明我们的优化策略是很有用的。并且，因为我们在存储标记的时候进行了压缩优化，cache步长可以设置更大，感觉目前设定值（8e6）并未完全挖掘程序潜能，相信更加精确的值可以使程序在16进程数下最快运行时间低于2s。  表8-5 optimizer4运行时间及加速比对比   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | optimizer4.cpp运行时间（单位：s） | | | | | | | 次数\进程数 | 1 | 2 | 4 | 8 | 16 | | 1 | 35.724446 | 17.704309 | 8.649820 | 4.280815 | 2.268144 | | 2 | 35.778934 | 17.635614 | 8.645706 | 4.312398 | 2.145523 | | 3 | 37.362213 | 17.643357 | 8.646744 | 4.324361 | 2.629479 | | 4 | 38.209094 | 17.639470 | 8.644777 | 4.812564 | 2.184965 | | 5 | 35.691701 | 17.651804 | 8.649580 | 4.521962 | 2.255717 | | 6 | 38.656181 | 17.645218 | 8.649001 | 4.308948 | 2.203937 | | 7 | 37.207597 | 17.636334 | 8.646910 | 4.903062 | 2.294772 | | 8 | 35.679522 | 17.630108 | 8.649185 | 4.249633 | 2.236092 | | 9 | 35.755938 | 17.616034 | 8.841162 | 4.301786 | 2.188624 | | 10 | 35.725390 | 17.633907 | 8.653453 | 4.275288 | 2.181244 | | 平均运  行时间 | 36.579102 | 17.643616 | 8.667634 | 4.429082 | 2.258850 | | 进程加速比 | 1.00 | 2.07 | 4.22 | 8.26 | 16.19 | | |
| **九、实验结论：**  1．成功使用MPI编程实现埃拉托斯特尼筛法并行算法，并对程序进行性能分析以及调优。  2．由实验数据得知，总体性能：base<optimizer1<optimizer2<optimizer3<optimizer4，符合预期的实验结果。该结果也证明了每一次优化都是有用的，实验成功。  3．“去掉偶数”、“cache优化”和自选优化能带来较大的性能提升，而“消除广播”带来的性能提升较小，因为广播通信并不是限制程序运行效率的瓶颈，并且本身计算前sqrt(n)个素数也带来了一定的时间开销。  4．在一定范围内，随着进程的增多，并行程序的性能就越好，但是提升的速度越来越慢。原因在于多进程带来的管理进程的开销可能会造成系统资源的浪费，并且多进程只能优化代码中的可并行部分，存在理论最大加速比。因此，在实际应用中应该注重时间效率与其它开销的折中。  5．cache优化提升cache命中率时，块的大小的选择很重要。通过实验选取合适的块大小，才能得到较好的性能。块过小或过大都会导致性能的降低。 | |
| **十、总结及心得体会：**  通过本次实验，我复习了课程中所学的知识，对课程中学到的一些较为抽象的并行程序的概念有了更深入的了解，也对并行程序对串行程序性能的提升有了更深入的认识。在编写、调试、修改并行程序的过程中，巩固课堂知识的同时，我还锻炼了编程能力与实践能力。在对基准程序进行一步步优化的过程中，我学会了在网上查找相关资料，拓宽了知识面。总的来说，通过本次实验，我锻炼了分析问题、解决问题、得出结论的能力，获益匪浅。 | |
| **十一、对本实验过程及方法、手段的改进建议：**  无 | |
| **报告评分：**  **指导教师签字：** | |

电子签名：

