**电 子 科 技 大 学**

**实 验 报 告**

|  |  |
| --- | --- |
| **学生姓名：关文聪** | **学 号：2016060601008** |
| **一、实验室名称：无** | |
| **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：**  **Eratosthenes素数筛选原理：**  Eratosthenes（公元前276~194）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。  Eratosthenes筛法的伪代码如下：   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。   **Eratosthenes筛法MPI实现：**  **(1)数据块分配方法**  先分配每个进程所需要计算的元素，需要解决两个问题：每个进程的最小元素和最大元素、给定元素属于哪个进程。  方法1：  当n(总元素数量)%p(进程数)等于0时，每个进程分配n/p空间大小。  当n(总元素数量)%p(进程数)不等于0时，令r=n%p,则前r个进程数据长度为n/p+1，后n-r个进程数据长度为n/p。  进程i的第一个元素：i\*(n/p)+min(i,r)  进程i的最后一个元素：（i+1）\*（n/p）+min(i+1,r)-1  给定元素j属于哪个进程：min(j/(n/p+1),(j-r)/(n/p))  方法2:  进程i的第一个元素：i\*n/p  进程i的最后一个元素:(i+1)\*n/p-1  给定元素j属于哪个进程:(p\*(j+1)-1)/n  **初始版并行代码：**  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。 | |
| **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：**  操作系统：Windows 10 x64  编程环境：Microsoft MPI（MSMPI）、MinGW-W64 gcc version 8.1.0  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）：**  1.**C/C++编译器环境配置：**  首先，到MinGW官方网站：[http://www.mingw-w64.org](http://www.mingw-w64.org/) 下载并安装MinGW-W64（详细过程略）。安装完成后，进行环境变量配置：将mingw64目录下的bin文件夹添加到系统环境变量的Path中；新建环境变量“INCLUDE”，将mingw64目录下的include文件夹添加到该环境变量“INCLUDE”中；新建环境变量“LIB”，将mingw64目录下的lib文件夹添加到该环境变量“LIB”中，保存退出。  运行cmd.exe打开命令行窗口，输入“gcc -v”，能成功输出gcc版本信息，即为配置成功，如图所示：  **2.MS-MPI环境配置：**  首先，到Microsoft官方网站：<https://www.microsoft.com/en-us/download/details.aspx?id=57467> 下载MS-MPI，注意“msmpisdk.msi”和“msmpisetup.exe”两个文件都要下载，如图所示：    然后，分别进行安装。  二者均安装完成后，运行cmd.exe打开命令行窗口，输入“set MSMPI”可以看到MSMPI环境变量已经自动配置完成。输入“mpiexec”可以正常输出相关信息，则MSMPI环境配置成功，如图所示： | |
| **七、实验步骤及操作：**  首先，将实验源代码保存并命名为“sieve1.c”，使用gcc+命令行参数编译运行，将编译生成的可执行文件命名为“sieve1.exe”。具体命令：$ gcc -o sieve1.exe sieve1.c -l msmpi -L "C:\Users\Eternity-Myth\Desktop\Microsoft SDKs\MPI\Lib\x64" -I "C:\Users\Eternity-Myth\Desktop\Microsoft SDKs\MPI\Include" （说明：在使用gcc编译时，-o参数用于自定义编译生成的文件名；-l参数用来指定程序要链接的库，此处编译时需要链接msmpi.lib库；-L参数用于指定程序编译时寻找库的路径；-I参数用于指定引入的头文件所在目录。库文件与头文件所在目录可通过命令“set MSMPI”查看系统环境变量得到，注意在64位系统的机器上运行需要使用x64的库）    如图所示，编译出现了若干警告和提示信息，但无错误，程序编译成功。生成可执行文件“sieve1.exe”。接下来只要直接调用mpiexec.exe并指定process个数与筛选范围运行程序即可：命令“mpiexec -np n sieve1.exe number”可以指定启动n个process并在小于等于number范围的正整数范围筛选素数。通过指定参数运行程序，得到结果如下图所示    显然，可以发现，程序在这两次运行中都出现了不正确的结果，因此，需要对程序进行修改。  首先解决process数为1时的异常情况：  查看并分析源代码sieve1.c，注意到在源代码的90、91行处只判断和处理了“p>1”的情况，如下图所示：    对此，在该语句后添加处理即可，修改后的程序语句如下图所示：    修改后保存退出，按照之前所述命令重新编译并运行程序，指定参数-np 1时，程序能够输出正确的结果，如下图所示：      接下来解决素数个数统计错误的问题：  查看并分析源代码sieve1.c，注意到在统计素数的个数时，是通过判断数组marked[]中每个元素的值是否为0来统计的。由经验猜测，对于这种统计个数相差1个的类似情况，原因往往是数组的初始化或者数组的下标问题，在某种特定的条件或边界值下会出错。进一步观察可以发现，marked[]数组分配内存空间位于源代码第62行，随后，在第70行，对数组中的元素赋予了初始值0，如下图所示：      进一步调试后修改程序：注释掉或者直接删除第70行的赋值操作，并修改第62行代码如下：    修改后保存退出，按照之前所述命令重新编译并运行程序，程序运行结果正确，如图所示：    至此，程序已经能够成功运行并且得到正确的结果，但我们可以进一步对程序进行拓展，使程序可以处理更大范围的数据：将源代码中的“low\_value”与“high\_value”变量修改为“long int”类型变量，并且，对源代码中第45行-47行的代码（如下图所示）作如下修改：  （原始）  （修改后）  接下来对sieve1程序在不同的数据规模和不同进程配置下的性能进行测试。在本次测试中，测试进程数分别为1、2、4、8和16，选取的数据规模有：1000、10000、100000、1000000、10000000和100000000。也就是说，一共需要测试5\*6=30组数据。为避免偶然性，对于每组数据，均在相同的测试环境下测试10次，并取平均值作为最终结果。测试结果列表如下：   |  |  |  |  | | --- | --- | --- | --- | | 组号 | 进程数（np） | 数据范围 | 10次测试平均时间（秒） | | 1 | 1 | 1000 | 0.000015 | | 2 | 1 | 10000 | 0.0000827 | | 3 | 1 | 100000 | 0.0008434 | | 4 | 1 | 1000000 | 0.0117579‬ | | 5 | 1 | 10000000 | 0.1666059‬ | | 6 | 1 | 100000000 | 1.9808614‬ | | 7 | 2 | 1000 | 0.0000741 | | 8 | 2 | 10000 | 0.0001282‬ | | 9 | 2 | 100000 | 0.0006611‬ | | 10 | 2 | 1000000 | 0.0065801 | | 11 | 2 | 10000000 | 0.1108784 | | 12 | 2 | 100000000 | 1.4257667‬ | | 13 | 4 | 1000 | 0.0004401 | | 14 | 4 | 10000 | 0.0005051‬ | | 15 | 4 | 100000 | 0.0009826‬ | | 16 | 4 | 1000000 | 0.0042348‬ | | 17 | 4 | 10000000 | 0.0885877‬ | | 18 | 4 | 100000000 | 1.2055877 | | 19 | 8 | 1000 | 0.0012258‬ | | 20 | 8 | 10000 | 0.0011222‬ | | 21 | 8 | 100000 | 0.0028717‬ | | 22 | 8 | 1000000 | 0.0051616‬ | | 23 | 8 | 10000000 | 0.0653903‬ | | 24 | 8 | 100000000 | 1.1574096‬ | | 25 | 16 | 1000 | 0.0037298‬ | | 26 | 16 | 10000 | 0.0042704‬ | | 27 | 16 | 100000 | 0.0046095‬ | | 28 | 16 | 1000000 | 0.0080269 | | 29 | 16 | 10000000 | 0.0623539‬ | | 30 | 16 | 100000000 | 1.1127653‬ |   表一：sieve1程序测试结果表  根据表一结果，可以绘制sieve1程序在不同进程数与不同数据规模下的并行性能曲线图（篇幅所限，图与结果详细分析请参见：八、实验数据及结果分析 部分）  接下来，在sieve1程序的基础上，对算法进行性能优化。  **优化思路1 去掉待筛选偶数：**  显然，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。  基于以上思想，修改源代码sieve1.c（注，由于代码改动处较多，此处不一一列举出，优化后的完整代码将在实验报告后的附录中给出），记经过第一次优化去掉待筛选偶数后的代码为modification1.c。  接下来对modification1程序在不同的数据规模和不同进程配置下的性能进行测试。在本次测试中，测试进程数分别为1、2、4、8和16，选取的数据规模有：1000、10000、100000、1000000、10000000和100000000。也就是说，一共需要测试5\*6=30组数据。为避免偶然性，对于每组数据，均在相同的测试环境下测试10次，并取平均值作为最终结果。测试结果列表如下：   |  |  |  |  | | --- | --- | --- | --- | | 组号 | 进程数（np） | 数据范围 | 10次测试平均时间（秒） | | 1 | 1 | 1000 | 0.0000411 | | 2 | 1 | 10000 | 0.0000791 | | 3 | 1 | 100000 | 0.0004915 | | 4 | 1 | 1000000 | 0.0056681‬ | | 5 | 1 | 10000000 | 0.075265‬ | | 6 | 1 | 100000000 | 0.9681718‬ | | 7 | 2 | 1000 | 0.0000594 | | 8 | 2 | 10000 | 0.0001057‬ | | 9 | 2 | 100000 | 0.0003701‬ | | 10 | 2 | 1000000 | 0.0032979‬ | | 11 | 2 | 10000000 | 0.0469701‬ | | 12 | 2 | 100000000 | 0.6943679‬ | | 13 | 4 | 1000 | 0.0003699‬ | | 14 | 4 | 10000 | 0.0003012‬ | | 15 | 4 | 100000 | 0.0005021‬ | | 16 | 4 | 1000000 | 0.0026812‬ | | 17 | 4 | 10000000 | 0.0289978‬ | | 18 | 4 | 100000000 | 0.5885879‬ | | 19 | 8 | 1000 | 0.0011759‬ | | 20 | 8 | 10000 | 0.0010012‬ | | 21 | 8 | 100000 | 0.0015086‬ | | 22 | 8 | 1000000 | 0.0031416 | | 23 | 8 | 10000000 | 0.0265533‬ | | 24 | 8 | 100000000 | 0.5455628‬ | | 25 | 16 | 1000 | 0.0031812 | | 26 | 16 | 10000 | 0.0039769‬‬ | | 27 | 16 | 100000 | 0.0044567 | | 28 | 16 | 1000000 | 0.0069956‬ | | 29 | 16 | 10000000 | 0.0245085‬ | | 30 | 16 | 100000000 | 0.4910188 |   表二：modification1程序测试结果表  根据表二结果，可以绘制modification1程序在不同进程数与不同数据规模下的并行性能曲线图（篇幅所限，图与结果详细分析请参见：八、实验数据及结果分析 部分）  **优化思路2 去掉广播通信：**  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，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。  假设每个任务除了分配到n/p个整数外,还拥有一个单独的数组，用于包含整数3,5,7,…,。 在寻找3~n的素数之前， 每个任务可以先用串行算法计算出3~的素数。一旦这一步骤完成， 每个任务就拥有了所有的数组，其中包含了所有3~的素数。在这些任务可以在消除了广播步骤的情况下，对整个数组进行筛选。  基于以上思想，修改第一次优化后的代码modification1.c（注，由于代码改动处较多，此处不一一列举出，优化后的完整代码将在实验报告后的附录中给出），记经过第二次优化，进一步去除广播通信后的代码为modification2.c。  接下来对modification2程序在不同的数据规模和不同进程配置下的性能进行测试。在本次测试中，测试进程数分别为1、2、4、8和16，选取的数据规模有：1000、10000、100000、1000000、10000000和100000000。也就是说，一共需要测试5\*6=30组数据。为避免偶然性，对于每组数据，均在相同的测试环境下测试10次，并取平均值作为最终结果。测试结果列表如下：   |  |  |  |  | | --- | --- | --- | --- | | 组号 | 进程数（np） | 数据范围 | 10次测试平均时间（秒） | | 1 | 1 | 1000 | 0.0000315 | | 2 | 1 | 10000 | 0.0000665 | | 3 | 1 | 100000 | 0.0004293 | | 4 | 1 | 1000000 | 0.0055843 | | 5 | 1 | 10000000 | 0.0744138‬ | | 6 | 1 | 100000000 | 0.9665156‬ | | 7 | 2 | 1000 | 0.0000416 | | 8 | 2 | 10000 | 0.0000659 | | 9 | 2 | 100000 | 0.0002724 | | 10 | 2 | 1000000 | 0.0026030 | | 11 | 2 | 10000000 | 0.0447923‬ | | 12 | 2 | 100000000 | 0.6876903‬ | | 13 | 4 | 1000 | 0.0002676 | | 14 | 4 | 10000 | 0.0002753 | | 15 | 4 | 100000 | 0.0003567 | | 16 | 4 | 1000000 | 0.0028537 | | 17 | 4 | 10000000 | 0.0275986‬ | | 18 | 4 | 100000000 | 0.5763973‬ | | 19 | 8 | 1000 | 0.0009297 | | 20 | 8 | 10000 | 0.0007689 | | 21 | 8 | 100000 | 0.0010852 | | 22 | 8 | 1000000 | 0.0028321 | | 23 | 8 | 10000000 | 0.0198515‬ | | 24 | 8 | 100000000 | 0.5180015‬ | | 25 | 16 | 1000 | 0.0030115 | | 26 | 16 | 10000 | 0.0034173 | | 27 | 16 | 100000 | 0.0034758 | | 28 | 16 | 1000000 | 0.0045161 | | 29 | 16 | 10000000 | 0.0180183 | | 30 | 16 | 100000000 | 0.4042959‬ |   表三：modification2程序测试结果表  根据表三结果，可以绘制modification2程序在不同进程数与不同数据规模下的并行性能曲线图（篇幅所限，图与结果详细分析请参见：八、实验数据及结果分析 部分）  **优化思路3分块筛选，提高cache命中率：**  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- sqrt(n)中的素数进行标记筛选，从而提高cache命中率。  并行筛法算法的大部分执行时间都用在对一个非常巨大的数组的分散的元素进行标记上了，造成了很差的Cache命中率。我们把算法的核心内容看成2个循环，外层循环为3~的素数之间迭代， 内层循环在属于该进程的3~n的整数之间迭代。如果我们把内外层循环交换一下， 就可以改进程序的Cache命中率。我们可以将数组的一部分放入Cache, 标记其中所有小于的素数的倍数，然后再读入数组的下一个部分。  在Windows系统中，可以通过任务管理器的“性能”选项查看机器的Cache缓存信息，在实验所用机器中，缓存信息如下：    由于测试中可能会涉及到比较大的数据范围，为保证统一，也为了方便进行比较，统一选取L3级缓存。由于在系统中，一个int类型数据占4B，因此，L3缓存共可以容纳4\*1024\*1024/4=1048576个int类型数据，因此，可以将1048576作为划分块的大小。  基于以上思想，再次修改第二次优化后的代码modification2.c（注，由于代码改动处较多，此处不一一列举出，优化后的完整代码将在实验报告后的附录中给出），记经过第三次优化，重新组织循环并提升Cache命中率后的代码为modification3.c。  接下来对modification3程序在不同的数据规模和不同进程配置下的性能进行测试。在本次测试中，测试进程数分别为1、2、4、8和16，选取的数据规模有：1000、10000、100000、1000000、10000000和100000000。也就是说，一共需要测试5\*6=30组数据。为避免偶然性，对于每组数据，均在相同的测试环境下测试10次，并取平均值作为最终结果。测试结果列表如下：   |  |  |  |  | | --- | --- | --- | --- | | 组号 | 进程数（np） | 数据范围 | 10次测试平均时间（秒） | | 1 | 1 | 1000 | 0.0000299 | | 2 | 1 | 10000 | 0.0000713 | | 3 | 1 | 100000 | 0.0004567 | | 4 | 1 | 1000000 | 0.0056906 | | 5 | 1 | 10000000 | 0.0726635‬ | | 6 | 1 | 100000000 | 0.8029341‬ | | 7 | 2 | 1000 | 0.0000384 | | 8 | 2 | 10000 | 0.0000649 | | 9 | 2 | 100000 | 0.0003198 | | 10 | 2 | 1000000 | 0.0027058‬ | | 11 | 2 | 10000000 | 0.0398478‬ | | 12 | 2 | 100000000 | 0.4318809‬ | | 13 | 4 | 1000 | 0.0002306 | | 14 | 4 | 10000 | 0.0002322 | | 15 | 4 | 100000 | 0.0004712 | | 16 | 4 | 1000000 | 0.0022883 | | 17 | 4 | 10000000 | 0.0264559‬ | | 18 | 4 | 100000000 | 0.2665422‬ | | 19 | 8 | 1000 | 0.0008931 | | 20 | 8 | 10000 | 0.0006660‬ | | 21 | 8 | 100000 | 0.0012072 | | 22 | 8 | 1000000 | 0.002158‬0 | | 23 | 8 | 10000000 | 0.0181239‬ | | 24 | 8 | 100000000 | 0.2044096‬ | | 25 | 16 | 1000 | 0.0028701‬ | | 26 | 16 | 10000 | 0.0025690‬ | | 27 | 16 | 100000 | 0.0026986‬ | | 28 | 16 | 1000000 | 0.0037125‬ | | 29 | 16 | 10000000 | 0.0160945‬ | | 30 | 16 | 100000000 | 0.1731379‬ |   表四：modification3程序测试结果表  根据表四结果，可以绘制modification3程序在不同进程数与不同数据规模下的并行性能曲线图（篇幅所限，图与结果详细分析请参见：八、实验数据及结果分析 部分） | |
| **八、实验数据及结果分析：**  根据以上四个表格的实验数据结果，可以绘制4个程序在不同数据规模以及不同进程数量下的运行性能曲线。分别固定数据规模，在同一张图中作出4个程序的折线图如下：  通过分析以上图表数据，可以得出，在相同的数据规模和进程数量条件下，从总体上看，每一次优化后的程序性能都要优于之前的程序，也就是说从总体的性能上看：sieve1<modification1<modification2<modification3。这是符合预期的实验结果，该结果也证明了以上所进行的每一次优化是行之有效的，其作用也直接体现在了程序平均运行时间的减少上。因此，至此实验基本取得成功。  进一步分析以上图表数据，还可以得出更多的结论与信息。可以注意到当数据的规模不太大时（如1000000以内），无论是原程序还是优化后的任一程序，其平均运行时间都随着进程数增多而增加。也就是说，在数据规模较小且一定时，增加进程数未必能提升性能加速计算，相反，反而还可能使程序运行的时间延长。出现这种情况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（这种开销体现在系统资源上和时间消耗上），在任务量比较小时，多创建进程会使得系统主要的开销都花在这些进程上，反而是不利于计算的。也就是说，此时，多进程并行计算带来的收益微乎其微，远远不及系统资源的开销，造成了系统资源的浪费。所以，在计算数据规模比较小时，并不适宜开启太多进程计算。  在计算数据规模足够大时，优化算法以及多进程并行计算带来的性能提升就显得非常明显。比较最初始版本sieve1程序与进行了第一次优化去掉偶数后的modification1程序，可以发现筛除偶数后的程序运行时间几乎比原来减少了一半。因为去除偶数实际上直接让数据规模减少了一半，也就是程序的计算量直接减少了一半，在数据规模足够大可以使程序性能充分发挥时，带来的性能提升就非常可观。  进一步比较第一次优化后的程序modification1与第二次优化去除广播通信的程序modification2，这组比较相对于前面一组的比较来说，性能的提升不是那么明显，二者的性能差距并不是很大，在计算数据规模不是很大时或是进程数不多时性能很接近。从原理上来理解，进程间的广播通信时间消耗相比于计算时间消耗并不大，本身可优化和性能提升的空间就比较有限，另外，由于在测试时指定的进程数都比较少，因此不能完全发挥其作用。可能当数据规模足够大并且进程数足够多时，这种优化才能体现较为明显的性能提升。  最后比较以上程序与最终优化的程序modification3，可以看出，通过提升Cache命中率来对程序进行优化同样需要计算数据规模足够大才能发挥明显的作用。从原理上来理解，由于机器的Cache块本身能容纳一定量的数，如果数据量明显小于Cache块容纳量，那么Cache块就可以放得下所有的数，也就不存在所谓的Cache miss情况，这时也就相当于没有提升。只有数据量足够大时，才需要考虑数据在Cache块的存放，此时才需要对数据进行分块，从而通过减少Cache miss，提高Cache命中率来提升性能。  总结来说，优化算法对程序性能的提升是行之有效的，因此，针对不同任务，应该设计和采用有效的算法提升性能。但是，程序的性能也受到其它方面的制约，应该综合考虑数据规模以及机器和系统本身的性能和资源量来决定，这往往需要经过尝试。 | |
| **九、实验结论：**  1.针对不同的具体任务，应该先分析问题，并设计和采用高效的算法，这有利于程序性能的提升。好的算法对程序性能的提升是非常明显的，能达到事半功倍的效果。  2.并不是越多进程并行计算就越好，不应该盲目应用多进程。对于计算数据规模较小的任务，多进程反而会造成系统资源的浪费，也影响程序运行时间和效率，可能事倍功半。  3.在不同情况、不同环境下，同样的程序性能可能有明显差异，应该综合考虑任务数据规模、机器软硬件、系统当前状态、性能和资源情况来决定应该如何应用并行程序，需要经验和尝试。 | |
| **十、总结及心得体会：**  分布式并行计算让计算机能够处理很大规模的问题，在现如今的应用是十分广泛的。通过本次实验，结合课程所学知识，我第一次接触、认识、学习了分布式并行计算程序，也进行了并行计算程序的编写、修改、调试和应用，拓宽了知识面的同时，锻炼了编程能力与动手能力。通过实践，锻炼了解决问题、分析数据、提炼结论的能力，获益匪浅。 | |
| **十一、\*对本实验过程及方法、手段的改进建议：**  暂无 | |
| **报告评分：**  **指导教师签字：** | |

附录：实验程序源代码

1. sieve1.c（已修复已知Bug）

#include "mpi.h"

#include <math.h>

#include <stdio.h>

#define MIN(a,b) ((a)<(b)?(a):(b))

int main (int argc, char \*argv[])

{

int count; /\* Local prime count \*/

double elapsed\_time; /\* Parallel execution time \*/

int first; /\* Index of first multiple \*/

int global\_count; /\* Global prime count \*/

int high\_value; /\* Highest value on this proc \*/

int i;

int id; /\* Process ID number \*/

int index; /\* Index of current prime \*/

int low\_value; /\* Lowest value on this proc \*/

char \*marked; /\* Portion of 2,...,'n' \*/

int n; /\* Sieving from 2, ..., 'n' \*/

int p; /\* Number of processes \*/

int proc0\_size; /\* Size of proc 0's subarray \*/

int prime; /\* Current prime \*/

int size; /\* Elements in 'marked' \*/

MPI\_Init (&argc, &argv);

/\* Start the timer \*/

MPI\_Comm\_rank (MPI\_COMM\_WORLD, &id);

MPI\_Comm\_size (MPI\_COMM\_WORLD, &p);

MPI\_Barrier(MPI\_COMM\_WORLD);

elapsed\_time = -MPI\_Wtime();

if (argc != 2) {

if (!id) printf ("Command line: %s <m>\n", argv[0]);

MPI\_Finalize();

exit (1);

}

n = atoi(argv[1]);

/\* Figure out this process's share of the array, as

well as the integers represented by the first and

last array elements \*/

low\_value = 2 + id\*(n-1)/p;

high\_value = 1 + (id+1)\*(n-1)/p;

size = high\_value - low\_value + 1;

/\* Bail out if all the primes used for sieving are

not all held by process 0 \*/

proc0\_size = (n-1)/p;

if ((2 + proc0\_size) < (int) sqrt((double) n)) {

if (!id) printf ("Too many processes\n");

MPI\_Finalize();

exit (1);

}

/\* Allocate this process's share of the array. \*/

marked = (char \*) calloc(size, sizeof(char));

if (marked == NULL) {

printf ("Cannot allocate enough memory\n");

MPI\_Finalize();

exit (1);

}

if (!id) index = 0;

prime = 2;

do {

if (prime \* prime > low\_value)

first = prime \* prime - low\_value;

else {

if (!(low\_value % prime)) first = 0;

else first = prime - (low\_value % prime);

}

for (i = first; i < size; i += prime) marked[i] = 1;

if (!id) {

while (marked[++index]);

prime = index + 2;

}

if (p > 1) MPI\_Bcast (&prime, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

} while (prime \* prime <= n);

count = 0;

for (i = 0; i < size; i++)

if (!marked[i]) count++;

if (p > 1) MPI\_Reduce (&count, &global\_count, 1, MPI\_INT, MPI\_SUM,

0, MPI\_COMM\_WORLD);

else{

global\_count = count;

}

/\* Stop the timer \*/

elapsed\_time += MPI\_Wtime();

/\* Print the results \*/

if (!id) {

printf ("There are %d primes less than or equal to %d\n",

global\_count, n);

printf ("SIEVE (%d) %10.6f\n", p, elapsed\_time);

}

MPI\_Finalize ();

return 0;

}

1. modification1.c

/\* Modification 1: 筛除除2以外的偶数 \*/

#include <mpi.h>

#include <math.h>

#include <stdio.h>

#define MIN(a,b) ((a)<(b) ? (a) : (b))

#define BLOCK\_LOW(id,p,n) ((id)\*(n)/(p))

#define BLOCK\_HIGH(id,p,n) (BLOCK\_LOW((id)+1,p,n)-1)

#define BLOCK\_SIZE(id,p,n) (BLOCK\_HIGH(id,p,n)-BLOCK\_LOW(id,p,n)+1)

int main(int argc, char \*argv[])

{

int count; /\* Local prime count \*/

double elapsed\_time; /\* Parallel execution time \*/

int first; /\* Index of first multiple \*/

int global\_count; /\* Global prime count \*/

int high\_value; /\* Highest value on this proc \*/

int i;

int id; /\* Process ID number \*/

int index; /\* Index of current prime \*/

int low\_value; /\* Lowest value on this proc \*/

char \*marked; /\* Portion of 2,...,'n' \*/

int n; /\* Sieving from 2, ..., 'n' \*/

int p; /\* Number of processes \*/

int proc0\_size; /\* Size of proc 0's subarray \*/

int prime; /\* Current prime \*/

int size; /\* Elements in 'marked' \*/

int m;

int loc;

MPI\_Init (&argc, &argv);

/\* Start the timer \*/

MPI\_Comm\_rank (MPI\_COMM\_WORLD, &id);

MPI\_Comm\_size (MPI\_COMM\_WORLD, &p);

MPI\_Barrier(MPI\_COMM\_WORLD);

elapsed\_time = -MPI\_Wtime();

if (argc != 2) {

if (!id) printf ("Command line: %s <m>\n", argv[0]);

MPI\_Finalize();

exit (1);

}

n = atoi(argv[1]);

m = (n-3)/2 + 1;

/\* Figure out this process's share of the array, as

well as the integers represented by the first and

last array elements \*/

low\_value = 2 \* BLOCK\_LOW(id, p, m) + 3;

high\_value = 2 \* BLOCK\_HIGH(id, p, m) + 3;

size = BLOCK\_SIZE(id, p, m);

/\* Bail out if all the primes used for sieving are

not all held by process 0 \*/

proc0\_size = m/p;

if ((2\*(proc0\_size-1) + 3) < (int) sqrt((double) n)) {

if (!id) printf("Too many processes\n");

MPI\_Finalize();

exit(1);

}

/\* Allocate this process' share of the array \*/

marked = (char \*) calloc(size, sizeof(char));

if (marked == NULL) {

printf("Cannot allocate enough memory\n");

MPI\_Finalize();

exit(1);

}

if (!id) index = 0;

prime = 3;

do {

if (prime\*prime > low\_value)

first = (prime\*prime-3)/2 - (low\_value-3)/2;

else {

loc = low\_value % prime;

if (!loc) first = 0;

else {

first = prime - loc;

if (!((low\_value+first)%2))

first = (first+prime)/2;

else first /= 2;

}

}

for (i=first; i <size; i+=prime)

marked[i] = 1;

if (!id) {

while (marked[++index]);

prime = 2\*index + 3;

}

MPI\_Bcast(&prime, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

} while (prime\*prime <= n);

count = 0;

for (i=0; i<size; i++)

if (!marked[i]) count++;

MPI\_Reduce(&count, &global\_count, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

/\* Stop the timer \*/

elapsed\_time += MPI\_Wtime();

/\* Print the results \*/

if (!id) {

printf ("There are %d primes less than or equal to %d\n",

global\_count+1, n);

printf ("SIEVE (%d) %10.6f\n", p, elapsed\_time);

}

MPI\_Finalize ();

return 0;

}

1. modification2.c

/\* Modification 1: 筛除除2以外的偶数 \*/

/\* Modification 2: 去掉广播通信 \*/

#include <mpi.h>

#include <math.h>

#include <stdio.h>

#define MIN(a,b) ((a)<(b) ? (a) : (b))

#define BLOCK\_LOW(id,p,n) ((id)\*(n)/(p))

#define BLOCK\_HIGH(id,p,n) (BLOCK\_LOW((id)+1,p,n)-1)

#define BLOCK\_SIZE(id,p,n) (BLOCK\_HIGH(id,p,n)-BLOCK\_LOW(id,p,n)+1)

int main(int argc, char \*argv[])

{

int count; /\* Local prime count \*/

double elapsed\_time; /\* Parallel execution time \*/

int first; /\* Index of first multiple \*/

int global\_count; /\* Global prime count \*/

int high\_value; /\* Highest value on this proc \*/

int i;

int id; /\* Process ID number \*/

int index; /\* Index of current prime \*/

int low\_value; /\* Lowest value on this proc \*/

char \*marked; /\* Portion of 2,...,'n' \*/

int n; /\* Sieving from 2, ..., 'n' \*/

int p; /\* Number of processes \*/

int proc0\_size; /\* Size of proc 0's subarray \*/

int prime; /\* Current prime \*/

int size; /\* Elements in 'marked' \*/

int m;

int loc;

char \*primes;

int primes\_size;

MPI\_Init(&argc, &argv);

/\* Start the timer \*/

MPI\_Comm\_rank (MPI\_COMM\_WORLD, &id);

MPI\_Comm\_size (MPI\_COMM\_WORLD, &p);

MPI\_Barrier(MPI\_COMM\_WORLD);

elapsed\_time = -MPI\_Wtime();

if (argc != 2) {

if (!id) printf("Command line: %s <n>\n", argv[0]);

MPI\_Finalize();

exit(1);

}

n = atoi(argv[1]);

m = (n-3)/2 + 1;

/\* Figure out this process's share of the array, as

well as the integers represented by the first and

last array elements \*/

low\_value = 2 \* BLOCK\_LOW(id, p, m) + 3;

high\_value = 2 \* BLOCK\_HIGH(id, p, m) + 3;

size = BLOCK\_SIZE(id, p, m);

/\* Allocate this process' share of the array \*/

marked = (char \*) calloc(size, sizeof(char));

if (marked == NULL) {

printf("Cannot allocate enough memory\n");

MPI\_Finalize();

exit(1);

}

primes\_size = (sqrt(n) - 3)/2 + 1;

primes = (char \*) malloc(primes\_size);

if (primes == NULL) {

printf("Cannot allocate enough memory\n");

free(marked);

MPI\_Finalize();

exit(1);

}

for (i=0; i<primes\_size; i++) primes[i] = 0;

index = 0;

prime = 3;

do {

for (i = (prime\*prime-3)/2; i < primes\_size; i += prime)

primes[i] = 1;

while (primes[++index]);

prime = 2\*index + 3;

} while (prime\*prime <= sqrt(n));

index = 0;

prime = 3;

do {

if (prime\*prime > low\_value)

first = (prime\*prime-3)/2 - (low\_value-3)/2;

else {

loc = low\_value % prime;

if (!loc) first = 0;

else {

first = prime - loc;

if (!((low\_value+first)%2))

first = (first+prime)/2;

else first /= 2;

}

}

for (i=first; i <size; i+=prime)

marked[i] = 1;

while (primes[++index]);

prime = 2\*index + 3;

} while (prime\*prime <= n);

count = 0;

for (i=0; i<size; i++)

if (!marked[i]) count++;

MPI\_Reduce(&count, &global\_count, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

/\* Stop the timer \*/

elapsed\_time += MPI\_Wtime();

/\* Print the results \*/

if (!id) {

printf ("There are %d primes less than or equal to %d\n",

global\_count+1, n);

printf ("SIEVE (%d) %10.6f\n", p, elapsed\_time);

}

MPI\_Finalize ();

return 0;

}

1. modification3.c

/\* Modification 1: 筛除除2以外的偶数 \*/

/\* Modification 2: 去掉广播通信 \*/

/\* Modification 3: 重新组织循环，提高cache块命中率 \*/

#include <mpi.h>

#include <math.h>

#include <stdio.h>

#define MIN(a,b) ((a)<(b) ? (a) : (b))

#define BLOCK\_LOW(id,p,n) ((id)\*(n)/(p))

#define BLOCK\_HIGH(id,p,n) (BLOCK\_LOW((id)+1,p,n)-1)

#define BLOCK\_SIZE(id,p,n) (BLOCK\_HIGH(id,p,n)-BLOCK\_LOW(id,p,n)+1)

int main(int argc, char \*argv[])

{

int count; /\* Local prime count \*/

double elapsed\_time; /\* Parallel execution time \*/

int first; /\* Index of first multiple \*/

int global\_count; /\* Global prime count \*/

int high\_value; /\* Highest value on this proc \*/

int i;

int id; /\* Process ID number \*/

int index; /\* Index of current prime \*/

int low\_value; /\* Lowest value on this proc \*/

char \*marked; /\* Portion of 2,...,'n' \*/

int n; /\* Sieving from 2, ..., 'n' \*/

int p; /\* Number of processes \*/

int proc0\_size; /\* Size of proc 0's subarray \*/

int prime; /\* Current prime \*/

int size; /\* Elements in 'marked' \*/

int m;

int loc;

char \*primes;

int primes\_size;

int lv;

int sec;

int chunk;

MPI\_Init(&argc, &argv);

/\* Start the timer \*/

MPI\_Comm\_rank (MPI\_COMM\_WORLD, &id);

MPI\_Comm\_size (MPI\_COMM\_WORLD, &p);

MPI\_Barrier(MPI\_COMM\_WORLD);

elapsed\_time = -MPI\_Wtime();

if (argc != 3) {

if (!id) printf("Command line: %s <n> <chunk>\n", argv[0]);

MPI\_Finalize();

exit(1);

}

n = atoi(argv[1]);

m = (n-3)/2 + 1;

chunk = atoi(argv[2]);

/\* Figure out this process's share of the array, as

well as the integers represented by the first and

last array elements \*/

low\_value = 2 \* BLOCK\_LOW(id, p, m) + 3;

high\_value = 2 \* BLOCK\_HIGH(id, p, m) + 3;

size = BLOCK\_SIZE(id, p, m);

/\* Allocate this process' share of the array \*/

marked = (char \*) calloc(size, sizeof(char));

if (marked == NULL) {

printf("Cannot allocate enough memory\n");

MPI\_Finalize();

exit(1);

}

primes\_size = (sqrt(n) - 3)/2 + 1;

primes = (char \*) malloc(primes\_size);

if (primes == NULL) {

printf("Cannot allocate enough memory\n");

free(marked);

MPI\_Finalize();

exit(1);

}

for (i=0; i<primes\_size; i++) primes[i] = 0;

index = 0;

prime = 3;

do {

for (i = (prime\*prime-3)/2; i < primes\_size; i += prime)

primes[i] = 1;

while (primes[++index]);

prime = 2\*index + 3;

} while (prime\*prime <= sqrt(n));

for (sec = 0; sec < size; sec += chunk) {

index = 0;

prime = 3;

lv = 2\*((low\_value-3)/2+sec)+3;

do {

if (prime\*prime > lv)

first = (prime\*prime-3)/2 - (lv-3)/2;

else {

loc = lv % prime;

if (!loc) first = 0;

else {

first = prime - loc;

if (!((lv+first)%2))

first = (first+prime)/2;

else first /= 2;

}

}

for (i = first+sec; i < first+sec+chunk && i < size; i += prime)

marked[i] = 1;

while (primes[++index]);

prime = 2\*index + 3;

} while (prime\*prime <= n);

}

count = 0;

for (i=0; i<size; i++)

if (!marked[i]) count++;

MPI\_Reduce(&count, &global\_count, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

/\* Stop the timer \*/

elapsed\_time += MPI\_Wtime();

/\* Print the results \*/

if (!id) {

printf ("There are %d primes less than or equal to %d\n",

global\_count+1, n);

printf ("SIEVE (%d) %10.6f\n", p, elapsed\_time);

}

MPI\_Finalize ();

return 0;

}