**实验1**

学号 姓名

**实验题目:**

**用openMP与MPI分别实现pi迭代算法和范围内质数个数统计算法**

**实验环境(操作系统,编译器,硬件配置等):**

**本次实验运行在NVIDIA DGX（本次实验只使用了CPU：Intel Xeon E5-2698 v4 @ 2.2GHz, 20 cores/40 threads）硬件平台的ubuntu 16.04 RTS系统上，采用了支持C++11标准的gcc 5.4编译器来进行编译工作和cmake 3.9来辅助编译（只是为了程序写起来比较简单）。**

**算法设计与分析(写出解题思路和实现步骤)**

**计算pi时：通过不同迭代次数的4 / (1 + x ^ 2)在[0, 1]上的定积分计算来逼近pi**

**统计质数个数时：基本思路基于筛法：先用相对短的时间单线程计算出sqrt(n)规模的素数表，然后用这个表在剩下来的所有数中进行筛选（把所有的非质数筛去）。计算素数表与筛选过程皆使用了位运算来节省空间以及加速代码运行。**

**核心代码(写出算法实现的关键部分,如核心的循环等)**

**具体的代码都可以在https://github.com/thoh-testarossa/pc-exp上找到**

**Pi：**

**for**(i = tid; i < STEP\_NUM; i++)  
{  
 x = (i + 0.5) \* STEP\_LENGTH;  
 sum[tid] += 4.0 / (1.0 + x \* x);  
}

**Prime：**

**char** countZero(**char** x)  
{  
 x = (x & 0x55) + ((x >> 1) & 0x55);  
 x = (x & 0x33) + ((x >> 2) & 0x33);  
 x = (x & 0xf) + ((x >> 4) & 0xf);  
 **return** 8 - x;  
}

**int** judgeAllFromPrimeTable(**int** from, **int** to, **int** \*primeTable, **int** pCount)  
{  
 **unsigned char** \*notPrimeBitArray = **new unsigned char** [(to - from) >> 4];  
 **for**(**int** i = 0; i < (to - from) >> 4; i++) notPrimeBitArray[i] = 0;  
  
 notPrimeBitArray[0] |= 1;  
  
 **for**(**int** i = 0; i < pCount && primeTable[i] \* primeTable[i] < to; i++)  
 {  
 **int** startNum = from - (from % primeTable[i]) + primeTable[i];  
 **if**((startNum & 1) == 0) startNum += primeTable[i];  
 **if**(startNum == primeTable[i]) startNum += 2 \* primeTable[i];  
  
 **for**(**int** j = startNum; j < to; j += 2 \* primeTable[i])  
 {  
 **int** index = j - from;  
 notPrimeBitArray[index >> 4] |= 1 << (((index & 0xf) >> 1));  
 }  
 }  
 **int** ans = 0;  
 **for**(**int** i = 0; i < (to - from) >> 4; i++)  
 ans += countZero(notPrimeBitArray[i]);  
 **return** ans;  
}

**实验结果:**

**Pi\_MPI运行时间**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 规模\线程 | 1 | 2 | 4 | 8 | 16 | 32 |
| 2.5 \* 10 ^ 8 | **0.986** | **0.528** | **0.298** | **0.249** | **0.216** | **0.302** |
| 5 \* 10 ^ 8 | **1.943** | **1.094** | **0.569** | **0.444** | **0.339** | **0.357** |
| 1 \* 10 ^ 9 | **3.814** | **2.000** | **1.025** | **0.661** | **0.530** | **0.502** |
| 2 \* 10 ^ 9 | **7.570** | **3.915** | **2.197** | **1.110** | **0.689** | **0.768** |

**Pi\_MPI加速比**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 规模\线程 | 1 | 2 | 4 | 8 | 16 | 32 |
| 2.5 \* 10 ^ 8 | **1** | **1.867** | **3.309** | **3.96** | **4.565** | **3.265** |
| 5 \* 10 ^ 8 | **1** | **1.776** | **3.415** | **4.376** | **5.732** | **5.443** |
| 1 \* 10 ^ 9 | **1** | **1.907** | **3.721** | **5.77** | **7.196** | **7.598** |
| 2 \* 10 ^ 9 | **1** | **1.934** | **3.446** | **6.82** | **10.987** | **9.857** |

**Pi\_openMP运行时间**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 规模\线程 | 1 | 2 | 4 | 8 | 16 | 32 |
| 2.5 \* 10 ^ 8 | **1.115** | **0.620** | **0.371** | **0.318** | **0.283** | **0.376** |
| 5 \* 10 ^ 8 | **2.148** | **1.232** | **0.664** | **0.530** | **0.416** | **0.435** |
| 1 \* 10 ^ 9 | **4.169** | **2.210** | **1.157** | **0.764** | **0.622** | **0.592** |
| 2 \* 10 ^ 9 | **8.226** | **4.278** | **2.423** | **1.249** | **0.794** | **0.879** |

**Pi\_MPI加速比**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 规模\线程 | 1 | 2 | 4 | 8 | 16 | 32 |
| 2.5 \* 10 ^ 8 | **1** | **1.798** | **3.005** | **3.506** | **3.94** | **2.965** |
| 5 \* 10 ^ 8 | **1** | **1.744** | **3.235** | **4.053** | **5.163** | **4.938** |
| 1 \* 10 ^ 9 | **1** | **1.886** | **3.603** | **5.457** | **6.703** | **7.042** |
| 2 \* 10 ^ 9 | **1** | **1.923** | **3.395** | **6.586** | **10.36** | **9.358** |

**Prime\_MPI运行时间**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 规模\线程 | 1 | 2 | 4 | 8 | 16 | 32 |
| 2.5 \* 10 ^ 8 | **1.576** | **0.826** | **0.441** | **0.270** | **0.233** | **0.378** |
| 5 \* 10 ^ 8 | **3.133** | **1.648** | **0.835** | **0.498** | **0.387** | **0.463** |
| 1 \* 10 ^ 9 | **6.302** | **3.278** | **1.659** | **0.868** | **0.532** | **0.659** |
| 2 \* 10 ^ 9 | **12.858** | **6.710** | **3.580** | **2.066** | **1.093** | **1.016** |

**Prime\_MPI加速比**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 规模\线程 | 1 | 2 | 4 | 8 | 16 | 32 |
| 2.5 \* 10 ^ 8 | **1** | **1.908** | **3.574** | **5.837** | **6.764** | **4.169** |
| 5 \* 10 ^ 8 | **1** | **1.901** | **3.752** | **6.291** | **8.096** | **6.767** |
| 1 \* 10 ^ 9 | **1** | **1.923** | **3.799** | **7.26** | **11.846** | **9.563** |
| 2 \* 10 ^ 9 | **1** | **1.916** | **3.592** | **6.224** | **11.764** | **12.656** |

**Prime\_openMP运行时间**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 规模\线程 | 1 | 2 | 4 | 8 | 16 | 32 |
| 2.5 \* 10 ^ 8 | **1.702** | **0.765** | **0.431** | **0.315** | **0.148** | **0.154** |
| 5 \* 10 ^ 8 | **3.141** | **1.563** | **0.790** | **0.455** | **0.301** | **0.247** |
| 1 \* 10 ^ 9 | **6.407** | **3.145** | **1.586** | **0.843** | **0.505** | **0.458** |
| 2 \* 10 ^ 9 | **12.812** | **6.612** | **3.377** | **1.810** | **1.019** | **0.814** |

**Prime\_openMP加速比**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 规模\线程 | 1 | 2 | 4 | 8 | 16 | 32 |
| 2.5 \* 10 ^ 8 | **1** | **2.225** | **3.949** | **5.403** | **11.5** | **11.052** |
| 5 \* 10 ^ 8 | **1** | **2.01** | **3.976** | **6.903** | **10.435** | **12.717** |
| 1 \* 10 ^ 9 | **1** | **2.037** | **4.04** | **7.6** | **12.687** | **13.989** |
| 2 \* 10 ^ 9 | **1** | **1.938** | **3.794** | **7.078** | **12.573** | **15.74** |

**分析与总结**

**从上面的实验可以看出，我们可以在很短的时间之内完成比实验要求其实多得多的计算任务。并且可以看到，即使是最大规模的数据（2 \* 10 ^ 9），这些计算任务在扩放到32线程时，加速比都显著下降了。并且数据越小，加速比开始出现偏离的线程速越低，从中间能看出来这些并行/线程通信模型的通信同步代价所在。**

**另外，在筛法使用时，在各种地方都能采用位运算来加速运行，并且节约存储空间。**

**备注(可选)**

**限制迭代时间在1s时，其实所有的任务的最大规模时，都能在1s内完成，若以16线程运行**