**《并行计算》上机报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 姓名： | 王章瀚 | | 学号： | PB18111697 | 日期： | 2021.5.15 |
| 上机题目： | MPI并行编程实验 | | | | | |
| 实验环境：  CPU： 8 块 Intel(R) Core(TM) i7-10875H CPU @ 2.30GHz ;  内存: 32GB ;  操作系统：Ubuntu 20.04.1 ;  软件平台：MPICH 3.0.4 ; | | | | | | |
| **一、算法设计与分析：**  题目一:  总体上就是遵从矩形求积分的方法来完成计算.    在 MPI的框架下, 需要做的就是分配给各个进程其相应的计算任务, 然后最后做一个 MPI\_Reduce以MPI\_SUM来收集各个进程计算的部分, 最后直接输出即可.  题目二：  本题目要求使用MPI实现 PSRA 算法进行排序. 该算法的输入是n个元素 A[1..n] 分成 p组, 每组 A[(i-1)n/p+1..in/p], i=1~p, 而排序主要分为以下几步:   1. 均匀划分: 将 n 个元素A[1..n]均匀划分成p段, 每个处理A[(i-1)n/p+1..in/p] 2. 局部排序: 调用串行排序算法对A[(i-1)n/p+1..in/p]排序 3. 选取样本: 从其有序子序列A[(i-1)n/p+1..in/p]中选取p个元素 4. 样本排序: 用一台处理器对 个样本元素进行串行排序 5. 选取主元: 用一台处理器从排好序的样本序列中选取p-1个主元, 并播送给其他 6. 主元划分:按主元将有序段A[(i-1)n/p+1..in/p] 划分成 p 段 7. 全局交换: 各处理器将其有序段按段号交换到对应的处理器中 8. 归并排序: 各处理器对接收到的元素进行归并排序   下面是一个上述流程的图解(摘自老师徐云的PPT):    由于用到了局部排序, 局部有 n/p 个数据, 其他部分的时间复杂度都小于此, 因此本算法的时间复杂度是的.  **二、核心代码：**  题目一:    关于矩形法求的方法, 前面已经讲过. 这里主要是让各个进程计算 my\_rank+i\*group\_size 的 sum 结果. 最后做一个 MPI\_Reduce 进行 reduce 求和即可.  值得注意的是, 应当只允许一个进程输出最终结果: 在本代码中即为 my\_rank 为0的进程进行输出的任务.  题目二：  首先当然要有一个 MPI 初始化的代码:     1. 均匀划分   这里要计算各个进程的数据量, 然后调用MPI\_Scatterv从进程0把各段数据传播给其他进程:     1. 局部排序   调用C++算法库的串行排序算法     1. 选取样本   根据各线程长度选择抽样间距，并进行抽样     1. 样本排序   调用MPI\_Gather收集各个进程的sample结果, 然后做一个简单排序.     1. 选择主元      1. 主元划分   选择主元的第一步是寻找交换前各个段的划分索引号 my\_seg\_idx    而后把这个计算结果散播给各个进程, 让各个进程知道他们将会接收到怎么样的数据大小.     1. 全局交换   依据前面划分结果进行全局交换     1. 归并排序   值得一提的是这里有一个多路归并排序, 我这里利用的是各个线程给一个优先队列, 保持优先队列至多只有 p 个元素, 这些元素都是待处理的子段数组首, 其思想与二路归并排序几乎一致.    与 OpenMP 不同的是, 这里还需要最后将所有进程的计算结果 Gather 到一个进程上, 以提供和串行程序一致的API.    最后会有一个与C++算法库比较结果的操作, 并输出时间与加速比:    **三、结果与分析：**  题目一:  这个算法比较简单, 为了能体现时间差异, 使用1000000000步的迭代, 编译参数为: mpicxx pi.cpp -o pi -O3.计时, 使用 time 命令计时, 注释掉 printf 以得到实际计算时间.  测试结果如下表所示, 其中各个时间数据均为5次测试取平均值的结果:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 进程数 | 1 | 2 | 4 | 8 | 16 | | 耗时(ns) | 1002967915 | 522141233 | 270407752 | 198152524 | 145617106 | | 加速比 | 1 | 1.92 | 3.71 | 5.06 | 6.89 |   可以看出来, 随着进程数的增大, 耗时也会减小. 当进程数为 1到4左右的时候, 耗时能够和进程数比较成比例地缩小, 但当进程数更大的时候, 耗时减小的程度并不大. 下面是该分析的示意图:    题目二：  首先考虑 4 个进程的运行情况, 单位是ns:   |  |  |  |  | | --- | --- | --- | --- | | 数据量 | std::sort运行时间(ns) | PSRS运行时间(ns) | 加速比 | | 1e2 | 1909 | 247480 | 0.007714 | | 1e3 | 26841 | 249792 | 0.107453 | | 1e4 | 347211 | 432954 | 0.801958 | | 1e5 | 4460476 | 2168342 | 2.057091 | | 1e6 | 53265060 | 17889299 | 2.977482 | | 1e7 | 600714131 | 194817606 | 3.083469 | | 1e8 | 7943134440 | 2270424459 | 3.498524 |   在图中反映如下:    可见随着数据量不断增大, PSRS算法相较于 std::sort的加速比逐渐趋近于理论值4.  再考虑改变线程数的情况, 固定数据量为1e8, 取线程数分别为1,2,4,8,16, 则有如下结果:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 进程数 | 1 | 2 | 4 | 8 | 16 | | PSRS运行时间(ns) | 6919596034 | 4046962291 | 2223212791 | 1398991919 | 1268076705 | | 加速比 | 0.903078 | 1.663439 | 3.016509 | 5.449541 | 8.764564 |     由此可见, 虽然更多进程能够带来更大的加速比, 但是并不总是和进程数成正比, 这是由于多进程及其通信自身的开销及数据集大小造成的. 因此实际应用中我们需要根据具体的需求来分析到底需要多少个线程.  **四、备注（\* 可选）：**  有可能影响结论的因素：  MPI 的通信方式  MPI的 CPU 调度方案可能影响结果  算法的具体实现也可能影响结果(如PSRS算法最后一步的归并排序有的同学使用了快速排序替代, 理论上会慢一点, 因为最后一步的时候是局部有序的 , 但也有效果相近的. )  CPU 核心是物理核还是逻辑核可能也有一定影响. | | | | | | |
| **总结：本次实验通过学习老师给出的计算 π的MPI实现方法的例子, 学习了如何使用MPI实现一个并行算法, 并实现了一个性能还算不错的 PSRS算法. 同时也从 scalability 等多个维度测试了本算法的性能, 让本人对MPI编程及一些并行算法有了更深刻的认识.** | | | | | | |
| 附录（源代码） | | 算法源代码（C/C++/JAVA描述）  这里就不放计算π的算法了, 毕竟已经给出了. 以下是PSRS的源代码: | | | | |