**《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： | OpenMP并行编程实验 | | | | | |
| 实验环境：  CPU：i7-10875H x 16; 内存: 32GB ;  操作系统：Ubuntu 20.04.1 LTS ;软件平台：gcc/g++ 9.3.0 ; | | | | | | |
| **一、算法设计与分析：**  题目一:  总体上四个算法的流程都是差不多的, 即利用矩形法求 π 值, 其公式如下:    而各个算法的主要区别如下:  算法一: 每个进程处理相隔为 NUM\_THREAD 的数据, 最后串行地将每个线程得到的 sum 求和.  算法二: 利用 #pragma omp for 自动划分for循环, 最后串行地将每个线程得到的 sum 求和.  算法三: 利用 #pragma omp critical 让各个线程竞争关键区然后再直接加入 sum, 避免串行求和.  算法四: 利用 reduction(+:sum) 直接规约sum 得到 sum.  其中，算法一就是简单地将求和任务划分给各个线程执行, 最后串行求和. 算法二则利用了 OpenMP 给出的for循环自动划分方案, 但同样也是串行求和, 仍有改进空间; 而算法三利用 critical 区使得各个进程可以并发对sum做加法, 从而理论上会一定程度上提高了程序的速度;最后算法四利用了reduction操作来求和, 理论上应当有利于指导编译器做进一步优化.  题目二：  本题目要求使用OpenMP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:  (用蓝色框圈出了重要部分)  算法一:    算法二:    算法三:    算法四:    题目二：   1. 均匀划分   均匀划分时, 计算各个线程的 begin 和 end即可     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C++算法库进行样本排序     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 seg\_idx    而后依据这个计算出交换后的各线程的各段起始位置和长度     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.    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: gcc pi-vx.c -o pi -O3 -fopenmp.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:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 线程数  算法 | 串行 | 算法一 | 算法二 | 算法三 | 算法四 | | 1 | 0.973s | 0.983 | 0.995 | 0.987 | 0.991 | | 2 | - | 0.493 | 0.501 | 0.501 | 0.511 | | 4 | - | 0.265 | 0.263 | 0.265 | 0.259 | | 8 | - | 0.166 | 0.189 | 0.153 | 0.170 | | 16 | - | 0.146 | 0.145 | 0.156 | 0.151 |   可见其随线程数增加而增速的能力有一定程度的限制, 加速比并不总是等于线程数. 但各个算法的加速是比较稳定的, 也就是说对于这样一个简单的程序, 上述四种写法都是可以接受的.  题目二：  首先考虑 4 个线程的运行情况, 单位是ns:   |  |  |  |  | | --- | --- | --- | --- | | 数据量 | std::sort运行时间(ns) | PSRS运行时间(ns) | 加速比 | | 1e2 | 2118 | 122776 | 0.017 | | 1e3 | 27347 | 109961 | 0.249 | | 1e4 | 350337 | 244583 | 1.433 | | 1e5 | 4207802 | 1395777 | 3.015 | | 1e6 | 48754399 | 15271155 | 3.193 | | 1e7 | 577245529 | 169966541 | 3.396 | | 1e8 | 6797659276 | 1945353220 | 3.494 |   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:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 线程数 | 1 | 2 | 4 | 8 | 16 | | PSRS运行时间(ns) | 6919596034 | 3892762202 | 1913142240 | 1125923880 | 872956794 | | 加速比 | 0.914382 | 1.720616 | 3.354127 | 6.374795 | 8.355579 |   在图中反映如下:    由此可见, 虽然更多线程能够带来更大的加速比, 但是并不总是和线程数成正比, 这是由于多线程自身的开销及数据集大小造成的. 比如用16个线程运行的时候, 当数据量为1e8时, 加速比只有8.36, 但若数据量为1e9, 则能达到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(主要考虑PSRS算法实现)：  内存结构与布局可能影响(如NUMA结构等)  OpenMP 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) | | | | | | |
| **总结：本次实验通过学习老师给出的计算 π的几种OpenMP实现方法的例子, 学习了如何使用OpenMP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OpenMP编程及一些并行算法有了更深刻的认识.** | | | | | | |
| 附录  （源码） | | 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: | | | | |