**实验3**

学号 姓名

**实验题目:**

**利用MPI进行n体模拟**

**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 7.3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)**

**按照题目要求，每一时刻每个分线程根据上一时刻的位置计算出各自管辖的实体的加速度，速度与位置，然后将计算出来的结果返回主线程，主线程再将距离的结果广播到所有线程，用以下一时刻的计算。这是一个非常朴素的算法。**

**注意到加速度的值只需要临时计算，速度的值只会被当前的加速度（由上一步的位置坐标决定）所影响，故这些量不需要进入通信的部分中。**

**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**

**计算相互作用力：**

**void** computeForce(**int** my\_particles, **double** masses[], vector my\_forces[], vector positions[], **int** num\_particles, **int** chunk)  
{  
  
 **double** m\_g;  
 vector f\_part\_k;  
 **double** len, len\_3, fact;  
  
 **int** part = my\_rank \* chunk + my\_particles;  
 my\_forces[my\_particles][X] = my\_forces[my\_particles][Y] = 0.0;  
  
 **for** (**int** k = 0; k < num\_particles; k++)  
 {  
 **if** (k != part)  
 {  
 f\_part\_k[X] = positions[part][X] - positions[k][X];  
 f\_part\_k[Y] = positions[part][Y] - positions[k][Y];  
  
  
 len = sqrt(pow(f\_part\_k[X], 2) + pow(f\_part\_k[Y], 2));  
 len\_3 = pow(len, 3);  
  
  
 m\_g = G \* masses[part] \* masses[k];  
 fact = m\_g / len\_3;  
  
 f\_part\_k[X] \*= fact;  
 f\_part\_k[Y] \*= fact;  
  
 my\_forces[my\_particles][X] += f\_part\_k[X];  
 my\_forces[my\_particles][Y] += f\_part\_k[Y];  
 }  
 }  
}

**根据相互作用力逐步计算剩下来的三个量：（也就是compute\_a(), compute\_v(), compute\_s()三个函数）。**

**void** updateParticles(**int** my\_particles, **double** masses[], vector my\_forces[], vector my\_positions[], vector my\_velocities[], **int** num\_particles, **int** chunk, **double** delta\_t)  
{  
  
 **int** part;  
 **double** fact;  
  
 part = my\_rank\*chunk + my\_particles;  
 fact = delta\_t/masses[part];  
  
 my\_positions[my\_particles][X] += delta\_t \* my\_velocities[my\_particles][X];  
 my\_positions[my\_particles][Y] += delta\_t \* my\_velocities[my\_particles][Y];  
 my\_velocities[my\_particles][X] += fact \* my\_forces[my\_particles][X];  
 my\_velocities[my\_particles][Y] += fact \* my\_forces[my\_particles][Y];  
  
}

**实验结果:**

**运行时间**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 规模, 步数\线程 | 1 | 2 | 4 | 8 | 16 | 32 |
| 64, 100000 | **38.806** | **19.583** | **10.080** | **5.549** | **3.642** | **2.485** |
| 256, 50000 | **309.857** | **156.863** | **78.805** | **39.752** | **23.023** | **15.5303** |

**运行时间**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 规模, 步数\线程 | 1 | 2 | 4 | 8 | 16 | 32 |
| 64, 100000 | **1** | **1.982** | **3.850** | **6.993** | **10.655** | **15.616** |
| 256, 50000 | **1** | **1.975** | **3.932** | **7.795** | **13.459** | **19.952** |

**分析与总结**

**从上面的实验可以看出，程序的并行性能被发挥得很好，基本都能以理论最大加速比来运行。**

**另外也可以看出来，即使在最下面的那一组实验，步数比其他三组多得多（也就是通信量多得多）的情况下，程序的加速比仍然稳定在一个非常高的水平，不如说还由于数据规模的显著增加而提升了，也可以看出MPI进程间通信的开销其实是很小的。**

**但是这个算法本身是一个复杂度很高的算法：****O(n2)，因此即使此种模型下的并行度体现得很好，但是对于这个问题本身并不一定是最适合的解决方案。而另一种O(nlgn)的算法更难被并行，但是在这个问题下的表现可能比并行算法更好。**