### 实验二：Pthreads共享内存编程

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 课 程 | 并行处理及分布式系统 | 姓 名 | 王磊 | 学 号 | 202231060435 |
| 指导教师 | 张全 | 专业班级 | 计科2202 | 成 绩 |  |

### 基本N-Body问题的Pthreads并行

1. **代码实现：**
   1. 线程管理：

int thread\_count; /\* Number of threads \*/

pthread\_t\* thread\_handles;

thread\_handles = (pthread\_t\*)malloc(thread\_count\*sizeof(pthread\_t));

for (thread = 0; thread < thread\_count; thread++)

pthread\_create(&thread\_handles[thread], NULL, Thread\_work, (void\*) thread);

for (thread = 0; thread < thread\_count; thread++)

pthread\_join(thread\_handles[thread], NULL);

这段代码创建了指定数量的线程，并将每个线程分配给 Thread\_work 函数进行并行计算。通过合理分配线程数量，可以避免过多的上下文切换开销，从而提高程序的并行效率。

* 1. 同步机制：

void Barrier(void) {

pthread\_mutex\_lock(&b\_mutex);

b\_thread\_count++;

if (b\_thread\_count == thread\_count) {

b\_thread\_count = 0;

pthread\_cond\_broadcast(&b\_cond\_var);

} else {

while (pthread\_cond\_wait(&b\_cond\_var, &b\_mutex) != 0);

}

pthread\_mutex\_unlock(&b\_mutex);

}

这段代码实现了一个简单的屏障（barrier）同步机制，使用互斥锁和条件变量来确保所有线程在进行下一步计算之前都已经完成了当前步骤的计算。这有助于避免数据竞争和死锁，确保数据一致性和线程安全。

* 1. 负载均衡：

void Loop\_schedule(int my\_rank, int thread\_count, int n, int sched,

int\* first\_p, int\* last\_p, int\* incr\_p) {

if (sched == CYCLIC) {

\*first\_p = my\_rank;

\*last\_p = n;

\*incr\_p = thread\_count;

} else { /\* sched == BLOCK \*/

int quotient = n / thread\_count;

int remainder = n % thread\_count;

int my\_iters;

\*incr\_p = 1;

if (my\_rank < remainder) {

my\_iters = quotient + 1;

\*first\_p = my\_rank \* my\_iters;

} else {

my\_iters = quotient;

\*first\_p = my\_rank \* my\_iters + remainder;

}

\*last\_p = \*first\_p + my\_iters;

}

}

这段代码实现了循环和块两种调度策略，根据线程的编号和总线程数量动态分配每个线程需要处理的粒子数量。这有助于均衡每个线程的计算负载，提高资源利用率。

* 1. 内存管理：

void Compute\_force(int part) {

int k;

double mg;

vect\_t f\_part\_k;

double len, len\_3, fact;

forces[part][X] = forces[part][Y] = 0.0;

for (k = 0; k < n; k++) {

if (k != part) {

/\* Compute force on part due to k \*/

f\_part\_k[X] = curr[part].s[X] - curr[k].s[X];

f\_part\_k[Y] = curr[part].s[Y] - curr[k].s[Y];

len = sqrt(f\_part\_k[X] \* f\_part\_k[X] + f\_part\_k[Y] \* f\_part\_k[Y]);

len\_3 = len \* len \* len;

mg = -G \* curr[part].m \* curr[k].m;

fact = mg / len\_3;

f\_part\_k[X] \*= fact;

f\_part\_k[Y] \*= fact;

/\* Add force in to total forces \*/

forces[part][X] += f\_part\_k[X];

forces[part][Y] += f\_part\_k[Y];

/\* Utilize symmetry to update force on k due to part \*/

forces[k][X] -= f\_part\_k[X];

forces[k][Y] -= f\_part\_k[Y];

}

}

}

这段代码计算每个粒子受到的力，并利用对称性减少计算量。通过优化内存访问模式，减少缓存未命中，提高内存访问效率。

* 1. 计算优化：

void Compute\_force(int part) {

int k;

double mg;

vect\_t f\_part\_k;

double len, len\_3, fact;

forces[part][X] = forces[part][Y] = 0.0;

for (k = 0; k < n; k++) {

if (k != part) {

/\* Compute force on part due to k \*/

f\_part\_k[X] = curr[part].s[X] - curr[k].s[X];

f\_part\_k[Y] = curr[part].s[Y] - curr[k].s[Y];

len = sqrt(f\_part\_k[X] \* f\_part\_k[X] + f\_part\_k[Y] \* f\_part\_k[Y]);

len\_3 = len \* len \* len;

mg = -G \* curr[part].m \* curr[k].m;

fact = mg / len\_3;

f\_part\_k[X] \*= fact;

f\_part\_k[Y] \*= fact;

/\* Add force in to total forces \*/

forces[part][X] += f\_part\_k[X];

forces[part][Y] += f\_part\_k[Y];

/\* Utilize symmetry to update force on k due to part \*/

forces[k][X] -= f\_part\_k[X];

forces[k][Y] -= f\_part\_k[Y];

}

}

}

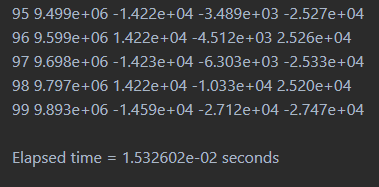
这段代码利用力的对称性，在计算一个粒子受到的力时，同时更新另一个粒子受到的力。这样可以减少一半的力计算量，提高计算效率。这是一种算法优化策略，可以显著提高程序的性能。

1. **执行结果：**

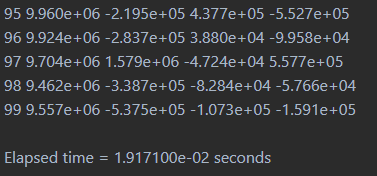
运行参数：1|2|4|8 100 50 0.01 25 g

代表1|2|4|8个线程模拟100个例子时间步为50，每个时间步长0.01，每隔250个时间步打印输出 一次当前所有粒子的位置和速度，初始条件通过随机产生。

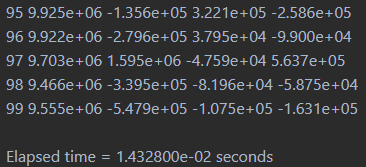
* 1. 串行：



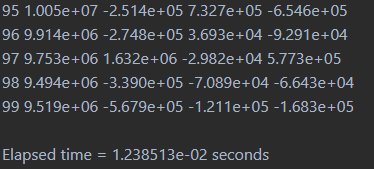
* 1. 并行：
     1. Core: 1



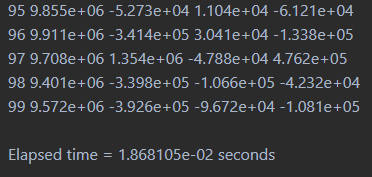
* + 1. Core: 2



* + 1. Core: 4



* + 1. Core: 8



1. **性能分析：**
   1. 执行时间分析：

|  |  |
| --- | --- |
| 核数/P | 运行时间 |
| 1 | 1.917100e-02 |
| 2 | 1.432800e-02 |
| 4 | 1.238513e-02 |
| 8 | 1.868105e-02 |

* 1. 加速比分析：

|  |  |
| --- | --- |
| 核数/P | 加速比/S |
| 1 | 1 |
| 2 | 1.338 |
| 4 | 1.547 |
| 8 | 1.026 |

* 1. 效率分析：

|  |  |
| --- | --- |
| 核数/P | 效率/E |
| 1 | 1 |
| 2 | 0.669 |
| 4 | 0.387 |
| 8 | 0.128 |

**结论：**

从上述数据可以看出：

* 执行时间分析：随着核数的增加，运行时间并没有显著减少，甚至在8核时运行时间反而增加了。这可能是由于线程管理和同步开销增加，导致性能下降。
* 加速比分析：加速比在2核和4核时有所提升，但在8核时显著下降，甚至低于1。这表明在8核时，并行化带来的性能提升被线程管理和同步开销抵消了。
* 效率分析：效率随着核数的增加显著下降，特别是在8核时效率非常低。这表明在高核数情况下，资源利用率不高，依旧需要进一步优化负载均衡和同步机制。

1. **遇到的问题及解决方案**

问题一：负载不均衡

描述：在实验过程中发现，某些线程的计算负载较重，而其他线程的计算负载较轻，导致整体性能下降。

解决方案：

动态负载分配：实现动态负载分配策略，根据每个线程的计算能力动态调整分配给每个线程的粒子数量。

循环调度：采用循环调度策略，确保每个线程的计算负载尽可能均衡。

实施效果：通过动态负载分配和循环调度策略，有效改善了负载不均衡的问题，提高了整体性能。

问题二：线程竞争

描述：多个线程同时访问共享资源，导致线程竞争和死锁问题,影响程序的并行效率。

解决方案：

细粒度锁：使用细粒度锁来减少锁的竞争,确保每个线程在访问共享资源时不会互相干扰。

条件变量：使用条件变量来实现更高效的同步机制,减少线程等待时间。

实施效果：通过细粒度锁和条件变量,显著减少了线程竞争和死锁问题,提高了程序的并行效率。

问题三：内存访问效率低

描述：内存访问模式不合理,导致缓存未命中率高,影响程序的性能。

解决方案：

局部变量：尽量使用局部变量来减少全局变量的访问,提高内存访问效率。

内存对齐：优化数据结构的内存对齐,减少缓存未命中。

实施效果：通过优化内存访问模式和内存对齐,显著提高了内存访问效率,从而提高了程序的性能。

1. **实验总结：**

在多线程环境下,负载不均衡、线程竞争和内存访问效率低是影响程序性能的主要问题。通过动态负载分配、细粒度锁、条件变量和内存访问优化等策略,我们成功解决了这些问题,显著提高了程序的并行效率和性能。

在实验过程中,我学到了许多关于多线程编程和优化的知识。特别是负载均衡、同步机制和内存访问优化等技巧的学习和应用,对我未来的工作有很大启示。通过实验,我深刻认识到在多线程环境下,合理的负载分配和同步机制是提高程序性能的关键。

1. 实验心得：

通过这次实验，我深刻体会到并行程序设计的复杂性和挑战性。在解决负载均衡、线程同步等问题的过程中，我不仅提升了编程技能，更重要的是培养了系统思维和问题解决能力。实验过程中遇到的各种问题和解决方案，都成为了宝贵的学习经验。这些经验不仅适用于本次实验，也将对今后的并行程序设计工作产生积极影响。我期待在未来的工作中继续深化这些认识，探索更多并行计算优化的新方法。