### 实验一：MPI分布式内存编程

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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** |  |

1. 基本N-Body问题的MPI并行
2. **代码实现：**
   1. MPI并行实现：

MPI并行实现的目标是将任务分配给多个进程，每个进程处理一部分粒子的计算。

#include <mpi.h>

int main(int argc, char\* argv[]) {

int rank, size;

MPI\_Init(&argc, &argv); // 初始化MPI环境。

// 获取当前进程的排名（rank）

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

//获取进程总数（size）

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

// 其他代码...

MPI\_Finalize(); // 终止MPI环境

return 0;

}

该部分代码初始化MPI环境，并在程序结束时终止MPI环境。

* 1. 通信优化：

通信优化的目标是减少通信开销，通过优化消息大小和使用非阻塞通信来实现。

MPI\_Datatype particle\_type;

MPI\_Type\_contiguous(5, MPI\_DOUBLE, &particle\_type);

MPI\_Type\_commit(&particle\_type);

int\* sendcounts = malloc(size \* sizeof(int));

int\* displs = malloc(size \* sizeof(int));

for (int i = 0; i < size; i++) {

sendcounts[i] = n / size;

if (i < n % size) sendcounts[i]++;

displs[i] = (i > 0) ? displs[i - 1] + sendcounts[i - 1] : 0;

}

MPI\_Request request;

MPI\_Iscatterv(all\_particles, sendcounts, displs, particle\_type, curr, local\_n, particle\_type, 0, MPI\_COMM\_WORLD, &request);

MPI\_Wait(&request, MPI\_STATUS\_IGNORE);

MPI\_Type\_contiguous：创建一个新的MPI数据类型来表示粒子结构。

MPI\_Iscatterv：使用非阻塞通信将粒子数据分散到各个进程。

MPI\_Wait：等待非阻塞通信完成。

* 1. 负载均衡：

负载均衡的目标是确保每个处理器的工作量大致相等，以提高资源利用率。

int local\_n = n / size; // 计算每个进程负责的粒子数量

if (rank < n % size) local\_n++;

struct particle\_s\* curr;

// 使用MPI的内存分配器来分配内存

MPI\_Alloc\_mem(local\_n \* sizeof(struct particle\_s), MPI\_INFO\_NULL, &curr);

这部分代码实现了负载均衡。将计算任务均匀分配给每个进程完成，提高了系统的整体性能、可靠性和可拓展性

* 1. 内存管理：

MPI\_Alloc\_mem(local\_n \* sizeof(struct particle\_s), MPI\_INFO\_NULL, &curr);

MPI\_Alloc\_mem(local\_n \* sizeof(vect\_t), MPI\_INFO\_NULL, &forces);

// 使用完内存后释放

MPI\_Free\_mem(curr);

MPI\_Free\_mem(forces);

MPI\_Alloc\_mem：使用MPI的内存分配器来分配内存。

MPI\_Free\_mem：释放分配的内存。

* 1. 计算优化：

void Compute\_force(int part, vect\_t forces[], struct particle\_s curr[], int n) {

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) {

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;

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

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

}

}

}

Compute\_force：计算每个粒子受到的力。

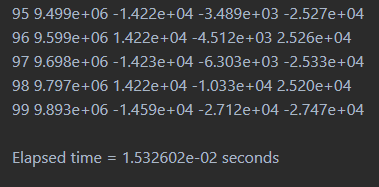
优化计算过程，减少重复计算。

1. **执行结果：**

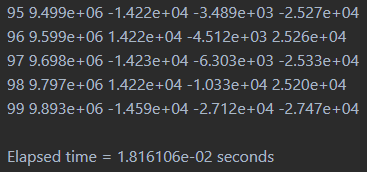
运行参数：100 50 0.01 25 g

代表模拟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。

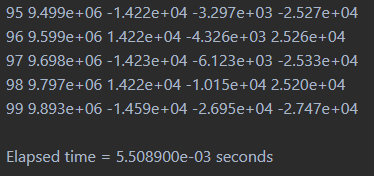
* 1. 串行：



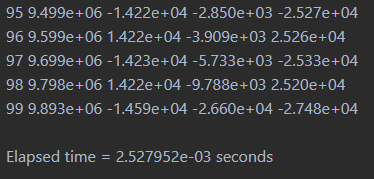
* 1. 并行：
     1. Core: 1



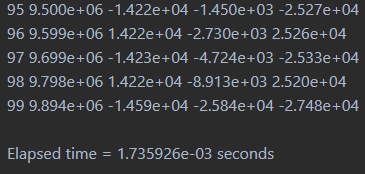
* + 1. Core: 2



* + 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 | 1.816106e-02 |
| 2 | 5.508900e-03 |
| 4 | 2.527952e-03 |
| 8 | 1.735926e-03 |

随着核数的增加，运行时间呈现减少趋势

* 1. 加速比分析：

|  |  |
| --- | --- |
| 核数/P | 加速比/S |
| 1 | 1 |
| 2 | 3.284751 |
| 4 | 7.184100 |
| 8 | 10.461886 |

加速比的逐步增加表明，程序的并行化有效，且核数越多时，性能提升越显著。随着核数进一步增加，加速比提升的幅度变小，表明加速逐渐趋于饱和。

* 1. 效率分析：

|  |  |
| --- | --- |
| 核数/P | 效率/E |
| 1 | 1 |
| 2 | 1.642376 |
| 4 | 1.796025 |
| 8 | 1.307736 |

核数为1-4时，效率随着核数的增加而增加，这表明每个处理器的利用率在增加。当核数高于四时，效率随着核数的增加逐渐减少，这表明每个处理器的利用率在减少，这是由进程间逐步增加的通信开销不能被计算工作充分吸收，且负载不完全均衡等原因导致的。

结论：程序的并行化在核数较少时（1到2核）获得了明显的性能提升，而随着核 数的增加，性能提升变得越来越缓慢，最终趋于饱和。

1. **遇到的问题及解决方案**
   1. 负载不均衡

**问题描述：**

在并行计算中，不同进程处理的粒子数量不均匀，导致某些进程的计算时间较长，从而影响整体性能。

**解决方案：**

块划分：将粒子数据按块划分，确保每个进程处理的粒子数量大致相等。

动态负载均衡：在计算过程中动态调整任务分配，确保每个进程的工作量大致相等。

**实施效果**：

通过块划分和动态负载均衡，显著减少了负载不均衡的问题，提高了整体计算效率。

* 1. 通信瓶颈

**问题描述：**

在并行计算中，通信开销较大，特别是在数据量较大时，通信时间占据了较大比例，影响了计算性能。

**解决方案：**

非阻塞通信：使用非阻塞通信函数（如MPI\_Iscatterv和MPI\_Igatherv）来减少通信开销。

优化消息大小：优化消息大小，减少通信次数，提高通信效率。

**实施效果：**

通过使用非阻塞通信和优化消息大小，显著减少了通信开销，提高了计算性能。

* 1. 内存管理

**问题描述：**

在并行计算中，内存分配和释放不当，导致内存碎片和泄漏，影响了计算性能。

**解决方案：**

MPI内存分配器：使用MPI的内存分配器（如MPI\_Alloc\_mem）来管理内存，减少内存拷贝和碎片。

内存池：使用内存池技术，提高内存分配和释放的效率。

**实施效果：**

通过使用MPI内存分配器和内存池技术，显著提高了内存管理效率，减少了内存碎片和泄漏。

1. **实验总结**

实验结果的总结：

* + 1. 执行时间：随着核数的增加，执行时间显著减少，表明并行化有效地减少了计算时间。
    2. 加速比：加速比随着核数的增加而增加，但增加的幅度逐渐变小，表明并行化的效果在一定程度上受到限制。
    3. 效率：效率随着核数的增加而减少，表明每个处理器的利用率在降低。

反思：

在实验过程中，我深刻体会到并行计算中的挑战，特别是负载均衡和通信瓶颈。并行计算不仅仅是简单地将任务分配到多个进程，还需要考虑如何最小化进程间的依赖、如何平衡各进程的负载，以及如何减少通信的开销。此外，Amdahl 法则提醒我们，在优化并行计算时，串行部分的影响不容忽视。

从这次实验中，我学到了如何有效地分析和优化并行程序的性能，也认识到并行计算中的一些常见问题和挑战。未来的工作可以进一步探讨如何通过更细粒度的并行化、更高效的通信协议和更先进的负载均衡算法，进一步提升并行程序的性能。

### **对未来工作的启示：**

1. **GPU加速**：对于计算密集型的 N-Body 问题，可以考虑使用 GPU 来进行计算加速，尤其是在大规模数据计算时，GPU 能够提供比 CPU 更高的并行处理能力。
2. **大规模并行化**：在核数较多的情况下，如何通过更高效的负载均衡和通信策略，充分利用所有计算资源，依然是一个值得深入研究的方向。
3. **更高效的算法**：除了通信和负载均衡的优化，还可以考虑优化 N-Body 算法本身。例如，使用 Barnes-Hut 算法来加速 N-Body 问题中的力计算，减少每个粒子间的计算量。

通过不断的优化策略和新技术的引入，可以将并行计算的效率和可扩展性提高到新的水平，尤其在大规模模拟中具有重要的应用价值。

1. **实验心得：**

通过这次 N-Body 问题的并行优化实验,我对并行计算有了更深入的认识。实验过程中我发现,高效的并行计算远不止简单地分配任务给多个处理器,还需要考虑任务的均衡分配、进程间通信开销的控制以及内存的高效管理等多个方面。为了提升性能,我尝试了包括任务块划分、使用非阻塞通信、优化内存分配等多种方法,虽然在调试过程中遇到了数据同步和内存泄漏等问题,但通过反复实验和分析最终都得到了解决。这次实验让我深刻体会到理论知识必须结合实践才能真正掌握,性能优化是一个需要不断迭代的过程,同时也认识到调试和性能分析工具的重要性。这些经验对我今后的学习和工作都很有帮助,在未来的并行计算相关工作中,我会更加注重前期的任务划分设计、及时的性能分析以及良好的代码风格维护。

附件代码：

