**《并行计算》课程实验报告**

**实验3：基于华为云环境的MPI并行编程**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 姓名 | 常添 | | 院系 | | 计算学部 | | 学号 | | 2022111699 | | |
| 任课教师 | | 张伟哲 | | | | 指导教师 | |  | | | |
| 实验地点 | | 格物楼213 | | | | 实验时间 | | 2024年10月11日 | | | |
| 实验课表现 | | 出勤、表现得分 | |  | | 实验报告  得分 | |  | | 实验总分 |  |
| 操作结果得分 | |  | |
| **一、实验目的** | | | | | | | | | | | |
| 实验的主要目的包括：  1.掌握N体问题和素数计算程序的基本原理：  理解N体问题的物理模型，包括天体间的相互作用及其通过牛顿力学模拟的基本过程。  理解素数计算算法，在指定范围内统计素数的基本方法。  2.应用MPI进行并行优化：  通过MPI将N体问题和素数计算的任务并行分配到多个进程上，体验并行编程的基本思想，提高计算效率。  3.评估并行性能：  通过加速比和效率的计算，分析程序在不同数据规模和进程数下的可扩展性，评估负载均衡和通信开销对并行性能的影响。  实现方法：  1. N体问题并行实现  问题概述：N体问题涉及多个天体之间的相互作用和力的计算，根据牛顿万有引力定律，模拟其运动。每个天体受到其他天体的引力作用，导致加速度变化，并最终影响速度和位置。  并行化实现：采用MPI的SPMD（单程序多数据流）模式，将天体分配给不同的进程。每个进程计算自己负责的天体所受的力，并更新其速度和位置。使用 MPI\_Send 和 MPI\_Recv 实现进程间数据交换，使所有进程都获得最新的天体信息，以便在下一步的计算中使用。  2. 素数计算程序的MPI并行优化  问题概述：素数计算程序在给定范围内统计素数的数量。串行实现通过逐个遍历数字并判断是否为素数来完成。  并行化实现：使用MPI将素数计算的任务划分给多个进程。根据进程数量，使用静态任务分配的方式，将需要检查的数范围分配给各个进程。  结果汇总：利用 MPI\_Reduce 将每个进程的部分结果汇总到主进程，最后在主进程中输出总结果。  负载均衡优化：在优化过程中，通过修改任务划分方式，确保每个进程获得均衡的计算负载，尽量减少通信开销，进一步提升并行效率。 | | | | | | | | | | | |
| **二、实验内容** | | | | | | | | | | | |
| 在本次实验中，我主要完成了以下工作：  1. N体问题的并行实现  编写串行代码：首先实现了N体问题的串行代码，计算每个天体受其他天体引力的合力，并据此更新其加速度、速度和位置。  编写MPI并行代码：将N体问题代码并行化，使用MPI将天体分配给多个进程，进程之间通过MPI\_Send和MPI\_Recv交换信息，以更新每个时间步的天体位置。  实现进程同步：在每个时间步结束后，使用 MPI\_Barrier 实现进程同步，以确保所有进程在下一步计算前获得最新的天体信息。  配置多机环境并运行测试：创建配置文件（config\_simple\_1, config\_simple\_2, 等），并在多台机器上运行实验。记录了不同进程数和数据规模（天体数量）下的运行时间。  2. 素数计算程序的并行优化  编写串行代码：实现了一个计算指定范围内素数数量的串行程序，用于在N从2到给定值之间统计素数个数。  修改代码以支持MPI并行：添加了MPI函数，分配每个进程检查特定的数字子集。通过静态划分任务，确保每个进程获得相似的计算负载。  使用MPI\_Reduce汇总结果：在并行计算结束后，使用 MPI\_Reduce 汇总每个进程计算的部分结果，并在主进程中输出总数和运行时间。  优化代码负载均衡：在初次测试时发现多进程效率差异，进一步调整了代码逻辑，确保负载均衡，减少通信和同步开销。  配置文件创建及多机测试：创建了不同的配置文件（config\_1, config\_2, config\_4, config\_6），在单机及多机环境下测试多种进程数与数据规模组合，收集运行时间数据。  3. 性能分析  计算加速比和效率：根据实验结果，计算了每个实验组合的加速比和效率，分析了并行程序在不同条件下的扩展性。  数据整理和图表生成：将运行时间和计算结果整理成表格，并生成图表，用于直观地展示加速比和效率的变化趋势。  对结果进行分析：通过实验结果分析N体问题和素数计算程序在并行下的扩展性和负载均衡问题，指出了通信开销对性能的影响，并讨论了潜在的优化方向。  实验源代码：prime\_mpi.c：  #include <stdlib.h>  #include <stdio.h>  #include <time.h>  #include <mpi.h>  int prime\_part(int id, int p, int n);  int main(int argc, char \*argv[])  {  int id, p, n, total, total\_part;  double start\_time, end\_time;  // 初始化MPI环境  MPI\_Init(&argc, &argv);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &p);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &id);  // 从命令行读取N值  if (argc < 2)  {  if (id == 0) printf("Usage: %s <N>\n", argv[0]);  MPI\_Finalize();  return 1;  }  n = atoi(argv[1]);  start\_time = MPI\_Wtime(); // 开始计时  // 每个进程计算自己分配的部分素数  total\_part = prime\_part(id, p, n);  // 使用MPI\_Reduce收集所有进程的结果到进程0  MPI\_Reduce(&total\_part, &total, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);  end\_time = MPI\_Wtime(); // 结束计时  // 仅进程0输出最终结果和计算时间  if (id == 0)  {  printf("\nBetween 2 and %d, there are %d primes\n", n, total);  printf("Execution Time: %f seconds\n", end\_time - start\_time);  }  // 结束MPI环境  MPI\_Finalize();  return 0;  }  int prime\_part(int id, int p, int n)  {  int i, j, prime, total\_part = 0;  for (i = 2 + id; i <= n; i += p)  {  prime = 1;  for (j = 2; j \* j <= i; j++)  {  if (i % j == 0)  {  prime = 0;  break;  }  }  if (prime)  {  total\_part++;  }  }  return total\_part;  }  实验源代码：prime\_mpi\_2.c 优化了负载均衡的问题：  #include <stdlib.h>  #include <stdio.h>  #include <mpi.h>  int is\_prime(int number);  int prime\_part\_range(int start, int end);  int main(int argc, char \*argv[])  {  int id, p, n, total = 0;  double start\_time, end\_time;  MPI\_Status status;  // 初始化MPI环境  MPI\_Init(&argc, &argv);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &p);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &id);  // 从命令行读取N值  if (argc < 2)  {  if (id == 0) printf("Usage: %s <N>\n", argv[0]);  MPI\_Finalize();  return 1;  }  n = atoi(argv[1]);  start\_time = MPI\_Wtime(); // 开始计时  if (p == 1) {  // 单进程模式，直接计算所有素数  total = prime\_part\_range(2, n);  } else if (id == 0) {  // 多进程模式，进程0分配任务范围  for (int i = 1; i < p; i++) {  int start = (i - 1) \* (n / (p - 1)) + 2;  int end = (i == p - 1) ? n : i \* (n / (p - 1)) + 1;  MPI\_Send(&start, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD);  MPI\_Send(&end, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD);  }  // 接收每个进程的部分结果  int partial;  for (int i = 1; i < p; i++) {  MPI\_Recv(&partial, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD, &status);  total += partial;  }  } else {  // 工作进程接收任务范围并计算素数  int start, end;  MPI\_Recv(&start, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &status);  MPI\_Recv(&end, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &status);  int local\_count = prime\_part\_range(start, end);  // 发送部分结果回到进程0  MPI\_Send(&local\_count, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);  }  end\_time = MPI\_Wtime(); // 结束计时  // 仅进程0输出最终结果和计算时间  if (id == 0) {  printf("\nBetween 2 and %d, there are %d primes\n", n, total);  printf("Execution Time: %f seconds\n", end\_time - start\_time);  }  MPI\_Finalize();  return 0;  }  // 判断是否为素数的函数  int is\_prime(int number) {  if (number < 2) return 0;  for (int i = 2; i \* i <= number; i++) {  if (number % i == 0) return 0;  }  return 1;  }  // 计算指定范围内的素数数量  int prime\_part\_range(int start, int end) {  int local\_count = 0;  for (int i = start; i <= end; i++) {  if (is\_prime(i)) {  local\_count++;  }  }  return local\_count;  } | | | | | | | | | | | |
| **三、实验结果** | | | | | | | | | | | |
| 实验1.2.1：单机上，数据规模为6000时，随每机进程数变化的运行时间；  运行结果截图：  图片包含 文本  描述已自动生成  数据统计表格：   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 进程数 | 1 | 2 | 3 | 4 | | 时间 | 7.507479 | 3.765938 | 2.511349 | 1.884503 |     实验1.2.2：相同数据规模为6000，随每机进程数变化的运行时间  运行结果截图：      数据统计表格：   |  |  |  |  | | --- | --- | --- | --- | | 每机进程数 | 单机 | 双机 | 三机 | | 1 | 7.507479 | 3.765938 | 2.511349 | | 2 | 3.765938 | 1.889411 | 1.278232 | | 3 | 2.511349 | 1.272890 | 0.861549 | | 4 | 1.884503 | 0.952066 | 0.639768 |     实验1.2.3：每机1个进程，随数据规模变化的n-body并行程序运行时间。  运行结果截图：        数据统计表格：   |  |  |  |  | | --- | --- | --- | --- | | 粒子数n | 单机 | 双机 | 三机 | | 150 | 0.004830 | 0.016591 | 0.018397 | | 300 | 0.018955 | 0.019619 | 0.011379 | | 600 | 0.075471 | 0.053722 | 0.043577 | | 1200 | 0.301185 | 0.153330 | 0.107714 | | 2400 | 1.209091 | 0.605584 | 0.409054 | | 4800 | 4.818647 | 2.414285 | 1.617593 | | 9600 | 19.285291 | 9.653732 | 6.449484 |     步骤三：根据记录的数据计算加速比与效率（给出数据并绘图）  实验1.2.4：单机上，粒子数为6000，随进程数变化加速比（Sp）统计。  加速比（Sp）统计表   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 进程数 | 1 | 2 | 3 | 4 | | 加速比 | 1.00 | 1.99 | 2.99 | 3.98 |     效率（Ep）统计表   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 进程数 | 1 | 2 | 3 | 4 | | 加速比 | 1.000 | 0.995 | 0.997 | 0.995 |     实验1.2.5：粒子数为6000，随每机进程数变化的加速比。  加速比（Sp）统计表   |  |  |  |  | | --- | --- | --- | --- | | 每机进程数 | 单机 | 双机 | 三机 | | 1 | 1.00 | 1.99 | 2.98 | | 2 | 1.99 | 3.97 | 5.87 | | 3 | 2.99 | 5.89 | 8.71 | | 4 | 3.98 | 7.88 | 11.73 |     效率（Ep）统计表   |  |  |  |  | | --- | --- | --- | --- | | 每机进程数 | 单机 | 双机 | 三机 | | 1 | 1.000 | 0.995 | 0.993 | | 2 | 0.995 | 0.993 | 0.978 | | 3 | 0.997 | 0.982 | 0.968 | | 4 | 0.995 | 0.985 | 0.978 |     实验1.2.6：每机1个进程，随数据规模变化的n-body并行程序加速比和效率。  加速比（Sp）统计表   |  |  |  |  | | --- | --- | --- | --- | | 粒子数n | 单机 | 双机 | 三机 | | 150 | 1.00 | 0.29 | 0.26 | | 300 | 1.00 | 0.97 | 1.67 | | 600 | 1.00 | 1.40 | 1.73 | | 1200 | 1.00 | 1.96 | 2.80 | | 2400 | 1.00 | 2.00 | 2.96 | | 4800 | 1.00 | 2.00 | 2.98 | | 9600 | 1.00 | 2.00 | 2.99 |     效率（Ep）统计表   |  |  |  |  | | --- | --- | --- | --- | | 粒子数n | 单机 | 双机 | 三机 | | 150 | 1.00 | 0.15 | 0.09 | | 300 | 1.00 | 0.49 | 0.56 | | 600 | 1.00 | 0.70 | 0.58 | | 1200 | 1.00 | 0.98 | 0.93 | | 2400 | 1.00 | 1.00 | 0.99 | | 4800 | 1.00 | 1.00 | 0.99 | | 9600 | 1.00 | 1.00 | 1.00 |     实验1.3.1  运行结果截图：    **文本  描述已自动生成**  **文本  描述已自动生成**  **文本  描述已自动生成**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | N  进程数 | 1 | 2 | 4 | 6 | | 100000 | 0.006631 | 0.007353 | 0.003964 | 0.004479 | | 200000 | 0.017357 | 0.017963 | 0.009230 | 0.009282 | | 400000 | 0.045851 | 0.046528 | 0.023570 | 0.023413 | | 800000 | 0.122154 | 0.122650 | 0.061738 | 0.061105 |   进行N在不同取值下的实验并记录数据：  计算N在不同取值下的加速比如下：   |  |  |  |  |  | | --- | --- | --- | --- | --- | | N  进程数 | 1 | 2 | 4 | 6 | | 100000 | 1.00 | 0.90 | 1.67 | 1.48 | | 200000 | 1.00 | 0.97 | 1.88 | 1.87 | | 400000 | 1.00 | 0.99 | 1.94 | 1.96 | | 800000 | 1.00 | 1.00 | 1.98 | 2.00 |   实验1.3.2  使用任务队列方法解决负载均衡问题，运行结果截图：      **文本  描述已自动生成**     |  |  |  |  |  | | --- | --- | --- | --- | --- | | N  进程数 | 1 | 2 | 4 | 6 | | 100000 | 0.006737 | 0.007447 | 0.004405 | 0.004104 | | 200000 | 0.017532 | 0.018280 | 0.008921 | 0.006941 | | 400000 | 0.046499 | 0.047012 | 0.021380 | 0.014670 | | 800000 | 0.122738 | 0.123483 | 0.055256 | 0.035512 |   数据统计表格：  计算N在不同取值下的加速比如下：   |  |  |  |  |  | | --- | --- | --- | --- | --- | | N  进程数 | 1 | 2 | 4 | 6 | | 100000 | 1.00 | 0.90 | 1.53 | 1.64 | | 200000 | 1.00 | 0.96 | 1.97 | 2.53 | | 400000 | 1.00 | 0.99 | 2.17 | 3.17 | | 800000 | 1.00 | 0.99 | 2.22 | 3.46 | | | | | | | | | | | | |
| **四、思考题** | | | | | | | | | | | |
| 思考题1：从算法层面上分析素数计算程序负载不均衡的原因，并提出改进方法。  原因分析：  1. 任务划分的非均匀性：在静态划分的情况下，某些进程负责的数字区间中包含了更多较大的数。素数判断的计算复杂度随着数值的增大而增加，因为较大的数需要更长的时间进行素性验证。即使每个进程被分配了相同数量的数字，但因大数的计算复杂度更高，导致不同进程耗时不均。  2.素数密度的变化：在较小的数范围内，素数的密度较大，而在较大的数范围内，素数变得稀疏。因此，某些进程负责的区间中可能包含更多的素数，这些进程在验证素数时会消耗更多计算时间。  改进方法：  1.动态负载平衡：  任务动态分配：不事先将区间划分给每个进程，而是在运行时由主进程动态分配。各个进程完成当前任务后再请求新的任务。这样可以确保所有进程保持高效工作直到所有任务完成。  任务窃取（Work Stealing）：如果某些进程的任务量显著减少或提前完成，可以从尚有任务的进程窃取任务，实现负载均衡。  2.分块策略（Chunking）：  将数字区间划分为多个较小的块，然后将这些块分配给各进程。通过控制块的大小，可以更好地实现负载均衡。例如，让每个进程每次处理一个小块，完成后再分配新块。这样即使任务量不均匀，由于每次只分配一个块，可以有效避免某些进程因计算任务量较大而被拖慢。  3.适当调整任务划分：  采用素数分布预测，将较大数的素数判断任务平均分配给各进程。例如，可以基于平方根分割的方法，将更大的数集中在不同的进程中分配，以降低计算量较大的任务集中在某一进程的问题。  思考题2：根据该实验，简述并行程序编程和优化的步骤。  1.选择并行模型：  根据问题特性选择适合的并行模型，如共享内存模型、分布式内存模型（如MPI）、混合模型等。对于分布式内存模型，还需选择数据和任务的分配策略。  2.代码实现并行化：  在串行代码的基础上，将适合并行的部分分割成独立任务。插入并行编程的库函数（如MPI、OpenMP等），实现并行执行。  使用MPI\_Init()、MPI\_Comm\_size()、MPI\_Comm\_rank()等函数进行进程初始化和管理。  3.数据和任务划分：  基于负载均衡和数据局部性原则划分任务。可以选择静态划分或动态划分方法，以适应不同的任务特性。  确保每个进程分配到的任务尽量均衡，并合理控制进程间的通信量。  4.同步与通信：  使用适当的通信方式（如点对点通信、集合通信），在各进程之间共享必要的数据。  使用同步原语（如MPI的MPI\_Barrier）确保进程按照正确的步骤完成任务，避免竞争条件和数据不一致问题。  5.性能评估：  使用加速比、效率等指标评估并行程序的性能。通过实验测试不同进程数和数据规模下的表现，观察并行化的效果。  使用 MPI\_Wtime() 等函数进行性能计时，分析各进程的运行时间、通信开销和同步开销等。  6.负载均衡和优化：  针对负载不均衡问题，调整任务划分策略。可尝试动态分配、任务窃取、分块策略等方法。  尽量减少不必要的通信开销和同步开销，优化数据传输路径，合理安排进程之间的协作，提升并行效率。  7.调试和验证：  对并行程序进行测试和调试，确保其正确性和鲁棒性。在多进程环境中调试时要特别注意数据同步和通信问题。  验证并行结果是否与串行程序一致，确保并行化过程不会影响最终计算结果的准确性。  通过这些步骤，逐步实现并行程序的编写和优化，从而提升计算效率并充分利用计算资源。 | | | | | | | | | | | |
| **五、实验心得体会** | | | | | | | | | | | |
| 1.深入理解并行编程的基本概念：  实验过程中，我加深了对并行编程中关键概念的理解，例如任务划分、负载均衡、通信和同步等。尤其是在N体问题和素数计算这两个问题上，如何合理地划分任务，直接影响到程序的性能表现。  2.学会了使用MPI进行分布式内存的编程，并掌握了基本的MPI函数，如MPI\_Init、MPI\_Comm\_size、MPI\_Comm\_rank、MPI\_Reduce等，理解了这些函数在并行计算中的重要作用。  3.性能优化的思路与方法：  本实验的核心在于优化并行性能，通过加速比和效率的计算来分析并行化的效果。实验中发现，即使增加了进程数，性能提升并不总是线性的。这让我认识到负载均衡和通信开销是影响并行程序扩展性的关键因素。  在素数计算中，由于任务划分不均，部分进程在等待其他进程完成，影响了整体效率。通过动态分配任务、分块策略等方法，可以有效地改善负载均衡问题，从而提高程序的扩展性。  4.并行编程中的挑战：  并行编程不仅仅是任务划分，还涉及到进程间的通信、同步和数据共享。尤其是在多机环境下，通信开销显著，增加了并行编程的复杂性。  实验过程中出现了进程数增加时性能反而下降的问题，这让我意识到并行化不能简单地通过增加进程数来提升性能，需要综合考虑计算和通信之间的权衡，确保计算量足够大时并行化才能真正发挥作用。  5.调试和测试的重要性：  并行程序的调试比串行程序更为复杂，因为需要同步多进程的执行过程。通过逐步调试，找出并行程序中的问题，并验证每个进程的计算结果是否与预期一致，这个过程帮助我提升了调试并行程序的能力。 | | | | | | | | | | | |
| 指导教师评语：  日期： | | | | | | | | | | | |