**《并行计算》课程实验报告**

**实验2：基于共享内存的并行程序设计**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 姓名 | 彭锦杰 | | 院系 | | 计算学部 | | | 学号 | | 2022112885 | |
| 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9月27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** | | | | | | | | | | | |
| 要求：需分析本次实验的基本目的，并综述你是如何实现这些目的的？  实验目的：  1. 掌握多线程编程技术，尤其是使用 PThread和OpenMP来实现并行计算。  2. 学习如何使用线程同步机制（如互斥锁）来解决并发访问共享数据时的竞争问题。  3. 使用蒙特卡洛方法和中值积分法计算π值，并通过并行化提升运算效率。  4. 理解多线程编程中线程安全、数据同步以及性能优化的基本概念。  实现方式：  为了实现上述目的，我采用了以下方法：  PThread 实现并行计算：  使用 pthread\_create() 创建多个线程，分配不同的任务（如计算π值或输出消息）。  使用互斥锁（pthread\_mutex\_t）保护共享变量，避免竞争条件。  通过 pthread\_join() 等待所有线程完成计算。  OpenMP 实现并行化：  在中值积分法和蒙特卡洛算法中，使用 OpenMP 的 #pragma omp parallel 和 #pragma omp for 来并行化循环结构，让多个线程同时进行计算。  使用 reduction 子句来安全地累加多个线程计算的结果，从而避免数据冲突。  线程同步机制的学习：  掌握了如何使用互斥锁、条件变量等机制，确保多个线程安全访问共享资源。  性能优化：  通过调整线程数量（如 OMP\_NUM\_THREADS）以及减少不必要的同步，提升了程序的运行效率。  使用 OpenMP 并行化 for 循环，充分利用多核处理器，减少了计算时间。 | | | | | | | | | | | |
| **二、实验内容** | | | | | | | | | | | |
| 该部分填写在实验过程中，你都完成了哪些工作。  1. PThread 多线程编程实验：  编写了 pthread\_hello\_world.c，并通过 pthread\_create() 创建多个线程，使每个线程输出 "Hello Thread" 信息。  使用 pthread\_join() 确保主线程等待所有线程完成工作，观察并记录线程的执行顺序，分析其随机性和不可预见性。  学习了如何通过互斥锁（pthread\_mutex\_t）保护共享资源，避免数据竞争。  2.使用 OpenMP 并行化计算π值：  通过蒙特卡洛算法计算π值，首先编写了串行版本的 pimonte\_serial.c，并使用随机数生成点的坐标，判断其是否落在单位圆内。  进一步，使用 OpenMP 将该算法并行化，编写 pimonte\_parallel.c，利用 #pragma omp parallel for 并行化生成随机点和圆内点计数的操作。  通过 reduction 子句安全地对各线程的结果进行累加。  3.中值积分法并行化计算π值：  编写了串行版本的 pi.c，通过中值积分法计算π值。  在串行代码的基础上，使用 OpenMP 对积分过程进行并行化，通过 #pragma omp parallel for 指令优化 for 循环，分配计算任务给多个线程，提升运算效率。  4.性能分析与优化：  对比串行和并行版本的执行时间，通过调整线程数量和优化数据同步（减少锁的使用）来提升计算性能。  学习了 OMP\_NUM\_THREADS 环境变量的设置，分析其对程序性能的影响。  本次实验的源代码如下：  **实验2.2**  **实验2.2.2 多线程程序实现π值的计算源代码：**  **calculate\_pi.c**  #include <stdio.h>  #include <pthread.h>  #include <sys/time.h>  #define NUM\_THREADS 4 // 设置线程数量  long long num\_steps = 1000000000;  double step;  double sum = 0.0; // 全局共享变量 sum  pthread\_mutex\_t mut; // 互斥量  // 每个线程的计算函数  void\* CalculatePi(void\* arg)  {  int id = \*(int\*)arg; // 线程id  double x, partial\_sum = 0.0;  long long start = id \* (num\_steps / NUM\_THREADS);  long long end = (id + 1) \* (num\_steps / NUM\_THREADS);  for (long long i = start; i < end; i++) {  x = (i + 0.5) \* step;  partial\_sum += 4.0 / (1.0 + x \* x); // 计算部分结果  }  pthread\_mutex\_lock(&mut); // 加锁，保护对共享变量 sum 的访问  sum += partial\_sum; // 将线程的部分结果加到 sum 中  pthread\_mutex\_unlock(&mut); // 解锁  return NULL;  }  int main()  {  pthread\_t threads[NUM\_THREADS];  int thread\_ids[NUM\_THREADS];  struct timeval TimeStampStart, TimeStampStop;  double ExeTime;  // 初始化互斥量  pthread\_mutex\_init(&mut, NULL);  // 计算步长  step = 1.0 / (double)num\_steps;  // 开始计时  gettimeofday(&TimeStampStart, NULL);  // 创建线程  for (int i = 0; i < NUM\_THREADS; i++) {  thread\_ids[i] = i;  pthread\_create(&threads[i], NULL, CalculatePi, &thread\_ids[i]);  }  // 等待所有线程完成  for (int i = 0; i < NUM\_THREADS; i++) {  pthread\_join(threads[i], NULL);  }  // 结束计时  gettimeofday(&TimeStampStop, NULL);  ExeTime = (double)(TimeStampStop.tv\_sec - TimeStampStart.tv\_sec) + (double)(TimeStampStop.tv\_usec - TimeStampStart.tv\_usec) \* 1e-6;  // 计算最终的 Pi 值  double pi = sum \* step;  // 销毁互斥量  pthread\_mutex\_destroy(&mut);  // 输出结果  printf("The value of PI is %15.12f\n", pi);  printf("The time to calculate PI was %f seconds\n", ExeTime);  return 0;  }  **实验2.2.3 使用并行化方法执行CountWords的源代码：**  **Parallel.c**  #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <stdbool.h>  #include <sys/time.h>  #define MAX\_LINE\_LENGTH 132 // 每行最大长度  FILE \*fd;  int TotalEvenWords = 0, TotalOddWords = 0, TotalWords = 0;  pthread\_mutex\_t mut; // 声明互斥量  char \*\*Lines; // 用于动态存储文件的每一行内容  int num\_lines = 0; // 总行数  int allocated\_lines = 1000; // 初始分配的行数  // 获取单词和字母数量  int GetWordAndLetterCount(char \*Line, int\* evenWords, int\* oddWords)  {  int Word\_Count = 0, Letter\_Count = 0;  for (int i = 0; i < MAX\_LINE\_LENGTH; i++)  {  if ((Line[i] != ' ') && (Line[i] != 0) && (Line[i] != '\n')) {  Letter\_Count++;  } else {  if (Letter\_Count % 2) {  (\*oddWords)++; // 局部增加奇数单词计数  } else {  (\*evenWords)++; // 局部增加偶数单词计数  }  Word\_Count++;  Letter\_Count = 0;  if (Line[i] == 0) break;  }  }  return Word\_Count;  }  // 每个线程处理一部分行  void\* ProcessLines(void\* arg)  {  int thread\_id = \*(int\*)arg;  int local\_evenWords = 0;  int local\_oddWords = 0;  int local\_totalWords = 0;    int lines\_per\_thread = num\_lines / 4; // 假设有4个线程  int start = thread\_id \* lines\_per\_thread;  int end = (thread\_id == 3) ? num\_lines : start + lines\_per\_thread; // 最后一个线程处理剩余的行  for (int i = start; i < end; i++) {  local\_totalWords += GetWordAndLetterCount(Lines[i], &local\_evenWords, &local\_oddWords);  }  // 统一加到全局变量中，减少锁的使用次数  pthread\_mutex\_lock(&mut); // 加锁，保护共享变量  TotalWords += local\_totalWords;  TotalEvenWords += local\_evenWords;  TotalOddWords += local\_oddWords;  pthread\_mutex\_unlock(&mut); // 解锁  return NULL;  }  int main()  {  // 动态分配内存用于存储行指针  Lines = (char \*\*)malloc(allocated\_lines \* sizeof(char \*));  for (int i = 0; i < allocated\_lines; i++) {  Lines[i] = (char \*)malloc(MAX\_LINE\_LENGTH \* sizeof(char));  }  // 打开文件  fd = fopen("./InFile1.txt", "r");  if (fd == NULL) {  printf("Error: Unable to open file.\n");  return -1;  }  // 初始化互斥量  pthread\_mutex\_init(&mut, NULL);  // 读取文件内容到内存，并动态调整行数大小  while (fgets(Lines[num\_lines], MAX\_LINE\_LENGTH, fd) != NULL) {  num\_lines++;  if (num\_lines >= allocated\_lines) {  // 动态扩展内存  allocated\_lines \*= 2;  Lines = (char \*\*)realloc(Lines, allocated\_lines \* sizeof(char \*));  for (int i = num\_lines; i < allocated\_lines; i++) {  Lines[i] = (char \*)malloc(MAX\_LINE\_LENGTH \* sizeof(char));  }  }  }  fclose(fd);  struct timeval TimeStampStart, TimeStampStop;  double ExeTime;  // 开始计时  gettimeofday(&TimeStampStart, NULL);  // 创建线程  int num\_threads = 4; // 假设使用4个线程  pthread\_t threads[num\_threads];  int thread\_ids[num\_threads];  for (int i = 0; i < num\_threads; i++) {  thread\_ids[i] = i;  pthread\_create(&threads[i], NULL, ProcessLines, &thread\_ids[i]);  }  // 等待所有线程完成  for (int i = 0; i < num\_threads; i++) {  pthread\_join(threads[i], NULL);  }  // 结束计时  gettimeofday(&TimeStampStop, NULL);  ExeTime = (double)(TimeStampStop.tv\_sec - TimeStampStart.tv\_sec) + (double)(TimeStampStop.tv\_usec - TimeStampStart.tv\_usec) \* 1e-6;  // 销毁互斥量  pthread\_mutex\_destroy(&mut);  // 输出结果  printf("Total Words = %8d\n", TotalWords);  printf("Total Even Words = %7d\nTotal Odd Words = %7d\n", TotalEvenWords, TotalOddWords);  printf("The time to read words was %f seconds\n", ExeTime);  // 释放内存  for (int i = 0; i < allocated\_lines; i++) {  free(Lines[i]);  }  free(Lines);  return 0;  }  **实验2.3**  **实验2.3.2 使用并行化方法对pi.c进行omp方式优化的源代码：**  **pi\_omp.c**  #include <stdio.h>  #include <sys/time.h>  #include <omp.h>  long long num\_steps = 1000000000; // 计算步数  double step;  int main(int argc, char\* argv[])  {  struct timeval TimeStampStart, TimeStampStop;  double ExeTime;  double x, pi, sum = 0.0;  int i;  // 计算步长  step = 1.0 / (double)num\_steps;  // 开始计时  gettimeofday(&TimeStampStart, NULL);  // 使用 OpenMP 并行化 for 循环，并通过 reduction 子句累加 sum  #pragma omp parallel for private(x) reduction(+:sum)  for (i = 0; i < num\_steps; i++)  {  x = (i + 0.5) \* step;  sum += 4.0 / (1.0 + x \* x); // 每个线程累加部分结果  }  // 计算 pi 的最终值  pi = sum \* step;  // 结束计时  gettimeofday(&TimeStampStop, NULL);  ExeTime = (double)(TimeStampStop.tv\_sec - TimeStampStart.tv\_sec) +  (double)(TimeStampStop.tv\_usec - TimeStampStart.tv\_usec) \* 1e-6;  // 输出结果  printf("The value of PI is %15.12f\n", pi);  printf("The time to calculate PI was %f seconds\n", ExeTime);  return 0;  }  **实验2.3.3使用并行化方法对pimonte\_serial.c进行omp方式优化的源代码：**  **pimonte\_parallel.c**  #include <stdio.h>  #include <stdlib.h>  #include <sys/time.h>  #include <time.h>  #include <omp.h>  #define BLOCK\_SIZE 500  int main() {  struct timeval TimeStampStart, TimeStampStop;  double ExeTime;  unsigned int iter = 200000000; // 迭代次数  int i, j;  double x, y;  double dUnderCurve = 0.0;  double pi = 0.0;  double r[BLOCK\_SIZE \* 2]; // 生成随机数的数组  // 获取开始时间  gettimeofday(&TimeStampStart, NULL);  // 并行执行，reduction 用于累加 dUnderCurve  #pragma omp parallel private(i, j, x, y, r) reduction(+:dUnderCurve)  {  unsigned int seed = time(NULL) ^ omp\_get\_thread\_num(); // 每个线程独立的随机数种子  // 生成随机数和计算  #pragma omp for  for (j = 0; j < iter / BLOCK\_SIZE; j++) {  // 创建随机数，生成 BLOCK\_SIZE\*2 个在 0.0-1.0 内均匀分布的随机数  for (i = 0; i < BLOCK\_SIZE \* 2; i++) {  r[i] = 0.0 + 1.0 \* rand\_r(&seed) / RAND\_MAX \* (1.0 - 0.0);  }  // 计算哪些点落在圆内  for (i = 0; i < BLOCK\_SIZE; i++) {  x = r[i]; // X 坐标  y = r[i + BLOCK\_SIZE]; // Y 坐标  if (x \* x + y \* y <= 1.0) { // 判断点是否落在圆内  dUnderCurve++;  }  }  }  }  // 计算π的值  pi = dUnderCurve / (double)iter \* 4;  // 获取结束时间  gettimeofday(&TimeStampStop, NULL);  ExeTime = (double)(TimeStampStop.tv\_sec - TimeStampStart.tv\_sec) +  (double)(TimeStampStop.tv\_usec - TimeStampStart.tv\_usec) \* 1e-6;  // 输出结果  printf("pi = %10.9f\n", pi);  printf("The time to calculate PI was %f seconds\n", ExeTime);  return 0;  } | | | | | | | | | | | |
| **三、实验结果** | | | | | | | | | | | |
| 实验二：  编译和执行pthread\_hello\_world.c 的实验结果：    编译和运行Pi.c的实验结果：    多线程程序实现π值的计算，编译和运行calculate\_pi.c的实验结果：    串行执行CountWords，编译和运行Serial.c的实验结果：    使用并行化方法执行CountWords，编译和运行Parallel.c的实验结果：  实验三：  编译和执行openmp\_hello\_world.c 的实验结果：    编译和执行pi.c的实验结果：    使用并行化方法对pi.c进行omp方式优化，编译和执行pi\_omp.c的实验结果：    编译和执行pimonte\_serial.c的实验结果：    使用并行化方法执行蒙特卡洛计算π值，编译和执行pimonte\_parallel.c的实验结果： | | | | | | | | | | | |
| **四、思考题** | | | | | | | | | | | |
| **思考题1：串行程序采用PThread进行线程化基本步骤？**  将串行程序转为使用 PThread 进行多线程化的基本步骤如下：  1.创建线程：使用 pthread\_create() 函数创建多个线程，指定每个线程的执行函数和参数。  2.编写线程函数：定义线程函数，该函数会在线程中并行执行。函数返回 void\* 类型的值。  3.等待线程完成：使用 pthread\_join() 等待每个线程执行完成并合并结果。  4.处理共享数据：如果线程需要共享数据，可以通过传递指针的方式给线程函数。对于共享资源，应该通过互斥锁等机制避免数据竞争。  5.销毁线程和资源：程序结束前使用 pthread\_exit() 来销毁线程，并销毁使用的互斥锁等同步资源。  **思考题2：PThread有几种共享数据互斥机制可以采用，如何应用？**  PThread 提供以下几种互斥机制来避免共享数据的竞争：  1.互斥锁 (Mutex)：使用 pthread\_mutex\_t 来确保同一时间只有一个线程能够访问共享数据。通过 pthread\_mutex\_lock() 加锁，pthread\_mutex\_unlock() 解锁。  应用：当多个线程同时需要访问同一个全局变量时，使用互斥锁保证数据一致性。  2.读写锁 (Read-Write Lock)：使用 pthread\_rwlock\_t，允许多个线程同时读取共享资源，但在写入时，其他线程不能进行读取或写入。  应用：适用于读操作远多于写操作的情况，比如配置文件读取。  3.信号量 (Semaphore)：使用 sem\_t，信号量用于控制对有限资源的访问，支持对资源计数。  应用：控制线程对有限资源（如多个文件、数据库连接）的并发访问。  4.条件变量 (Condition Variable)：使用 pthread\_cond\_t，条件变量允许线程等待某些条件发生再继续执行。  应用：用于线程之间的复杂同步，比如等待某些事件的发生。  **思考题3：OpenMP的主要功能，基本构成体有哪些？**  OpenMP 是用于并行编程的 API，主要功能是帮助程序员通过简单的指令对循环、任务等代码块进行并行化。它的主要构成体有：  1.并行区域 (#pragma omp parallel)：定义代码的并行区域，启动多个线程执行区域内的代码。  2.循环并行 (#pragma omp for)：用于并行化 for 循环，让多个线程分担循环的不同迭代。  任务 (#pragma omp task)：定义独立的任务，每个任务可并行执行。  3.同步机制：  #pragma omp critical：保证代码块在同一时刻只能有一个线程执行，用于共享资源的保护。  #pragma omp barrier：所有线程必须在此处等待，直到所有线程都到达此点。  4.数据共享：  shared：指定变量在所有线程中共享。  private：为每个线程创建该变量的副本。  reduction：将线程独立计算的结果合并到一个全局结果中。  **思考题4：试分析如何使用OpenMP实现多线程并行运算，提高系统运算效能，其引入环节应如何选取？**  使用 OpenMP 实现多线程并行运算可以通过以下步骤提高系统运算效能：  1.识别并行区域：找到程序中可以并行化的部分，比如 for 循环、任务分配等。  引入点：通常选择计算密集型的循环结构进行并行化，如大量数据的处理和矩阵计算等。  2.选择并行结构：根据任务类型选择适当的 OpenMP 结构。对于简单的循环，可以使用 #pragma omp for；对于复杂的任务分解，可以使用 #pragma omp task。  3.减少同步开销：尽量减少线程间的同步，比如减少 critical 区域的使用，避免锁竞争。  4.负载均衡：使用 OpenMP 的 schedule 子句来优化循环的负载均衡，避免线程空闲。  5.线程数调优：选择合适的线程数量，一般设置为处理器的核心数量，以最大化 CPU 的利用率。  **思考题5：对本次实验最开始时的pthread\_hello\_world.c修改后编译并多次运行程序，记录线程执行顺序，分析线程程序执行顺序是否不可预见及其产生原因。**  多次运行 pthread\_hello\_world.c 程序后会发现线程的执行顺序是不可预见的。这是因为：  1.线程调度的非确定性：操作系统的线程调度器以非确定性方式分配 CPU 时间片给不同的线程，这意味着线程的执行顺序不固定，可能会由于 CPU 负载、线程优先级和其他运行的进程等因素而有所变化。  2.多核处理器：在多核系统中，线程可能被分配到不同的 CPU 核心，这也会影响线程的执行顺序。 因此，PThread 程序的执行顺序通常是不可预见的，取决于系统的线程调度策略。  **思考题6：根据你的实验，OMP\_NUM\_THREADS是不是设置得越大越好？说明理由。**  OMP\_NUM\_THREADS 并不是设置得越大越好，原因如下：  1.CPU 核心限制：每个线程需要占用一个 CPU 核心，如果线程数超过物理核心数，线程会在核心之间进行上下文切换，导致性能下降。  2.过多的线程开销：每个线程的创建、销毁和上下文切换都有开销，过多的线程会增加系统开销，反而导致性能降低。  3.负载不均衡：如果任务的负载分配不均衡，可能会导致某些线程空闲，造成资源浪费。 | | | | | | | | | | | |
| **五、实验心得体会** | | | | | | | | | | | |
| 1.多线程编程的必要性：在处理大量计算任务时，单线程的串行程序往往会因资源利用率低而效率不高。多线程编程可以有效利用多核 CPU 资源，显著提升程序的执行效率，尤其在计算密集型任务中，如本次实验的π值计算。  2.线程同步与数据竞争：在多线程编程中，线程之间共享数据时容易出现数据竞争，导致结果不一致或错误。通过使用互斥锁和其他同步机制（如 critical 区域和 reduction 子句），可以确保线程安全，但也要注意同步带来的性能开销。  3.PThread 与 OpenMP 的对比：PThread 是底层的线程管理机制，能够提供更精细的线程控制，但相对复杂；而 OpenMP 更加高层次、易于使用，能够通过简单的指令快速实现并行化，适合处理大量循环结构的任务。两者各有优缺点，适用场景也不同。  4.线程调度的不可预见性：通过对 pthread\_hello\_world.c 程序的实验，我了解到操作系统的线程调度是非确定性的，线程的执行顺序不可预见，这对于并发编程的调试和优化提出了挑战。  5.并行化并非总是越多线程越好：通过设置不同的 OMP\_NUM\_THREADS 并观察程序性能，我发现线程数并不是越多越好，超过系统的物理核心数反而会引入额外的上下文切换开销，导致性能下降。因此，合理选择线程数是提升性能的关键。 | | | | | | | | | | | |
| 指导教师评语：  日期： | | | | | | | | | | | |