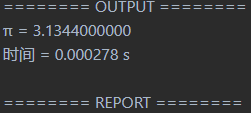
编程作业3.2：

**编程题**

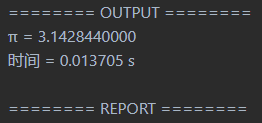
1. 程序代码（填入如下表格）

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <mpi.h>  #include <math.h>  void Get\_input(long long int\* number\_of\_tosses, int my\_rank, MPI\_Comm comm) {  if (my\_rank == 0) {  // printf("投掷飞镖次数: ");  scanf("%lld", number\_of\_tosses);  }  MPI\_Bcast(number\_of\_tosses, 1, MPI\_LONG\_LONG\_INT, 0, comm);  }  long long int Monte\_carlo(long long number\_of\_tosses, int my\_rank) {  long long int local\_number\_in\_circle = 0;  unsigned int seed = my\_rank; // Seed for random number generator  for (long long int i = 0; i < number\_of\_tosses; i++) {  double x = (rand\_r(&seed) / (double)RAND\_MAX) \* 2.0 - 1.0;  double y = (rand\_r(&seed) / (double)RAND\_MAX) \* 2.0 - 1.0;  if (x \* x + y \* y <= 1.0) {  local\_number\_in\_circle++;  }  }  return local\_number\_in\_circle;  }  int main(void) {  int comm\_sz, my\_rank;  long long int number\_of\_tosses, local\_tosses;  long long int local\_number\_in\_circle, total\_number\_in\_circle;  double pi\_estimate;  double start\_time, end\_time, local\_time, total\_time;  // 初始化 MPI 环境  MPI\_Init(NULL, NULL);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);  // 获取输入并广播  Get\_input(&number\_of\_tosses, my\_rank, MPI\_COMM\_WORLD);  local\_tosses = number\_of\_tosses / comm\_sz;  // 开始计时  MPI\_Barrier(MPI\_COMM\_WORLD); // 确保所有进程同步后再计时  start\_time = MPI\_Wtime();  // 执行 Monte Carlo 计算  local\_number\_in\_circle = Monte\_carlo(local\_tosses, my\_rank);  // 规约结果  MPI\_Reduce(&local\_number\_in\_circle, &total\_number\_in\_circle, 1, MPI\_LONG\_LONG\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);  // 结束计时  end\_time = MPI\_Wtime();  local\_time = end\_time - start\_time;  // 获取全局最大时间（最慢进程时间）  MPI\_Reduce(&local\_time, &total\_time, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);  // 打印结果  if (my\_rank == 0) {  pi\_estimate = 4.0 \* total\_number\_in\_circle / number\_of\_tosses;  printf("π = %.15f\n", pi\_estimate);  printf("并行运行时间 %.6f seconds\n", total\_time);  }  // 释放资源  MPI\_Finalize();  return 0;  } |
|  |

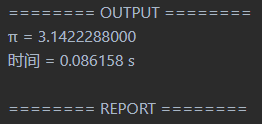
运行结果截图：



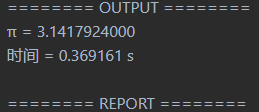
图一 问题规模（104 ）进程数（1）



图二 问题规模（106 ）进程数（2）



图三 问题规模（107 ）进程数（4）



图四 问题规模（108 ）进程数（8）

1. 程序运行结果

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| comm\_sz/进程数 | 问题规模 | | | |
| 104 | 106 | 107 | 108 |
| 1 | π = 3.1344000000 时间 = 0.000278s | π = 3.1419560000 时间 = 0.027075s | π = 3.1421176000 时间 = 0.279910s | π = 3.1416637600 时间 = 2.760553s |
| 2 | π = 3.1360000000 时间 = 0.000172s | π = 3.1428440000 时间 = 0.013705s | π = 3.1416200000 时间 = 0.141394s | π = 3.1416459200 时间 = 1.309626s |
| 4 | π = 3.1248000000 时间 = 0.000099s | π = 3.1428160000 时间 = 0.007022s | π = 3.1422288000 时间 = 0.086158s | π = 3.1415504400 时间 = 0.677212s |
| 8 | π = 3.1308000000 时间 = 0.000074s | π = 3.1440040000 时间 = 0.005216s | π = 3.1415640000 时间 = 0.047603s | π = 3.1417924000 时间 = 0.369161s |

1. 结果分析
2. 加速比分析

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| comm\_sz/进程数 | 问题规模 | | | |
| 104 | 106 | 107 | 108 |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1.62 | 1.97 | 1.98 | 2.11 |
| 4 | 2.81 | 3.85 | 3.24 | 4.08 |
| 8 | 3.75 | 5.19 | 5.85 | 7.48 |

1. 效率分析

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| comm\_sz/进程数 | 问题规模 | | | |
| 104 | 106 | 107 | 108 |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 0.81 | 0.99 | 0.99 | 1.06 |
| 4 | 0.70 | 0.96 | 0.81 | 1.02 |
| 8 | 0.47 | 0.65 | 0.73 | 0.93 |

1. 扩展性分析

分析：

* **加速比**：随着进程数的增加，加速比逐渐提高，但也会受限于通信开销，尤其在小规模问题下。大规模问题（10^8）下加速比接近理论最大值。
* **效率**：在增加进程数时，效率略有下降，但对于较大问题规模，效率仍保持在一个较高的水平。

结论：

* **在大问题规模时，拓展性较好**：随着进程数的增加，运行时间显著降低，计算资源得到了充分利用。
* **在小问题规模时，通信开销成为瓶颈**：增加进程数带来的性能提升较小，效率下降。
* **拓展性受限于通信开销和负载均衡**：在更大规模的计算中，通信开销对程序性能的影响将变得更加显著，且需要确保负载均衡以优化拓展性。

编程作业3.3：

**编程题**

1. 程序代码（填入如下表格）

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// 获取高精度时间戳

double Get\_time() {

return MPI\_Wtime();

}

// 树形结构通信实现全局和

int Global\_sum(int\* my\_ints, int num\_ints, int my\_rank, int comm\_sz, MPI\_Comm comm) {

int phase = 0;

int sum = 0;

// 计算本进程的部分和

for (int i = 0; i < num\_ints; i++) {

sum += my\_ints[i];

}

int partner;

while ((1 << phase) < comm\_sz) {

if (my\_rank % (1 << (phase + 1)) == 0) {

// 如果当前进程是接收进程

partner = my\_rank + (1 << phase);

if (partner < comm\_sz) {

int temp;

MPI\_Recv(&temp, 1, MPI\_INT, partner, 0, comm, MPI\_STATUS\_IGNORE);

sum += temp;

}

} else {

// 如果当前进程是发送进程

partner = my\_rank - (1 << phase);

MPI\_Send(&sum, 1, MPI\_INT, partner, 0, comm);

break; // 发送数据后退出循环

}

phase++;

}

return sum; // 返回部分和或全局和

}

int main(void) {

int my\_rank, comm\_sz;

int\* my\_ints;

int total\_num\_ints, num\_ints\_per\_process, global\_sum;

MPI\_Comm comm;

MPI\_Init(NULL, NULL); // 初始化 MPI 环境

MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz); // 获取进程总数

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank); // 获取当前进程号

comm = MPI\_COMM\_WORLD;

// 仅 0 号进程读取问题规模（即总随机数的数量）

if (my\_rank == 0) {

printf("请输入总随机数的数量：");

scanf("%d", &total\_num\_ints); // 输入总随机数数量

}

// 广播问题规模给所有进程

MPI\_Bcast(&total\_num\_ints, 1, MPI\_INT, 0, comm);

// 每个进程负责处理一部分随机数

// 确保总数可以均匀分配给每个进程

if (total\_num\_ints % comm\_sz != 0) {

if (my\_rank == 0) {

printf("警告：总随机数数量不能被进程数整除，结果可能不正确。\n");

}

}

// 每个进程分配生成的随机数数量

num\_ints\_per\_process = total\_num\_ints / comm\_sz;

// 每个进程动态分配数组，生成多个随机整数

my\_ints = malloc(num\_ints\_per\_process \* sizeof(int));

srand(time(NULL) + my\_rank); // 保证每个进程生成不同的随机数

for (int i = 0; i < num\_ints\_per\_process; i++) {

my\_ints[i] = rand() % 100 + 1; // 生成随机数，范围是 1 到 100

}

// 开始计时

double start\_time = Get\_time();

// 计算全局和

global\_sum = Global\_sum(my\_ints, num\_ints\_per\_process, my\_rank, comm\_sz, comm);

// 结束计时

double end\_time = Get\_time();

// 0号进程收集并打印结果

if (my\_rank == 0) {

int\* all\_ints = malloc(total\_num\_ints \* sizeof(int)); // 动态分配数组，存储所有进程的随机数

MPI\_Gather(my\_ints, num\_ints\_per\_process, MPI\_INT, all\_ints, num\_ints\_per\_process, MPI\_INT, 0, comm);

printf("全局和：%d\n", global\_sum); // 打印全局和

printf("并行运行时间：%f 秒\n", end\_time - start\_time); // 打印并行运行时间

free(all\_ints);

} else {

MPI\_Gather(my\_ints, num\_ints\_per\_process, MPI\_INT, NULL, 0, MPI\_INT, 0, comm);

}

free(my\_ints); // 释放本进程的随机数数组

MPI\_Finalize(); // 释放 MPI 资源

return 0;

}

运行结果截图：



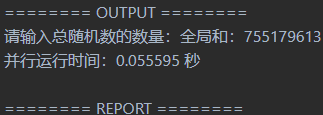
图五 问题规模（104 ）进程数（1）



图六 问题规模（106 ）进程数（2）



图七 问题规模（107 ）进程数（4）



图八 问题规模（108 ）进程数（4）

1. 程序运行结果

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| comm\_sz/进程数 | 问题规模 | | | |
| 104 | 106 | 107 | 108 |
| 1 | 0.000041s | 0.004089s | 0.041181s | 0.412617s |
| 2 | 0.000030s | 0.002077s | 0.017293s | 0.191885s |
| 4 | 0.000022s | 0.001141s | 0.010085s | 0.087296s |
| 8 | 0.000031s | 0.000493s | 0.005888s | 0.051269s |

运行时间

1. 结果分析
2. 加速比分析

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| comm\_sz/进程数 | 问题规模 | | | |
| 104 | 106 | 107 | 108 |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1.37 | 1.97 | 2.38 | 2.15 |
| 4 | 1.86 | 3.59 | 4.08 | 4.73 |
| 8 | 1.32 | 8.29 | 7.00 | 8.05 |

1. 效率分析

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| comm\_sz/进程数 | 问题规模 | | | |
| 104 | 106 | 107 | 108 |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 0.685 | 0.985 | 1.19 | 1.08 |
| 4 | 0.465 | 0.898 | 1.02 | 1.18 |
| 8 | 0.165 | 1.04 | 0.88 | 1.01 |

1. 扩展性分析
2. 加速比增长的趋势：

* 从表格来看，随着问题规模的增大，程序在增加进程数时的加速比有所增长，特别是在 108 这个大规模问题中，加速比接近理论最佳（8.05），这表明程序在大规模数据上具备良好的扩展性。

1. 效率变化趋势：

* 对于小规模问题（如 104），效率下降较快，尤其是随着进程数的增加，通信开销逐渐占据更大的比重。
* 对于大规模问题（如 108），效率基本稳定，甚至随着进程数的增加，效率有所上升。这表明在大规模问题上，程序的扩展性较好，能够有效利用多进程进行计算。

结论：

1. 小问题规模（104）：扩展性差，增加进程数对性能提升的影响有限。通信开销成为主要瓶颈。
2. 中等问题规模（106 和 107）：随着问题规模增大，增加进程数会带来明显的加速比提升，但效率会略微下降。
3. 大问题规模（108）：扩展性最佳，增加进程数时，加速比接近理论最佳，效率接近 1，表明程序能够充分利用多进程资源，通信开销对性能的影响较小。