

基于蒙特卡洛法算圆的面积

诸晓婉(922110800509)

张雨馨(923104780210)

拓欣(922114740127)

2024 年 12 月

1 介绍

蒙特卡罗方法, 也称统计模拟方法, 是 1940 年代中期由于科学技术的发展和电子计算机的发明, 而提出的一种以概率统计理论为指导的数值计算方法. 使用随机数来解决很多计算问题的方法

2 代码实现

2.1 串行

```
1  program normal
2      use, intrinsic :: iso_fortran_env, only: dp => real64
3      implicit none
4
5      real(dp) :: x, y, area, true_area, dis
6      real(dp) :: r
7      real(dp) :: start_cpu, end_cpu
8      integer :: start_time, end_time
9      integer :: i, inside_points, samples
10
11     ! 参数
12     r = 1.0_dp
13     samples = 1000000
14     inside_points = 0
15     true_area = r*r*3.141592653589793_dp
16
17     ! 开始时间
18     call cpu_time(start_cpu)
19     call system_clock(count=start_time)
20
21     ! 蒙特卡洛
22     do i = 1, samples
23         call random_number(x)
24         call random_number(y)
25         x = x*r*2.0_dp - r
26         y = y*r*2.0_dp - r
27
28         if (x*x + y*y <= r*r) then
29             inside_points = inside_points + 1
30         end if
```

```

31  end do
32
33  ! 圆面积
34  area = (real(inside_points, dp)/real(samples, dp))*(4.0_dp*r*r)
35  dis = abs(area - true_area)
36
37  ! 结束时间
38  call cpu_time(end_cpu)
39  call system_clock(count=end_time)
40
41  ! 结果
42  print '(A,F15.8)', '计算结果(面积): ', area
43  print '(A,F15.8)', '半径: ', r
44  print '(A,I10)', '样本量: ', samples
45  print '(A,F15.8)', '结果精度(误差): ', dis
46  print '(A,F15.8,A)', 'Wall Time: ', (end_time - start_time)/1000.0_dp, '秒'
47  print '(A,F15.8,A)', 'CPU Time: ', (end_cpu - start_cpu), '秒'
48  end program normal

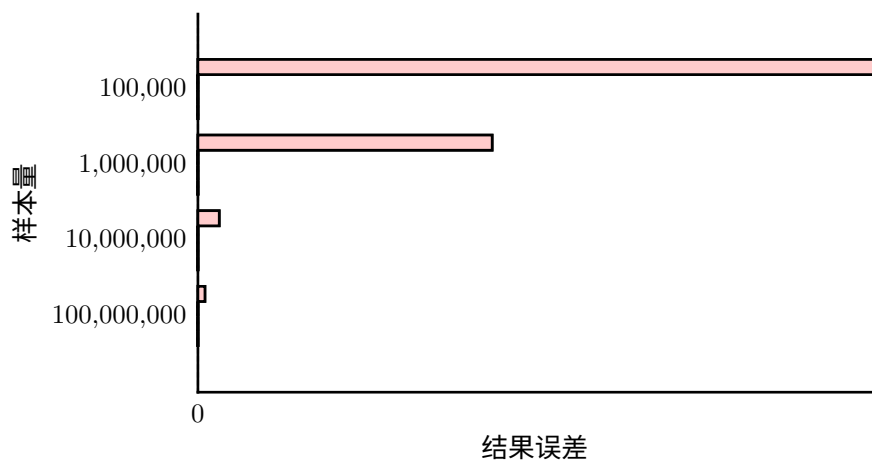
```

编译指令

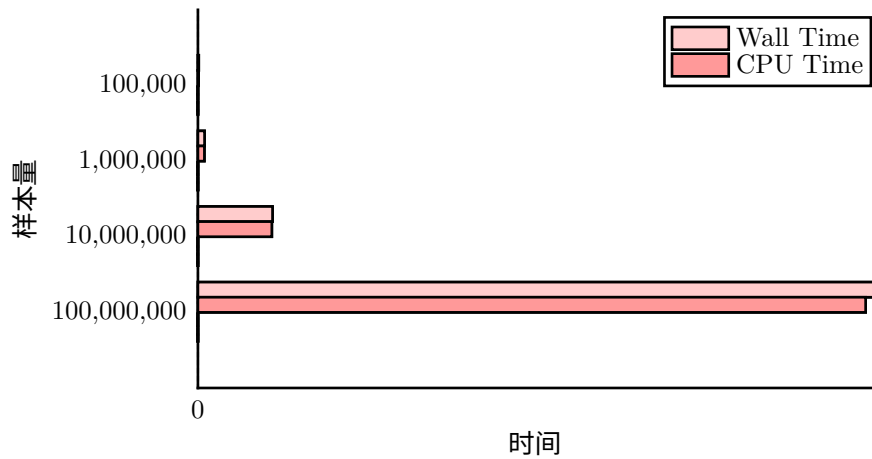
```
1 gfortran -O3 -march=native src/normal.f90 -o dist/normal
```

Shell

不同样本量下结果误差的变化(值越小越好)



不同样本量下 Wall Time/CPU Time 的变化



2.2 OpenMP

```

1  program omp
2      use, intrinsic :: iso_fortran_env, only: dp => real64
3      implicit none
4
5      real(dp) :: x, y, area, true_area, dis
6      real(dp) :: r
7      real(dp) :: start_cpu, end_cpu
8      integer :: start_time, end_time
9      integer :: i, inside_points, samples
10     integer :: tid
11
12     ! 参数
13     r = 1.0_dp
14     samples = 1000000
15     inside_points = 0
16     true_area = r*r*3.141592653589793_dp
17
18     ! 开始时间
19     call cpu_time(start_cpu)
20     call system_clock(count=start_time)
21
22     !$omp parallel private(x, y, tid) reduction(+:inside_points)
23     !$omp do
24     do i = 1, samples
25         call random_number(x)
26         call random_number(y)
27         x = x*r*2.0_dp - r
28         y = y*r*2.0_dp - r
29
30         if (x*x + y*y <= r*r) then
31             inside_points = inside_points + 1
32         end if
33     end do
34     !$omp end do

```

```

35     !$omp end parallel
36
37     ! 圆面积
38     area = (real(inside_points, dp)/real(samples, dp))*(4.0_dp*r*r)
39     dis = abs(area - true_area)
40
41     ! 结束时间
42     call cpu_time(end_cpu)
43     call system_clock(count=end_time)
44
45     ! 结果
46     print '(A,F15.8)', '计算结果(面积): ', area
47     print '(A,F15.8)', '半径: ', r
48     print '(A,I10)', '样本量: ', samples
49     print '(A,F15.8)', '结果精度(误差): ', dis
50     print '(A,F15.8,A)', 'Wall Time: ', (end_time - start_time)/1000.0_dp, '秒'
51     print '(A,F15.8,A)', 'CPU Time: ', (end_cpu - start_cpu), '秒'
52 end program omp

```

编译指令

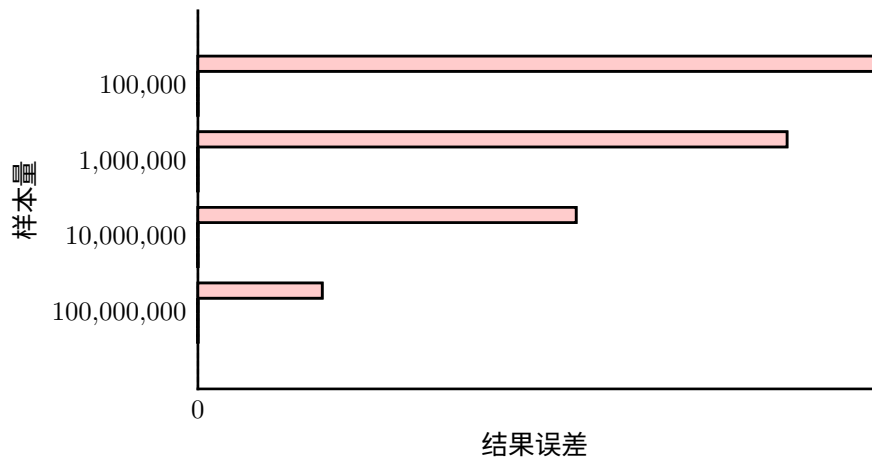
```

1 gfortran -O3 -march=native -openmp src/omp.f90 -o dist/omp-g
2 mpiifx -O3 -march=native -qopenmp -qmkl src/omp.f90 -o dist/omp-i

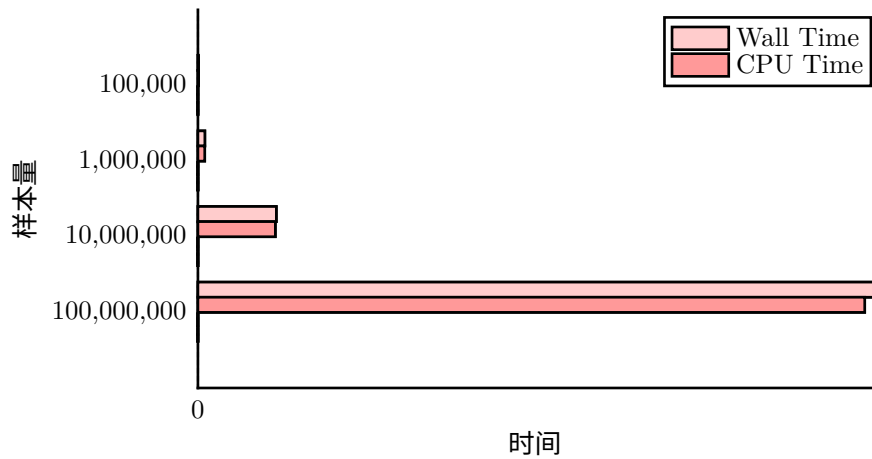
```

Shell

不同样本量下结果误差的变化(值越小越好)



不同样本量下 Wall Time/CPU Time 的变化



2.3 OpenMP+MPI

```

1  program hybrid
2      use, intrinsic :: iso_fortran_env, only: dp => real64
3      use mpi
4      implicit none
5
6      real(dp) :: x, y, pi, area, true_area, dis
7      real(dp) :: r
8      real(dp) :: start_cpu, end_cpu
9      integer :: start_time, end_time
10     integer :: i, local_inside, global_inside, samples, local_points
11     integer :: ierr, rank, size
12
13     ! MPI
14     call MPI_Init(ierr)
15     call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
16     call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
17
18     ! 参数
19     r = 1.0_dp
20     samples = 1000000
21     local_points = samples/size
22     local_inside = 0
23     true_area = r*r*3.141592653589793_dp
24
25     ! 开始时间
26     if (rank == 0) then
27         call cpu_time(start_cpu)
28         call system_clock(count=start_time)
29     end if
30
31     !$omp parallel private(x, y) reduction(+:local_inside)
32     !$omp do
33     do i = 1, local_points
34         call random_number(x)

```

```

35      call random_number(y)
36      x = x*r*2.0_dp - r
37      y = y*r*2.0_dp - r
38
39      if (x*x + y*y <= r*r) then
40          local_inside = local_inside + 1
41      end if
42  end do
43  !$omp end do
44  !$omp end parallel
45
46  ! 汇总所有进程的结果
47  call MPI_Reduce(local_inside, global_inside, 1, MPI_INTEGER, &
48                  MPI_SUM, 0, MPI_COMM_WORLD, ierr)
49
50  if (rank == 0) then
51      ! 圆面积
52      area = (real(global_inside, dp)/real(samples, dp))*(4.0_dp*r*r)
53      dis = abs(area - true_area)
54
55      ! 结束时间
56      call cpu_time(end_cpu)
57      call system_clock(count=end_time)
58
59      ! 结果
60      print '(A,F15.8)', '计算结果(面积): ', area
61      print '(A,F15.8)', '半径: ', r
62      print '(A,I10)', '样本量: ', samples
63      print '(A,F15.8)', '结果精度(误差): ', dis
64      print '(A,F15.8,A)', 'Wall Time: ', (end_time - start_time)/1000.0_dp, '秒'
65      print '(A,F15.8,A)', 'CPU Time: ', (end_cpu - start_cpu), '秒'
66  end if
67
68  call MPI_Finalize(ierr)
69 end program hybrid

```

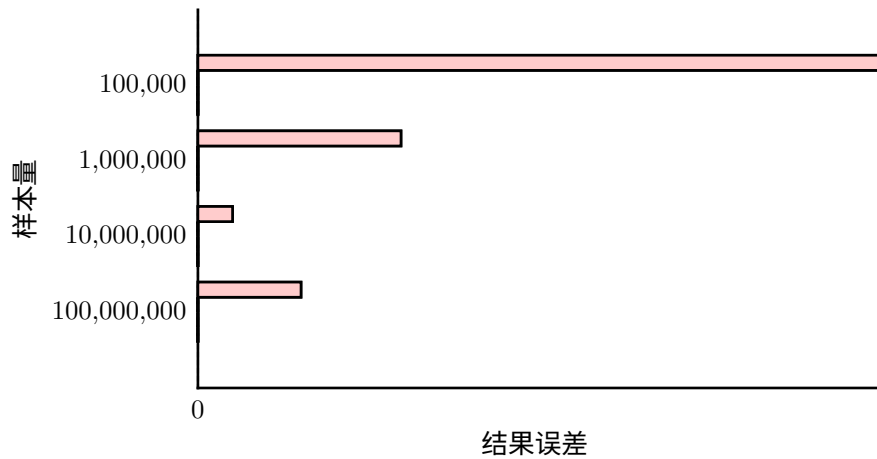
编译指令

```

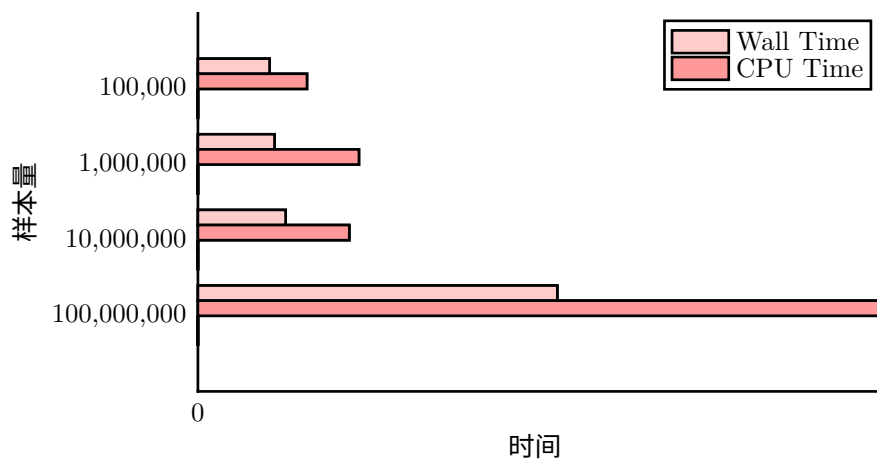
1 mpif90 -O3 -fopenmp src/hybrid.f90 -o dist/hybrid-g
2 mpirun -np 8 ./dist/hybrid-g
3
4 mpiifx -O3 -march=native -qopenmp -lmpi -qmk1 src/hybrid.f90 -o dist/hybrid-i && \
5 mpirun -np 8 ./dist/hybrid-i"

```

不同样本量下结果误差的变化(值越小越好)



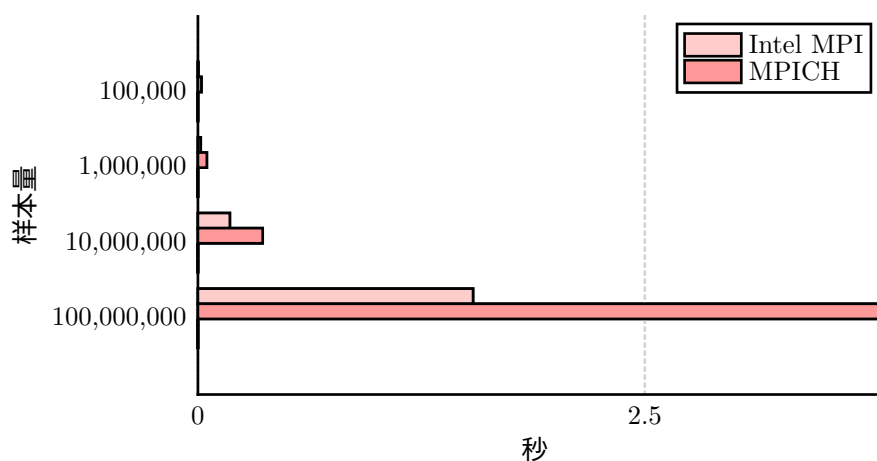
不同样本量下 Wall Time/CPU Time 的变化



2.4 MPICH/BLAS vs Intel MPI/MKL

这里我们分别使用 GFortran 编译器和 Intel Fortran 编译器来对比两者的性能

不同样本量下执行速度(CPU Time)的变化



3 项目展望

因为时间和技术的限制, 我们只实现了串行、OpenMP 和 OpenMP+MPI 三种版本, 但是还有很多其他的优化方法可以尝试

3.1 Coarray

Coarray 是 Fortran 2008 的一个特性, 可以实现分布式内存并行计算, 无需自己编写 OpenMP、MPI 等代码, 但是需要编译器支持

3.2 使用 Julia 等更热门的语言

3.2.1 Julia

Julia 是一个高性能的动态编程语言, 有很多并行计算的库, 可以实现分布式内存并行计算

- Threads.jl
- Distributed.jl

3.2.2 Python

Python 虽然是一个解释型语言, 但是有很多高性能的库, 可以实现并行计算

- Cython
- Ray

3.2.3 Rust

Rust 是一门近年来非常流行的系统编程语言, 有一些高性能的库可以实现并行计算

- Rayon

3.3 使用 GPU 加速

我们的显卡擅长浮点运算, 可以使用 GPU 加速来提高计算速度

而 NVIDIA 公司的 CUDA 是一个非常流行的 GPU 编程框架, 可以使用 CUDA C/C++、CUDA Python、CUDA Fortran 等语言来编写 GPU 程序

- NVIDIA CUDA Fortran

3.4 更高级的算法

- 分层采样 把蒙特卡洛法进一步微分, 把一个任务拆分成了数个任务
- 重要性采样 需要一定的概论相关的数学知识
- 自适应采样 通过一定的算法/机器学习的方法来自适应的调整采样的方法
- 拟蒙特卡洛序列
 - Sobol 序列
 - Halton 序列
 - Faure 序列