****

中国地质大学（北京）

高性能计算导论

实验报告

学 院：信息工程学院

专 业：计算机科学与技术

班 级：10041811

学 号：1005183121

姓 名：周子杰

联系方式：18006163783

邮 箱：2476331552@qq.com

指导老师：王玉柱

日 期：2020年12月6日

目录

[基于MPI和OpenMP的并行程序设计与实现 3](#_Toc57300922)

[一、实验目的和要求 3](#_Toc57300923)

[二、实验内容 3](#_Toc57300924)

[（1）使用SPMD编程模式编写求解MPI程序。 3](#_Toc57300925)

[（2）编写求解OpenMP程序 4](#_Toc57300926)

[<2.1>并行域并行求解 4](#_Toc57300927)

[<2.2>使用for循环制导计算 6](#_Toc57300928)

[<2.3>使用带reduction子句的for循环制导 7](#_Toc57300929)

[<2.4>通过private子句和critical制导计算 8](#_Toc57300930)

[（3）基于MPI+OpenMP编写求解程序。 9](#_Toc57300931)

[（4）设计并实现矩阵乘的并行算法，分析所设计的并行算法的性能，比如加速比、并行效率和可扩展性。（选做） 11](#_Toc57300932)

[（5）MPI与OpenMP有什么区别？谈谈对高性能计算的理解和认识。 13](#_Toc57300933)

# 基于MPI和OpenMP的并行程序设计与实现

# 一、实验目的和要求

1. 掌握高性能计算相关基础知识和并行程序设计的方法。
2. 掌握Linux系统的安装与使用。
3. 掌握Linux下并行计算环境的搭建与使用。
4. 掌握基于MPI的并行编程方法。
5. 掌握基于OpenMP的并行编程方法。
6. 掌握MPI+OpenMP的并行编程方法。
7. 练习课件第5、6章的相关例子。

# 二、实验内容

## （1）使用SPMD编程模式编写求解MPI程序。

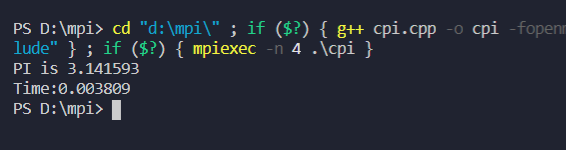
分析：

这是根据PPT上的串行代码改的MPI代码，实际上就是通过不断的计算，然后将各个线程运算的结果记录在mypi中，并通过归约的方式传入pi并对多个线程的结果作累加（MPI\_SUM）。运行时间很短，这里我们将循环次数都设为1000000，方便下面的对比。

#### 代码：

|  |
| --- |
| #include <stdio.h>  #include <mpi.h>  const *int* n = 1000000;  *int* main(*int* *argc*, *char* \**argv*[])  {  *int* my\_rank, num\_procs;  *long* i;  *double* w, local, mypi = 0.0, pi;  *double* start = 0.0, stop = 0.0;      MPI\_Init(&*argc*, &*argv*);      MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_procs);      MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);      if (my\_rank == 0)          start = MPI\_Wtime();      w = 1.0 / n;      for (i = my\_rank; i < n; i += num\_procs)      {          local = (i + 0.5) \* w;          mypi += 4.0 / (1.0 + local \* local);      }      mypi \*= w;      MPI\_Reduce(&mypi, &pi, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);      if (my\_rank == 0)      {          printf("PI is %\n", pi);          stop = MPI\_Wtime();          printf("Time:%f\n", stop - start);          fflush(stdout); //清空缓冲区      }      MPI\_Finalize();      return 0;  } |

#### 运行结果



## （2）编写求解OpenMP程序

### <2.1>并行域并行求解

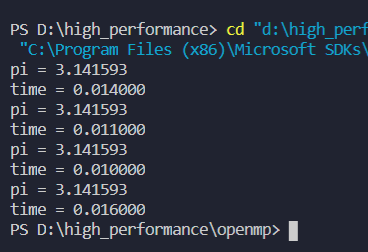
分析：

和上面的MPI的方式类似，实际只不过是换成openmp的样子而已，最后作个累加即可。不过运行时间比mpi多了。

代码：

|  |
| --- |
| #include <omp.h>  #include <stdio.h>  static *long* num\_steps = 1000000;  *double* step;  #define NUM\_THREADS 2  *int* main(*int* *argc*, *char* \**argv*[])  {  *int* i;  *double* x, pi, sum[NUM\_THREADS];  *double* start = 0.0, stop = 0.0;      step = 1.0 / (*double*)num\_steps;      omp\_set\_num\_threads(NUM\_THREADS);      start = omp\_get\_wtime();  #pragma omp parallel private(i)      {  *double* x;  *int* id;          id = omp\_get\_thread\_num();          for (i = id, sum[id] = 0.0; i < num\_steps; i += NUM\_THREADS)          {              x = (i + 0.5) \* step;              sum[id] += 4.0 / (1.0 + x \* x);          }      }      for (i = 0, pi = 0.0; i < NUM\_THREADS; ++i)          pi += sum[i] \* step;      stop = omp\_get\_wtime();      printf("pi = %lf\ntime = %lf\n", pi, stop-start);      return 0;  } |

运行结果：



### <2.2>使用for循环制导计算

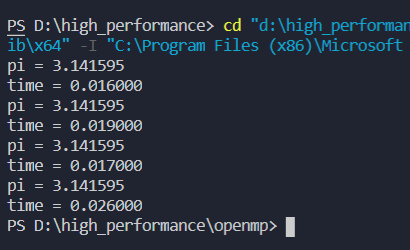
分析：

加入了for循环制导之后，系统能够自己去将各个任务分配给各个线程，我们的循环里头只需从0到num\_steps不停运算即可。运行时间与for循环类似。

代码：

|  |
| --- |
| #include <omp.h>  #include <stdio.h>  static *long* num\_steps = 100000;  *double* step;  #define NUM\_THREADS 4  *int* main(*int* *argc*, *char* \**argv*[])  {  *int* i, id;  *double* x, pi, sum[NUM\_THREADS];  *double* start = 0.0, stop = 0.0;      step = 1.0 / (*double*)num\_steps;      omp\_set\_num\_threads(NUM\_THREADS);      start = omp\_get\_wtime();  #pragma omp parallel private(x, id)      {          id = omp\_get\_thread\_num();          sum[id] = 0;  #pragma omp for          for (i = 0; i <= num\_steps; ++i)          {              x = (i + 0.5) \* step;              sum[id] += 4.0 / (1.0 + x \* x);          }      }      for (i = 0, pi = 0.0; i < NUM\_THREADS; ++i)          pi += sum[i] \* step;      stop = omp\_get\_wtime();      printf("pi = %lf\ntime = %lf\n", pi, stop-start);      return 0;  } |

运行结果：



### <2.3>使用带reduction子句的for循环制导

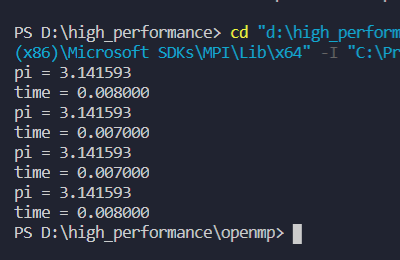
分析：

有了reduction之后，我们就可以删去上面代码中求和的部分，因为reduction中有sum这个操作，它会自动帮我们求和。运行时间大大缩短，但还是不如mpi。

代码：

|  |
| --- |
| #include <omp.h>  #include <stdio.h>  static *long* num\_steps = 100000;  *double* step;  #define NUM\_THREADS 2  *int* main(*int* *argc*, *char* \**argv*[])  {  *int* i;  *double* x = 0.0, pi = 0.0, sum = 0.0;  *double* start = 0.0, stop = 0.0;      step = 1.0 / (*double*)num\_steps;      omp\_set\_num\_threads(NUM\_THREADS);      start = omp\_get\_wtime();  #pragma omp parallel for reduction(+ : sum) private(x)      for (i = 1; i <= num\_steps; ++i)      {          x = (i - 0.5) \* step;          sum += 4.0 / (1.0 + x \* x);      }      pi += sum \* step;      stop = omp\_get\_wtime();      printf("pi = %lf\ntime = %lf\n", pi, stop-start);      return 0;  } |

运行结果：



### <2.4>通过private子句和critical制导计算

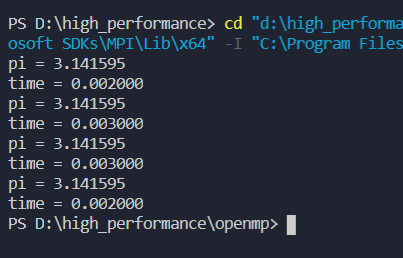
分析：

private是用来申明私有变量的，在并行计算中，如果参与运算的变量不声明为私有，那么可能线程的运算会相互干扰。而critical的加入能够用来保护对共享变量的修改。运行时间甚至低于mpi。

代码：

|  |
| --- |
| #include <omp.h>  #include <stdio.h>  static *long* num\_steps = 100000;  *double* step;  #define NUM\_THREADS 4  *int* main(*int* *argc*, *char* \**argv*[])  {  *int* i, id;  *double* x, pi, sum;  *double* start = 0.0, stop = 0.0;      step = 1.0 / (*double*)num\_steps;      omp\_set\_num\_threads(NUM\_THREADS);      start = omp\_get\_wtime();  #pragma omp parallel private(id, i, x, sum)      {          id = omp\_get\_thread\_num();          for (i = id, sum = 0.0; i <= num\_steps; i += NUM\_THREADS)          {              x = (i + 0.5) \* step;              sum += 4.0 / (1.0 + x \* x);          }  #pragma omp critical          pi += sum;      }      pi \*= step;      stop = omp\_get\_wtime();      printf("pi = %lf\ntime = %lf\n", pi, stop-start);      return 0;  } |

运行结果：



## （3）基于MPI+OpenMP编写求解程序。

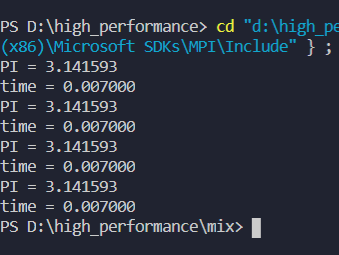
分析：

这里呢是基于<2.4>进行的一个修改，将计算求和的工作交给openmp，而传输数据的工作交给mpi。运行时间和带reduction的for循环制导类似。

代码：

|  |
| --- |
| #include <stdio.h>  #include <mpi.h>  #include <omp.h>  static *long* num\_steps = 100000;  *double* step;  #define NUM\_THREADS 2  // const int n = 1000000;  *int* main(*int* *argc*, *char* \**argv*[])  {  *int* myrank, nprocs;  *long* i;  *double* mypi = 0.0, x = 0.0, pi = 0.0, sum = 0.0;  *double* start, stop;      step = 1.0 / (*double*)num\_steps;      MPI\_Status status;      MPI\_Init(&*argc*, &*argv*);      MPI\_Comm\_size(MPI\_COMM\_WORLD, &nprocs);      MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myrank);      omp\_set\_num\_threads(NUM\_THREADS);      start = omp\_get\_wtime();  #pragma omp parallel for reduction(+ : sum) private(x)      for (i = 1; i <= num\_steps; ++i)      {          x = (i - 0.5) \* step;          sum += 4.0 / (1.0 + x \* x);      }      pi += sum \* step;      MPI\_Send(&pi, 1, MPI\_DOUBLE, (myrank + 1) % nprocs, 99, MPI\_COMM\_WORLD);      MPI\_Recv(&mypi, 1, MPI\_DOUBLE, (myrank - 1 + nprocs) % nprocs, 99, MPI\_COMM\_WORLD, &status);      stop = omp\_get\_wtime();      // if (myrank == 0)      printf("PI = %lf\ntime = %lf\n", pi, stop-start);      MPI\_Finalize();      return 0;  } |

运行结果：



## （4）设计并实现矩阵乘的并行算法，分析所设计的并行算法的性能，比如加速比、并行效率和可扩展性。（选做）

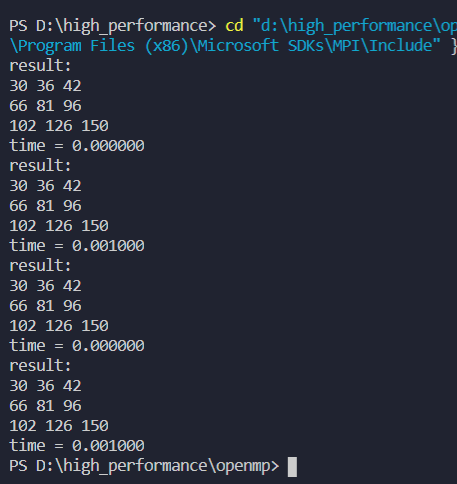
设计思路：

我的思路很朴素，就是设立两个矩阵matrix1[A][B]，matrix2[B][C]，然后呢对matrix1[A][B]的每一行进行并行的计算，去乘matrix2[B][C]的每一列，然后分别存到结果矩阵中去。

代码：

|  |
| --- |
| //计算 matrix1(A\*B) \* matrix2(B\*C) = matrixAns(A\*C)  #include <stdio.h>  #include <omp.h>  #define A 3  #define B 3  #define C 3  #define nThreads 4  //第一个A\*B矩阵  *int* matrix1[A][B] = {      {1, 2, 3},      {4, 5, 6},      {7, 8, 9}};  //第二个B\*C矩阵  *int* matrix2[B][C] = {      {1, 2, 3},      {4, 5, 6},      {7, 8, 9}};  *int* matrixAns[A][C]; //结果矩阵  *int* main()  {  *double* start, stop;      omp\_set\_num\_threads(nThreads);  *int* temp;      start = omp\_get\_wtime();  #pragma omp parallel      {  #pragma omp for private(temp)          for (*int* a = 0; a < A; a++)          {              for (*int* c = 0; c < C; c++)              {                  temp = 0;                  for (*int* b = 0; b < B; b++)                  {                      temp += matrix1[a][b] \* matrix2[b][c];                  }                  matrixAns[a][c] = temp;              }          }      }      stop = omp\_get\_wtime();      //输出      printf("result:\n");      for (*int* i = 0; i < A; ++i)      {          for (*int* j = 0; j < C; ++j)              printf("%d ", matrixAns[i][j]);          printf("\n");      }      printf("time = %lf\n", stop-start);      return 0;  } |

运行结果：



性能分析：

对于matrix1[A][B]的每一行，我采用了并行的方式，但是对于mareix1[A][B]的每一行和matrix2[B][C]的每一列的乘法累加，我用的仍然是串行的方式。这样的结果就是当矩阵很大的时候，我的代码的串行运行时间实际上会远远大于并行运行的时间，所以其加速比并不会很高，其效率也比较普通。

对于更大规模的问题，正如上面提到的，串行计算的时间会远远大于并行计算的时间，其并行效率也会相应地有所下降，所以其可扩展性较差。

## （5）MPI与OpenMP有什么区别？谈谈对高性能计算的理解和认识。

MPI与OpenMP的区别还是蛮大的。首先从编程的角度来讲，MPI的代码要比OpenMP的代码来得复杂，这一点在我做实验的时候深有体会。很多时候，OpenMP只需要在原有循环的基础上加上并行语句即可，不需要我们去考虑怎么传输，怎么接收，怎么分配，而MPI则截然不同。

而且，在兼容性方面，OpenMP是针对单主机上的多核/多CPU并行计算而设计的工具，所以说无法用在超级计算机上。而MPI作为分布式额你存之间实现信息通讯的一种标准库，具有很好的可移植性和可扩展性，适用于超级计算机的多主机多核心环境。

但是，在可靠性方面，MPI如果一个进程出问题，那么整个程序都将出现错误，这样就提高了对MPI程序进程调试的难度，所以其可靠性是不如OpenMP的。

百度上对高性能计算有如下定义：指通常使用很多处理器（作为单个机器的一部分）或者某一集群中组织的几台计算机（作为单个计算资源操作）的计算系统和环境。也就是说，这是一种通过并发的方式来高效地协调使用多个处理器或者多个核心来提高计算效率。

其前景是非常开阔的，尤其是在当今人工智能等这些上层应用已经接近饱和的状态下。如今的很多大型工程运算，诸如气象预测等等，都离不开作为底层的并行运算来提升它们的计算速度。如果有可能的话，我希望在未来能够加入到高性能计算的研究行列中去。