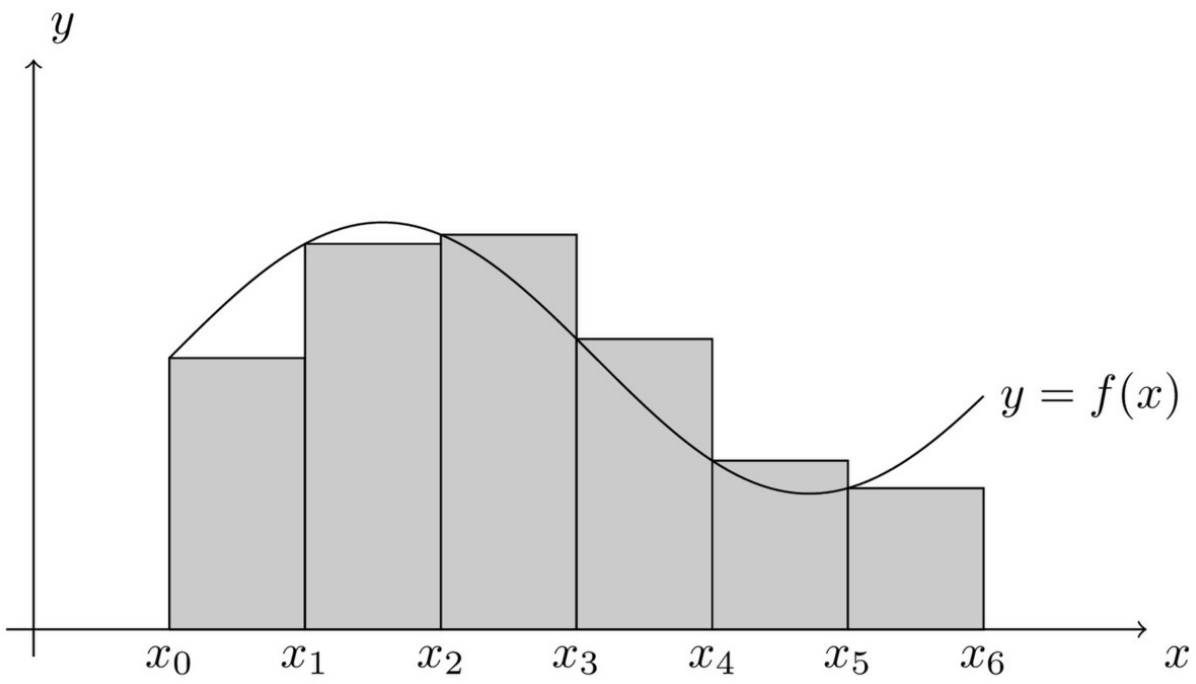
MPI和OpenMP简介

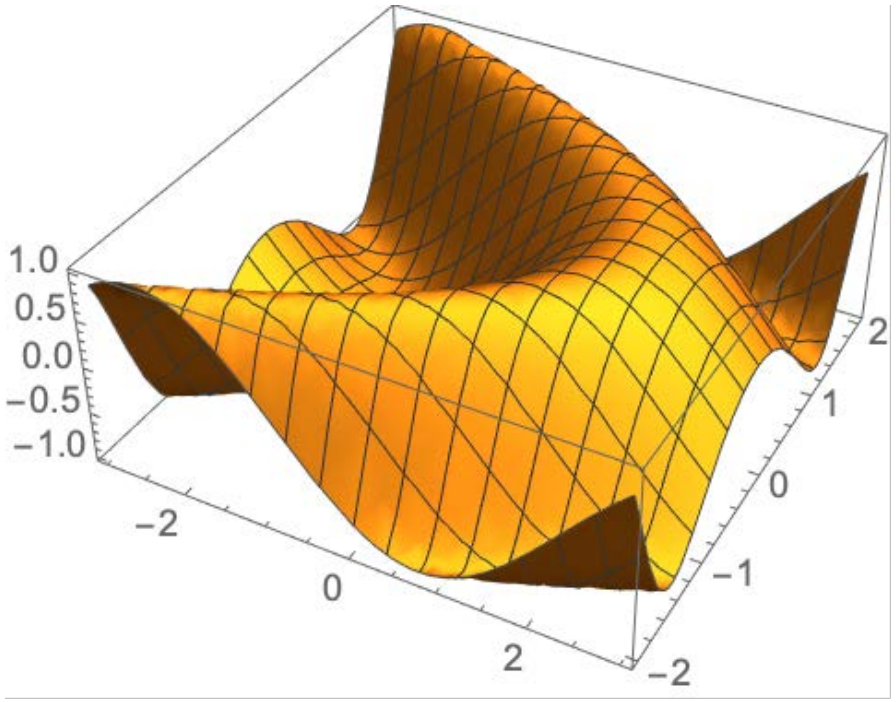
王政力
中科院理论物理研究所
wangzhengli@itp.ac.cn
2020年9月23日

问题描述

求和



扫点



Outline

- MPI
- OpenMP

什么是MPI

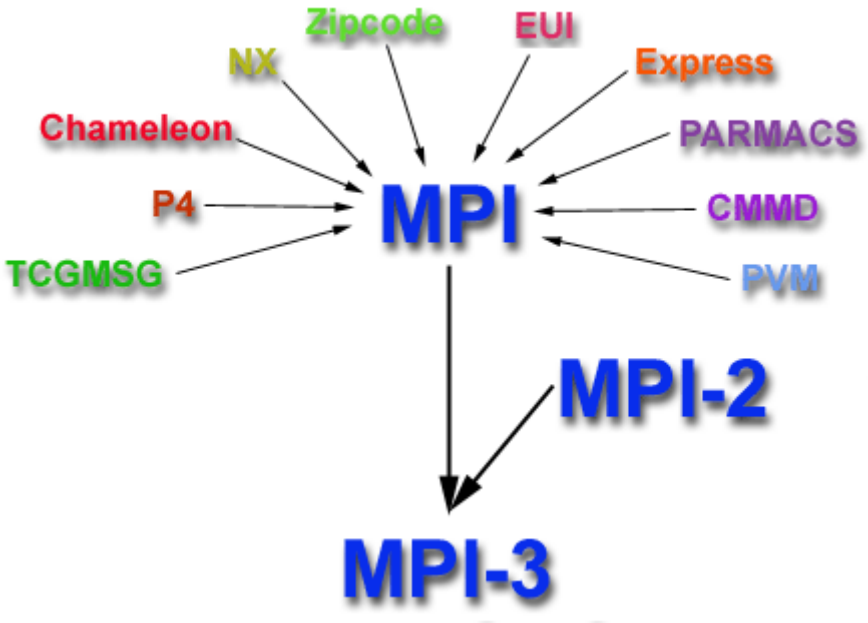
- Message Passing Interface（MPI）是一种消息传递模型，由MPI Forum开发，官方支持C/C++，Fortran，社区支持Python，Ruby，Java，Julia，D等等

- MPI是一种库描述，并不是一种语言，有上百个函数调用接口，可以在C/C++，Fortran中直接对这些函数进行调用

- MPI是一种标准或者规范，并不是某种特指的具体实现

MPI的发展历程

- 1994年5月： MPI-1.0
  - 1995年6月： MPI-1.1
  - 1997年7月： MPI-1.2
  - 2008年5月： MPI-1.3
- 1998年： MPI-2.0
  - 2008年9月： MPI-2.1
  - 2009年9月： MPI-2.2
- 2012年9月： MPI-3.0
  - 2015年6月： MPI-3.1
- 目前： MPI-4.0正在开发中



Portability, efficiency, functionality

常见的MPI实现

- 主要有两大类库：Open MPI和MPICH（MPICH/MPICH2/MPICH3，OSU MVAPICH/MVAPICH2 ，Intel MPI，IBM MPI，Cray MPI）
- Intel MPI，Open MPI和MPICH3：支持InfiniBand和以太网
- MPICH/MPICH2：支持以太网，不支持InfiniBand
- MVAPICH/MVAPICH2：支持InfiniBand和以太网
- 区别：Open MPI的设计针对普遍通用的情况；MPICH跟进最新的MPI标准，并为其他有特殊需求的MPI实现提供了高质量的参考
- https://stackoverflow.com/questions/2427399/mpich-vs-openmpi

MPI基本概念

- 消息（变量）：address, length
- 源/目的地：source/destination，指明消息发往哪个进程或者从哪个进程接收消息
- 标签：tag，标记消息，选择性地发送或者接受指定的消息

Send(address, length, destination, tag)

Receive(address, length, source, tag, actlen)

MPI基本概念

- **MPI_Send**(address, count, datatype, destination, tag, comm)
- **MPI_Recv**(address, maxcount, datatype, source, tag, comm, status)
  - (address, count, datatype): (A, 300, **MPI_REAL**)代表Fortran中长度为300的实数数组
  - Comm: 通信域，MPI中最大的通信域为**MPI_COMM_WORLD**，进入并行环境时自动创建
  - Status: 接收到的消息的实际信息，包括消息的长度，来源和标签，接收进程可以接受特定来源特定标签的消息，也可以接受任意来源任意标签的消息，**MPI_ANY_SOURCE**, **MPI_ANY_TAG**

最基本的MPI函数

- **MPI_Init**：初始化MPI
- **MPI_Comm_size**：取得总进程数
- **MPI_Comm_rank**：取得每个进程的进程号，从0开始
- **MPI_Send**：发送消息
- **MPI_Recv**：接收消息
- **MPI_Finalize**：结束MPI

虽然MPI有上百个调用接口，但是其中最基本的六个函数就能编写一个完整的MPI程序，去解决很多问题

第一个MPI程序

各大编译器之间的mod文件并不兼容，
甚至同一编译器的不同版本也可能不兼容，
碰到编译错误可考虑更换编译器或者使用mpif.h
变量名千万不要取成mpi！！！

Fortran

```fortran
program main
  !use mpi
  implicit none
  include 'mpif.h'
  integer :: ierr,myid,np
  call mpi_init(ierr)
  call mpi_comm_rank(mpi_comm_world,myid,ierr)
  call mpi_comm_size(mpi_comm_world,np,ierr)
  write(*,*) myid,np
  call mpi_finalize(ierr)
end program main
```

C

```c
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]){
  int myid,np;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  printf("%d %d\n",myid,np);
  MPI_Finalize();
  return 0;
}
```

```
1           4
2           4
3           4
0           4
```

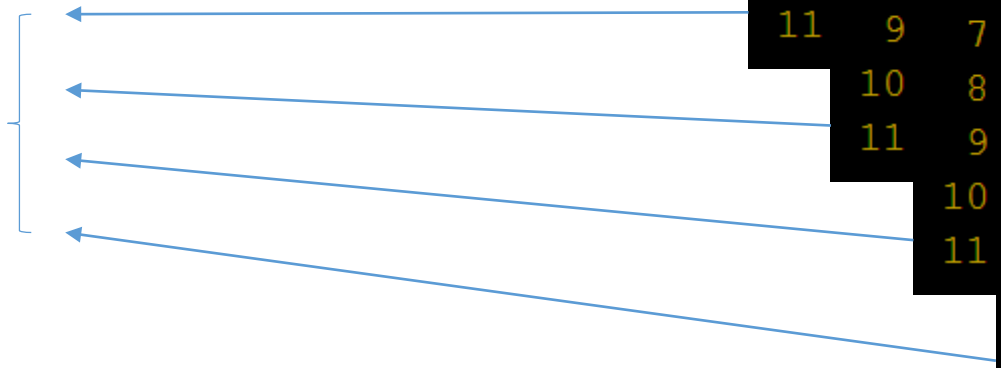
```
0 4
2 4
1 4
3 4
```

```c
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]){
  int myid,np;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  printf("%d %d\n",myid,np);
  MPI_Finalize();
  return 0;
}
```

SPMD: Single Program, Multiple Data

各个进程之间没有影响，每个语句独立地在各个进程中执行，即使是输出语句，所以如果输出很多，输出结果可能很乱

```
0 4
2 4
1 4
3 4
```



MPI程序的编译

```
mpicc -show
gcc -I/public/software//mpi/intelmpi/2017.4.239/intel64
ntelmpi/2017.4.239/intel64/lib -Xlinker --enable-new-dt
r -rpath -Xlinker /public/software//mpi/intelmpi/2017.4
-rpath -Xlinker /opt/intel/mpi-rt/2017.0.0/intel64/lib
```

- Open MPI（不分编译器）
  - C: mpicc
  - C++: mpic++, mpicxx ,mpiCC
  - Fortran: mpifort(mpif77, mpif90)
- Intel MPI（分编译器）
  - GNU编译器: mpicc, mpigcc, mpicxx, mpigxx, mpifc(mpif77, mpif90)
  - Intel编译器: mpiicc, mpiicpc, mpiifort
- 这些命令只是一层封装，实际会调用系统编译器，编译时自动添加MPI头文件搜索路径，链接时自动链接MPI库文件，方便使用

MPI程序的运行

- 常见的MPI实现都提供以下命令
  - mpirun –np 4 ./a.out
- MPI Forum建议使用以下命令
  - mpiexec –n 4 ./a.out

Fortran & C bindings

Fortran	C
MPI_INIT(ierror) integer ierror	int **MPI_Init**(int *argc, ***argv)
MPI_COMM_SIZE(comm, size, ierror) integer comm, size, ierror	int **MPI_Comm_size**(MPI_Comm comm, int *size)
MPI_COMM_RANK(comm, rank, ierror) integer comm, rank, ierror	int **MPI_Comm_rank**(MPI_Comm comm, int *rank)
MPI_FINALIZE(ierror) integer comm	int **MPI_Finalize**()

第二个MPI程序：求和

$$f(x) = \frac{4}{1+x^2}$$

$$\Delta x = \frac{1}{n}$$

$$x_i = \frac{i-0.5}{n}$$

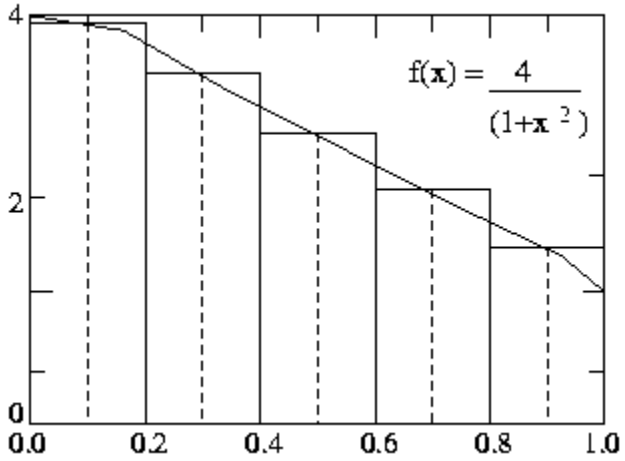
$$\pi \approx \sum_{i=1}^{n} f(x_i)\Delta x$$

$$\int_0^1 \frac{1}{1+x^2}\,dx = \arctan(x)\Big|_0^1$$

$$= \arctan(1) - \arctan(0)$$

$$= \arctan(1)$$

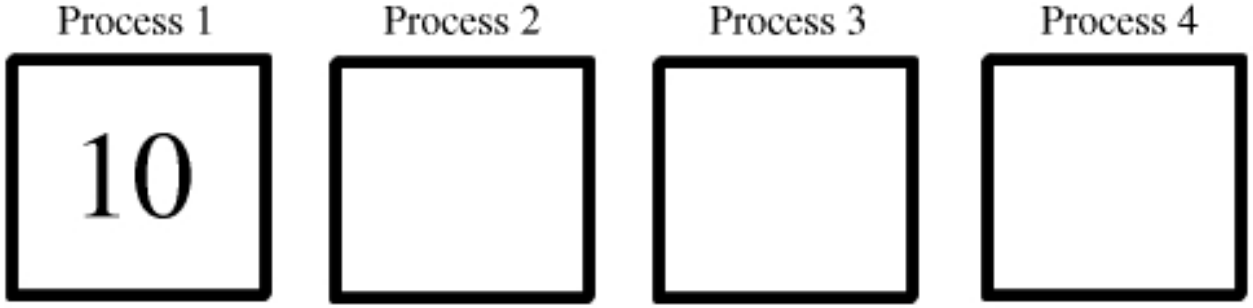
$$= \frac{\pi}{4}$$



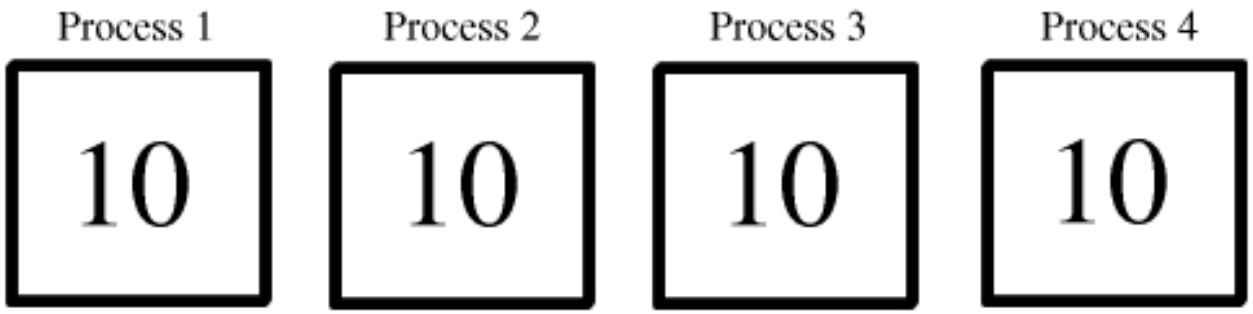
```fortran
program main
  use mpi
  implicit none
  integer, parameter :: dbl = selected_real_kind(15)
  real(dbl), parameter :: pi = acos(-1.0_dbl)
  integer :: ierr,myid,np,i,n
  real(dbl) :: dx,xi,mypi,tpi
  call mpi_init(ierr)
  call mpi_comm_rank(mpi_comm_world,myid,ierr)
  call mpi_comm_size(mpi_comm_world,np,ierr)
  do
    if ( myid == 0 ) then
      write(*,*) "Enter the number of intervals:(0 quits)"
      read(*,*) n
    end if
    call mpi_bcast(n,1,mpi_integer,0,mpi_comm_world,ierr)
    if ( n <= 0 ) exit
    dx = 1.0_dbl/n
    mypi = 0.0_dbl
    do i = myid+1, n, np
      xi = (i-0.5_dbl)/n
      mypi = mypi + 4.0_dbl/(1.0_dbl + xi**2)*dx
    end do
    call mpi_reduce(mypi,tpi,1,mpi_double_precision,&
        mpi_sum,0,mpi_comm_world,ierr)
    if ( myid == 0 ) then
      write(*,*) pi
      write(*,*) tpi
      write(*,*) abs(pi-tpi)/pi*100,"%"
    end if
  end do
  call mpi_finalize(ierr)
end program main
```

```c
#include "mpi.h"
#include <stdio.h>
#include <math.h>
void main(int argc,char *argv[])  {
  const double pi = acos(-1.0);
  double dx,xi,mypi,tpi;
  int myid,np,n,i;
  MPI_Init( &argc, &argv);
  MPI_Comm_rank( MPI_COMM_WORLD, &myid);
  MPI_Comm_size( MPI_COMM_WORLD, &np);
  while (1)  {
    if (myid == 0)  {
      printf("Enter the number of intervals:(0 quits)\n");
      scanf("%d",&n);
    }
    MPI_Bcast( &n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n<=0) break;
    dx = 1.0/n;
    mypi = 0.0;
    for (i = myid+1; i <= n; i+=np)  {
      xi = (i-0.5)/n;
      mypi += 4.0/(1.0+xi*xi)*dx;
    }
    MPI_Reduce( &mypi, &tpi, 1, MPI_DOUBLE, MPI_SUM, 0,
        MPI_COMM_WORLD);
    if (myid==0)  {
      printf("%.16f\n",pi);
      printf("%.16f\n",tpi);
      printf("%.16e%%\n",fabs(pi-tpi)/pi*100);
    }
  }
  MPI_Finalize();
}
```

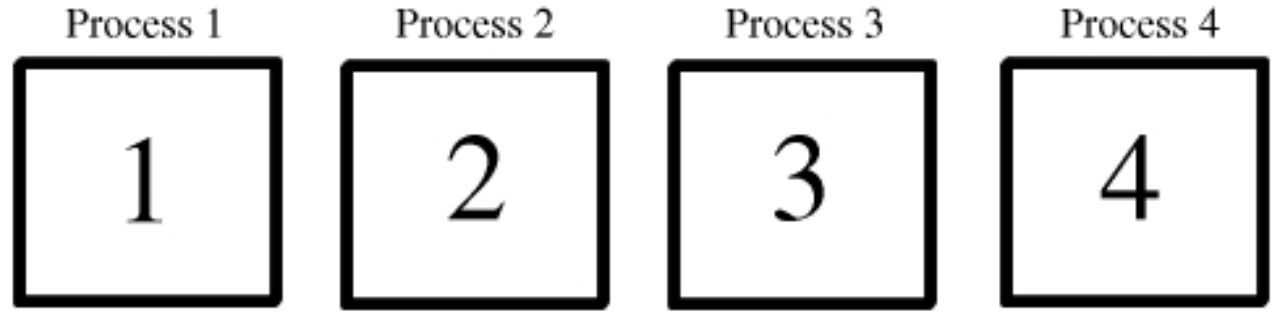
Before MPI_Bcast



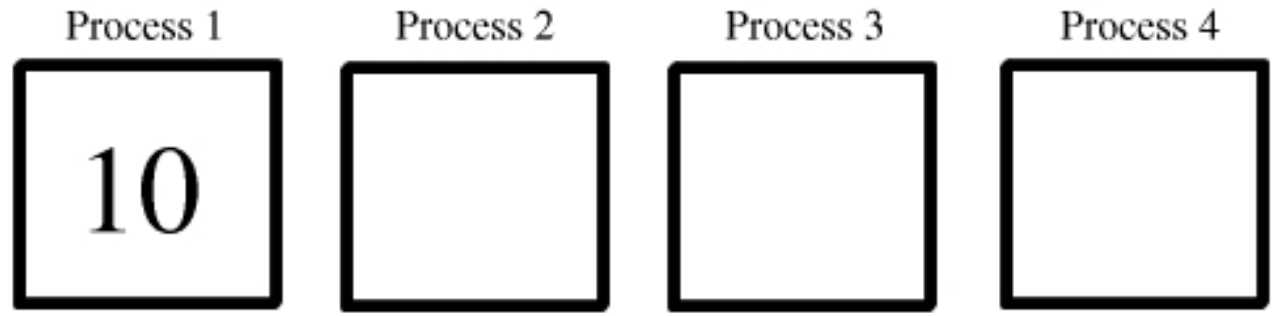
After MPI_Bcast



Before MPI_Reduce



After MPI_Reduce



```
 Enter the number of intervals:(0 quits)
9
   3.14159265358979
   3.14262145655761
  3.2747815559220 22E-002 %
 Enter the number of intervals:(0 quits)
99
   3.14159265358979
   3.14160115612355
  2.706440551352709E-004 %
 Enter the number of intervals:(0 quits)
999
   3.14159265358979
   3.14159273709004
  2.657895530102677E-006 %
 Enter the number of intervals:(0 quits)
9999
   3.14159265358979
   3.14159265442329
  2.653108456047908E-008 %
 Enter the number of intervals:(0 quits)
99999
   3.14159265358979
   3.14159265359813
  2.653006678298102E-010 %
 Enter the number of intervals:(0 quits)
0
```

```
Enter the number of intervals:(0 quits)
9
3.1415926535897931
3.1426214565576127
3.2747815559220218e-02%
Enter the number of intervals:(0 quits)
99
3.1415926535897931
3.1416011561235466
2.706440551352709 2e-04%
Enter the number of intervals:(0 quits)
999
3.1415926535897931
3.1415927370900434
2.6578955159668783e-06%
Enter the number of intervals:(0 quits)
9999
3.1415926535897931
3.1415926544232935
2.653114110367342 1e-08%
Enter the number of intervals:(0 quits)
99999
3.1415926535897931
3.1415926535981287
2.6532893942697868e-10%
Enter the number of intervals:(0 quits)
0
```

Fortran & C bindings

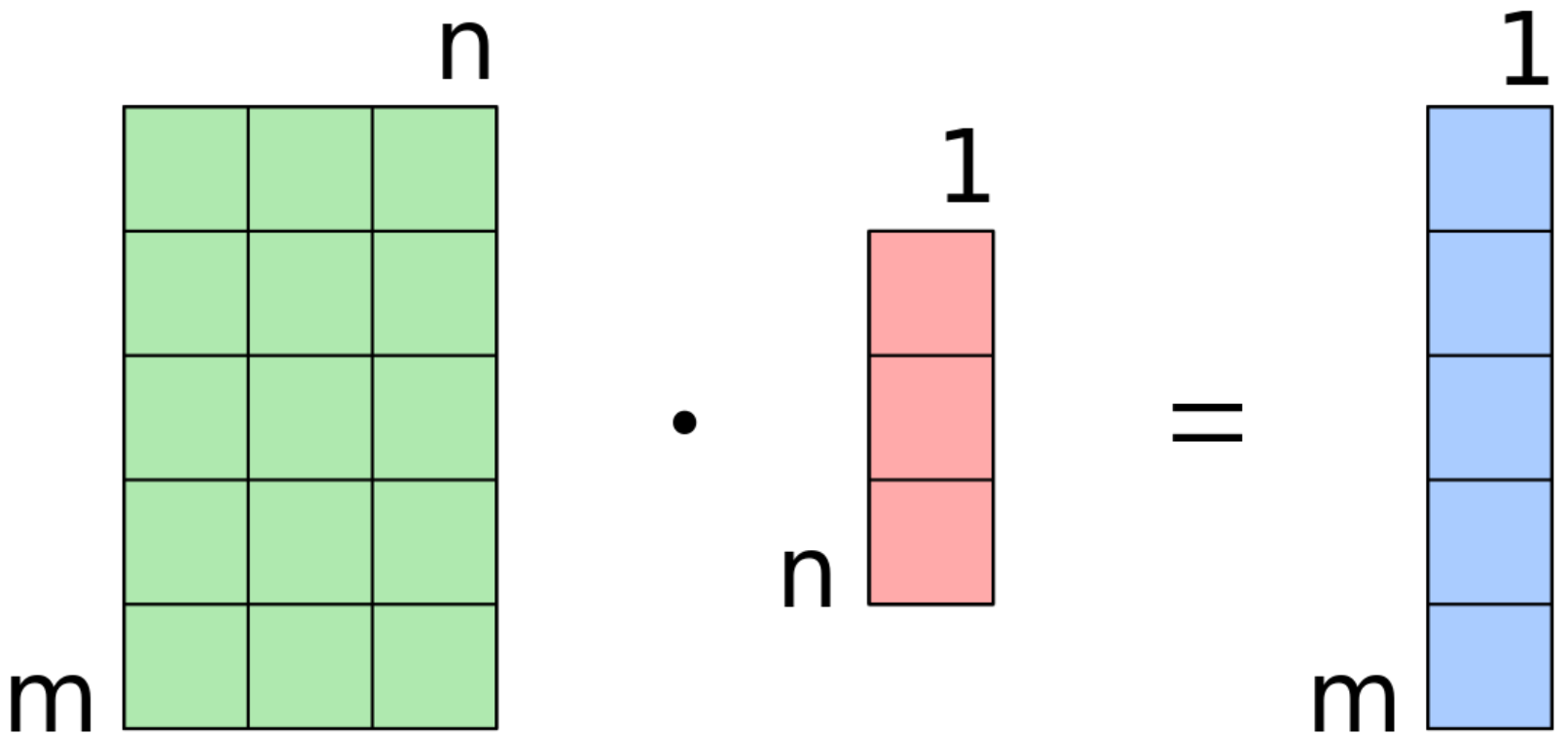
Fortran	**MPI_BCAST**(buffer, count, datatype, root, comm, ierror) <type> buffer(*) integer count, datatype, root, comm, ierror
	MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierror) <type> sendbuf(*), recvbuf(*) integer count, datatype, op, root, comm, ierror
C	int **MPI_Bcast**(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
	int **MPI_Reduce**(const void *sendbuf, void *recvbuf, int count, MPI_Datatype, MPI_Op op, int root, MPI_Comm comm)

MPI归约操作	
MPI_MAX	最大值
MPI_MIN	最小值
MPI_MAXLOC	最大值且相应位置
MPI_MINLOC	最小值且相应位置
MPI_SUM	求和
MPI_PROD	乘积

MPI基本数据类型

MPI Fortran		MPI C	
MPI_CHARACTER	character(1)	MPI_CHAR	signed char
MPI_INTEGER	integer	MPI_INT	signed int
MPI_REAL	real	MPI_FLOAT	float
MPI_DOUBLE_PRECISION	double precision	MPI_DOUBLE	double
MPI_COMPLEX	complex	MPI_LONG	signed long int
MPI_DOUBLE_COMPLEX	double complex	MPI_LONG_DOUBLE	long double
MPI_LOGICAL	logical	MPI_UNSIGNED	unsigned int

第三个MPI程序：矩阵矢量乘积



A · x = y

Self-scheduling or manager-worker algorithm

- 主要思想是一个进程负责分配任务，其余进程负责计算
- 这个机制非常适合其余进程（worker）不需要相互通信或者每个worker需要执行的任务量不确定的情况，所以scheduling可以理解为负载均衡
- 几个例子：
  - 矩阵乘积，计算每个矩阵元的计算量一样
  - 扫描T矩阵，有时候阈值上下计算量并不一样
  - 用试除法求N以内的素数个数，判断一个数是否为素数所需要的计算量并不相同

Common part

$$c = a \times b$$

```
 1 program main
 2   use mpi
 3   implicit none
 4   integer, parameter :: dbl = selected_real_kind(15)
 5   integer, parameter :: n = 1000
 6   real(dbl) :: a(n,n),b(n),c(n),buffer(n),ans
 7   integer :: ierr,myid,np,manager,stat(mpi_status_size)
 8   integer :: i,numsent,source,tag
 9   call mpi_init(ierr)
10   call mpi_comm_rank(mpi_comm_world,myid,ierr)
11   call mpi_comm_size(mpi_comm_world,np,ierr)
12   manager = 0
13   if(myid == manager) then
14   else
15   end if
16   call mpi_finalize(ierr)
17 end program main
```

Manager part

- 先把b广播到各个进程
- 再把a的行发送到每个worker
- 进入循环，每接收一个c矩阵元就再发送a的一行到worker，直到填满c为止，这时候发送一个终止信号到各个worker

```
13  if(myid == manager) then
14    a = 1.0_dbl
15    b = 1.0_dbl
16    numsent = 0
17    call mpi_bcast(b,n,mpi_double_precision,&
18        manager,mpi_comm_world,ierr)
19    do i = 1, np-1
20      buffer = a(i,:)
21      call mpi_send(buffer,n,mpi_double_precision,&
22          i,i,mpi_comm_world,ierr)
23      numsent = numsent + 1
24    end do
25    do i = 1, n
26      call mpi_recv(ans,1,mpi_double_precision,&
27          mpi_any_source,mpi_any_tag,mpi_comm_world,stat,ierr)
28      source = stat(mpi_source)
29      tag = stat(mpi_tag)
30      c(tag) = ans
31      if(numsent<n)then
32        buffer = a(numsent+1,:)
33        call mpi_send(buffer,n,mpi_double_precision,&
34            source,numsent+1,mpi_comm_world,ierr)
35        numsent = numsent + 1
36      else
37        call mpi_send(mpi_bottom,0,mpi_double_precision,&
38            source,0,mpi_comm_world,ierr)
39      endif
40    end do
41    write(*,*) maxval(abs(c-n))
42  else
```

Worker part

- 先得到b
- 进入循环，包括接收a的行，计算矩阵元，把结果发送给manager，直到收到终止信号结束循环

```
42    else
43      call mpi_bcast(b,n,mpi_double_precision,&
44           manager,mpi_comm_world,ierr)
45      do
46        call mpi_recv(buffer,n,mpi_double_precision,&
47             manager,mpi_any_tag,mpi_comm_world,stat,ierr)
48        tag = stat(mpi_tag)
49        if(tag == 0) exit
50        ans = dot_product(buffer,b)
51        call mpi_send(ans,1,mpi_double_precision,&
52             manager,tag,mpi_comm_world,ierr)
53      end do
54    end if
```

几点说明

- 默认情况下tag的取值范围0-32767
- Status是输出参数，包含接收到的消息的source，tag，length
- 在Fortran中，status(**MPI_STATUS_SIZE**)是整数数组，status(**MPI_SOURCE**)为消息来源，status(**MPI_TAG**)为消息标签
- 在C中，status是**MPI_Status**类型的结构体，status.**MPI_SOURCE**表示消息的来源，status.**MPI_TAG**表示消息的标签

Fortran	**MPI_SEND**(buf, count, datatype, dest, tag, comm, ierror) <type> buf(*) integer count, datatype, dest, tag, comm, ierror
	MPI_RECV(buf, count, datatype, source, tag, comm, status, ierror) <type> buf(*) integer count, datatype, source, tag, comm, status(MPI_STATUS_SIZE), ierror
C	int **MPI_Send**(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
	int **MPI_Recv**(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Outline

- MPI

- OpenMP

什么是OpenMP

- OpenMP(Open Multi-Processing)是一种针对共享式内存的多线程并行变成标准（不能像MPI那样跨节点），支持C/C++, Fortran
- 由编译指令，库函数，环境变量三部分组成，通常由编译器提供支持，在编译时打开OpenMP编译选项即可
- 已有的代码不需要大幅度修改，只需加上几条OpenMP指令即可并行化，当编译器不支持OpenMP时，程序又可退化为串行程序
- 行业标准：
  - DEC, Intel, IBM, HP, Sun, SGI, Fujitsu, 美国能源部等等
  - 跨平台：Windows/Linux/macOS

历史回顾

- 过去每个厂商都有自己的指令和库标准，虽然有ANSI X3H5标准，但未被正式采用；二来有消息传递库PVM和MPI，能够代替内存共享机
- 但在1996-1997年，厂商恢复了对共享内存编程架构的兴趣，还有部分厂商认为消息传递模型太过复杂
- 进入多核cpu时代后，用操作系统API编写多线程程序不如OpenMP来得方便，比如扩展性问题，可移植性问题等等

Release History

Date	Version
Oct 1997	Fortran 1.0
Oct 1998	C/C++ 1.0
Nov 1999	Fortran 1.1
Nov 2000	Fortran 2.0
Mar 2002	C/C++ 2.0
May 2005	OpenMP 2.5
May 2008	OpenMP 3.0
Jul 2011	OpenMP 3.1
Jul 2013	OpenMP 4.0
Nov 2015	OpenMP 4.5
Nov 2018	OpenMP 5.0

GNU	GCC C/C++/Fortran	Free and open source – Linux, Solaris, AIX, MacOSX, Windows, FreeBSD, NetBSD, OpenBSD, DragonFly BSD, HPUX, RTEMS ▪ From GCC 4.2.0, OpenMP 2.5 is fully supported for C/C++/Fortran. ▪ From GCC 4.4.0, OpenMP 3.0 is fully supported for C/C++/Fortran. ▪ From GCC 4.7.0, OpenMP 3.1 is fully supported for C/C++/Fortran. ▪ In GCC 4.9.0, OpenMP 4.0 is supported for C and C++, but not Fortran. ▪ From GCC 4.9.1, OpenMP 4.0 is fully supported for C/C++/Fortran. ▪ From GCC 6.1, OpenMP 4.5 is fully supported for C and C++. ▪ From GCC 7.1, OpenMP 4.5 is partially supported for Fortran. ▪ From GCC 9.1, OpenMP 5.0 is partially supported for C and C++. Compile with -fopenmp to enable OpenMP. Online documentation: https://gcc.gnu.org/onlinedocs/libgomp/ OpenMP support history: https://gcc.gnu.org/projects/gomp/
IBM	XL C/C++/Fortran	XL C/C++ for Linux V16.1.1 and XL Fortran for Linux V16.1.1 fully support OpenMP 4.5 features including the target constructs. Compile with -qsmp=omp to enable OpenMP directives and with -qoffload for offloading the target regions to GPUs. For more information, please visit IBM XL C/C++ for Linux and IBM XL Fortran for Linux.
Intel	C/C++/Fortran	Windows, Linux, and MacOSX. ▪ OpenMP 3.1 C/C++/Fortran fully supported in version 12.0, 13.0, 14.0 compilers ▪ OpenMP 4.0 C/C++/Fortran supported in version 15.0 and 16.0 compilers ▪ OpenMP 4.5 C/C++/Fortran supported in version 17.0, 18.0, and 19.0 compilers ▪ OpenMP 4.5 and subset of OpenMP 5.0 C/C++/Fortran supported in 19.1 compilers under -qnextgen -fiopenmp. Compile with -Qopenmp on Windows, or just -openmp or -qopenmp on Linux or Mac OSX More information

常用编译器的OpenMP编译选项

编译器	GNU	Intel	Nvidia（原PGI）
OpenMP选项	-fopenmp	-qopenmp	-mp

2013年7月29日， PGI被Nvidia收购
2020年8月05日， PGI成为NVIDIA HPC SDK的一部分， 可以从Nvidia免费下载

第一个OpenMP程序

```fortran
program main
  implicit none
  write(*,*) 'Hi'
  !$omp parallel
  write(*,*) 'Hello'
  !$omp end parallel
  write(*,*) 'Bye'
end program main
```

```
gfortran hello.f90 && ./a.out
 Hi
 Hello
 Bye
```

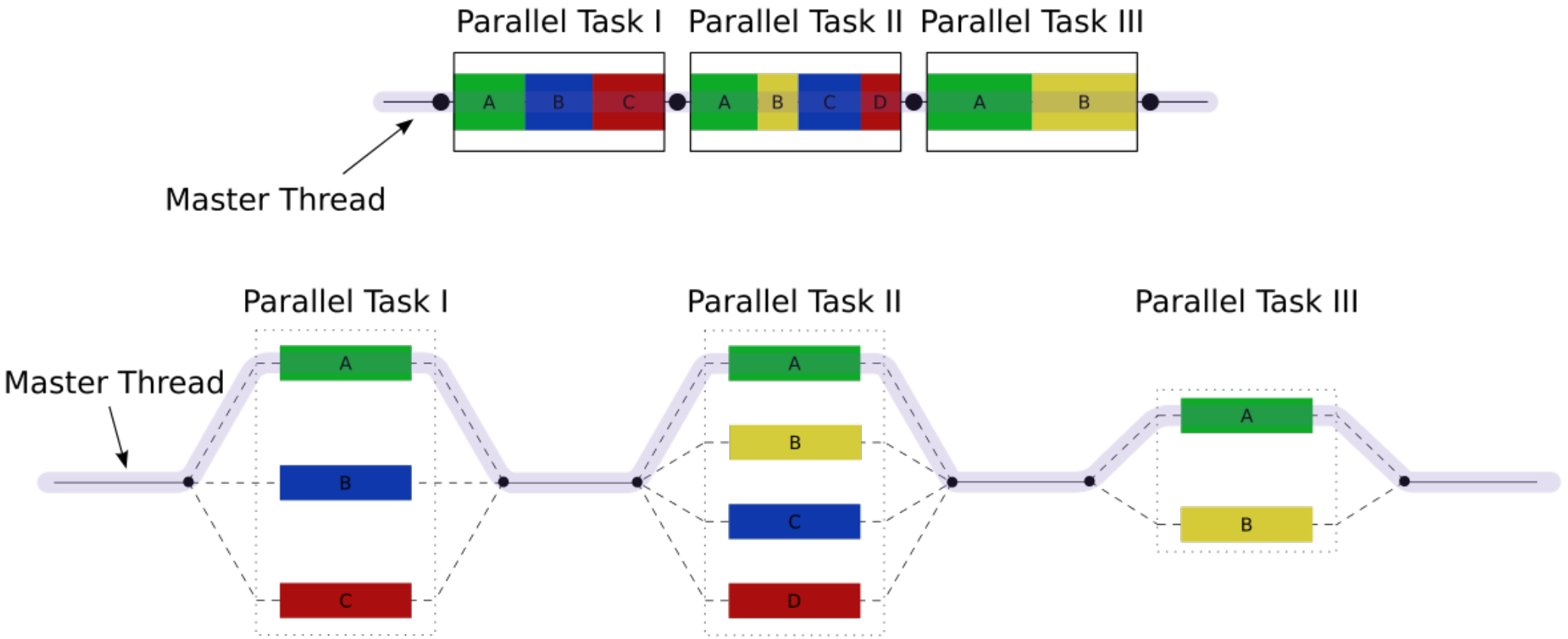
```
gfortran hello.f90 -fopenmp && ./a.out
 Hi
 Hello
 Hello
 Hello
 Hello
 Bye
```

```c
#include <stdio.h>
void main(int argc, char *argv[]){
  printf("Hi\n");
  #pragma omp parallel
  {
  printf("Hello\n");
  }
  printf("Bye\n");
}
```

```
gcc hello.c && ./a.out
Hi
Hello
Bye
```

```
gcc hello.c -fopenmp && ./a.out
Hi
Hello
Hello
Hello
Hello
Bye
```

Fork-Join Model



OpenMP API Overview

- Compiler Directives
  - Syntax: sentinel directive-name [clause, ...]
  - Fortran: !$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)
  - C/C++: #pragma omp parallel default(shared) private(beta,pi)
- Runtime Library Routines
  - omp_get_thread_num, omp_get_num_threads
  - Fortran: omp_lib.h(use omp_lib)
  - C/C++: omp.h
- Environment Variables
  - OMP_NUM_THREADS

Directives

Work-sharing constructs	Combined parallel work-sharing constructs	Synchronization constructs	Data environment constructs
do/for sections single	parallel do/parallel for parallel sections	master critical barrier atomic flush ordered	threadprivate

do/for

```
1  program main
2    implicit none
3    integer, parameter :: dbl = selected_real_kind(15)
4    integer, parameter :: n = 2000
5    real(dbl), dimension(n,n) :: a,b,c
6    integer :: i,j,k
7    a = 1.0_dbl
8    b = 1.0_dbl
9    c = 0.0_dbl
10   !$omp parallel
11   !$omp do
12   do i = 1, n
13     do j = 1, n
14       do k = 1, n
15         c(i,j) = c(i,j) + a(i,k)*b(k,j)
16       end do
17     end do
18   end do
19   !$omp end do
20   !$omp end parallel
21   write(*,*) maxval(abs(c-n))
22 end program main
```

```
1  #include <stdio.h>
2  void main(int argc, char *argv[]){
3    const int n = 2000;
4    double a[n][n],b[n][n],c[n][n];
5    for (int i=0; i<n; i++){
6      for (int j=0; j<n; j++){
7        a[i][j] = 1.0;
8        b[i][j] = 1.0;
9        c[i][j] = 0.0;
10     }
11   }
12   #pragma omp parallel
13   {
14   #pragma omp for
15   for (int i=0; i<n; i++){
16     for (int j=0; j<n; j++){
17       for (int k=0; k<n; k++){
18         c[i][j] += a[i][k]*b[k][j];
19       }
20     }
21   }
22   }
23   printf("%f\n",c[0][0]-n);
24 }
```

```
gfortran -O2 mm.f90 && time ./a.out
   0.0000000000000000

real    0m12.518s
user    0m12.492s
sys     0m0.026s
```

```
gcc -O2 mm.c && time ./a.out
0.000000

real    0m12.080s
user    0m12.058s
sys     0m0.022s
```

```
gfortran -O2 mm.f90 -fopenmp && time ./a.out
   0.0000000000000000

real    0m3.702s
user    0m14.576s
sys     0m0.031s
```

```
gcc -O2 mm.c -fopenmp && time ./a.out
0.000000

real    0m3.688s
user    0m14.303s
sys     0m0.046s
```

sections/single/master

```
 1 program main
 2   implicit none
 3   !$omp parallel sections
 4   !$omp section
 5   write(*,*) 'Hello'
 6   !$omp section
 7   write(*,*) 'Hi'
 8   !$omp section
 9   write(*,*) 'Bye'
10   !$omp end parallel sections
11 end program main
```

```
gfortran -fopenmp tmp.f90 && ./a.out
 Hello
 Hi
 Bye
```

```
 1 program main
 2   implicit none
 3   !$omp parallel
 4   write(*,*) 'Hello'
 5   !$omp single
 6   write(*,*) 'Hi'
 7   !$omp end single
 8   !$omp end parallel
 9 end program main
```

```
gfortran -fopenmp tmp.f90 && ./a.out
 Hello
 Hi
 Hello
 Hello
 Hello
```

```
 1 program main
 2   implicit none
 3   !$omp parallel
 4   write(*,*) 'Hello'
 5   !$omp master
 6   write(*,*) 'Hi'
 7   !$omp end master
 8   !$omp end parallel
 9 end program main
```

```
gfortran -fopenmp tmp.f90 && ./a.out
 Hello
 Hi
 Hello
 Hello
 Hello
```

critical/atomic

```
 1 program main
 2   implicit none
 3   integer, parameter :: n = 100
 4   integer :: i,s
 5   s = 0
 6   !$omp parallel do
 7   do i=1,n
 8     !$omp critical
 9     s = s + i
10     !$omp end critical
11   end do
12   !$omp end parallel do
13   write(*,*) s
14 end program main
```

```
gfortran -fopenmp tmp.f90 && ./a.out
        5050
```

```
 1 program main
 2   implicit none
 3   integer, parameter :: n = 100
 4   integer :: i,s
 5   s = 0
 6   !$omp parallel do
 7   do i=1,n
 8     s = s + i
 9   end do
10   !$omp end parallel do
11   write(*,*) s
12 end program main
```

```
gfortran -fopenmp tmp.f90 && ./a.out
        3150
```

```
 1 program main
 2   implicit none
 3   integer, parameter :: n = 100
 4   integer :: i,s
 5   s = 0
 6   !$omp parallel do
 7   do i=1,n
 8     !$omp atomic
 9     s = s + i
10   end do
11   !$omp end parallel do
12   write(*,*) s
13 end program main
```

```
gfortran -fopenmp tmp.f90 && ./a.out
        5050
```

Clauses

Data scope attribute clauses	Other clauses
PRIVATE(list) SHARED(list) DEFAULT(PRIVATE | SHARED | NONE) FIRSTPRIVATE(list) LASTPRIVATE(list) COPYIN(list) COPYPRIVATE(list) REDUCTION(operator:list)	IF(scalar logical expression) NUM_THREADS(scalar integer expression) NOWAIT SCHEDULE(type, chunk)

Reduction(openrator:list)

Operation	Fortran	C/C++	Initialization
Addition	+	+	0
Multiplication	*	*	1
Subtraction	-	-	0
Logical AND	.and.	&&	.true. / 1
Logical OR	.or.	||	.false. / 0
AND bitwise	Iand	&	All bits on / ~0
OR bitwise	Ior	|	0
Exclusive OR bitwise	Ieor	^	0
Equivalent	.eqv.		.true.
Not Equivalent	.neqv.		.false.
Maximum	Max	Max	Most negative
Minimum	Min	Min	Largest positive

```fortran
program main
  implicit none
  integer, parameter :: dbl = selected_real_kind(15)
  integer, parameter :: n = 100
  real(dbl), parameter :: pi = acos(-1.0_dbl)
  real(dbl) :: mypi,dx,xi
  integer :: i
  mypi = 0.0_dbl
  dx = 1.0_dbl/dble(n)
  !$omp parallel private(xi)
  !$omp do reduction(+:mypi)
  do i=1,n
    xi = (dble(i)-0.5_dbl)/dble(n)
    mypi = mypi + 4.0_dbl/(1.0_dbl+xi**2)*dx
  end do
  !$omp end do
  !$omp end parallel
  write(*,*) pi
  write(*,*) mypi
  write(*,*) abs(pi-mypi)/pi*100,'%'
end program main
```

```
gfortran -fopenmp pi.f90 && ./a.out
   3.1415926535897931
   3.1416009869231249
   2.6525823843875473E-004 %
```

```c
#include <stdio.h>
#include <math.h>
void main(int argc, char *argv[]){
  const int n = 100;
  const double pi = acos(-1.0);
  double mypi,dx,xi;
  mypi = 0.0;
  dx = 1.0/(double)n;
  #pragma omp parallel private(xi)
  {
  #pragma omp for reduction(+:mypi)
    for (int i=1; i<=n; i++){
      xi = ((double)i-0.5)/(double)n;
      mypi += 4.0/(1.0+xi*xi)*dx;
    }
  }
  printf("%f\n",pi);
  printf("%f\n",mypi);
  printf("%f%%\n",fabs(pi-mypi)/pi*100);
}
```

```
gcc -fopenmp pi.c && ./a.out
3.141593
3.141601
0.000265%
```

私有&公有变量

- 默认情况下循环计数器和threadprivate变量是私有变量，其余都是公有变量，私有变量在并行开始处其值不确定，并行结束时其值也不确定
- 通过private，share，default来声明变量属性
- firstprivate在并行开始处对私有变量赋值，lastprivate在并行结束时更新私有变量的值，copyprivate可在单一线程执行single指令对中的指令后将私有变量的值传递给其他线程

Schedule(type, chunk)

- 并行执行循环语句时，每个线程执行相同数量的迭代，这并不总是最好的办法，因为每个线程的工作量可能会不相同
- Static：chunk不赋值时，默认每个线程执行相同数量的迭代，比如1-25，26-50，51-75，76-100
- Dynamic：chunk不赋值时，默认chunk=1，每个线程执行一次迭代，比如1，2，3，4，最先执行完的线程继续执行下一个迭代
- Guided：与静态方法相比，动态方法提高了执行能力和效率，但处理和分配任务时产生了冗余，工作快越小，冗余越多；另一种动态方法是每个块的工作量按指数递减，后一个块的工作量是前一个的一半
- Runtime：通过环境变量OMP_SCHEDULE来指定

static

```
gfortran -O2 prime.f90 -fopenmp && time ./a.out
     5761455

real    0m19.608s
user    0m56.780s
sys     0m0.176s
```

dynamic

```
gfortran -O2 prime.f90 -fopenmp && time ./a.out
     5761455

real    0m15.105s
user    0m59.481s
sys     0m0.162s
```

guided

```
gfortran -O2 prime.f90 -fopenmp && time ./a.out
     5761455

real    0m14.326s
user    0m56.540s
sys     0m0.151s
```

```
 1 function p(n)
 2   implicit none
 3   integer :: n,p,i
 4   do i=3,nint(sqrt(dble(n))),2
 5     if(mod(n,i)==0)then
 6       p = 0
 7       return
 8     end if
 9   end do
10   p = 1
11 end function p
12
13 program main
14   implicit none
15   integer, parameter :: n = 1e8
16   integer :: a(n),i
17   integer, external :: p
18   a = 0
19   !$omp parallel do schedule(dynamic)
20   do i=3,n,2
21     a(i) = p(i)
22   end do
23   !$omp end parallel do
24   write(*,*) sum(a)+1
25 end program main
```

num_threads & if

```
 1 program main
 2   implicit none
 3   !$omp parallel num_threads(3)
 4   !$omp sections
 5   !$omp section
 6   write(*,*) 'Hi'
 7   !$omp section
 8   write(*,*) 'Hello'
 9   !$omp section
10   write(*,*) 'Bye'
11   !$omp end sections
12   !$omp end parallel
13 end program main
```

```
 1 function p(n)
 2   implicit none
 3   integer :: n,p,i
 4   do i=3,nint(sqrt(dble(n))),2
 5     if(mod(n,i)==0)then
 6       p = 0
 7       return
 8     end if
 9   end do
10   p = 1
11 end function p
12
13 program main
14   implicit none
15   integer, parameter :: n = 1e8
16   integer :: a(n),i
17   integer, external :: p
18   a = 0
19   !$omp parallel if(n>1e5)
20   !$omp do schedule(dynamic)
21   do i=3,n,2
22     a(i) = p(i)
23   end do
24   !$omp end do
25   !$omp end parallel
26   write(*,*) sum(a)+1
27 end program main
```

库函数

Execution environment routines
OMP_set_num_threads
OMP_get_num_threads
OMP_get_max_threads
OMP_get_thread_num
OMP_get_num_procs
OMP_in_parallel
OMP_set_dynamic
OMP_get_dynamic
OMP_set_nested
OMP_get_nested

```
 1 program main
 2   !use omp_lib
 3    implicit none
 4    include 'omp_lib.h'
 5    integer,save :: myid,np
 6    !$omp threadprivate(myid,np)
 7    !$omp parallel
 8    myid = omp_get_thread_num()
 9    np = omp_get_num_threads()
10    !$omp end parallel
11    !$omp parallel
12    write(*,*) myid,np
13    !$omp end parallel
14 end program main
```

```
gfortran -fopenmp tmp.f90 && ./a.out
           2           4
           0           4
           1           4
           3           4
```

环境变量

The environment variables
OMP_NUM_THREADS
OMP_SCHEDULE
OMP_DYNAMIC
OMP_NESTED

优先级：
NUM_THREADS >
OMP_SET_NUM_THREADS >
OMP_NUM_THREADS

总结

- MPI：进程级，分布式存储，效率高，可扩展性好，程序复杂
- OpenMP：线程级，共享式存储，可扩展性差（单机，不可跨节点），程序简单，要注意racing condition

References

- Using MPI Portable Parallel Programming with the Message-Passing Interface
- Using MPI-2 Advanced Features of the Message-passing Interface
- Parallel Programming in Fortran 95 using OpenMP

谢谢！