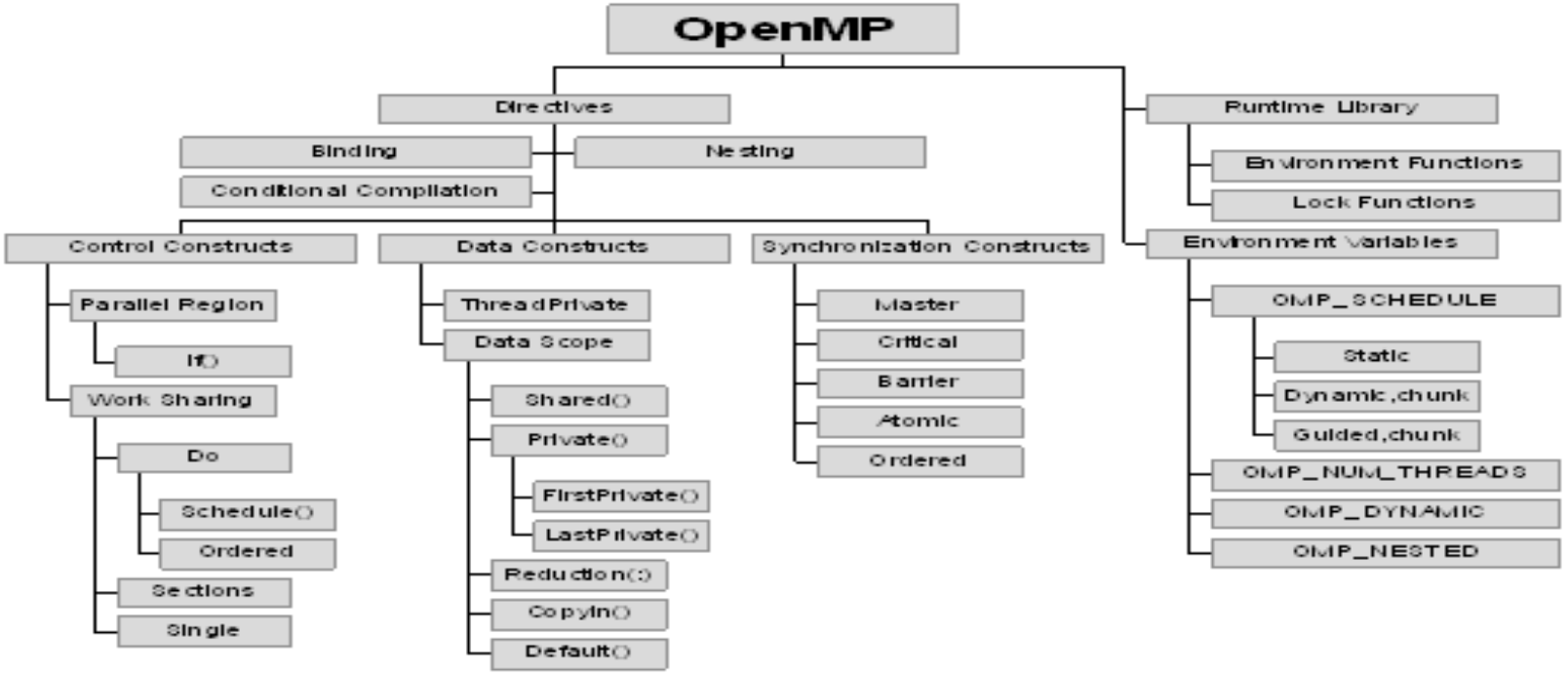
OpenMP编程技术

习题课

1.A.3 OpenMP Constructs



1. Overview 16

- Parallel and work sharing directives
- data environment directives
- synchronization directives

上节课的回顾

- OpenMP
  - 一种简单的方法支持共享存储编程模型（多线程编程），易用，支持快速开发
  - Fork-join model，可动态管理并行度
  - 支持增量开发
- We explored basic OpenMP coding on how to:
  - 定义并行区 (**omp parallel**)
  - 设置并行度
  - 并行结构 (**omp for; omp sections**)
  - 任务分配(**schedule**)
  - 数据管理/变量分类 (**omp private/shared**)
  - 同步控制 (**omp critical, atomic** …)

线程数控制的讨论

- 通常情况下线程组内线程数目由环境变量OMP_NUM_THREADS控制

- 如果parallel语句有num_threads子句，或者用户调用了omp_set_num_threads函数，线程数目由它们给出，num_threads具有高优先级

- 环境变量、 num_threads与omp_set_num_threads作用域分别为系统、并行块级以及程序级

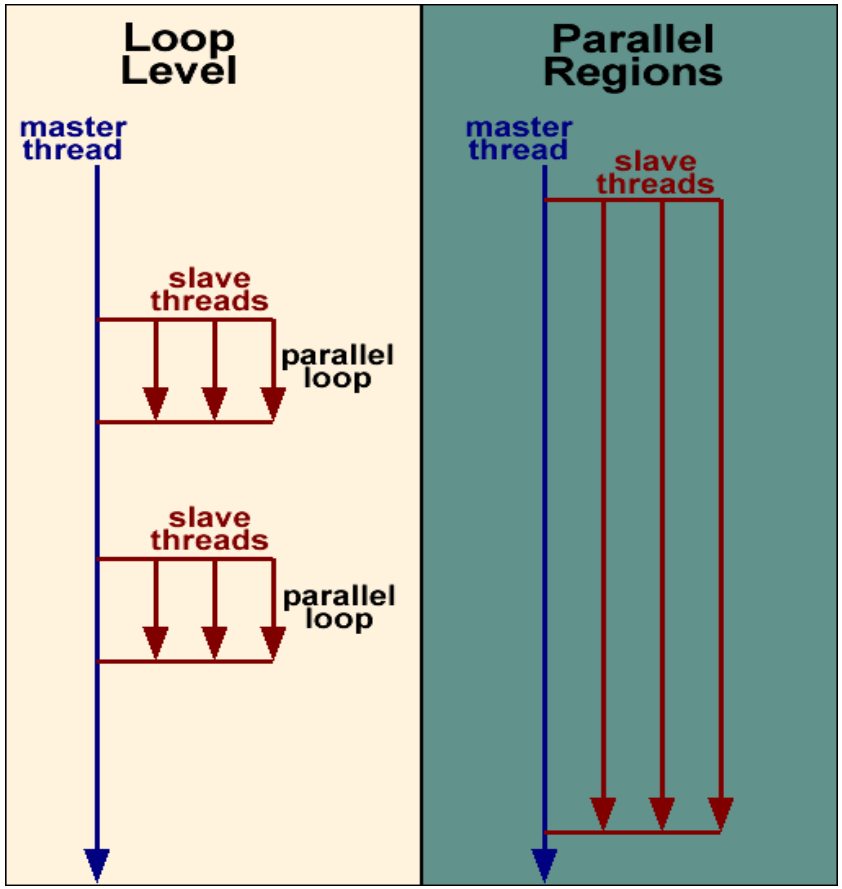
- 这里给出的线程数目可以大于系统中处理器个数，它是一个上限值，系统实际产生的线程数目可能由于资源的限制而比上限值要小

OpenMP并行编程

1. 如何发掘并行机会？选择并行结构
2. 如何分配任务？任务划分
3. **如何协调？分析数据依赖关系，管理同步**
4. 性能123

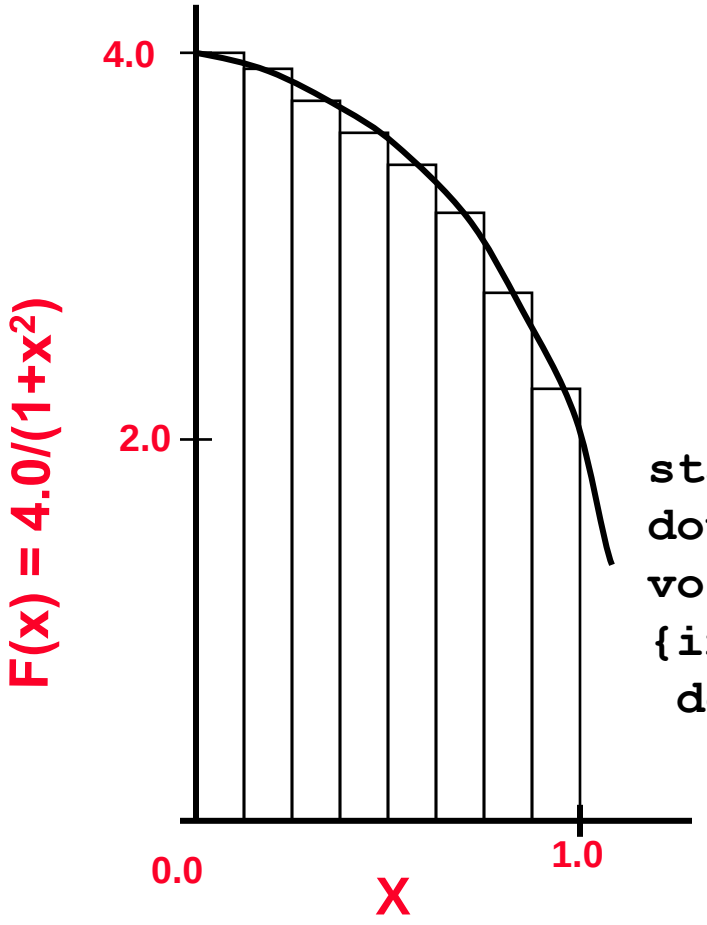
并行结构

Divide loop iterations among threads: We will focus mainly on loop level parallelism in this lecture



Divide various sections of code between threads

任务划分



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

```
static long num_steps = 100000;
double step;
void main ()
{int I;
 double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
       x = (i+0.5)*step;
       sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

任务划分 - 人工

```
#include <omp.h>
#define NUM_THREADS 2
static long num_steps = 100000;
double step;
void main ()
{       int i;
        double x, pi, sum[NUM_THREADS] ={0};
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
        {       double x;
                int id, i;
                id = omp_get_thread_num();
                int nthreads = omp_get_num_threads();
                for (i=id;i< num_steps; i=i+nthreads){
                  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;
}
```

SPMD Programs:

Each thread runs the same code with the thread ID selecting any thread specific behavior.

任务划分 - 自动

```
#include <omp.h>
#define NUM_THREADS 2
static long num_steps = 100000;
double step;
void main ()
{       int i;
        double x, pi, sum[NUM_THREADS] ={0.0};
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
        {       double x;
                int i, id;
                id = omp_get_thread_num();
#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;
}
```

Work Sharing Programs:

Each thread runs the same code with the system selecting the proper iteration count for each thread.

Scheduling Strategies

OpenMP supports three scheduling strategies:

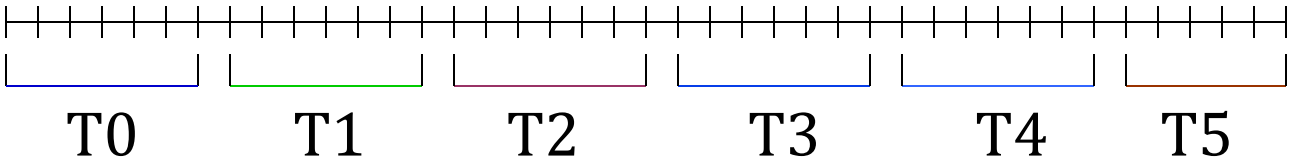
- ***Static***: **The default**, as described in the previous slides – good for iterations that are inherently load balanced.

- ***Dynamic***: Each thread gets a chunk of a few iterations, and when it finishes that chunk it goes back for more, and so on until all of the iterations are done – good when iterations aren't load balanced at all.

- ***Guided***: Each thread gets smaller and smaller chunks over time – a compromise.

Static Scheduling

For N_i iterations and N_t threads, each thread gets one chunk of N_i/N_t loop iterations:



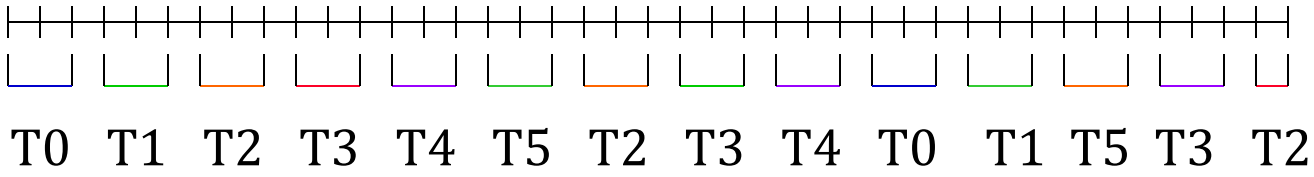
- Thread #0: iterations 0 through N_i/N_t-1
- Thread #1: iterations N_i/N_t through $2N_i/N_t$-1
- Thread #2: iterations $2N_i/N_t$ through $3N_i/N_t$-1

…

- Thread #N_t-1: iterations $(N_t\text{-}1)N_i/N_t$ through N_i-1

Dynamic Scheduling

For N_i iterations and N_t threads, each thread gets a fixed-size chunk of k loop iterations:

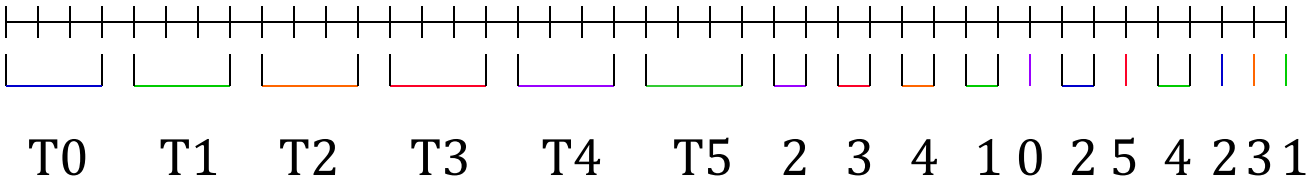


When a particular thread finishes its chunk of iterations, it gets assigned a new chunk. So, the relationship between iterations and threads is nondeterministic.

- Advantage: very flexible
- Disadvantage: high overhead – lots of decision making about which thread gets each chunk

Guided Scheduling

For N_i iterations and N_t threads, initially each thread gets a fixed-size chunk of $k < N_i/N_t$ loop iterations:



After each thread finishes its chunk of k iterations, it gets a chunk of $k/2$ iterations, then $k/4$, etc. Chunks are assigned dynamically, as threads finish their previous chunks.

- Advantage over static: can handle imbalanced load
- Advantage over dynamic: fewer decisions, so less overhead

任务划分

Schedule Clause	When To Use
STATIC	Predictable and similar work per iteration
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

How to Know Which Schedule in practice?

Try and Test:

- Test all three using a typical case as a ***benchmark***.
- Whichever wins is probably the one you want to use most of the time on that particular platform.
- This may vary depending on problem size, new versions of the compiler, who's on the machine, what day of the week it is, etc, so you may want to benchmark the three schedules from time to time.

- 选择并行结构(受限)
- 任务划分
- **分析数据依赖关系，管理同步**
  - **数据变量管理**
  - **数据相关性分析**
  - **同步机制**

数据变量管理

- 变量共享的“规则”
  - Most variables are shared by default
  - Global variables are SHARED among threads
    - **Fortran: COMMON blocks, SAVE variables, MODULE variables**
    - **C: File scope variables, static**
  - But not everything is shared...
    - **Stack variables in sub-programs called from parallel regions are PRIVATE**
    - **Automatic variables within a statement block are PRIVATE.**

数据变量管理

- 主动管理变量
  - SHARED
  - PRIVATE
    - The value is **uninitialized**
    - Private copy is *not* storage associated with the original
  - FIRSTPRIVATE
    - Initializes each private copy with the corresponding value from the master thread.
  - THREADPRIVATE
    - Preserves global scope for per-thread storage
    - Legal for name-space-scope and file-scope
    - Use copyin to initialize from master thread
  - LASTPRIVATE
    - the value of a private from the last iteration to a global variable.
  - **DEFAULT (PRIVATE | SHARED | NONE)**

共享变量真的共享了吗?

Memory Model

- OpenMP provides a "relaxed-consistency" and "temporary" view of thread memory (in their words). In other words, threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time.

- When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is FLUSHed by all threads as needed.

flush

- The FLUSH directive identifies a synchronization point at which the implementation must provide a consistent view of memory. Thread-visible variables are written back to memory at this point.

Fortran	!$OMP FLUSH *(list)*
C/C++	#pragma omp flush *(list) newline*

默认flush情况

Fortran	C/C++
BARRIER END PARALLEL CRITICAL and END CRITICAL END DO END SECTIONS END SINGLE ORDERED and END ORDERED	barrier parallel - upon entry and exit critical - upon entry and exit ordered - upon entry and exit for - upon exit sections - upon exit single - upon exit

Thread stack

- Each thread has its own memory region called the thread stack
- This can grow to be quite large, so default size may not be enough
- This can be increased (e.g. to 16 MB):

csh:

limit stacksize 16000; setenv KMP_STACKSIZE 16000000

bash:

ulimit -s 16000; export KMP_STACKSIZE=16000000

数据相关性分析

```
#pragma omp parallel for
for(int i=0; i<N; i++)
{
    a[i] = a[i]+1.0;
}

#pragma omp parallel for
for(int i=1; i<N; i++)
{
    a[i] = a[i]+a[i-1];
}
```

上面的例子分析

```
#pragma omp parallel for
for(int i=1; i<=4; i++)
{
    a[i] = a[i]+a[i-1];
}
```

- 假设条件：2个线程，静态调度
- a=[1,1,1,1,1,...,1]

Thread 1	
a[1]=a[1]+a[0];	2
a[2]=a[2]+a[1];	3

Thread 2	
a[3]=a[3]+a[2];	2
a[4]=a[4]+a[3];	3

实际上

```
Serial
a[1]=a[1]+a[0];      2
a[2]=a[2]+a[1];      3
a[3]=a[3]+a[2];      4
a[4]=a[4]+a[3];      5
```

- 多线程程序打破了原有串行程序中的读写顺序，导致程序错误

```
Thread 1
a[1]=a[1]+a[0];
a[2]=a[2]+a[1];
```

```
Thread 2
a[3]=a[3]+a[2];
a[4]=a[4]+a[3];
```

如何来定义这种问题—数据相关性

- 多条指令访问同一内存位置
- 其中至少有一条指令是赋值操作

```
#pragma omp parallel for
for(int i=0; i<N; i++)
{
    a[i] = a[i]+1.0;
}
```

- 同一内存位置如何理解？
  - 可实现标量赋值
- 还记得以前提过的数据竞争嘛？
- **竞争**
  - **两个处理器(或两个线程) 访问一个变量，且其中至少有一个写者**
  - **两个访问同时发生**

OpenMP关心的数据相关性

```
#pragma omp parallel for
for(int i=1; i<=4; i++)
{
        a[i] = 1.0;
        a[i] = a[i]+ 2.0;
}
```

这两个程序中存在数据相关性嘛？

```
#pragma omp parallel for
for(int i=1; i<=4; i++)
{
    a[i] = a[i]+a[i-1];
}
```

Loop-carried

数据相关性分析

```
for(int i=1; i<=N; i=i+2)
{
    a[i] = a[i]+a[i-1];
}

for(int i=0; i<N/2; i++)
{
    a[i] = a[i]+a[i+N/2];
}

for(int i=0; i<N/2+1; i++)
{
    a[i] = a[i]+a[i+N/2];
}

for(int i=0; i<N; i=i+2)
{
    a[idx[i]] = a[idx[i]]+b[idx[i]];
}
```

数据相关性分析

```
for(int i=1; i<=N; i=i+2)
{
    a[i] = a[i]+a[i-1];          // 数据不相关
}

for(int i=0; i<N/2; i++)
{
    a[i] = a[i]+a[i+N/2];    // 数据不相关
}

for(int i=0; i<N/2+1; i++)
{
    a[i] = a[i]+a[i+N/2];    // 数据相关
}

for(int i=0; i<N; i=i+2)
{
    a[idx[i]] = a[idx[i]]+b[idx[i]];
                                        // 不一定,要看idx的具体情况
}
```

数据相关性分析

```
//矩阵相乘

for(int j=1; j<=N; j=j++)
{
        for(int i=1;i<=N;i++)
        {
                c[i][j]=0;
                for(int k=1;k<=N;k++)
                {
                        c[i][j]=c[i][j]+a[i][k]*b[k][j];
                }
        }
}
```

数据相关性分析

```
for(int j=1; j<=N; j=j++)
{           // 外循环loop-carried数据不相关
            for(int i=1;i<=N;i++)
            {
                        c[i][j]=0;
                        for(int k=1;k<=N;k++)
                        {           // 内循环数据相关
                                    c[i][j]=c[i][j]+a[i][k]*b[k][j];
                        }
            }
}
```

数据相关性分类

```
S1:  A = 1.0;
S2:  B = A + 3.14;
S3:  A = 1/3 * (C – D);
. . . . . .
S4:  A = (B * 3.8) / 2.7;
```

数据相关性的分类

- Consider the serial code:

```
S1:  A = 1.0;
S2:  B = A + 3.14;
S3:  A = 1/3 * (C – D);
. . . . . .
S4:  A = (B * 3.8) / 2.7;
```

- Flow dependence between S1 and S2
  - Value of A updated in S1 is used in S2
- Anti dependence between S2 and S3
  - Value of A is read in S2 before written in S3
- Output dependence between S3 and S4
  - Value of A assigned in S3 must occur before assignment in S4

讨论

```
for(int i=2; i<N; i++)
{
S1:     x=d[i]+i;
S2:     a[i]=a[i+1]+x;
S3:     b[i]=b[i]+b[i-1]+d[i-1];
S4:     C[2]=2*i;
}
```

- Tell me how many dependences in this case?
  - Loop carried + non loop-carried

Solution

变量	前面的语句 (语句编号;迭代次数; 访问类型)	后面的语句 (语句编号;迭代次数; 访问类型)	Loop-carried ?	相关性类型
x	S1; i; write	S2; i; read	no	flow
x	S1; i; write	S1; i+1; write	yes	output
a(i+1)	S2; i; read	S2; i+1; write	yes	anti
b(i)	S3; i; write	S3; i+1; read	yes	flow
C[2]	S4; i; write	S3; i+1; write	yes	output
x	S2; i; read	S1; i+1; write	yes	anti
x	S1; i; write	S2; i+1; read	yes	flow

存在loop-carried数据相关性如何寻找并行性？

1. Loop 转换

2. 同步控制

在数据相关的条件下寻求并行性(case 1)

```
for(int i=1; i<N; i++)
{
        x=(b[i]+c[i])/2;
        a[i]=a[i+1]+x;
}
```

在数据相关的条件下寻求并行性(case 1)

```
for(int i=1; i<N; i++)
{
        x=(b[i]+c[i])/2;
        a[i]=a[i+1]+x;
}
```

```
#pragma omp parallel for shared(a,a2)
for(int i=1;i<N;i++)
        a2[i]=a[i+1];

#pragma omp parallel for private(x) shared(a,a2,b,c)
for(int i=1; i<N; i++)
{
        x=(b[i]+c[i])/2;
        a[i]=a2[i]+x;
}
```

在数据相关的条件下寻求并行性(case 2)

```
for(int i=1; i<N; i++)
{
        x=(b[i]+c[i])/2;
        a[i]=a[i]+x;
        d[1]=2*x;
}

y=x+d[1]+d[2];
```

Lastprivate Clause

- Variables update shared variable using value from last iteration
- C++ objects are updated as if by assignment

```
void sq2(int n,  double *lastterm)
{
  double x; int i;
  #pragma omp parallel
  #pragma omp for lastprivate(x)
  for (i = 0; i < n; i++){
    x = a[i]*a[i] + b[i]*b[i];
    b[i] = sqrt(x);
  }
  lastterm = x;
}
```

solution

```
#pragma omp parallel for lastprivate(x,d1) shared(a,b,c)
for(int i=1; i<N; i++)
{
        x=(b[i]+c[i])/2;
        a[i]=a[i]+x;
        d1=2*x;
}
d[1]=d1;
y=x+d[1]+d[2];
```

Any improvement?

在数据相关的条件下寻求并行性(case3)

```
for(int i=1; i<N; i++)
{
        x=(b[i]+c[i])/2;
        a[i]=a[i]+x;
        d[1]=2*x;                    --- if(i%2==0)…
}

y=x+d[1]+d[2];
```

在数据相关的条件下寻求并行性(Case 4)

```
x=0;
for(int i=1; i<N; i++)
{
        x= x + a[i];
}
```

在数据相关的条件下寻求并行性(Case 4)

```
x=0;
for(int i=1; i<N; i++)
{
        x= x + a[i];
}
```

```
x=0;
#pragma omp parallel for reduction(+: x)
for(int i=1; i<N; i++)
{
        x= x + a[i];
}
```

在数据相关的条件下寻求并行性(Case 5)

```
for(int i=2; i<=N; i++)
{
S1:      b[i]= b[i] + a[i-1];
S2:      a[i]=    a[i] + c[i];
}
```

在数据相关的条件下寻求并行性(Case 5)

```
for(int i=2; i<=N; i++)
{
S1:      b[i]= b[i] + a[i-1];
S2:      a[i]=    a[i] + c[i];
}
```

```
b[2]=b[2]+a[1];
#pragma omp parallel for shared(a,b,c)
for(int i=2; i<N; i++)
{
        a[i]= a[i] + c[i];
        b[i+1]= b[i+1] + a[i];
}
a[N]=a[N]+c[N];
```

在数据相关的条件下寻求并行性(Case 6)

```
for(int i=2; i<=N; i++)
{
        a[i]= (a[i] + a[i-1])/2;
        y = y + c[i];
}
```

在数据相关的条件下寻求并行性(Case 6)

```
for(int i=2; i<=N; i++)
{
        a[i]= (a[i] + a[i-1])/2;
        y = y + c[i];
}
```

```
for(int i=2; i<=N; i++)
{
        a[i]= (a[i] + a[i-1])/2;
}
#pragma omp parallel for reduction(+:y)
for(int i=2; i<=N; i++)
{
        y = y + c[i];
}
```

在数据相关的条件下寻求并行性(Case 7)

```
for(int i=1; i<=N; i++)
{
        y = y + a[i];
        b[i]= (b[i] + c[i])*y;
}
```

在数据相关的条件下寻求并行性(Case 7)

```
for(int i=1; i<=N; i++)
{
        y = y + a[i];
        b[i]= (b[i] + c[i])*y;
}
```

```
y1[1] = y+a[1];
for(int i=2; i<=N; i++)
{
        y1[i] = y1[i-1] + a[i];
}
y=y1[N];

#pragma omp parallel for shared(b,c,y1)
for(int i=1; i<=N; i++)
{
        b[i]= (b[i] + c[i])*y1[i];
}
```

同步机制

- OpenMP has the following constructs to support synchronization:
  - critical
  - atomic(受限)
  - barrier
  - ordered(跟随的结构块串行执行)
  - single
  - Master
  - Reduce(语义受限)
- 隐式同步——性能问题

  parallel
  for (except when nowait is used)
  sections (except when nowait is used)
  single (except when nowait is used)

小结

- 数据相关是导致多线程程序出错的原因
- 今天的内容：解决OpenMP(多线程) loop-carried 数据相关问题
  - 分析数据相关
  - 分类数据相关
  - 解决各类数据相关
    - 数据访问局部性分析
    - 数据相关性解耦
      - Expanding a scalar into array
      - loop transform
      - Fission into serial and parallel parts
      - Lastprivate
      - Reduce
    - 同步管理

MPI编程技术

习题课

消息传递功能需求

- 共享存储编程(OpenMP)
  - 定义并行区
  - 设置并行度
  - 并行结构
  - 任务分配
  - 数据管理/变量分类
  - 同步控制
- 消息传递编程(MPI)
  - 定义并行区
  - 设置并行度(静态、动态)
  - 程序员完成并行结构设计
  - 程序员完成任务分配
  - 通信管理
  - 同步控制(MPI_Barrier)

MPI在使用上更加灵活，接口也更加复杂

MPI接口卡(1)

Message Passing Interface
Quick Reference in C

```
#include <mpi.h>
```

Blocking Point-to-Point

Send a message to one process. (§3.2.1)
```
int MPI_Send (void *buf, int count,
   MPI_Datatype datatype, int dest, int
   tag, MPI_Comm comm)
```
Receive a message from one process. (§3.2.4)
```
int MPI_Recv (void *buf, int count,
   MPI_Datatype datatype, int source, int
   tag, MPI_Comm comm, MPI_Status *status)
```
Count received data elements. (§3.2.5)
```
int MPI_Get_count (MPI_Status *status,
   MPI_Datatype datatype, int *count)
```
Wait for message arrival. (§3.8)
```
int MPI_Probe (int source, int tag,
   MPI_Comm comm, MPI_Status *status)
```
Related Functions: MPI_Bsend, MPI_Ssend, MPI_Rsend, MPI_Buffer_attach, MPI_Buffer_detach, MPI_Sendrecv, MPI_Sendrecv_replace, MPI_Get_elements

Non-blocking Point-to-Point

Begin to receive a message. (§3.7.2)
```
int MPI_Irecv (void *buf, int count,
   MPI_Datatype, int source, int tag,
   MPI_Comm comm, MPI_Request *request)
```
Complete a non-blocking operation. (§3.7.3)
```
int MPI_Wait (MPI_Request *request,
   MPI_Status *status)
```
Check or complete a non-blocking operation. (§3.7.3)
```
int MPI_Test (MPI_Request *request, int
   *flag, MPI_Status *status)
```
Check message arrival. (§3.8)
```
int MPI_Iprobe (int source, int tag,
   MPI_Comm comm, int *flag, MPI_Status
   *status)
```
Related Functions: MPI_Isend, MPI_Ibsend, MPI_Issend, MPI_Irsend, MPI_Request_free, MPI_Waitany, MPI_Testany, MPI_Waitall, MPI_Testall, MPI_Waitsome, MPI_Testsome, MPI_Cancel, MPI_Test_cancelled

Persistent Requests

Related Functions: MPI_Send_init, MPI_Bsend_init, MPI_Ssend_init, MPI_Rsend_init, MPI_Recv_init, MPI_Start, MPI_Startall

Derived Datatypes

Create a strided homogeneous vector. (§3.12.1)
```
int MPI_Type_vector (int count, int
   blocklength, int stride, MPI_Datatype
   oldtype, MPI_Datatype *newtype)
```
Save a derived datatype (§3.12.4)
```
int MPI_Type_commit (MPI_Datatype
   *datatype)
```
Pack data into a message buffer. (§3.13)
```
int MPI_Pack (void *inbuf, int incount,
   MPI_Datatype datatype, void *outbuf,
   int outsize, int *position, MPI_Comm
   comm)
```
Unpack data from a message buffer. (§3.13)
```
int MPI_Unpack (void *inbuf, int insize,
   int *position, void *outbuf, int
   outcount, MPI_Datatype datatype,
   MPI_Comm comm)
```
Determine buffer size for packed data. (§3.13)
```
int MPI_Pack_size (int incount,
   MPI_Datatype datatype, MPI_Comm comm,
   int *size)
```
Related Functions: MPI_Type_contiguous, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_hindexed, MPI_Type_struct, MPI_Address, MPI_Type_extent, MPI_Type_size, MPI_Type_lb, MPI_Type_ub, MPI_Type_free

Collective

Send one message to all group members. (§4.4)
```
int MPI_Bcast (void *buf, int count,
   MPI_Datatype datatype, int root,
   MPI_Comm comm)
```
Receive from all group members. (§4.5)
```
int MPI_Gather (void *sendbuf, int
   sendcount, MPI_Datatype sendtype, void
   *recvbuf, int recvcount, MPI_Datatype
   recvtype, int root, MPI_Comm comm)
```
Send separate messages to all group members. (§4.6)
```
int MPI_Scatter (void *sendbuf, int
   sendcount, MPI_Datatype sendtype, void
   *recvbuf, int recvcount, MPI_Datatype
   recvtype, int root, MPI_Comm comm)
```
Combine messages from all group members. (§4.9.1)
```
int MPI_Reduce (void *sendbuf, void
   *recvbuf, int count, MPI_Datatype
   datatype, MPI_Op op, int root, MPI_Comm
   comm)
```
Related Functions: MPI_Barrier, MPI_Gatherv, MPI_Scatterv, MPI_Allgather, MPI_Allgatherv, MPI_Alltoall, MPI_Alltoallv, MPI_Op_create, MPI_Op_free, MPI_Allreduce, MPI_Reduce_scatter, MPI_Scan

Groups

Related Functions: MPI_Group_size, MPI_Group_rank, MPI_Group_translate_ranks, MPI_Group_compare, MPI_Comm_group, MPI_Group_union, MPI_Group_intersection, MPI_Group_difference, MPI_Group_incl, MPI_Group_excl, MPI_Group_range_incl, MPI_Group_range_excl, MPI_Group_free

Basic Communicators

Count group members in communicator. (§5.4.1)
```
int MPI_Comm_size (MPI_Comm comm, int
   *size)
```
Determine group rank of self. (§5.4.1)
```
int MPI_Comm_rank (MPI_Comm comm, int
   *rank)
```
Duplicate with new context. (§5.4.2)
```
int MPI_Comm_dup (MPI_Comm comm, MPI_Comm
   *newcomm)
```
Split into categorized sub-groups. (§5.4.2)
```
int MPI_Comm_split (MPI_Comm comm, int
   color, int key, MPI_Comm *newcomm)
```
Related Functions: MPI_Comm_compare, MPI_Comm_create, MPI_Comm_free,

MPI接口卡(2)

MPI_Comm_test_inter, MPI_Comm_remote_size, MPI_Comm_remote_group, MPI_Intercomm_create, MPI_Intercomm_merge

Communicators with Topology

Create with cartesian topology. (§6.5.1)
```
int MPI_Cart_create (MPI_Comm comm_old,
    int ndims, int *dims, int *periods, int
    reorder, MPI_Comm *comm_cart)
```
Suggest balanced dimension ranges. (§6.5.2)
```
int MPI_Dims_create (int nnodes, int
    ndims, int *dims)
```
Determine rank from cartesian coordinates. (§6.5.4)
```
int MPI_Cart_rank (MPI_Comm comm, int
    *coords, int *rank)
```
Determine cartesian coordinates from rank. (§6.5.4)
```
int MPI_Cart_coords (MPI_Comm comm, int
    rank, int maxdims, int *coords)
```
Determine ranks for cartesian shift. (§6.5.5)
```
int MPI_Cart_shift (MPI_Comm comm, int
    direction, int disp, int *rank_source,
    int *rank_dest)
```
Split into lower dimensional sub-grids. (§6.5.6)
```
int MPI_Cart_sub (MPI_Comm comm, int
    *remain_dims, MPI_Comm *newcomm)
```
Related Functions: MPI_Graph_create, MPI_Topo_test, MPI_Graphdims_get, MPI_Graph_get, MPI_Cartdim_get, MPI_Cart_get, MPI_Graph_neighbors_count, MPI_Graph_neighbors, MPI_Cart_map, MPI_Graph_map

Communicator Caches

Related Functions: MPI_Keyval_create, MPI_Keyval_free, MPI_Attr_put, MPI_Attr_get, MPI_Attr_delete

---

LAM & MPI Information

1224 Kinnear Rd.
Columbus, Ohio 43212
614-292-8492

lam@tbag.osc.edu

http://www.osc.edu/lam.html
ftp://tbag.osc.edu/pub/lam

Error Handling

Related Functions: MPI_Errhandler_create, MPI_Errhandler_set, MPI_Errhandler_get, MPI_Errhandler_free, MPI_Error_string, MPI_Error_class

Environmental

Determine wall clock time. (§7.4)
```
double MPI_Wtime (void)
```
Initialize MPI. (§7.5)
```
int MPI_Init (int *argc, char ***argv)
```
Cleanup MPI. (§7.5)
```
int MPI_Finalize (void)
```
Related Functions: MPI_Get_processor_name, MPI_Wtick, MPI_Initialized, MPI_Abort, MPI_Pcontrol

Constants

Wildcards (§3.2.4)
```
MPI_ANY_TAG, MPI_ANY_SOURCE
```
Elementary Datatypes (§3.2.2)
```
MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG,
MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT,
MPI_UNSIGNED, MPI_UNSIGNED_LONG,
MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE,
MPI_BYTE, MPI_PACKED
```
Reserved Communicators (§5.2.4)
```
MPI_COMM_WORLD, MPI_COMM_SELF
```
Reduction Operations (§4.9.2)
```
MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD,
    MPI_BAND, MPI_BOR, MPI_BXOR, MPI_LAND,
    MPI_LOR, MPI_LXOR
```

LAM Quick Reference

LAM / MPI Extensions

Spawn processes.
```
int MPIL_Spawn (MPI_Comm comm, char *app,
    int root, MPI_Comm *child_comm);
```
Get communicator ID.
```
int MPIL_Comm_id (MPI_Comm comm, int *id);
```
Deliver an asynchronous signal.
```
int MPIL_Signal (MPI_Comm comm, int rank,
    int signo);
```
Enable trace collection.
```
int MPIL_Trace_on (void);
```
Related Functions: MPIL_Comm_parent, MPIL_Universe_size, MPIL_Type_id, MPIL_Comm_gps, MPIL_Trace_off

Session Management

Confirm a group of hosts.
```
recon -v <hostfile>
```
Start LAM on a group of hosts.
```
lamboot -v <hostfile>
```
Terminate LAM.
```
wipe -v <hostfile>
```
Hostfile Syntax
```
# comment
<hostname> <userid>
<hostname> <userid>
...etc...
```

Compilation

Compile a program for LAM / MPI.
```
hcc -o <binary> <source> -I<incdir>
    -L<libdir> -l<lib> -lmpi
```

Processes and Messages

Start an SPMD application.
```
mpirun -v -s <src_node> -c <copies>
    <nodes> <program> -- <args>
```
Start a MIMD application.
```
mpirun -v <appfile>
```
Appfile Syntax
```
# comment
<program> -s <src_node> <nodes> -- <args>
<program> -s <src_node> <nodes> -- <args>
...etc...
```
Examine the state of processes.
```
mpitask
```
Examine the state of messages.
```
mpimsg
```
Cleanup all processes and messages.
```
lamclean -v
```

Message Passing Libraries

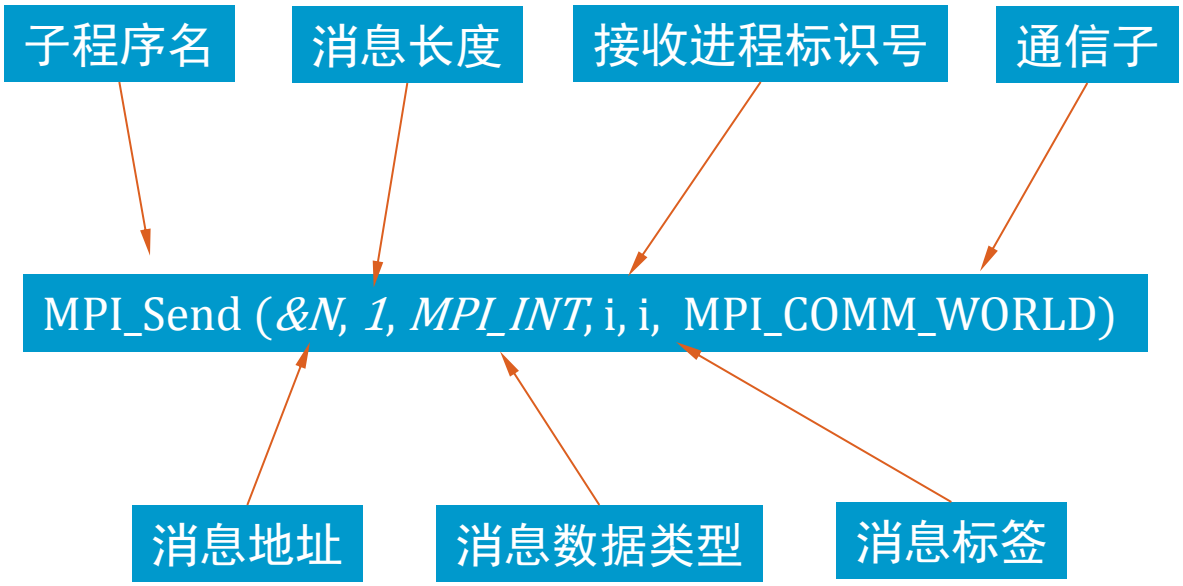
- All communication, synchronization require subroutine calls
  - No shared variables
  - Program runs on a single processor just like any uniprocessor program, except for calls to message passing library
- Subroutines for
  - Communication
    - Pairwise or point-to-point: Send and Receive
    - Collectives all processor get together to
      - Move data: Broadcast, Scatter/gather
      - Compute and move: sum, product, max, … of data on many processors
  - Synchronization
    - Barrier
    - **No locks because there are no shared variables to protect**
  - Enquiries
    - How many processes? Which one am I? Any messages waiting?

Novel Features of MPI

- Communicators encapsulate communication spaces for library safety
- Datatypes reduce copying costs and permit heterogeneity
- Extensive collective operations for scalable global communication
- Multiple communication modes allow precise buffer management
- Process topologies permit efficient process placement, user views of process layout
- Profiling interface encourages portable tools

Slide source: Bill Gropp, ANL

MPI中的消息

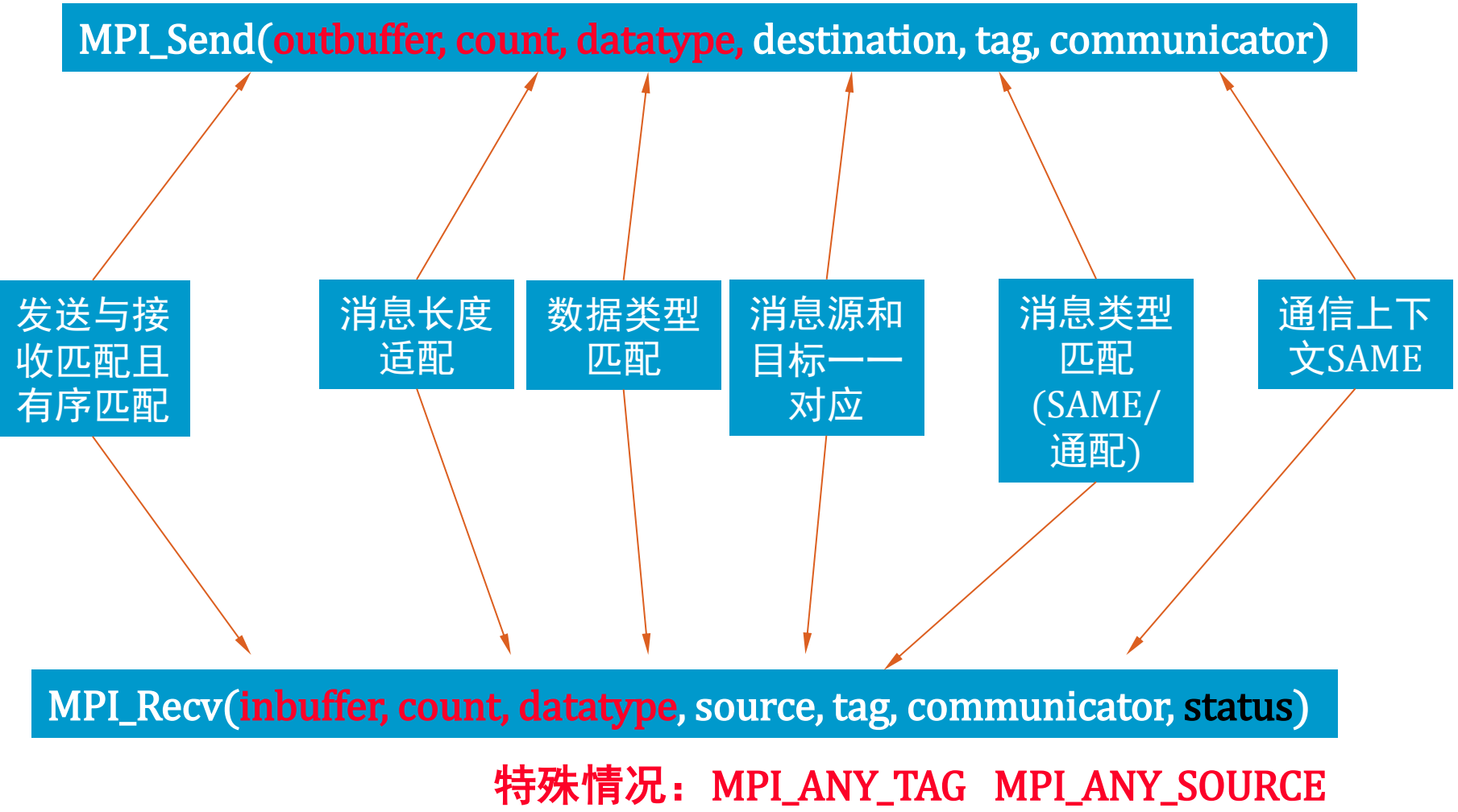


MPI_Send (buffer, count, datatype, destination, tag, communicator)

(buffer, count, datatype)　　消息缓冲

(destination, tag, communicator)　　消息信封

消息管理



特殊情况：MPI_ANY_TAG　MPI_ANY_SOURCE

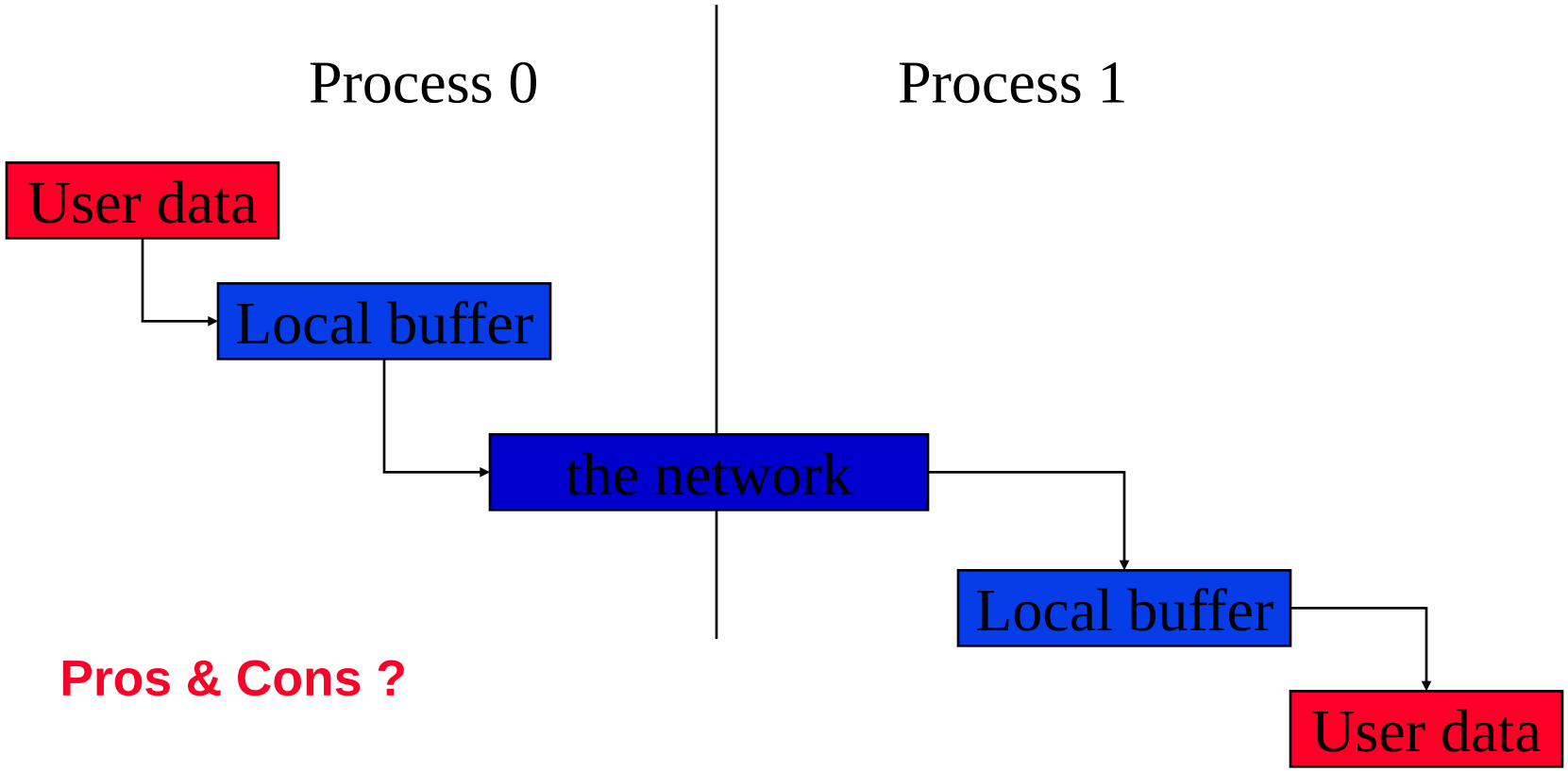
More on Message Passing

- Message passing is a simple programming model, but there are some special issues
  - Buffering and deadlock
  - Deterministic execution (老生常谈)
  - Performance (future talk)

Slide source: Bill Gropp, ANL

Buffers

- When you send data, where does it go? One possibility is:



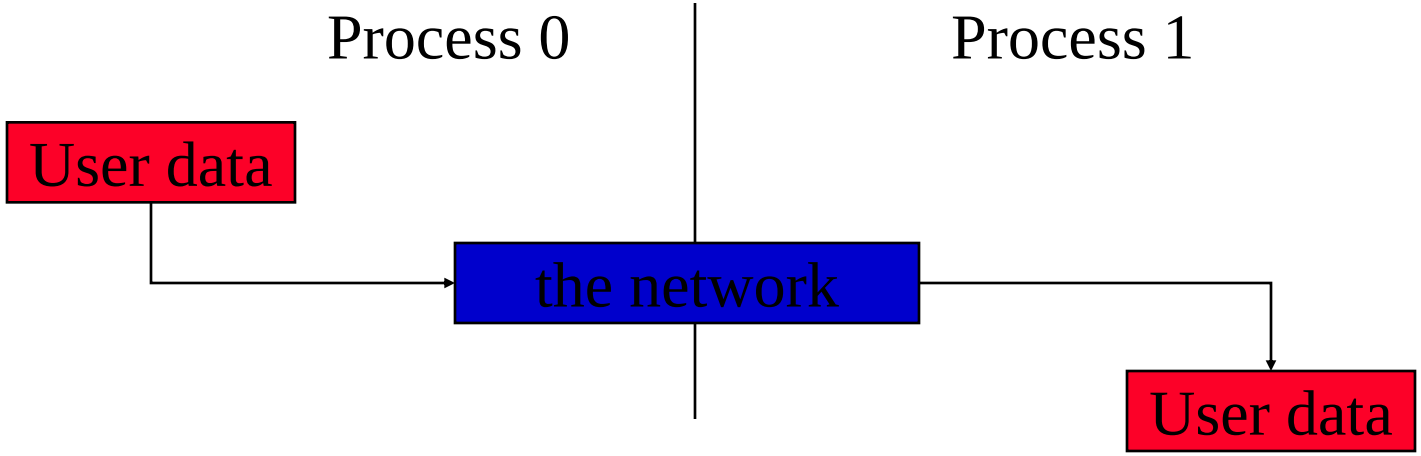
Pros & Cons ?

Slide source: Bill Gropp, ANL

Avoiding Buffering

- It is better to avoid copies:

Pros & Cons ?



This requires that **MPI_Send** wait on delivery, or that **MPI_Send** return before transfer is complete, and we wait later.

Slide source: Bill Gropp, ANL

Blocking Communication

- So far we have been using *blocking* communication:
  - MPI_Recv does not complete until the buffer is full (available for use).
  - MPI_Send does not complete until the buffer is empty (available for use).
- Completion depends on size of message and amount of system buffering.

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

如果调整send
与recv先后顺序会怎样？

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

- This is called “unsafe” because it depends on the availability of system buffers in which to store the data sent until it can be received

Slide source: Bill Gropp, ANL

Some Solutions to the "unsafe" Problem

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

- Supply receive buffer at same time as send:

Process 0	Process 1
`Sendrecv(1)`	`Sendrecv(0)`

Slide source: Bill Gropp, ANL

Send-Receive

- Combine into one call sending of message to one destination and receipt of message from another process – not necessarily same one, but within same communicator
- Message sent by send-receive can be received by regular receive or probe
- Send-receive can receive message sent by regular send operation
- Send-receive is blocking
- `int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`

More Solutions to the "unsafe" Problem

- Supply own space as buffer for send

Process 0	Process 1
`Bsend(1)`	`Bsend(0)`
`Recv(1)`	`Recv(0)`

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`
`Waitall`	`Waitall`

非阻塞通信

```
int MPI_Isend(void *buf,   int count,  MPI_Datatype dtype,
              int dest,    int tag,    MPI_Comm comm,
              MPI_Request *req)
```

```
int MPI_Irecv(void *buf,   int count,  MPI_Datatype dtype,
              int src,     int tag,    MPI_Comm comm,
              MPI_Request *req)
```

非阻塞通信

- 非阻塞通信完成检查
  - 函数MPI_WAIT和MPI_TEST用于完成一个非阻塞通信
    - 一个是阻塞检查
    - MPI_Wait(MPI_Request * request, MPI_Status * status)
      INOUT request 请求(句柄)
      OUT status 状态对象(状态类型)
    - 另一个是非阻塞检查
    - MPI_TEST(MPI_Request * request, int *flag, MPI_Status * status)
      INOUT request 通信请求(句柄)
      OUT flag 如果操作完成则为真(逻辑型)
      OUT status 状态对象(状态类型)

Multiple Completions

- Want to await completion of any, some, or all communications, instead of specific message
- Use `MPI_{Wait,Test}{any,all,some}` for this purpose
  - `any`: `Waits` or `Tests` for any one option in array of requests to complete
  - `all`: `Waits` or `Tests` for all options in array of requests to complete
  - `some`: `Waits` or `Tests` for all enabled operations in array of requests to complete

Blocking vs. Non-blocking

Blocking

- A blocking send routine will only return after it is *safe* to modify the buffer.
- *Safe* means that modification will not affect the data to be sent.
- *Safe* does not imply that the data was actually received.

Non-blocking

- Send/receive routines return immediately.
- Non-blocking operations request that the MPI library perform the operation "when possible".
- It is **unsafe** to modify the buffer until the requested operation has been performed. There are *wait* routines used to do this (MPI_Wait).
- Primarily used to overlap computation with communication.

点对点的通信

- 最佳性能可能是采用MPI_Ssend写出的程序
- 最常用的是MPI_Send
- 用MPI_Isend / MPI_Irecv去开发计算与通信重叠
- MPI_Bsend 只在不方便应用 MPI_Isend的场合使用
- 其他接口在应用程序中甚少使用

Exercises (网络学堂：mpiexmpl.tar.gz)

Simple Parallel Data Structures

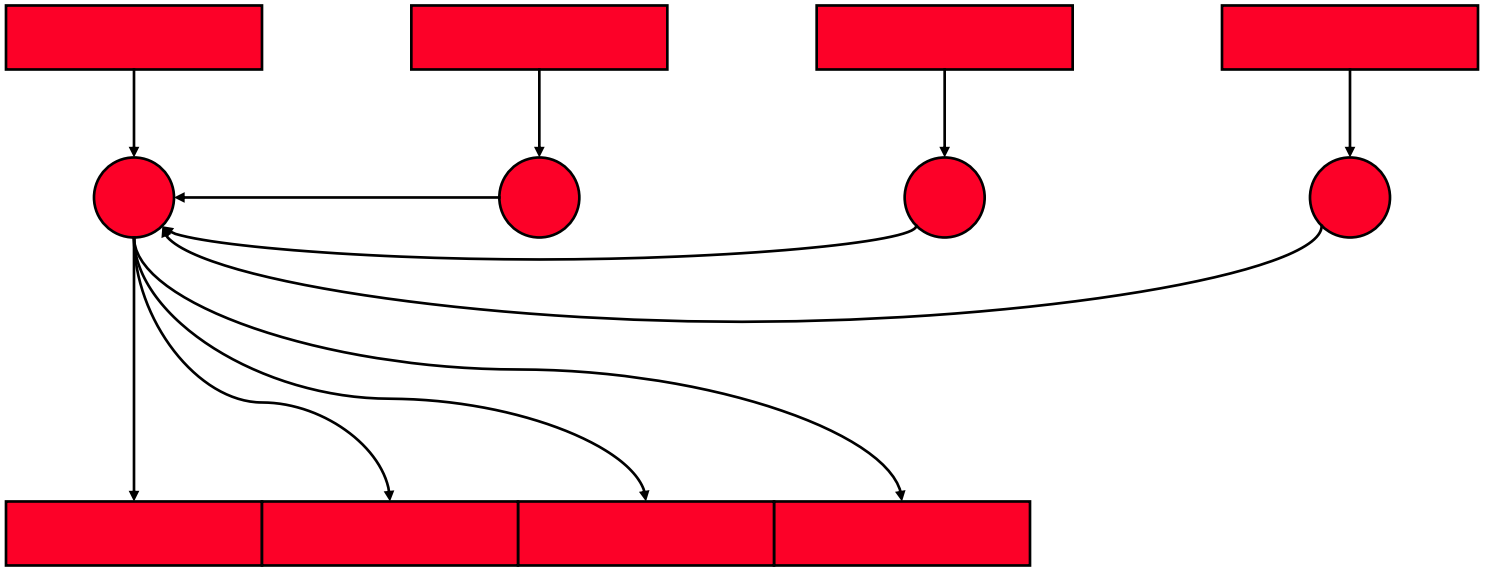
- Getting started with "Hello World"
- Sharing Data
  - Using MPI datatypes to share data
  - Using MPI_Pack to share data
- Sending in a ring (broadcast by ring)
  - Using topologies to find neighbors
- Finding PI using MPI collective operations
- Fairness in message passing
  - Implementing Fairness using Waitsome
- A Parallel Data Structure
  - Using nonblocking operations
  - Shifting data around
  - Exchanging data with MPI_Sendrecv

MPI-IO

From Thakur-MPI-IO

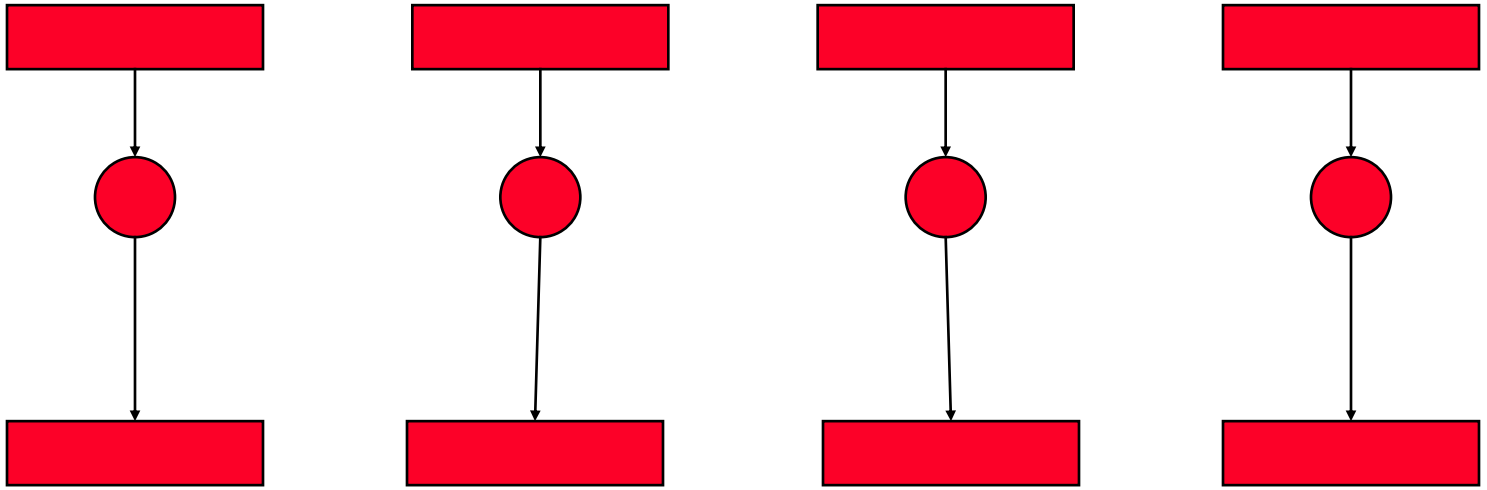
Common Ways of Doing I/O in Parallel Programs

- Sequential I/O:
  - All processes send data to rank 0, and 0 writes it to the file



Another Way

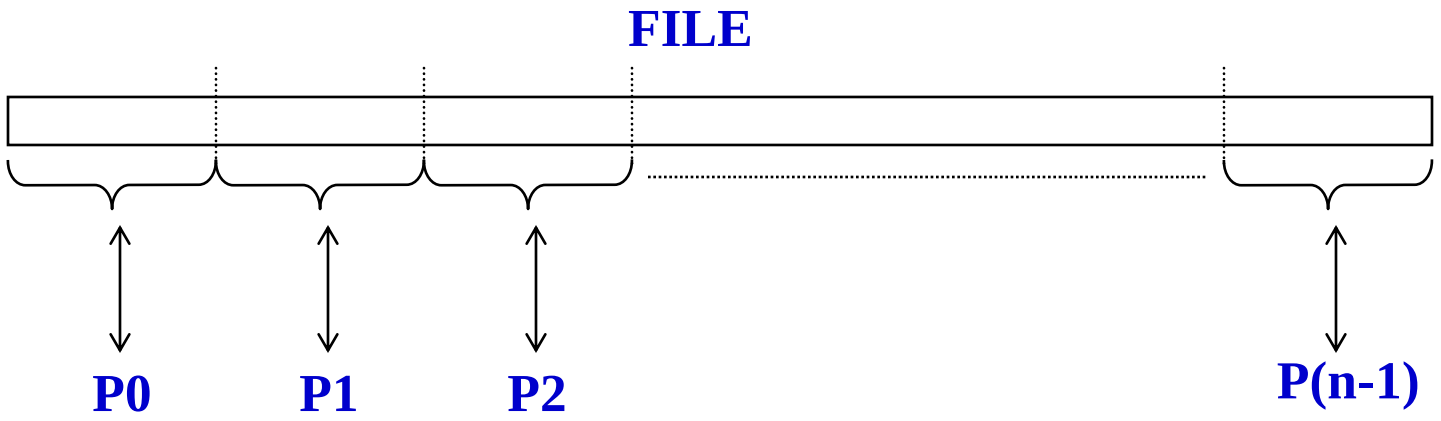
- Each process writes to a separate file



- **Pros:**
  - **parallelism, high performance**
- **Cons:**
  - **lots of small files to manage**
  - **difficult to read back data from different number of processes**

What is Parallel I/O?

- Multiple processes of a parallel program accessing data (reading or writing) from a *common* file



Why Parallel I/O?

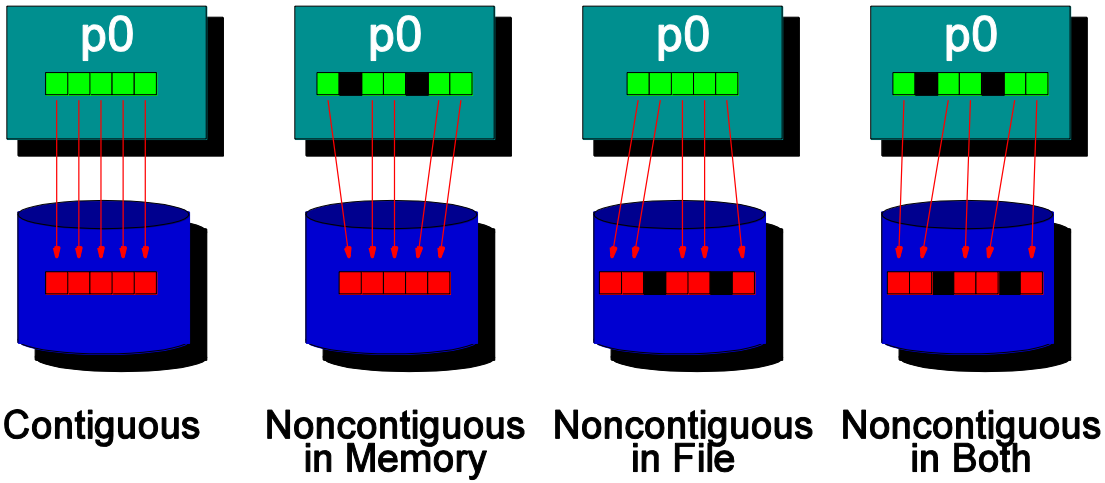
- Non-parallel I/O is simple but
  - Poor performance (single process writes to one file) or
  - Awkward and not interoperable with other tools (each process writes a separate file)

- Parallel I/O
  - Provides high performance
  - Can provide a single file that can be used with other tools (such as visualization programs)

MPI I/O standard

- All calls start with MPI_File_
  - open, read, write, seek, close
- Asynchronous modifier “i”: iread etc.
- Absolute position modifier “_at”: read_at
- Collective modifier “_all”: read_all etc
- split collective modifier “_begin” “_end”
- shared file pointer modifier: “_shared”
- MPI_Type to create derived data types

Noncontiguous I/O



- Contiguous I/O moves data from a single block in memory into a single region of storage
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O

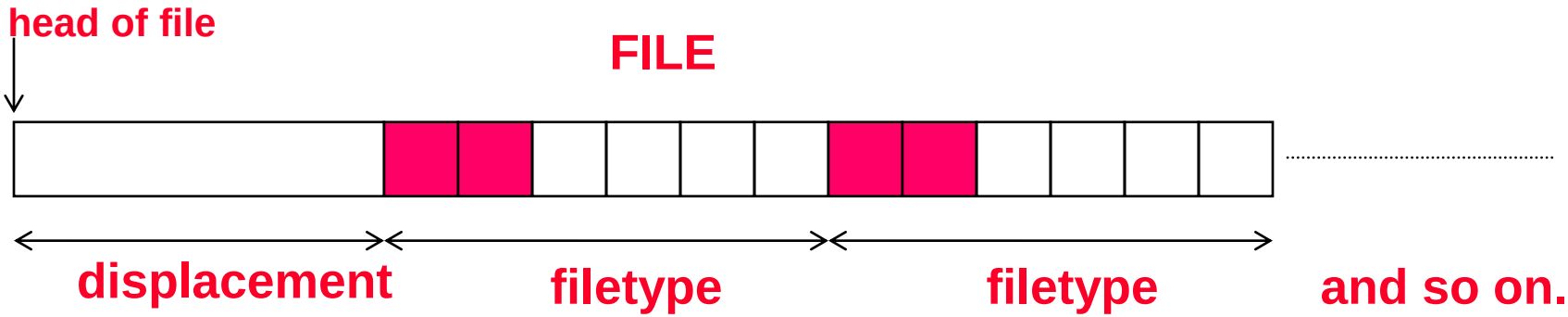
Fileview

- Displacement, etype and filetype creates a fileview
- **fileview allows simultaneous writing/reading of noncontiguous interleaved data by multiple processes**
- MPI_File_set_view call
- **each process has a different fileview of a single file**

A Simple Noncontiguous File View Example

etype = MPI_INT

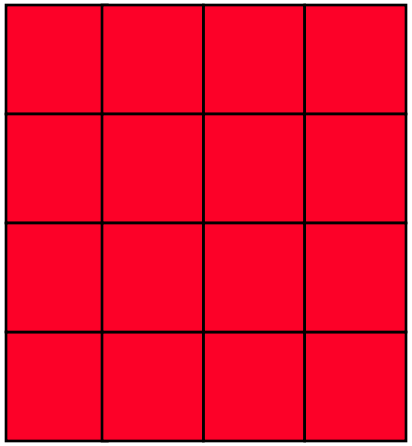
filetype = two MPI_INTs followed by
a gap of four MPI_INTs




```
MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);
```

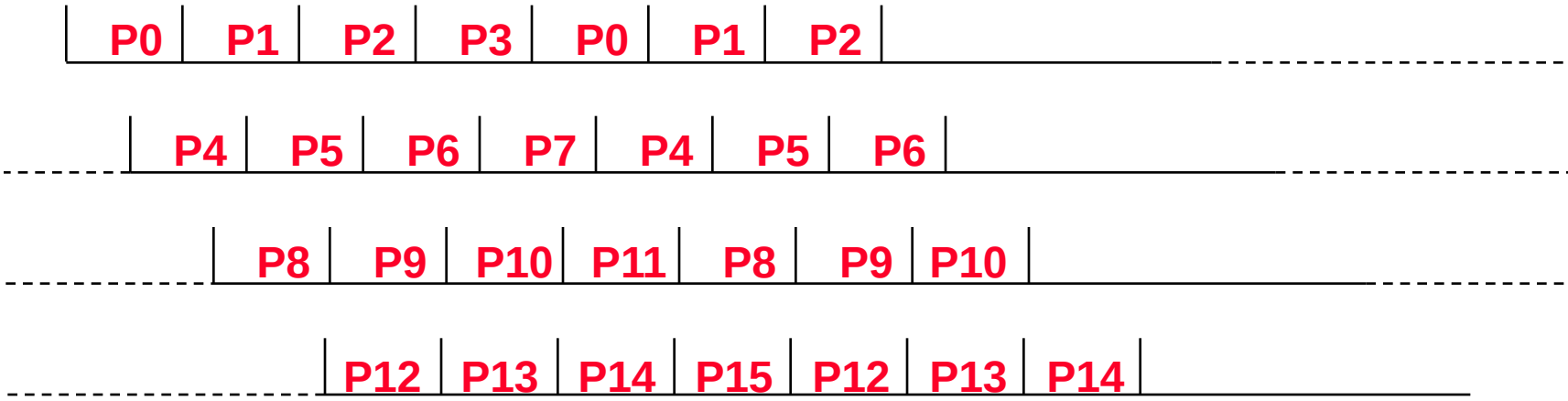
Example: Distributed Array Access

Large array distributed among 16 processes



Each square represents a subarray in the memory of a single process

Access Pattern in the file



Level-0 Access (C)

- Each process makes one independent read request for each row in the local array (as in Unix)

```
MPI_File_open(..., file, ..., &fh)
for (i=0; i<n_local_rows; i++) {
    MPI_File_seek(fh, ...);
    MPI_File_read(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

Level-1 Access (C)

- Similar to level 0, but each process uses collective I/O functions

```
 MPI_File_open(MPI_COMM_WORLD, file, ...,
&fh);
for (i=0; i<n_local_rows; i++) {
    MPI_File_seek(fh, ...);
    MPI_File_read_all(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

Level-2 Access (C)

- Each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent I/O functions

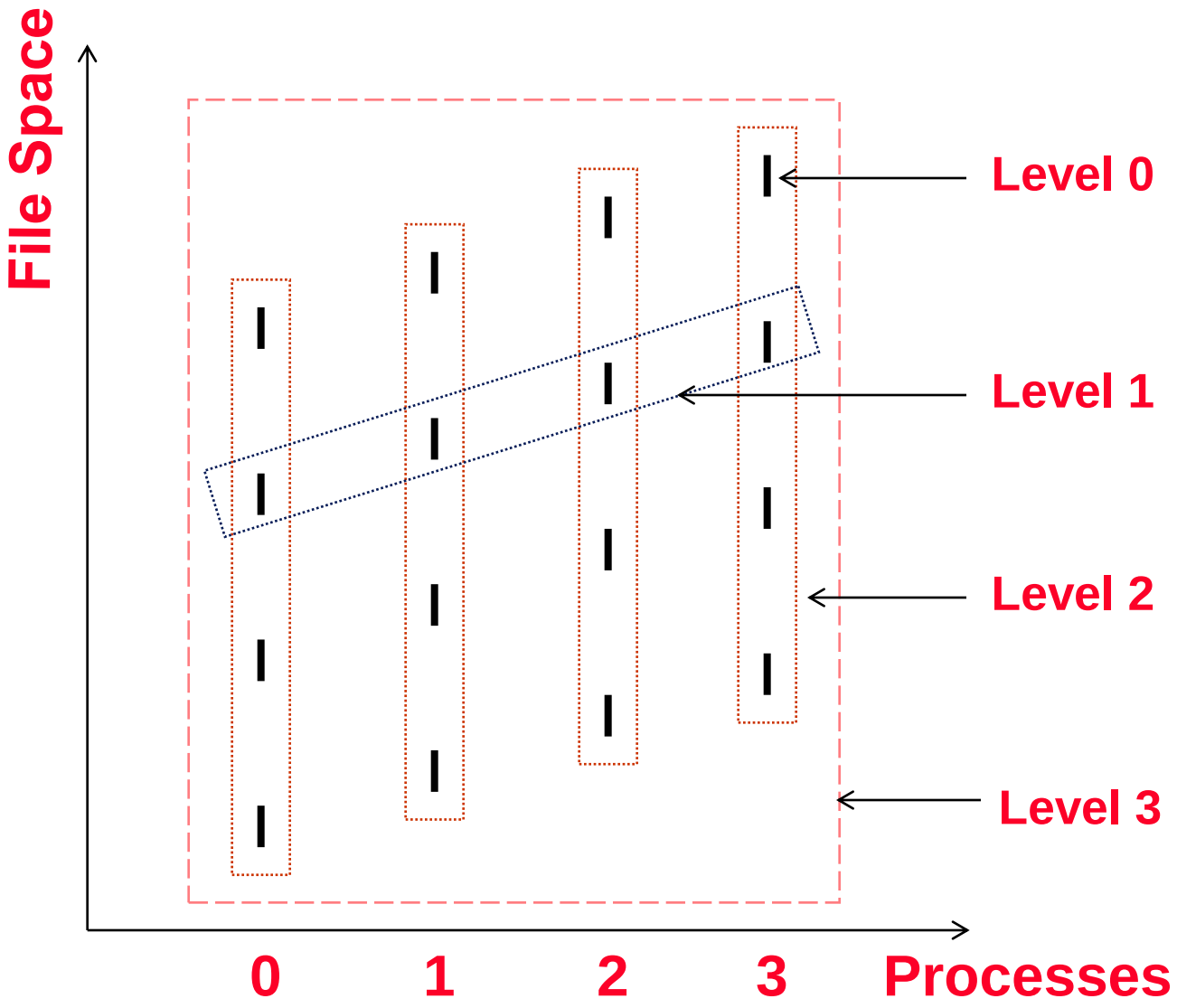
```
 MPI_Type_create_subarray(..., &subarray,
...);
MPI_Type_commit(&subarray);
 MPI_File_open(..., file, ..., &fh);
 MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read(fh, A, ...);
MPI_File_close(&fh);
```

Level-3 Access (C)

- Similar to level 2, except that each process uses collective I/O functions

```
MPI_Type_create_subarray(..., &subarray,
...);
MPI_Type_commit(&subarray);
 MPI_File_open(MPI_COMM_WORLD, file, ...,
&fh);
 MPI_File_set_view(fh, ..., subarray,
...);
MPI_File_read_all(fh, A, ...);
MPI_File_close(&fh);
```

The Four Levels of Access



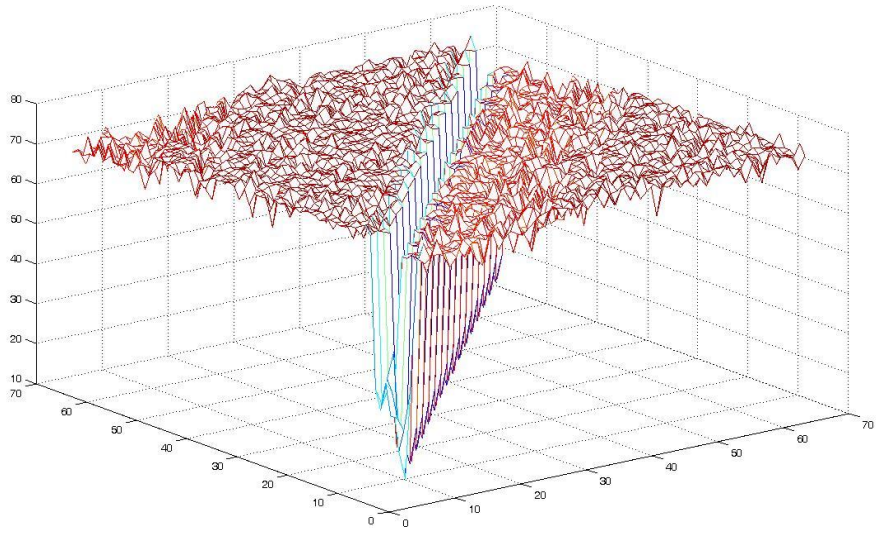
General Guidelines for Achieving High I/O Performance

- Buy sufficient I/O hardware for the machine
- Use fast file systems, not NFS-mounted home directories
- Do not perform I/O from one process only
- To achieve good performance:
  - Write as large chunks as possible
  - use derived data types to read/write non-contiguous data
  - Use collective I/O calls
  - use non-blocking I/O calls
  - provide hints through “info” parameter
  - provide complete picture of the total I/O operation on the whole file by all the processes

Others might be helpful in MPI programming

- MPI 进程拓扑MPI_Cart_*

- 进程映射
  - LSB_HOSTS获取资源
  - MPI命令行参数-machinefile设定进程向资源的映射



System view

Application view