L07 OpenMP

1. Introduction to OpenMP

OpenMP: Prevailing **shared memory** programming approach

·共享内存

·声明串行程序的一部分代码为并行执行（增量式并行化 Incremental parallelization）

·基于编译器 Compiler based

·fork-join model 最开始只有一个主线程 master thread，并行时 fork 出从线程 worker threads，线程间同步使用隐式 implicit barrier，并行结束后只有主线程继续执行，从线程空闲等待下一次 fork，最后 join 到一起

·Pragmas->预处理指令，OpenMP 是**基于指令的共享内存 API**

```
#pragma omp parallel
#pragma omp 指令名 \
    下一行指令
```

·检查处理器是否支持

```
#ifdef _OPENMP
#include <omp.h>
#endif
```

·shared 共享变量，private 私有变量，默认为 shared，loop index 为 private

```
#pragma omp parallel \
 shared (bigdata) \
 private (tid)
{
    //并行代码
}
```

·int omp_get_num_threads(void)返回当前并行线程总数（不一定根据用户定义线程数，根据处理器配置，一般一个处理器一个）

·int omp_get_thread_num(void) 返回某一线程编号，主线程为 0，最后一个线程编号为 omp_get_num_threads()-1

2. The trapezoidal rule

·#pragma omp parallel **num_threads**(thread_count) 用 thread_count 个线程并行执行

·

```
sum=0;
 #pragma omp parallel for reduction(+:sum)
   for(i=0; i<100; i++) {
       sum+=array[i];
   }
```

Parallel for -> 结构必须为一个 for 循环

+:sum -> 要规约的为加法，结果加入 sum

每一线程有一个私有 sum（执行时），最终加入共享 sum。

规约操作（需满足结合律）初始值：

```
+0 -0 *1
Bitwise &0 |0 ^0
Logical &1 |0
```

3. Parallel for

自动分配，自动同步，所有线程都同步到#pragma omp parallel for 处，所有线程都运行完 for 后，主线程继续运行，其余等待。

迭代次数需要提前知道，对于 loop index 的加减只能在条件中，循环过程中不能有 break，不能存在 loop carried dependency。

```
double factor = 1.0;
double sum = 0.0;
#pragma omp parallel for num_threads(thread_count) \
    default(none) reduction(+:sum) private(i,factor) shared(n)
 for(i=0;i<n;i++) {
    factor=(i%2==0)?1.0:-1.0;
    sum+=factor/(2*i+1);
}
pi=4.0*sum;
```

- default(none)后所有变量要明确给出私有 or 共享，reduction(+:sum)已隐含 sum 变量为共享，每个线程有自己私有 sum。

4. Schedule Clause

```
#pragma omp parallel for num_threads(thread_count) \
    reduction(+:sum) schedule(static,1)
```

schedule(<TYPE>,<CHUNKSIZE>)

- static 表示运行时不变，1 表示每次给线程划分几次迭代

TYPE：

- static：每个线程划分 chunk 次迭代
- dynamic：有调度队列，动态分配。哪个线程先运行完再分配。
- guided：chunksize 每次变为剩下大小一半，只有最后一次迭代才能小于 chunksize
- RUNTIME：动态设置环境变量 OMP_SCHEDULE，不用设置 size
- AUTO：系统自己计算评估
- NOWAIT：取消 loop 结束后的默认同步
- ORDERED：按串行顺序执行循环迭代

5. Synchronization

- 显式同步：
  - critical：一段代码
  - atomic：一条语句 x<op> = <expression> 只对 x 保护（如 x+=y++，x++）
  - barrier
- 条件并行：
  - 只有满足该条件才并行

```
#pragma omp parallel if(n>threshold) \
    shared(n,x,y) private(i) {
    #pragma omp for
    for(i=0;i>n;i++)
        x[i]+=y[i];
    }
```

- 其余语句只需要一个线程运行

```
#pragma omp single {
    //code
}
```

• 主线程运行

```
#pragma omp master {
    //code
}
```

• 每个线程只运行其中一个 section

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            //code1
        #pragma omp section
            //code2
    }//end sections
}//end parallel
```

L06 MPI

1. Introduction to Message Passing

- 没有共享存储空间，逻辑上共享的数据被划分到本地进程中。进程通过明确的 send/receive 对进行通信
- MPI 为一个函数库 message passing library，基于单程序多数据 SPMD（不同进程根据条件运行不同代码段），无数据竞争，但可能出现通信问题（接收端悬挂）
- 所有通信、同步都要调用子程序 subroutine calls，没有共享变量，除了对 MPI 的调用以外程序再单个处理器上运行。
  - 子程序用于：
    - ①communication
      - 成对/点对点通信：一个消息从特定发送进程发送到一个特定接收进程
      - 涉及多处理器的集合：
        - 移动数据：Broadcast，Scatter/gather
        - 计算&移动：Reduce，Allreduce
    - ②synchronization
      - barrier
      - 没有锁，因为无共享变量
    - ③查询
      - 进程数量、自己进程编号、有无消息等待

2. Hello World

```c
#include <stdio.h>
#include <string.h>
#include <mpi.h>
//每个程序都发送消息给 0 号进程并打印
const int MAX_STRING = 100;
int main(void){
    char greeting[MAX_STRING];
    int comm_sz;
    int my_rank;
    int q;
    MPI_Init(NULL,NULL);  //初始化
    MPI_Comm_size(MPI_COMM_WORLD,&comm_sz); //获得一共有多少个进程处理任务
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank); //返回当前进程进程号（initial 时已分配好）
    if(my_rank != 0){
        sprintf(greeting,"Greatings from process %d of %d!",my_rank,comm_sz);
        MPI_Send(greeting,strlen(greeting)+1,MPI_CHAR,0,0,MPI_COMM_WORLD);
            //发送 greeting 信号给 0 号进程
    }else{
        printf("Greetings from process %d of %d\n",my_rank,comm_sz);
        for(q = 1;q < comm_sz; q++){
            MPI_Recv(greeting,MAX_STRING,MPI_CHAR,q,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                //send 和 recv 次数一一对应
            printf("%s\n",greeting);
        }
    }
    MPI_Finalize();
    return 0;
```

编译：

```
mpicc -g -Wall -o mpi_hello mpi_hello.c
```

运行：

```
mpiexec -n <进程数> ./mpi_hello
```

进程数即用几个进程运行并行

MPI 函数：

①int MPI_Init(
    int*   argc_p /*in/out*/,
    char***   argv_p /*in/out*/);
    分配存储空间，决定 rank，定义一个通信子(初始化为最大进程数 MPI_COMM_WORLD)

②int MPI_Finalize(void)
    释放资源，MPI 部分已完成
    MPI 调用在 init 和 finalize 之间完成

③int MPI_Comm_size(
    MPI_Comm   comm   /*in*/,
    int*   comm_sz_p   /*out*/);
    通信子类型 MPI_Comm，包含哪些进程、及其编号
    通过&comm_sz_p 获得进程数量

④int MPI_Comm_rank(
    MPI_Comm   comm   /*in*/,
    int*   my_rank_p   /*out*/);
    通过&my_rank_p 获得当前进程 rank

⑤int MPI_Send(
    void*   msg_buf_p   /*in，一个指向一块内存的指针，将该空间内的数据发送出*/,
    int   msg_size   /*in，数据 amount*/,
    MPI_Datatype   msg_type   /*in，数据类型*/,
    int   dest   /*in*/,
    int   tag   /*in，区分打印/计算的数据*/,
    MPI_Comm   communicator   /*in，声明通信域*/);
    配对，哪一个 send 对应哪一个 receive
    MPI_Datatype：不同于 C 语言的数据类型，要使用 MPI 数据类型

    MPI_Send(buffer, size, datatype, dest, tag, comm);
    - 消息 buffer 用 buffer，size，datatype 描述
    - 目标进程号用 dest 声明
    - 函数返回不代表数据已经被接收，函数返回时只表明数据已经被送到系统中，缓冲区可以被重新使用

⑥int MPI_Recv(
    void*   msg_buf_p   /*out，指针指向接收端 buffer*/,
    int   buf_size   /*in*/,
    MPI_Datatype   buf_type   /*in*/,
    int   source   /*in，从哪一个进程接收的[通配符 MPI_ANY_SOURCE 任何一个发送进程都可匹配]*/,
    int   tag   /*in*/,
    MPI_Comm   communicator   /*in*/,
    MPI_Status*   status_p   /*out*/);

    MPI_Recv(buffer, size, datatype, source, tag, comm, status);
    - 等待，直到从系统中收到一个匹配的（包括 source 和 tag）消息，并且可以使用缓冲区
    - source 是由 comm 指定的通信子中的 rank，或者 MPI_ANY_SOURCE
    - tag 是一个要匹配的标签，或者 MPI_ANY_TAG
    - 收到少于 size 的 datatype 是可以的，但收到更多则 error。
    - status 包含进一步的信息（如消息的大小），MPI_STATUS_IGNORE

匹配：

进程 q 调用 MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag, send_comm)

过程 r 调用 MPI_Recv(recv_buf_p, recv_buf_sz, rece_type, src, recv_tag, recv_comm, &status)

send 和 receive 匹配 if：

recv_comm=send_comm

recv_tag=send_tag

dest=r && src=q

可以接收信息 if：

recv_type=send_type

recv_buf_sz>=send_buf_sz

组织进程：

- 进程分为组 groups
- 每个消息接收&发送都在同一上下文 context 中
- 一个通信子 communicator 由一个 group 和 context 构成
- 一个进程用其所在通信子相关组中的 rank 标识
- 默认通信子 MPI_COMM_WORLD

数据类型 MPI_Datatype：

- 一个连续的 MPI 数据类型数组
- 一个数据类型的串联块
- 一个由数据类型块组成的索引数组
- 一个派生数据类型结构

Table 3.1 Some Predefined MPI Datatypes

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Tag：

- 消息在发送时附带一个用户定义的 int tag，以帮助接收进程识别消息
- 消息可以通过指定一个特定的 tag 在接收端进行筛选，或者通过指定 MPI_ANY_TAG 作为接收中的 tag 而不进行筛选。

Send&Recv 语义：

- 发送进程可以 buffer/block 消息
- MPI 发送的确切行为是由 MPI 实现决定的
  - 典型的实现方式有一个默认的 cutoff 消息大小
    - 如果消息的大小小于截止值，它将被 buffer。
    - 如果消息的大小大于截止值，MPI_Send 将 block。
- MPI_Recv 总是 block，直到收到一个匹配的消息。
- MPI 要求消息是非超越式 non-overtaking（recv 方接收同一 send 方的消息顺序与发送顺序相同，不同 send 方则可能不同）

3. The trapezoidal rule
4. Collective communication

①int MPI_Reduce(

```
void*          input_data_p      /*in*/,
void*          output_data_p     /*out*/,
int            count             /*in*/,
MPI_Datatype   datatype          /*in*/,
MPI_Op         operator          /*in,操作子*/,
int            dest_process      /*in*/,
MPI_Comm       comm              /*in*/);
```

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

- 存储全局操作的结果到指定的进程

• 所有进程都提供相同长度的输入 buffer，其元素的类型与 root 的输出 buffer 相同

• 有目的进程，发送结果给目的进程

• 通信子中的所有进程必须调用相同的集体函数

• 每个进程传递给 MPI 集体通信的参数必须兼容

• output_data_p 参数只在 dest_process 上使用。 然而，所有的进程仍然需要传入一个与 output_data_p 相对应的实际参数，即使它只是 NULL。

• 点对点通信是在 tag 和通信子的基础上匹配的。集体通信不使用 tag。它们只根据通信子和它们被调用的顺序来匹配。（每个进程调用的第一个/第二个...）

②int MPI_Allreduce(

```
    void*          input_data_p      /*in*/,
    void*          output_data_p     /*out*/,
    int            count             /*in*/,
    MPI_Datatype   datatype          /*in*/,
    MPI_Op         operator          /*in*/,
    MPI_Comm       comm              /*in*/);
```

• MPI_Allreduce 的行为与 MPI_Reduce 相同，只是其结果出现在所有进程的接收缓冲区中。

• 可逆树形结构，分发结果给每个进程

③int MPI_Bcast(

```
    void*          data_p            /*in/out*/,
    int            count             /*in*/,
    MPI_Datatype   datatype          /*in*/,
    int            source_proc       /*in，发送进程号*/,
    MPI_Comm       comm              /*in*/);
```

• 只允许 0 号进程读 buffer 后分发，其他进程要执行接收

Data distributions

$$\begin{aligned}\mathbf{x}+\mathbf{y} &= (x_0,x_1,\ldots,x_{n-1})+(y_0,y_1,\ldots,y_{n-1})\\ &= (x_0+y_0,x_1+y_1,\ldots,x_{n-1}+y_{n-1})\\ &= (z_0,z_1,\ldots,z_{n-1})\\ &= \mathbf{z}\end{aligned}$$

n=12, Comm_sz=3

Table 3.4 Different Partitions of a 12-Component Vector among Three Processes

Process	Components: Block	Components: Cyclic	Components: Block-Cyclic Blocksize = 2
0	0 1 2 3	0 3 6 9	0 1 6 7
1	4 5 6 7	1 4 7 10	2 3 8 9
2	8 9 10 11	2 5 8 11	4 5 10 11

④int MPI_Scatter(

```
    void*          send_buf_p        /*in，要发送的数据*/,
    int            send_count        /*in，发送给每个进程的数量*/,
    MPI_Datatype   send_type         /*in，数据类型，要与 recv_type 相同*/,
    void*          recv_buf_p
                                     /*out，每个进程接收的数据，必须大于 send_count*send_type*/,
    int            recv_count        /*in*/,
    MPI_Datatype   recv_type         /*in*/,
    int            src_proc          /*in，发送者*/,
    MPI_Comm       comm              /*in*/);
```

• 分发数据，平均分，分成 n 个相等的片段，第 i 个片段被发送给第 i 个进程

⑤int MPI_Scatterv(

```
    const void*    sendbuf,       /*可以为不连续的，根据 displs[]偏移量找到*/
    const int      sendcounts[],  /*发送给每个进程的数据数量可变*/
    const int      displs[],      /*发送给进程 j 的数据被放置在根进程的 sendbuf 中，从偏移量 displs[j]元素开始*/
    MPI_Datatype   sendtype,
    void*          recvbuf,
    int            recvcount,
    MPI_Datatype   recvtype,
    int            root,
    MPI_Comm       comm);
```

• 发送给每个进程的数据数量可变

⑥int MPI_Gather(

```
    void*          send_buf_p     /*in*/,
    int            send_count     /*in*/,
    MPI_Datatype   send_type      /*in*/,
    void*          recv_buf_p     /*out，连续存放的地址空间*/,
    int            recv_count     /*in，从每个进程接收的 msg 数量*/,
    MPI_Datatype   recv_type      /*in*/,
    int            dest_proc      /*in*/,
    MPI_Comm       comm           /*in*/);
```

• 收集，相对于 scatter，放入目的线程的 recv_buffer

• All processes execute:

MPI_Send(sendbuf, sendcount, sendtype, root , ...),

The root execute:

MPI_Recv(recvbuf+i*recvcount*extent(recvtype), recvcount, recvtype, i,...),

⑦int MPI_Gatherv(

```
    const void*    sendbuf,
    int            sendcount,
    MPI_Datatype   sendtype,
    void*          recvbuf,
    const int      recvcounts[], /*非负整数数组（长度为 group 大小），包含了从每个进程收到的元素的数量（仅在 root 有意义）*/
    const int      displs[], /*从进程 j 收到的数据被放入根进程的 recvbuf，从偏移量 displs[j] 元素开始（就 recvtype 而言）*/
    MPI_Datatype   recvtype,
    int            root,
    MPI_Comm       comm);
```

⑧int MPI_Allgather(

```
    void*          send_buf_p     /*in*/,
    int            send_count     /*in*/,
    MPI_Datatype   send_type      /*in*/,
    void*          recv_buf_p     /*out*/,
    int            recv_count     /*in*/,
    MPI_Datatype   recv_type      /*in*/,
    MPI_Comm       comm           /*in*/);
```

- MPI_Allgather 可以被认为是 MPI_Gather，但是所有进程都会收到结果，而不仅仅是根节点。
- 从第 j 个进程发送的数据块被每个进程接收，并放在缓冲区 recv_buf_p 的第 j 个块中。

⑨int MPI_Allgatherv(
```
const void*    sendbuf,
int            sendcount,
MPI_Datatype   sendtype,
void*          recvbuf,
const int      recvcounts[],
const int      displs[],
MPI_Datatype   recvtype,
MPI_Comm       comm)
```

- MPI_ALLGATHERV 可以被认为是 MPI_GATHERV，但所有进程都会收到结果，而不仅仅是根节点。
- 从第 j 个进程发送的数据块被每个进程接收，并放在缓冲区 recvbuf 的第 j 个块中。
- 这些块不一定都是相同大小的。
- 所有线程都要调用集合通信相关函数

5. MPI derived datatypes 派生数据类型

- count 参数：可用于将连续的数组元素分组到一个信息中。
- 派生数据类型：可以用来表示内存中任何数据项的集合，方法是同时存储这些项的类型和它们在内存中的相对位置。
- MPI Pack/Unpack。将给定数据类型的数据打包到连续的内存中并进行通信。接收方解包。
- 一个派生数据类型由两部分组成：
  一个基本数据类型的序列 + 一个整数（byte）位移的序列（距离开始地址的偏移）
- 类型映射：一对{($type_0$, $disp_0$),...,($type_{n-1}$,$disp_{n-1}$)}的序列，其中 $type_i$ 是基本类型，$disp_i$ 是偏移量。
- 数据类型的类型签名：基本数据类型的序列（忽略位移）。即一共包含多少种数据类型。

①int MPI_Type_contiguous(
```
int            count,
MPI_Datatype   oldtype,
MPI_Datatype   *newtype);
```

- oldtype 复制 count 次形成 newtype
- oldtype 的 type map 为{(double,0), (char,8)},偏移量 16
  MPI_Type_contiguous(3,oldtype,newtype)
  ->newtype 的 type map:
  {(double,0),(char,8), (double,16),(char,24), (double,32),(char,40)}

②int MPI_Type_vector(
```
int            count,
int            blocklength,
int            stride,
MPI_Datatype   oldtype,
MPI_Datatype   *newtype);
```

- MPI_Type_vector 是一个更通用的构造函数，允许将一个数据类型复制到由等距块组成的位置。
- oldtype 的 type map 为{(double,0), (char,8)},偏移量 16
  MPI_Type_vector(2,3,4,oldtype,newtype)
  - 2->newtype 两个 block
  - 3->每个 block 三个 oldtype
  - 4->4*偏移量 16=64，第二个 block 起始位置 64

  ->newtype 的 type map:

①{(double,0),(char,8), (double,16),(char,24), (double,32),(char,40),
②(double,64),(char,72), (double,80),(char,88), (double,96),(char,104)}

③int MPI_Type_indexed(
```
int            count,     /*多少 block*/
const int      array_of_blocklengths[],     /*每个 block 对应一个 length（不同）*/
const int      array_of_displacements[],    /*每个 block 的偏移量多少个 extent*/
MPI_Datatype   oldtype,
MPI_Datatype   *newtype);
```

•MPI_Type_indexed 允许将一个旧的数据类型复制到一连串的块中（每个块都是旧的数据类型的串联），其中每个块可以包含不同数量的拷贝，并有不同的位移。

•oldtype 的 type map 为{(double,0), (char,8)},偏移量 16

B=(3,1), D=(4,0)

MPI_Type_indexed(2,B,D,oldtype,newtype);

{(double,64),(char,72), (double,80),(char,88), (double,96),(char,104),
(double,0),(char,8)}

④int MPI_Type_create_struct(
```
int            count,
int            array_of_blocklengths[],
MPI_Aint       array_of_displacements[],
MPI_Datatype   array_of_types[],
MPI_Datatype*  new_type_p);
```

•使用 MPI_Type_create_struct 来建立一个派生数据类型，它由具有不同基本类型的单个元素组成。

•type1 的 type map 为{(double,0), (char,8)},偏移量 16

B=(2,1,3), D=(0,16,26), T=(MPI_FLOAT, type1, MPI_CHAR)

MPI_Type_create_struct(3,B,D,T,newtype)

{(float,0),(float,4), (double,16),(char,24), (char,26),(char,27),(char,28)}

2*float　　1*type1　　3*char

address 和 size 相关函数：

①MPI_Bottom：表示地址空间的初始地址为零。

②MPI_Get_address：找到内存中某个位置的地址。

```
int MPI_Get_address(
    void       location_p     /*in*/,
    MPI_Aint*  address_p      /*out，返回*/);
```

③MPI_Aint_add：产生一个新的 MPI_Aint 值，相当于 base 和 disp 参数之和。

```
MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp);
```

④MPI_Aint_diff：产生一个新的 MPI_Aint 值，相当于 addr1 和 addr2 参数之间的差。(add2-add1)

```
MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2);
```

⑤MPI_Type_size：将 size 的值设置为与数据类型相关的类型签名中的条目的总大小，单位为字节。

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

获取偏移量：

```
MPI_Aint a_addr, b_addr, n_addr;
MPI_Get_address(&a, &a_addr);
MPI_Get_address(&b, &b_addr);
MPI_Get_address(&n, &n_addr);
array_of_displacements[0]=0;
array_of_displacements[1]=MPI_Aint_diff(&b_addr,&a_addr); //b-a
array_of_displacements[2]=MPI_Aint_diff(&n_addr,&a_addr); //n-a
```

a	b	n

commit 和 free 相关函数：

①MPI_Type_commit：一个数据类型对象在可以用于通信之前必须被提交。提交操作提交的是数据类型，也就是通信缓冲区的形式描述，而不是该缓冲区的内容。创建新类型后要 commit。

```
int MPI_Type commit(MPI_Datatype* new_mpi_t_p /*in/out*/);
```

②MPI_Type_free：标记与 datatype 相关联的数据类型对象为去分配，并将 datatype 设置为 MPI_DATATYPE_NULL。即释放构造数据类型所用空间。

```
int MPI_Type_free(MPI_Datatype *datatype);
```

pack 和 unpack 相关函数：

①int MPI_Pack(

```
    void*          data_to_be_packed     /*in*/,
    int            to_be_packed_count    /*in*/,
    MPI_Datatype   datatype              /*in*/,
    void*          contig_buf            /*out*/,
    int            contig_buf_size       /*in*/,
    int*           position_p            /*in/out*/,
    MPI_Comm       comm                  /*in*/);
```

- 将数据打包到一个连续内存的缓冲区中。
- 将由 data_to_be_packed、to_be_paced_count、datatype 指定的发送缓冲区中的信息打包到由 contig_buf 和 contig_buf_size 指定的缓冲区空间。输出缓冲区是一个包含 contig_buf_size 字节的连续存储区域，从地址 contig_buf 开始。

②int MPI_Unpack(

```
    void*          contig_buf            /*in*/,
    int            contig_buf_size       /*in*/,
    int*           position_p            /*in/out*/,
    void*          unpacked_data         /*out*/,
    int            unpack_count          /*in*/,
    MPI_Datatype   datatype              /*in*/,
    MPI_Comm       comm                  /*in*/)
```

- 从连续内存的缓冲区中解压数据。
- 从 contig_buf 和 contig_buf_size 指定的缓冲区空间将消息解压到 unpacked_data，unpack_count，datatype 指定的接收缓冲区。输出缓冲区可以是 MPI_RECV 中允许的任何通信缓冲区。

内通信子 Inter-Communicators：

内部通信子将组和上下文的概念结合起来。

MPI 通信操作引用通信子来确定点对点或集体操作的范围和 "通信宇宙"。

每个通信器都包含一个有效参与者的组；这个组总是包括本地进程。 **send 方和 recv 方在一组进程内。**

对于集体通信，内部通信器指定了参与集体操作的进程的集合（以及它们的顺序，如果重要的话）。

一旦调用 MPI_INIT 或 MPI_INIT_THREAD，就会定义本地进程在初始化后可以与之通信的所有进程的初始内部通信器 MPI_COMM_WORLD（包括它自己）。

内通信是不同组中的进程之间的点对点通信。

对内通信和内通信子属性的总结：

点对点通信和集体通信的语法对外部通信和内部通信都是一样的。同一通信器可用于发送和接收操作。

无论是发送还是接收，目标进程都是通过它在远程组中的等级来寻址。

使用内部通信子的通信保证不会与使用不同通信子的任何通信相冲突。

一个通信器要么提供内部通信，要么提供外部通信，绝不会同时提供。

L05-Pthreads

1. Introduction to POSIX Threads

(1) 共享内存编程 vs 分布式内存编程

共享内存：主线程/进程 fork 出执行计算的线程，线程执行工作，线程间通过共享内存通信和同步。

分布式内存：多个进程在多个系统，进程执行工作，进程间通过消息传递通信和同步。

(2) 共享内存编程

动态线程：当任务请求来时调用 fork()，无则不调用，主线程等待。优化资源使用，但动态生成&销毁造成性能下降。

静态线程：分配号工作后直接生成所有线程，所有计算线程完成后再 terminate，主线程 cleanup。性能更好，但是可能浪费系统资源。

(3) POSIX Threads

POSIX: Portable Operating System Interface for UNIX——用于 UNIX 的便携式操作系统接口

2. Hello World

(1) 创建线程 Fork

①int pthread_create(

```
pthread_t*,              //指针，指向 pthread_t 对象，存储线程信息的地址
const pthread_attr_t*,   //线程属性，默认为 NULL
void* (*) (void*),       //函数->线程要运行的函数
void*);                  //传递给该函数的参数
```

```
errcode = pthread_create(&thread_id, &thread_attribute, &thread_fun, &fun_arg);
    //创建的每一个线程都要运行 thread_fun
    //如创建失败则 errcode 非 0
```

```
int main() {
    pthread_t threads[16];
    int tn;
    //先 fork 后 join
    for(tn=0; tn<16; tn++) {
        pthread_create(&threads[tn], NULL, ParFun, NULL);
        //每个线程 ParFun 并行执行
    }
    for(tn=0; tn<16; tn++) {
        pthread_join(threads[tn], NULL); //进程结束前把线程 join 到主进程，释放空间
    }
    return 0;
}
```

编译时 gcc ... -lpthread ...

(2) 共享数据

- main 函数之外声明的变量为共享
- 如果传递指针，则在堆中分配的对象可能为共享
- 栈中的变量是私有的

```
char* message = "Hello World!\n";
```

```
        pthread_create(&thread1, NULL, (void*)&print_fun, (void*)message);
```

```
#include <sdtio.h>
#include <stdlib.h>
#include <pthread.h>

int thread_count; //共享全局变量

void *Hello(void* rank) {
    long my_rank = (long) rank; //分配线程号，不能用 int 因为是 64bit
    printf("Hello from thread %ld of %d\n", my_rank, thread_count);
    return NULL;
}

int main(int argc, char* argv[]) {
    long thread;
    pthread_t* thread_handles; //pthread 数据结构

    thread_count = strtol(argv[1], NULL, 10);
        //从命令行输入获得线程数量，10 为十进制的长整型转换
    thread_handles = malloc(thread_count*sizeof(pthread_t));
        //为每个线程分配空间

    for(thread=0; thread<thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL, Hello, (void*)thread);
            //(void*)thread 指针类型 64bit，强制转换

    for(thread=0; thread<thread_count; thread++)
        pthread_join(&thread_handles[thread], NULL);

    free(thread_handles); //释放创建线程所用空间
    return 0;
}
```

编译：gcc -g -Wall -o pth_hello pth_hello.c -lpthread

./pth_hello 4(线程数)

3. Martrix vector multiplication
4. Critical Section

临界区：一次只能有一个线程操作，更新一个共享资源。必须用串行方式访问。

(1) busy-waiting 忙等

```
y = Compute(my_rank);
while(flag!=my_rank);
x = x+y; //临界区
flag++; // or flag = (flag+1)%thread_count
```

每个线程有自己私有变量，最终再对私有 sum 进行加和操作，减少 while 循环测试时间。

(2) Pthread Mutexes 互斥锁

互斥锁数据类型：pthread_mutex_t

①int pthread_mutex_init(

```
    pthread_mutex_t*              mutex_p;
    const pthread_mutexattr_t*    attr_p); //默认为 NULL
```

必须初始化。分配空间，初始值可以是 1。

②int pthread_mutex_destroy(pthread_mutex_t* mutex_p);

释放锁空间

③int pthread_mutex_lock(pthread_mutex_t* mutex_p);

线程调用，为获得对临界区的访问权。如获得，mutex 从 1 变 0，即占用状态，其余线程检测/挂起状态交替。

④int pthread_mutex_unlock(pthread_mutex_t* mutex_p);

一个线程运行完临界区的代码，将 mutex+1。

```
pthread_mutex_lock(&mutex);
sum+=my_sum;
pthread_mutex_unlock(&mutex);
```

线程执行顺序随机。

(3) semaphore 信号量

不是 pthreads 一部分，所以需要#include <semaphore.h>。

一个 unsigned int，有两种操作：sem_wait（即 sem--）和 sem_post（即 sem++）。比互斥锁更强大因为其能初始化为任何非负值。

生产者-消费者：内部并行，外部必须先生产再消费，p 生产消息放入消息池，c 再从消息池中取出。

```
①int sem_init(
    sem_t*      semaphore_p    /*out*/,
    int         shared         /*in*/,
    unsigned    initial_val    /*in，信号量初始化的值*/);

②int sem_destroy(sem_t* semaphore_p /*in/out*/);

③int sem_post(sem_t* semaphore_p /*in/out*/);

④int sem_wait(sem_t* semaphore_p /*in/out*/);
```

5. Barrier and Condition Variables 条件变量实现 barrier 同步

barrier：同步线程以确保它们都处于程序中的同一点。在所有线程都到达 barrier 之前，任何线程都不能跨越 barrier。

```
pthread_barrier_t b;
pthread_barrier_init(&b, NULL, 3); //NULL 为默认属性，设置线程数为 3

pthread_barrier_wait(&b); //wait 一个 barrier
```

进行静态初始化：赋值给一个用宏指令创建的初始值

```
PTHREAD_BARRIER_INITIALIZER(3) //3 个线程到了才运行
```

barrier 使用结构：

```
Point in program we want to reach;
Barrier;
if (my_rank==0){
```

```
        printf("All threads reached this point\n");
        fflush(stdout);
    }
```

使用忙等和 mutex 实现一个 barrier：

我们使用一个由 mutex 保护的共享 counter。

当计数器显示每个线程都已经进入过临界区时，线程都可以离开临界区。

```
    /*Barrier*/。
    pthread_mutex_lock(&barrier_mutex);
    counter++; //每个线程到后对 counter+1
    pthread_mutex_unlock(&barrier_mutex);
    while(counter < thread_count); //小于 n，一直忙等
```

使用信号量实现 barrier：

```
    sem_t count_sem; /*Initialize to 1*/ //对 counter 的访问
    sem_t barrier_sem; /*Initialize to 0*/  //阻塞线程的信号量
    void* Thread_work(){
        /*Barrier*/
        sem_wait(&count_sem);
        if(counter == thread_count-1){  //最后一个进入的线程
            counter = 0;
            sem_post(&count_sem);
            for(j=0;j<thread_count-1;j++) //放行阻塞线程
                sem_post(&barrier_sem);
        }
        else{
            counter++;
            sem_post(&count_sem);
            sem_wait(&barrier_sem); //阻塞线程
        }
    }
```

counter 和 count_sem 可以重复使用，barrier_sem 不能被重用！

(4)条件变量 condition variables

条件变量是一个数据对象，它允许一个线程暂停执行，直到某个事件或条件发生。当事件或条件发生时，另一个线程可以向该线程发出信号 "唤醒"。

一个条件变量总是与一个所有线程共享的互斥锁一起使用（阻塞线程）。

A 线程挂起，B 线程发送信号唤醒 A。

①int pthread_cond_signal(pthread_cond_t* cond_var_p /*in/out*/);

只唤醒一个线程

②int pthread_cond_broadcast(pthread_cond_t* cond_var_p /*in/out*/);

唤醒所有线程

③int pthread_cond_wait(

```
    pthread_cond_t*    cond_var_p     /*in/out*/,
    pthread_mutex_t*   mutex_p        /*in/out*/);
```

阻塞，分三步：

①unlock 条件变量互斥锁

②wait for signal（其他线程唤醒）

③其他线程调用 signal/broadcast->lock mutex，恢复到阻塞前的状态

条件变量使用结构：

```
lock mutex;
if condition has occurred
    signal thread(s); //唤醒所有被挂起的线程
else {
    unlock the mutex and block; //当线程被 unblock，互斥锁被重新 lock
}
unlock mutex;
```

用条件变量实现 barrier：

```
/*Shared*/
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
...
void* Thread_work(...) {
    ...
    /*Barrier*/
    pthread_mutex_lock(&mutex);
    counter++;
    if(counter == thread_count) {
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    }
    else {
        while(pthread_cond_wait(&cond_var, &mutex)!=0); //线程挂起
    }
    pthread_mutex_unlock(&mutex);
    ...
}
```

6. Read-Write Locks

多个线程可以同时通过调用 read-lock 函数获得锁，而只有一个线程可以通过调用 write-lock 函数获得锁。
因此，如果任何线程拥有读取的锁，任何想要获得写入的锁的线程都会在调用写锁函数的过程中阻塞。
如果任何线程拥有写入锁，任何想要获得读取或写入锁的线程都会在各自的锁函数中阻塞。

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
//读
...
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
//写
...
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
//写
```

L04-Dependence

1. Data Dependence 数据依赖

• 定义：两个访问同一个存储地址，至少其中一个是写操作。前面程序产生一个值供后面程序使用。保证数据依赖不变，则并行正确。

Pj 读 Ij，写 Oj

Pk 读 Ik，写 Ok

①Ij∩Ok=空集

②Ik∩Oj=空集

③Oj∩Ok=空集（即 Pj,k 无顺序）

满足上述三个条件则程序可并行。

• 四种模式：read after write 先写后读，write after read 先读后写，write after write，read after read（只有这个绝对安全）

• i 和 i’之间存在数据依赖 当且仅当

①两操作之间有写操作

②两操作指向同一变量

③两操作执行有顺序，i 在 i’之前

2. Recording Transformation

• 定义 1：相同输入的两个计算等价当：①产生同一输出 ②输出执行顺序相同

• 定义 2：重排转换 reordering transformation：改变语句执行顺序，不增加 or 删除任何语句的执行。

• 定义 3：重排转换可保持数据依赖当保持 source（源）和 sink（结束）的执行顺序

• 定理 1：任何保留了程序中每一个依赖关系的重排转换都保留了该程序的意义（符合结合律的）。

• why 重排转换？保持代码依赖性。

• 并行化

在同步点之间并行执行的计算有可能被重新排序。 这种重新排序是否安全？ 根据我们的定义，如果它保留了代码中的依赖关系，就是安全的。

• 局部性最优

假设我们想修改一个计算访问内存的顺序，使其更有可能在缓存（数据访问开销减少）中。 这也是一种重排转换，如果它保留了代码中的依赖关系，那么它就是安全的。

• 减少计算

对于还原，我们必须放松这一规则。 对于换元和关联操作来说，重排还原是安全的。

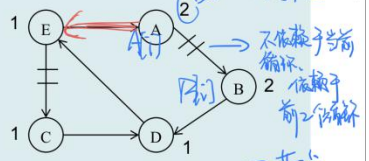
3. Parallelization

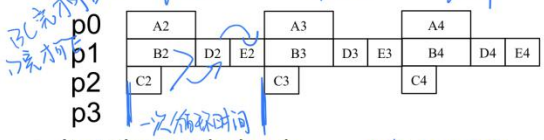
• 流水线：①取址 ②译码，取源操作数 ③运行 ALU 执行 ④保存在内存 ⑤写回

• 数据流图：

Parallelization

```
for (i = 2; i < n; i++) {
    A[i] = E[i] * 2;
    B[i] = A[i-2] * A[i-2];
    C[i] = E[i-2] + 5;
    D[i] = B[i] + C[i];
    E[i] = D[i] + 9;
}
```

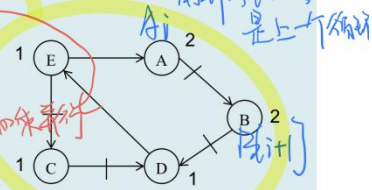


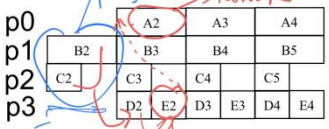


- Iteration period = 4
- Iteration period: The execution latency of an iteration

Parallelization

```
for (i = 2; i < n; i++) {
    A[i] = E[i] * 2;
    B[i+1] = A[i-1] * A[i-1];
    C[i+1] = E[i-1] + 5;
    D[i] = B[i] + C[i];
    E[i] = D[i] + 9;
}
```

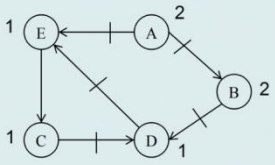


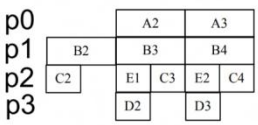


- Iteration period = 2

Parallelization

```
for (i = 2; i < n; i++) {
    A[i] = E[i] * 2;
    B[i+1] = A[i-1] * A[i-1];
    E[i-1] = D[i-1] + 9;
    C[i+1] = E[i-1] + 5;
    D[i] = B[i] + C[i];
}
```





- Iteration period = 2

4. Locality

- 数据划分/数据分配问题

  减少 cache miss 的概率，时间局部性↑，数据重用↑（同一块数据放在一起计算）。

  - 减少内存延迟 memory latency

    内存访问的延迟是指内存请求和完成之间的时间（通常以周期为单位）。

  - 最大化内存带宽 memory bandwidth

    带宽是指在一个时间间隔内可以检索到的有用数据的数量。

  - 管理开销

    执行优化（如复制）的开销应小于预期

- 数据重用&数据局部性

  考虑数据的访问方式

  - 数据重用 reuse（cache hit 即为一种数据重用）

    多次使用相同或相近的数据

    计算中的内在因素

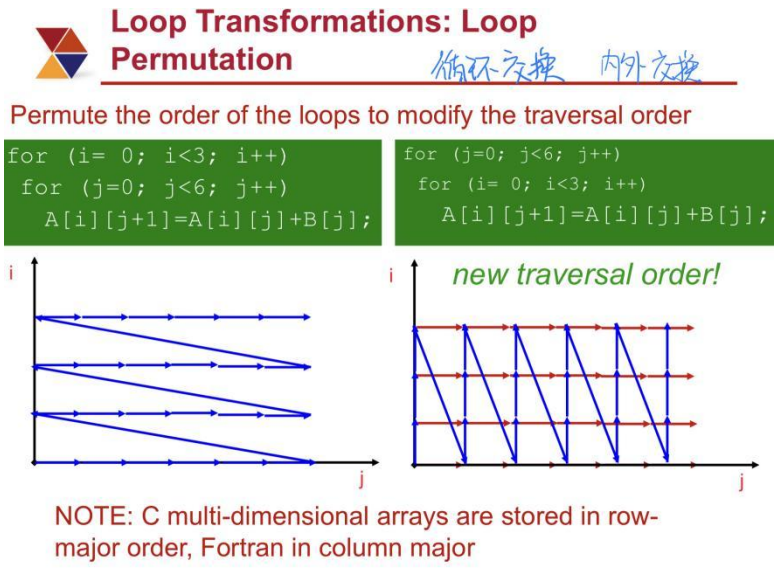
  - 数据局部性 locality

    数据被重复使用并存在于 "快速内存 "中

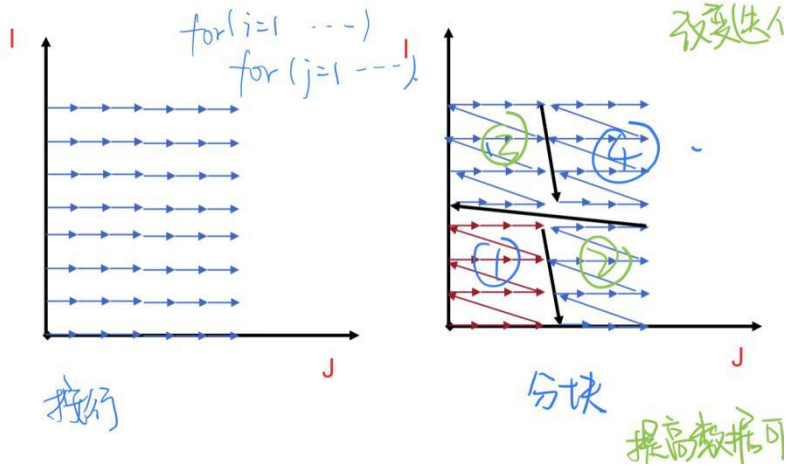
    相同的数据或相同的数据传输

    如果一个计算有重复使用，我们可以做什么来获得局部性？

    适当的数据放置和布局&代码重排转换

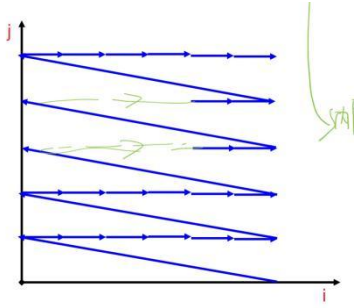


改变迭代顺序，提高数据可重用性。



- Tiling
  - 改变迭代执行顺序：

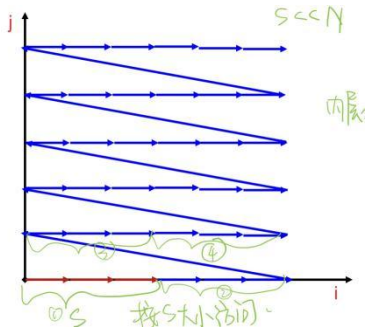
```
for(j=1;j<M;j++)
    for(i=1;i<N;i++)
        D[i]=D[i]+B[j,i]
```



假设一个 cache line 可放 b 个数组元素
内层循环 N/b 次 miss，外层 M
总 cache miss=2M*N/b

  - 分块划分 循环交换 Strip mine

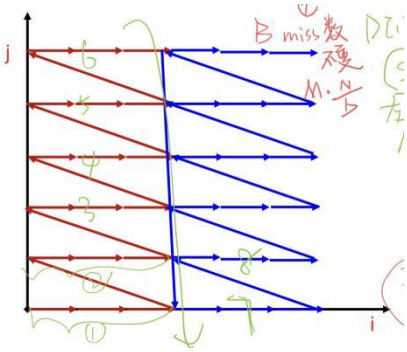
```
for(j=1;j<M;j++)
    for(ii=1;ii<N;ii+=s)
        for(i=ii;i<min(ii+s,N);i++)
            D[i]=D[i]+B[j,i]
```



s<<N，假设 D[s]和 D[1]可存在于同一行 cache line 中
内循环通过分块变为 2 层

  - 重新排列 Permute

```
for(ii=1;ii<N;ii+=s)
    for(j=1;j<M;j++)
        for(i=ii;i<min(ii+s,N);i++)
            D[i]=D[i]+B[j,i]
```



B 的 miss 数不变，仍然为 M*N/b
假设 D[s]和 D[1]可存在于同一行 cache line 中
每一小块是 s/b 次 miss，分块数与 j 无关，为 N/s 个块
所以对于 D 第 i 个元素，miss 为 s/b*N/s = N/b
总 cache miss=(1+M)N/b

- Unroll&Unroll-and-jam
  - Unroll 只是简单地复制循环中的语句，复制的数量称为 unroll 系数。
    只要副本不超过原始循环的迭代次数，它就始终是安全的。（可能需要 "清理"代码）
  - Unroll-and-jam 包括解开**外循环**，并将内循环的副本融合在一起（不一定安全）。
    这是最有效的优化方法之一，但是 unroll 太多的话会有危险。

Original:

```
for (i=0; i<4; i++)
    for (j=0; j<8; j++)
```

Unroll j（j=0&j=1 合并为一个迭代）

```
for (i=0; i<4; i++)
    for (j=0; j<8; j+=2)
        A[i][j] =B[j+1][i];
        A[i][j+1] = B[j+2][i];
```

Unroll-and-jam i

```
for (i= 0; i<4; i+=2)
    for (j=0; j<8; j++)
        A[i][j] = B[j+1][i];
        A[i+1][j] = B[j+1][i+1];
```

Unroll-and-jam i and j loops

```
for (i=0; i<4; i+=2)
    for (j=0; j<8; j+=2) {
        A[i][j]   = B[j+1][i] + B[j+1][i+1];
        A[i+1][j] = B[j+1][i+1] + B[j+1][i+2];
        A[i][j+1] = B[j+2][i] + B[j+2][i+1];
        A[i+1][j+1]= B[j+2][i+1] +
```

B 时间上的重用（放在一次循环）

5. Safety of Recording Transformation

- 数组间的数据依赖

①Loop-Carried dependency

迭代之间，一次迭代中无

```
for(i=2;i<5;i++)
    A[i] = A[i-2]+1;
```

i 之后才可以进行 i+2，i 与 i+2 次迭代之间有依赖。

②Loop-Independent dependency

同一迭代中存在依赖关系

```
for(i=1;i<=3;i++)
    A[i] = B[i]+1;
    C[i] = A[i]*2;
```

只存在 loop-independent 则可以并行。

- 字典序 lexicographic order：迭代 i 的字典序小于 i’
- 距离向量：有依赖的两次迭代间距离（迭代次数差）

```
for(i=1;i<N;i++)
    for(j=1;j<N;j++)
        A[i+1,j+1] = A[i,j]*2.0;
```

I[1,1]->I[2,2]

D = I’-I = [1,1] [外循环,内循环]

D 必须>=0（保持字典序）

正确的循环转换：保持距离向量方向不变。

- 方向向量：相减>0 为<，相减<0 为>，相等为=。只要方向。

距离向量[1,1]，方向向量[<,<]

①[=,=]：可以交换。loop independent，无依赖，可任意交换。

②[=,<]：可以交换。在内层 i 循环存在>0 的。交换后依赖变为[<,=]，转换后未改变字典序，依赖关系仍然在 i 循环中，可以做 permute。

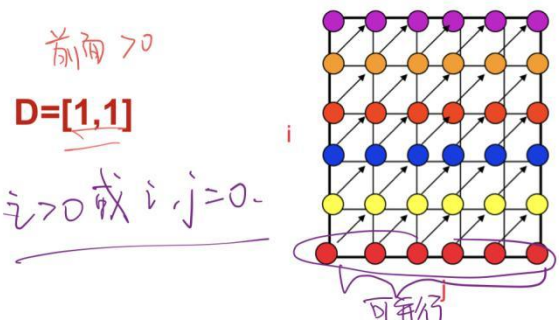
③[<,=]：可以交换。在外层 j 循环存在。内外层可以交换。

④[<,<]：可以交换。交换后仍大于 0。

⑤[<,>]：不可交换。外循环存在依赖，交换后变为[>,<]，前面负的是不存在的（字典序改变）。

[>,*],[=,>]非法。

- 一层 loop 无 loop-carried 则可以并行。
- 定义 1：D=(d1,...,dn)在第 i 层 loop 存在 loop-carried 当 di 是 D 的第一个非 0 值。
- 定义 2：对于 D=(d1,...,dn)，第 j 层 loop 可以并行当(d1,...,dj-1)>0 或(d1,...,dj)=0。



- 定义 3：对于 D，n 层 loop 都可以完全互换当(d1,...,di-1)>0 或 for all k, i<=k<=j, dk>=0。

Cache

①全相联：每个缓存行可以放在 cache 任意位置

②直接映射：每个缓存行在 cache 中有唯一的位置

③n 路组相联：每个缓存行都能放置到 cache 中 n 个不同区域位置的一个

缺失率 miss_rate=miss/access

AMAT(Average Memory Access Time) = hit + miss_rate*miss_penalty

CPI(每条指令所花时间周期) = CPI_0 + MPI(Miss Per Instruction)*miss_penalty（命中时间忽略）

Cache 一致性

①监听 cache 一致性协议：监听总线。广播通知

②基于目录的 cache 一致性协议：目录存储每个高速缓存行的状态，有哪些包含该变量高速缓存行。

页表 TLB

指令级并行：通过让多个处理器部件或者功能单元同时执行指令来提高处理器的性能。

①流水线：将功能单元分阶段安排

②多发射：让多条指令同时启动。多发射处理器通过复制功能单元来同时执行程序中的不同指令。

静态多发射：编译时调度

动态多发射：运行时调度（支持动态多发射的处理器——超标量）

预测 speculation：编译器 or 处理器对一条指令进行猜测，然后在猜测的基础上执行代码。

线程级并行 TLP：同时执行不同线程来提供并行性（粗粒度）

硬件多线程：当前执行的任务被阻塞时，系统能够继续其他有用的工作。

细粒度多线程：处理器在每条指令执行完后切换线程，从而跳过被阻塞的线程

同步多线程 SMT：细粒度多线程的变种，通过允许多个线程同时使用多个功能单元来利用超标量处理器的性能。

粗粒度多线程：只切换需要等待较长时间才能完成操作而被阻塞的线程

Some definitions:

- Token: A group of inputs processed to produce a group of outputs（一轮任务）
- Latency 延迟: Time for one token to pass from start to end（运行一次 token 耗时）
- Throughput 吞吐量: The number of tokens that can be produced per unit time（单位时间内访问的任务量/数据量）

Parallelism increases throughput.（可以运行几轮任务）

任务并行/数据并行

负载均衡：在进程 or 线程之间平均分配任务使每个进程 or 线程获得大致相等的工作量

并行化：将串行程序或者算法转换为并行程序的过程

非确定性：给定的输入可能产生不同输出

并行程序加速比：S=T 并行/T 串行

并行程序效率：E=T 串行/(p 核数*T 并行)

可扩展性：在输入规模也以相应增长率增加的情况下，该程序的效率值一直为 E

加速比：S=T 串行/[(并行百分比*T 串行)/p+(串行百分比*T 串行)]

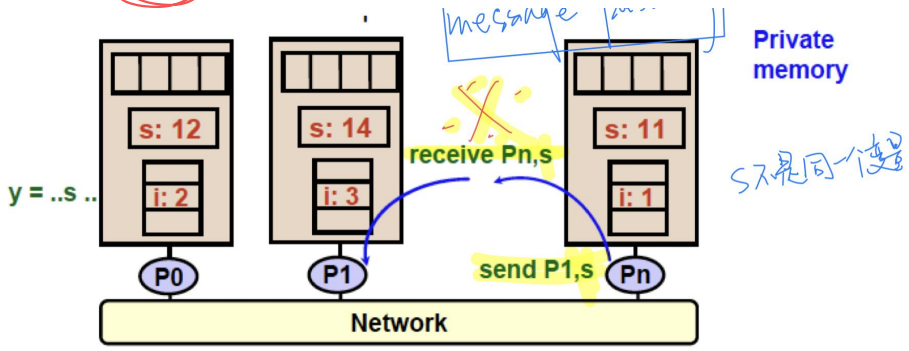
Foster 方法（并行化步骤）：①划分 ②通信 ③凝聚/聚合 ④分配

共享内存架构

- 一个连接到内存系统的自主处理器的集合。支持一个全局地址空间，每个处理器可以访问每个内存位置。
- 一个共享内存程序是一个控制线程的集合。
  - 线程是在程序开始时创建的，也可能是动态的
  - 每个线程都有私有变量，例如本地堆栈变量
- 也有一组共享变量，如静态变量、共享的共用块，或全局堆。
  - 线程通过写入和读取共享变量进行隐式通信。
  - 线程通过使用共享变量实现的锁和 barrier 进行协调。

分布式内存架构

- 一个由互连连接的自主系统的集合。每个系统都有自己独特的地址空间，处理器必须显式通信以共享数据。
- 一个分布式内存程序由命名的进程组成。
- 进程是一个控制线程加上本地地址空间——**没有共享数据**。
- 逻辑上的共享数据被划分给本地进程。
- 进程通过显式的 send/receive 对进行通信。协调隐含在每个通信事件中。



共享内存互连网络：

①总线互联：一组并行的通信线和控制访问总线的硬件组成。核心特征：连接到总线上的设备共享通信线。

②交换互联：使用交换机控制连接设备之间的数据传递。

crossbar 交叉开关矩阵

等分宽度：衡量同时通信的数量。将节点分成 2 组，最坏情况下两部分之间同时通信的链路数目 = 去除最少的链路数从而将节点分成 2 等分。

等分带宽：衡量网络的质量。计算链路的带宽，将链接的带宽相加。

分布式内存互连网络：

(1) 直接互连

①环

②环面：正方形二维环面网格等分宽度=2 √ p

③全相连：每个交换器与每一个其他的交换器直接连接，等分宽度 $p^2/4$

④超立方体：维度为 d 的超立方体有 $p=2^d$ 个节点，等分宽度 p/2

(2) 间接互连

①交叉开关矩阵：p^2 个交换器，等分宽度 p

②omega 网络：$2p\log_2(p)$个交换器，等分宽度 p/2

延迟：从发送源开始传送数据到目的地开始接收数据之间的时间。

带宽：目的地在开始接受数据后接收数据的速度。

一个互连网络延迟 l 秒，带宽 b 字节每秒，传输一个 n 字节的消息

传输时间=l+n/b