Parallel Programming with Processes, Threads and Message Passing

TDDE35

Christoph Kessler

PELAB / IDA
Linköping University
Sweden

Outline

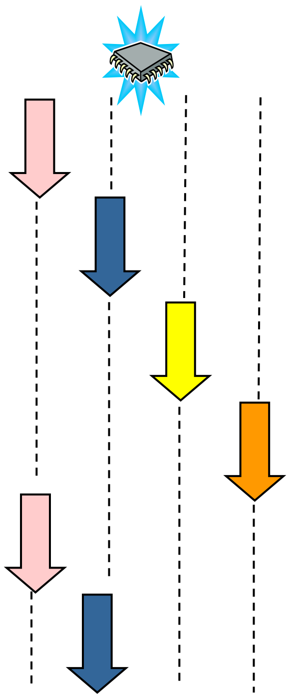
Lecture 2a: Parallel programming with threads

- Shared Memory programming model
- Revisiting processes, threads, synchronization
- Pthreads
- OpenMP (very shortly)

Lecture 2b: Parallel programming with message passing

- Distributed Memory programming model
- MPI introduction

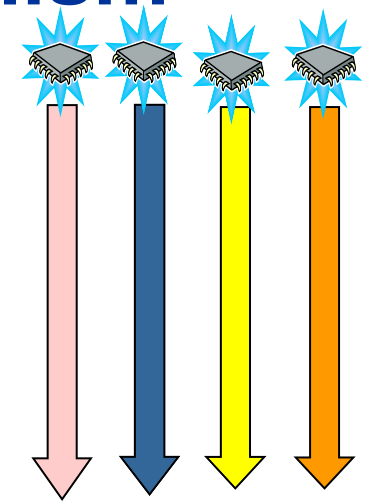
Concurrency vs. Parallelism



Concurrent computing

1 or few CPUs

Quasi-simultaneous execution



Parallel computing

Many CPUs

Simultaneous execution of many / all threads of the *same application*

Common issues:
- threads/processes for overlapping execution
- synchronization, communication
- resource contention, races, deadlocks

Goals of concurrent execution:
- Increase CPU utilization
- Increase responsitivity of a system
- Support multiple users

Central issues: Scheduling, priorities, …

Goals of parallel execution:
- Speedup of 1 application (large problem)

Central issues: Parallel algorithms and data structures, Mapping, Load balancing…

Parallel Programming Models

Parallel Programming Model

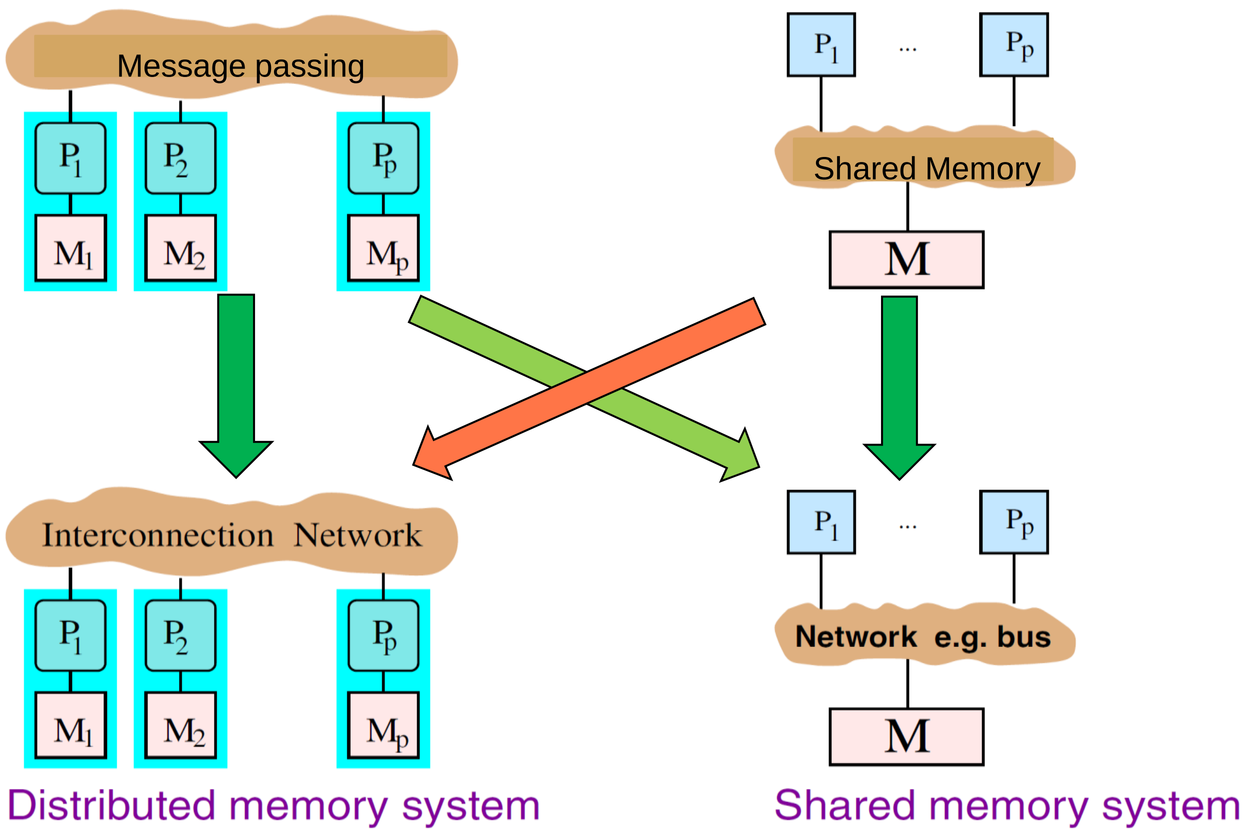
- System-software-enabled **programmer's view** of the underlying hardware
- Abstracts from details of the underlying architecture, e.g. network topology
- Focuses on **a few characteristic properties**, e.g. memory model

→ **Portability** of algorithms/programs across a family of parallel architectures

Programmer's view of the underlying system (Lang. constructs, API, …)
→ **Programming model**

Mapping(s) performed by programming toolchain (compiler, runtime system, library, OS, …)

Underlying parallel computer **architecture**

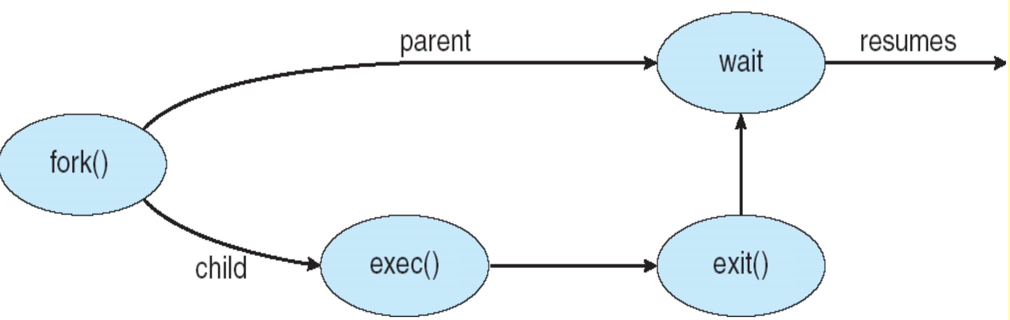


Processes

(Refresher from TDDB68)

Example: Process Creation in UNIX

- **fork** system call
  - creates new child process
- **exec** system call
  - used after a **fork** to replace the process' memory space with a new program
- **wait** system call
  - by parent, suspends parent execution until child process has terminated



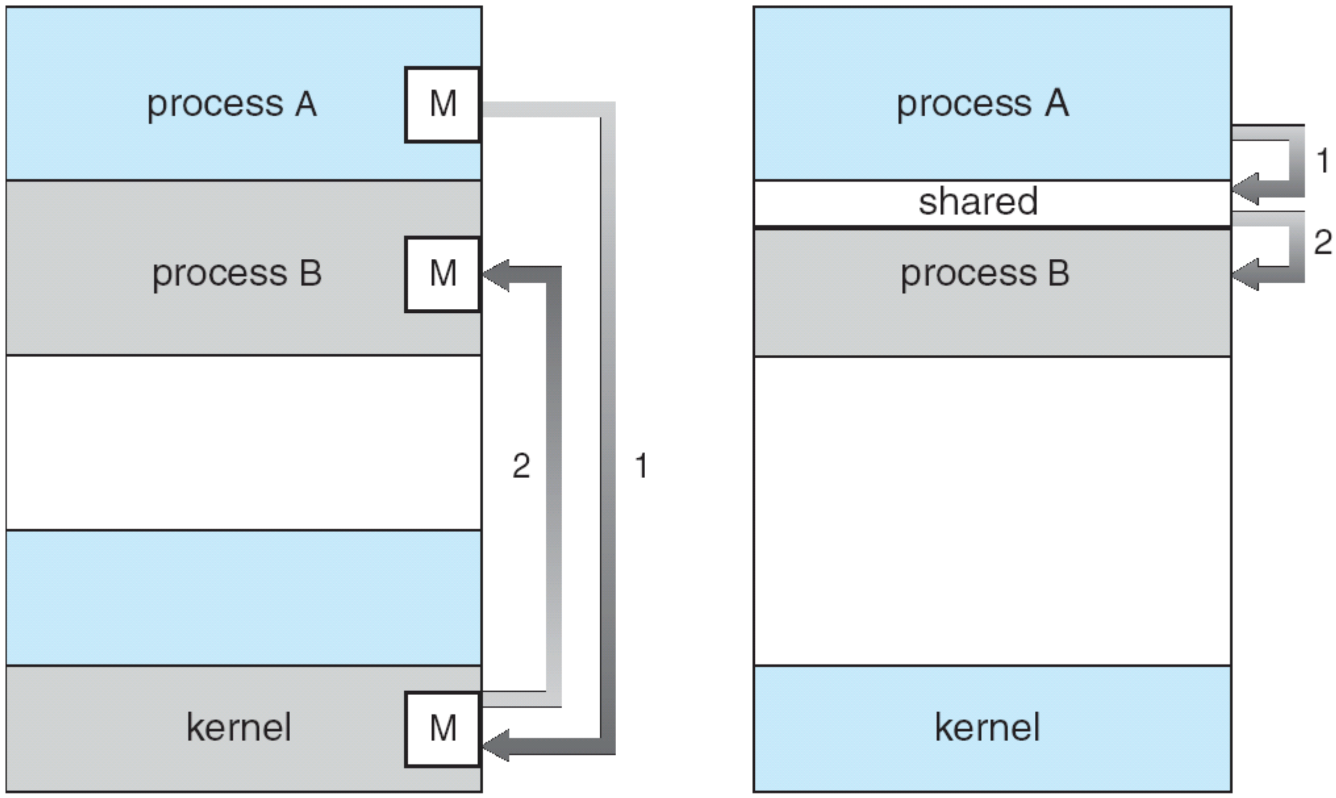
C program forking a separate process

```
int main()
{
     Pid_t  ret;
     /* fork another process: */
     ret = fork();
     if (ret < 0)  {  /* error occurred */
           fprintf ( stderr, "Fork Failed“ );
           exit(-1);
     }
     else if (ret == 0)  {  /* child process */
           execlp ( "/bin/ls", "ls", NULL );
     }
     else  { /*parent process: ret=childPID *
           /* will wait for child to complete: *
           wait (NULL);
           printf ("Child Complete");
           exit(0);
     }
}
```

Parallel programming with processes

- Processes can create new processes that execute concurrently with the parent process
- OS scheduler – also for single-core CPUs
- Different processes share nothing by default
  - Inter-process communication via OS only,
    via shared memory (write/read)
    or message passing (send/recv)
- **Threads** are a more light-weight alternative for programming shared-memory applications
  - Sharing memory (except local stack) by default
  - Lower overhead for creation and scheduling/dispatch
    - E.g. Solaris: creation 30x, switching 5x faster

IPC Models – Realization by OS



(a) IPC via **Message Passing**

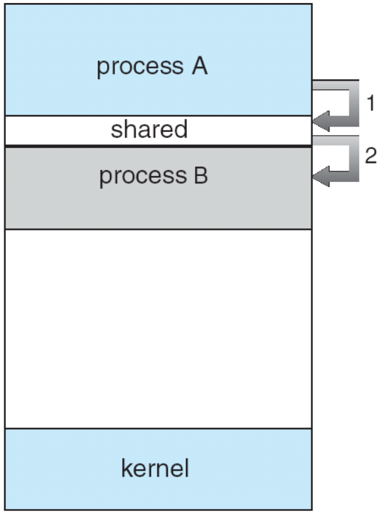
(b) IPC via **Shared Memory**

Syscalls: send, recv

Syscalls: shmget, shmat, then load / store

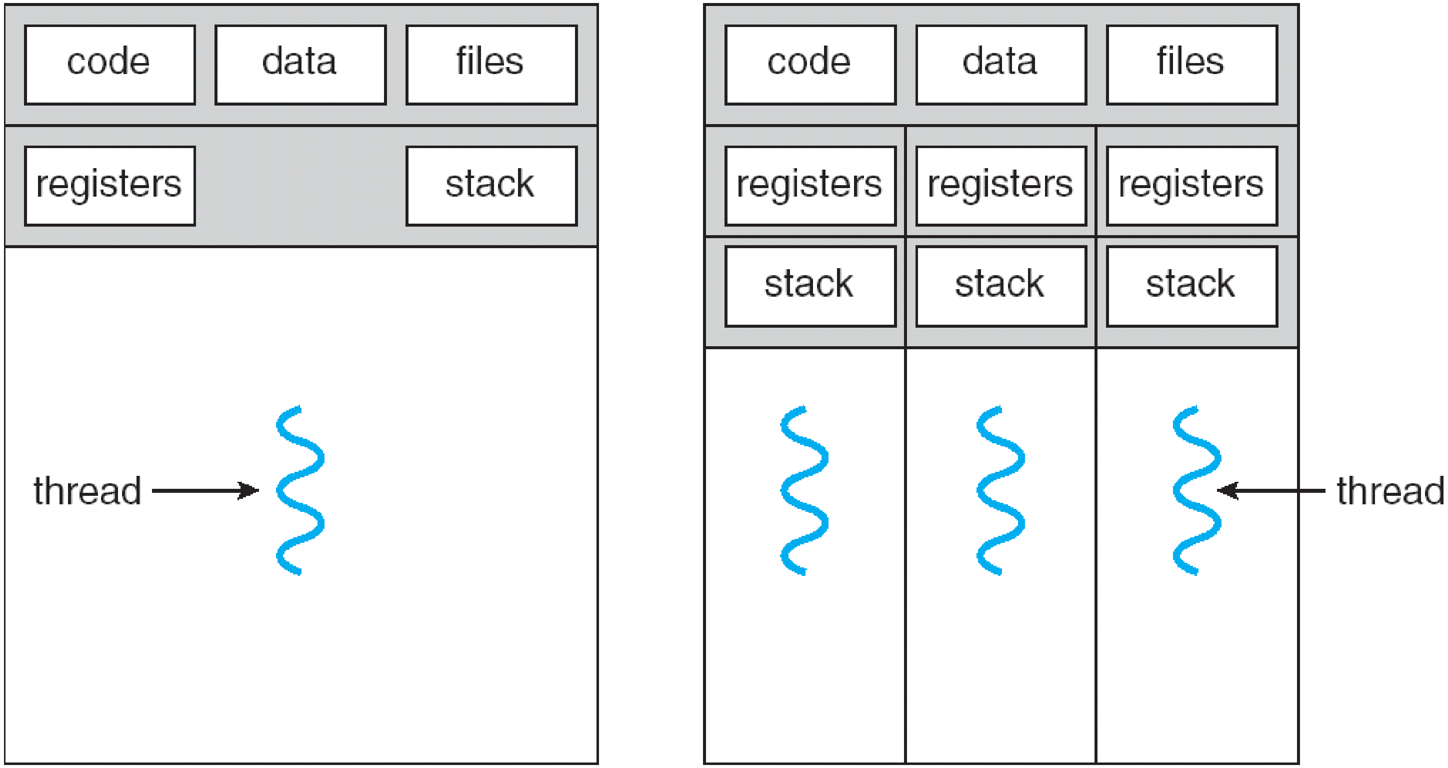
Example: POSIX Shared Memory API

- `#include <sys/shm.h>`
  `#include <sys/stat.h>`
- Let OS create a shared memory segment (system call):
  - int segment_id = **shmget** (IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
- Attach the segment to the executing process (system call):
  - void *shmemptr = **shmat** (segment_id, NULL, 0);
- Now access it:
  - strcpy ((char *)shmemptr, ”Hello world”);   // Example: copy a string into it
  - …
- Detach it from executing process
  when no longer accessed:
  - **shmdt** (shmemptr);
- Let OS delete it when no longer used:
  - **shmctl** (segment_id, IPC_RMID, NULL);



Threads

Single- and Multithreaded Processes



single-threaded process

multithreaded process

A **thread** is a basic unit of CPU utilization:

- Thread ID, program counter, register set, stack.

A process may have one or several threads.

Benefits of Multithreading

- Responsiveness
  - Interactive application can continue even when part of it is blocked
- Resource Sharing
  - Threads of a process share its memory by default.
- Economy
  - Light-weight
  - Creation, management, context switching for threads is much faster than for processes
- **Utilization of Multiprocessor Architectures**
  - Convenient (but low-level) shared memory programming

POSIX Threads (Pthreads)

- A POSIX standard (IEEE 1003.1c) API for thread programming in C
  - start and terminate threads
  - coordinate threads
  - regulate access to shared data structures
- API specifies behavior, not implementation, of the thread library
- C interface, e.g.
  - **int** pthread_create (pthread_t *thread, **const** pthread_attr_t *attr, **void** *(*start_routine)(**void***), **void** *arg);
- Note: as a library, rely on underlying OS and hardware!
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Starting a Thread (1)

- Thread is started with function

```
int pthread_create ( pthread_t *thread,
                     const  pthread_attr_t *attr,
                     void *(*func)(void*),
                     void *arg);
```

  - Called func must have parameter and ret values void*
    - Exception: first thread is started with main()
  - Thread terminates when called function terminates, or by   pthread_exit (**void** *retval)
  - Threads started one by one
  - Threads represented by data structure of type pthread_t

Starting a Thread (2)

- Example:

```
#include <pthread.h>

int main ( int argc, char *argv[] )
{
 int *ptr;
 pthread_t thr;

 pthread_create( &thr,
                 NULL,
                 foo,
                 (void*)ptr  );
 …
 pthread_join( &thr, NULL );
 return 0;
}
```

```
void *foo ( void *vp )
{
   int  i  =  (int) vp;;
   …
}


// alternative
 //  – pass a parameter block:

void *foo ( void *vp )
{
   Userdefinedstructtype *ptr;
   ptr=(Userdefinedstructtype*)vp;
  …
}
```

Access to Shared Data (0)

- Globally defined variables are globally shared and visible to all threads.
- Locally defined variables are visible to the thread executing the function.
- But all data in shared memory publish an address of data: all threads could access…
- Take care: typically no protection between thread data – thread1 (foo1) could even write to thread2's (foo2) stack frame

- **Example 0: Parallel incrementing**

```
int a[N];   // shared,  assume P | N
pthread_t  thr[P];

int main( void )
{
   int t;
   for (t=0; t<P; t++)
     pthread_create(&(thr[t]), NULL,
                          incr, a + t*N/P );
   for (t=0; t<P; t++)
     pthread_join( thr[t], NULL );
    …
}
void *incr ( void *myptr_a )
{   int i;
   for (i=0; i<N/P; i++)
      ((int*)myptr_a[i])++;     }
```

Access to Shared Data (1)

- Globally defined variables are globally shared and visible to all threads.
- Locally defined variables are visible to the thread executing the function.
- But all data in shared memory publish an address of data: all threads could access…
- Take care: typically no protection between thread data – thread1 (foo1) could even write to thread2's (foo2) stack frame

- **Example 1**

```
int *globalptr = NULL;    // shared ptr

void *foo1 ( void *ptr1 )
{
   int i = 15;
   globalptr = &i;  // ??? dangerous!
     // if foo1 terminates, foo2 writes
     // somewhere, unless globalptr
     // value is reset to NULL manually
   …
}

void *foo2 ( void *ptr2 )
{
   if (globalptr) *globalptr = 17;
  …
}
```

Access to Shared Data (2)

- Globally defined variables are globally shared and visible to all threads
- Locally defined variables are visible to the thread executing the function
- But all data in shared memory publish an address of data: all threads could access…
- Take care: typically no protection between thread data – thread1 could even write to thread2's stack frame

- **Example 2**

```
int *globalptr = NULL;    // shared ptr

void *foo1 ( void *ptr1 )
{
   int i = 15;
   globalptr =(int*)malloc(sizeof(int));
   // safe, but possibly memory leak;
   // OK if garbage collection ok


}

void *foo2 ( void *ptr2 )
{
   if (globalptr) *globalptr = 17;
  …
}
```

Coordinating Shared Access (3)

What if several threads need to write a shared variable?

- If they simply write: ok if write order does not matter
- If they read and write: encapsulate (critical section, monitor) and protect e.g. by mutual exclusion using mutex locks)
- **Example: Access to a taskpool**
  - Maintain shared list of tasks to be performed
  - If a thread is idle, it gets a task and performs it

```
// each thread:
while (! workdone)
{
   task = gettask( Pooldescr );
   performtask ( task );
}

// may be called concurrently:

Tasktype gettask ( Pool p )
{
  // begin critical section

  task = p.queue [ p.index ];
  p.index++;

  // end critical section

   return task;
}
```

Race Conditions lead to Nondeterminism

- Example: p.index++
- could be implemented in machine code as

```
39: register1 = p.index              // load
40: register1 = register1 + 1        // add
41: p.index = register1              // store
```

Not atomic!

- Consider this execution interleaving, with "index = 5" initially:

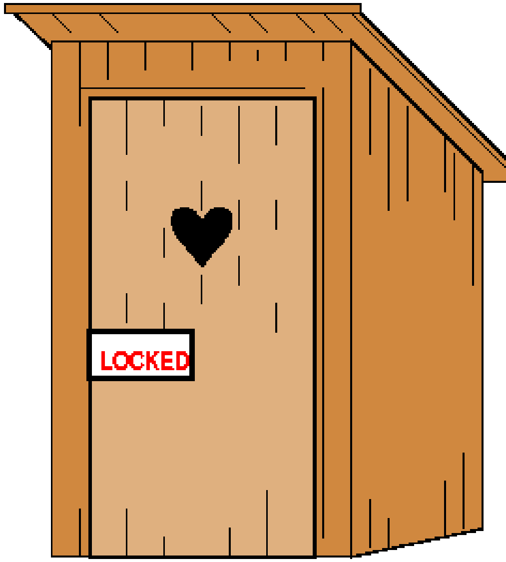
```
39: thread1 executes register1 = p.index          { T1.register1 = 5 }
39: thread2 executes register1 = p.index          { T2.register1 = 5 }
40: thread1 executes register1 = register1 + 1    { T1.register1 = 6 }
40: thread2 executes register1 = register1 + 1    { T2.register1 = 6 }
41: thread1 executes p.index = register1          { p.index = 6 }
41: thread2 executes p.index = register1          { p.index = 6 }
```

- Compare to a different interleaving,
  e.g., 39,40,41, 39,40,41…

→ Result depends on relative speed of the accessing threads
(race condition)

Critical Section

- **Critical Section**: A set of instructions, operating on shared data or resources, that should be executed by a single thread at a time without interruption
  - **Atomicity** of execution
  - **Mutual exclusion**: At most one process should be allowed to operate inside at any time
  - **Consistency**: inconsistent intermediate states of shared data not visible to other processes outside
- May consist of different program parts for different threads
  - that access the same shared data



- General structure, with structured control flow:

  ...

  Entry of critical section C

  … critical section C: operation on shared data

  Exit of critical section C

Coordinating Shared Access (4)

```
pthread_mutex_t  mutex;    // global variable - shared
    …
 // in main:
    pthread_mutex_init( &mutex, NULL );
    …

 // in gettask:
    …
    pthread_mutex_lock( &mutex );
    task = p.queue [p.index];
    p.index++;
    pthread_mutex_unlock( &mutex );
     …
```

Often implemented using test_and_set or other atomic instruction where available

Hardware Support for Synchronization

- Most systems provide hardware support for protecting critical sections
- Uniprocessors – could ***disable interrupts***
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this are not broadly scalable
- Modern machines provide special ***atomic instructions***
  - **TestAndSet**: test memory word <u>and</u> set value atomically
    - Atomic = non-interruptable
    - If multiple TestAndSet instructions are executed *simultaneously* (each on a different CPU in a multiprocessor), then they take effect sequentially in some arbitrary order.
  - **AtomicSwap**: swap contents of two memory words atomically
  - **CompareAndSwap**
  - **Load-linked / Store-conditional**

TestAndSet Instruction

- Definition in pseudocode:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;      // return the OLD value
}
```

atomic

Mutual Exclusion using TestAndSet

- Shared boolean variable lock, initialized to FALSE (= unlocked)
- do {

```
    while ( TestAndSet (&lock ))
        ;   // do nothing but spinning on the lock (busy waiting)
```

```
    //  …  critical section

    lock = FALSE;
```

```
    //  …    remainder section

} while ( TRUE);
```

Pitfalls with Semaphores

- Correct use of mutex operations:
  - Protect all possible entries/exits of control flow into/from critical section:

    pthread_mutex_lock (&mutex)
    ….
    pthread_mutex_unlock (&mutex)

- Possible sources of synchronization errors:
  - Omitting **lock**(&mutex) or **unlock**(&mutex) (or both) ??
  - **lock**(&mutex) …. **lock**(&mutex) ??
  - **lock**(&mutex1) …. **unlock**(&mutex2) ??
  - if-statement in critical section, **unlock** in then-branch only

Problems: Deadlock and Starvation

- **Deadlock** – two or more threads are waiting indefinitely for an event that can be caused only by one of the waiting threads
  - Typical example: *Nested critical sections*
    - Guarded by locks S and Q, initialized to unlocked

P_0	P_1
mutex_lock(S);	mutex_lock(Q);
mutex_lock(Q);	mutex_lock(S);
…	…
mutex_unlock(S);	mutex_unlock(Q);
mutex_unlock(Q);	mutex_unlock(S);

time ↓

- **Starvation** – indefinite blocking. A thread may never get the chance to acquire a lock if the mutex mechanism is not *fair*.

Deadlock Characterization [Coffman *et al.* 1971]

Deadlock can arise only if **four conditions** hold simultaneously:

- **Mutual exclusion:** only one thread at a time can use a resource.
- **Hold and wait:** a thread holding at least one resource is waiting to acquire additional resources held by other threads.
- **No preemption of resources:** a resource can be released only voluntarily by the thread holding it, *after* that thread has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting threads such that
  - P_0 is waiting for a resource that is held by P_1,
  - P_1 is waiting for a resource that is held by P_2, …,
  - P_{n-1} is waiting for a resource that is held by P_n, and
  - P_n is waiting for a resource that is held by P_0.

Coordinating Shared Access (5)

- Must also rely on implementation for efficiency
- Time to lock / unlock mutex or synchronize threads varies widely between different platforms
- A mutex that all threads access serializes the threads!
  - Convoying
  - Goal: Make critical section as short as possible

```
// in gettask():
int tmpindex;  // local (thread-private) variable
pthread_mutex_lock( &mutex );
tmpindex = p.index++;
pthread_mutex_unlock( &mutex );
task = p.queue [ tmpindex ];
```

Possibly slow shared memory access now outside critical section

Coordinating Shared Access (6)

- When programming on this level of abstraction: can minimize serialization, but not avoid
  - Example: Fine-grained locking

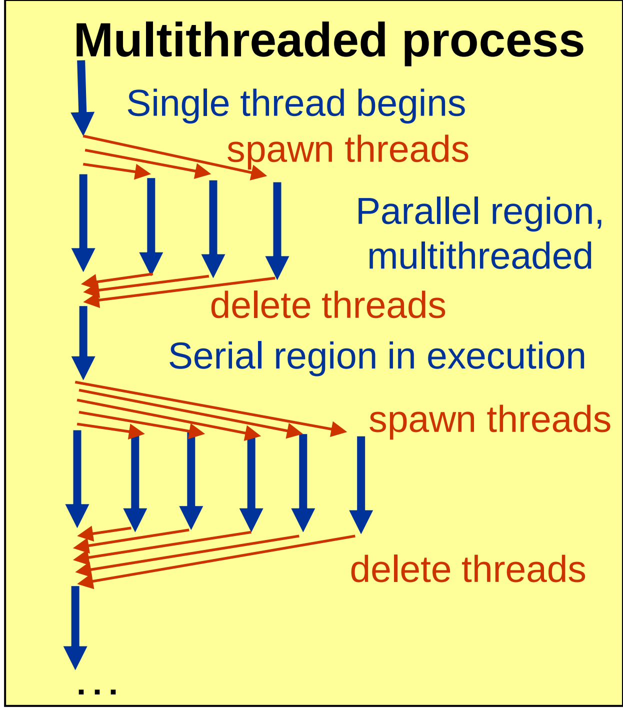
- Better: avoid mutex and similar constructs, and use higher-level data structures that are lock-free
  - Example: NOBLE library

- Also: Transactional memory

More about this in TDDD56

Performance Issues with Threads on Multicores

Performance Issue: Thread Pools



- For a multithreaded process:
  Create a number of threads in a pool where they await work
- Advantages:
  - Faster to service a request with an existing thread than to create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
- Win32 API
- OpenMP

Performance Issue: Spinlocks on Multiprocessors

- Recall busy waiting at spinlocks:

```
// … lock initially 0 (unlocked)
while ( ! test_and_set( &lock ))
      ;
// … the critical section …
lock = 0;
```

- Test_and_set in a tight loop
  → high bus traffic on multiprocessor
  - Cache coherence mechanism must broadcast all writing accesses (incl. t&s) to lock immediately to all writing processors, to maintain a consistent view of lock's value

  → contention

  → degrades performance

Solution 1: TTAS

- Combine with ordinary read:

```
while ( ! test_and_set( &lock ))
    while ( lock )
          ;
// … the critical section …
```

- Most accesses to lock are now reads
  → less contention,
  as long as lock is not released.

Solution 2: Back-Off

- ```
  while ( ! test_and_set( &lock ))
      do_nothing_for ( short_time );
  // … the critical section …
  ```

- Exponential / random back-off
  ```

Performance Issue: Manual Avoidance of Idle Waiting

- Thread that unsuccessfully tried to acquire mutex is blocked but not suspended
  - busy waiting, idle ☹

- Can find out that mutex is locked and do something else:

  pthread_mutex_trylock (&mutex_lock);

  - If mutex is unlocked, returns 0
    If mutex is locked, returns EBUSY

- Useful for locks that are not accessed too frequently and for threads having the chance to do something else

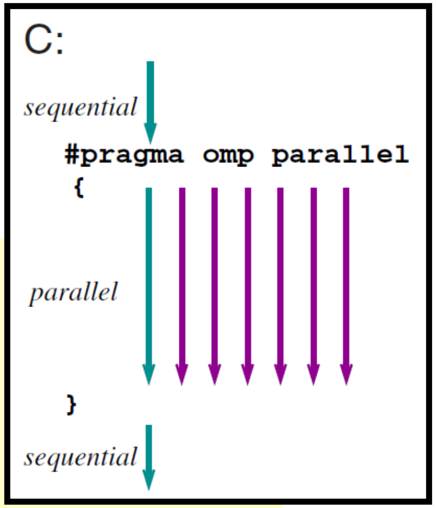
Better Programmability for Thread Programming

Short overview of OpenMP™

(see TDDC78 for in-depth treatment of OpenMP)

OpenMP™

- Standard for shared-memory thread programming
- Developed for incremental parallelization of HPC code
- Directives (e.g. #pragma omp parallel)
- Support in recent C compilers, e.g. gcc from v.4.3 and later
- High-level constructs for data and work sharing
  - Low-level thread programming still possible




```
#include <omp.h>
…
#pragma omp parallel  shared(N)  private(i)
{    // creating a team of OMP_NUM_THREADS threads
    …
#pragma omp for  schedule(static)
    for (i=0;  i<N;  i++)
        domuchwork( i );
}
```

Work (here: iterations of for loop) shared among all threads of the current team

Performance Issue: Load Balancing

- Parallel execution time (”makespan” in scheduling terminology) is determined by the longest-running process / thread
- Minimized by **load balancing**
  - Static – mapping of tasks to cores *before* runtime, no OH
  - Dynamic – mapping done *at* runtime
    - Shared (critical section) or distributed work pool
    - On-line problem – don’t know the future, only the past
      - Heuristics such as best-fit, random work stealing

Example: Parallel loop, iterations of unknown+varying workload

```
#pragma omp parallel for  schedule(dynamic)
for (i=0;  i<N; i++)   work ( i, unknownworkload(i) );
```

Example: Sequential sum in C

```
#define N 2048

int sum, arr[N];

void main()
{
  // … initialize arr



    for (i=0; i<N; i++)  {
       sum = sum + arr[i];
    }

  // … output sum
}
```

Example: Parallel sum in OpenMP

```
#include <omp.h>

#define N 2048

int sum, arr[N];

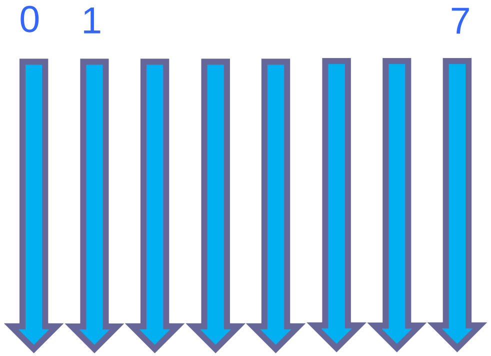
void main()
{
  // … initialize arr
#pragma omp parallel private(i)
  {
#pragma omp for reduction(+:sum)
    for (i=0; i<N; i++)  {
       sum = sum + arr[i];
    }
  }
  // … output sum
}
```

Message Passing

MPI – Program Startup

- MPI (implementation) is a **library** of message passing operations, linked with the application's executable code.

- **SPMD** execution style
  - all started processes (at least, 1 per node) execute **main**() of the same program



- Startup script (platform-dependent), e.g.:

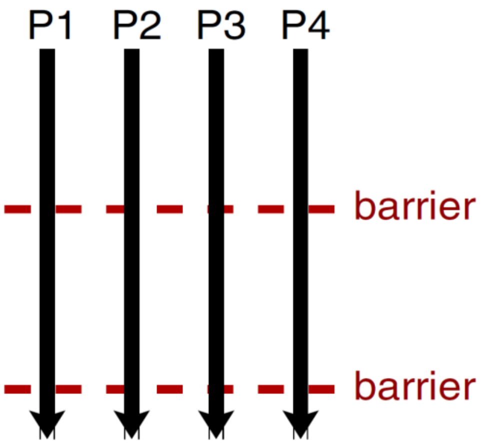
  mpirun –np 8 a.out

  launches 8 MPI processes, each executing main() of a.out
  - Distinguished only by their **MPI rank** (unique ID in 0 … #processes – 1)

Background: SPMD vs. Fork-Join

Parallel program execution styles

SPMD style



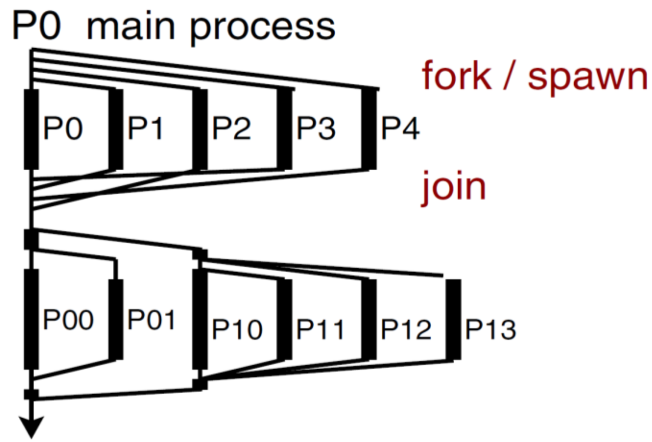
constant number of parallel activities
(processors / processes / threads)

static mapping to processors

mostly flat parallelism
(nested parallelism by group splitting)

Example: MPI, HPF, UPC, NestStep

Fork–join–style



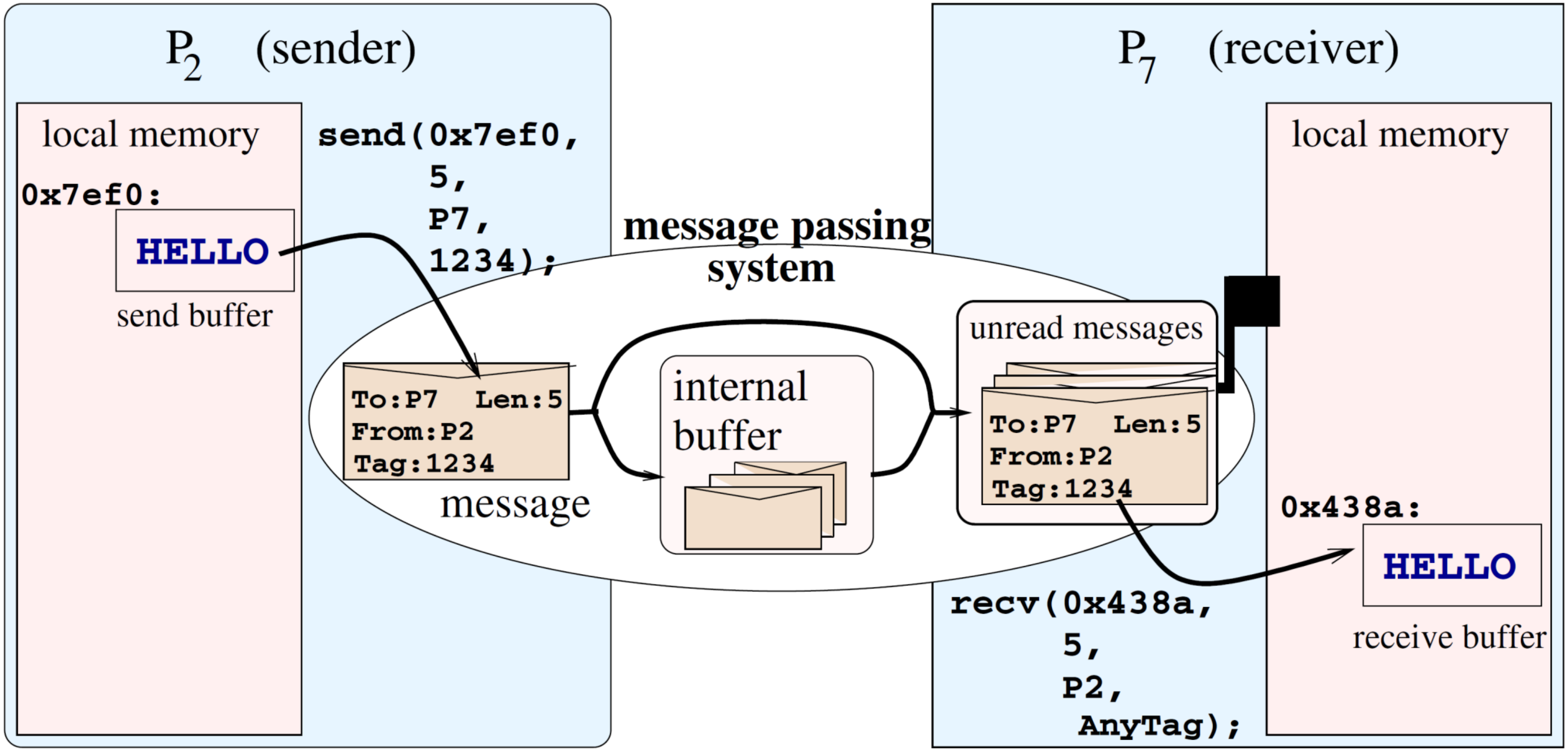
dynamic creation and deletion
of parallel activities

needs dynamic scheduling (overhead)

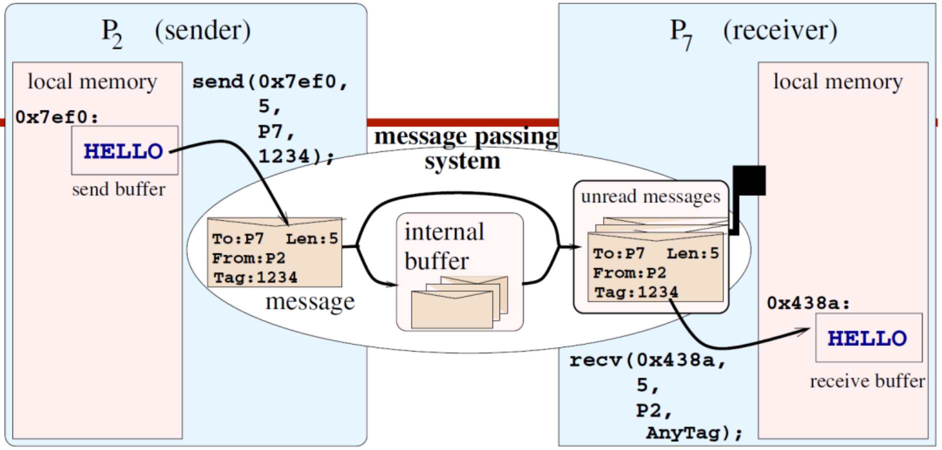
naturally nested parallelism
(nested parallelism by nested spawning)

Example: pthreads, Java threads, Unix–fork,
OpenMP, PVM, MPI–2, Cilk

Hello World (1)



Hello World (2)




```
#include <mpi.h>

void main( void )
{
 MPI_Status status;
 char *string = "xxxxx";  // receive buffer
 int myid;

 MPI_Init( NULL, NULL );
 MPI_Comm_rank( MPI_COMM_WORLD, &myid );
 if (myid==2)
   MPI_Send( "HELLO", 5, MPI_CHAR, 7, 1234, MPI_COMM_WORLD );
 if (myid==7) {
   MPI_Recv( string, 5, MPI_CHAR, 2, MPI_ANY_TAG,
             MPI_COMM_WORLD, &status );
   printf( "Got %s from P%d, tag %d\n",
           string, status.MPI_SOURCE, status.MPI_TAG );
 }
 MPI_Finalize();
}
```

C

MPI Core Routines (C API)

```
MPI_Init( int *argc, char ***argv );

MPI_Finalize( void );

MPI_Send( void *sbuf, int count, MPI_Datatype datatype,
                              int dest, int tag, MPI_Comm comm );

int MPI_Recv( void *dbuf, int count, MPI_Datatype datatype,
     int source, int tag, MPI_Comm comm, MPI_Status *status );

MPI_Comm_size( MPI_Comm comm, int *psize);

MPI_Comm_rank( MPI_Comm comm, int *prank);
```

Status object:

- *status*->MPI_SOURCE indicates the sender of the message received;
- *status*->MPI_TAG indicates the tag of the message received;
- *status*->MPI_ERROR contains an error code.

MPI – Determinism

Message passing is generally nondeterministic:

Arrival order of two sent messages is unspecified.

MPI guarantees that two messages sent from processor A to B will arrive in the order sent.

Messages can be distinguished by sender and a tag (integer).

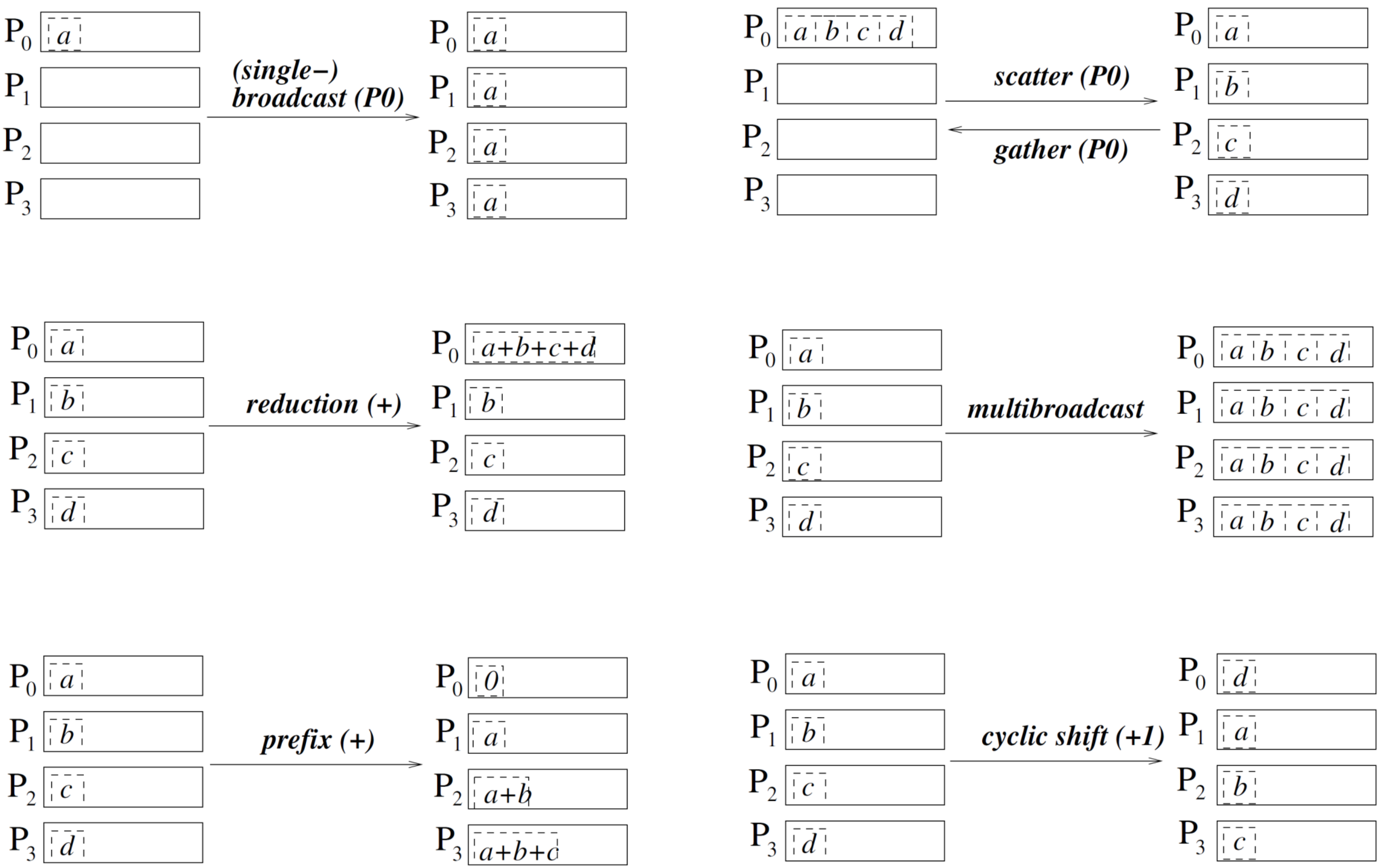
User-defined nondeterminism in receive operations:

wildcard `MPI_ANY_SOURCE`

wildcard `MPI_ANY_TAG`

MPI blocking vs. nonblocking communication operations → TDDC78
MPI communication modes (synchronous, buffered, …) → TDDC78

Collective Communication Operations



Some Collective Communication Operations in MPI

Single-Broadcast:

```
MPI_Bcast( void *srbuf, int count, MPI_Datatype datatype,
                              int rootrank, MPI_Comm comm );
```

Reduction:

```
MPI_Reduce( void *sbuf, void *rbuf, int count,
              MPI_Datatype datatype, MPI_Op op, int rootrank,
                                            MPI_Comm comm );
```

with predefined *op* $\in$ { `MPI_SUM`, `MPI_MAX`, ... }
or user-defined by `MPI_Op_Create`.

`MPI_Allreduce`

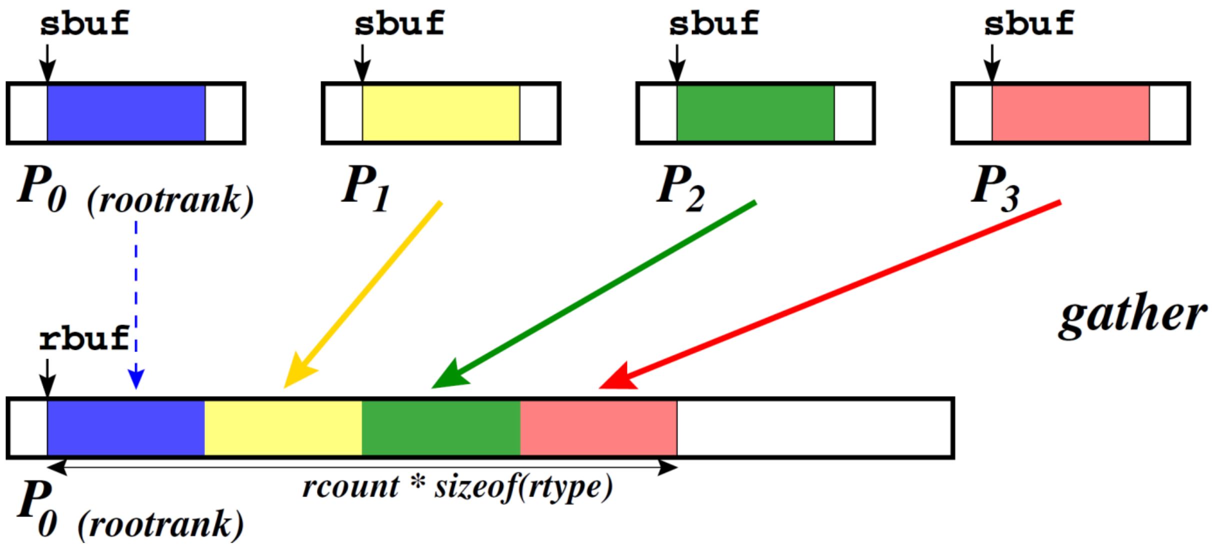
Barrier synchronization:

```
int MPI_Barrier( MPI_Comm comm );
```

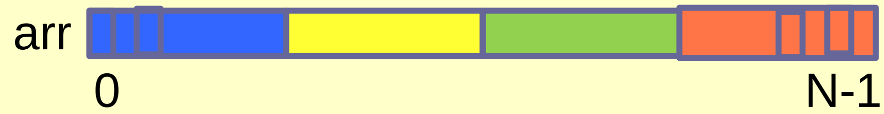
Collective Communication in MPI
Example: Scatter and Gather

```
int MPI_Scatter( void *sbuf, int scount, MPI_datatype stype,
                 void *rbuf, int rcount, MPI_datatype rtype,
                 int rootrank, MPI_Comm comm );

int MPI_Gather( void *sbuf, int scount, MPI_datatype stype,
                void *rbuf, int rcount, MPI_datatype rtype,
                int rootrank, MPI_Comm comm );
```



Example: Global Sum in MPI

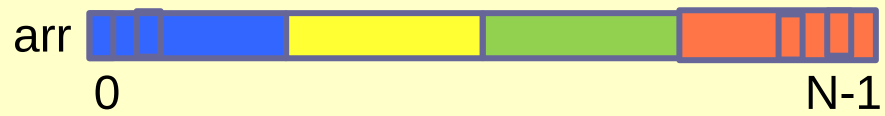



```
#include <mpi.h>
#define N 2048
…
void main( int argc, int argv )
{
  int rank, p, i, sum, arr[N], *myarr, myN, mysum;




}
```

Example: Global Sum in MPI




```
#include <mpi.h>
#define N 2048
…
void main( int argc, int argv )
{
  int rank, p, i, sum, arr[N], *myarr, myN, mysum;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &p );



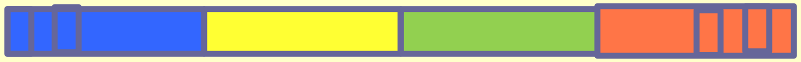






}
```

Example: Global Sum in MPI

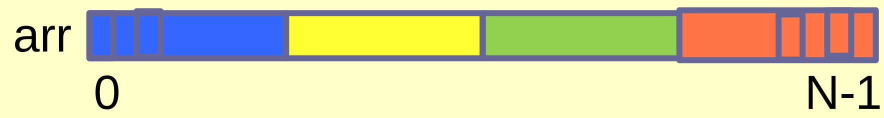



```
#include <mpi.h>
#define N 2048
…
void main( int argc, int argv )
{
  int rank, p, i, sum, arr[N], *myarr, myN, mysum;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &p );
  if (rank==0)     // initialize on P0 only:
     for (i=0; i<N; i++)
        arr[i] = …;
  myN = N / p;     // assume p divides N
  myarr = (int *) malloc( myN * sizeof(int));




}
```

Example: Global Sum in MPI




```
#include <mpi.h>
#define N 2048
…
void main( int argc, int argv )
{
  int rank, p, i, sum, arr[N], *myarr, myN, mysum;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &p );
  if (rank==0)      // initialize on P0 only:
     for (i=0; i<N; i++)
         arr[i] = …;
  myN = N / p;      // assume p divides N
  myarr = (int *) malloc( myN * sizeof(int));
  MPI_Scatter( arr, N, MPI_INT,  myarr, myN, MPI_INT,  0,  MPI_COMM_WORLD);

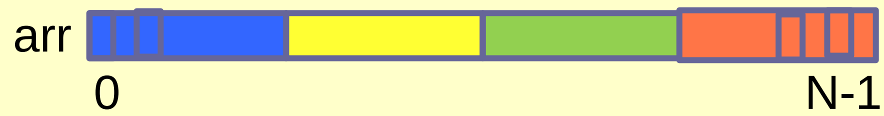






}
```

Example: Global Sum in MPI

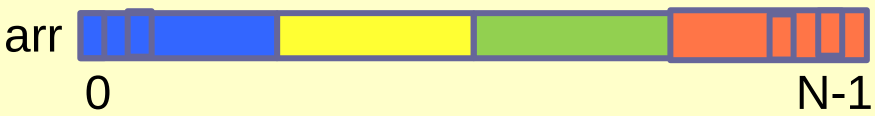



```
#include <mpi.h>
#define N 2048
…
void main( int argc, int argv )
{
  int rank, p, i, sum, arr[N], *myarr, myN, mysum;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &p );
  if (rank==0)     // initialize on P0 only:
     for (i=0; i<N; i++)
        arr[i] = …;
  myN = N / p;     // assume p divides N
  myarr = (int *) malloc( myN * sizeof(int));
  MPI_Scatter( arr, N, MPI_INT,  myarr, myN, MPI_INT,  0,  MPI_COMM_WORLD);
  mysum = 0;
  for (i=0; i<myN; i++)
     mySum += myarr[i];      // each process calculates partial sum of N/p elements



}
```

Example: Global Sum in MPI




```
#include <mpi.h>
#define N 2048
…
void main( int argc, int argv )
{
  int rank, p, i, sum, arr[N], *myarr, myN, mysum;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &p );
  if (rank==0)      // initialize on P0 only:
     for (i=0; i<N; i++)
         arr[i] = …;
  myN = N / p;      // assume p divides N
  myarr = (int *) malloc( myN * sizeof(int));
  MPI_Scatter( arr, N, MPI_INT,  myarr, myN, MPI_INT,  0,  MPI_COMM_WORLD);
  mysum = 0;
  for (i=0; i<myN; i++)
      mySum += myarr[i];       // each process calculates partial sum of N/p elements
  MPI_Reduce( &mysum, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  // … now output sum
  MPI_Finalize();
}
```

More about MPI → TDDC78

- MPI Communication modes for point-to-point communication
- MPI Communicators and Groups
- MPI Datatypes
- MPI One-Sided Communication (Remote Memory Access)
- MPI Virtual Topologies
- Labs: Image filter, Particle simulation

Questions?

Further Reading (Selection)

- C. Lin, L. Snyder: *Principles of Parallel Programming*. Addison Wesley, 2008. (general introduction; Pthreads)
- B. Wilkinson, M. Allen: *Parallel Programming, 2e*. Prentice Hall, 2005. (general introduction; pthreads, OpenMP, MPI)
- M. Herlihy, N. Shavit: *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. (threads; nonblocking synchronization)
- Chandra, Dagum, Kohr, Maydan, McDonald, Menon: *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.
- Barbara Chapman *et al*.: *Using OpenMP - Portable Shared Memory Parallel Programming*. MIT press, 2007.
- OpenMP: www.openmp.org
- MPI: www.mpi-forum.org