Computers Parallelism

Grau en Ciència i Enginyeria de Dades

Xavier Martorell, Xavier Verdú

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC)

2019-2020 Q2

Creative Commons License

Table of Contents

Introduction to Parallelism

- Parallelism is the capacity to perform several things at the same time

- Parallel computing: multiple calculations are executed at the same time…
  - …but what sort of calculations?

- Summary of Parallelism Levels
  - Bit-Level: the finest grain parallelism (e.g. one 64-bit vs two 32-bit calculations)
  - Instruction-Level: multiple instructions per cycle (e.g. 1 ALU and 1 FPU instructions at a time)
  - Data-Level: same calculation to multiple data (e.g. 3D-graphics processing)
  - Task-Level: different calculations to the same or different datasets (e.g. complex applications)

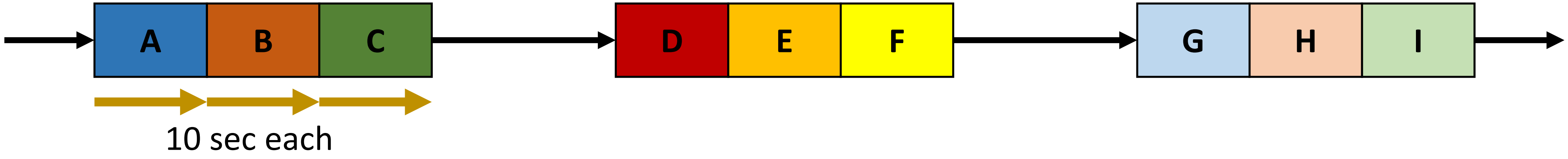
- **This lesson focuses on Data and Task levels of parallelism**

Generic Use Cases of Parallel Programming

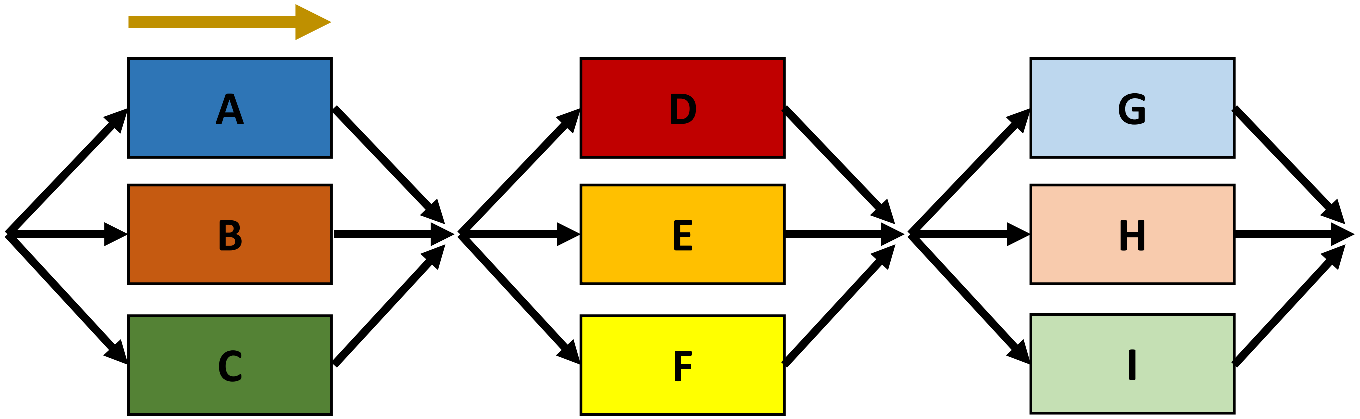
- Parallelism exploitation
  - Workload can be split into multiple workload subsets that can be processed in parallel (with no or few dependencies)
  - Multiple independent inputs can be processed in parallel (with no or few dependencies)
- Task encapsulation
  - Task specific modules focused on a particular task execution to improve performance (e.g. AI and physics engines in videogames)
- I/O efficiency
  - Decouple I/O treatment from the main execution workload
- Service request pipelining (sustain required Quality-of-Service)
  - Enhance distribution of pipeline processing stages over hardware resources towards improve performance

Sequential vs Parallel Execution

- **Sequential Execution:** code can only be executed by a single Sw thread. It is independent of Hw resource availability.



- **Parallel Execution:** multiple Sw threads can execute code at the same time. The code may show occasional dependencies or synchronization barriers



Parallelism vs Concurrency

- **Parallelism:** N Sw threads run at a given time in N Hw threads

Time(each CPU executes one process)

CPU0	Proc. 0
CPU1	Proc. 1
CPU2	Proc 2

- **Concurrency:** N Sw threads could be potentially executed in parallel, but there are not enough Hw resources to do it
  - The OS selects what Sw thread can run and what Sw thread has to wait

Time (CPU is shared among processes)

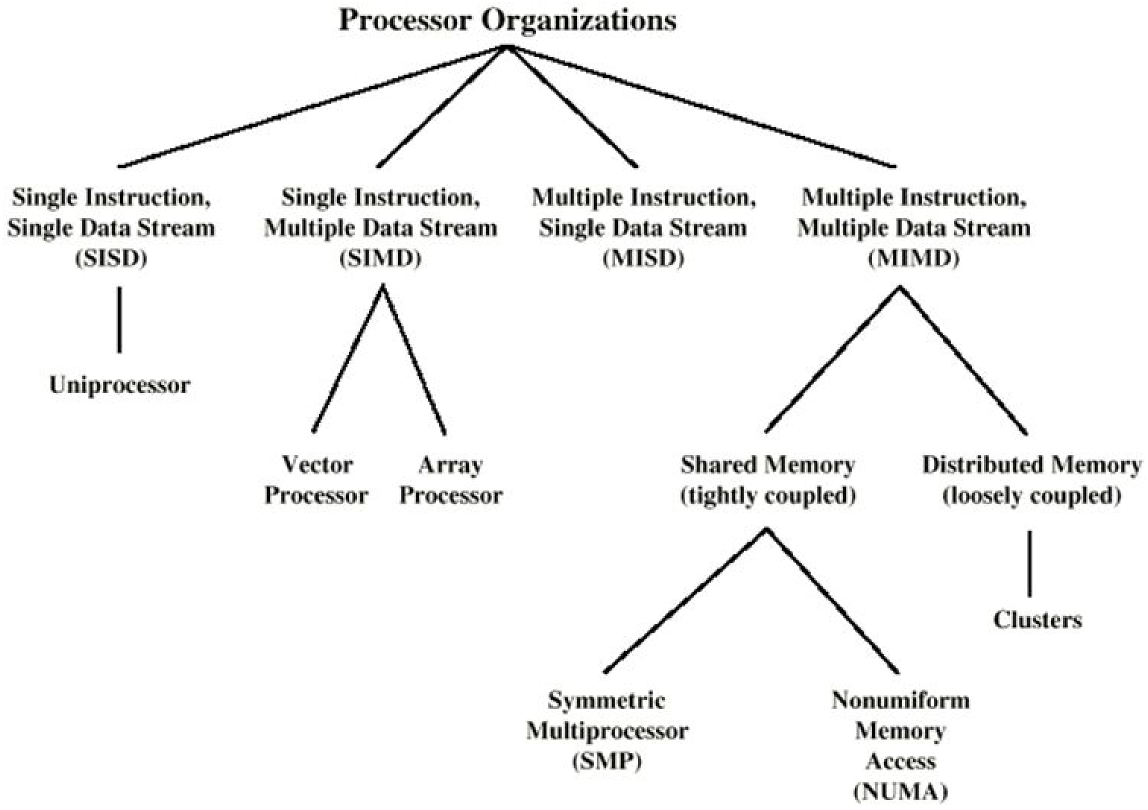
CPU0	**Proc. 0**	**Proc. 1**	**Proc. 2**	**Proc. 0**	**Proc. 1**	**Proc. 2**	**Proc. 0**	**Proc. 1**	**Proc. 2**	**...**

Parallelism: Hardware point of view

- Number of instructions executed per cycle
  - ILP vs TLP

- Resource limitations
  - E.g.: #processors, #cores, #HwThreads
  - Structural hazards
    - A planned instruction cannot be executed because the resource is still being used

- Flynn's taxonomy
  - https://en.wikipedia.org/wiki/Flynn's_taxonomy



Parallelism: Software point of view

- Code execution is inherently limited by dependencies
  - Data dependency
    - A given value of the current instruction depends on the outcome of another instruction
  - Control dependency
    - The execution flow path depends on the outcome of a given instruction

- Library/Compiler support
  - Optimizations at compile-time
  - **Parallel programming models**

Parallelism: Software point of view

- Degrees of Parallelism
  - Coarse-grained parallelism (thousands of instructions or independent programs; ~secs.)
    - Parts of a program, Tasks
    - More difficult detection, less communication requirements
    - Low communication and synchronization overhead, but difficult for load balancing
  - Medium-grained parallelism (<2000 instructions; ~msecs.)
    - Procedures, Routines
  - Fine-grained parallelism (10s-100s instructions; ~usecs)
    - Loops, Groups of instructions
    - Assisted by compiler (difficult to detect by programmers) and parallel programming models
    - Suitable for shared memory approaches and easy for load balancing

- **Developer design decisions may have important consequences!!!!**

Parallelism: Operating System point of view

- Process and thread management
  - User-Level Threads vs Kernel-Level Threads → **Important impact on scheduling**
- Scheduling policies
  - Affinity based, priority based
- Load balancing across hardware resources
  - Dealing with dependencies and shared-resource contention
- Communication and synchronization capabilities
  - **Inter-process communication (IPC)**
    - Addressed in this lesson

Tradeoffs of Parallelism

- Shared-Memory vs Distributed-Memory Computing
  - Shared-memory
    - suitable for threaded programs
  - Distributed-memory
    - suitable for computational demanding workloads

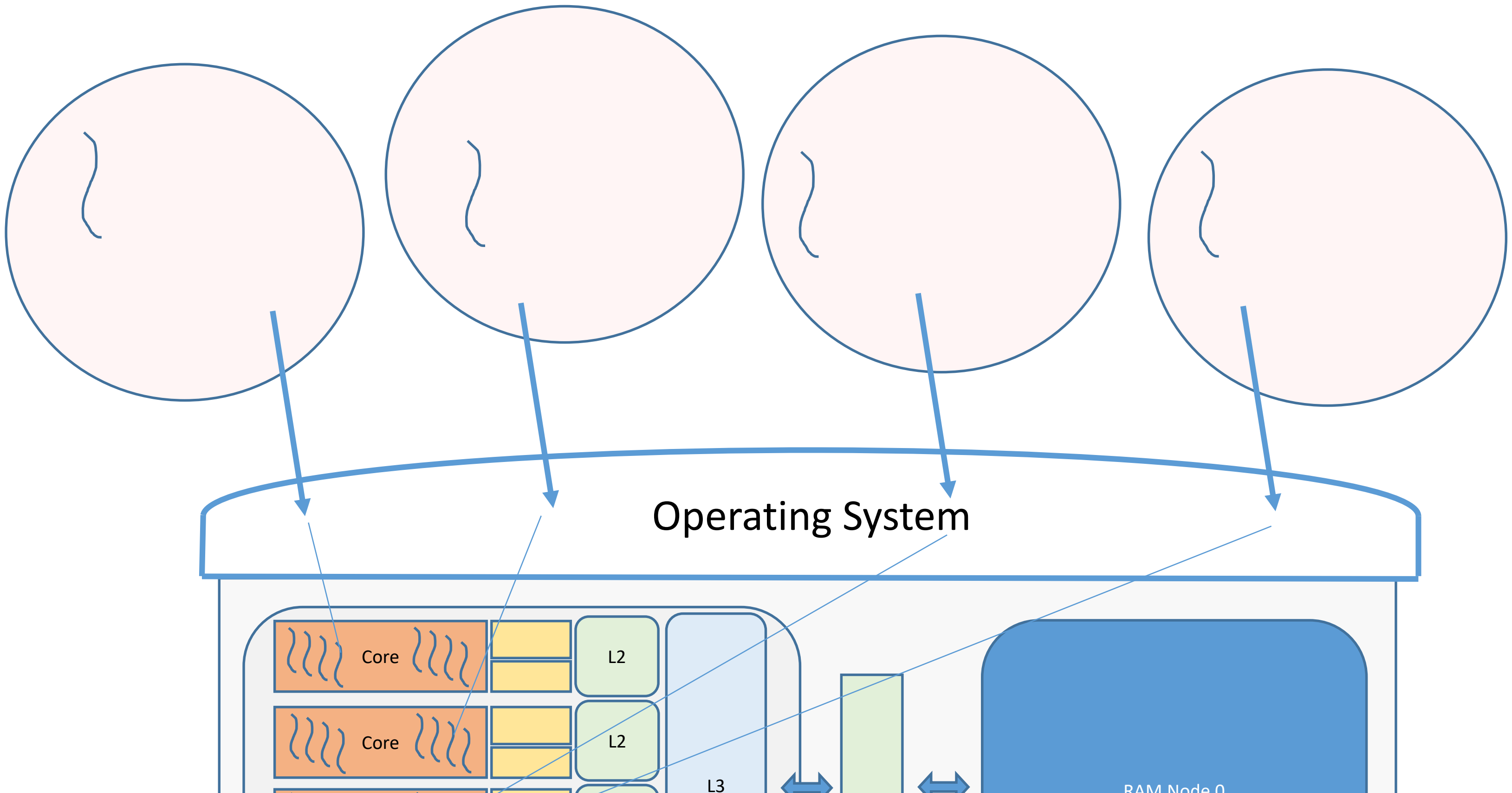
- Hybrid programming techniques
  - combine the advantages of both of them

Tradeoffs of Parallelism

- Processes vs Threads
  - Multi-processing:
    - a global problem is split into several processes that can run on shared or distributed memory devices. Every process has access to its own memory space
      - E.g.: **OpenMPI**
  - Multi-threading:
    - a code snippet is split into several flows that can run on a shared memory device. Every thread has access to all data
      - E.g.: **OpenMP**

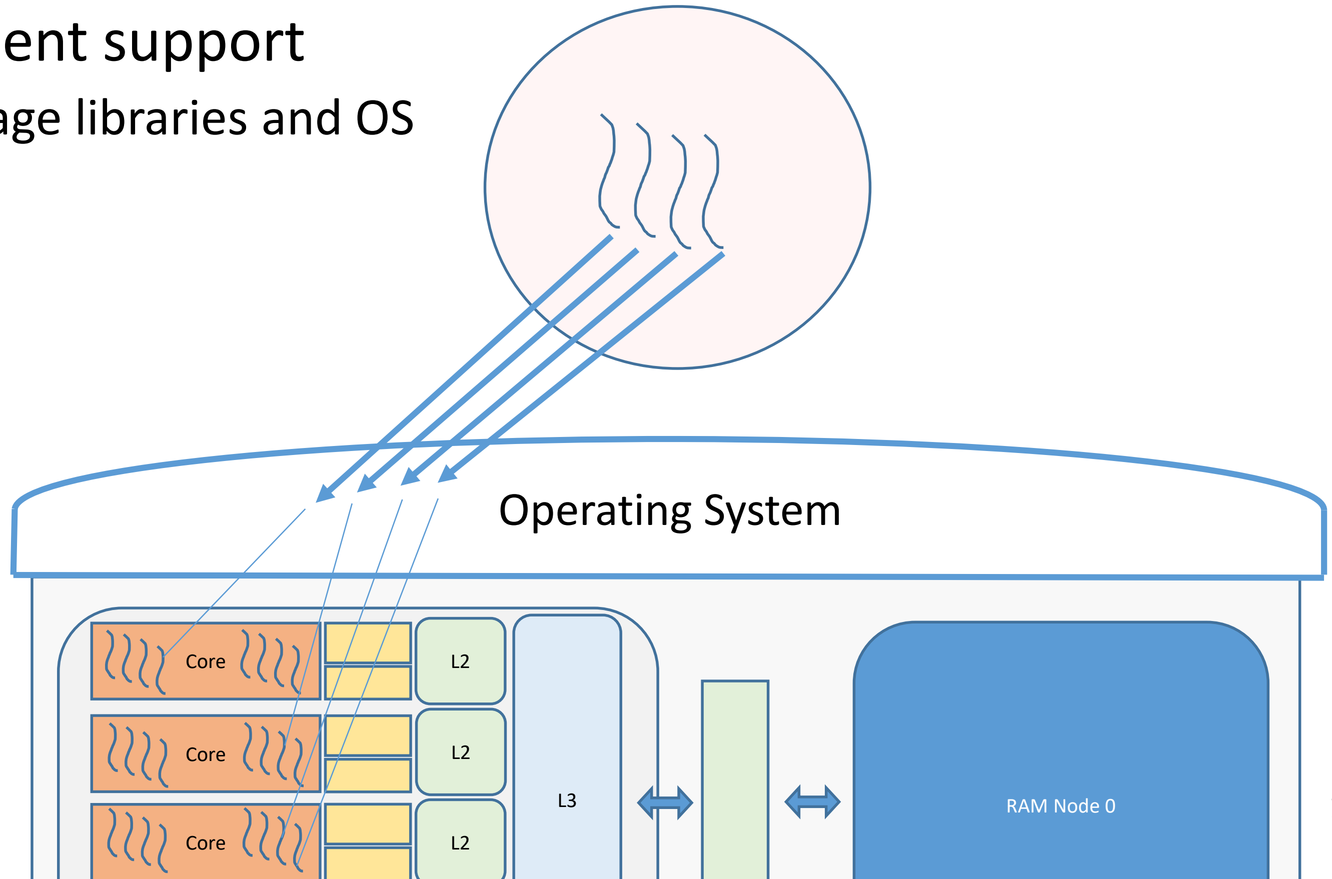
Utilization of resources (Processes)

- Multi-process approach
  - No memory sharing in this case or complexity on the memory management



Utilization of resources (Threads)

- Multi-thread approach
  - Allow to exploit parallelism inside each process
  - Require thread management support
    - From programming language libraries and OS



Inter-Process Communication (IPC)

- Cooperating processes/threads need communication and synchronization
  - Data Exchange / Event Notification
  - Blocking (synchronous) / Non-blocking (asynchronous) communication
  - Direct / Indirect communication

- Processes can run on a single host or on several remote hosts

- IPC Method selection based on…
  - Latency, bandwidth, type of data exchanged

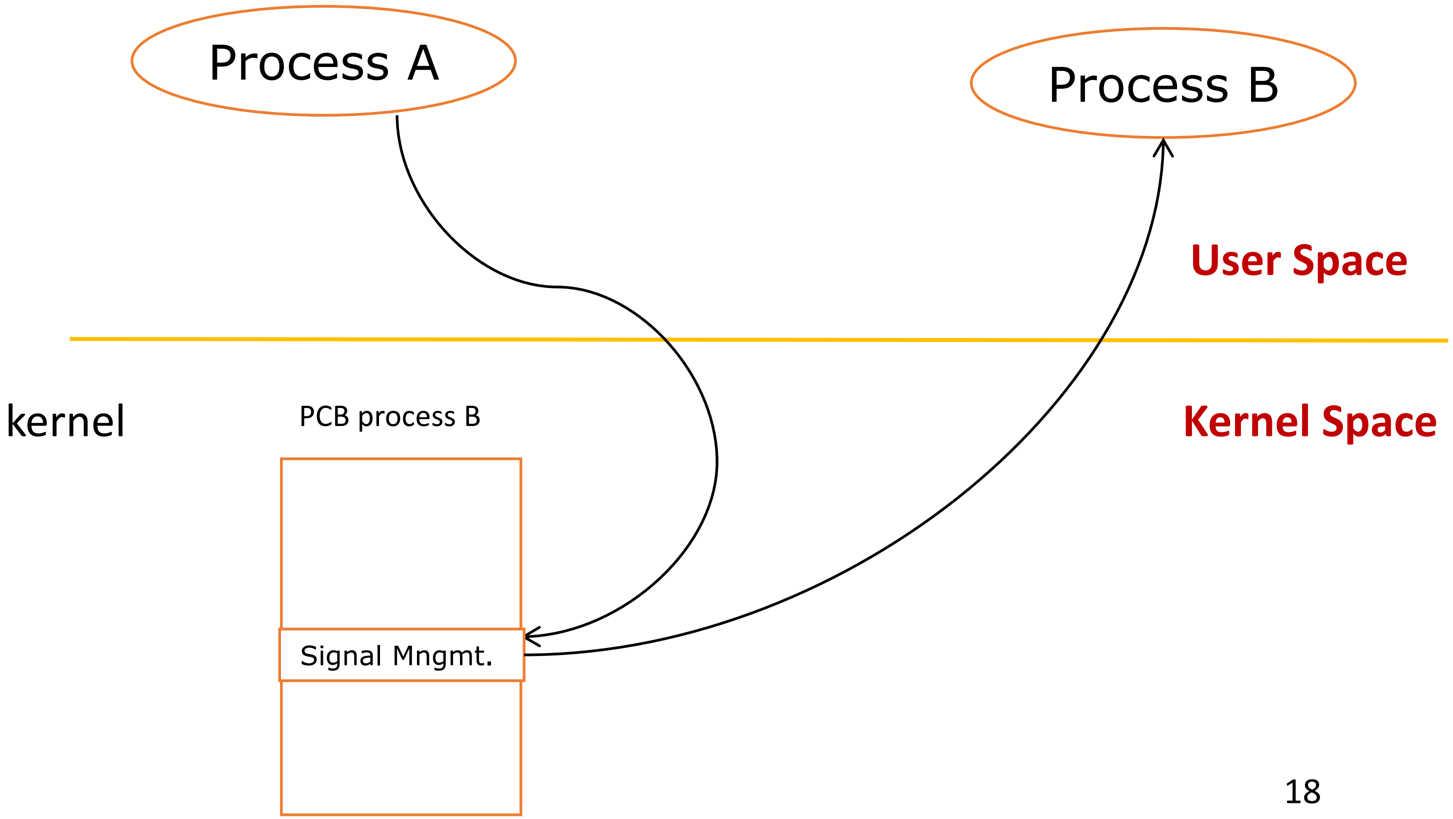
- **Bugs really difficult to find and solve!!!!**

Inter-Process Communication (IPC)

- Communication Mechanisms
  - Signals
  - Pipes/named pipes
  - Sockets
  - Message Passing
    - **Message-Passing Interface (e.g. OpenMPI)**
  - Shared Memory
  - Memory-Mapped files

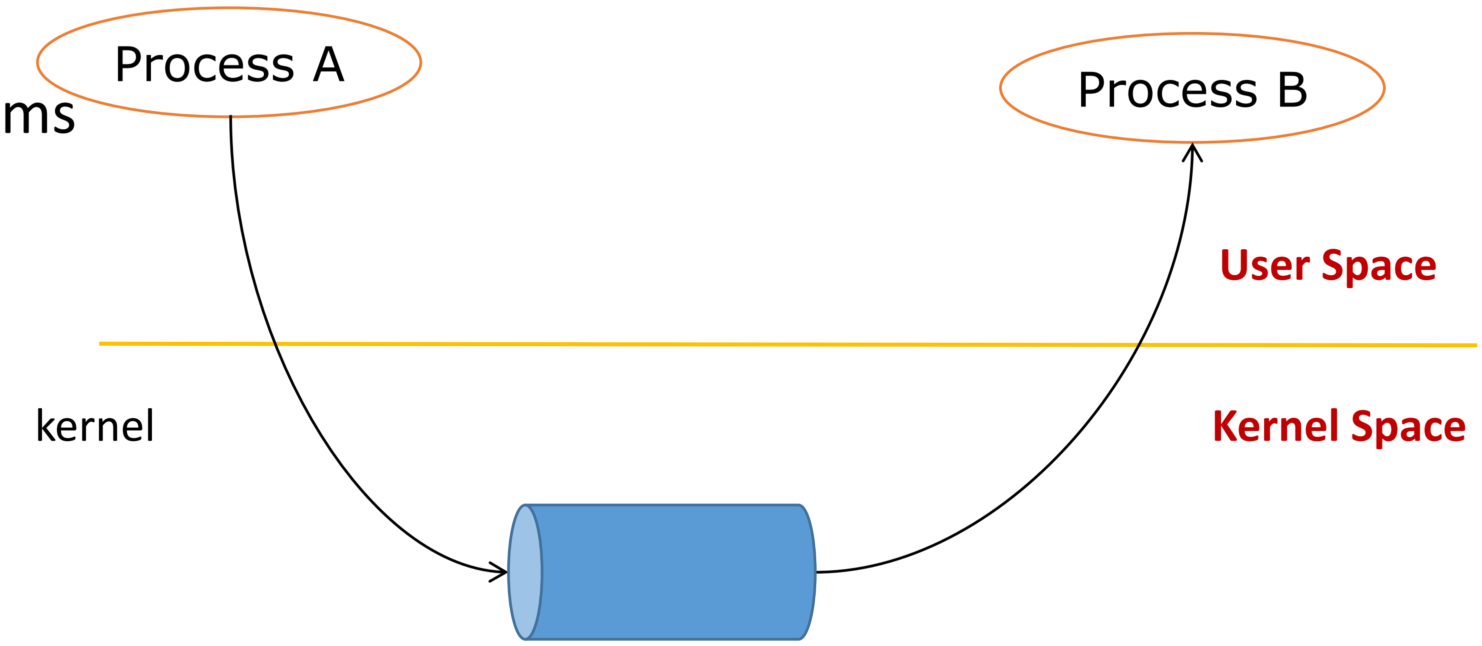
Inter-Process Communication (IPC)

- Signals
  - Event notification
  - NO data exchange
  - Change execution flow
  - Programmable actions



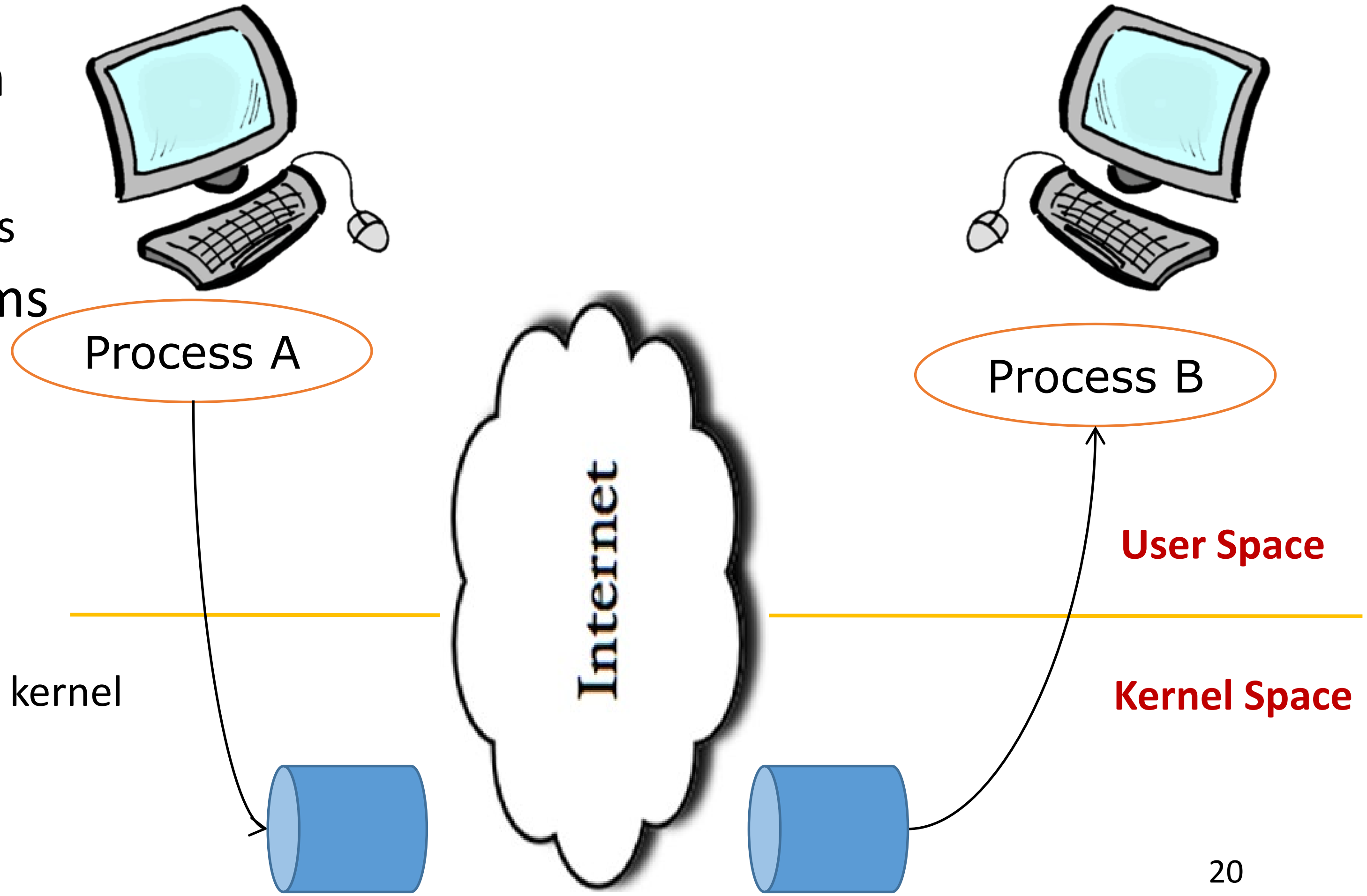
Inter-Process Communication (IPC)

- Pipes/Named Pipes
  - FIFO memory buffer
  - N:M end-points
  - Synchronization mechanisms



Inter-Process Communication (IPC)

- Sockets
  - Full-duplex communication
  - Between two processes
    - Alternative implementations
  - Synchronization mechanisms
  - Multiple socket types

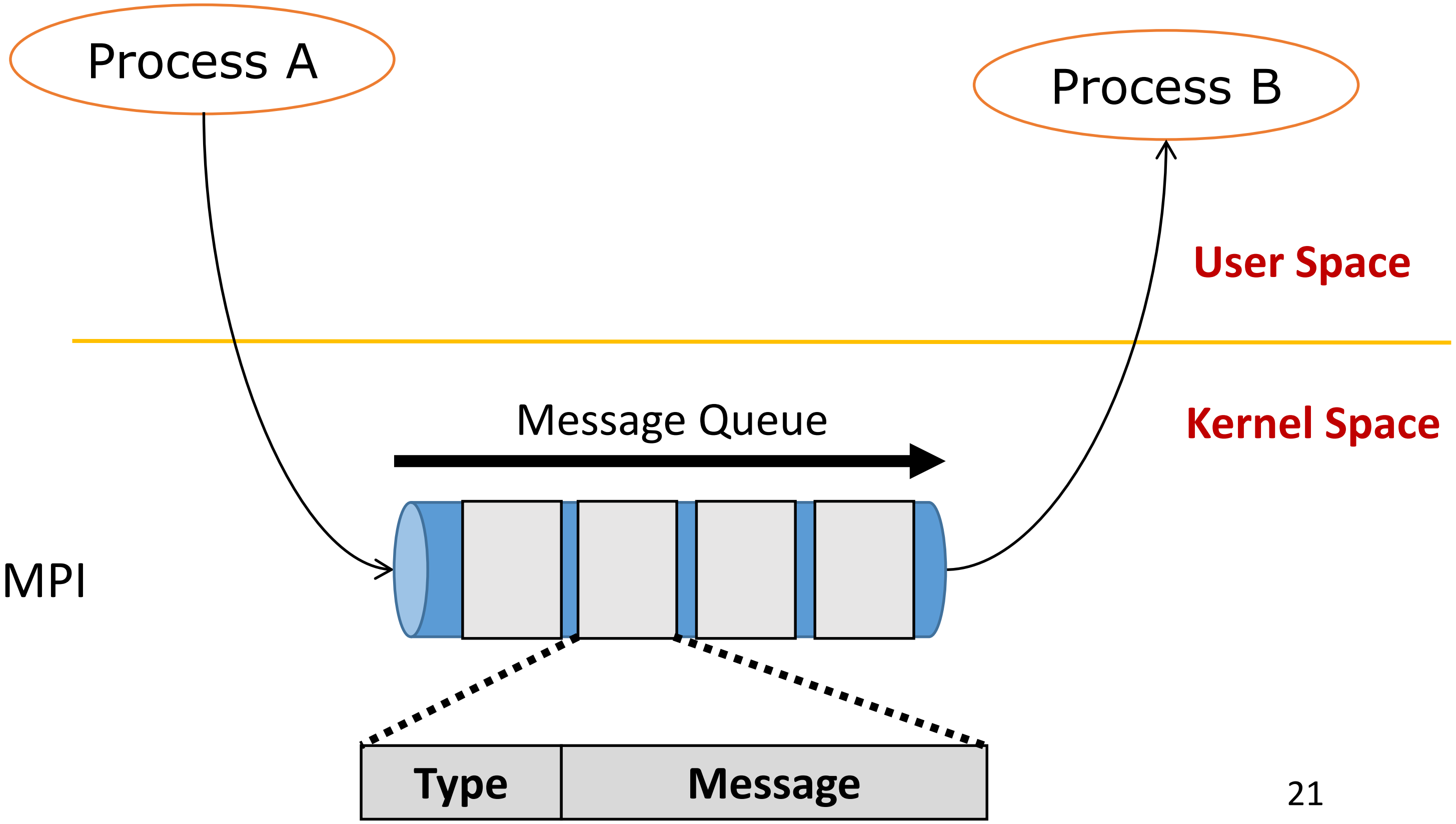


Inter-Process Communication (IPC)

- Message Passing
  - Implemented through syscalls
    - Overhead
  - Message Queues
  - Structured messages

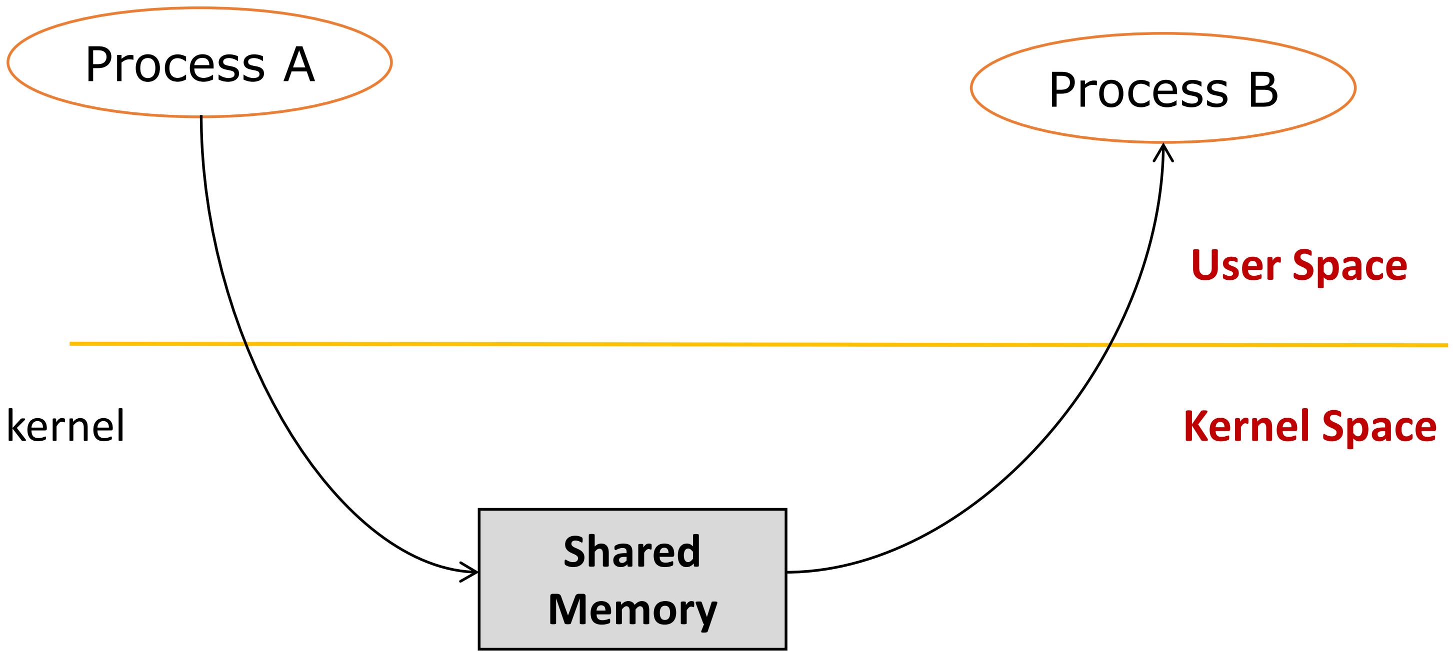
- MPI
  - Message Passing Interface
  - Library specification
  - Aim at parallel computing

- **OpenMPI**
  - Open source implementation of MPI
  - https://www.open-mpi.org/



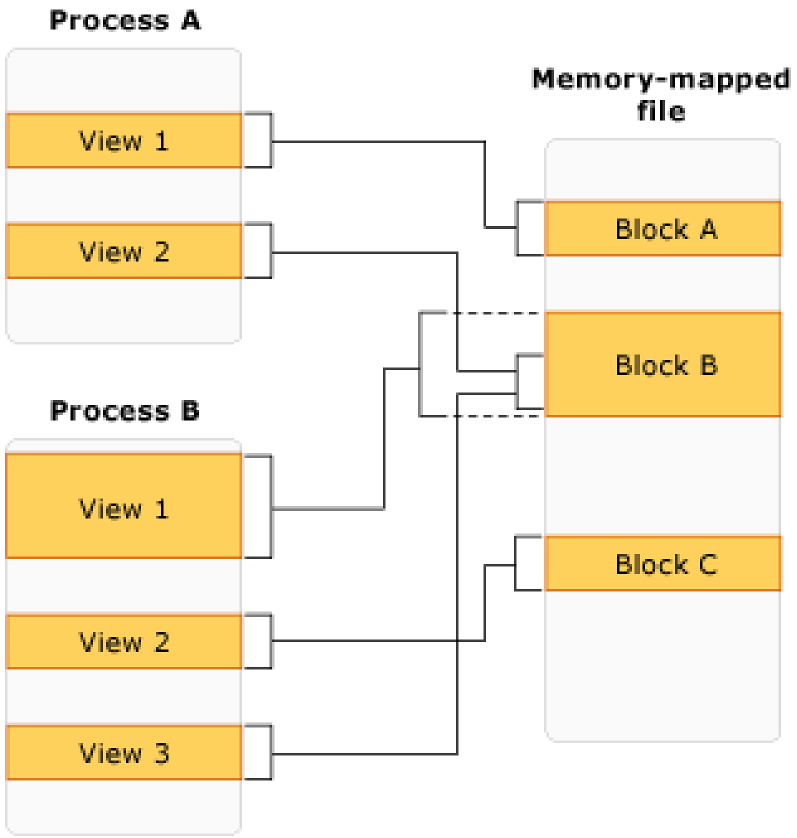
Inter-Process Communication (IPC)

- Shared Memory
  - Multiple processes share virtual memory space
  - Very fast
  - No synchronization



Inter-Process Communication (IPC)

- Memory-Mapped files
  - A file allocated in virtual memory
  - Can be shared among processes

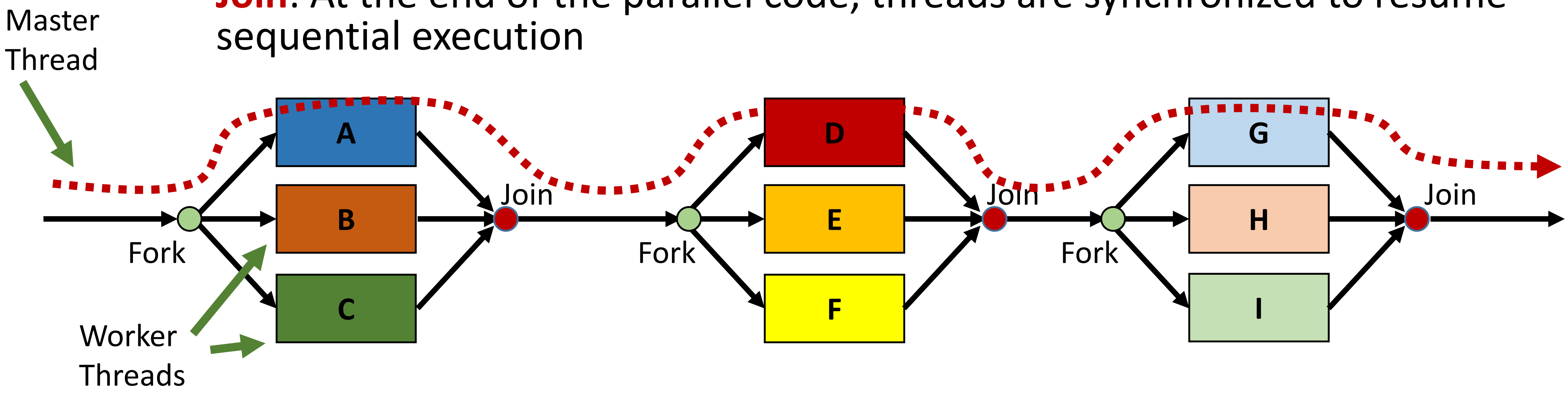


https://docs.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files

Table of Contents

Fork-Join Model

- Parallel programs present code snippets that can be parallelized
  - **Fork**: A piece of code branches off to be executed by multiple threads
  - **Join**: At the end of the parallel code, threads are synchronized to resume sequential execution



- Master thread spawns multiple worker threads in parallel regions

Parallel Programming Models

- This lesson focuses on
  - **OpenMP** (Open specification for Multi Processing)
    - Standard API to write shared memory parallel programs (C, C++ and Fortran)
    - http://www.openmp.org
  - Pthreads (POSIX Threads)
    - POSIX (Portable Operating System Interface) API and parallel execution model
    - Supported by multiple Oss

- Spawn multiple threads that share memory (access to all data)

- Code size increases more than multithreaded code does
  - Codes are more complex, but sometimes performance is higher

Difficulties of Shared Data

- The correct behavior of the program MUST NOT depend on the sequence of processes or threads execution (a.k.a. **race condition**)
  - It is out of our control

- Shared data MUST BE properly protected to guarantee data consistency
  - **Critical sections** (code that accesses to shared data) have to be correctly protected and accessed
  - Does a given critical section really needs to be protected???

- We have to avoid **deadlocks:** two or more threads wait for each other for eternity

- Over protection of parallel code degrades performance
  - We have protected code that is not necessary

- It is difficult to totally guarantee the correct behavior of multithreaded codes
  - Possible collisions among threads depend on the sequence of processes/threads execution
    - E.g. Debug deterministic sequences

The Big Issue: Sharing Data

- Threads can potentially access to any shared data
- Critical Section
  - Code that accesses to shared data

```
Func_assign(int value){
  …
  pos = pos_free;
  vector[pos] = value;
  pos_free++;
  …
}
```

Time

Th_0

```
Func_assign(10){
  …
  pos = pos_free;
  vector[pos] = 10;
  pos_free++;
  …
}
```

Th_1

```
Func_assign(20){
  …
  pos = pos_free;
  vector[pos] = 20;
  pos_free++;
  …
}
```

CRITICAL SECTION

The Big Issue: Sharing Data

- We cannot control when multiple threads may collide in a shared data
- Shared data MUST BE properly protected to guarantee data consistency

//at this moment: "pos_free == 2"

```
Func_assign(int value){
  …
  pos = pos_free;
  vector[pos] = value;
  pos_free++;
  …
}
```

Time

Th$_0$

```
Func_assign(10){
  …
  pos = pos_free;//2?? 3???
  vector[pos] = 10;
  pos_free++;//3?? 4???
  …
}
```

Th$_1$

```
Func_assign(20){
  …
  pos = pos_free;//2?? 3???
  vector[pos] = 20;
  pos_free++;//3?? 4???
  …
}
```

//at this moment we cannot guarantee the values of "pos_free", "vector[2]", and "vector[3]"

Synchronization Mechanisms

- **Mutex** (MUTual EXclusion)
  - The running thread waits till the lock (mutex) is avilable to get its ownership
- **Barriers**
  - A thread waits for other threads to reach a given point
- Hardware support
  - Atomically read and modify a memory location
- Semaphores
  - A common resource can be accessed by multiple threads
- Spinlocks
  - Active checking for a lock

Synchronization: Mutual Exclusion

- Only a single thread can be in a critical section at a time

- Utilization

  Directive [lock name]

- Limitations
  - It is not exception safe
  - It is not allowed to break out
  - All criticals wait for each other
    - Lock_name helps
- Alternatives
  - Lock based alternatives

Time

Th_0

```
Func_assign(10){
  …
  Directive {
      pos = pos_free;
      vector[pos] = 10;
      pos_free++;
  }
  …
}
```

Th_1

```
Func_assign(20){
  …
      Thread
        is
      waiting
  Directive {
      pos = pos_free;
      vector[pos] = 10;
      pos_free++;
  }
  …
}
```

Synchronization: Barriers

- All threads wait for the remaining threads to reach the checkpoint
- There are implicit barriers at the end of parallel regions

- Utilization

  {

  …

  Barrier Directive

  …

  }

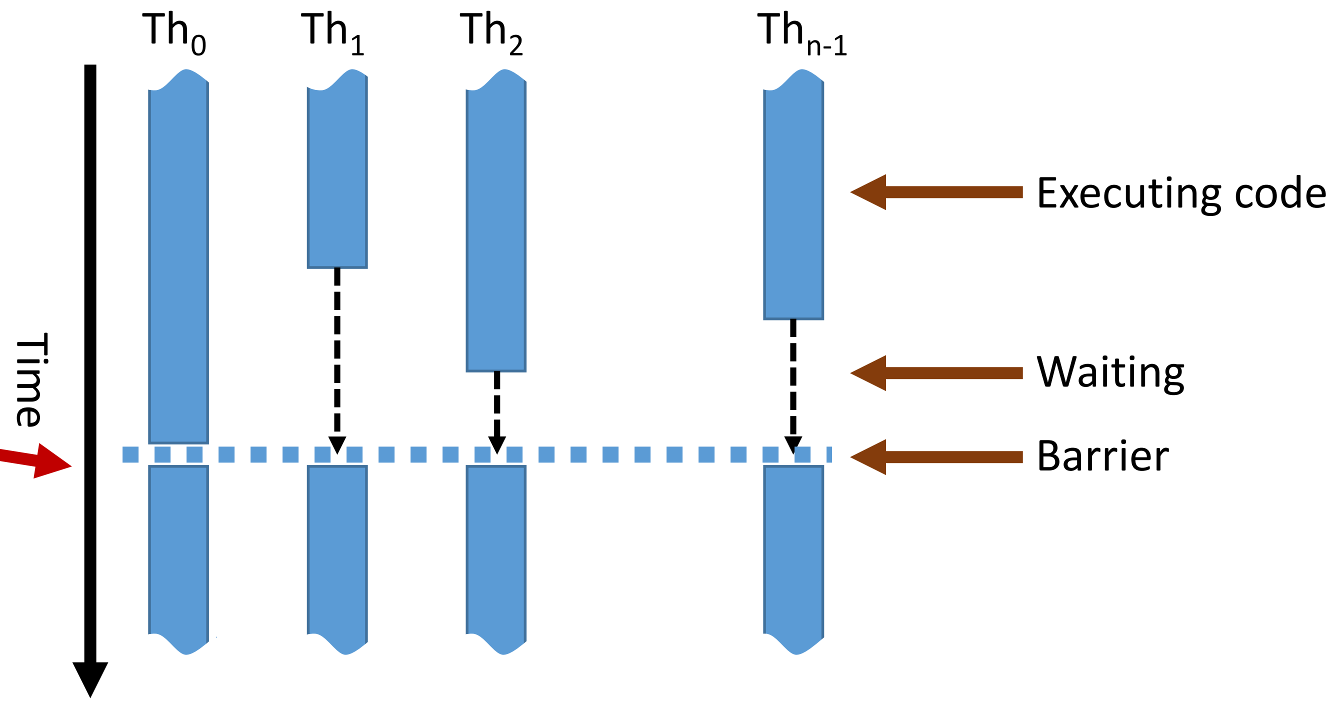


Table of Contents

- Introduction
- Points-of-View
- Tradeoffs
- Approaches
  - Inter-Process Communication
  - Parallel Programming Models
- Let’s code!!!
  - **OpenMPI**
  - OpenMP

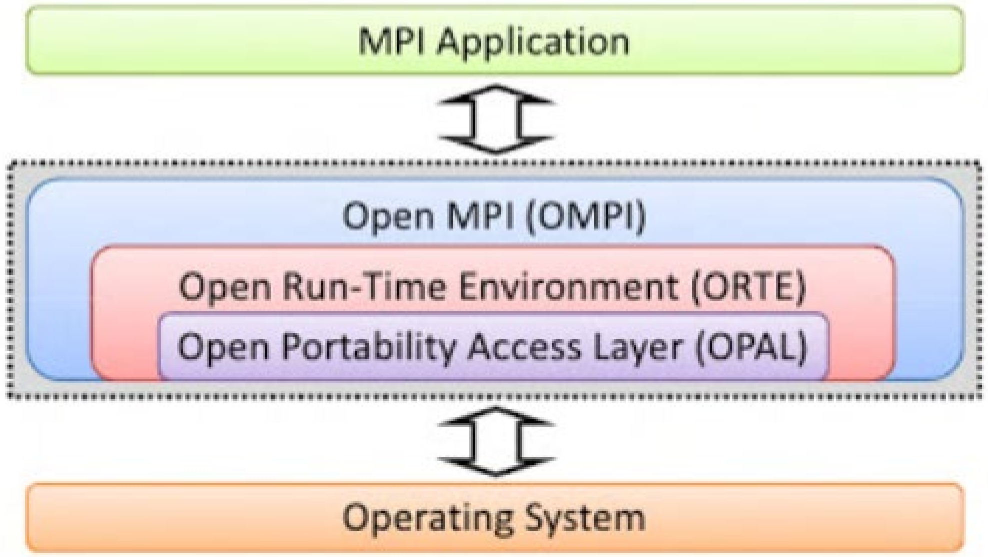
Message Passing Interface: OpenMPI

- Message Passing Interface (MPI) defines routines based on standards
  - From MPI-1 (early '90s) to MPI-3 (2015). Currently MPI-4 (under development)
  - Currently supports hybrid memory systems: distributed & shared
  - Over 430 routines in MPI-3, but most of the MPI apps use a **dozen or less routines**
- MPI Programming Model
  - Data moved from the address space of a process to another process using cooperative routines on each process
- Several free implementations (libraries)
  - E.g. OpenMPI (based on MPI-2 standard)
    - Open source implementation developed and supported by a consortium of academic, research, and industry partners

Code Sections of an OpenMPI Application

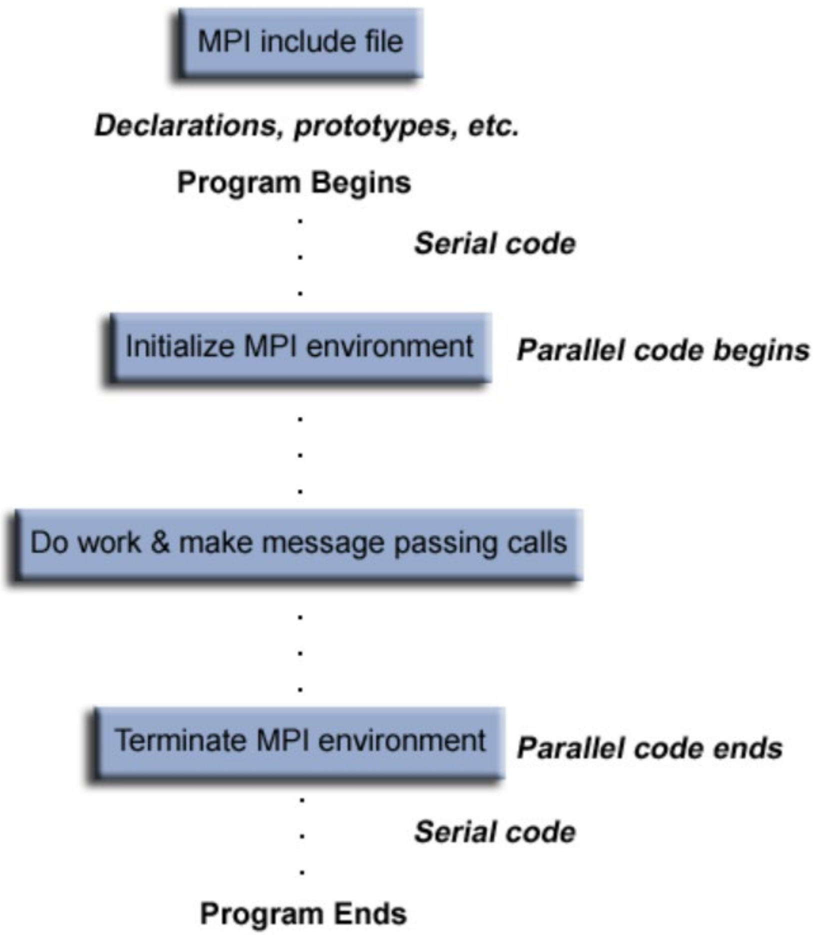
- Framework with functional interrelated interfaces

- OpenMPI apps present are structured:
  - OMPI: OpenMPI
    - Interfaces with apps through MPI API
  - ORTE: Open Run-Time Environment
    - Provides a parallel runtime environment and other services to the upper section
  - OPAL: Open Portable Access Layer
    - Abstraction layer to interface with the OS



Example: Hello World

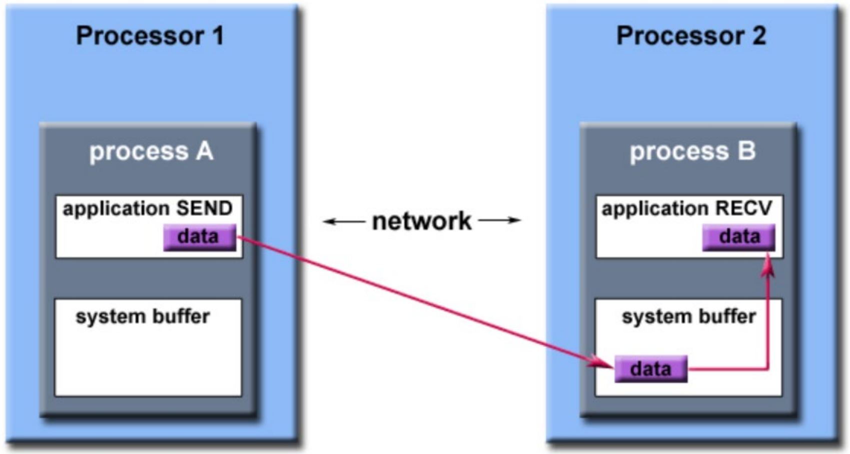
```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
    int numtasks, taskid;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);

    printf ("Hello from task %d of %d!\n", taskid, numtasks);

    MPI_Finalize();
}
```

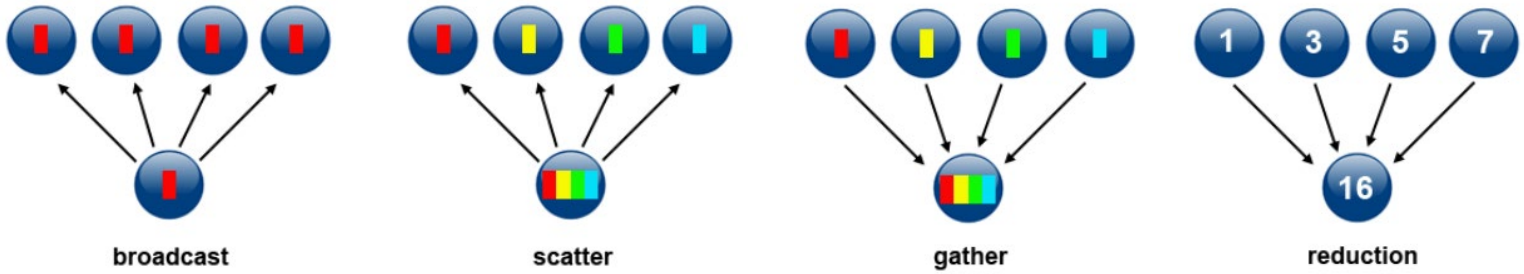



```
#>mpiexec  -n 4  ./hello.exe
```

Message based Communication

- Basic idea: divide the work into multiple communicated tasks
  - Blocking vs non-blocking; Synchronous vs asynchronous
  - System buffer (data in transit) vs app buffer (user managed address spcace)
- Point-to-Point communication
  - Messages between only two MPI tasks
- Collective communication
  - All processes of a given scope





Example: Hello World with blocking messages

```
MPI_Send(buffer, count, type, dest, tag, comm);
MPI_Recv(buffer, count, type, source, tag, comm, status);
```

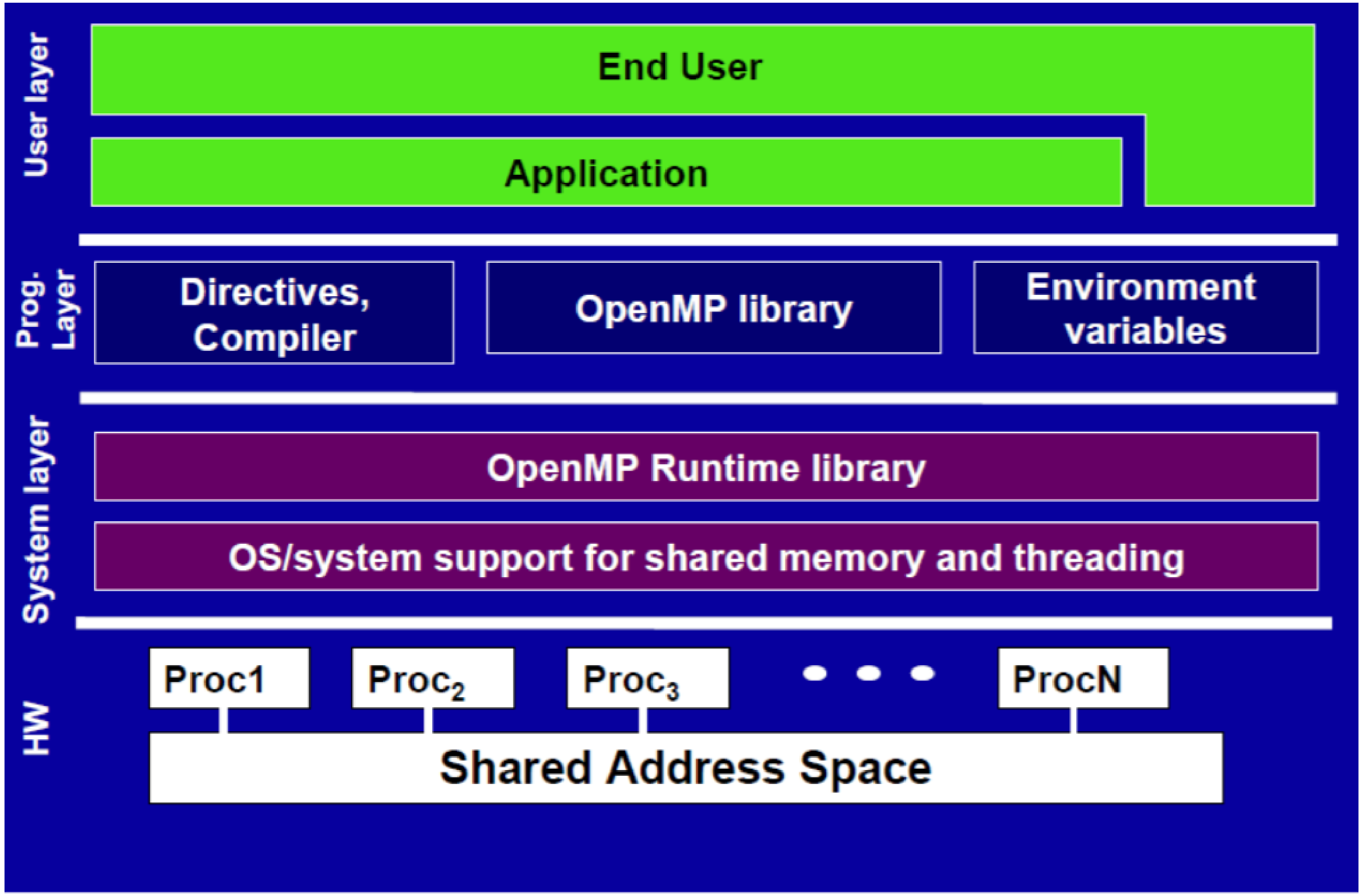
```
…
  /* determine partner and then send/receive with partner */
  if (taskid < numtasks/2) {
    partner = numtasks/2 + taskid;
    MPI_Send(&taskid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD);
    MPI_Recv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &status);
    }
  else if (taskid >= numtasks/2) {
    partner = taskid - numtasks/2;
    MPI_Recv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &status);
    MPI_Send(&taskid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD);
    }

  /* print partner info and exit*/
  printf("Task %d is partner with %d\n",taskid,message);
  }
…
```

Table of Contents

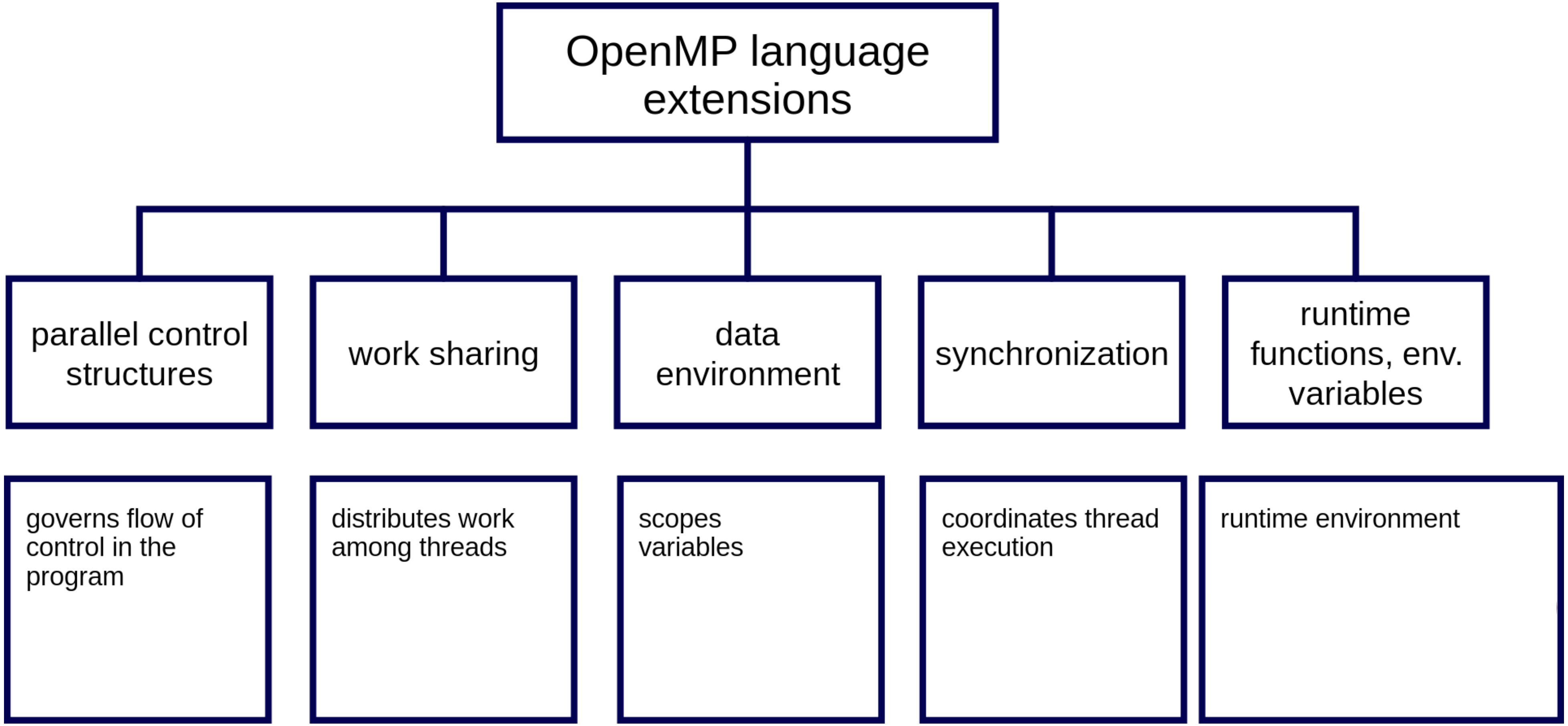
OpenMP

- OpenMP: open source API that supports shared memory multiprocessing programming
  - From OpenMP-1 (end of ’90s) to version 5.0 (released on 2018)
- Directives
  - Additional information for the compiler: how to process directive annotated code
  - **#pragma omp** directive [clause [clause …]]
  - Clause example: data scoping
    - Private: each thread has its own copy of the variable
    - Shared: threads share a single copy of the specified variable
    - …
- Format of Runtime Library Routines & Environment Variables
  - omp_... & OMP_...

OpenMP Components



OpenMP key extensions



Example: Hello World

```
int main() {
    …
    printf(buf, "Hello!\n");
    …
}
```

Output

Hello!

Parallel Control Structures

- GCC flag: **"-fopenmp"** → Enables handling of OpenMP directives
  - gcc –fopenmp -o hello hello.c

```
#include <omp.h>
int main() {
    …
    #pragma omp parallel
    {
        printf(buf, "Hello!\n");
    }
    …
}
```

Directive

Parallel Region

Runtime Environment

- Environment variables
  - OMP_NUM_THREADS
    - E.g.: export OMP_NUM_THREADS = 4

```
#include <omp.h>
int main() {
    …
    #pragma omp parallel
    {
        printf(buf, “Hello!\n”);
    }
    …
}
```

Output

Hello!
Hello!
Hello!
Hello!

Runtime Functions

- Runtime Library Routines
  - omp_get_thread_num()
    - Returns the ID of the current thread

```
#include <omp.h>
int main() {
    …
    #pragma omp parallel num_threads(4)
    {
        thid = omp_get_thread_num();
        printf(buf, “Hello %d!\n”, thid);
    }
    …
}
```

Output

Hello 0!
Hello 3!
Hello 1!
Hello 2!

Work Sharing

- Parallelizing regions (e.g. for, task, sections,…)

```
#include <omp.h>
int main() {
    …
    #pragma omp parallel
    {
        #pragma omp single
        {
            numthreads = omp_get_num_threads();
        }
        thid = omp_get_thread_num();
        #pragma omp for
        for (i=0; i<10; i++)
            printf(buf, “Hello i %d th %d!\n”, i, thid);
    }
    …
}
```

Output

Hello i 0 th 0!
Hello i 1 th 0!
Hello i 2 th 1!
Hello i 3 th 1!
Hello i 4 th 1!
Hello i 5 th 2!
Hello i 6 th 2!
Hello i 7 th 2!
Hello i 8 th 3!
Hello i 9 th 3!

Data Environment

- **Private**: Each thread has its own copy of the data
  - Other threads cannot access this data
  - Changes are only visible to the thread owning the data
  - By default, the loop iteration counters are private
- **Shared**: Threads share a single copy of the data
  - Other threads can access this data
  - Threads can **read and write** the data **simultaneously**
  - By default, all variables in a work sharing region are shared except the loop iterator
- **Default (shared|none)**: explicitly determines the default data sharing

- The default data scoping is … depends 🙄

- Utilization
  - **#pragma omp parallel shared(x,y) private(thid)**

Synchronization Mechanisms

- Mutex (MUTual EXclusion)
  - The running thread waits till the lock (mutex) is avilable to get its ownership
- Barriers
  - A thread waits for other threads to reach a given point
- Hardware support
  - Atomically read and modify a memory location
- Semaphores
  - A common resource can be accessed by multiple threads
- Spinlocks
  - Active checking for a lock

Some Synchronization Clauses

- **Critical**
  - Restricts execution of the associated structured block to a single thread at a time

    #pragma omp critical [(name) [hint (hint-expression)]]
    structured-block
- **Barrier**
  - Specifies an explicit barrier at the point at which the construct appears

    #pragma omp barrier
- **Taskwait**
  - Specifies a wait on the completion of **child tasks** of the current **task**

    #pragma omp taskwait
- **Atomic**
  - Ensures that a specific storage location is accessed atomically

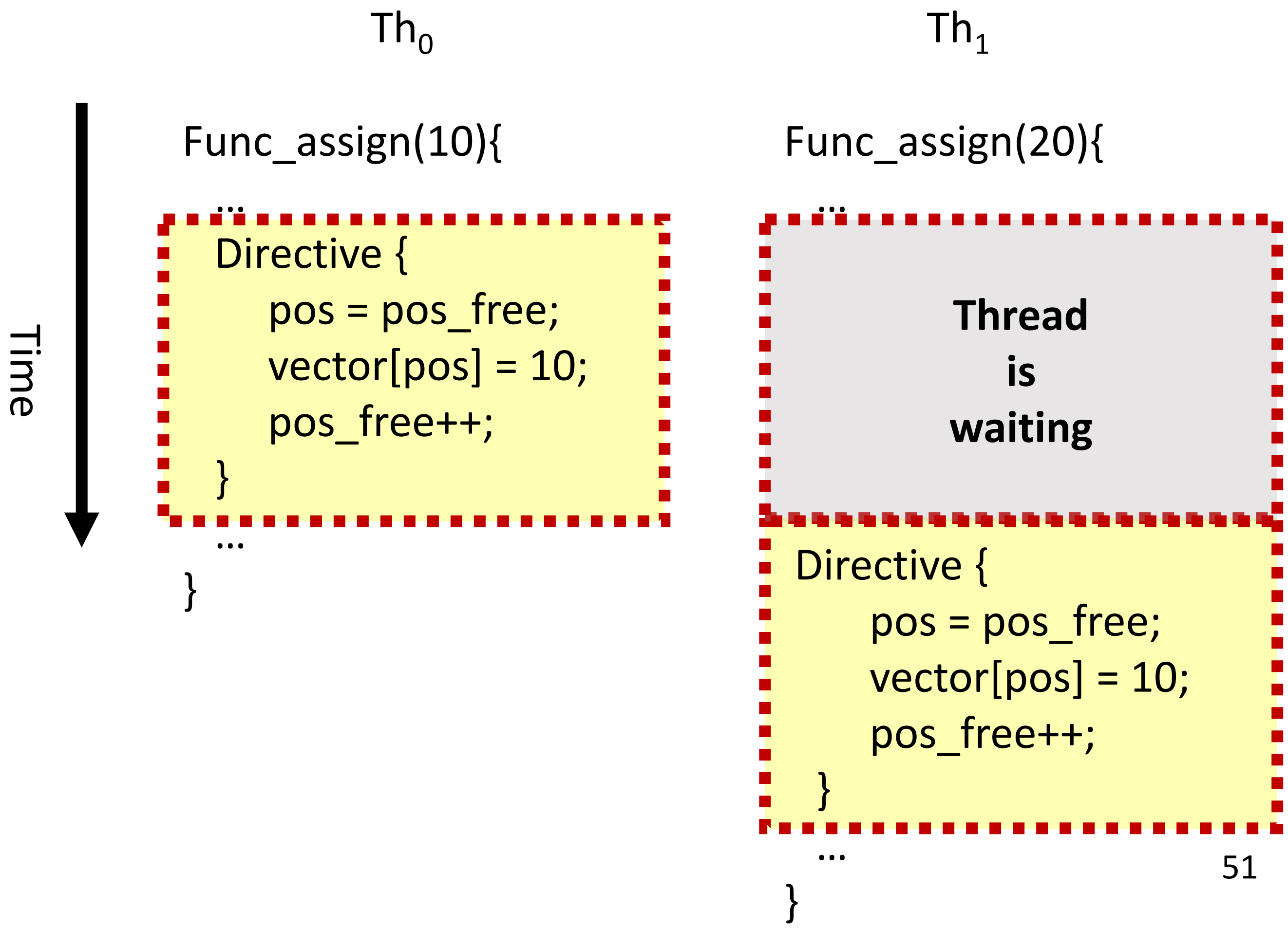
    #pragma omp atomic
    expression

Synchronization: Mutual Exclusion

- Only a single thread can be in a critical section at a time
- Utilization

  Directive [lock name]

- Limitations
  - It is not exception safe
  - It is not allowed to break out
  - All criticals wait for each other
    - Lock_name helps
- Alternatives
  - Lock based alternatives

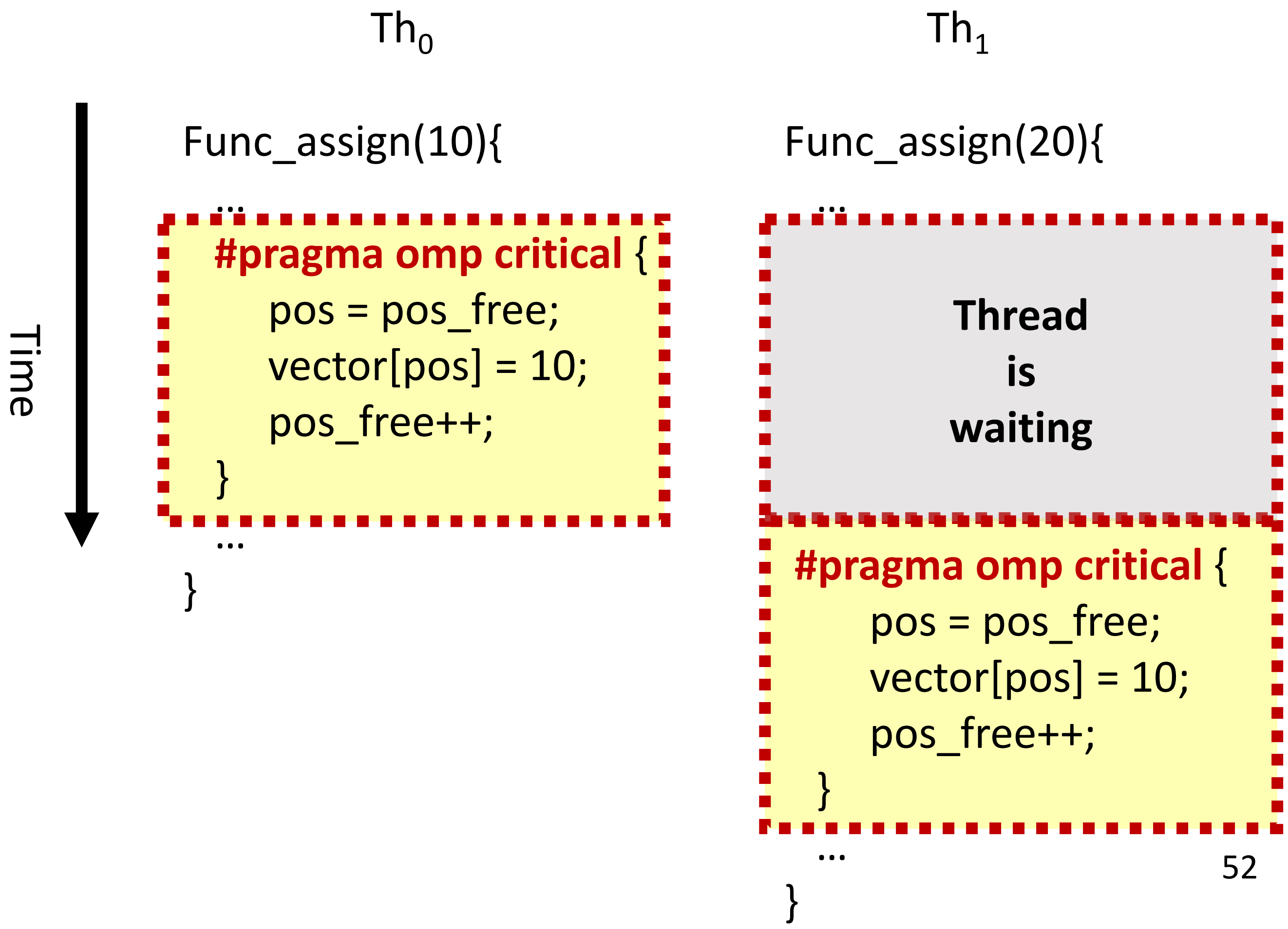


Synchronization: Mutual Exclusion

- Only a single thread can be in a critical section at a time
- Directive **Critical**
- Utilization

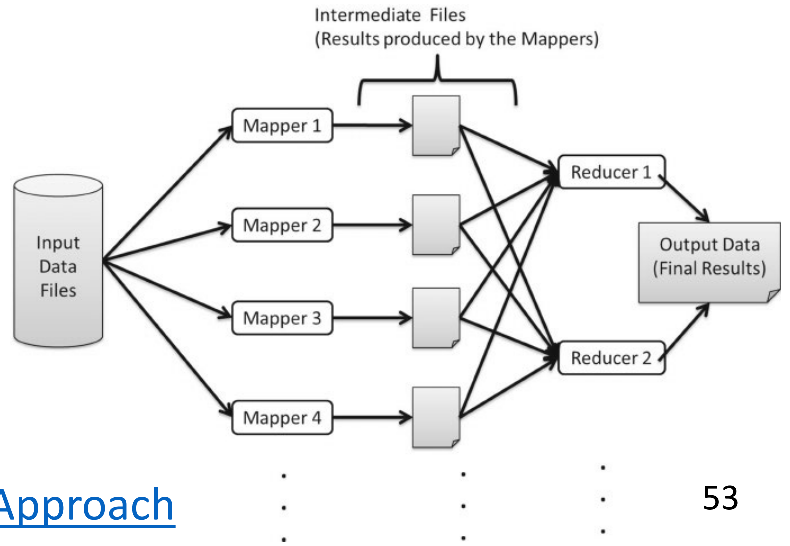
  #pragma omp critical [lock name]

- Limitations
  - It is not exception safe
  - It is not allowed to break out
  - All criticals wait for each other
    - Lock_name helps
- Alternatives
  - Lock based alternatives



Other Parallel Programming Models such as…

- CUDA (for [GPGPU])
  - NVIDIA's general purpose parallel computing architecture & programming model
    - GPU accelerates applications other tan 3D graphics
  - Scale code to hundreds of cores running thousands of threads
    - E.g.: 1 Tesla: 240 cores; 128 threads per core → 30720 threads total
- MapReduce
  - Suitable for data-intensive parallel processing
  - Data is partitioned across the cluster in a distributed file system
  - Move computation to data and compute across nodes in parallel



A Scalable Expressive Ensemble Learning Using Random Prism: A MapReduce Approach

Bibliography

- OpenMPI
  - https://www.open-mpi.org/
- OpenMP
  - https://www.openmp.org
  - OpenMP 5 reference guide C/C++ summary
    - https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-111802-web.pdf