

# HPC 101

Arnaud LEGRAND, CR CNRS, LIG/INRIA/Mescal

Vincent DANJEAN, MCF UJF, LIG/INRIA/Moais

October, 15th 2012

## Goals of the two next lectures

Learn and understand low-level software in HPC

Understand the internal of HPC programming model implementations

Limitation of mixing HPC programming models

# Hardware in HPC

- 2 Computational units
  - Parallel Machines with Shared Memory
  - Parallel Machines with Distributed Memory
  - Current Architectures in HPC
  
- 3 Networks
  - (Fast|Giga)-Ethernet
  - Legacy hardware
  - Current networking hardware
  
- 4 Summary

# Low-level software primitives in HPC

- 5 basic programming models for computational units
  - Hardware support for synchronization
  - Threads
- 6 Low-level communication interface/libraries
  - BIP and MX/Myrinet
  - SiSCI/SCI
  - VIA
- 7 Classical low-level techniques for efficient communications
  - Interacting with the network card: PIO and DMA
  - Zero-copy communications
  - Handshake Protocol
  - OS Bypass
- 8 Summary of low-level software primitives in HPC

# Low-level API in HPC

- 9 Synchronization
  - Semaphores
  - Monitors
  
- 10 PThread
  - Normalization of the threads interface
  - Basic POSIX Thread API
  
- 11 MPI
  - Message Passing
  - Introduction to MPI
  - Point-to-Point Communications
  - Collective Communications

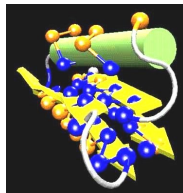
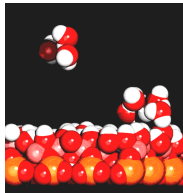
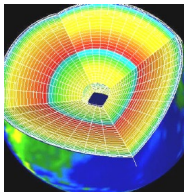
# Part I

## Hardware in HPC



# High-Performance Computing

- Simulation complements theory and experiments
  - Climatology, seismology, astrophysics, nano-sciences, material chemistry, molecular biology, ...

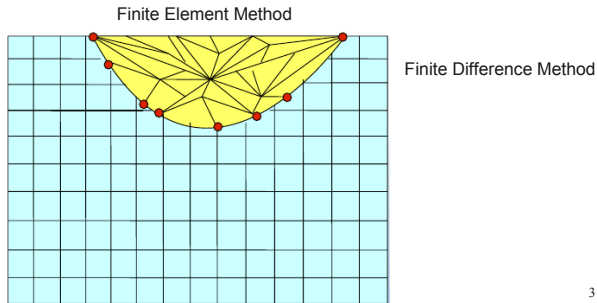


- Computation needs are always higher
  - Faster or better results



## Irregular applications

- Multi-scale simulation
- Code coupling

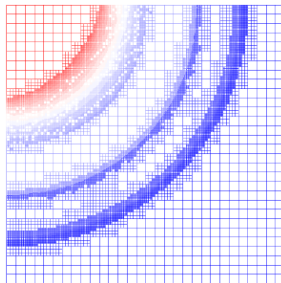
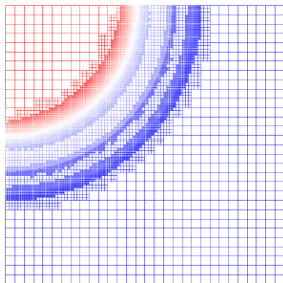






## Irregular applications

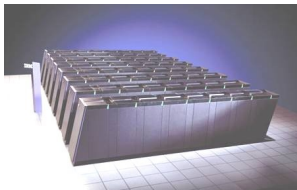
- Adaptive Mesh Refinement (AMR)
  - Behavior not known *a priori*



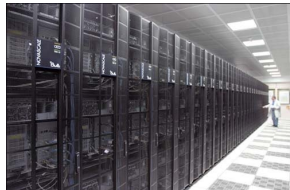


## Towards more and more hierarchical computers

- Landscape has changed
  - From super-computers to clusters
  - With more and more parallelism



Blue Gene, 106,496 cores



Tera10, 8704 cores

# High Performance Computing

## Needs are always here

- numerical or financial simulation, modelisation, virtual reality virtuelle
- more data, more details, . . .

Computing power will never be enough

## One way to follow: using parallelism

**Idea: change space into time**

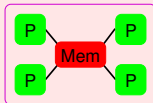
more resources to gain some time

# Parallel Architectures

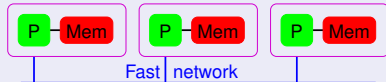
## Two main kinds

Architectures with shared memory and architectures with distributed memory.

### Multiprocessors



### Clusters



# Outlines: Hardware in HPC

- 2 Computational units
  - Parallel Machines with Shared Memory
  - Parallel Machines with Distributed Memory
  - Current Architectures in HPC
- 3 Networks
- 4 Summary

## Why several processors/cores ?

### Limits for monocoressors

- superscalar processors: instruction level parallelism
- frequency
- electrical power

### What to do with place available on chips ?

- caches (bigger and quicker)
- several series of registers (hyperthreaded processors)
- several series of cores (multi-core processors)
- all of that

# Symmetric Multi Processors

- all processors have access to the same memory and I/O
- in case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors
- rarely used nowadays but on processors with few cores (2 or 4)

## Non Uniform Memory Access Architectures

- memory access time depends on the memory location relative to a processor
- better scaling hardware architecture
- harder to program efficiently: trade off needed between load-balancing and memory data locality

# Clusters

## Composed of a few to hundreds of machines

- often homogeneous
  - same processor, memory, etc.
- often linked with a high speed, low latency network
  - Myrinet, InfinityBand, Quadrix, etc.

## Biggest clusters can be split in several parts

- computing nodes
- I/O nodes
- front (interactive) node



# Grids

## Lots of heterogeneous resources

- aggregation of clusters and/or standalone nodes
- high latency network (Internet for example)
- often dynamic resources (clusters/nodes appear and disappear)
- different architectures, networks, etc.

# Computational units

## Hierarchical Architectures

- HT technology
- multi-core processor
- multi processors machine
- cluster of machines
- grid of clusters and individual machines

## Even more complexity

- computing on GPU
  - require specialized codes but hardware far more powerful
- FPGA
  - hardware can be specialized on demand
  - still lots of work on interface programming here

# Outlines: Hardware in HPC

- 2 Computational units
- 3 Networks**
  - (Fast|Giga)-Ethernet
  - Legacy hardware
  - Current networking hardware
- 4 Summary

# High Speed Networks

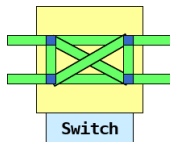
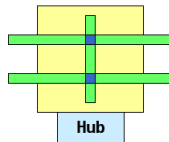
## High Speed Networks are used in clusters

- low distance
- very interesting performance
  - low latency: about  $1 \mu\text{s}$
  - high bandwidth: about 10 Gb/s and more
- specific light protocols
  - static routing of messages
  - no required packet fragmentation
  - sometimes, no packet required

Myrinet, Quadrics, SCI, . . .

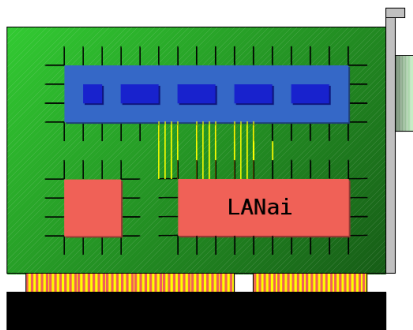
# (Fast|Giga)-Ethernet

- Interconnect:
  - Hub or switch
- Wires:
  - Copper or optical fiber
- Latency:
  - about  $10 \mu\text{s}$
- Bandwidth:
  - From 100 Mb/s to 10 Gb/s (100 Gb/s, june 2010)
- Remark:
  - compatible with traditional Ethernet



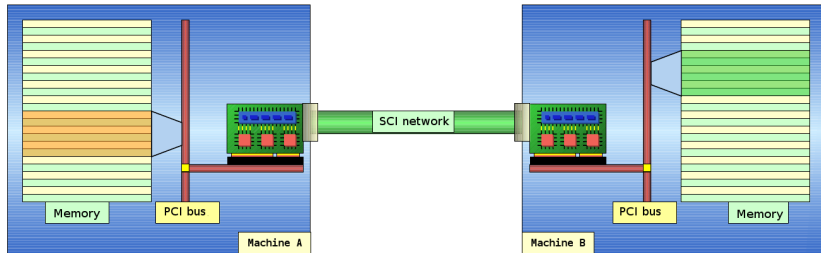
# Myrinet

- Myricom corporate
- Interconnect:
  - Switch
- PCI card with:
  - a processor: LANai
  - SRAM memory: about 4 MB
- Latency:
  - about 1 or 2  $\mu$ s
- Bandwidth:
  - 10 Gb/s
- Remark:
  - static, wormhole routing
  - can you RJ45 cables



# SCI

- Scalable Coherent Interface
  - IEEE norm (1993)
  - Dolphin corporate
- Uses remote memory access:
  - Address space remotely mapped



# InfiniBand

- Several manufacturers (Cisco, HP, Intel, IBM, etc.)
- Interconnect:
  - Optical links
  - Serial, point-to-point connections
  - Switched fabric (possibility of several paths)
- Bandwidth
  - single line of 2, 4, 8, 14 or 25 Mb/s
  - possibility of bonding 4 or 12 lines
- Latency:
  - about 100 or 200 ns *for hardware only*
  - about 1 or 2  $\mu$ s for some hardware with its driver
- Remark:
  - can interconnect buildings
  - RDMA operations available



# Quadrics

- One manufacturer (Quadrics)
- Interconnect:
  - Bi-directional serial links
  - Switched fabric (possibility of several paths)
- Bandwidth
  - 1 to 2 Gb/s on each direction
- Latency:
  - about  $1.3 \mu\text{s}$  in MPI
- Remark:
  - selected by Bull for the fastest supercomputer in Europe: Tera100 at CEA
  - global operations (reduction, barrier) available in hardware

# Outlines: Hardware in HPC

- 2 Computational units
- 3 Networks
- 4 Summary**

# Hardware in HPC

## Performance, performance and performance

HPC machines: as many TFlops as possible for the user

- very complex multi-cores, multi-processors machines
- specialized hardware accelerators (GPU, FPGA, etc.)
- dedicated specialized networks

## Some limitations

- not the price
- but the power consumption  
dedicated EDF electrical line for the last French  
super-calculator
- and the physics  
an electric signal cannot go quicker than the light speed

basic programming models for computational units

Low-level communication interface/libraries

Classical low-level techniques for efficient communications

Summary of low-level software primitives in HPC

## Part II

# Low-level software primitives in HPC

## Outlines: Low-level software primitives in HPC

- 5 basic programming models for computational units
  - Hardware support for synchronization
  - Threads
- 6 Low-level communication interface/libraries
- 7 Classical low-level techniques for efficient communications
- 8 Summary of low-level software primitives in HPC

# Synchronization requires hardware support

What happens with incrementations in parallel?

```
var++;
```

```
var++;
```

# Synchronization requires hardware support

What happens with incrementations in parallel?

```
for (i=0; i<10; i++){ | for (i=0; i<10; i++){  
    var++;              |    var++;  
}                       | }
```

# Synchronization requires hardware support

## What happens with incrementations in parallel?

```
for (i=0; i<10; i++){ | for (i=0; i<10; i++){  
    var++;              |    var++;  
}                       | }
```

## Hardware support required

**TAS** atomic test and set instruction

**cmpexchge** compare and exchange

**atomic operation** incrementation, decrementation, adding, etc.



# Critical section with busy waiting

## Example of code

```
while (TAS(&var))  
    ;  
/* in critical section */  
var=0;
```

# Critical section with busy waiting

## Example of code

```
while (TAS(&var))  
    while (var) ;  
/* in critical section */  
var=0;
```

# Critical section with busy waiting

## Example of code

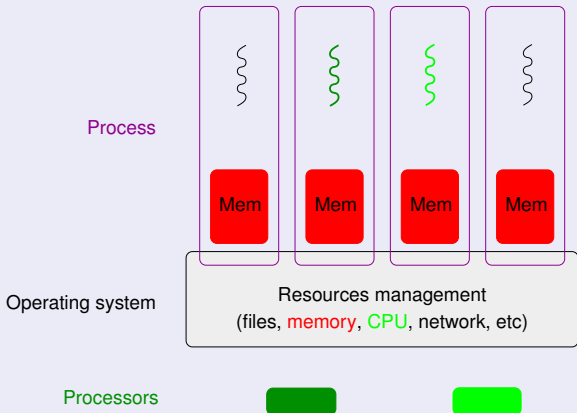
```
while (TAS(&var))  
    while (var) ;  
/* in critical section */  
var=0;
```

## Busy waiting

- + very reactive
- + no OS or lib support required
- use a processor while not doing anything
- does not scale if there are lots of waiters
- very difficult to do it correctly (compiler, processor, etc.)

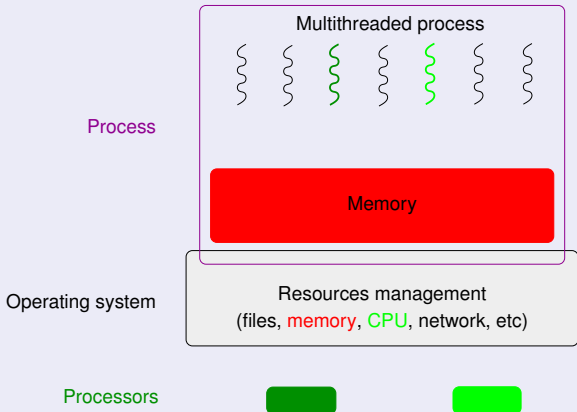
# Programming on Shared Memory Parallel Machines

## Using process



# Programming on Shared Memory Parallel Machines

## Using threads



# basic programming models for computational units

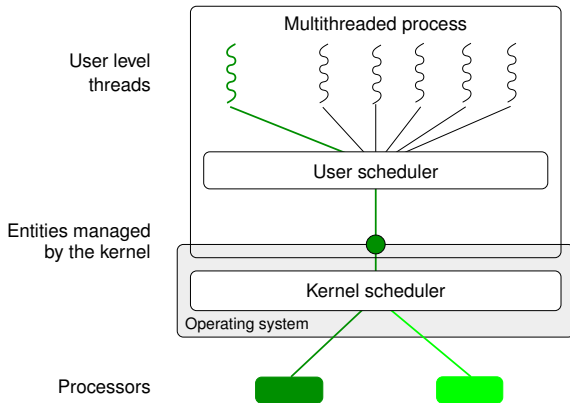
## Why threads ?

- To take profit from shared memory parallel architectures
  - SMP, hyperthreaded, multi-core, NUMA, etc. processors
  - future Intel processors: several hundreds cores
- To describe the parallelism within the applications
  - independent tasks, I/O overlap, etc.

## What will use threads ?

- User application codes
  - directly (with thread libraries)
    - POSIX API (IEEE POSIX 1003.1c norm) in C, C++, ...
  - with high-level programming languages (Ada, OpenMP, ...)
- Middleware programming environments
  - demonized tasks (garbage collector, ...), ...

# User threads



Efficiency



Flexibility



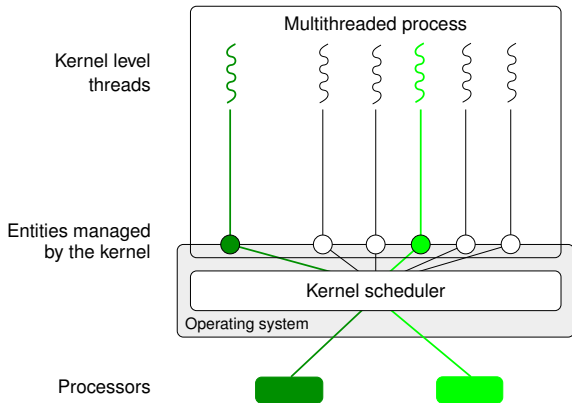
SMP



Blocking syscalls



# Kernel threads



Efficiency



Flexibility



SMP

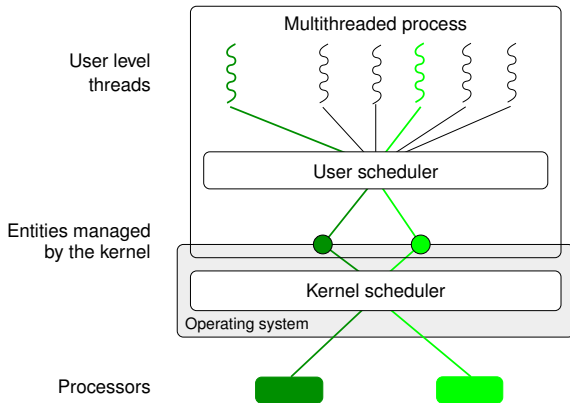


Blocking syscalls





# Mixed models



Efficiency



Flexibility



SMP



Blocking syscalls limited

# Thread models characteristics

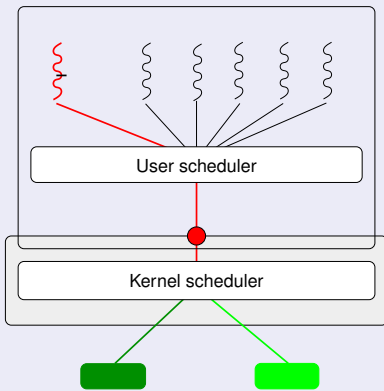
Library	Characteristics			
	Efficiency	Flexibility	SMP	Blocking syscalls
User	+	+	-	-
Kernel	-	-	+	+
Mixed	+	+	+	limited

## Summary

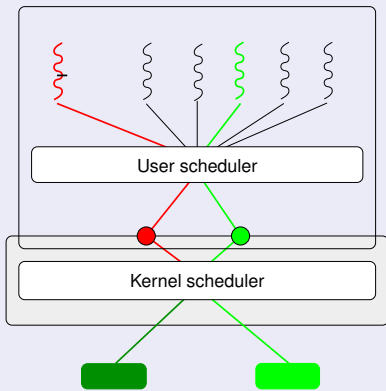
Mixed libraries seems more attractive however they are more complex to develop. They also suffer from the blocking system call problem.

# User Threads and Blocking System Calls

User level library



Mixed library



# Scheduler Activations

Idea proposed by Anderson et al. (91)

Dialogue (and not monologue) between the user and kernel schedulers

- the user scheduler uses system calls
- **the kernel scheduler uses upcalls**

## Upcalls

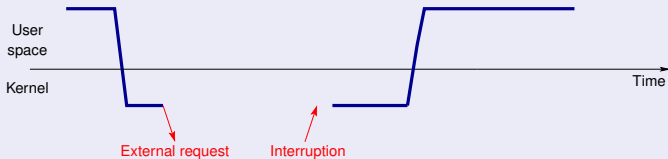
Notify the application of scheduling kernel events

## Activations

- a new structure to support upcalls  
a kind of **kernel thread** or **virtual processor**
- creating and destruction managed by the kernel

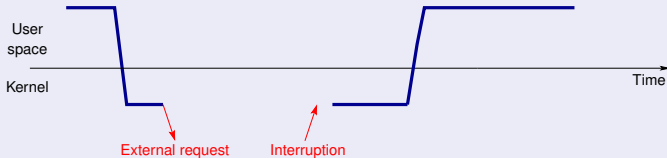
# Scheduler Activations

Instead of:

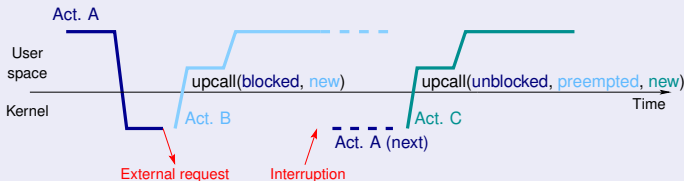


# Scheduler Activations

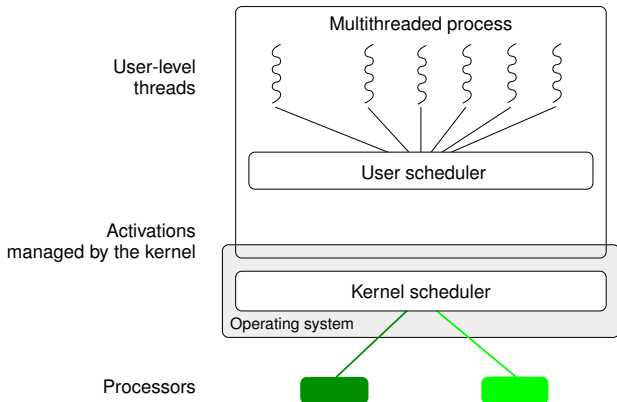
Instead of:



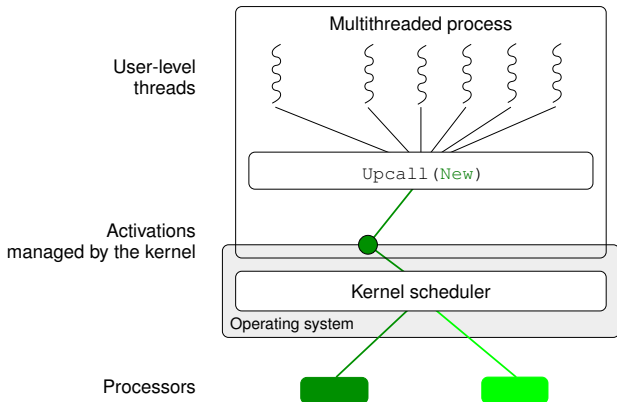
...better use the following schema:



# Working principle

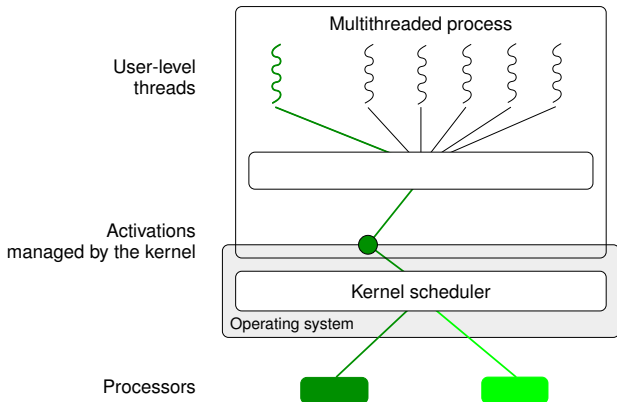


# Working principle

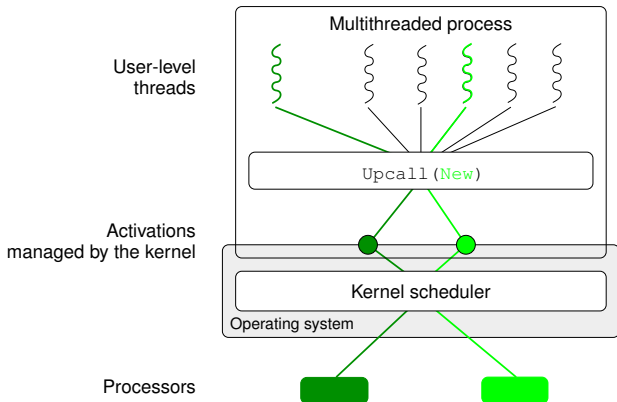




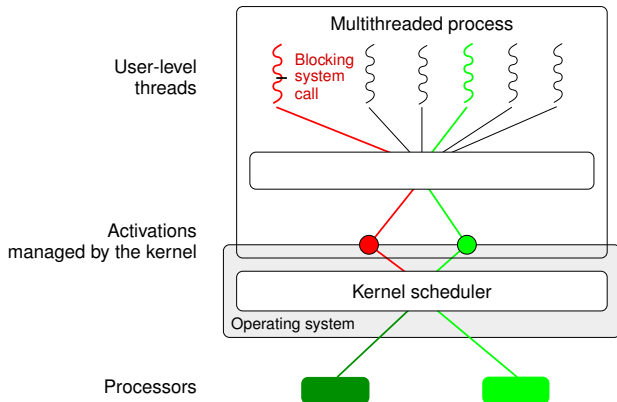
# Working principle



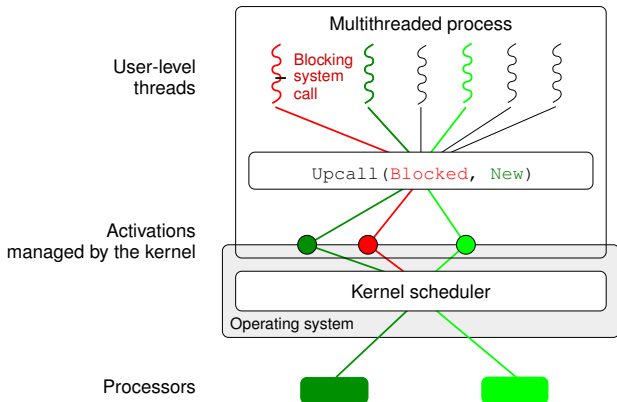
# Working principle



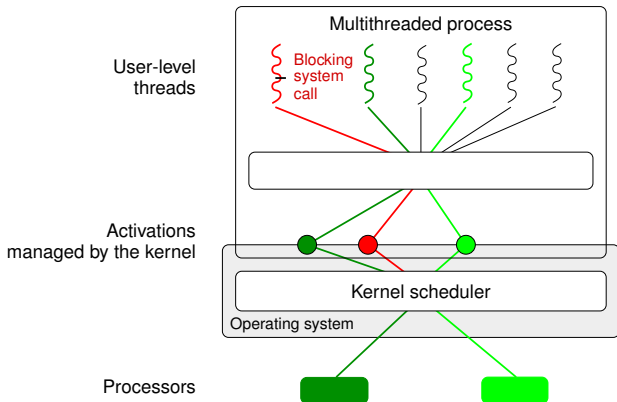
# Working principle



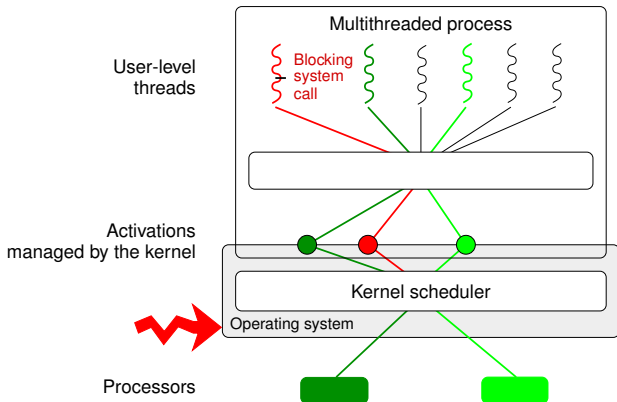
# Working principle



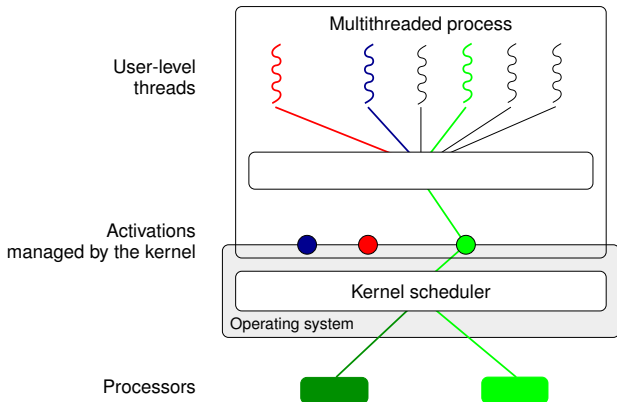
# Working principle



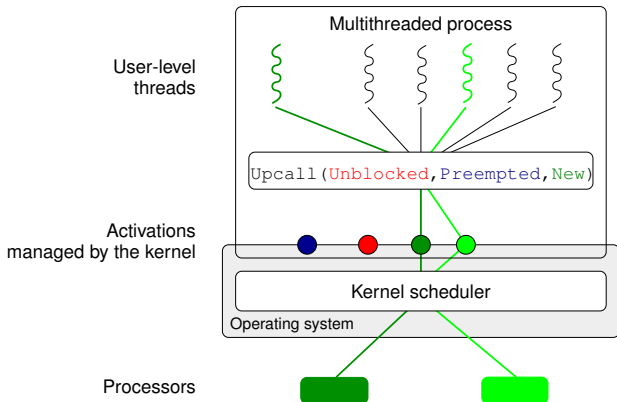
# Working principle



# Working principle



# Working principle





# Exploiting computational resources

## Some recurring patterns

- hardware synchronization primitives
- threads

## Common feature: very hard to program correctly such hardware

One cannot expect HPC applications to handle directly these kind of resources. It is too complex for an scientific application programmer.

# Outlines: Low-level software primitives in HPC

- 5 basic programming models for computational units
- 6 Low-level communication interface/libraries**
  - BIP and MX/Myrinet
  - SiSCI/SCI
  - VIA
- 7 Classical low-level techniques for efficient communications
- 8 Summary of low-level software primitives in HPC

# BIP/Myrinet

- Basic Interface for Parallelism
  - L. Prylli and B. Tourancheau
- Dedicated to Myrinet networks
- Characteristics
  - Asynchronous communication
  - No error detection
  - No flow control
    - Small messages are copied into a fixed buffer at reception
    - Big messages are lost if the receiver is not ready

# MX/Myrinet

- Myrinet eXpress
  - Official driver from Myricom
- Very simplistic interface to allow easy implementation of MPI
  - Flow control
  - Reliable communications
  - Non contiguous messages
  - Multiplexing

# SiSCI/SCI

- Driver for SCI cards
- Programming model
  - Remote memory access
    - Explicit: RDMA
    - Implicit: memory projections
- Performance
  - Explicit use of some operation required:
    - memory “flush”
    - `SCI_memcpy`
    - RDMA

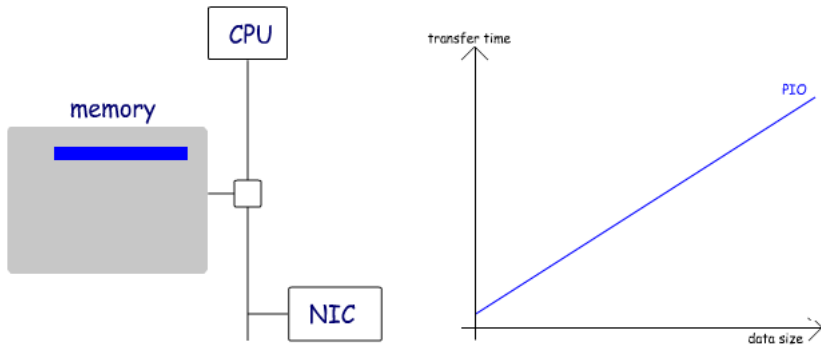
# VIA

- Virtual Interface Architecture
- A new standard
  - Lots of industrials
    - Microsoft, Intel, Compaq, etc.
  - Use for InfiniBand networks
- Characteristics
  - Virtual interfaces objects
    - Queues of descriptors (for sending and receiving)
  - Explicit memory recording
  - Remote reads/writes
    - RDMA

# Outlines: Low-level software primitives in HPC

- 5 basic programming models for computational units
- 6 Low-level communication interface/libraries
- 7 Classical low-level techniques for efficient communications**
  - Interacting with the network card: PIO and DMA
  - Zero-copy communications
  - Handshake Protocol
  - OS Bypass
- 8 Summary of low-level software primitives in HPC

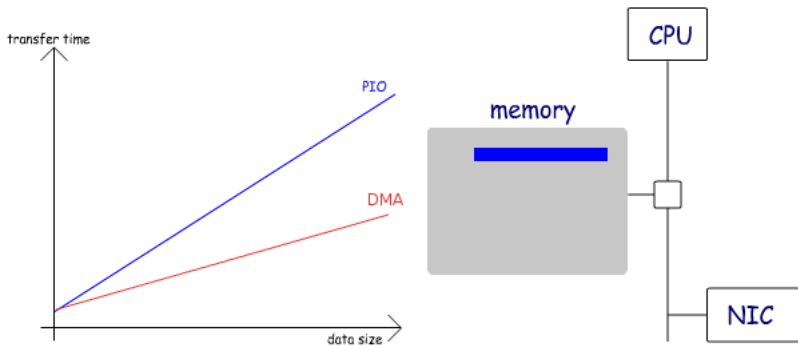
## Interacting with the network card: PIO mode



**Programmed Input/Output**



## Interacting with the network card: DMA mode



**Direct Memory Access**

# Zero-copy communications

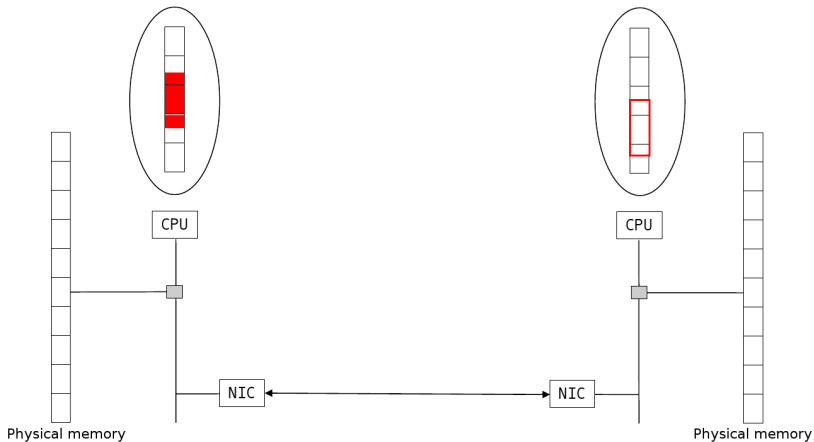
## Goals

- Reduce the communication time
  - Copy time cannot be neglected
    - but it can be partially recovered with pipelining
- Reduce the processor use
  - currently, `memcpy` are executed by processor instructions

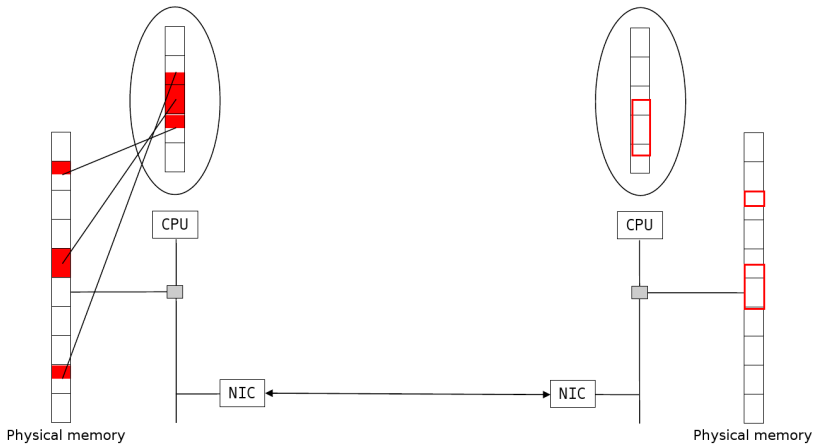
## Idea

The network card directly read/write data from/to the application memory

# Zero-copy communications



# Zero-copy communications



# Zero-copy communications for emission

## PIO mode transfers

- No problem for zero-copy

## DMA mode transfers

- Non contiguous data in physical memory
- Headers added in the protocol
  - linked DMA
  - limits on the number of non contiguous segments

## Zero-copy communications for reception

A network card cannot “freeze” the received message on the physical media

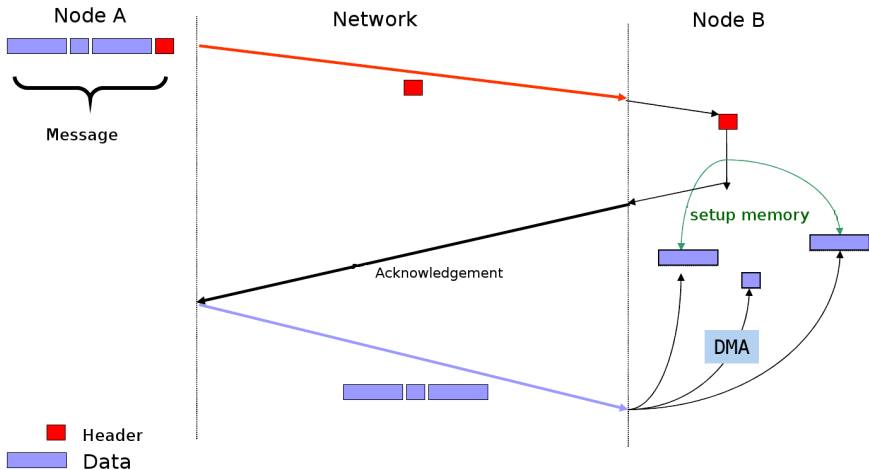
If the receiver posted a “recv” operation before the message arrives

- zero-copy OK if the card can filter received messages
- else, zero-copy allowed with bounded-sized messages with optimistic heuristics

If the receiver is not ready

- A handshake protocol must be setup for big messages
- Small messages can be stored in an internal buffer

# Using a Handshake Protocol



## A few more considerations

### The receiving side plays an important role

- Flow-control is mandatory
- Zero-copy transfers
  - the sender has to ensure that the receiver is ready
  - a handshake (REQ+ACK) can be used

### Communications in user-space introduce some difficulties

- Direct access to the NIC
  - most technologies impose “pinned” memory pages

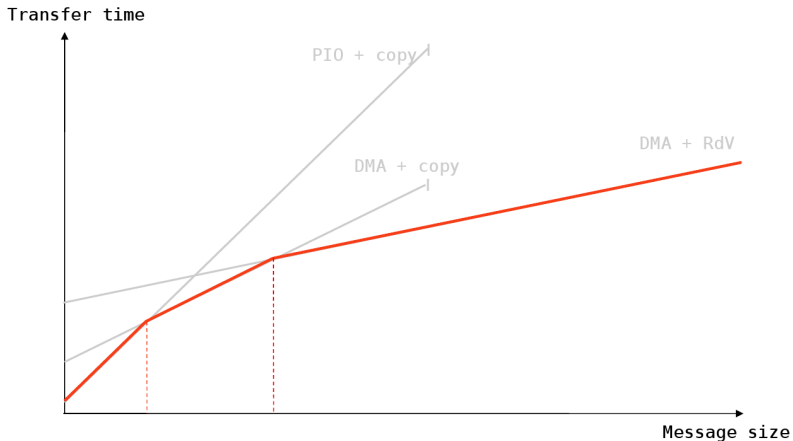
### Network drivers have limitations



# Communication Protocol Selection

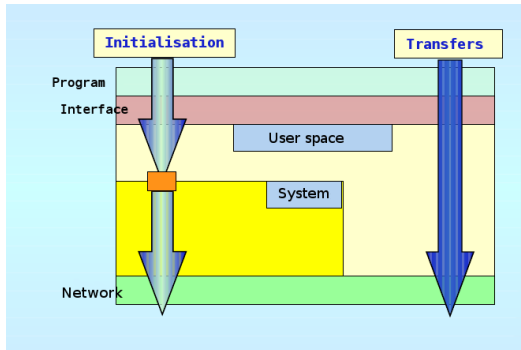


# Communication Protocol Selection



# Operating System Bypass

- Initialization
  - traditional system calls
  - only at session beginning
- Transfers
  - direct from user space
  - no system call
  - “less” interrupts
- Humm... And what about security ?

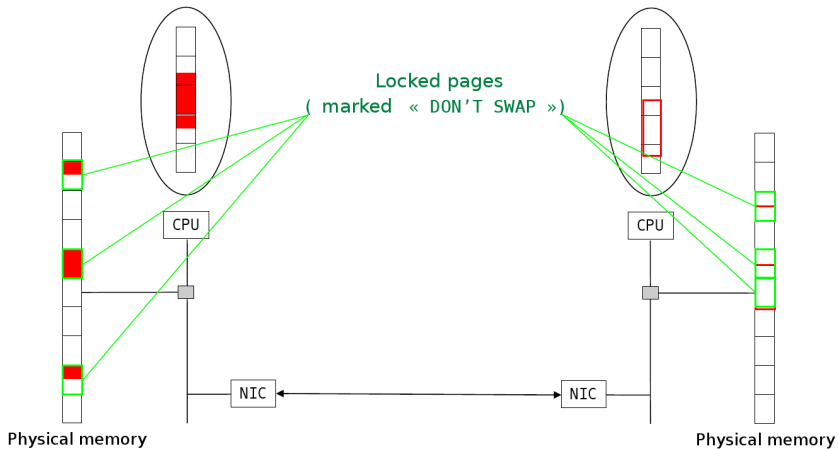


# OS-bypass + zero-copy

## Problem

- Zero-copy mechanism uses DMA that requires physical addresses
- Mapping between virtual and physical address is only known by:
  - the processor (MMU)
  - the OS (pages table)
- We need that
  - the library knows this mapping
  - this mapping is not modified during the communication
    - ex: swap decided by the OS, copy-on-write, etc.
- No way to ensure this in user space !

# OS-bypass + zero-copy



# OS-bypass + zero-copy

## First solution

- Pages “recorded” in the kernel to avoid swapping
- Management of a cache for virtual/physical addresses mapping
  - in user space or on the network card
- Diversion of system calls that can modify the address space

## Second solution

- Management of a cache for virtual/physical addresses mapping on the network card
- OS patch so that the network card is informed when a modification occurs
- Solution chosen by MX/Myrinet and Elan/Quadrics

## Direct consequences

- Latency measure can vary whether the memory region used
  - Some pages are “recorded” within the network card
- Ideal case are ping-pong exchanges
  - The same pages are reused hundred of times
- Worst case are applications using lots of different data regions. . .

## Outlines: Low-level software primitives in HPC

- 5 basic programming models for computational units
- 6 Low-level communication interface/libraries
- 7 Classical low-level techniques for efficient communications
- 8 Summary of low-level software primitives in HPC**



# Summary of low-level software primitives in HPC

## Efficient hardware

- numerous parallel threads
- very low latency and high bandwidth networks
- complex hardware to be programmed efficiently
  - synchronization, onboard CPU, onboard MMU for DMA, etc.

## Very specific programming interfaces

- specialized assembly instructions for synchronization
- dedicated to specific technologies (but VIA)
- different programming models
- quasi no portability

It is not reasonable to program a scientific application directly with such programming interfaces

## Part III

# Low-level API in HPC

# Outlines: Low-level API in HPC

- 9 Synchronization
  - Semaphores
  - Monitors
- 10 PThread
- 11 MPI

# Semaphores

- Internal state: a counter initialised to a positive or null value
- Two methods:
  - $P(s)$  wait for a positive counter then decrease it once
  - $V(s)$  increase the counter

## Common analogy: a box with tokens

- Initial state: the box has  $n$  tokens in it
- One can put one more token in the box (V)
- One can take one token from the box (P) waiting if none is available

# Monitors

## Mutex

- Two states: locked or not
- Two methods:
  - `lock(m)` take the mutex
  - `unlock(m)` release the mutex (must be done by the thread owning the mutex)

## Conditions

- waiting thread list (conditions are not related with tests)
- Three methods:
  - `wait(c, m)` sleep on the condition. The mutex is released atomically during the wait.
  - `signal(c)` one sleeping thread is wake up
  - `broadcast(c)` all sleeping threads are wake up

# Outlines: Low-level API in HPC

- 9 Synchronization
- 10 PThread
  - Normalization of the threads interface
  - Basic POSIX Thread API
- 11 MPI

# Normalisation of the thread interface

## Before the norm

- each Unix had its (slightly) incompatible interface
- but same kinds of features was present

## POSIX normalization

- IEEE POSIX 1003.1c norm (also called POSIX threads norm)
- Only the API is normalised (not the ABI)
  - POSIX thread libraries can easily be switched at source level but not at runtime
- POSIX threads own
  - processor registers, stack, etc.
  - signal mask
- POSIX threads can be of any kind (user, kernel, etc.)

# Basic POSIX Thread API

## Creation/destruction

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg)`
- `void pthread_exit(void *value_ptr)`
- `int pthread_join(pthread_t thread, void **value_ptr)`

## Synchronisation (semaphores)

- `int sem_init(sem_t *sem, int pshared, unsigned int value)`
- `int sem_wait(sem_t *sem)`
- `int sem_post(sem_t *sem)`
- `int sem_destroy(sem_t *sem)`



## Basic POSIX Thread API (2)

### Synchronisation (mutex)

- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)`
- `int pthread_mutex_lock(pthread_mutex_t *mutex)`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex)`
- `int pthread_mutex_destroy(pthread_mutex_t *mutex)`

### Synchronisation (conditions)

- `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)`
- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
- `int pthread_cond_signal(pthread_cond_t *cond)`

## Basic POSIX Thread API (3)

### Per thread data

- `int pthread_key_create(pthread_key_t *key, void (*destr_function) (void*))`
- `int pthread_key_delete(pthread_key_t key)`
- `int pthread_setspecific(pthread_key_t key, const void *pointer)`
- `void * pthread_getspecific(pthread_key_t key)`

## Basic POSIX Thread API (3)

### Per thread data

- `int pthread_key_create(pthread_key_t *key, void (*destr_function) (void*))`
- `int pthread_key_delete(pthread_key_t key)`
- `int pthread_setspecific(pthread_key_t key, const void *pointer)`
- `void * pthread_getspecific(pthread_key_t key)`

### The new `__thread` C keyword

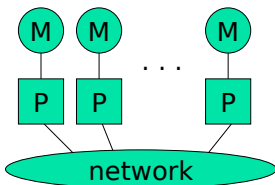
- used for a global per-thread variable
- need support from the compiler and the linker at compile time and execute time
- libraries can have efficient per-thread variables without disturbing the application

# Outlines: Low-level API in HPC

- 9 Synchronization
- 10 PThread
- 11 MPI
  - Message Passing
  - Introduction to MPI
  - Point-to-Point Communications
  - Collective Communications



# Message Passing



- Each processor runs a process
  - Processes communicate by exchanging messages
  - They cannot share memory in the sense that they cannot address the same memory cells
- 
- The above is a programming model and things may look different in the actual implementation (e.g., MPI over Shared Memory)
  - **Message Passing is popular because it is general:**
    - Pretty much any distributed system works by exchanging messages, at some level
    - Distributed- or shared-memory multiprocessors, networks of workstations, uniprocessors
  - **It is not popular because it is easy (it's not)**



# Code Parallelization

---

- Shared-memory programming
  - Parallelizing existing code can be very easy
    - OpenMP: just add a few pragmas
    - Pthreads: wrap work in `do_work` functions
  - Understanding parallel code is easy
  - Incremental parallelization is natural
- Distributed-memory programming
  - parallelizing existing code can be very difficult
    - No shared memory makes it impossible to “just” reference variables
    - Explicit message exchanges can get really tricky
  - Understanding parallel code is difficult
    - Data structured are split all over different memories
  - Incremental parallelization can be challenging



# Programming Message Passing

---

- Shared-memory programming is simple conceptually (sort of)
- Shared-memory machines are expensive when one wants a lot of processors
- It's cheaper (and more scalable) to build distributed memory machines
  - Distributed memory supercomputers (IBM SP series)
  - Commodity clusters
- But then how do we program them?
- At a basic level, let the user deal with explicit messages
  - difficult
  - but provides the most flexibility



# Message Passing

---

- Isn't exchanging messages completely known and understood?
  - That's the basis of the IP idea
  - Networked computers running programs that communicate are very old and common
    - DNS, e-mail, Web, ...
- The answer is that, yes it is, we have "Sockets"
  - Software abstraction of a communication between two Internet hosts
  - Provides an API for programmers so that they do not need to know anything (or almost anything) about TCP/IP and write code with programs that communicate over the internet





## Using Sockets for parallel programming?

---

- One could think of writing all parallel code on a cluster using sockets
  - n nodes in the cluster
  - Each node creates n-1 sockets on n-1 ports
  - All nodes can communicate
- Problems with this approach
  - Complex code
  - Only point-to-point communication
  - No notion of types messages
  - But
    - All this complexity could be “wrapped” under a higher-level API
    - And in fact, we’ll see that’s the basic idea
  - **Does not take advantage of fast networking within a cluster/ MPP**
    - Sockets have “Internet stuff” in them that’s not necessary
    - TPC/IP may not even be the right protocol!



## Message Passing for Parallel Programs

---

- Although “systems” people are happy with sockets, people writing parallel applications need something better
  - easier to program to
  - able to exploit the hardware better within a single machine
- This “something better” right now is MPI
  - We will learn how to write MPI programs
- Let’s look at the history of message passing for parallel computing

# Outlines: Low-level API in HPC

- 9 Synchronization
- 10 PThread
- 11 MPI
  - Message Passing
  - Introduction to MPI
  - Point-to-Point Communications
  - Collective Communications



# The MPI Standard

---

- MPI Forum setup as early as 1992 to come up with a de facto standard with the following goals:
  - source-code portability
  - allow for efficient implementation (e.g., by vendors)
  - support for heterogeneous platforms
- MPI is not
  - a language
  - an implementation (although it provides hints for implementers)
- June 1995: MPI v1.1 (we're now at MPI v1.2)
  - <http://www-unix.mcs.anl.gov/mpi/>
  - C and FORTRAN bindings
  - We will use MPI v1.1 from C in the class
- Implementations:
  - well-adopted by vendors
  - free implementations for clusters: MPICH, LAM, CHIMP/MPI
  - research in fault-tolerance: MPICH-V, FT-MPI, MPIFT, etc.



# SPMD Programs

---

- It is rare for a programmer to write a different program for each process of a parallel application
- In most cases, people write Single Program Multiple Data (SPMD) programs
  - the same program runs on all participating processors
  - processes can be identified by some *rank*
  - This allows each process to know which piece of the problem to work on
  - This allows the programmer to specify that some process does something, while all the others do something else (common in master-worker computations)

```
main(int argc, char **argv) {  
    if (my_rank == 0) { /* master */  
        ... load input and dispatch ...  
    } else { /* workers */  
        ... wait for data and compute ...  
    }  
}
```

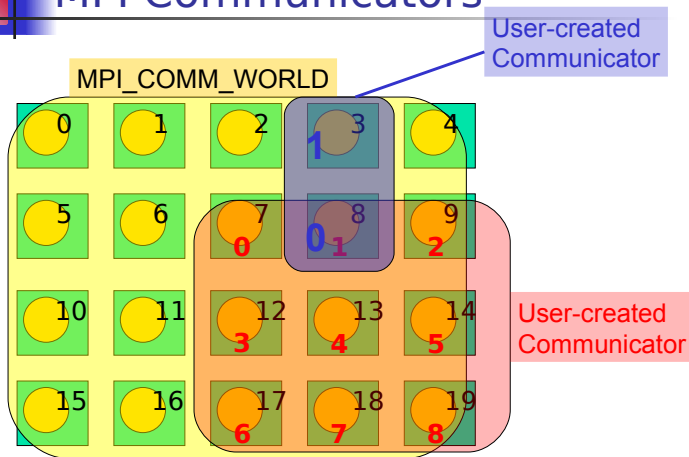
The logo consists of a vertical line and a horizontal line intersecting at the center. The vertical line is black, and the horizontal line is grey. There are four colored squares: a yellow square in the top-left, a red square in the top-right, a blue square in the bottom-left, and a blue square in the bottom-right.

# MPI Concepts

---

- Fixed number of processors
  - When launching the application one must specify the number of processors to use, which remains unchanged throughout execution
- Communicator
  - Abstraction for a group of processes that can communicate
  - A process can belong to multiple communicators
  - Makes is easy to partition/organize the application in multiple layers of communicating processes
  - Default and global communicator: ***MPI\_COMM\_WORLD***
- Process Rank
  - The index of a process within a communicator
  - Typically user maps his/her own virtual topology on top of just linear ranks
    - ring, grid, etc.

# MPI Communicators





# A First MPI Program

```
#include <unistd.h>
#include <mpi.h>
int main(int argc, char **argv) {
    int my_rank, n;
    char hostname[128];
    MPI_init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n);
    gethostname(hostname, 128);
    if (my_rank == 0) { /* master */
        printf("I am the master: %s\n", hostname);
    } else { /* worker */
        printf("I am a worker: %s (rank=%d/%d)\n",
              hostname, my_rank, n-1);
    }
    MPI_Finalize();
    exit(0);
}
```

Has to be called first, and once

Has to be called last, and once





# Compiling/Running it

- Compile with `mpicc`
- Run with `mpirun`
  - `% mpirun -np 4 my_program <args>`
    - requests 4 processors for running `my_program` with command-line arguments
    - see the `mpirun` man page for more information
    - in particular the `-machinefile` option that is used to run on a network of workstations
- Some systems just run all programs as MPI programs and no explicit call to `mpirun` is actually needed
- Previous example program:

```
% mpirun -np 3 -machinefile hosts my_program
I am the master: somehost1
I am a worker: somehost2 (rank=2/2)
I am a worker: somehost3 (rank=1/2)
```

(stdout/stderr redirected to the process calling mpirun)

# Outlines: Low-level API in HPC

- 9 Synchronization
- 10 PThread
- 11 MPI
  - Message Passing
  - Introduction to MPI
  - Point-to-Point Communications
  - Collective Communications



# Point-to-Point Communication



- Data to be communicated is described by three things:
  - address
  - data type of the message
  - length of the message
- Involved processes are described by two things
  - communicator
  - rank
- Message is identified by a “tag” (integer) that can be chosen by the user



# Point-to-Point Communication

---

- Two modes of communication:
  - Synchronous: Communication does not complete until the message has been received
  - Asynchronous: Completes as soon as the message is “on its way”, and hopefully it gets to destination
- MPI provides four versions
  - synchronous, buffered, standard, ready



## Synchronous/Buffered sending in MPI

---

- Synchronous with MPI\_Ssend
  - The send completes only once the receive has succeeded
    - copy data to the network, wait for an ack
    - The sender has to wait for a receive to be posted
    - No buffering of data
- Buffered with MPI\_Bsend
  - The send completes once the message has been buffered internally by MPI
    - Buffering incurs an extra memory copy
    - Does not require a matching receive to be posted
    - May cause buffer overflow if many bsend's and no matching receives have been posted yet



## Standard/Ready Send

---

- Standard with `MPI_Send`
  - Up to MPI to decide whether to do synchronous or buffered, for performance reasons
  - The rationale is that a correct MPI program should not rely on buffering to ensure correct semantics
- Ready with `MPI_Rsend`
  - May be started *only* if the matching receive has been posted
  - Can be done efficiently on some systems as no hand-shaking is required

The logo for MPI\_RECV features a stylized graphic on the left consisting of overlapping yellow, red, and blue squares with a black crosshair. To the right of this graphic, the text "MPI\_RECV" is written in a large, blue, sans-serif font. A thin horizontal line extends from the end of the text across the slide.

# MPI\_RECV

- There is only one `MPI_Recv`, which returns when the data has been received.
  - only specifies the **MAX** number of elements to receive
- **Why all this junk?**
  - Performance, performance, performance
  - MPI was designed with constructors in mind, who would endlessly tune code to extract the best out of the platform (LINPACK benchmark).
  - Playing with the different versions of `MPI_?send` can improve performance without modifying program semantics
  - Playing with the different versions of `MPI_?send` can modify program semantics
  - Typically parallel codes do not face very complex distributed system problems and it's often more about performance than correctness.
  - You'll want to play with these to tune the performance of your code in your assignments

# Example: Sending and Receiving

```
#include <unistd.h>
#include <mpi.h>
int main(int argc, char **argv) {
    int i, my_rank, nprocs, x[4];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if (my_rank == 0) { /* master */
        x[0]=42; x[1]=43; x[2]=44; x[3]=45;
        MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
        for (i=1; i<nprocs; i++)
            MPI_Send(x, 4, MPI_INT, i, 0, MPI_COMM_WORLD);
    } else { /* worker */
        MPI_Status status;
        MPI_Recv(x, 4, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
    exit(0);
}
```

destination and source

user-defined tag

Max number of elements to receive

Can be examined via calls like MPI\_Get\_count(), etc.





## Example: Deadlock

```
...  
MPI_Ssend()  
MPI_Recv()
```

**Deadlock**

```
...  
MPI_Ssend()  
MPI_Recv()
```

```
...  
...  
MPI_Buffer_attach()  
MPI_Bsend()  
MPI_Recv()
```

**No  
Deadlock**

```
...  
...  
MPI_Buffer_attach()  
MPI_Bsend()  
MPI_Recv()
```

```
...  
...  
MPI_Buffer_attach()  
MPI_Bsend()  
MPI_Recv()  
...
```

**No  
Deadlock**

```
...  
...  
MPI_Ssend()  
MPI_Recv()  
...
```



## What about MPI\_Send?

- MPI\_Send is either synchronous or buffered....
- With , running “some” version of MPICH

**Deadlock**

...		...
<i>MPI_Send()</i>	<b>Data size &gt; 127999 bytes</b>	<i>MPI_Send()</i>
<i>MPI_Recv()</i>	<b>Data size &lt; 128000 bytes</b>	<i>MPI_Recv()</i>
...		...

**No  
Deadlock**

- Rationale: a correct MPI program should not rely on buffering for semantics, just for performance.
- So how do we do this then? ...



# Non-blocking communications

---

- So far we've seen blocking communication:
  - The call returns whenever its operation is complete (MPI\_SEND returns once the message has been received, MPI\_BSEND returns once the message has been buffered, etc..)
- MPI provides non-blocking communication: the call returns immediately and there is another call that can be used to check on completion.
- Rationale: Non-blocking calls let the sender/receiver do something useful while waiting for completion of the operation (without playing with threads, etc.).



# Non-blocking Communication

- MPI\_Issend, MPI\_Ibsend, MPI\_Isend, MPI\_Irsend, MPI\_Irecv

```
MPI_Request request;
```

```
MPI_Isend(&x, 1, MPI_INT, dest, tag, communicator, &request);
```

```
MPI_Irecv(&x, 1, MPI_INT, src, tag, communicator, &request);
```

- Functions to check on completion: MPI\_Wait, MPI\_Test, MPI\_Waitany, MPI\_Testany, MPI\_Waitall, MPI\_Testall, MPI\_Waitsome, MPI\_Testsome.

```
MPI_Status status;
```

```
MPI_Wait(&request, &status) /* block */
```

```
MPI_Test(&request, &status) /* doesn't block */
```



## Example: Non-blocking comm

```
#include <unistd.h>
#include <mpi.h>
int main(int argc, char **argv) {
    int i, my_rank, x, y;
    MPI_Status status;
    MPI_Request request;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if (my_rank == 0) { /* P0 */
        x=42;
        MPI_Isend(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
        MPI_Recv(&y, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
        MPI_Wait(&request, &status);
    } else if (my_rank == 1) { /* P1 */
        y=41;
        MPI_Isend(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
        MPI_Recv(&x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        MPI_Wait(&request, &status);
    }
    MPI_Finalize(); exit(0);
}
```



**No  
Deadlock**



## Use of non-blocking comms

---

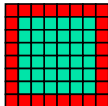
- In the previous example, why not just swap one pair of send and receive?
- Example:
  - A logical linear array of N processors, needing to exchange data with their neighbor at each iteration of an application
  - One would need to orchestrate the communications:
    - all odd-numbered processors send first
    - all even-numbered processors receive first
  - Sort of cumbersome and can lead to complicated patterns for more complex examples
  - In this case: just use `MPI_Isend` and write much simpler code
- Furthermore, using `MPI_Isend` makes it possible to overlap useful work with communication delays:

```
MPI_Isend()  
<useful work>  
MPI_Wait()
```



# Iterative Application Example

```
for (iterations)
  update all cells
  send boundary values
  receive boundary values
```



- Would deadlock with MPI\_Ssend, and maybe deadlock with MPI\_Send, so must be implemented with MPI\_Isend
- Better version that uses non-blocking communication to achieve communication/computation overlap (aka latency hiding):

```
for (iterations)
  initiate sending of boundary values to neighbours;
  initiate receipt of boundary values from neighbours;
  update non-boundary cells;
  wait for completion of sending of boundary values;

  wait for completion of receipt of boundary values;
  update boundary cells;
```
- Saves cost of boundary value communication if hardware/software can overlap comm and comp



# Non-blocking communications

---

- Almost always better to use non-blocking
  - communication can be carried out during blocking system calls
  - communication and communication can overlap
  - less likely to have annoying deadlocks
  - synchronous mode is better than implementing acks by hand though
- However, everything else being equal, non-blocking is slower due to extra data structure bookkeeping
  - The solution is just to benchmark
- When you do your programming assignments, you will play around with different communication types





## More information

---

- There are many more functions that allow fine control of point-to-point communication
- Message ordering is guaranteed
- Detailed API descriptions at the MPI site at ANL:
  - Google “MPI”. First link.
  - Note that you should check error codes, etc.
- Everything you want to know about deadlocks in MPI communication

<http://andrew.ait.iastate.edu/HPC/Papers/mpicheck2/mpicheck2.htm>

# Outlines: Low-level API in HPC

9 Synchronization

10 PThread

11 MPI

- Message Passing
- Introduction to MPI
- Point-to-Point Communications
- Collective Communications



# Collective Communication

---

- Operations that allow more than 2 processes to communicate simultaneously
  - barrier
  - broadcast
  - reduce
- All these can be built using point-to-point communications, but typical MPI implementations have optimized them, and it's a good idea to use them
- In all of these, all processes place the **same call** (in good SPMD fashion), although depending on the process, some arguments may not be used



# Barrier

---

- Synchronization of the calling processes
  - the call blocks until all of the processes have placed the call
- No data is exchanged
- Similar to an OpenMP barrier

```
...  
MPI_Barrier(MPI_COMM_WORLD)  
...
```



# Broadcast

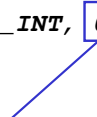
---

- One-to-many communication
- Note that multicast can be implemented via the use of communicators (i.e., to create processor groups)

...

```
MPI_Bcast (x, 4, MPI_INT, 0,  
           MPI_COMM_WORLD)
```

...



Rank of the root



## Broadcast example

---

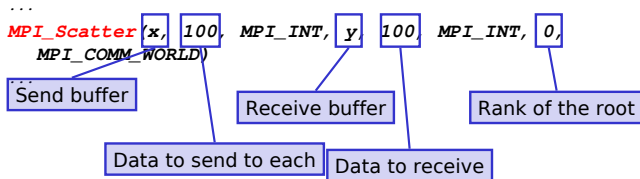
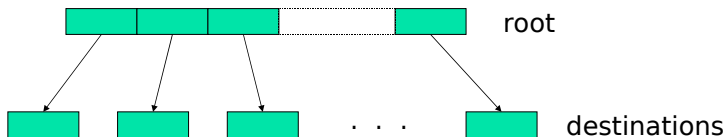
- Let's say the master must send the user input to all workers

```
int main(int argc, char **argv) {
    int my_rank;
    int input;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if (argc != 2) exit(1);
    if (sscanf(argv[1], "%d", &input) != 1) exit(1);
    MPI_Bcast(&input, 1, MPI_INT, 0, MPI_COMM_WORLD);
    ...
}
```



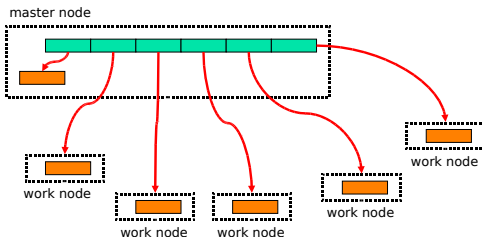
# Scatter

- One-to-many communication
- Not sending the same message to all



# This is actually a bit tricky

- The root sends data to itself!



- Arguments #1, #2, and #3 are only meaningful at the root





## Scatter Example

- Partitioning an array of input among workers

```
int main(int argc, char **argv) {
    int *a;
    double *recvbuffer;
    ...
    MPI_Comm_size(MPI_COMM_WORLD, &n);
    <allocate array recvbuffer of size N/n>

    if (my_rank == 0) { /* master */
        <allocate array a of size N>
    }
    MPI_Scatter(a, N/n, MPI_INT,
               recvbuffer, N/n, MPI_INT,
               0, MPI_COMM_WORLD);
    ...
}
```



# Scatter Example

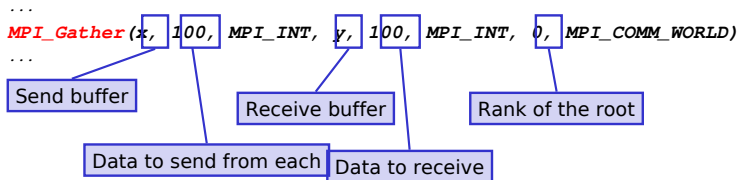
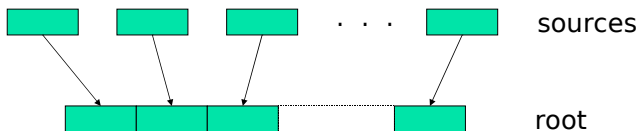
- Without redundant sending at the root

```
int main(int argc, char **argv) {
    int *a;
    double *recvbuffer;
    ...
    MPI_Comm_size(MPI_COMM_WORLD, &n);
    if (my_rank == 0) { /* master */
        <allocate array a of size N>
        <allocate array recvbuffer of size N/n>
        MPI_Scatter(a, N/n, MPI_INT,
                   MPI_IN_PLACE, N/n, MPI_INT,
                   0, MPI_COMM_WORLD);
    } else { /* worker */
        <allocate array recvbuffer of size N/n>
        MPI_Scatter(NULL, 0, MPI_INT,
                   recvbuffer, N/n, MPI_INT,
                   0, MPI_COMM_WORLD);
    }
    ...
}
```



# Gather

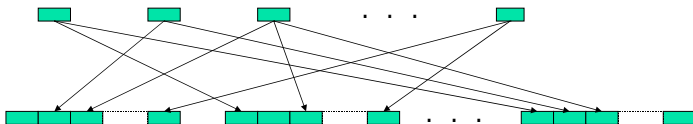
- Many-to-one communication
- Not sending the same message to the root





# Gather-to-all

- Many-to-many communication
- Each process sends the same message to all
- Different Processes send different messages



...  
`MPI_Allgather(x, 100, MPI_INT, y, 100, MPI_INT, MPI_COMM_WORLD)`  
...

Send buffer

Data to send to each

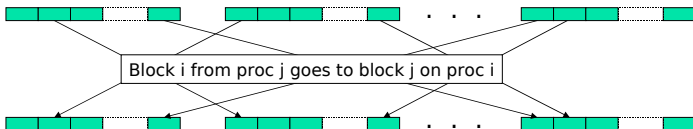
Receive buffer

Data to receive



# All-to-all

- Many-to-many communication
- Each process sends a different message to each other process



```
...  
MPI_Alltoall(x, 100, MPI_INT, y, 100, MPI_INT, MPI_COMM_WORLD)  
...
```

Send buffer

Data to send to each

Receive buffer

Data to receive



# Reduction Operations

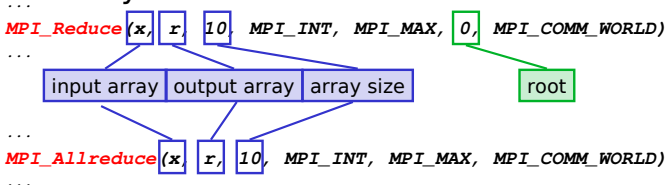
---

- Used to compute a result from data that is distributed among processors
  - often what a user wants to do anyway
    - e.g., compute the sum of a distributed array
  - so why not provide the functionality as a single API call rather than having people keep re-implementing the same things
- Predefined operations:
  - `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, etc.
- Possibility to have user-defined operations



# MPI\_Reduce, MPI\_Allreduce

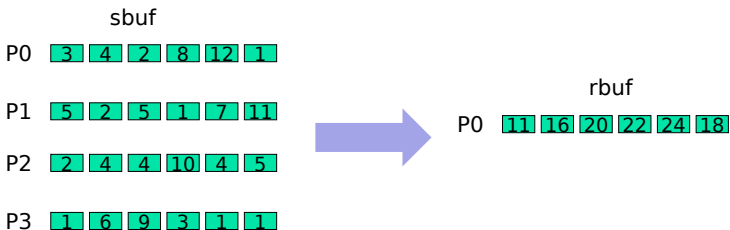
- MPI\_Reduce: result is sent out to the root
  - the operation is applied element-wise for each element of the input arrays on each processor
  - An **output array** is returned
- MPI\_Allreduce: result is sent out to everyone





# MPI Reduce example

***MPI\_Reduce***(sbuf, rbuf, 6, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD)

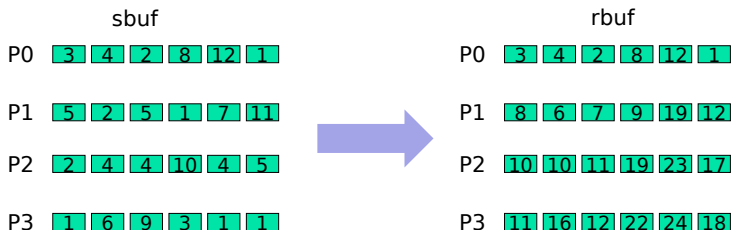






## MPI\_Scan: Prefix reduction

- Process  $i$  receives data reduced on process 0 to  $i$ .



***MPI\_Scan***(sbuf, rbuf, 6, MPI\_INT, MPI\_SUM, MPI\_COMM\_WORLD)



## And more...

---

- Most broadcast operations come with a version that allows for a stride (so that blocks do not need to be contiguous)
  - `MPI_Gatherv()`, `MPI_Scatterv()`, `MPI_Allgatherv()`, `MPI_Alltoallv()`
- `MPI_Reduce_scatter()`: functionality equivalent to a reduce followed by a scatter
- All the above have been created as they are common in scientific applications and save code
- All details on the MPI Webpage