

Parallel programming

Programming the memory models

- Shared memory: all processors share the same address space
 - OpenMP: directives-based programming
 - PGAS languages (UPC, Titanium, X10)
- Distributed memory: every processor has its own address space
 - MPI: Message Passing Interface

Ideal vs Practice

- Shared memory (or SMP: Symmetric MultiProcessor) is easy to program (OpenMP) but hard to build
 - bus-based systems can become saturated
 - large, fast (high bandwidth, low latency) crossbars are expensive
 - cache-coherency is hard to maintain at scale

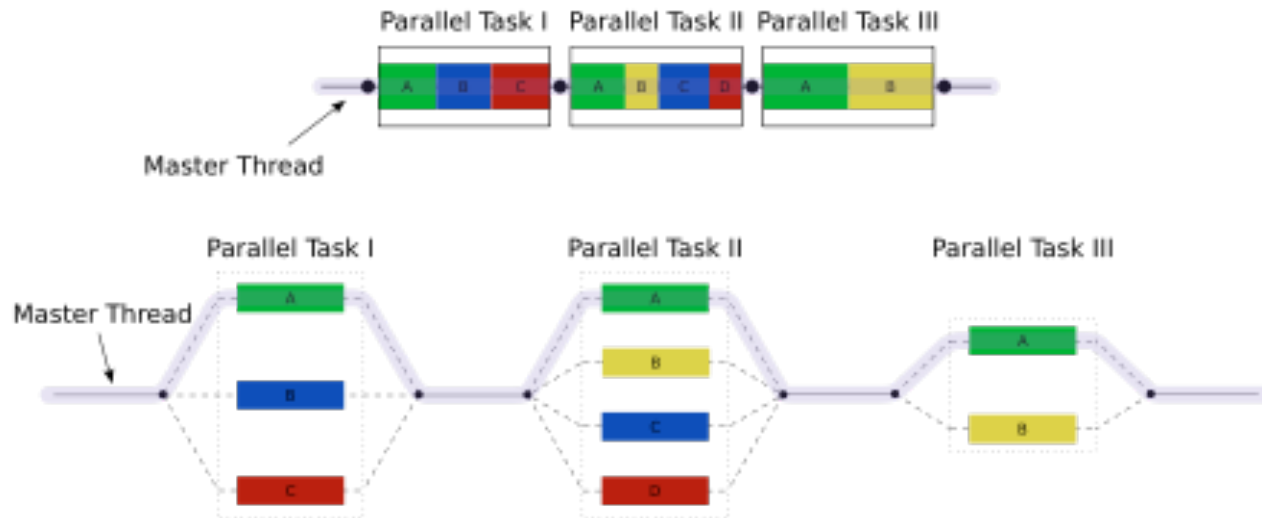
Ideal vs Practice

- Distributed memory is easy to build (bunch of PCs, ethernet) but hard to program (MPI)
 - You have to spell it all out
 - interconnects have higher latency, so data is not immediately there
 - makes parallel algorithm development and programming harder

Programmer's view vs Hard reality

- It is possible for distributed hardware to act like shared
- Middle layer: programmatic, OS, hardware support
- New machines: SGI UV, Cray Gemini

Shared memory programming in OpenMP



- Shared memory.
- Various issues: critical regions, binding, thread overhead

Thread programming

- Threads have shared address space (unlike processes)
- Great for parallel processing on shared memory
- Ex: quad-core => use 4 threads (8 with HT)
- OpenMP declares parallel tasks, the threads execute them in some order (shared memory essential!)
- Obvious example: loop iterations can be parallel

OpenMP programming

- “pragma”-based: directives to the compiler

```
#pragma omp parallel default(none) \  
    shared(n,x,y) private(i)  
{  
  #pragma omp for  
    for (i=0; i<n; i++)  
        x[i] += y[i];  
} /*-- End of parallel region --*/
```

clauses

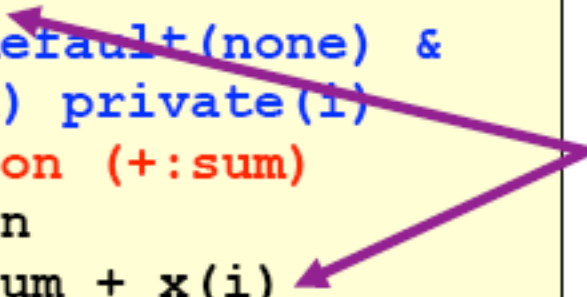
```
!$omp parallel default(none) &  
!$omp shared(n,x,y) private(i)  
!$omp do  
  do i = 1, n  
    x(i) = x(i) + y(i)  
  end do  
!$omp end do  
!$omp end parallel
```

OpenMP programming

- Handling of private and shared data

```
sum = 0.0
!$omp parallel default(none) &
!$omp shared(n,x) private(i)
!$omp do reduction (+:sum)
  do i = 1, n
    sum = sum + x(i)
  end do
!$omp end do
!$omp end parallel
print *,sum
```

Variable SUM is a shared variable



Now that threads have come up...

- Your typical core can handle one thread (two with HT)
- `Context switching` is expensive
- GPU handles many threads with ease, in fact relies on it
- => GPU is even more SIMD than you already realized

On to Distributed Memory

Parallel algorithms vs parallel programming

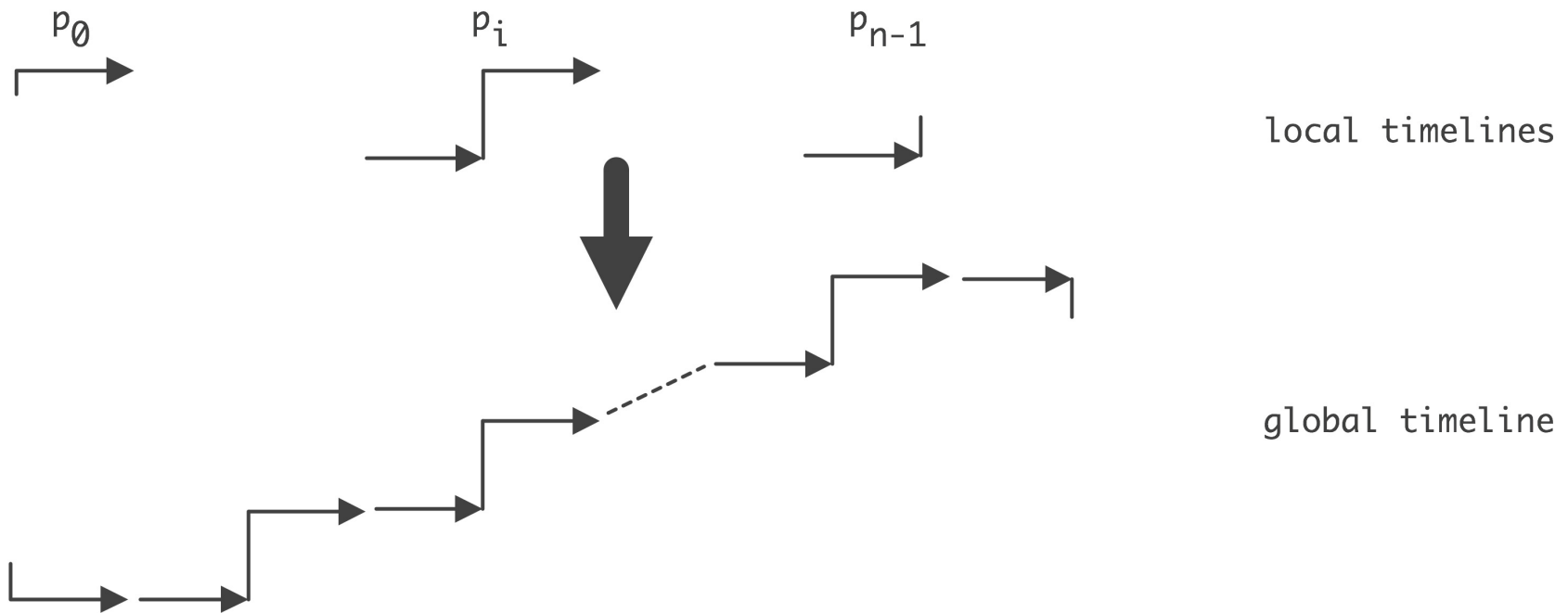
- Example: two arrays x and y ; n processors; p_i stores x_i and y_i
- Algorithm: $y_i := y_i + x_{i-1}$
- Global description:
 - Processors $0..n-2$ send their x element to the right
 - Processors $1..n-1$ receive an x element from the left
 - Add the received number to their y element

Local implementations

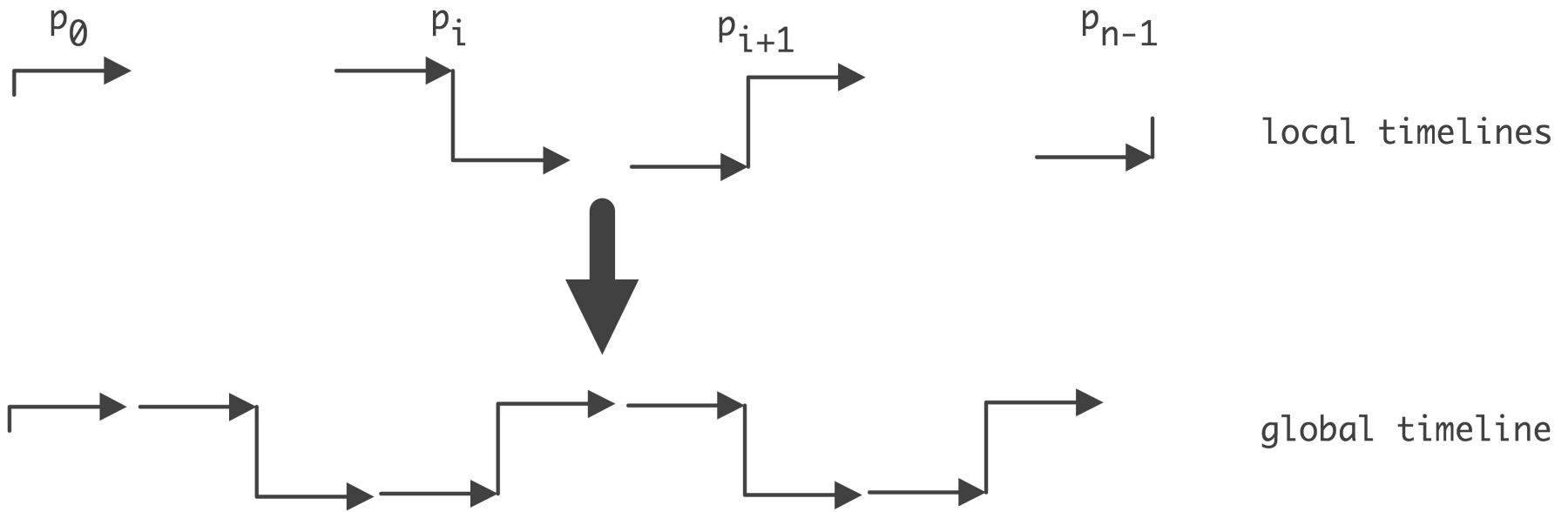
- One implementation:
 - If my number >0 : receive a x element, add it to my y element
 - If my number $<n-1$: send my x element to the right
- Other implementation
 - If my number $<n-1$: send my x element to the right
 - If my number >0 : receive a x element, add it to my y element

- Other implementation

- If my number $< n-1$: send my x element to the right
- If my number > 0 : receive a x element, add it to my y element



- Better implementation
 - If my number odd: receive then send
 - If my number even: send then receive



Blocking operations

- Send & recv operations are *blocking*: a send does not finish until the message is actually received
- Parallel operation becomes sequentialized; in a ring even loads to *deadlock*

Non-Blocking operations

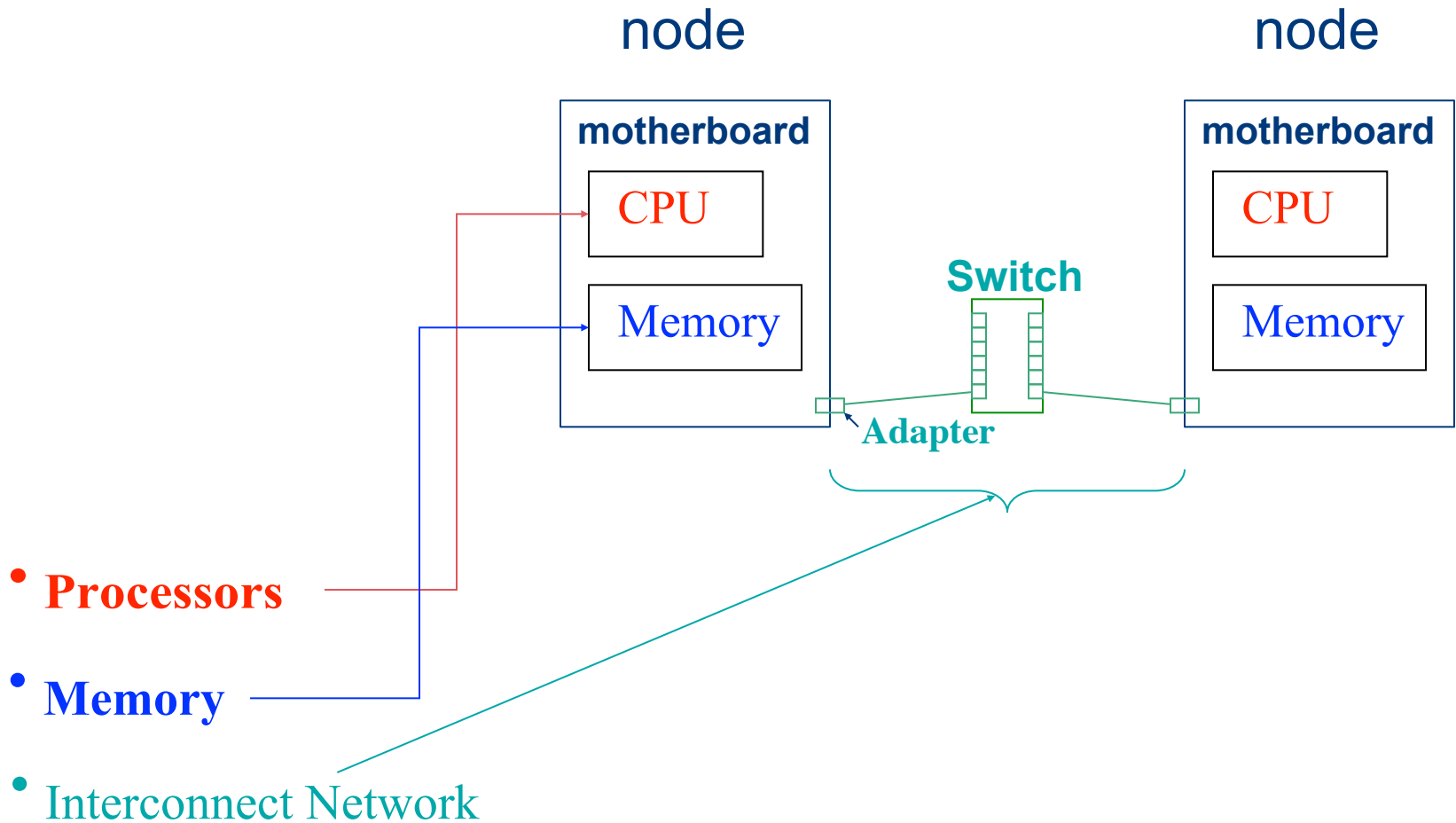
- Non-blocking send & recv:
 - Give a buffer to the system to send from / recv into
 - Continue with next instruction
 - Check for completion later

MPI: message passing

- Message Passing Interface: library for explicit communication
- Point-to-point and collective communication
- Blocking semantics, buffering
- Looks harder than it is

```
if(myid == 0)
{
    printf("WE have %d processors\n", numprocs);
    for(i=1;i<numprocs;i++)
    {
        sprintf(buff, "Hello %d", i);
        MPI_Send(buff, 128, MPI_CHAR,
                i, 0, MPI_COMM_WORLD);
    }
    for(i=1;i<numprocs;i++)
    {
        MPI_Recv(buff, 128, MPI_CHAR,
                i, 0, MPI_COMM_WORLD, &stat);
        printf("%s\n", buff);
    }
}
else
{
    MPI_Recv(buff, 128, MPI_CHAR,
            0, 0, MPI_COMM_WORLD, &stat);
    sprintf(idstr, " Processor %d ", myid);
    strcat(buff, idstr);
    strcat(buff, "reporting for duty\n");
    MPI_Send(buff, 128, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
}
```

Basic Anatomy of a Server/Desktop/ Laptop/Cluster-node



RAID

- Was: Redundant Array of Inexpensive Disks
- Now: Redundant Array of Independent Disks
- Multiple disk drives working together to:
 - increase capacity of a single logical volume
 - increase performance
 - improve reliability/add fault tolerance
- 1 Server with RAIDed disks can provide disk access to multiple nodes with NFS

Parallel Filesystems

- Use multiple servers together to aggregate disks
 - utilizes RAIDed disks
 - improved performance
 - even higher capacities
 - may use high-performance network
- Vendors/Products
 - CFS/Lustre
 - IBM/GPFS
 - IBRIX/IBRIXFusion
 - RedHat/GFS
 - ...

Summary

- Why so much parallel talk?
 - Every computer is a parallel computer now
 - Good serial computing skills central to good parallel computing
 - Cluster and MPP nodes are largely like desktops and laptops
 - Processing units: CPUs, FPUs, GPUs
 - Memory hierarchies: Registers, Caches, Main memory
 - Internal Interconnect: Buses and Switch-based networks
 - Clusters and MPPs built via fancy connections.