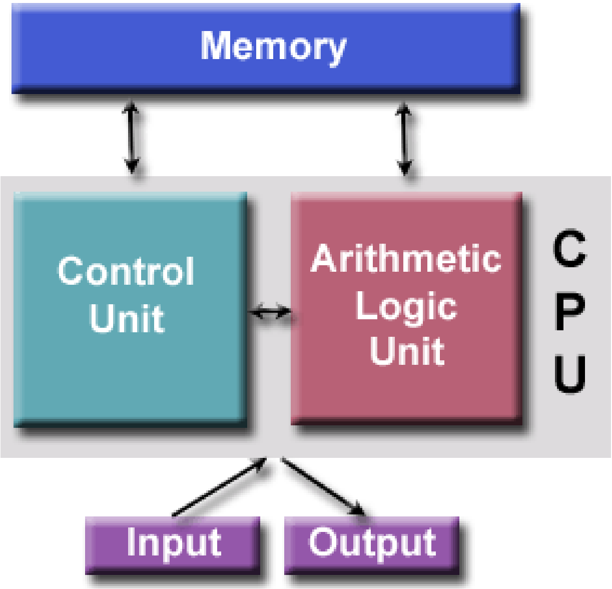
Parallel Computing Paradigms, etc

von Neumann Architecture

- For over 50 years, every computer has adopted a common model of architecture known as the von Neumann computer, named after the Hungarian mathematician John von Neumann
- A von Neumann computer uses the concept of "stored-program". The CPU executes a stored program that specifies a sequence of reads and writes to memory

Basic architecture:

- The **memory** is used to store both program instructions and data
- The **instructions** in the program encode data which tell the computer what to do
- **Data** is simply information that is used by the program
- A **Central Processing Unit (CPU)** fetches the information and / or data from memory, decodes the instructions and executes them sequentially



Flynn’s Taxonomy

In 1966, Michael J. Flynn classifies Computing Systems according to the multiplicity of the instruction stream and the data stream that can be handled

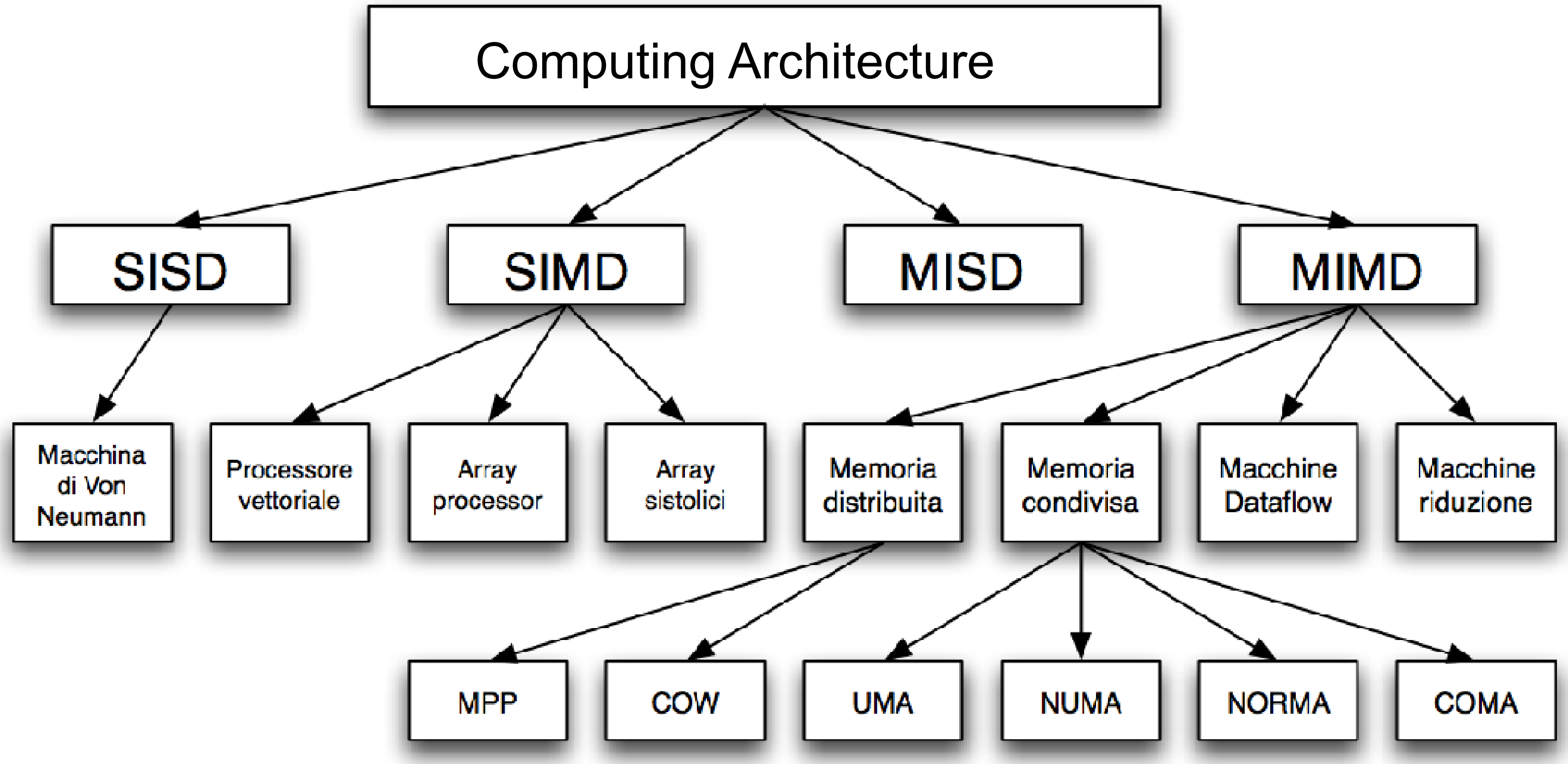
SISD (Single Instruction Single Data)

SIMD (Single Instruction Multiple Data)

MISD (Multiple Instruction Single Data)

MIMD (Multiple Instruction Multiple Data)

Flynn’s Taxonomy



Computing Architecture

SISD

SIMD

MISD

MIMD

Macchina di Von Neumann

Processore vettoriale

Array processor

Array sistolici

Memoria distribuita

Memoria condivisa

Macchine Dataflow

Macchine riduzione

MPP

COW

UMA

NUMA

NORMA

COMA

Single Instruction, Single Data (SISD)

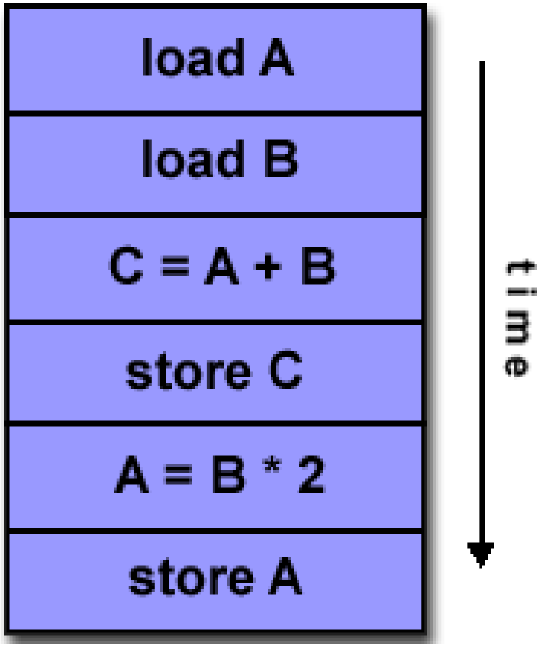
- A sequential (non-parallel) calculator

- Single instruction: only one instruction stream is executed during a machine cycle

- Single data: only one data stream is used as input during a machine cycle

- Deterministic execution

- Examples: the majority of PCs, workstations, and mainframes single CPU



SISD Machines

UNIVAC

CRAY 1

PDP1

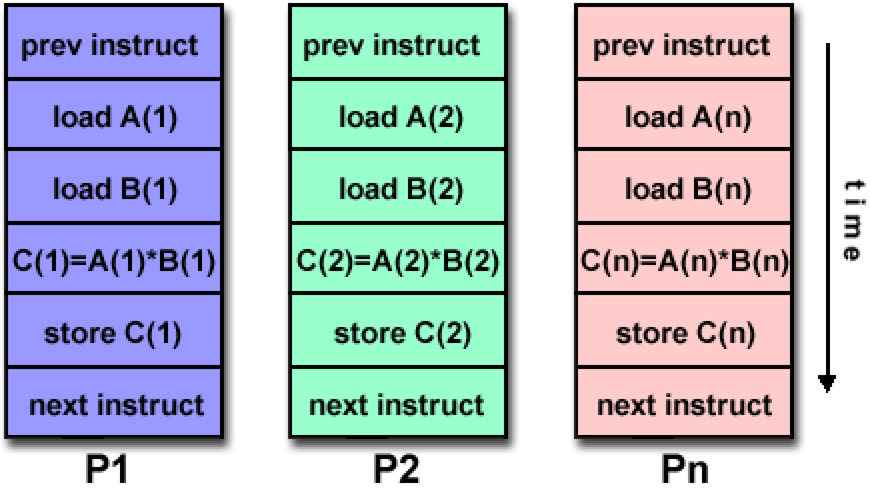
NOTEBOOK

Single Instruction, Multiple Data (SIMD)

•**Single instruction**: All computing units execute the same instruction at each cycle clock

•**Multiple data**: Each processing unit element can operate on a different date

•Typically, this machine has as **instruction dispatcher**, an internal high bandwidth network and a large number of **small units** for instructions



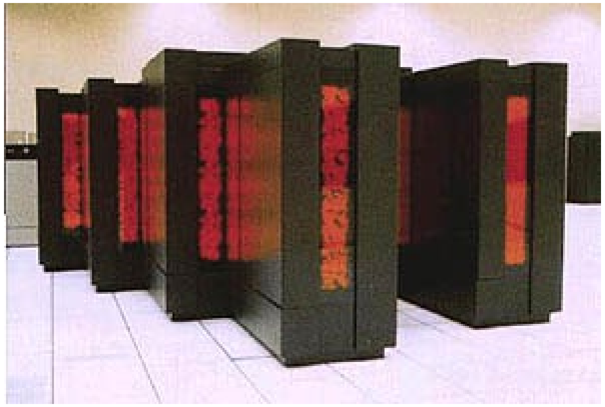
•Used mainly for "specialized‟ problems characterized by a high degree of regularity, as image processing (eg. GPU cards)

•Synchronous (**lockstep**) and deterministic execution

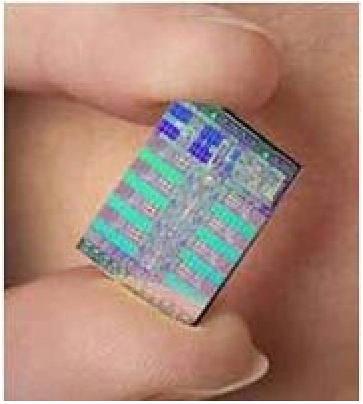
•Examples:

- Processor Arrays: Connection Machine CM-2, MasPar MP-1, MP-2
- Vector Pipelines: IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820
- GPU processors of standard computers (NVIDIA, ATI, etc.) or dedicated (TESLA)

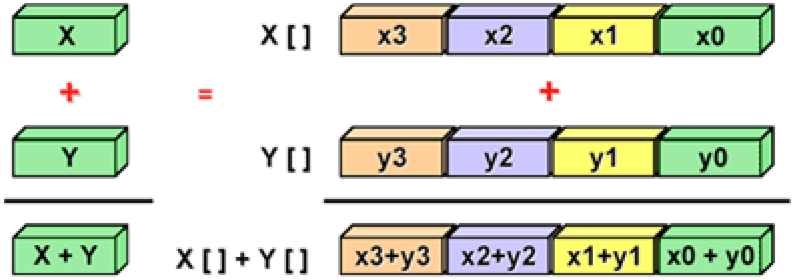
SIMD Machines



Thinking Machines (CM2)



Cray XMP



X X [ ] x3 x2 x1 x0

+ = +

Y Y [ ] y3 y2 y1 y0

X + Y X [ ] + Y [ ] x3+y3 x2+y2 x1+y1 x0 + y0

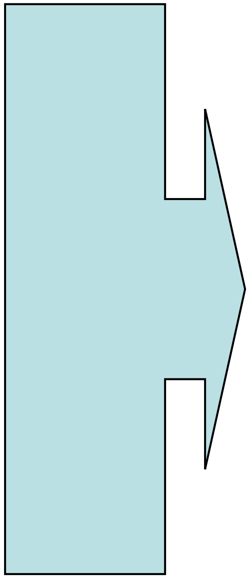
Cell Processors (GPU)

Pipeline Architectures

```
float a[100], b[100], c[100];
for (i=0; i<100; i++)
    c[i]= a[i] + b[i];
```

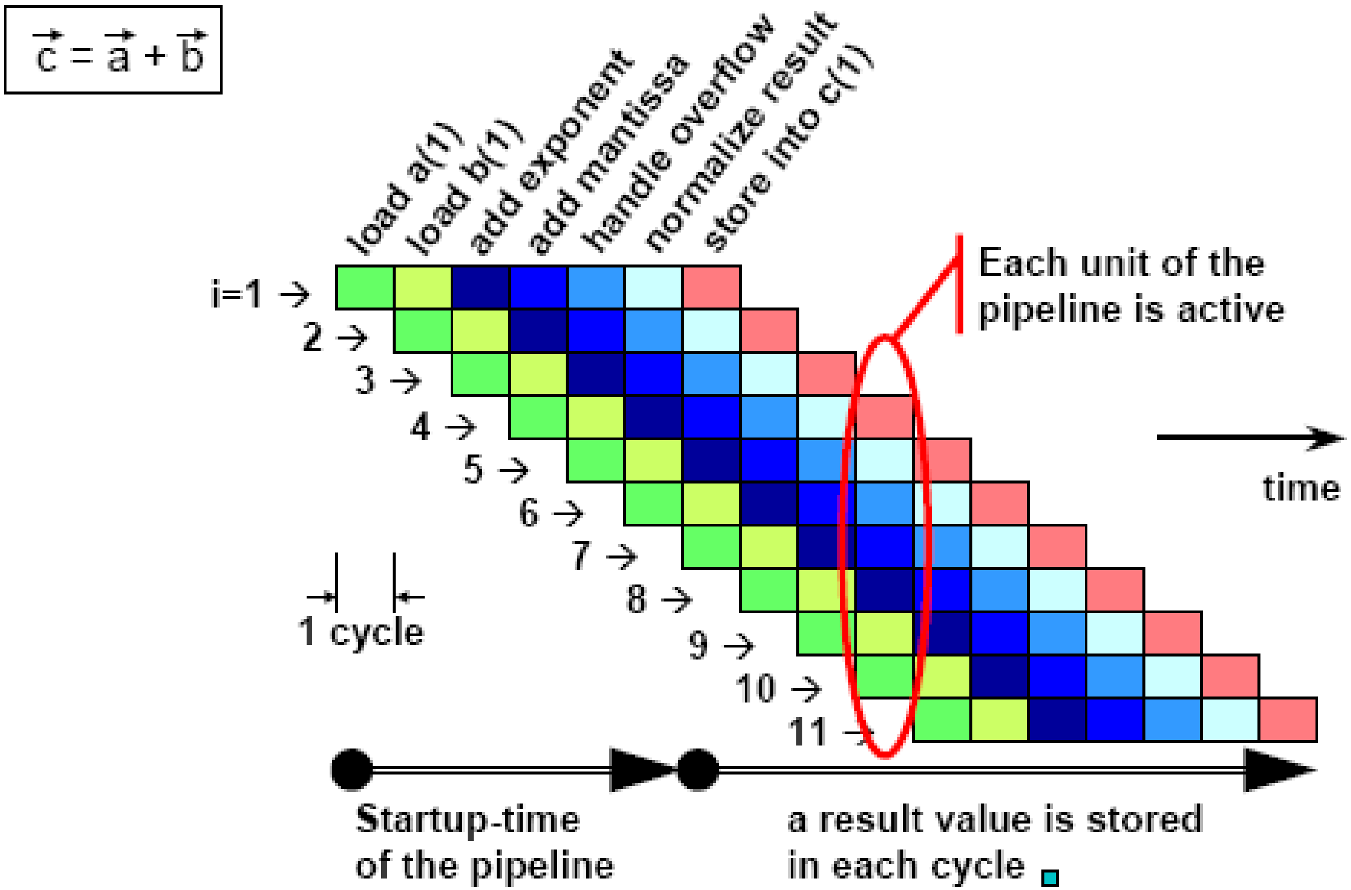
A single instruction can consist in:

1. Fetching operand `a` from memory
2. Fetching operand `b` from memory
3. Exponent summation
4. Mantissa summation
5. Overflow management
6. Normalization
7. Store `c`



7 functional pipeline units

Pipeline Architectures



Each unit of the pipeline is active

Startup-time of the pipeline

a result value is stored in each cycle

Vector Architectures?

```
float x[100], y[100], z[100];
for (i=0; i<100; i++)
   z[i]= x[i] + y[i];
```

Becomes (Fortran 90):

```
z(1:100) = x(1:100) + y(1:100)
```

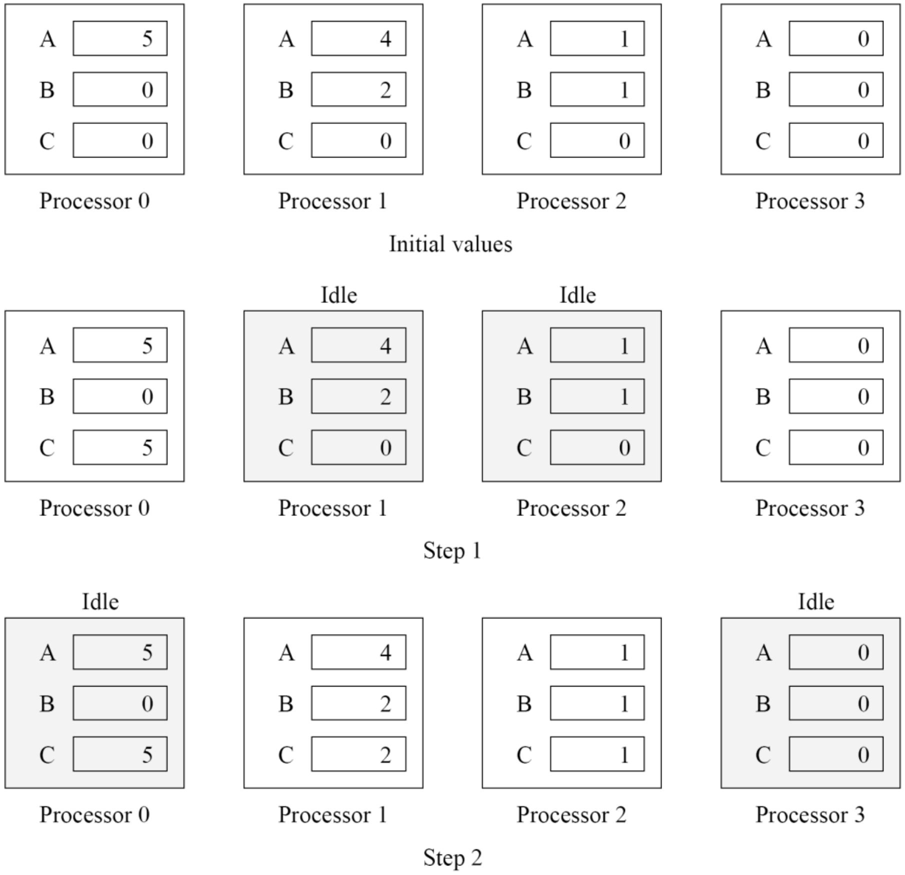
SIMD Programing

```
for (i=0; i<1000; i++)
      if (y[i]!=0.0)
            z[i]=x[i]/y[i];
      else
            z[i]=x[i];
```

Assuming an element for each processor, running synchronous (in **lockstep** - at each step each processor does the same thing):

```
1.   if (local_y!=0.0)
2.   if (local_y!=0.0) then local_z = local_x / local_y
     if (local_y==0.0) then do nothing
3.   if (local_y!=0.0) then do nothing
     if (local_y==0.0) then local_z = local_x
if (B == 0)
        C = A;
else
        C = A/B;
```

(a)



(b)

Figure 2.4 Executing a conditional statement on an SIMD computer with four processors: (a) the conditional statement; (b) the execution of the statement in two steps.

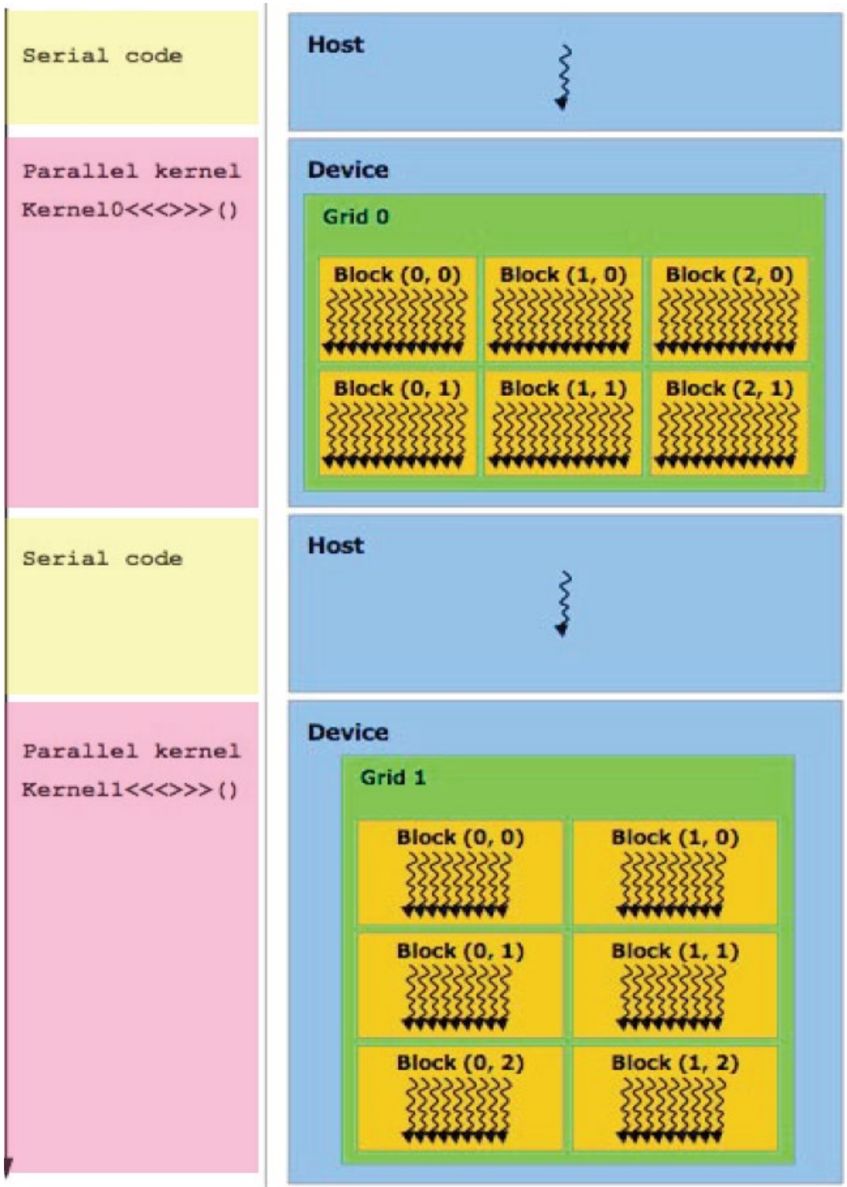
CUDA: Execution model

A Cuda code alternates portions of the **serial code**, **executed by the CPU** and **parallel code, executed on the GPU**.

The portions of code executed on the GPU are known as **kernels** (i.e., functions in C / C + +)
A kernel is defined as a **grid of blocks** that are assigned to various multiprocessors, and represent a coarse-grained parallelism.

Each block performs the fundamental units of computation, the **thread**.

A thread can belong to only one block, and is uniquely identified by an ID



Hello World in CUDA C

```
// Prototypes
__global__ void helloWorld(char*);

// Host function
int
main(int argc, char** argv)
{
  int i;

  // desired output
  char str[] = "Hello World!";

  // mangle contents of output
  // the null character is left intact for
//simplicity
  for(i = 0; i < 12; i++)
    str[i] -= i;

  // allocate memory on the device
  char *d_str;
  size_t size = sizeof(str);
  cudaMalloc((void**)&d_str, size);

  // copy the string to the device
  cudaMemcpy(d_str, str, size,
          cudaMemcpyHostToDevice);

  // set the grid and block sizes
  dim3 dimGrid(2);   // one block per word
  dim3 dimBlock(6); // one thread per
character
```

```
 // invoke the kernel
  helloWorld<<< dimGrid, dimBlock >>>(d_str);

  // retrieve the results from the device
  cudaMemcpy(str, d_str, size,
          cudaMemcpyDeviceToHost);

  // free up the allocated memory on the device
  cudaFree(d_str);

  // everyone's favorite part
  printf("%s\n", str);

  return 0;
}

// Device kernel
__global__ void
helloWorld(char* str)
{
  // determine where in the thread grid we are
  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  // unmangle output
  str[idx] += idx;
}
```

Multiple Instruction, Single Data (MISD)

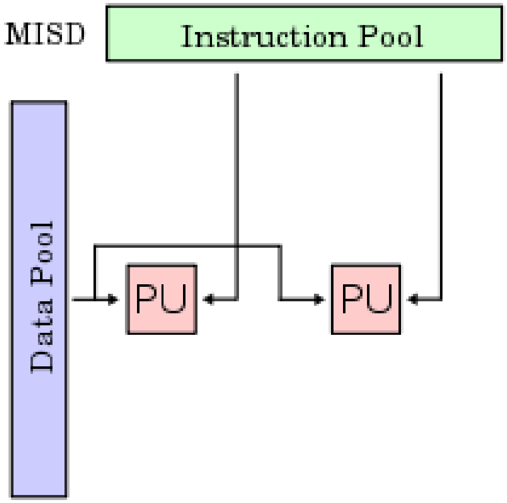
- Few real and practical examples exist

Some examples might be:

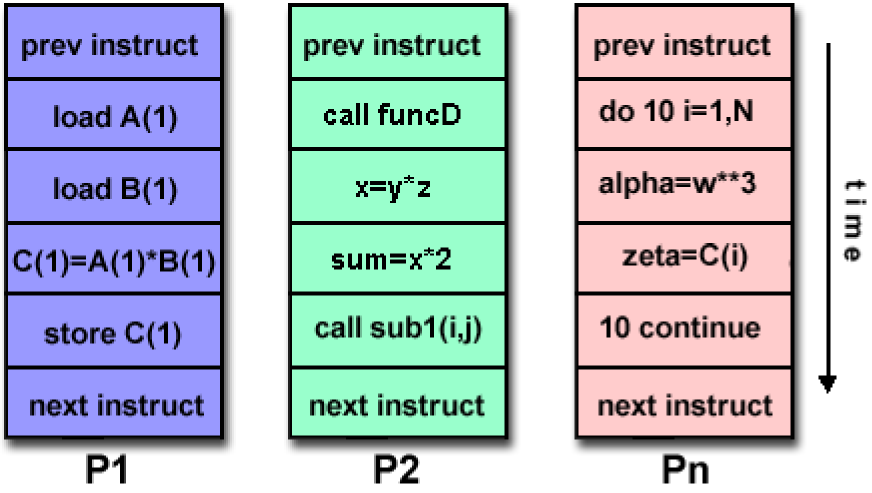
- Multiple frequency filters that operate on a single flow signal

- Multiple encryption algorithms that try to crack a single message

...



Multiple Instruction, Multiple Data (MIMD)



- The most common type of parallel computer

- **Multiple Instruction**: Each processor can execute a different stream of instructions

- **Multiple Data**: Each processor can work with a different data stream

- Execution can be synchronous or asynchronous, deterministic or non-deterministic

- **Examples**: current supercomputers, "grid" of parallel computers based on network computers and multi-processor SMP, all existing PC / notebook (eg Dual Core i7)

MIMD Machines

Compaq/HP Alpha server

AMD Opteron

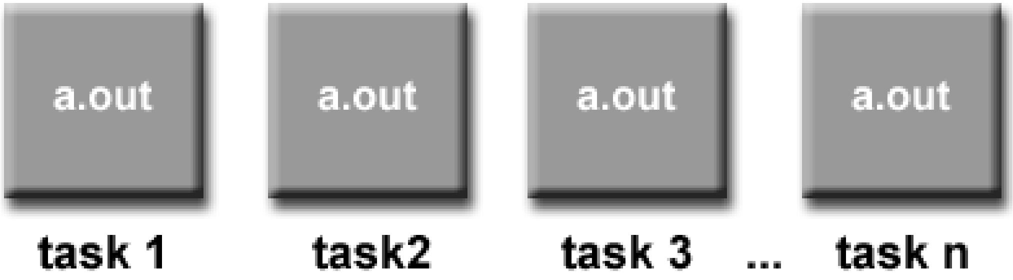
IBM Blue Gene

MIMD Paradigms

- **MPMD** - Each processor has a different program to be executed
- **SPMD** - Each processor has the same program running, although independent and asynchronous. MPI (Message Passing Interface) is based on this paradigm
- However, with branching techniques, the model can be efficiently emulated on MPMD models SPMD

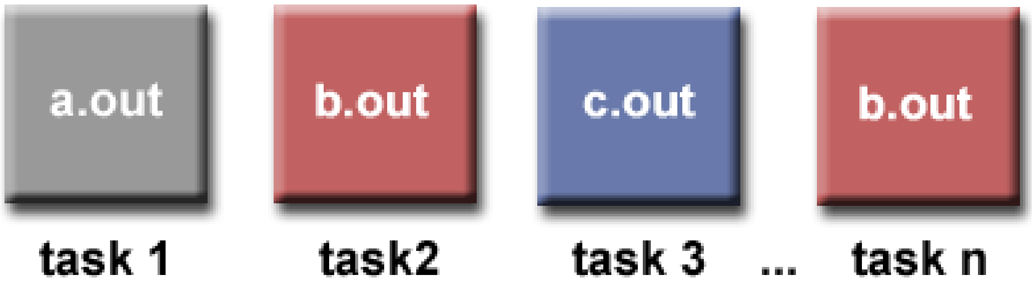
SPMD Model

- The **Single Program Multiple Data (SPMD)** model is a high-level parallel programming model that is based on the models mentioned previously
- A **program** (task) is performed by all individual processes simultaneously
- At any given time, processes can execute the same instructions or other instructions.
- **SPMD** programs usually have a specific logic that allows them to run, by "**branching**", only those parts of the program for which they are designed. That is, the process does not need to run the entire program, but only part of it.
- All tasks usually work on different data



MPMD Model

- As the SPMD model, the **Multiple Program Multiple Data (MPMD)** model is a parallel programming model that can be constructed on any combination of the previous models
- MPMD typical applications consist of multiple executable programs. While the parallel application is running, each task can run the same program or a different one
- All tasks usually work on different data



Communication Models in MIMD Platforms

- As we have seen, there are two types of data exchange between parallel tasks –
  - access to a **shared data space**
  - **through messages exchange**

- Platforms that provide a common space shared are called **shared-address-space or Multiprocessors** (shared memory)

- Platforms that allow the exchange of messages are called **message passing platforms or multicomputers** (distributed-memory)

Shared-Address-Space Platforms (Shared Memory)

- Part (or all) the memory is directly accessible by all processors

- Processors interact by modifying data stored in this "space" shared

- If the time taken by a processor to access any memory word of the system (local or global) is identical, the platform is classified as **uniform memory access (UMA)**, otherwise, as the machine **non-uniform memory access (NUMA)**

NUMA and UMA Shared-Address-Space Platforms

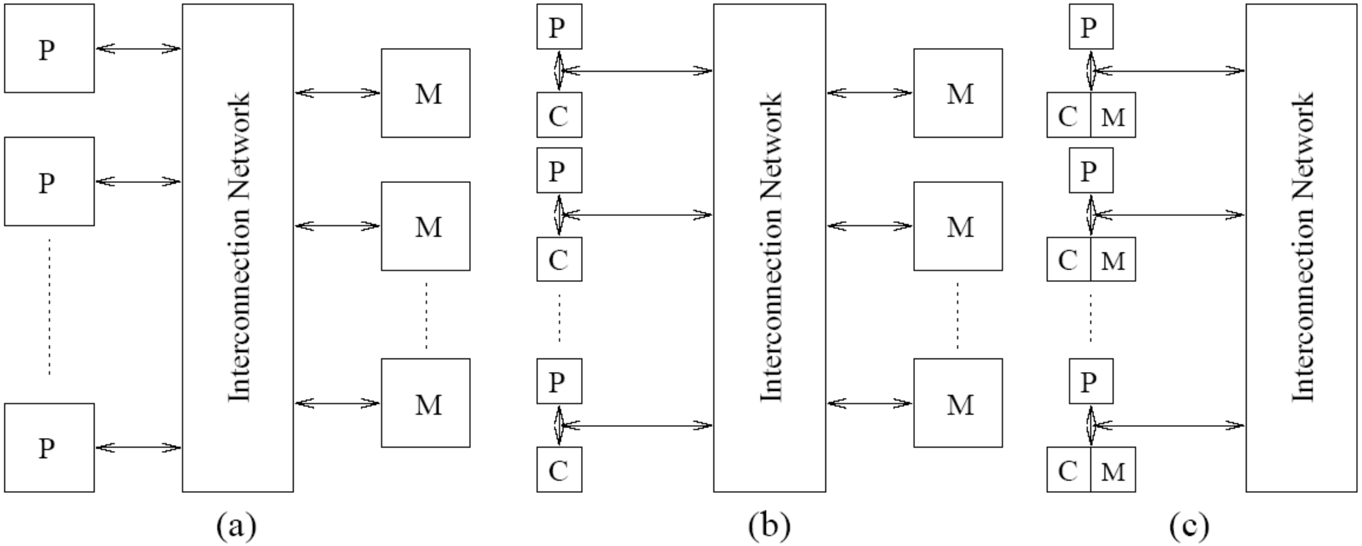


Figure 2.5 Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.

Shared-Address-Space Classes

Uniform Memory Access (UMA):

- Mainly represented by symmetric multiprocessor (SMP) machines
- Identical processors
- Identical memory access times
- Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, other processors are aware of the change. The **cache coherency** is done at the hardware level

Non-Uniform Memory Access (NUMA):

- Implemented by physically connecting two or more SMP
- One SMP can directly access the memory of the SMP
- Not all processors have equal access time to memory
- Access times slower through the links
- If the cache coherency is maintained, it is called CC-NUMA - Cache Coherent NUMA

Pros & Cons (Shared-Address-Space)

Advantages:

The space addressing allows a user-friendly type programming

The sharing of data between tasks is fast and uniform due to the "proximity" of the memory to the CPU

Disadvantages:

The main drawback is the lack of scalability between memory and CPUs. Adding more CPUs, there is an “geometrical” increase traffic on the shared bus memory CPU, and cache coherent systems, a geometric increase in traffic cache / memory

Programmer's responsibility to ensure that **synchronization** constructs a correct access to the global memory

Cost: it is **difficult and expensive** to design and produce shared memory machines with a greater number of processors ... (max. 64-128 proc)

NUMA and UMA Shared-Address-Space Platforms

- The distinction between UMA and NUMA platforms is important from the algorithmic design point of view. NUMA machines require “**locality"** in the algorithms to achieve good performance
- However, read-write operations to shared data must be **coordinated**
- **Caches of these machines require coordinated access to multiple copies**. This leads to the problem of **cache coherence**

Methods for Multiprocessor Programming (Shared Memory)

- **Threads** (Pthreads, Java, etc), where the programmer decomposes the program into individual parallel sequences, each being a "thread", and able to access variables declared outside this thread
- A (standard) sequential language, with preprocessor compilation directives, suitable for the specification of shared variables and the specification of parallelism. ES: **OpenMP** (industry standard)
- A (standard) compiler language with libraries (eg, methods, etc) for the declaration and access to shared variables
- A parallel programming language with parallelism syntax, in which the compiler creates the appropriate execution code for each processor (es: **Julia**)
- A sequential programming language, combined with a "parallel" ad hoc compiler for the conversion into parallel code

Threads Model

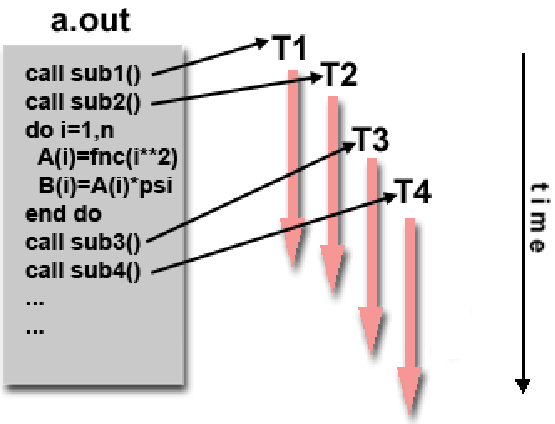
In the threads model, a single process can have several concurrent "paths"

- The main program `a.out` is scheduled to be run by the native operating system. `a.out` is loaded and acquires the system and user resources needed to be executed

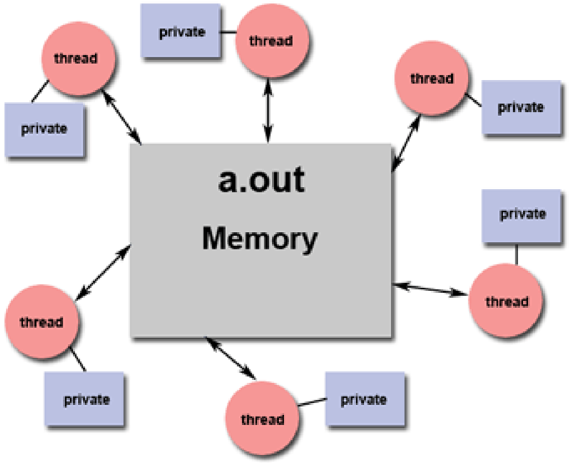
- `a.out` performs work sequentially, then creates a number of tasks (ie, threads) that can be scheduled and executed by the operating system concurrently

- Each thread has local data, but also shares the entire resources of `a.out`. This saves overhead generated by the replication of program resources for each thread. Each thread also benefits from the global view of shared memory of `a.out`.

- Threads are commonly associated with shared memory architectures and operating systems



```
call sub1()
call sub2()
do i=1,n
  A(i)=fnc(i**2)
  B(i)=A(i)*psi
end do
call sub3()
call sub4()
...
...
```



Threads Implementions

From a programming point of view, the main thread implementations include:

- A library of subroutines that are called from within the parallel code
- A set of compiler directives embedded in both parallel and sequential code

POSIX Threads

Based on libraries; requires parallel coding

Specified by the IEEE POSIX 1003.1c (1995).

Only for the C language

Commonly known as Pthreads.

Most hardware vendors offer Pthreads in addition to their proprietary implementations.

Very explicit parallelism, requires a very significant level of detail by the programmer

OTHERS … (Qt Threads, Openthreads) … OpenMP!

Cache Coherence in Multiprocessor Systems

- Interconnection networks **provide** the basic mechanisms for the transfer of data

- However, in the case of shared address space machines, additional hardware is required to **coordinate access** to the data that may have **multiple copies** in the network

Cache Coherence in Multiprocessor Systems

When the value of a variable changes, all other copies must be **invalidated** or **updated**

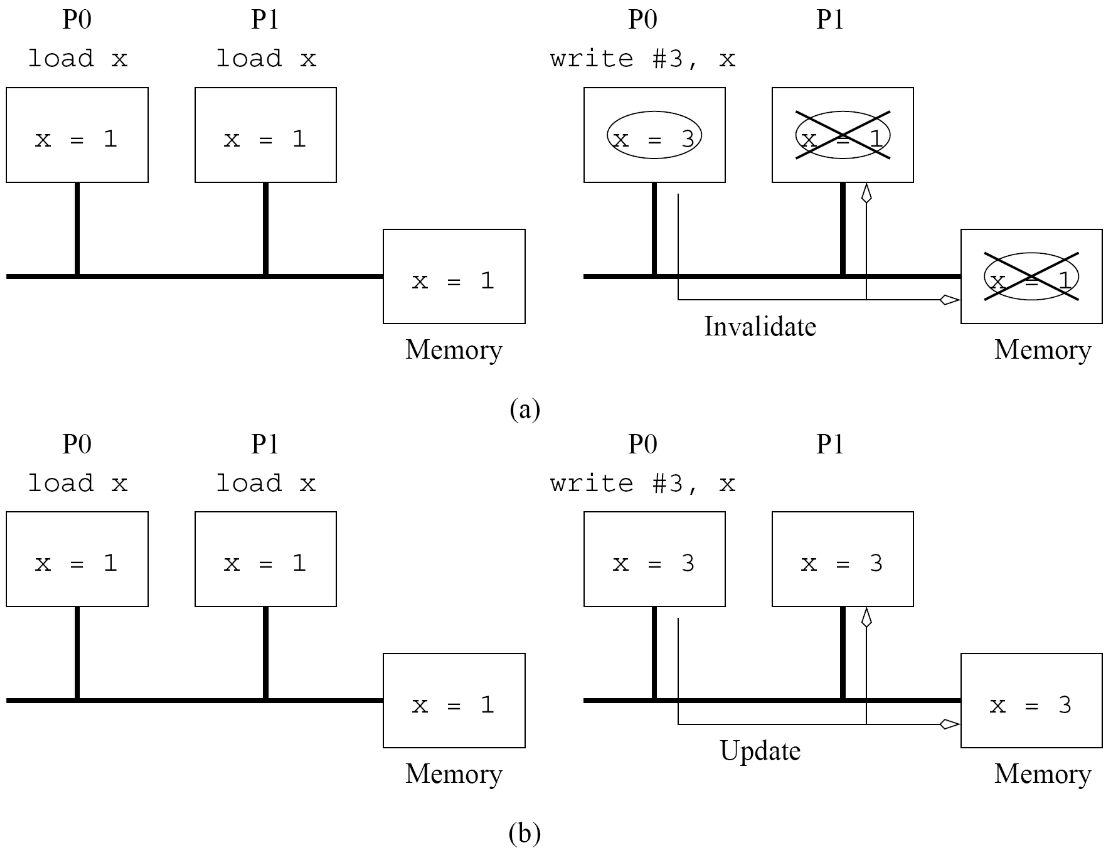


Figure 2.21 Cache coherence in multiprocessor systems: (a) Invalidate protocol; (b) Update protocol for shared variables.

Cache Coherence: Update or Invalidate Protocols

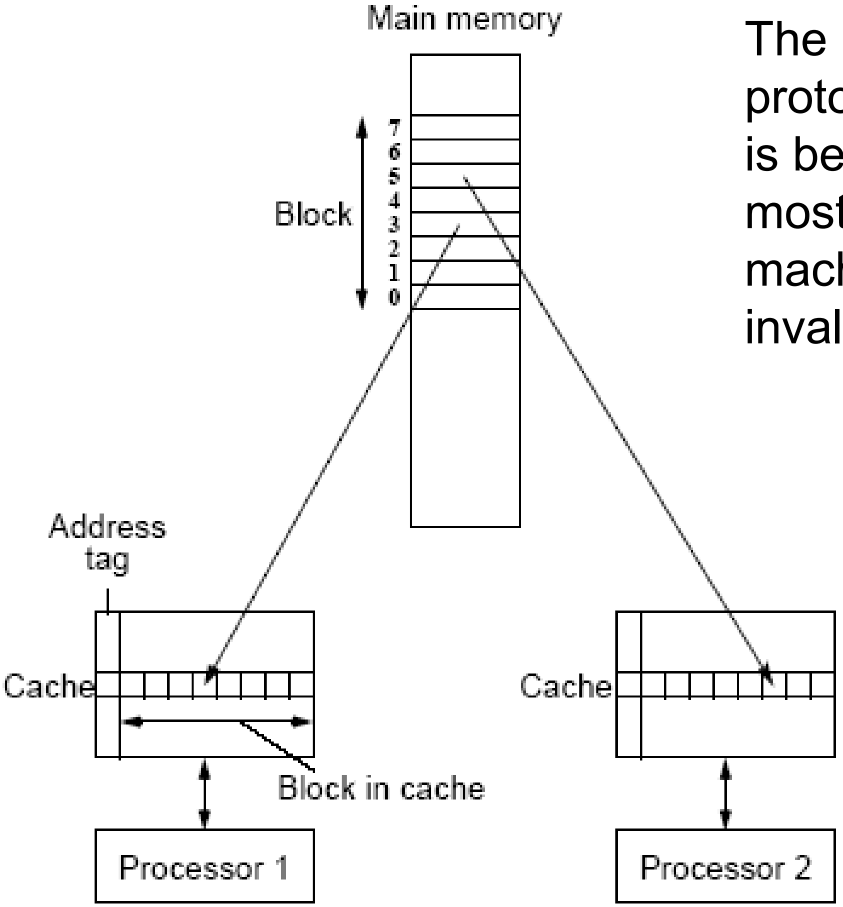
- If a processor reads a value only once, and it is not needed any more, an **update protocol** can generate a **significant overhead**
- If two processors working in “interleaved” access a variable, it is better to use an update protocol
- Both protocols suffer from **false sharing** overheads, a typical performance-degrading usage pattern

Most machines use invalidating protocols

False Sharing

Different parts of a block (**cache line**) required by different processors but **not the same data.**

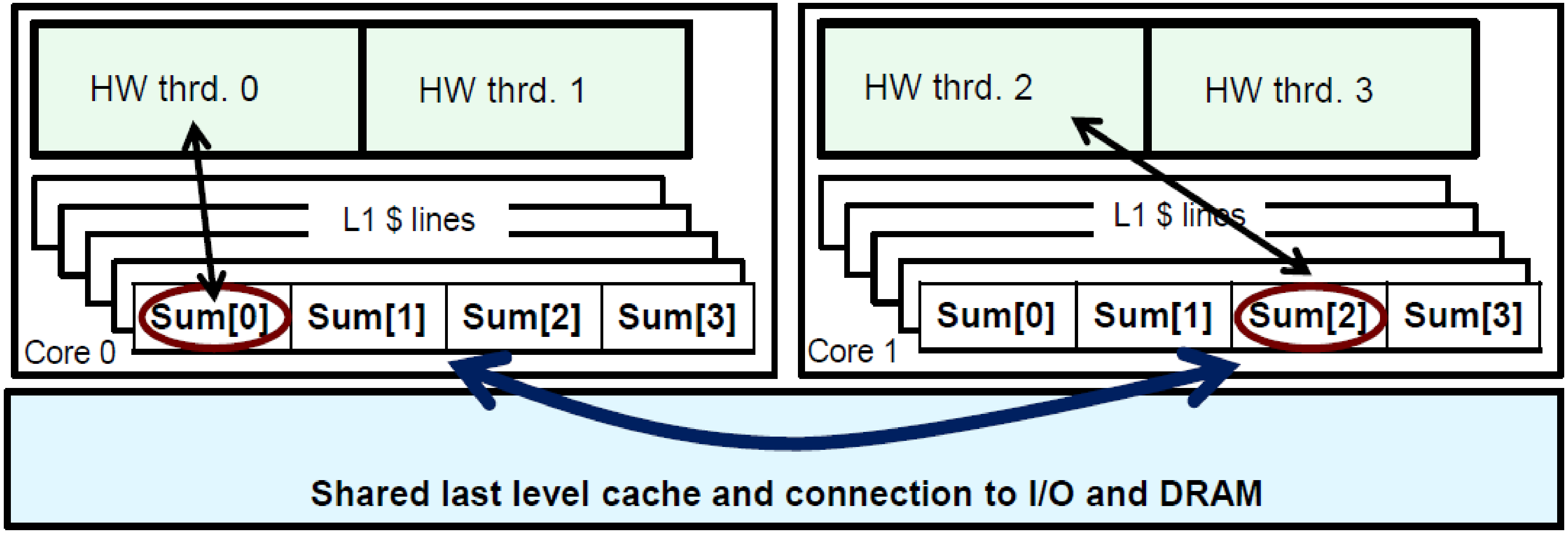
In this case, if a processor writes a part of the block, copies of the complete block in other caches need to be updated or invalidated, **<u>degrading the performance</u>** of the system, given that the actual data (i.e., the entire block) is **not shared**



The use of update protocols
is better, although most modern machines adopt invalidate protocols

Why such poor scaling? False sharing!

- If independent data elements happen to sit on the same cahe line, each update will cache lines to «slosh back and forth» between threads…. FALSE SHARING.



- If you promote scalars to an array to support creation of a SPDM program, the array elements are contiguous in memory and hence share cache lines… results in poor scalability
- Solution: Pad arrays so elements you see are on distinct cache lines

Example

```
struct foo {
    int x;
    int y;
};

static struct foo f;

/* The two following functions are running concurrently: */

int sum_a(void)
{
    int s = 0;
    int i;
    for (i = 0; i < 1000000; ++i)
        s += f.x;
    return s;
}

void inc_b(void)
{
    int i;
    for (i = 0; i < 1000000; ++i)
        ++f.y;
}
```

Here, `sum_a` may need to continually re-read `x` from main memory (instead of from cache) even though `inc_b`'s concurrent modification of `y` is irrelevant for `sum_a` !

The Maintenance of Coherence by Invalidating Protocols

- Each copy of the data is associated with a **state**
- The possible states of a variable are: **shared, invalid or dirty**

- In the **shared state**, there are multiple valid copies of the data
- If a processor modifies a value, an invalidation must be generated on other copies of other processors. The copy becomes **dirty**

- In the **invalid state**, the copy of the data is invalid, and a reading generates a data request
- *In the dirty state, only one copy exists, and therefore no invalidation must be generated*

The Maintenance of Coherence by Invalidating Protocols

→ processor

⇢ cache

C_* = coherent

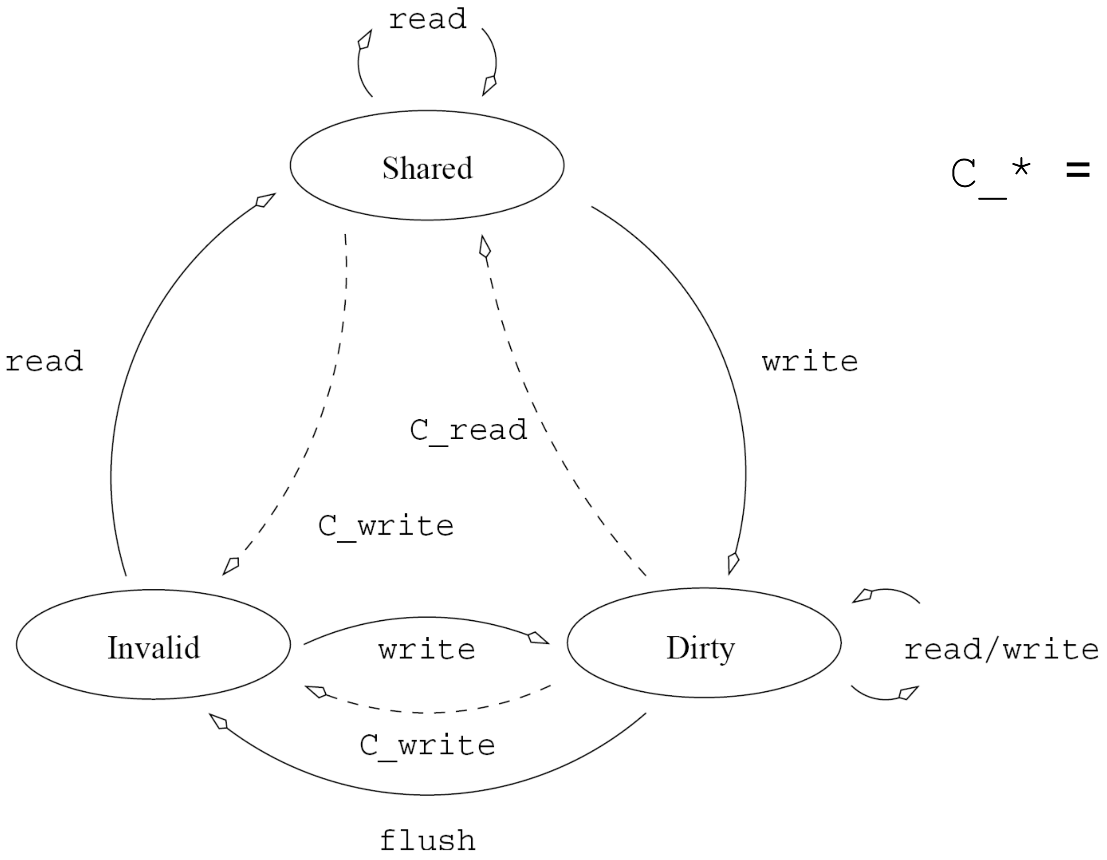


Figure 2.22 State diagram of a simple three-state coherence protocol.

The Maintenance of Coherence

Time	Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.
					x = 5, D y = 12, D
	read x		x = 5, S		x = 5, S
		read y		y = 12, S	y = 12, S
	x = x + 1		x = 6, D		x = 5, I
		y = y + 1		y = 13, D	y = 12, I
	read y		y = 13, S	y = 13, S	y = 13, S
		read x	x = 6, S	x = 6, S	x = 6, S
	x = x + y		x = 19, D	x = 6, I	x = 6, I
		y = x + y	y = 13, I	y = 19, D	y = 13, I
	x = x + 1		x = 20, D		x = 6, I
		y = y + 1		y = 20, D	y = 13, I

Figure 2.23 Example of parallel program execution with the simple three-state coherence protocol discussed in Section 2.4.6.

Snoopy Cache Systems

How are invalidates sent to the right processor?

In **snoopy caches** (buses, rings), there is a broadcast device that "listens" and invalidates all requests for readings, performing the appropriate operations on the basis of the previous diagram

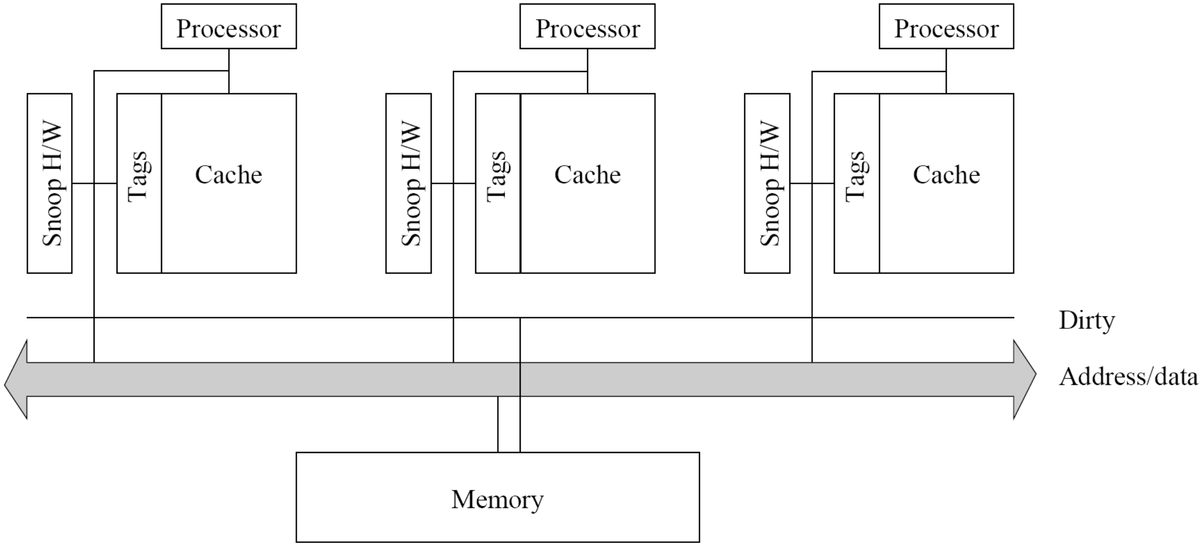
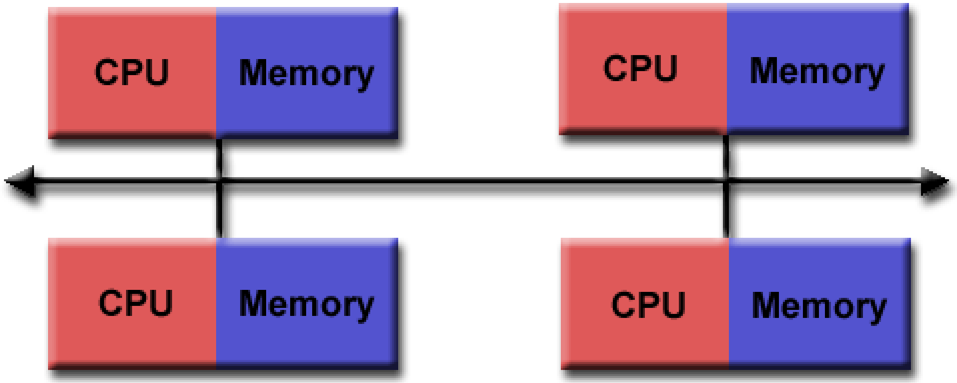


Figure 2.24 A simple snoopy bus based cache coherence system.

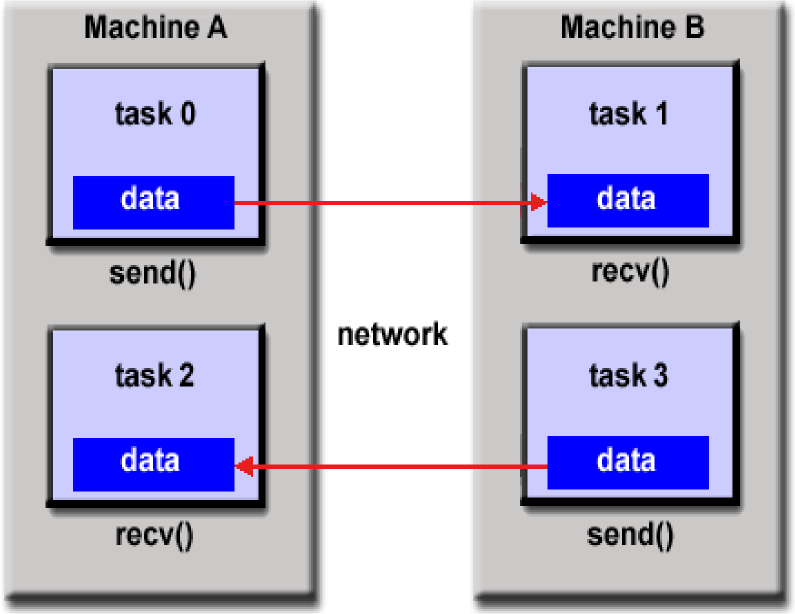
Message-Passing Platforms (Distributed Memory)

- These platforms include a set of processors with their own **exclusive memory** (from the logical point of view)
- Examples are clusters of workstations and multicomputer with space-address-not-shared (non-shared-address-space multicomputers)
- These platforms are programmed using (variants of) **send** and **receive** primitives
- Libraries such as MPI and PVM provide such primitives
- They do not need, obviously, the use of cache-coherence protocols



Message Passing Model

1. A set of tasks that use their memory during the computation. Multiple tasks can reside on the same physical machine and on different machines
2. The tasks exchange data through communications by sending and receiving messages.
3. The data transfer usually requires cooperative operations for each process. For example, a send operation must have a corresponding receive



From a programmer point of view, the message passing implementations usually include a library of functions that are incorporated in the code. The programmer is responsible for determining the parallelism.

MPI (Message Passing Interface), standard de-facto for the Message Passing model

Message Passing Model

- Single program Multiple data (SMPD) Approach

```
if (my_process_rank ==0)
   MPI_Send(&x, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
else
   if (my_process_rank ==1)
     MPI_Recv(&x, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
            &status);
```

The MPI paradigm

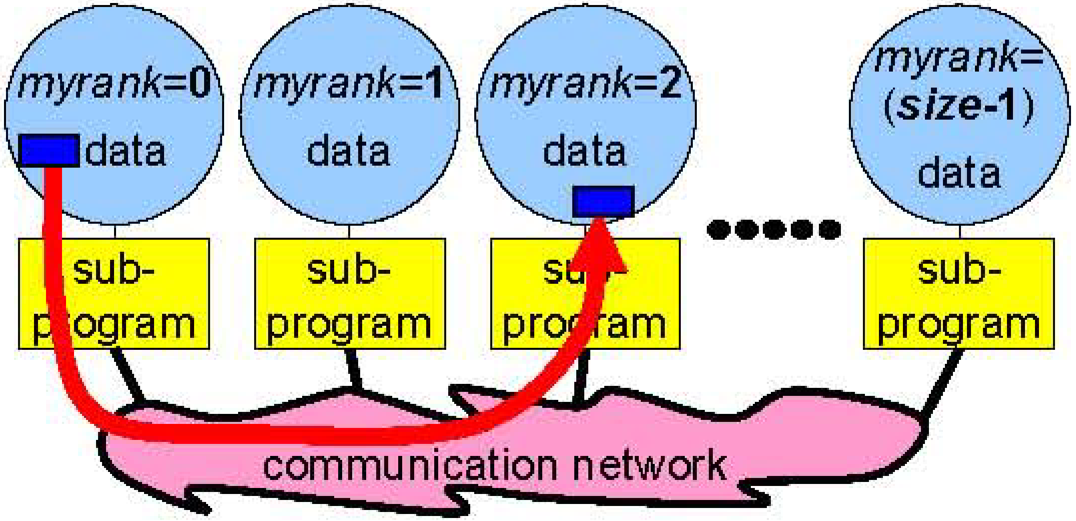
Each processor in a message passing program runs a ***sub-program***

- written in a conventional sequential language, e.g., C or Fortran,
- typically the same on each processor (SPMD)

All work and data distribution is based on value of ***myrank***

- returned by special library routine

Communication via special send & receive routines (***message passing***)



Communication

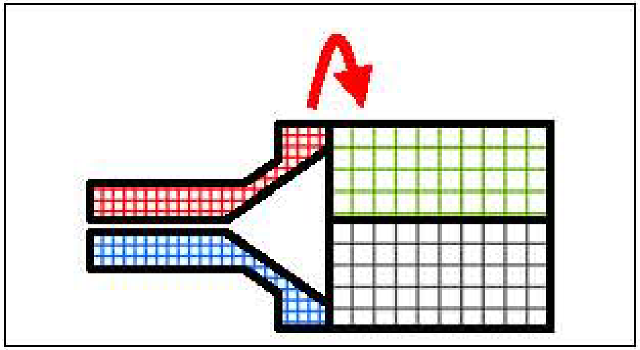
```
Do i=2,99
    b(i) = a(i) + f*(a(i-1)+a(i+1)-2*a(i))
Enddo
```

Communication is necessary on the boundaries

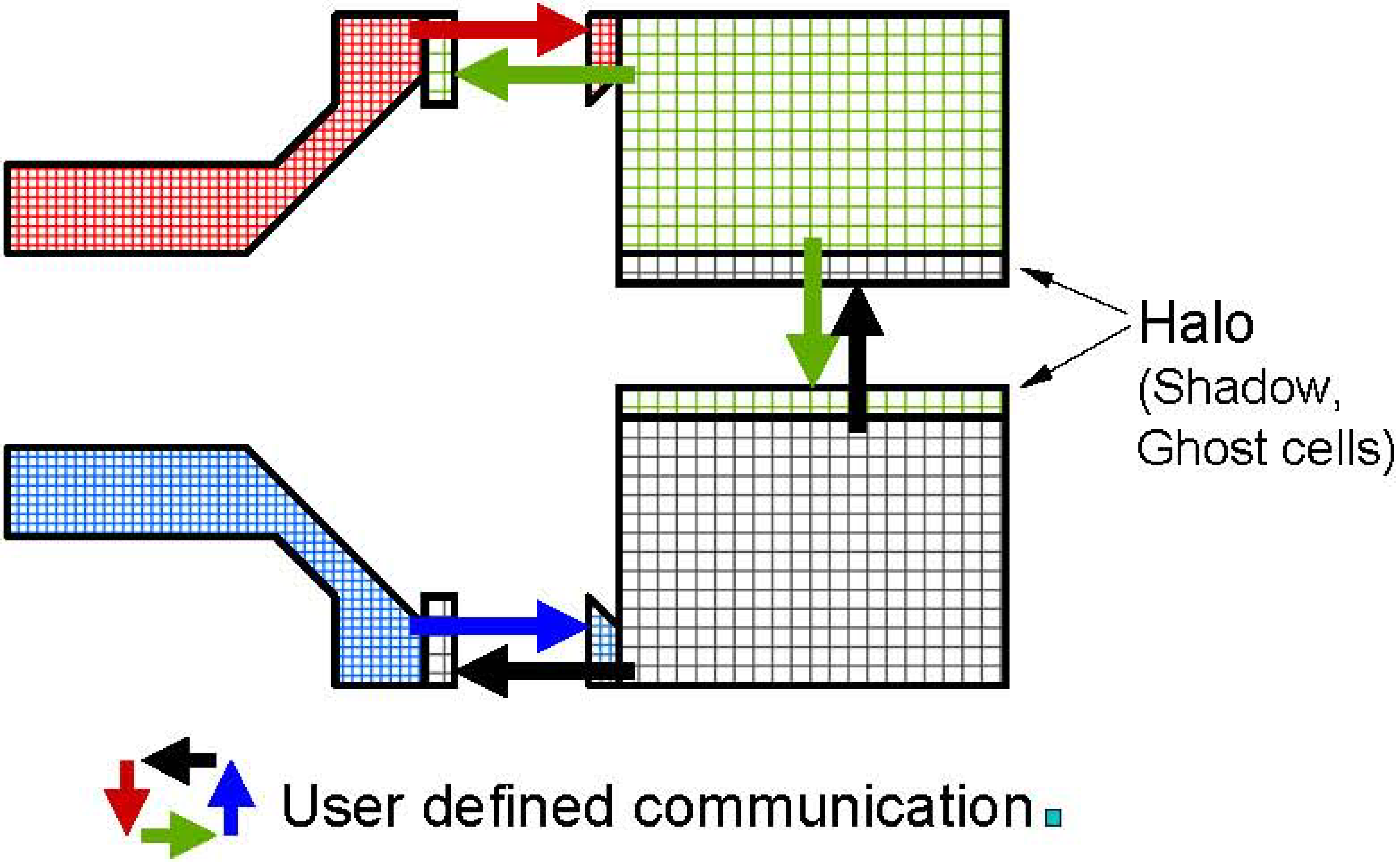
- e.g. b(26) = a(26) + f*(a(25)+a(27)-2*a(26))

a(1:25),	b(1:25)
a(26,50),	b(51,50)
a(51,75),	b(51,75)
a(76,100),	b(76,100)

- e.g. at domain boundaries



Ghost/Halo cells



Pros & Cons (Message Passing)

Advantages:

Memory is **scalable** with the number of processors. If you increase the number of processors, memory size is increased proportionally.

Each processor can rapidly access its memory without overhead caused to cache coherency maintenance

Actual cost: You can use common "off-the-shelf" processors and networks

Disadvantages:

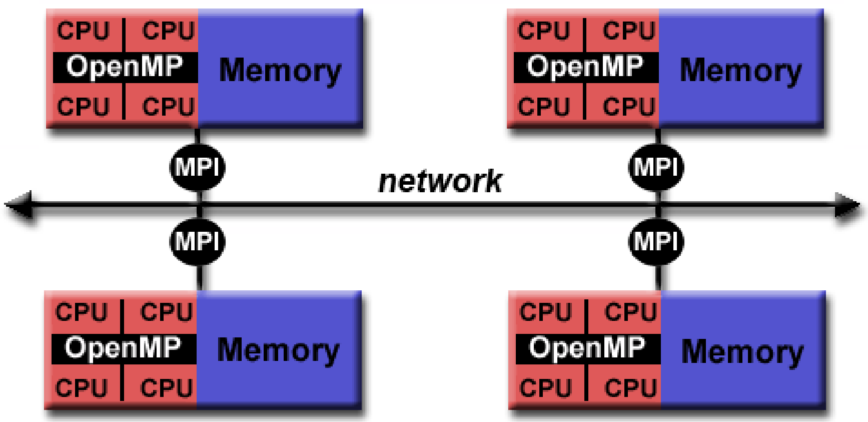
The **programmer is responsible** for most of the details associated with the communication of data between processors

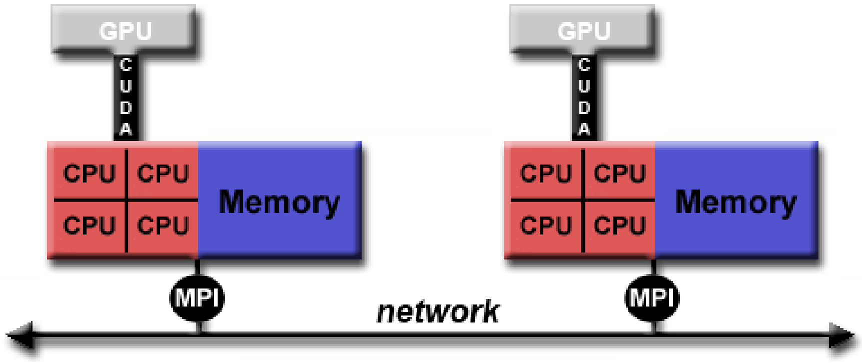
It may be difficult to map existing data structures, due to the global memory in this type of organization.

Non-uniform access times (NUMA)

Hybrid architectures

- **The most powerful computer architectures employ both shared and distributed memory**
- The shared memory component is usually a cache coherent SMP and so processors of a given SMP access a global address memory
- The distributed memory component is the network of SMP
- The current trend is the use of this kind of machines
- **Advantages and disadvantages**: what is common to the various components taken individually
- Joint use MPI – Pthreads - OpenMP (and GPGPU) !





Concepts, methodologies, etc

Recap: (Software) Models in Parallel Programming

- Message Passing
- Data Parallel
- Shared Memory
- Threads
- Hybrid(Ex: OpenMP + MPI)
- SPMD (Single Program Multiple Data)
- MPMD (Multiple Program Multiple Data)

Although not apparent, such models are not specific to the particular type of machine or memory architecture. For example it is possible to emulate Message Passing on Shared Memory architectures and vice versa (although in a very "expensive‟ manner)

Synchronization

Synchronous communication operations

Involves only those tasks that perform a communication operation

When a task executes a communication, a form of coordination is required with other tasks involved in the communication

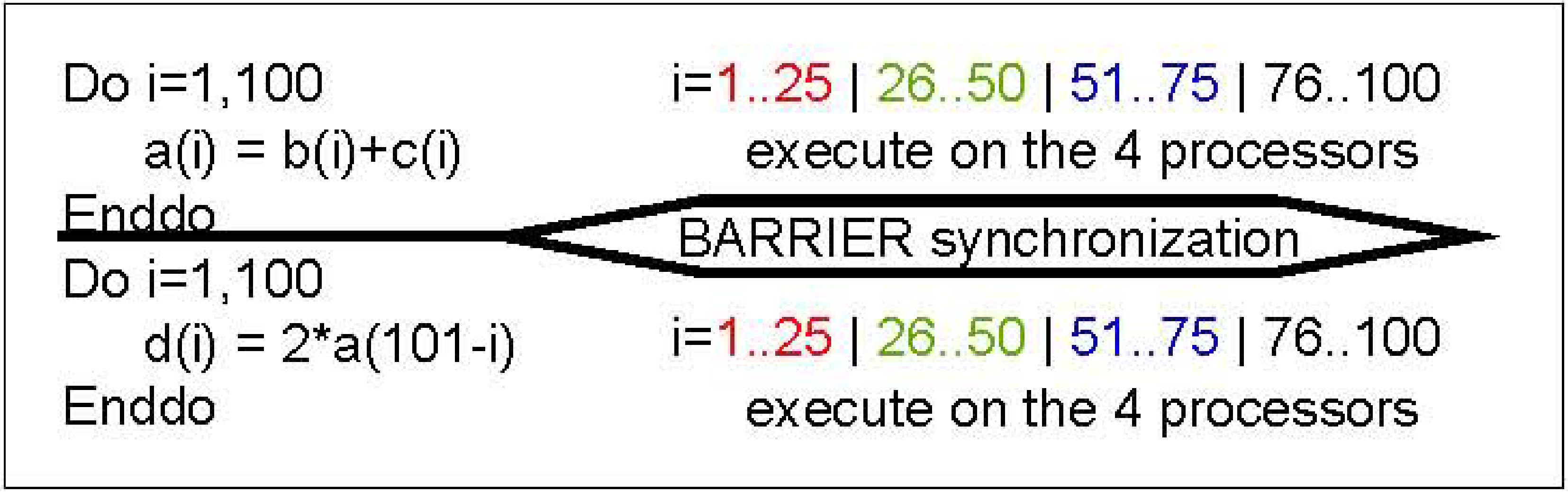
For example, before a task can execute a send operation, it must first receive an acknowledgment from the receiving task, which indicates that the "okay to send ..."

Synchronization

Barrier

- Usually involves all tasks
- Each task executes the code until it "reaches" a barrier. Thereafter, it stops or "freezes"
- When the last task reaches the barrier, all tasks are synchronized

Synchronization



```
Do i=1,100
    a(i) = b(i)+c(i)
Enddo
```

i=1..25 | 26..50 | 51..75 | 76..100

execute on the 4 processors

BARRIER synchronization

```
Do i=1,100
    d(i) = 2*a(101-i)
Enddo
```

i=1..25 | 26..50 | 51..75 | 76..100

execute on the 4 processors

Synchronization

- is necessary
- may cause
  - **idle time on some processors**
  - **overhead to execute the synchronization primitive**

Synchronization

Lock / semaphore

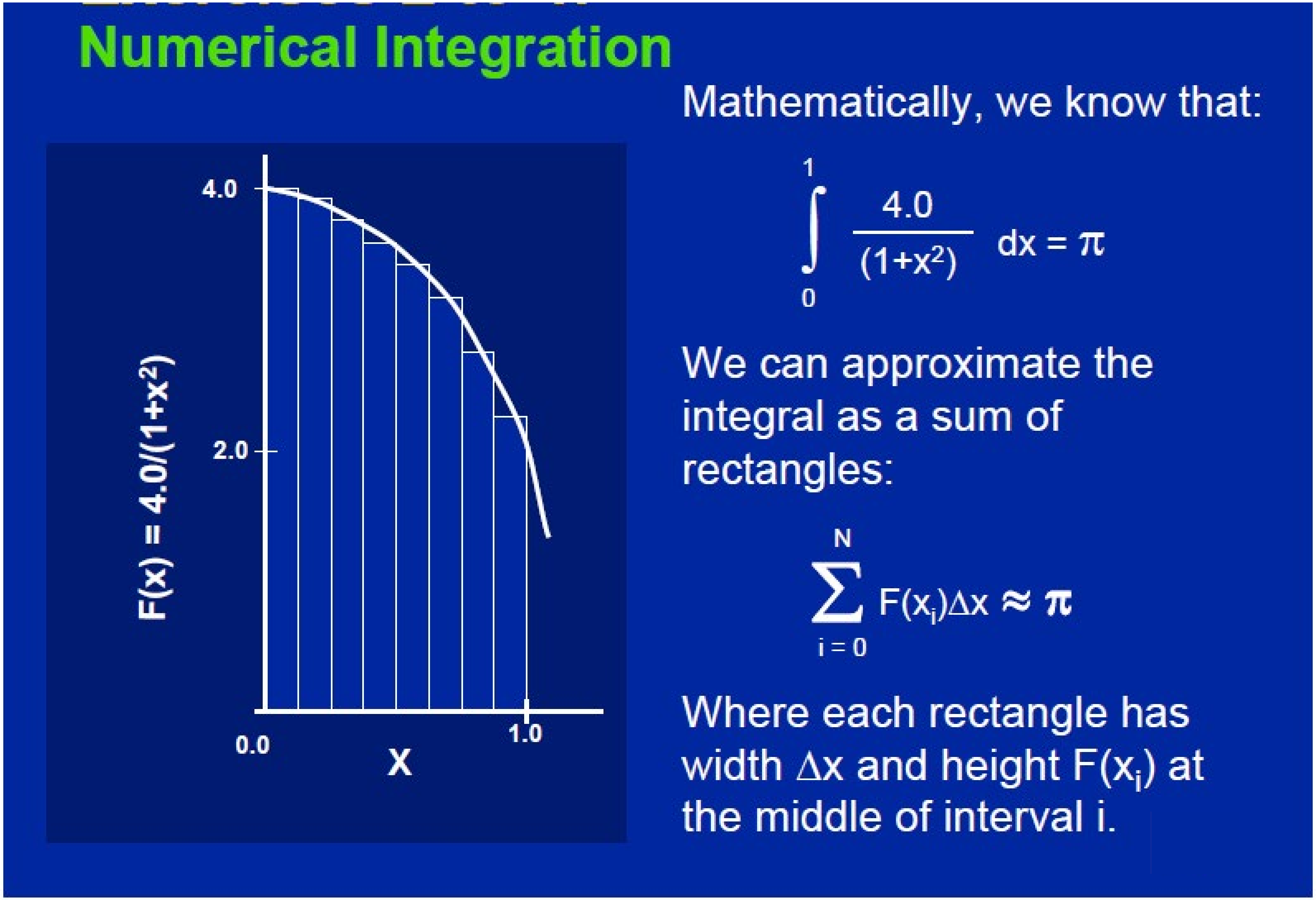
- Involves any number of tasks
- It is typically used to **serialize** (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag
- The first task acquires the lock sets it.
- This task can therefore, in safe mode, access data or code
- Other tasks can try to acquire the lock, but must wait until the task that owns it releases it
- It may be **blocking** or **non-blocking**

Shared Memory Model

- Creation of static or dynamic processes (fork, exec, joins, etc.)

- Coordination between processes is typically done via 3 primitives:
  - Shared variables
  - Critical section (mutex, semaphores, etc)
  - Synchronization (Barrier)

π computation



Numerical Integration

Mathematically, we know that:

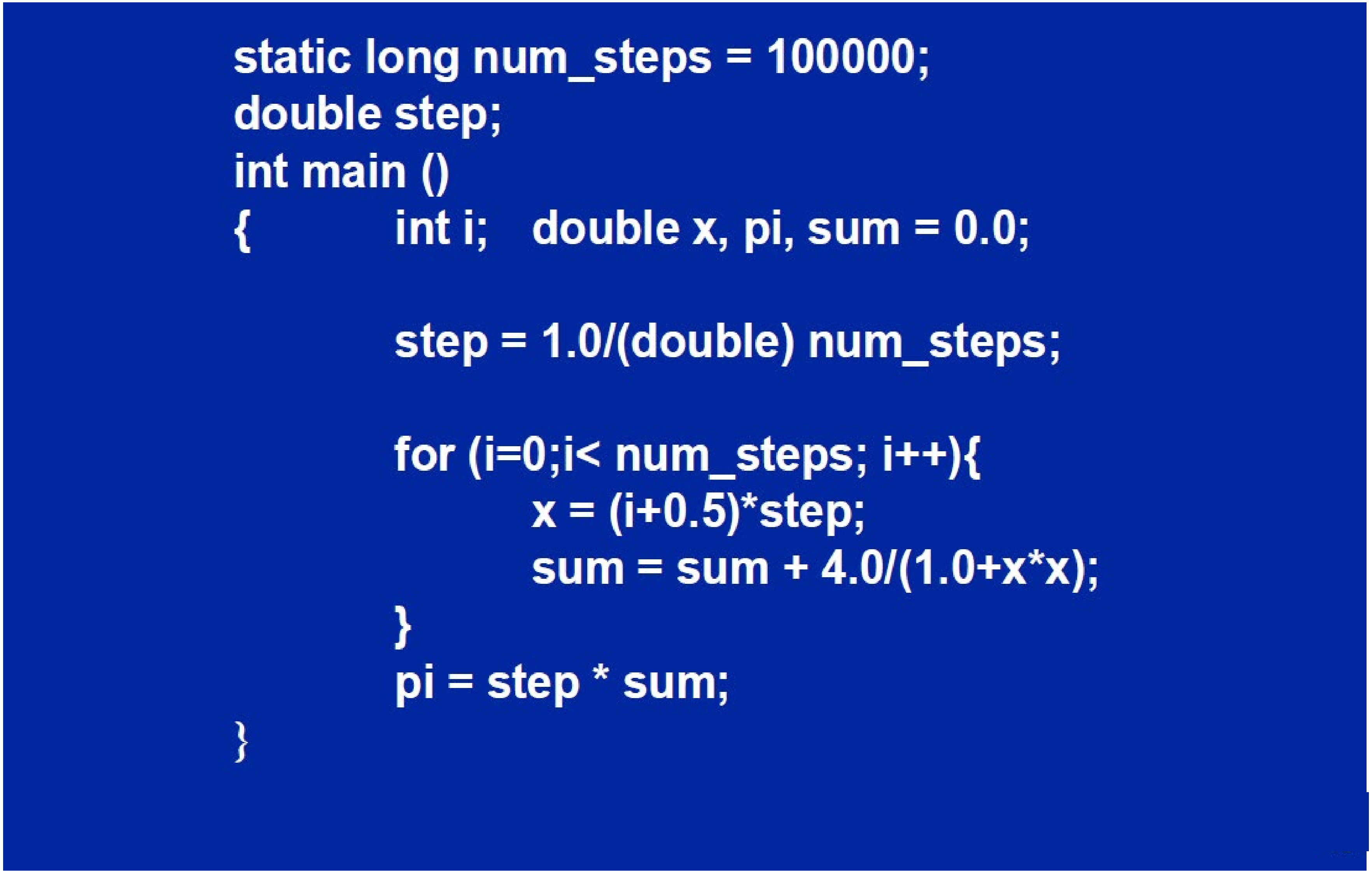
$$\int_{0}^{1} \frac{4.0}{(1+x^2)}\, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial program



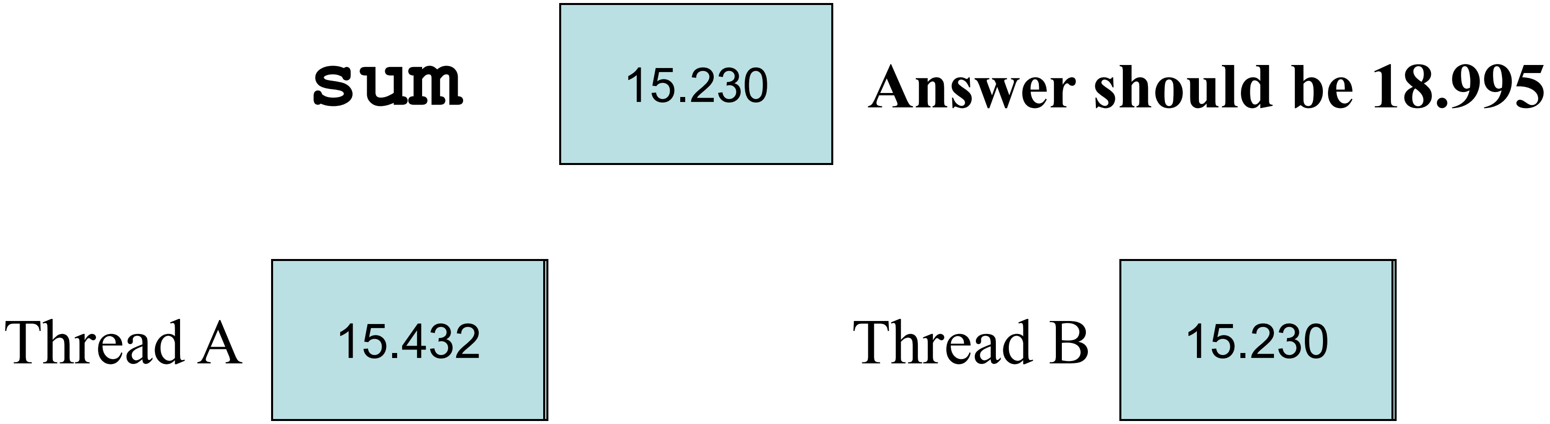
```
static long num_steps = 100000;
double step;
int main ()
{          int i;   double x, pi, sum = 0.0;

           step = 1.0/(double) num_steps;

           for (i=0;i< num_steps; i++){
                    x = (i+0.5)*step;
                    sum = sum + 4.0/(1.0+x*x);
           }
           pi = step * sum;
}
```

Race Condition (cont.)

- ... we set up a **race condition** in which one process may “race ahead” of another and not see its change to shared variable **`sum`**



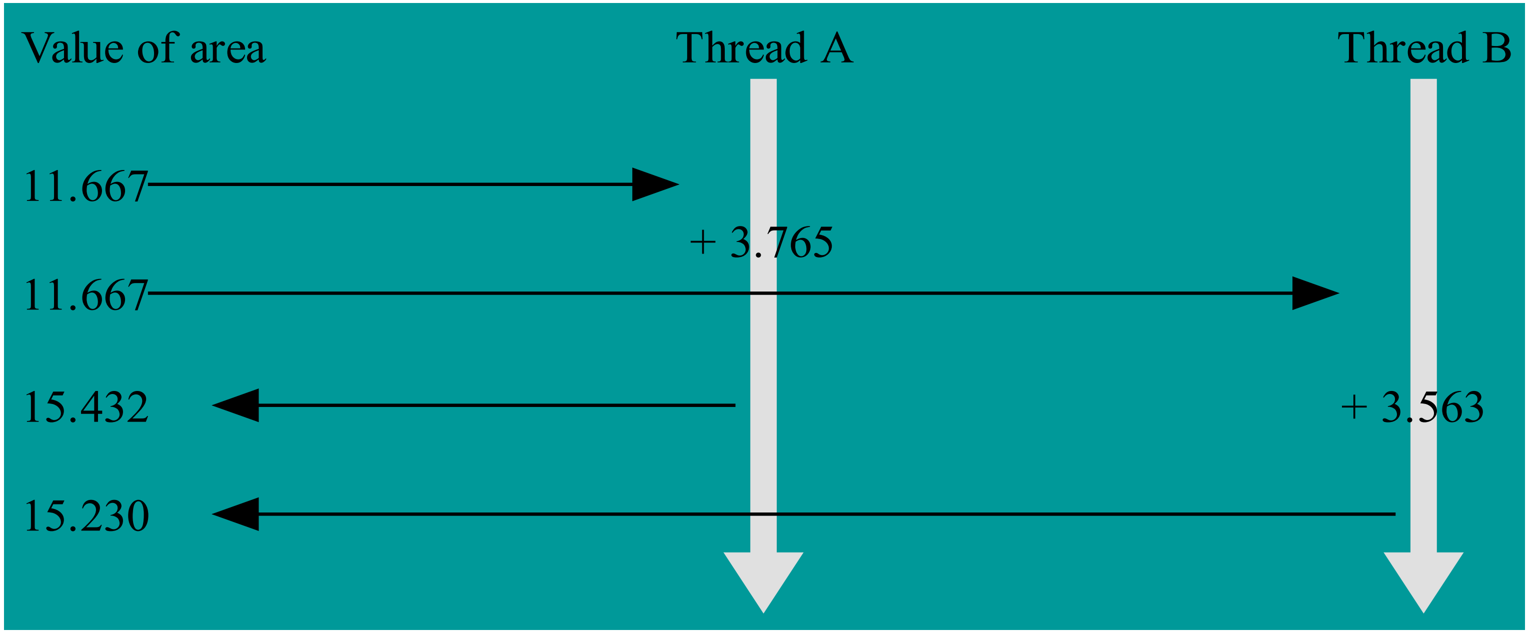
`sum` 15.230 **Answer should be 18.995**

Thread A 15.432

Thread B 15.230

```
sum += 4.0/(1.0 + x*x)
```

Race Condition Time Line



Another example

```
int private_x;  //local variable
shared int sum = 0;
…
sum = sum + private_x;
```

Es: Code executed on each proc

Example: Possible scenario on 2 processes:

Step	**Process 0**	**Process 1**
0	get `sum=0`	end `private_x` computation
1	get `private_x=2`	get `sum=0`
2	sum `0 + 2`	get `private_x=3`
3	save `sum`	sum `0 + 3`
4		save `sum 3`

Sum (`sum`) should be 5 !

Shared Memory Model

- Critical section!
- Binary semaphore!
- The operation `sum = sum + private_x` MUST be performed by a process at a time!

```
shared int s = 1;
while (!s);                 // wait until s=1
s=0;                        // close access
sum = sum + private_x;      // critical sect.
s = 1;                      // reopen access
```

Shared Memory Model

- **Atomicity**
- While a process checks `s = 1` to test if it is okay to enter the critical section, another process might store `s = 0` !
- Two special functions are adopted:

```
void P(int* s);
void V(int* s);
```

- `P` has the effect of preventing other processes from accessing `s` once a process exits the loop
- `V` sets `s` to 1, but ATOMICALLY
- Identified by Dijkstra in 1968. The letter `P` in Dutch stands for **passeren**, which means "go" and the letter `V` comes from the Dutch **vrijgeven**, which means "release"

Shared Memory Model

- Finally, to wait until all the processes have finished before printing, we use a barrier:

```
int private_x;
shared int sum =0;
shared int s = 1;
...
P(&s);
sum = sum + private_x;
V(&s);
Barrier();
if (I am process 0)
      printf("sum = %d\n", sum);
```