

# L7 – INTRODUCTION TO PARALLEL COMPUTING

BMCS3003  
DISTRIBUTED  
SYSTEMS AND  
PARALLEL  
COMPUTING

# CONTENTS

Parallel Computing

Components of Parallel Computing Systems

The needs of Parallelism

Parallel Programming Models

Designing Parallel Programs



## MOTIVATING PARALLELISM

The role of parallelism in accelerating computing speeds has been recognized for several decades.

Its role in providing multiplicity of datapaths and increased access to storage elements has been significant in commercial applications.

The scalable performance and lower cost of parallel platforms is reflected in the wide variety of applications.

---

## FOUR DECADES OF COMPUTING

- Batch Era
- Time sharing Era
- Desktop Era
- Network Era

# BATCH ERA

- Batch processing
  - Is execution of a series of programs on a computer without manual intervention
  - The term originated in the days when users entered programs on punch cards



---

## TIME-SHARING ERA

- **time-sharing** is the sharing of a computing resource among many users by means of multiprogramming and multi-tasking
- Developing a system that supported multiple users at the same time

---

## DESKTOP ERA

- Personal Computers (PCs)
- With WAN



## NETWORK ERA

- Systems with:
  - Shared memory
  - Distributed memory





# FLYNN'S TAXONOMY OF COMPUTER ARCHITECTURE

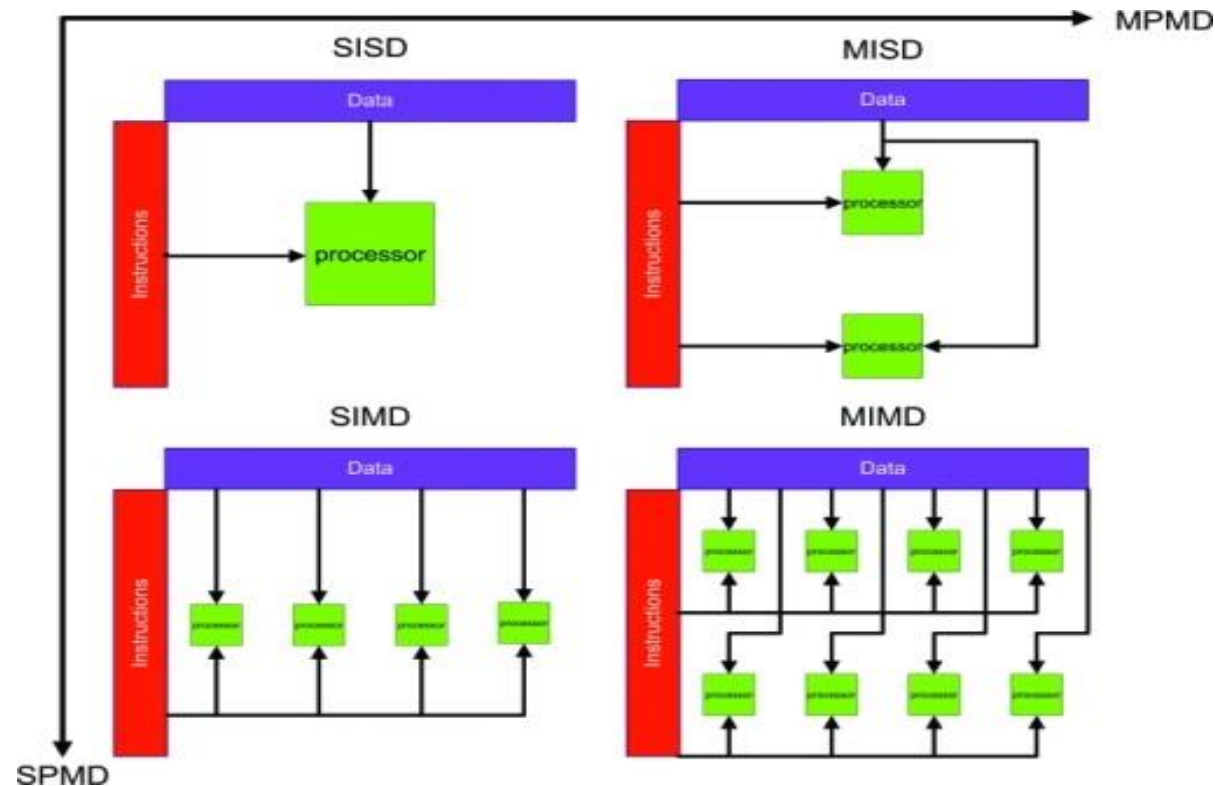
Two types of information flow into processor:

- Instructions
- Data

what are instructions and data?

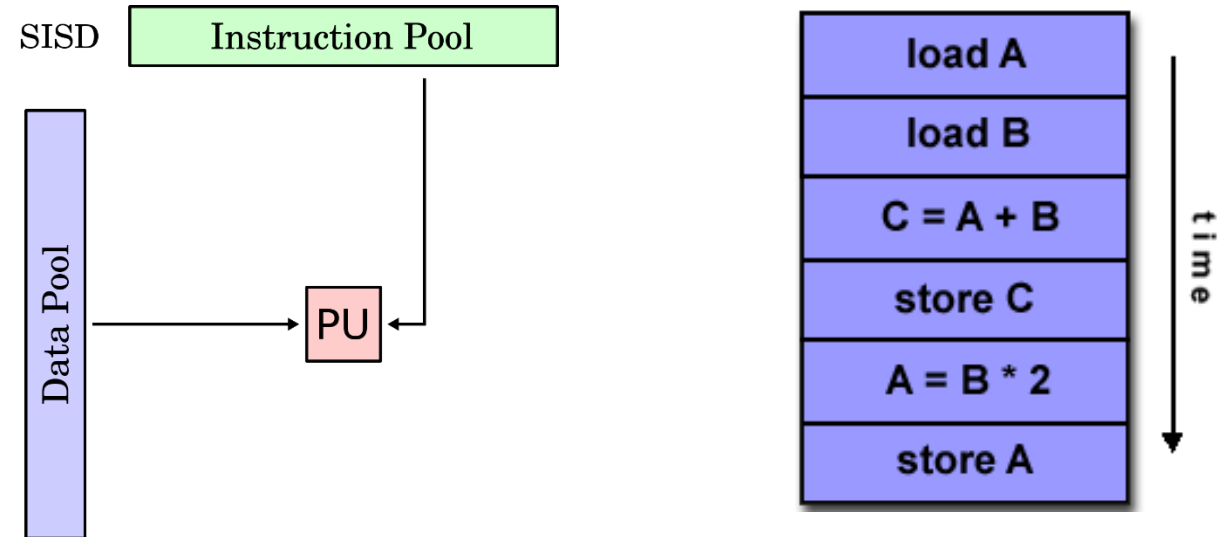
# FLYNN'S TAXONOMY OF COMPUTER ARCHITECTURE

1. single-instruction single-data streams (SISD)
2. single-instruction multiple-data streams (SIMD)
3. multiple-instruction single-data streams (MISD)
4. multiple-instruction multiple-data streams (MIMD)



# SINGLE INSTRUCTION SINGLE DATA (SISD)

- A serial (non-parallel) computer
- This is the oldest type of computer



UNIVAC1



IBM 360



CRAY1



CDC 7600

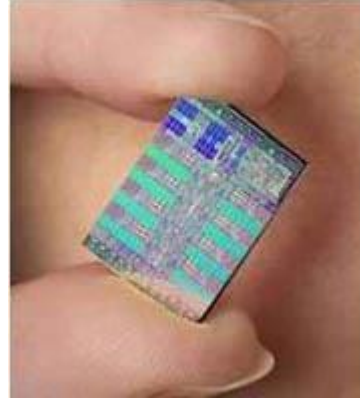


PDP1

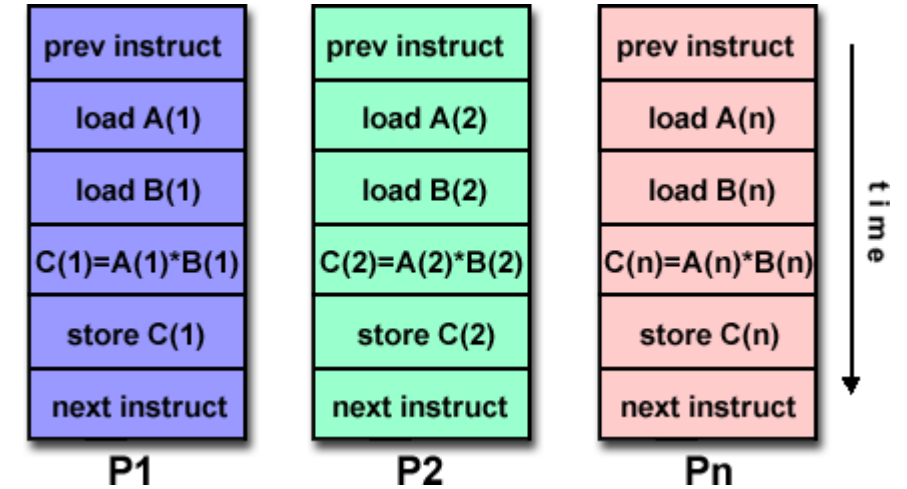
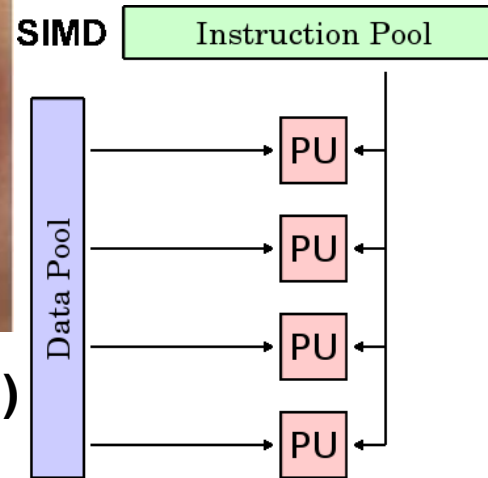
# SINGLE INSTRUCTION MULTIPLE DATA (SIMD)



ILLIAC IV



Cell Processor (GPU)



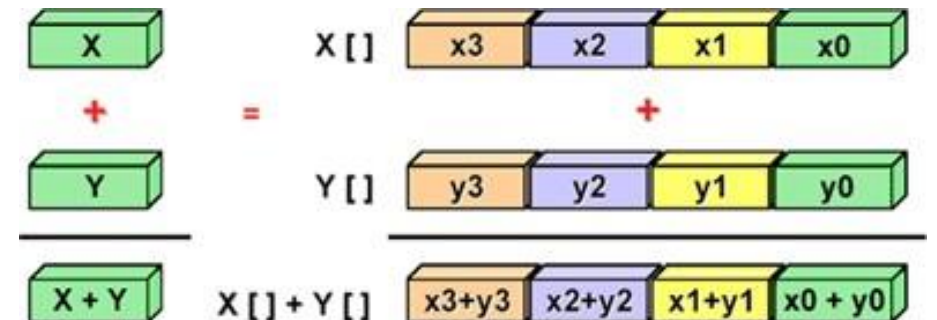
MasPar



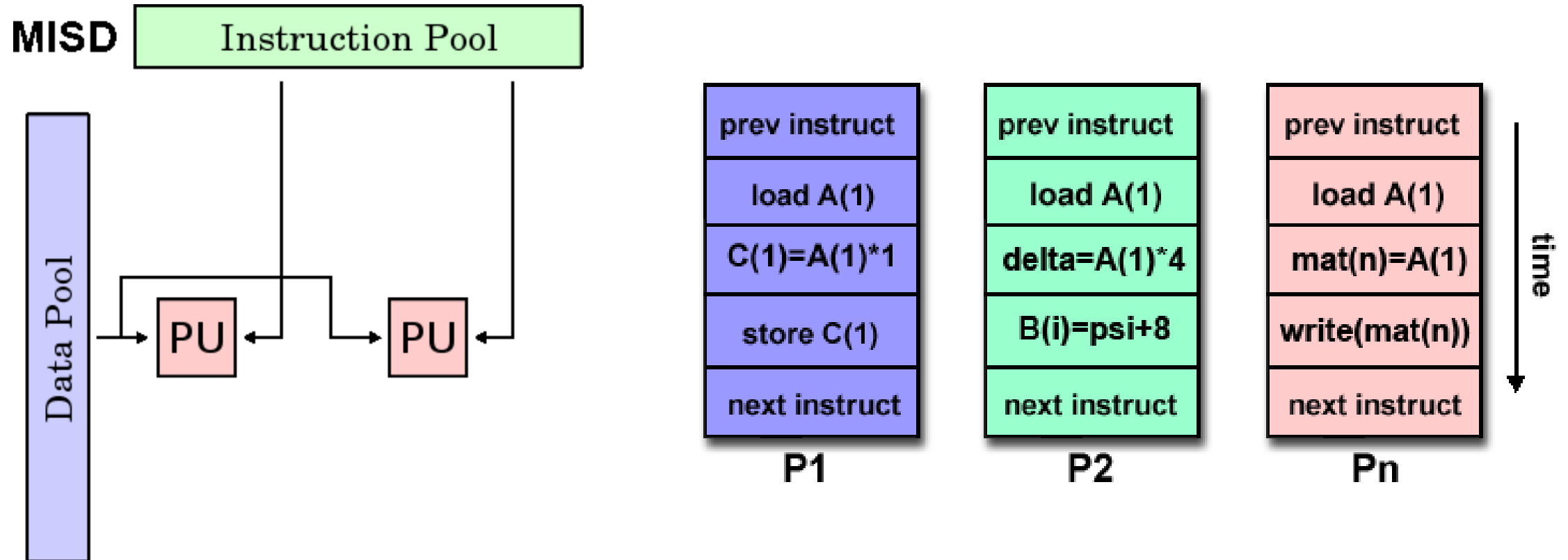
Cray X-MP



Cray Y-MP

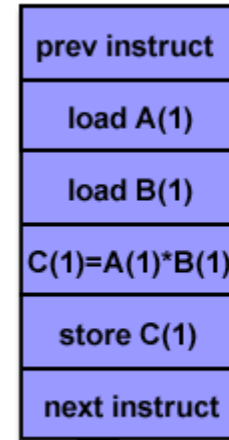
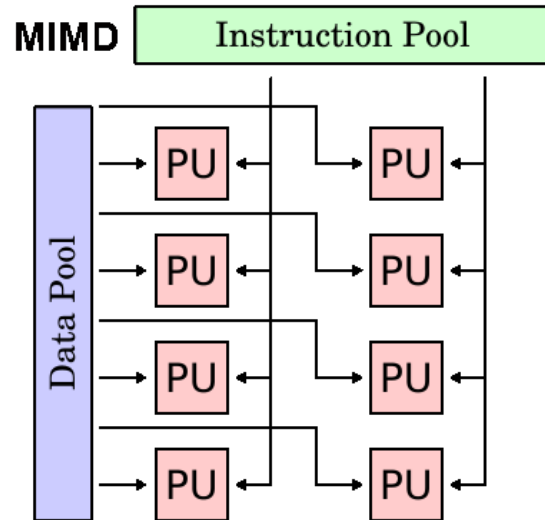


## MULTIPLE INSTRUCTION SINGLE DATA (MISD)

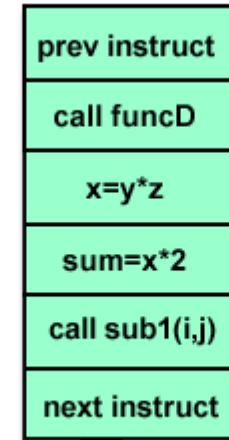


The Space Shuttle flight control computers

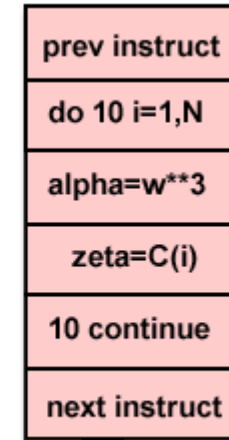
# MULTIPLE INSTRUCTION MULTIPLE DATA (MIMD)



P1



P2



Pn

time

**IBM POWER5**



**AMD Opteron**



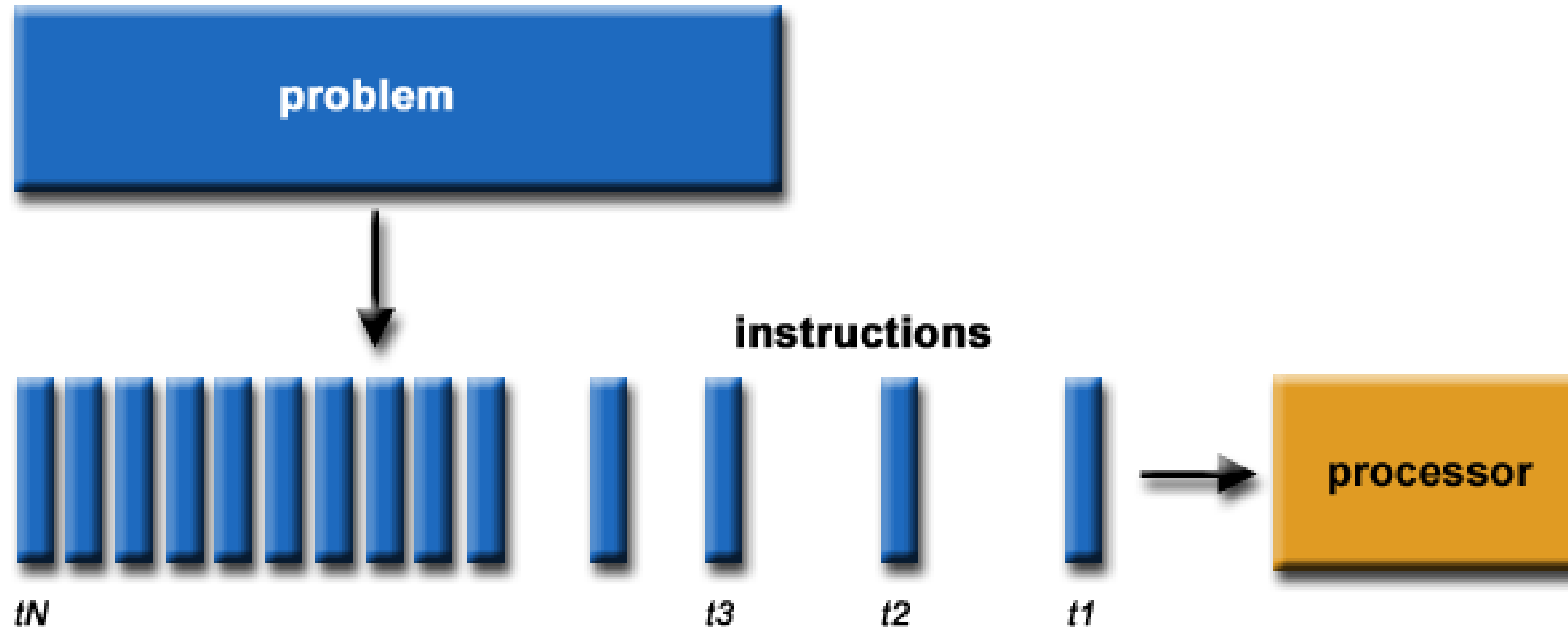
**HP/Compaq Alphaserver**



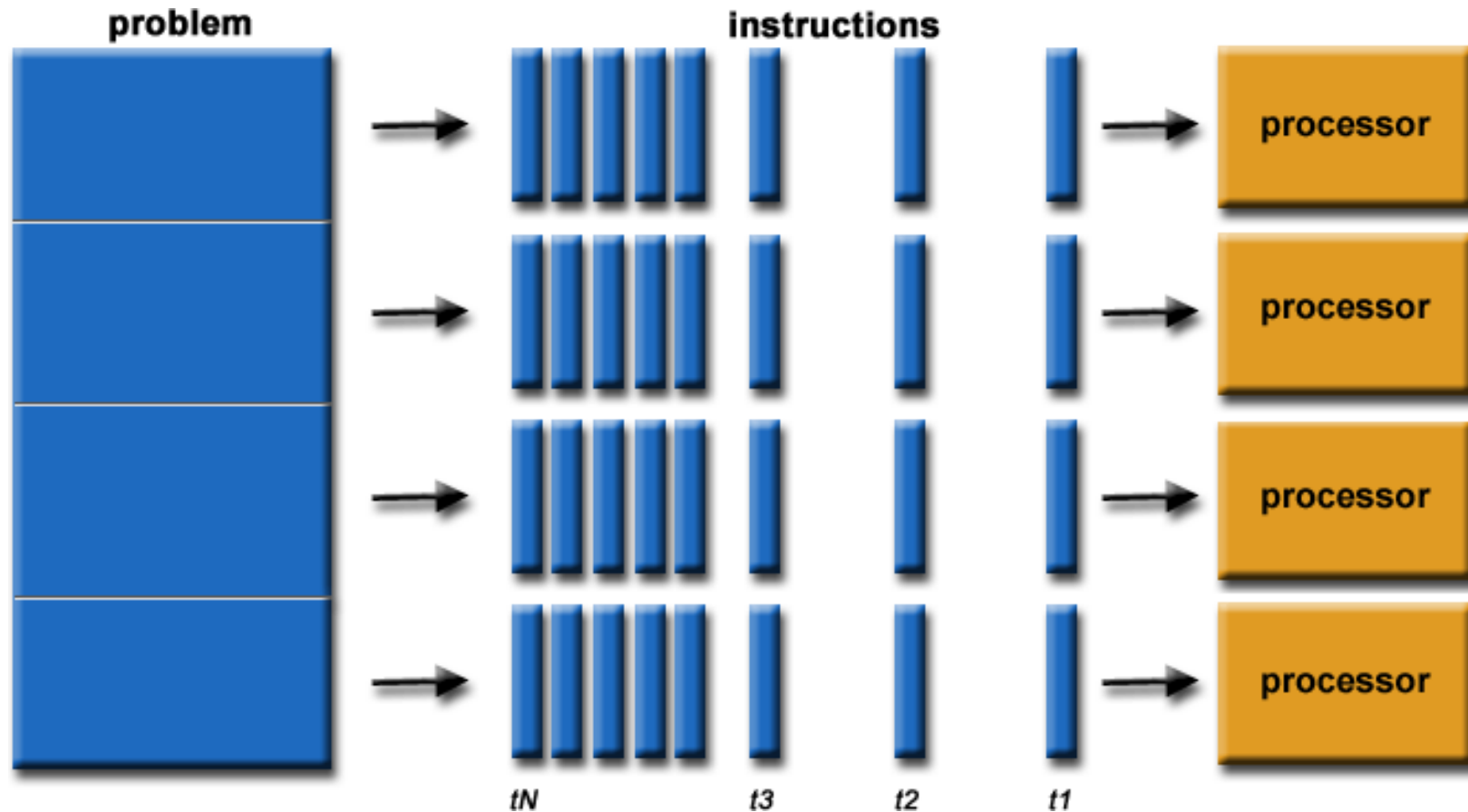
**Intel IA32**

# Parallel computing?

Serial computing

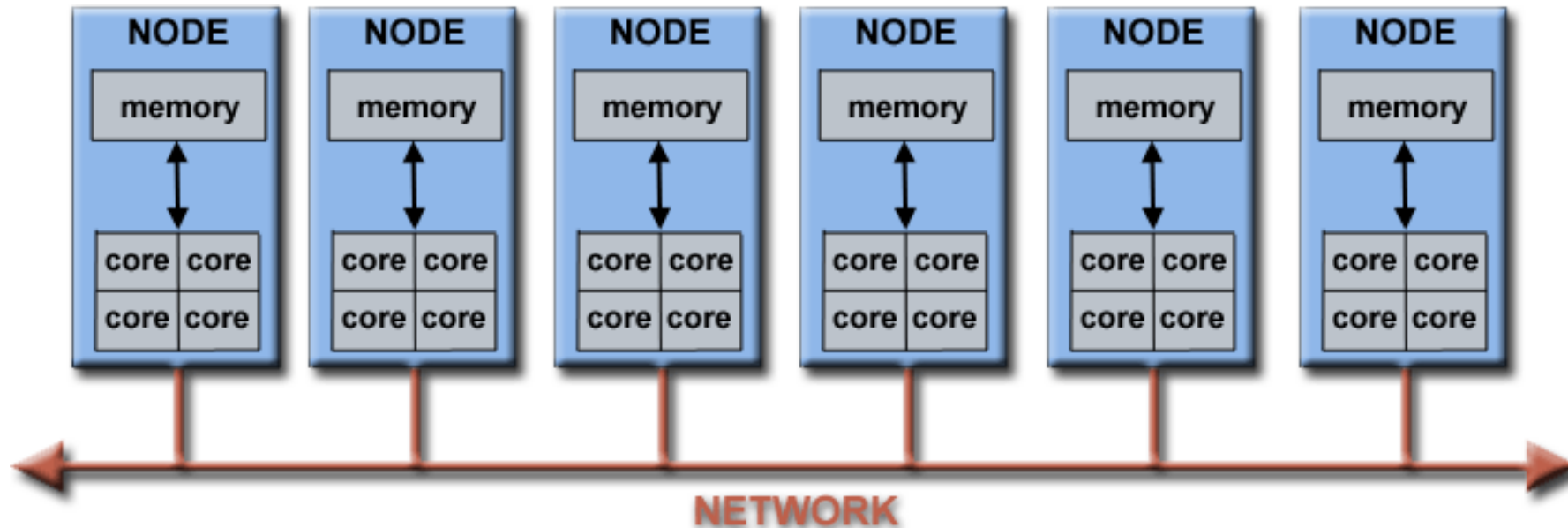


# PARALLEL COMPUTING?



# PARALLEL COMPUTERS

- Networks connect multiple stand-alone computers (nodes) to make larger parallel computer clusters.



# WHY USE PARALLEL COMPUTING?

- SAVE TIME AND/OR MONEY:



# WHY USE PARALLEL COMPUTING?

- SOLVE LARGER / MORE COMPLEX PROBLEMS



# WHY USE PARALLEL COMPUTING?

- PROVIDE CONCURRENCY



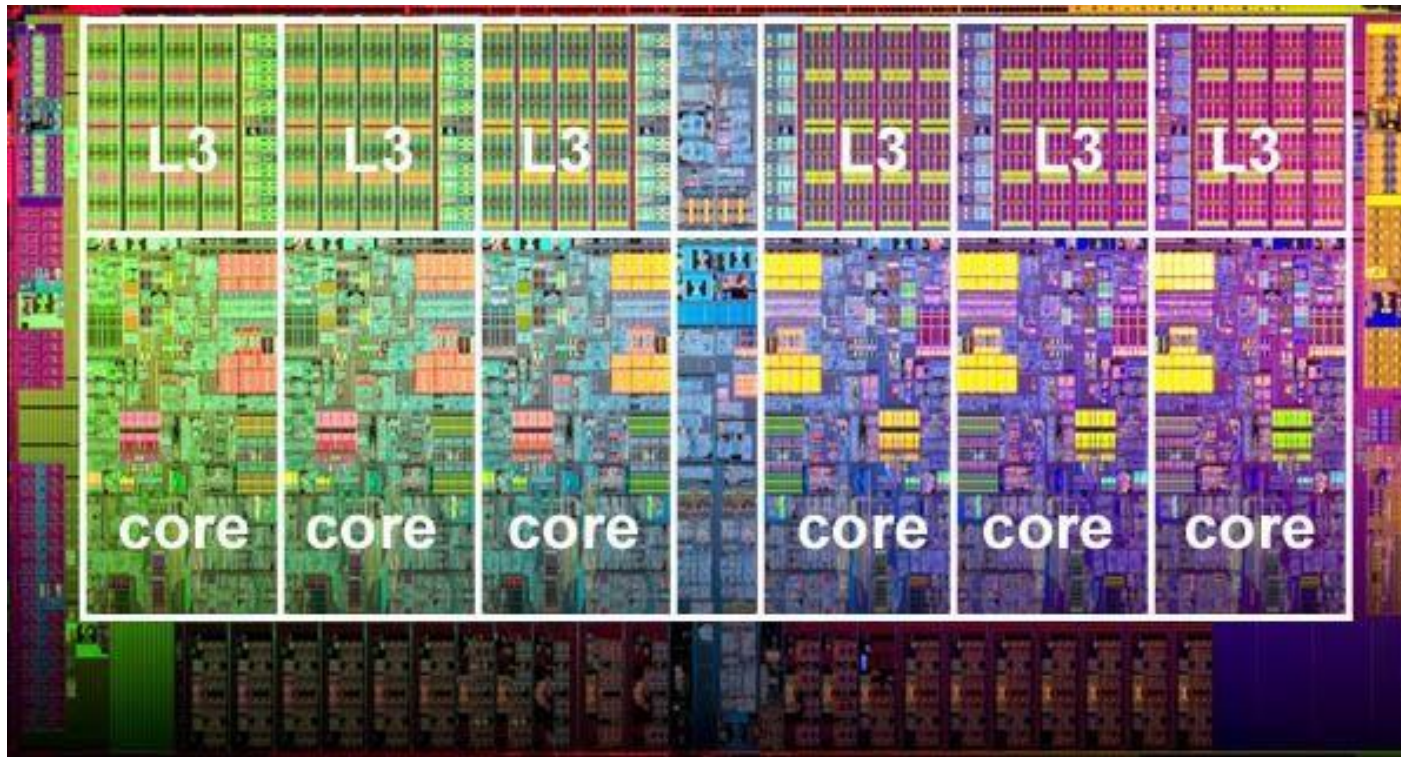
---

## WHY USE PARALLEL COMPUTING?

- TAKE ADVANTAGE OF NON-LOCAL RESOURCES:

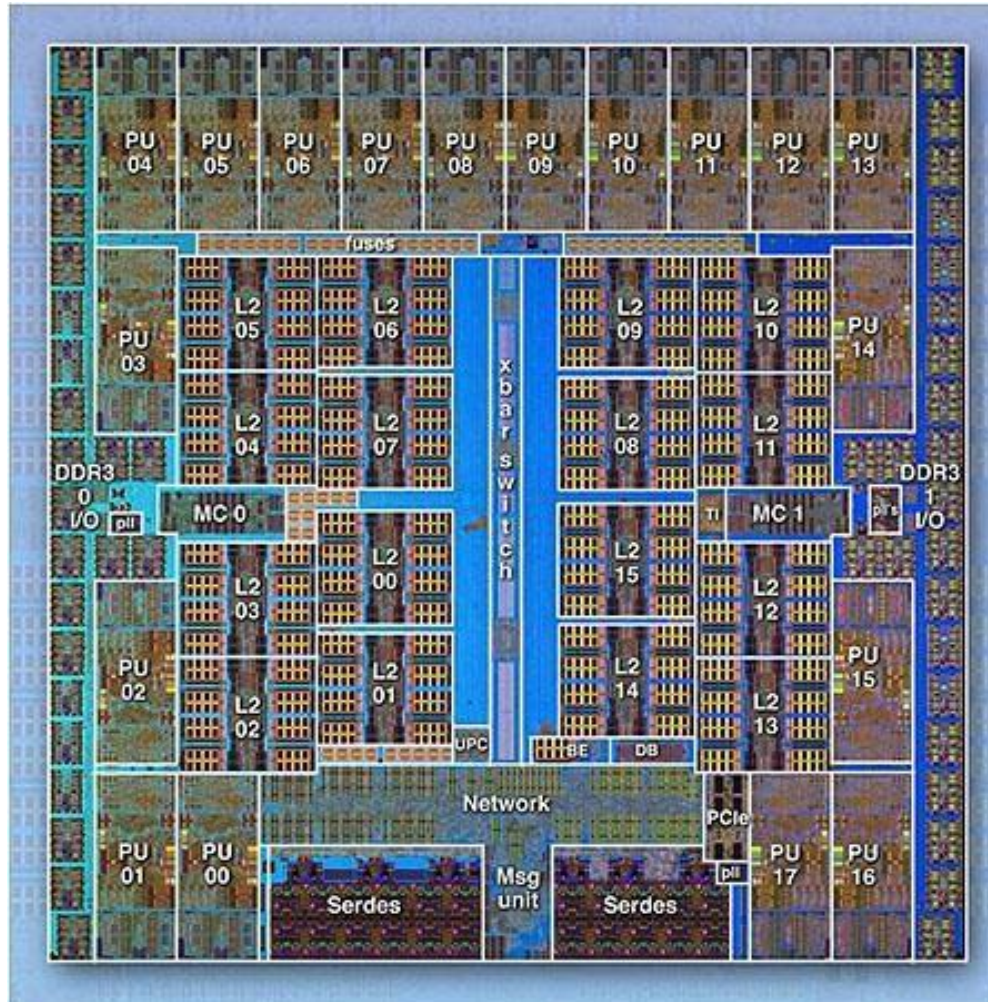
# WHY USE PARALLEL COMPUTING?

- MAKE BETTER USE OF UNDERLYING PARALLEL HARDWARE
  - Modern computers, even laptops, are parallel in architecture with multiple processors/cores



# PARALLEL COMPUTERS

- all stand-alone computers today are parallel from a hardware perspective



# MOTIVATING PARALLELISM

Developing parallel hardware and software has traditionally been time and effort intensive.

If one is to view this in the context of rapidly improving uniprocessor speeds, one is tempted to question the need for parallel computing.

There are some unmistakable trends in hardware design, which indicate that uniprocessor (or implicitly parallel) architectures may not be able to sustain the rate of *realizable* performance increments in the future.

This is the result of a number of fundamental physical and computational limitations.

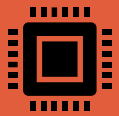
The emergence of standardized parallel programming environments, libraries, and hardware have significantly reduced time to (parallel) solution.



# IMPLICIT PARALLELISM: TRENDS IN MICROPROCESSOR ARCHITECTURES

- Microprocessor clock speeds have posted impressive gains over the past two decades (two to three orders of magnitude).
- Higher levels of device integration have made available a large number of transistors.
- The question of how best to utilize these resources is an important one.
- Current processors use these resources in multiple functional units and execute multiple instructions in the same cycle.
- The precise manner in which these instructions are selected and executed provides impressive diversity in architectures.

## LEVEL OF PARALLELISM -JOB



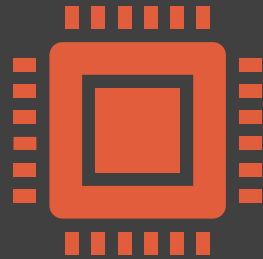
Full jobs are running completely in parallel on different processors with weak or no interaction between those jobs.

improved throughput of computer  
shorter real time of jobs,



e.g.: workstation with multiple processors and multitasking.

## LEVEL OF PARALLELISM - PROGRAM



Parts of one program are  
running on multiple processors.  
shorter real time.

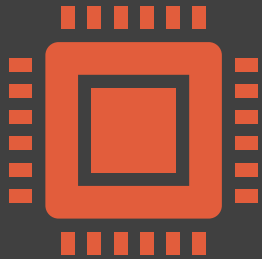


e.g.: parallel computer

# LEVEL OF PARALLELISM - INSTRUCTION

- Parallelization between phases of execution.
  - Accelerated execution of the whole instruction.
- e.g.: sequential computer / single processor.

## LEVEL OF PARALLELISM - ARITHMETIC, BIT LEVEL



Hardware inherent parallelism of integer arithmetic. Bit-wise parallel access but word-wise sequential access and vice versa.

Less clocks for executing an instruction.

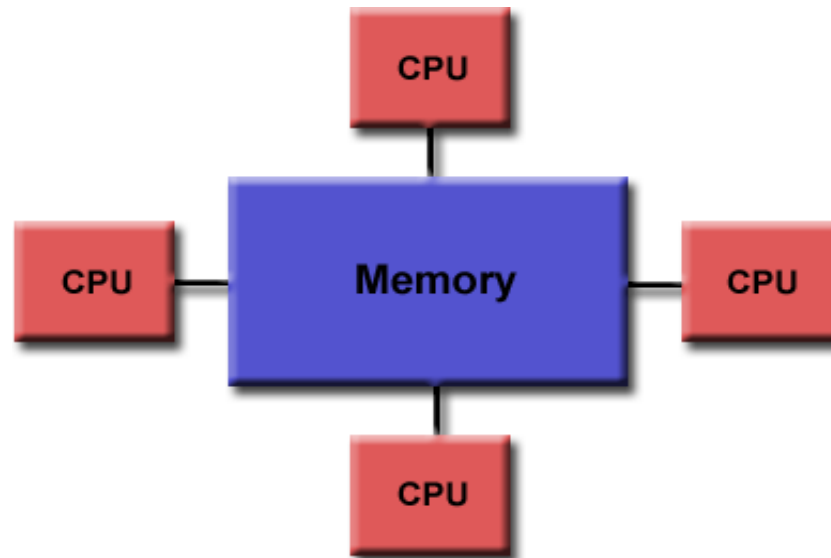


e.g.: superscalar processors, 32/64 bit bus system

# PARALLEL COMPUTER MEMORY ARCHITECTURES

## - SHARED MEMORY

### Uniform Memory Access (UMA)

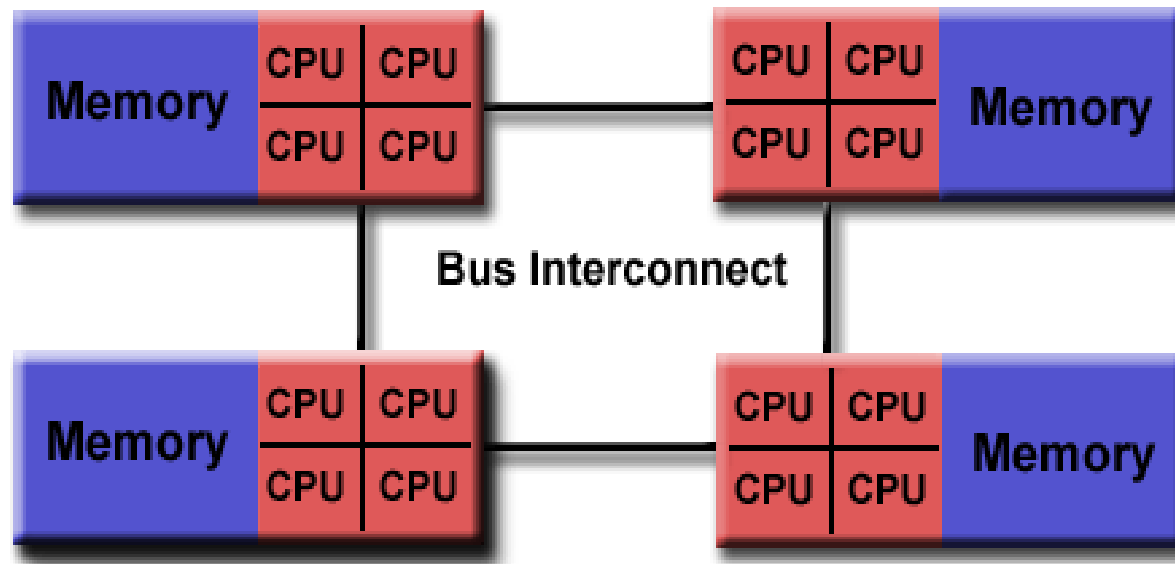


- Equal accesses and access times to memory
- Most commonly represented today by ***Symmetric Multiprocessor (SMP)*** machines

# PARALLEL COMPUTER MEMORY ARCHITECTURES

## - SHARED MEMORY

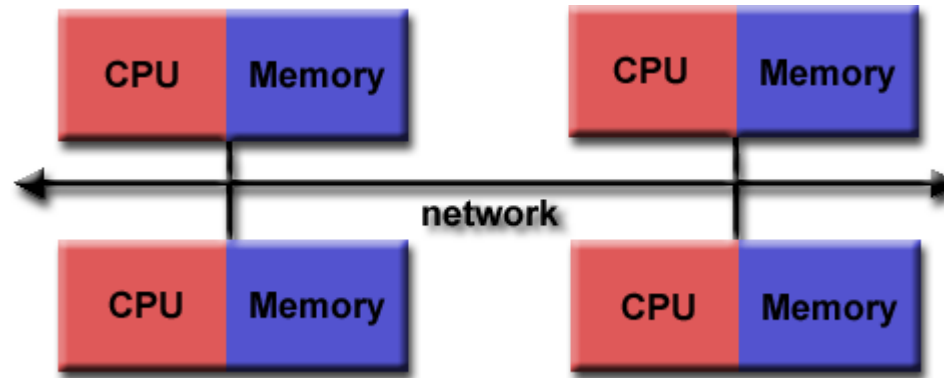
Non - Uniform Memory Access (NUMA)



- Not all processors have equal memory access time

# PARALLEL COMPUTER MEMORY ARCHITECTURES

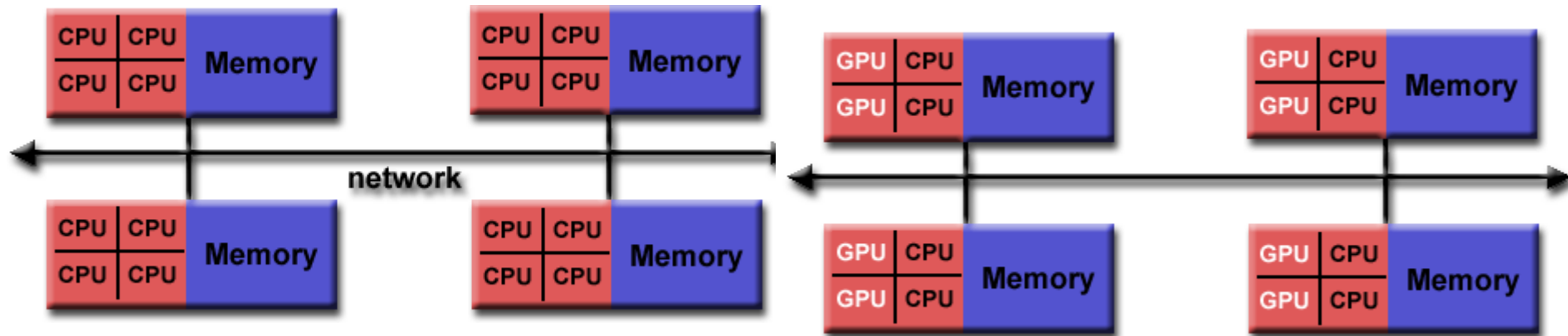
## - DISTRIBUTED MEMORY



- Processors have own memory  
(There is no concept of global address space)
- It operates independently
- communications in message passing systems are performed via **send** and **receive** operations

# PARALLEL COMPUTER MEMORY ARCHITECTURES

## – Hybrid Distributed-Shared Memory



- Use in largest and Fastest computers in the world today



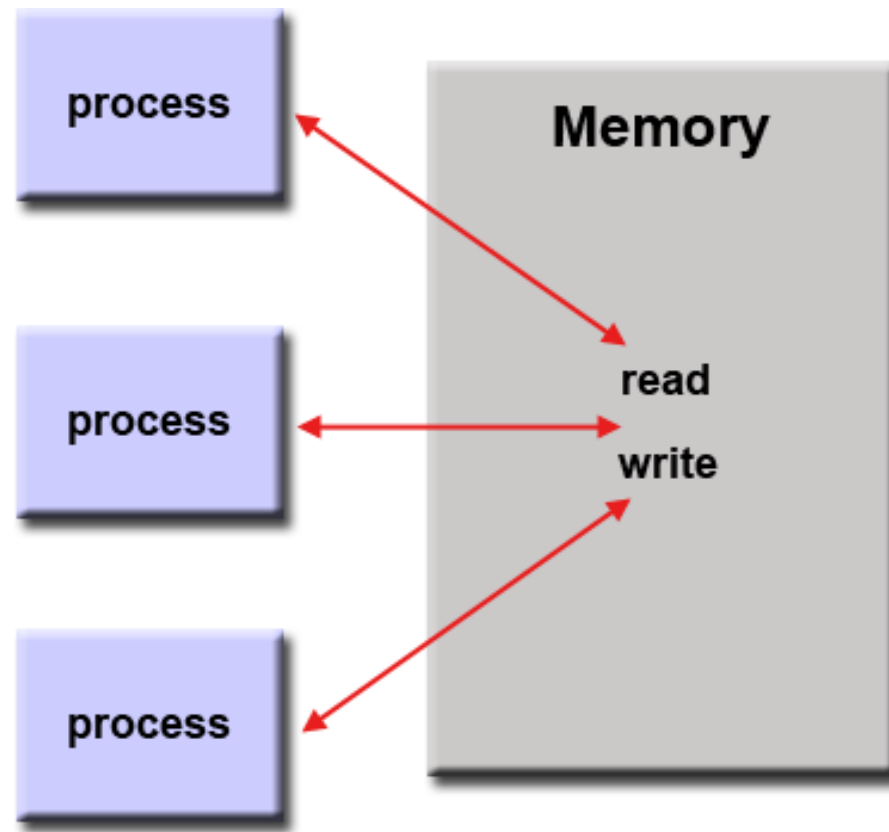
# PARALLEL PROGRAMMING MODELS



# PARALLEL PROGRAMMING MODELS

## Shared Memory Model (without threads)

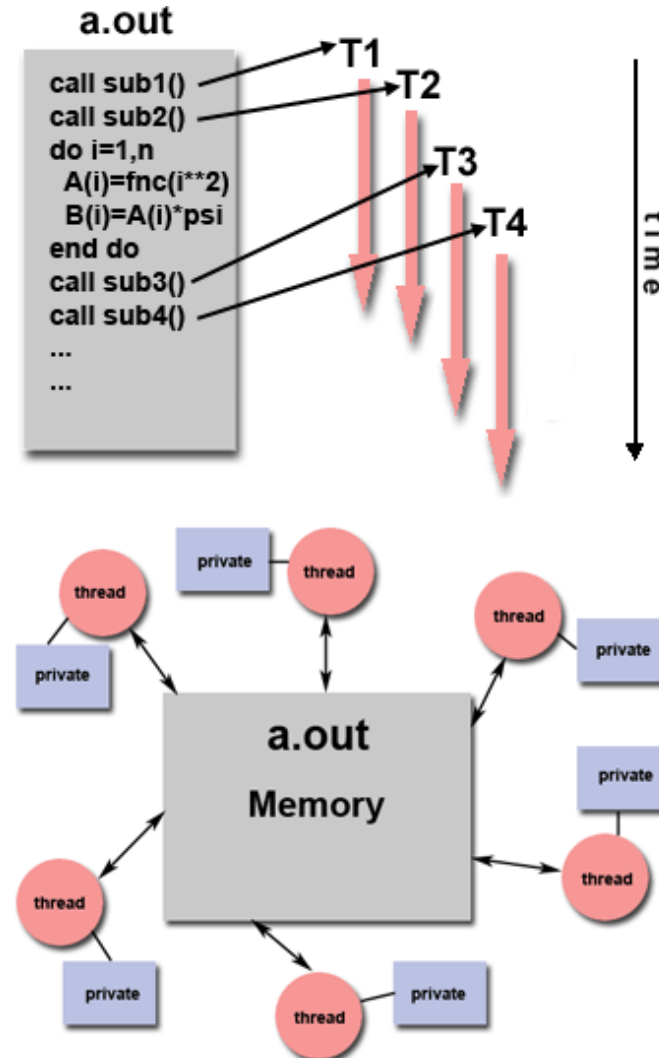
- In this programming model, processes/tasks share a common address space, which they read and write to asynchronously.



# PARALLEL PROGRAMMING MODELS

## Threads Model

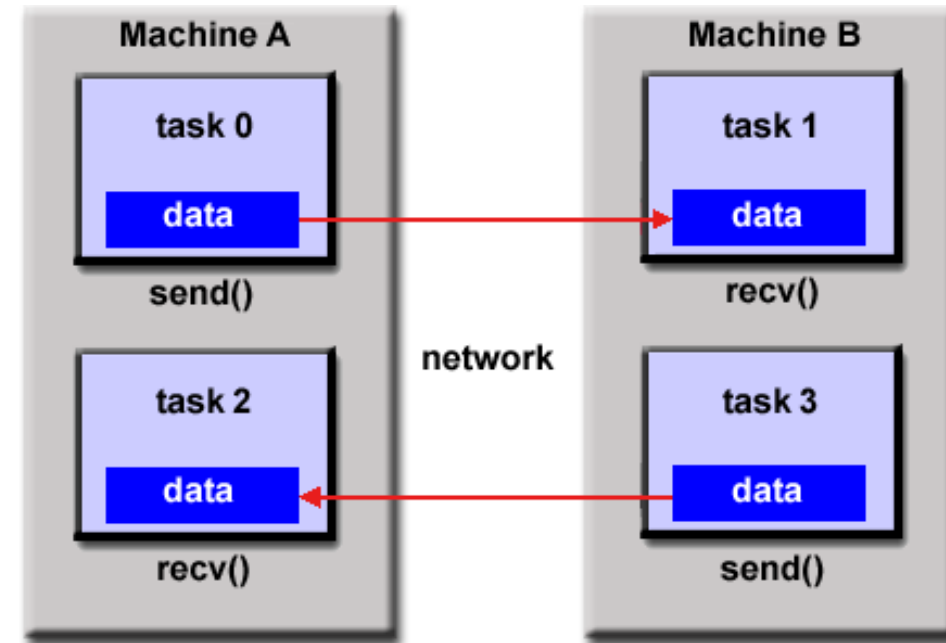
- This programming model is a type of shared memory programming.
- In the threads model of parallel programming, a single "heavy weight" process can have multiple "light weight", concurrent execution paths.
- Ex: **POSIX Threads, OpenMP, Microsoft threads, Java Python threads, CUDA threads for GPUs**



# PARALLEL PROGRAMMING MODELS

## Distributed Memory / Message Passing Model

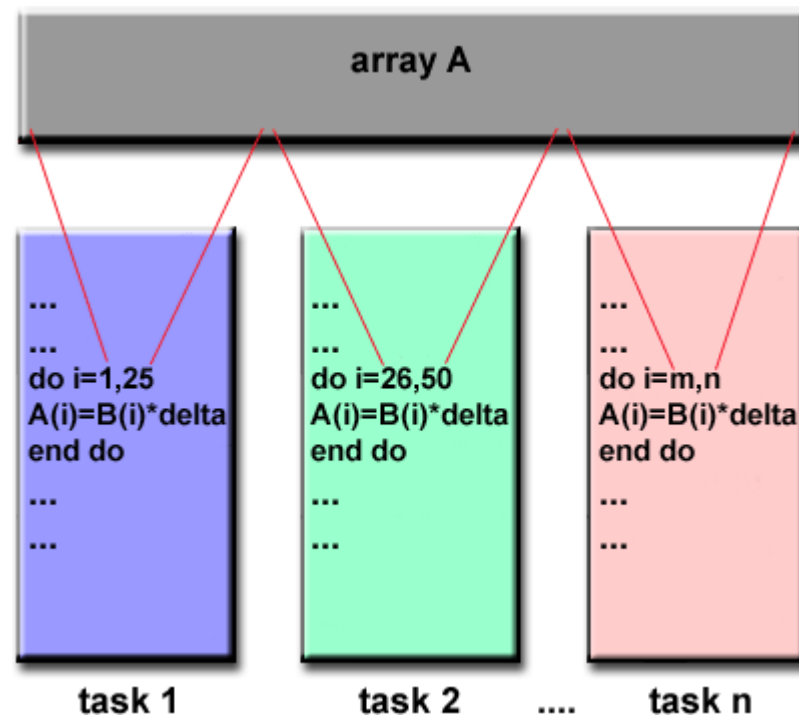
- A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.
- Tasks exchange data through communications by sending and receiving messages.
- Ex:
- Message Passing Interface (MPI)



# PARALLEL PROGRAMMING MODELS

## Data Parallel Model

- May also be referred to as the **Partitioned Global Address Space (PGAS)** model.
- Ex: Coarray Fortran, Unified Parallel C (UPC), X10



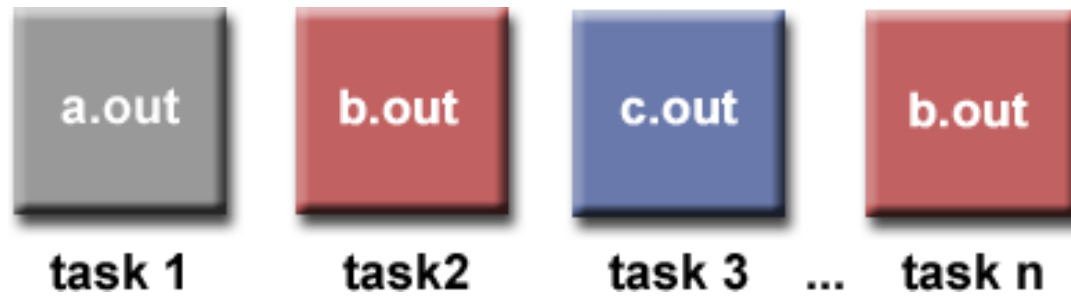
# PARALLEL PROGRAMMING MODELS

## SPMD and MPMD

Single Program Multiple Data (SPMD)



Multiple Program Multiple Data (MPMD)





# DESIGNING PARALLEL PROGRAMS



# DESIGNING PARALLEL PROGRAMS

## Automatic vs. Manual Parallelization

- **Fully Automatic**
  - The compiler analyzes the source code and identifies opportunities for parallelism.
  - The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance.
  - Loops (do, for) are the most frequent target for automatic parallelization.
- **Programmer Directed**
  - Using "compiler directives" or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code.
  - May be able to be used in conjunction with some degree of automatic parallelization also.

# DESIGNING PARALLEL PROGRAMS

## Understand the Problem and the Program

- **Easy to parallelize problem**
  - Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.
- **A problem with little-to-no parallelism**
  - Calculation of the Fibonacci series (0,1,1,2,3,5,8,13,21,...) by use of the formula:
  - $F(n) = F(n-1) + F(n-2)$

---

# DESIGNING PARALLEL PROGRAMS

## Partitioning

- One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as decomposition or partitioning.

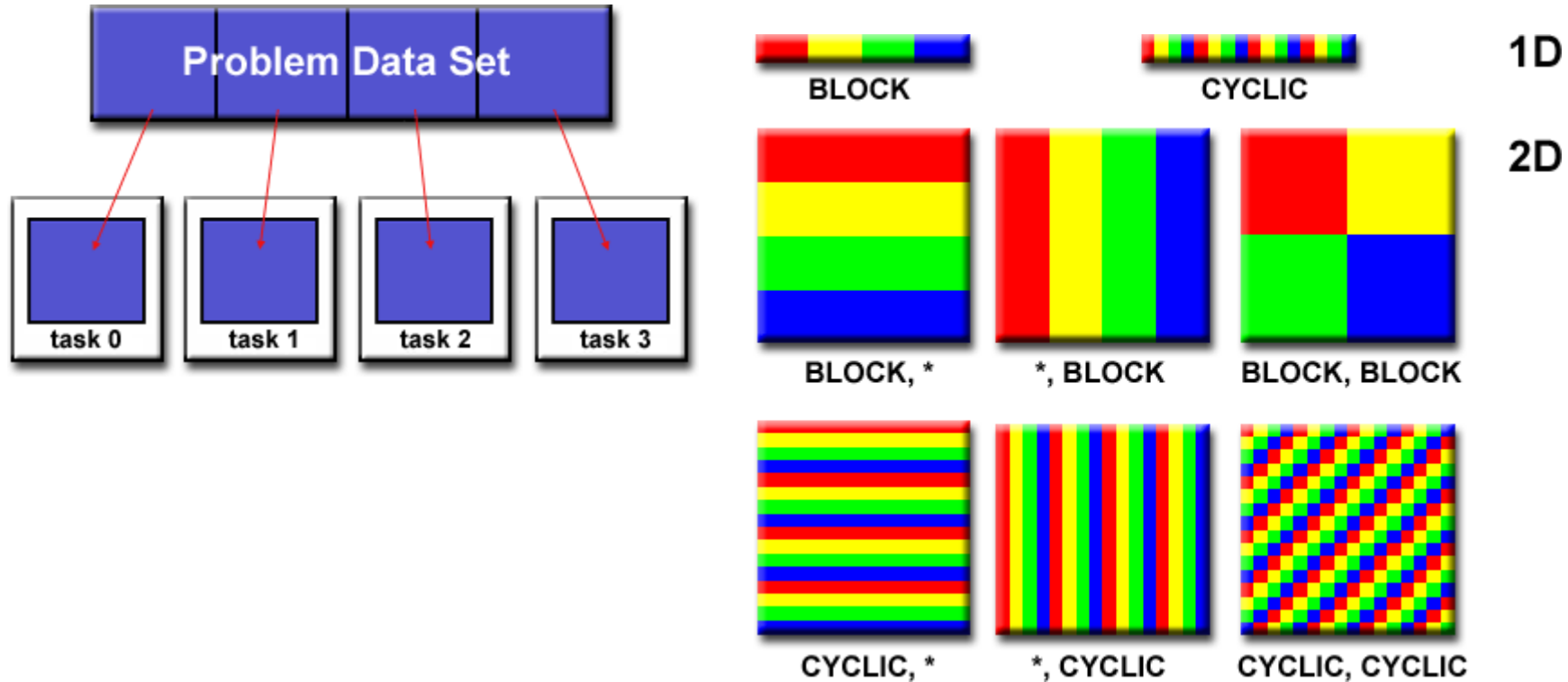
Two ways:

- **Domain decomposition**
- **Functional decomposition**

# DESIGNING PARALLEL PROGRAMS

## Domain Decomposition

The data associated with a problem is decomposed

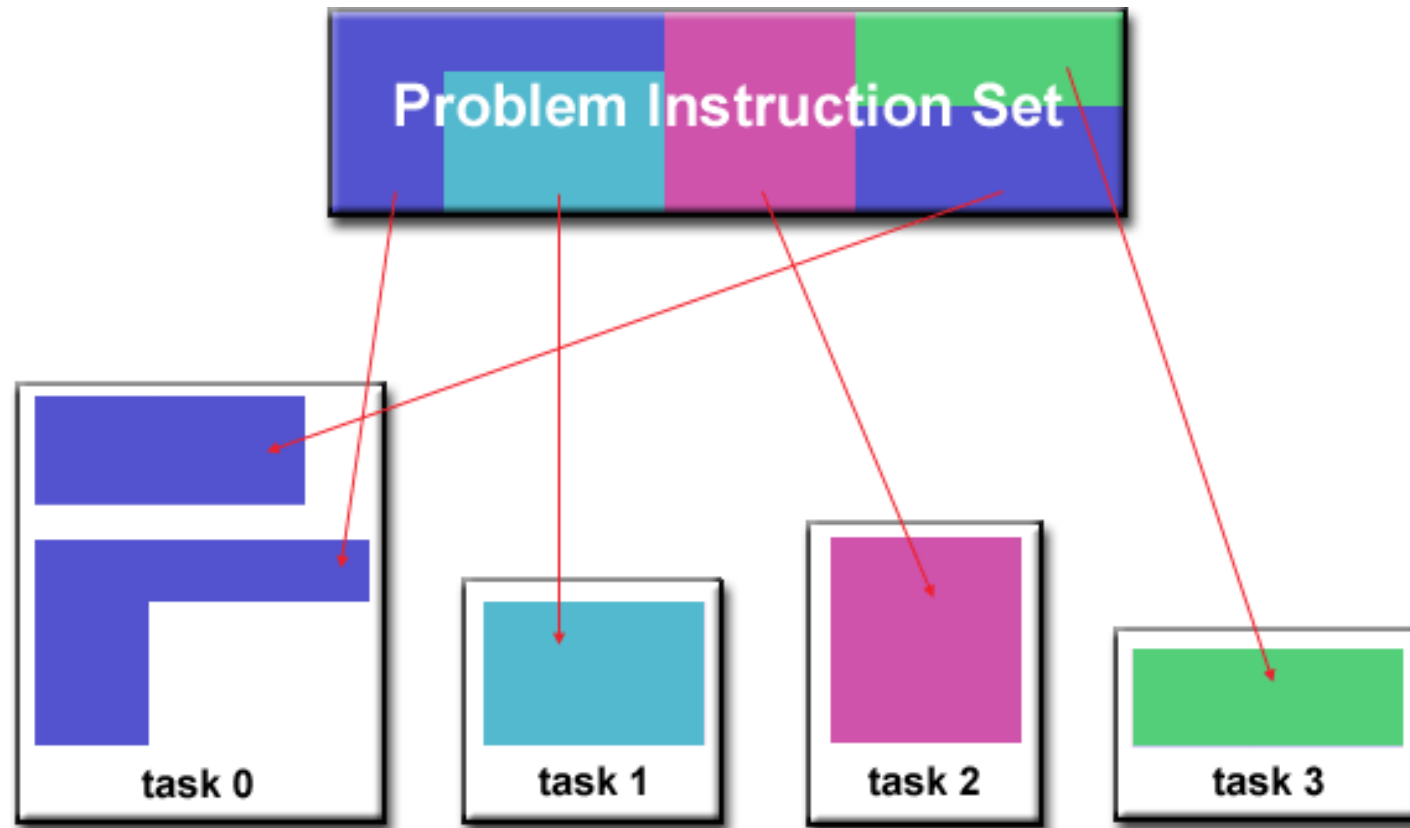


There are different ways to partition data:

# DESIGNING PARALLEL PROGRAMS

## Functional Decomposition

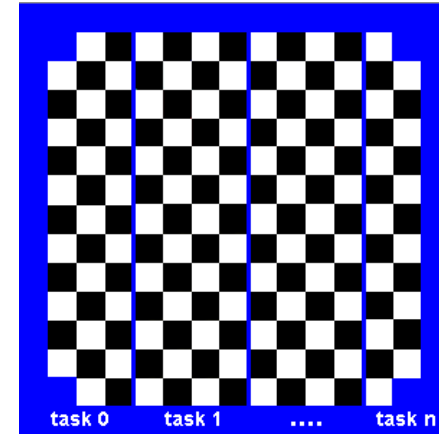
The problem is decomposed according to the work that must be done



# DESIGNING PARALLEL PROGRAMS

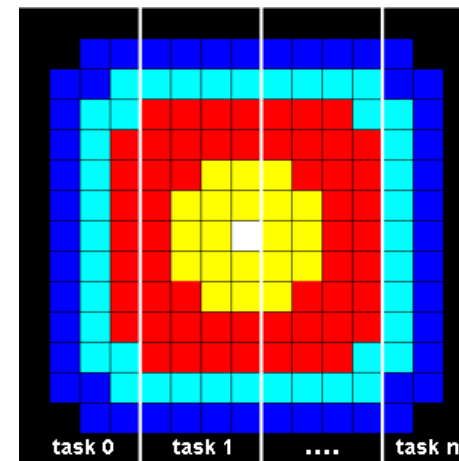
## You DON'T need communications

- Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data.
- Ex: Every pixel in a black and white image needs to have its color reversed



## You DO need communications

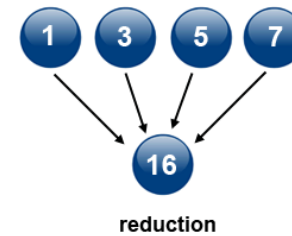
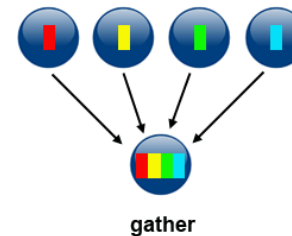
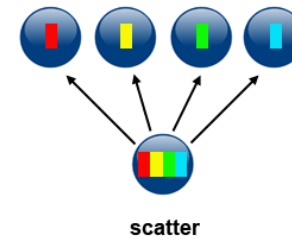
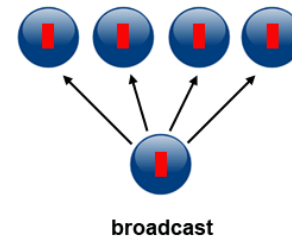
- This requires tasks to share data with each other
- A 2-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data



# DESIGNING PARALLEL PROGRAMS

## Factors to Consider (designing your program's inter-task communications)

- Communication overhead
- Latency vs. Bandwidth
- Visibility of communications
- Synchronous vs. asynchronous communications
- Scope of communications
- Efficiency of communications



# DESIGNING PARALLEL PROGRAMS

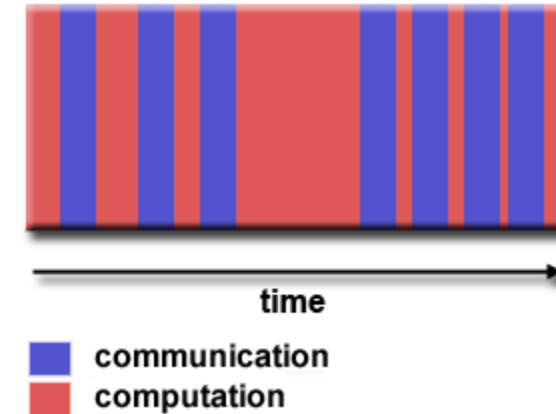
## Granularity

- In parallel computing, granularity is a qualitative measure of the ratio of computation to communication. (**Computation / Communication**)
- Periods of computation are typically separated from periods of communication by synchronization events.
- Fine-grain Parallelism
- Coarse-grain Parallelism

# DESIGNING PARALLEL PROGRAMS

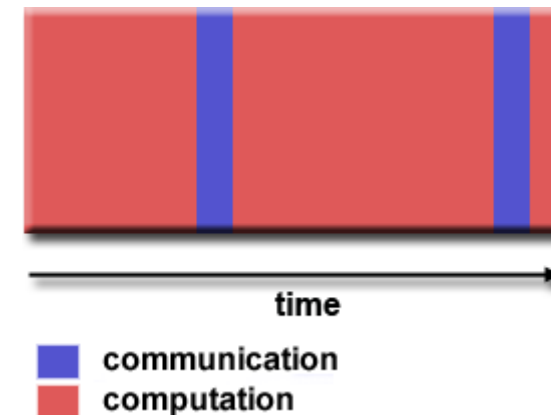
- **Fine-grain Parallelism**

- Relatively small amounts of computational work are done between communication events
- Low computation to communication ratio
- Facilitates load balancing
- Implies high communication overhead and less opportunity for performance enhancement
- If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.



- **Coarse-grain Parallelism**

- Relatively large amounts of computational work are done between communication/synchronization events
- High computation to communication ratio
- Implies more opportunity for performance increase
- Harder to load balance efficiently

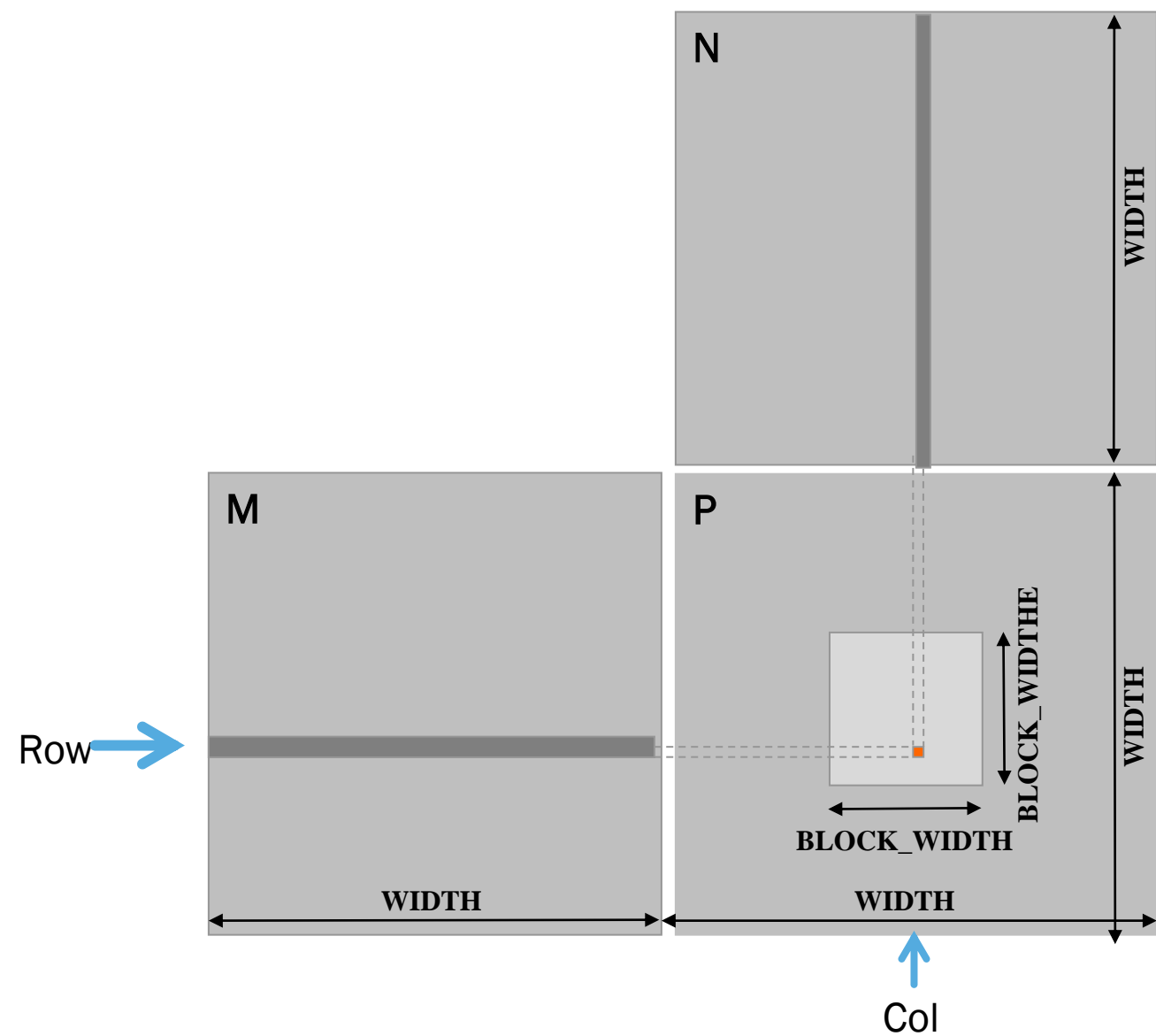


# DESIGNING PARALLEL PROGRAMS

## I/O

- Rule #1: Reduce overall I/O as much as possible
- If you have access to a parallel file system, use it.
- Writing large chunks of data rather than small chunks is usually significantly more efficient.
- Fewer, larger files performs better than many small files.
- Confine I/O to specific serial portions of the job, and then use parallel communications to distribute data to parallel tasks. For example, Task 1 could read an input file and then communicate required data to other tasks. Likewise, Task 1 could perform write operation after receiving required data from all other tasks.
- Aggregate I/O operations across tasks - rather than having many tasks perform I/O, have a subset of tasks perform it.

# EXAMPLE – MATRIX MULTIPLICATION



# A BASIC MATRIX MULTIPLICATION

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {  
    // Calculate the row index of the P element and M  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
  
    // Calculate the column index of P and N  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        // each thread computes one element of the block sub-matrix  
        for (int k = 0; k < Width; ++k) {  
            Pvalue += M[Row*Width+k]*N[k*Width+Col];  
        }  
        P[Row*Width+Col] = Pvalue;  
    }  
}
```

# CUDA BUILT-IN VARIABLES

- **blockIdx.x, blockIdx.y, blockIdx.z** are built-in variables that returns the block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code.
- **threadIdx.x, threadIdx.y, threadIdx.z** are built-in variables that return the thread ID in the x-axis, y-axis, and z-axis of the thread that is being executed by this stream processor in this particular block.
- **blockDim.x, blockDim.y, blockDim.z** are built-in variables that return the “block dimension” (i.e., the number of threads in a block in the x-axis, y-axis, and z-axis).
- So, you can express your collection of blocks, and your collection of threads within a block, as a 1D array, a 2D array or a 3D array.
- These can be helpful when thinking of your data as 2D or 3D.
- The full global thread ID in x dimension can be computed by:
  - $x = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x};$

# THREAD IDENTIFICATION EXAMPLE: X-DIRECTION

Global Thread ID

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
threadIdx.x								threadIdx.x								threadIdx.x								threadIdx.x							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
blockIdx.x = 0								blockIdx.x = 1								blockIdx.x = 2								blockIdx.x = 3							

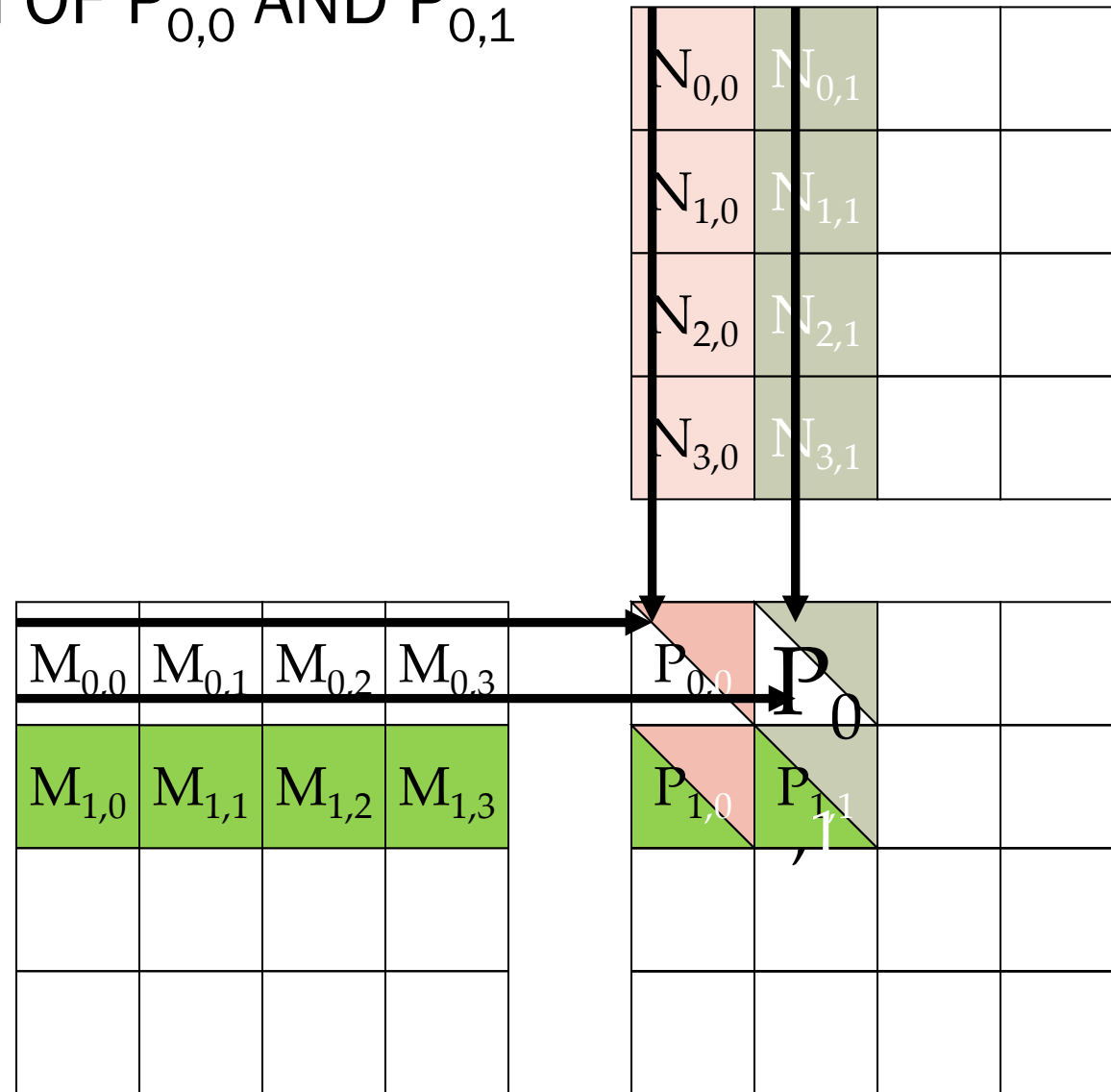
- Assume a hypothetical ID grid and ID block architecture: 4 blocks, each with 8 threads.
- For Global Thread ID 26:
  - $\text{gridDim.x} = 4 \times 1$
  - $\text{blockDim.x} = 8 \times 1$
  - $\text{Global Thread ID} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$
  - $= 3 \times 8 + 2 = 26$

## EXAMPLE – MATRIX MULTIPLICATION

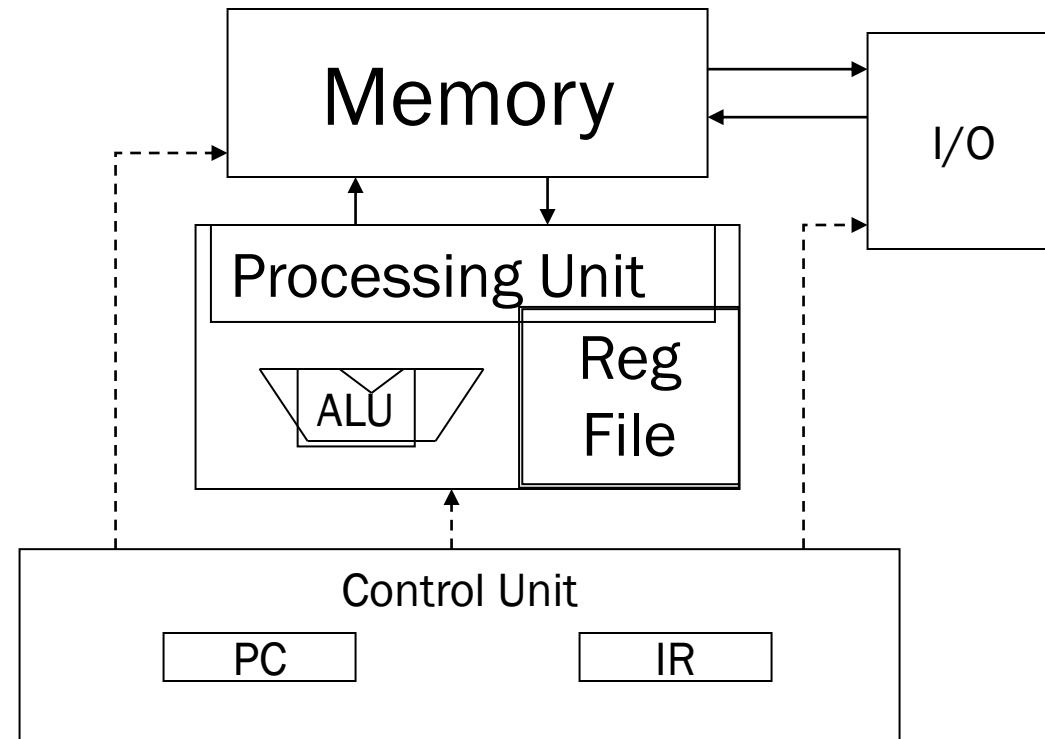
```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {  
  
    // Calculate the row index of the P element and M  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
  
    // Calculate the column index of P and N  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        // each thread computes one element of the block sub-matrix  
        for (int k = 0; k < Width; ++k) {  
            Pvalue += M[Row*Width+k]*N[k*Width+Col];  
        }  
        P[Row*Width+Col] = Pvalue;  
    }  
}
```



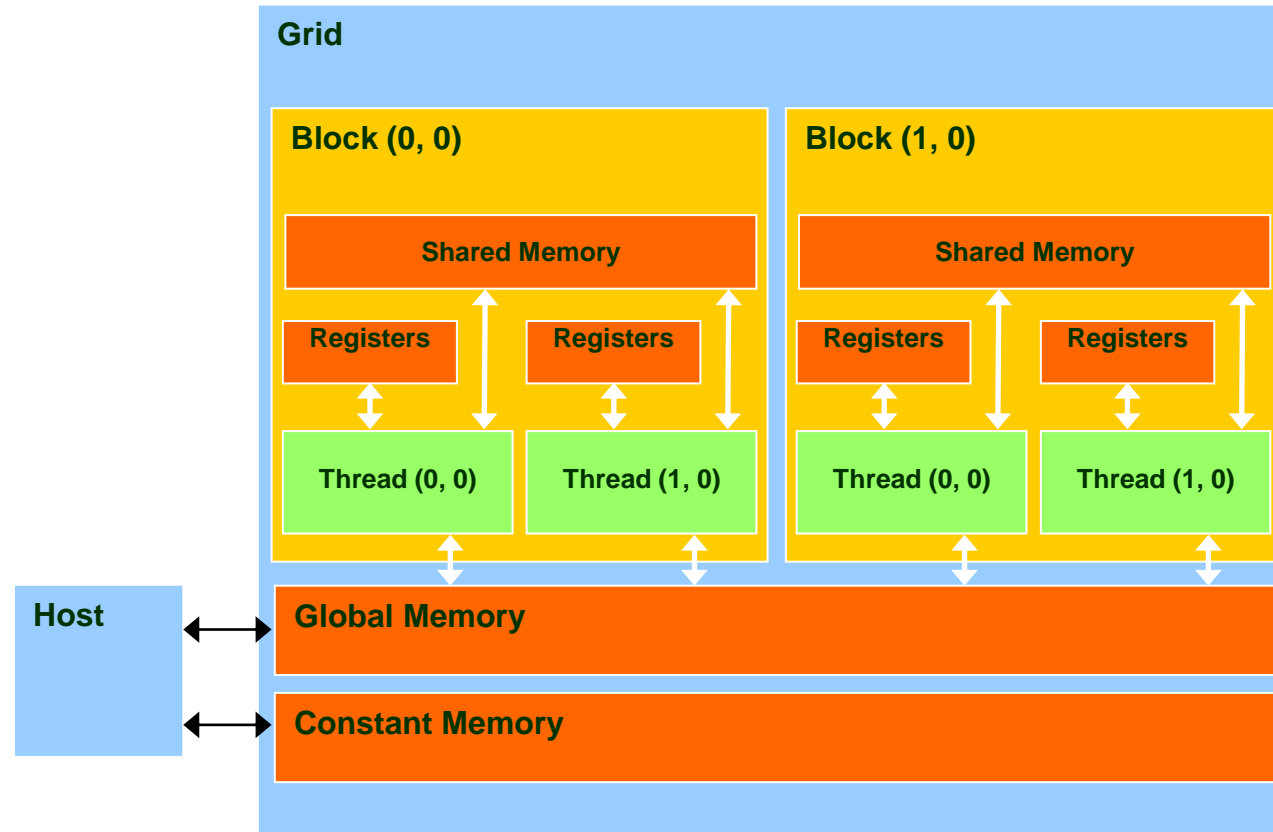
# CALCULATION OF $P_{0,0}$ AND $P_{0,1}$



# MEMORY AND REGISTERS IN THE VON-NEUMANN MODEL



# PROGRAMMER VIEW OF CUDA MEMORIES



# DECLARING CUDA VARIABLES

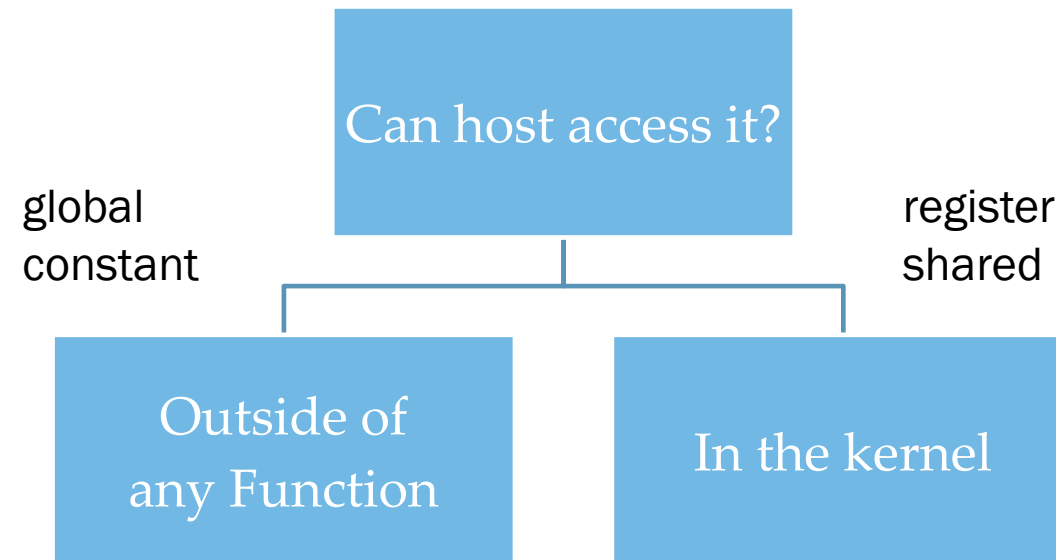
Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- **\_\_device\_\_** is optional when used with **\_\_shared\_\_**, or **\_\_constant\_\_**
- Automatic variables reside in a **register**
  - Except per-thread arrays that reside in global memory

## EXAMPLE: SHARED MEMORY VARIABLE DECLARATION

```
void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
{
    __shared__ float ds_in[TILE_WIDTH][TILE_WIDTH];
    ...
}
```

# WHERE TO DECLARE VARIABLES?



## SHARED MEMORY IN CUDA

- A special type of memory whose contents are explicitly defined and used in the kernel source code
  - One in each SM
  - Accessed at much higher speed (in both latency and throughput) than global memory
  - Scope of access and sharing - thread blocks
  - Lifetime – thread block, contents will disappear after the corresponding thread finishes terminates execution
  - Accessed by memory load/store instructions
  - A form of scratchpad memory in computer architecture

# HARDWARE VIEW OF CUDA MEMORIES

