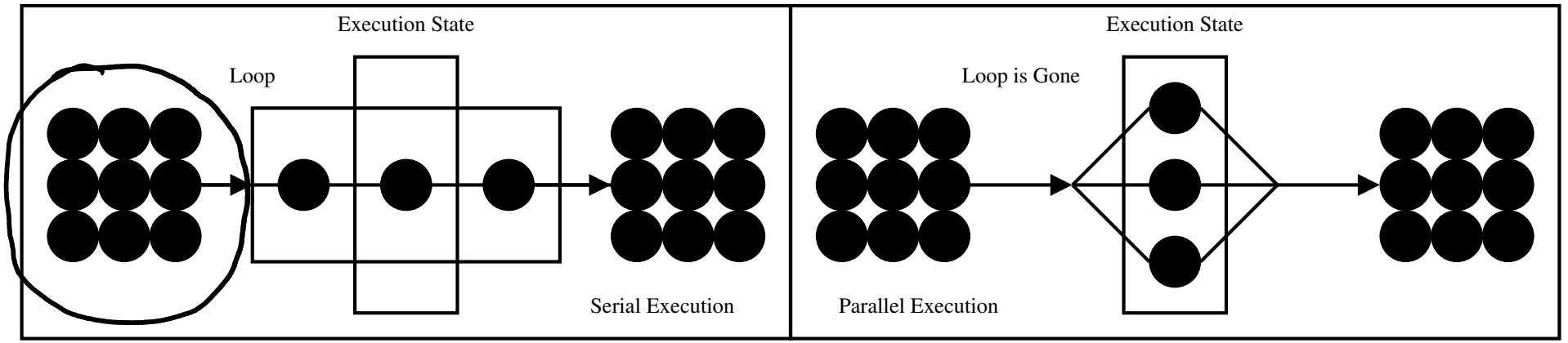
Serial vs Parallel vs Distributed (cont.)

Parallel Computing Systems

- Several processors that are located within a small distance of each other
- Their main purpose is to perform a computation task jointly
- Communication between processors, if required, is reliable, fast, and predictable.



Units of work

Figure 6: Difference between serial and parallel execution

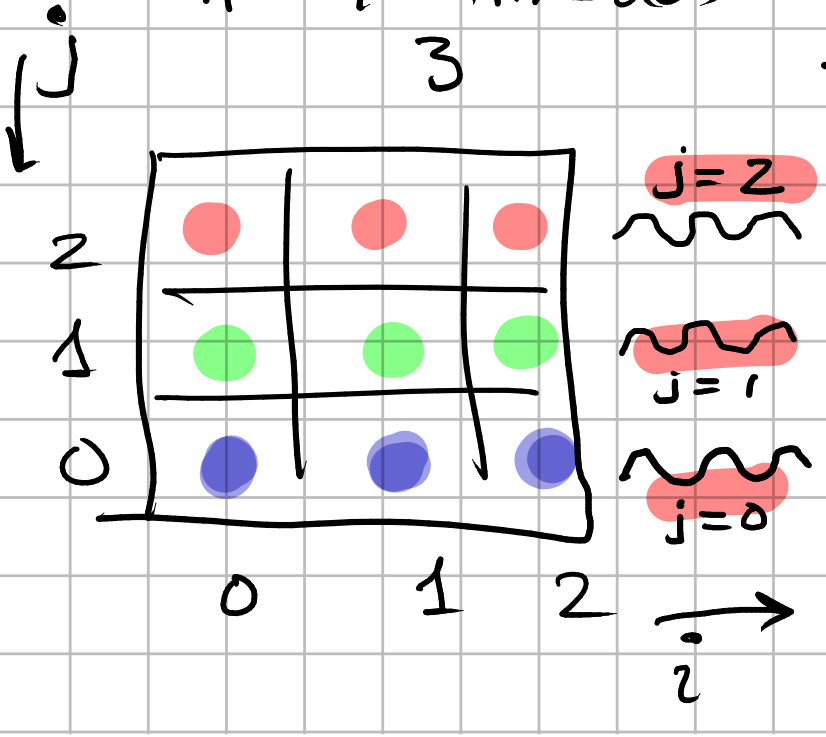
j → replace with thread..

```
for ( j=0 ; j<3 ; j++ )        j = 0,1,2
    for ( i=0 ; i<3 ; i++ )
        a[i][j]
```

— serial

# of threads 3 | work / thread 3

total work = 3 x 3 = 9

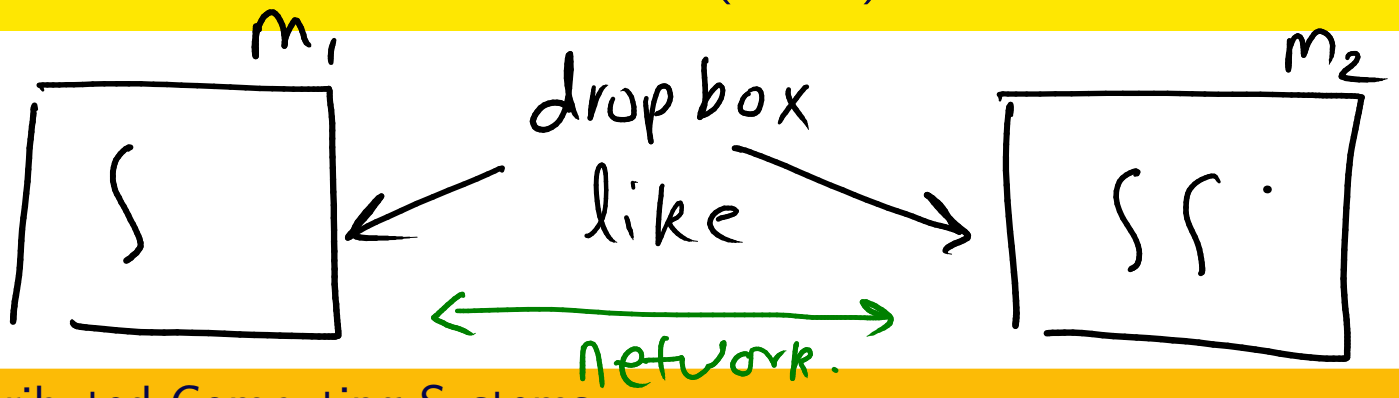


three threads start

```
for ( i=0 ; i<3 ; i++ )
    a[i][j]
```

three threads end.

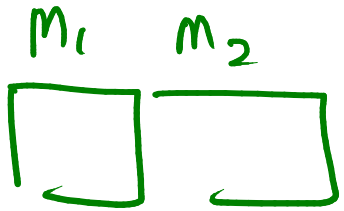
Serial vs Parallel vs Distributed (cont.)

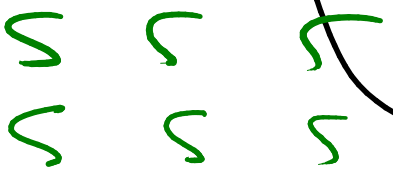


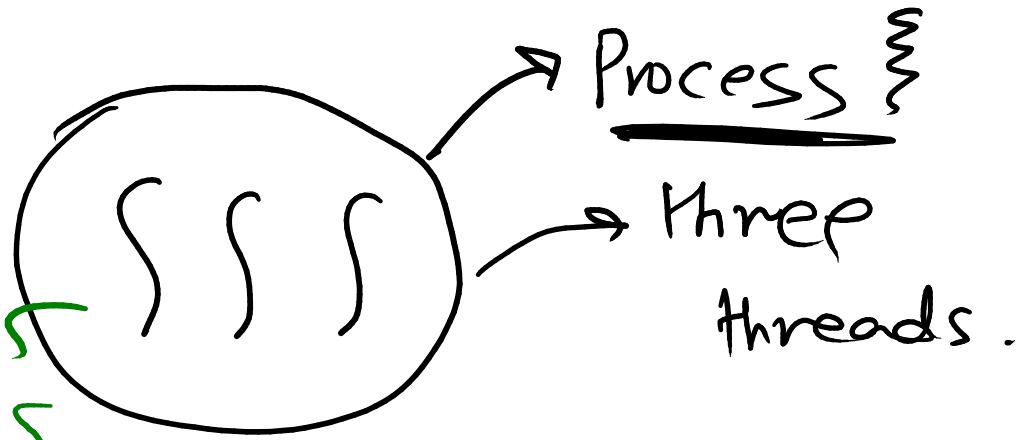
Distributed Computing Systems

- Processors may be far apart (are loosely coupled and usually independent of others)
- Challenges
  - Communication latency between processors is large and unpredictable
  - Communication links may be unreliable
  - Topology of system may undergo changes.
  - Load Balancing difficulties → (synchronization)

Work items > # of machines







Serial vs Parallel vs Distributed (cont.)

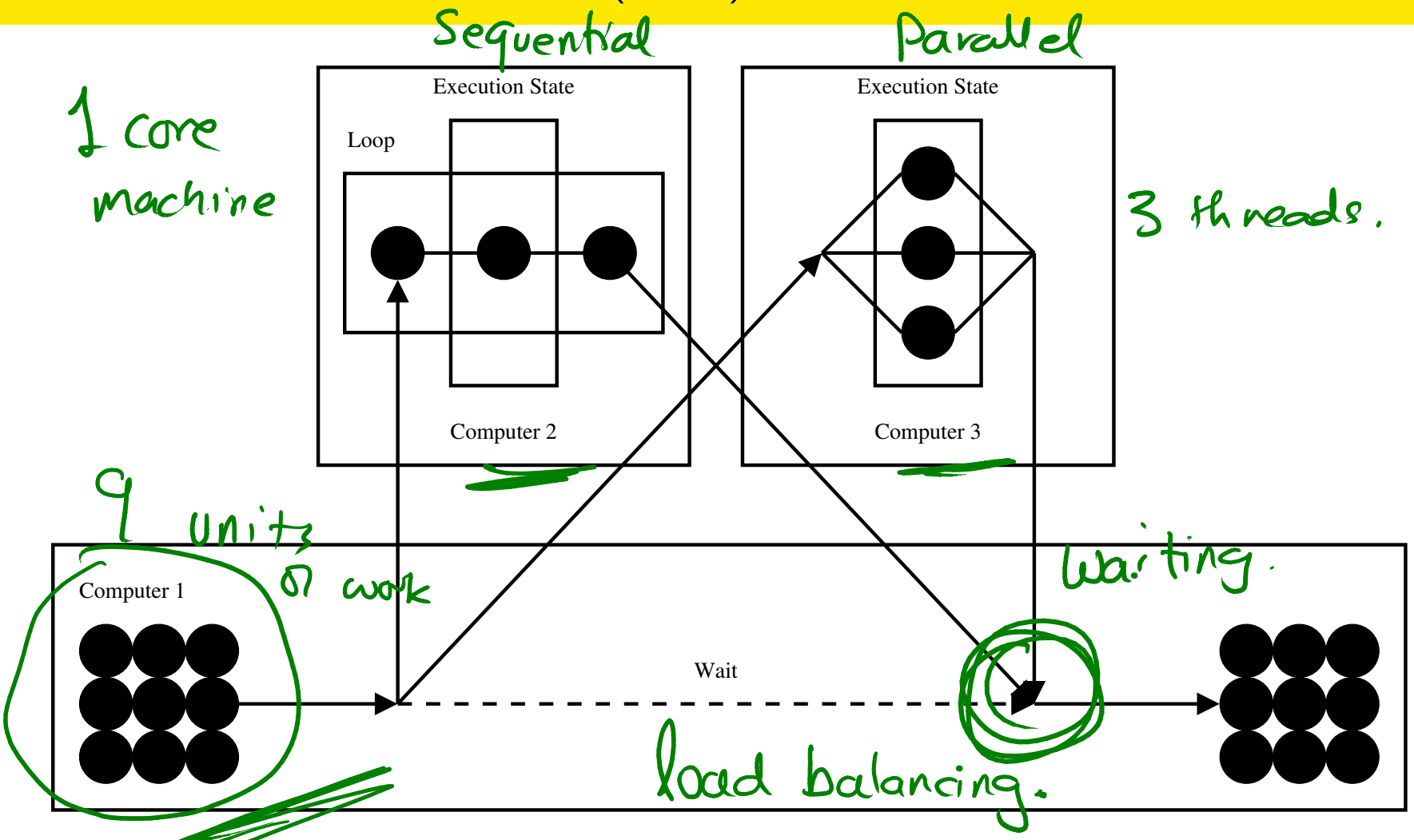
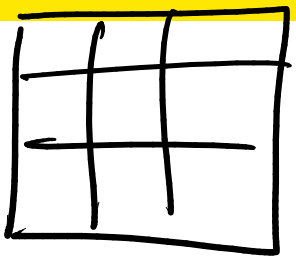


Figure 7: Model of execution on a distributed system (non-client/server). This shows a hybrid. But usually developers don't like these setups.

What is there in Parallel, that is not available in Serial?

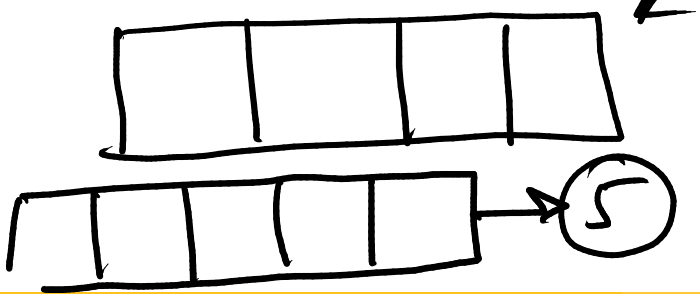
Some Examples . . .



total work → decompose into smaller tasks

threads x work done per thread

- **Task Allocation**: Breakdown of total workload into smaller tasks, such that each smaller task is assigned to a different processor
- **Interim Communication**: Since each processor is working on independent tasks, they may need to communicate outcome of computation with other processors.
  - Is communication automatic? or defined by programmer?
  - Depends on type of multi-threading tool that is being used
- **Synchronization**: A waiting interval by a processor for certain tasks
  - E.g., arrival of data (before computation on it can be performed, or before bring printed).
  - This would have positive effect on accuracy but negative effect on performance.
- . . .



4 work units.

1 threads — 4 w.units. = 4

2 threads — 2 w.units. = 4

4 threads — 1 w.unit = 4

Applications and Examples

Task Parallel Threads

threads are doing different types of works.

General Purpose Programs and Purposes

- Word processor (spell check, auto save, . . .)
  - Spreadsheet (automatic background recalculations after cell edits)
- Operating system (User threads, Kernel threads, . . .)
  - Server (multi client handling, database servers serving millions of transactions per second)
  - GUI's (Event listeners)
  - Video games (rendering of animation, taking care of physics, artificial intelligence, network connections, user I/O, . . .))
- Web browser (tabs, multi-segment download accelerators, . . .)
  - Search Engines (Information searching, retrieval)
  - Encoding/Decoding of video (compression/decompression)
  - . . .

Applications and Examples (cont.)

Data Parallel threads.

Scientific Purposes

- Mathematical problems (Ordinary differential equations, partial differential equations, linear algebra, numerical analysis)
  - Climate modelling and weather prediction
  - Fluid dynamics (video games, simulation of dam bursts, tsunamies, etc.)
  - Computational Biology, Finance, Physics, Chemistry
  - Machine learning, statistical methods
  - Image processing (medical images, spectral images, satellite images)
- Finding Aliens, Finding cure for Alzheimers, Cancers, Parkinson's, ...

Figure 8: The Folding@Home project by Stanford University

Data Parallel example

$$\begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

```
for( i=0; i<2; i++ )        Removed
    C[i] = A[i] + B[i]
```

```
i=Create two threads
    C[i] = A[i] + B[i]
end two threads
```

multiple threads

doing the same task on diff. locations.

task parallel variation

$C_1 = A_1 + B_1$

$C_2 = A_2 - B_2$

```
for( i=0; i<2; i++ )
    if(i%2==0) c[i] = A[i] + B[i]
    else       c[i] = A[i] - B[i]
```

```
i=Create two threads

    if(i%2==0) c[i] = A[i] + B[i]
    else       c[i] = A[i] - B[i]

end two threads
```

Multiple threads different tasks on different locations.

Task parallel variation

$C_1 = A_1 + B_1$

$C_1 = A_1 - B_1$

```
for( i=0; i<2; i++)
    if(i%2==0) C = A + B
    else       C = A - B
```

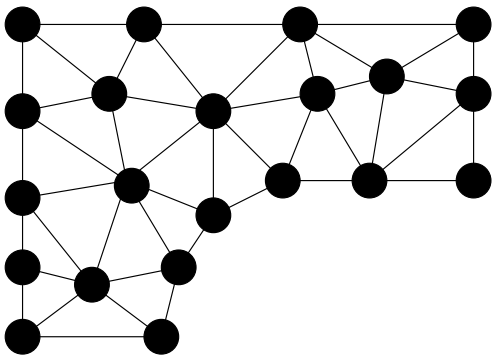
```
i=Create two threads

    if(i%2==0) C = A + B
    else       C = A - B

end two threads
```

Multiple threads different tasks on same location

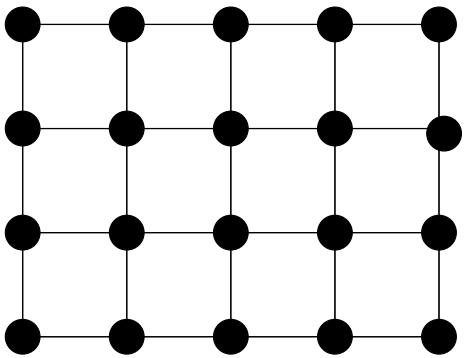
Difference between both types of Applications

Skip

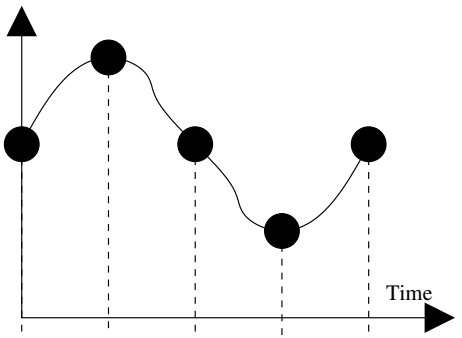
- General Purpose Applications: Threads are usually involved in doing **different things**. This is known as **task-parallelism**.
- Scientific Applications: Threads are usually involved in doing **similar things**, but at different locations. This is known as **data-parallelism**
- Different locations? Take a problem and perform both **spatial decomposition**, as well as **time decomposition**.
- Each decomposed location (involving predominantly local interactions) will be handled ideally by a single processor. If it involves non-local interactions, then processors would need to communicate.
- What level of decomposition would be required ?? A question answered by concerns of cost, speed, storage memory, functional requirements, etc.



Non–Uniform Spatial Decomposition

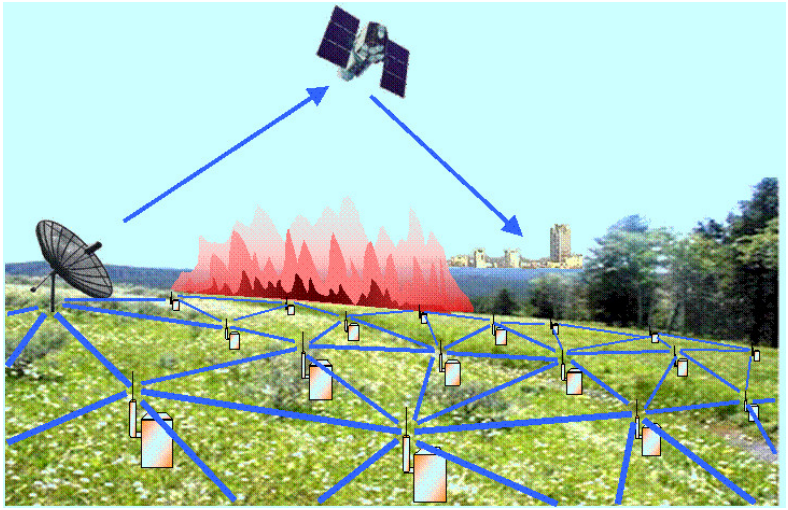


Uniform Spatial Decomposition



Uniform Time Decomposition

Applications



← Fig: Sensor Networks Deployment

- Approach-1: Client/Server Distributed Models
- Approach-2: Decentralized Distributed Models (P2P, WSN, MANET)

- Functions: Information Acquisition, Extraction, Aggregation and Control in geographically distributed systems
- E.g., sensor networks, where geographically distributed sensors **acquire** local information). This information is communicated to a control station through multiple hops. At each hop, information may be **extracted**, **aggregated** (to remove redundancies), and repackaged before forwarded. The control station ultimately recieves the packet and performs **control** operations (e.g., fire control, smart cooling, etc.)
- Concerns: unreliable communication, device failures, cost, speed, deployment, energy constraints

Examples of Processors

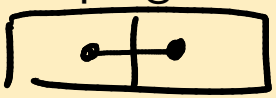
prog: task parallel vs Data parallel.

- Hundreds of architectures, dozens in main-stream, hundreds now obsolete
- Many ways of classifying these architectures. The most simple classification is those that are "parallel", and those that are "distributed"

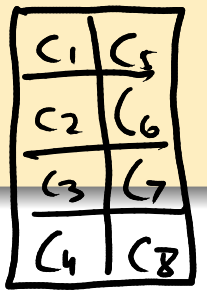
GPU's → Stream processors.

Array Processors (Vector Processors)

- Array of coupled processors (with nearest neighbor inter-connects)
- Most array processors perform "combined" or "group" execution.
- Ideal for spatial decomposed problems, mage processing, scientific computing, etc. (not good for task-parallelism) Data Parallel.
- A host processor may monitor progress on the processor array and serves as a "control processor"



- Distributed/shared memory array processor possible
- More about this in Hardware Classifications (Flynn)

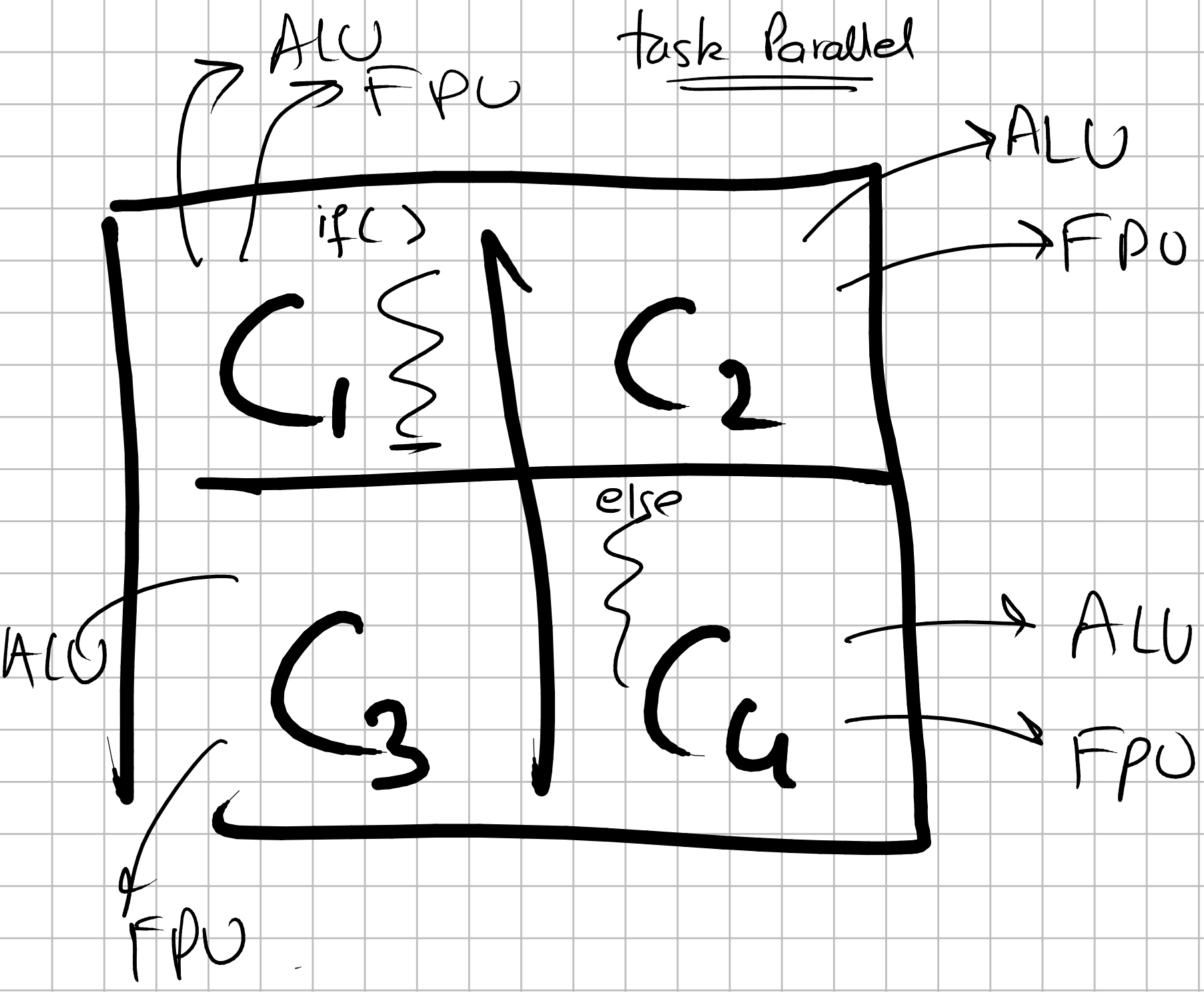


MP
8 cores.

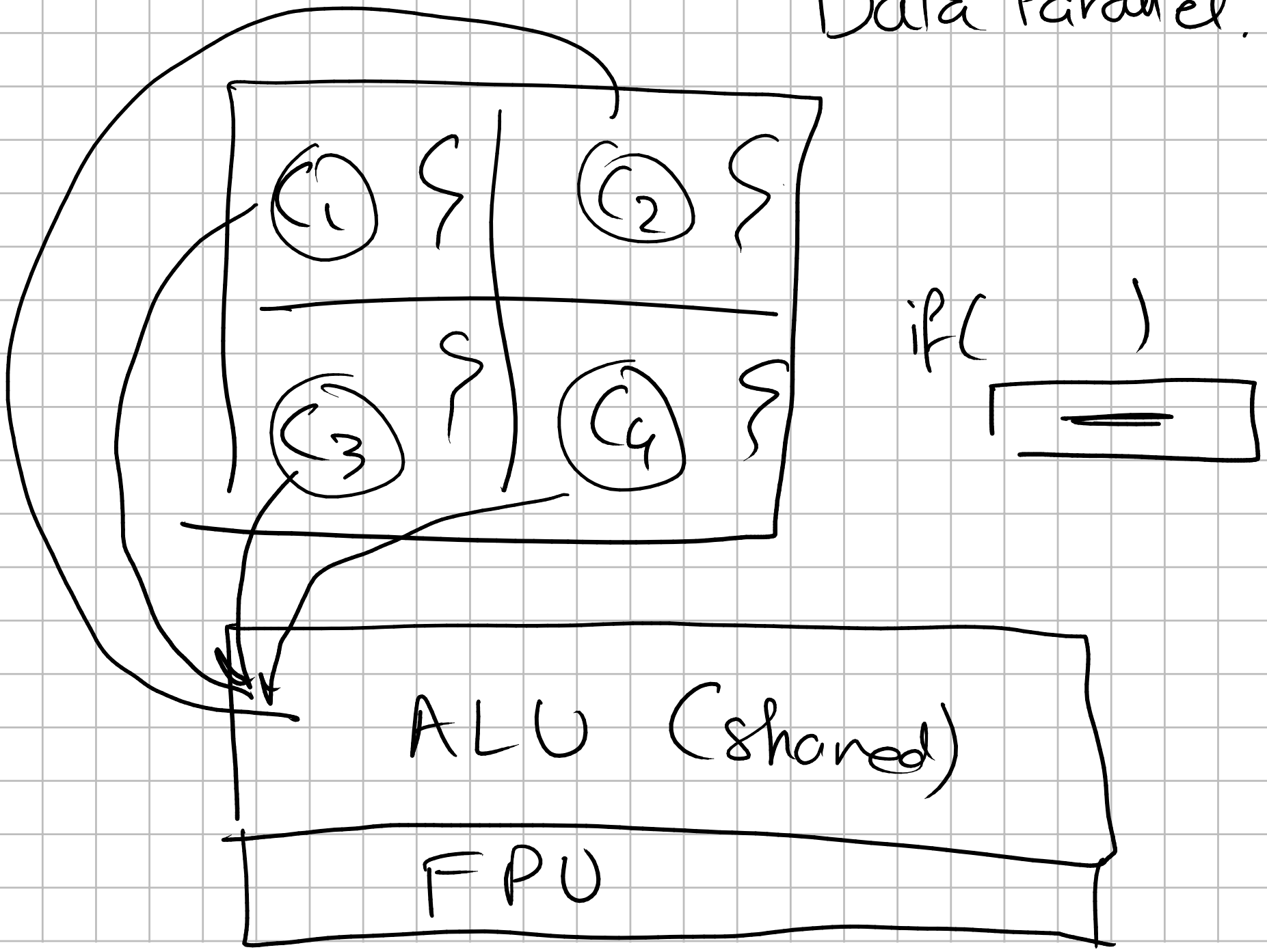
ALU
FPU

$$d_i = a_i + b_i$$

array.


Task Parallel
ALU
FPU
ALU
FPU
if( )
C1
C2
else
ALU
C3
C4
ALU
FPU
FPU

Data Parallel.



Examples of Processors (cont.)

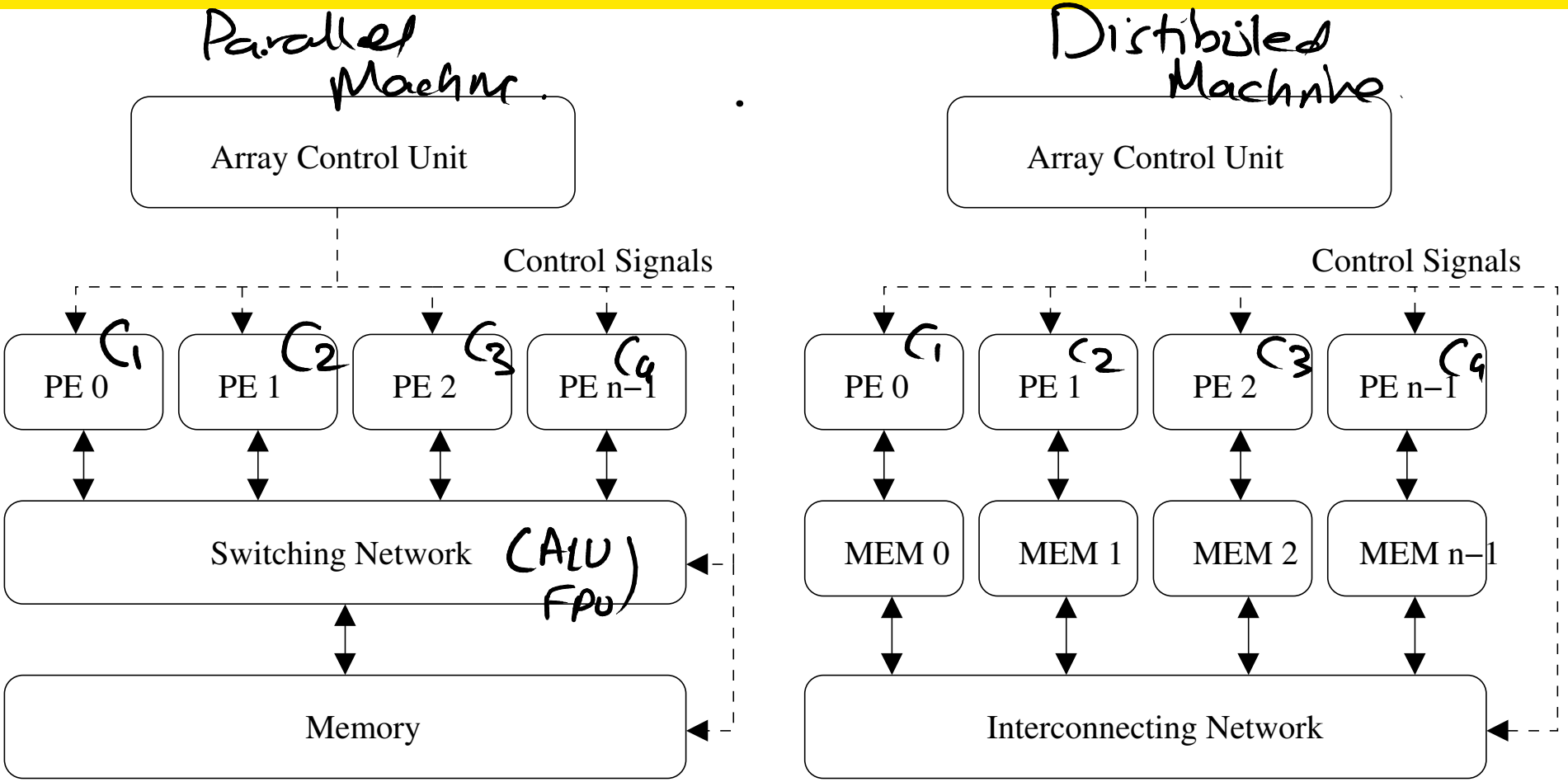


Figure 9: Array Processors (left) Shared Memory (right) distributed memory (IMG: Roland N Ibbett & Nigel P Topham, 1996) (Key: PE = Processing Element)

Examples of Processors (cont.)

MP – multiprocessor.

CPU

Symmetric Multiprocessor.

Multiprocessors

- Loosely coupled processors, where each processor has more control on what to do and when to do.
- Well suited for task parallel problems (general purpose multi-threaded applications)
- Most common type is known as "symmetric multiprocessor system"

Systolic Arrays

→ Specialized Processor. (one + only one task. (ex. Fast))

- Makes use of "Very Large Scale Integration" (VLSI) Technology, where all computation elements are placed on a single chip. FPGA
- Processors are designed for a single purpose only and show fixed behaviour (code is embedded in system hardware)
- E.g., solving a linear system of equations, performing a fast Fourier transform
- Data "moves" across each processor (must be aligned before execution can commence)

Examples of Processors (cont.) Not Important

Matrix Multiplication.

3×3

A00 A01 A02	B00 B01 B02	A00*B00 + A01*B10 + A02*B20	A00*B01 + A01*B11 + A02*B21	A00*B02 + A01*B12 + A02*B22
A10 A11 A12 x	B10 B11 B12 =	A10*B00 + A11*B10 + A12*B20	A10*B01 + A11*B11 + A12*B21	A10*B02 + A11*B12 + A12*B22
A20 A21 A22	B20 B21 B22	A20*B00 + A21*B10 + A22*B20	A20*B01 + A21*B11 + A22*B21	A20*B02 + A21*B12 + A22*B22

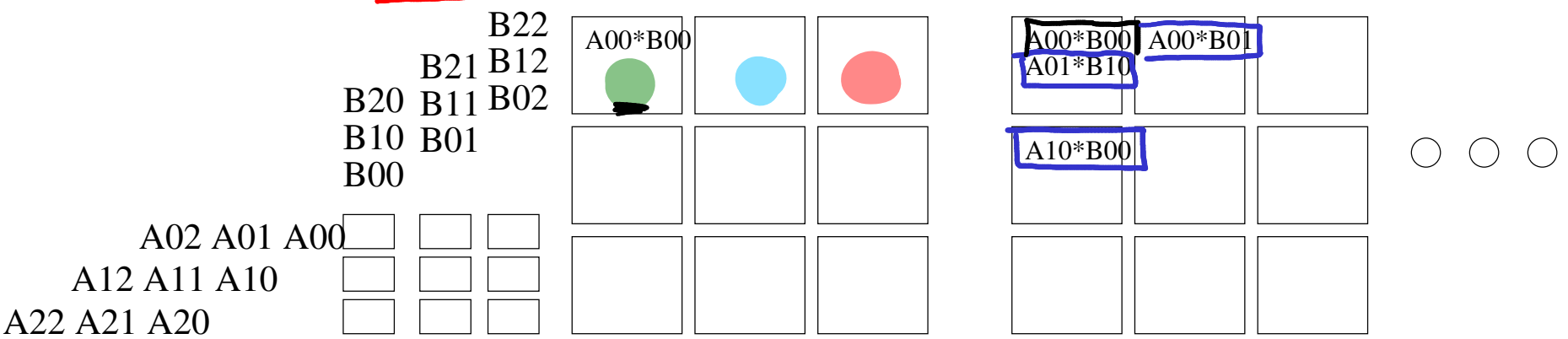
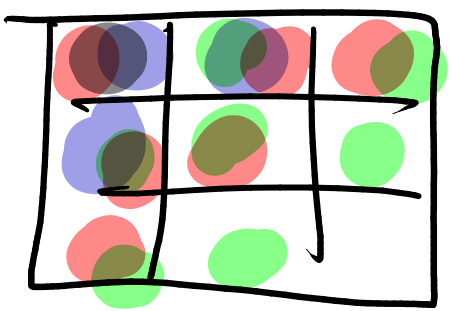


Figure 10: Movement of Data across systolic array for Matrix Multiplication



t_0 t_1 t_3

Flynn-Johnson Taxonomy (Classification)

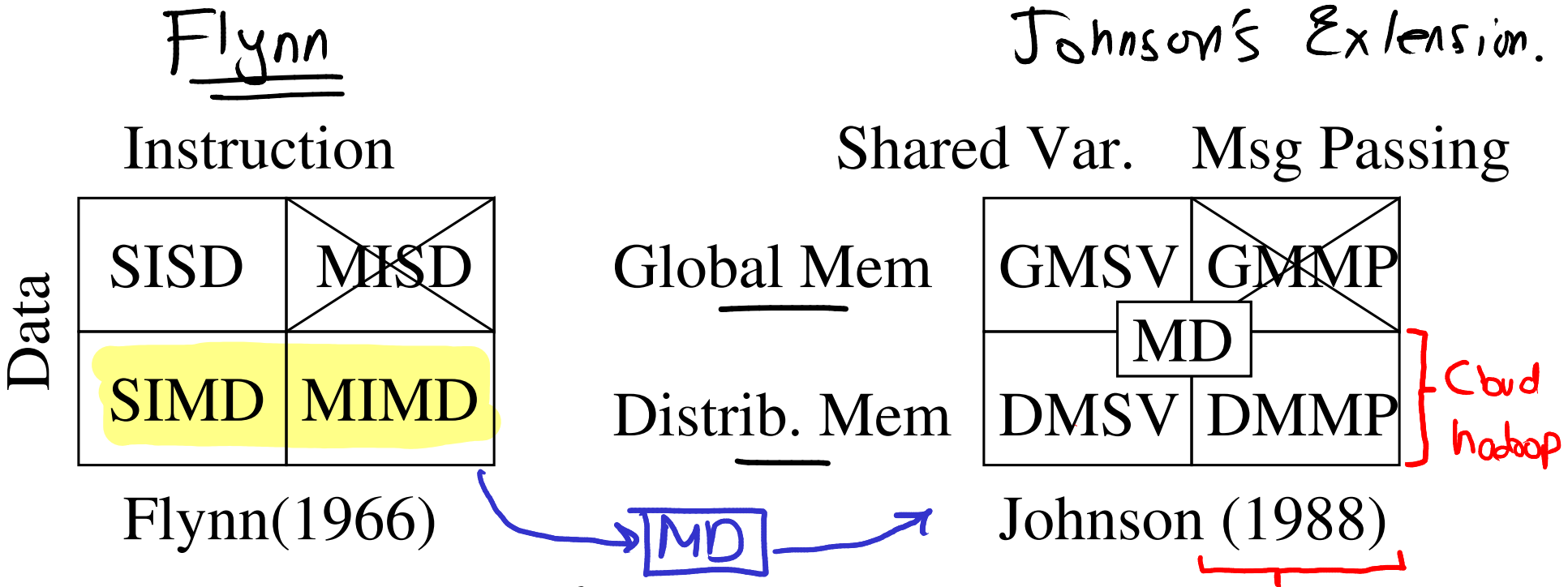
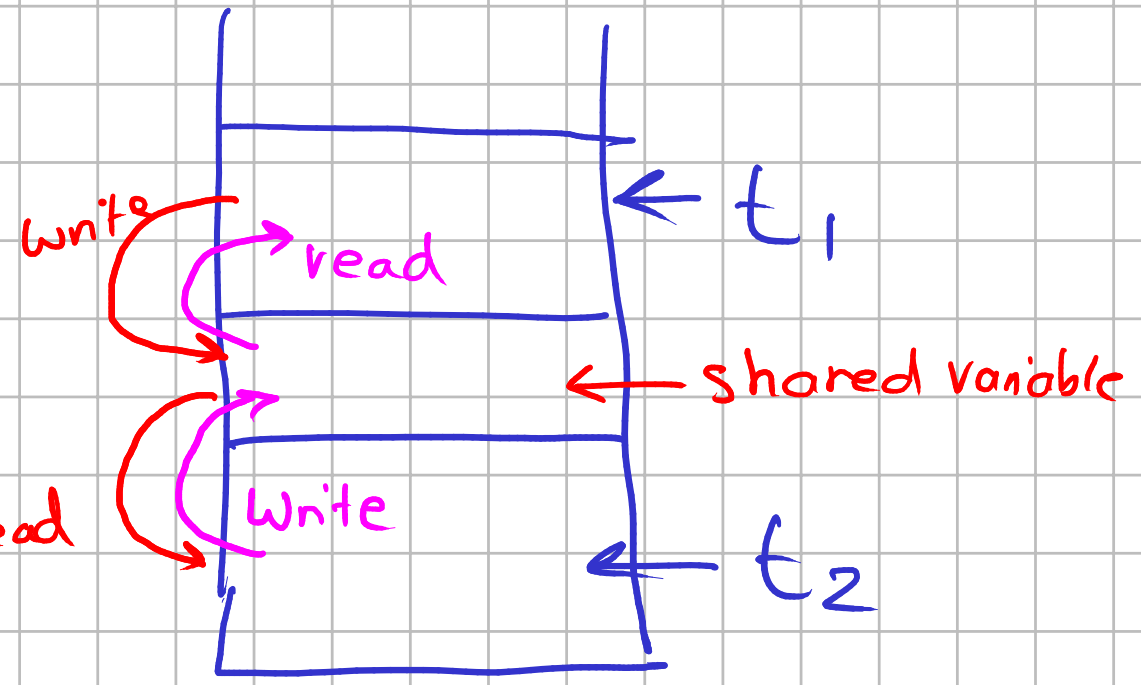


Figure 13: (l) Flynn's Classification (1966) based on instruction and data streams, and (r) Johnson's Classification (1988) based on memory and communication structures

- **Instruction Stream**: Sequence of Instructions
- **Data Stream**: Sequence of Data

Shared Variable



Synchronization

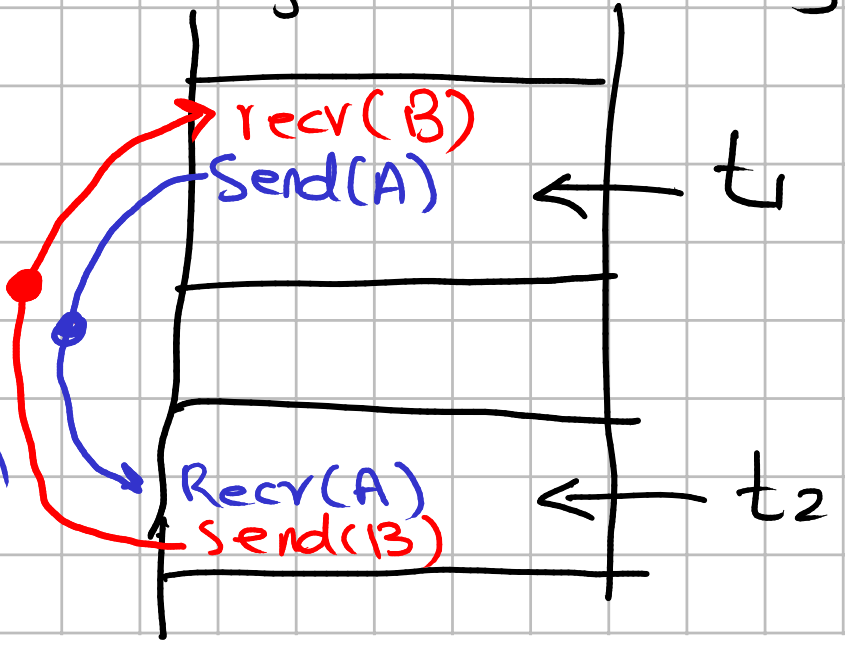
MPI → Distributed

Message Passing → Blocking vs non-blocking

Send(t_1, t_2, A)

Send(t_2, t_1, A)

src dest. payload



Flynn

Code & Data

Instructions → Code.

Data Stream	SI	MI
SD	SISD (Single core CPU's)	MISD
MD	SIMD	MIMD

SIMD, MIMD → array processors

MISD → X

Data.

A: a_0 | a_1 | a_2 | a_3

B: b_0 | b_1 | b_2 | b_3

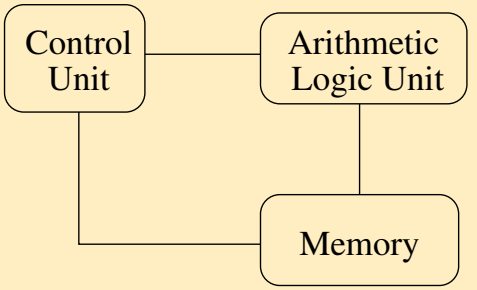
```
if( )
    a_i = a_i + b_i        → MI, MD
else()
    a_i = a_i - b_i
```

$a_i = a_i + b_i$ → SI, MD

Flynn-Johnson Taxonomy (Classification) (cont.)

SISD

- Uni-Processor architectures
- Logic of program is sequential (sequential flow of instructions)
- Concurrent execution using context-switching can be possible
- *It's still SISD if you follow sequential programming on multi-core machines*



Typical Fetch/Execute Cycle: (1) Instruction Fetch, (2) Ins. Decode, (3) Operand Address Calculation, (4) Operand Fetch, (5) Execute, (6) Store Result

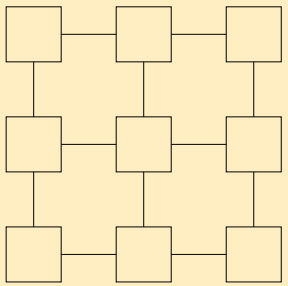
MISD

- Not commercially implemented
- E.g. multiple processors working on decrypting a single encrypted object (using different decrypting algorithms.

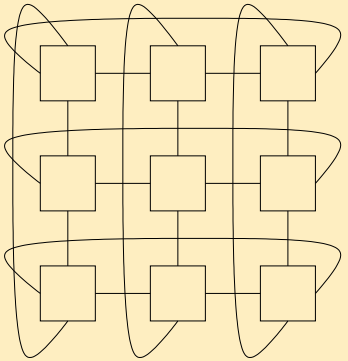
Flynn-Johnson Taxonomy (Classification) (cont.)

SIMD/MIMD

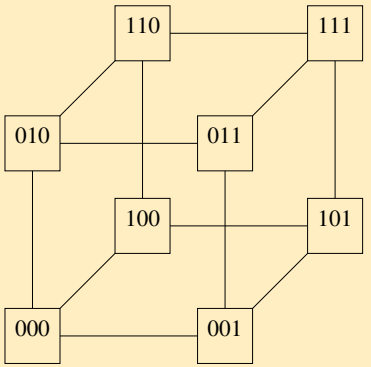
- Array/Streaming/Vector Processors interconnected using nearest-neighbour inter-connects.
- Processing elements contain only "Arithmetic Unit" instead of "Arithmetic and Logic Unit" (hence combined execution)
- Possible to disable some processing elements, if not needed
- Possible Issues
  - Data Alignment Issues
  - Conditional execution



Simple Nearest Neighbor Inter–connects



3 by 3 Nearest Neighbor Inter–connects



3–cube Inter–connects

Flynn-Johnson Taxonomy (Classification)

Johnson's Taxonomy

- **GMSV** Symmetric Multi-Processors (OpenMP, PThreads)
- **DMMP** Distributed systems (If simulated: Omnet++, NS2/3, etc.), NUMA based Cluster systems (OpenMPI, RPC), GPU Computing (OpenCL, CUDA), HPC based systems
- **DMSV** Cloud Model: Memory is distributed but access is through same address space (Hadoop, MapReduce)

MD

↳ Separate programming models.