INTRODUCTION TO PARALLELISM

THIS WEEK IN PDC…

- What is Parallelism
  - Speedup and Efficiency
- Why can’t everything be run in parallel?

PARALLELISM

Ability to execute different parts of a program concurrently on different machines

GOAL OF PARALLELISM

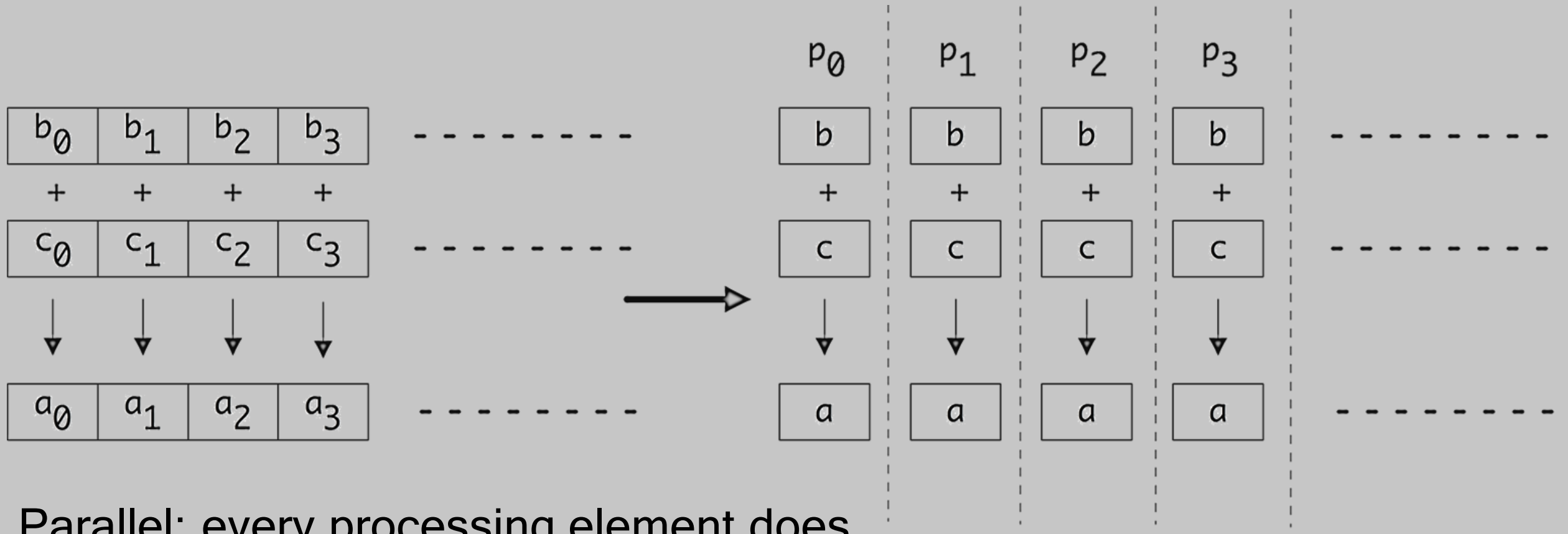
Shorten execution time

SIMPLE EXAMPLE:

```
1. for (i=0; i<n; i++)
2.   a[i] = b[i] + c[i];
```

- Summing two arrays together

SIMPLE EXAMPLE



Parallel: every processing element does

```
for ( i in my_subset_of_indices )
    a[i] = b[i] + c[i];
```

Difference b/w operations

```
1. for (i=0; i<n; i++)
2.   a[i] = b[i] + c[i];
```

- Summing two arrays together

- Compare operation counts

```
3. s = 0;
4. for (i=0; i<n; i++)
5.   s += x[i]
```

- Compare behavior on single processor. What about multi-core?

- Other thoughts about parallel execution?

MEASURES OF PERFORMANCE

- To computer scientists: speedup, execution time.

- To applications people: size of problem, accuracy of solution, etc.

FOR NEXT SLIDES KEEP IN MIND

Single processor time $T1$,

on p processors Tp

SPEEDUP

- Single processor time T1, on p processors Tp

- Speedup is $Sp = \frac{T1}{Tp}$,
  - $S_P \leq p$

- efficiency is $Ep = \frac{Sp}{p}$,
  - $0 < E_p \leq 1$

SPEEDUP

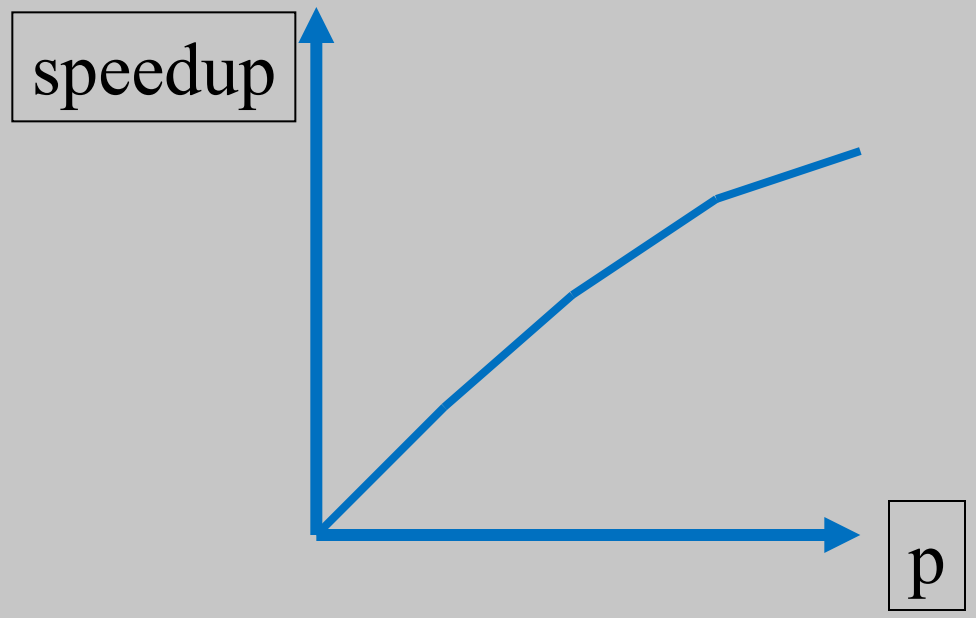
- Is T1 based on the same algorithm? The parallel code?

- Sometimes superlinear speedup.

- Is T1 measurable? Can the problem be run on a single processor?

SPEEDUP OF ALGORITHM

- Speedup of algorithm = sequential execution time / execution time on p processors (with the same data set).



SPEEDUP ON PROBLEM

- Speedup on problem =

  sequential execution time of best-known sequential algorithm

  ÷

  execution time on p processors.

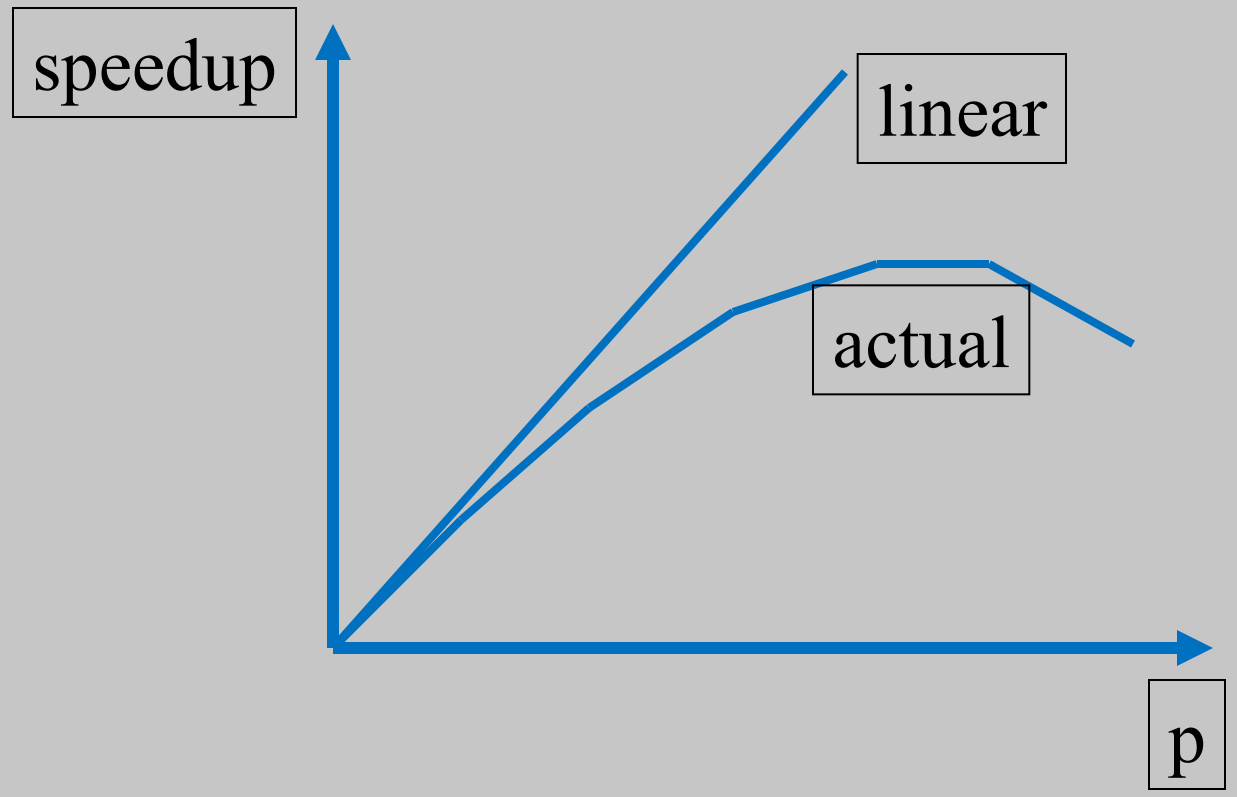
- A more honest measure of performance.
- Avoids picking an easily parallelizable algorithm with poor sequential execution time.

What speedups can you get?

- Linear speedup
  - Confusing term: implicitly means a 1-to-1 speedup per processor.
  - (almost always) as good as you can do.

- Sub-linear speedup: more normal due to overhead of startup, synchronization, communication, etc.

SPEEDUP



WISHFULLY…

If you put in n processors, you should get n times Speedup (and 100% Speedup Efficiency), right?

Wrong!

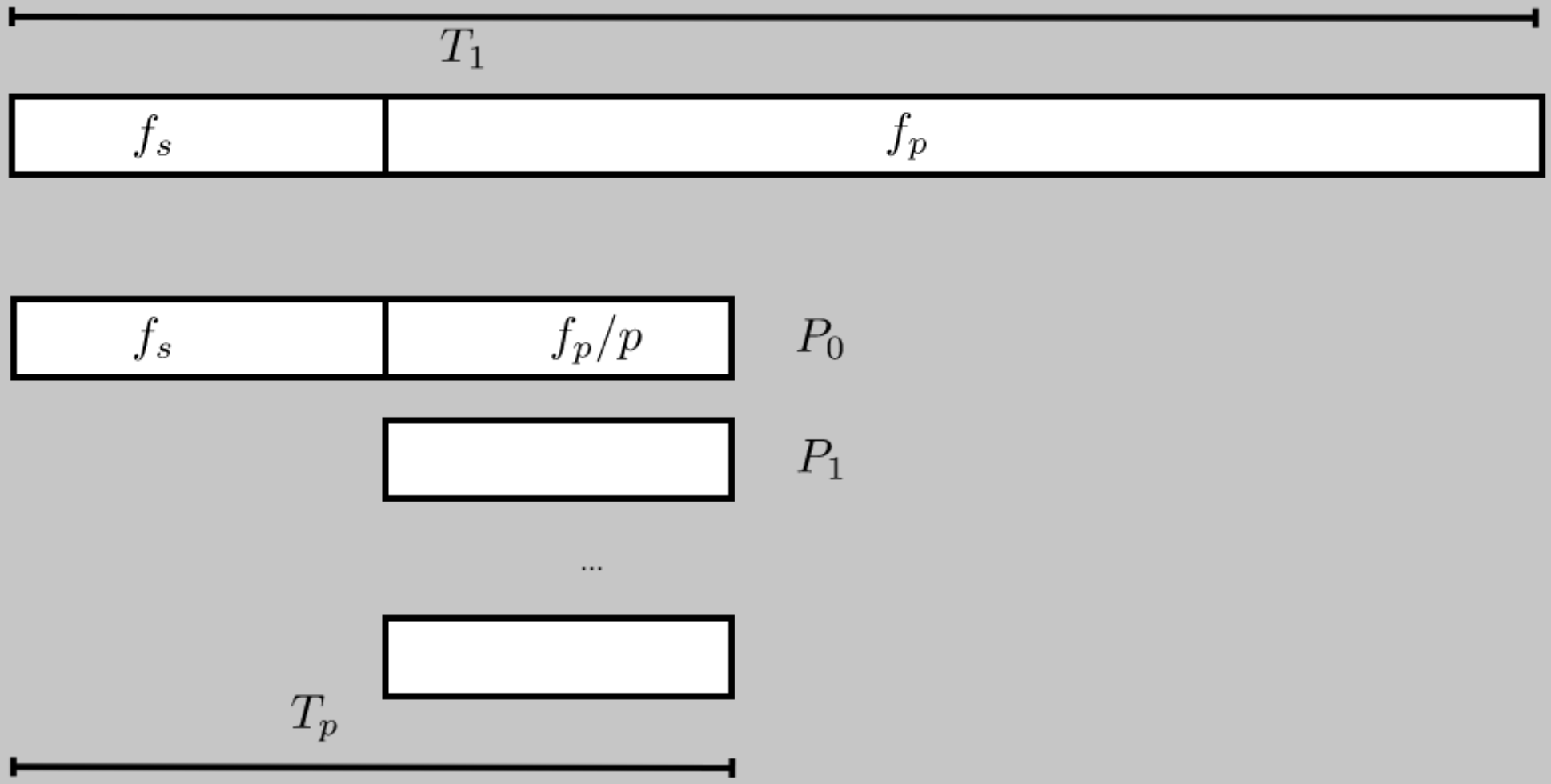
AMDAHL'S LAW

- There are always some fraction of the total operation that is inherently sequential and cannot be parallelized no matter what you do.
- This includes reading data, setting up calculations, control logic, storing results, etc.

AMDAHL'S LAW

- Let's assume that part of the application can be parallelized, part not. (Examples?)
  - Fs sequential fraction, Fp parallelizable fraction
  - Fs + Fp = 1

AMDAHL'S LAW



AMDAHL'S LAW

- T1 = (Fs + Fp)T1 = Fs T1 + Fp T1
- Amdahl's law: Tp = Fs T1 + Fp T1/p
- $Sp = \frac{T1}{Fs \times T1 + \left(\frac{Fp \times T1}{p}\right)} = \frac{T1}{(1-Fp)T1 + \left(\frac{Fp\ T1}{p}\right)}$

AMDAHL'S LAW

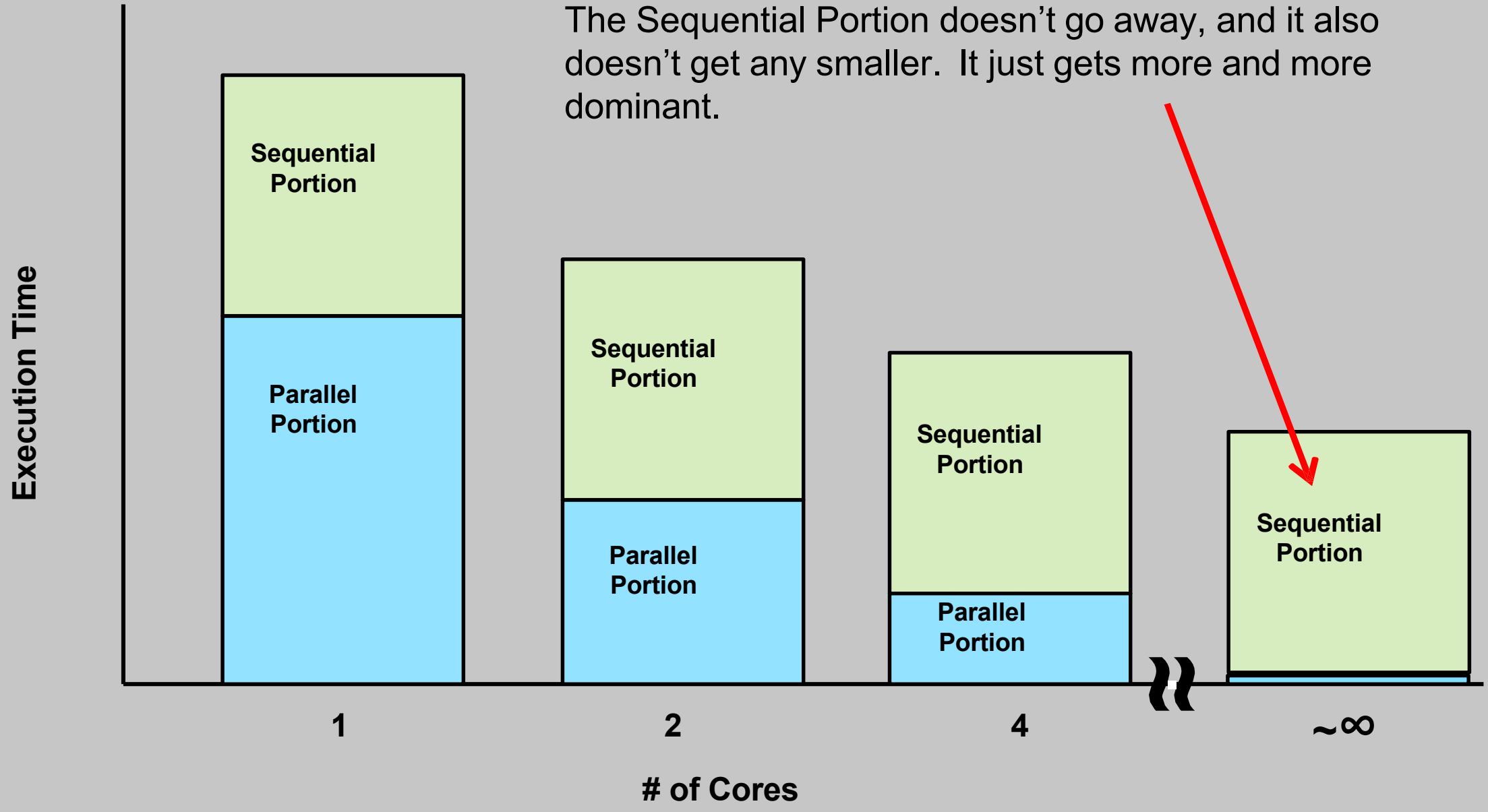
- $P \rightarrow \infty$: TP $\downarrow$ FsT1

- Speedup is limited by $Sp \leq 1/Fs$,
  efficiency is a decreasing function $E \sim 1/p$.

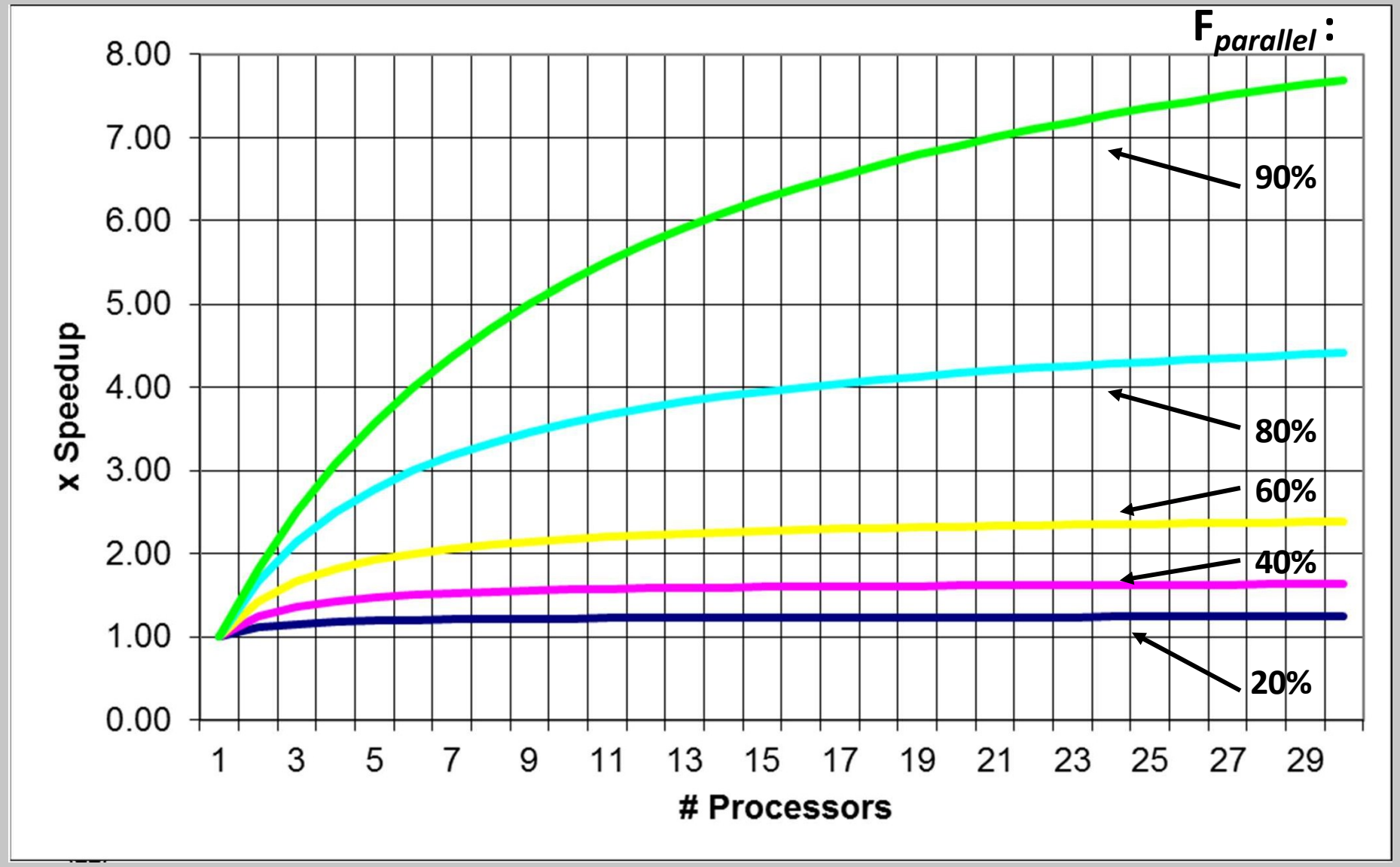
- Do you see problems with this?

Visual Representation of Amdahl's Law

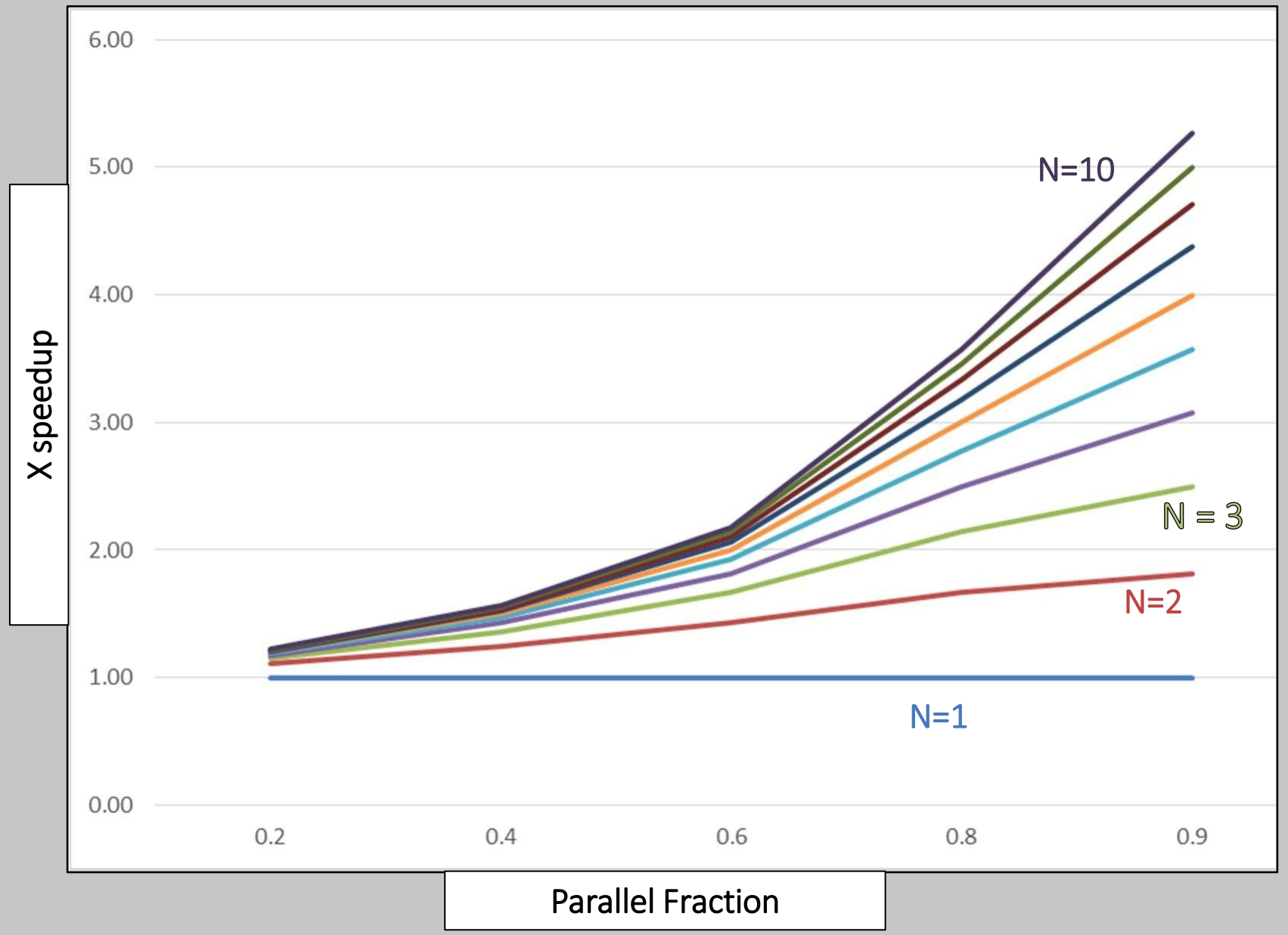
The Sequential Portion doesn't go away, and it also doesn't get any smaller. It just gets more and more dominant.



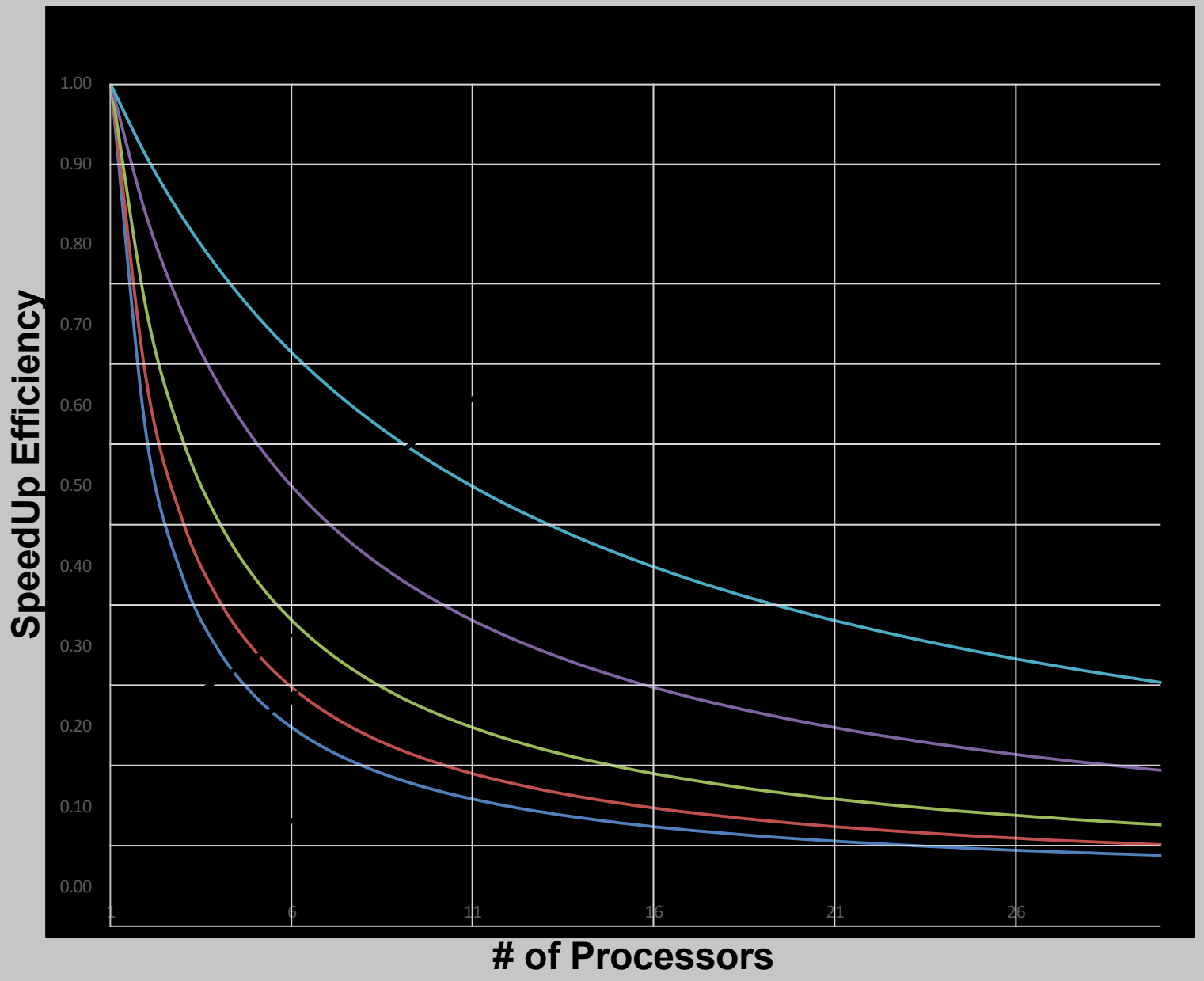
SPEEDUP AS FUNCTION OF NUMBER OF PROCESSORS



SPEED-UP AS FUNCTION OF FP



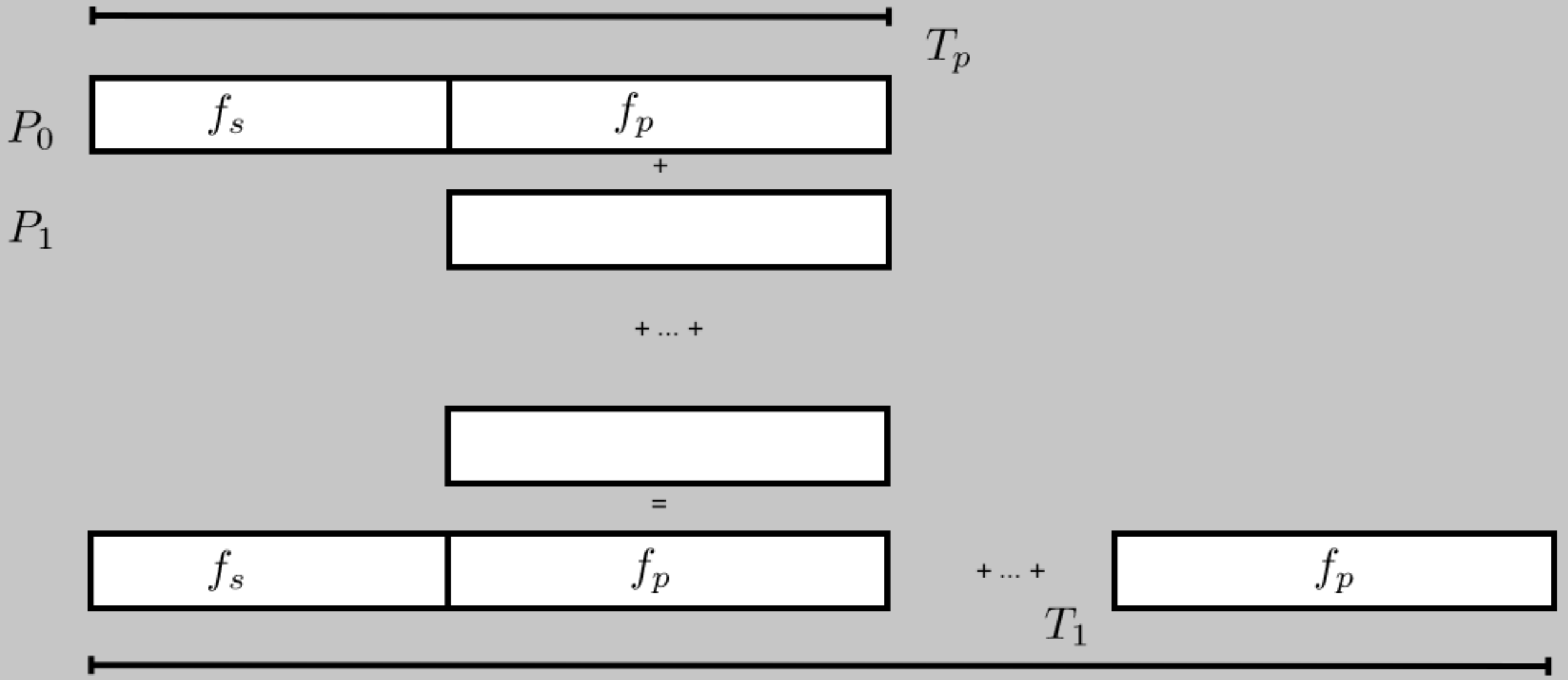
EFFICIENCY AS A FUNCTION OF NUMBER OF PROCESSORS



GUSTAFSON-BARIS OBSERVATION

- Gustafson observed that: -
  - as you increase the number of processors, you have a tendency to attack larger and larger versions of the problem.
  - also, that when you use the same parallel program on larger datasets, the parallel fraction, Fp, increases.

GUSTAFSON'S LAW



GUSTAFSON'S LAW

- Let $T_p = F_s + F_p \equiv 1$
- then $T_1 = F_s + p \,.\, F_p$
- Speedup:

$$S_p = \frac{T_1}{T_p} = \frac{F_s + p.F_p}{F_s + F_p}$$

Assuming $F_s + F_p = 1 \Rightarrow F_s + p.F_p$

$$= F_s + p.(1 - F_s) = p - pF_s + F_s$$

$$= p - (p-1)Fs \rightarrow eq(1)$$

$$= p + (1-p)F_s \rightarrow eq(2)$$

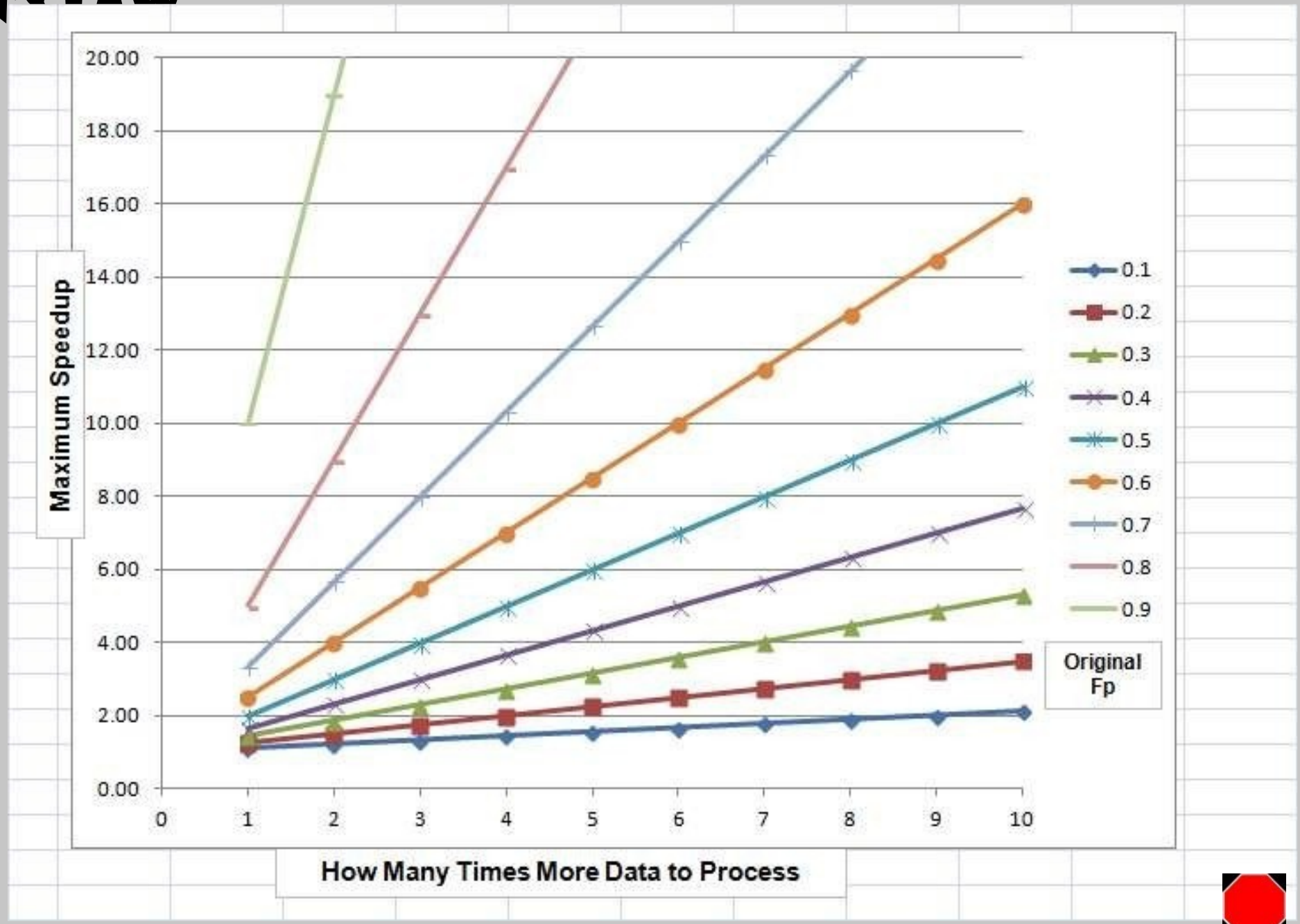
GUSTAFSON'S LAW

$$\text{Speedup} = p - (p - 1)Fs$$

How does this help?

If our problem, size increases, Parallel Fractional part increases.

GUSTAFSON'S LAW



ADDING OVERHEADS

- Communication independent of p: Tp = T1(Fs + Fp/P) + Tc
- assume fully parallelizable: Fp = 1

- then $S_p = \frac{T1}{\frac{T1}{p} + Tc}$

- For reasonable speedup: Tc ≪ T1/p or p ≪ T1/Tc :
- number of processors limited by ratio of scalar execution time and communication overhead

SCALABILITY

- No really precise decision.
- Roughly speaking, a program is said to scale to a certain number of processors p, if going from p-1 to p processors results in some acceptable improvement in speedup (for instance, an increase of 0.5).

SCALABILITY

- Amdahl's law: strong scaling
- same problem over increasing processors

- Often more realistic: weak scaling
- increase problem size with number of processors, for instance keeping memory constant

WHY KEEP SOMETHING SEQUENTIAL?

- Some parts of the program are not parallelizable (because of dependences)

- Some parts may be parallelizable, but the overhead dwarfs the increased speedup.

When can two statements execute in parallel?

When can two statements execute in parallel?

- On one processor:

  statement 1;

  statement 2;

- On two processors:

processor1:	processor2:
statement1;	statement2;

FUNDAMENTAL ASSUMPTION

- Processors execute independently: no control over order of execution between processors

When can 2 statements execute in parallel?

- Possibility 1

Processor1:

statement1;

Processor2:

statement2;

- Possibility 2

Processor1:

statement1;

Processor2:

statement2:

When can 2 statements execute in parallel?

Their order of execution must not matter!

When can 2 statements execute in parallel?

- Their order of execution must not matter!

- In other words,

  statement1; statement2;

  must be equivalent to

  statement2; statement1;

EXAMPLE 1

```
a = 1;

b = a;
```

- Statements cannot be executed in parallel
- Program modifications may make it possible.

EXAMPLE 2

```
a = f(x);

b = a;
```

- May not be wise to change the program (sequential execution would take longer).

EXAMPLE 3

```
a = 1;
a = 2;
```

- Statements cannot be executed in parallel.

TRUE DEPENDENCE

Statements S1, S2

S2 has a true dependence on S1

iff

S2 reads a value written by S1

ANTI-DEPENDENCE

Statements S1, S2.

S2 has an anti-dependence on S1

iff

S2 writes a value read by S1.

OUTPUT DEPENDENCE

Statements S1, S2.

S2 has an output dependence on S1

iff

S2 writes a variable written by S1.

When can 2 statements execute in parallel?

S1 and S2 can execute in parallel

iff

there are no dependences between S1 and S2

- true dependences
- anti-dependences
- output dependences

Some dependences can be removed.

EXAMPLE 4

- Most parallelism occurs in loops.

```
for(i=0; i<100; i++)
    a[i] = i;
```

- No dependences.
- Iterations can be executed in parallel.

EXAMPLE 5

```
for(i=0; i<100; i++) {
    a[i] = i;
    b[i] = 2*i;
}
```

Iterations and statements can be executed in parallel.

EXAMPLE 6

```
for(i=0;i<100;i++) a[i] = i;
for(i=0;i<100;i++) b[i] = 2*i;
```

Iterations and loops can be executed in parallel.

EXAMPLE 7

```
for(i=0; i<100; i++)
    a[i] = a[i] + 100;
```

- There is a dependence … on itself!
- Loop is still parallelizable.

EXAMPLE 8

```
for( i=0; i<100; i++ )
  a[i] = f(a[i-1]);
```

- Dependence between a[i] and a[i-1].
- Loop iterations are not parallelizable.

Loop-carried dependence

- A loop carried dependence is a dependence that is present only if the statements are part of the execution of a loop.
- Otherwise, we call it a loop-independent dependence.
- Loop-carried dependences prevent loop iteration parallelization.

EXAMPLE 9

```
for(i=0; i<100; i++ )
  for(j=0; j<100; j++ )
      a[i][j] = f(a[i][j-1]);
```

- Loop-independent dependence on i.
- Loop-carried dependence on j.
- Outer loop can be parallelized, inner loop cannot.

EXAMPLE 10

```
for( j=0; j<100; j++ )
  for( i=0; i<100; i++ )
      a[i][j] = f(a[i][j-1]);
```

- Inner loop can be parallelized, outer loop cannot.
- Less desirable situation.
- Loop interchange is sometimes possible.

Level of Loop-Carried Dependence

- Is the nesting depth of the loop that carries the dependence.
- Indicates which loops can be parallelized.

BE CAREFUL … EXAMPLE 11

```
printf(“a”);
printf(“b”);
```

Statements have a hidden output dependence due to the output stream.

BE CAREFUL … EXAMPLE 12

```
a = f(x);
b = g(x);
```

Statements could have a hidden dependence if f and g update the same variable.

Also depends on what f and g can do to x.

BE CAREFUL … EXAMPLE 13

```
for(i=0; i<100; i++)
  a[i+10] = f(a[i]);
```

- Dependence between a[10], a[20], …
- Dependence between a[11], a[21], …
- …
- Some parallel execution is possible.

BE CAREFUL … EXAMPLE 14

```
for( i=1; i<100;i++ ) {
        a[i] = …;
        … = a[i-1];
}
```

- Dependence between a[i] and a[i-1]
- Complete parallel execution impossible
- Pipelined parallel execution possible

BE CAREFUL … EXAMPLE 15

```
for( i=0; i<100; i++ )
    a[i] = f(a[indexa[i]]);
```

- Cannot tell for sure.
- Parallelization depends on user knowledge of values in indexa[].
- User can tell, compiler cannot.

AN ASIDE

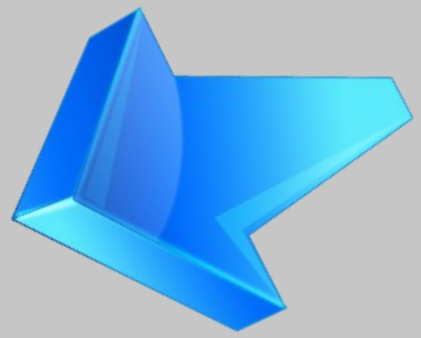
- Parallelizing compilers analyze program dependences to decide parallelization.
- In parallelization by hand, user does the same analysis.
- Compiler more convenient and more correct
- User more powerful, can analyze more patterns.

TO REMEMBER

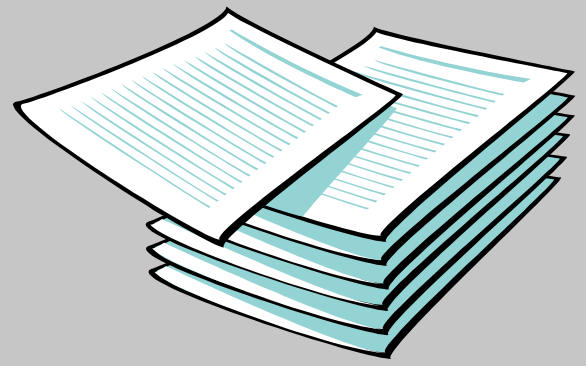
- Statement order must not matter.
- Statements must not have dependences.
- Some dependences can be removed.
- Some dependences may not be obvious.

Serial Hardware and Software

input



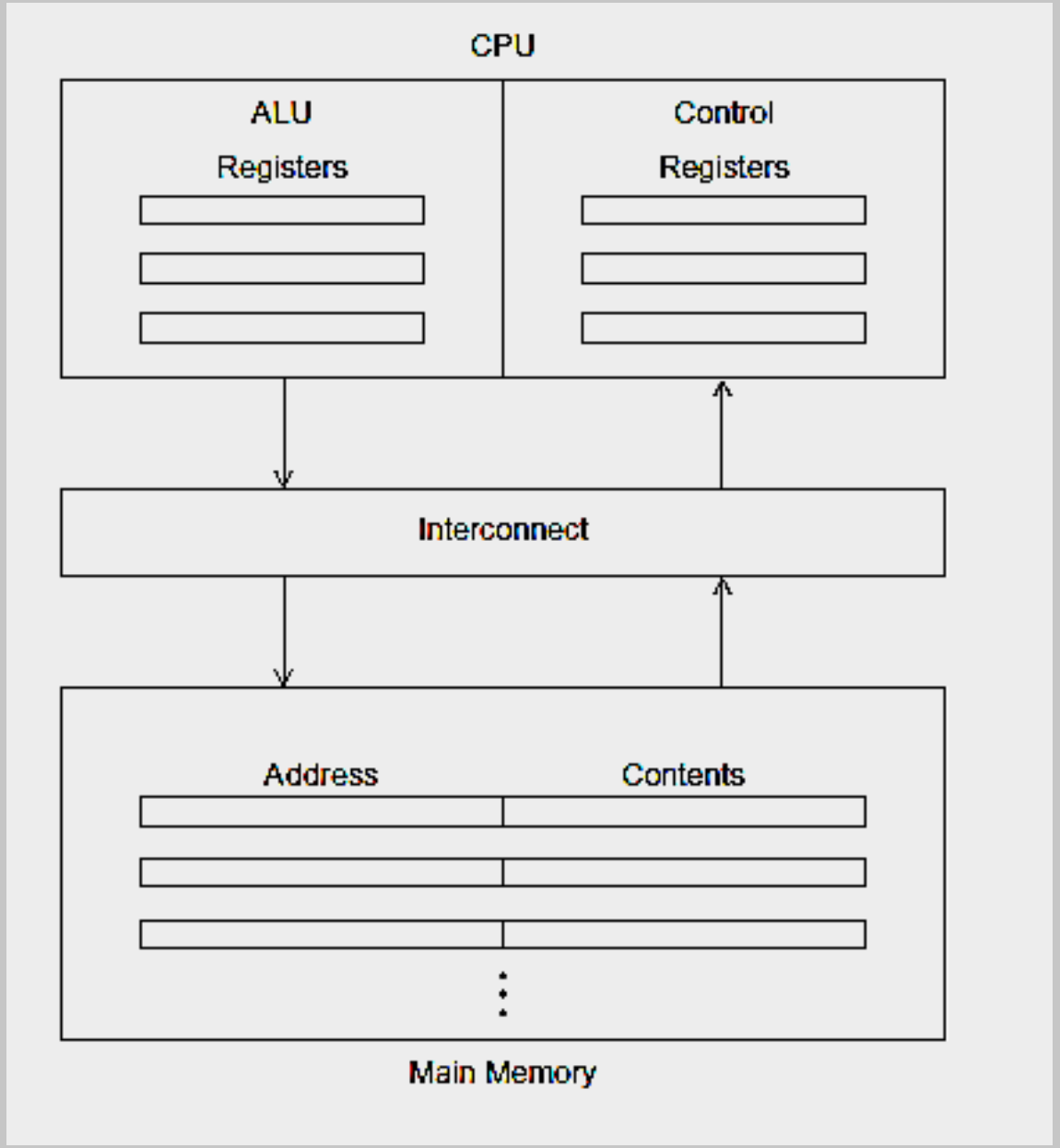
programs



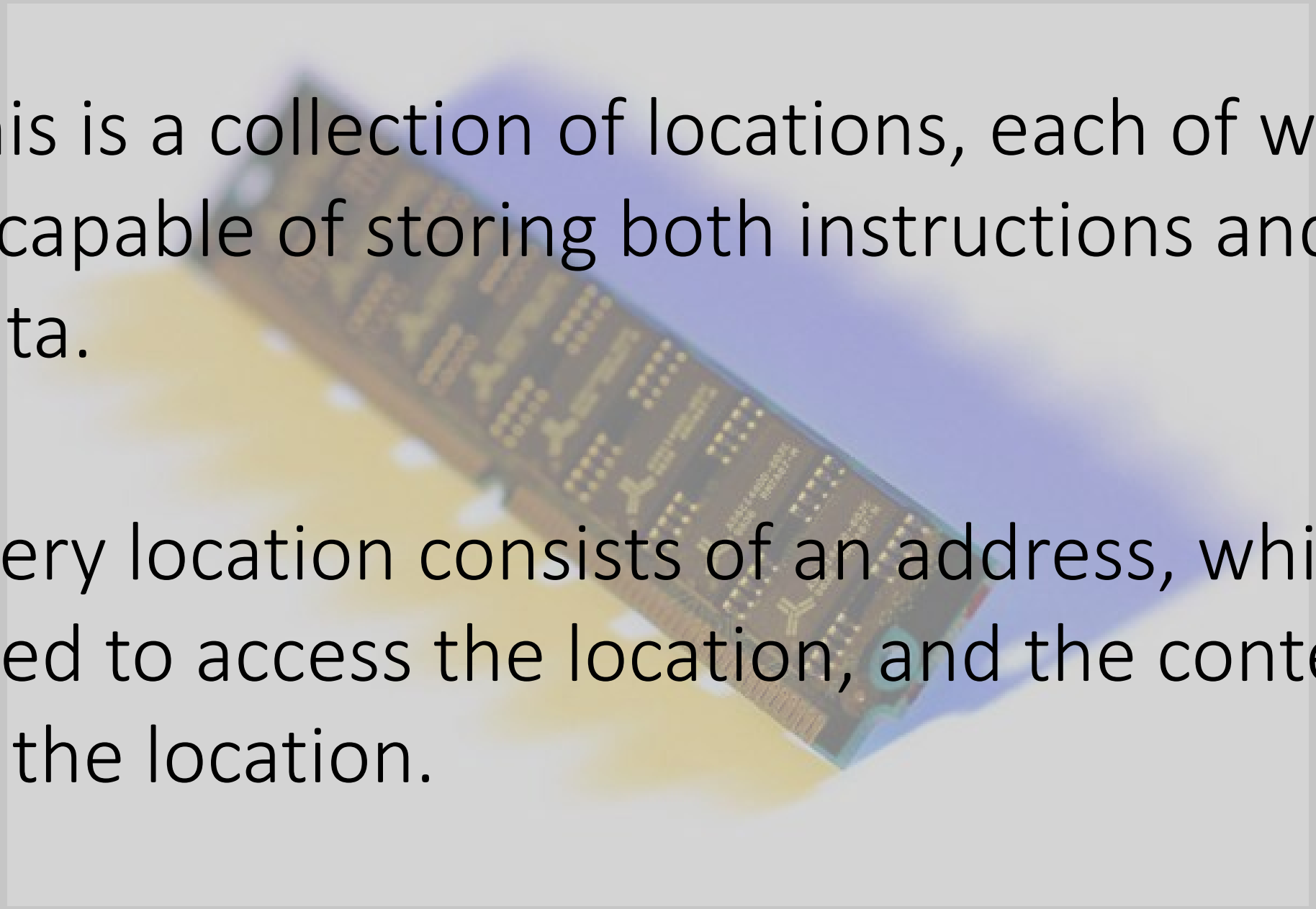
output

Computers runs one program at a time.

THE VON NEUMANN ARCHITECTURE

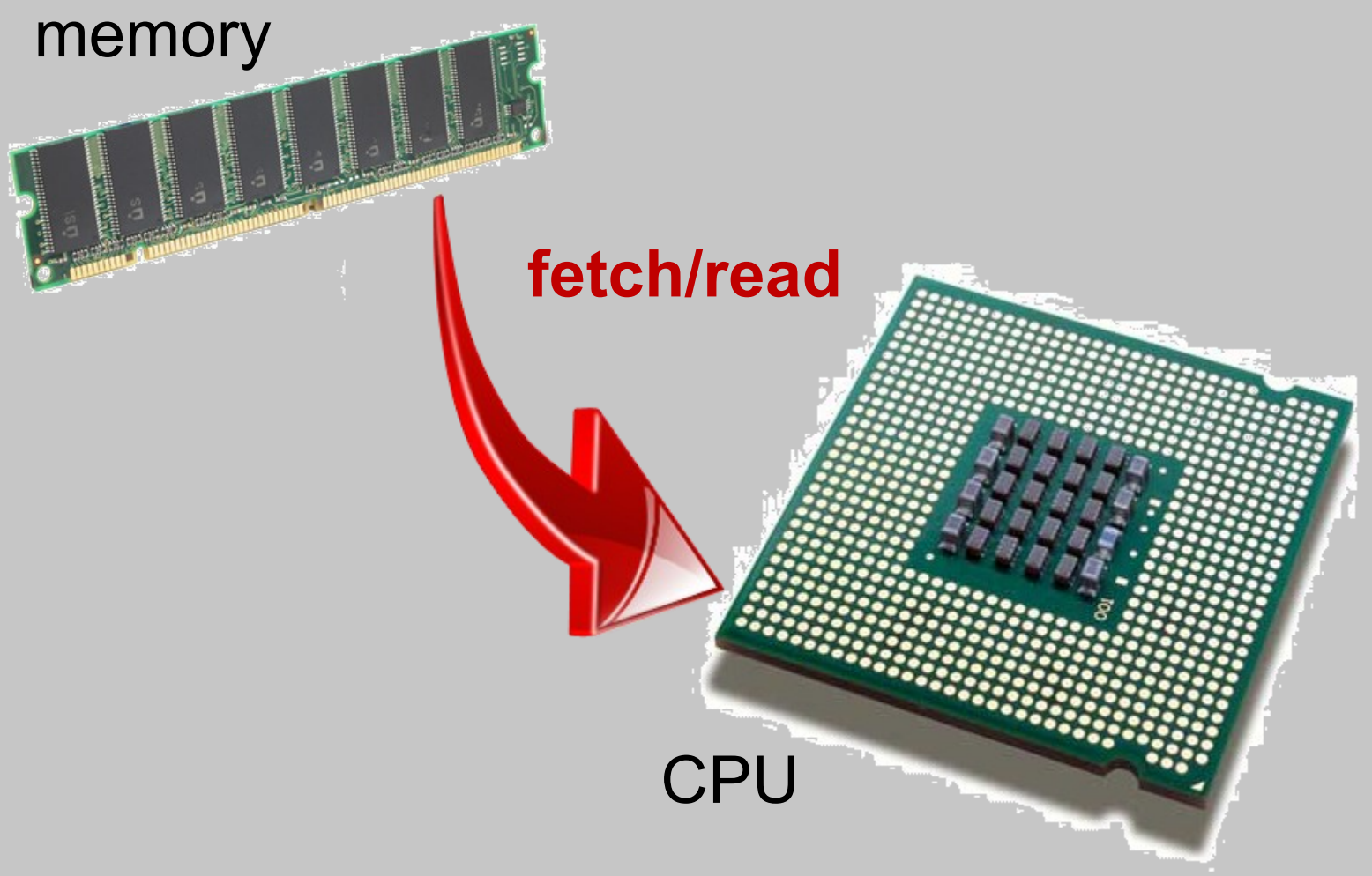


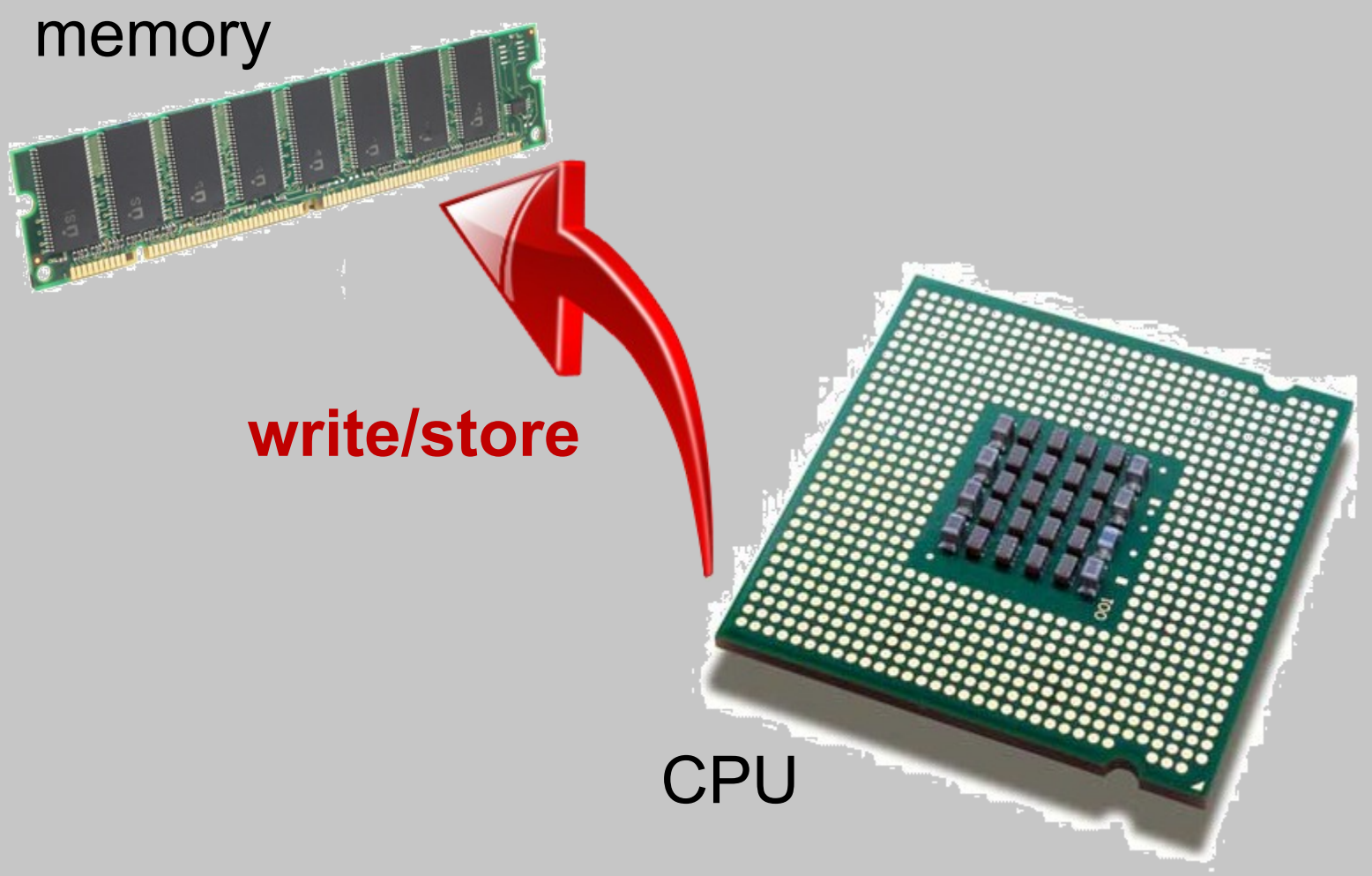
MAIN MEMORY



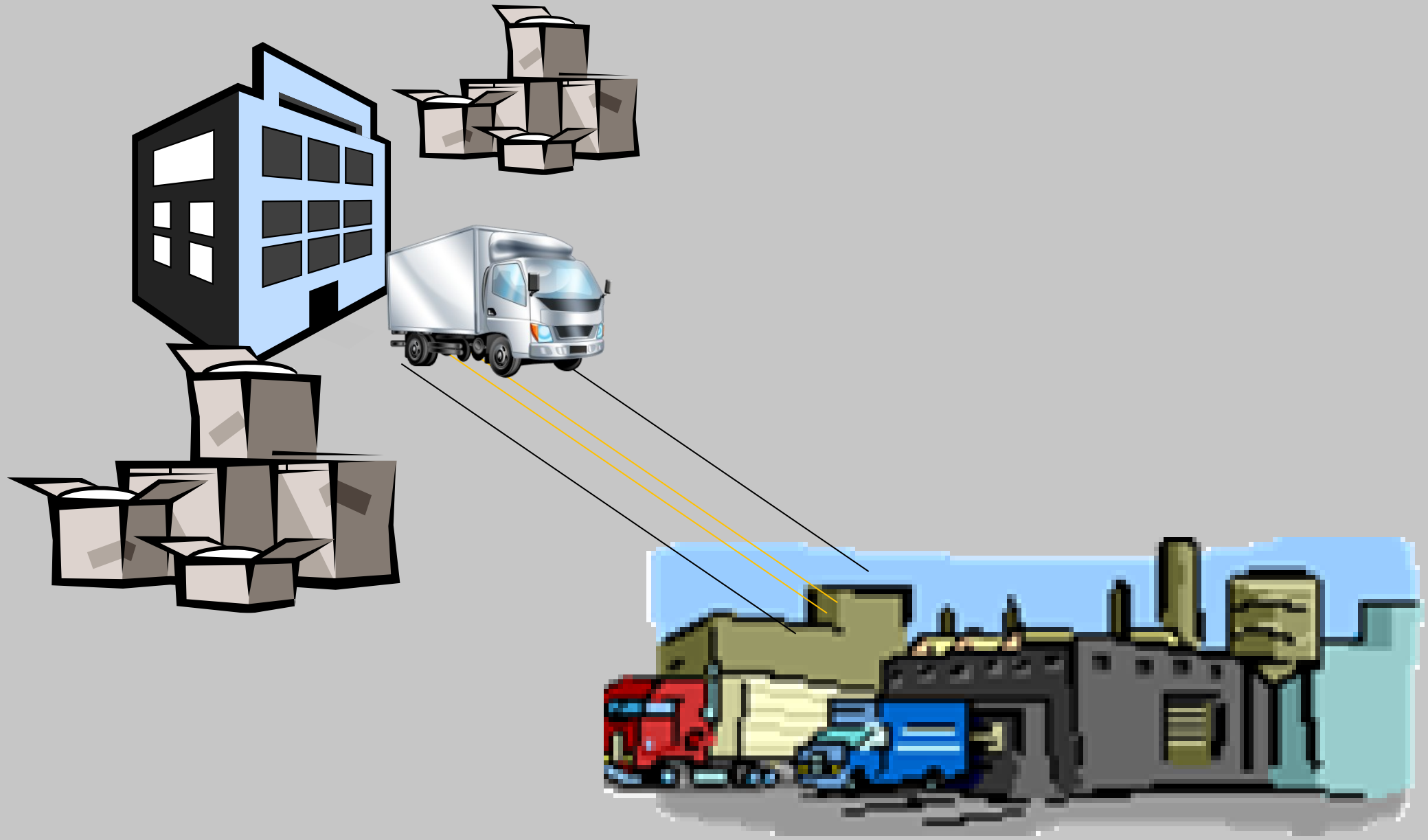
- This is a collection of locations, each of which is capable of storing both instructions and data.

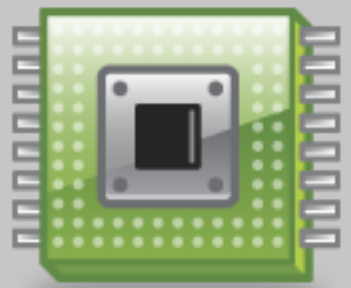
- Every location consists of an address, which is used to access the location, and the contents of the location.





VON NEUMANN BOTTLENECK





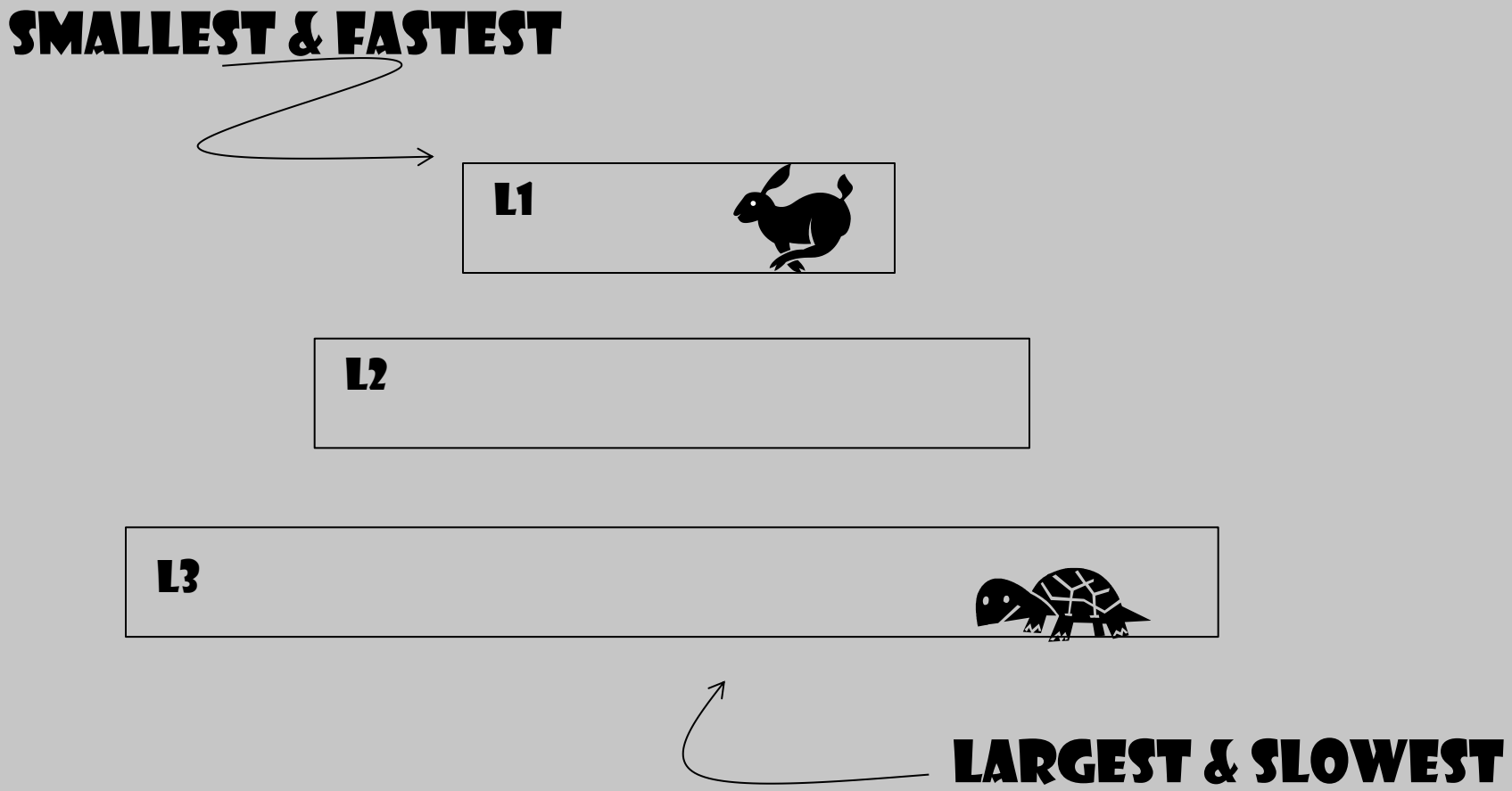
Modifications to the von Neumann model

BASICS OF CACHING

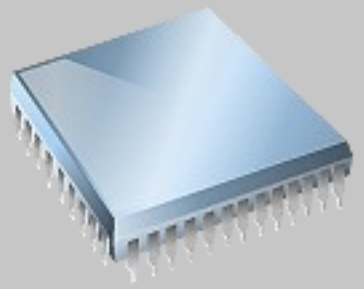
- A collection of memory locations that can be accessed in less time than some other memory locations.

- A CPU cache is typically located on the same chip, or one that can be accessed much faster than ordinary memory.

LEVELS OF CACHE



CACHE HIT



FETCH X

L1

L2

L3

ISSUES WITH CACHE

- When a CPU writes data to cache, the value in cache may be inconsistent with the value in main memory.

- **Write-through** caches handle this by updating the data in main memory at the time it is written to cache.

- **Write-back** caches mark data in the cache as dirty. When the cache line is replaced by a new cache line from memory, the dirty line is written to memory.

Instruction Level Parallelism (ILP)

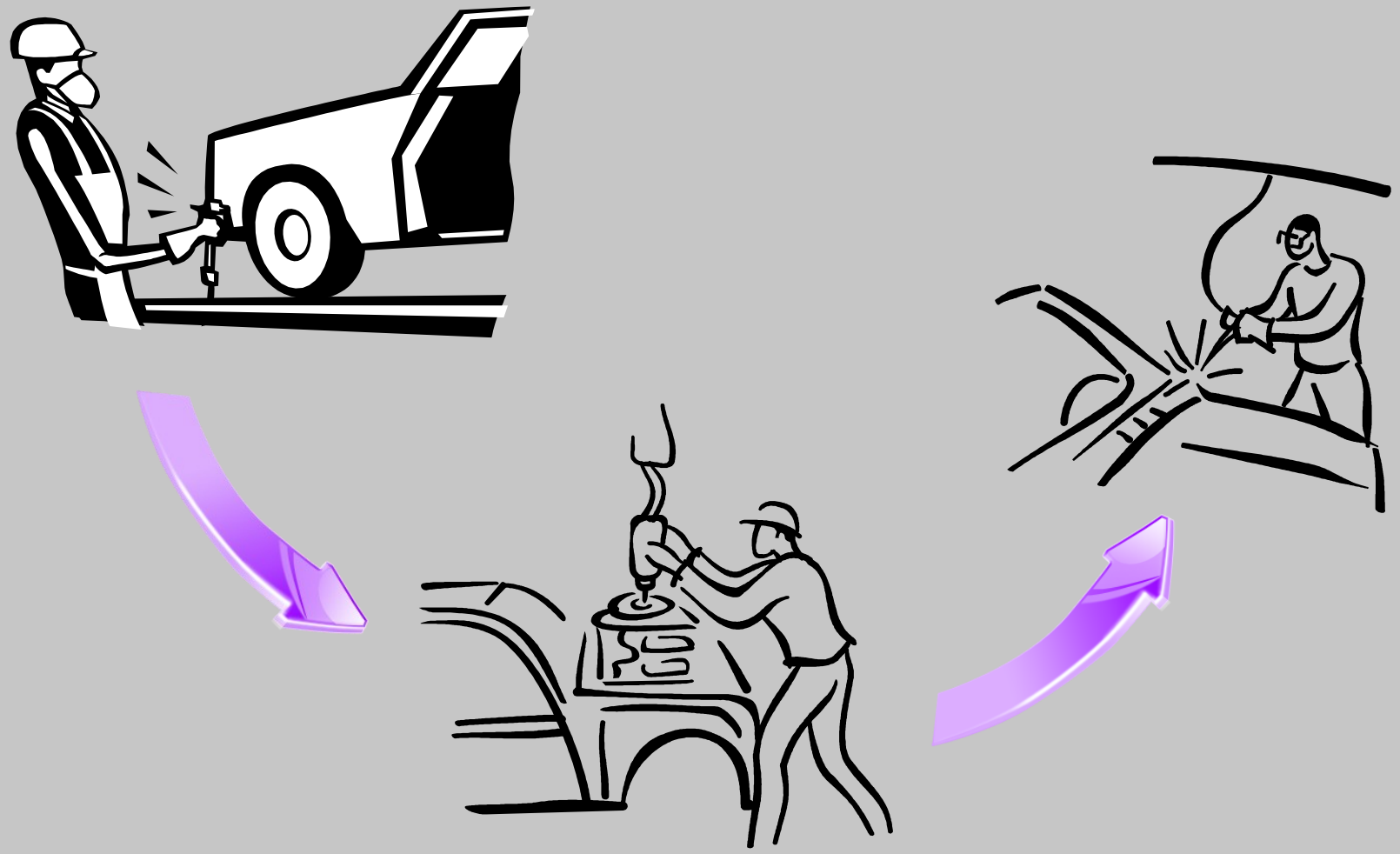
- Attempts to improve processor performance by having multiple processor components or functional units simultaneously executing instructions.

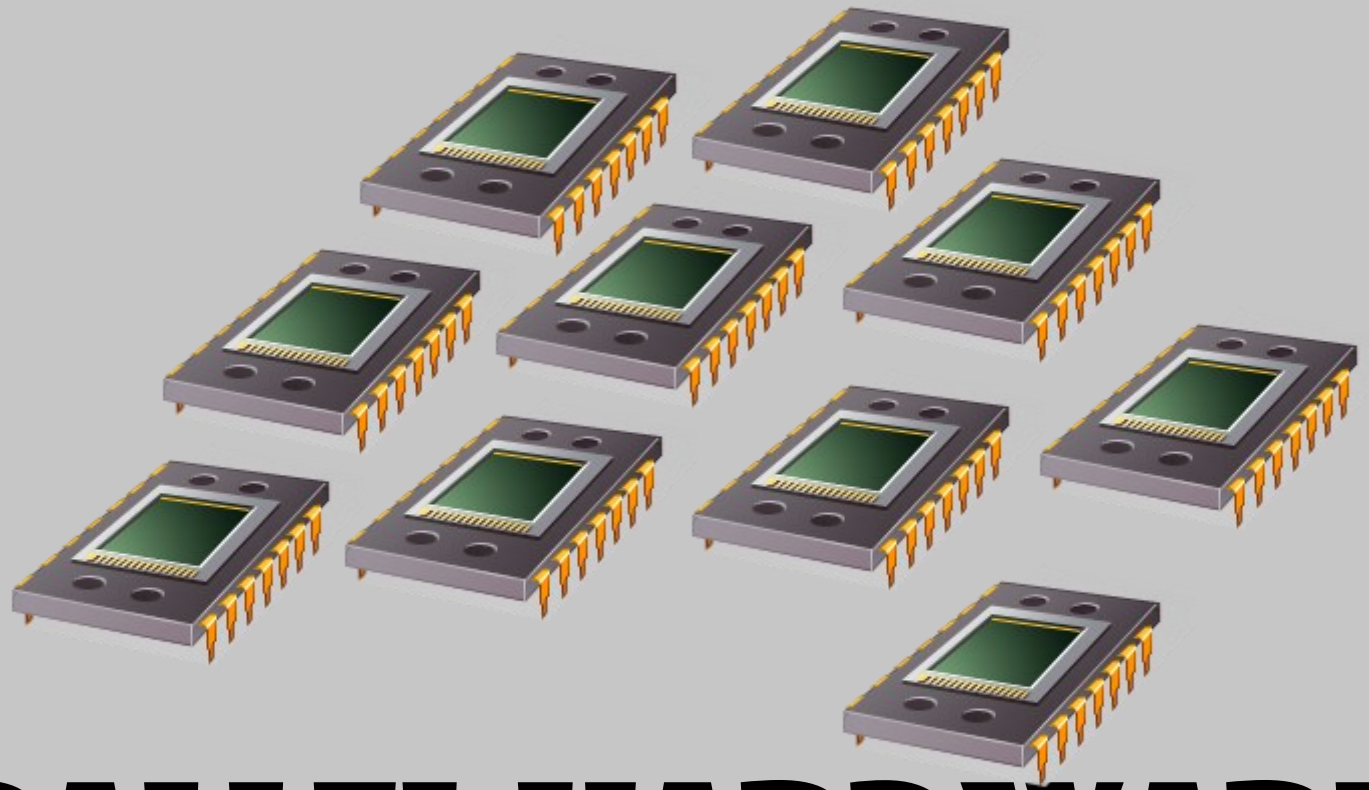
Instruction Level Parallelism (2)

- Pipelining - functional units are arranged in stages.

- Multiple issue - multiple instructions can be simultaneously initiated.

PIPELINING





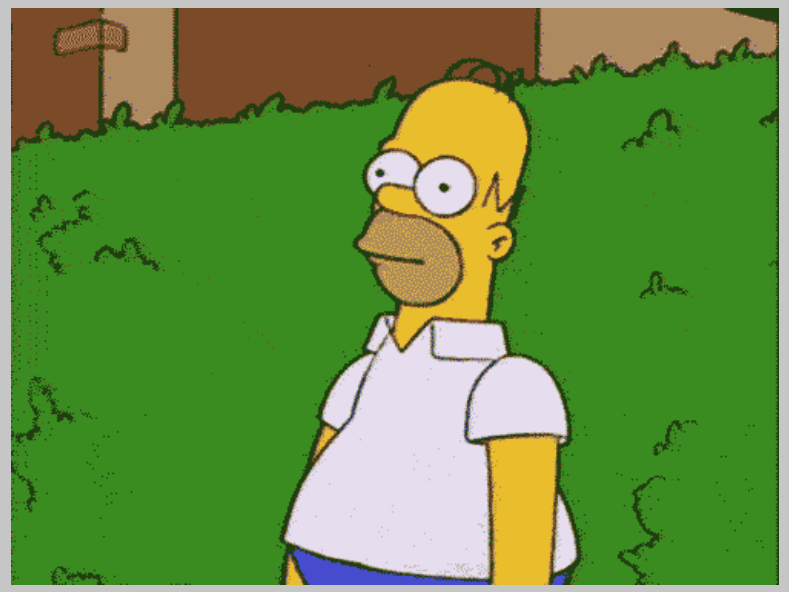
Parallel Hardware

A programmer can write code to exploit.

MODELS OF PARALLELISM

MISD

redundant computing for fault tolerance?



SIMD

data parallelism, pipelining, array processing, vector instructions

SISD

used to be single processor, now single core

Flynns Taxonomy

MIMD

independent processors, clusters, MPPs

LINKS

- https://medium.datadriveninvestor.com/towards-multithreading-an-introduction-to-flynns-taxonomy-bee0761ac90b

- https://en.namu.wiki/w/%ED%94%8C%EB%A6%B0%20%EB%B6%84%EB%A5%98

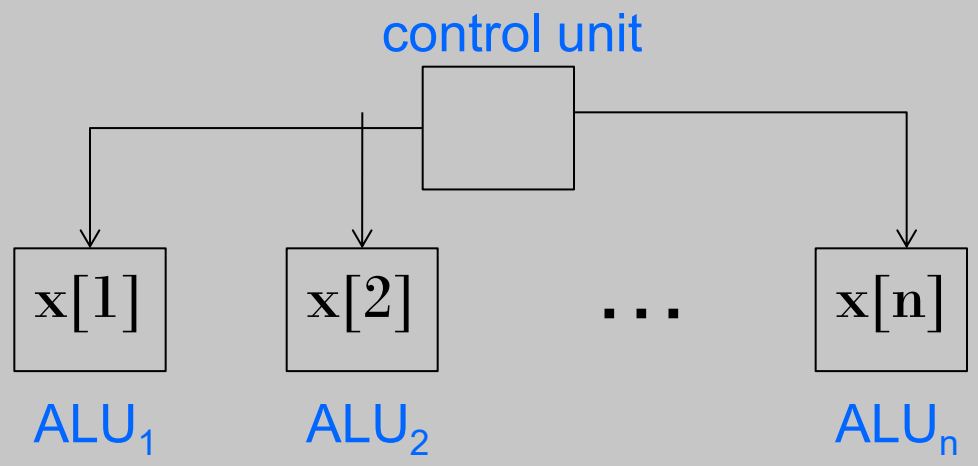
Flynn's Taxonomy

SISD Single instruction stream Single data stream classic von Neumann	(SIMD) Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream not covered	(MIMD) Multiple instruction stream Multiple data stream

SIMD

- Parallelism achieved by dividing data among the processors.
- Applies the same instruction to multiple data items.
- Called data parallelism.
- Relies on streams of identical operations
- Recurrences hard to accommodate

SIMD EXAMPLE




```
for (i = 0; i < n; i++)
    x[i] += y[i];
```

n data items

n ALUs

SIMD Drawbacks

- All ALUs are required to execute the same instruction, or remain idle.
- In classic design, they must also operate synchronously.
- The ALUs have no instruction storage.
- Efficient for large data parallel problems, but not other types of more complex parallel problems.

VECTOR PROCESSORS (1)

- Operate on arrays or vectors of data while conventional CPU's operate on individual data elements or scalars.

- Vector registers.
  - Capable of storing a vector of operands and operating simultaneously on their contents.

Vector Processors (2)

- Vectorized and pipelined functional units.
  - The same operation is applied to each element in the vector (or pairs of elements).

- Vector instructions.
  - Operate on vectors rather than scalars.

VECTOR PROCESSORS (3)

- Interleaved memory.
  - Multiple “banks” of memory, which can be accessed more or less independently.
  - Distribute elements of a vector across multiple banks, so reduce or eliminate delay in loading/storing successive elements.
- Strided memory access and hardware scatter/gather.
  - The program accesses elements of a vector located at fixed intervals.

Vector Processors - Pros

- Fast.
- Easy to use.
- Vectorizing compilers are good at identifying code to exploit.
- Compilers also can provide information about code that cannot be vectorized.
  - Helps the programmer re-evaluate code.
- High memory bandwidth.
- Uses every item in a cache line.

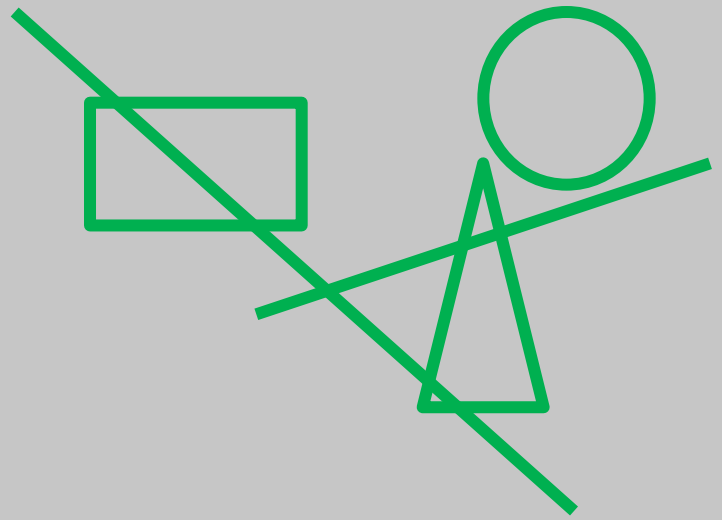
VECTOR PROCESSORS - CONS

- They don’t handle irregular data structures as well as other parallel architectures.

- A very finite limit to their ability to handle ever larger problems. (scalability)

GRAPHICS PROCESSING UNITS (GPU)

- Real time graphics application programming interfaces or API's use points, lines, and triangles to internally represent the surface of an object.



GPUS

- A graphics processing pipeline converts the internal representation into an array of pixels that can be sent to a computer screen.

- Several stages of this pipeline (called shader functions) are programmable.
  - Typically, just a few lines of C code.

GPUs

- Shader functions are also implicitly parallel, since they can be applied to multiple elements in the graphics stream.

- GPU's can often optimize performance by using SIMD parallelism.
- The current generation of GPU's use SIMD parallelism.
  - Although they are not pure SIMD systems.

MIMD

- Supports multiple simultaneous instruction streams operating on multiple data streams.

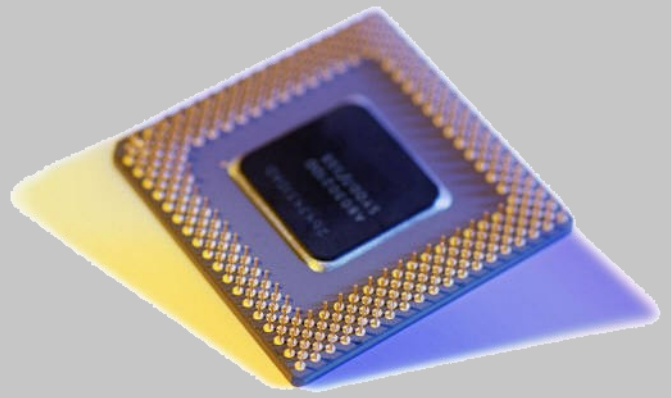
- Typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU.

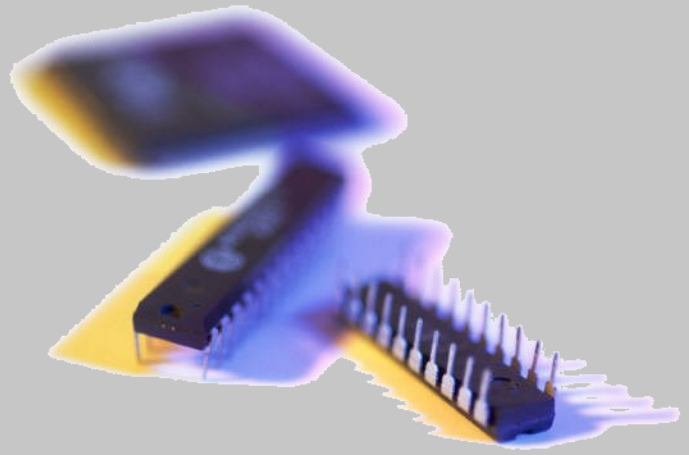
Shared Memory System (1)

- A collection of autonomous processors is connected to a memory system via an interconnection network.
- Each processor can access each memory location.
- The processors usually communicate implicitly by accessing shared data structures.

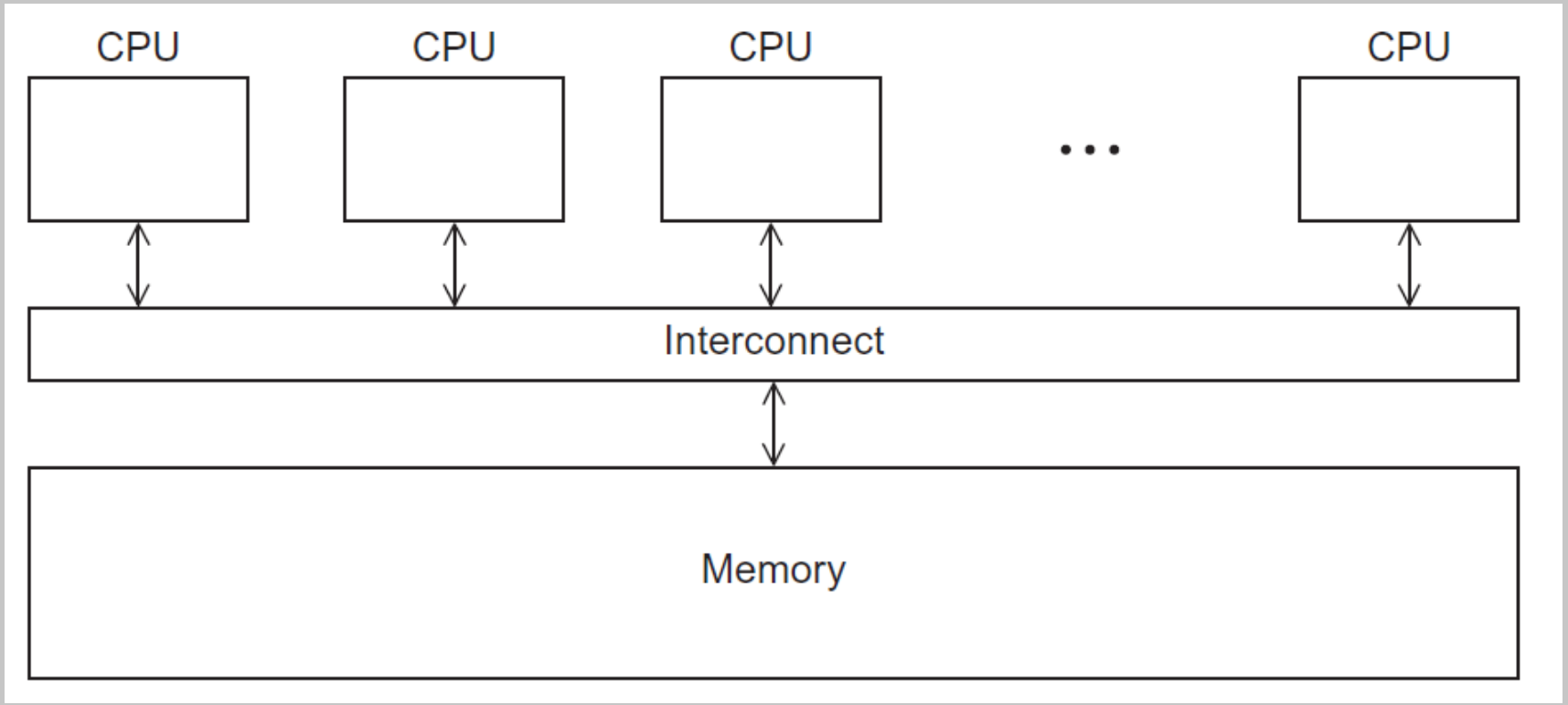
Shared Memory System (2)

- Most widely available shared memory systems use one or more multicore processors.
  - (multiple CPU's or cores on a single chip)

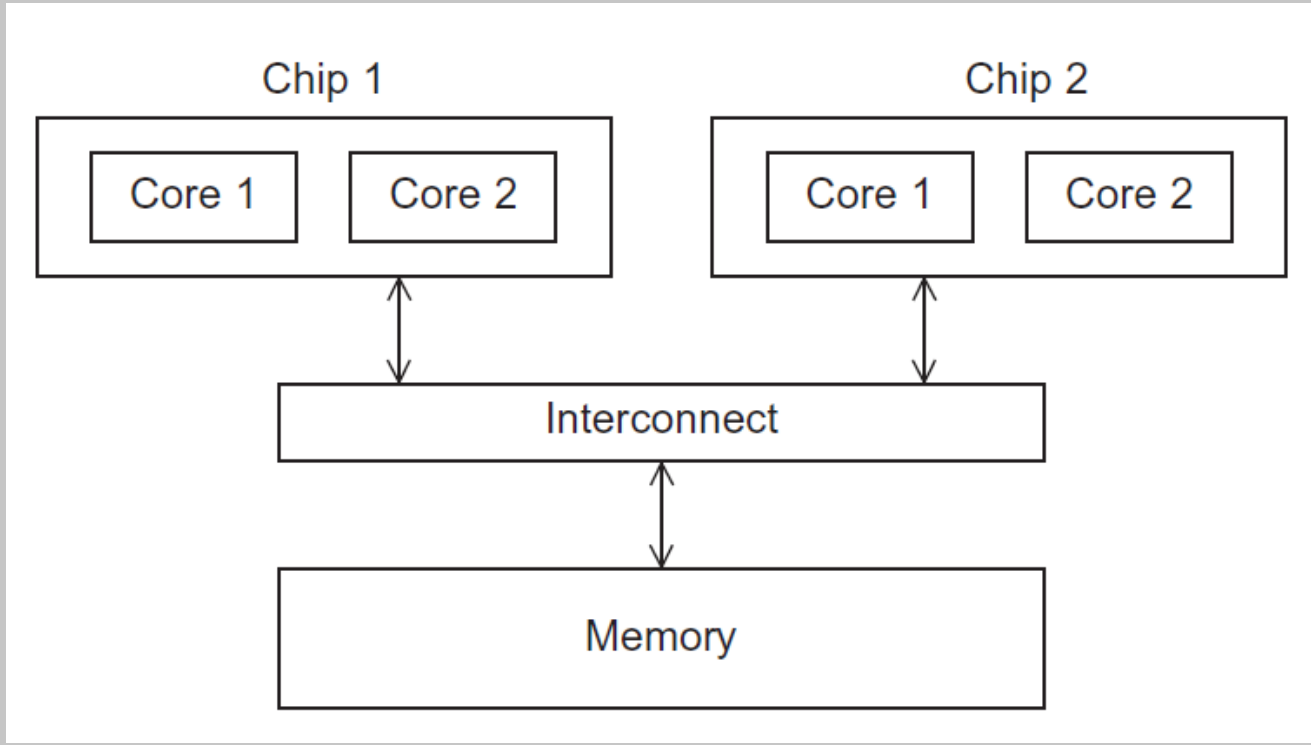




Shared Memory System

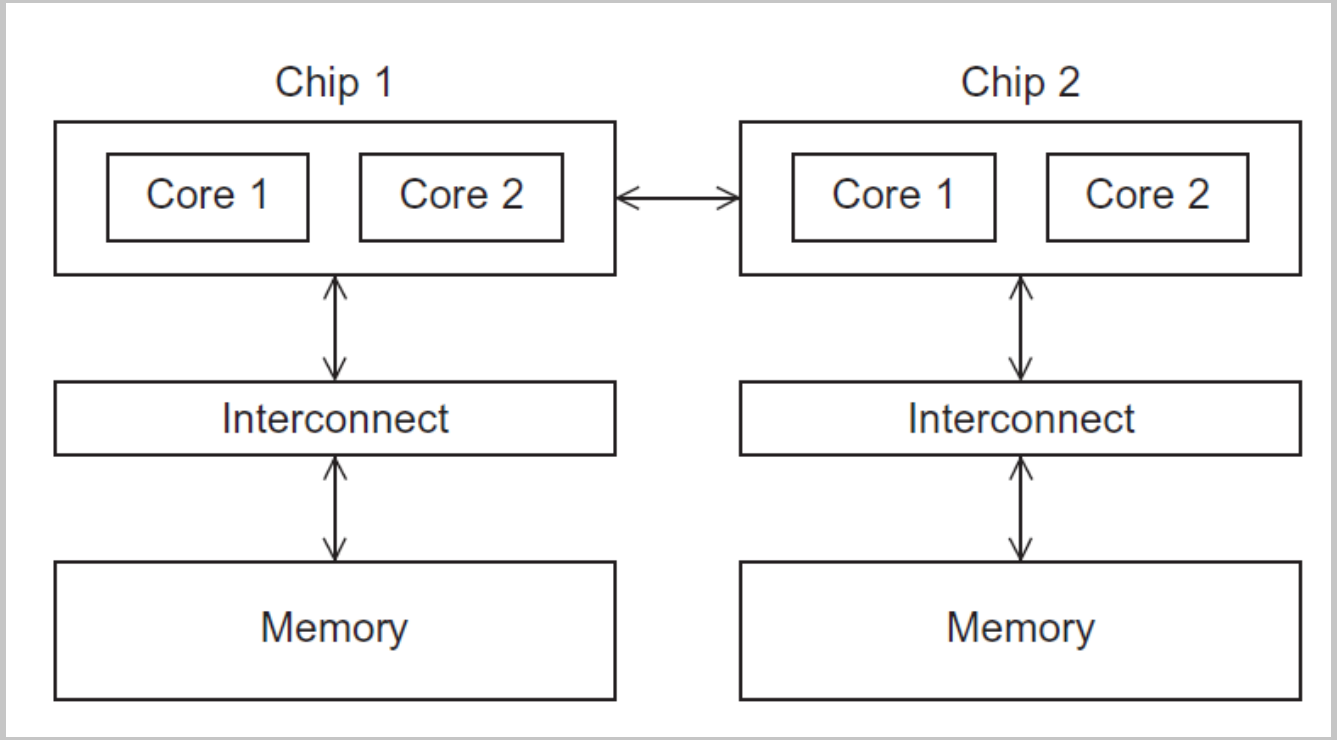


UMA MULTICORE SYSTEM



Time to access all the memory locations will be the same for all the cores.

NUMA MULTICORE SYSTEM



A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

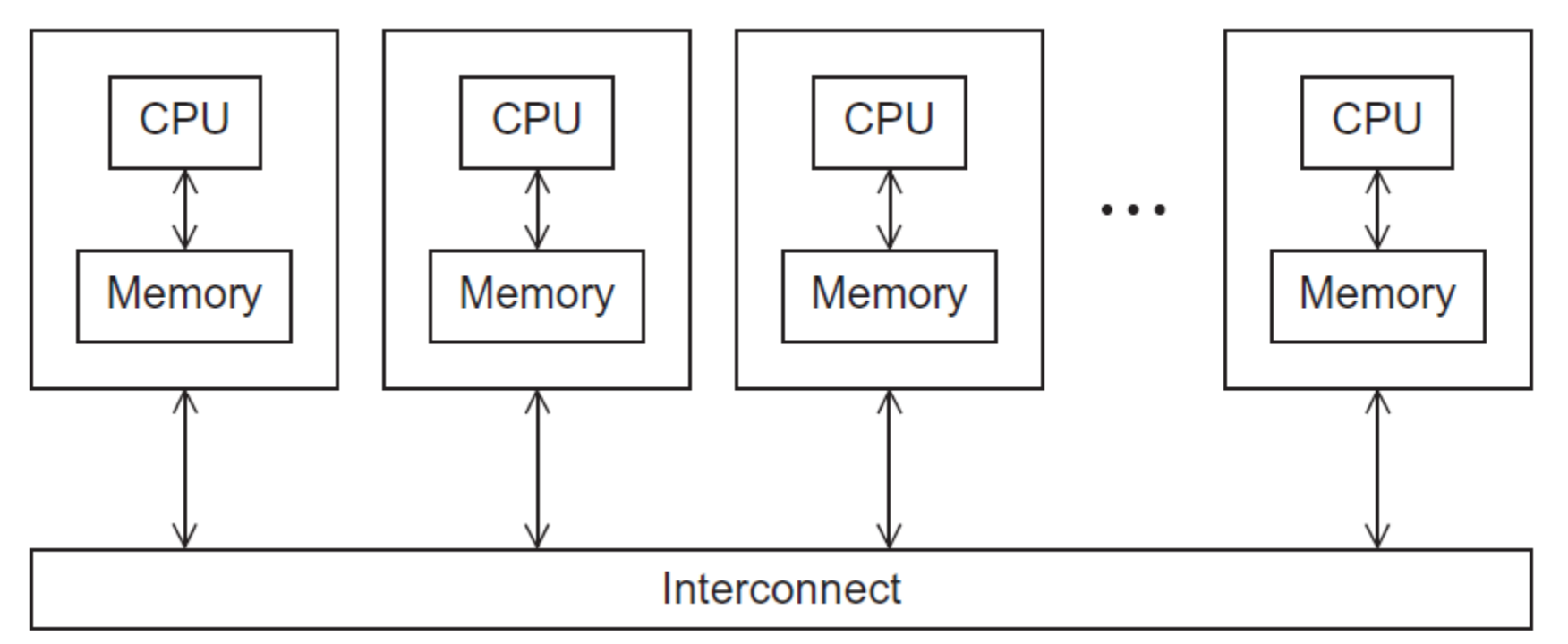
DISTRIBUTED MEMORY SYSTEM

- Clusters (most popular)
  - A collection of commodity systems.
  - Connected by a commodity interconnection network.

- Nodes of a cluster are individual computations units joined by a communication network.

a.k.a. hybrid systems

DISTRIBUTED MEMORY SYSTEM



Interconnection Networks

- Affects performance of both distributed and shared memory systems.

- Two categories:
  - Shared memory interconnects
  - Distributed memory interconnects

Shared Memory Interconnects

- Bus interconnect
  - A collection of parallel communication wires together with some hardware that controls access to the bus.
  - Communication wires are shared by the devices that are connected to it.
  - As the number of devices connected to the bus increases, contention for use of the bus increases, and performance decreases.

New Branches in the Taxonomy

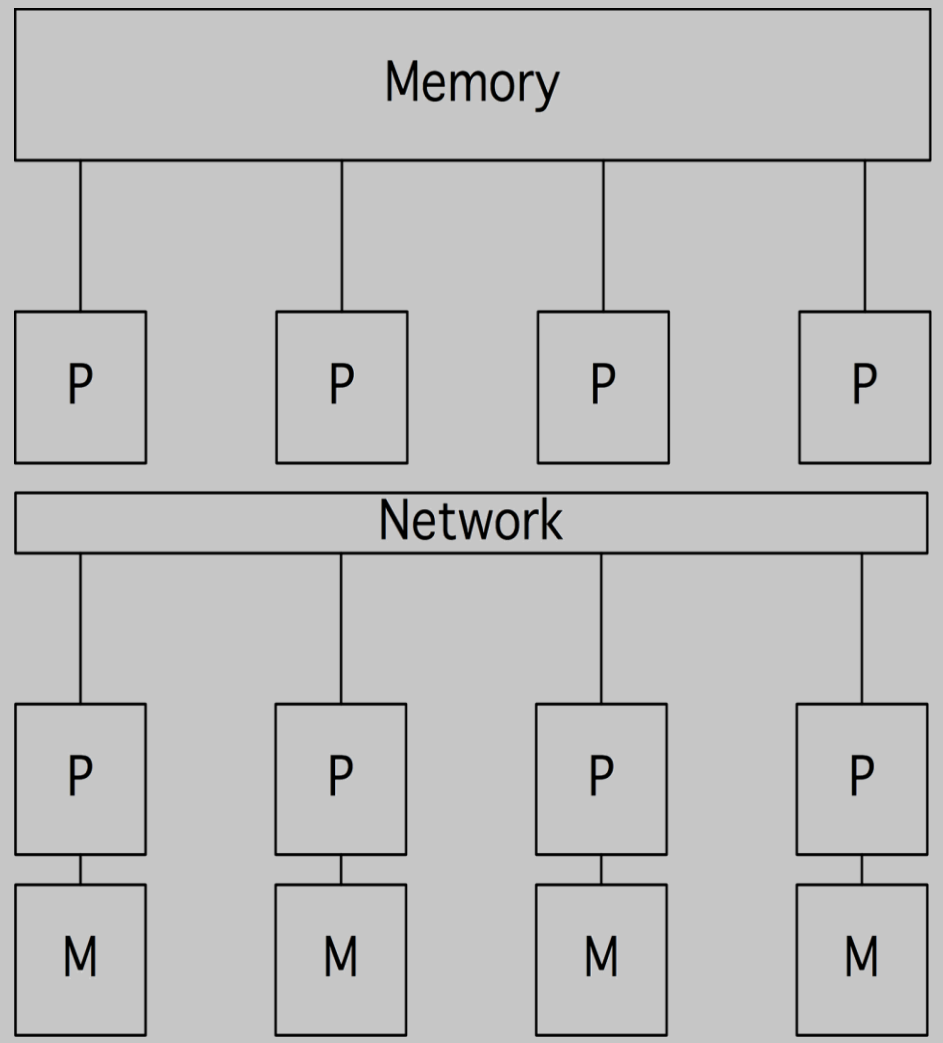
- SPMD: single program multiple data the way clusters are actually used

- SIMT: single instruction multiple threads the GPU model

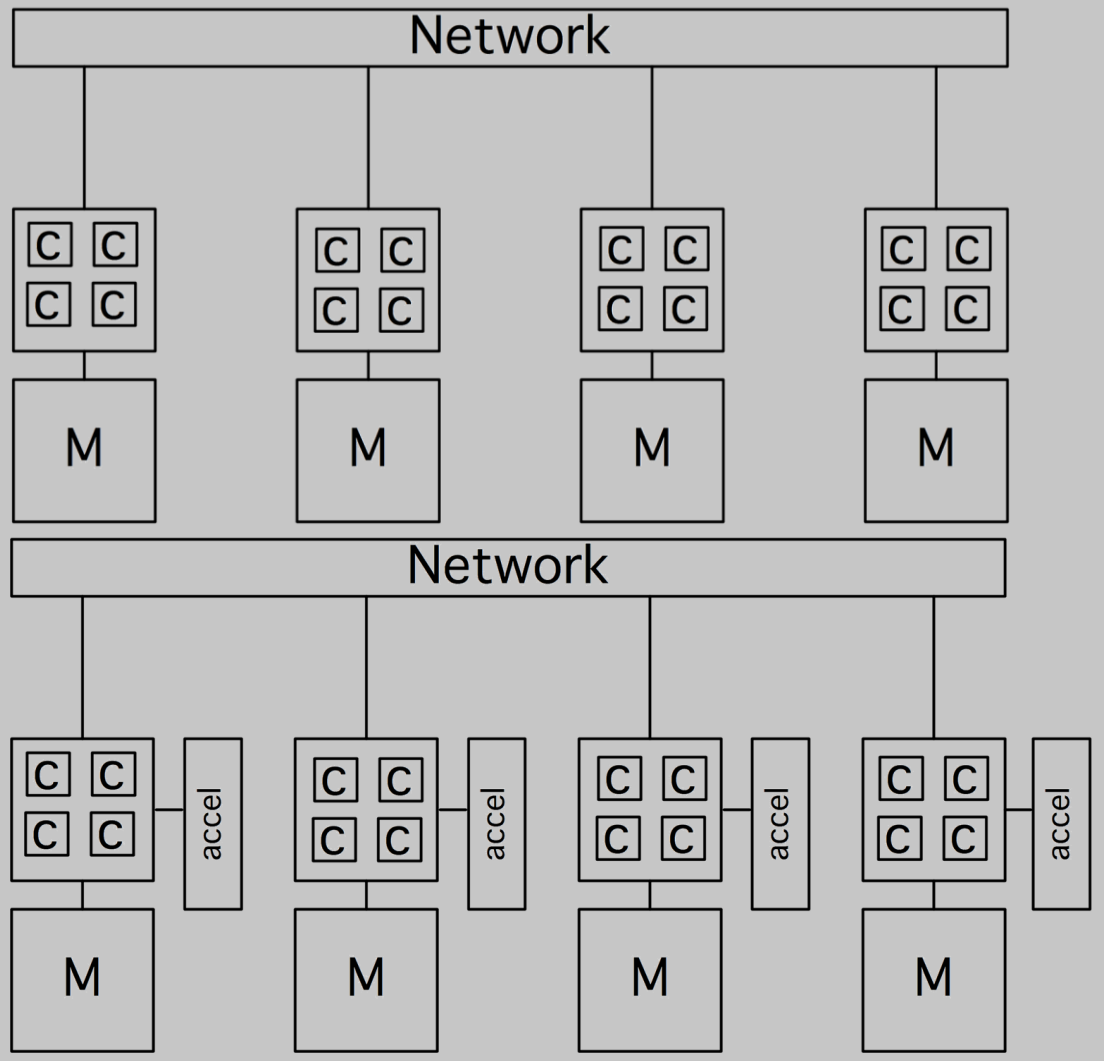
MIMD BECOMES SPMD

- MIMD: independent processors, independent instruction streams, independent data
- In practice very little true independence: usually the same executable
- Single Program Multiple Data
- Exceptional example: climate codes
- Old-style SPMD: cluster of single-processor nodes
- New-style: cluster of multicore nodes, ignore shared caches / memory
- (We'll get to hybrid computing in a minute)

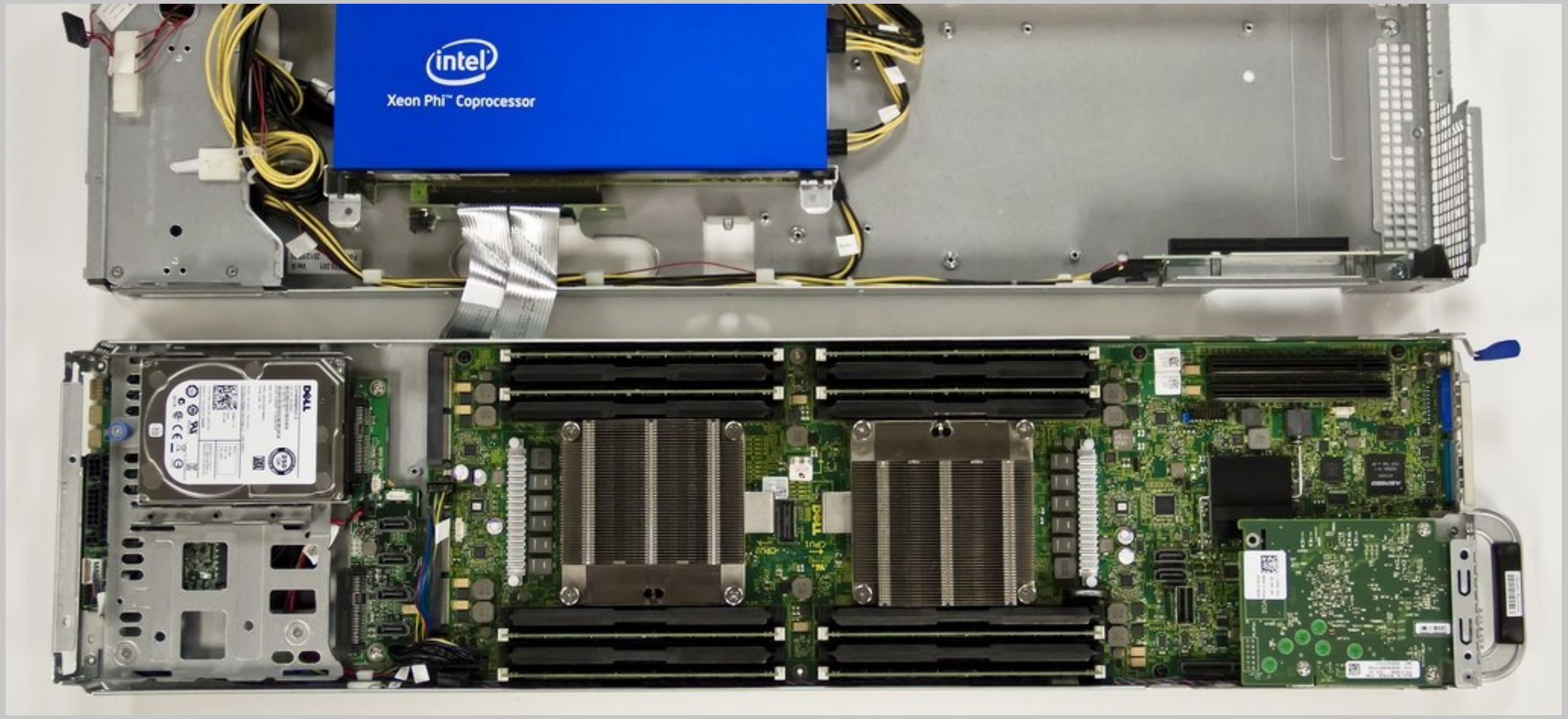
MAJOR TYPES OF MEMORY ORGANIZATION, CLASSIC



MAJOR TYPES OF MEMORY ORGANIZATION, CONTEMPORARY



PICTURE OF NUMA



MESH CLUSTERS

END OF SECTION

MPI PROGRAMMING

An introduction

IDENTIFYING MPI PROCESSES

- Common practice to identify processes by nonnegative integer ranks.

- *p* processes are numbered *0, 1, 2, .. p-1*

MPI HELLO WORLD

```
1.  int main(int argc, char* argv[]) {
2.  int my_rank, world_size;
3.  MPI_Init(NULL, NULL);
4.  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
5.  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
6.  // Get the name of the processor
7.  char processor_name[MPI_MAX_PROCESSOR_NAME];
8.  int name_len;
9.  MPI_Get_processor_name(processor_name, &name_len);
10. printf("Hello from CPU '%s' process %d out of %d
    processes!", processor_name,my_rank, world_size);
11. MPI_Finalize();
12. return 0; }
```

MPI_Init(argc , argv)

```
1.  int MPI_Init( int *argc,
2.                char ***argv );

3.  //Both can be null
```

- This function must be called by one thread only.
- That thread will be known as the “Main Thread” and
- must be the same thread to call MPI_Finalize.

MPI_FINALIZE

```
1.  int MPI_Finalize(void);
```

- Terminates the calling MPI process's execution environment.

- All MPI Processes must call this routine before exiting on the thread that called MPI_Init

RANK AND WORLD

1. `MPI_Comm_size(MPI_COMM_WORLD, &world_size);`

2. `MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);`

COMMUNICATORS

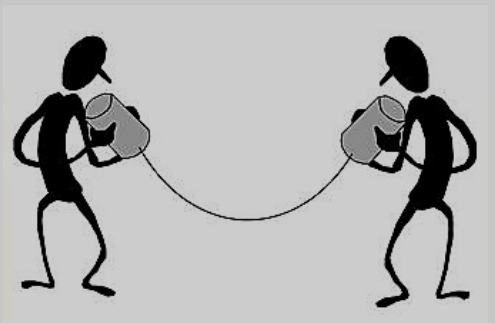
- A collection of processes that can send messages to each other.
- MPI_Init defines a communicator that consists of all the processes created when the program is started.
- Called MPI_COMM_WORLD.

SPMD

- Single-Program Multiple-Data
- We compile one program.
- Process 0 does something different.
  - Receives messages and prints them while the other processes do the work.
- The if-else construct makes our program SPMD.

COMMUNICATION

```
int MPI_Send(
      void*          msg_buf_p      /* in */,
      int            msg_size       /* in */,
      MPI_Datatype   msg_type       /* in */,
      int            dest           /* in */,
      int            tag            /* in */,
      MPI_Comm       communicator   /* in */);
```



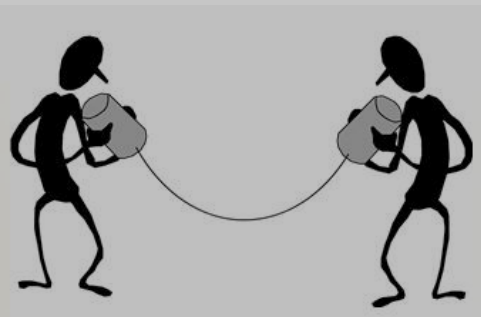
DATA TYPES

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

COMMUNICATION

```
int MPI_Recv(
      void*           msg_buf_p     /* out */,
      int             buf_size      /* in  */,
      MPI_Datatype    buf_type      /* in  */,
      int             source        /* in  */,
      int             tag           /* in  */,

 MPI_Comm        communicator  /* in  */,
 MPI_Status*     status_p      /* out */);
```



Message Matching

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

r

MPI_SEND

SRC = Q

MPI_RECV

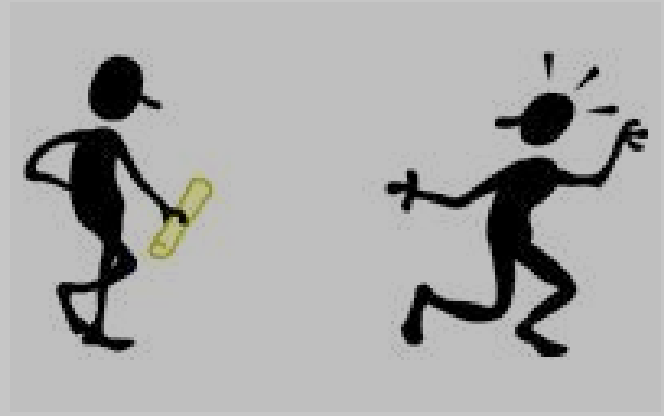
DEST = R

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

q

RECEIVING MESSAGES

- A receiver can get a message without knowing:
  - the amount of data in the message,
  - the sender of the message,
  - or the tag of the message.



END OF WEEK 2