# CME 194 Introduction to MPI

Disce Fundamenta — The Basics

http://cme194.stanford.edu

# CME 194 Introduction to MPI

This course is about writing programs for computer clusters

## Disce Fundamenta — The Basics

This course answers two questions

- Why do we want to write parallel programs?
- How do we write parallel programs?

# Why do we need parallel computers?

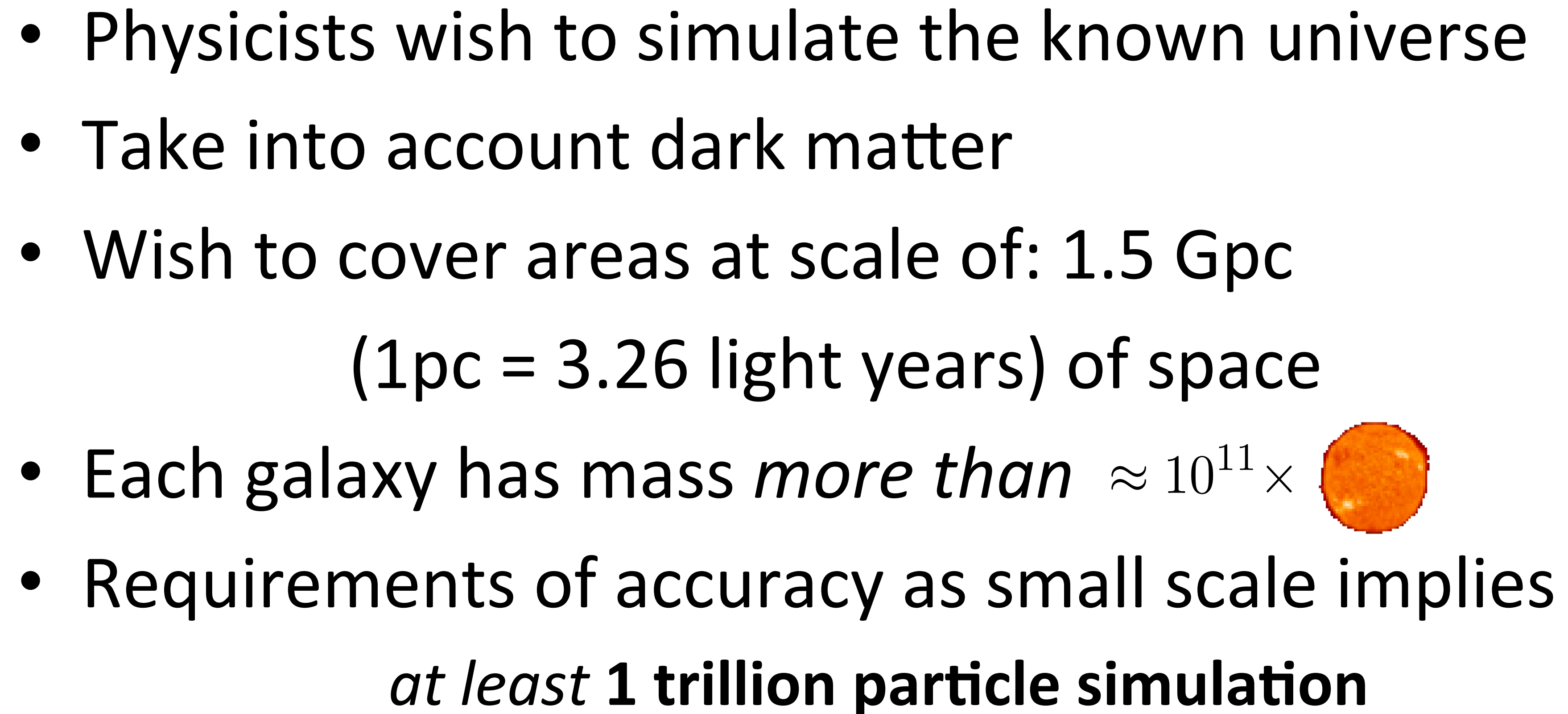
## Motivating Example: N-body Problem

**Given: -** A group of N points in space
**-** The laws of physics

**Goal:** Predict the motion of a group of N-bodies interacting gravitationally.

At the heart of all **direct** algorithms is computing $O(n^2)$ particle-particle forces

# N-body problem



- Physicists wish to simulate the known universe
- Take into account dark matter
- Wish to cover areas at scale of: 1.5 Gpc

  (1pc = 3.26 light years) of space
- Each galaxy has mass *more than* $\approx 10^{11} \times$
- Requirements of accuracy as small scale implies

  *at least* **1 trillion particle simulation**

# Why do we need parallel computers?

## Motivating Example: N-body Problem

- Imagine we only needed:
  - 10,000 floating point operation per point per time step

    $\Longrightarrow$ about 115 years per time step on 1Ghz CPU

    $\Longrightarrow$ under 3 hours with a 1Thz CPU

Note: Direct methods require $O(n)$ or $O(10^{12})$ operations per point.

Real codes are indirect, so our assumption may not be too far off

Why not build a CPU with a clock speed of

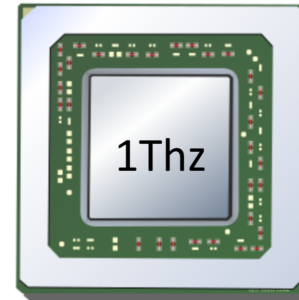
1 trillion operations / second?

# A 1Thz CPU Thought Experiment

**Execute this code:**

```
double x[ ONE_TRILLION ];
double y[ ONE_TRILLION ];
double z[ ONE_TRILLION ];
for ( size_t i = 0; i < ONE_TRILLION; ++i){
     z[i] = x[i] + y[i];
}
```

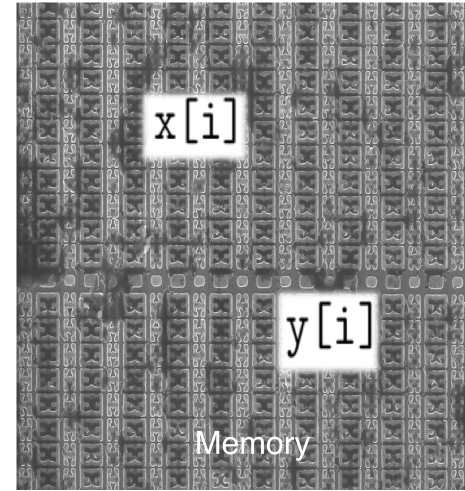
**On this CPU:**

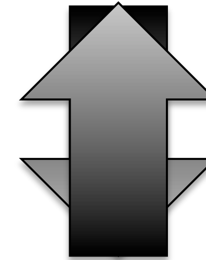


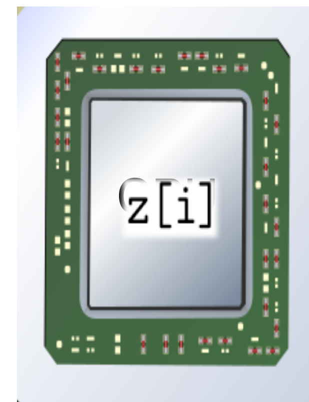

**In one second.**

**Execute this code:**

```
double x[ ONE_TRILLION ];
double y[ ONE_TRILLION ];
double z[ ONE_TRILLION ];
for ( size_t i = 0; i < ONE_TRILLION; ++i){
     z[i] = x[i] + y[i];
}
```

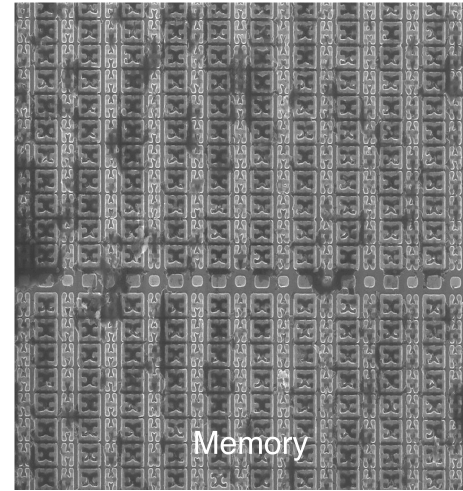


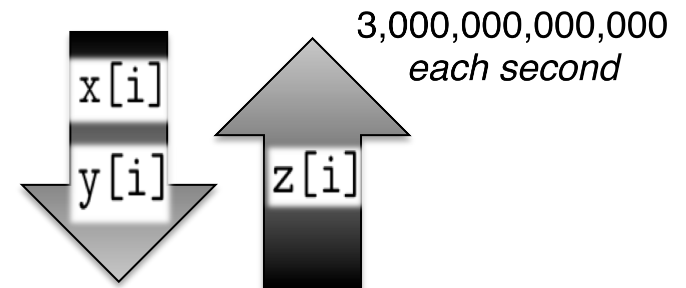



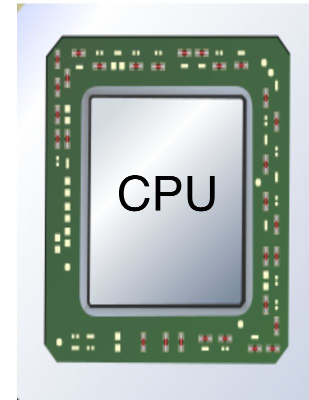


**Execute this code:**

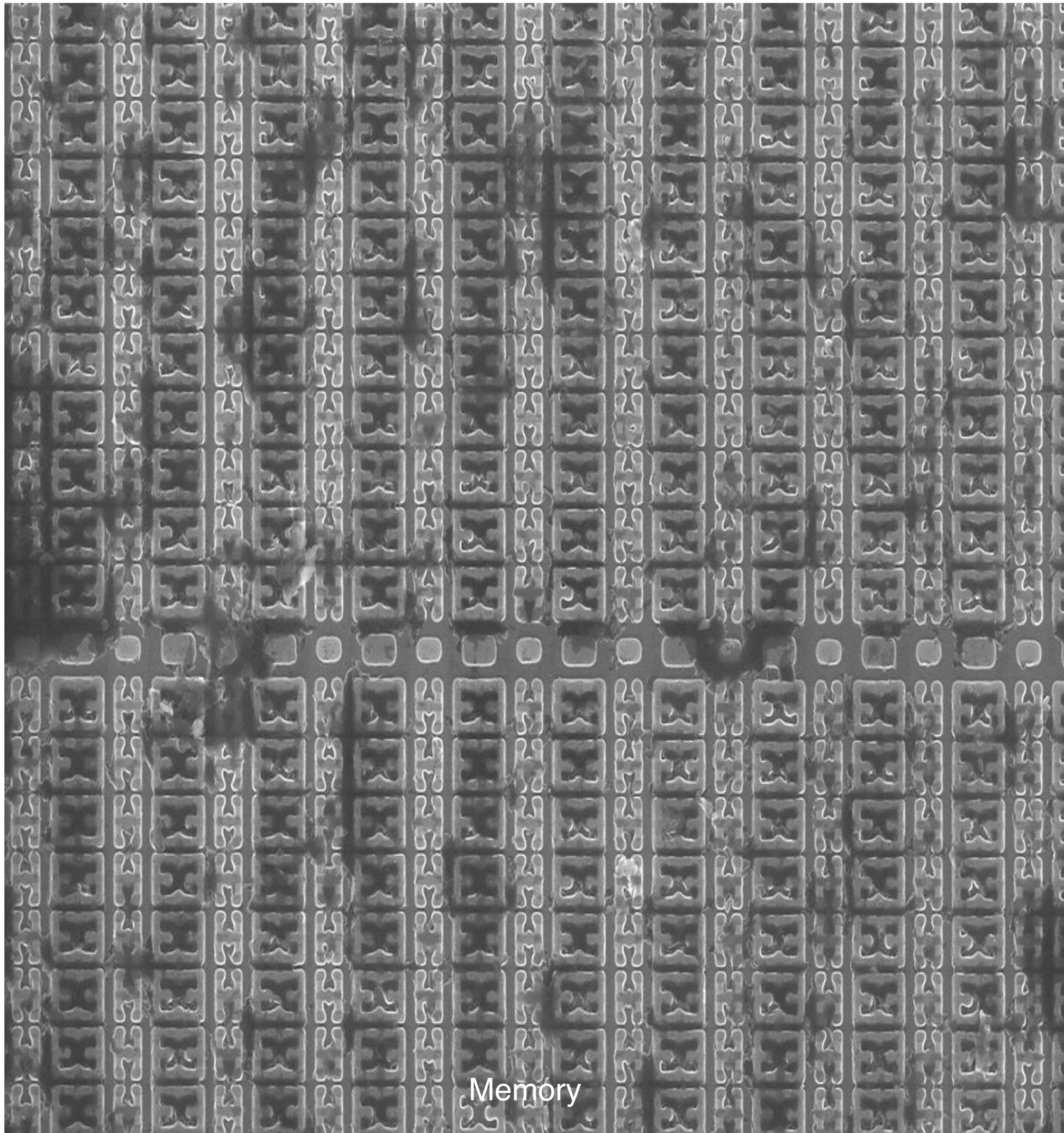
```
double x[ ONE_TRILLION ];
double y[ ONE_TRILLION ];
double z[ ONE_TRILLION ];
for ( size_t i = 0; i < ONE_TRILLION; ++i){
     z[i] = x[i] + y[i];
}
```



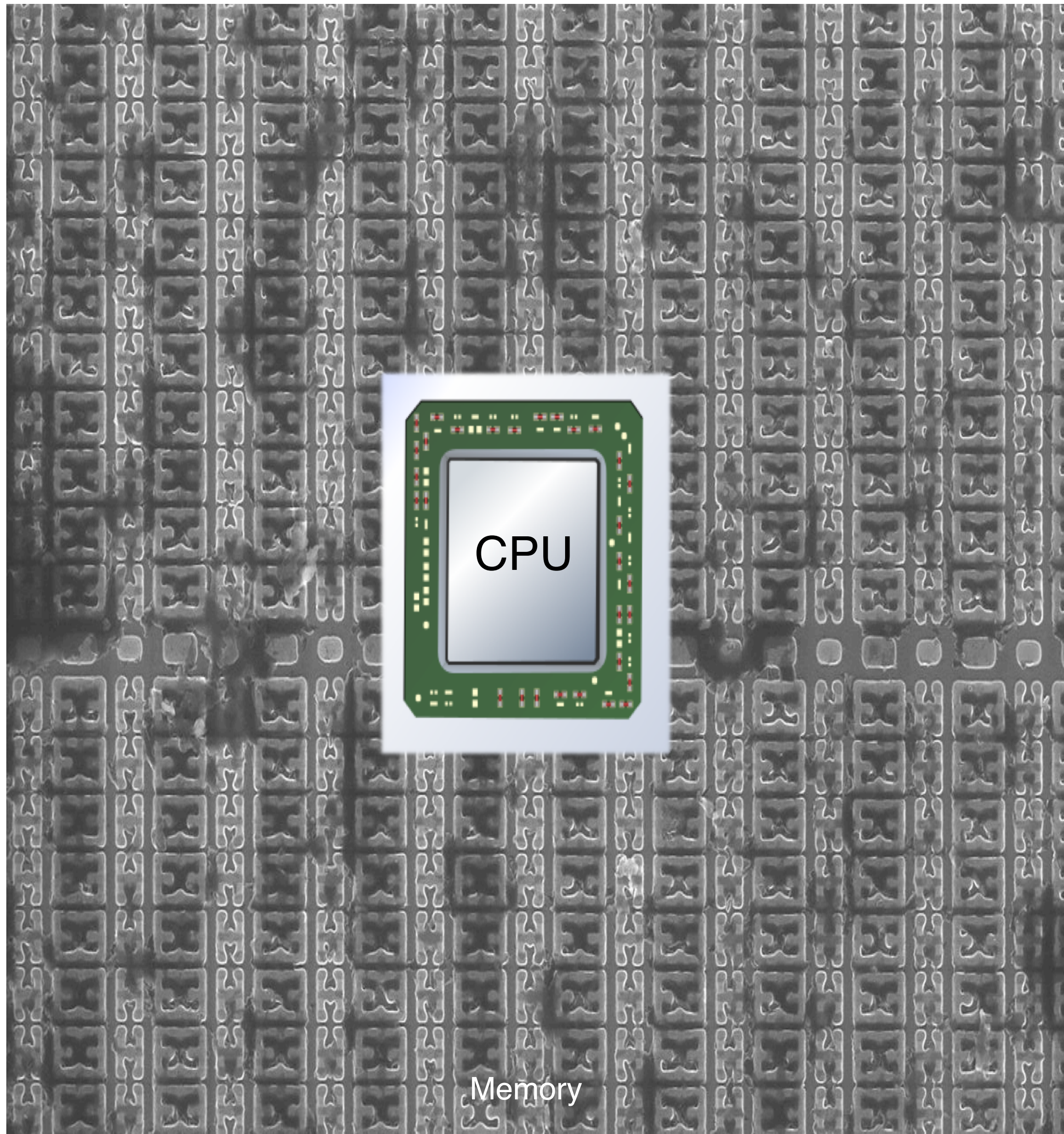



3,000,000,000,000 words

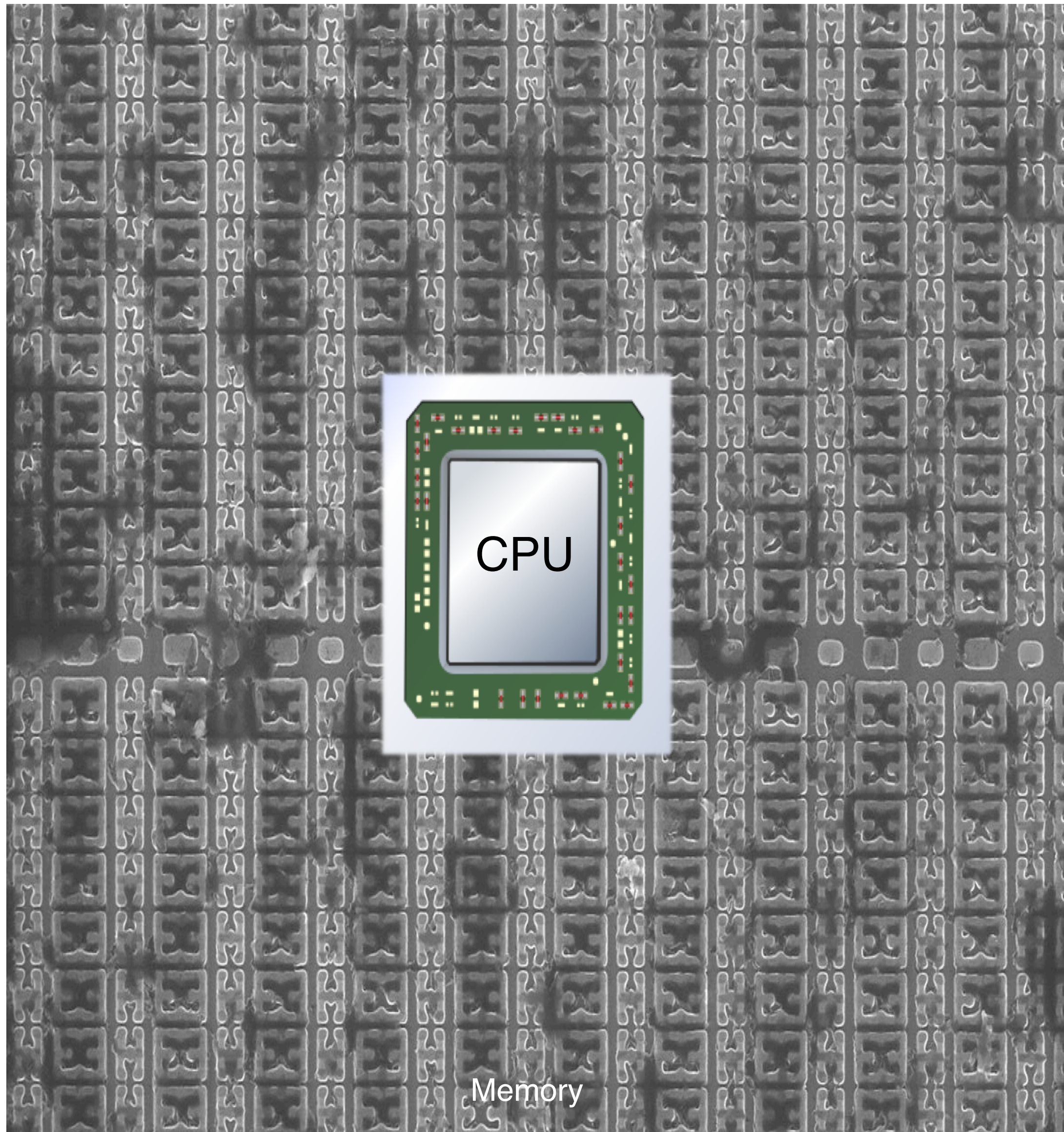


3,000,000,000,000 words




side length $s$

3,000,000,000,000 words

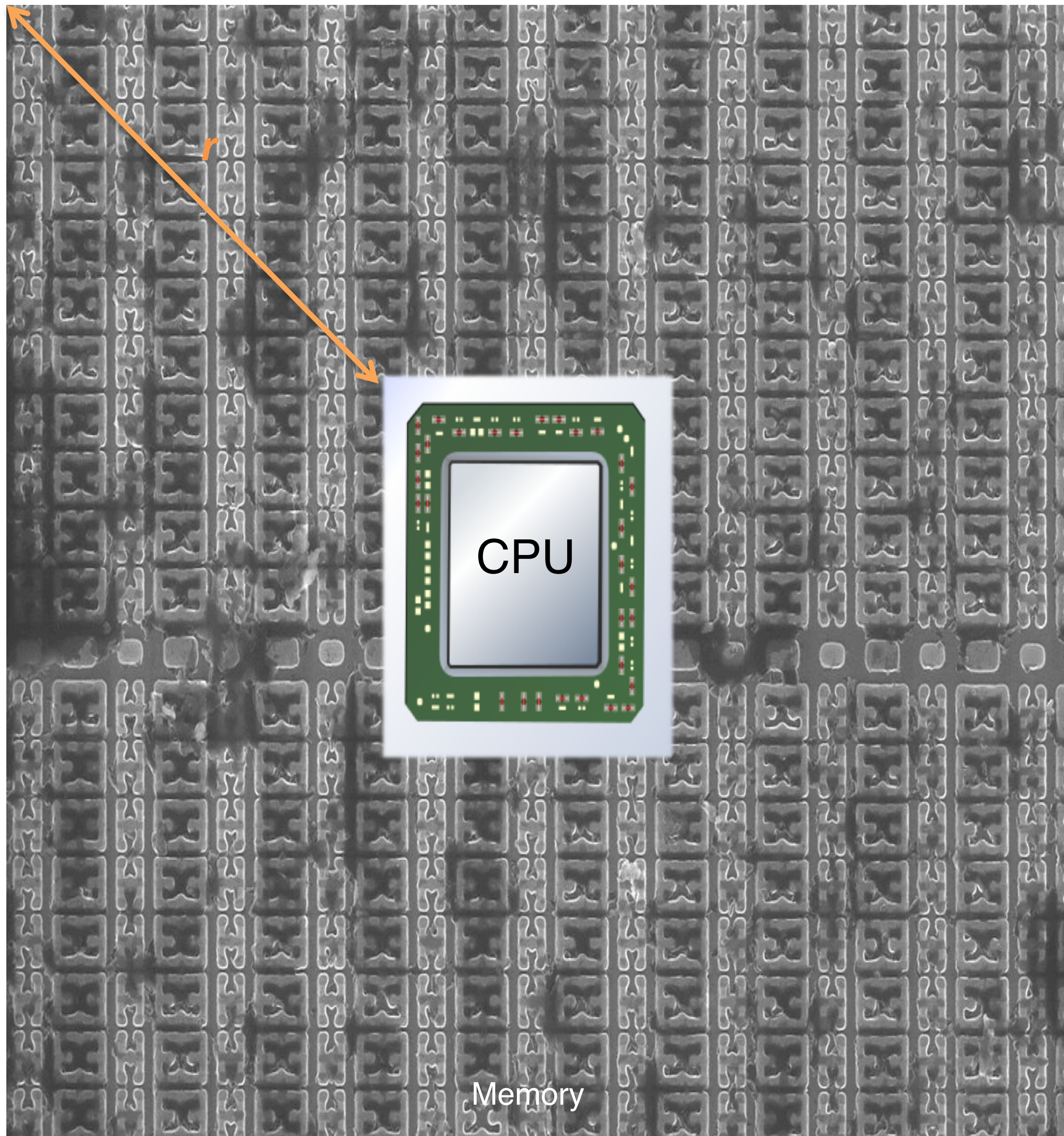



side length $s$

Words travel at

$$3 \cdot 10^8 \text{ meters/second}$$

3,000,000,000,000 words



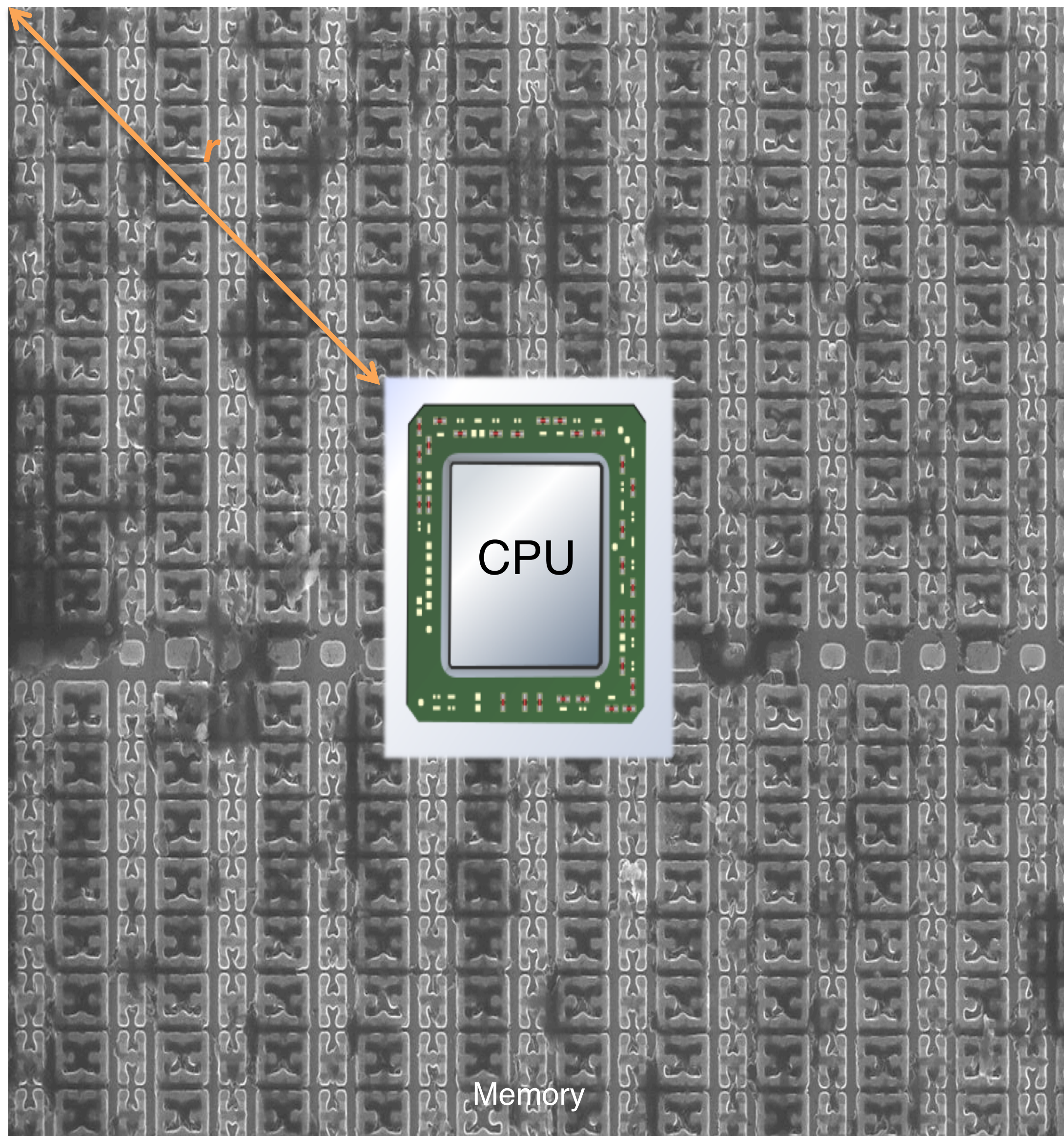

side length $s$

Words travel at

$$3 \cdot 10^8 \text{ meters/second}$$

$$(3 \cdot 10^{12}) \cdot r \text{ meters}$$
$$= 3 \cdot 10^8 \text{ meters/sec } \times 1 \text{ sec}$$

3,000,000,000,000 words




side length $s$

Words travel at

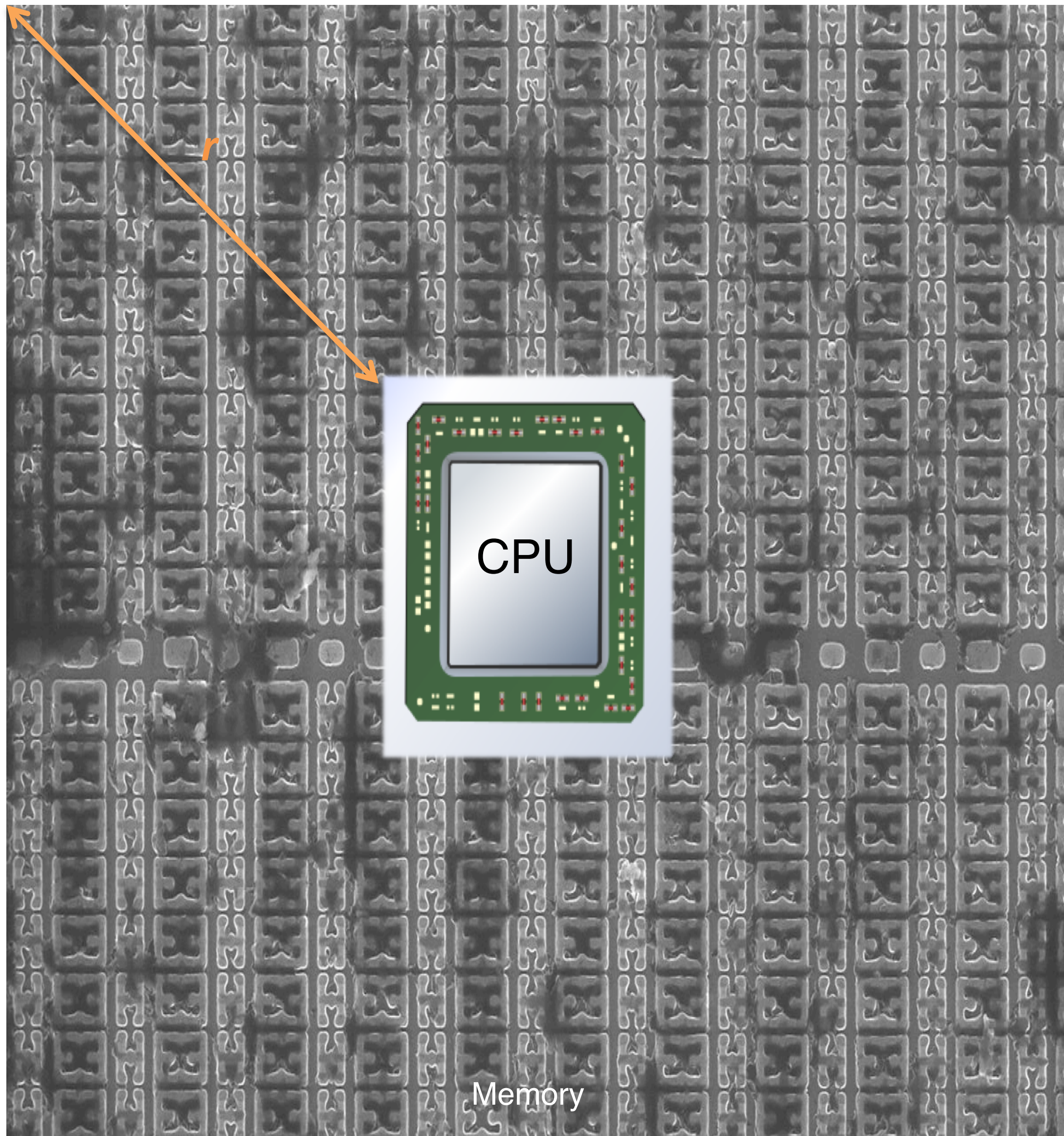
$$3 \cdot 10^8 \text{ meters/second}$$

$$(3 \cdot 10^{12}) \cdot r \text{ meters}$$

$$= 3 \cdot 10^8 \text{ meters/sec} \times 1 \text{ sec}$$

$$\implies r \approx 10^{-4} \text{ meters}$$

3,000,000,000,000 words




side length $s$

Words travel at

$$3 \cdot 10^8 \text{ meters/second}$$

$$(3 \cdot 10^{12}) \cdot r \text{ meters}$$

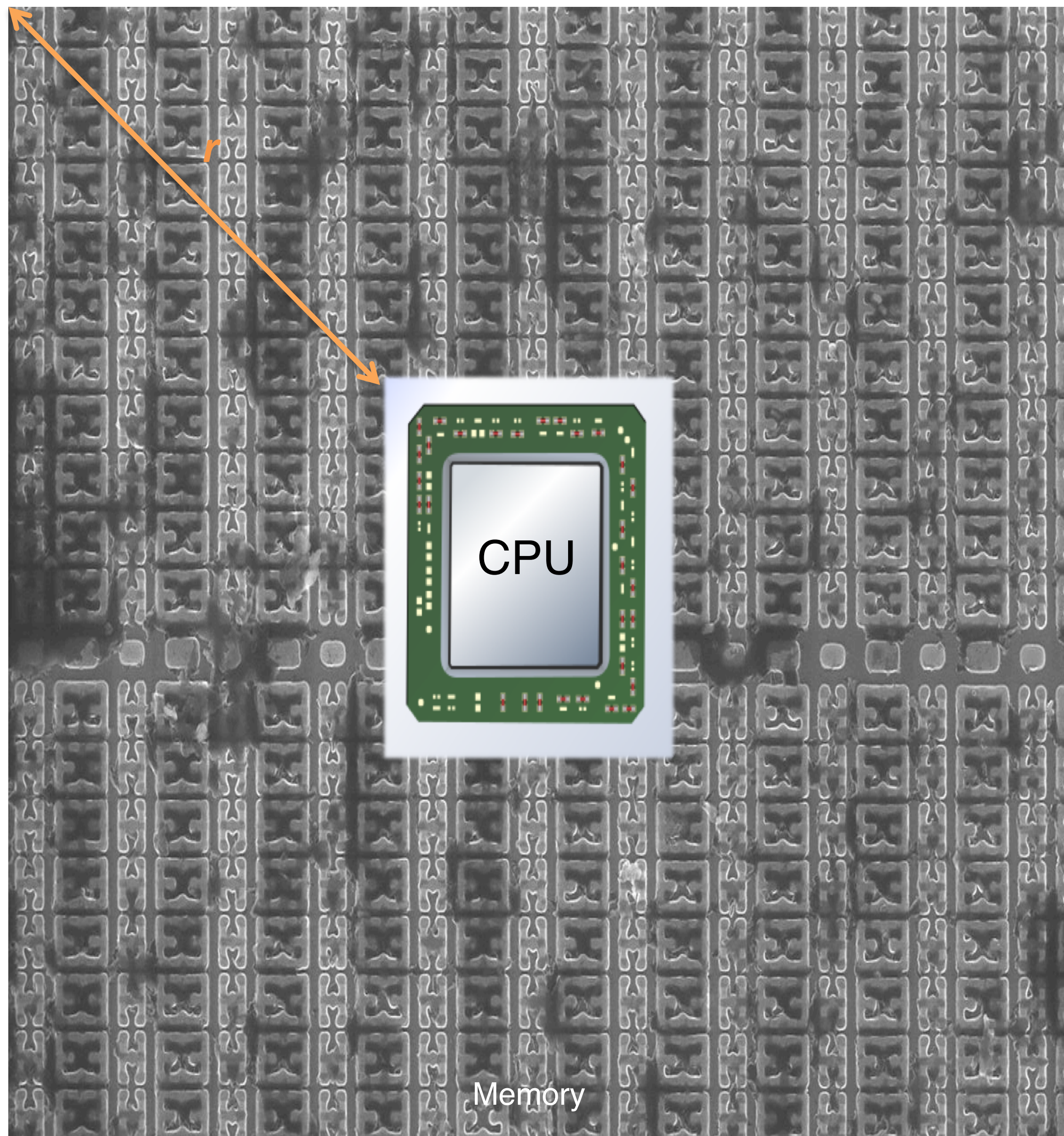
$$= 3 \cdot 10^8 \text{ meters/sec } \times 1 \text{ sec}$$

$$\implies r \approx 10^{-4} \text{ meters}$$

$$s = \frac{r}{2}$$

$$\implies s = 2 \times 10^{-4} \text{ meters}$$

3,000,000,000,000 words




side length $s$

Words travel at

$$3 \cdot 10^8 \text{ meters/second}$$

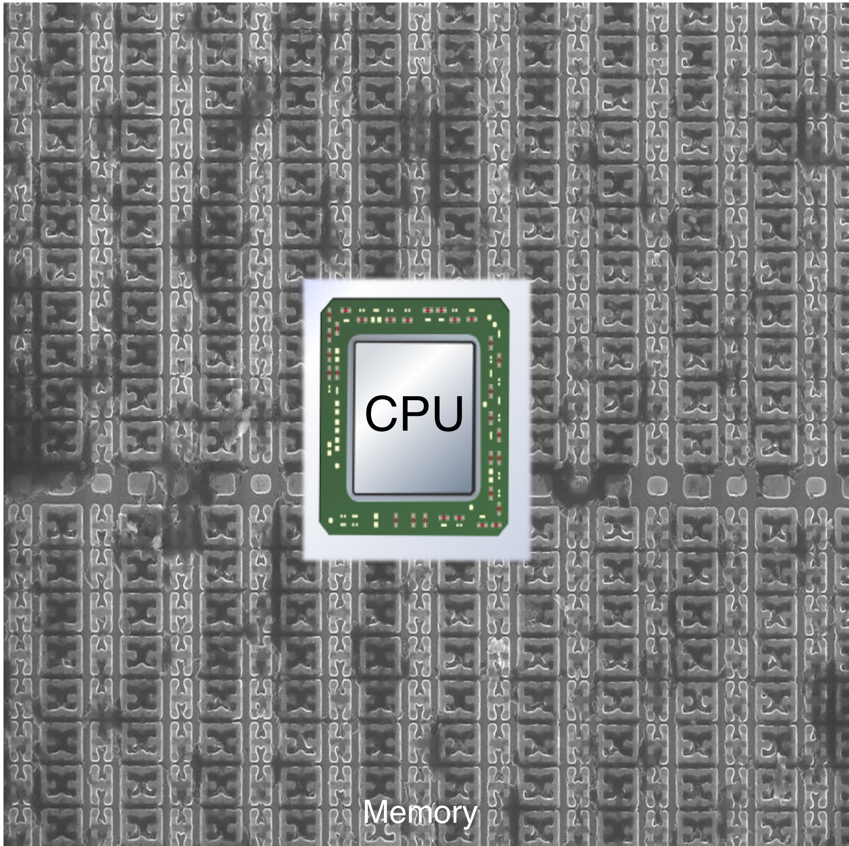
$$(3 \cdot 10^{12}) \cdot r \text{ meters}$$

$$= 3 \cdot 10^8 \text{ meters/sec } \times 1 \text{ sec}$$

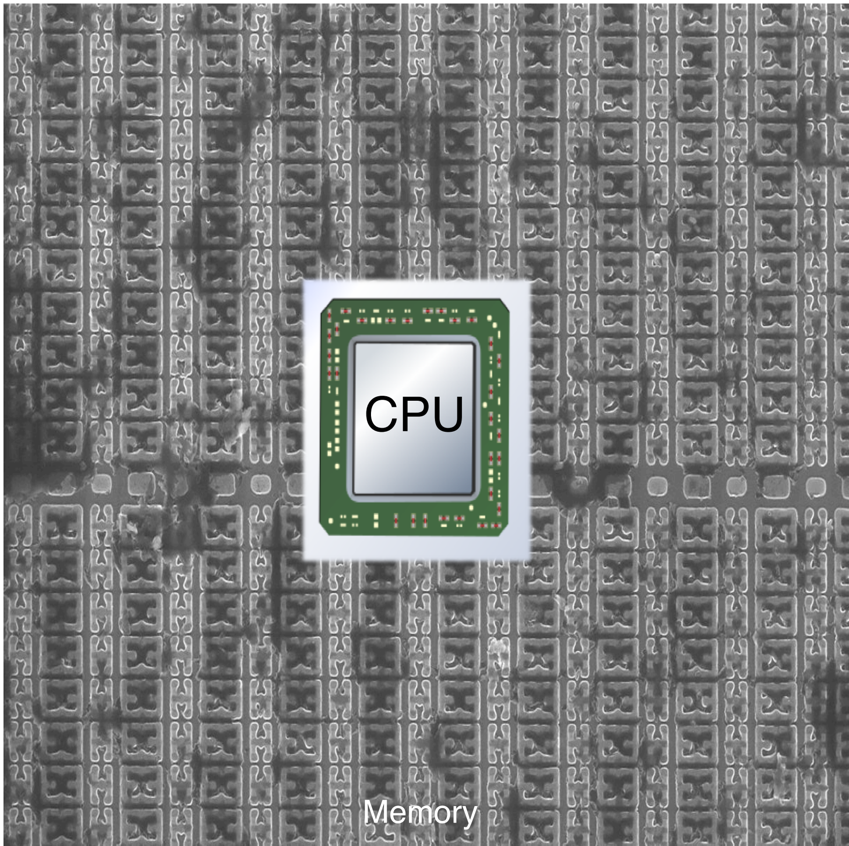
$$\implies r \approx 10^{-4} \text{ meters}$$

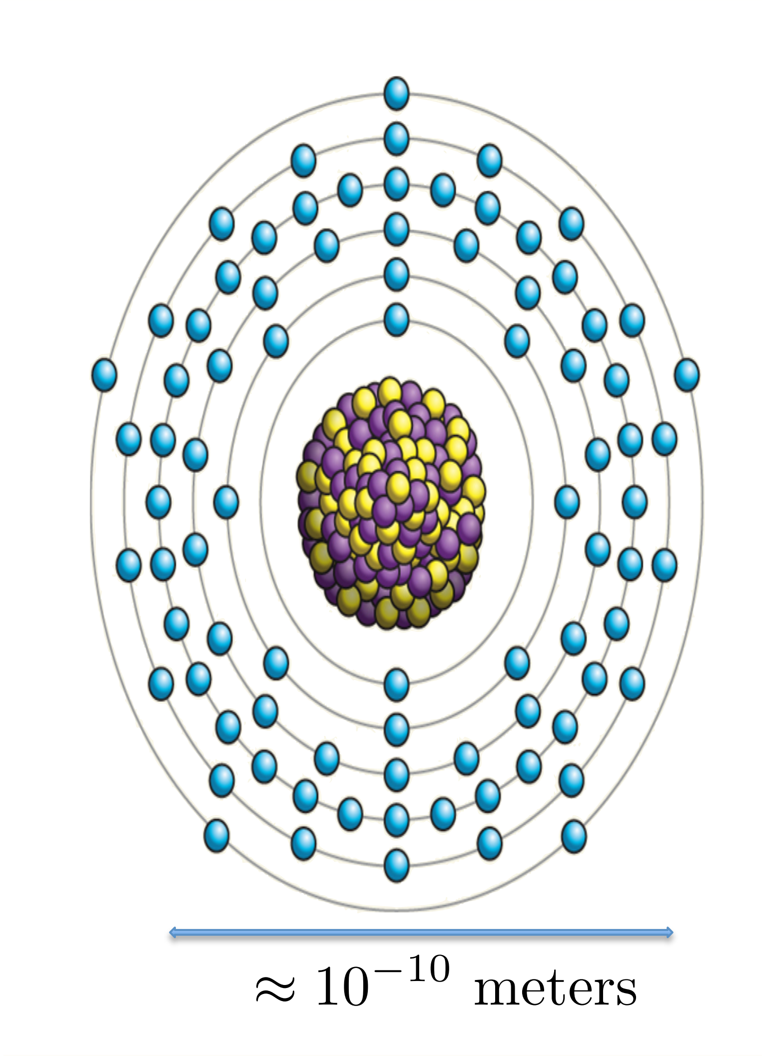
$$s = \frac{r}{2}$$

$$\implies s = 2 \times 10^{-4} \text{ meters}$$




2,000,000 words per row


CPU
Memory


$\approx 10^{-10}$ meters

# Why do we need parallel computers?

## Problem:

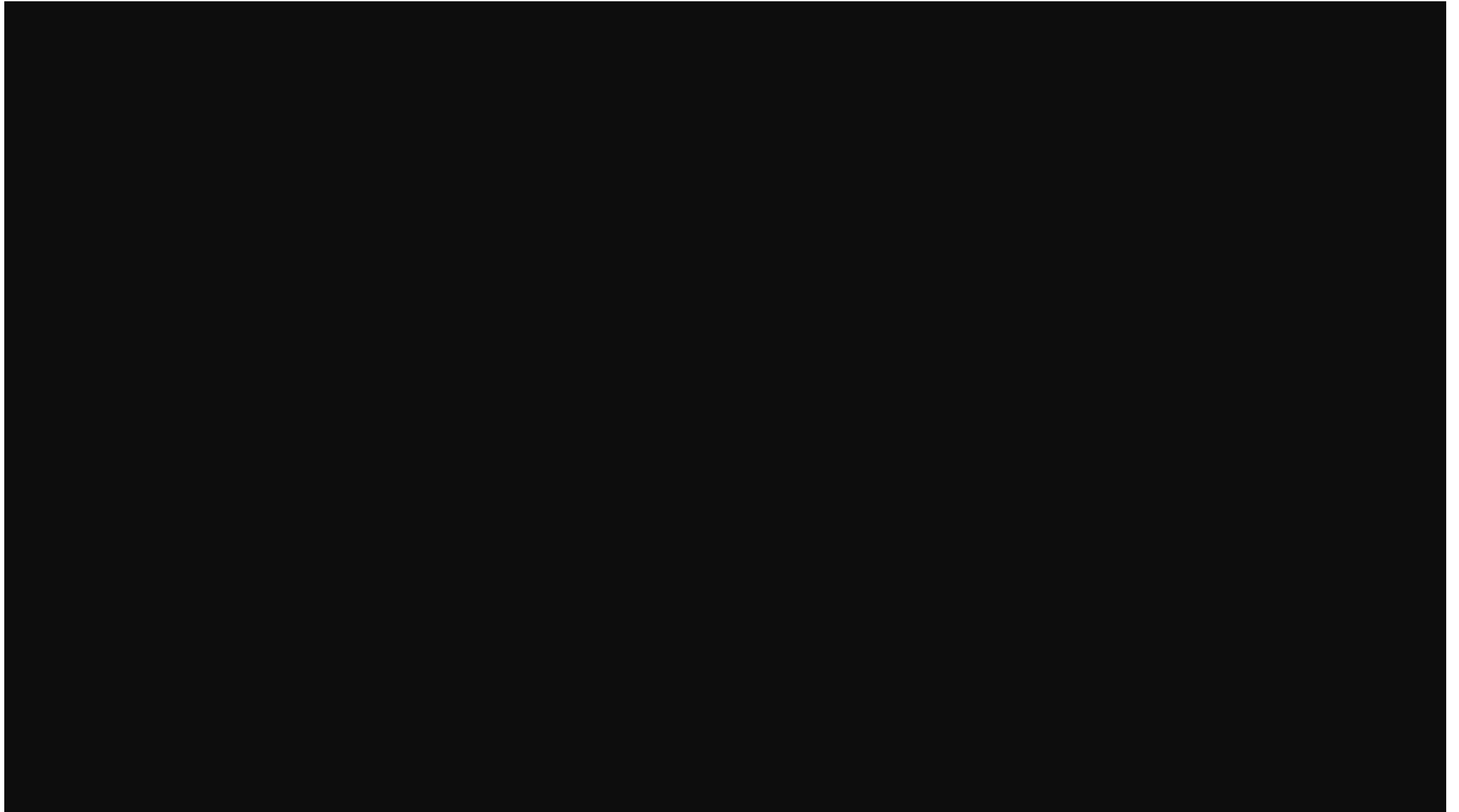
How do we get the necessary computational power?

## Solution:

1. Be clever! $\Longrightarrow$ Use Algorithms with lower complexity

2. Put many computers together $\Longrightarrow$ Solve problems faster

# Actual $10^6$–body simulation

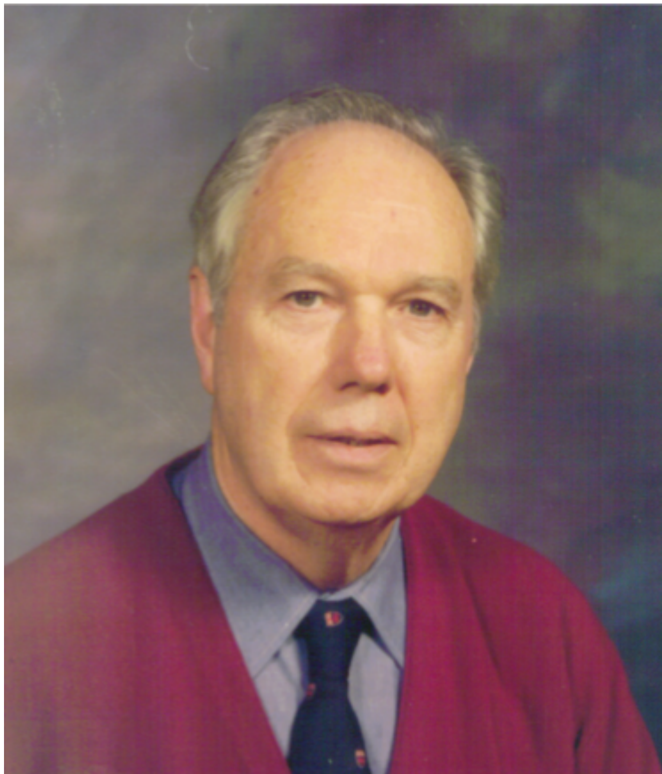


# How do we use parallel computers?

**Preliminary Question:** What exactly is a parallel computer?

**Answer:** Many *different* [parallel] computing architectures.

# Different Parallel Architectures



Dr. Michael Flynn
Stanford University



IEEE TRANSACTIONS ON COMPUTERS, VOL. C-21, NO. 9, SEPTEMBER 1972

## Some Computer Organizations and Their Effectiveness

MICHAEL J. FLYNN, MEMBER, IEEE

*Abstract*—A hierarchical model of computer organizations is developed, based on a tree model using request/service type resources as nodes. Two aspects of the model are distinguished: logical and physical.

General parallel- or multiple-stream organizations are examined as to type and effectiveness—especially regarding intrinsic logical difficulties.

The overlapped simplex processor (SISD) is limited by data dependencies. Branching has a particularly degenerative effect.

The parallel processors [single-instruction stream-multiple-data stream (SIMD)] are analyzed. In particular, a nesting type explanation is offered for Minsky's conjecture—the performance of a parallel processor increases as log $M$ instead of $M$ (the number of data stream processors).

Multiprocessors (MIMD) are subjected to a saturation syndrome based on general communications lockout. Simplified queuing models indicate that saturation develops when the fraction of task time spent locked out ($L/E$) approaches $1/n$, where $n$ is the number of processors. Resources sharing in multiprocessors can be used to avoid several other classic organizational problems.

*Index Terms*—Computer organization, instruction stream, overlapped, parallel processors, resource hierarchy.

### Introduction

ATTEMPTS to codify the structure of a computer have generally been from one of three points of view: 1) automata theoretic or microscopic; 2) individual problem oriented; or 3) global or statistical.

In the microscopic view of computer structure, relationships are described exhaustively. All possible interactions and parameters are considered without respect to their relative importance in a problem environment.

Measurements made by using individual problem yardsticks compare organizations on the basis of their relative performances in a peculiar environment. Such comparisons are usually limited because of their ad hoc more "macroscopic" view, yet without reference to a particular user environment. Clearly, any such effort must be sharply limited in many aspects; some of the more significant are as follows.

1) There is no treatment of I/O problems or I/O as a limiting resource. We assume that all programs of interest will either not be limited by I/O, or the I/O limitations will apply equally to all computer memory configurations. That is, the I/O device sees a "black box" computer with a certain performance. We shall be concerned with *how* the computer attained a performance potential, while it may never be realized due to I/O considerations.

2) We make no assessment of particular instruction sets. It is assumed that there exists a (more or less) ideal set of instructions with a basically uniform execution time—except for data conditional branch instructions whose effects will be discussed.

3) We will emphasize the notion of effectiveness (or efficiency) in the use of internal resources as a criterion for comparing organizations, despite the fact that either condition 1) or 2) may dominate a total performance assessment.

Within these limitations, we will first attempt to classify the forms or gross structures of computer systems by observing the possible interaction patterns between instructions and data. Then we will examine physical and logical attributes that seem fundamental to achieving efficient use of internal resources (execution facilities, memory, etc.) of the system.

### Classification: Forms of Computing Systems

*Gross Structures*

In order to describe a machine structure from a

Published September 1972

# Different Parallel Architectures

There is an alphabet soup of different computer architectures:

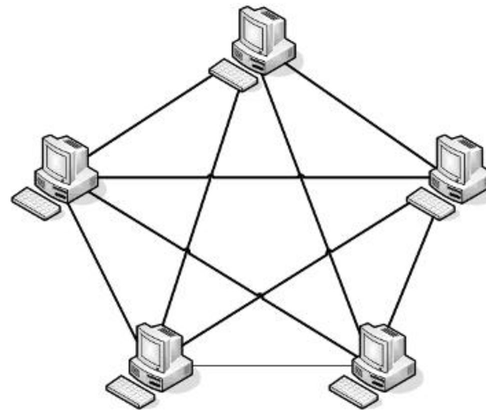
- SISD (Single instruction stream, single data stream)
- SIMD (Single instruction stream, multiple data streams)
- MISD (Multiple instruction streams, single data streams)
- MIMD (Multiple instruction streams, multiple data streams)

And each of them allow for some type of parallel computing

# Different Parallel Architectures

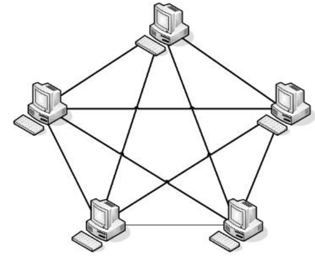
**This course:** MIMD Architecture

- A collection of machines each with their own memory
- Machines work **autonomously**
- Machines **communicate** by sending messages over a network
- We will assume this network is fully connected



Use MPI to orchestrate the communication between machines

# What is MPI?



- Stands for Message Passing Interface
- A **standard** for communication via **message passing**
- There are common implementations of this standard:
  - OpenMPI
  - MPICH

Ok, but how do MPI programs work?

# What is MPI?

- MPI Processes **initialize** and **finalize** in the same way
- Join a global **communicator**
- Each communicator assigns a unique process **rank**
- Communicators have a **size** = # of processes

How many MPI functions are there?

# What is MPI?

- 6 + 1
- 128+
  - 52 Point-to-Point Communication
  - 16 Collective Communication
  - 30 Groups, Contexts, and Communicators
  - 16 Process Topologies
  - 13 Environmental Inquiry
  - 1 Profiling

- MPI_Init(...)
  - Start MPI
- MPI_Comm_size(...)
  - Number of MPI processes
- MPI_Comm_rank(...)
  - Internal process number
- MPI_Get_processor_name(...)
  - External processor name
- MPI_Finalize(...)
  - Stop MPI

Lets look at some code

```
1 #include <iostream> //std::cout
2 #include "mpi.h" //MPI_*
3
4 int main(int argc, char* argv[]){
5   //MPI_Init handles the plumbing to make this process connect to the "network"
6   //This must be the first MPI_ function called
7   //And this function may only be called once
8   if( MPI_Init( &argc, &argv) != MPI_SUCCESS){
9         std::cerr << "MPI Failed to Initialize!" << std::endl;
10        return 1;
11  }
12
13  int rank=0,size=0;
14
15  //Get this processors ID within the communicator
16  MPI_Comm_rank( MPI_COMM_WORLD, &rank);
17
18  //Get the number of processes in the communicator
19  MPI_Comm_size( MPI_COMM_WORLD, &size);
20
21  std::cout << "Hello, World! I am process "
22            << rank << " out of " << size << std::endl;
23
24  MPI_Finalize();
25  return 0;
26 }
```

# Compiling and Running

- **Compiling & Linking:**

  $ mpicc hello_world.cc –o hello_world

  $ mpic++ hello_world.cpp –o hello_world

- **Running:**

  $ mpirun [ -np X ] [ --hostfile <filename> ] <program>

```
$ mpirun -np 3 hello_world
Hello, World! I am process 0 out of 3
Hello, World! I am process 1 out of 3
Hello, World! I am process 2 out of 3
$ mpirun -np 3 hello_world
Hello, World! I am process 1 out of 3
Hello, World! I am process 0 out of 3
Hello, World! I am process 2 out of 3
$ mpirun -np 3 hello_world
Hello, World! I am process 2 out of 3
Hello, World! I am process 0 out of 3
Hello, World! I am process 1 out of 3
$ mpirun -np 3 hello_world
Hello, World! I am process 0 out of 3
Hello, World! I am process 1 out of 3
Hello, World! I am process 2 out of 3
$
```

# What is MPI?

**Common Faux Pas**

- MPI is the standard
  - The library is the implementation
- There is no MPI Compiler.

**Source of confusion**

- All the libraries look the same
- Compile "script" mpicc and mpic++

```
$ mpic++ --showme:compile
$ mpic++ –showme:link
```

# Better example

- We may now achieve ideal parallelism!
- Solve problems with no communication

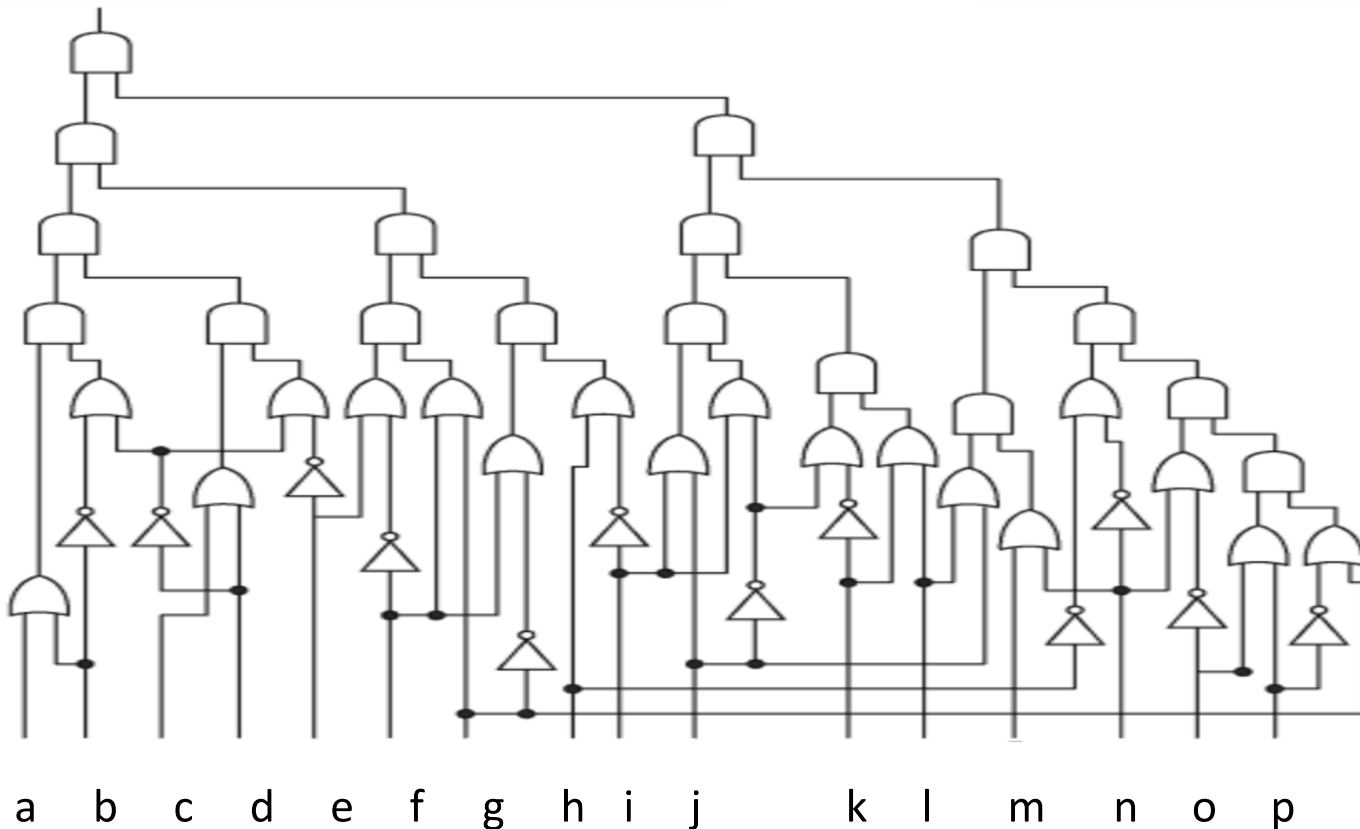
**Circuit Satisfiability**

# Circuit Satisfiability

a b c d e f g h i j k l m n o p

# Circuit Satisfiability

- **Goal:** Use MPI to find a satisfying solution to the circuit diagram shown before in parallel



# The Cluster

- Clusters are almost always shared resources
- Uses queues to manage them
- Important commands:

```
$  msub <submit.script>
$  showq
$  canceljob <job id>
```

# Machines

Use SSH

ICME MPI Cluster: icme-mpi1.stanford.edu
ICME Shared Memory Machine: icme-sharedmem.stanford.edu
Stanford general computers: corn.stanford.edu (Barley Cluster)

Accounts exist on the ICME systems.