

# **TDT 4200 Fall 2024**

## **Parallel Computing**

### **Lect 9: MPI\_Scatter/Gather – time analysis/Performance, More on Caching**

**Prof. Anne C. Elster, PhD**

**Dept. of Computer Science (IDI)**

**Norwegian Univ. of Science & Technology**

**(NTNU Trondheim, Norway)**

**&**

**Univ. of Texas at Austin (Senior Visiting Scientist)**

Anne C. Elster NTNU  
TDT4200, Sept 4, 2024


# TDT4200 F2024 Course Information

– See also <https://www.ntnu.edu/studies/courses/TDT4200/>

- Instructor: Professor **Anne C. Elster** ([elster@ntnu.no](mailto:elster@ntnu.no))
- Recitation lectures and assignments: **Tobias Dyngeland**
- Course supporter: Assoc. Prof. **Jan Christian Meyer**
- Web page: <http://www.idi.ntnu.no/~elster/tdt4200/fall2024> **TBA**
- **Lectures:**
  - Tuesdays: 10:15-12:00 in R5
  - Wednesdays: 9:15-10:00 in R7
- **Recitation lectures (Øvingstime):**
  - Tuesdays: 16:15-17:00 in Kjel 5
  - ~~Wednesdays: 8:15-9:00 in Kjel 1~~ – **replaced by Cybele hrs on Mondays:** (Updated under “Detailed TimeTable”)
- **Lab help Mondays 16-18 @ Cybele**
- **Course Tool: BlackBoard**

# TDT4200 F2024 Course Information

– See also <https://www.ntnu.edu/studies/courses/TDT4200/>


[Studies](#)
[Research and innovation](#)
[Life and housing](#)
[About NTNU](#)

[About](#)
[Timetable](#)
[Examination](#)

Autumn 2024/ Spring 2025

List view  
[Detailed timetable](#)
[ical](#)

Studieprogram [ALLE](#) Filter

Day [Week](#) Month All

Sep 2 – 6, 2024

< today >

## Week 36

	Monday 2/9	Tuesday 3/9	Wednesday 4/9
8			
9			
10			1Forelesning
11		1Forelesning Lecture 10:15 - 12:00 A.C. Elster	
12			
13			
14			
15			
16	3Øving Øving i datasal Cybele 16:15 - 18:00	3Øving	
17			

# Course Information – continued

See also <https://www.ntnu.edu/studies/courses/TDT4200/>

NOTE: Compulsory assignments:

- You need to do and **pass ALL Problem Sets/Exercises** in order to take the final  
... **also those that are Pass/Fail!!**



# BlackBoard – Problem Sets -- Updated dates!



## Problem Sets -- Tentative Dates 🗓️ ⚡

Tentative schedule for the problem sets :

Problem set (Exercise)	Available (Tuesdays in Recitation)	Due Mondays 10pm	Topic	Grading
PS 0	Aug 20	Sep 2/9*	C - intro -- pointers ++ (optional, but highly recommended)	Pass/Fail -- Optional
PS 1a	Aug 27	Mon Sep 9*	C - Wave Equation	Pass/Fail -- Required
PS 1b	Aug 29	Mon Sep 9*	MPI Intro (Optional, but highly recommended)	Pass/Fail -- Optional
PS2	Sep 10	Mon Sep 16	MPI 1D Wave eqn	Pass/Fail - Required
PS3	Sep 17	TBD	MPI 2D Wave eqn	Graded - Required
PS4	Oct 1	TBD	Pthreads/OpenMP	Pass/Fail - Required
PS5	TBD	TBD	CUDA Intro	Pass/Fail - Required
PS6	TBD	TBD	CUDA 2D Wave eqn	Graded - Required

\* You can technically submit these until the end of the add/drop period, but are **STRONGLY** advised to do them *ASAP*.



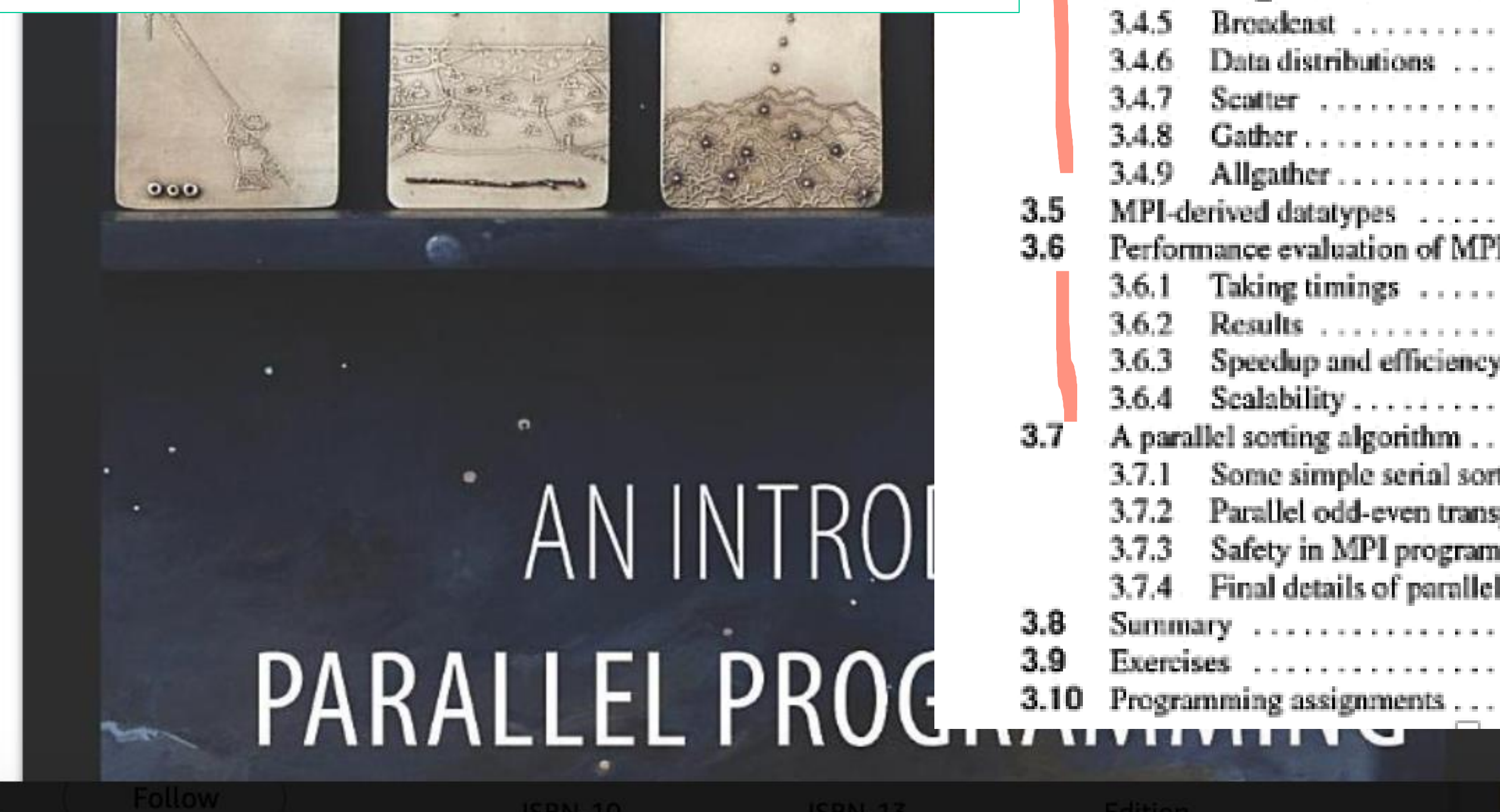
Preface .....	xv
<b>CHAPTER 1 Why parallel computing .....</b>	<b>1</b>
1.1 Why we need ever-increasing performance .....	2
1.2 Why we're building parallel systems .....	3
1.3 Why we need to write parallel programs .....	3
1.4 How do we write parallel programs? .....	6
1.5 What we'll be doing .....	8
1.6 Concurrent, parallel, distributed .....	10
1.7 The rest of the book .....	11
1.8 A word of warning .....	11
1.9 Typographical conventions .....	12
1.10 Summary .....	12
1.11 Exercises .....	14
<b>CHAPTER 2 Parallel hardware and parallel software .....</b>	<b>17</b>
2.1 Some background .....	17
2.1.1 The von Neumann architecture .....	17
2.1.2 Processes, multitasking, and threads .....	19
2.2 Modifications to the von Neumann model .....	20
2.2.1 The basics of caching .....	21
2.2.2 Cache mappings .....	23
2.2.3 Caches and programs: an example .....	24
2.2.4 Virtual memory .....	25
2.2.5 Instruction-level parallelism .....	27
2.2.6 Hardware multithreading .....	30
2.3 Parallel hardware .....	31
2.3.1 Classifications of parallel computers .....	31
2.3.2 SIMD systems .....	31
2.3.3 MIMD systems .....	34
2.3.4 Interconnection networks .....	37
2.3.5 Cache coherence .....	45
2.3.6 Shared-memory vs. distributed-memory .....	49
2.4 Parallel software .....	49
2.4.1 Caveats .....	50
2.4.2 Coordinating the processes/threads .....	50
2.4.3 Shared-memory .....	51
2.4.4 Distributed-memory .....	55
2.4.5 GPU programming .....	58
2.4.6 Programming hybrid systems .....	60



<b>CHAPTER 3</b>	<b>Distributed memory programming with MPI</b>	<b>89</b>
3.1	Getting started	90
3.1.1	Compilation and execution	91
3.1.2	MPI programs	92
3.1.3	MPI_Init and MPI_Finalize	93
3.1.4	Communicators, MPI_Comm_size, and MPI_Comm_rank	94
3.1.5	SPMD programs	94
3.1.6	Communication	95
3.1.7	MPI_Send	95
3.1.8	MPI_Recv	97
3.1.9	Message matching	97
3.1.10	The status_p argument	98
3.1.11	Semantics of MPI_Send and MPI_Recv	99
3.1.12	Some potential pitfalls	100
3.2	The trapezoidal rule in MPI	100
3.2.1	The trapezoidal rule	101
3.2.2	Parallelizing the trapezoidal rule	102
3.3	Dealing with I/O	104
3.3.1	Output	105
3.3.2	Input	107



3.4	Collective communication	108
3.4.1	Tree-structured communication	108
3.4.2	MPI_Reduce	110
3.4.3	Collective vs. point-to-point communications	112
3.4.4	MPI_Allreduce	113
3.4.5	Broadcast	113
3.4.6	Data distributions	116
3.4.7	Scatter	117
3.4.8	Gather	119
3.4.9	Allgather	121
3.5	MPI-derived datatypes	123
3.6	Performance evaluation of MPI programs	127
3.6.1	Taking timings	127
3.6.2	Results	130
3.6.3	Speedup and efficiency	133
3.6.4	Scalability	134
3.7	A parallel sorting algorithm	135
3.7.1	Some simple serial sorting algorithms	135
3.7.2	Parallel odd-even transposition sort	137
3.7.3	Safety in MPI programs	140
3.7.4	Final details of parallel odd-even sort	143
3.8	Summary	144
3.9	Exercises	148
3.10	Programming assignments	155





# BlackBoard – Forum

<div>→ Thread Actions ▾ Collect Delete</div>								
<input type="checkbox"/>		DATE ▾	THREAD	AUTHOR	STATUS	UNREAD POSTS	UNREAD REPLIES TO ME	TOTAL POSTS
<input type="checkbox"/>		9/1/24 6:54 PM	<a href="#">Problem with movie creation</a>	<b>Arthus Guy Daimez</b>	Published	0	0	3
<input type="checkbox"/>		8/28/24 1:58 PM	<a href="#">Issue Running Assignment 1 on Snotra Server</a>	<b>Simone Deidier</b>	Published	0	0	2
<input type="checkbox"/>		8/25/24 8:13 PM	<a href="#">Can i deliver assignment 0 on the 2nd of september?</a>	<b>Kristian Sørli</b>	Published	0	0	4
<input type="checkbox"/>		8/23/24 2:30 PM	<a href="#">Problem Set resubmission</a>	<b>Eivind Kløvjan</b>	Published	0	0	2
<input type="checkbox"/>		8/21/24 10:57 AM	<a href="#">Comments/Questions -- Main lectures</a>	<b>Anne Cathrine Elster</b>	Published	0	0	2
<input type="checkbox"/>		8/20/24 4:40 PM	<a href="#">Recommendations for learning C</a>	<b>Henriette Marie Eltvik</b>	Published	0	0	3
<input type="checkbox"/>		8/20/24 1:00 PM	<a href="#">Exercise submission</a>	<b>Guillaume Carraux</b>	Published	0	0	3
<input type="checkbox"/>		8/19/24 1:53 PM	<a href="#">Welcome to the TDT4200 Forum / Discussion Board for Fall 2024</a> ▾	<b>Anne Cathrine Elster</b>	Published	0	0	1
<div>→ Thread Actions ▾ Collect Delete</div>								



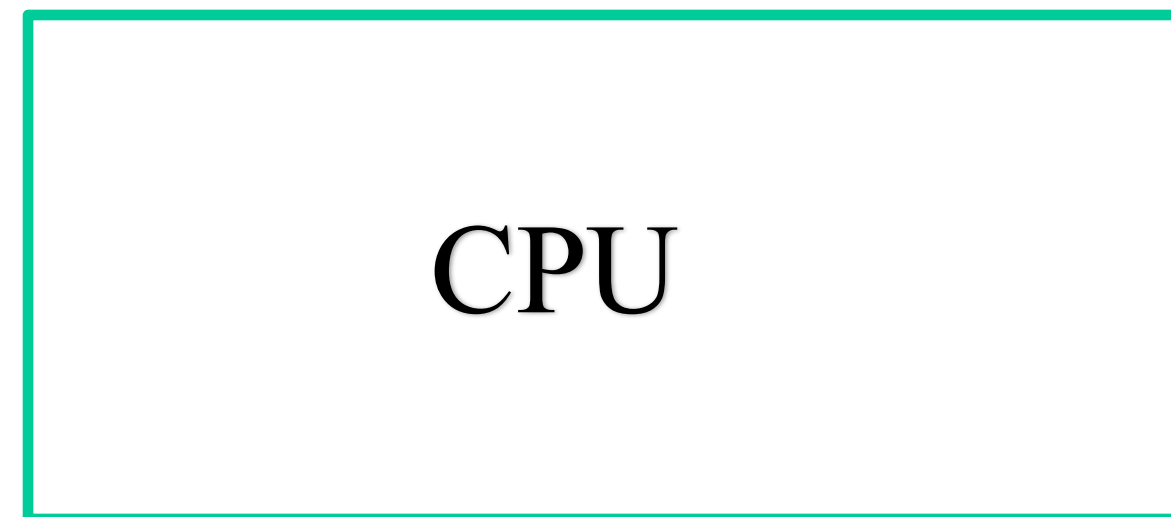


# Outline

- **Course Info.**
- **MPI Scatter/gather and timing** – on blackboard and JCM slides13, slides 14-32
  - Timing, incl. Latency & Bandwidth
  - Implementation impact on performance (serial vs trees etc)
- **More on Caching** – on black board, these slides + JCM slides24

# The Von Neuman Architecture

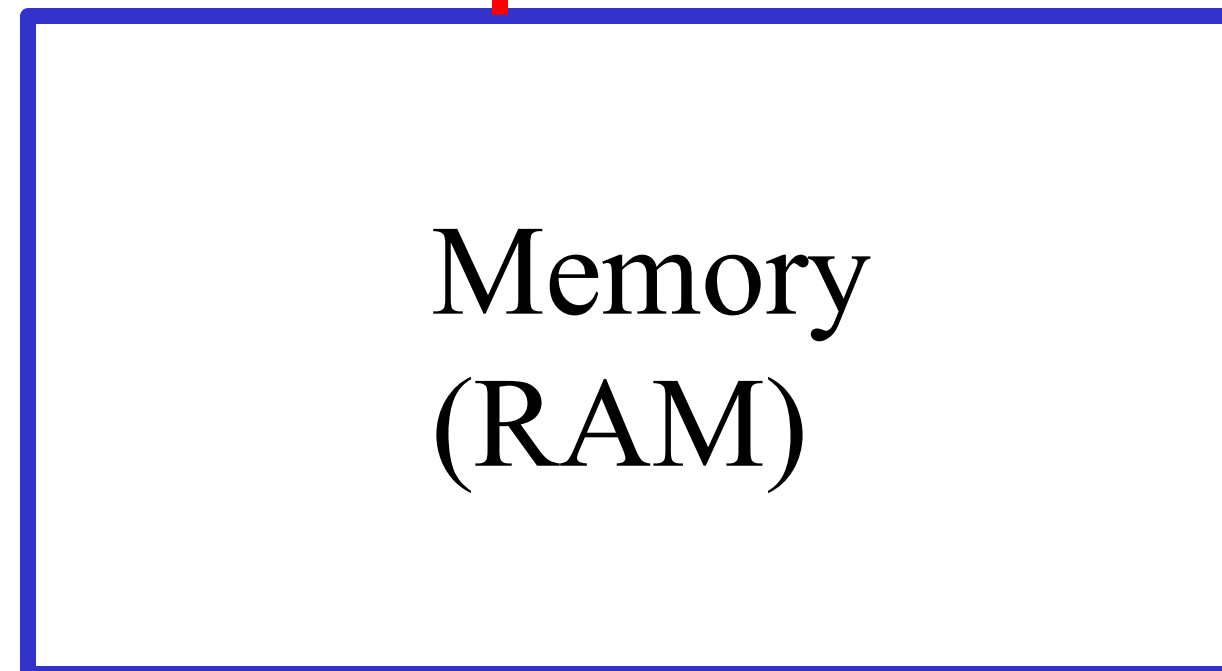
→ sequential computer (SISD)



CPU

## Von Neuman bottleneck

– since CPU 500-1000 times faster than RAM!



Memory  
(RAM)

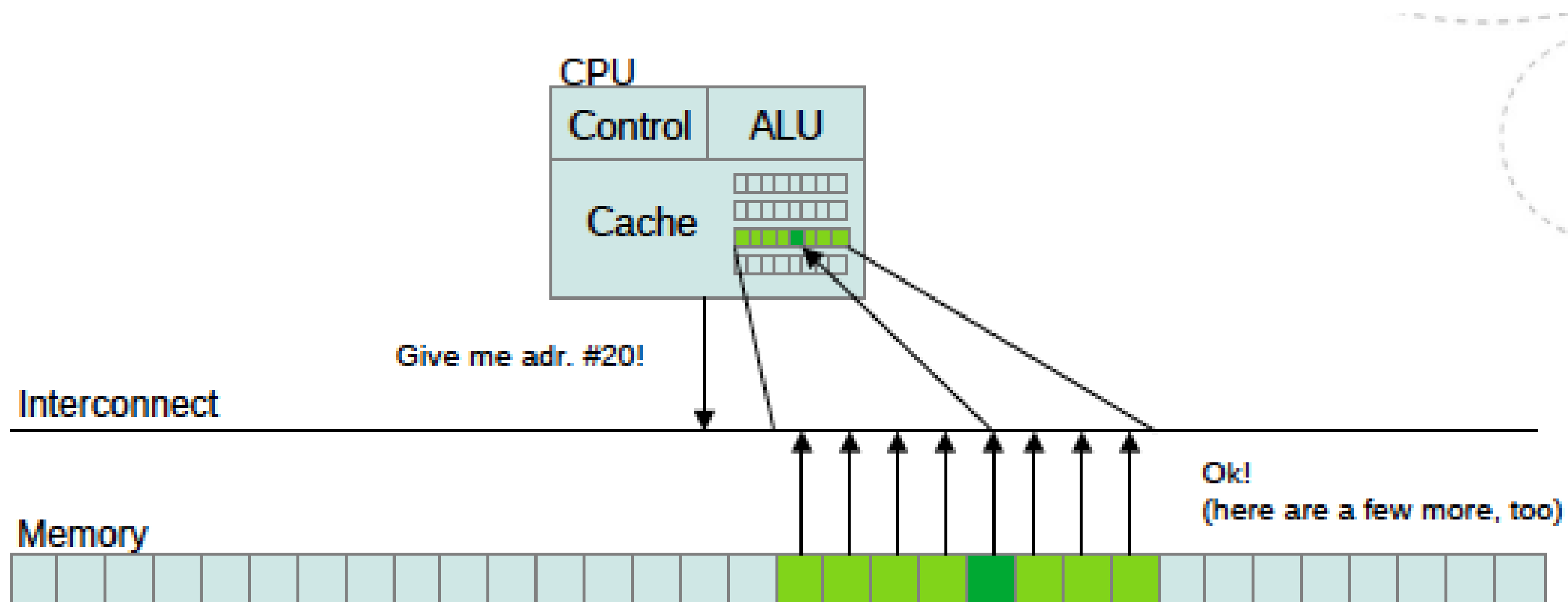
→ Add caches!

(Illustrated system with 3 cache levels on the blackboard)

# Caching

(from JCM slides26)

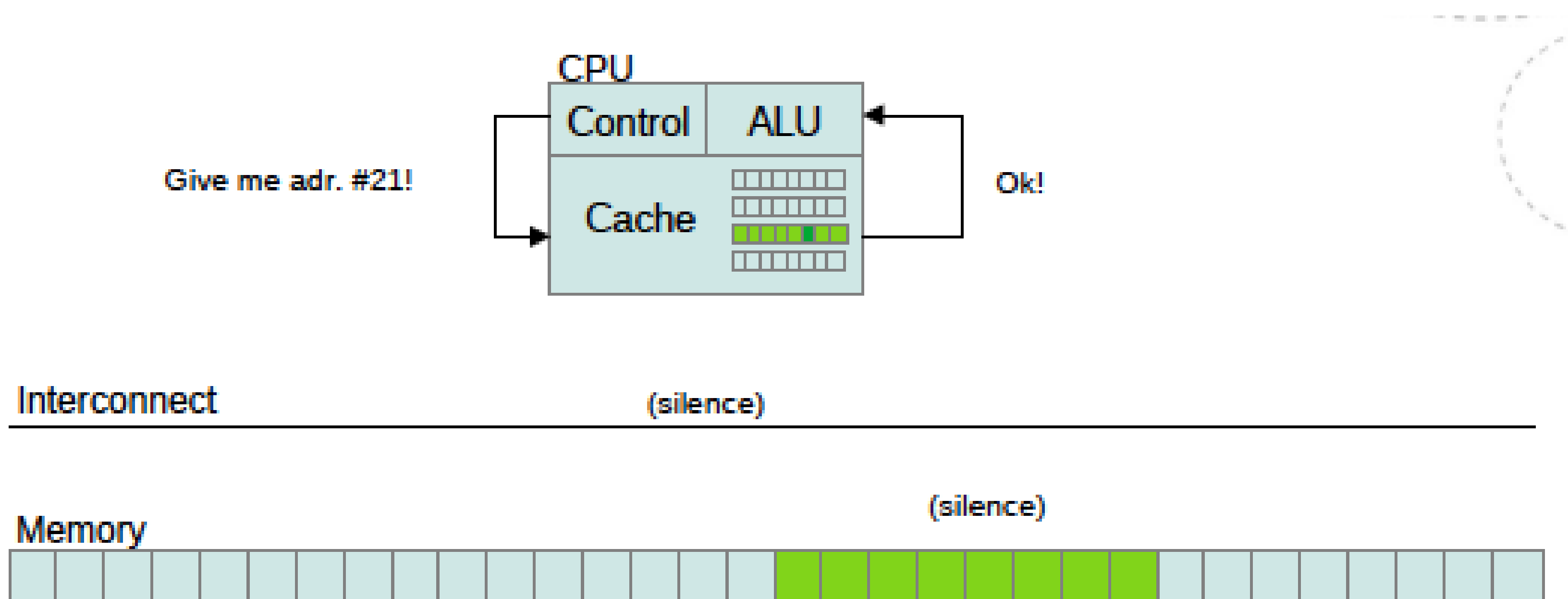
Reading un-cached value:



# Caching

(based on JCM slides26)

Reading another (now cached)  
value:

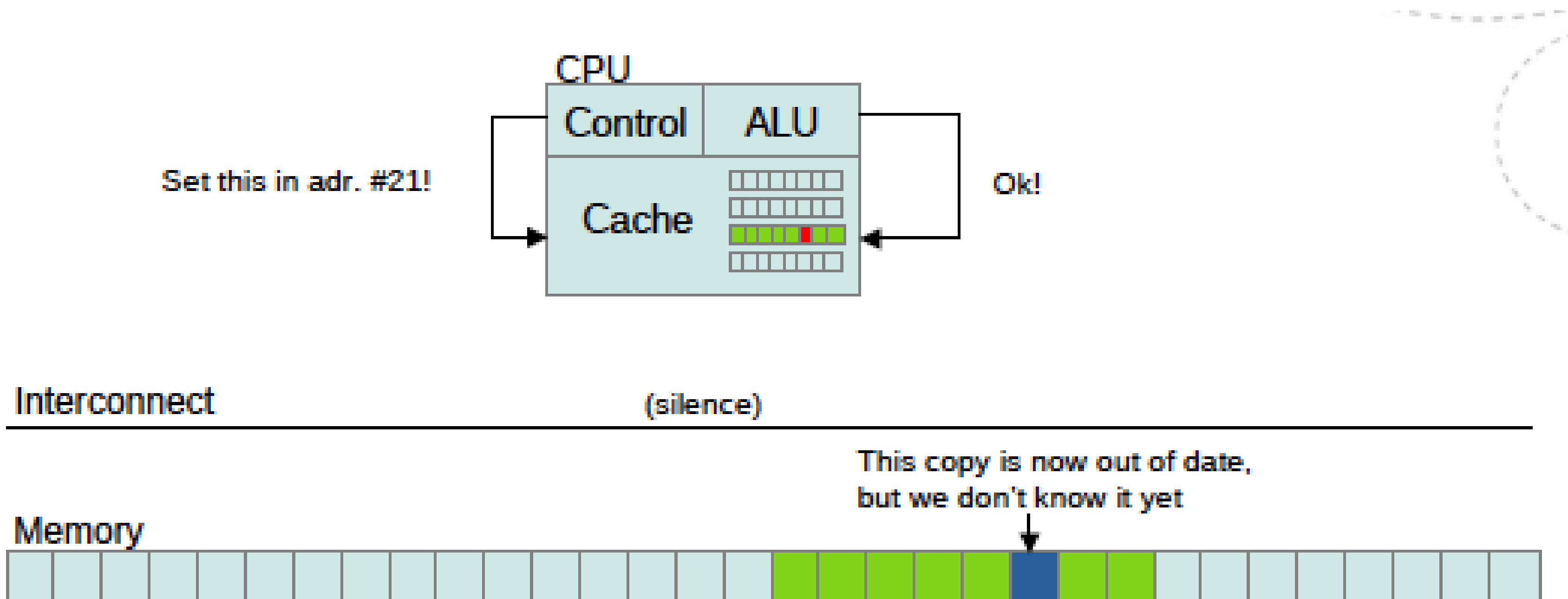




# Caching

(based on JCM slides26)

Writing a cached value:



# Caching – writing cached value

(based on JCM slides26)

We have 2 options:

1) Push write operations straight through cache memory, and update main memory ASAP

- + Simple and inexpensive implementation
- May create constant memory traffic

2) Delay write operations in memory, continue working with the cached copy

- + Doesn't do unnecessary work
- Requires more complex circuitry and wiring

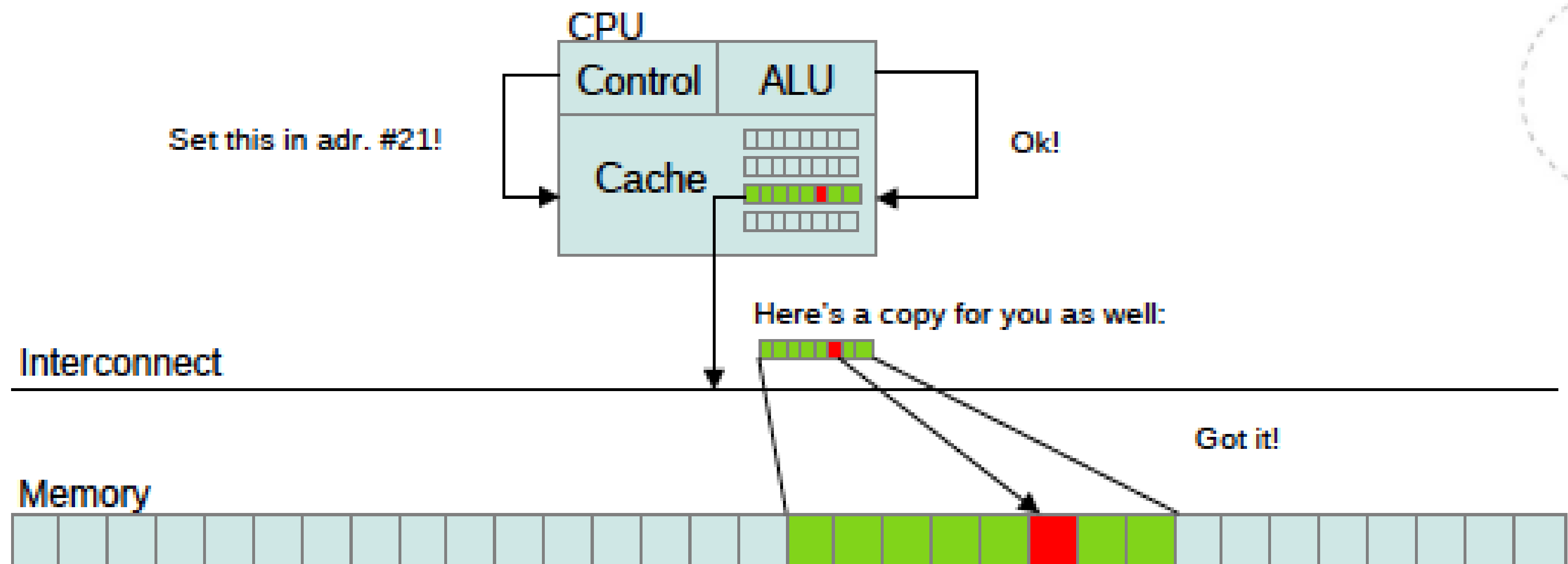
(Similar tradeoff to eager vs. lazy evaluation in programming languages)

# Caching – writing cached value

(JCM slides26 – slide 8)

## *Write-through* caching

- Write-through caches immediately pass their updates to main memory via the interconnect



# Caching – writing cached value

(based on JCM slides26)

See JCM slides24, slides 8-24



# MPI is SPMD!

- Single Program, Multiple Data
- Model proposed in 1984
  - by **Dr. Frederica Darema** (IBM)
    - later @ DARPA, NSF and now Director of US AirForce Research.
    - also known for [Dynamic Data Driven Application Systems](https://www.af.mil/About-Us/Biographies/Display/Article/3271084/dr-frederica-darema/) (DDDAS) proposed in 2000.



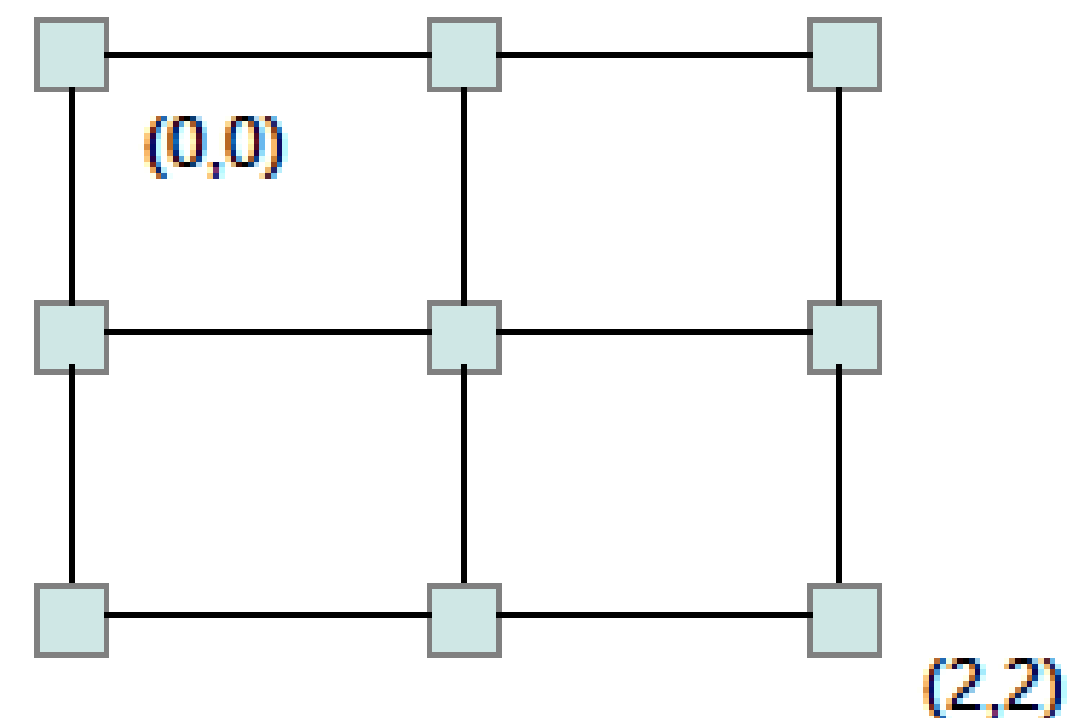
<https://www.af.mil/About-Us/Biographies/Display/Article/3271084/dr-frederica-darema/>

- Can be thought of as subcategory of MIMD
- **MPI is SPMD!**

Unlike SIMD, can use control statements like if/else to do execute different instructions concurrently

# MPI Virtual Topologies (from JCM Slides 08)

- The world communicator has no internal structure, everyone just gets a number for rank
- MPI lets you declare communicators that have structure, e.g. the *Cartesian* flavor, where every rank has a set of coordinates:
- This way you can send/receive messages with “rank at (1,1)” instead of having to calculate an indexing scheme yourself
- We can get communicators shaped like arbitrary graphs as well, but this rectangular thing is common



# MPI Communication modes

(based on JCM Slides 08)

Point-to-point messages ( e.g. MPI\_Send and MPI\_Recv) can be sent with **4 different guarantees for how they are transmitted:**

- **Standard** (implementation default, more on this later)
- **Synchronized** (Send-function will not return until reception is acknowledged)
- **Buffered** (Explicitly manage the memory that's used for sending/receiving)
- **Ready** (Assume that the receiver has already initiated the receive)

# MPI Non-Blocking Communication

(based on JCM Slides 08)

- Usually, send and receive operations cause the program to stop and wait for the message to come through, and only resume the program afterwards
- This is not 100% true, but close enough for now
- Non-blocking sending and receiving immediately returns a request instead, so that you can continue calculating
- In order to make sure that the message has gone/come through, you **must issue a wait-for-completion** call for the request later on
  - Whenever you can no longer proceed without the comms being complete



# More MPI

## MPI Collectives :

- MPI\_Barrier,
- MPI\_Bcast,
- MPI\_Scatter /MPI\_Gather (+ all-gather /all-scatter)

## MPI Derived datatypes:

- combining several datatypes in a reusable buffer

## MPI Parallel I/O

- Lecture with Jan Chr. Meyer on Sept 11!



# MPI functions -- OpenMPI

## MPI API (section 3 man pages)

<a href="#">MPI</a>	<a href="#">MPI File call errhandler</a>	<a href="#">MPI Ineighbor allgather</a>	<a href="#">MPI T init thread</a>
<a href="#">MPIX Allgather init</a>	<a href="#">MPI File close</a>	<a href="#">MPI Ineighbor allgatherv</a>	<a href="#">MPI T pvar get info</a>
<a href="#">MPIX Allgatherv init</a>	<a href="#">MPI File create errhandler</a>	<a href="#">MPI Ineighbor alltoall</a>	<a href="#">MPI T pvar get num</a>
<a href="#">MPIX Allreduce init</a>	<a href="#">MPI File delete</a>	<a href="#">MPI Ineighbor alltoallv</a>	<a href="#">MPI T pvar handle alloc</a>
<a href="#">MPIX Alltoall init</a>	<a href="#">MPI File f2c</a>	<a href="#">MPI Ineighbor alltoallw</a>	<a href="#">MPI T pvar handle free</a>
<a href="#">MPIX Alltoallv init</a>	<a href="#">MPI File get amode</a>	<a href="#">MPI Info c2f</a>	<a href="#">MPI T pvar read</a>
<a href="#">MPIX Alltoallw init</a>	<a href="#">MPI File get atomicity</a>	<a href="#">MPI Info create</a>	<a href="#">MPI T pvar readreset</a>
<a href="#">MPIX Barrier init</a>	<a href="#">MPI File get byte offset</a>	<a href="#">MPI Info delete</a>	<a href="#">MPI T pvar reset</a>
<a href="#">MPIX Bcast init</a>	<a href="#">MPI File get errhandler</a>	<a href="#">MPI Info dup</a>	<a href="#">MPI T pvar session create</a>
<a href="#">MPIX Exscan init</a>	<a href="#">MPI File get group</a>	<a href="#">MPI Info env</a>	<a href="#">MPI T pvar session free</a>
<a href="#">MPIX Gather init</a>	<a href="#">MPI File get info</a>	<a href="#">MPI Info f2c</a>	<a href="#">MPI T pvar start</a>
<a href="#">MPIX Gatherv init</a>	<a href="#">MPI File get position</a>	<a href="#">MPI Info free</a>	<a href="#">MPI T pvar stop</a>
<a href="#">MPIX Neighbor allgather init</a>	<a href="#">MPI File get position shared</a>	<a href="#">MPI Info get</a>	<a href="#">MPI T pvar write</a>
<a href="#">MPIX Neighbor allgatherv init</a>	<a href="#">MPI File get size</a>	<a href="#">MPI Info get nkeys</a>	<a href="#">MPI Test</a>
<a href="#">MPIX Neighbor alltoall init</a>	<a href="#">MPI File get type extent</a>	<a href="#">MPI Info get nthkey</a>	<a href="#">MPI Test cancelled</a>
<a href="#">MPIX Neighbor alltoallv init</a>	<a href="#">MPI File get view</a>	<a href="#">MPI Info get valuelen</a>	<a href="#">MPI Testall</a>
<a href="#">MPIX Neighbor alltoallw init</a>	<a href="#">MPI File iread</a>	<a href="#">MPI Info set</a>	<a href="#">MPI Testany</a>
<a href="#">MPIX Query_cuda_support</a>	<a href="#">MPI File iread all</a>	<a href="#">MPI Init</a>	<a href="#">MPI Testsome</a>
<a href="#">MPIX Reduce init</a>	<a href="#">MPI File iread at</a>	<a href="#">MPI Init thread</a>	<a href="#">MPI Topo test</a>
<a href="#">MPIX Reduce scatter block init</a>	<a href="#">MPI File iread at all</a>	<a href="#">MPI Initialized</a>	<a href="#">MPI Type c2f</a>
<a href="#">MPIX Reduce scatter init</a>	<a href="#">MPI File iread shared</a>	<a href="#">MPI Intercomm create</a>	<a href="#">MPI Type commit</a>
<a href="#">MPIX Scan init</a>	<a href="#">MPI File iwrite</a>	<a href="#">MPI Intercomm merge</a>	<a href="#">MPI Type contiguous</a>
<a href="#">MPIX Scatter init</a>	<a href="#">MPI File iwrite all</a>	<a href="#">MPI Iprobe</a>	<a href="#">MPI Type create darray</a>
<a href="#">MPIX Scatterv init</a>	<a href="#">MPI File iwrite at</a>	<a href="#">MPI Irecv</a>	<a href="#">MPI Type create f90_complex</a>
<a href="#">MPI Abort</a>	<a href="#">MPI File iwrite at all</a>	<a href="#">MPI Ireduce</a>	<a href="#">MPI Type create f90_integer</a>
<a href="#">MPI Accumulate</a>	<a href="#">MPI File iwrite shared</a>	<a href="#">MPI Ireduce scatter</a>	<a href="#">MPI Type create f90_real</a>
<a href="#">MPI Add error class</a>	<a href="#">MPI File open</a>	<a href="#">MPI Ireduce scatter block</a>	<a href="#">MPI Type create hindexed</a>
<a href="#">MPI Add error code</a>	<a href="#">MPI File preallocate</a>	<a href="#">MPI Irsend</a>	<a href="#">MPI Type create hindexed block</a>
<a href="#">MPI Add error string</a>	<a href="#">MPI File read</a>	<a href="#">MPI Is thread main</a>	<a href="#">MPI Type create hvector</a>
<a href="#">MPI Address</a>	<a href="#">MPI File read all</a>	<a href="#">MPI Iscan</a>	<a href="#">MPI Type create indexed block</a>

[illegible][illegible][illegible][illegible]

# Observing re. MPI

(from JCM slides09)

Every MPI function is called something like

**MPI\_Abcd\_efg\_h**

- “MPI\_” to begin with
  - First letter in the function name is capitalized
  - The rest of the name is all in lowercase, with underscore separation
- MPI uses arguments to pass variables in and out of functions
    - For the vast, vast majority of functions, return value is an error code that indicates whether the function completed in style or not
    - In order to obtain the answer from a function, you pass it a pointer to an area you have sized up to contain it, and let the function write it there



# Observing re. MPI

(from JCM slides09)

## Why use pointer-arguments instead of C's own return values?

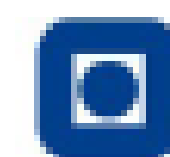
- There is actually a reasonable rationale behind this, you will find that system libraries and many other libraries do it as well
- The purpose is to give the programmer complete control over allocation
- If you're coming from an OO language, it's tempting to build 'constructors' for your structs like this:

```
my_thing * create_thing( int a, int b, int c ) { /* malloc in here */ }  
void destroy_thing ( my_thing *dead ) { free ( dead ); }
```

and use them like this

```
my_thing *newThing = create_thing (1,2,3);  
destroy_thing ( newThing );
```

- This will force all my\_things into the heap



# Parallel Computing is Fun!

