TDT 4200 Fall 2024
Parallel Computing

Lect 10-11: Non-blocking send/recv
Persistent communication,
MPI_AllReduce, MPI_Wtime(),
Hockney model (α + n*1/β)
MPI graphs and MPI Cartesian

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)

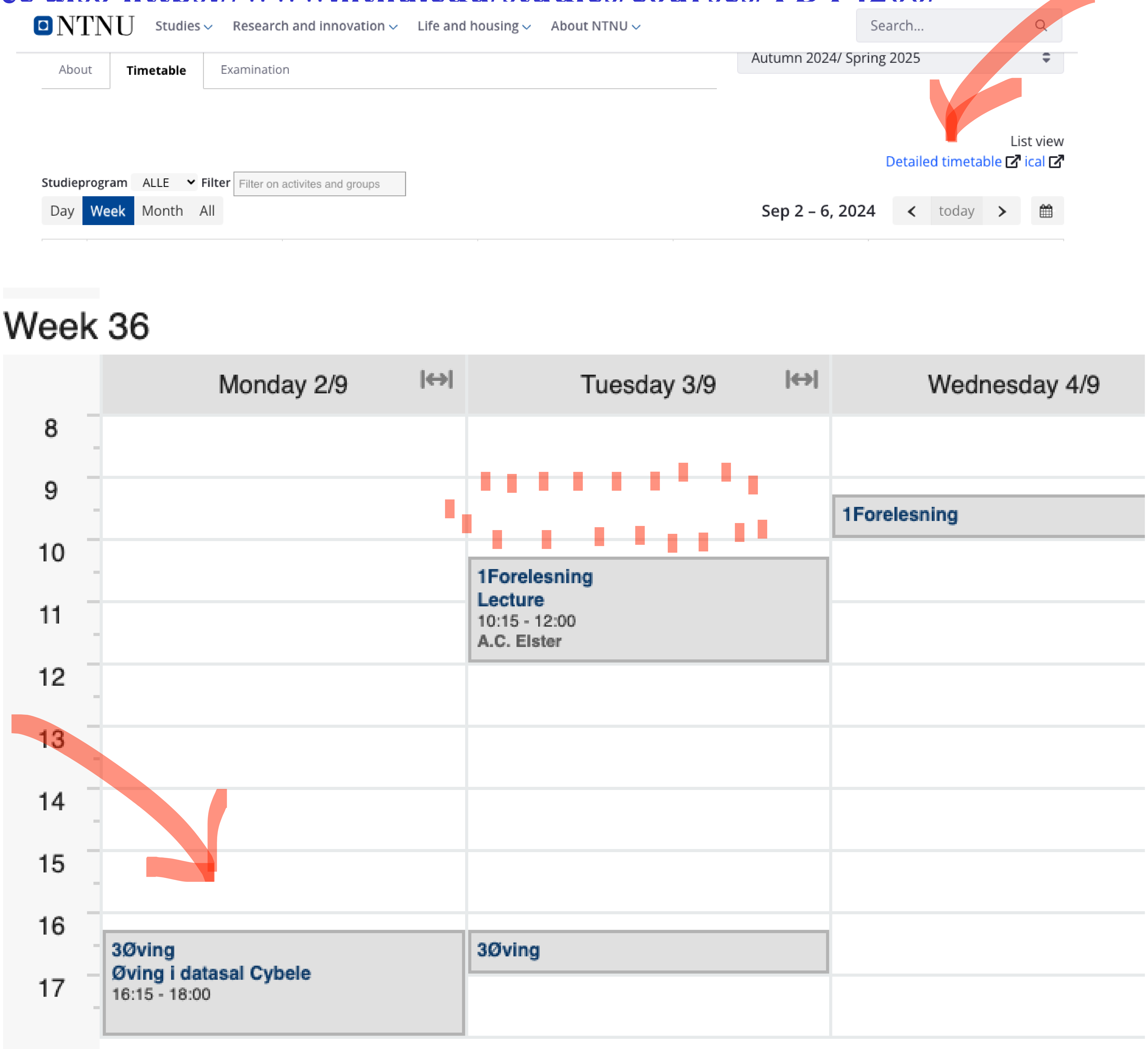
TDT4200 F2024 Course Information

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

- Instructor: Professor **Anne C. Elster** (elster@ntnu.no)
- Recitation lectures and assignments: **Tobias Dyngeland**
- Course supporter: Assoc. Prof. **Jan Christian Meyer**
- **2 part-time TAs (Læringsassistenter) positions announced**
- 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):**
  - **Lab help Mondays 16-18 @ Cybele**
  - Tuesdays: 16:15-17:00 in Kjel 5
- **Course Tool: BlackBoard**

TDT4200 F2024 Course Information

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



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!

New dates!

Problem Sets -- Tentative Dates

Tentative schedule for the problem sets :

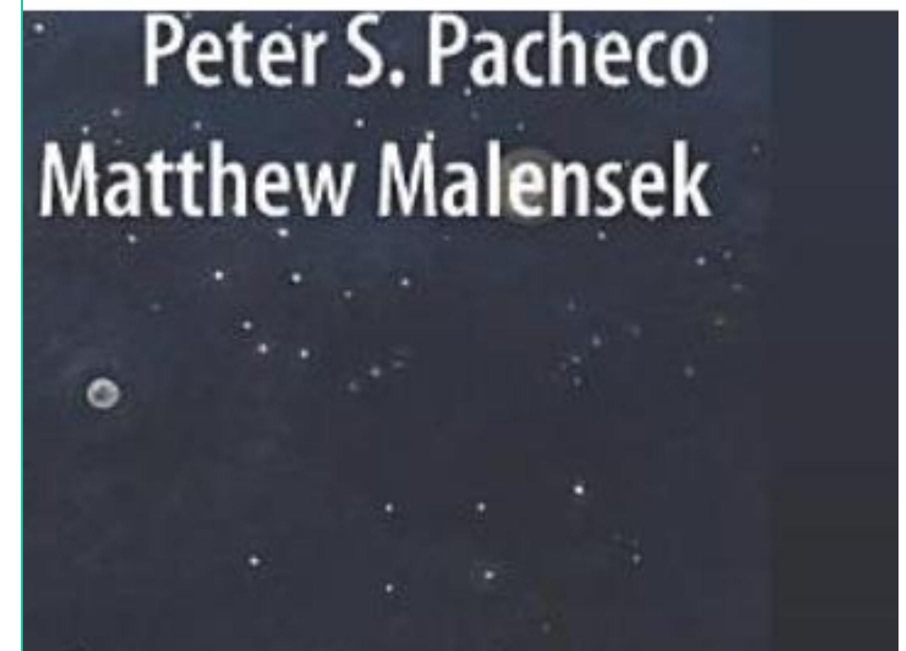
Problem set (Exercise)	Available (Tuesdays in Recitation)	Due Mondays 10pm	Topic	Grading
PS 0	Aug 20	Sep 2/9*	C - intro -- pointers ++ (optional, but highly recommended)	Pass/Fail -- Optional
PS 1a	Aug 27	Mon Sep 9*	C - Wave Equation	Pass/Fail -- Required
PS 1b	Aug 29	Mon Sep 9*	MPI Intro (Optional, but highly recommended)	Pass/Fail -- Optional
PS2	Sep 10	Mon Sep 23	MPI 1D Wave eqn	Pass/Fail - Required
PS3	Sep 17	Mon Sep 30/ Oct 7	**MPI 2D Wave eqn**	**Graded - Required**
PS4	Oct 1	Mon Oct 14/ 21	Pthreads/OpenMP	Pass/Fail - Required
PS5	Oct 15	Mon Oct 28	CUDA Intro	Pass/Fail - Required
PS6	Oct 27	Nov 11	**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.

BlackBoard – Problem Sets – Please do them!

Assignment Needs Grading (3)

- Excercise /PS 1B --- MPI Intro - Mandelbrot (41)
  TDT4200 Parallelle beregninger (2024 HØST)
- Exercise 0 (81)
  TDT4200 Parallelle beregninger (2024 HØST)
- Exercise 1A (125)
  TDT4200 Parallelle beregninger (2024 HØST)



Peter S. Pacheco
Matthew Malensek

AN INTRO
PARALLEL PROG

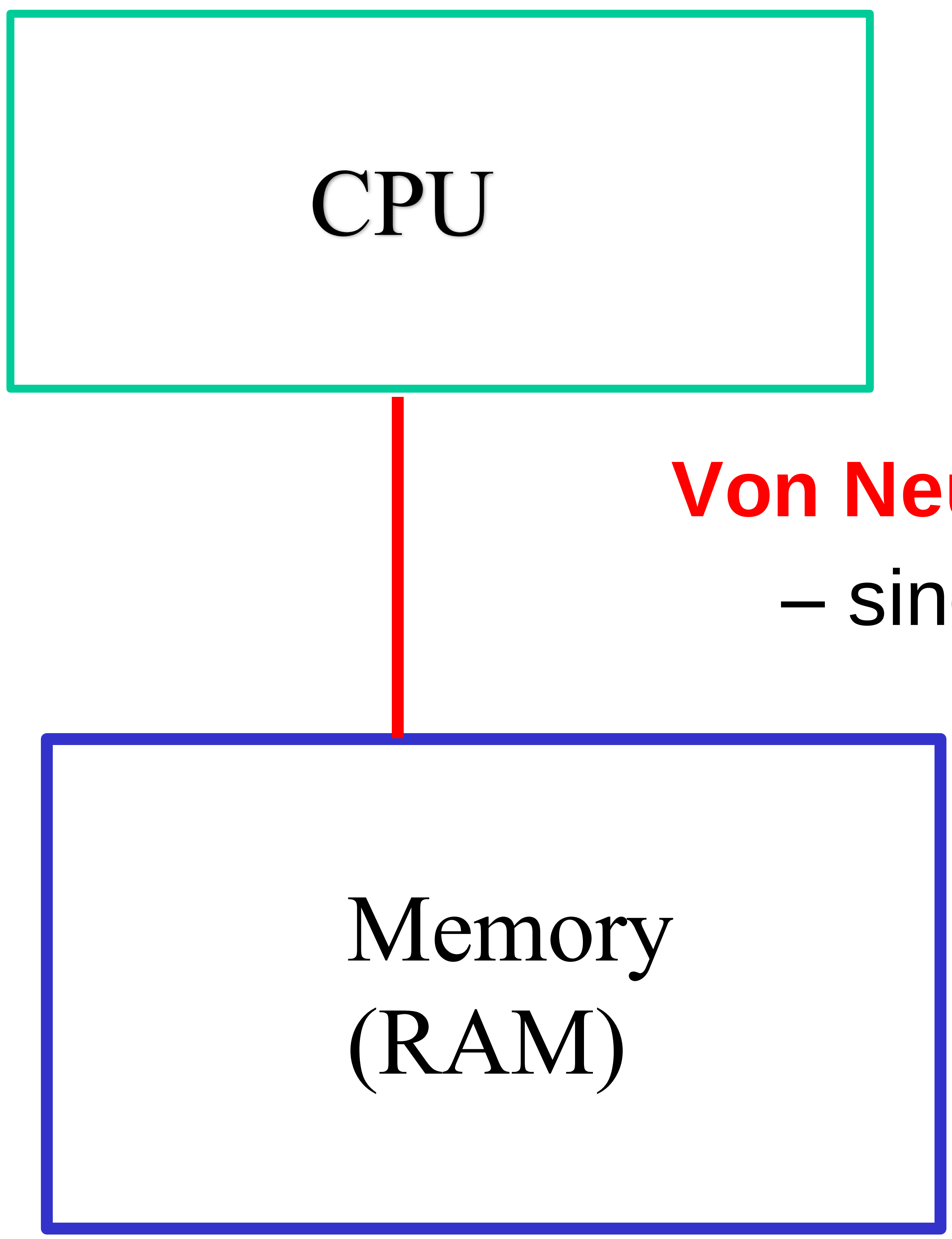
BlackBoard – Forum

Thread Actions | Collect | Delete

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

Thread Actions | Collect | Delete

The Von Neuman Architecture → sequential computer (SISD)



Von Neuman bottleneck

– since CPU 500-1000 times faster than RAM!

→ Add caches!

(Illustrated system with 3 cache levels on the blackboad)

HPC Challenge
-- Programming for Performance:

The challenges of data locality

1. Feeding the parallel cores registers fast enough
2. Moving compute to data (incl. compression of data)
3. Picking the right algorithm (that addresses NUMA memory /distributed memory)

I.e.

Computing is now like real estate – it is all about:

LOCATION, LOCATION & LOCATION!!

Come il prezzo degli immobili!

Outline

- Course Info.

- **MPI quick-review**
- **MPI_Reduce /MPI_Allreduce (JCM slides11, slide 16-17)**
- **MPI timing** – on blackboard and JCM slides13, slides 14-32
- MPI_Cartesian

MPI is SPMD!

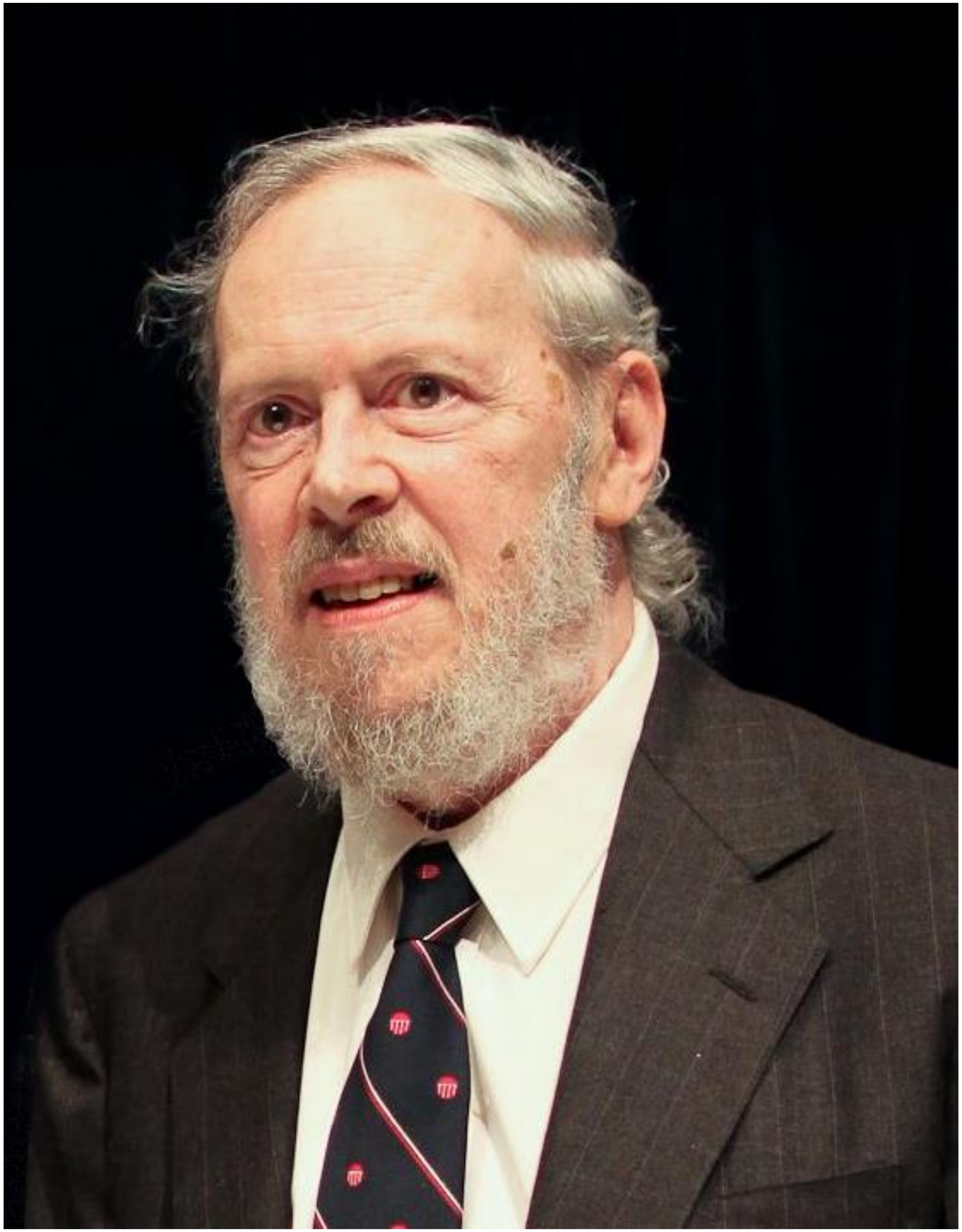
- Single Program, Multiple Data
- Model proposed in 1984
  - by **Dr. Frederica Darema** (IBM)
    - later @ DARPA, NSF and now Director of US AirForce Research.
    - also known for Dynamic Data Driven Application Systems (DDDAS) proposed in 2000.

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

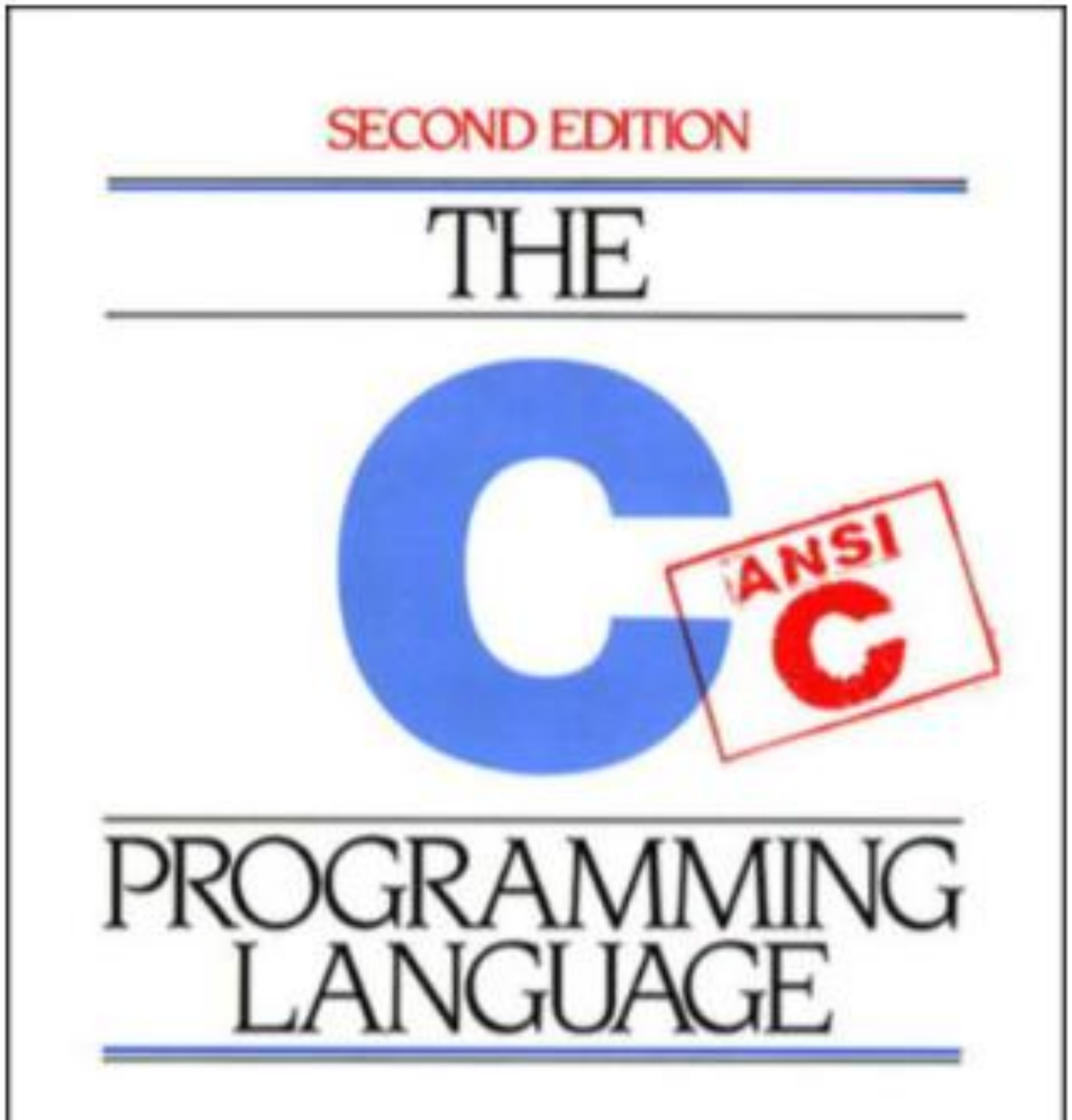
- Can be thought of as subcategory of MIMD
- **MPI is SPMD!**

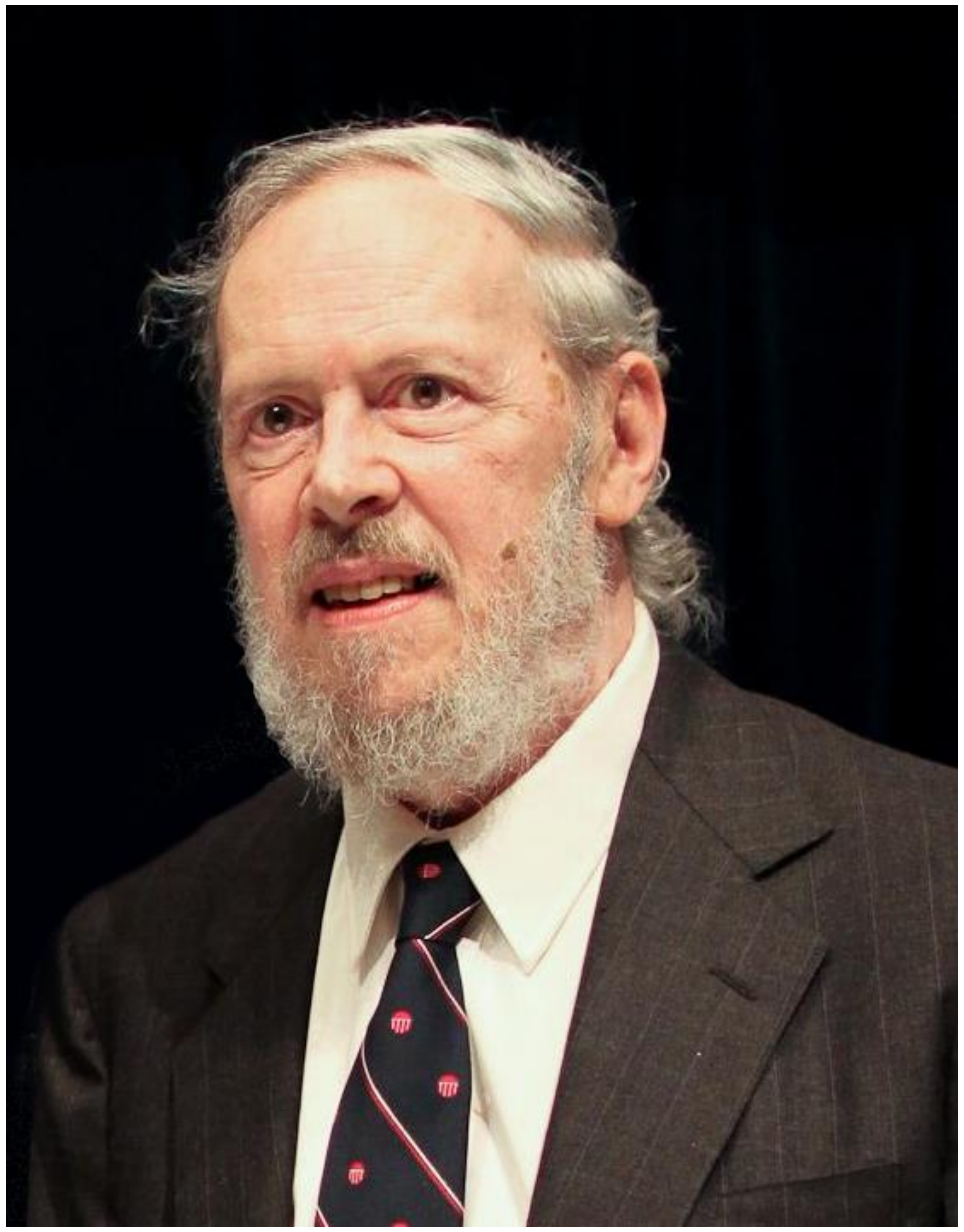
  Unlike SIMD, can use control statements like if/else to do execute different instructions concurrently

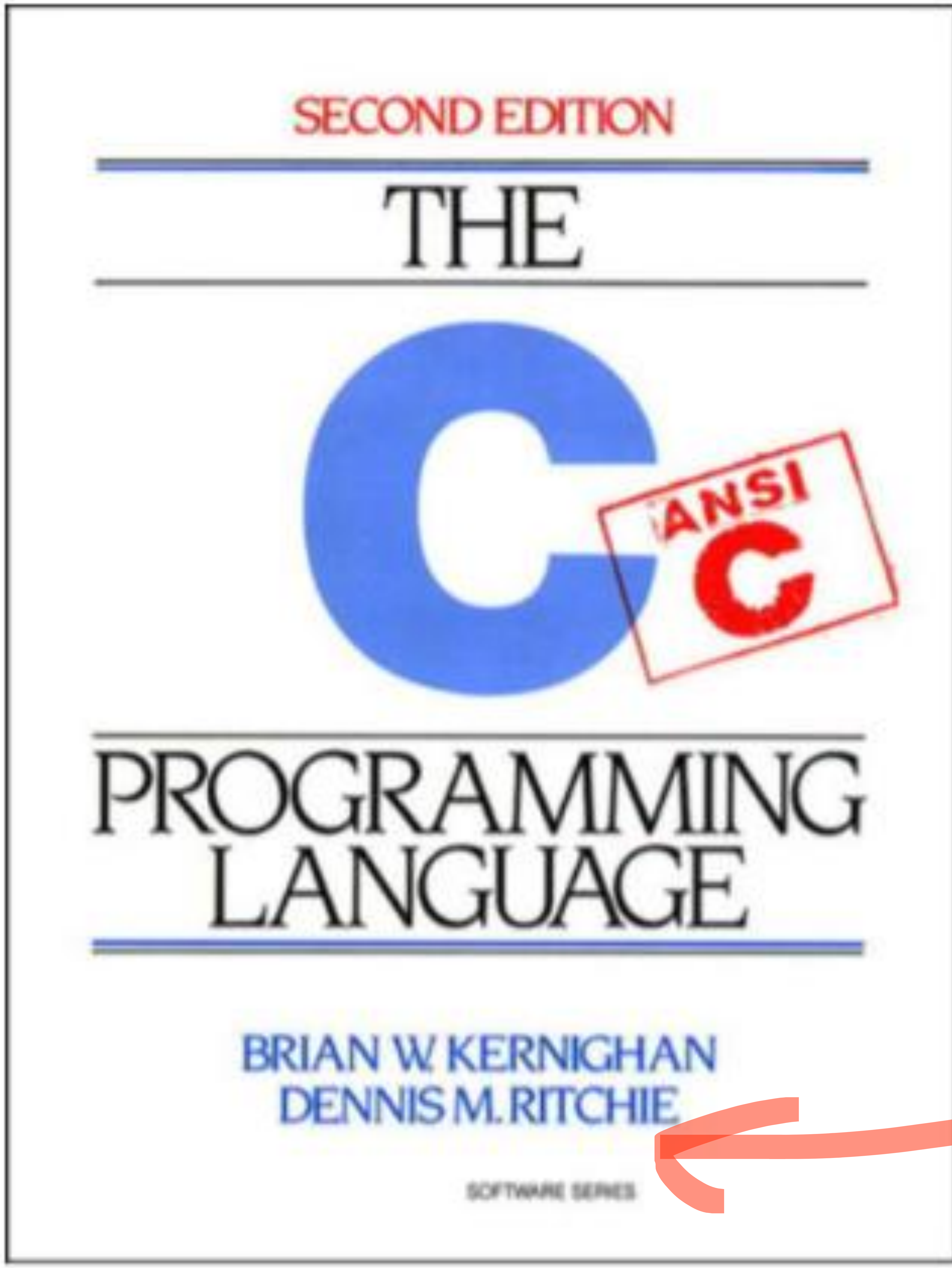
Who is this?

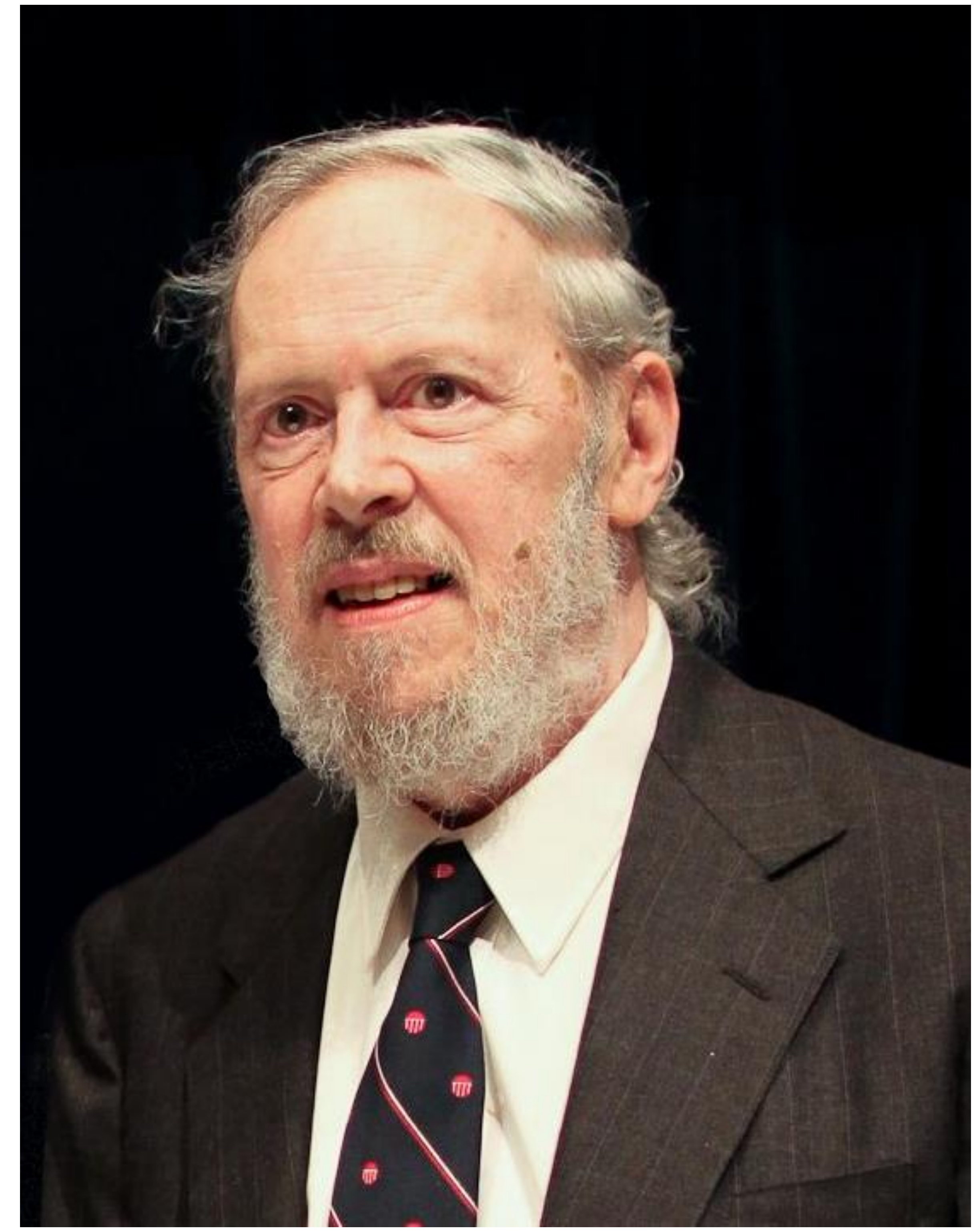


Who is this?






SECOND EDITION
THE
C
ANSI C
PROGRAMMING LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
SOFTWARE SERIES



K&R Book!

Dennis McAlistair Ritchie
-- “father of C” & “parent of Unix”

(September 9, 1941 – c. October 12, 2011)

- PhD work (not degree) @ Harvard 1968
- Father Bell Lab Alistaire Ritchie (switching crkts)
- Created **C prog. Lang.**
- With **Ken Thompson**, long-time colleague

  at Bell Labs also Multics work that led to **Unix** & **B language**

Dennis Ritchie & Ken Thompson
-- “father of C” & “parent of Unix”

- With Ken Thompson, long-time colleague
  at Bell Labs also Multics work that led to
  Unix & **B language (** Unix name suggested by
  Canadian Bell Lab colleague **Brian Kernighan**

- Ritchie & Thompson awarded the Turing (ACM) in 1983
- IEEE Richard W. Hamming Medal from IEEE in 1990
- National Medal of Technology from President Bill Clinton in 1999

- Ritchie was the head of Lucent Technologies System Software Research Department when he retired in 2007.

MPI C-bindings
(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

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 slides08)

- 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

MPI Non-Blocking Communication
(based on JCM slides12)

- **MPI_Isend**
- **MPI_Issend**
- **MPI_Ibsend**
- **MPI_Irsend**
- **MPI_Irecv**

There are even **non-blocking collectives**

They were only introduced in **MPI 3.0,** so not seen in a lot of production code yet.

Persistent Communication
(based on JCM slides12)

The **MPI_Request-objects** of **Isend** also have another application:

• If you're going to use the same communication pattern over and over (e.g. running neighbor exchanges every iteration) you can let MPI prepare them once and for all, and just call on them every time you want to activate them

– saves a little bit of time with setting up the transmission

– saves a bit of code complexity in middle of a loop that you're probably filling up with other complicated expressions

Persistent Sends and Receives
(based on JCM slides12)

All our sending and receiving calls can be initialized like this:

int **MPI_Send**_init (<all the usual stuff>, MPI_Request *req);

int **MPI_Recv**_init (<all the usual stuff>, MPI_Request *req);

triggered like this:

int **MPI_Start** (MPI_request *req);

There is also an **MPI_Startall** that takes a count and an array of requests and waited for if they're non-blocking, as before.

MPI Non-Blocking Communication
(based on JCM slides12)

- **MPI_Isend**
- **MPI_Issend**
- **MPI_Ibsend**
- **MPI_Irsend**
- **MPI_Irecv**

There are even **non-blocking collectives**
They were only introduced in **MPI 3.0,** so not seen in a lot of production code yet.

More MPI + related JCM slide references

MPI Collectives (JCM slides11):

- MPI_Barrier,
- MPI_Bcast,
- MPI_Scatter /MPI_Gather (+ all-gather /all-scatter)
- **MPI_Reduce /MPI_Allreduce** (slide 16 &17)

Timing in MPI (JCM slides13)

MPI Derived datatypes (JCM slides14):

- combing several datatypes in a reusable buffer

MPI_Cartesian (JCM slides16)

MPI Parallel I/O

- Lecture with Jan Chr. Meyer tomorrow (Sept 11, 2024)!

MPI_Reduce (based on JCM slide11, slide 16)

```
int MPI_Reduce (
    const void *sendbuf,
    void *recvbuf,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm comm
  );
```

MPI_Op -- name of an operation that can be applied to combine the contributions from arbitrary pairs of ranks

– These include **MPI_SUM, MPI_PROD, MPI_MAX, MPI_BAND (‘bitwise and’),** and so on...

– The main thing is that **they have to be commutative**

MPI_Reduce /MPI_Allreduce

MPI_Allreduce

- No root
- Need to allocate local buffer for each process since all processes receive copy of the result

The Pi example with reduction

- estimate_pi_reduction.c replaces our point-to-point construct with a collective op. that takes a single line of code
- There is also an unrooted MPI_Allreduce
- It's the same as Reduce, except that
  - There's no *root* argument
  - recv-buffer has to be allocated on all participants, because everyone gets a copy of the result
- estimate_pi_allreduce.c uses that instead

MPI_Wtime() (JCM slides13)

There’s no MPI requirement for what calendar year, time zone, country, or parallel universe the clock is relative to – It’s just some number of seconds

- That’s ok, because we mainly want to measure differences in it:

double t_start = MPI_Wtime();
do_something_useful();
double t_end = **MPI_Wtime();**
printf (“Something useful took %lf seconds!\n”,
t_end – t_start);

- **Hey, presto!**

MPI_Wtime() w/ many ranks (JCM slides13)

To isolate that your timings are only affected by the operations in the section you want to time, **synchronize the ranks first:**

```
MPI_Barrier ( MPI_COMM_WORLD );
   double t_start = MPI_Wtime();
   do_something_useful();
  double t_end = MPI_Wtime();
printf (
"Something useful took %ld seconds on rank %d!\n",
t_end – t_start, rank
);
```

- **You get P different timings** still, but you can collect them, find the average, variance, median, etc. etc. and figure out how long things take.

Approx. Communication time -- Hockney Model (JCM slides13)

If we know the size **n** of our message, we can estimate the transmission time as the **sum of latency** α **and n times the inverse bandwidth** β^{-1}:

$T\_comm(n) = \alpha + \mathbf{n}\, \beta^{-1}$

- first published by Roger W. Hockney
- Others call it the pingpong model, for reasons that will imminently be made clear

Approx. Communication time -- Hockney Model (JCM slides13)

9

Hockney's equipment

- Roger developed his model in order to estimate message costs on the Intel Paragon machine
  - The computer museum here at NTNU still has one
  - It doesn't run any more
- Communication links were equally fast throughout the entire machine
- Therefore, the α and β^{-1} could be measured between any pair of processors, and characterize the whole contraption

NTNU – Trondheim
Norwegian University of Science and Technology

Approx. Communication time -- Hockney Model (JCM slides13)

Ping-pong test of communication speed:
– Start the clock
– Repeat “a lot of” times:
- Send message from A to B (ping)
- Send message from B to A (pong)

– Stop the clock
– Divide the time difference by 2 (for both directions), and the number of messages

- “lot of” times has to be adjusted to whatever makes the procedure last long enough that you can reliably time it

– That depends on the speed of the equipment you’re using

Approx. Communication time -- Hockney Model (JCM slides13)

Latency lags bandwidth

- Latency is often the smaller part of transmission time
- It is, however, very difficult to improve upon:
  - Bandwidth can be expanded by adding extra lanes to the interconnect fabric
  - Latency is ultimately restricted by the speed of light, nothing can go faster from A to B
- Research in parallel computing is eagerly investigating *latency-masking techniques*
  - We can't get rid of it, but we can do something useful in the meantime
  - Overlapping computation with MPI_Isend is one such technique

Approx. Communication time -- Hockney Model (JCM slides13)

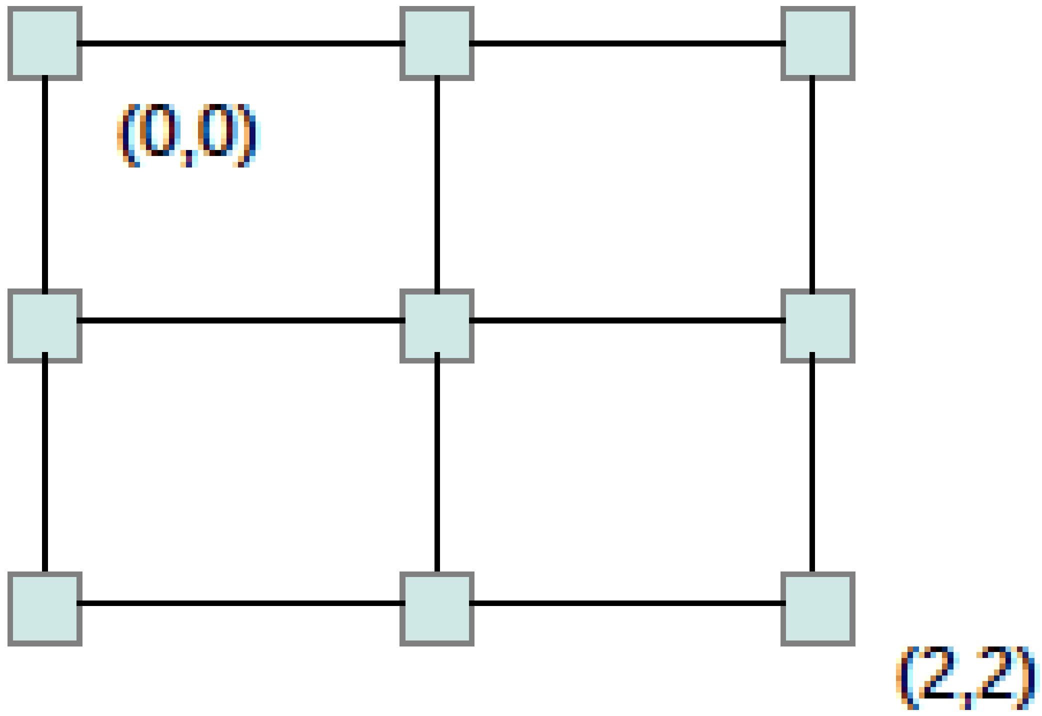
In modern times

- The days of uniform latency and bandwidth are long gone
  - The cost of sending messages between adjacent cores on a chip is wildly different from the cost of sending them to another computer across the room
- If you want to make sense of ping-pong results nowadays, you have to measure as many different α/β pairs as you have types of links in your platform
- It can still be useful, though, if you are careful about where your ranks are running

(There are also a couple of statistical techniques to make the measurements more stable and reliable, but I won't bother you with them in TDT4200)

MPI Virtual Topologies (from JCM Slides08)

- 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 Graphs (from JCM Slides15)

Create a graph communicator out of another communicator, by just imposing the graph structure:

```
int MPI_Graph_create (
    MPI_Comm old_communicator, ← Easy
    int number_of_nodes, ← Easy
    const int index[ ],
    const int edges[ ],
    int reorder, ← Easy
    MPI_Comm *new_communicator ← Easy
  );
```

Most of this is straightforward

- – “reorder” says whether or not MPI is allowed to give ranks in the new communicator different numbers, or whether it must keep the old values
    - – 0 means don’t reorder
    - – Not 0 means it’s ok to reorder (but it’s not an obligation)

MPI Graphs(from JCM Slides15)

Create a graph communicator out of another communicator, by just imposing the graph structure:

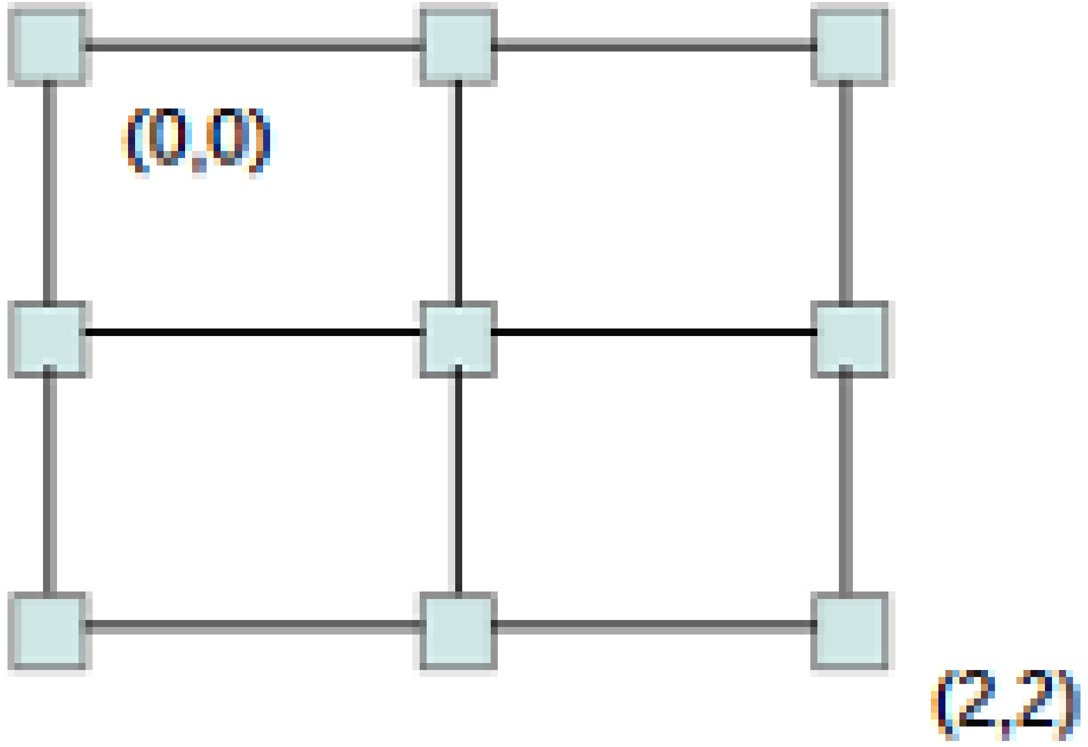
int MPI_Graph_create (
MPI_Comm old_communicator, ← Easy
int number_of_nodes, ← Easy
const int index[ ],
const int edges[ ],
int reorder, ← Easy
MPI_Comm *new_communicator ← Easy
);

The remaining two arguments ‘index’ and ‘edges’ are just some linear lists of integers. Sizing and contents can be a bit finickety, though. Let´s illustrate them using one particular graph topology -- the binary tree

MPI Trees. (from JCM Slides15)

See JCM slides 15, slides 15-23

MPI Cartesian (from JCM Slides16)



These are like a special case of graphs, but it's such a common case

– They arrange ranks into a regular array with neighbors to the

- left/right (1D),
- up/down (2D),
- in/out (3D), etc. etc.

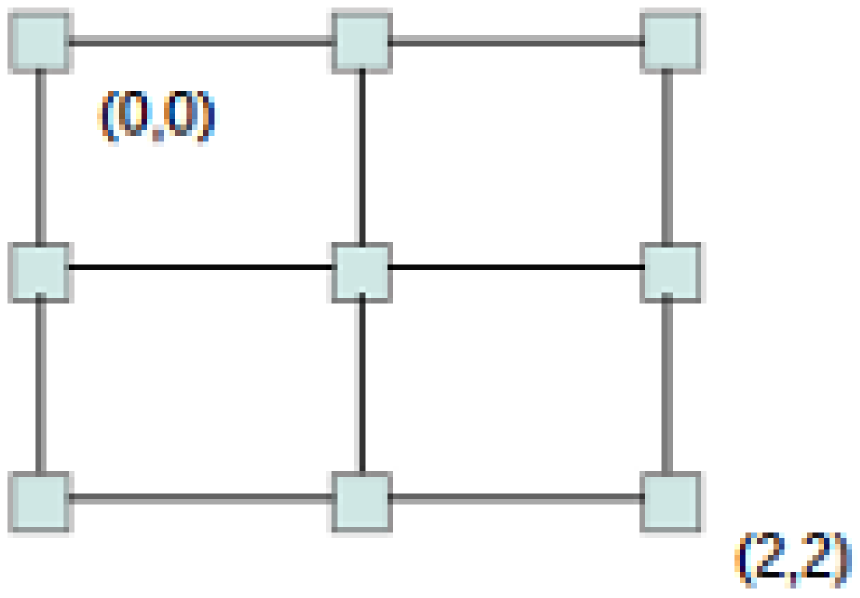
• The index/edge lists for this kind of graph become so regular that it would be a waste to write them out

– Each rank has 2 direct neighbors in each direction

– My neighbor's neighbor is my own 2nd neighbor

– ...and so on

Creating Cartesian Communicator (from JCM Slides16)

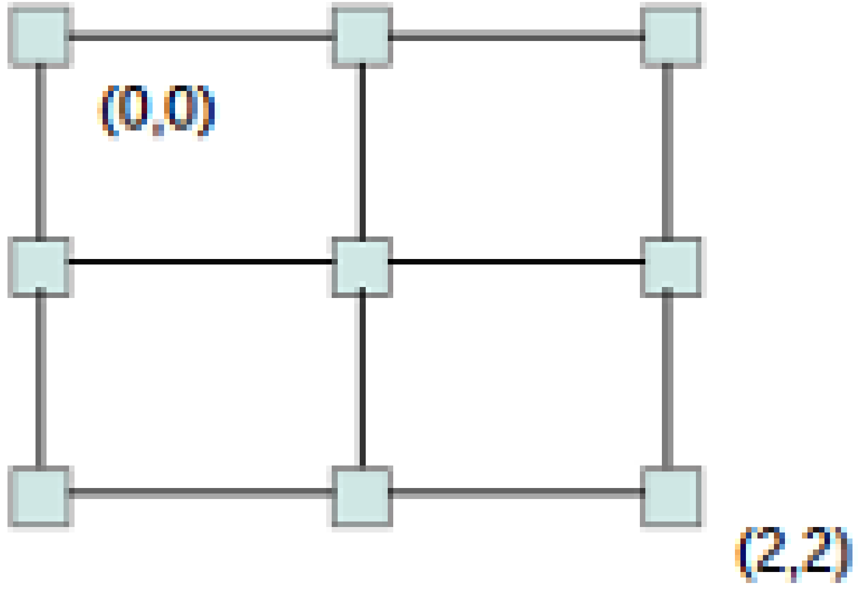


Like the graph communicator, it starts from another communicator, and just arranges all the ranks

- It needs some lists of integers as well:

int MPI_Cart_create (
MPI_Comm old_communicator,
int number_of_dimensions, ← Length of the next few lists
const int dims[ ], ← Size in each of n directions
const int periods[ ], ← Yes/no (1/0): wrap the edges?
int reorder, ← Same as for graph comms.
MPI_Comm *new_communicator ← Result
);

Creating Cartesian Communicator (from JCM Slides16)



See JCM slides16

MPI functions -- OpenMPI

MPI API
(section 3 man pages)

MPI
MPIX_Allgather_init
MPIX_Allgatherv_init
MPIX_Allreduce_init
MPIX_Alltoall_init
MPIX_Alltoallv_init
MPIX_Alltoallw_init
MPIX_Barrier_init
MPIX_Bcast_init
MPIX_Exscan_init
MPIX_Gather_init
MPIX_Gatherv_init
MPIX_Neighbor_allgather_init
MPIX_Neighbor_allgatherv_init
MPIX_Neighbor_alltoall_init
MPIX_Neighbor_alltoallv_init
MPIX_Neighbor_alltoallw_init
MPIX_Query_cuda_support
MPIX_Reduce_init
MPIX_Reduce_scatter_block_init
MPIX_Reduce_scatter_init
MPIX_Scan_init
MPIX_Scatter_init
MPIX_Scatterv_init
MPI_Abort
MPI_Accumulate
MPI_Add_error_class
MPI_Add_error_code
MPI_Add_error_string
MPI_Address

MPI_File_call_errhandler
MPI_File_close
MPI_File_create_errhandler
MPI_File_delete
MPI_File_f2c
MPI_File_get_amode
MPI_File_get_atomicity
MPI_File_get_byte_offset
MPI_File_get_errhandler
MPI_File_get_group
MPI_File_get_info
MPI_File_get_position
MPI_File_get_position_shared
MPI_File_get_size
MPI_File_get_type_extent
MPI_File_get_view
MPI_File_iread
MPI_File_iread_all
MPI_File_iread_at
MPI_File_iread_at_all
MPI_File_iread_shared
MPI_File_iwrite
MPI_File_iwrite_all
MPI_File_iwrite_at
MPI_File_iwrite_at_all
MPI_File_iwrite_shared
MPI_File_open
MPI_File_preallocate
MPI_File_read
MPI_File_read_all

MPI_Ineighbor_allgather
MPI_Ineighbor_allgatherv
MPI_Ineighbor_alltoall
MPI_Ineighbor_alltoallv
MPI_Ineighbor_alltoallw
MPI_Info_c2f
MPI_Info_create
MPI_Info_delete
MPI_Info_dup
MPI_Info_env
MPI_Info_f2c
MPI_Info_free
MPI_Info_get
MPI_Info_get_nkeys
MPI_Info_get_nthkey
MPI_Info_get_valuelen
MPI_Info_set
MPI_Init
MPI_Init_thread
MPI_Initialized
MPI_Intercomm_create
MPI_Intercomm_merge
MPI_Iprobe
MPI_Irecv
MPI_Ireduce
MPI_Ireduce_scatter
MPI_Ireduce_scatter_block
MPI_Irsend
MPI_Is_thread_main
MPI_Iscan

MPI_T_init_thread
MPI_T_pvar_get_info
MPI_T_pvar_get_num
MPI_T_pvar_handle_alloc
MPI_T_pvar_handle_free
MPI_T_pvar_read
MPI_T_pvar_readreset
MPI_T_pvar_reset
MPI_T_pvar_session_create
MPI_T_pvar_session_free
MPI_T_pvar_start
MPI_T_pvar_stop
MPI_T_pvar_write
MPI_Test
MPI_Test_cancelled
MPI_Testall
MPI_Testany
MPI_Testsome
MPI_Topo_test
MPI_Type_c2f
MPI_Type_commit
MPI_Type_contiguous
MPI_Type_create_darray
MPI_Type_create_f90_complex
MPI_Type_create_f90_integer
MPI_Type_create_f90_real
MPI_Type_create_hindexed
MPI_Type_create_hindexed_block
MPI_Type_create_hvector
MPI_Type_create_indexed_block

Parallel Computing is Fun!