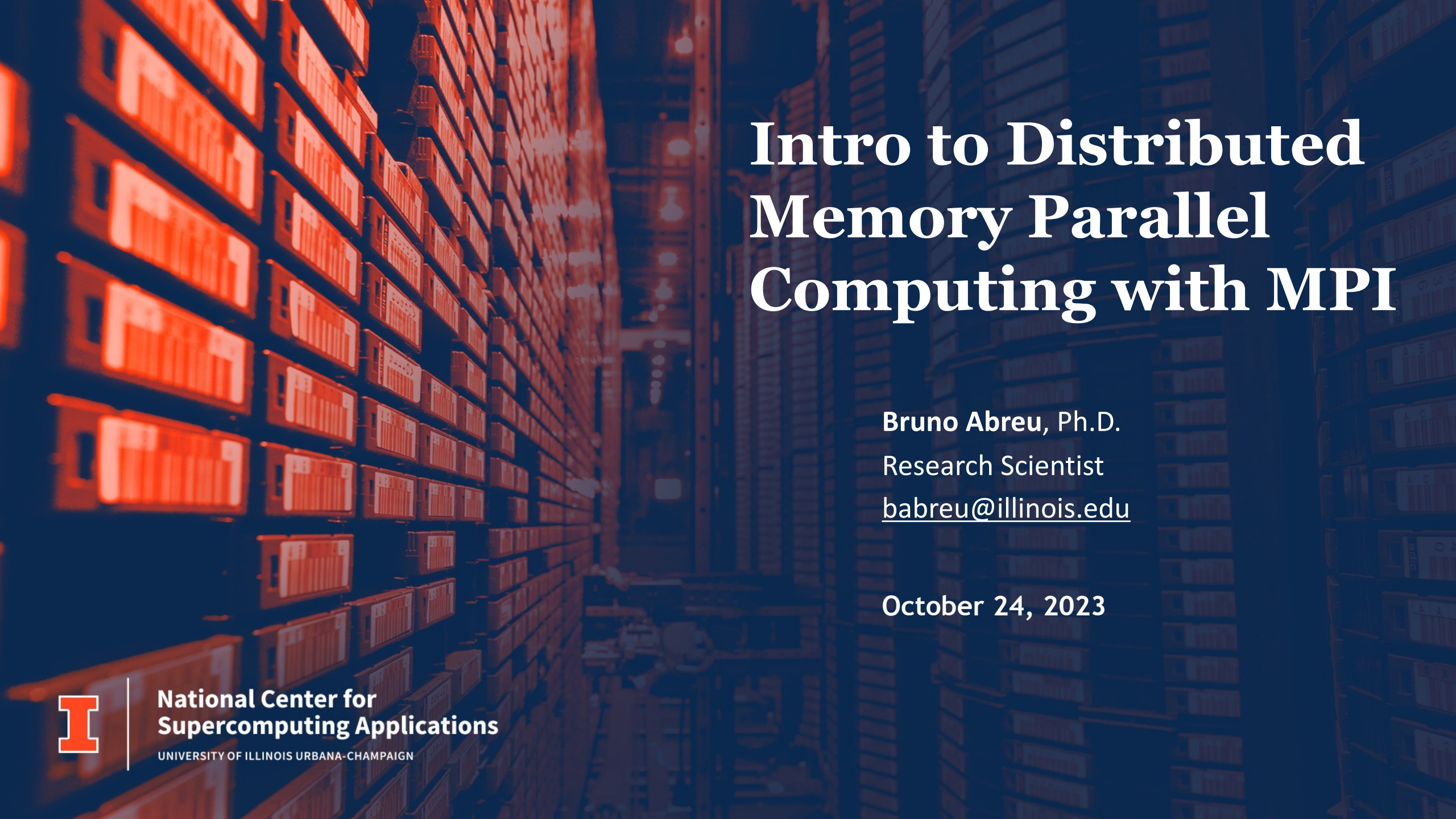


Intro to Distributed Memory Parallel Computing with MPI

Bruno Abreu, Ph.D.
Research Scientist
babreu@illinois.edu

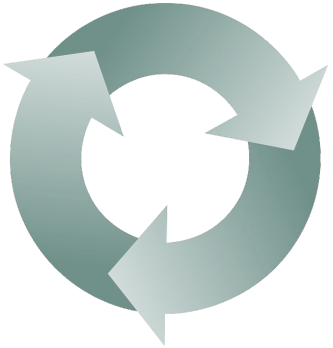
October 24, 2023

National Center for Supercomputing Applications
UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Outline

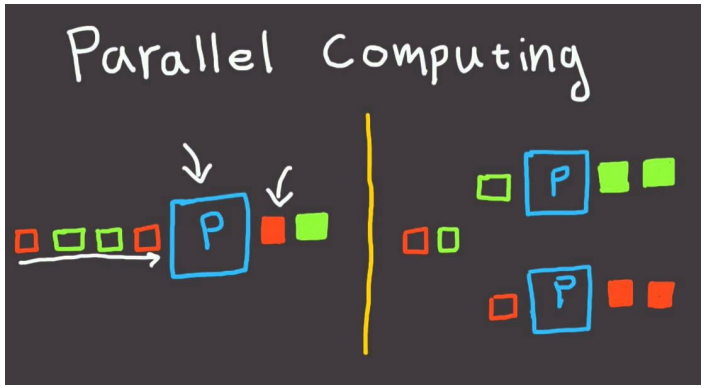
Part A: **Parallel Computing Quick Review** (10 min)

- Task and Data parallelism
- Parallel computers
- Threads, processes and more



Part B: **MPI Intro** (60 min)

- Core concepts
- Message passing
- Execution



Part C: **Exercises** (45+ min)

Ex 1: Hello World

Ex 2: Domain Decomposition

Ex 3: Parallel Linear Regression

Tools and resources we will cover

During the next 90 minutes, you will be directly or indirectly using/learning how to use:

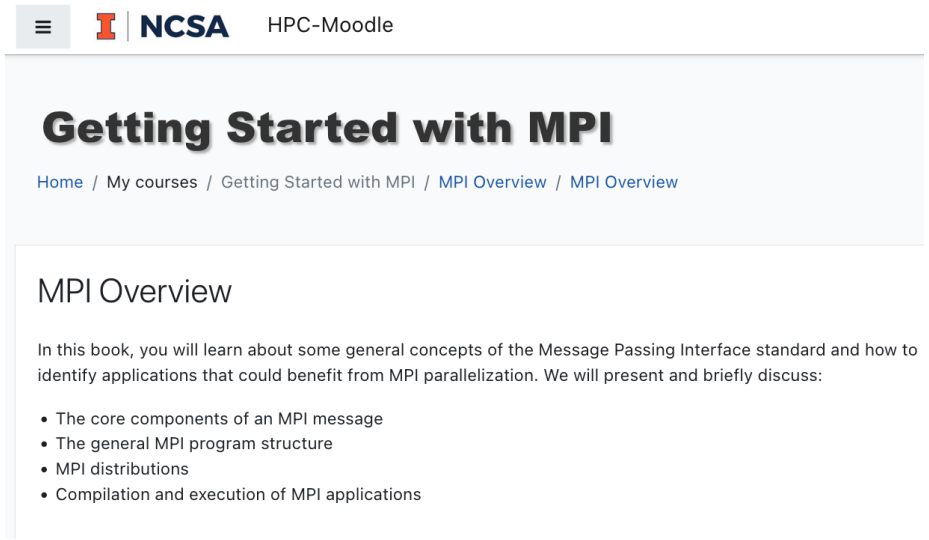
- Git and GitHub
- SLURM
- OpenMP
  - Loop scheduling
  - Data declarations
  - Reduction clauses
- Parallel computing
  - Measuring performance
  - Bottlenecks and common issues

- Illinois Campus Cluster

HPC Training Moodle

www.hpc-training.org

- Self-paced Tutorials
- Training Workshops
- Digital Badges

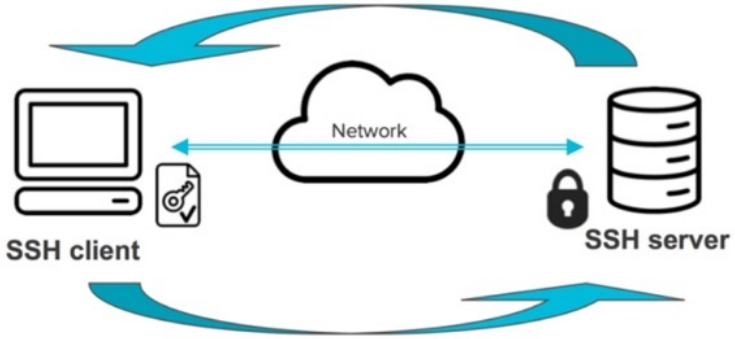


HPC-Moodle

- Intro to OpenMP
- Intro to MPI
- Intro to Visualization
- Getting Started on the Illinois Campus Cluster
- Developing Effective Training Webinars

We develop live training sessions (like this one!) on top of existing self-paced courses. If you want to revisit today's contents and/or learn more about other HPC tools and frameworks, this is the place!

Environment access



ssh

Illinois Campus Cluster Program
AT THE UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

macOS®

"Terminal" on *Spotlight Search* or *Finder*

Windows

Start -> Windows System -> Command Prompt

Linux

Ctrl+Alt+T

- `ssh -l <NetID> cc-login1.campuscluster.illinois.edu`

Cloning the repository

The exercises for today
can be found here:

https://github.com/babreu-ncsa/IntroToMPI/tree/uiuc-icc

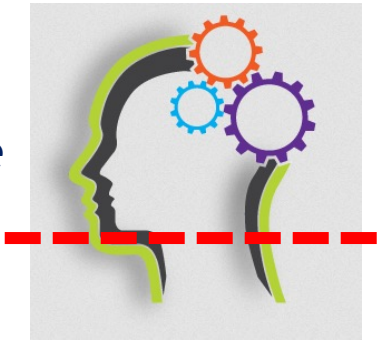
To copy these files to your ICC home directory, use:

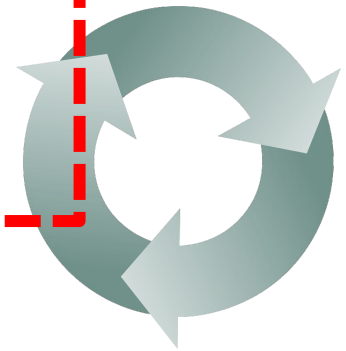
- `module load git`
- `git clone --branch uiuc-icc --single-branch https://github.com/babreu-ncsa/IntroToMPI.git`

Outline

Part A: **Parallel Computing Quick Review** (10 min)

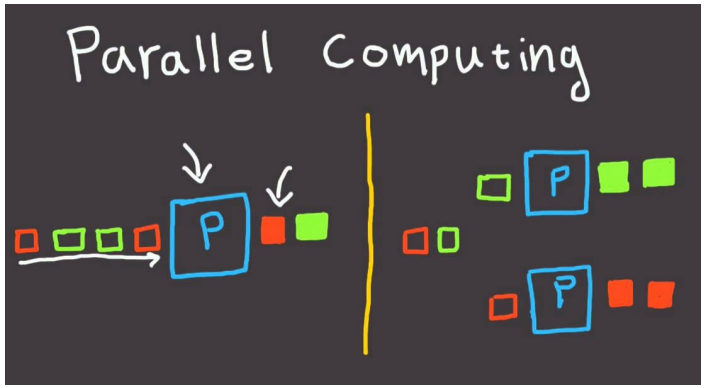
- Task and Data parallelism
- Parallel computers
- Threads, processes and more





Part B: **MPI Intro** (60 min)

- Core concepts
- Message passing
- Execution



Part C: **Exercises** (45+ min)

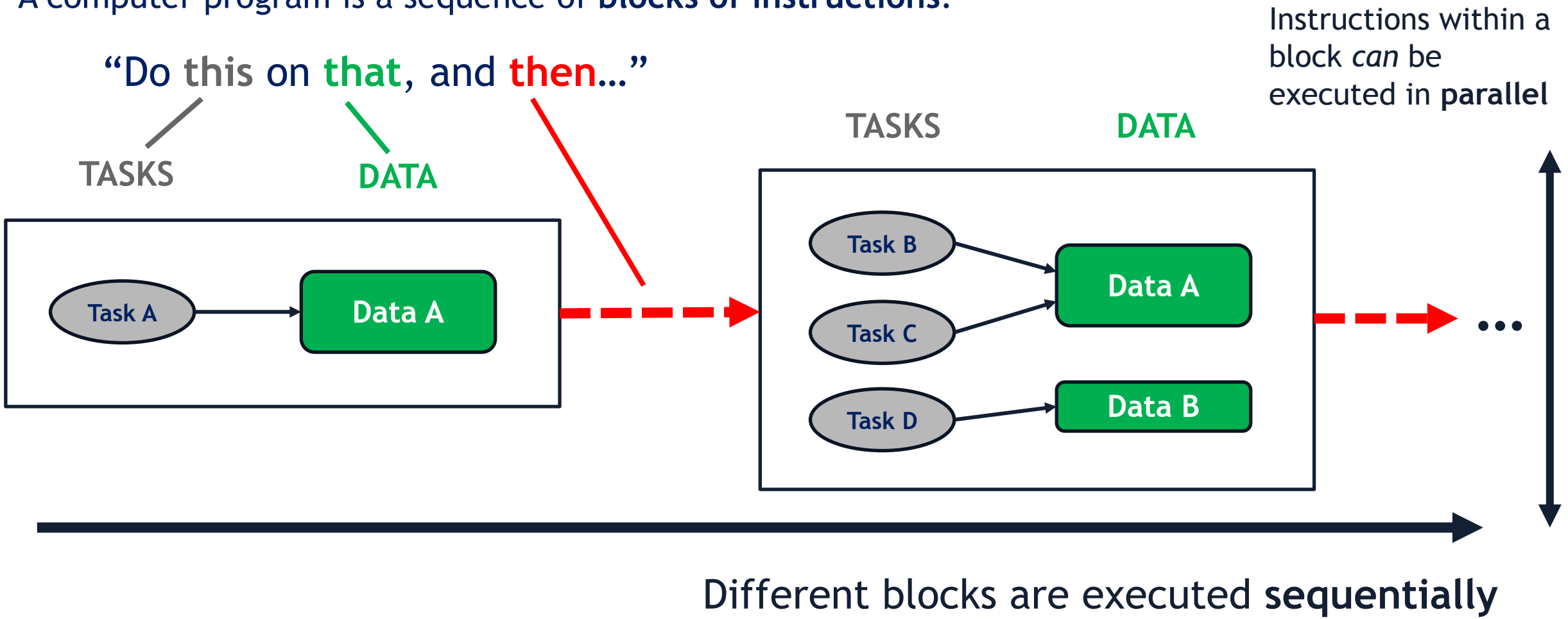
Ex 1: Hello World

Ex 2: Domain Decomposition

Ex 3: Parallel Linear Regression

Instruction blocks

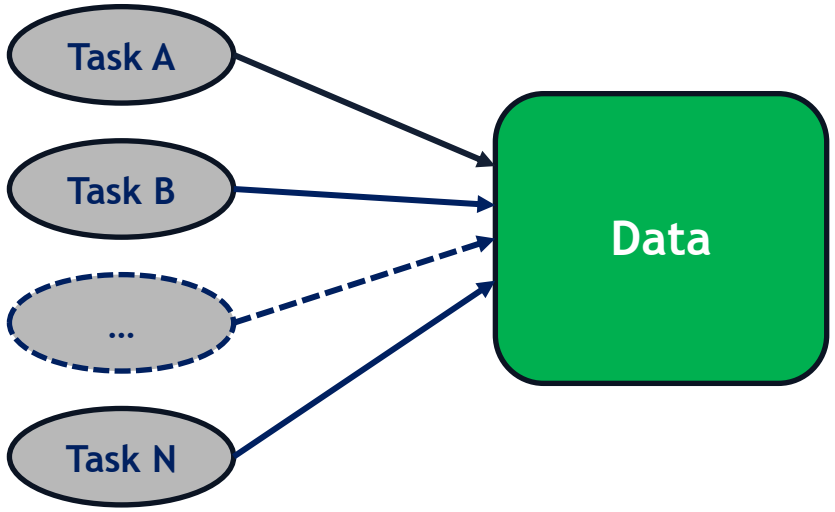
A computer program is a sequence of **blocks of instructions:**



Data & Task Parallelism

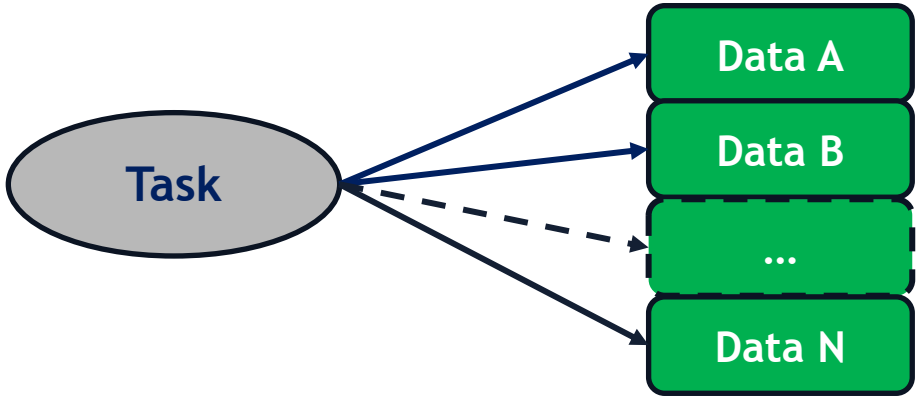
- An instruction is a pair (TASK, DATA)
- There are two “pure” ways of executing instructions in parallel:

Task Parallelism



Execute different tasks over the same data

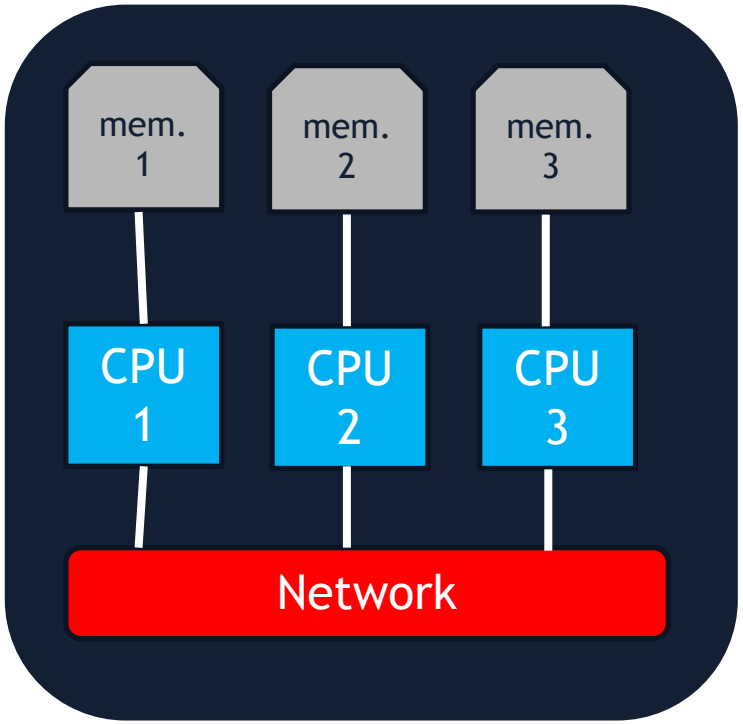
Data Parallelism



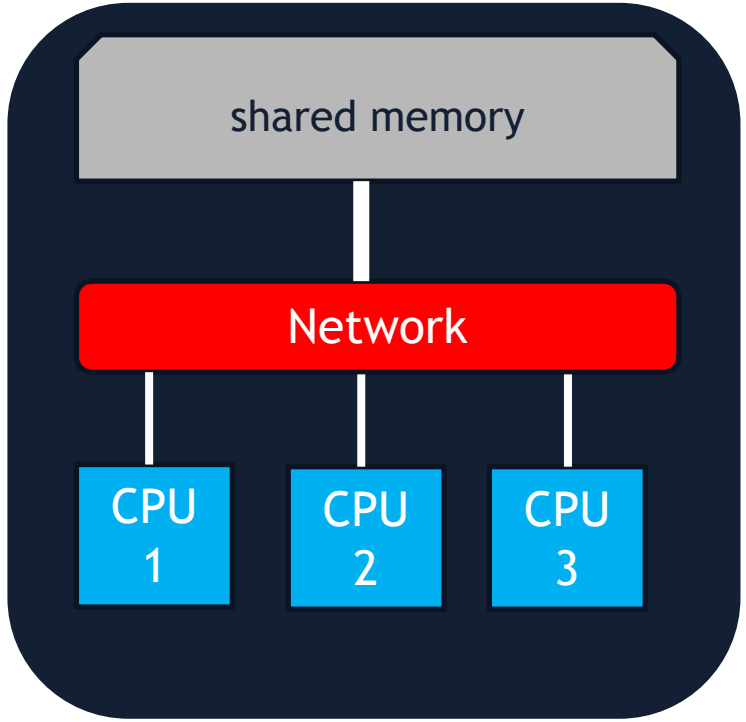
Execute the same task over different chunks of data

Parallel computers: memory

Distributed Memory

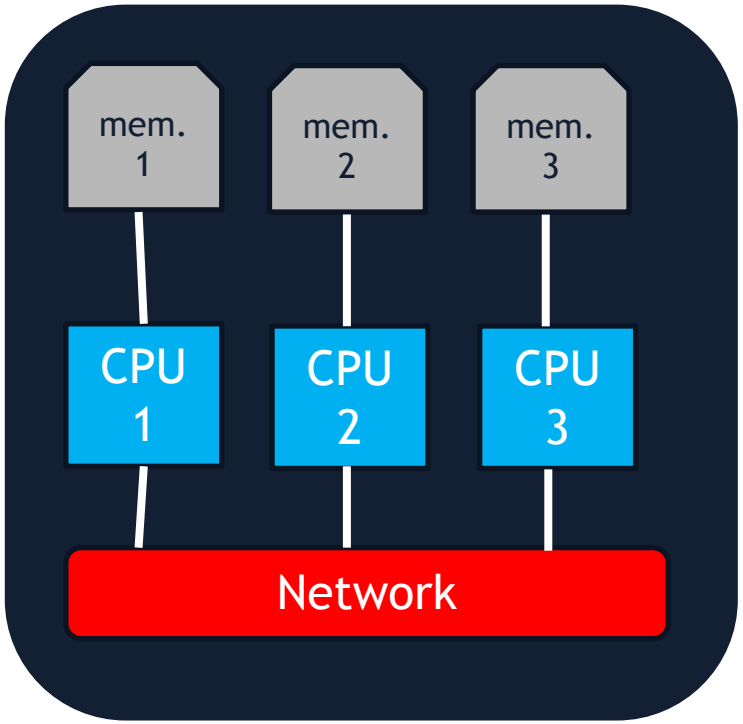


Shared Memory



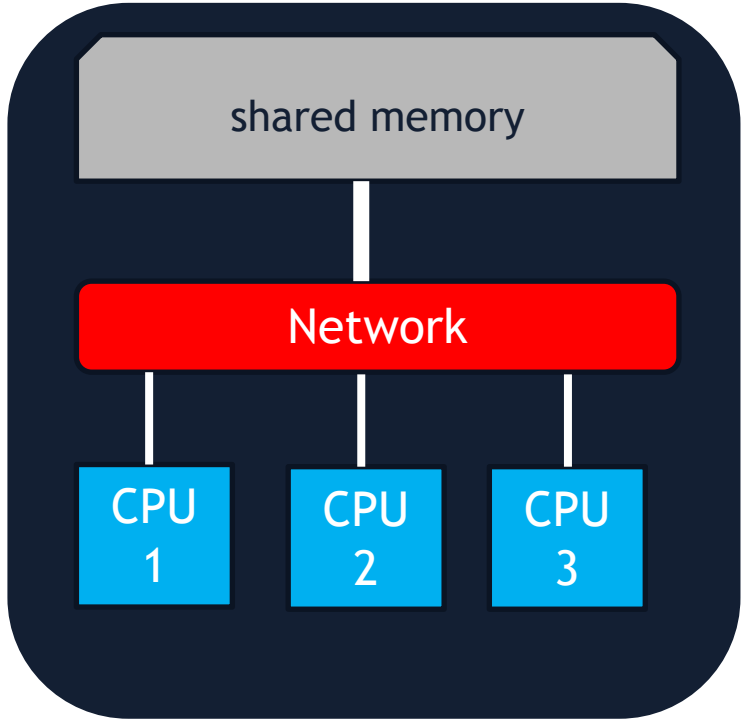
Parallel computers: memory

Distributed Memory



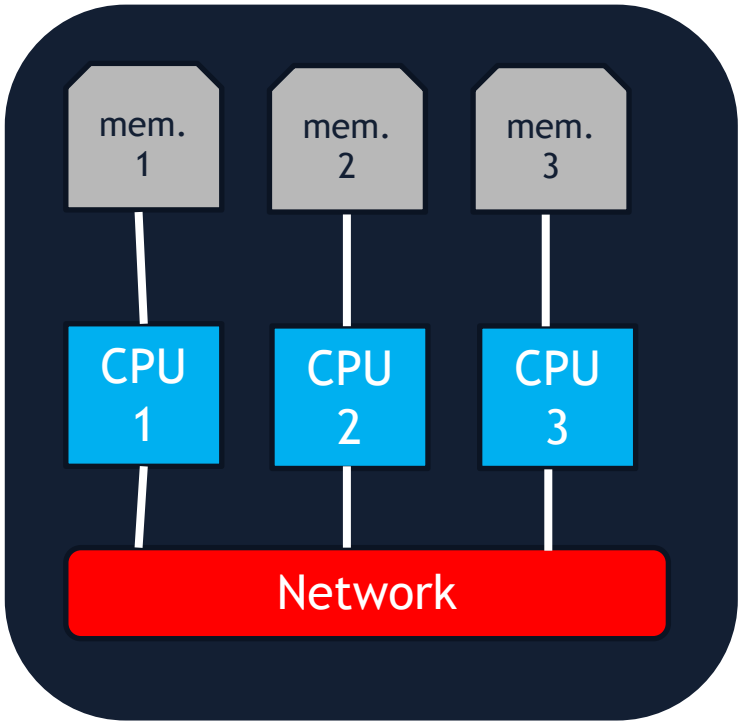
MPI

Shared Memory



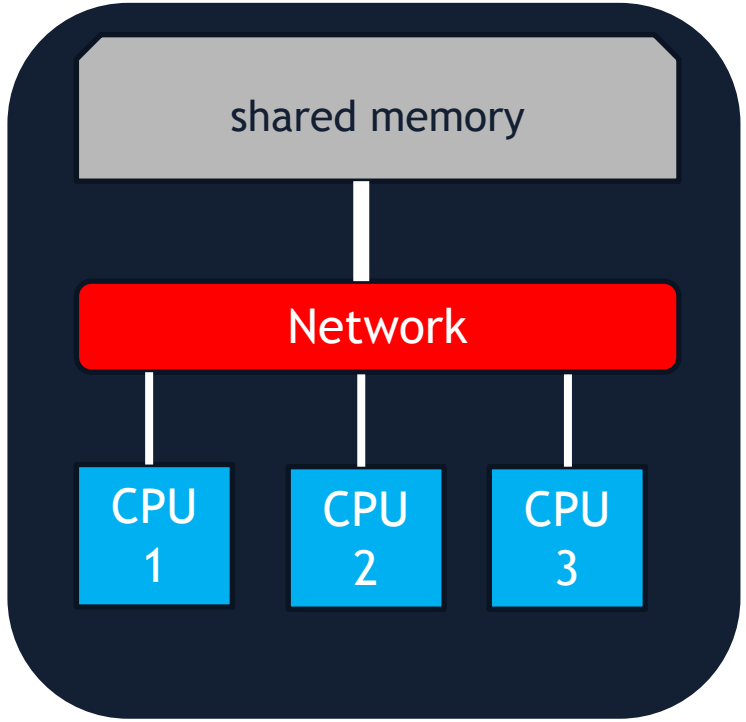
Parallel computers: memory

Distributed Memory

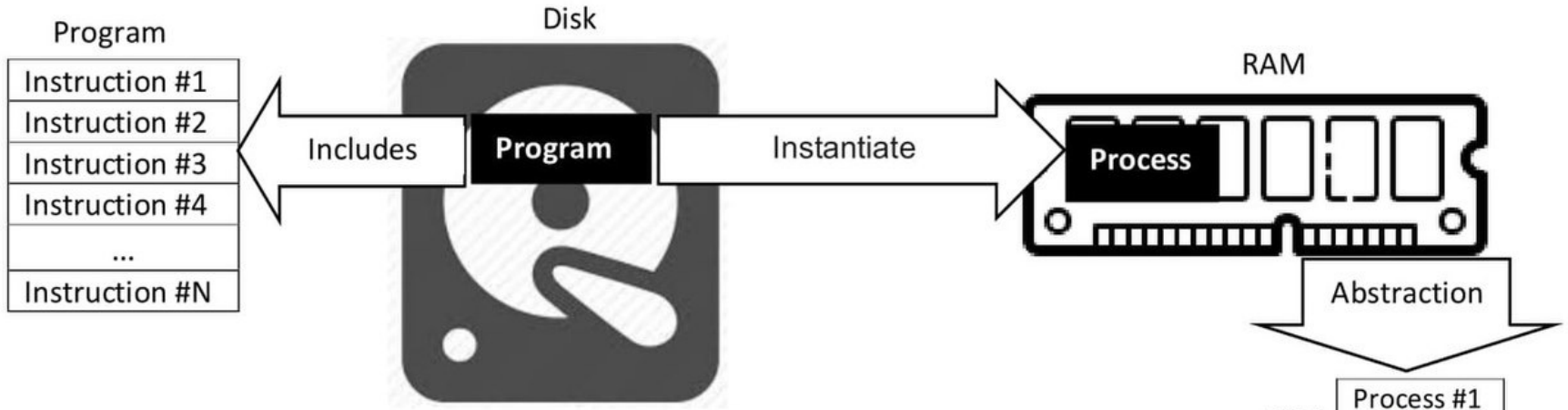


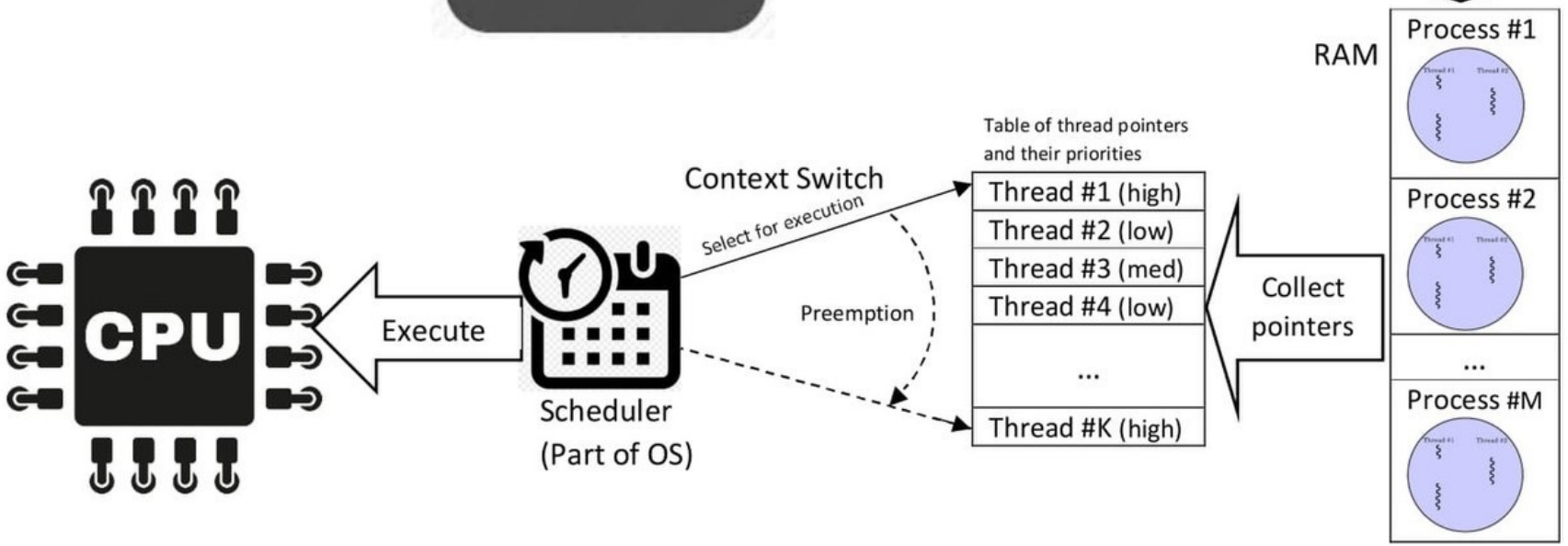
OpenMP™

Shared Memory



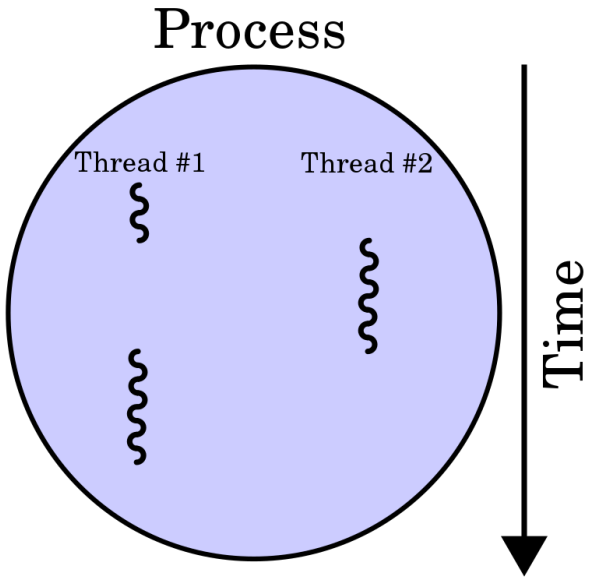
Processes, threads, and more



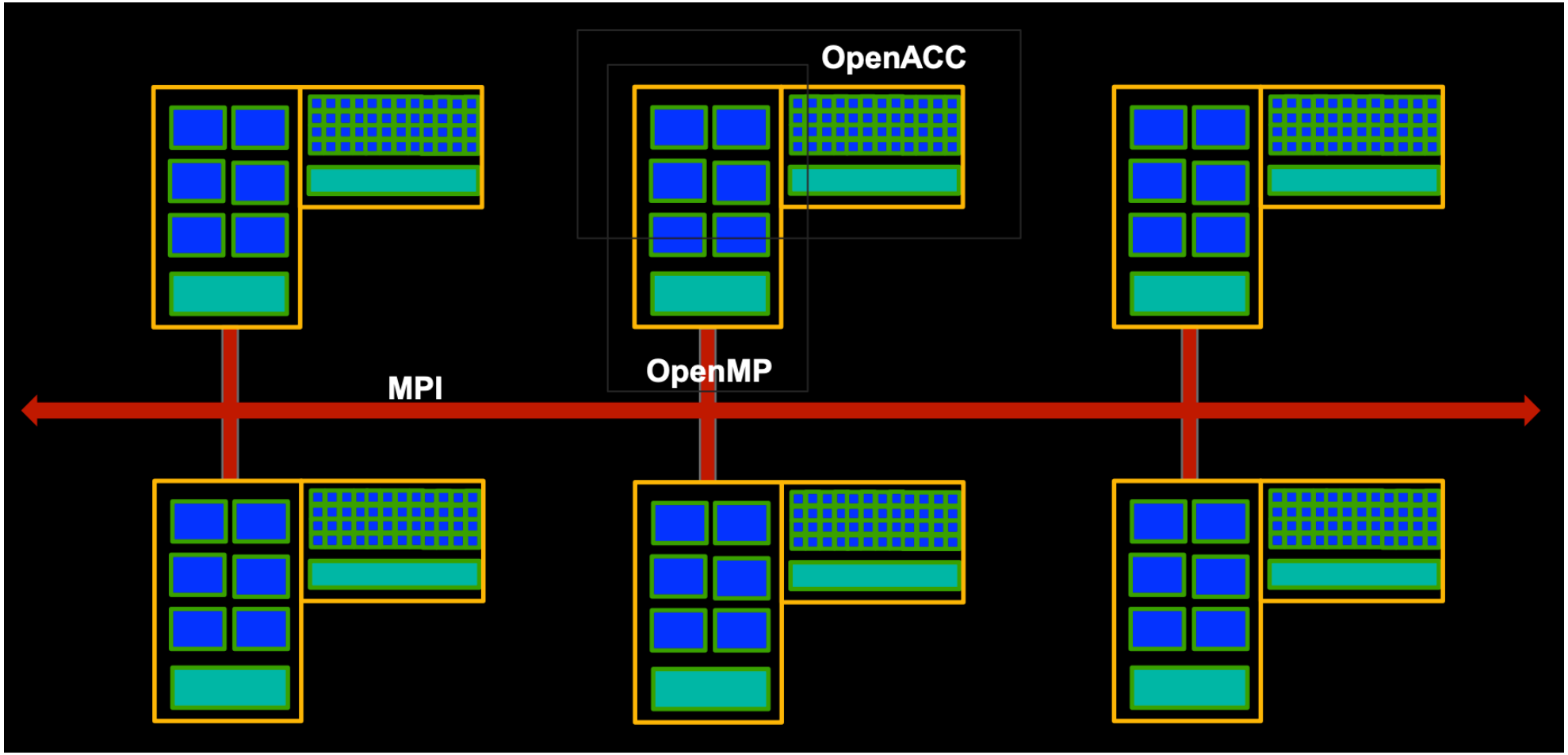


Thread: smallest sequence of instructions that the OS scheduler can manage

Process: composed of threads. Memory space allocated when program is loaded from disk.



Frameworks



Courtesy of John Urbanic @ PSC

Outline

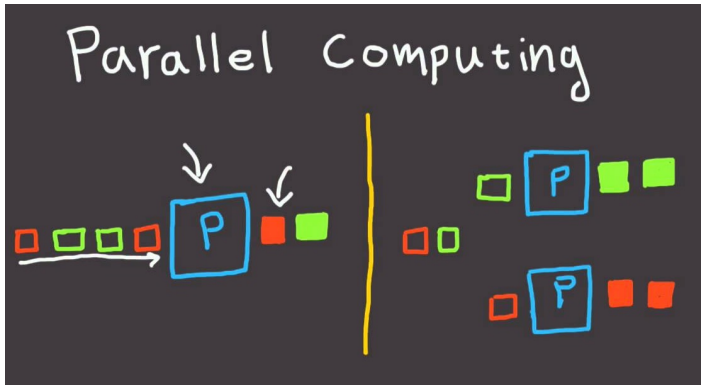
Part A: **Parallel Computing Quick Review** (10 min)

- Task and Data parallelism
- Parallel computers
- Threads, processes and more

Part B: **MPI Intro** (60 min)

- Core concepts
- Message passing
- Execution

MPI



Part C: **Exercises** (45+ min)

Ex 1: Hello World

Ex 2: Domain Decomposition

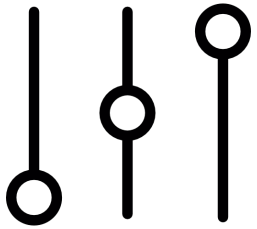
Ex 3: Parallel Linear Regression

What is MPI?

"Message Passing Interface (MPI) is a standardized and portable message-passing framework designed to function on parallel computing architectures."

https://www.mpi-forum.org/

- Library function/routines
- Different implementations



Application runtime behavior

- Established in the early 1990s by a group of researchers interested in standardizing message-passing
- Wide portability, applicable to multiple settings, independent of network speed or memory architecture
- Provides essential virtual topology, synchronization, and communication functionality between a set of processes in a language-independent way

MPI Pros & Cons

- Robust, mature, widely portable framework
- Fully scalable parallelism
- Allows for strong scaling, massive speedups
- Present in every supercomputing system and on the roadmap of the Exascale generation
- Integration with a variety of accelerators and their frameworks (MPI+X, e.g. CUDA)

- Initial learning barrier can be discouraging
- No incremental parallelism
- May require re-designing the entire application
- Local installation can be tricky. Several implementations available (open-source and vendor specific)
- Library routines are implemented either on C or Fortran only. Python binders are available.

When to (not) use MPI

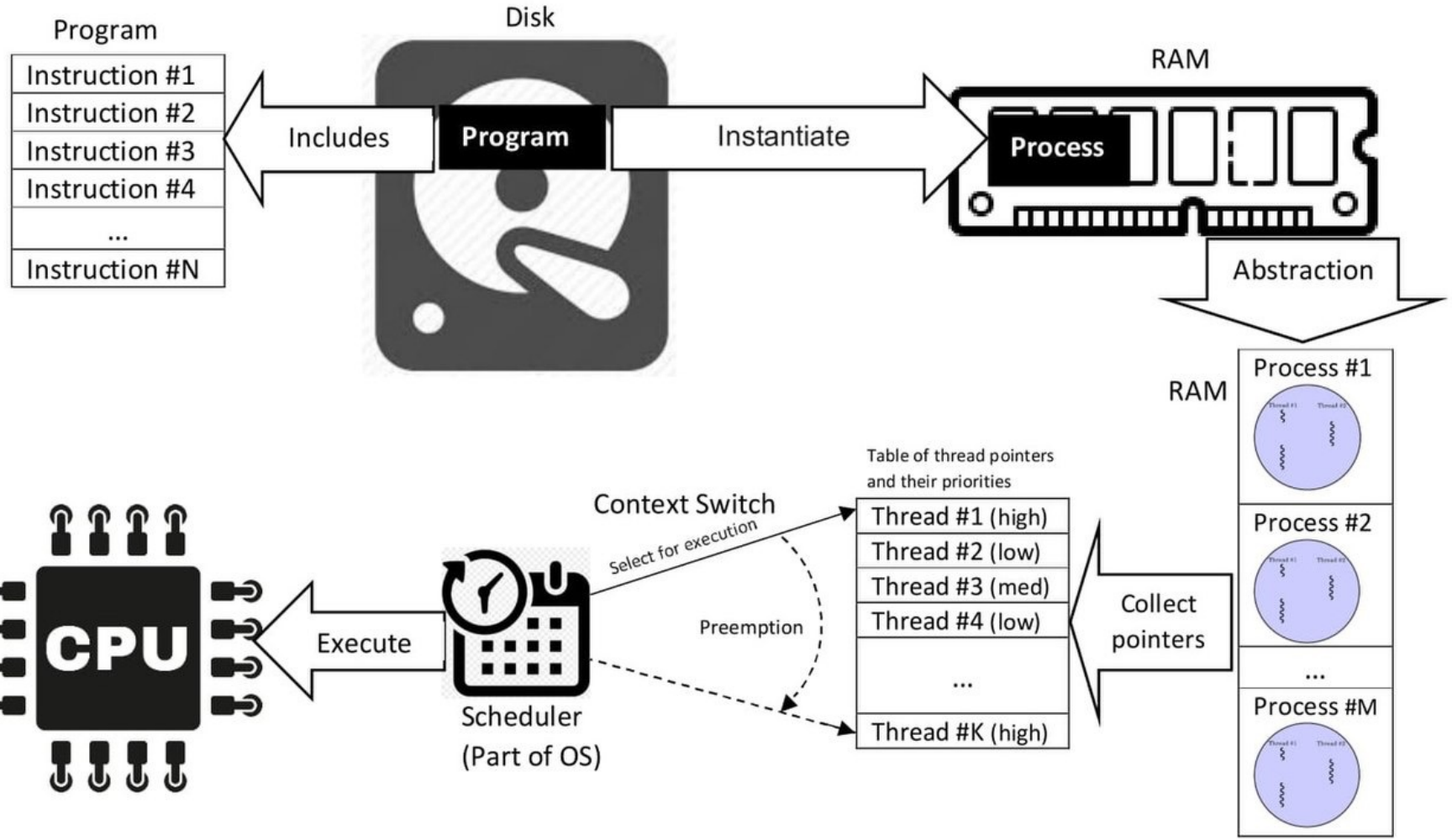
Parallel code needs to be portable across platforms

Large performance gain and/or strong scaling is required across the entire application

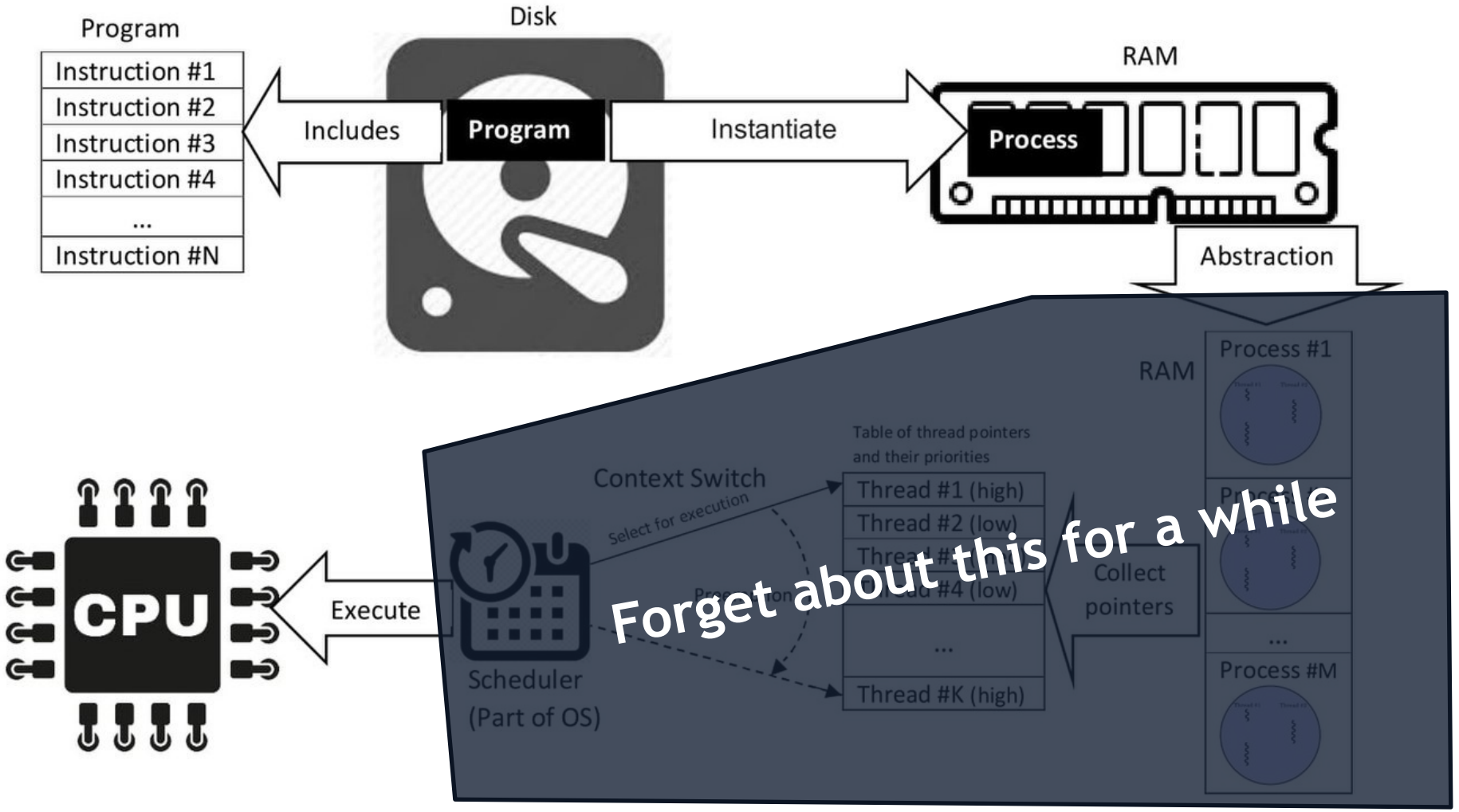
It's possible to achieve desired performance gain with only loop-level parallelism

Can use pre-existing parallel numerical libraries (e.g. ScaLAPACK)

How MPI works

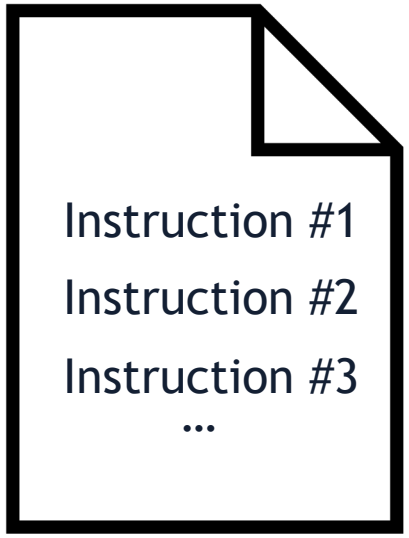


How MPI works



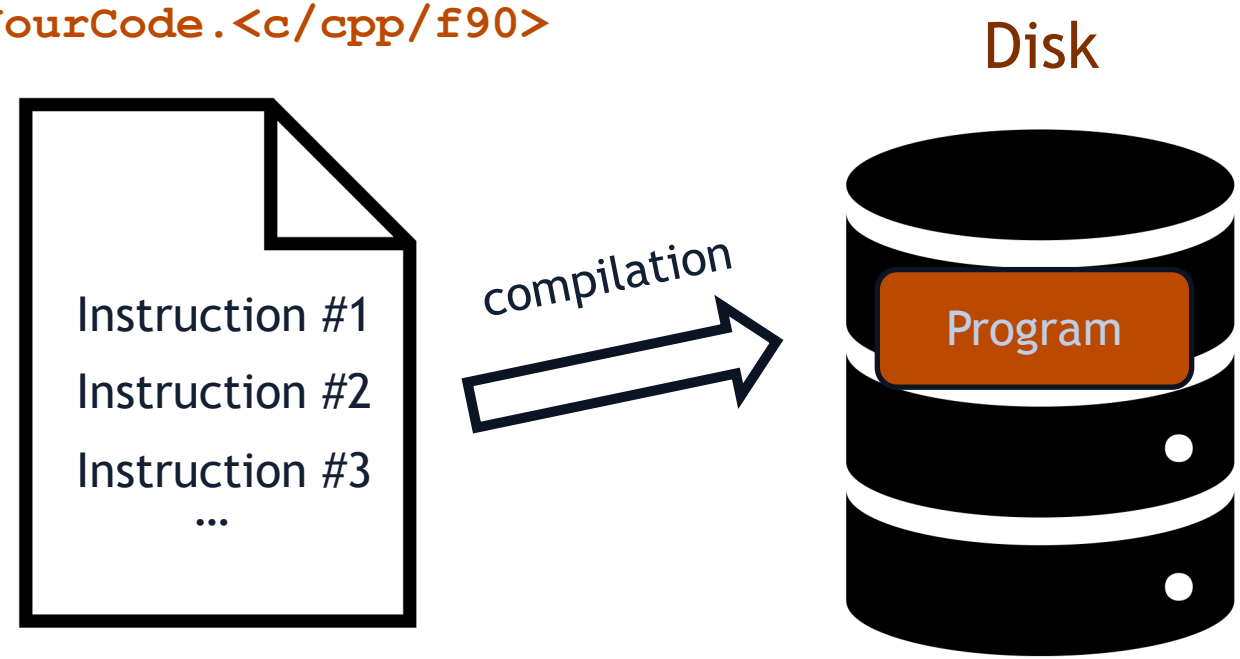
Serial code compilation and execution

`YourCode.<c/cpp/f90>`

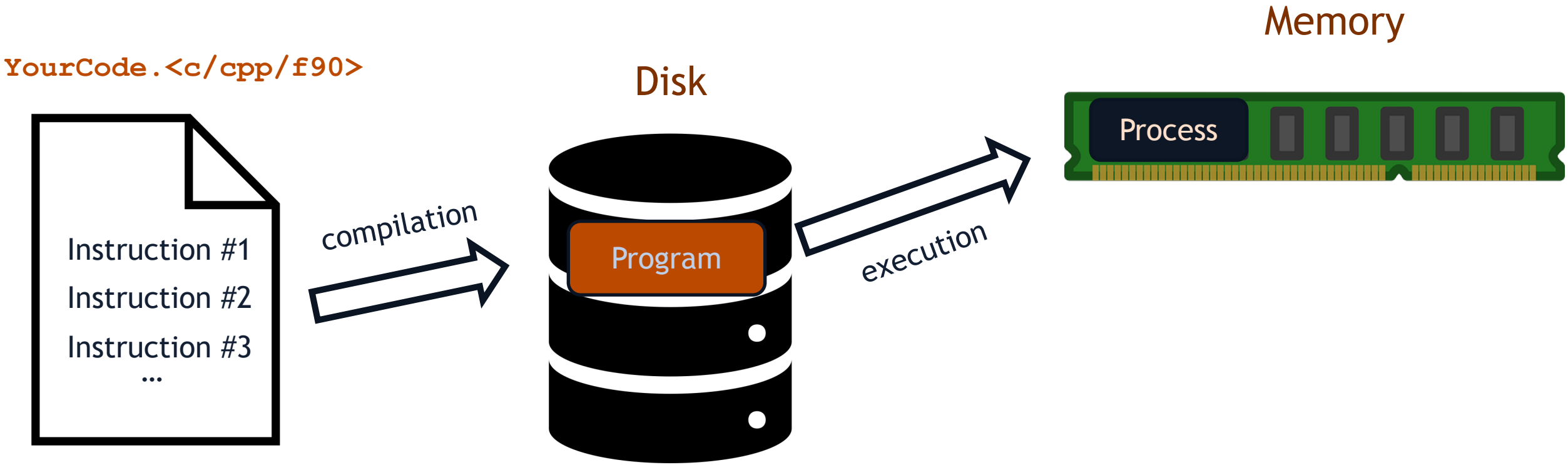


Serial code compilation and execution

`YourCode.<c/cpp/f90>`

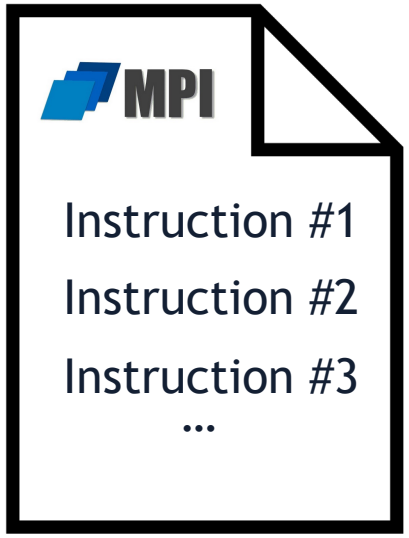


Serial code compilation and execution



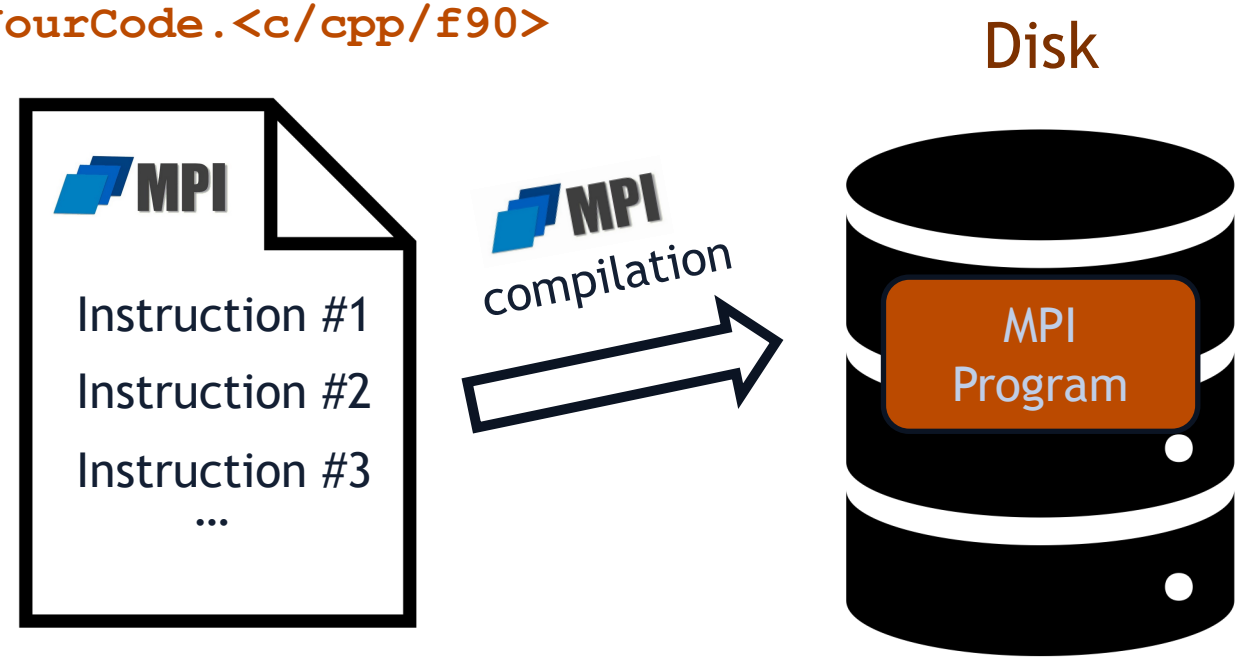
MPI code compilation and execution

`YourCode.<c/cpp/f90>`

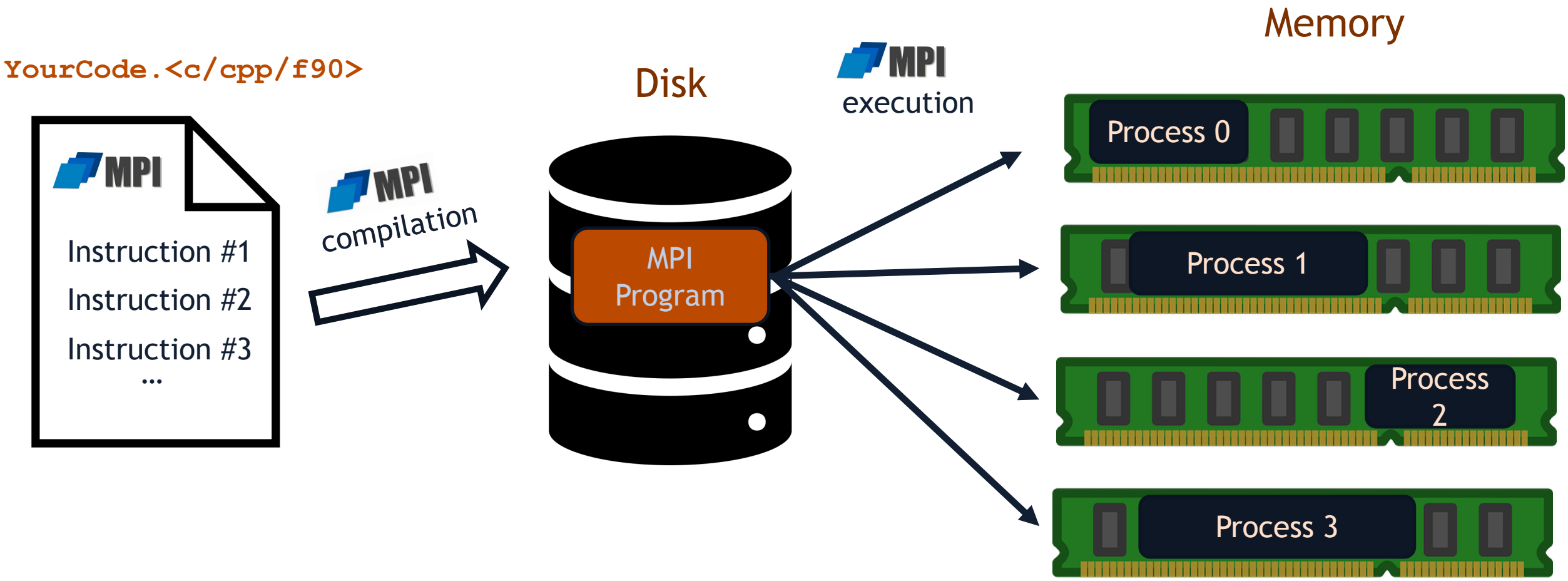


MPI code compilation and execution

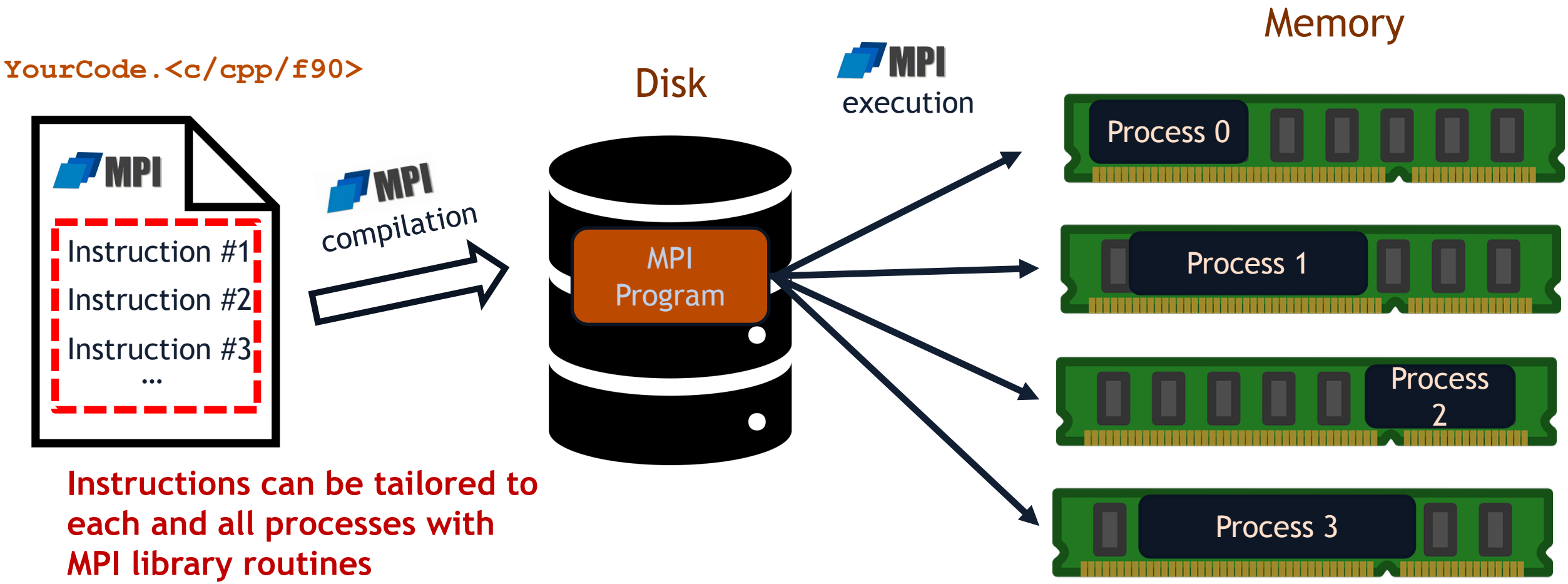
`YourCode.<c/cpp/f90>`



MPI code compilation and execution

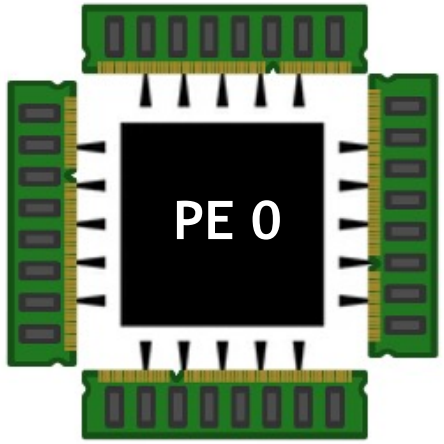


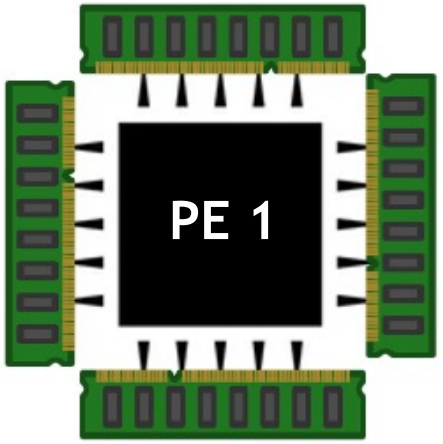
MPI code compilation and execution

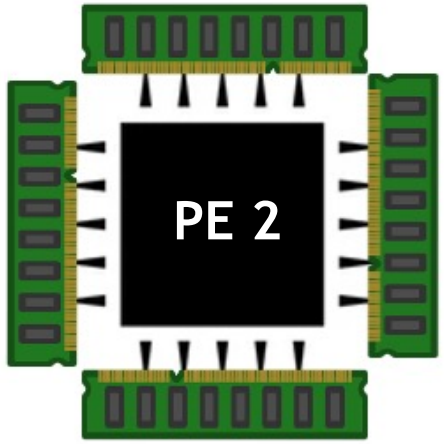


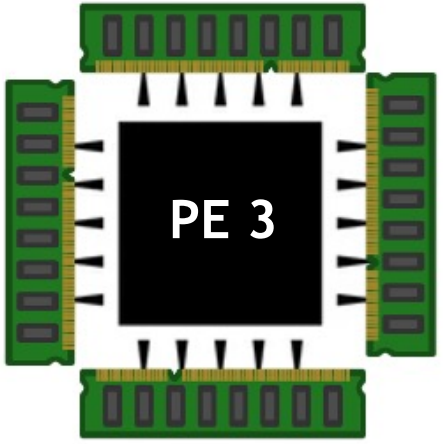
Instructions can be tailored to each and all processes with MPI library routines

MPI communication

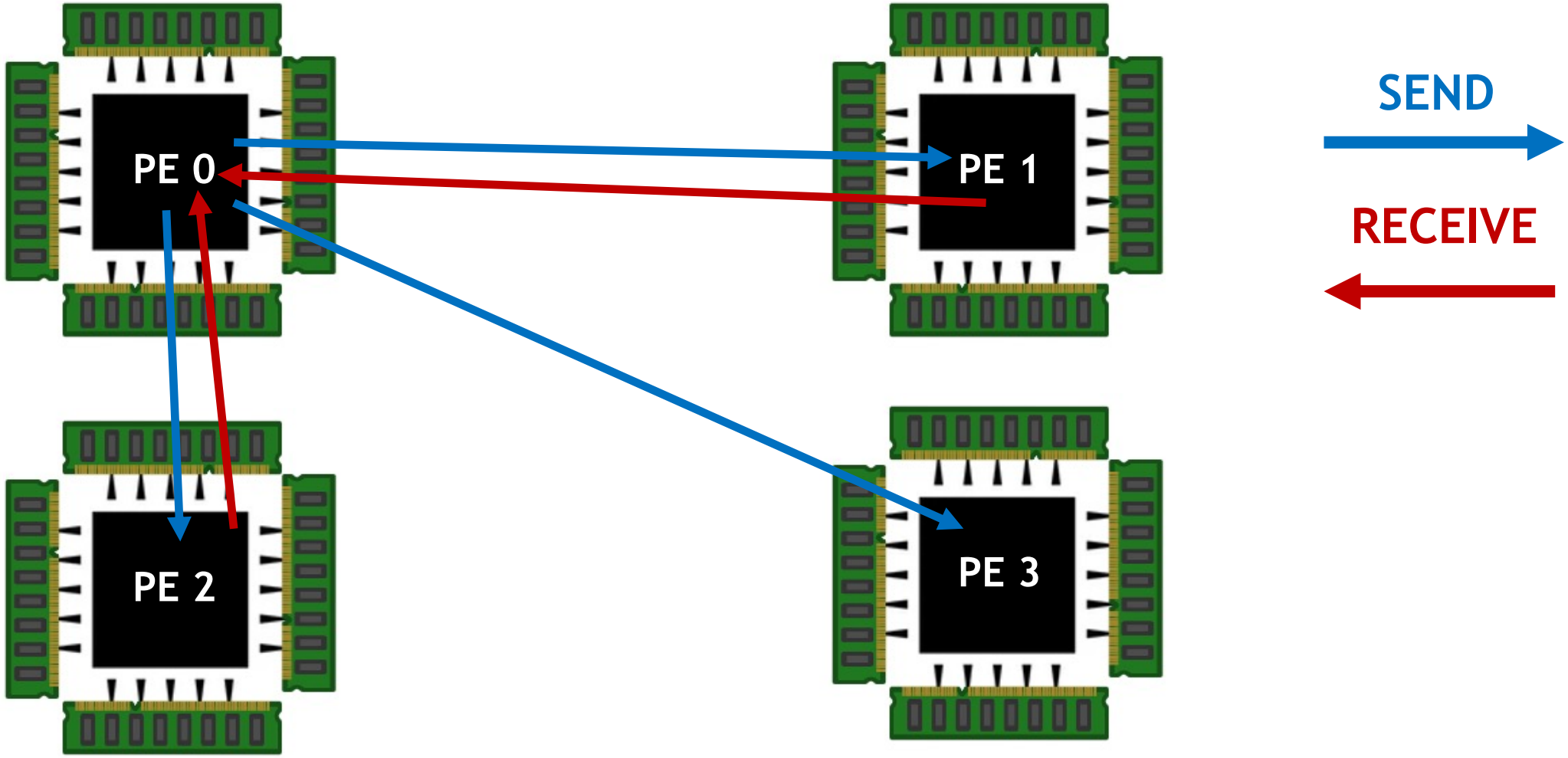




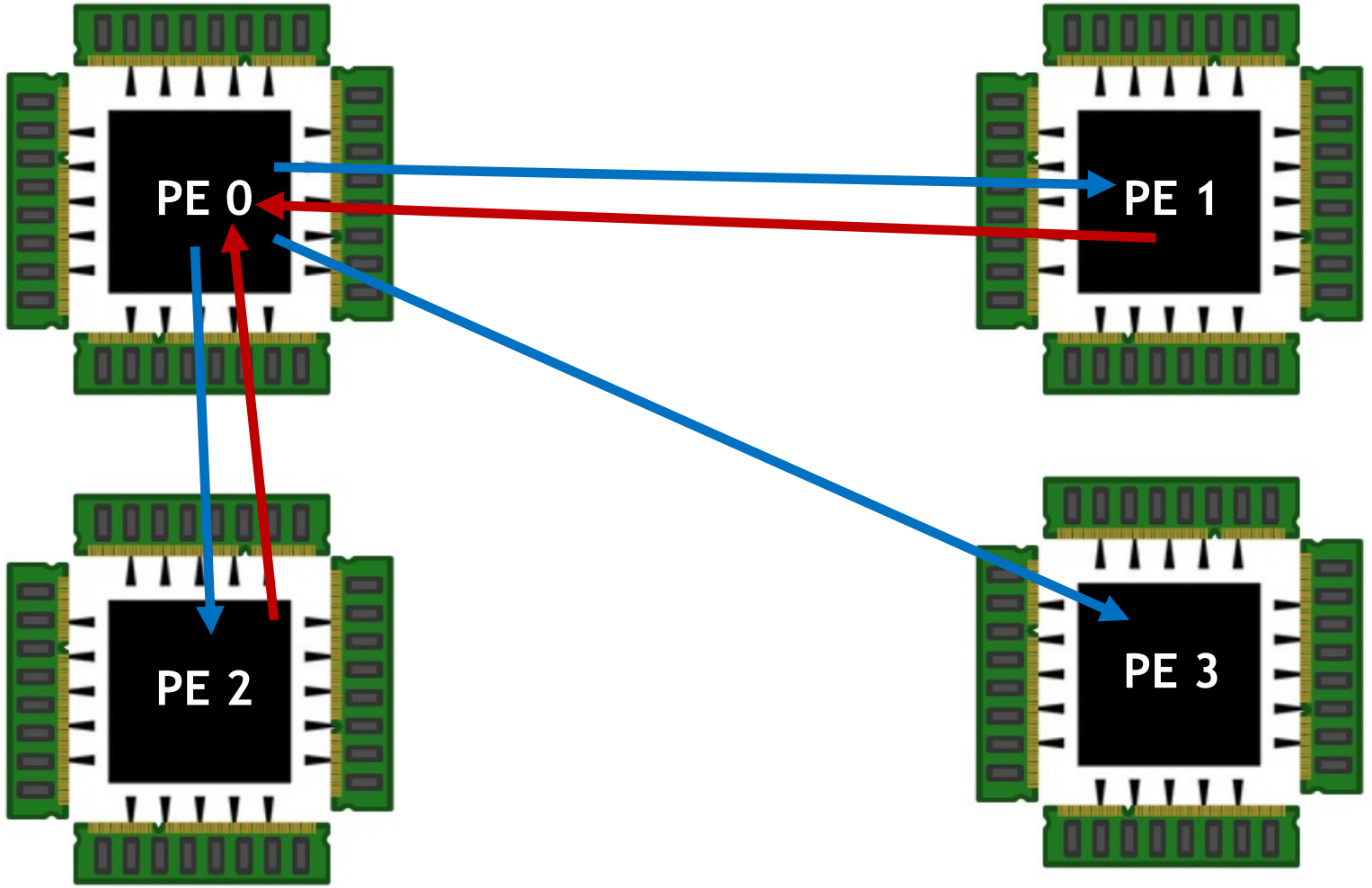




MPI communication



MPI communication



SEND

RECEIVE

MPI Messages

MPI messages

MPI messages

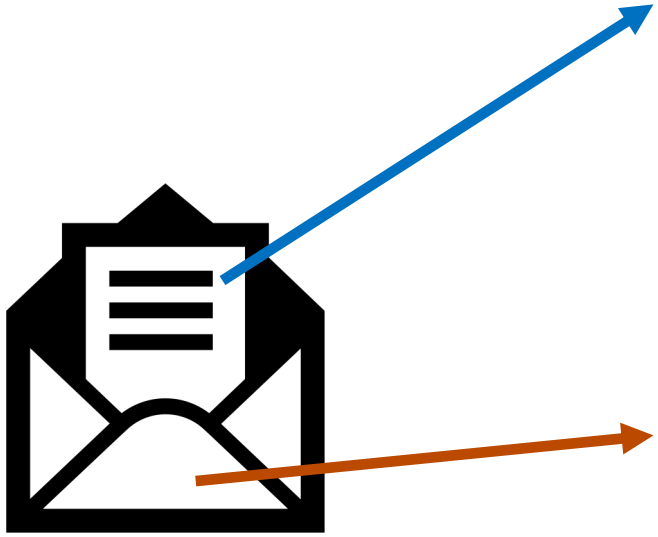
MPI messages

ENVELOPE

Destination address, return address, and information needed to ensure correct delivery

- Source
- Destination
- Communicator
- Tag

MPI messages



BODY

What is being transferred (payload, content)

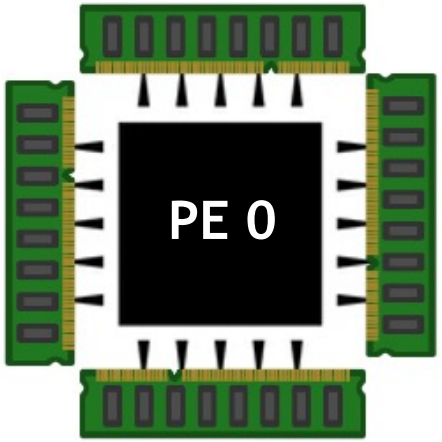
- Buffer (the data)
- Datatype (type of data in the message)
- Count (message size)

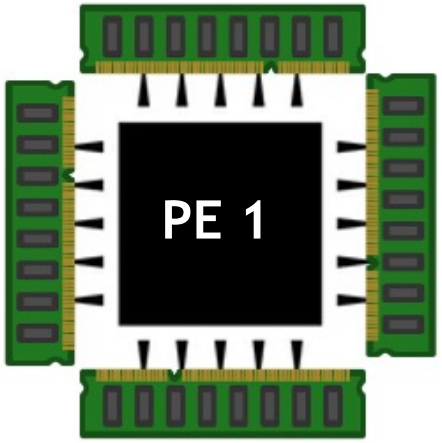
ENVELOPE

Destination address, return address, and information needed to ensure correct delivery

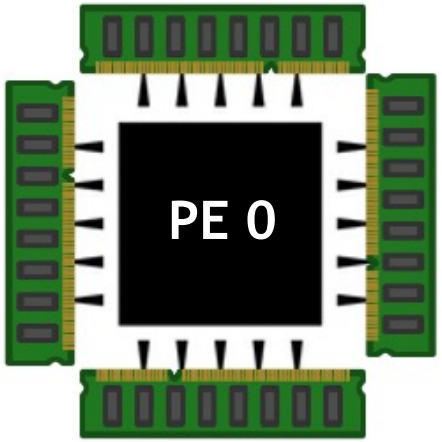
- Source
- Destination
- Communicator
- Tag

MPI communication protocol

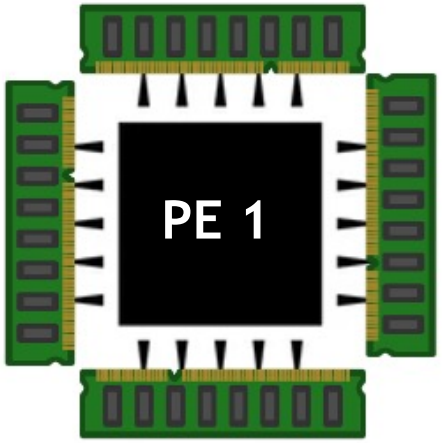




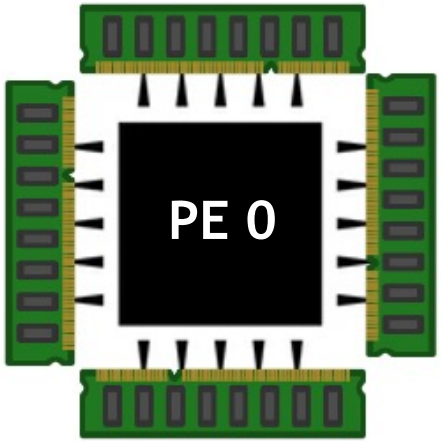
MPI communication protocol



MPI Buffer

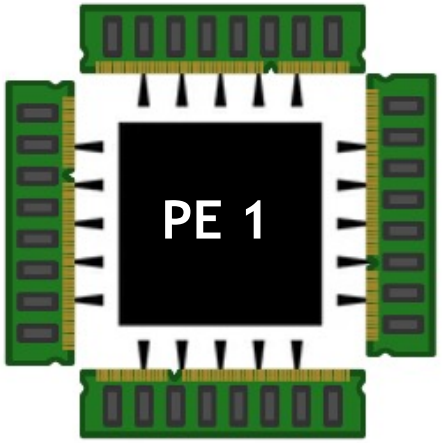


MPI communication protocol

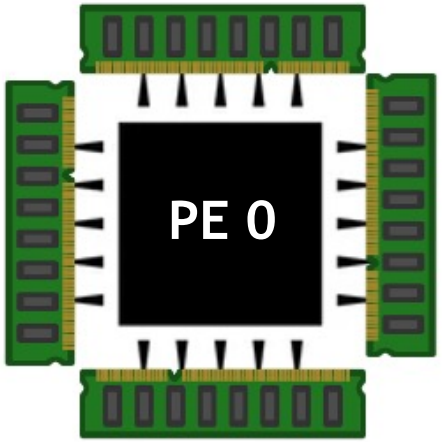


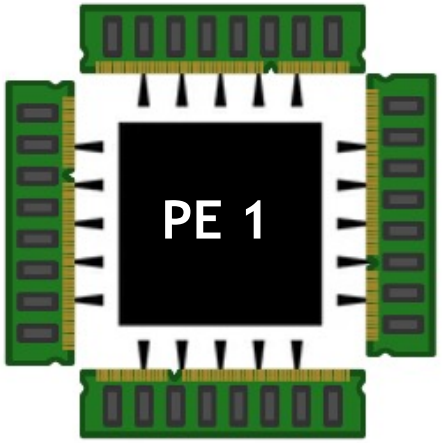
PE 0 writes
a message

MPI Buffer

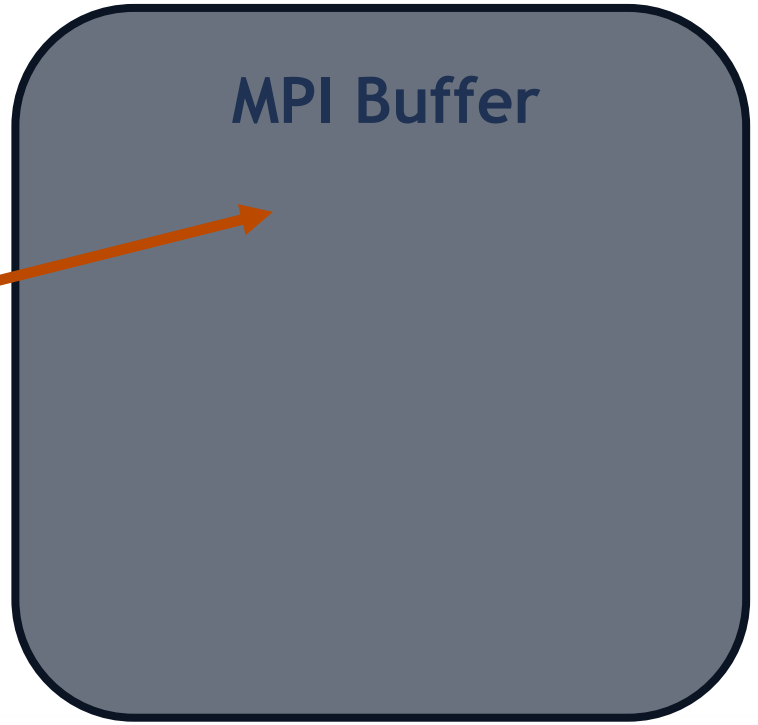


MPI communication protocol

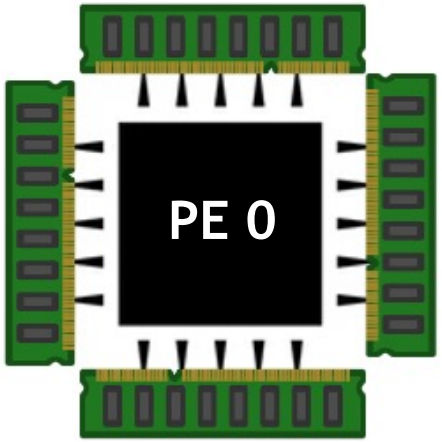


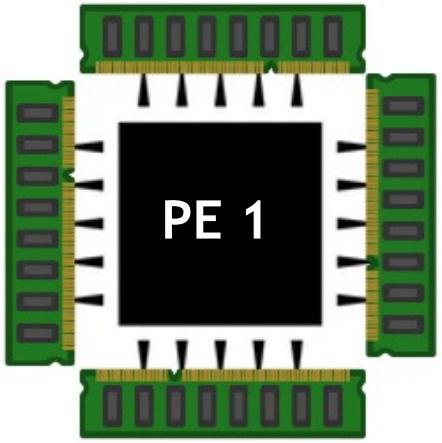


Sends it to the buffer



MPI communication protocol

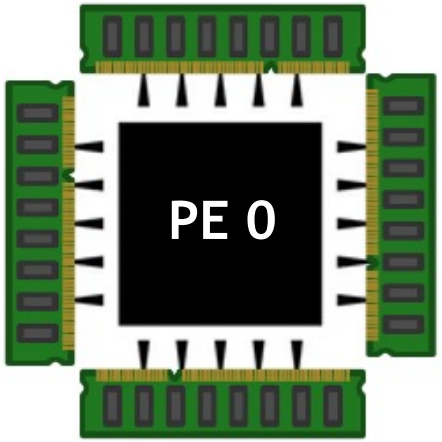


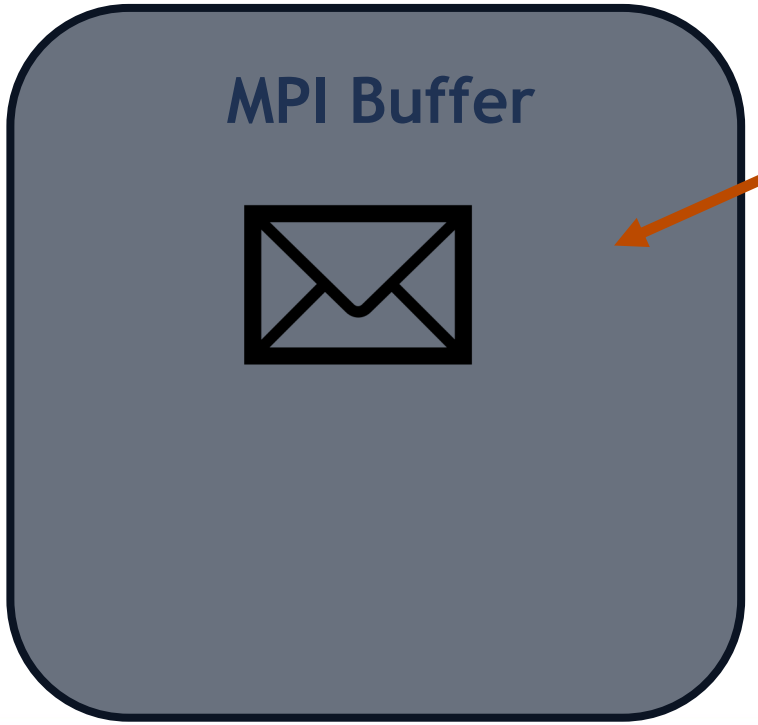


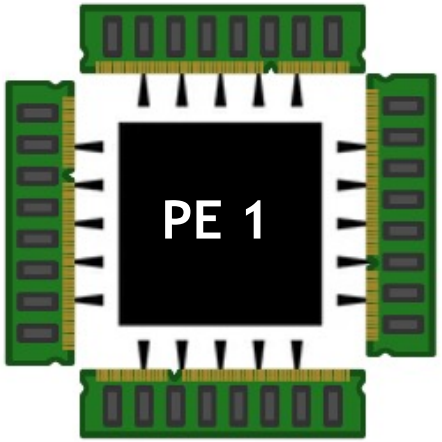
MPI Buffer

Buffer keeps the message until destination is ready to receive it

MPI communication protocol

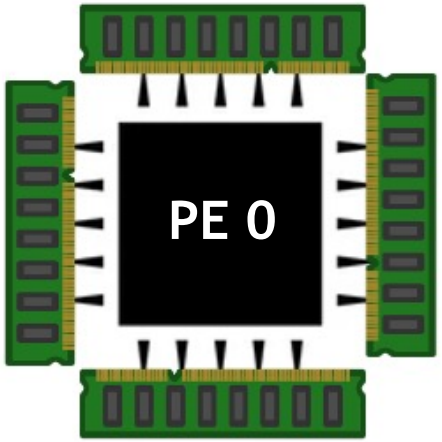




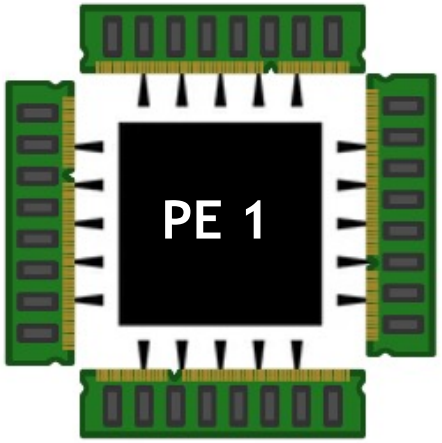


PE 1 checks the buffer

MPI communication protocol



MPI Buffer



PE 1 receives the message

MPI program structure

```cpp
#include <iostream>
#include <mpi.h>

using namespace std;

int main()
{
    int my_rank;
    int mpierr; // this will hold error codes for MPI calls

    // start MPI
    mpierr = MPI_Init(NULL, NULL);

    // get my rank
    mpierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // print message
    cout << "Hello from PE " << my_rank << endl;

    // terminate MPI
    mpierr = MPI_Finalize();

    // goodbye
    return 0;
}
```

MPI program structure

```cpp
#include <iostream>
#include <mpi.h>

using namespace std;

int main()
{
    int my_rank;
    int mpierr; // this will hold error codes for MPI calls

    // start MPI
    mpierr = MPI_Init(NULL, NULL);

    // get my rank
    mpierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // print message
    cout << "Hello from PE " << my_rank << endl;

    // terminate MPI
    mpierr = MPI_Finalize();

    // goodbye
    return 0;
}
```

1. Include MPI header file

MPI program structure

```cpp
#include <iostream>
#include <mpi.h>

using namespace std;

int main()
{
    int my_rank;
    int mpierr; // this will hold error codes for MPI calls

    // start MPI
    mpierr = MPI_Init(NULL, NULL);

    // get my rank
    mpierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // print message
    cout << "Hello from PE " << my_rank << endl;

    // terminate MPI
    mpierr = MPI_Finalize();

    // goodbye
    return 0;
}
```

1. Include MPI header file
2. Variable declarations

MPI program structure

```cpp
#include <iostream>
#include <mpi.h>

using namespace std;

int main()
{
    int my_rank;
    int mpierr; // this will hold error codes for MPI calls

    // start MPI
    mpierr = MPI_Init(NULL, NULL);

    // get my rank
    mpierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // print message
    cout << "Hello from PE " << my_rank << endl;

    // terminate MPI
    mpierr = MPI_Finalize();

    // goodbye
    return 0;
}
```

1. Include MPI header file
2. Variable declarations
3. Initialize MPI environment

MPI program structure

```cpp
#include <iostream>
#include <mpi.h>

using namespace std;

int main()
{
    int my_rank;
    int mpierr; // this will hold error codes for MPI calls

    // start MPI
    mpierr = MPI_Init(NULL, NULL);

    // get my rank
    mpierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // print message
    cout << "Hello from PE " << my_rank << endl;

    // terminate MPI
    mpierr = MPI_Finalize();

    // goodbye
    return 0;
}
```

1. Include MPI header file
2. Variable declarations
3. Initialize MPI environment
4. Do computation, MPI calls

MPI program structure

```cpp
#include <iostream>
#include <mpi.h>

using namespace std;

int main()
{
    int my_rank;
    int mpierr; // this will hold error codes for MPI calls

    // start MPI
    mpierr = MPI_Init(NULL, NULL);

    // get my rank
    mpierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // print message
    cout << "Hello from PE " << my_rank << endl;

    // terminate MPI
    mpierr = MPI_Finalize();

    // goodbye
    return 0;
}
```

1. Include MPI header file
2. Variable declarations
3. Initialize MPI environment
4. Do computation, MPI calls
5. Close MPI communications

MPI library

The MPI Library has a myriad of functions that provide complete control over

- Communication procedures
- Task and load distribution
- Reductions and more complex operations

https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

MPI library

The MPI Library has a myriad of functions that provide complete control over

- Communication procedures
- Task and load distribution
- Reductions and more complex operations

https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

MPI functions syntax

The good thing is that their syntax is quite standardized…

C/C++

Fortran

MPI functions syntax

The good thing is that their syntax is quite standardized…

C/C++	Fortran
`MPI_`	`MPI_`

- Indicates that an MPI function/routine is called

MPI functions syntax

The good thing is that their syntax is quite standardized…

C/C++	Fortran
`MPI_FunctionName`	`MPI_FUNCTIONNAME`

- Indicates that an MPI function/routine is called
- Typically a description of what the function does

MPI functions syntax

The good thing is that their syntax is quite standardized…

C/C++	Fortran
`MPI_FunctionName(...)`	`MPI_FUNCTIONNAME(...)`

- Indicates that an MPI function/routine is called
- Typically a description of what the function does
- Arguments to be passed and collected

MPI functions syntax

The good thing is that their syntax is quite standardized…

C/C++	Fortran
`MPI_FunctionName( BODY , ENVELOPE)`	`MPI_FUNCTIONNAME( BODY , ENVELOPE)`

- Indicates that an MPI function/routine is called
- Typically a description of what the function does
- Arguments to be passed and collected
  - Envelope
  - Body

MPI call: Send and Receive

C/C++

```
MPI_FunctionName( BODY , ENVELOPE )

MPI_FunctionName( BODY , ENVELOPE )
```

Fortran

```
MPI_FUNCTIONNAME( BODY , ENVELOPE )

MPI_FUNCTIONNAME( BODY , ENVELOPE )
```

MPI call: Send and Receive

C/C++

```
MPI_Send( BODY , ENVELOPE)

MPI_Recv( BODY , ENVELOPE)
```

Fortran

```
MPI_SEND( BODY , ENVELOPE )

MPI_RECV( BODY , ENVELOPE )
```

MPI call: Send and Receive

C/C++

```
MPI_Send( BODY , ENVELOPE)

MPI_Recv( BODY , ENVELOPE)
```

Fortran

```
MPI_SEND( BODY , ENVELOPE )

MPI_RECV( BODY , ENVELOPE )
```

BODY

- Buffer
- Count
- Data type

ENVELOPE

- Source
- Destination
- Tag
- Communicator

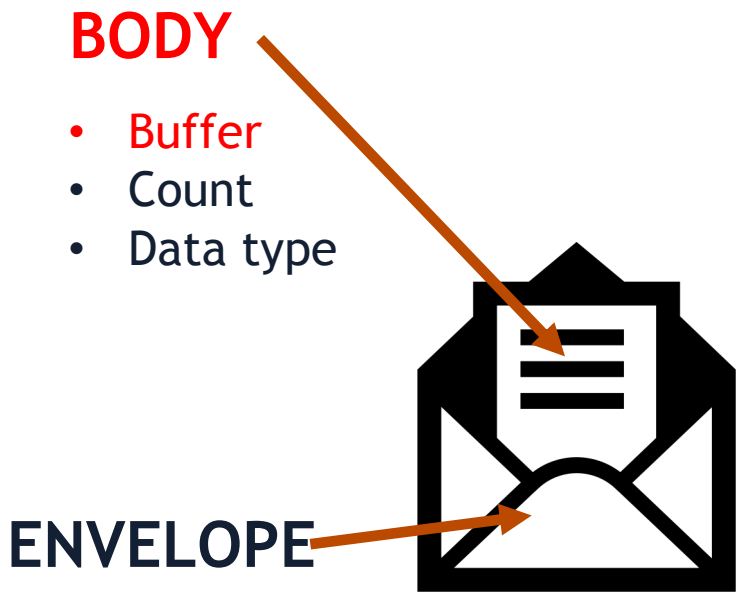
MPI call: Send and Receive

C/C++

```
MPI_Send(void *buf,    )

MPI_Recv(void *buf,    )
```

Fortran

```
MPI_SEND(BUF,    )

MPI_RECV(BUF,    )
```



- Source
- Destination
- Tag
- Communicator

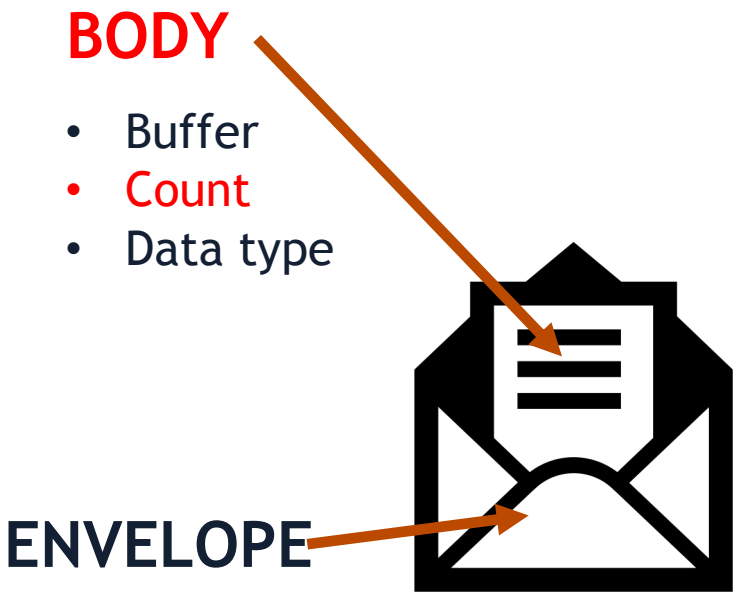
MPI call: Send and Receive

C/C++

```
MPI_Send(void *buf, int count,    )

MPI_Recv(void *buf, int count,    )
```

Fortran

```
MPI_SEND(BUF, COUNT,    )

MPI_RECV(BUF, COUNT,    )
```



- Source
- Destination
- Tag
- Communicator

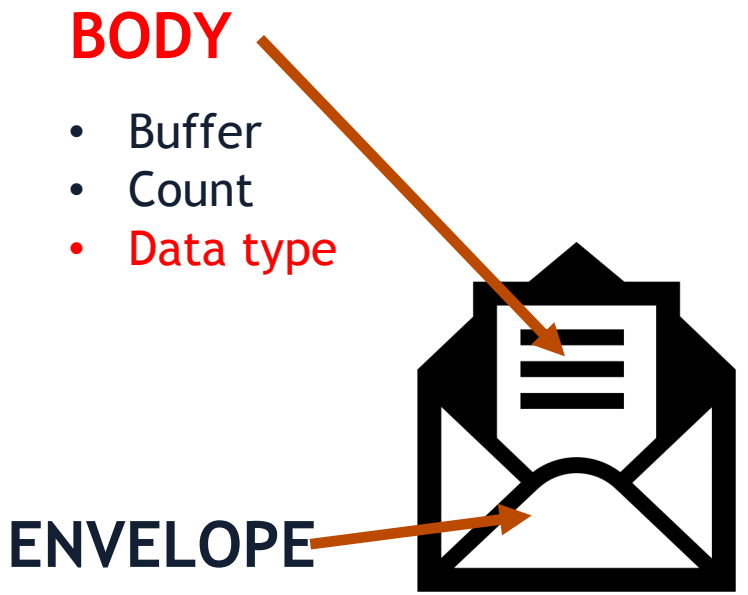
MPI call: Send and Receive

C/C++

```
MPI_Send(void *buf, int count, MPI_Datatype dtype,   )

MPI_Recv(void *buf, int count, MPI_Datatype dtype,   )
```

Fortran

```
MPI_SEND(BUF, COUNT, DTYPE,   )

MPI_RECV(BUF, COUNT, DTYPE,   )
```



- Source
- Destination
- Tag
- Communicator

MPI call: Send and Receive

C/C++

```
MPI_Send(void *buf, int count, MPI_Datatype dtype, )

MPI_Recv(void *buf, int count, MPI_Datatype dtype, int src,)
```

Fortran

```
MPI_SEND(BUF, COUNT, DTYPE,    )

MPI_RECV(BUF, COUNT, DTYPE, SOURCE,   )
```

BODY

- Buffer
- Count
- Data type

ENVELOPE

- Source
- Destination
- Tag
- Communicator

MPI call: Send and Receive

C/C++

```
MPI_Send(void *buf, int count, MPI_Datatype dtype, int dst,)

MPI_Recv(void *buf, int count, MPI_Datatype dtype, int src,)
```

Fortran

```
MPI_SEND(BUF, COUNT, DTYPE, DEST,  )

MPI_RECV(BUF, COUNT, DTYPE, SOURCE,  )
```

BODY

- Buffer
- Count
- Data type

ENVELOPE

- Source
- Destination
- Tag
- Communicator

MPI call: Send and Receive

C/C++

```
MPI_Send(void *buf, int count, MPI_Datatype dtype, int dst,
int tag,   )
MPI_Recv(void *buf, int count, MPI_Datatype dtype, int src,
int tag,   )
```

Fortran

```
MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG,  )

MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG,  )
```

BODY

- Buffer
- Count
- Data type

ENVELOPE

- Source
- Destination
- Tag
- Communicator

MPI call: Send and Receive

C/C++

```
MPI_Send(void *buf, int count, MPI_Datatype dtype, int dst,
int tag, MPI_Comm comm);

MPI_Recv(void *buf, int count, MPI_Datatype dtype, int src,
int tag, MPI_Comm comm,   )
```

Fortran

```
MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG, COMM,  )

MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM,  )
```

BODY

- Buffer
- Count
- Data type

ENVELOPE

- Source
- Destination
- Tag
- Communicator

MPI call: Send and Receive

C/C++

```
MPI_Send(void *buf, int count, MPI_Datatype dtype, int dst,
int tag, MPI_Comm comm);
MPI_Recv(void *buf, int count, MPI_Datatype dtype, int src,
int tag, MPI_Comm comm, MPI_Status *status);
```

Status about the message that was received

Fortran

```
MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG, COMM,  )

MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM, STATUS,   )
```

BODY

- Buffer
- Count
- Data type

ENVELOPE

- Source
- Destination
- Tag
- Communicator

MPI call: Send and Receive

C/C++

```
int ierr = MPI_Send(void *buf, int count, MPI_Datatype
dtype, int dst, int tag, MPI_Comm comm);
int ierr = MPI_Recv(void *buf, int count, MPI_Datatype
dtype, int src, int tag, MPI_Comm comm, MPI_Status *status);
```

Error code when routine/function returns

Fortran

```
MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, IERR)

MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM, STATUS, IERR)
```

BODY

- Buffer
- Count
- Data type

ENVELOPE

- Source
- Destination
- Tag
- Communicator

MPI call: Send and Receive

C/C++

```
int ierr = MPI_Send(void *buf, int count, MPI_Datatype
dtype, int dst, int tag, MPI_Comm comm);

int ierr = MPI_Recv(void *buf, int count, MPI_Datatype
dtype, int src, int tag, MPI_Comm comm, MPI_Status *status);
```

Fortran

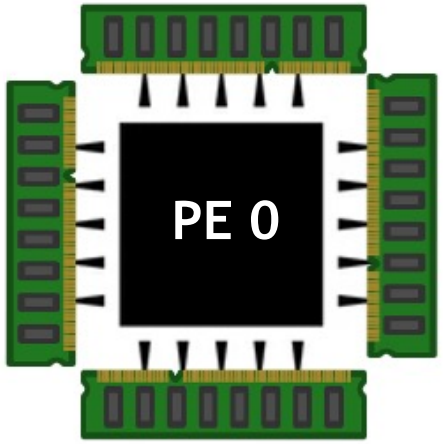
```
MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, IERR)

MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM, STATUS, IERR)
```

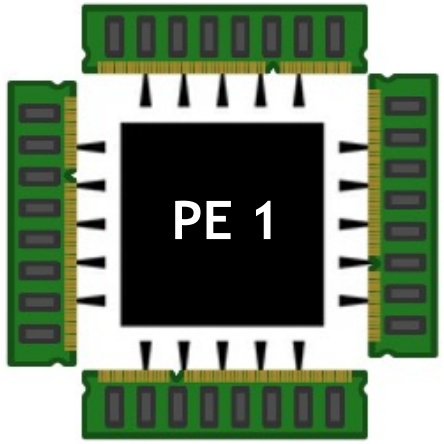
MPI collectives

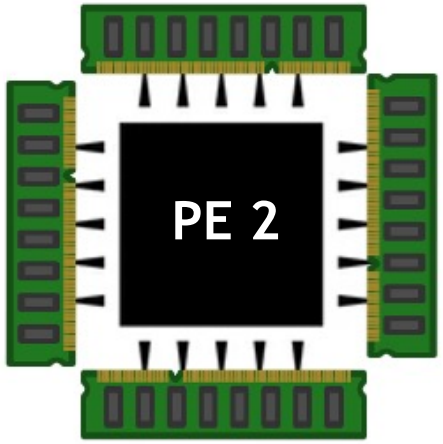
Collective communications are a great way to get started with MPI. They are:

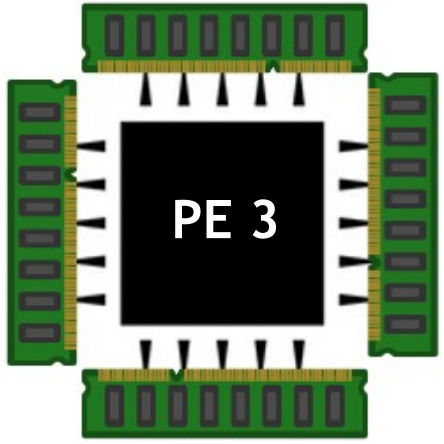
- Simple to understand
- Robust in execution
- Enclose a lot of functionality that solves a lot of common problems
- Avoid deadlocks (the beginner's nightmare)

MPI collectives

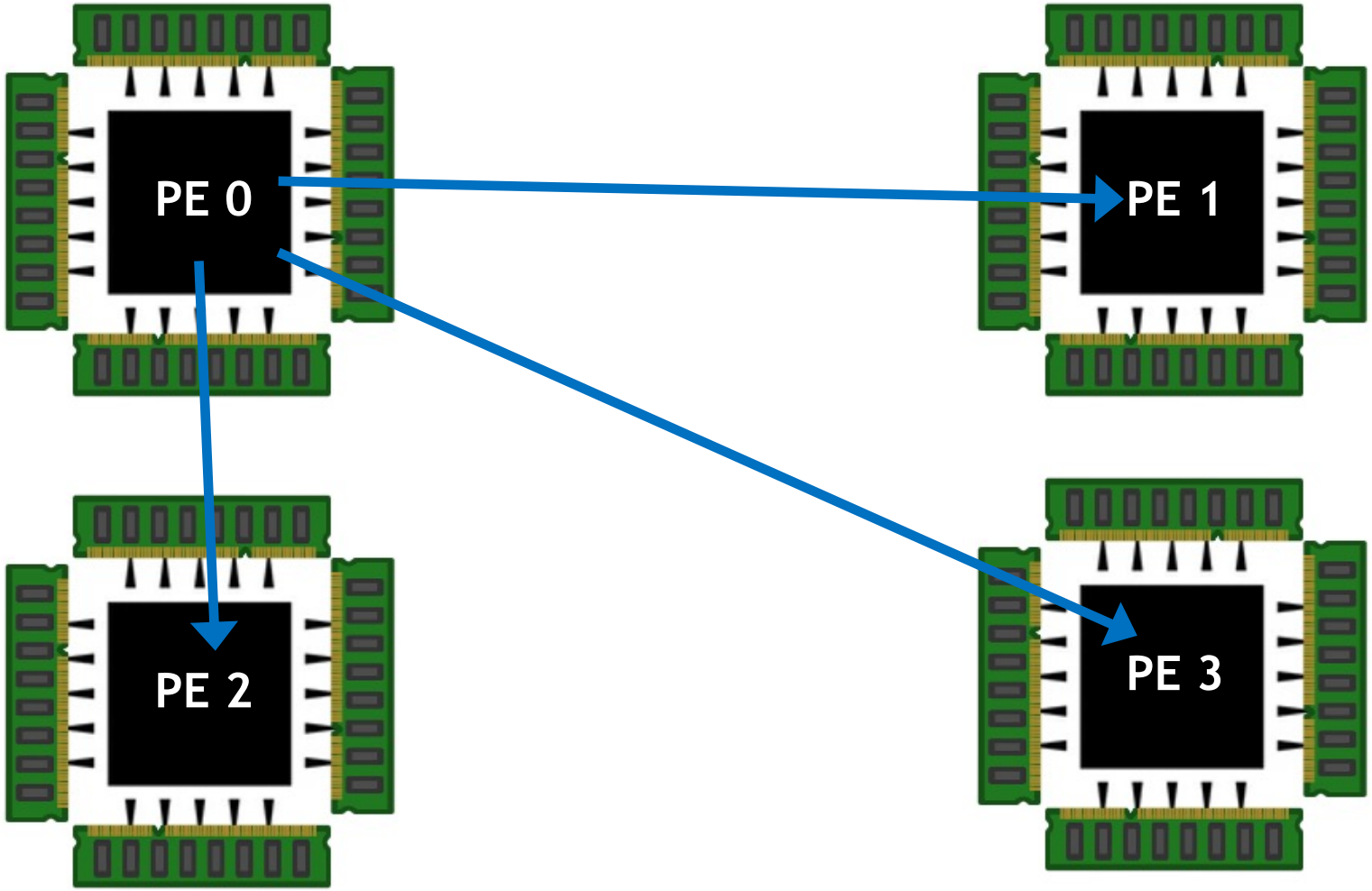




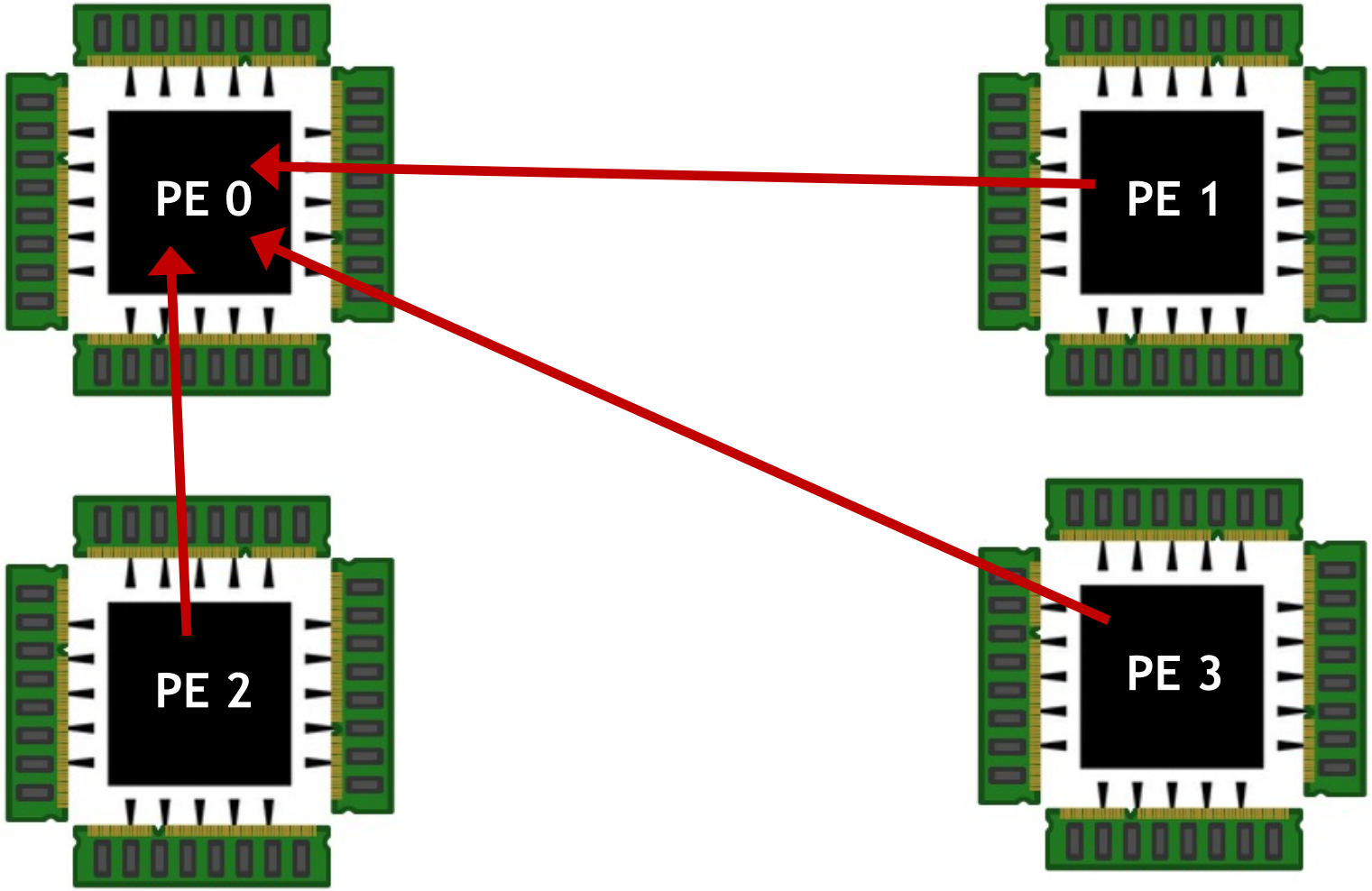




MPI collectives: one-to-all



MPI collectives: all-to-one



MPI Broadcast

Copies data from the memory of the root process to the same memory locations for the other processes within the communicator

MPI Broadcast

Copies data from the memory of the root process to the same memory locations for the other processes within the communicator

C/C++

```
int ierr = MPI_Bcast(  );
```

Fortran

```
MPI_BCAST( IERR )
```

MPI Broadcast

Copies data from the memory of the root process to the same memory locations for the other processes within the communicator

C/C++

```
int ierr = MPI_Bcast(void *send_buf,  );
```

Starting address of the data that is being sent

Fortran

```
MPI_BCAST(SEND_BUF, IERR)
```

MPI Broadcast

Copies data from the memory of the root process to the same memory locations for the other processes within the communicator

C/C++

```
int ierr = MPI_Bcast(void *send_buf, int send_count,  );
```

Number of elements to be sent

Fortran

```
MPI_BCAST(SEND_BUF, SEND_COUNT, IERR)
```

MPI Broadcast

Copies data from the memory of the root process to the same memory locations for the other processes within the communicator

C/C++

```
int ierr = MPI_Bcast(void *send_buf, int send_count, MPI_Datatype dtype, );
```

The data type of the elements being sent

Fortran

```
MPI_BCAST(SEND_BUF, SEND_COUNT, DTYPE, IERR)
```

MPI Broadcast

Copies data from the memory of the root process to the same memory locations for the other processes within the communicator

C/C++

```
int ierr = MPI_Bcast(void *send_buf, int send_count, MPI_Datatype dtype,
       int root_id, );
```

The ID of the root process

Fortran

```
MPI_BCAST(SEND_BUF, SEND_COUNT, DTYPE, ROOT_ID, IERR)
```

MPI Broadcast

Copies data from the memory of the root process to the same memory locations for the other processes within the communicator

C/C++

```
int ierr = MPI_Bcast(void *send_buf, int send_count, MPI_Datatype dtype,
       int root_id, MPI_Comm comm);
```

The communicator (group of processes)

Fortran

```
MPI_BCAST(SEND_BUF, SEND_COUNT, DTYPE, ROOT_ID, COMM, IERR)
```

MPI Broadcast

Copies data from the memory of the root process to the same memory locations for the other processes within the communicator

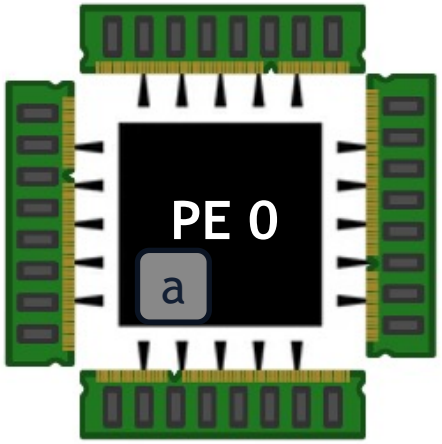
C/C++

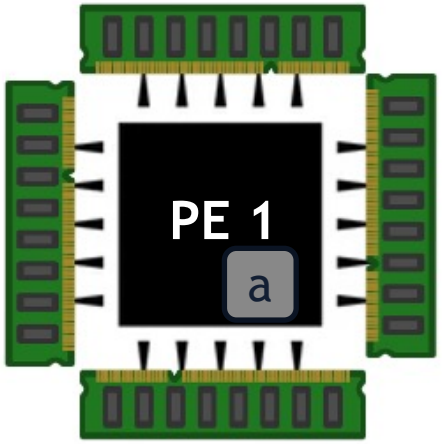
```
int ierr = MPI_Bcast(void *send_buf, int send_count, MPI_Datatype dtype,
       int root_id, MPI_Comm comm);
```

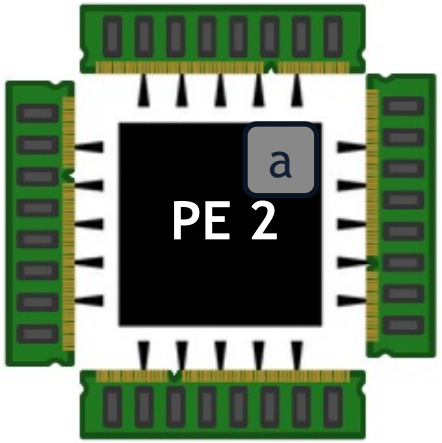
Fortran

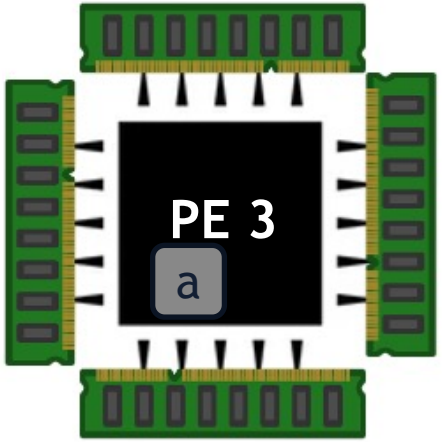
```
MPI_BCAST(SEND_BUF, SEND_COUNT, DTYPE, ROOT_ID, COMM, IERR)
```

MPI Broadcast

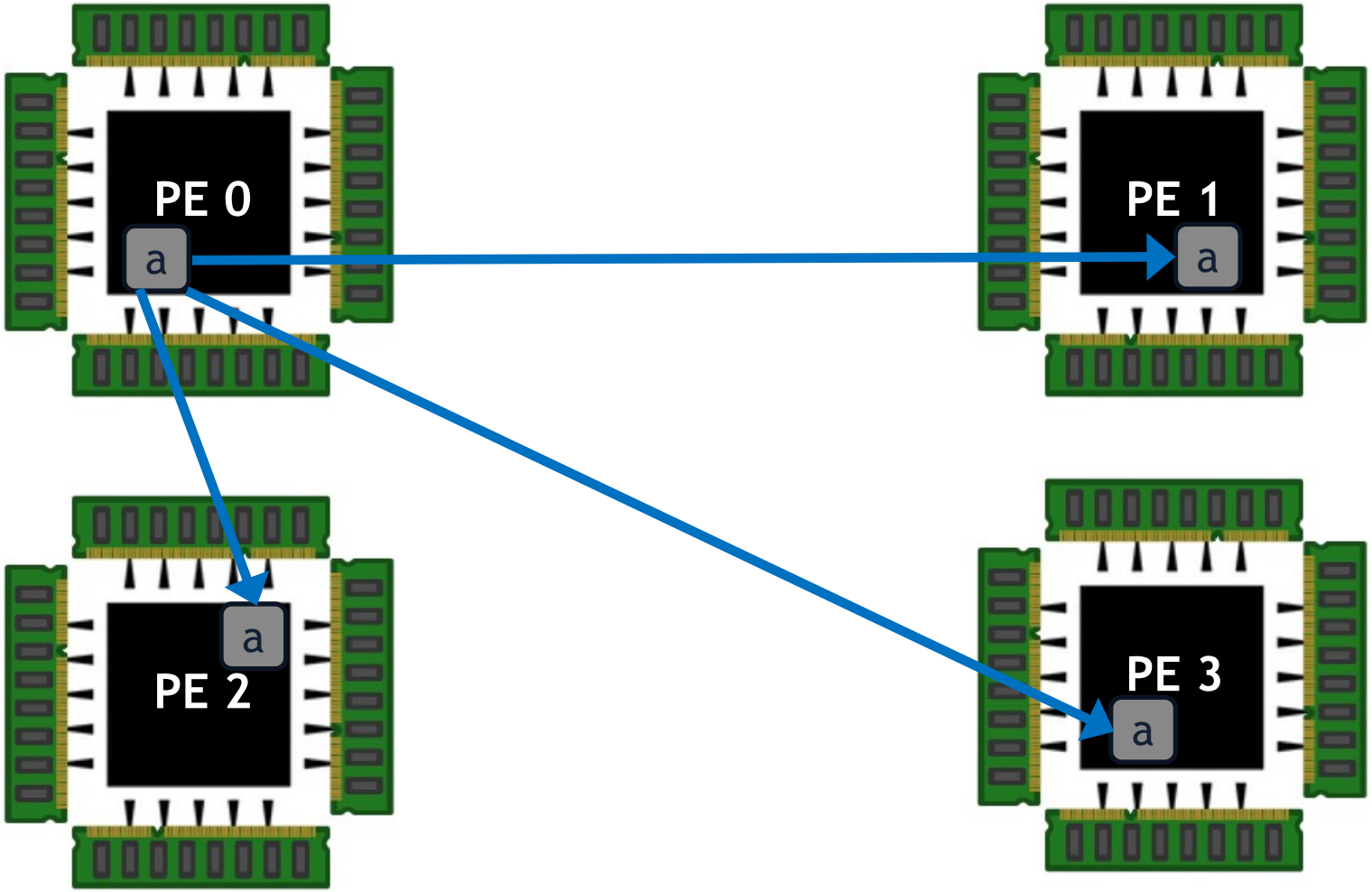








MPI Broadcast



MPI Reduce

Collects data from each process, reduces it to a single value via a specified operation, stores the reduced result on the root process

MPI Reduce

Collects data from each process, reduces it to a single value via a specified operation, stores the reduced result on the root process

C/C++

```
int ierr = MPI_Reduce(  );
```

Fortran

```
MPI_REDUCE( IERR )
```

MPI Reduce

Collects data from each process, reduces it to a single value via a specified operation, stores the reduced result on the root process

C/C++

```
int ierr = MPI_Reduce(void *send_buf,   );
```

Starting address of the data that is being sent

Fortran

```
MPI_REDUCE(SEND_BUF, IERR)
```

MPI Reduce

Collects data from each process, reduces it to a single value via a specified operation, stores the reduced result on the root process

C/C++

```
int ierr = MPI_Reduce(void *send_buf, void *recv_buf, );
```

Address of where the resulting operation is going to be sent to

Fortran

```
MPI_REDUCE(SEND_BUF, RECV_BUF, IERR)
```

MPI Reduce

Collects data from each process, reduces it to a single value via a specified operation, stores the reduced result on the root process

C/C++

```
int ierr = MPI_Reduce(void *send_buf, void *recv_buf, int count, );
```

Number of elements in the send buffer

Fortran

```
MPI_REDUCE(SEND_BUF, RECV_BUF, COUNT, IERR)
```

MPI Reduce

Collects data from each process, reduces it to a single value via a specified operation, stores the reduced result on the root process

C/C++

```
int ierr = MPI_Reduce(void *send_buf, void *recv_buf, int count,
      MPI_Datatype dtype, );
```

Data type of elements in the send buffer

Fortran

```
MPI_REDUCE(SEND_BUF, RECV_BUF, COUNT, DTYPE, IERR)
```

MPI Reduce

Collects data from each process, reduces it to a single value via a specified operation, stores the reduced result on the root process

C/C++

```
int ierr = MPI_Reduce(void *send_buf, void *recv_buf, int count,
      MPI_Datatype dtype, MPI_Op op, );
```

The reduction operation to be performed

Fortran

```
MPI_REDUCE(SEND_BUF, RECV_BUF, COUNT, DTYPE, OP, IERR)
```

MPI Reduce

Collects data from each process, reduces it to a single value via a specified operation, stores the reduced result on the root process

C/C++

```
int ierr = MPI_Reduce(void *send_buf, void *recv_buf, int count,
      MPI_Datatype dtype, MPI_Op op, int root_id, );
```

The ID of the root process

Fortran

```
MPI_REDUCE(SEND_BUF, RECV_BUF, COUNT, DTYPE, OP, ROOT_ID, IERR)
```

MPI Reduce

Collects data from each process, reduces it to a single value via a specified operation, stores the reduced result on the root process

C/C++

```
int ierr = MPI_Reduce(void *send_buf, void *recv_buf, int count,
      MPI_Datatype dtype, MPI_Op op, int root_id, MPI_Comm comm);
```

The communicator (group of processes)

Fortran

```
MPI_REDUCE(SEND_BUF, RECV_BUF, COUNT, DTYPE, OP, ROOT_ID, COMM, IERR)
```

MPI Reduce

Collects data from each process, reduces it to a single value via a specified operation, stores the reduced result on the root process

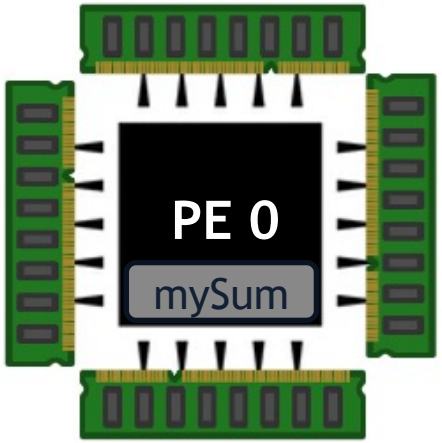
C/C++

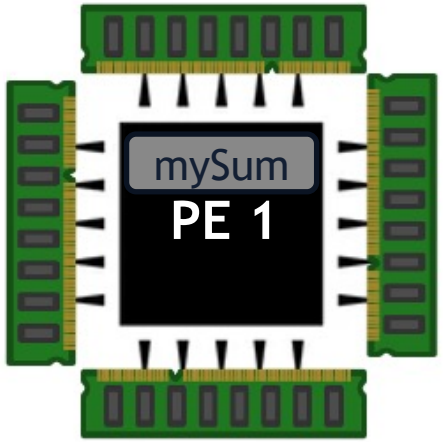
```
int ierr = MPI_Reduce(void *send_buf, void *recv_buf, int count,
      MPI_Datatype dtype, MPI_Op op, int root_id, MPI_Comm comm);
```

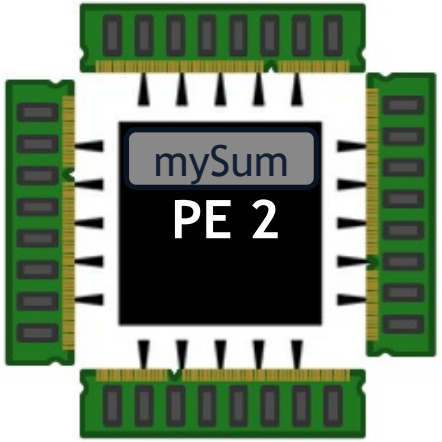
Fortran

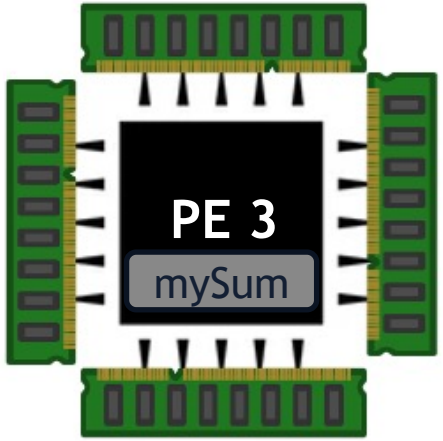
```
MPI_REDUCE(SEND_BUF, RECV_BUF, COUNT, DTYPE, OP, ROOT_ID, COMM, IERR)
```

MPI Reduce

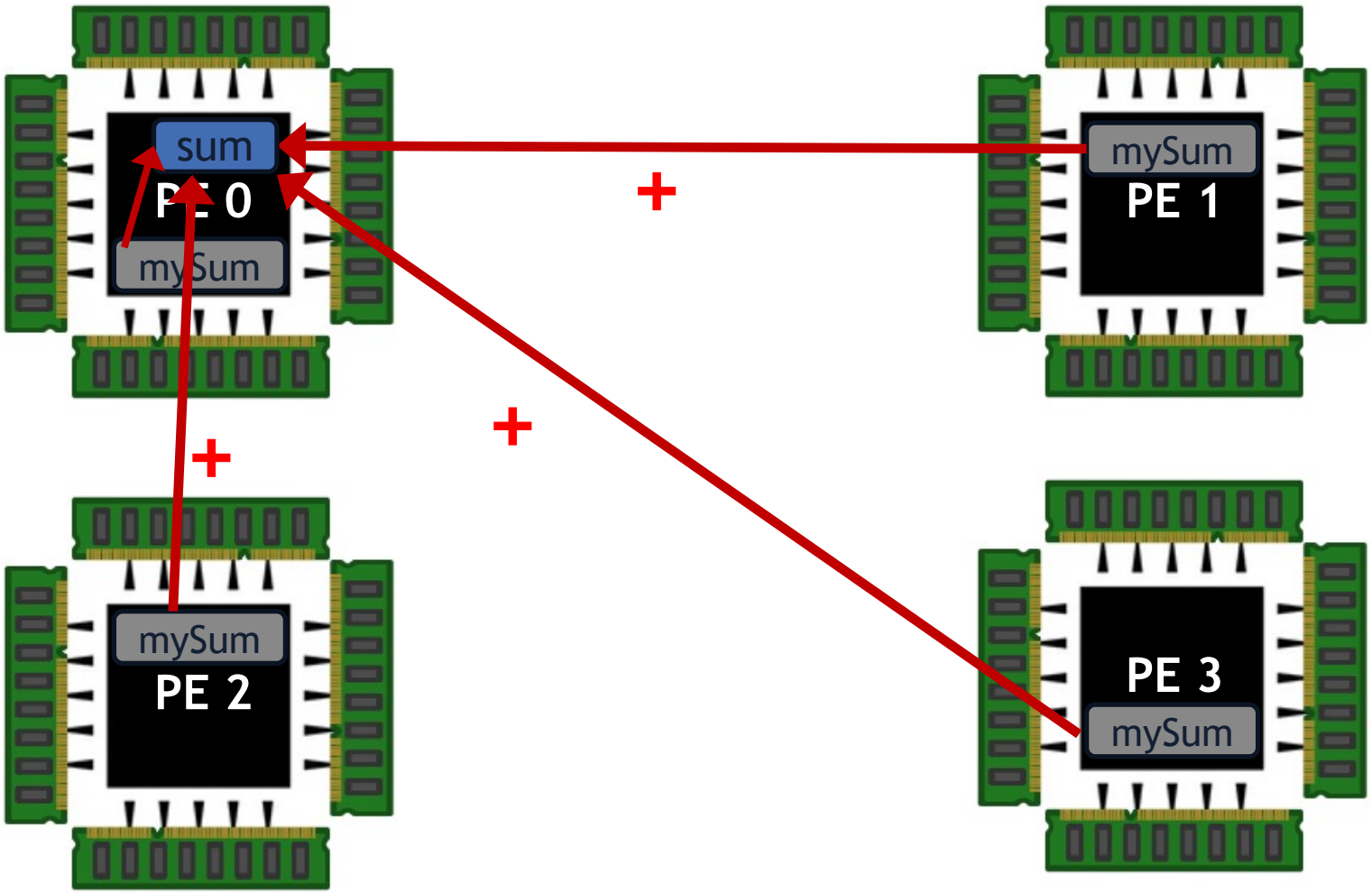








MPI Reduce



MPI library: Datatypes

https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf Section 3.2.2

C/C++

MPI datatype	C datatype
MPI_CHAR	`char` (treated as printable character)
MPI_SHORT	`signed short int`
MPI_INT	`signed int`
MPI_LONG	`signed long int`
MPI_LONG_LONG_INT	`signed long long int`
MPI_LONG_LONG (as a synonym)	`signed long long int`
MPI_SIGNED_CHAR	`signed char` (treated as integral value)
MPI_UNSIGNED_CHAR	`unsigned char` (treated as integral value)
MPI_UNSIGNED_SHORT	`unsigned short int`
MPI_UNSIGNED	`unsigned int`
MPI_UNSIGNED_LONG	`unsigned long int`
MPI_UNSIGNED_LONG_LONG	`unsigned long long int`
MPI_FLOAT	`float`
MPI_DOUBLE	`double`
MPI_LONG_DOUBLE	`long double`
MPI_WCHAR	`wchar_t` (defined in `<stddef.h>`) (treated as printable character)

Fortran

MPI datatype	Fortran datatype
MPI_INTEGER	`INTEGER`
MPI_REAL	`REAL`
MPI_DOUBLE_PRECISION	`DOUBLE PRECISION`
MPI_COMPLEX	`COMPLEX`
MPI_LOGICAL	`LOGICAL`
MPI_CHARACTER	`CHARACTER(1)`
MPI_BYTE	
MPI_PACKED	

MPI library: Datatypes

https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf Section 3.2.2

C/C++

MPI datatype	C datatype
MPI_CHAR	char (treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG (as a synonym)	signed long long int
MPI_SIGNED_CHAR	signed char (treated as integral value)
MPI_UNSIGNED_CHAR	
MPI_UNSIGNED_SHORT	
MPI_UNSIGNED	
MPI_UNSIGNED_LONG	
MPI_UNSIGNED_LONG_LONG	
MPI_FLOAT	
MPI_DOUBLE	
MPI_LONG_DOUBLE	
MPI_WCHAR	
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	
MPI_PACKED	

Fortran

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

MPI library: reduce operations

https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf Section 6.9.1

Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or (xor)
MPI_BXOR	bit-wise exclusive or (xor)
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

MPI library: communicators

`MPI_Comm` **`MPI_COMM_WORLD`**

Every process in your running application

You can construct custom communicators if you wish…

https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf Chapter 7

MPI library: ubiquitous functions

These are the ones that you will **always** use in an MPI program

C/C++

Fortran

MPI library: ubiquitous functions

These are the ones that you will **always** use in an MPI program

C/C++

```
int ierr = MPI_Init(&argc, &argv)
```

Fortran

```
MPI_INIT(IERR)
```

Initializes the MPI environment

MPI library: ubiquitous functions

These are the ones that you will **always** use in an MPI program

C/C++

```
int ierr = MPI_Init(&argc, &argv)
```

```
int ierr = MPI_Finalize()
```

Fortran

```
MPI_INIT(IERR)
```

```
MPI_FINALIZE(IERR)
```

Terminates the MPI environment

MPI library: ubiquitous functions

These are the ones that you will **always** use in an MPI program

C/C++

```
int ierr = MPI_Init(&argc, &argv)

int ierr = MPI_Finalize()

int ierr = MPI_Comm_size(MPI_Comm comm,
    &nprocs)
```

Fortran

```
MPI_INIT(IERR)

MPI_FINALIZE(IERR)

MPI_COMM_SIZE(COMM, NPROCS, IERR)
```

Gives the number of processes
in the communicator

MPI library: ubiquitous functions

These are the ones that you will **always** use in an MPI program

C/C++

```
int ierr = MPI_Init(&argc, &argv)

int ierr = MPI_Finalize()

int ierr = MPI_Comm_size(MPI_Comm comm,
    &nprocs)

int ierr = MPI_Comm_rank(MPI_Comm comm,
    &my_id)
```

Fortran

```
MPI_INIT(IERR)

MPI_FINALIZE(IERR)

MPI_COMM_SIZE(COMM, NPROCS, IERR)

MPI_COMM_RANK(COMM, MY_ID, IERR)
```

Gives the ID of each process in the communicator

MPI library: ubiquitous functions

These are the ones that you will **always** use in an MPI program

C/C++

```
int ierr = MPI_Init(&argc, &argv)

int ierr = MPI_Finalize()

int ierr = MPI_Comm_size(MPI_Comm comm,
    &nprocs)

int ierr = MPI_Comm_rank(MPI_Comm comm,
    &my_id)
```

Fortran

```
MPI_INIT(IERR)

MPI_FINALIZE(IERR)

MPI_COMM_SIZE(COMM, NPROCS, IERR)

MPI_COMM_RANK(COMM, MY_ID, IERR)
```

Outline

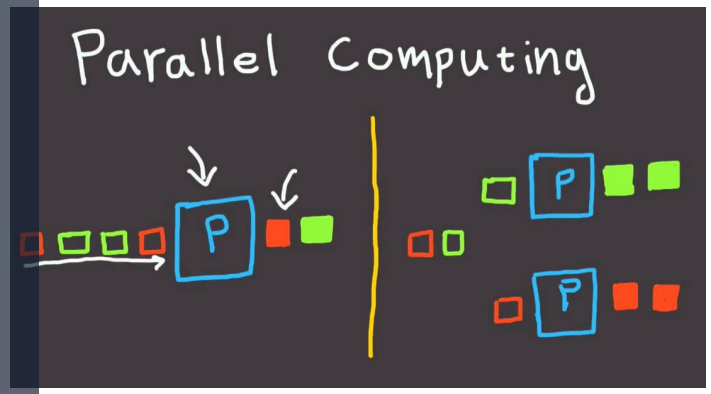
Part A: **Parallel Computing Quick Review** (10 min)

- Task and Data parallelism
- Parallel computers
- Threads, processes and more

Part B: **MPI Intro** (60 min)

- Core concepts
- Message passing
- Execution

MPI



Part C: **Exercises** (45+ min)

Ex 1: Hello World

Ex 2: Domain Decomposition

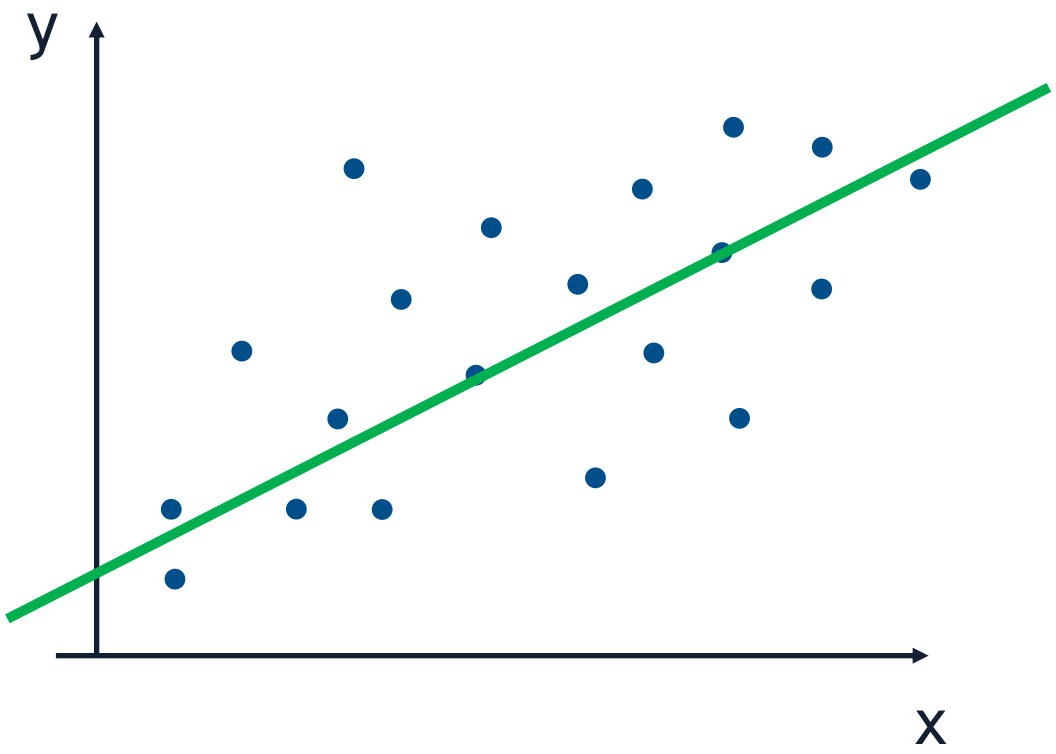
Ex 3: Parallel Linear Regression

Exercise 1 – MPI Hello World!

https://github.com/babreu-ncsa/IntroToMPI/tree/uiuc-icc/Exercises/Hello

1. `module load intel/18.0`
2. Parallelize the Hello World serial code and make each PE print their ID with the hello message

Linear Regression 101



x: inputs (features, control variable)
y: outputs (observations, response variable)

Model

$$y = a \cdot x + b$$

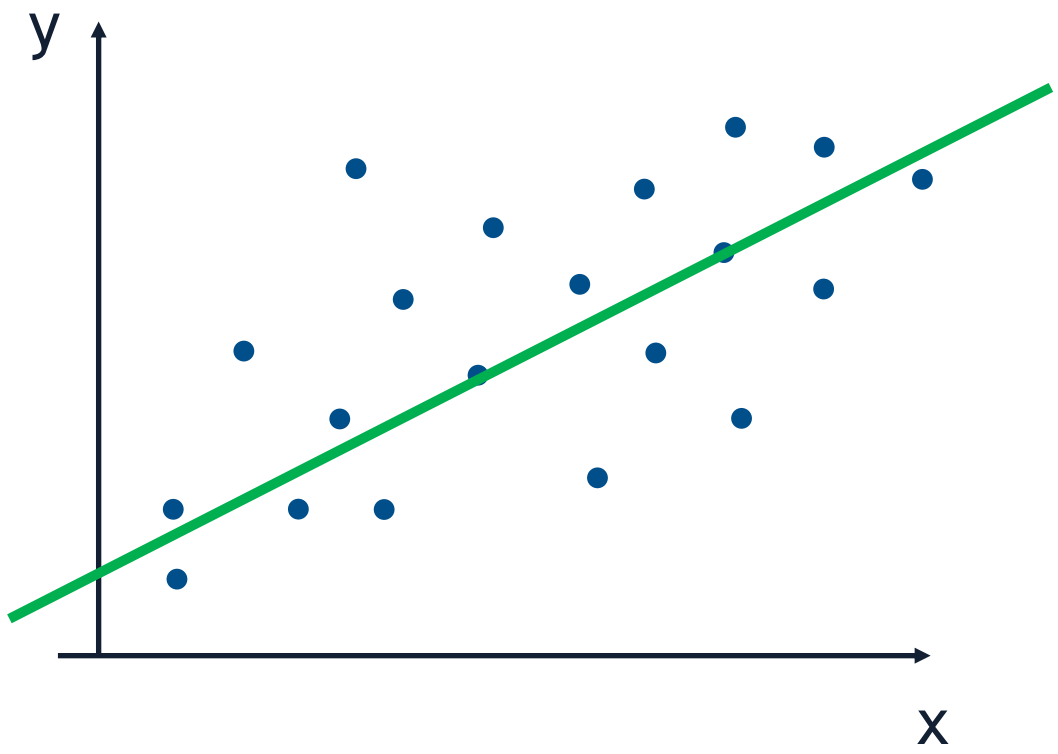
Parameters

Fitting

Find the "best" (a, b) pair

Inference

Linear Regression 101



x: inputs (features, control variable)
y: outputs (observations, response variable)

Model

$$y = a \cdot x + b$$

Parameters

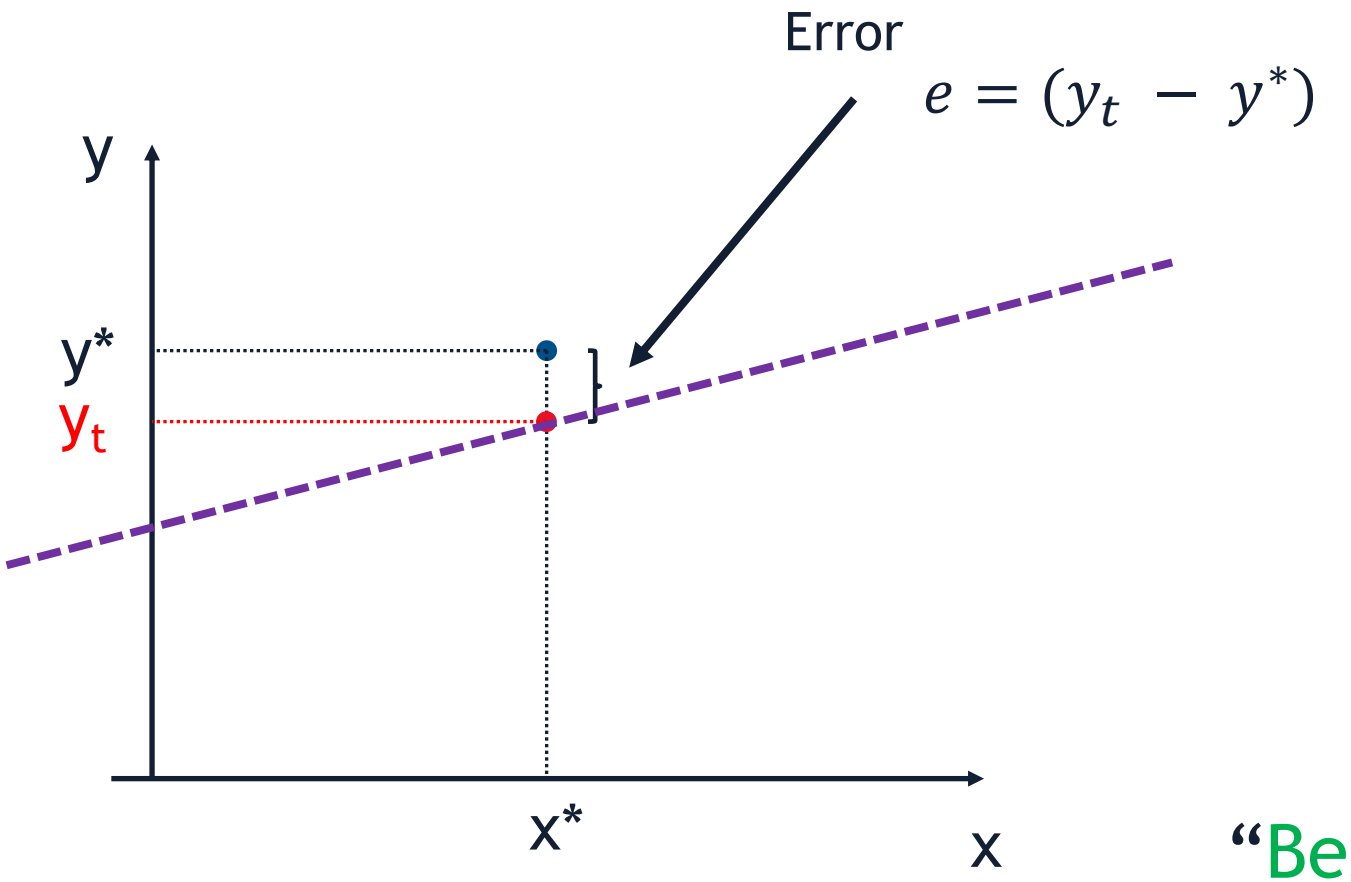
Fitting

Find the "best" (a, b) pair

?

Inference

Linear Regression 101



$$y_t = a_t \cdot x^* + b_t$$

Mean Squared Error (MSE)

$$MSE = \frac{1}{N}\sum_{i=1}^{N} e_i^2$$

"Best" (a, b) ➡ Smallest MSE

LinReg with MPI

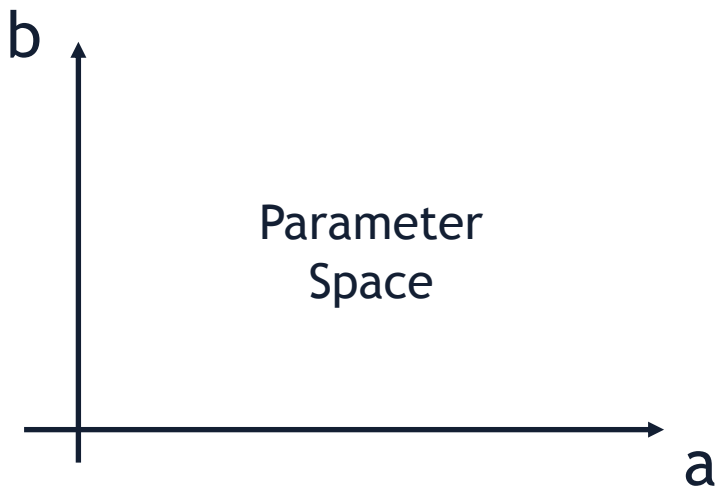
https://github.com/babreu-ncsa/IntroToMPI/tree/uiuc-icc/Exercises/LinearRegression

Serial LinReg

```
int main()
{
    // parameter map
    int na = 10, nb = 10;       // number of points for each parameter in grid space
    double da = 0.1, db = 0.1; // grid spacing in each direction
    vector<double> a, b;        // parameters in each direction
```

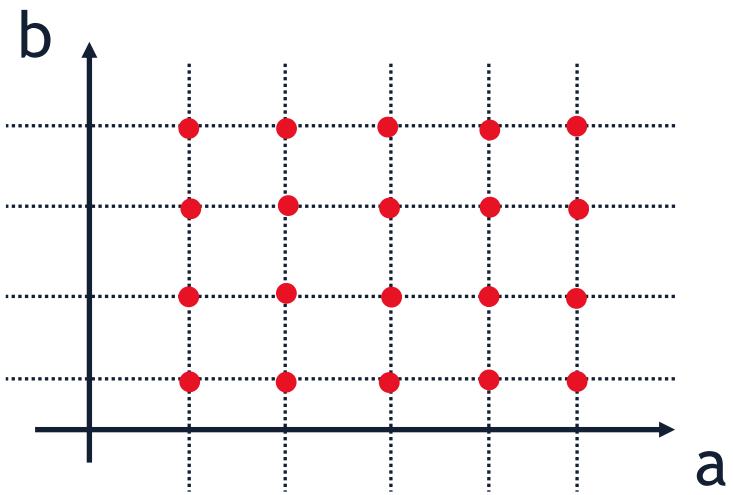
Serial LinReg

```
int main()
{
    // parameter map
    int na = 10, nb = 10;       // number of points for each parameter in grid space
    double da = 0.1, db = 0.1; // grid spacing in each direction
    vector<double> a, b;        // parameters in each direction
```



Serial LinReg

```
int main()
{
    // parameter map
    int na = 10, nb = 10;       // number of points for each parameter in grid space
    double da = 0.1, db = 0.1; // grid spacing in each direction
    vector<double> a, b;        // parameters in each direction
```



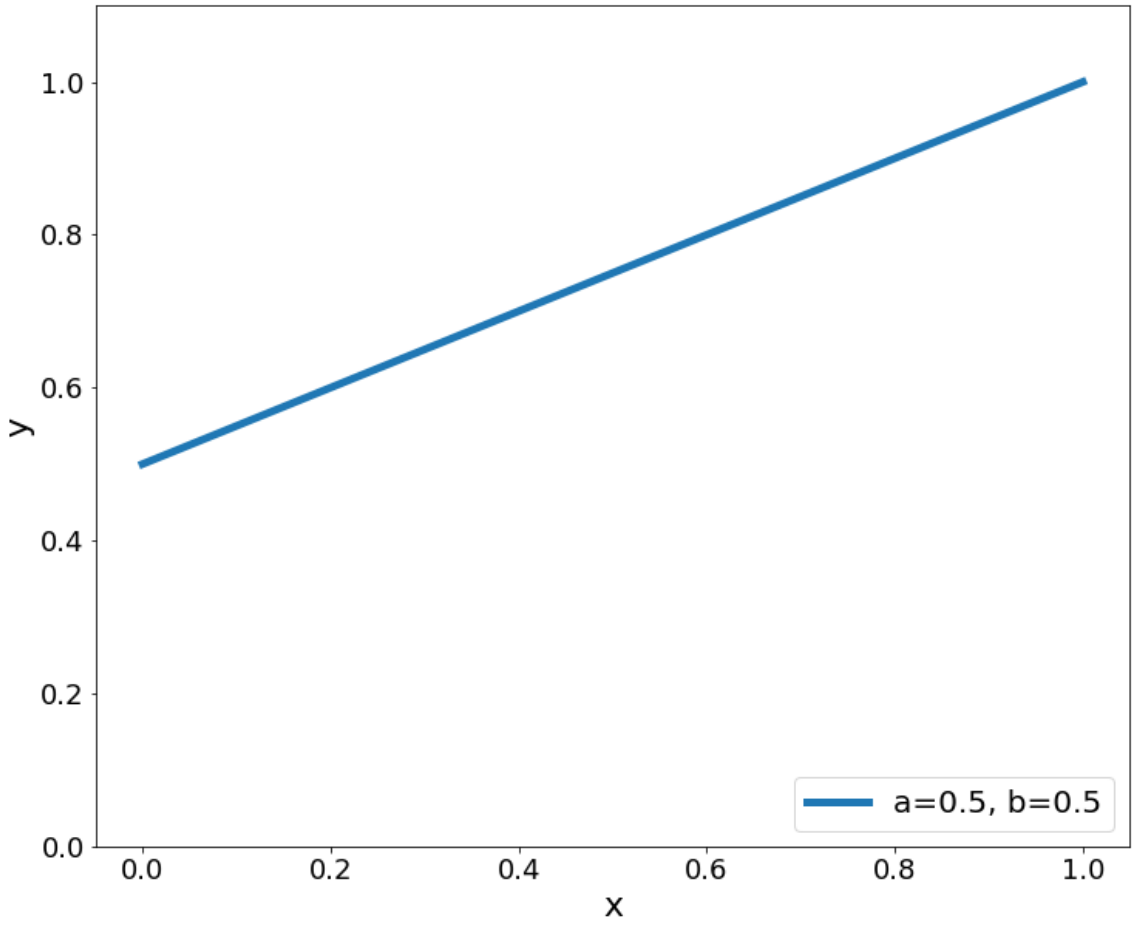

```
// 1. Build parameter map - square grid in (a,b) space
for (i = 0; i < na; i++)
{
    a.push_back((i + 1) * da);
}
for (j = 0; j < nb; j++)
{
    b.push_back((j + 1) * db);
}
```

Serial LinReg

```
// target straight line
int n = 1 << 23;            // number of "data" points
vector<double> x;           // control variable
double dx;                  // control variable spacing
vector<double> y;           // response variable
double at = 0.5, bt = 0.5; // target parameters
```

Serial LinReg

```
// target straight line
int n = 1 << 23;            // number of "data" points
vector<double> x;           // control variable
double dx;                  // control variable spacing
vector<double> y;           // response variable
double at = 0.5, bt = 0.5; // target parameters
```

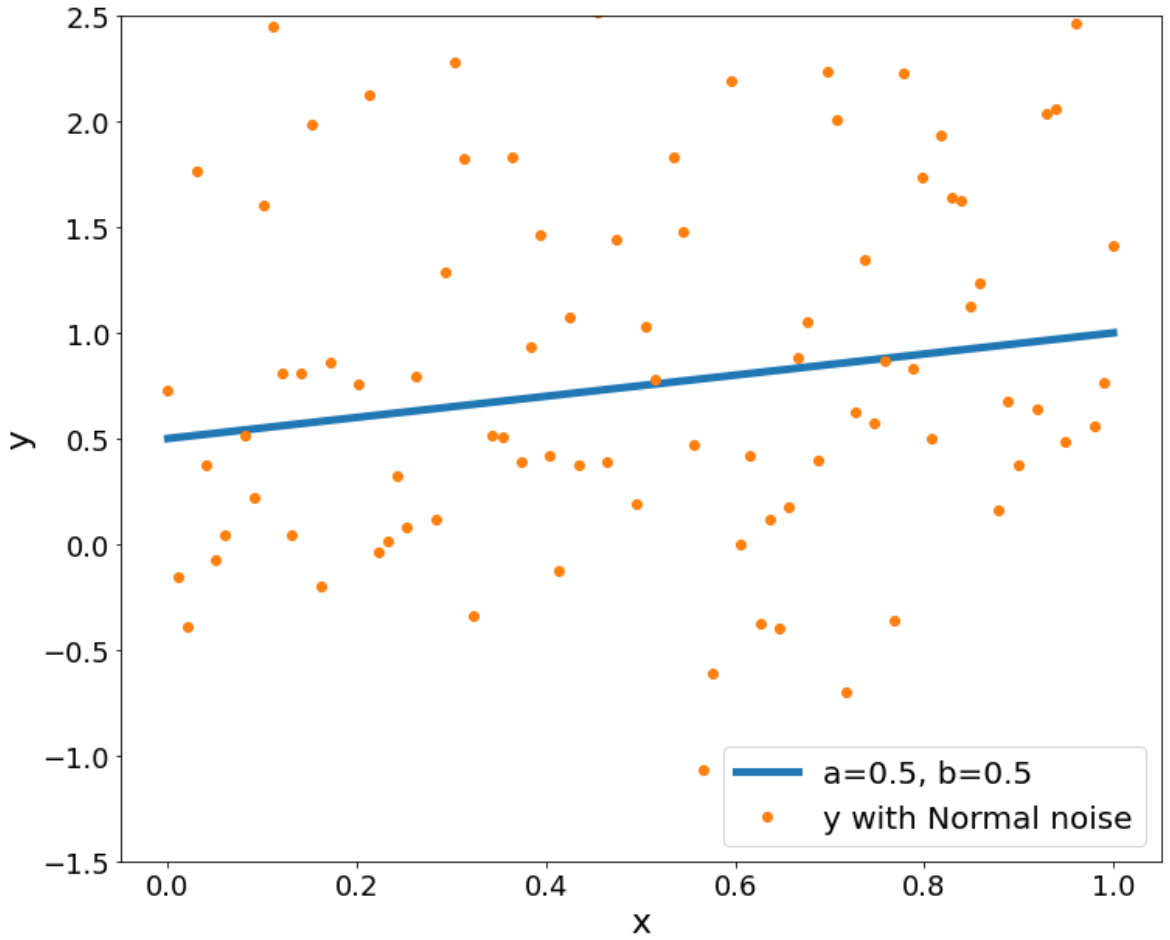


Serial LinReg

```
// 2. Build points to fit (dataset)
dx = 1.0 / double(n); // we'll make x go from 0 to 1
for (i = 0; i < n; i++)
{
    xp = i * dx;
    yp = at * xp + bt;
    rand1 = (double)rand() / RAND_MAX;
    rand2 = (double)rand() / RAND_MAX;
    z = sqrt(-2.0 * log(rand1)) * cos(2.0 * pi * rand2);
    yp = yp + z;
    x.push_back(xp);
    y.push_back(yp);
}
```

Serial LinReg

```
// 2. Build points to fit (dataset)
dx = 1.0 / double(n); // we'll make x go from 0 to 1
for (i = 0; i < n; i++)
{
    xp = i * dx;
    yp = at * xp + bt;
    rand1 = (double)rand() / RAND_MAX;
    rand2 = (double)rand() / RAND_MAX;
    z = sqrt(-2.0 * log(rand1)) * cos(2.0 * pi * rand2);
    yp = yp + z;
    x.push_back(xp);
    y.push_back(yp);
}
```



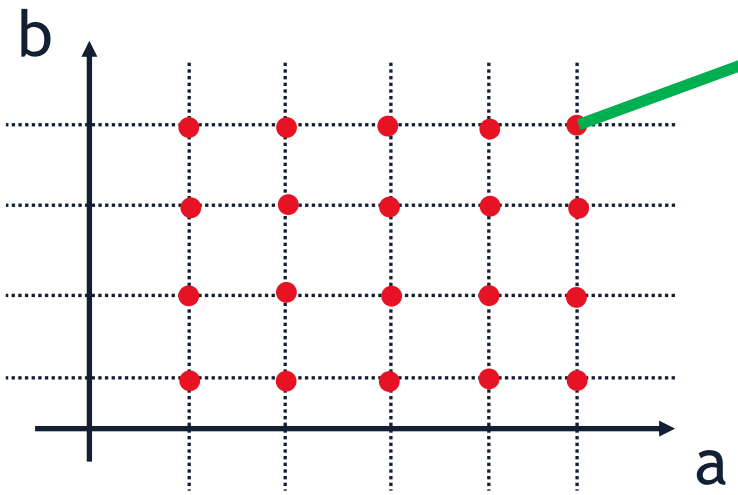
https://colab.research.google.com/drive/124oGew9s_GA3vKkLjr2-pRepEk3MdKze?usp=sharing

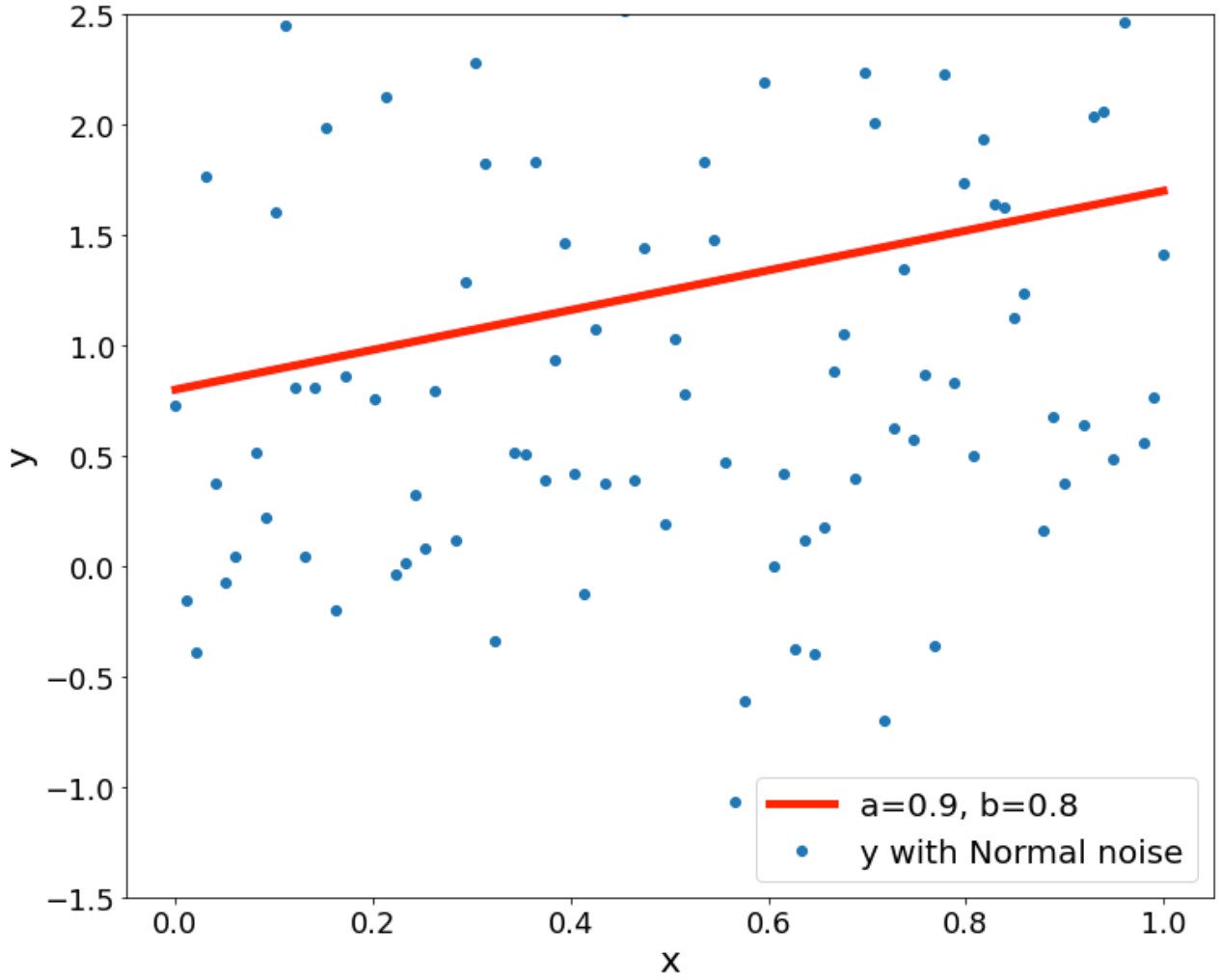
Serial LinReg

```
// 3. Explore parameter space
for (i = 0; i < na; i++)
{
    as = a[i];
    for (j = 0; j < nb; j++)
    {
        bs = b[j];
        rss = 0.0;
        for (k = 0; k < n; k++)
        {
            ys = as * x[k] + bs;
            rss = rss + pow((ys - y[k]), 2.0);
        }
        mse.push_back(rss / n);
        cout << "(a,b) = (" << a[i] << "," << b[j] << ")
    }
}
```

Serial LinReg

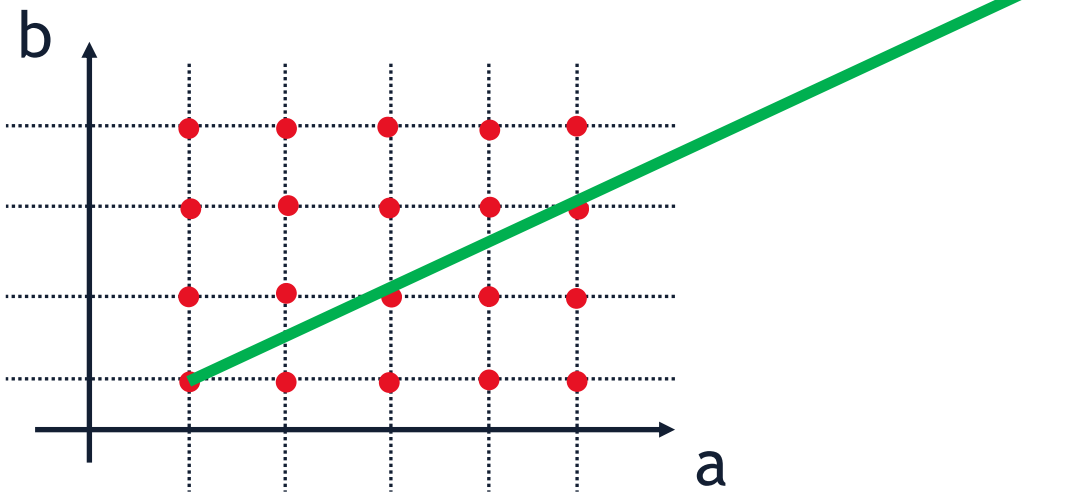
```
// 3. Explore parameter space
for (i = 0; i < na; i++)
{
    as = a[i];
    for (j = 0; j < nb; j++)
    {
        bs = b[j];
        rss = 0.0;
        for (k = 0; k < n; k++)
        {
            ys = as * x[k] + bs;
            rss = rss + pow((ys - y[k]), 2.0);
        }
        mse.push_back(rss / n);
        cout << "(a,b) = (" << a[i] << "," << b[j] << ")
    }
}
```

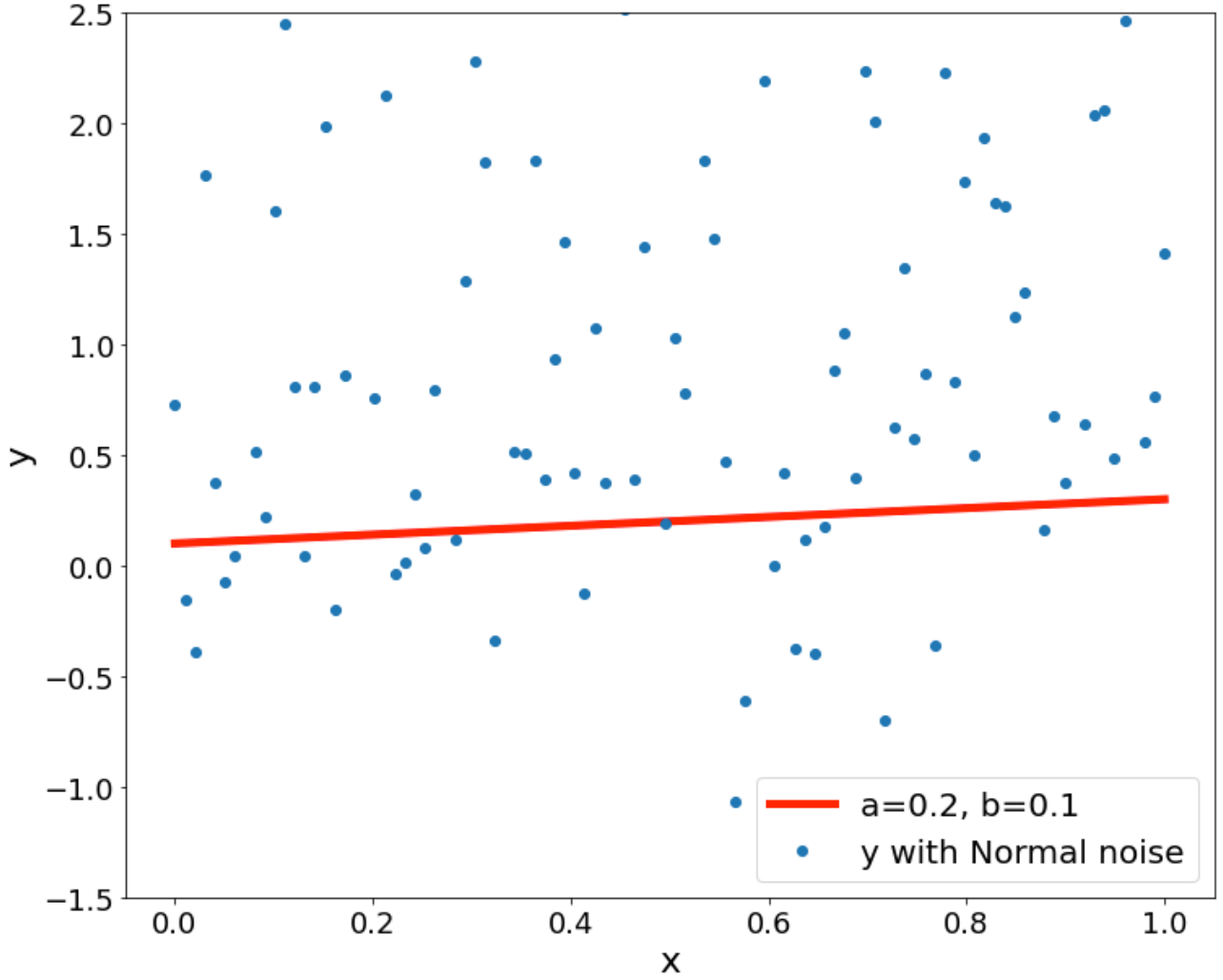




Serial LinReg

```
// 3. Explore parameter space
for (i = 0; i < na; i++)
{
    as = a[i];
    for (j = 0; j < nb; j++)
    {
        bs = b[j];
        rss = 0.0;
        for (k = 0; k < n; k++)
        {
            ys = as * x[k] + bs;
            rss = rss + pow((ys - y[k]), 2.0);
        }
        mse.push_back(rss / n);
        cout << "(a,b) = (" << a[i] << "," << b[j] << ")
    }
}
```





Serial LinReg

```
// 4. Look for best combination of (a,b)
counter = 0;
best_mse = mse[0];
best_i = 0;
best_j = 0;
for (i = 0; i < na; i++)
{
    for (j = 0; j < nb; j++)
    {
        if (mse[counter] < best_mse)
        {
            best_mse = mse[counter];
            best_i = i;
            best_j = j;
        }
        counter++;
    }
}
```

Serial LinReg

```
// 4. Look for best combination of (a,b)
counter = 0;
best_mse = mse[0];
best_i = 0;
best_j = 0;
for (i = 0; i < na; i++)
{
    for (j = 0; j < nb; j++)
    {
        if (mse[counter] < best_mse)
        {
            best_mse = mse[counter];
            best_i = i;
            best_j = j;
        }
        counter++;
    }
}
```

$$MSE = \frac{1}{N}\sum_{i=1}^{N} e_i^2$$

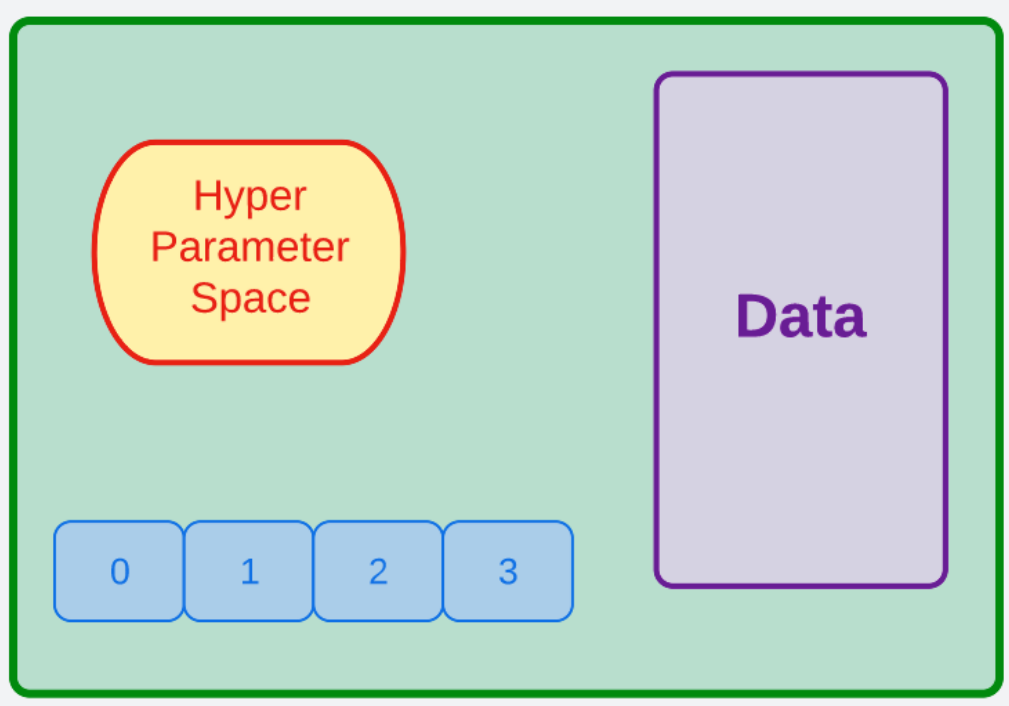
"Best" (a, b) ⟹ Smallest MSE

Exercise 2 – Domain decomposition

It’s time to parallelize the Linear Regression using MPI!

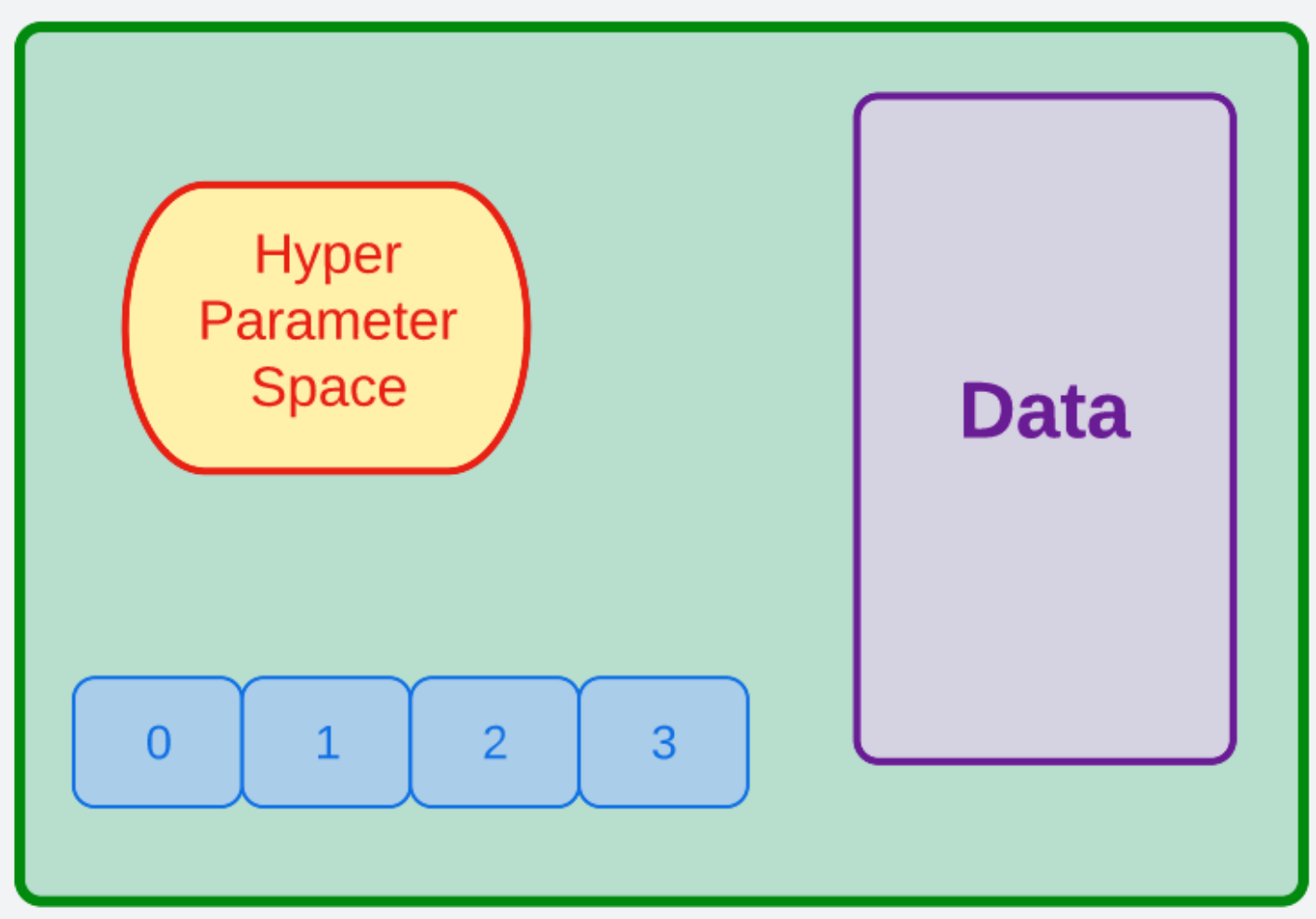
The first step to parallelization is called domain decomposition

- What each process will be doing?
- How do I distribute the tasks to each of them?
- How do I slice my entire dataset so that each process works on a particular part of it?
- Do I want my code to just run fast, or do I want to solve bigger and bigger problems?

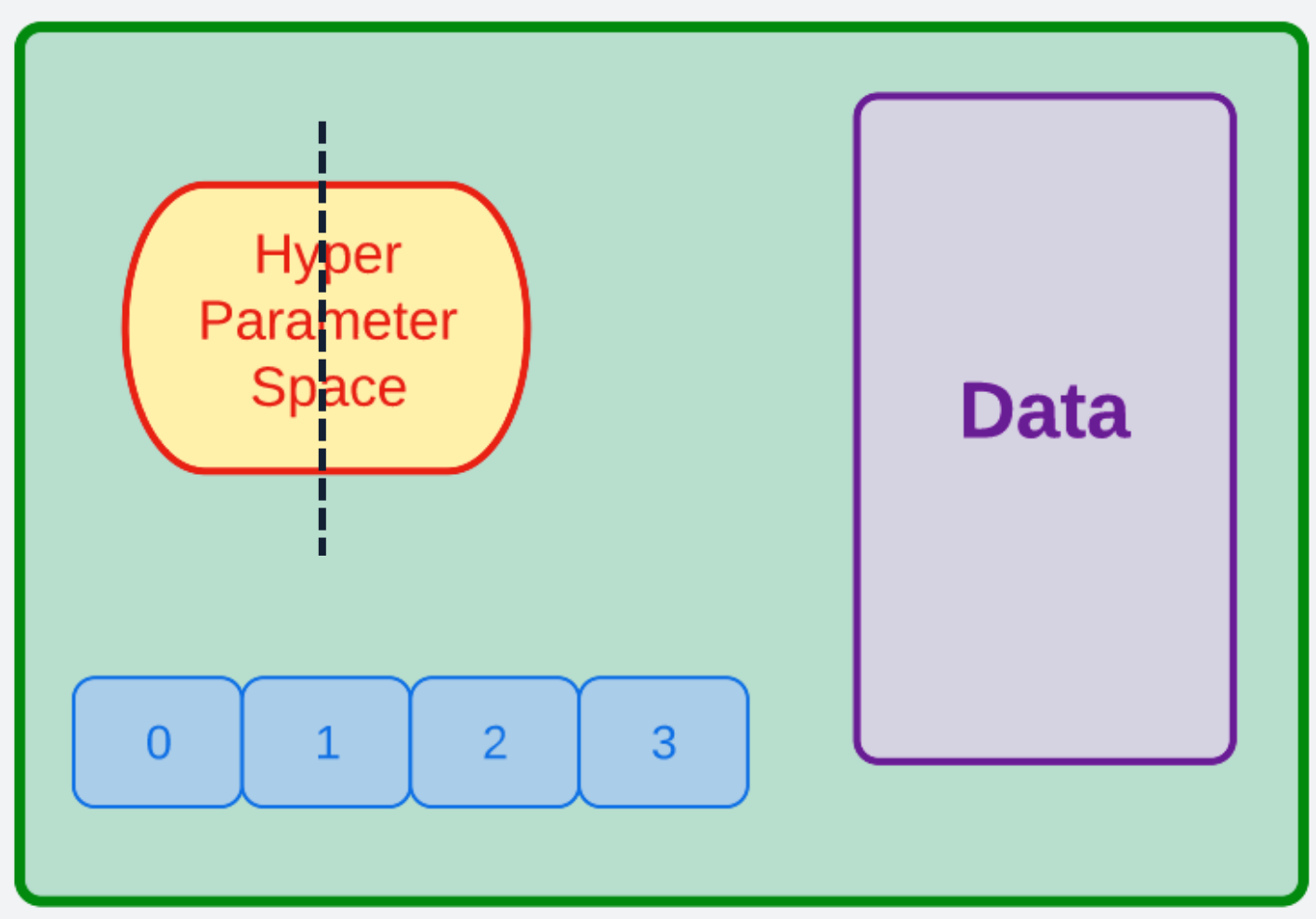


https://docs.google.com/presentation/d/13skCf6Wf__cB0bbF1CIaZ_z1ZOQSDQ5KxKqCA9Q5XOo/edit?usp=sharing

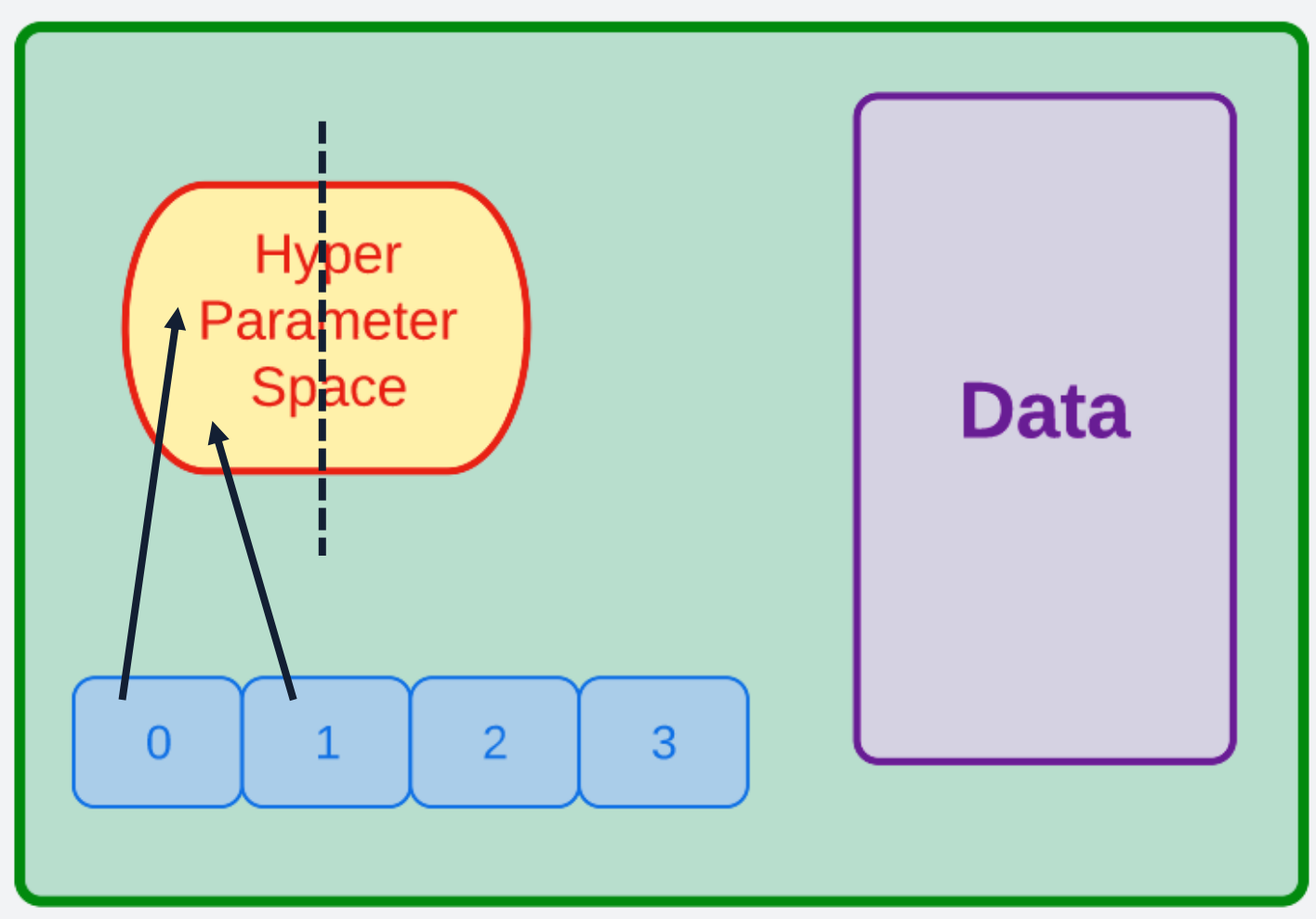
Decomposition example



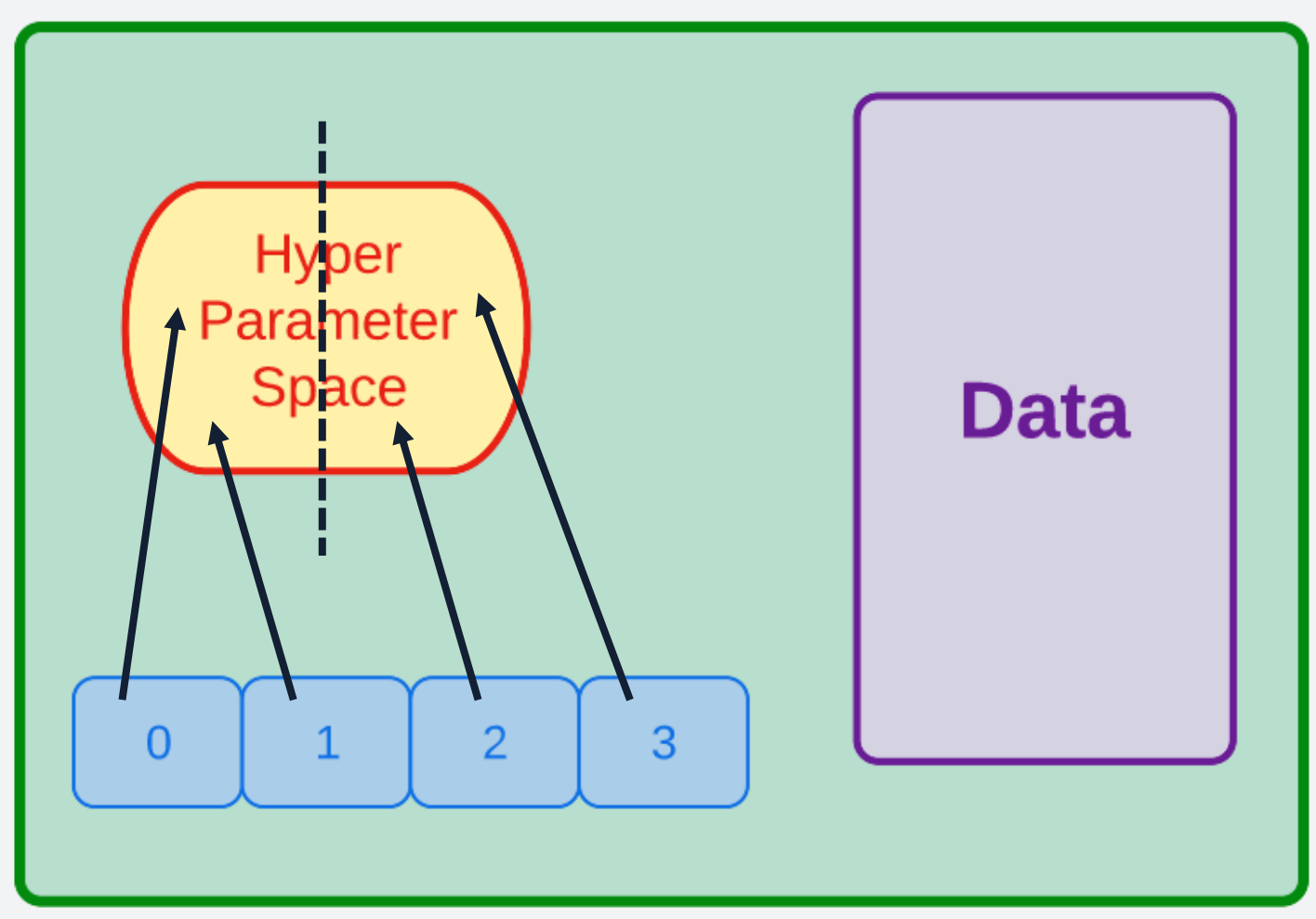
Decomposition example



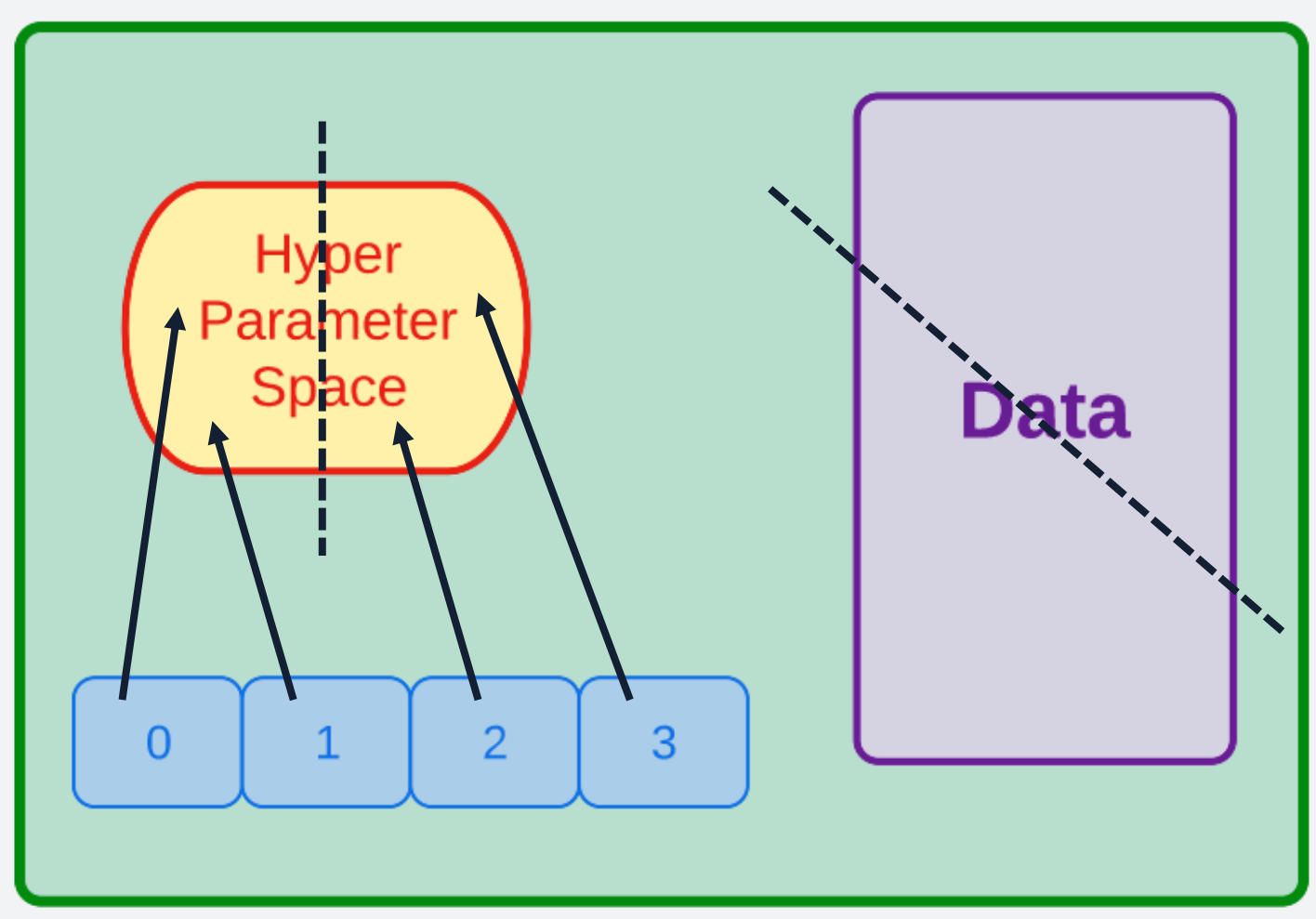
Decomposition example



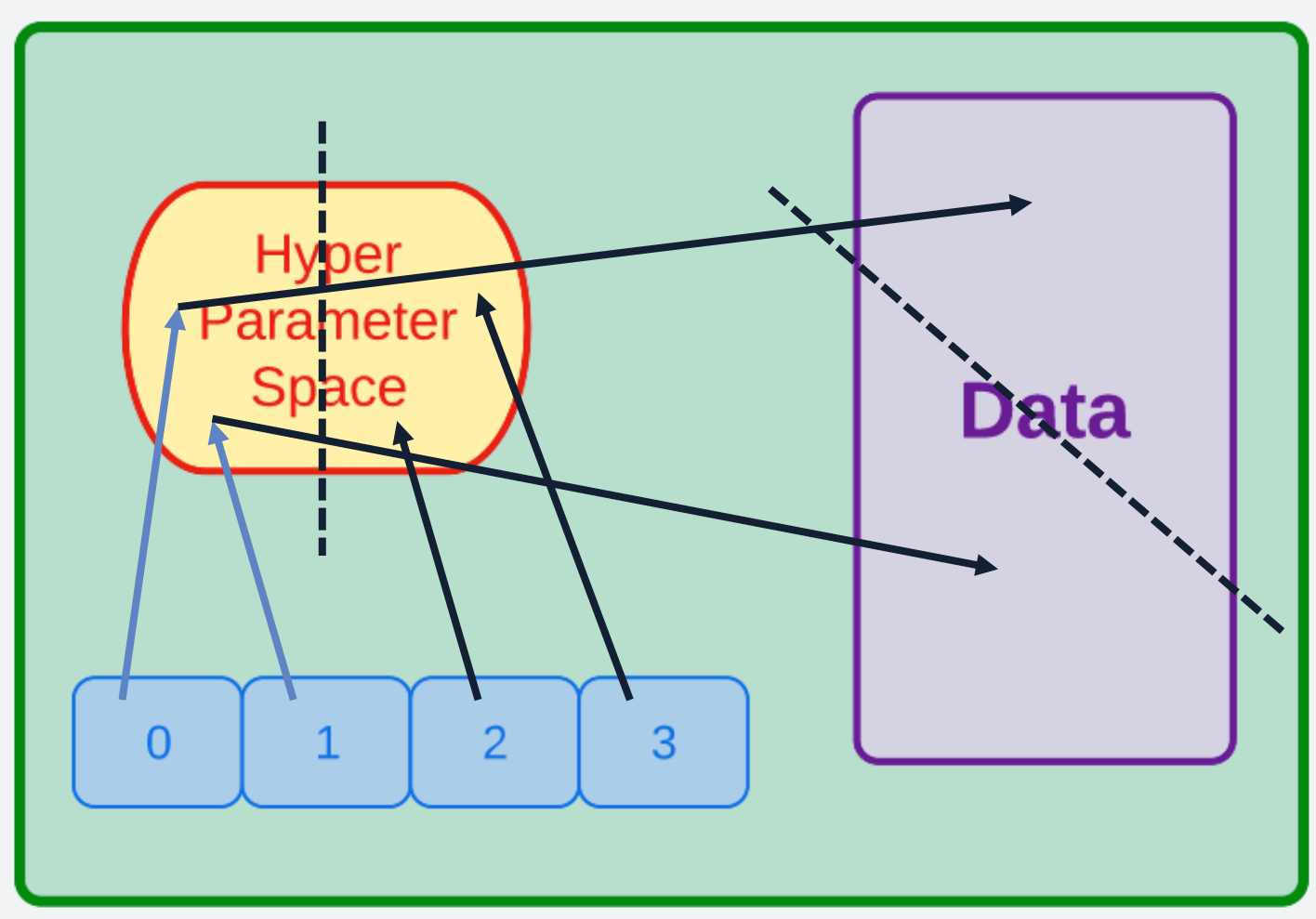
Decomposition example



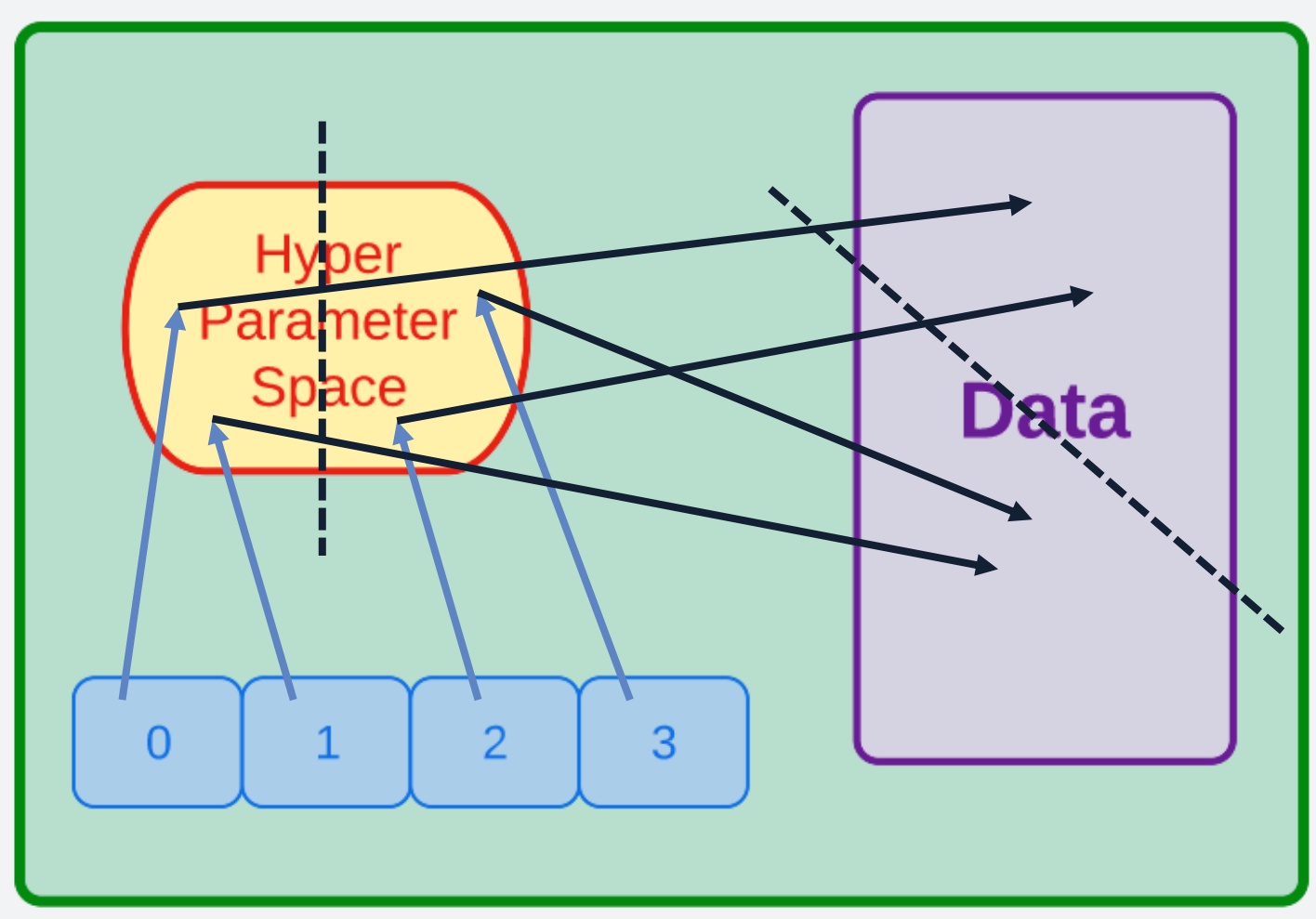
Decomposition example



Decomposition example



Decomposition example



Exercise 3 – MPI Linear Regression

1. Implement your domain decomposition using the MPI library functions
2. Check for consistency of your results
3. Perform a scaling analysis

Outline

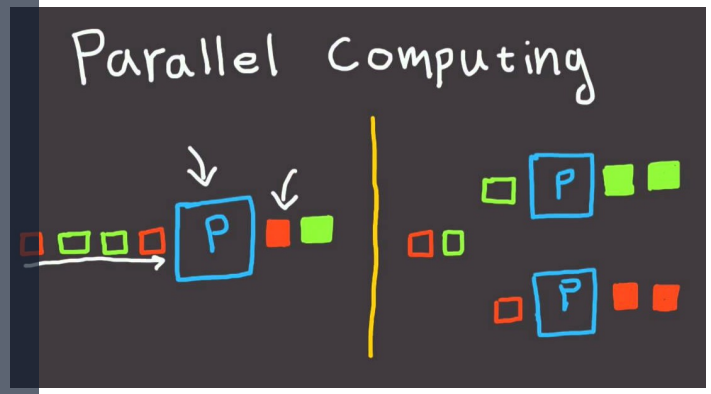
Part A: **Parallel Computing Quick Review** (10 min)

- Task and Data parallelism
- Parallel computers
- Threads, processes and more

Part B: **MPI Intro** (60 min)

- Core concepts
- Message passing
- Execution

MPI



Part C: **Exercises** (45+ min)

Ex 1: Hello World

Ex 2: Domain Decomposition

Ex 3: Parallel Linear Regression

Summary

Parallel computing is essential for modern scientific research computing

Computers today execute multiple instructions on multiple pieces of data (MIMD)

MPI allows for parallelization that can employ thousands of processes; redesigning an existing serial application is necessary

MPI collective communications are a good starting point; Point-to-point communication are often necessary for fully scalable parallelism

Final assignment (optional)

Write a simple code that is related to the scientific research problem you are interested in (you may already have it!)

Line up the steps to parallelize it: perform domain decomposition, map the time of required communications

Parallelize it using MPI library functions

Share your results! Let's add it to the collaborative GitHub folder!

https://github.com/babreu-ncsa/parallelization

Useful links

GitHub Parallelization repo
https://github.com/babreu-ncsa/parallelization

GitHub repo with this session's exercises
https://github.com/babreu-ncsa/IntroToMPI

HPC Training Moodle – Introduction to OpenMP course
https://www.hpc-training.org/xsede/moodle/course/view.php?id=72

MPI related contents
https://www.mpi-forum.org/docs/
https://mpitutorial.com/tutorials/

Google Drive folder with this session's material
https://drive.google.com/drive/folders/17WliPmA3vZ-PjXdiVlH1rx3WQtDClZZV?usp=sharing

Feed us back, please!

Feedback is vital for our continuous improvement as instructors/lecturers, and for our organization.

Please take a couple of minutes to answer this anonymous survey. Any sort of comments are extremely helpful and all of them are going to be carefully taken under consideration.

Thank you!

https://go.ncsa.illinois.edu/IntroToMPIFeedback

Q&A

Bruno Abreu, Ph.D.
Research Scientist
babreu@illinois.edu

https://go.ncsa.illinois.edu/IntroToMPIFeedback

National Center for Supercomputing Applications
UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN