“The Eight Fallacies of Distributed Computing

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences.

- *The network is reliable*
- *Latency is zero*
- *Bandwidth is infinite*
- *The network is secure*
- *Topology doesn’t change*
- *There is one administrator*
- *Transport cost is zero*
- *The network is homogeneous”*

Peter Deutsch, Engineer at Sun Microsystems, 1997

Lecture B.5:
Distributed-Memory Parallel Processing

CS205: Computing Foundations for Computational Science
Dr. David Sondak
Spring Term 2020

HARVARD
School of Engineering and Applied Sciences

INSTITUTE FOR APPLIED COMPUTATIONAL SCIENCE
AT HARVARD UNIVERSITY

Lectures developed by Dr. Ignacio M. Llorente

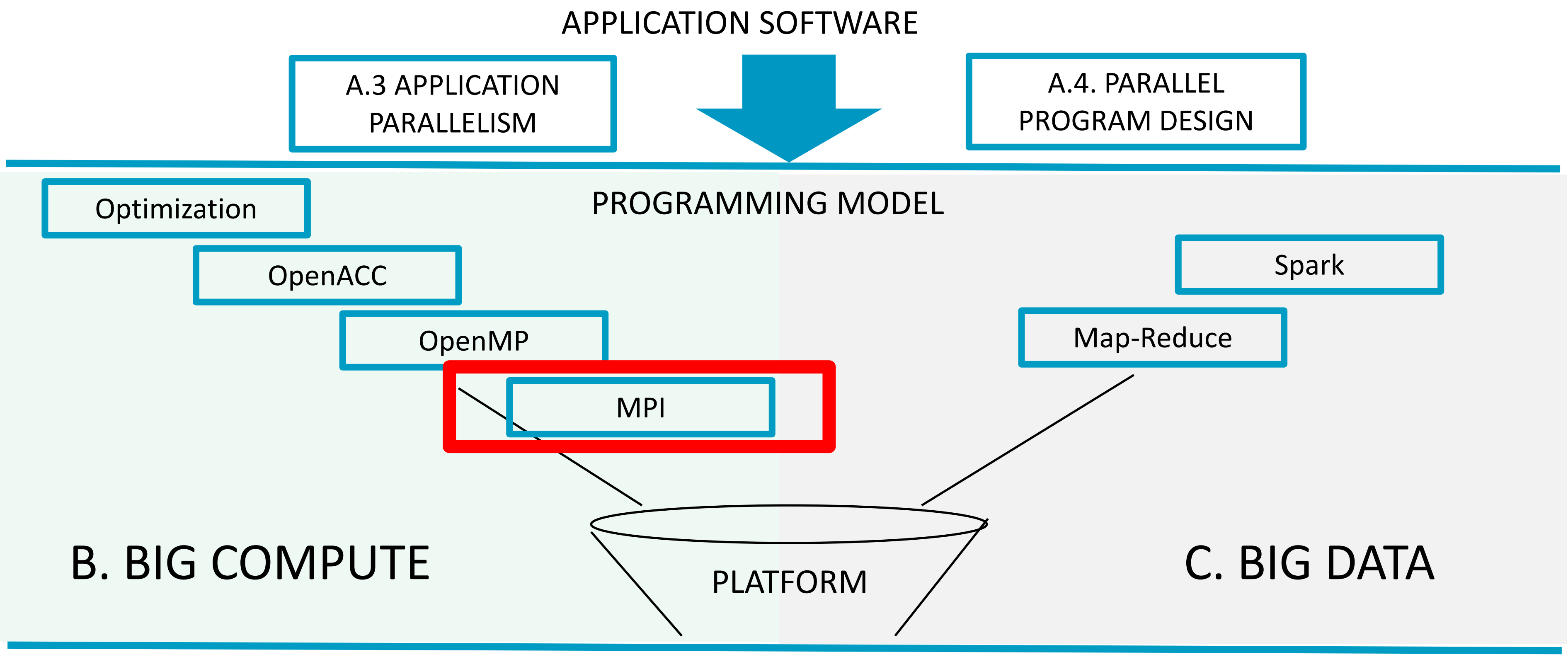
Before We Start

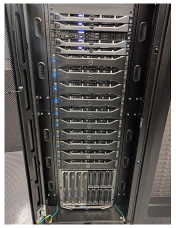
Where We Are

Computing Foundations for Computational and Data Science

How to use modern computing platforms in solving scientific problems

CS205: Contents





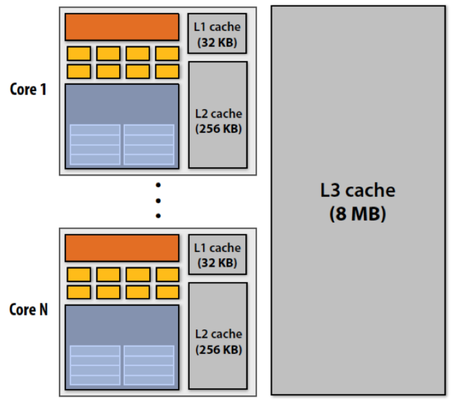
A.2. LARGE-SCALE PROCESSING ON CLOUD

A.1 PARALLEL ARCHITECTURES

Context

Distributed-Memory Parallel Processing

How can I make efficient use of multiple nodes?

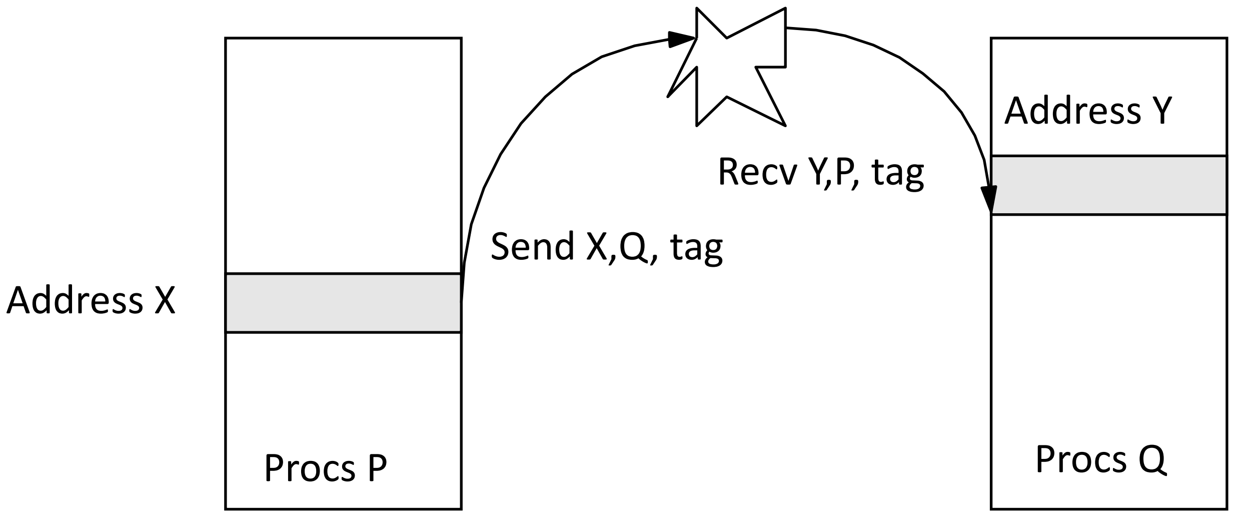


Multi-core

Multi-node

Context

Distributed-Memory Parallel Processing



Roadmap

Distributed-Memory Parallel Processing

Distributed-Memory basics

MPI Fundamentals

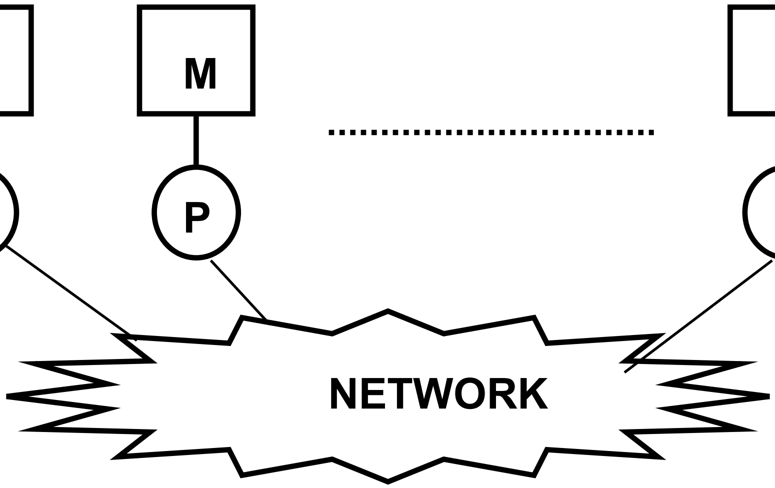
Hybrid Programming Model

A Summary of Big Compute Models

Distributed-Memory Basics

Elements of Programming

- Assumes that the machine consists of a collection of processors, each with local memory
- Each processor can access only the instructions/data stored in its own memory
- The machine has an interconnection network that supports passing messages between processors



Distributed-Memory Basics

Elements of Programming

- A user specifies a number of concurrent processes when program begins; the number of active processes typically remains constant throughout execution. However, dynamic parallelism is also possible

- Every process executes the same program, though the flow of execution should depend on the processor's unique ID number

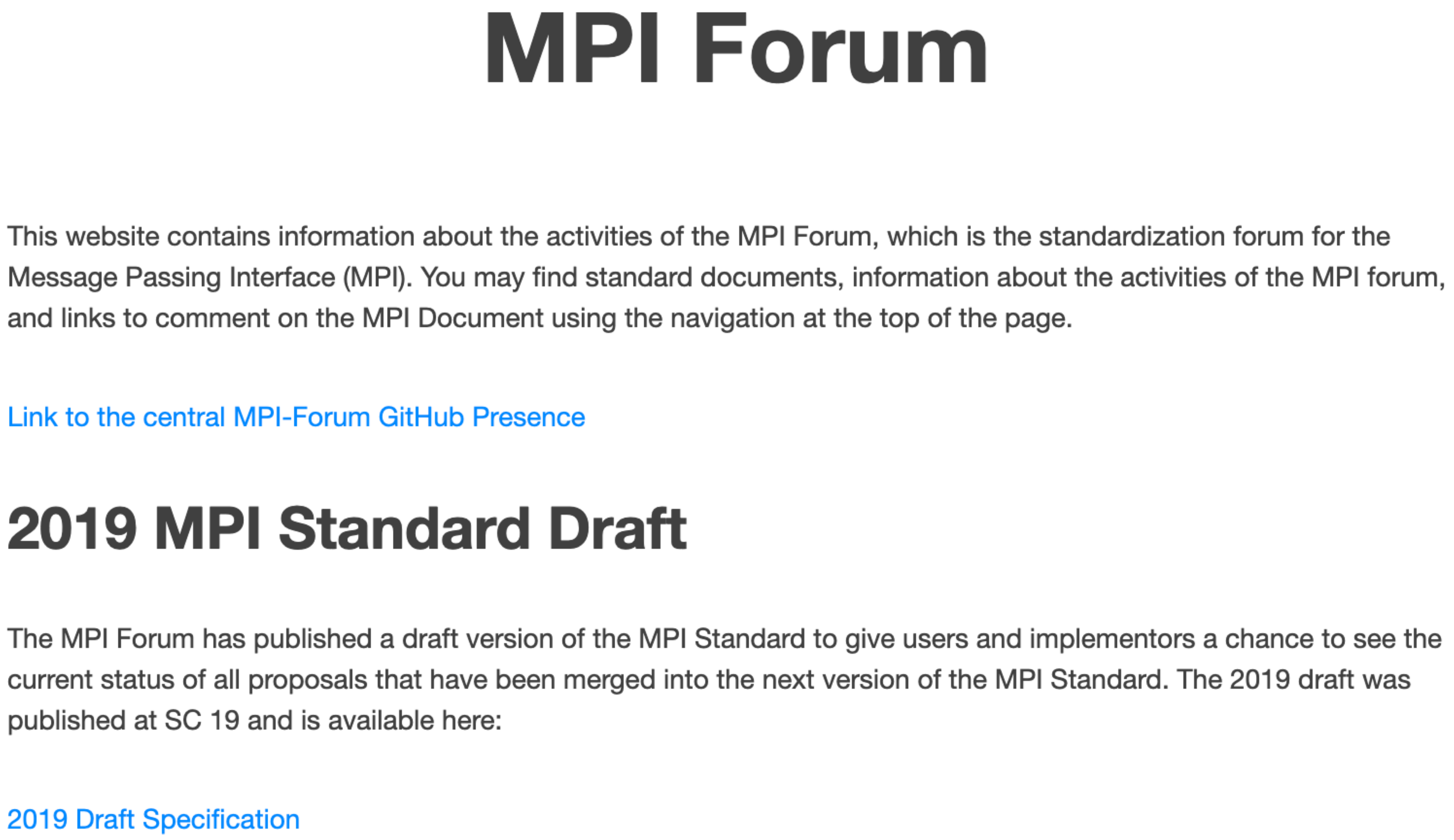
  (e.g. "`if (myid == 0) { }`")

- Each process performs computations on its local variables, then communicates with other processes, (and repeats), to eventually achieve the computed result

- In this model, message passing performs two roles: to send/receive information, and to synchronize with one another

MPI Fundamentals

What Is it?



MPI Forum

This website contains information about the activities of the MPI Forum, which is the standardization forum for the Message Passing Interface (MPI). You may find standard documents, information about the activities of the MPI forum, and links to comment on the MPI Document using the navigation at the top of the page.

Link to the central MPI-Forum GitHub Presence

2019 MPI Standard Draft

The MPI Forum has published a draft version of the MPI Standard to give users and implementors a chance to see the current status of all proposals that have been merged into the next version of the MPI Standard. The 2019 draft was published at SC 19 and is available here:

2019 Draft Specification

MPI Fundamentals

What IS it?

- API for distributed-memory programming
- De facto industry standard (MPI-3.1 released in 2015)
- Use from C/C++, Fortran, Python, R, ...
- More than 200 routines
- Using only 10 routines are enough in 99% of the cases

MPI Fundamentals

Why MPI?

- Standardization - MPI is the only message passing library which can be considered a standard
- Portability – Widely supported. There is no need to modify your source code when you port your application to a different platform
- Performance Opportunities - Vendor implementations should be able to exploit native hardware features to optimize performance and provide collective communication implementation
- Functionality – Rich set of features
- Availability - A variety of implementations are available, both vendor and public domain

MPI Fundamentals

A Simple Example

Basic Program Structure (Only 6 routines)

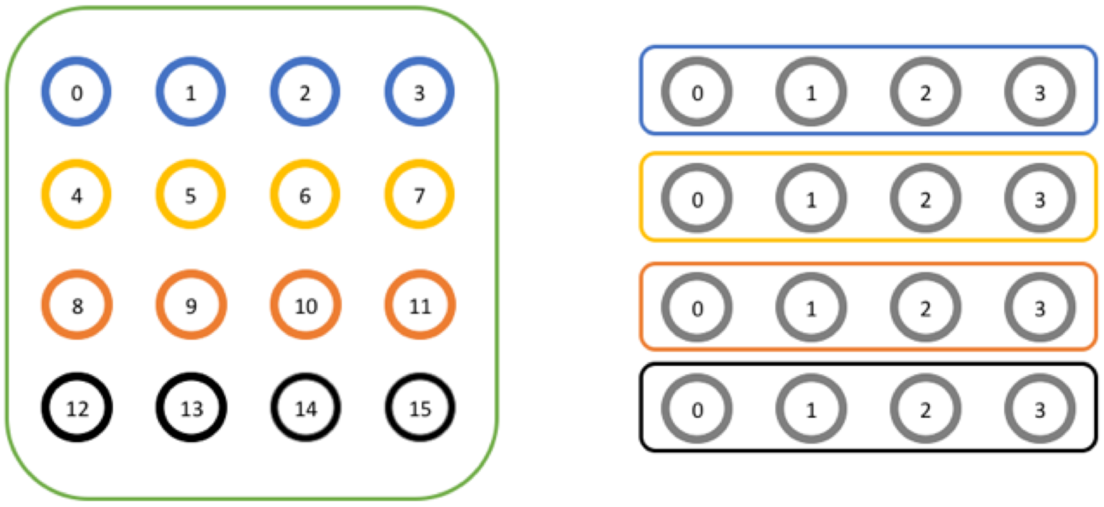
```
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
  {
    int rank, size, tag=50, destination=0, source;
    char message[100];
    MPI_Status state;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank !=0) {
      sprintf(message,"¡Greetings from process %d!", rank);
      MPI_Send(message, strlen(message)+1, MPI_CHAR, destination, tag, MPI_COMM_WORLD);
    } else {
      for (origen = 1; origen < size; origen++) {
        MPI_Recv(message, strlen(message)+1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &state);
        printf("%s\n", message);
      }
    }
    MPI_Finalize();
  }
```

MPI Fundamentals

Communicators

- A group of processes numbered 0,1,.. to N-1
  - ✓ Default communicator: `MPI_COMM_WORLD`
  - ✓ Contains all processes
- Query functions:
  - ✓ Number of processes: `MPI_Comm_size(MPI_COMM_WORLD, &nproc)`
  - ✓ My process number: `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`

Process can be part of several COMMs



`MPI_COMM_WORLD` `Colored communicators`

MPI Fundamentals

Communicators

```
#include "mpi.h”                                     // MPI header file
#include <stdio.h>
main(int argc, char *argv[]) {
  int np, pid;
  MPI_Init(&argc, &argv);                            // Initialize MPI

  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  printf("N. of procs = %d, proc ID = %d\n", np, pid);

  MPI_Finalize();                                    // Clean up
}
```

MPI Fundamentals

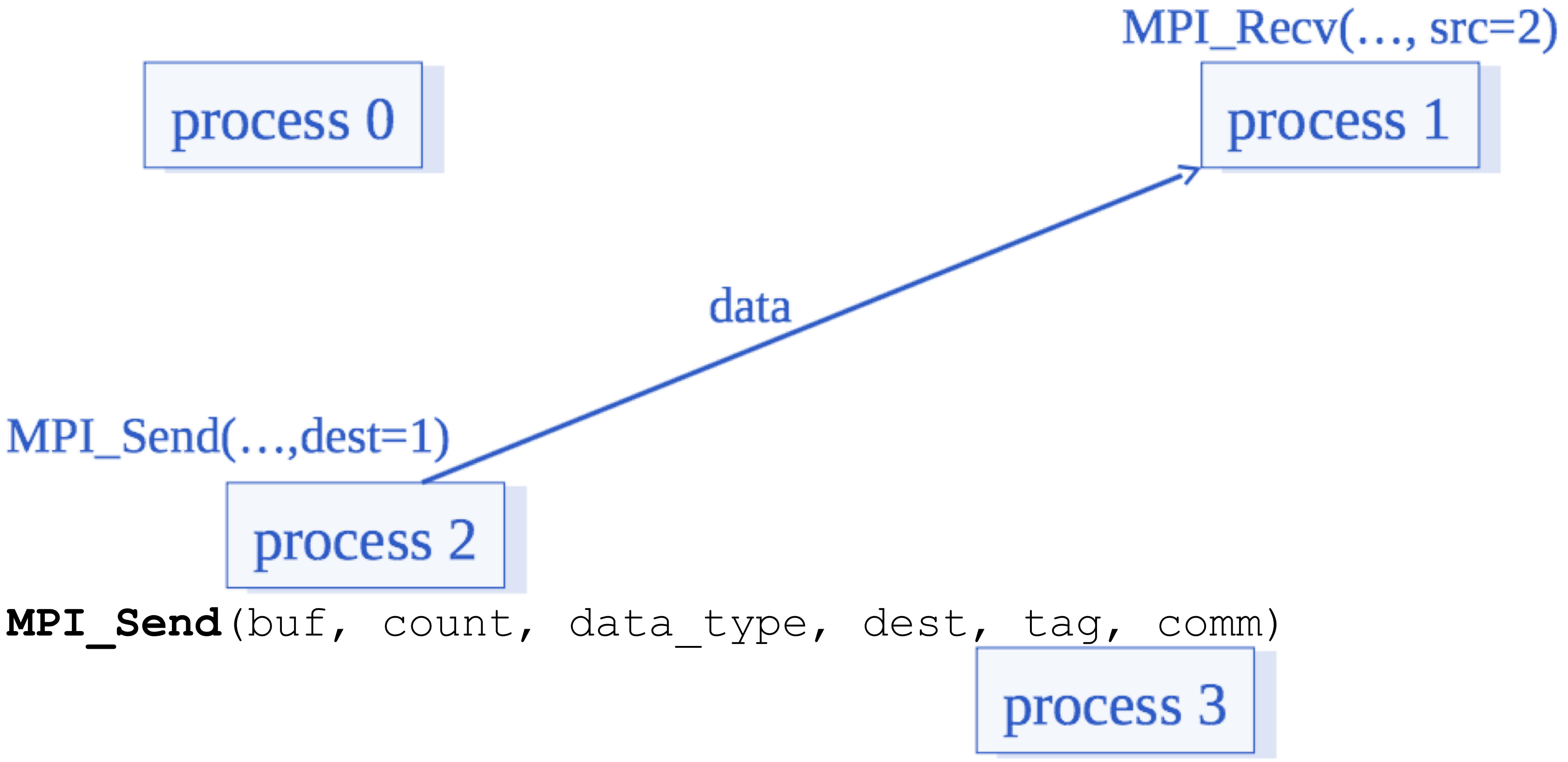
Data Types

Type	Comment
`MPI_CHAR`	char
`MPI_INT`	signed int
`MPI_LONG`	signed long int
`MPI_FLOAT`	float
`MPI_DOUBLE`	double
…	*many other….* *and with derived types*

MPI Fundamentals

Point to Point Communications

`MPI_Recv(buf, count, datatype, src, tag, comm, status)`



`MPI_Send(buf, count, data_type, dest, tag, comm)`

- A message is received when the following are matched: Source (sending process rank), Tag, Communicator (e.g. `MPI_COMM_WORLD`)
- Wildcard values may be used: `MPI_ANY_TAG` and `MPI_ANY_SOURCE`

MPI Fundamentals

Point to Point Communications

SEND Arguments	Meanings
`buf`	starting address of send buffer
`count`	# of elements
`data_type`	data type of each send buffer element
`dest`	processor ID (rank) destination
`tag`	message tag
`comm`	communicator

RECV Arguments	Meanings
`buf`	starting address of send buffer
`count`	# of elements
`data_type`	data type of each send buffer element
`src`	processor ID (rank) source
`tag`	message tag
`comm`	communicator
`status`	Status object

MPI Fundamentals

Point to Point Communications

```
int f[N], src=0, dest=1;
MPI_Status status;
// ...
MPI_Comm_rank( MPI_COMM_WORLD, &rank);


if (rank == src) // process "dest" ignores this
  MPI_Send(f, N, MPI_INT, dest, 0, MPI_COMM_WORLD);
if (rank == dest) // process "src" ignores this
  MPI_Recv(f, N, MPI_INT, src, 0, MPI_COMM_WORLD, &status);
//...
```

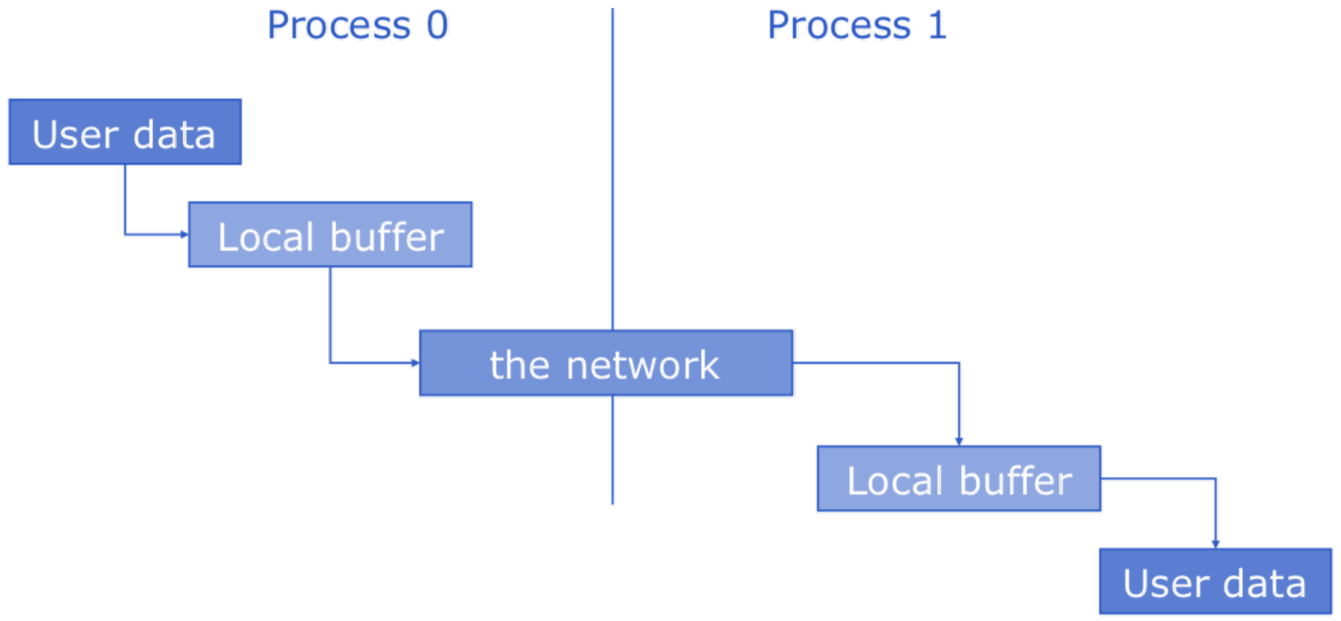
MPI Fundamentals

Blocking

- So far we have been using blocking communication:

✓ **`MPI_Send`** does not complete until buffer is empty (available for use).

✓ **`MPI_Recv`** does not complete until buffer is full (available for use).



- Completion depends on size of message and amount of system buffer

MPI Fundamentals

Blocking

Calling order matters

- ✓ It is possible to wait indefinitely, called “deadlock”
- ✓ Improper ordering results in serialization (loss of performance)

Deadlock

```
MPI_Comm_rank(comm, &rank);
if (rank == 0) {
MPI_Recv(recvbuf, cnt, MPI_INT, 1, tag, comm, &stat);
MPI_Send(sendbuf, cnt, MPI_INT, 1, tag, comm);
} else { /* rank==1 */
MPI_Recv(recvbuf, cnt, MPI_INT, 0, ta            &stat);
MPI_Send(sendbuf, cnt, MPI_INT, 0, t
}
```

This is called “unsafe” because it depends on the availability of system buffers

nsafe

```
MPI_Comm_rank(comm, &rank);
if (rank == 0) {
MPI_Send(sendbuf, cnt, MPI_INT, 1, tag, comm);
MPI_Recv(recvbuf, cnt, MPI_INT, 1, tag, comm, &stat);
} else { /* rank==1 */
MPI_Send(sendbuf, cnt, MPI_INT, 0, tag, comm);
MPI_Recv(recvbuf, cnt, MPI_INT, 0, tag, comm, &stat);}
```

MPI Fundamentals

Blocking

- Synchronous `MPI_Ssend`: The send does not complete until a matching receive has begun
- Buffered `MPI_Bsend`: The send does not complete until the message has been copied to internal buffer

In both cases it is safe to overwrite the memory areas where the data were originally stored

MPI Fundamentals

Non-Blocking

- Function call returns immediately, without completing data transfer
  - ✓ Only “starts” the communication (without finishing)
  - ✓ `MPI_Isend` and `MPI_Irecv`
  - ✓ Need an additional mechanism to ensure transfer completion (`MPI_Wait`)
- Avoid deadlock
- Possibly higher performance

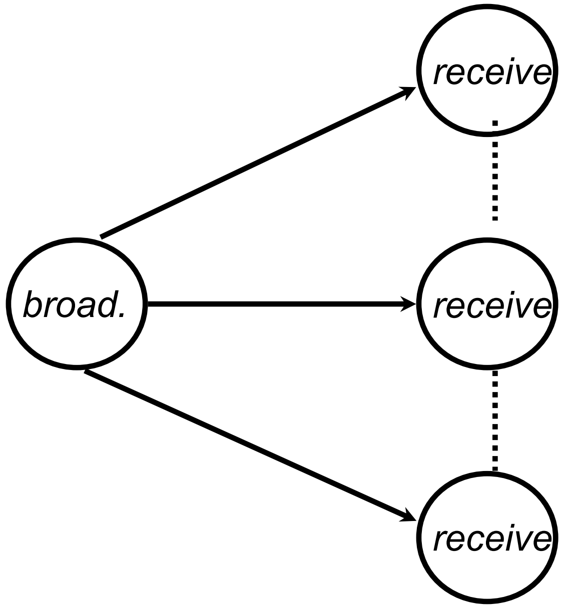
```
MPI_Request request_X, request_Y;
MPI_Isend(..., &request_X);
MPI_Isend(..., &request_Y);


//... more ground-breaking computations ...

MPI_Wait(&request_X, ...);
MPI_Wait(&request_Y,...);
```

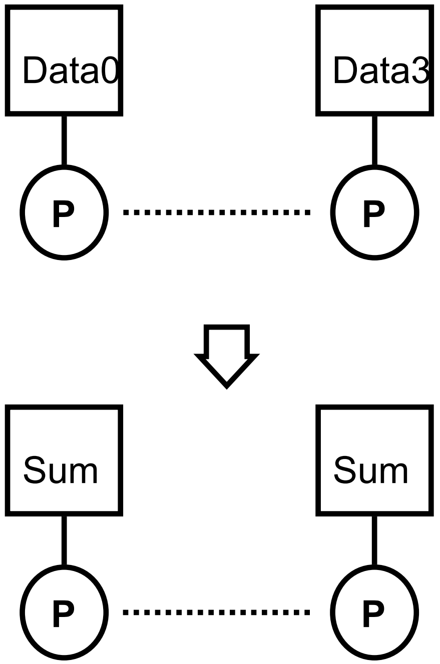
The sending process should not access the send buffer until the send completes

MPI Fundamentals

Collective Communications

- One to all
  - ✓ `MPI_Bcast, MPI_Scatter`
- All to one
  - ✓ `MPI_Reduce, MPI_Gather, MPI_Scatter`
- All to all
  - ✓ `MPI_AlltoAll`





MPI Fundamentals

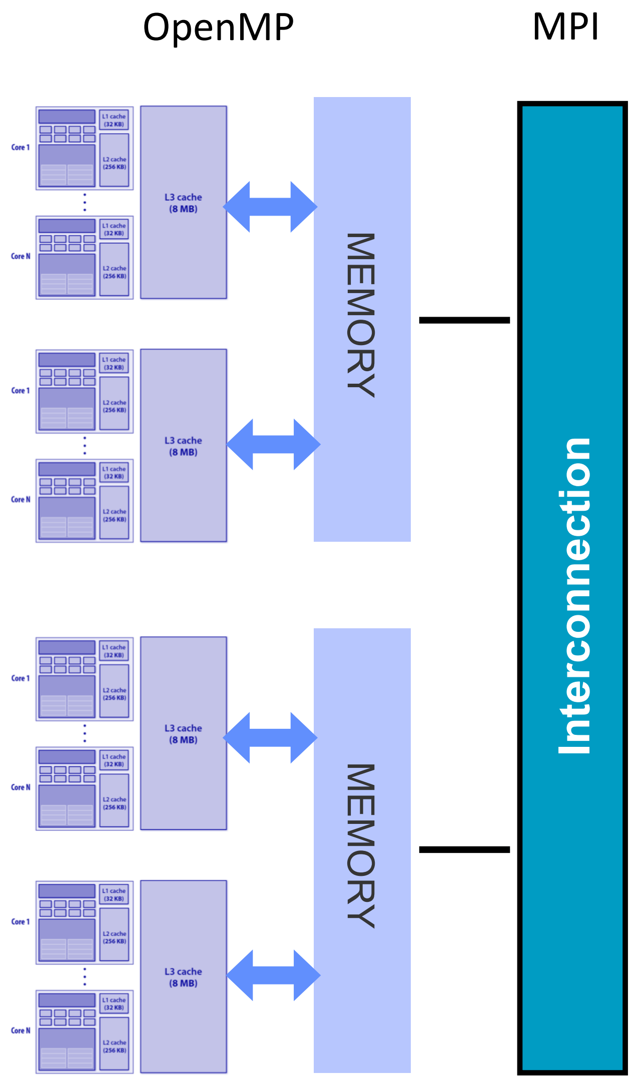
Example: Parallel Vector Inner Product

```
// loc_sum = local sum
float loc_sum = 0.0; // probably should use double
for (i = 0; i < N; i++)
  loc_sum += x[i] * y[i];

// sum = global sum
MPI_Reduce(&loc_sum, &sum, 1, MPI_FLOAT, MPI_SUM, root,
           MPI_COMM_WORLD);
```

Hybrid Model

Shared-Memory within and Distributed-Memory Across Nodes

- SMP Nodes
  - ✓ Single MPI task launched per node
  - ✓ Parallel Threads share all node memory, e.g 1 4-thread task in each node x 2 nodes
- SMP Sockets
  - ✓ Single MPI task launched on each socket
  - ✓ Parallel Threads share socket memory, e.g. 2 2-thread tasks in each node x 2 nodes
- No Shared Memory (all MPI)
  - ✓ Each core on a node is assigned an MPI task
  - ✓ Not really hybrid, e.g. 4 1-thread tasks in each node x 2 nodes



SLURM

-n : number of tasks I intend to run
-c : number of cores per task
-N : number of nodes on which distribute the tasks

Hybrid Model

Single-Threaded Messaging

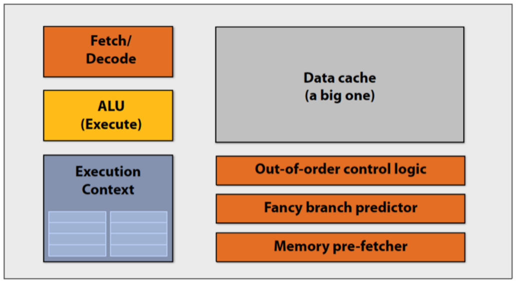
- Start with MPI initialization
- Create OMP parallel regions within MPI task (process)
  - ✓ Serial regions are the master thread or MPI task
  - ✓ MPI rank is known to all threads
- Call MPI library from serial region or a single thread within parallel region
- Finalize MPI

```
#include <mpi.h>
int main(int argc, char **argv){
int rank, size, ierr, i;
ierr= MPI_Init(&argc,&argv[]);
ierr= MPI_Comm_rank (...,&rank);
ierr= MPI_Comm_size (...,&size);
//Setup shared mem, compute & Comm
#pragma omp parallel for
  for(i=0; i<n; i++){
    <work>
  }
// compute & communicate
ierr= MPI_Finalize();
```

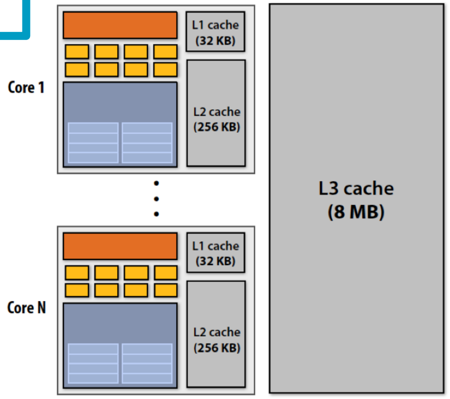
Big Compute

Path to Performance

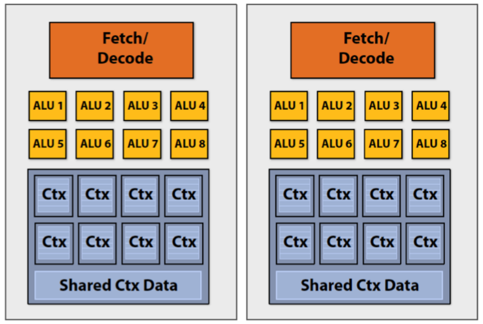
How to develop code that can make effective use of existing parallelism at different levels?



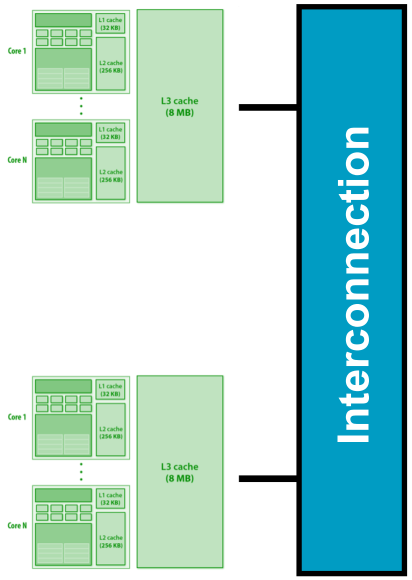
ILP/Data



Multi-core



Many-core

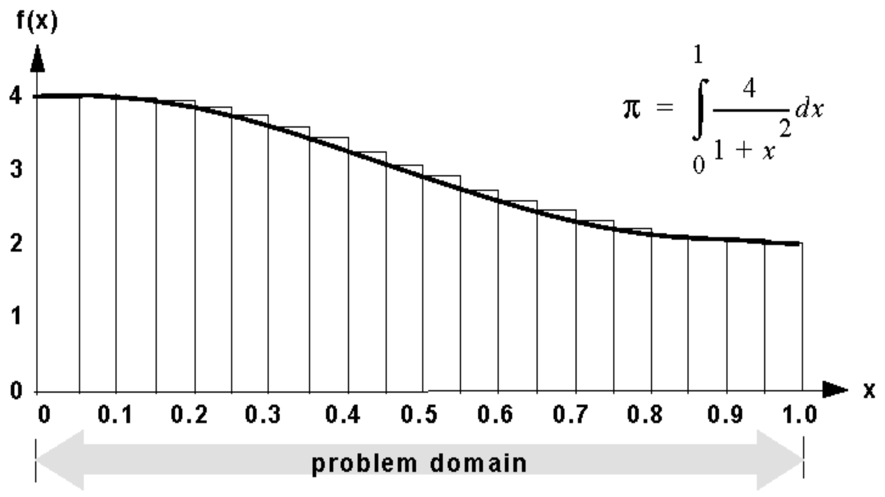


Multi-node

Big Compute

Calculating Pi

```
#include <stdio.h>
#define N 2000000000
#define vl 1024
int main(void) {
  double pi = 0.0f;
  long long i;
  for (i=0; i<N; i++) {
    double t= (double)((i+0.5)/N);
    pi +=4.0/(1.0+t*t);
}
  printf("pi=%11.10f\n",pi/N);
  return 0;
}
```



$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

Big Compute

Performance Optimization on Single-core

ILP: Loop Unrolling

```
#include <stdio.h>
#define N 2000000000
#define vl 1024
int main(void) {
  double pi = 0.0f;
  long long i;
  for (i=0; i<N; i+=2) {
    double t= (double)((i+0.5)/N);
    pi +=4.0/(1.0+t*t);
    double t= (double)((i+1+0.5)/N);
    pi +=4.0/(1.0+t*t);
  }
  printf("pi=%11.10f\n",pi/N);
  return 0;
}
```

- Level: Very fine-grained parallelism
- Overhead: Jumps
- Pros: Simple and available
- Cons: Limited scalability

Big Compute

GPU-based Accelerated Computing

OpenACC

```
#include <stdio.h>
#define N 2000000000
#define vl 1024
int main(void) {
  double pi = 0.0f;
  long long i;
  #pragma acc parallel vector_length(vl)
  #pragma acc loop reduction(+:pi)
  for (i=0; i<N; i++) {
    double t= (double)((i+0.5)/N);
    pi +=4.0/(1.0+t*t);
  }
  printf("pi=%11.10f\n",pi/N);
  return 0;
}
```

- Level: Loop-level parallelism
- Overhead: Data transfer
- Pros: Simple and best cost-performance
- Cons: Only for data parallelism

Big Compute

Shared-memory Multi-core Programming

OpenMP

```
#include <stdio.h>
#define N 2000000000
#define vl 1024
int main(void) {
  double pi = 0.0f;
  long long i;
  #pragma omp parallel for reduction(+:pi) private(i,t)
  for (i=0; i<N; i++) {
    double t= (double)((i+0.5)/N);
    pi +=4.0/(1.0+t*t);
  }
  printf("pi=%11.10f\n",pi/N);
  return 0;
}
```

- Level: Fine-grained parallelism
- Overhead: Memory sharing
- Pros: Simple and available
- Cons: Limited scalability

Big Compute

Distributed-memory Multi-mode Programming

MPI

- Level: Coarse-grained parallelism
- Overhead: Message passing
- Pros: Scalable and portable
- Cons: Complexity

```
int main(void) {
  …


  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  …
  for (i = myid + 1; i <= n; i += numprocs)
  {
    double t= (double)((i+0.5)/N);
    mypi +=4.0/(1.0+t*t);
  }
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  …
  MPI_Finalize();}
```

Big Compute

Hybrid Programming: From Bit to Application Parallelism

EXTERNAL

Coarse-grained

Application Level
- Independent applications within a system

Task Level
- Interrelated tasks within an application

Fine-grained

Procedure Level
- Regions of code within a task

Loop Level
- Iterations within a loop

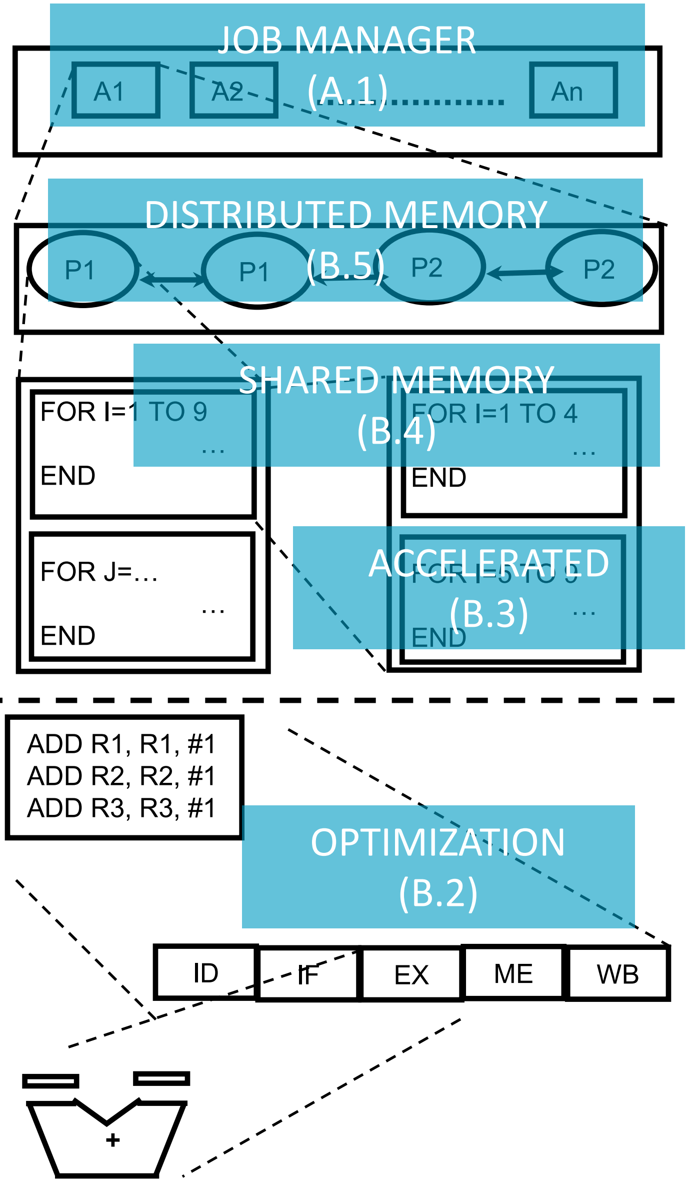
THREADS

INTERNAL

Instruction Level => SUPERSCALAR
- Instructions with an executable

Instruction Phase Level => PIPELINING
- Overlap execution of multiple instructions

Bit Level => ARITHMETIC UNIT
- Processor word size



Reading Assignments / Open Discussion

Parallel Programming at Scale: Present and Future

W. Gropp, M. Snir, *"Programming for Exascale Computers"*, Computing in Science & Engineering, 2013, 15(6), 27-35

What are the main design choices in parallel programming?

What are the existing parallel programming models?

What are the new programming models?

Next Steps

- HWB due on Monday 3/23!
- Runtime Optimization Competition! due on Wednesday 3/25!
- Get ready for tomorrow’s **lab session**
  I7 - MPI on AWS
- Get ready for third **hands-on:**
  H3. MPI Programming (on Cannon)
  <u>Check your Cannon account</u>