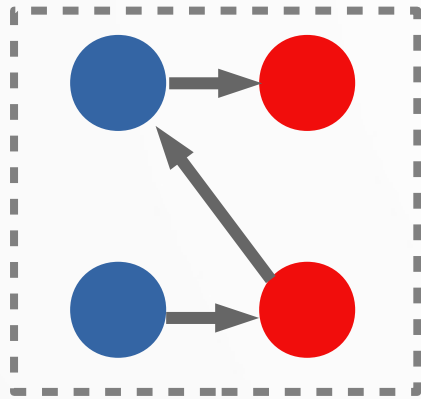


MPI Introduction

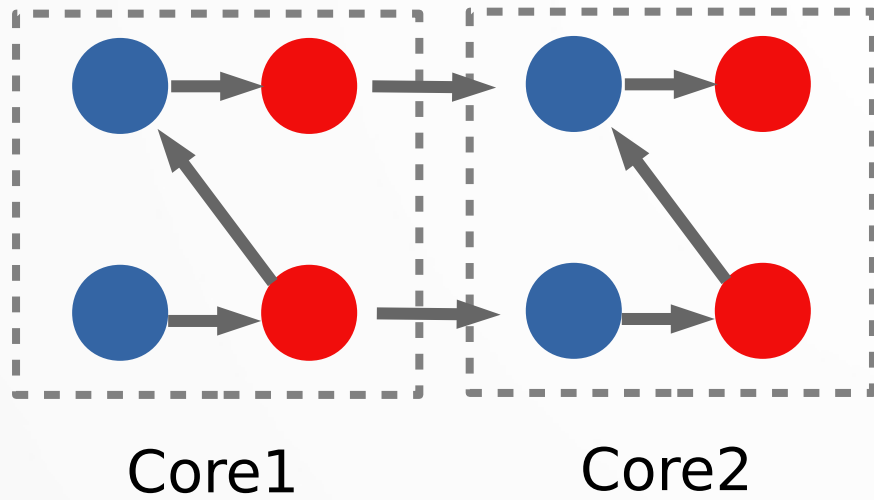
Programming Models for Emerging Platforms

Shared Memory Model

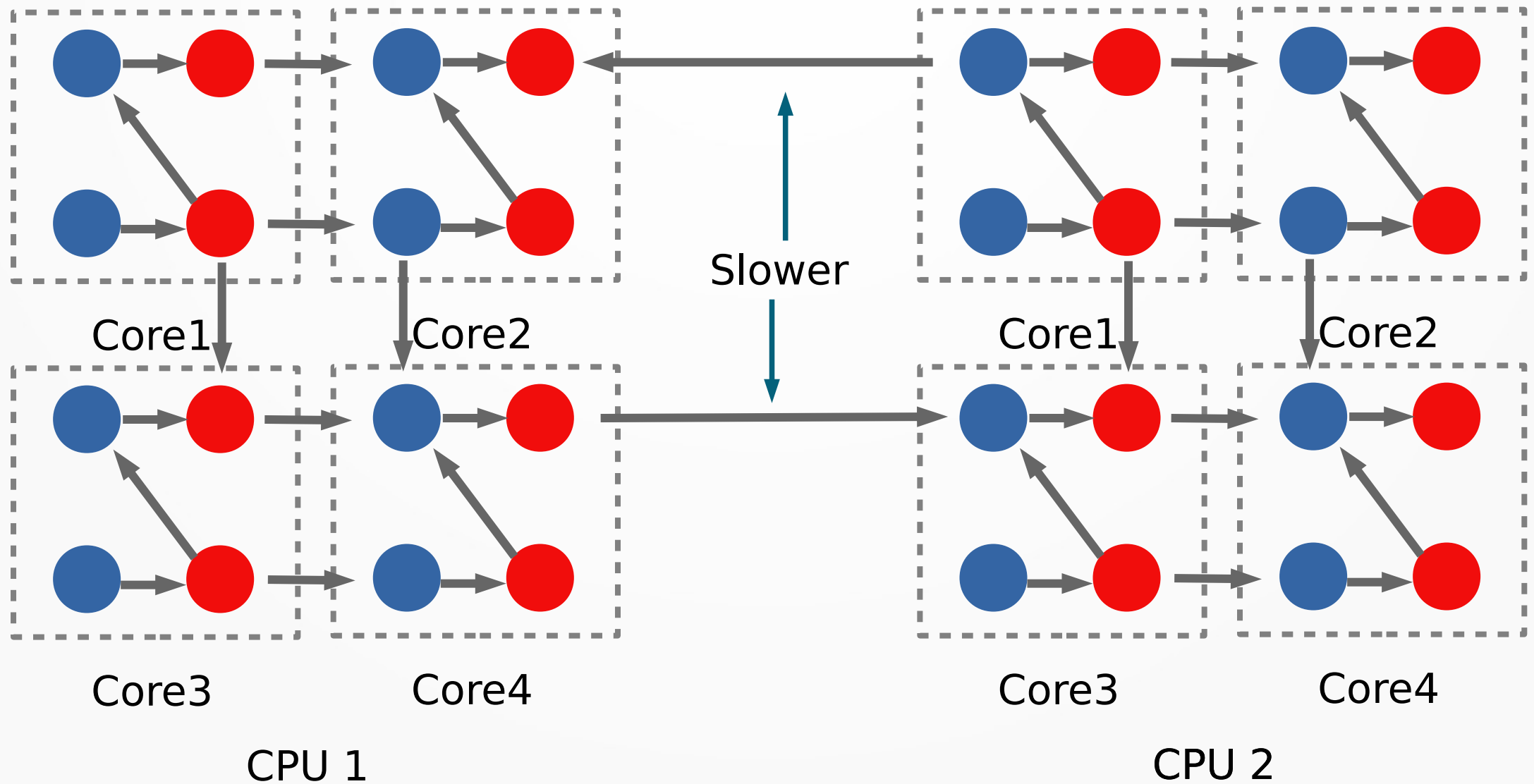


CPU1

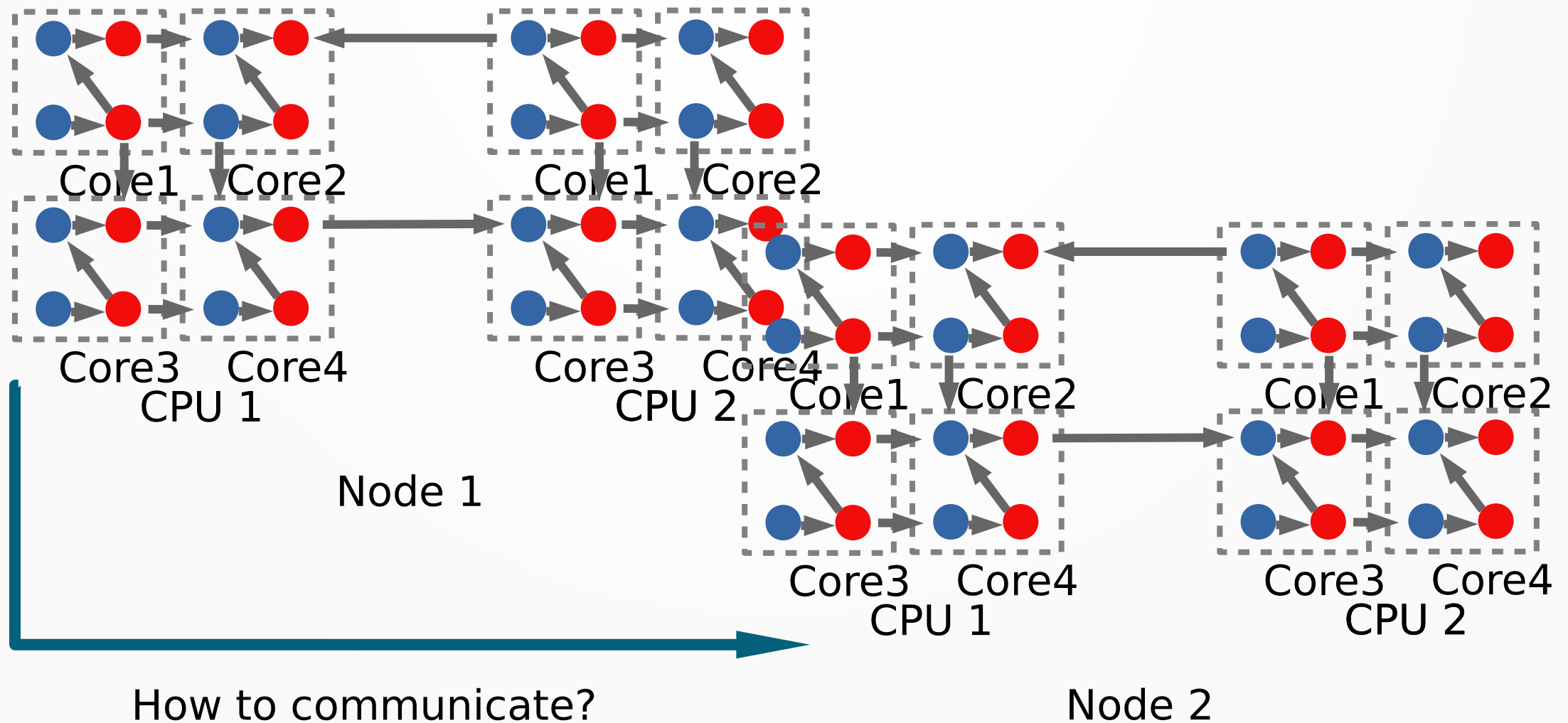
Shared Memory Model



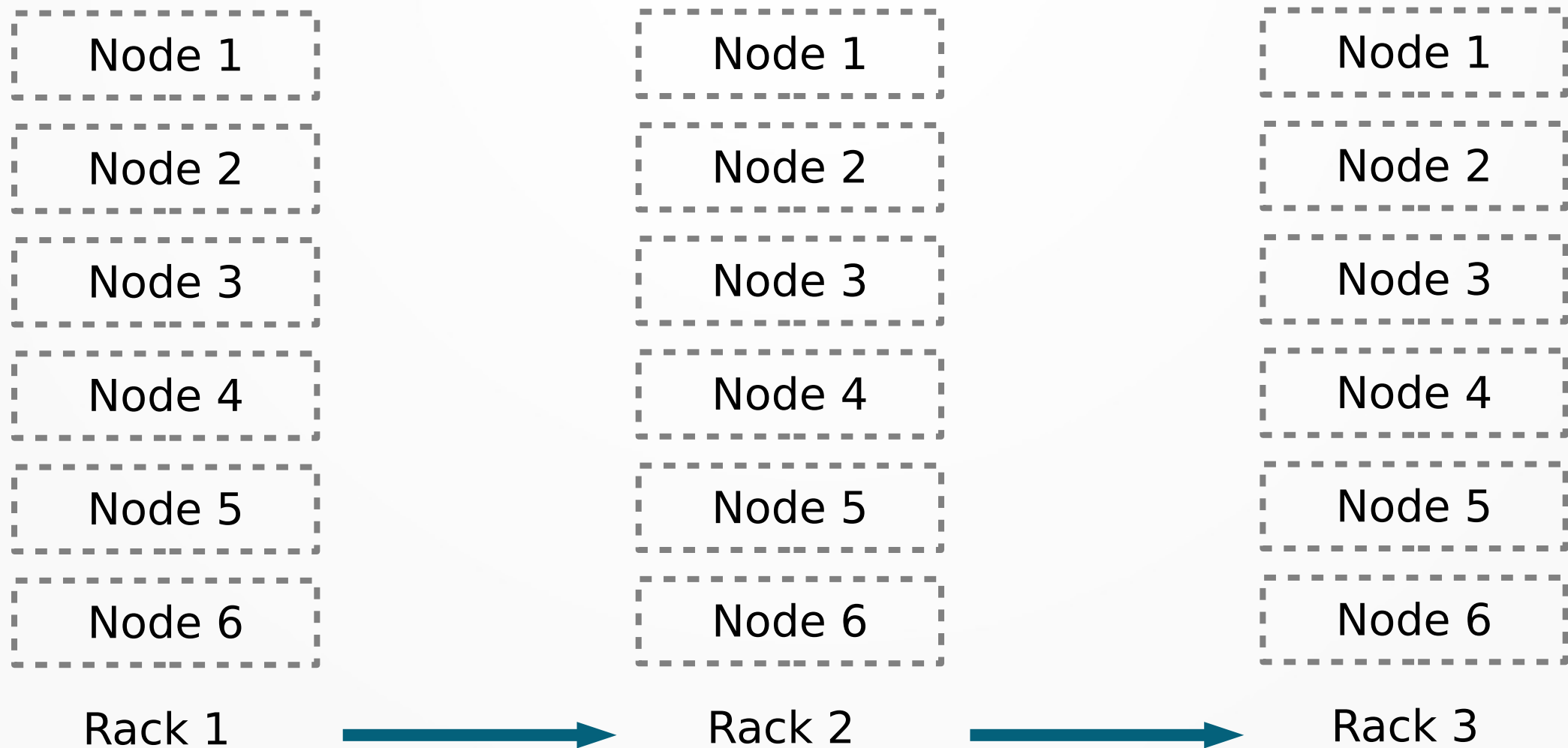
Shared Memory Model



Shared Memory Model



Shared Memory Model



Shared vs Non-Shared

- Shared

- Non-Shared

Shared vs Non-Shared

- Shared

- Immediate Updates
- Lower programming effort
(requires fewer loc / abstractions)
- Higher mental effort (consider
ALL program execution pathways)
- Local Synchronization and
Deadlock

- Non-Shared

Shared vs Non-Shared

- Shared

- Immediate Updates
- Lower programming effort (requires fewer loc / abstractions)
- Higher mental effort (consider ALL program execution pathways)
- Local Synchronization and Deadlock

- Non-Shared

- Must communicate updates
- Higher programming effort (need to send/gather data)
- Lower mental effort (consider each process execution pathway in isolation)
- Global synchronization and deadlock

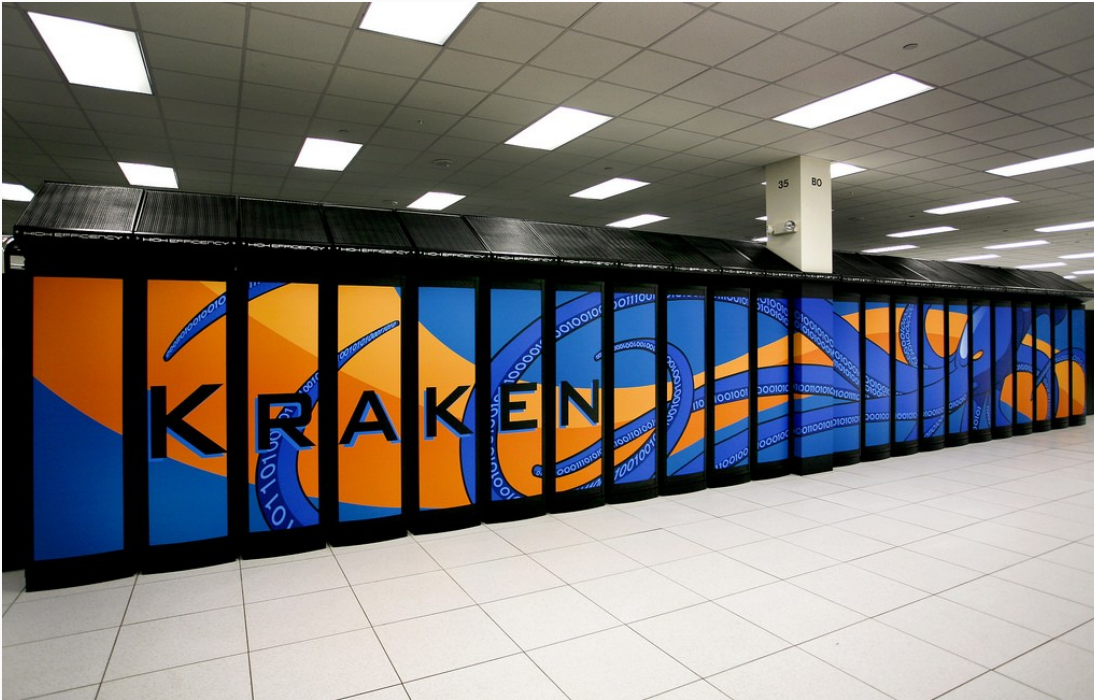
Shared vs Non-Shared

- From a PL model perspective
 - Trade-offs associated with both (although Non-Shared most certainly provides a better abstraction)
 - We will revisit

Shared vs Non-Shared

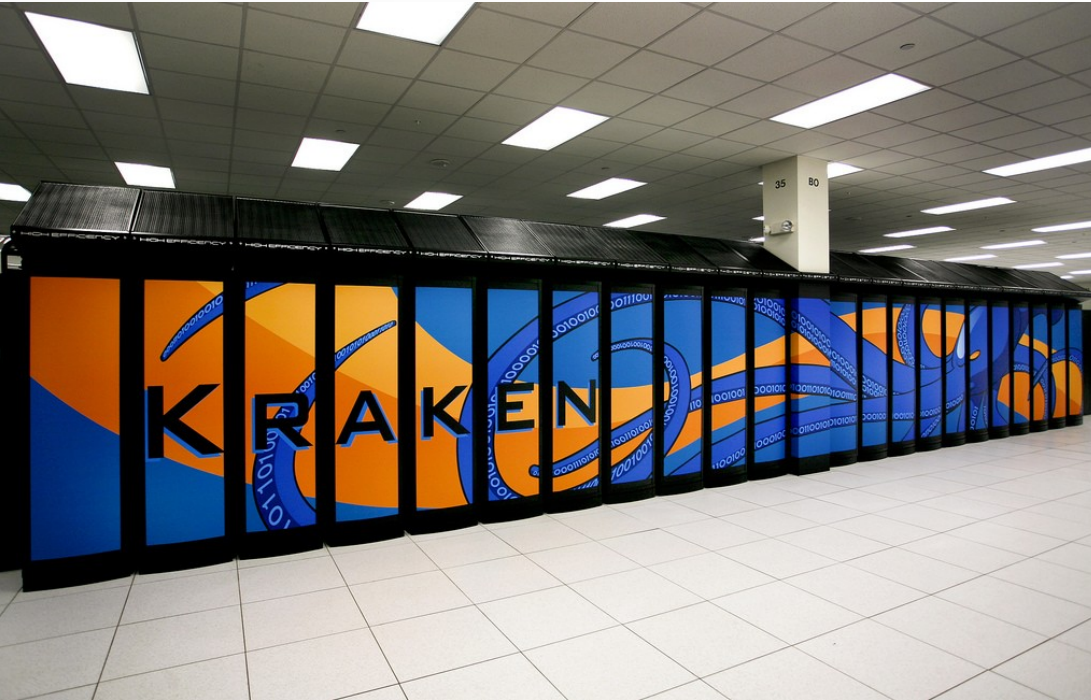
- From a PL model perspective
 - Trade-offs associated with both (although Non-Shared most certainly provides a better abstraction)
 - We will revisit
- From a implementation perspective
 - Two models are complementary (shared for *intraprocess*, non-shared for *interprocess*)

High Performance Computing



- 18,816 Processors
- 112,896 Cores
- 9,408 Nodes
- 147 TB Memory

High Performance Computing



- 18,816 Processors
- 112,896 Cores
- 9,408 Nodes
- 147 TB Memory

How do we program this machine?

Message Passing Interface (MPI)

- *Standard* interface for programming large-scale parallel platforms
 - Specific implementations exist, such as mpich, openmpi
 - Easily the most widely used parallel programming interface

Message Passing Interface (MPI)

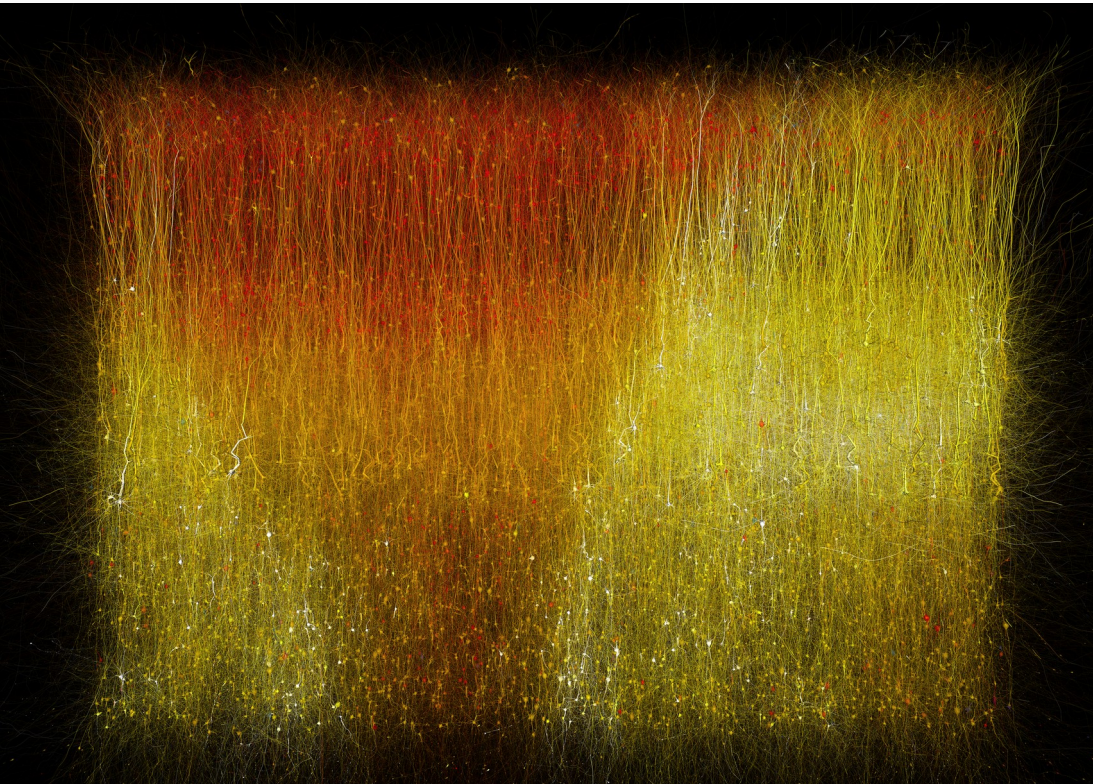
- *Standard* interface for programming large-scale parallel platforms
 - Specific implementations exist, such as mpich, openmpi
 - Easily the most widely used parallel programming interface
- Essentially, processes send and recv messages with MPI_Send, MPI_Recv
 - Lots of features to ease the “parallel process” part

A Word of Caution

Table 2 Shortest path computation times comparing a GPGPU with a parallel implementation of the $O(n^2)$ Bellman-Ford algorithm, and a serial CPU using either Bellman-Ford or the $O(n \log n)$ Dijkstra algorithm.

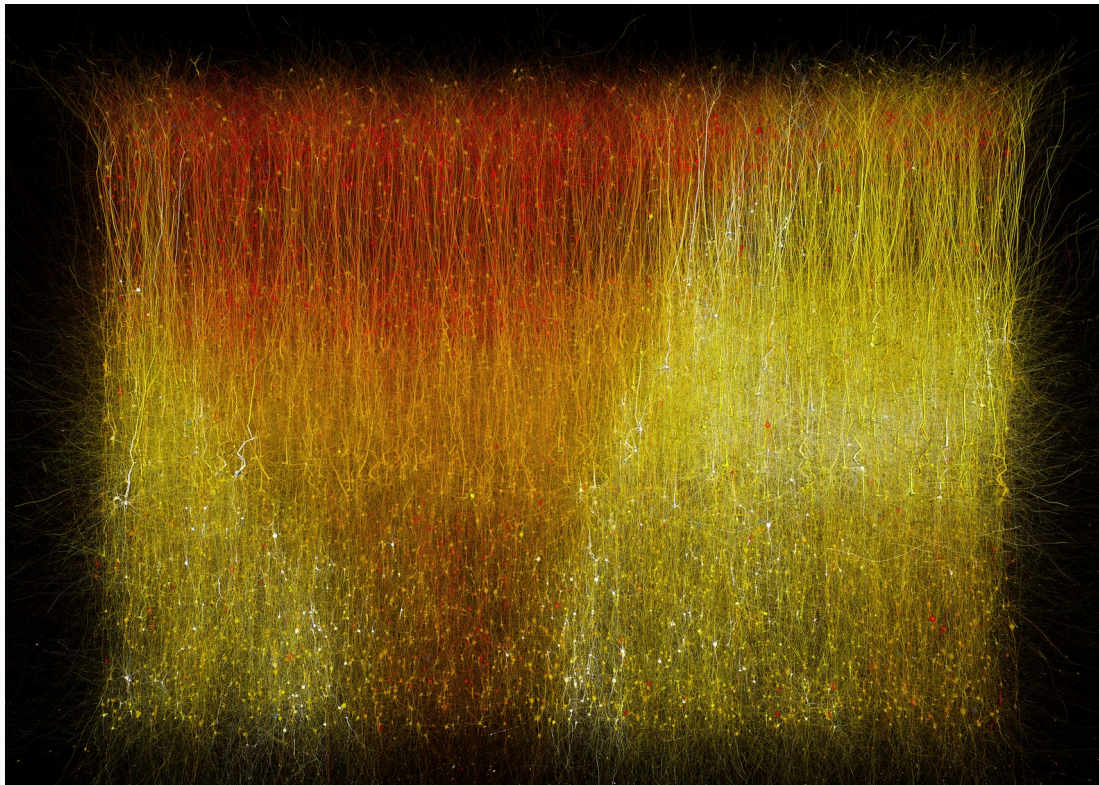
	CPU Bellman-Ford	GPGPU Bellman-Ford	CPU Dijkstra
Bunny (35k vertices)	2.9s	0.06s	0.08s
Buddha (544k vertices)	62.5s	3s	1.6s

Blue Brain Project



- Digital reconstruction of brain
- Mathematical models for individual neurons and synapses
- 100 billion neurons, 100 trillion synapses

Blue Brain Project



- Digital reconstruction of brain
- Mathematical models for individual neurons and synapses
- 100 billion neurons, 100 trillion synapses

Simulation fits parallel programming!

How does MPI work?

```
mpirun -n 2 ./a.out arg1 arg2
```

How does MPI work?

```
mpirun -n 2 ./a.out arg1 arg2
```

./a.out arg1 arg2
Rank 0

./a.out arg1 arg2
Rank 1

How does MPI work?

```
mpirun -n 8 ./a.out arg1 arg2
```

How does MPI work?

```
mpirun -n 8 ./a.out arg1 arg2
```

./a.out arg1 arg2
Rank 0

./a.out arg1 arg2
Rank 2

./a.out arg1 arg2
Rank 4

./a.out arg1 arg2
Rank 7

./a.out arg1 arg2
Rank 6

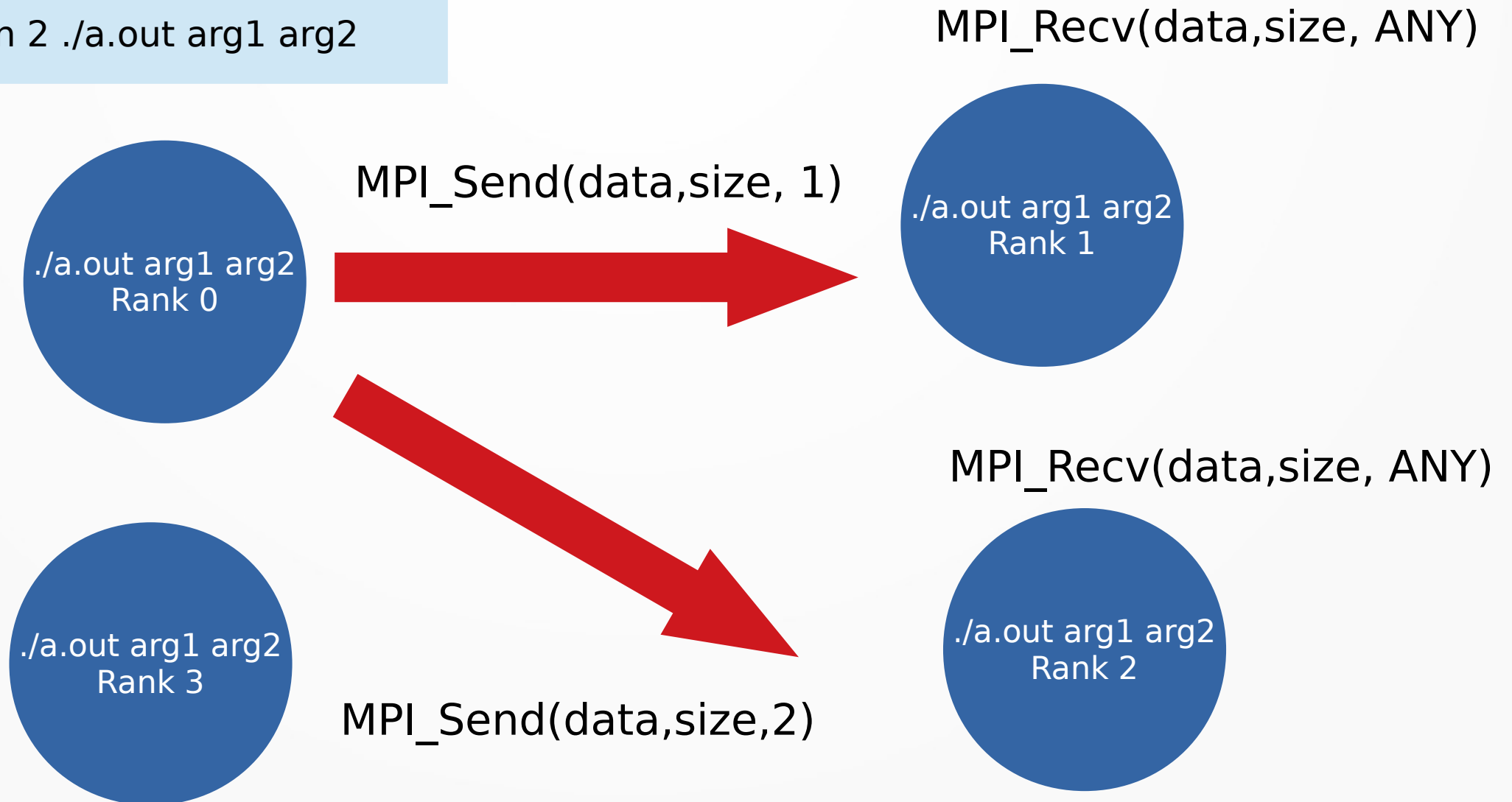
./a.out arg1 arg2
Rank 1

./a.out arg1 arg2
Rank 3

./a.out arg1 arg2
Rank 5

How does MPI work?

```
mpirun -n 2 ./a.out arg1 arg2
```



How does MPI work?

```
mpirun -n 2 ./a.out arg1 arg2
```

MPI_Recv(data,size, ANY)

No sockets
No connecting
Runtime takes care of message communication

Rank 3

MPI_Send(data,size,2)

a,size, ANY)

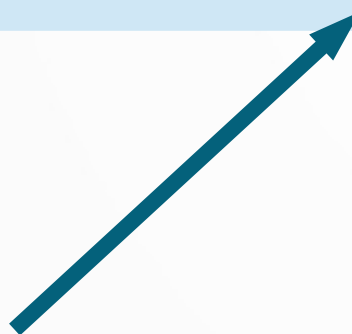
How does MPI work?

```
int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm com)
```


How does MPI work?

```
int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm com)
```

count refers to
number of type
items




MPI_CHAR
MPI_INT
MPI_FLOAT
...



How does MPI work?

```
int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm com)
```

count refers to
number of type
items



MPI_CHAR
MPI_INT
MPI_FLOAT
...

rank (id) of process
to send to

a way to “type” messages

How does MPI work?

```
int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm com)
```

Group of MPI processes



The diagram consists of two teal arrows. The first arrow originates from the text 'Group of MPI processes' and points diagonally upwards and to the right, terminating at the **MPI_Comm** parameter in the function signature. The second arrow originates from the text 'Default MPI_COMM_WORLD' and points diagonally upwards and to the left, also terminating at the **MPI_Comm** parameter.

Default MPI_COMM_WORLD

How does MPI work?

```
int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm com)
```

Group of MPI processes



Default MPI_COMM_WORLD

If MPI_Send succeeds, all data gets sent!

How does MPI work?

```
int MPI_Recv(void *b, int count, MPI_Datatype type, int src, int tag, MPI_Comm com, MPI_Status *stat)
```

rank of process to recv
from (can use MPI_ANY_SOURCE)

The diagram consists of two teal-colored arrows. The first arrow originates from the text 'rank of process to recv from (can use MPI_ANY_SOURCE)' and points diagonally upwards and to the right, terminating at the 'src' parameter in the function signature. The second arrow originates from the text 'type of message to recv (can use MPI_ANY_TAG)' and points diagonally upwards and to the right, terminating at the 'type' parameter in the function signature.

type of message to recv
(can use MPI_ANY_TAG)

How does MPI work?

```
int MPI_Recv(void *b, int count, MPI_Datatype type, int src, int tag, MPI_Comm com, MPI_Status *stat)
```

rank of process to recv
from (can use MPI_ANY_SOURCE)



type of message to recv
(can use MPI_ANY_TAG)

get recv information such as:
number items recv,
sending process,
sending tag etc

How does MPI work?

```
int MPI_Recv(void *b, int count, MPI_Datatype type, int src, int tag, MPI_Comm com, MPI_Status *stat)
```

rank of process to recv
from (can use MPI_ANY_SOURCE)

type of message to recv
(can use MPI_ANY_TAG)

get recv information such as:
number items recv,
sending process,
sending tag etc

May receive *up to* **count** items

How does MPI work?

```
int MPI_Recv(void *b, int count, MPI_Datatype type, int src, int tag, MPI_Comm com, MPI_Status *stat)
```

rank of process to recv
from (can use MPI_ANY_SOURCE)



type of message to recv
(can use MPI_ANY_TAG)

get recv information such as:
number items recv,
sending process,
sending tag etc

But no partial receives (if you can't recv
at least count items, error)

Send/Recv

- Example (hello.c, fib.c)

Practice

- circular.c on website
- Convert the circular prime search from Cilk into an MPI program
- Make a master split work, workers perform search and return found primes to master

For Reference / Self Study

- MPI Functions
 - MPI_Init
 - MPI_Comm_rank
 - MPI_Comm_size
 - MPI_Recv
 - MPI_Send
 - MPI_Finalize
- Compiling and linking
 - Compile: mpicc hello.c
 - Run: mpirun -n 4 ./a.out
- MPI Structs
 - MPI_Datatype
 - MPI_Status

Acknowledgments

- Dispelling the Myths of Parallel Computing
 - Madden
 - <http://www.cs.binghamton.edu/~pmadden/pubs/dispelling-ieeeedt-2013.pdf>
- Introduction to the Message Passing Interface (Irish Centre for High-End Computing (ICHEC) www.ichec.ie)
- A Comprehensive MPI Tutorial Resource (<http://mpitutorial.com/>)
- [Parallel Monte Carlo Computation](#)