NORTHWEST INSTITUTE for ADVANCED COMPUTING

Pacific Northwest NATIONAL LABORATORY
Proudly Operated by Battelle for the U.S. Department of Energy

UNIVERSITY of WASHINGTON

AMATH 483/583
High Performance Scientific Computing

Lecture 17:
Distributed memory, communicating sequential processes

Andrew Lumsdaine
Northwest Institute for Advanced Computing
Pacific Northwest National Laboratory
University of Washington
Seattle, WA

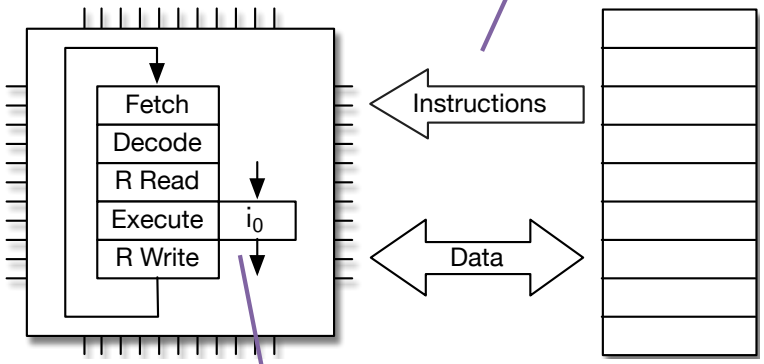
Overview

- Distributed memory systems
- Communicating sequential processes
- Message passing
- The message passing interface

Scaling progression of CPUs

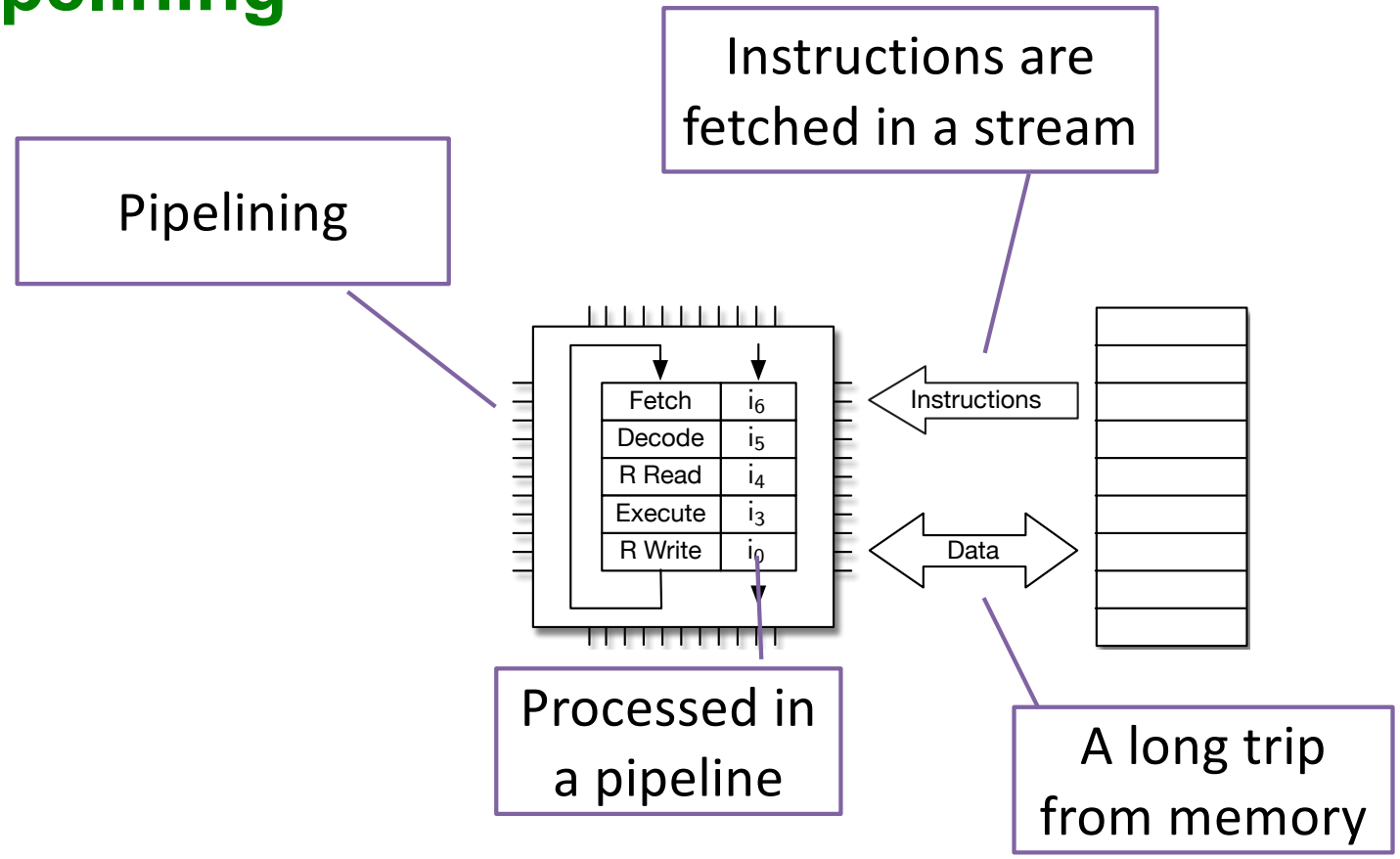
Simplest model

CPU fetches and executes instructions



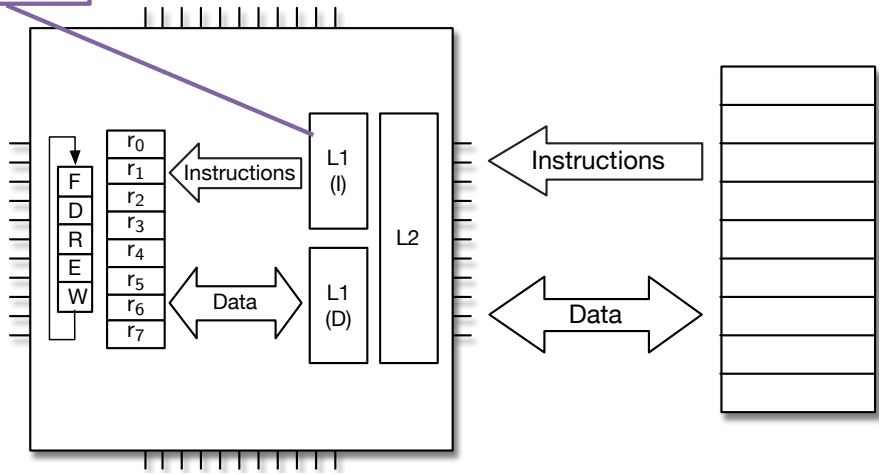
Many cycles per instruction

Pipelining


Pipelining
Instructions are fetched in a stream
Fetch
Decode
R Read
Execute
R Write
i_6
i_5
i_4
i_3
i_0
Instructions
Data
Processed in a pipeline
A long trip from memory

Hierarchical memory

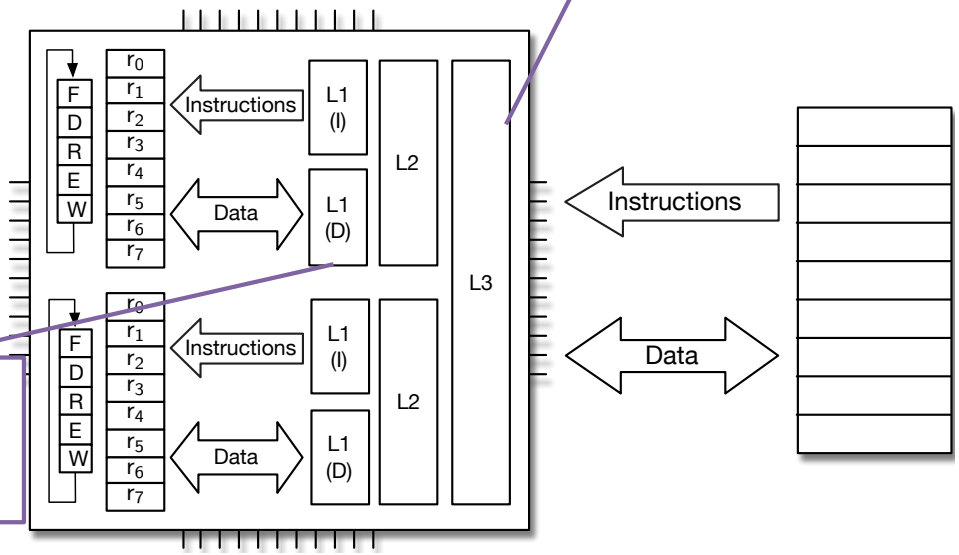
Use special, fast memory to keep data and instructions close



Multicore CPUs

Replicate 2X

Cores share slower memory



Caches need to be kept coherent

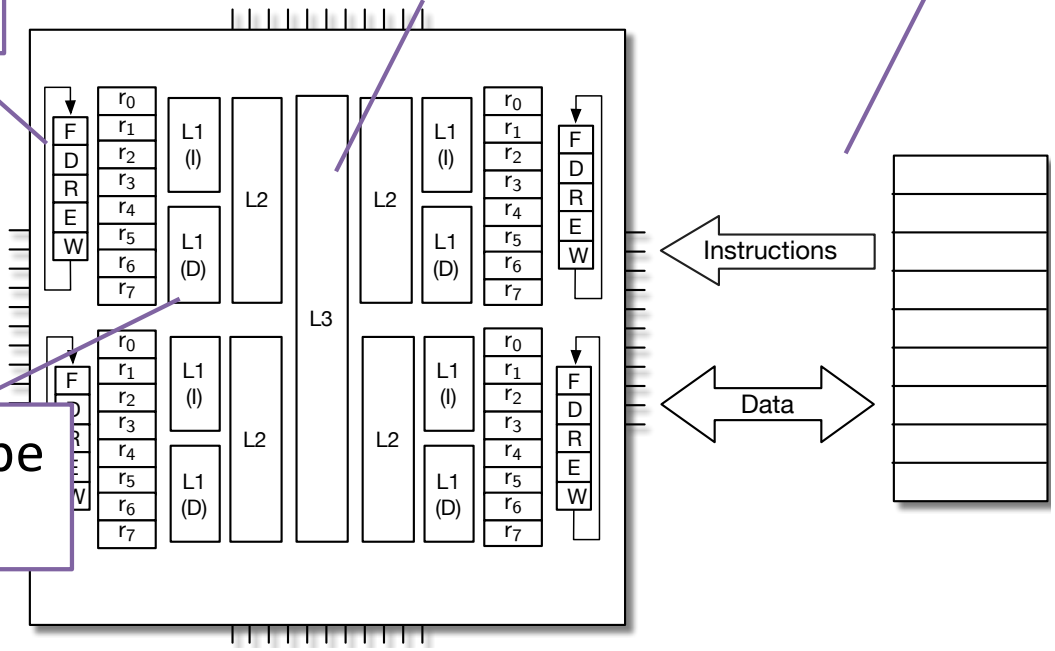
Even more cores

Replicate 4X

Cores share slower memory

Include super-slow DRAM

Caches need to be kept coherent

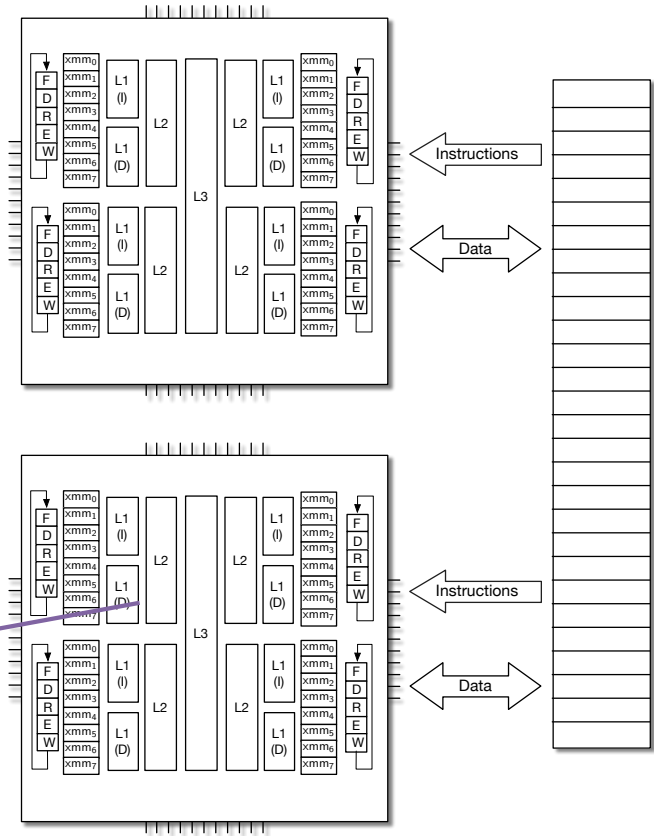


Symmetric Multi-Processor (SMP)

Multiple CPU chips

AKA “sockets”

Caches still need to be kept (somewhat) coherent



Memory may be uniformly shared among sockets

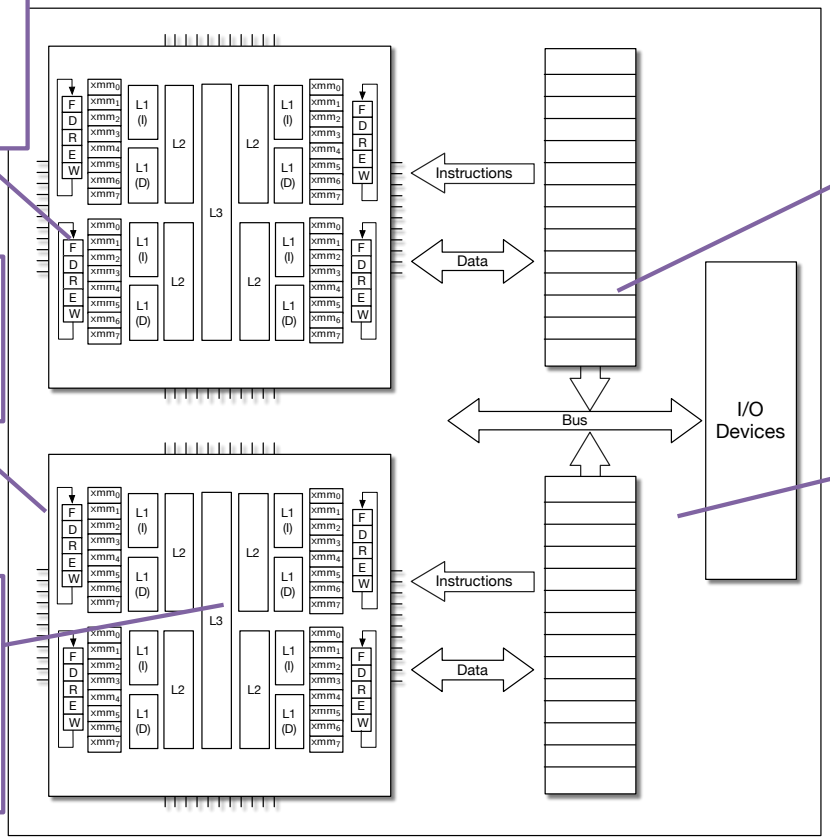
Uniform memory access (UMA)

Asymmetric

Multiple CPU chips

AKA “sockets”

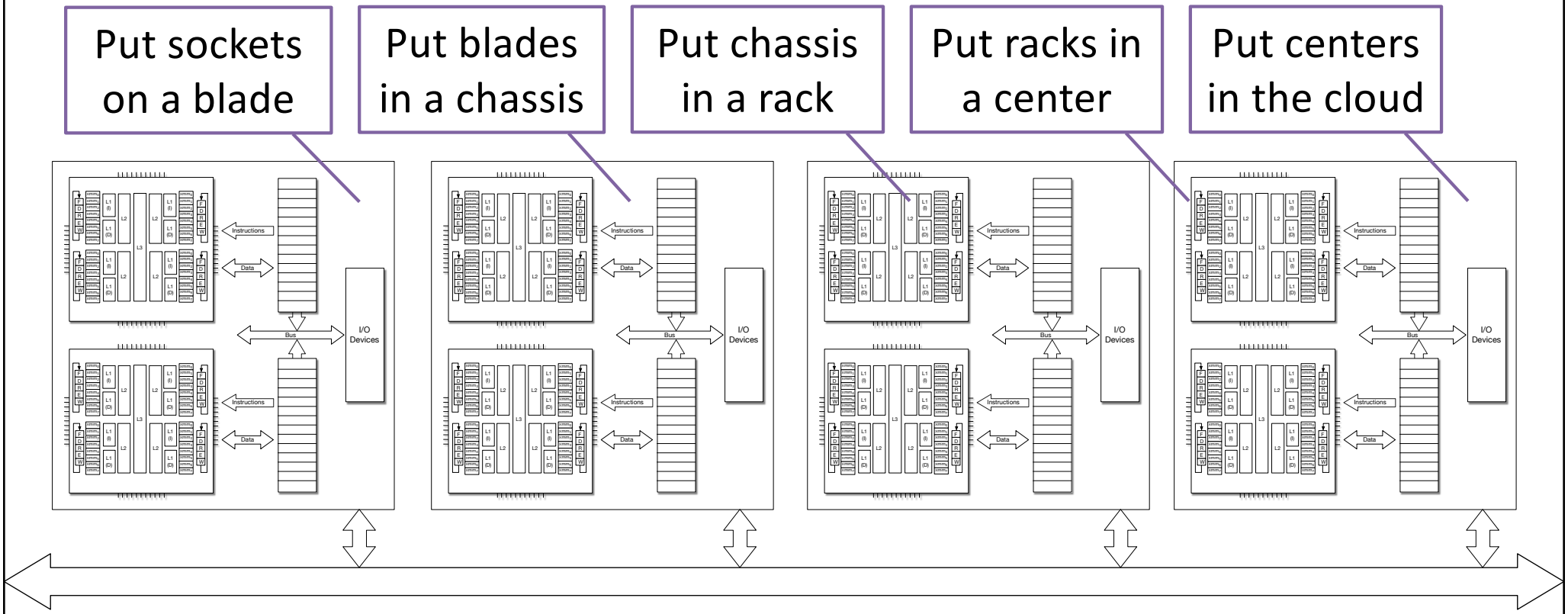
Caches still need to be kept (somewhat) coherent: CC-NUMA



Memory may be non-uniformly shared among sockets

Non-uniform memory access (NUMA – most common)

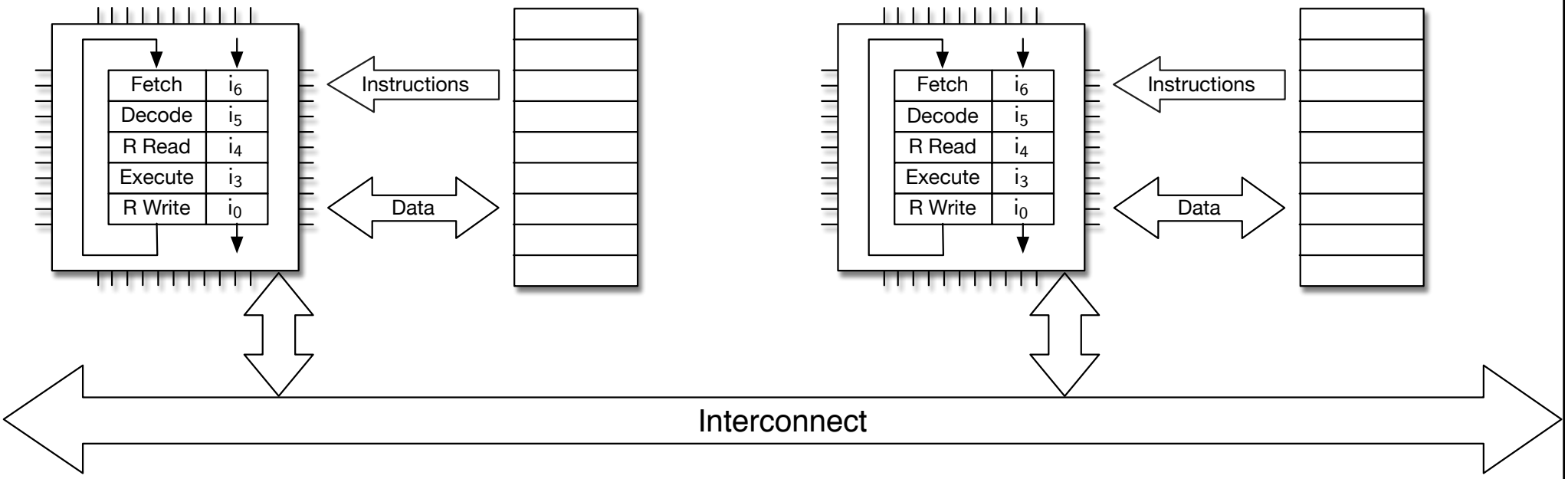
The Next Step



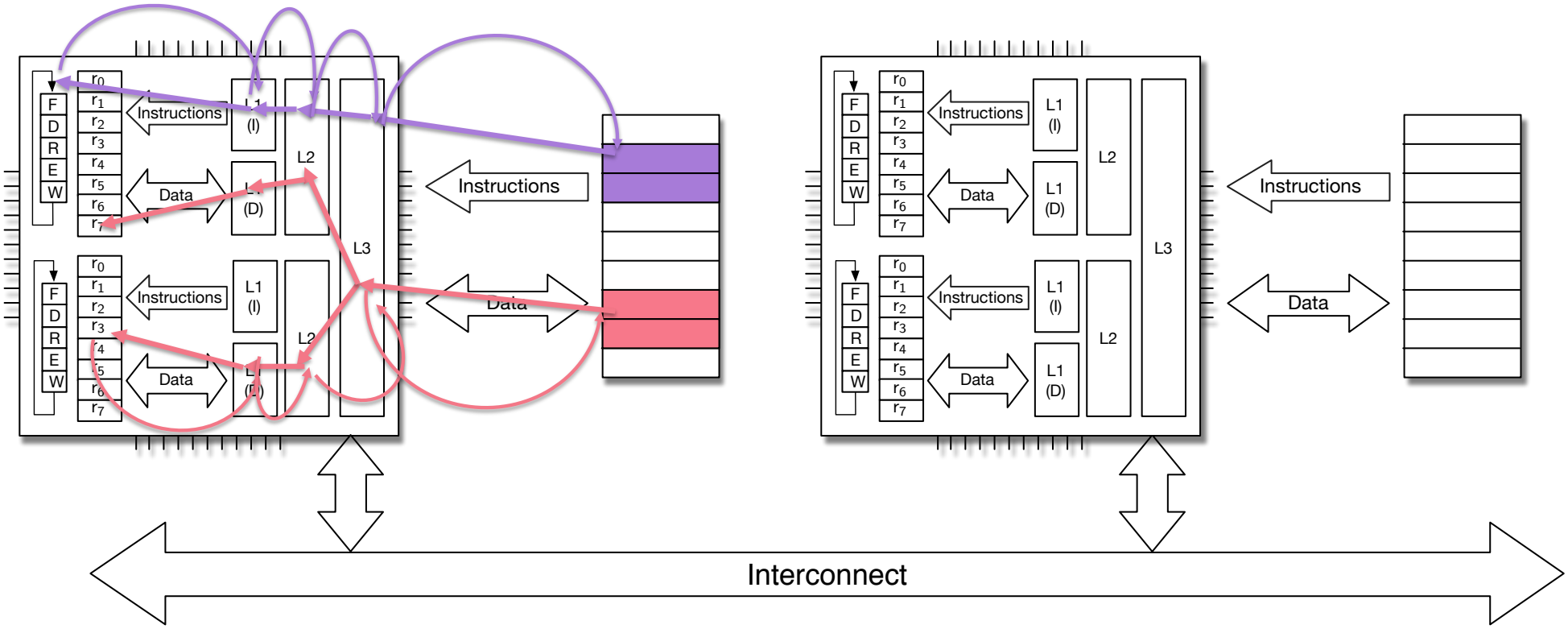
Then you have a supercomputer

But how do you use it?

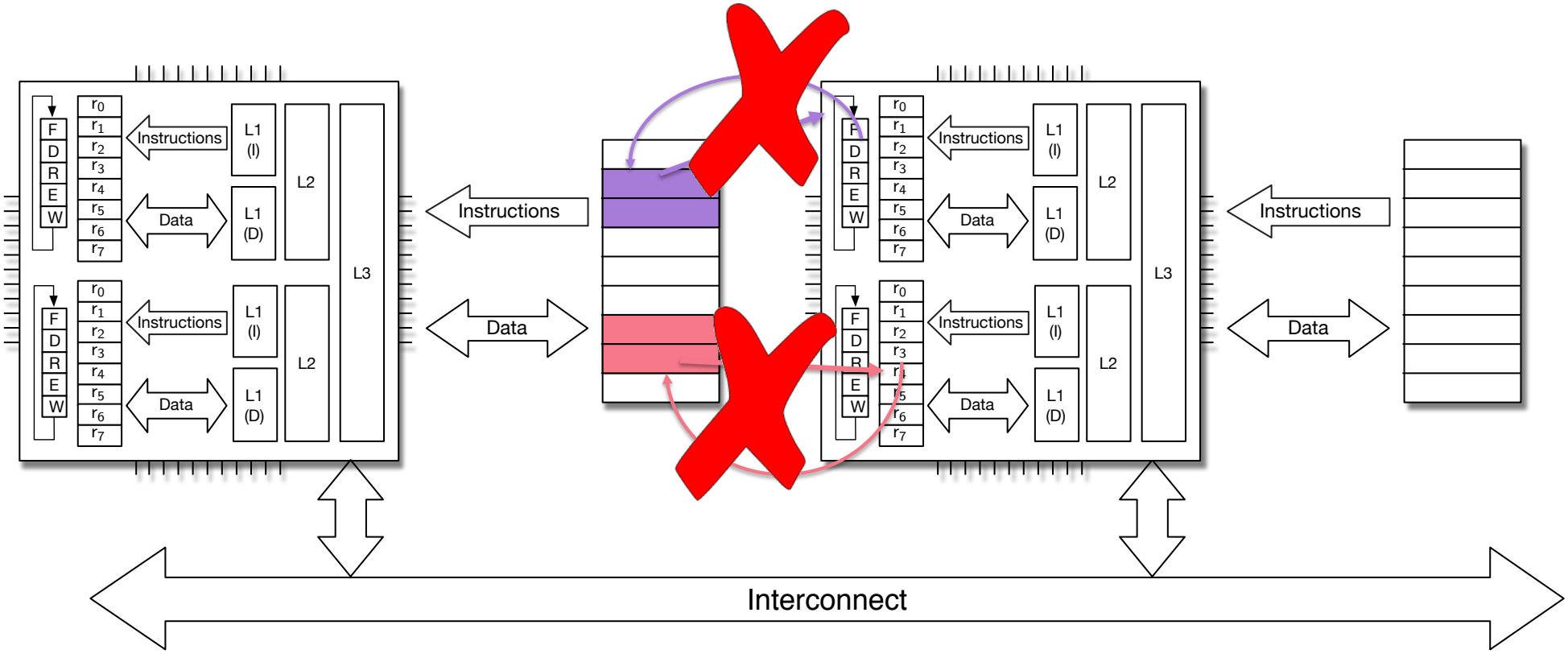
Distributed memory



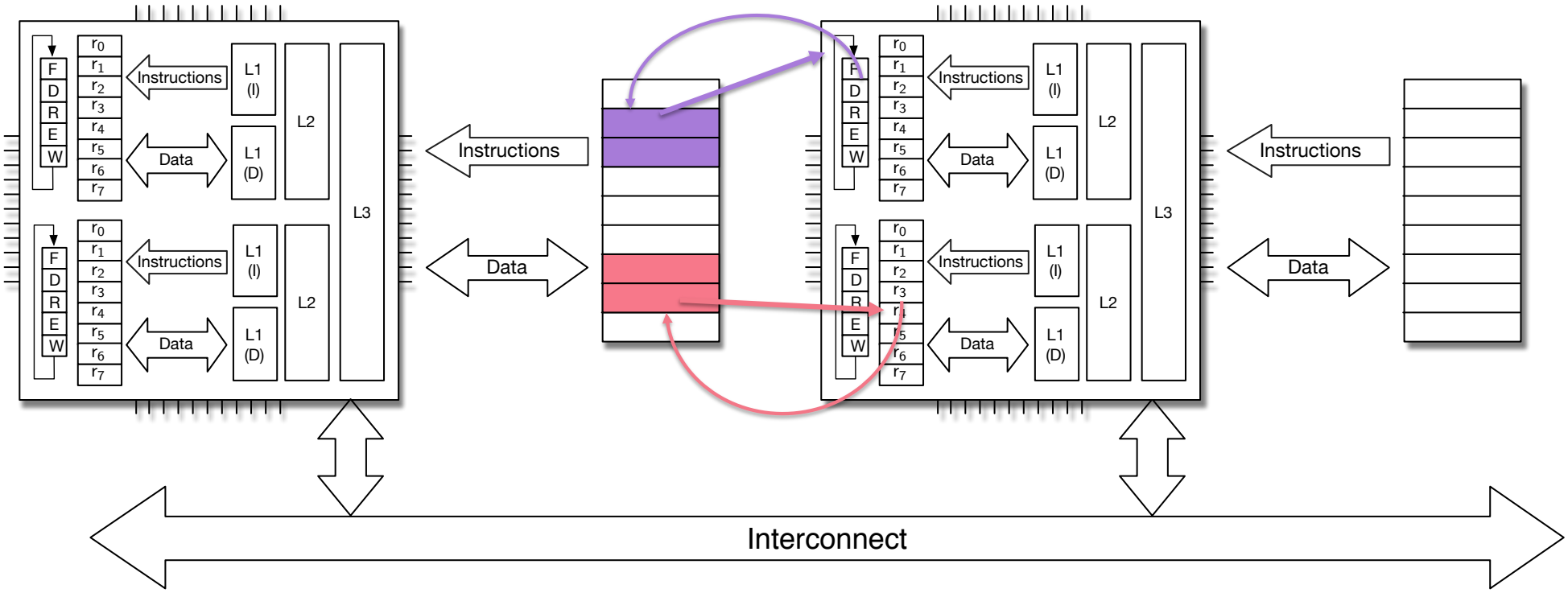
Distributed memory



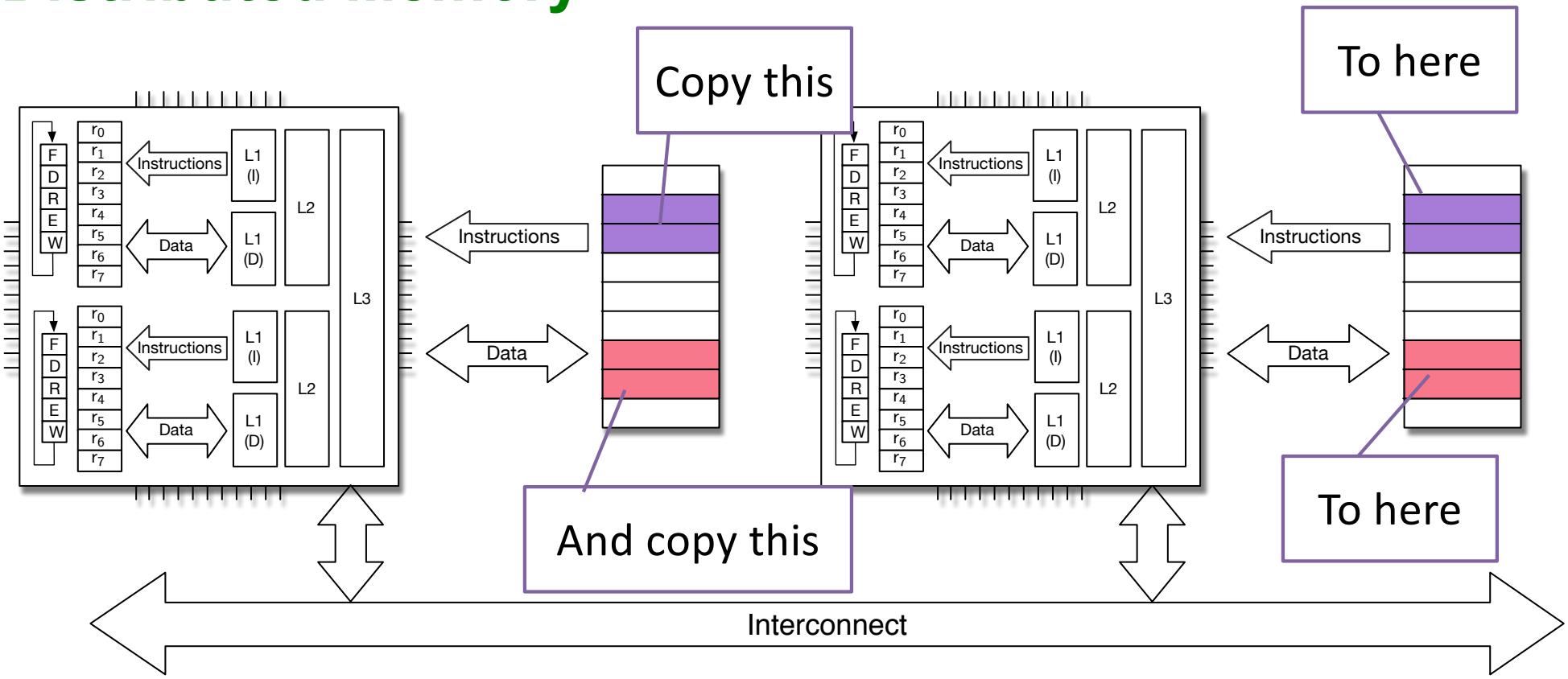
Distributed memory



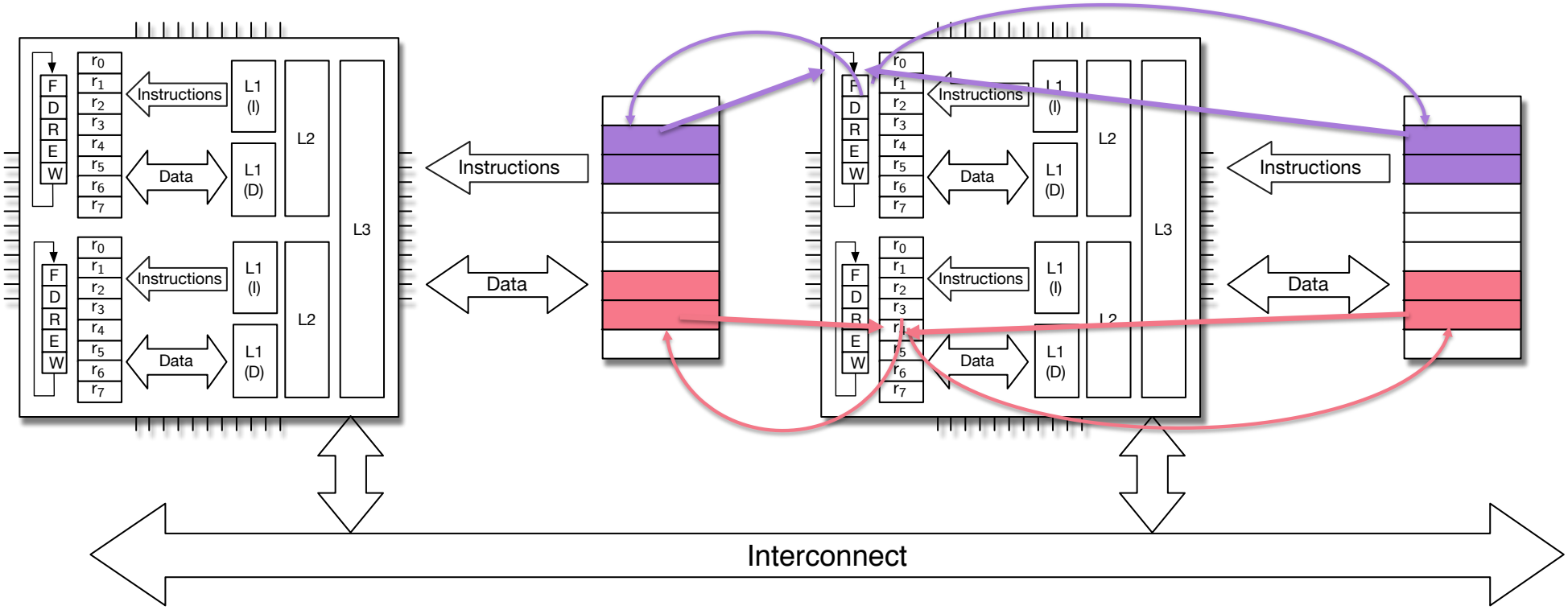
Distributed memory



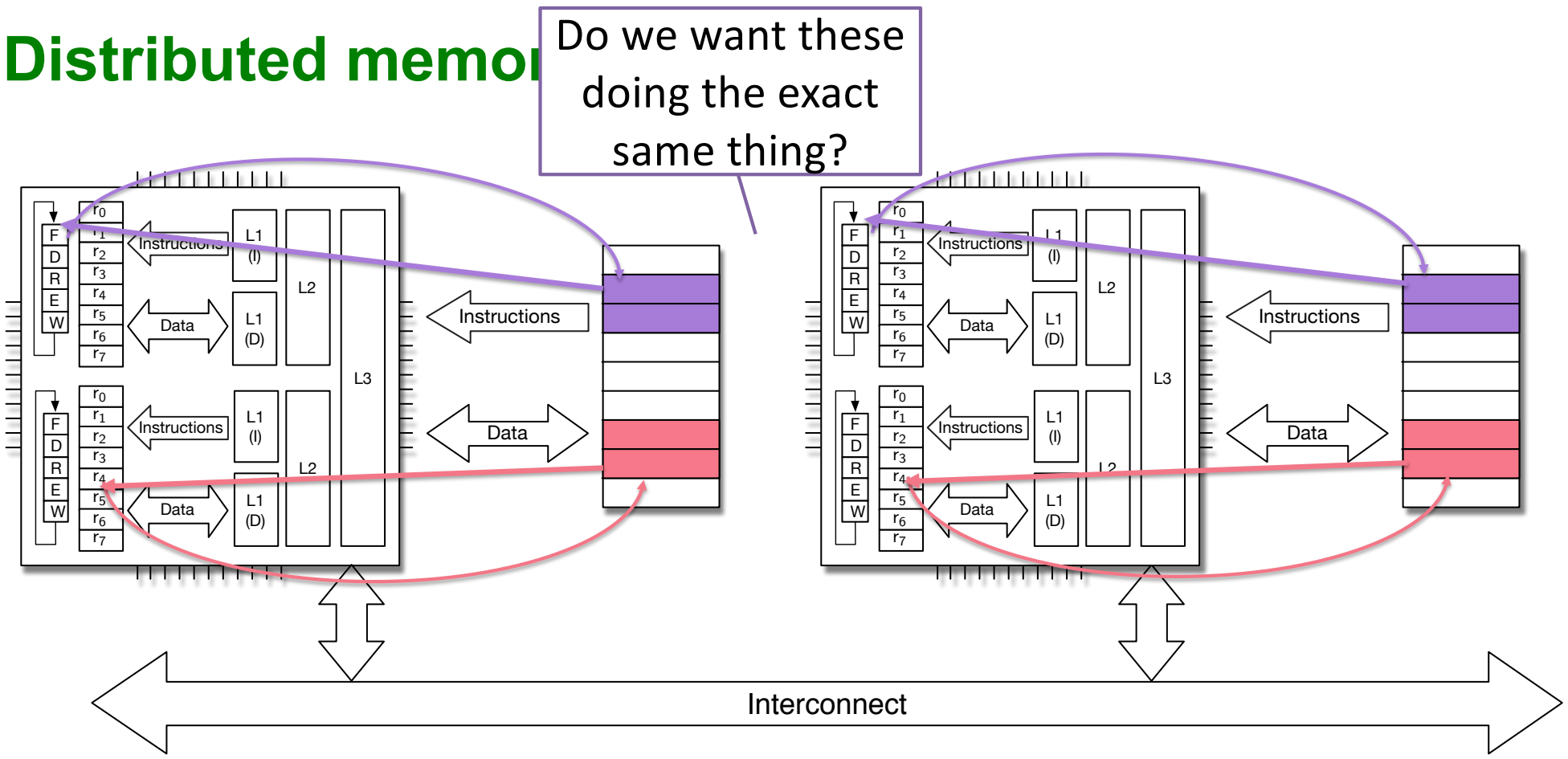
Distributed Memory



Distributed memory



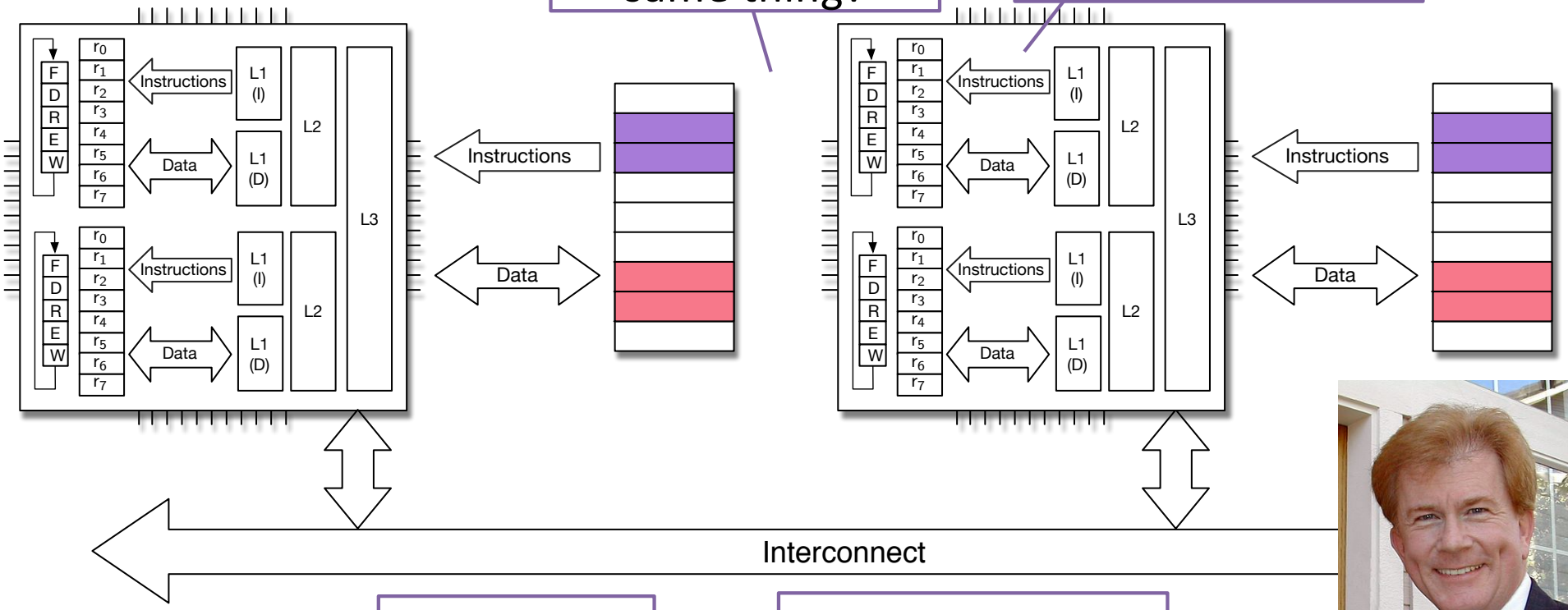
Distributed memor



Distributed memor

Do we want these doing the exact same thing?

Will there be speedup if we do?



What was his law?

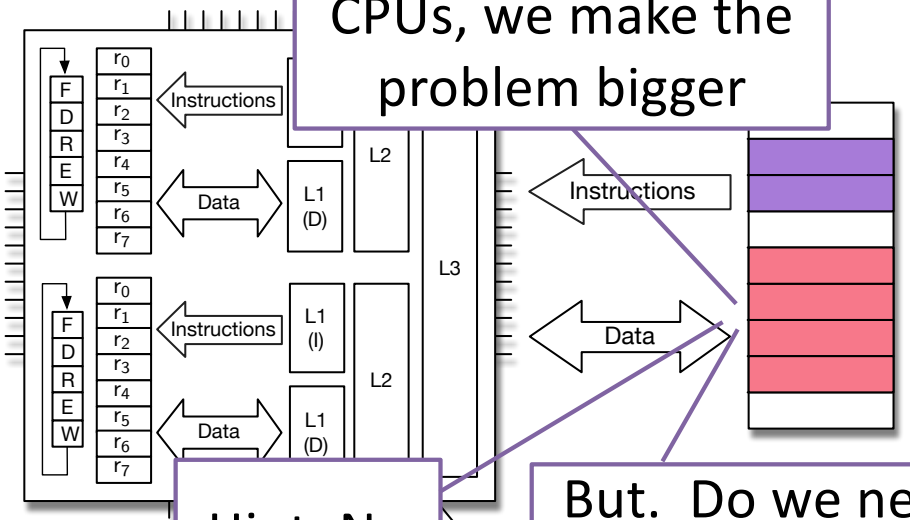
Remember this famous person?

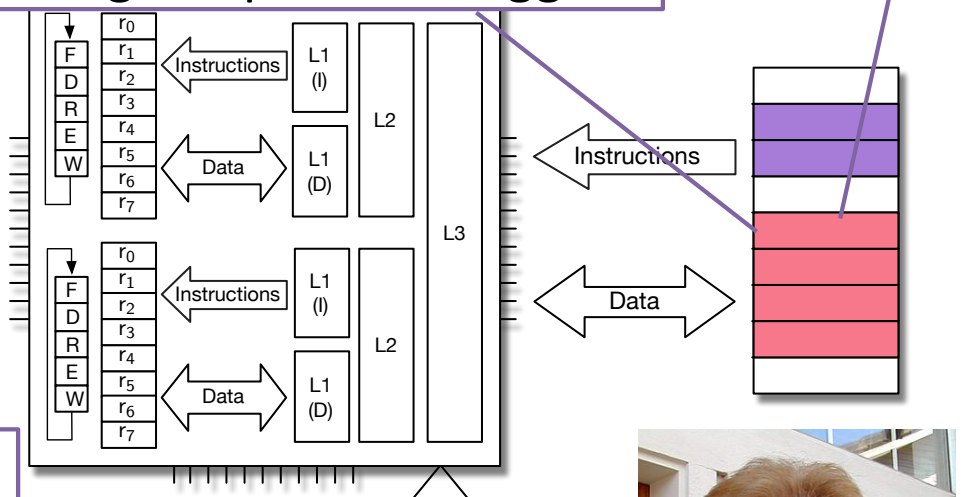
Distributed memory

As we add more CPUs, we make the problem bigger

Can we keep all the data on every node if we keep making the problem bigger?

Hint: No





Hint: No

But. Do we need all the data on every node?

Interconnect

What was his law?

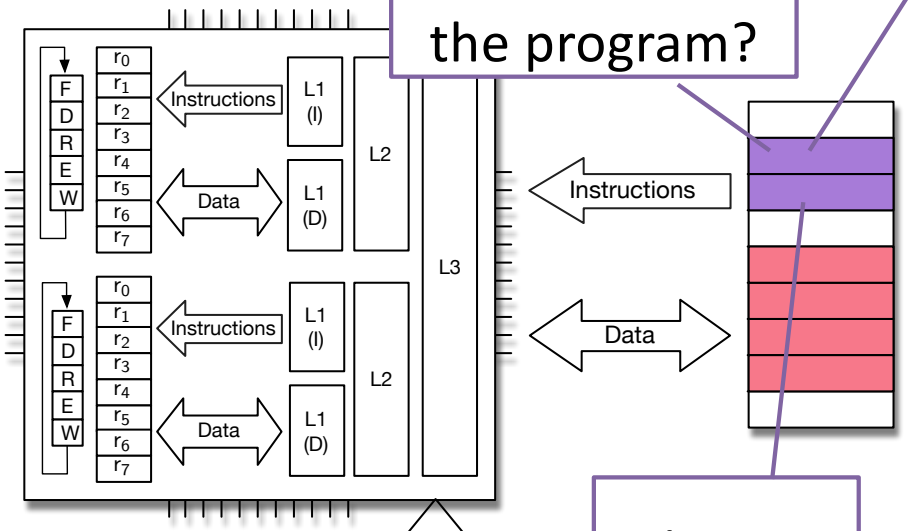
Remember this famous person?

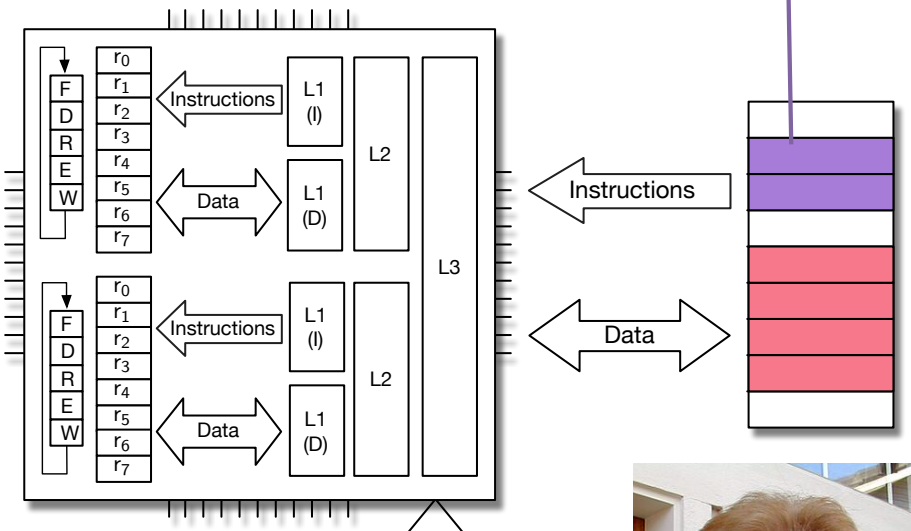
Distributed memory

Does it grow with problem size?

And we probably need all of it

What about the program?





Hint: No

Interconnect

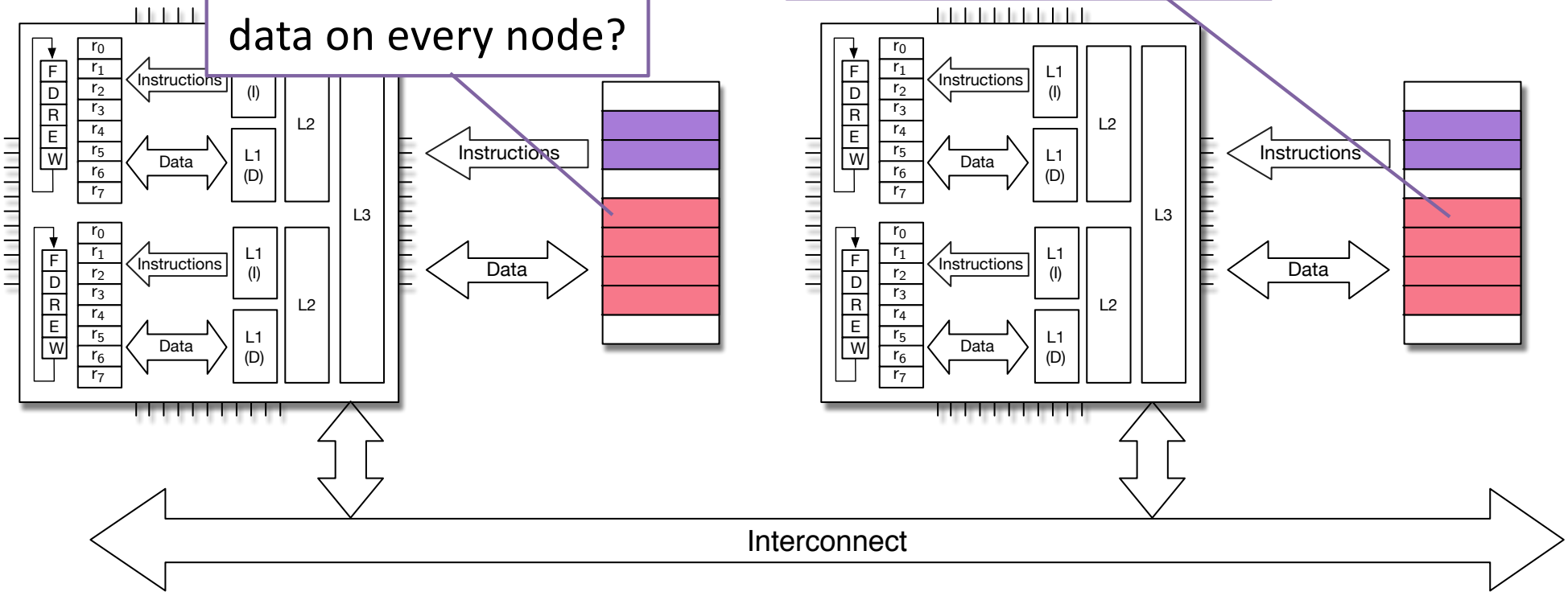
What was his law?

Remember this famous person?

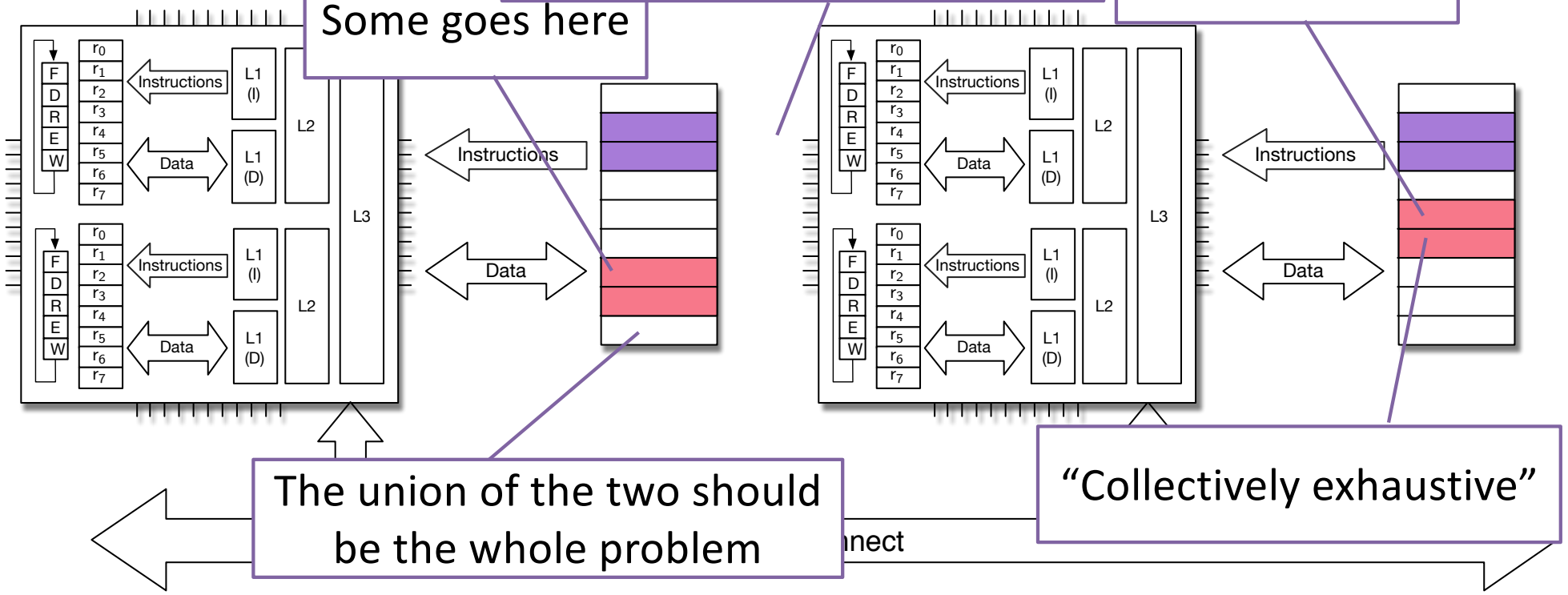
Distributed memory

Do we need all the data on every node?

What do we keep?
What do we not keep?

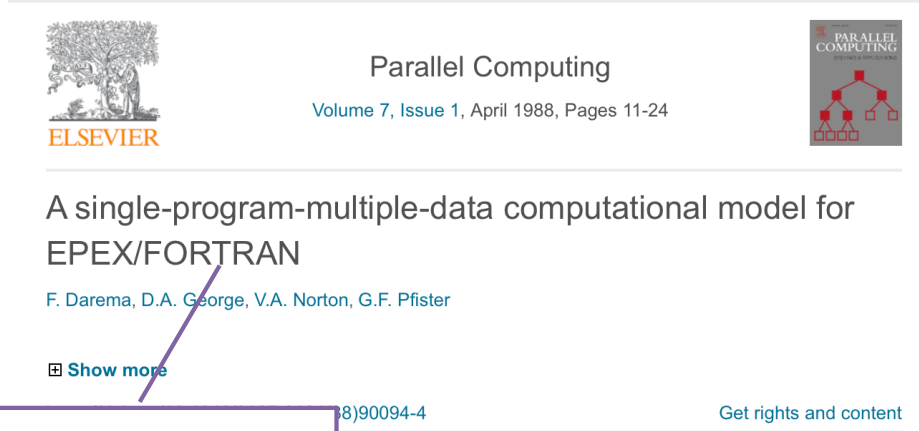


Distributed meme



Name this famous person

Frederica Darema (Director, Air Force Office of Scientific Research)



ELSEVIER

Parallel Computing

Volume 7, Issue 1, April 1988, Pages 11-24

A single-program-multiple-data computational model for EPEX/FORTRAN

F. Darema, D.A. George, V.A. Norton, G.F. Pfister

⊞ Show more

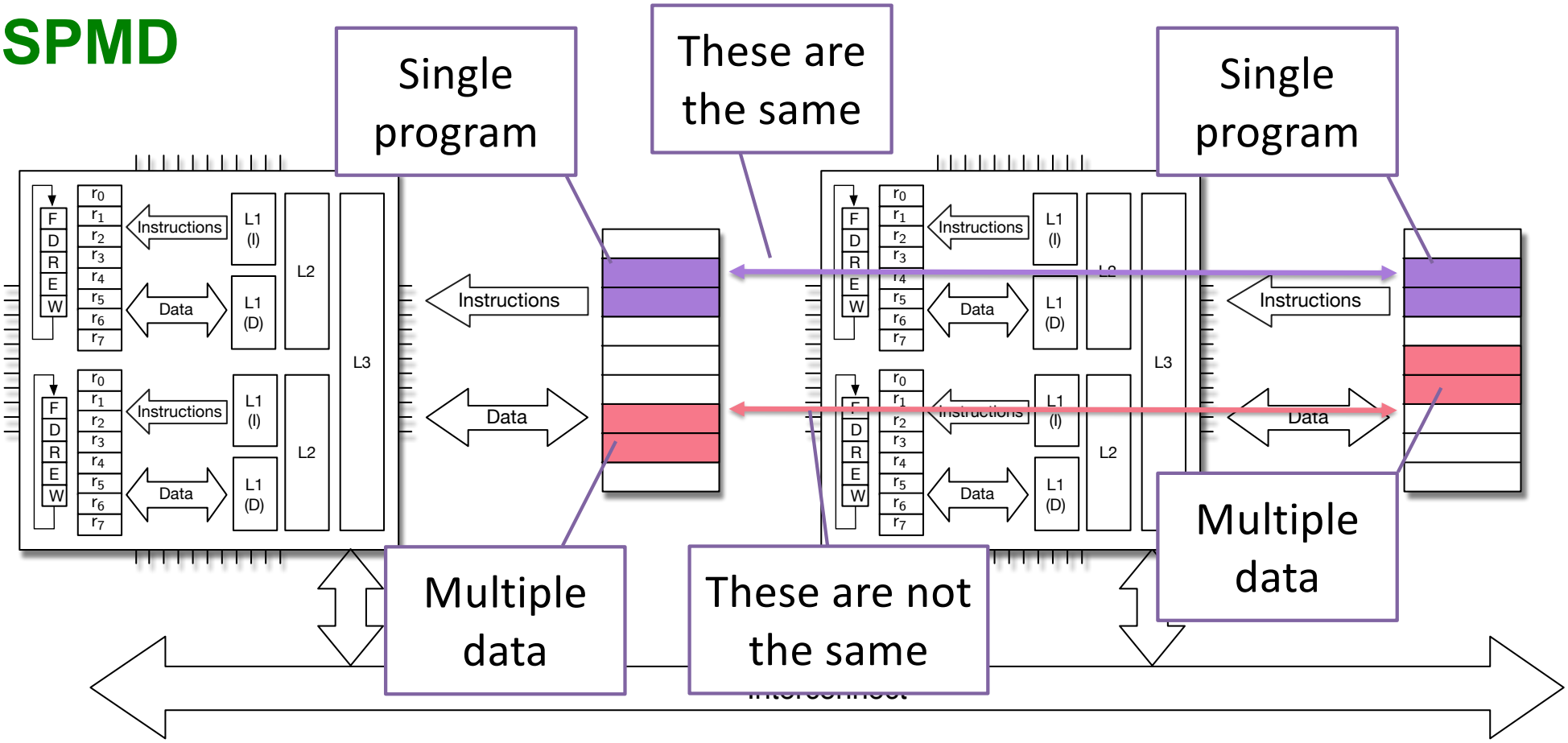
8)90094-4 Get rights and content

-multiple-data computational model which we have
stem to run in parallel mode FORTRAN scientific
mputational model assumes a shared memory
organization and is based on the scheme that all processes executing a program in parallel remain in existence for the entire execution; however, the tasks to be executed by each process are determined dynamically during execution by the use
ppriate synchronizing constructs that are imbedded in the program. We have
trated the applicability of the model in the parallelization of several
ions. We discuss parallelization features of these applications and
ance issues such as overhead, speedup, efficiency.

Single program multiple data model (SPMD)

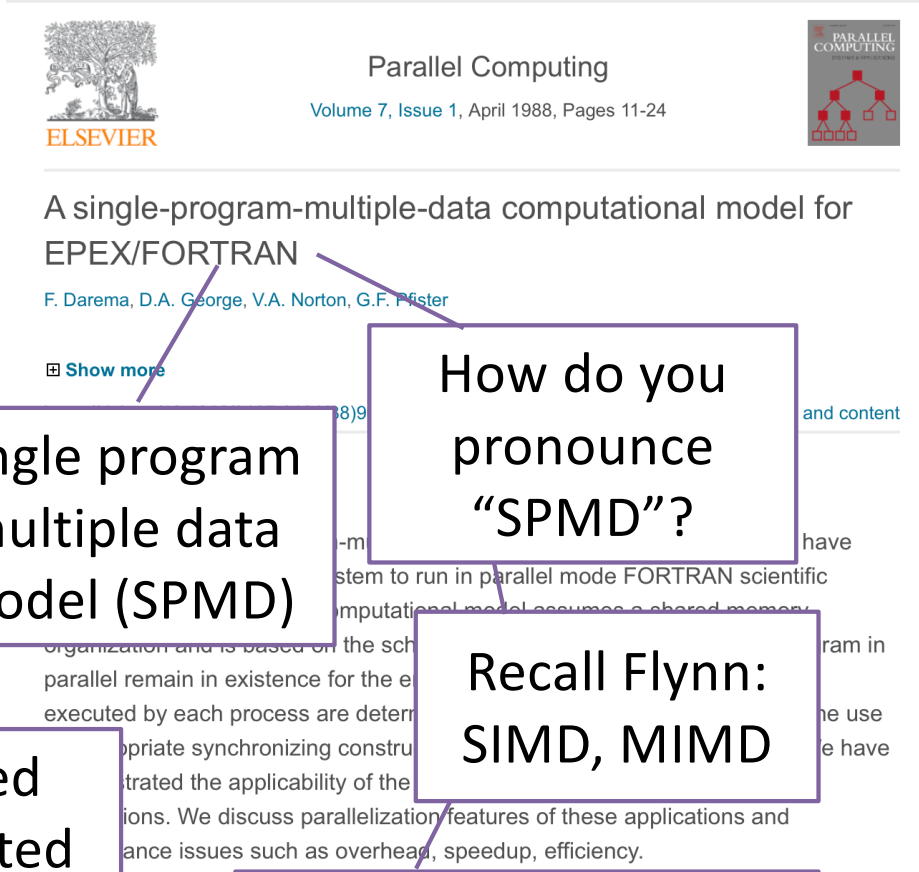
Most widely used model in distributed memory programming

SPMD


Single program
These are the same
Single program
Instructions
Data
L1 (I)
L1 (D)
L2
L3
Multiple data
These are not the same
Multiple data
Interconnect

Name this famous person

Frederica Darema (Director, Air Force Office of Scientific Research)



Single program multiple data model (SPMD)

How do you pronounce "SPMD"?

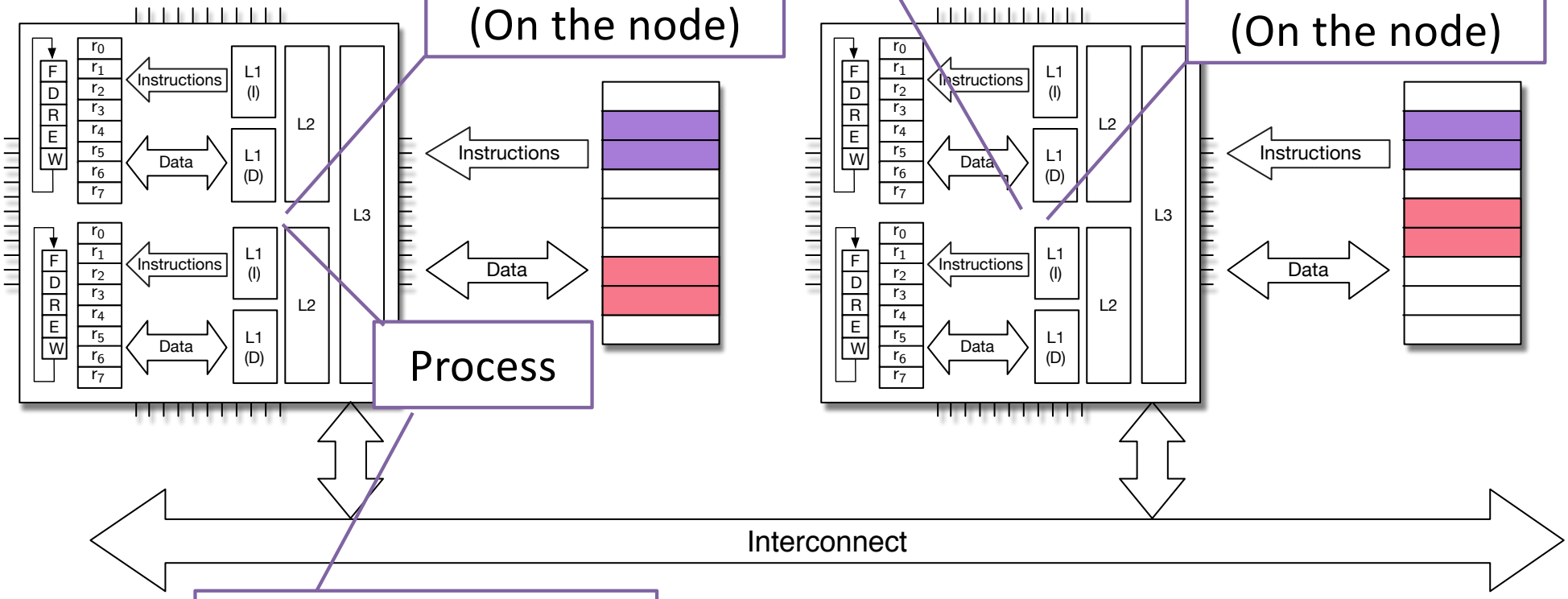
Recall Flynn: SIMD, MIMD

Most widely used model in distributed memory programming

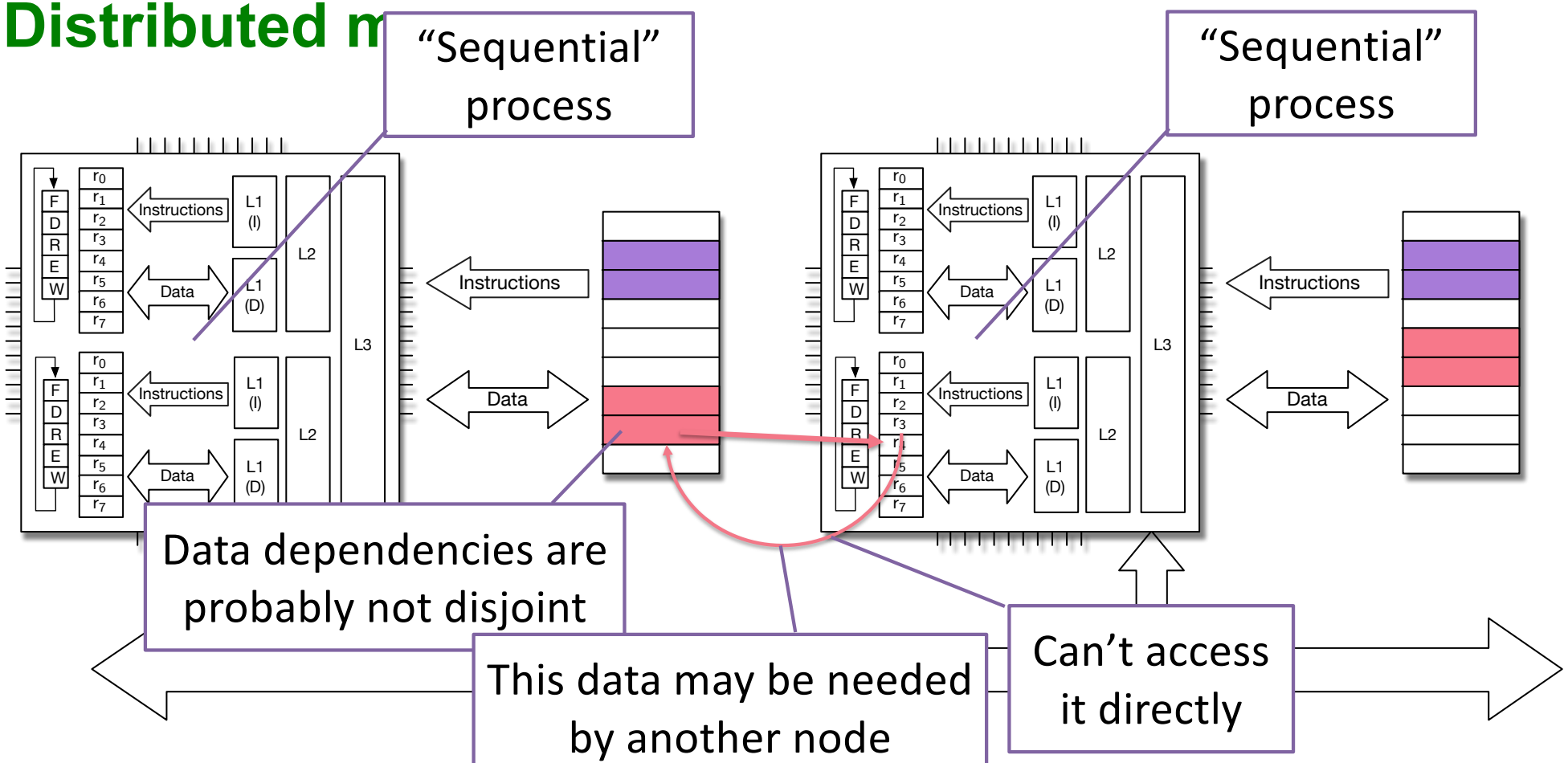
Better model for today's practice than Flynn's

SPMD is pronounced "spim dee"

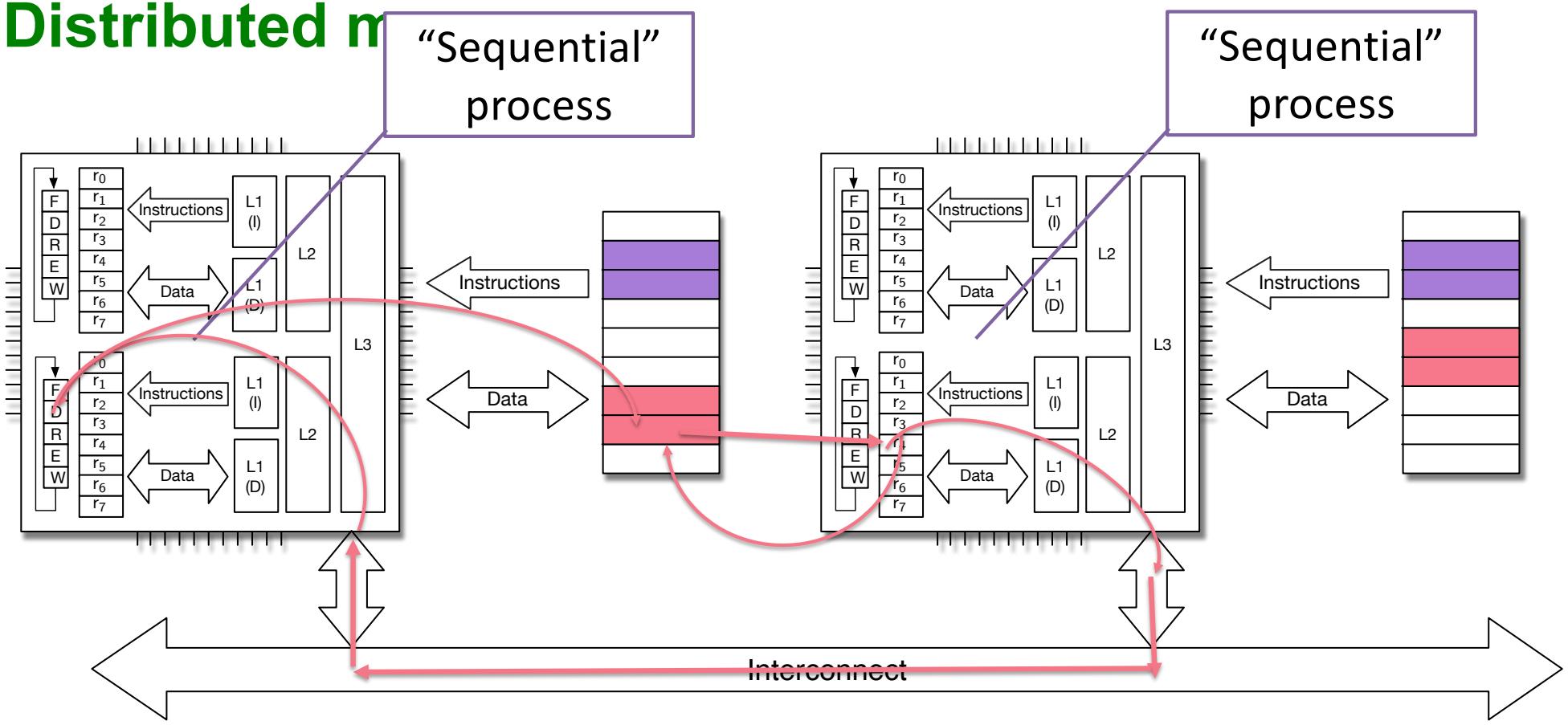
Distributed memory



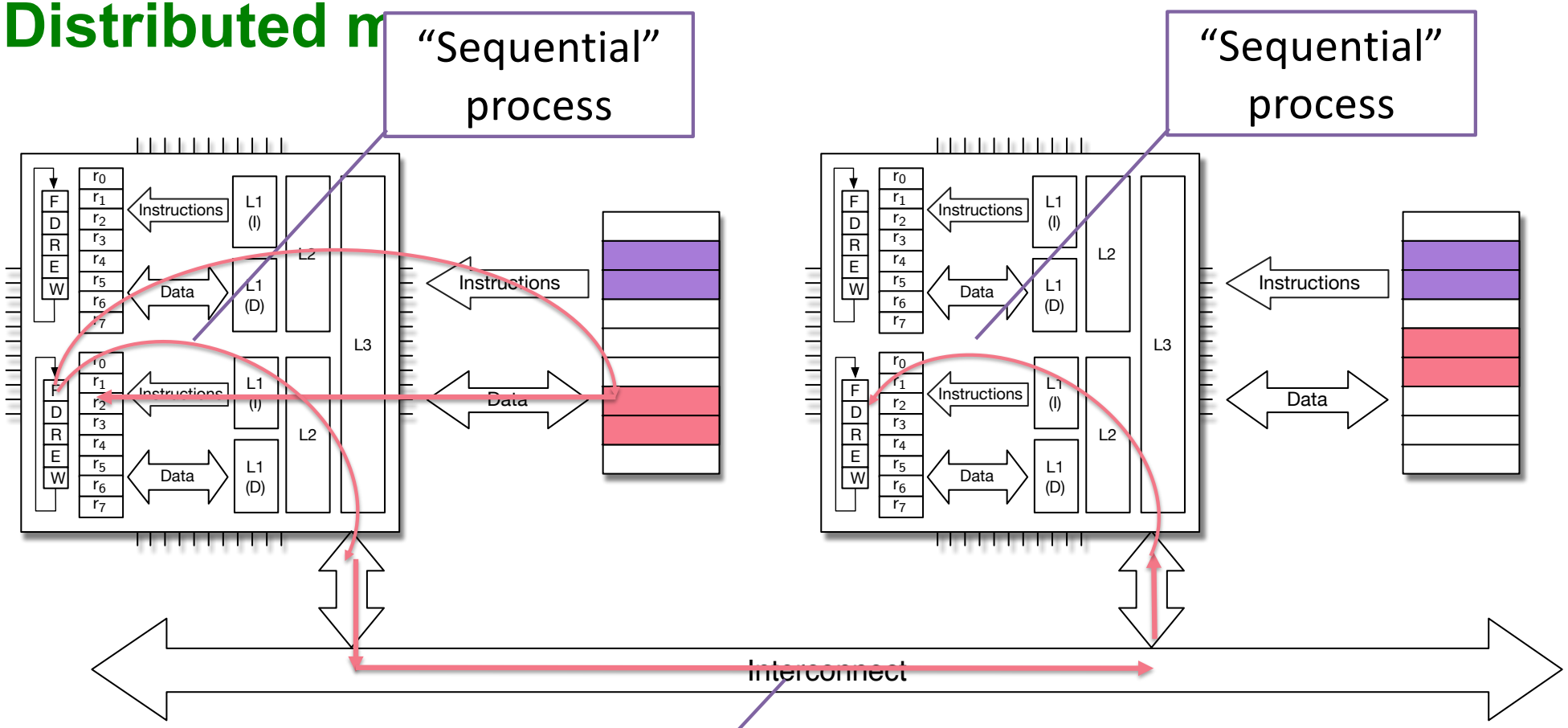
Distributed m



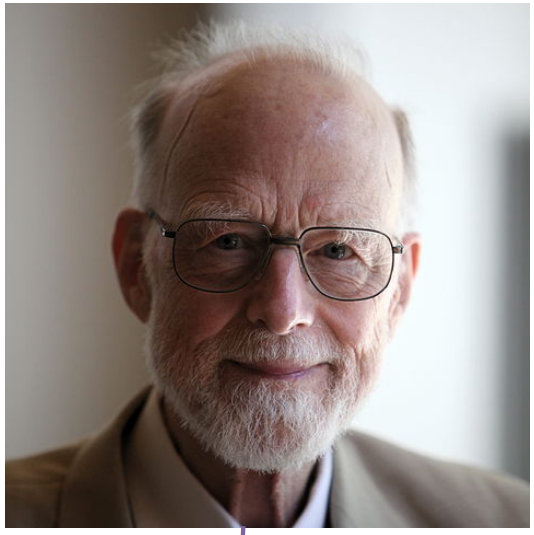
Distributed m



Distributed m



Recall this famous person



C.A.R (Tony) Hoare

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is n
tions of computer progran
were first applied in the s
been extended to other
volves the elucidation of se
which can be used in proc
programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

"CSP" (pronounced see ess pea)

Programming Techniques
S. L. Graham, R. L. Rivest Editors

Communicating Sequential Processes

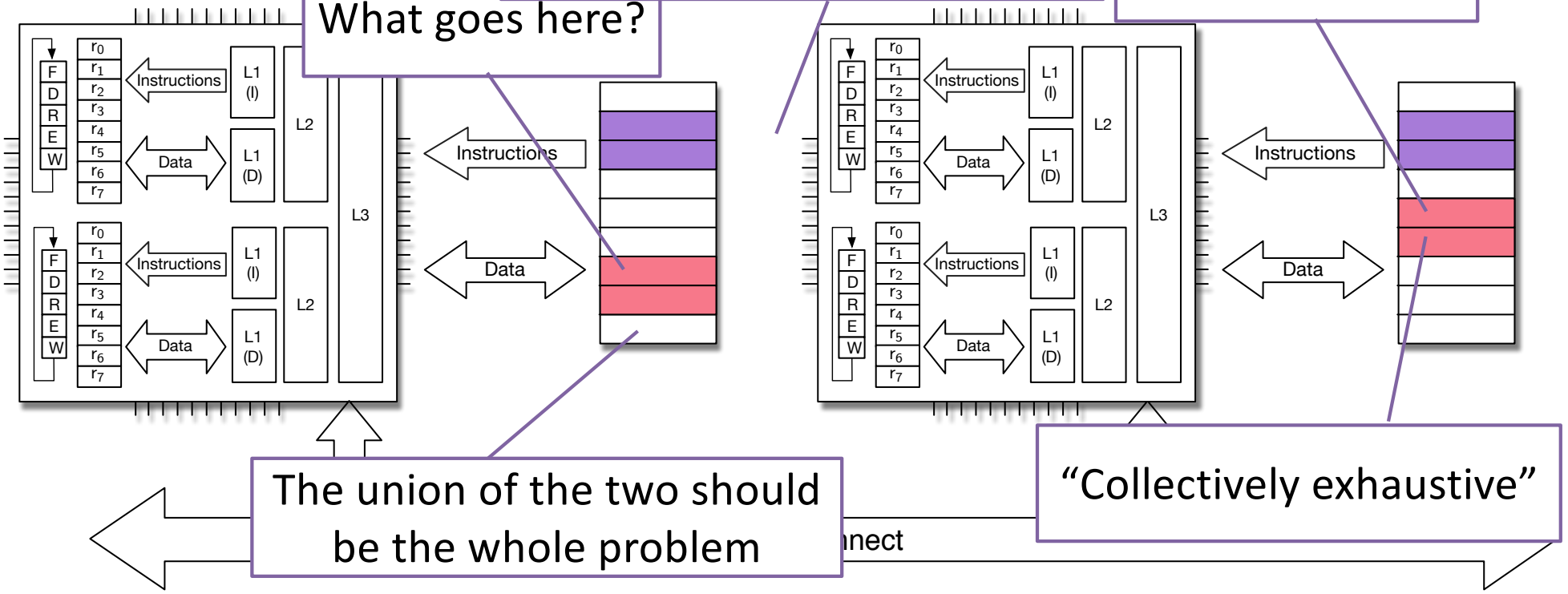
C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.

Key Words and Phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical regions, monitors, iterative arrays
CR Categories: 4.20, 4.22, 4.32

PS: These aren't even what he is most famous for

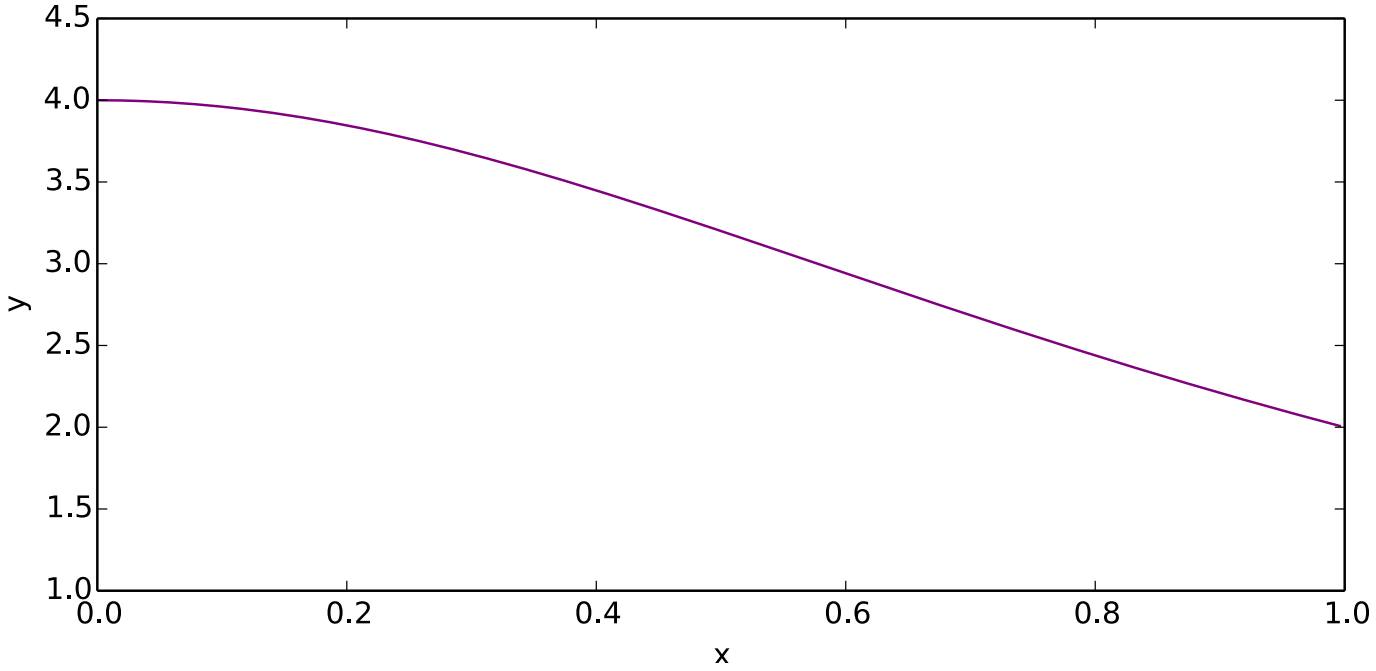
Distributed meme



Back to our trusty example (one of them)

- Find the value of π
- Using formula

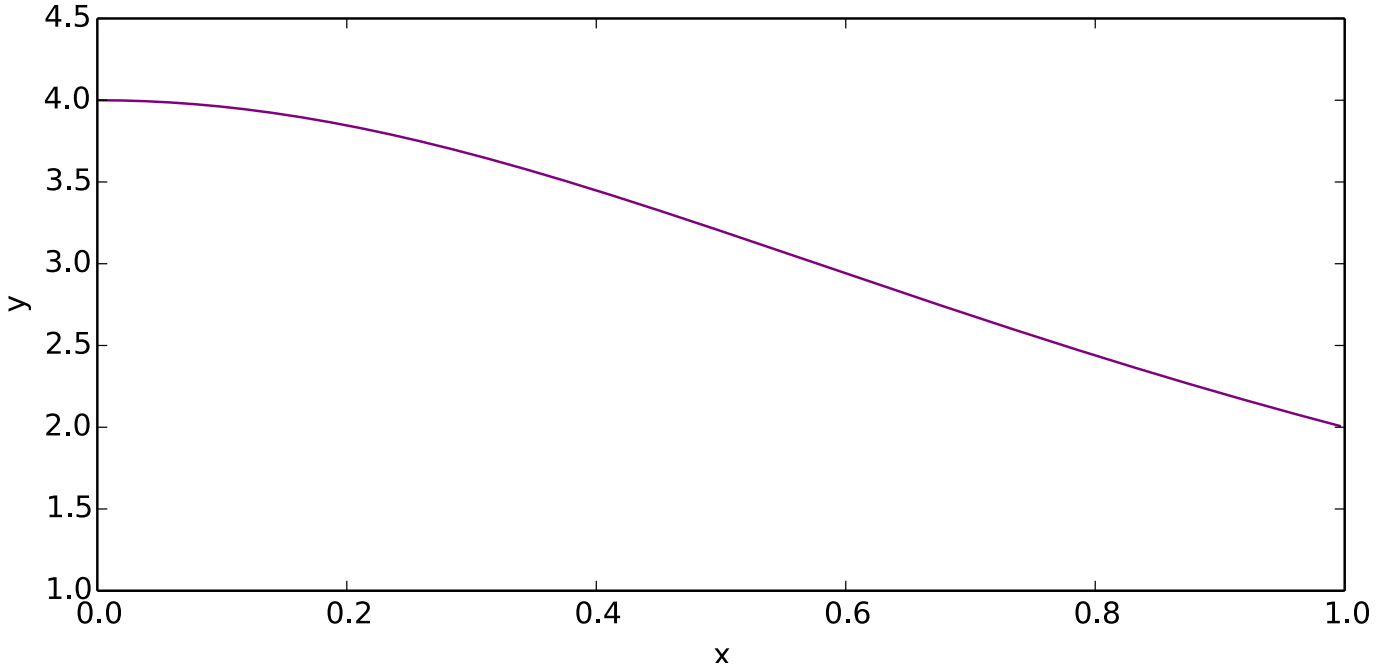
$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$



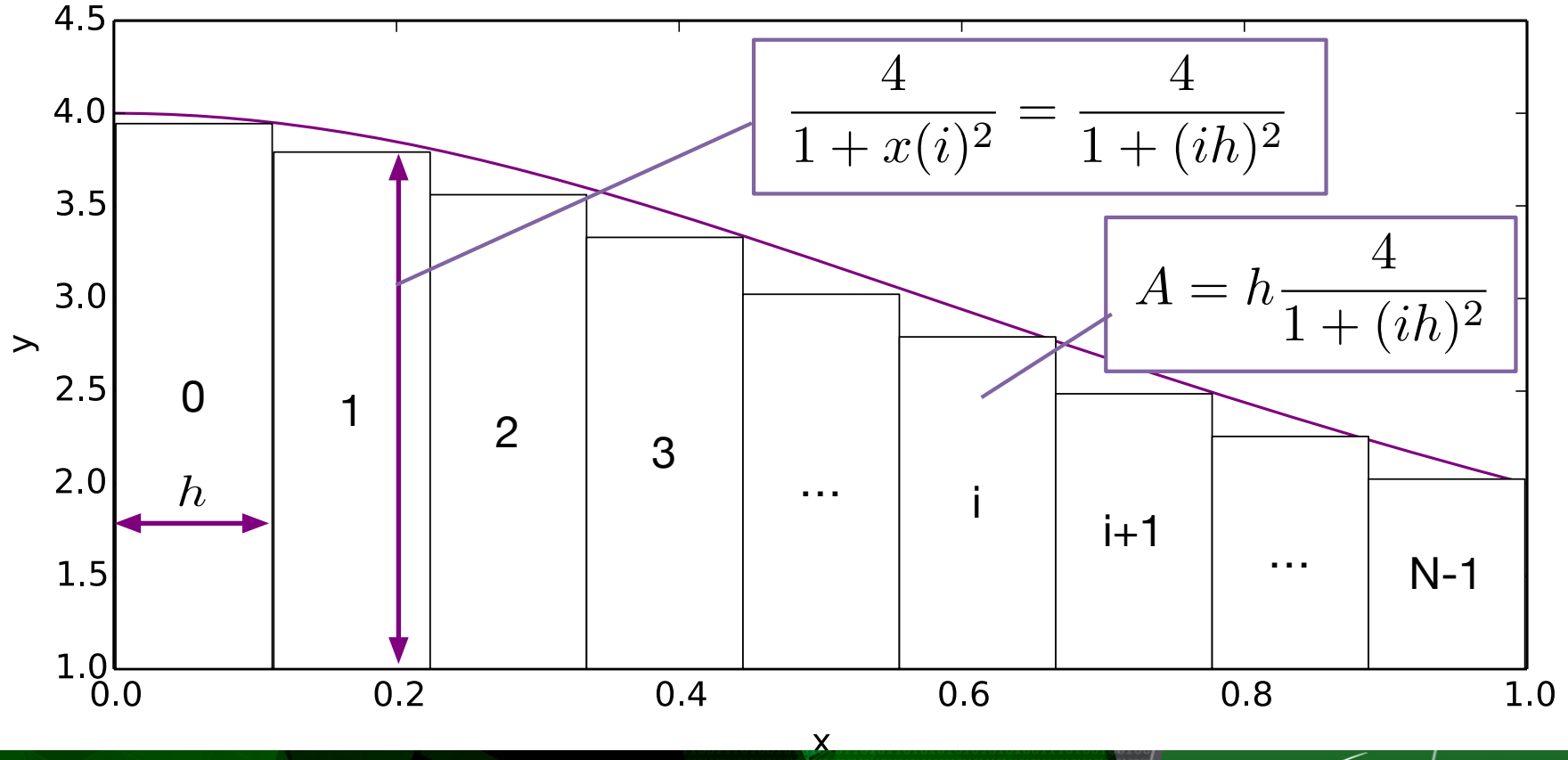
Example

- Find the value of π
- Using formula

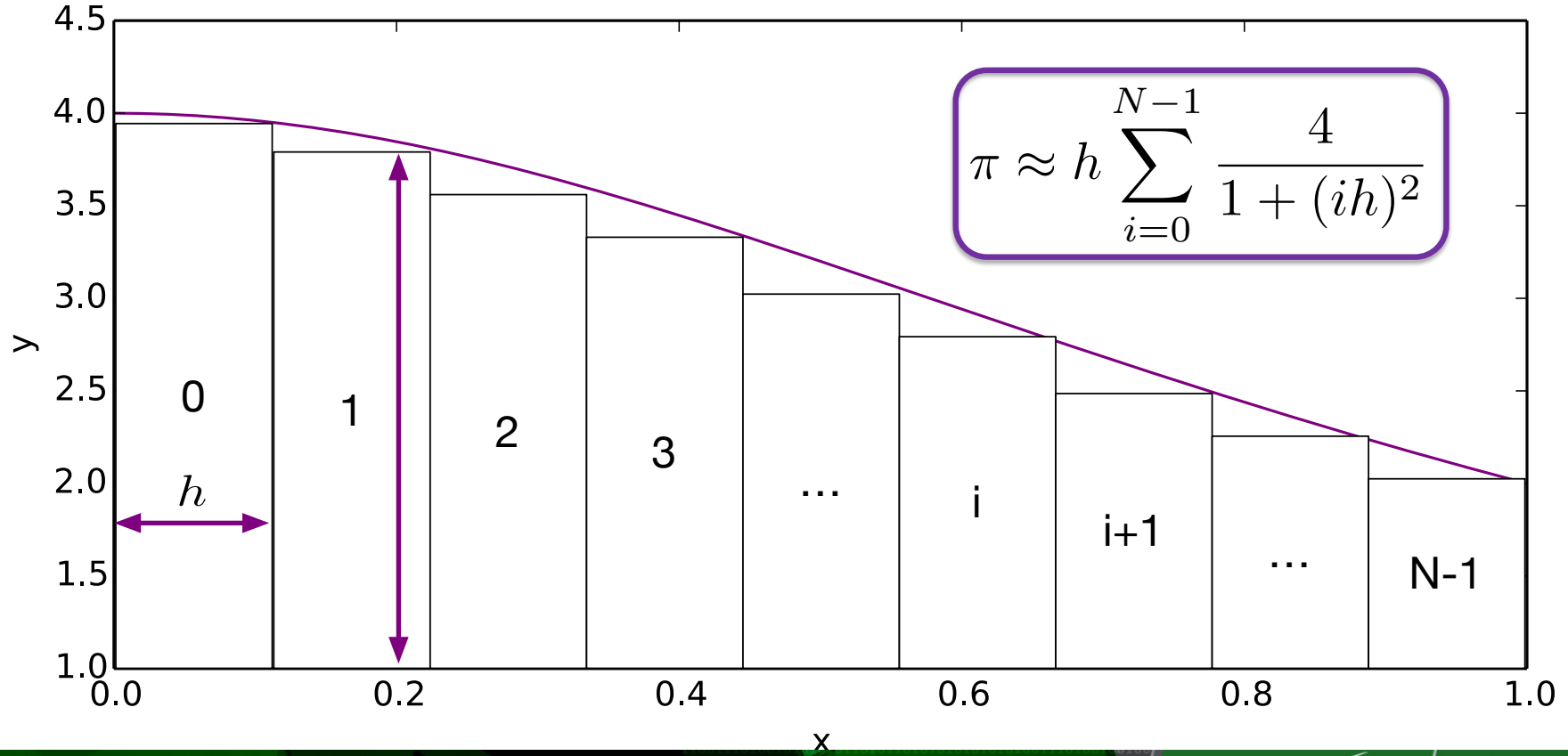
$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$



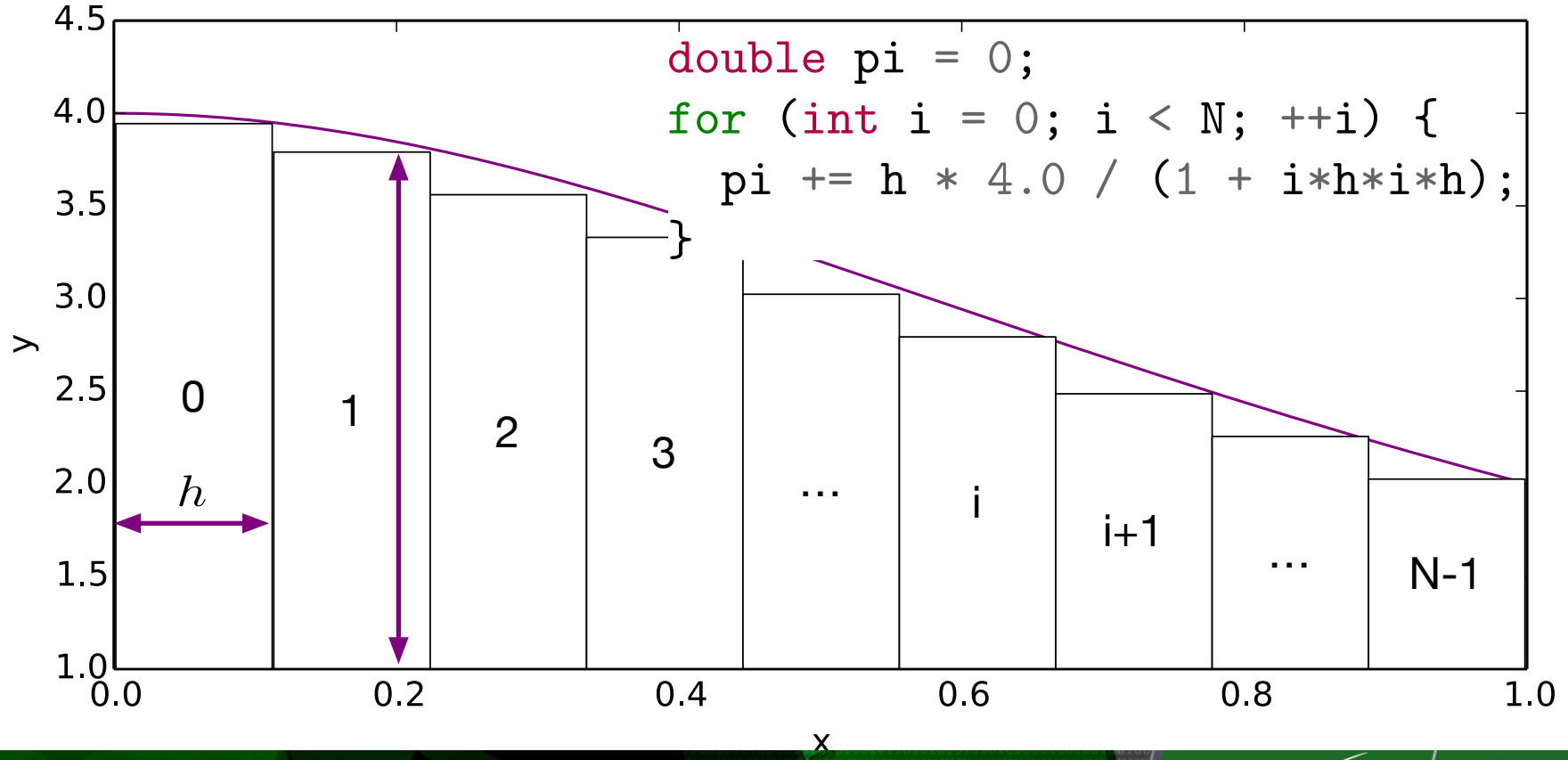
Numerical Quadrature



Numerical Quadrature

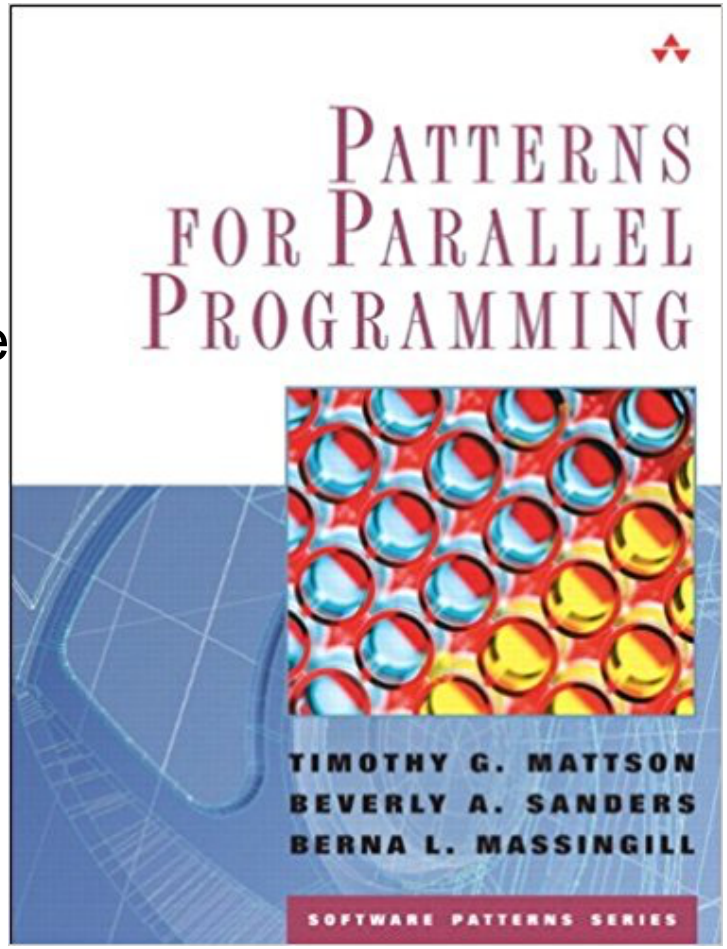


Numerical Quadrature (Sequential)



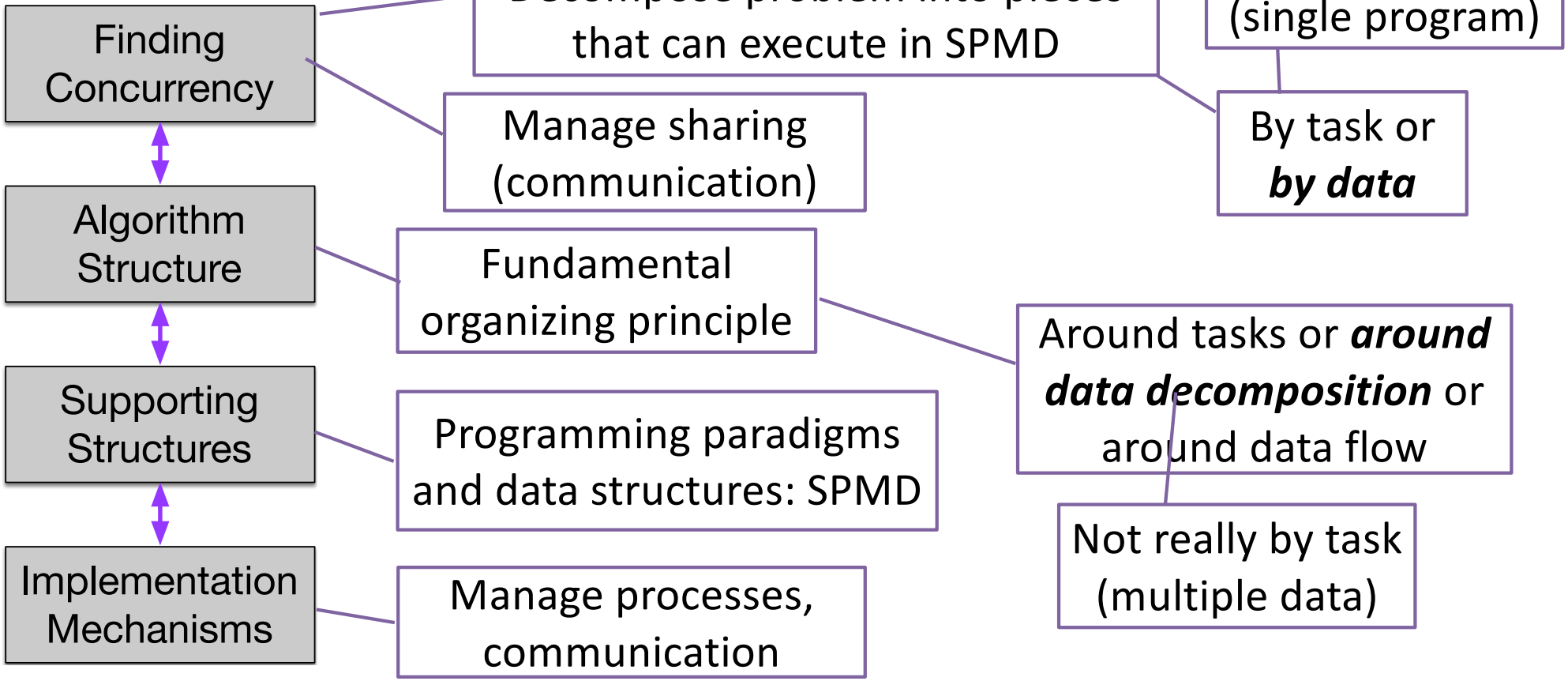
Parallelization Strategy

- How do we go from a problem want to solve
- And maybe know how to solve sequentially
- To a parallel program
- That scales



Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming* (First ed.). Addison-Wesley Professional.

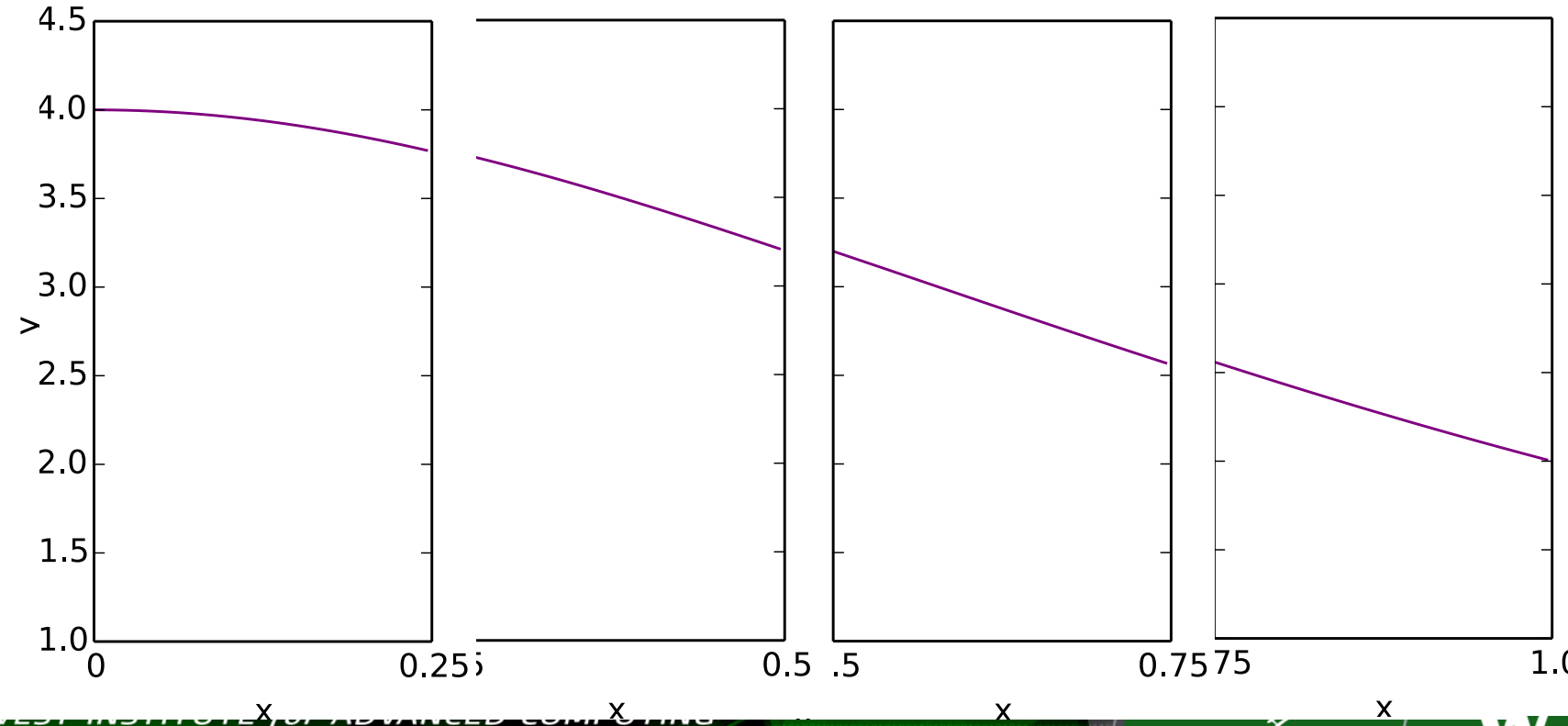
Parallelization strategy, SPMD version



Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming* (First ed.). Addison-Wesley Professional.

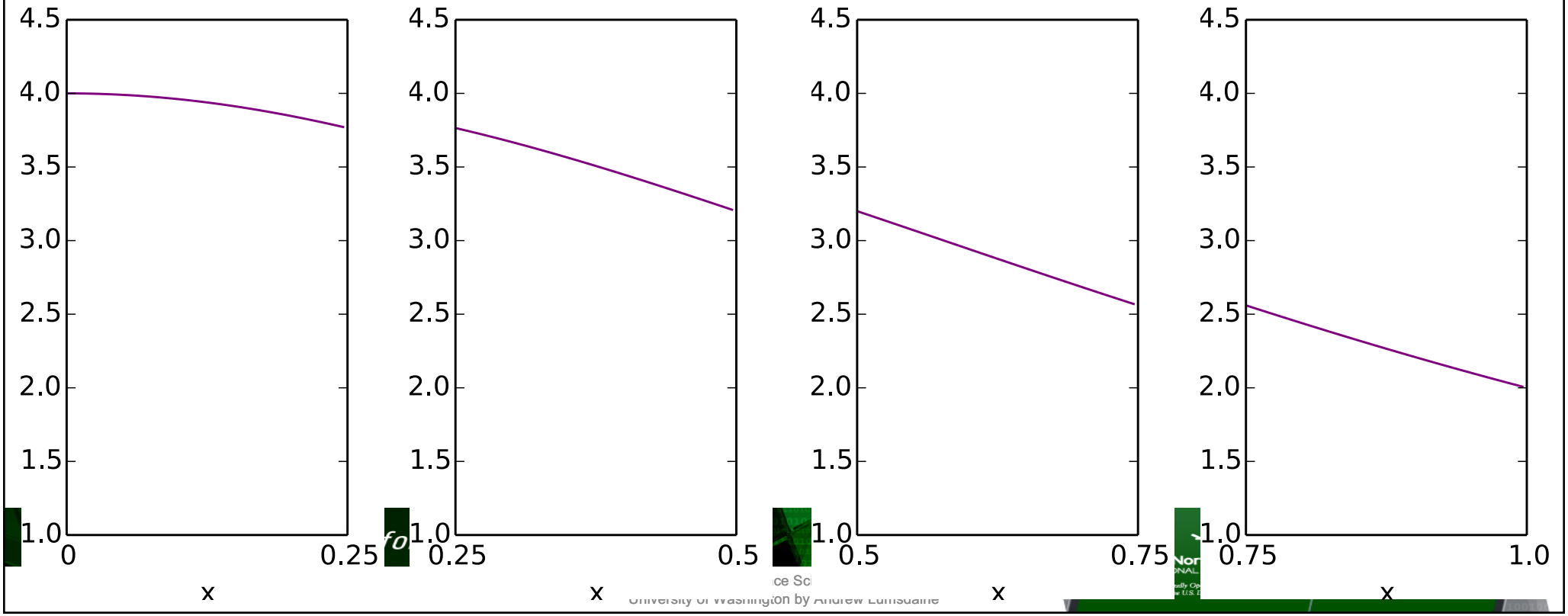
Finding Concurrency

$$\pi = \int_0^{0.25} \frac{4}{1+x^2}dx + \int_{0.25}^{0.5} \frac{4}{1+x^2}dx + \int_{0.5}^{0.75} \frac{4}{1+x^2}dx + \int_{0.75}^{1} \frac{4}{1+x^2}dx$$



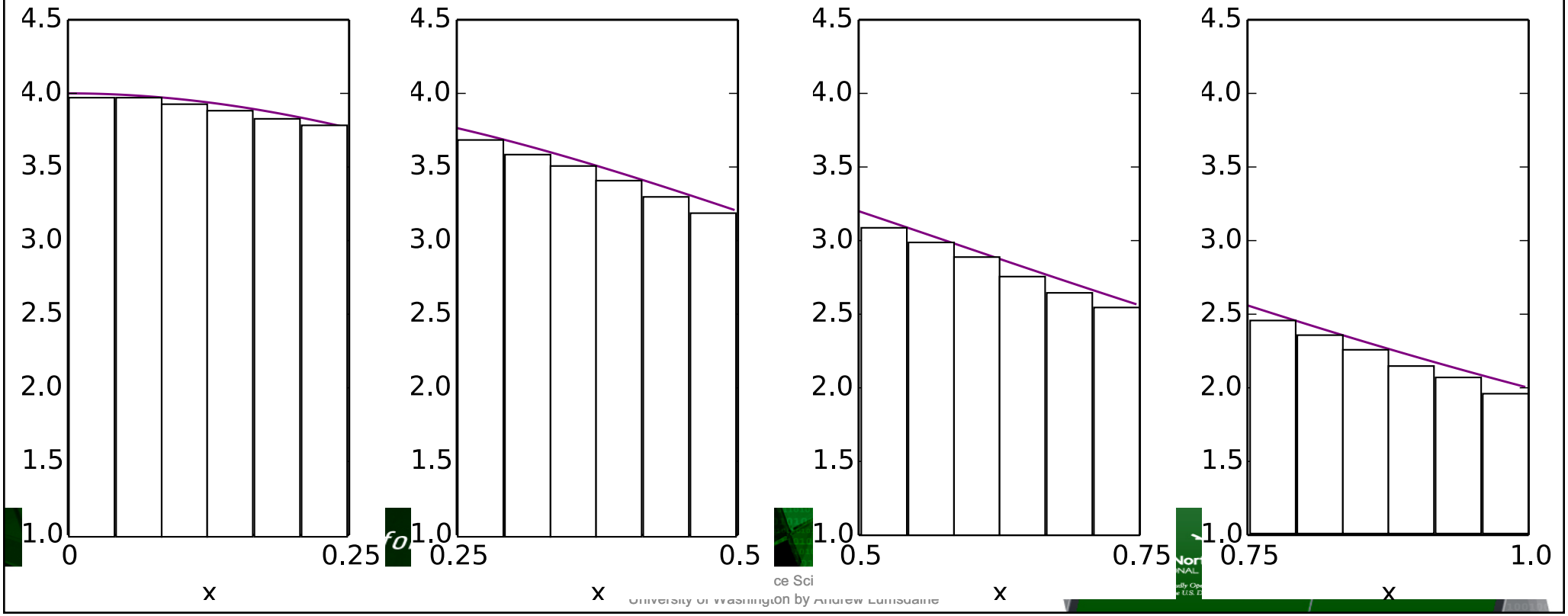
Finding Concurrency

$$\pi = \int_0^{0.25} \frac{4}{1+x^2}dx + \int_{0.25}^{0.5} \frac{4}{1+x^2}dx + \int_{0.5}^{0.75} \frac{4}{1+x^2}dx + \int_{0.75}^{1} \frac{4}{1+x^2}dx$$



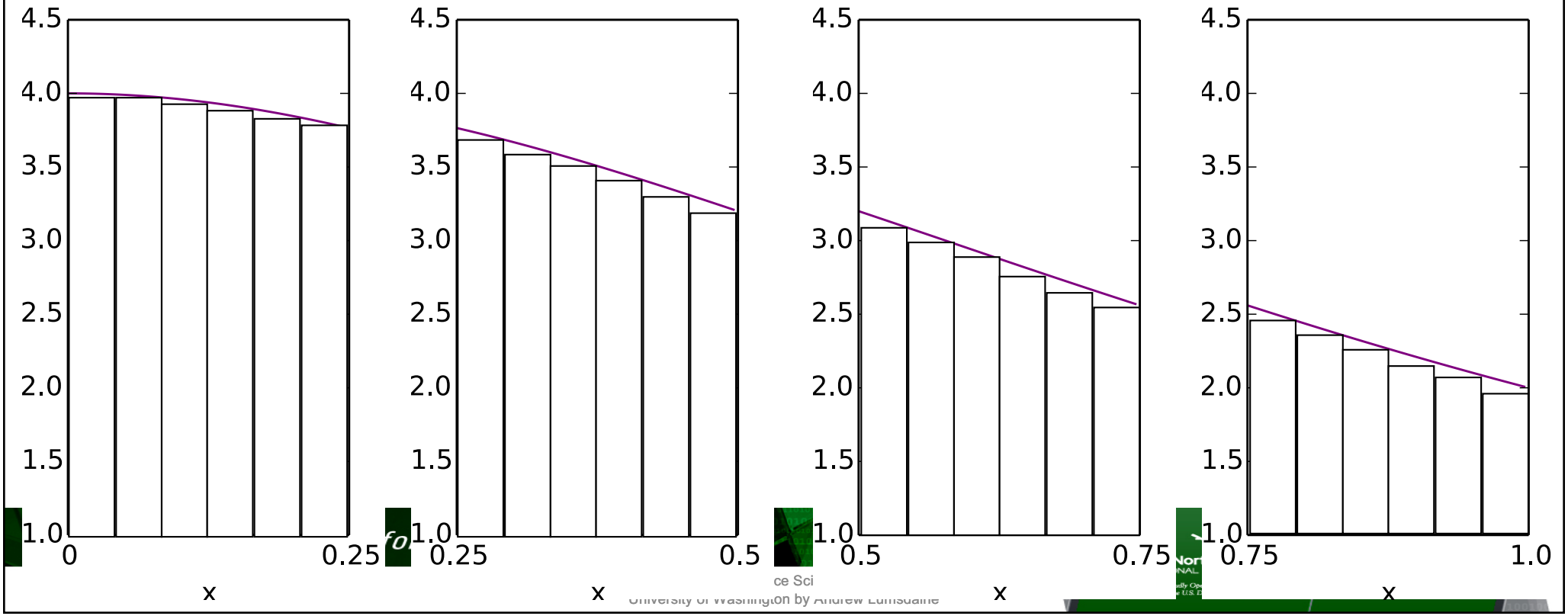
Finding Concurrency

$$\pi = \int_0^{0.25} \frac{4}{1+x^2}dx + \int_{0.25}^{0.5} \frac{4}{1+x^2}dx + \int_{0.5}^{0.75} \frac{4}{1+x^2}dx + \int_{0.75}^{1} \frac{4}{1+x^2}dx$$



Finding Concurrency

$$\pi \approx h\sum_{i=0}^{N/4-1} \frac{4}{1+(ih)^2} + h\sum_{i=N/4}^{N/2-1} \frac{4}{1+(ih)^2} + h\sum_{i=N/2}^{3N/4-1} \frac{4}{1+(ih)^2} + h\sum_{i=3N/4}^{3N-1} \frac{4}{1+(ih)^2}$$



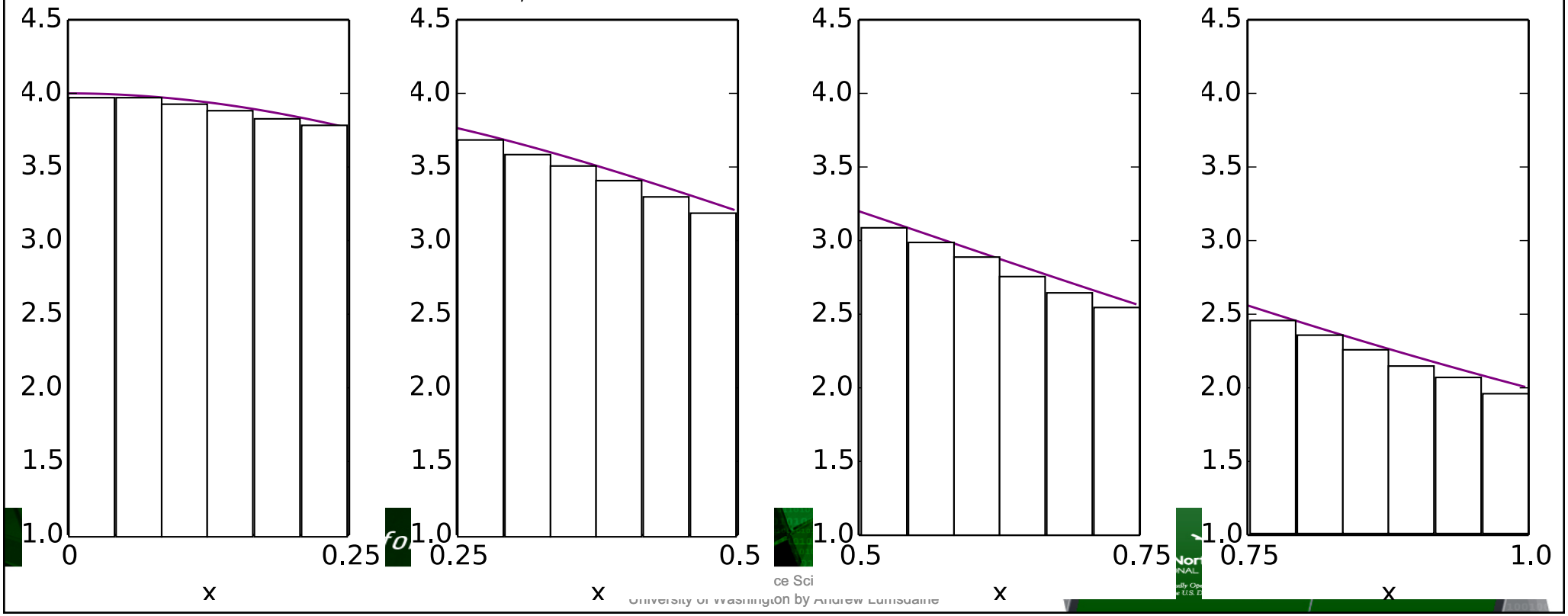
Finding Concurrency

$$h\sum_{i=0}^{N/4-1}\frac{4}{1+(ih)^2}$$

$$h\sum_{i=N/4}^{N/2-1}\frac{4}{1+(ih)^2}$$

$$h\sum_{i=N/2}^{3N/4-1}\frac{4}{1+(ih)^2}$$

$$h\sum_{i=3N/4}^{N-1}\frac{4}{1+(ih)^2}$$



Finding Concurrency

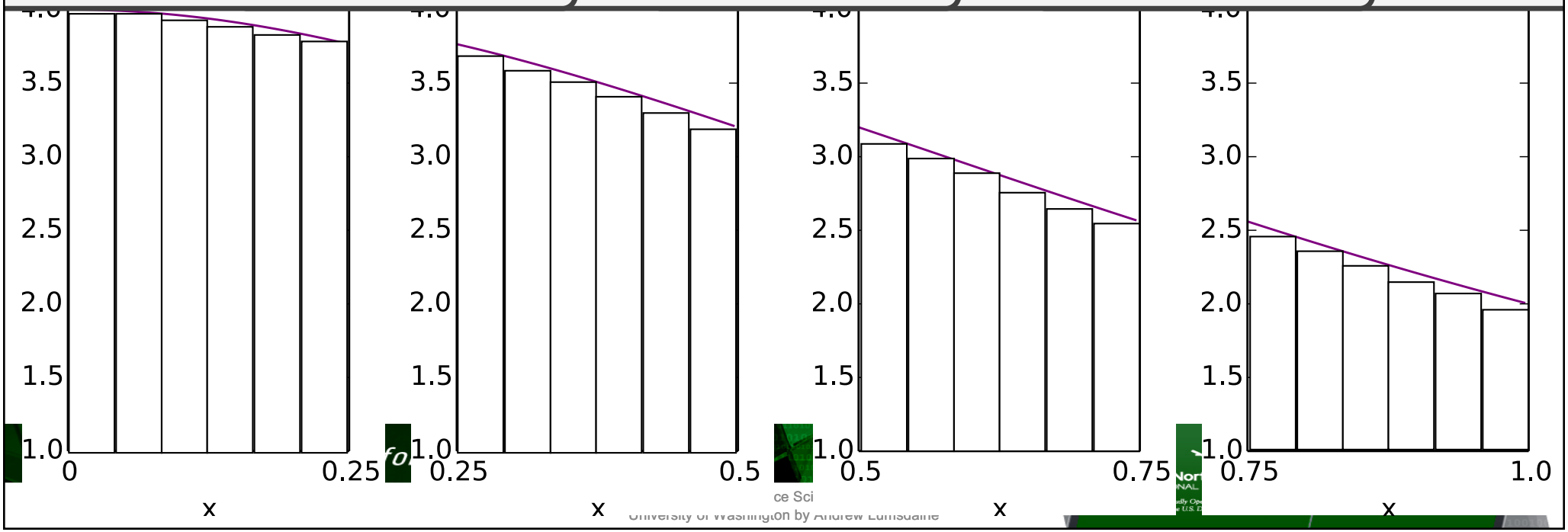
```
for (int i = begin; i < end; ++i) {
  pi += h * 4.0 / (1 + i*h*i*h);
}
```

```
int i = 0; i < N/4; ++i) {
+= h * 4.0 / (1 + i*h*i*h);
```

```
4; i < N/2; ++i) {
/ (1 + i*h*i*h);
```

```
/2; i < 3*N/4; ++i) {
0 / (1 + i*h*i*h);
```

```
N/4; i < N
/ (1 + i*
```



Finding Concurrency

$$h \sum_{i=0}^{N/4-1} \frac{4}{1+(ih)^2}$$

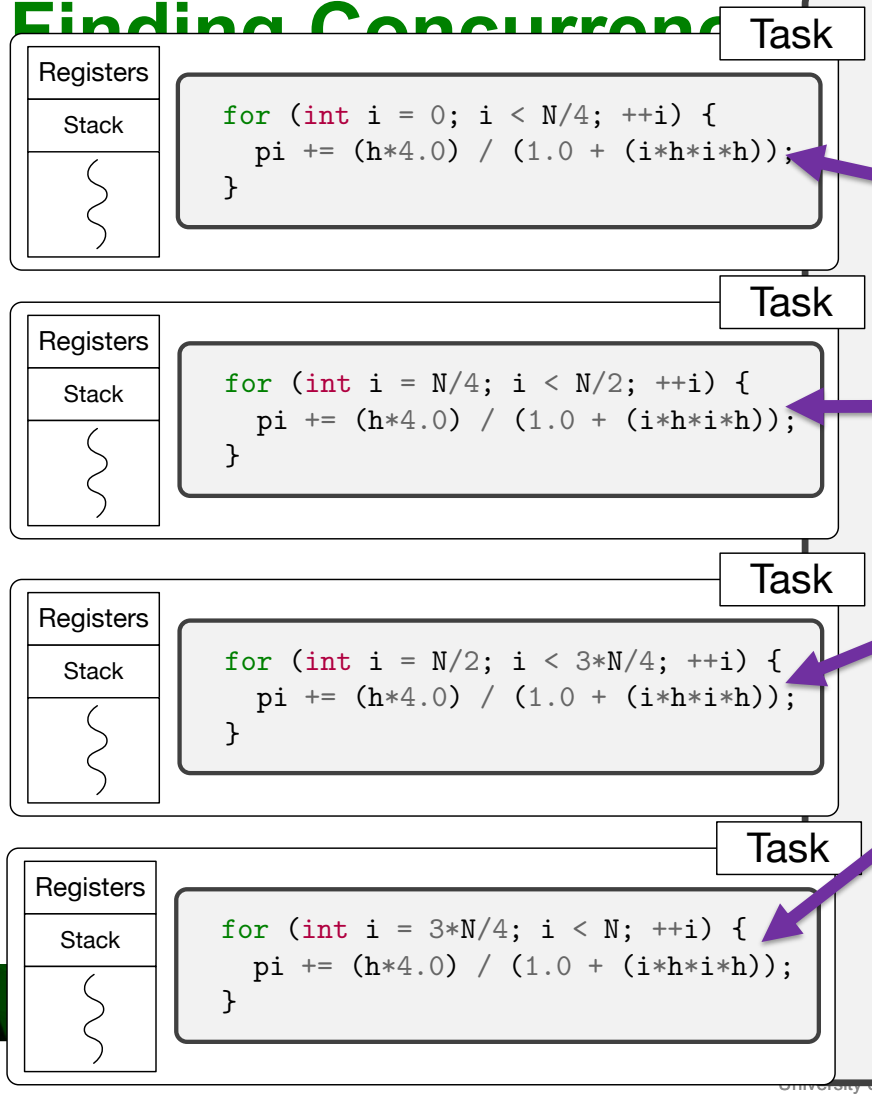
$$h \sum_{i=N/4}^{N/2-1} \frac{4}{1+(ih)^2}$$

$$h \sum_{i=N/2}^{3N/4-1} \frac{4}{1+(ih)^2}$$

$$h \sum_{i=3N/4}^{N-1} \frac{4}{1+(ih)^2}$$

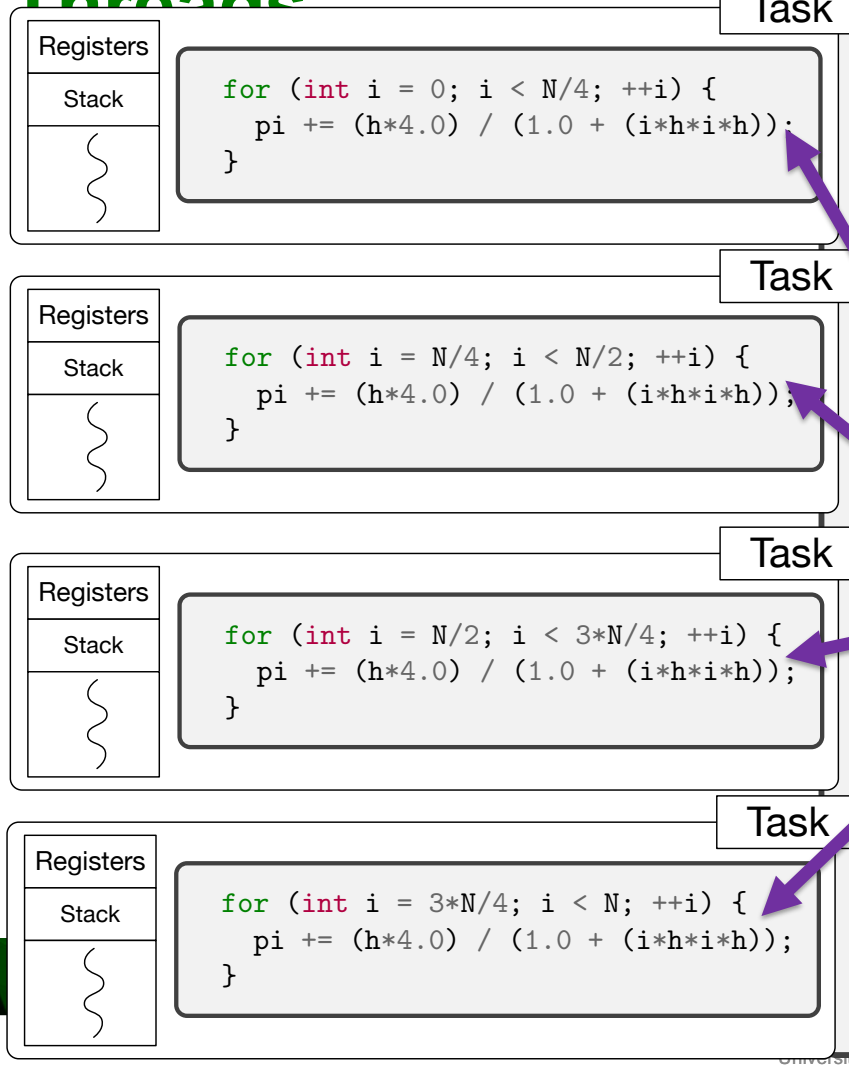
```
int main() {
  double pi = 0.0;  int N = 1024*1024;

  for (int i = 0; i < N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/4; i < N/2; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/2; i < 3*N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = 3*N/4; i < N; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  std::cout << "pi ~ " << pi << std::endl;
  return 0;
}
```

Finding Concurrency




```
int main() {
  double pi = 0.0;  int N = 1024*1024;

  for (int i = 0; i < N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/4; i < N/2; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/2; i < 3*N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = 3*N/4; i < N; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  std::cout << "pi ~ " << pi << std::endl;
  return 0;
}
```

Threads




```
double pi = 0.0;

void pi_helper(int begin, int end, double h) {
  for (int i = begin; i < end; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}

int main(int argc, char* argv[]) {
  int N = 1024 * 1024; double h = 1.0/ (double)N;

  std::thread t0(pi_helper, 0,     N/4,   h);
  std::thread t1(pi_helper, N/4,   N/2,   h);
  std::thread t2(pi_helper, N/2,   3*N/4, h);
  std::thread t3(pi_helper, 3*N/4, N,     h);

  t0.join();  t1.join();  t2.join();  t3.join();

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```

Finding Concurrency

$$h \sum_{i=0}^{N/4-1} \frac{4}{1+(ih)^2}$$

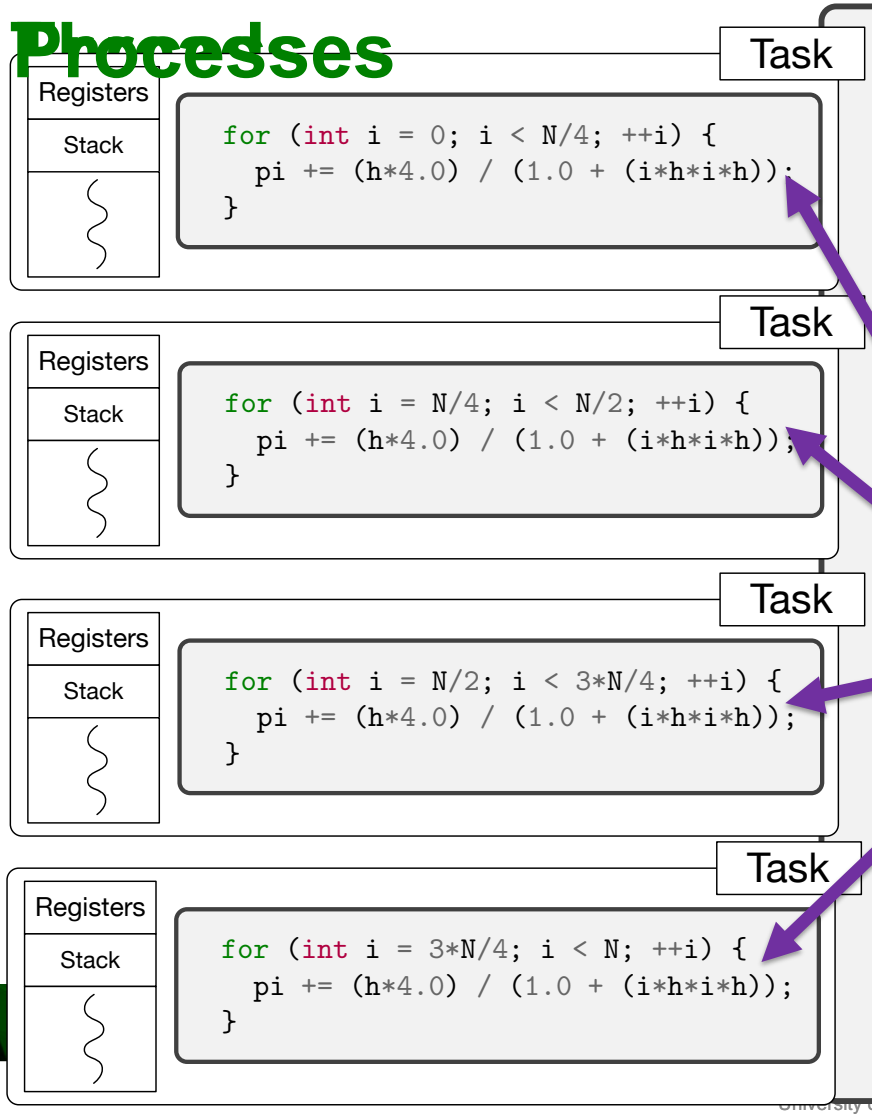
$$h \sum_{i=N/4}^{N/2-1} \frac{4}{1+(ih)^2}$$

$$h \sum_{i=N/2}^{3N/4-1} \frac{4}{1+(ih)^2}$$

$$h \sum_{i=3N/4}^{N-1} \frac{4}{1+(ih)^2}$$

```
int main() {
  double pi = 0.0;  int N = 1024*1024;

  for (int i = 0; i < N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/4; i < N/2; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/2; i < 3*N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = 3*N/4; i < N; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  std::cout << "pi ~ " << pi << std::endl;
  return 0;
}
```

Processes




```
double pi = 0.0;

void pi_helper(int begin, int end, double h) {
  for (int i = begin; i < end; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}

int main(int argc, char* argv[]) {
  int N = 1024 * 1024; double h = 1.0/ (double)N;

  std::thread t0(pi_helper, 0,     N/4,   h);
  std::thread t1(pi_helper, N/4,   N/2,   h);
  std::thread t2(pi_helper, N/2,   3*N/4, h);
  std::thread t3(pi_helper, 3*N/4, N,     h);

  t0.join();  t1.join();  t2.join();  t3.join();

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```

Processes

```
int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = 0; i < N/4; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << st
```

Process

```
int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = N/4; i < N/2; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << st
```

Process

```
int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = N/2; i < 3*N/4; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << st
```

Process

```
int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = 3*N/4; i < N; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

```
double pi = 0.0;

void pi_helper(int begin, int end, double h) {
  for (int i = begin; i < end; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}

int main(int argc, char* argv[]) {
  int N = 1024 * 1024; double h = 1.0/ (double)N;

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```

Communicating sequential processes / SPMD

```cpp
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = 0; i < N/4; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

```cpp
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = N/4; i < N/2; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

```cpp
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = N/2; i < 3*N/4; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

```cpp
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = 3*N/4; i < N; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

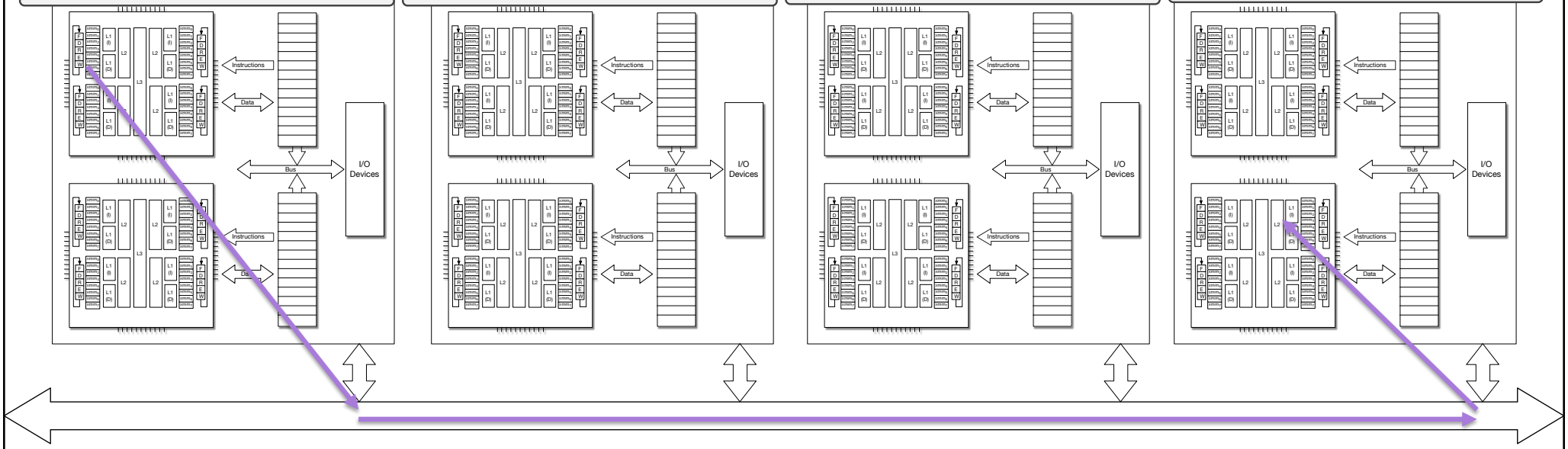
Communicating sequential processes / SPMD

```
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = 0; i < N/4; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

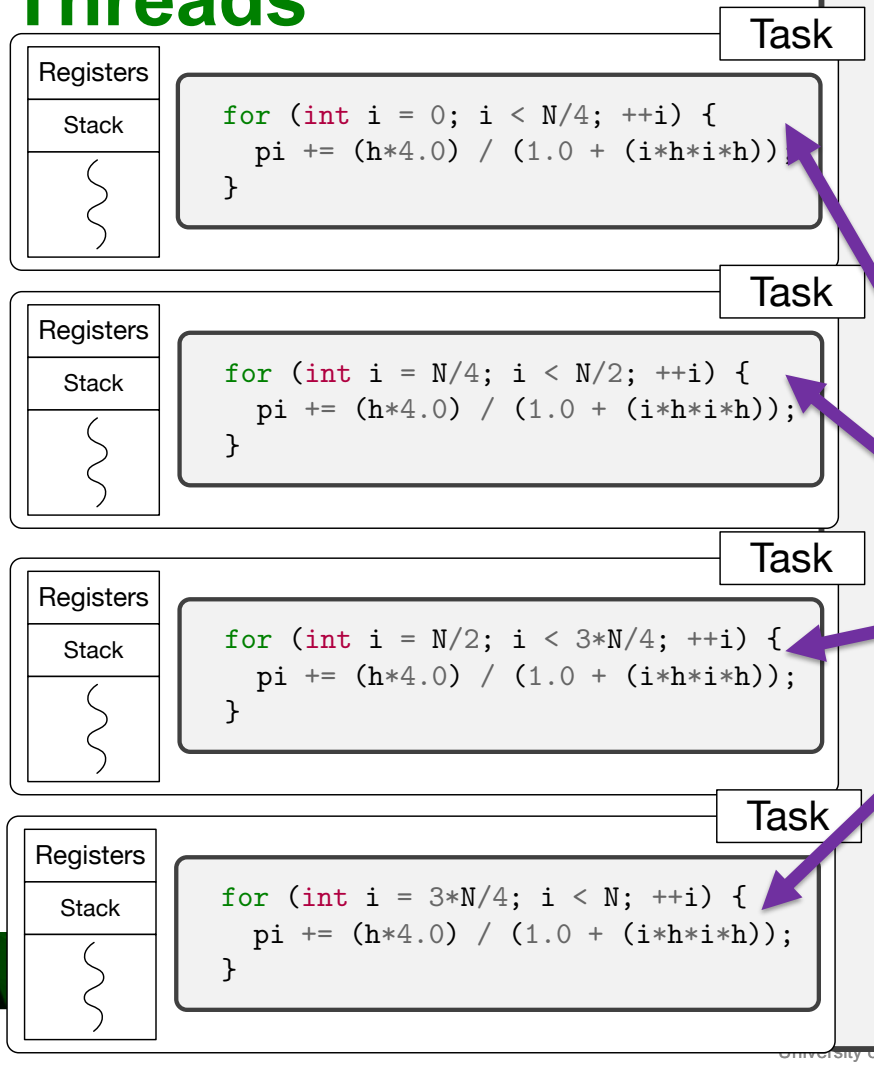
```
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = N/4; i < N/2; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

```
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = N/2; i < 3*N/4; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

```
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = 3*N/4; i < N; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

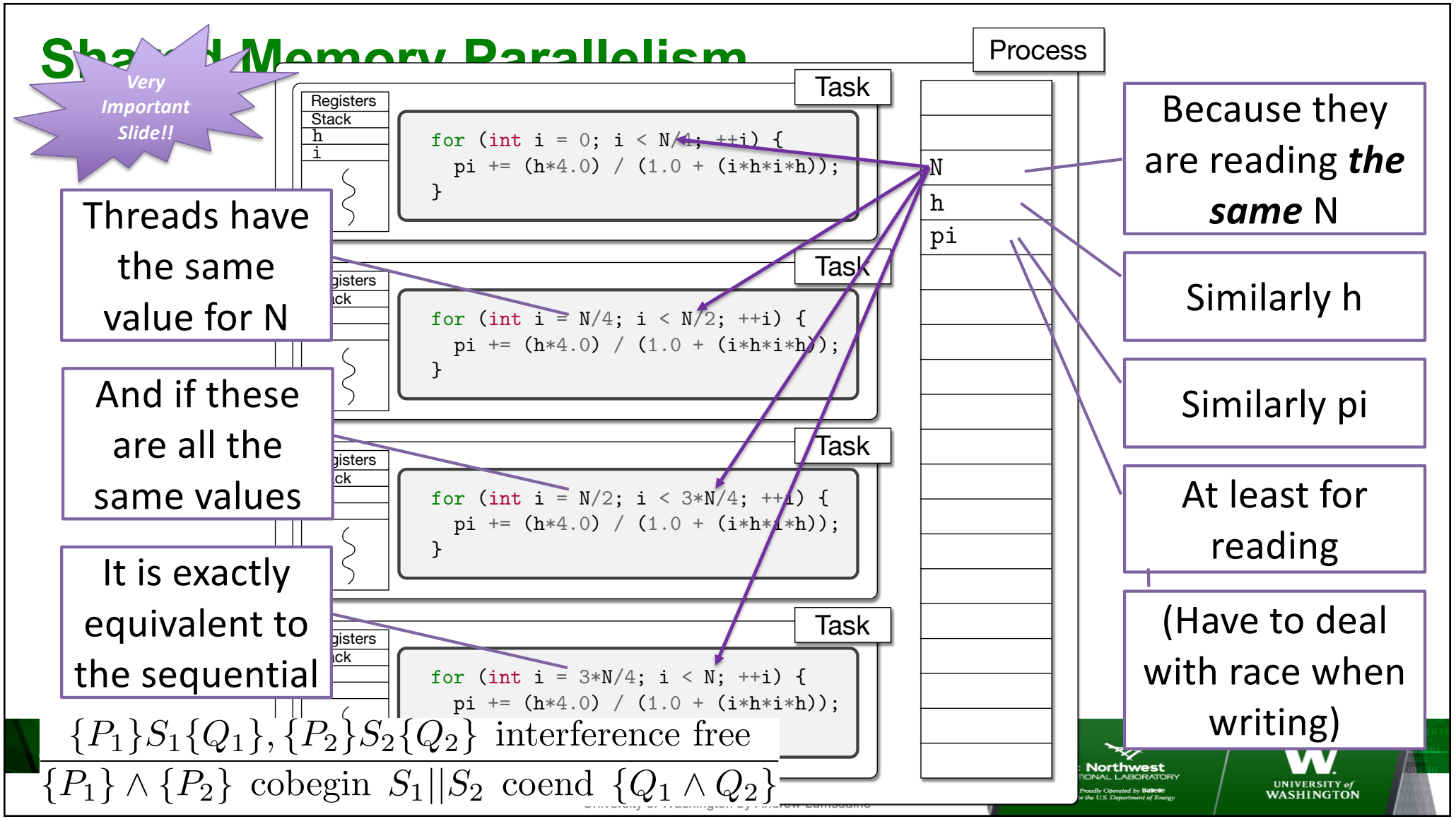


Threads



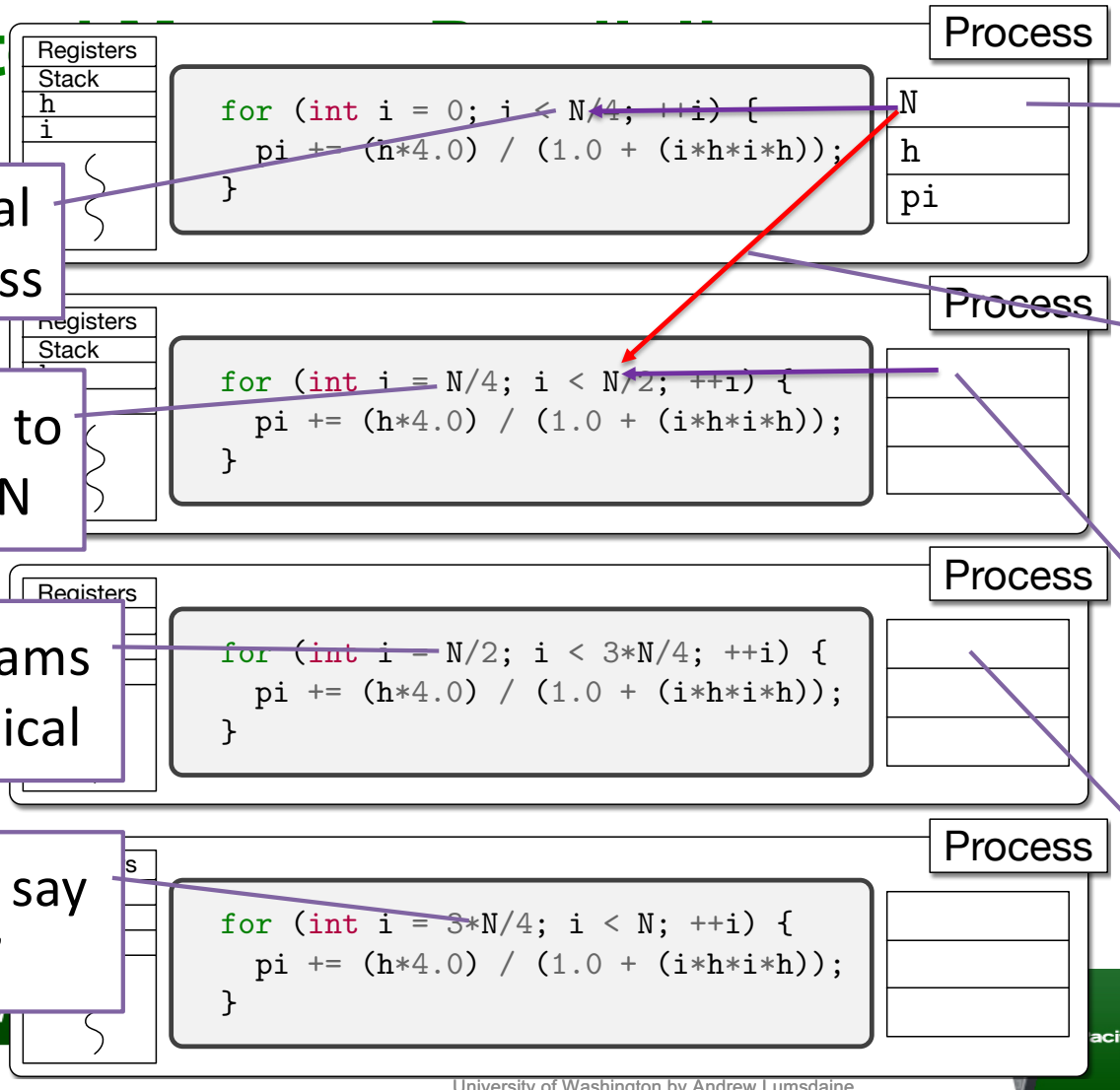

```
double pi = 0.0;

void pi_helper(int begin, int end, double h) {
  for (int i = begin; i < end; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}

int main(int argc, char* argv[]) {
  int N = 1024 * 1024; double h = 1.0/ (double)N;

  std::thread t0(pi_helper, 0,     N/4,   h);
  std::thread t1(pi_helper, N/4,   N/2,   h);
  std::thread t2(pi_helper, N/2,   3*N/4, h);
  std::thread t3(pi_helper, 3*N/4, N,     h);

  t0.join();  t1.join();  t2.join();  t3.join();

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```

Shared Memory Parallelism



$$\frac{\{P_1\}S_1\{Q_1\}, \{P_2\}S_2\{Q_2\} \text{ interference free}}{\{P_1\} \wedge \{P_2\} \text{ cobegin } S_1 || S_2 \text{ coend } \{Q_1 \wedge Q_2\}}$$

Distribut



This N is local to this process

This N is local to this process

But we need to read ***some*** N

These programs are all identical

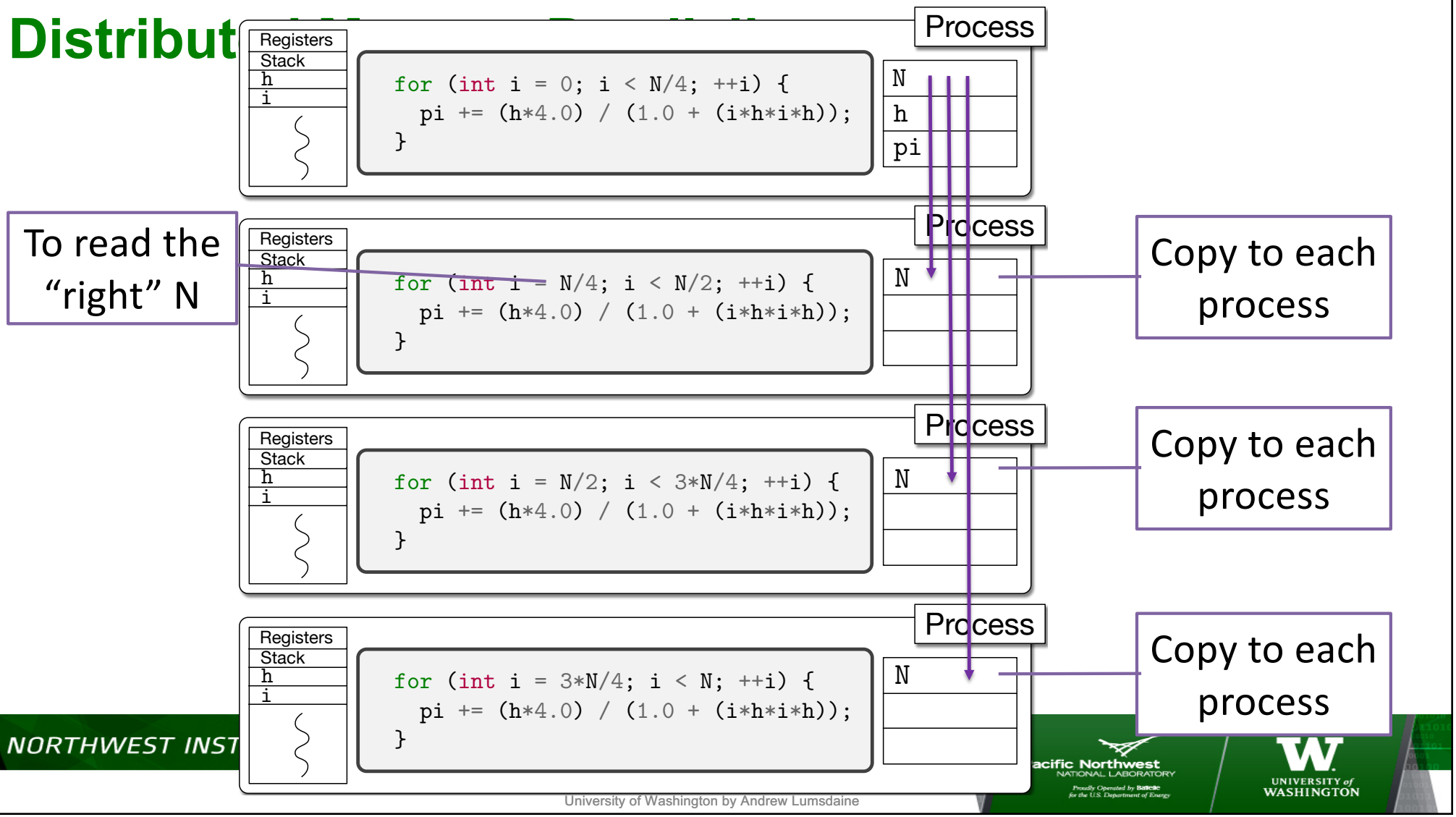
And they all say "read N"

Can ***not*** read from another process memory

(In some sense there is an N here)

How do we get the right ***value*** for N here?

Distribut


Process
Registers
Stack
h
i
for (int i = 0; i < N/4; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
}
N
h
pi
Process
Registers
Stack
h
i
for (int i = N/4; i < N/2; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
}
N
Process
Registers
Stack
h
i
for (int i = N/2; i < 3*N/4; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
}
N
Process
Registers
Stack
h
i
for (int i = 3*N/4; i < N; ++i) {
pi += (h*4.0) / (1.0 + (i*h*i*h));
}
N
To read the "right" N
Copy to each process
Copy to each process
Copy to each process

Distribut

Very Important Slide!!

Registers
Stack
h
i

```
for (int i = 0; i < N/4; ++i) {
  pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```

Process: N, h, pi

```
for (int i = N/4; i < N/2; ++i) {
  pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```

Process: N, h, pi

```
for (int i = N/2; i < 3*N/4; ++i) {
  pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```

Process: N, h, pi

```
for (int i = 3*N/4; i < N; ++i) {
  pi += (h*4.0) / (1.0 + (i*h*i*h));
}
```

Process: N, h, pi

Threads have the same value for N

And if these are all the same values

It is exactly equivalent to the sequential

Because they are reading ***copies of*** N

It is ***as if*** they were the same N

Similarly h, pi

At least for reading

Have to make consistent when writing to maintain ***as if***

$$\frac{\{P_1\}S_1\{Q_1\}, \{P_2\}S_2\{Q_2\} \text{ interference free}}{\{P_1\} \wedge \{P_2\} \text{ cobegin } S_1 || S_2 \text{ coend } \{Q_1 \wedge Q_2\}}$$

SPMD? Single program multiple data?

```
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = 0; i < N/4; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

Multiple data
(different limits)

Multiple program
(limits hard-coded)

```
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = N/2; i < 3*N/4; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

Single Program Multiple Data (SPMD)

```
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = begin; i < end; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

Multiple data (different limits)

Different, provided each process has a different begin, end

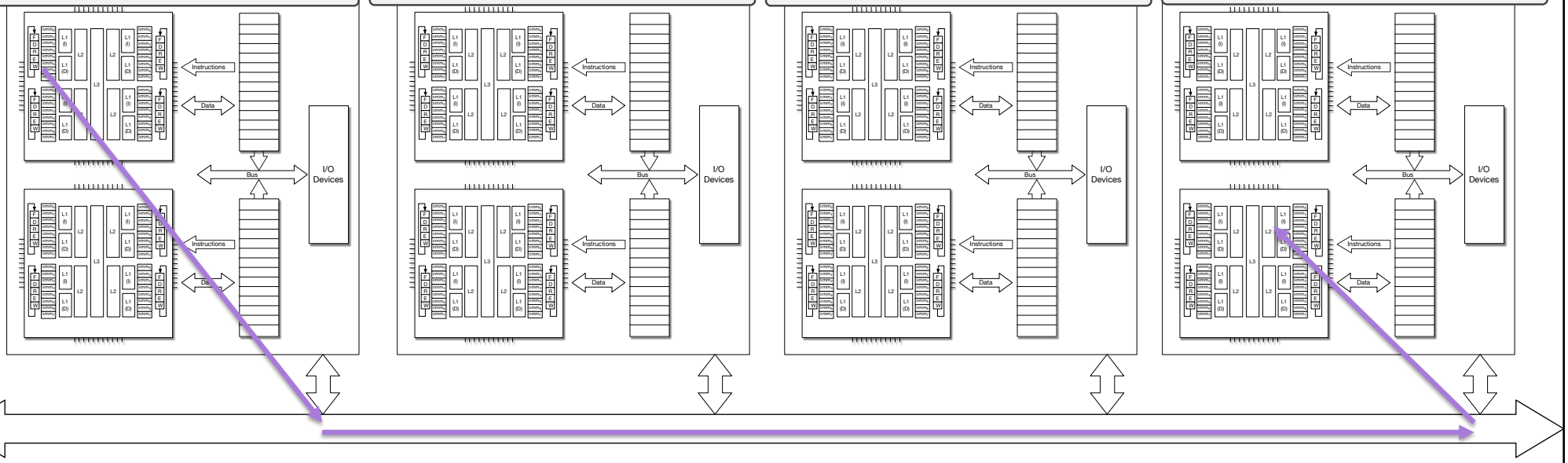
But this is now exactly the same program

Single program

```
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = begin; i < end; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

Single Program Multiple Data

```
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = 0; i < N/4; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```



Single Program Multiple Data (SPMD)

```
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = begin; i < end; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

How do we set N, begin, end?

How do we do these two things?

We need exactly the same N everywhere

```
#include <iostream>

int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = begin; i < end; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

These are exactly the same program

Each program computes same thing

But a different begin and end everywhere

Single Program Multiple Data

```
int main(size_t argc, char* argv[]) {
  size_t N    = atol(argv[1]);
  double h    = 1.0 / (double) N;
  double pi   = 0.0;

  for (size_t i = begin; i < end; ++i)
    pi_i += (h * 4.0) / (1.0 + (i * h * i

  std::cout << "pi is ~ " << pi << std::en

  return 0;
}
```

We can get N from the command line

From every node? That's a lot of typing

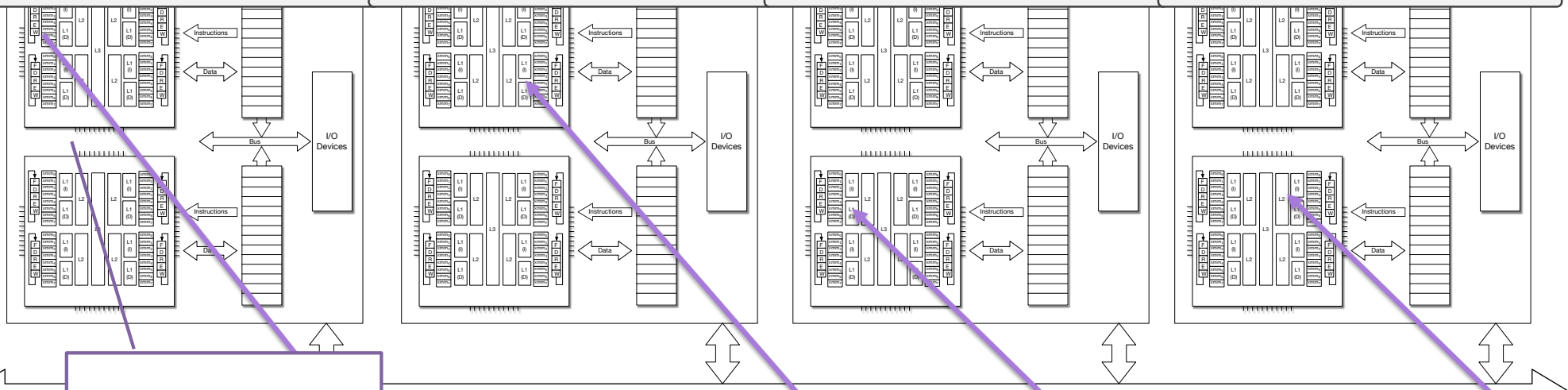
Better to get it at just one node and send it around

Single Program a

This node reads from the command line

Single program: all still read from the command line

```
int main(size_t argc, char* argv[]) {
  size_t N    = atol(argv[1]);
  double h    = 1.0 / (double) N;
  double pi   = 0.0;

  for (size_t i = begin; i < end; ++i)
    pi_i += (h * 4.0) / (1.0 + (i * h * i * h));

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```



One sends, the others receive

Single Program Multiple Data

```
int main(size_t argc, char* argv[]) {
  size_t N    = atol(argv[1]);
  double h    = 1.0 / (double) N;
  double pi   = 0.0;

  for (size_t i = begin; i < end; ++i)
    pi_i += (h * 4.0) / (1.0 + (i * h * i

  std::cout << "pi is ~ " << pi << std::e

  return 0;
```

How do we get the same program to different things

While keeping them the same?

Hint: multiple data

With, say, an if statement

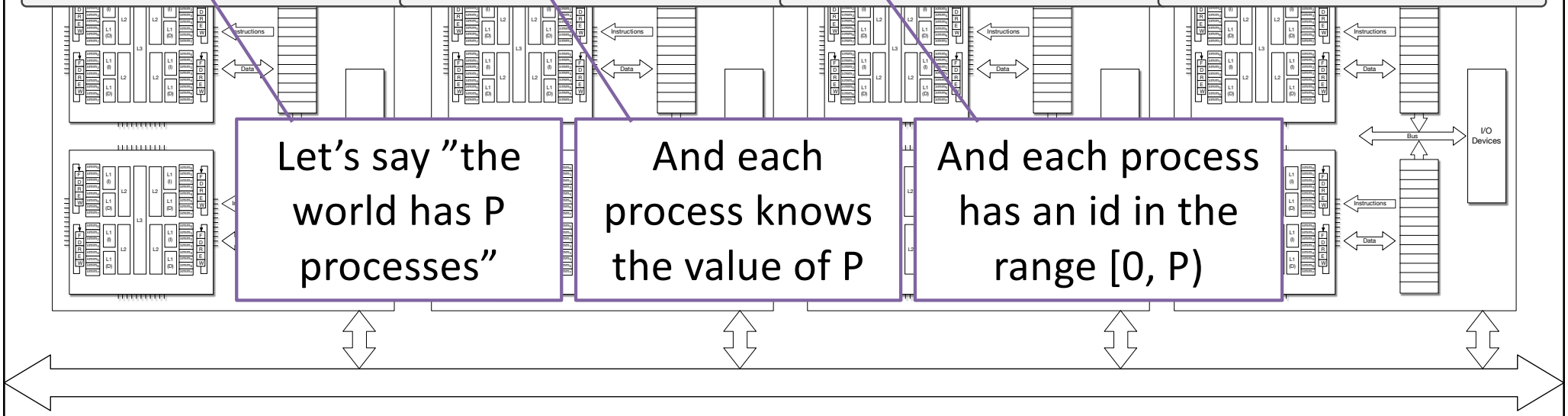
Where have we already seen identical functions that need to distinguish themselves?

fork()

How did they distinguish each other?

Single Program Multiple Data

```
int main(size_t argc, char* argv[]) {
  size_t N    = atol(argv[1]);
  double h    = 1.0 / (double) N;
  double pi   = 0.0;

  for (size_t i = begin; i < end; ++i)
    pi_i += (h * 4.0) / (1.0 + (i * h * i * h));

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```



A better name than MIMD or SPMD

Distinguished replicated processes, distributed data

(DRPDD)

Pronounced “drop dee”

Single Program

```
int main(size_t argc, char* argv[]) {
  size_t partitions = magically_get_P();
  size_t my_id      = magically_get_id();

  size_t N          = atol(argv[1]);
  size_t block_size = N / partitions;
  size_t begin      = block_size * my_id;
  size_t end        = block_size * (my_id + 1);
  double h          = 1.0 / (double) N;

  for (size_t i = begin; i < end; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```

Magically get P

Magically get id

Oops

Oops

This distinguishes the processes

Distinguished Replicated Process

```
int main(size_t argc, char* argv[]) {
  size_t partitions = magically_get_P();
  size_t my_id      = magically_get_id();

  size_t N          = -1;
  if (0 == my_id) {
    N = atol(argv[1]);
  }
  size_t block_size = N / partitions;
  size_t begin      = block_size * my_id;
  size_t end        = block_size * (my_id + 1);
  double h          = 1.0 / (double) N;

  for (size_t i = begin; i < end; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));

  if (0 == my_id) {
    std::cout << "pi is ~ " << pi << std::endl;
  }

  return 0;
}
```

No and no.

Only one node reads N

Is that going to be correct?

Compute begin and end

Is that going to be correct?

Only one node prints pi

Distinguished Replicated Process

```
int main(size_t argc, char* argv[]) {
  size_t partitions = magically_get_P();
  size_t my_id      = magically_get_id();

  size_t N          = -1;
  if (0 == my_id) {
    N = atol(argv[1]);
  }
  size_t block_size = N / partitions;
  size_t begin      = block_size * my_id;
  size_t end        = block_size * (my_id + 1);
  double h          = 1.0 / (double) N;

  for (size_t i = begin; i < end; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));

  if (0 == my_id) {
    std::cout << "pi is ~ " << pi << std::endl;
  }

  return 0;
}
```

What is this value?

Distingu eplicate ses

my_id == 0

my_id == 1

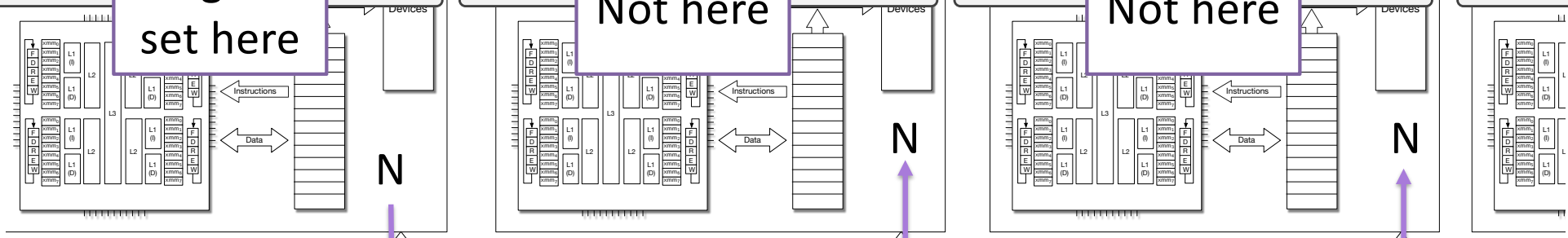
my_id == 2

```
int main(size_t argc, char* argv[]) {
  size_t partitions = magically_get_P();
  size_t my_id      = magically_get_id();

  size_t N          = -1;
  if (0 == my_id) {
    N = atol(argv[1]);
  }
  size_t block_size = N / partitions;
  size_t begin      = block_size * my_id;
  size_t end        = block_size * (my_id + 1);
  double h          = 1.0 / (double) N;

  for (size_t i = begin; i < end; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));

  if (0 == my_id) {
    std::cout << "pi is ~ " << pi << std::endl;
  }

  return 0;
}
```

N gets set here

Not here

Not here



Finally

Get our id and number of other nodes

id 0 gets N

id shares N

This pattern is ubiquitous

Everyone computes their own partial

id 0 collects all partials, adds them, and prints

```
int main(size_t argc, char* argv[]) {
  size_t partitions = magically_get_P();
  size_t my_id      = magically_get_id();

  size_t N          = -1;
  if (0 == my_id) {
    N = atol(argv[1]);
  }
  N                 = magically_share(N);
  size_t block_size = N / partitions;
  size_t begin      = block_size * my_id;
  size_t end        = block_size * (my_id + 1);
  double h          = 1.0 / (double) N;

  for (size_t i = begin; i < end; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));

  if (0 == my_id) {
    pi              = magically_combine(pi);
    std::cout << "pi is ~ " << pi << std::endl;
  }

  return 0;
}
```

MPI

Get our id and number of other nodes

id 0 gets N

This pattern is ubiquitous

id shares N

Everyone has same N

Everyone computes their own partial

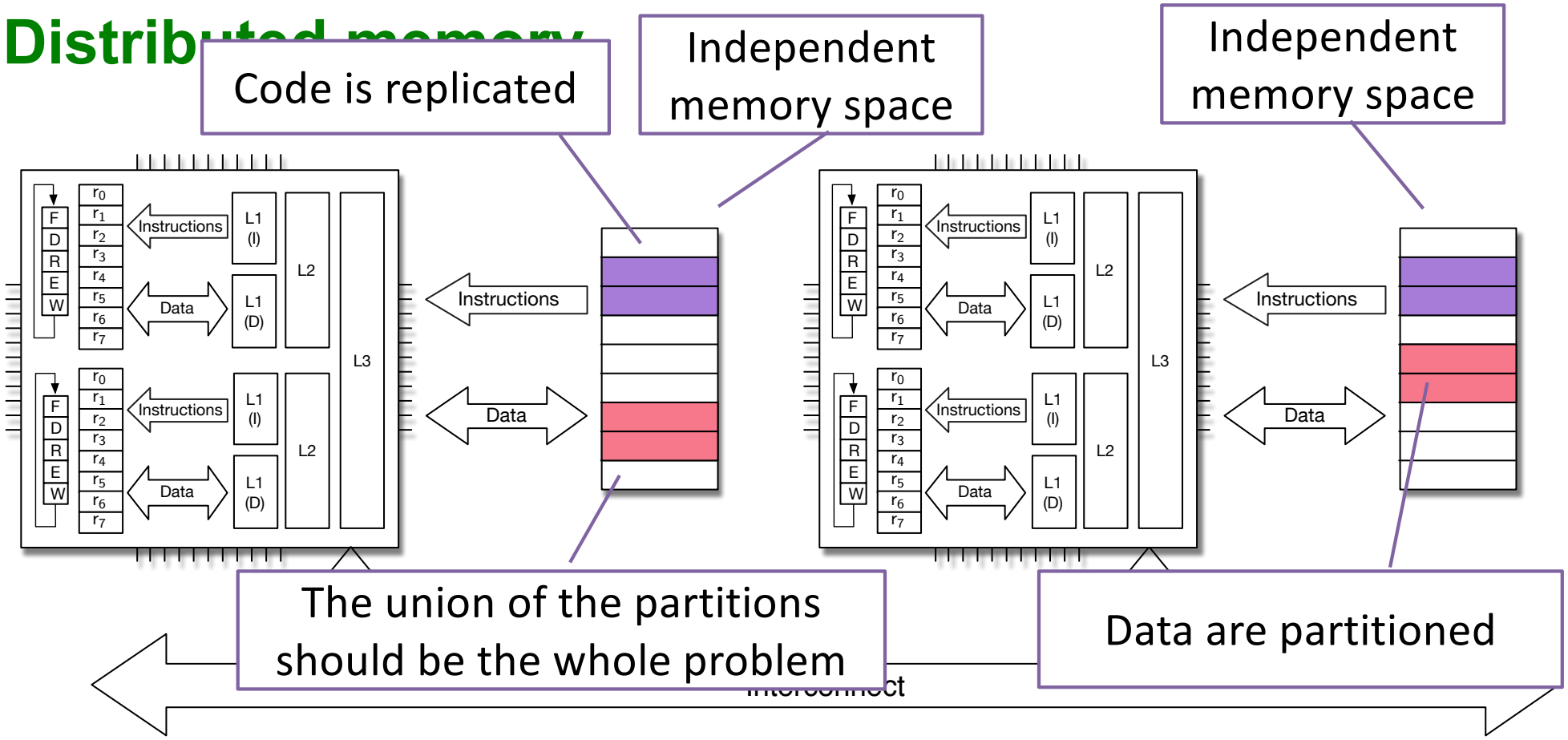
id 0 collects all partials, adds them, and prints

id 0 is root

```
int main(int argc, char* argv[]) {
  size_t intervals = 1024 * 1024;

  MPI::Init();

  int myrank = MPI::COMM_WORLD.Get_rank();
  int mysize = MPI::COMM_WORLD.Get_size();

  if (0 == myrank) {
    if (argc >= 2) intervals = std::atol(argv[1]);
  }

  MPI::COMM_WORLD.Bcast(&intervals, 1, MPI::UNSIGNED_LONG, 0);

  size_t blocksize = intervals / mysize;
  size_t begin     = blocksize * myrank;
  size_t end       = blocksize * (myrank + 1);
  double h         = 1.0 / ((double)intervals);

  double pi        = 0.0;
  for (size_t i = begin; i < end; ++i) {
    pi += 4.0 / (1.0 + (i * h * i * h));

  MPI::COMM_WORLD.Reduce(&mypi, &pi, 1, MPI::DOUBLE, MPI::SUM, 0);

  if (0 == myrank) {
    std::cout << "pi is approximately " << pi << std::endl;
  }

  MPI::Finalize();

  return 0;
}
```

The Message Passing Interface (MPI)

Distrib


Code is replicated
Independent memory space
Independent memory space
F
D
R
E
W
r_0
r_1
r_2
r_3
r_4
r_5
r_6
r_7
Instructions
Data
L1 (I)
L1 (D)
L2
L3
The union of the partitions should be the whole problem
Data are partitioned
Interconnect

Finding Concurrency

$$h \sum_{i=0}^{N/4-1} \frac{4}{1+(ih)^2}$$

$$h \sum_{i=N/4}^{N/2-1} \frac{4}{1+(ih)^2}$$

$$h \sum_{i=N/2}^{3N/4-1} \frac{4}{1+(ih)^2}$$

$$h \sum_{i=3N/4}^{N-1} \frac{4}{1+(ih)^2}$$

```
int main() {
  double pi = 0.0;  int N = 1024*1024;

  for (int i = 0; i < N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/4; i < N/2; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/2; i < 3*N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = 3*N/4; i < N; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  std::cout << "pi ~ " << pi << std::endl;
  return 0;
}
```

Processes

```
int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = 0; i < N/4; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << st
```

Process

```
int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = N/4; i < N/2; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << st
```

Process

```
int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = N/2; i < 3*N/4; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << st
```

Process

```
int main() {
  double pi = 0.0; double h = 1./(double) N;
  for (size_t i = 3*N/4; i < N; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  std::cout << "pi is ~ " <<  pi << std::endl

  return 0;
}
```

```
double pi = 0.0;

void pi_helper(int begin, int end, double h) {
  for (int i = begin; i < end; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}

int main(int argc, char* argv[]) {
  int N = 1024 * 1024; double h = 1.0/ (double)N;

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```

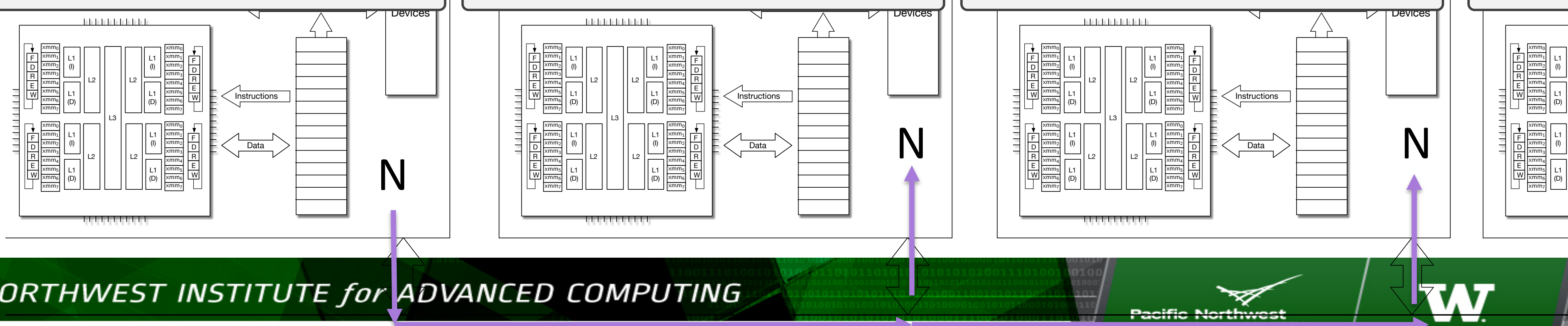
Distinguished Replicated Processes

```cpp
int main(size_t argc, char* argv[]) {
  size_t partitions = magically_get_P();
  size_t my_id      = magically_get_id();

  size_t N          = -1;
  if (0 == my_id) {
    N = atol(argv[1]);
  }
  size_t block_size = N / partitions;
  size_t begin      = block_size * my_id;
  size_t end        = block_size * (my_id + 1);
  double h          = 1.0 / (double) N;

  for (size_t i = begin; i < end; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));

  if (0 == my_id) {
    std::cout << "pi is ˜ " << pi << std::endl;
  }

  return 0;
}
```


```cpp
int main(size_t argc, char* argv[]) {
  size_t partitions = magically_get_P();
  size_t my_id      = magically_get_id();

  size_t N          = -1;
  if (0 == my_id) {
    N = atol(argv[1]);
  }
  size_t block_size = N / partitions;
  size_t begin      = block_size * my_id;
  size_t end        = block_size * (my_id + 1);
  double h          = 1.0 / (double) N;

  for (size_t i = begin; i < end; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));

  if (0 == my_id) {
    std::cout << "pi is ˜ " << pi << std::endl;
  }

  return 0;
}
```

```cpp
int main(size_t argc, char* argv[]) {
  size_t partitions = magically_get_P();
  size_t my_id      = magically_get_id();

  size_t N          = -1;
  if (0 == my_id) {
    N = atol(argv[1]);
  }
  size_t block_size = N / partitions;
  size_t begin      = block_size * my_id;
  size_t end        = block_size * (my_id + 1);
  double h          = 1.0 / (double) N;

  for (size_t i = begin; i < end; ++i)
    pi += (h * 4.0) / (1.0 + (i * h * i * h));

  if (0 == my_id) {
    std::cout << "pi is ˜ " << pi << std::endl;
  }

  return 0;
}
```


```cpp
int main(size_
  size_t parti
  size_t my_id

  size_t N
  if (0 == my_
    N = atol(a
  }
  size_t block
  size_t begin
  size_t end
  double h

  for (size_t
    pi += (h *

  if (0 == my_
    std::cout
  }

  return 0;
}
```



MPI

Get our id and number of other nodes

id 0 gets N

This pattern is ubiquitous

id shares N

Everyone has same N

Everyone computes their own partial

id 0 collects all partials, adds them, and prints

id 0 is root

```
int main(int argc, char* argv[]) {
  size_t intervals = 1024 * 1024;

  MPI::Init();

  int myrank = MPI::COMM_WORLD.Get_rank();
  int mysize = MPI::COMM_WORLD.Get_size();

  if (0 == myrank) {
    if (argc >= 2) intervals = std::atol(argv[1]);
  }

  MPI::COMM_WORLD.Bcast(&intervals, 1, MPI::UNSIGNED_LONG, 0);

  size_t blocksize = intervals / mysize;
  size_t begin     = blocksize * myrank;
  size_t end       = blocksize * (myrank + 1);
  double h         = 1.0 / ((double)intervals);

  double pi        = 0.0;
  for (size_t i = begin; i < end; ++i) {
    pi += 4.0 / (1.0 + (i * h * i * h));

  MPI::COMM_WORLD.Reduce(&mypi, &pi, 1, MPI::DOUBLE, MPI::SUM, 0);

  if (0 == myrank) {
    std::cout << "pi is approximately " << pi << std::endl;
  }

  MPI::Finalize();

  return 0;
}
```

The Message Passing Interface (MPI)

Thank You!

Creative Commons BY-NC-SA 4.0 License