

AMATH 483/583

High Performance Scientific

Lecture 17: Computing
Distributed memory, communicating sequential processes

Xu Tony Liu, PhD

Paul G. Allen School of Computer Science & Engineering

University of Washington

Seattle, WA

Overview

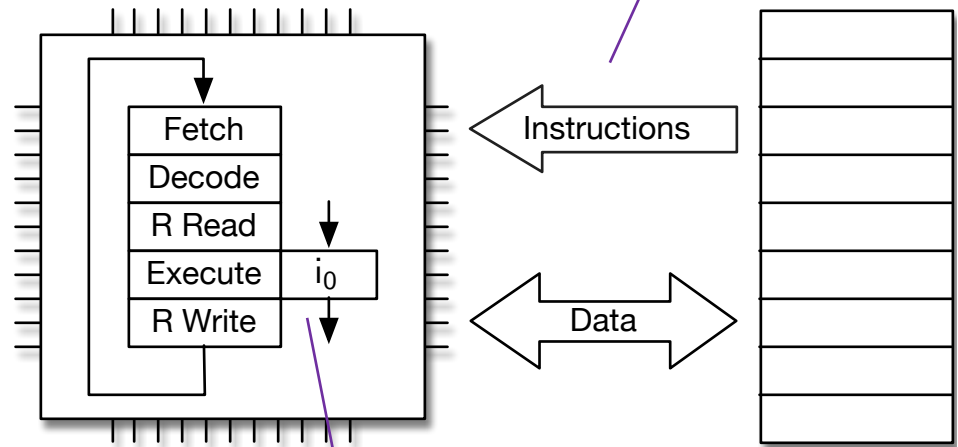
- Distributed memory systems
- Communicating sequential processes
- Message passing
- The message passing interface



Scaling progression of CPUs

Simplest model

CPU fetches and executes instructions

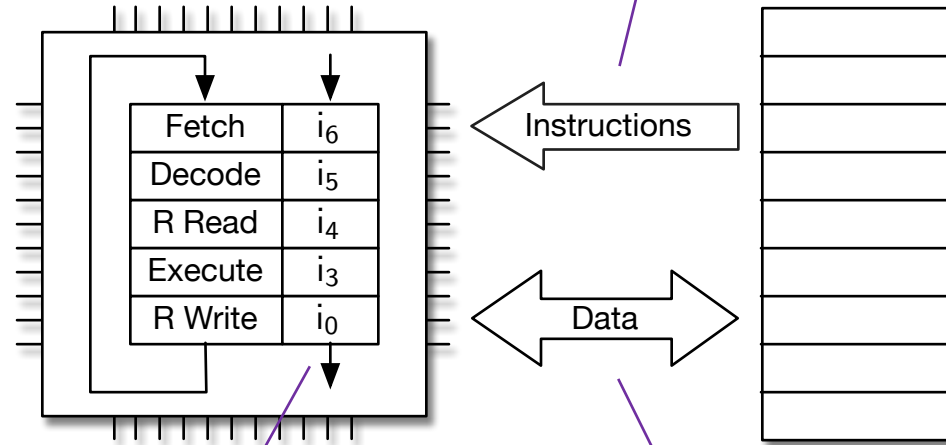


Many cycles per instruction

Pipelining

Pipelining

Instructions are fetched in a stream

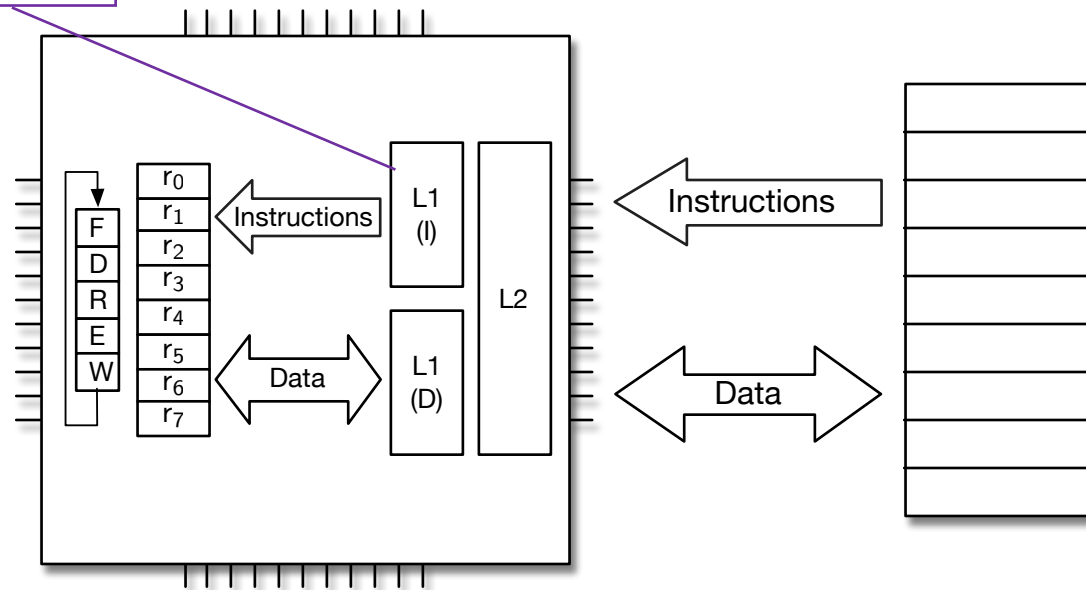


Processed in a pipeline

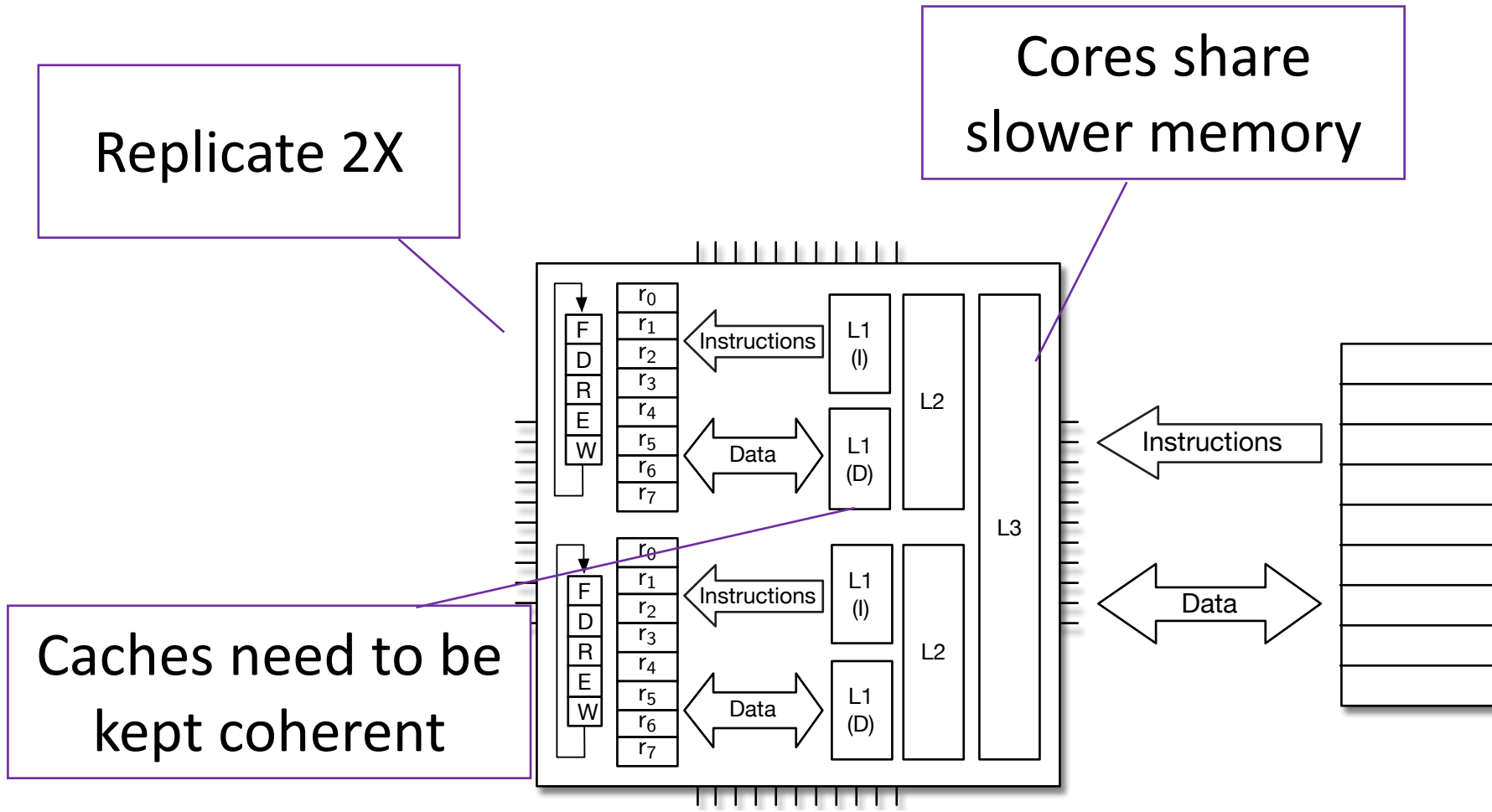
A long trip from memory

Hierarchical memory

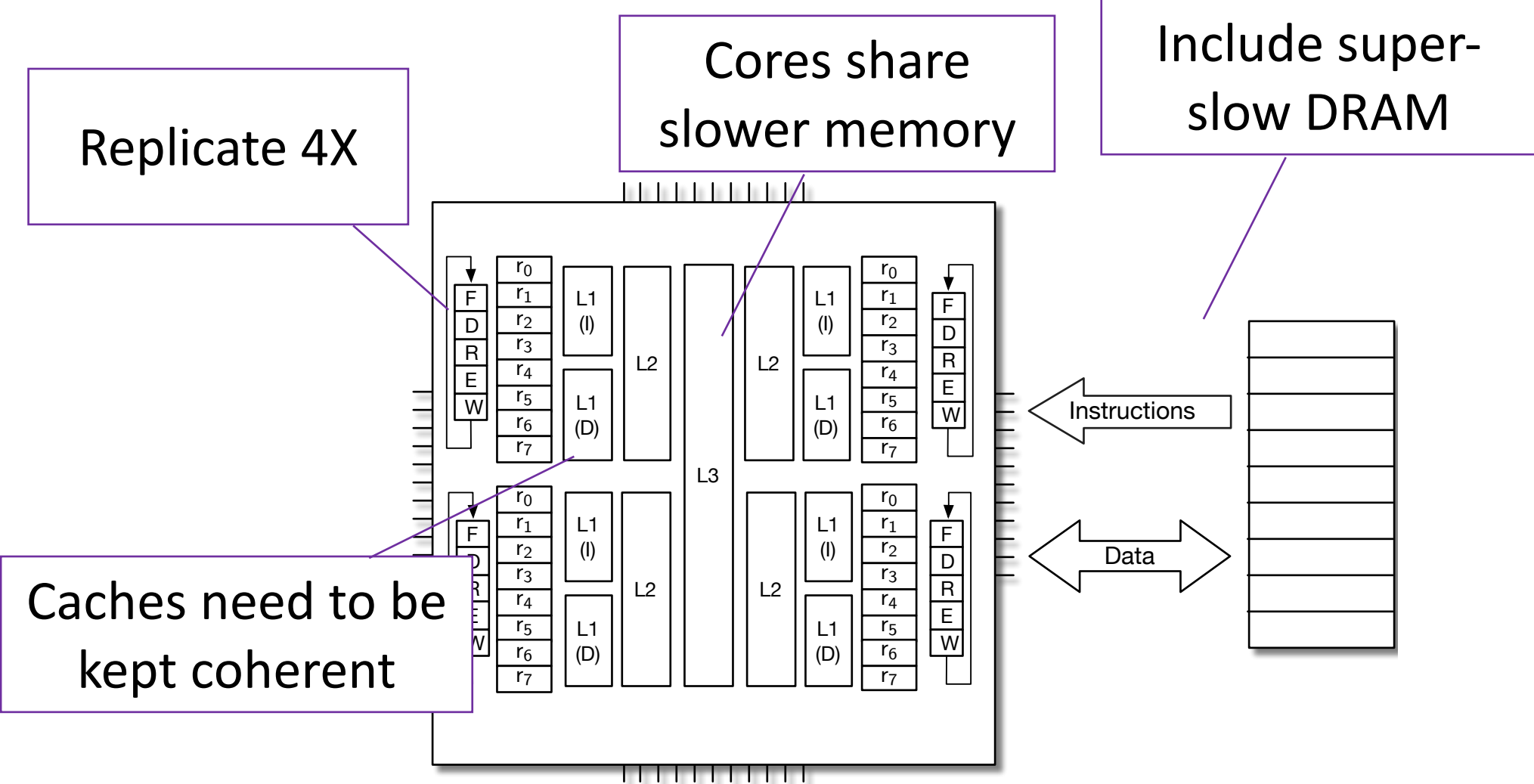
Use special, fast memory to keep data and instructions close



Multicore CPUs



Even more cores

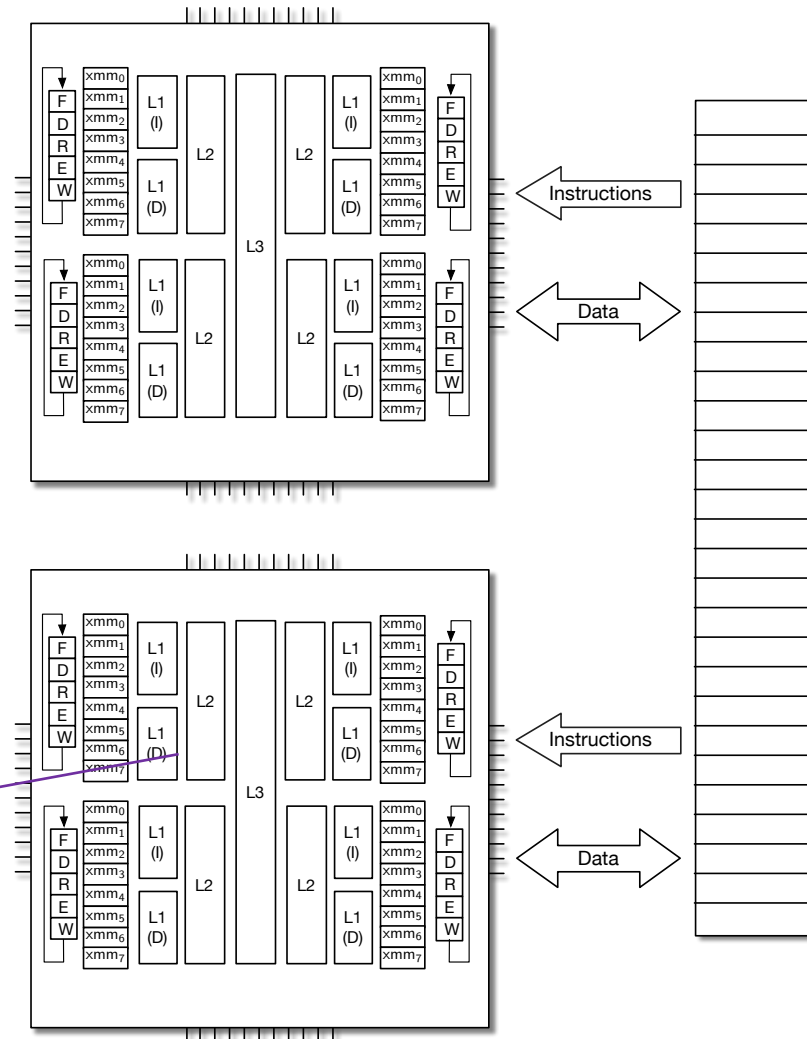


Symmetric Multi-Processor (SMP)

Multiple CPU chips

AKA "sockets"

Caches still need to be kept (somewhat) coherent



Memory may be uniformly shared among sockets

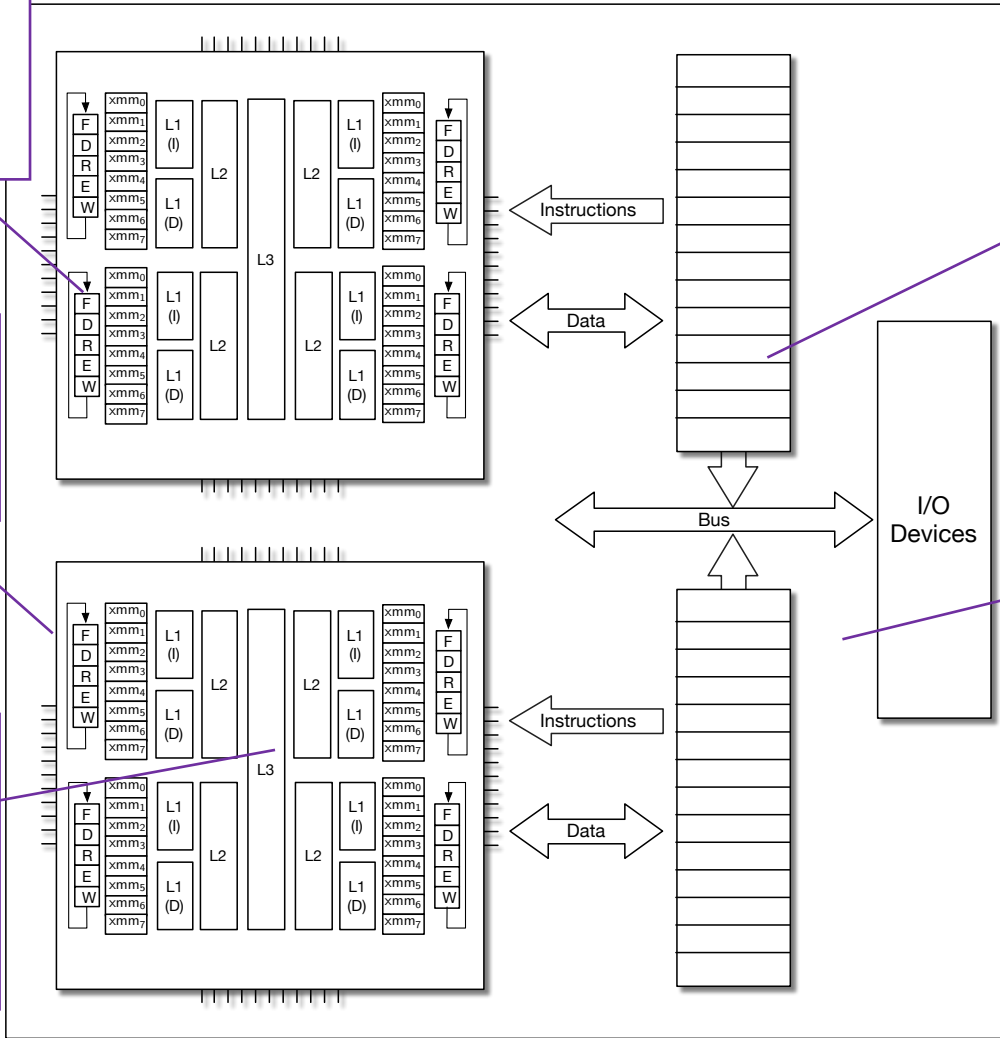
Uniform memory access (UMA)

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

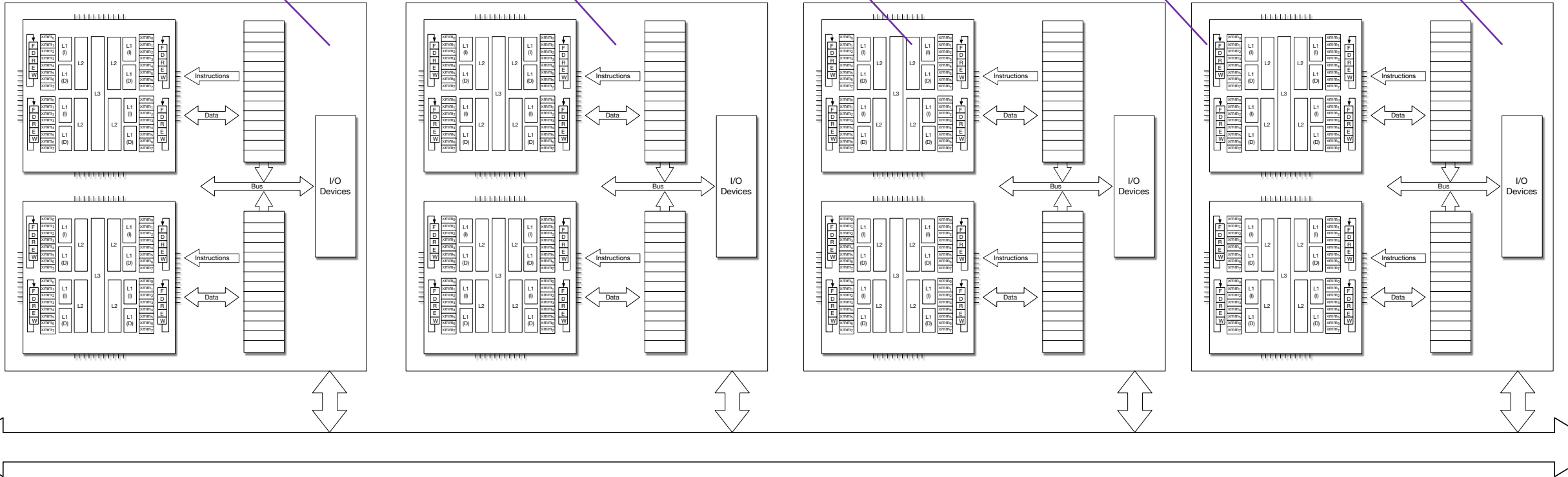
Put sockets on a blade

Put blades in a chassis

Put chassis in a rack

Put racks in a center

Put centers in the cloud



Then you have a supercomputer

But how do you use it?



Need More Power? Buy More Hardware!

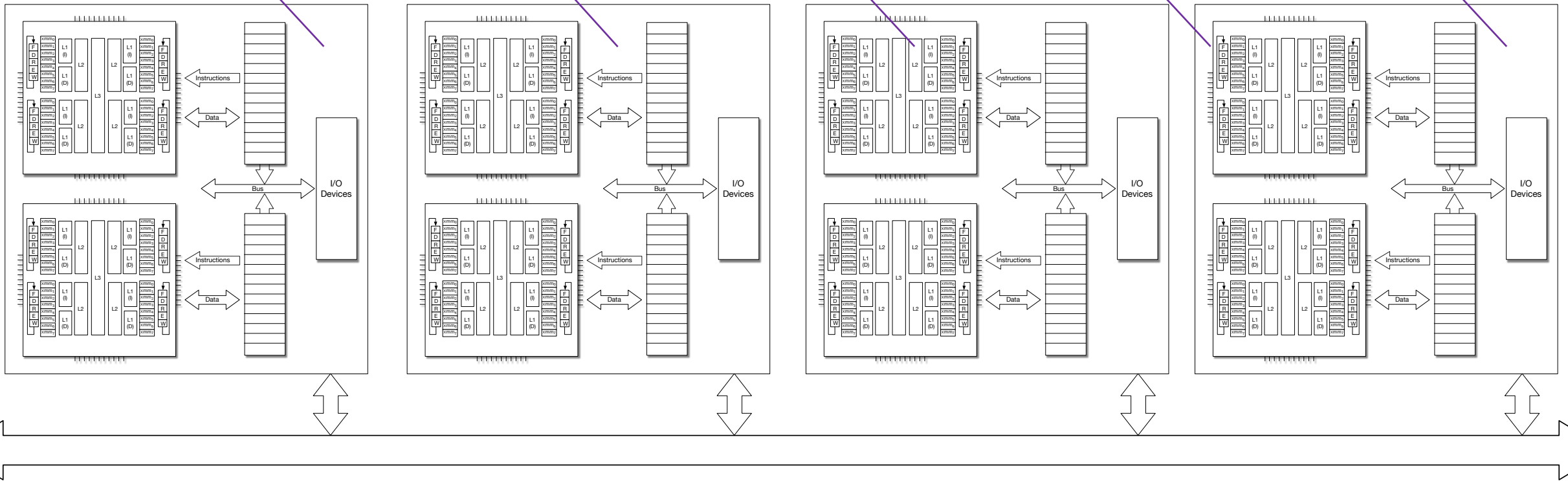
More cores!

More blades!

More chassis!

More racks!

More centers!



Top500 as of Nov 2021 (top500.org)

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)	
1	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899	7.6M cores
2	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096	2.4M cores
3	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438	1.6M cores
4	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371	10.6M cores

There are no parallel computers



It's really just a bunch of computers

Separate
memory

(Each has its
own memory)

(Each has its
own storage)



(Each has its
own OS)

There are no parallel programs



It's really just a bunch of programs

Separate
memory

(Each has its
own memory)

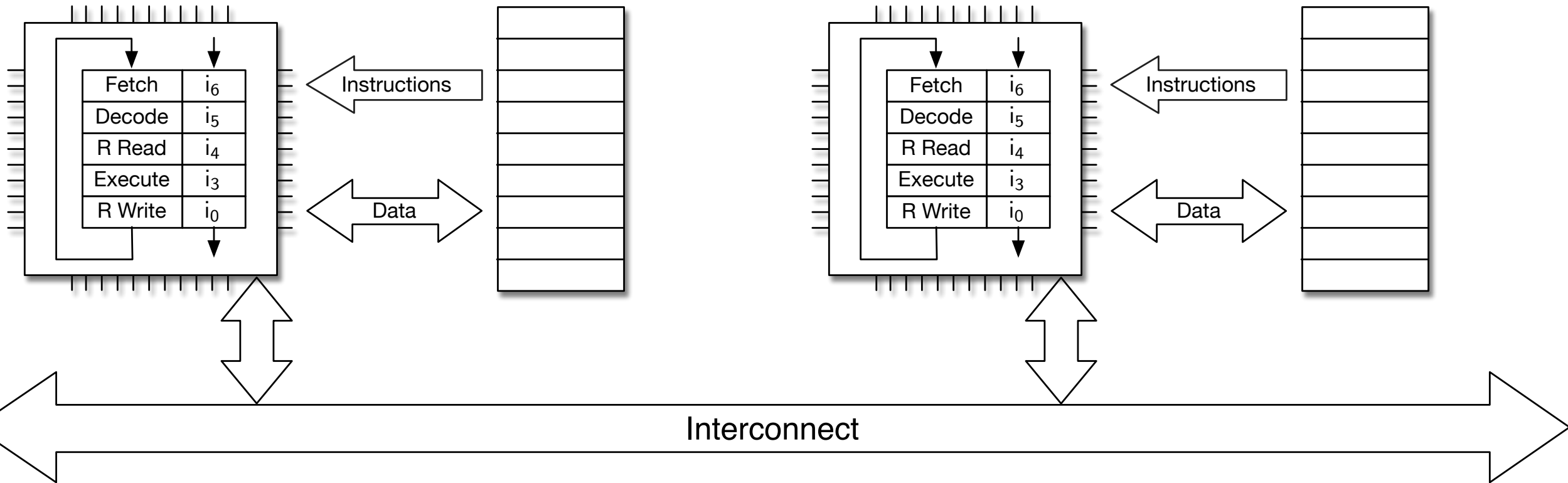
(Each has its
own storage)



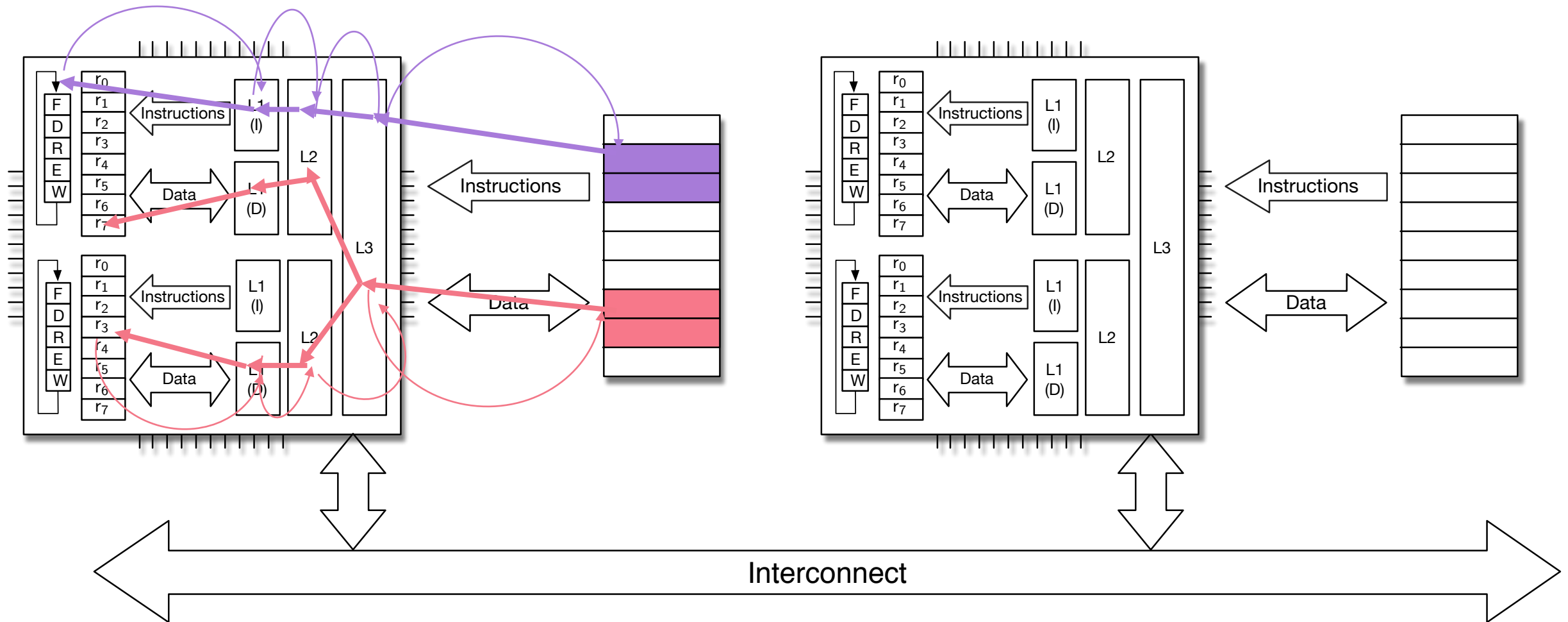
(Each has its
own OS)

(Each runs its
own programs)

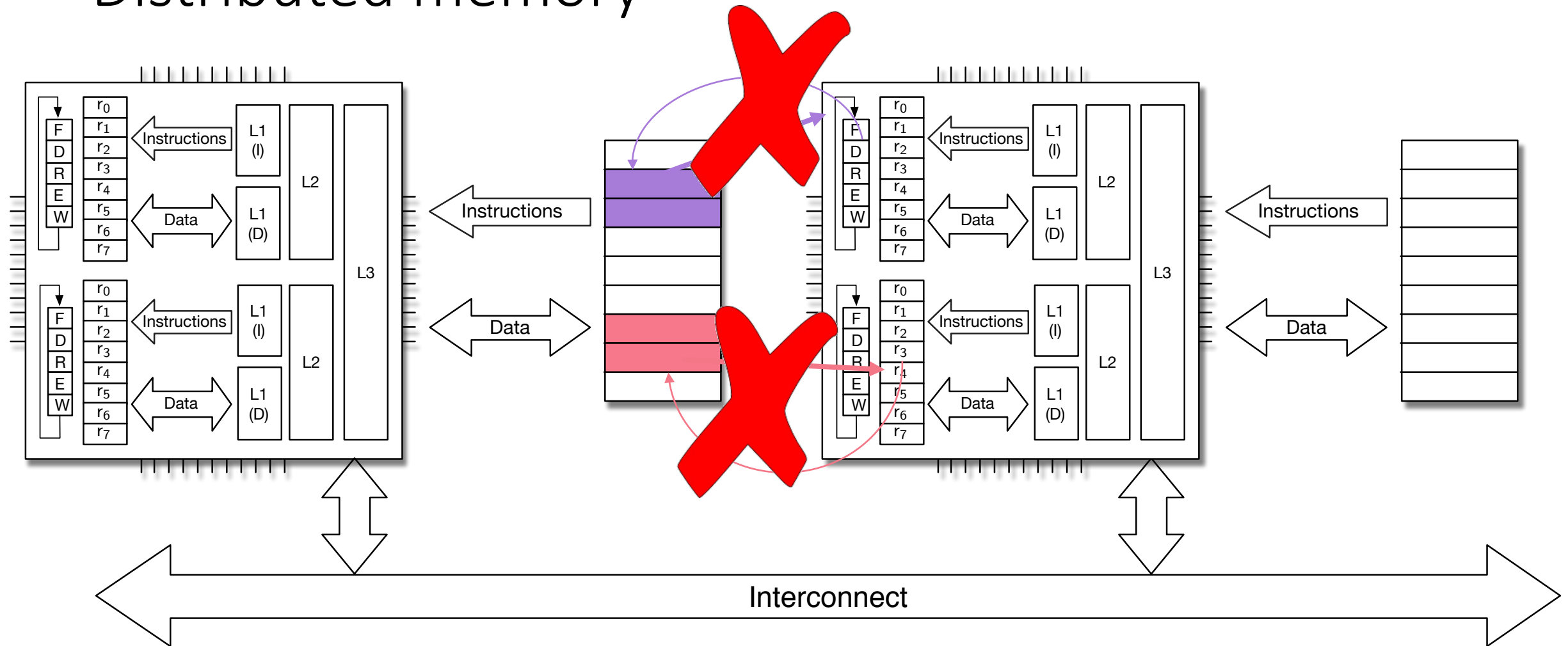
Distributed memory



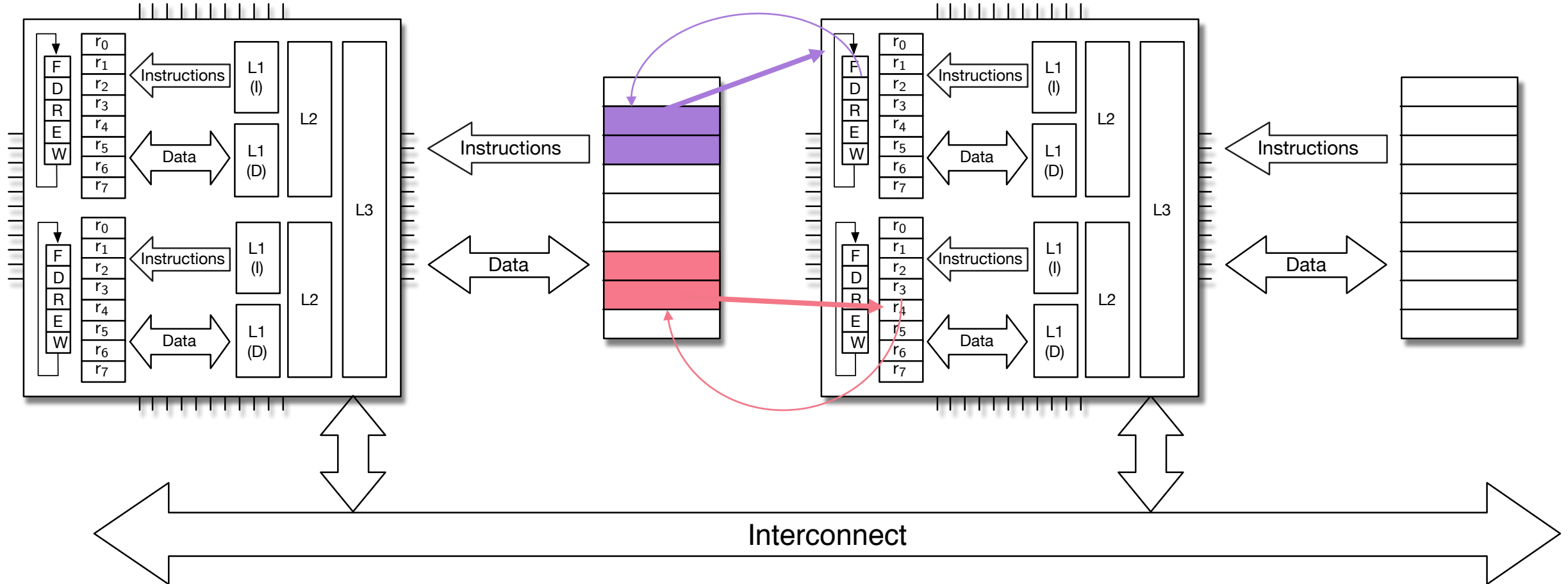
Distributed memory



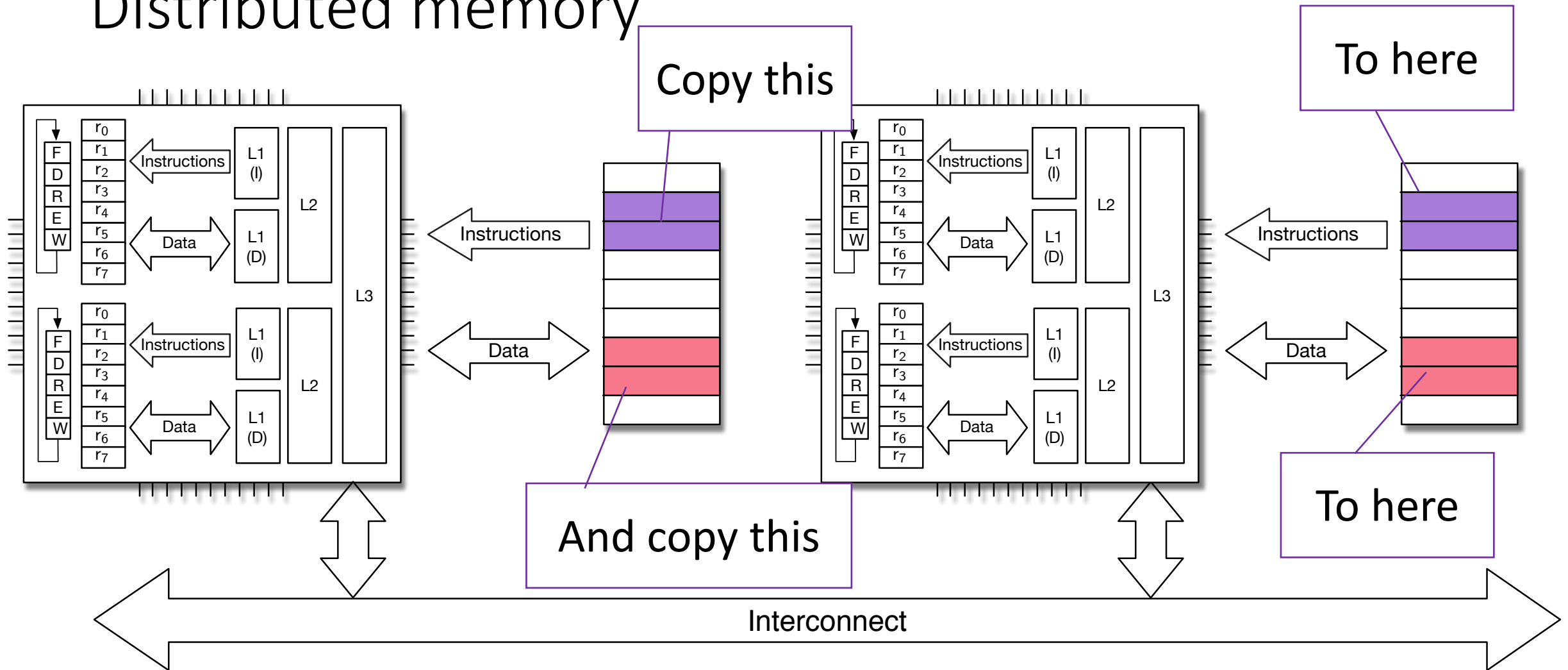
Distributed memory



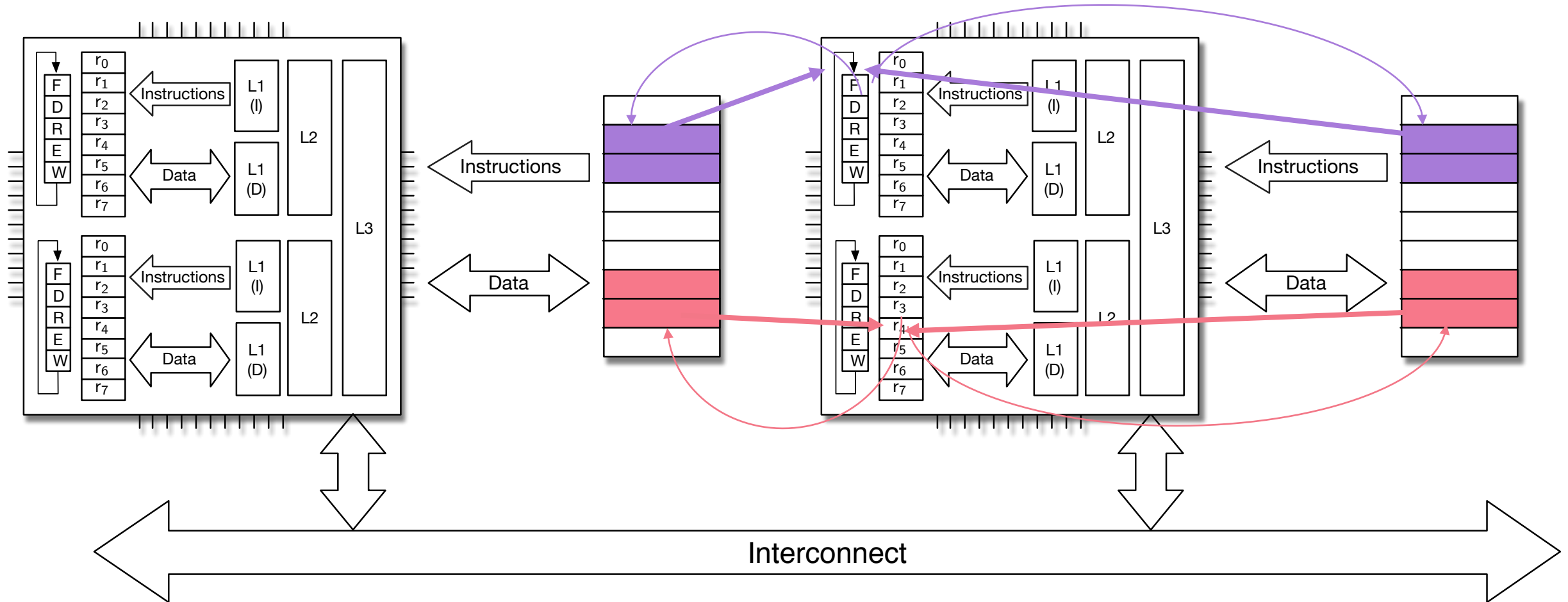
Distributed memory



Distributed memory

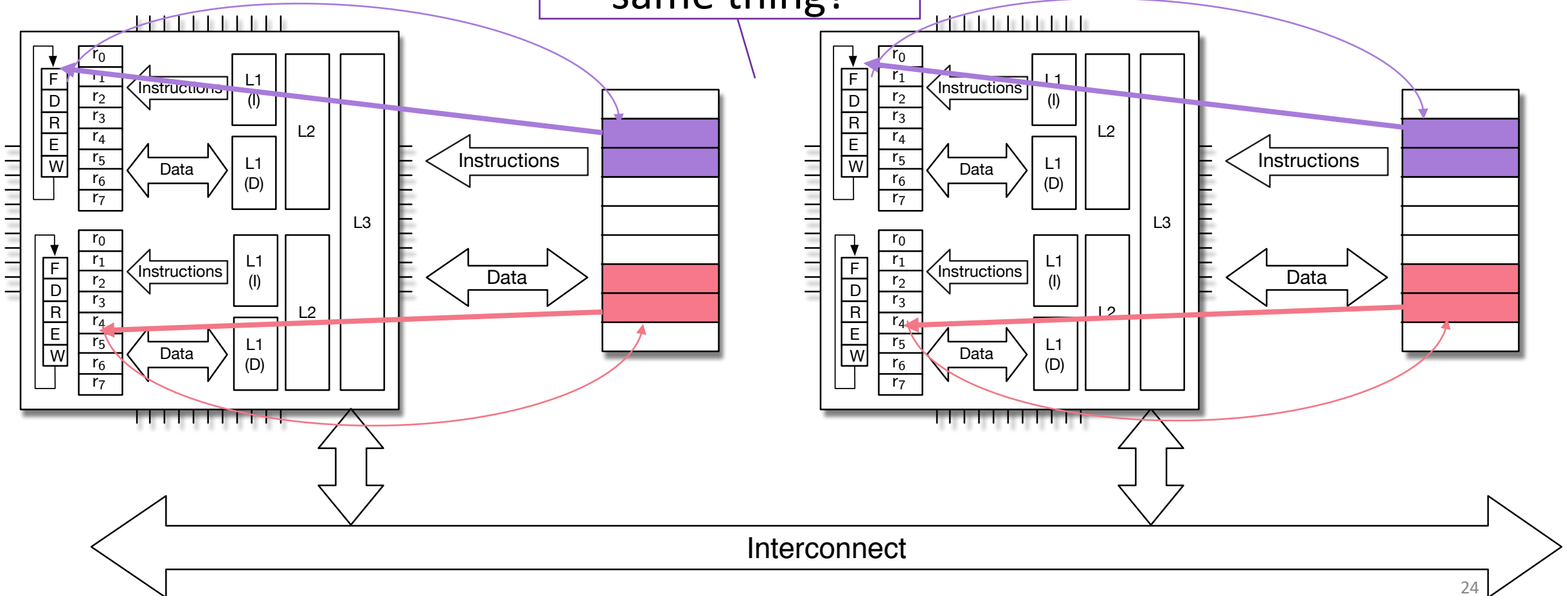


Distributed memory



Distributed memory

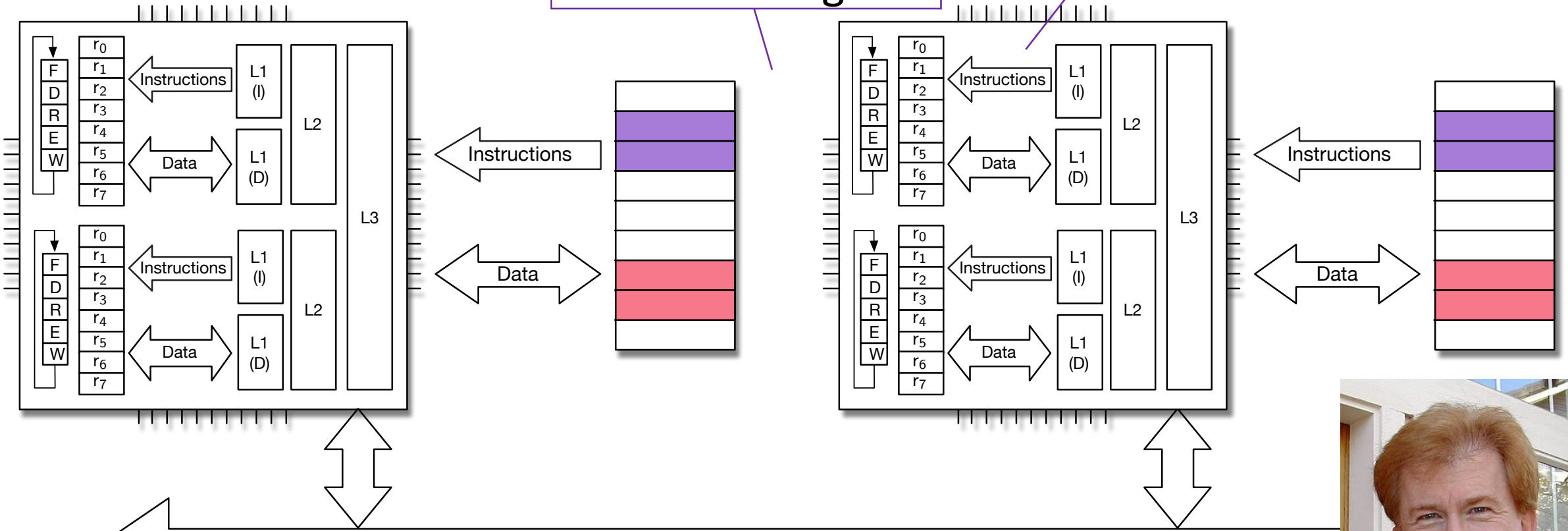
Do we want these
doing the exact
same thing?



Distributed mem

Do we want these doing the exact same thing?

Will there be speedup if we do?



What was his law?

Remember this famous person?

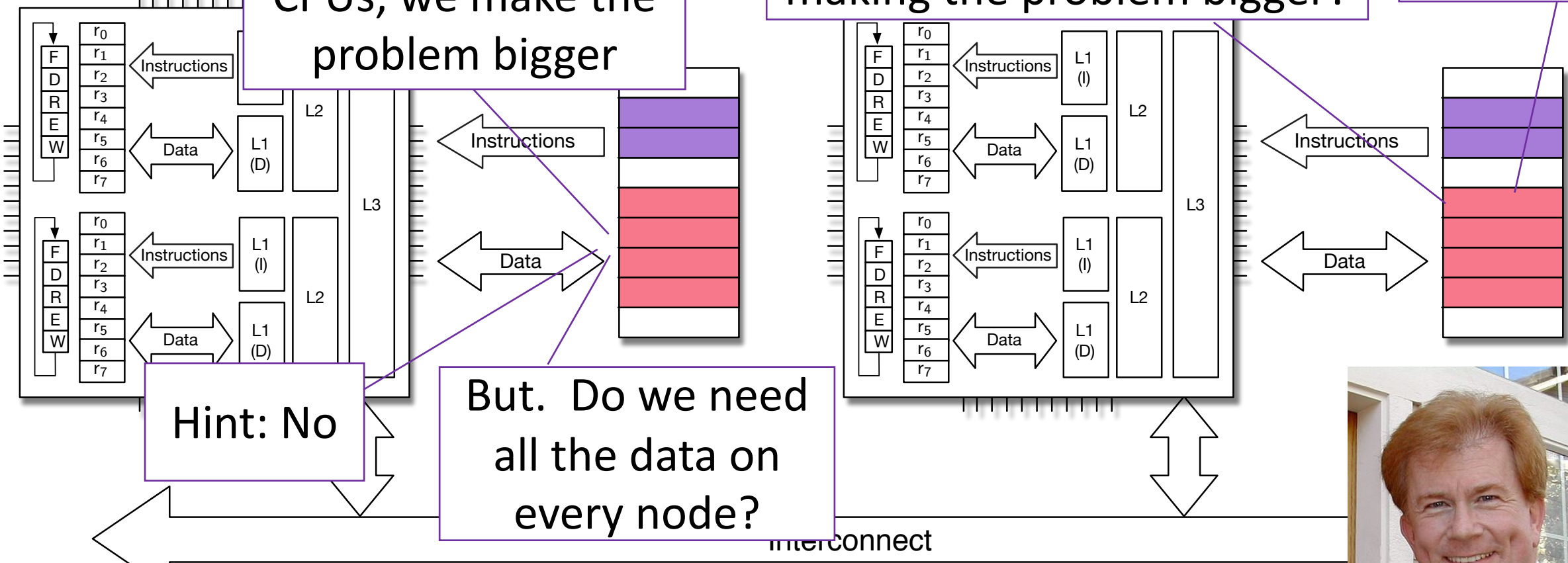


Distributed memory

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

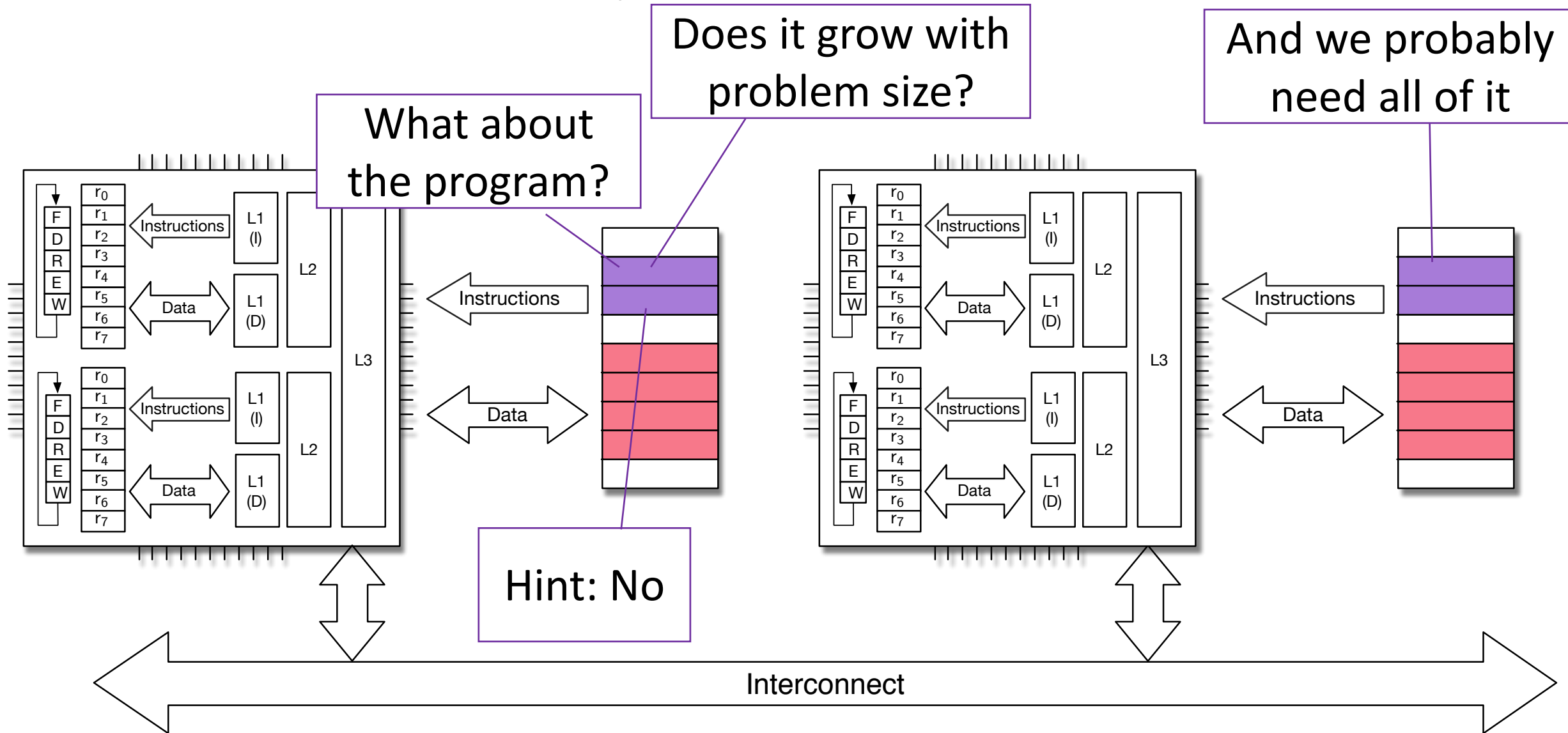
But. Do we need all the data on every node?

Gustafson's Law

Gustafson

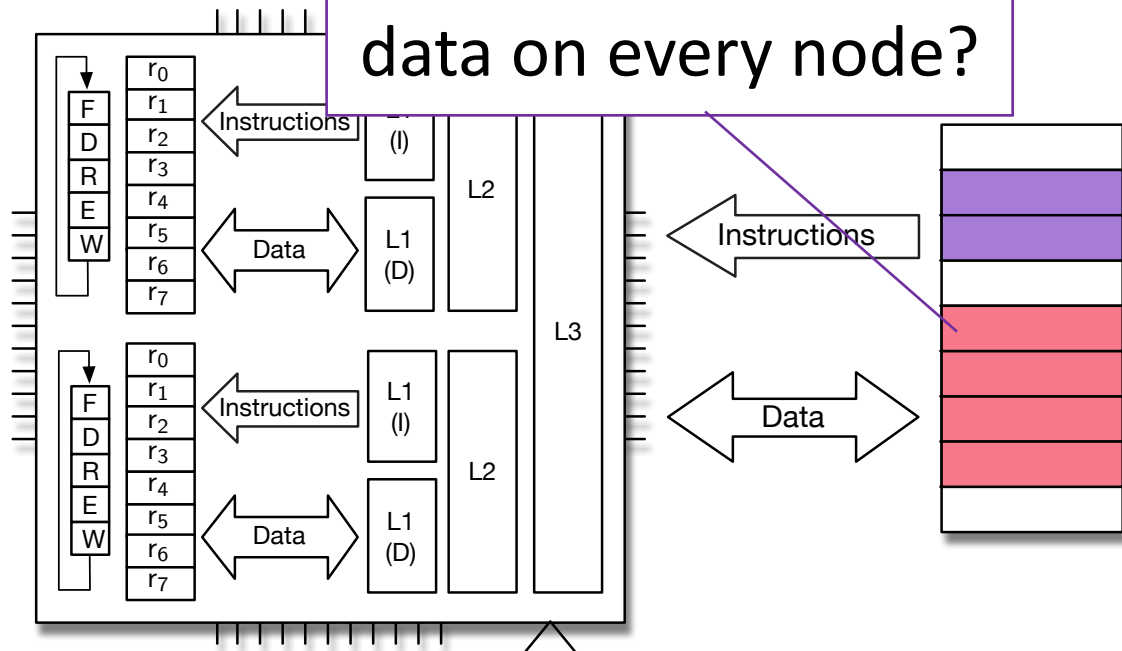


Distributed memory

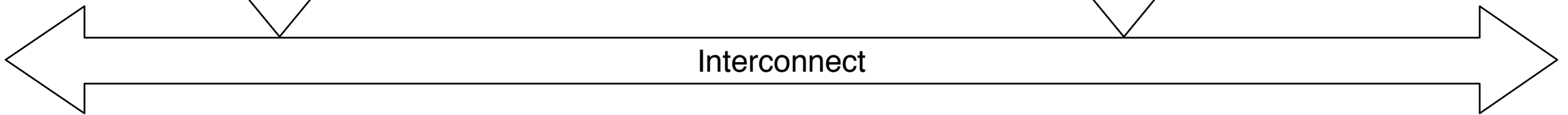
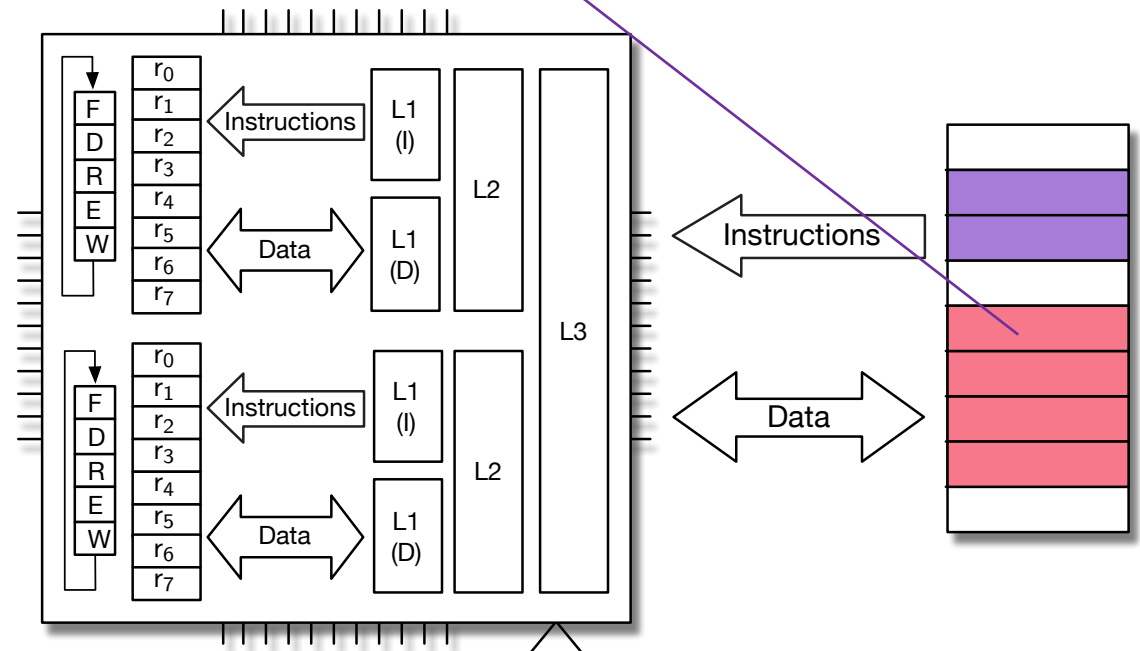


Distributed memory

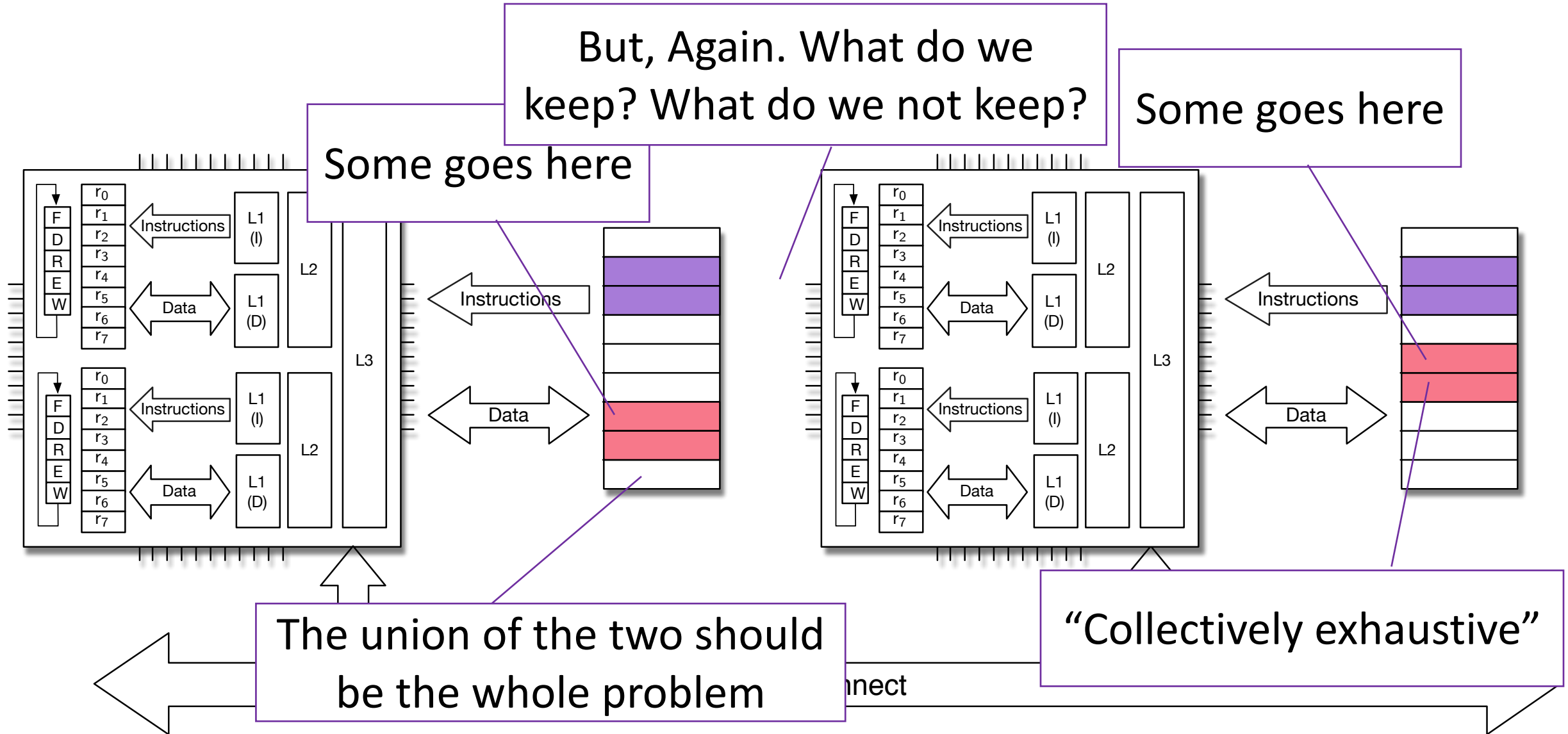
Do we need all the data on every node?



What do we keep?
What do we not keep?



Distributed memory



Name this famous person



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



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

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

Show more

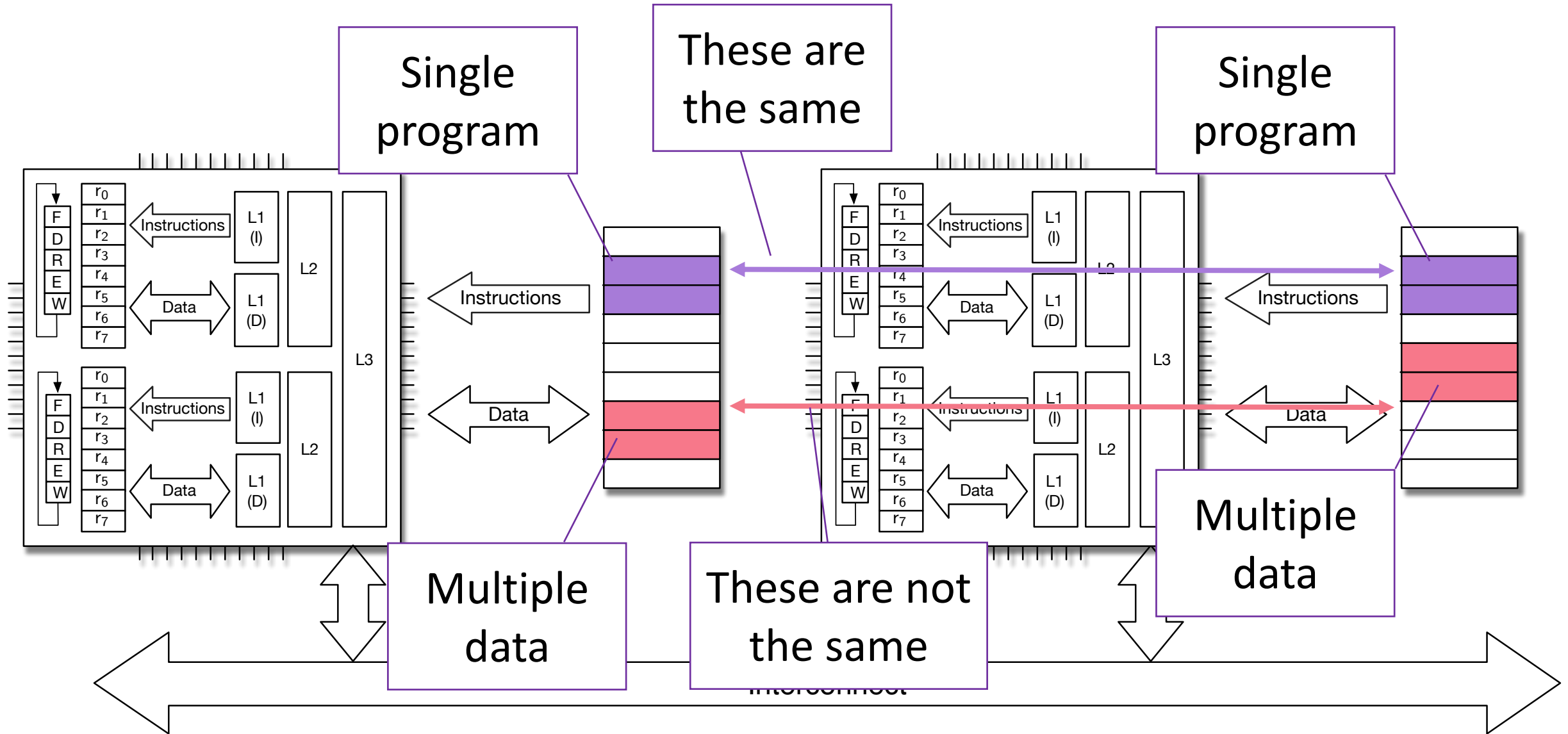
Single program
multiple data
model (SPMD)

Most widely used
model in distributed
memory programming

08)90094-4 [Get rights and content](#)

single-program-multiple-data computational model which we have
system to run in parallel mode FORTRAN scientific
computational 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
appropriate synchronizing constructs that are imbedded in the program. We have
demonstrated the applicability of the model in the parallelization of several
applications. We discuss parallelization features of these applications and
performance issues such as overhead, speedup, efficiency.

SPMD



Name this famous person



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

Better model for
today's practice
than Flynn's

Most widely used
model in distributed
memory programming

Single program
multiple data
model (SPMD)

How do you
pronounce
"SPMD"?

Recall Flynn:
SIMD, MIMD

SPMD is pronounced
"spim dee"



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

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

Show more

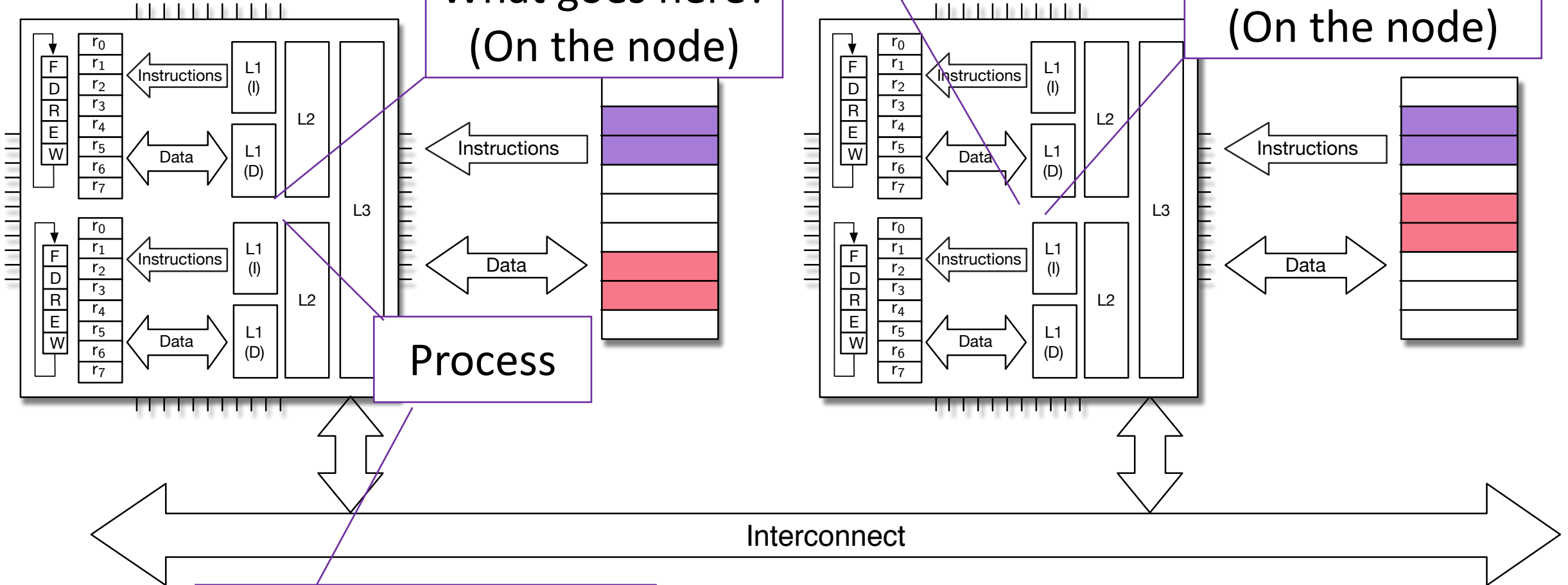
organization and is based on the sch
parallel remain in existence for the e
executed by each process are determ
appropriate synchronizing constru
strated the applicability of the
ions. We discuss parallelization featur
ance issues such as overhead, speedup, efficiency.

Distributed memory

What goes here?
(On the node)

Hint: Resource
allocation

What goes here?
(On the node)

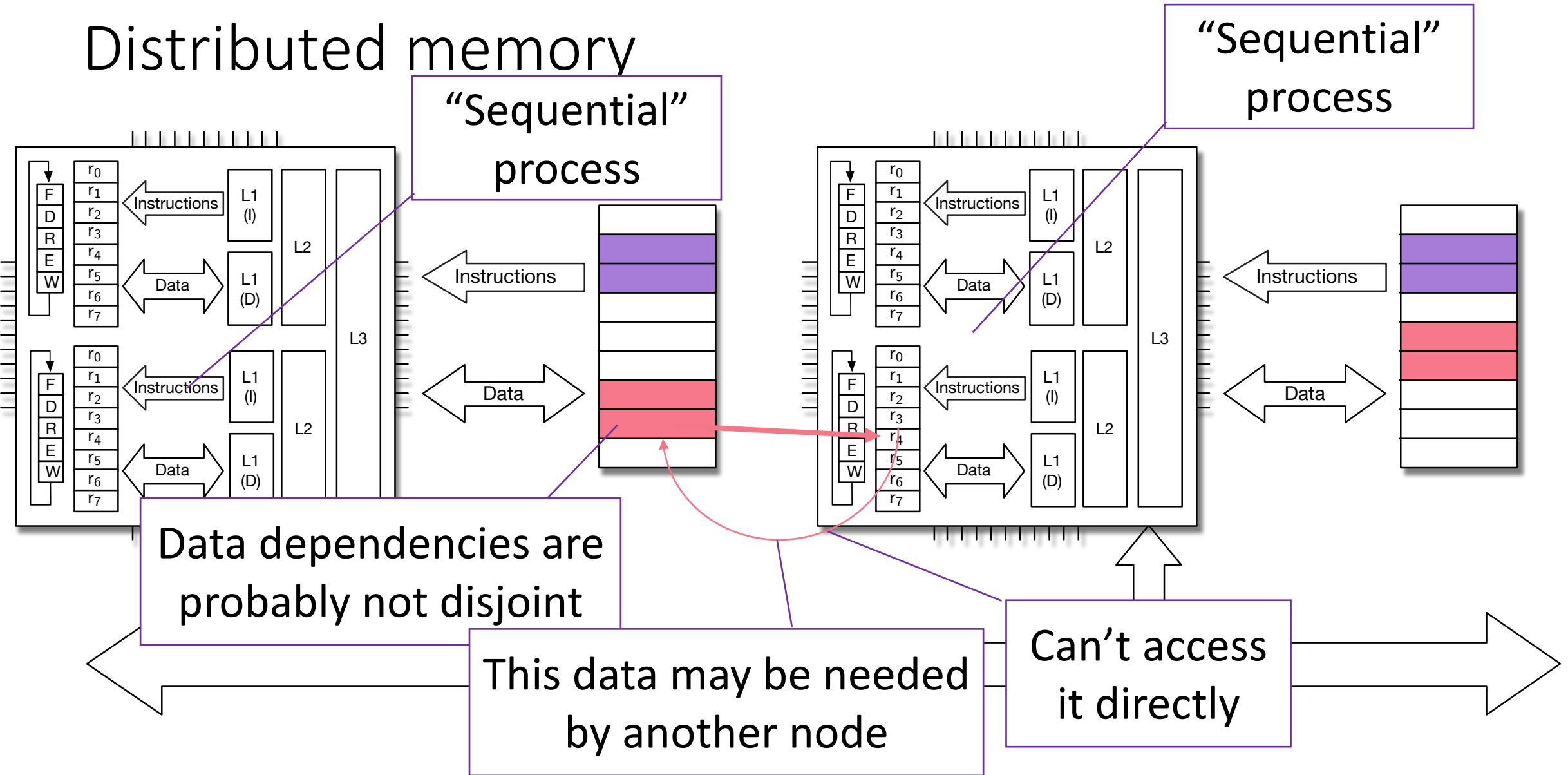


Process

Interconnect

And, back in the day,
a sequential process

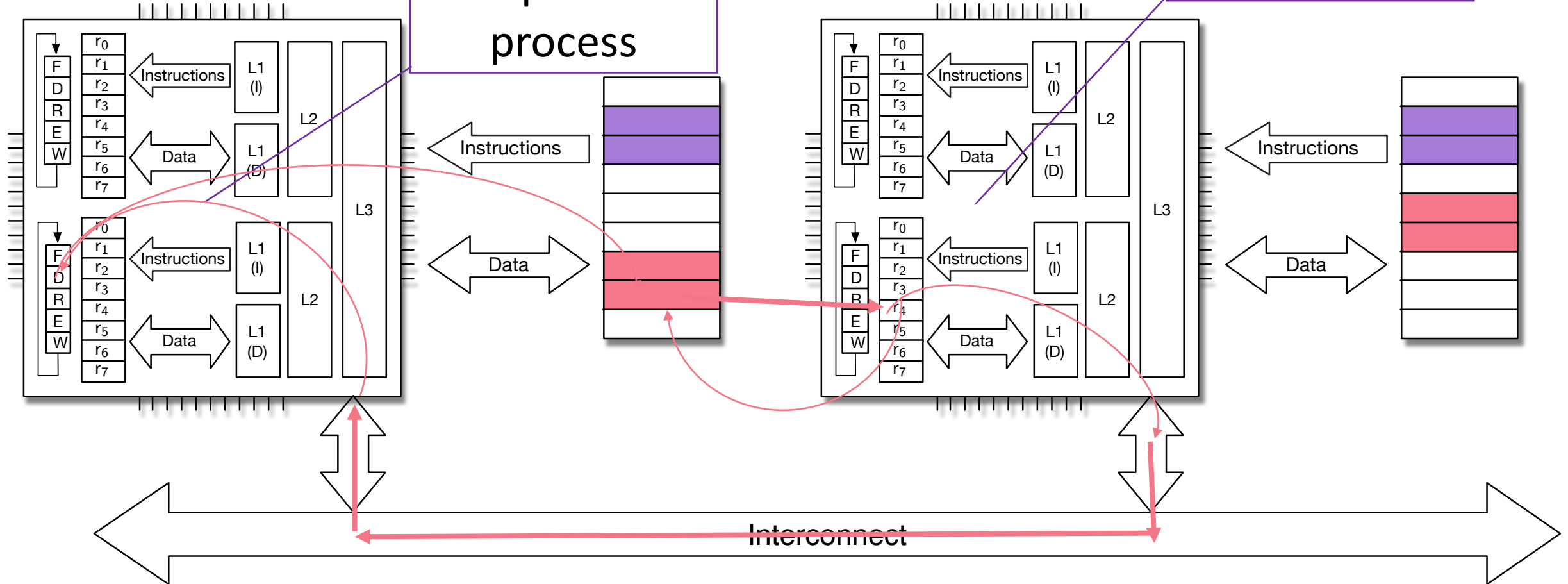
Distributed memory



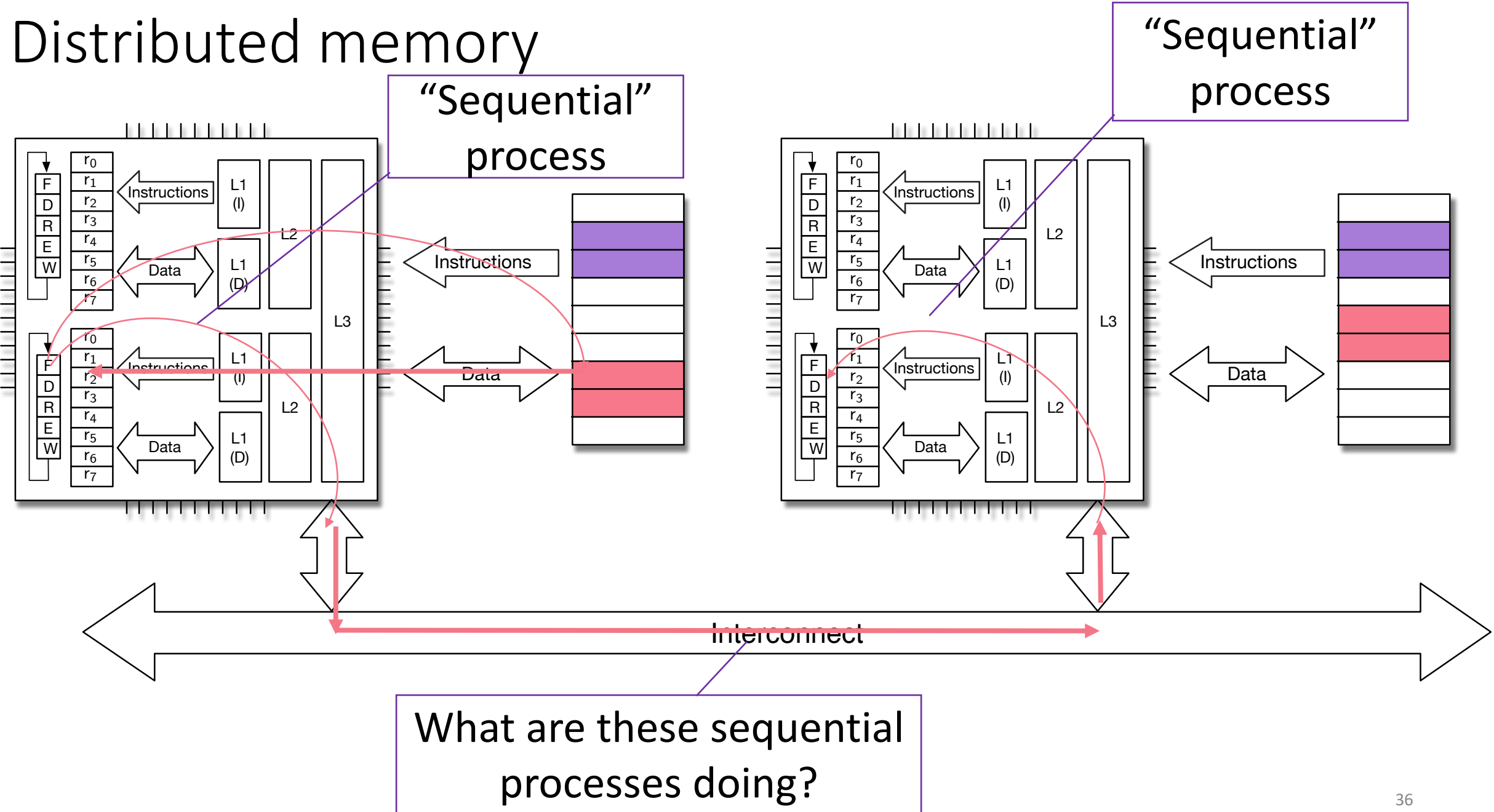
Distributed memory

“Sequential”
process

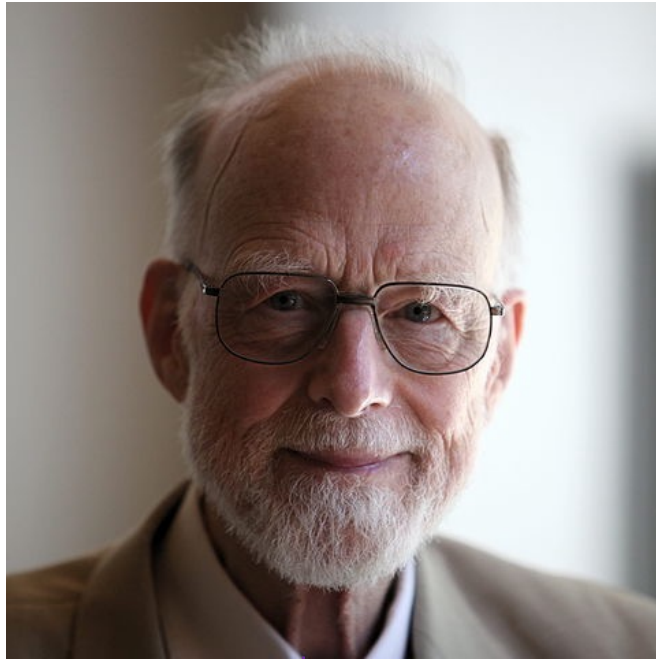
“Sequential”
process



Distributed memory



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 made to formalize the notions of computer programming. The axiomatic approach was first applied in the specification of the ALGOL 60 language and has since been extended to other programming languages. This paper involves the elucidation of some of the basic concepts which can be used in programming. The paper describes 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)

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

Programming
Techniques

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

Communicating Sequential Processes

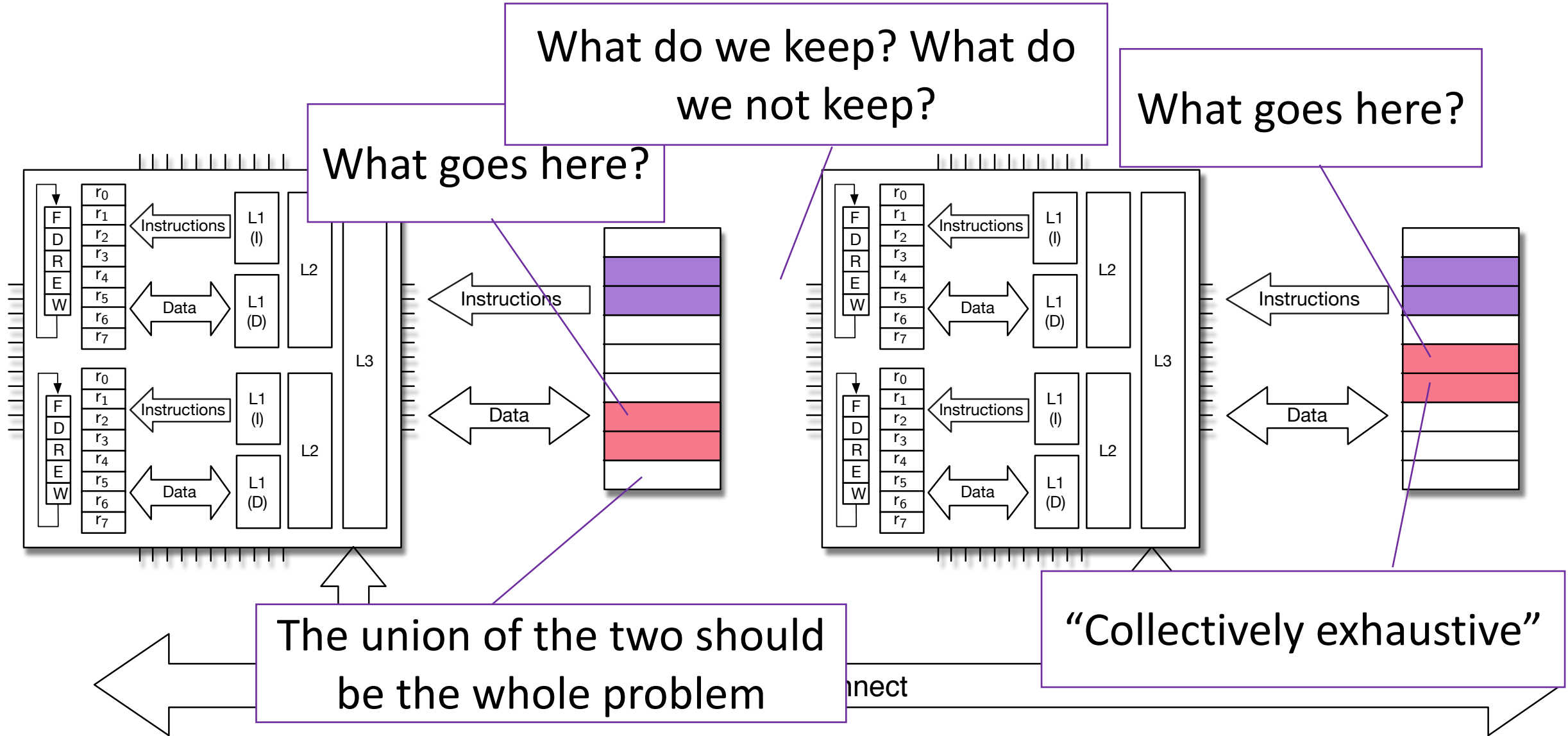
C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

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.

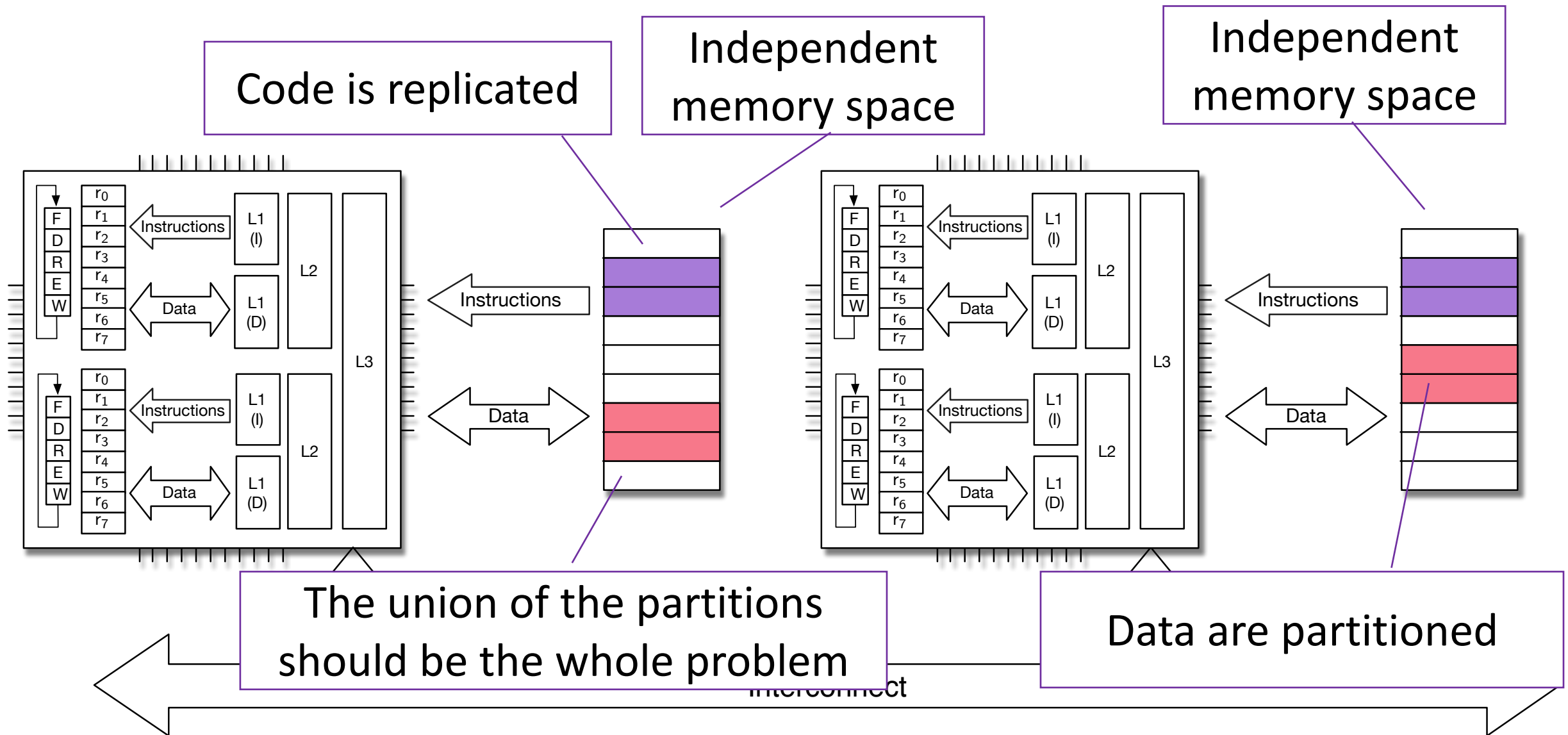
Key Words and Phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical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

CR Categories: 4.20, 4.22, 4.32

Distributed memory



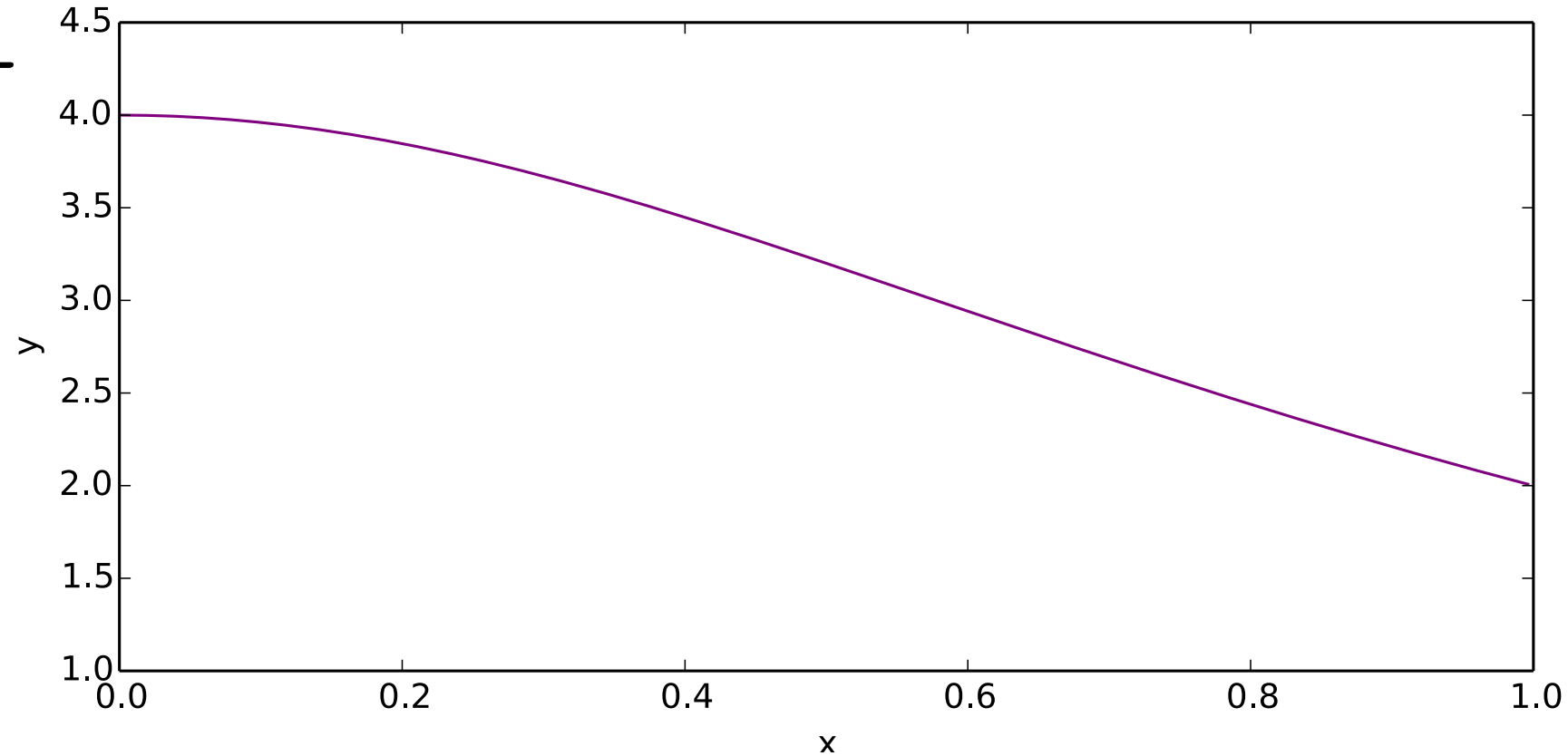
Distributed memory



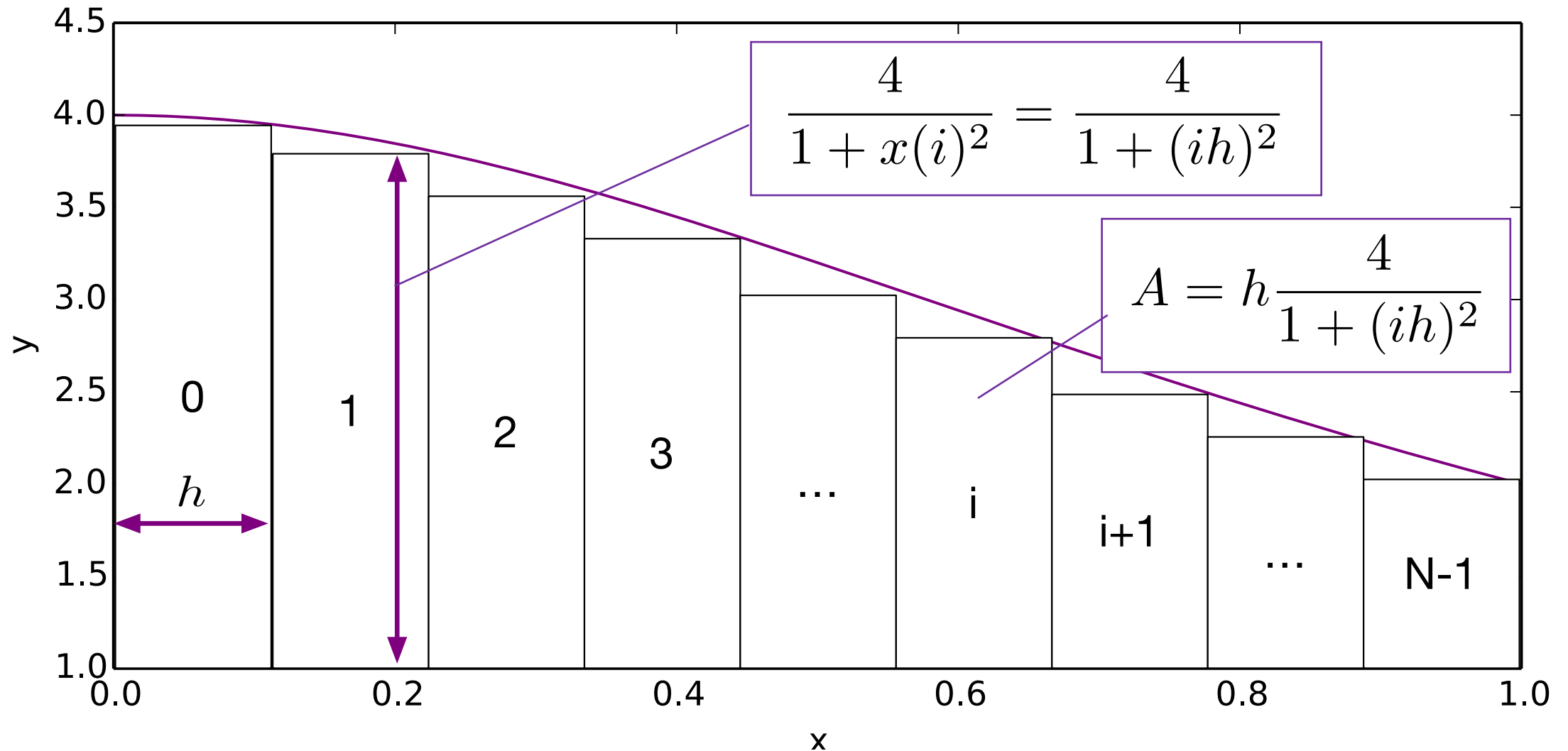
Back to our trusty example (one of them)

- Find the value of π
- Using formula

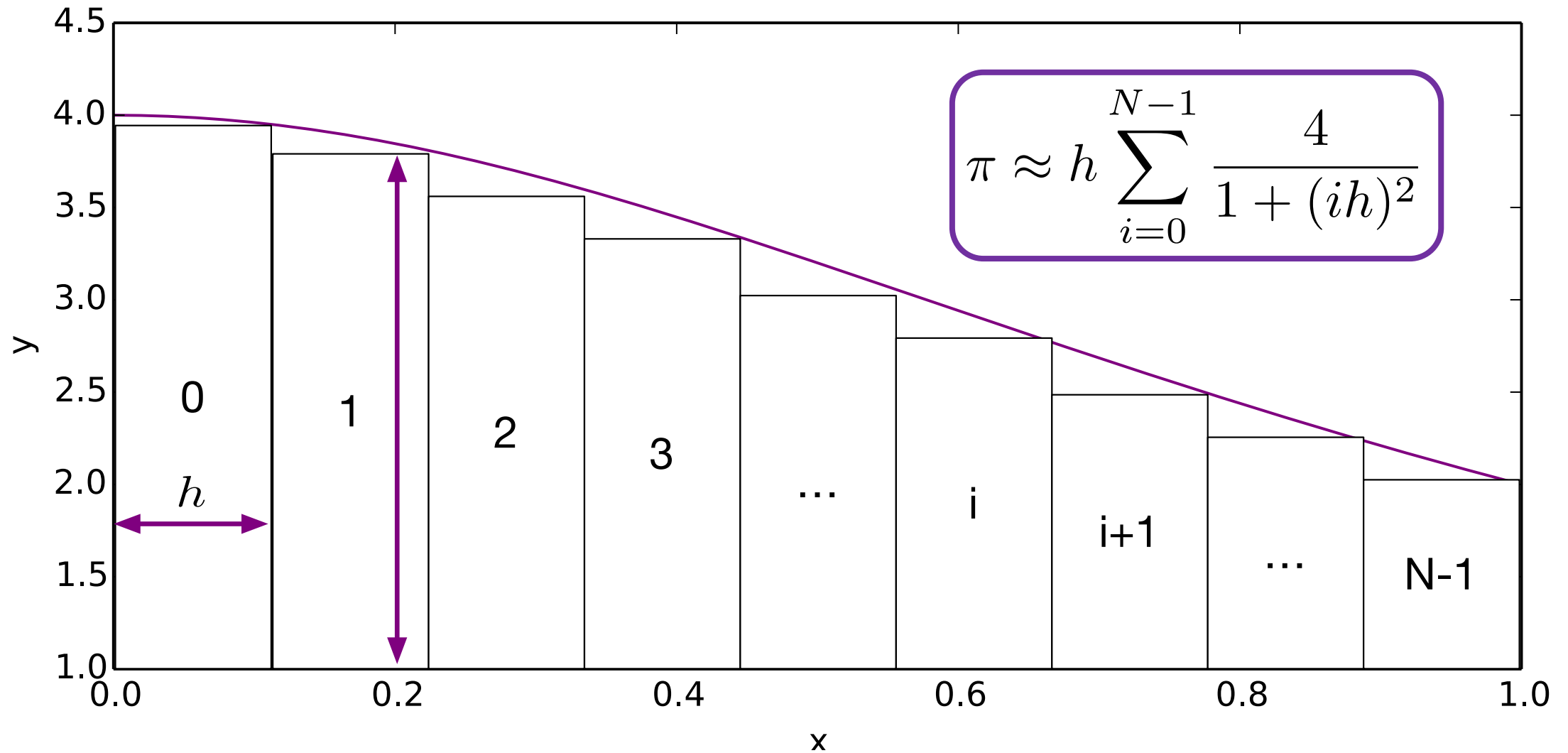
$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$



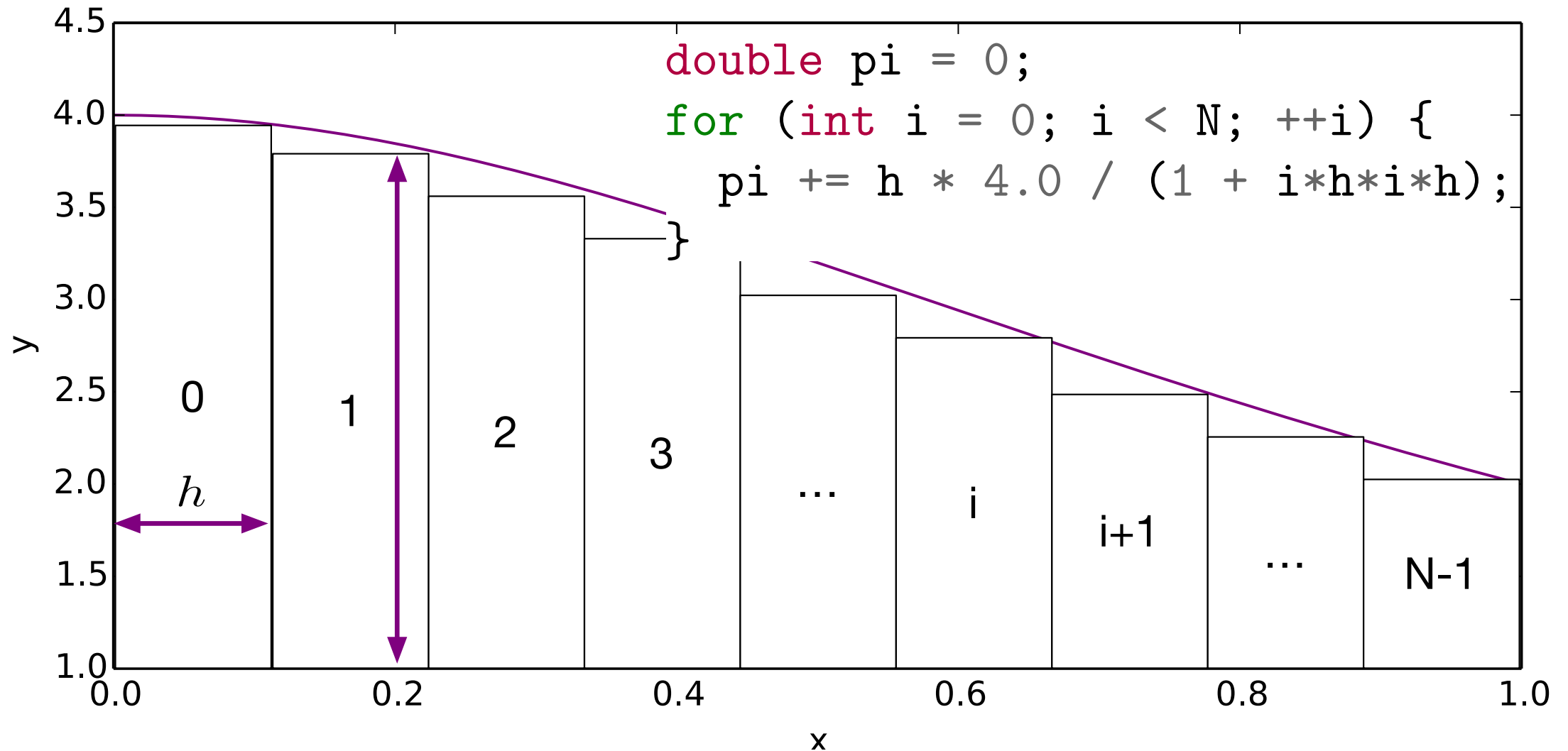
Numerical Quadrature



Numerical Quadrature

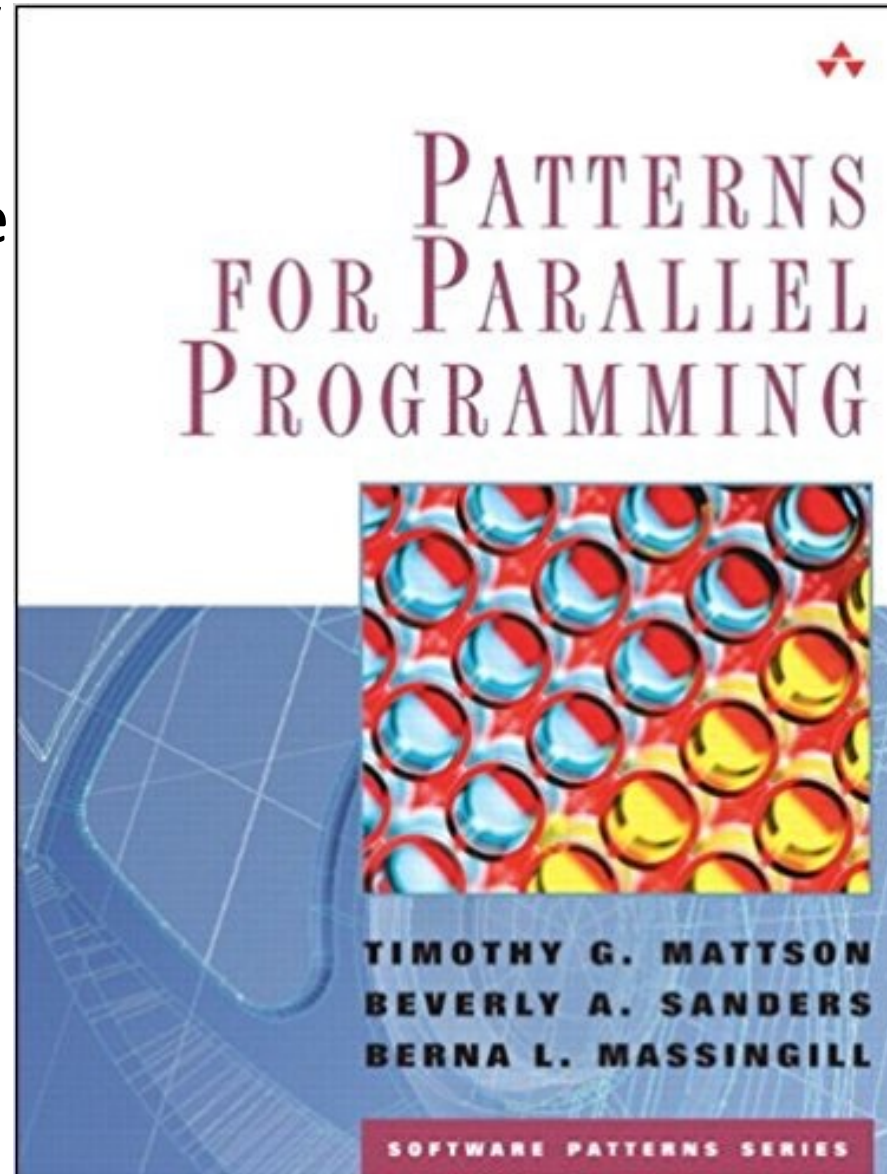


Numerical Quadrature (Sequential)

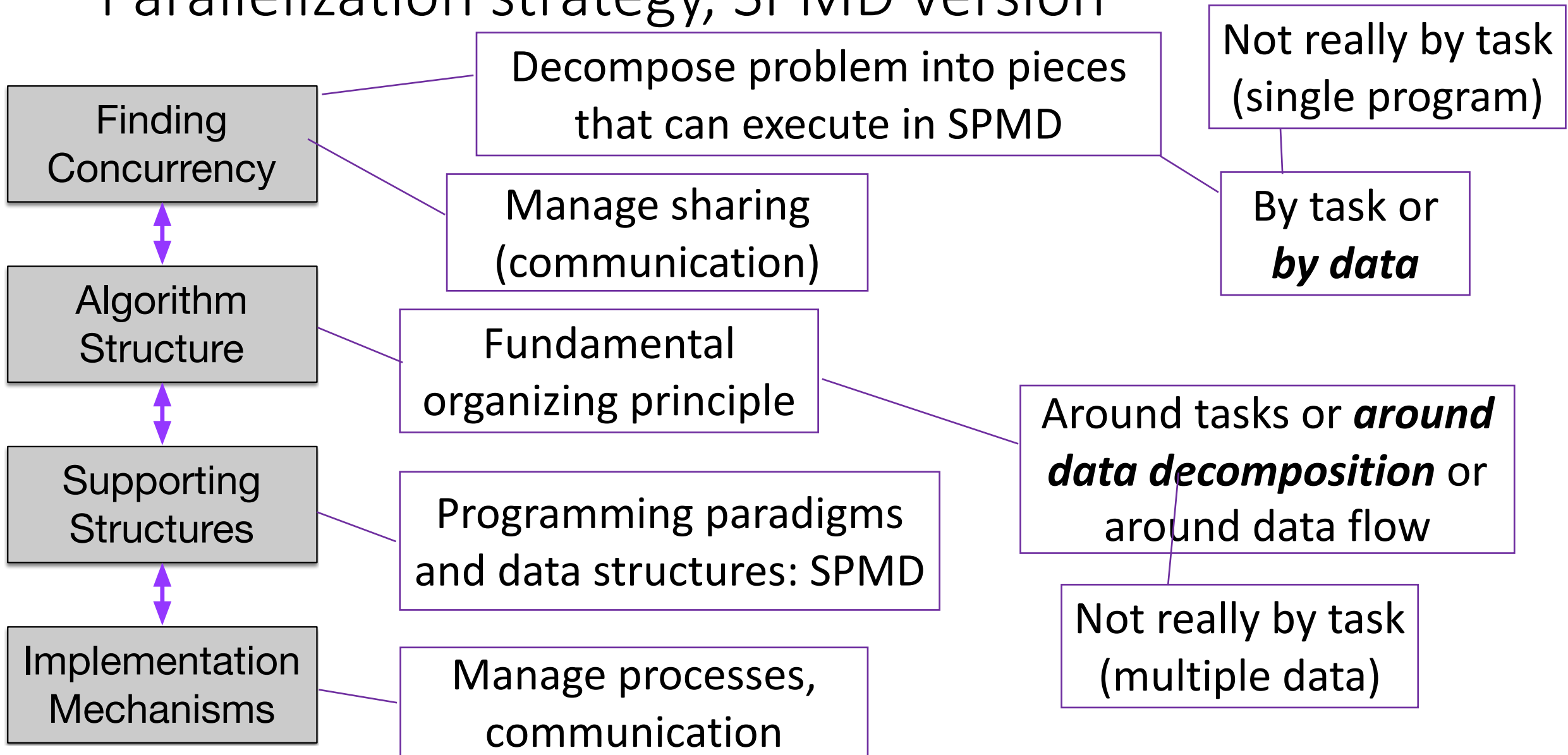


Parallelization Strategy

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

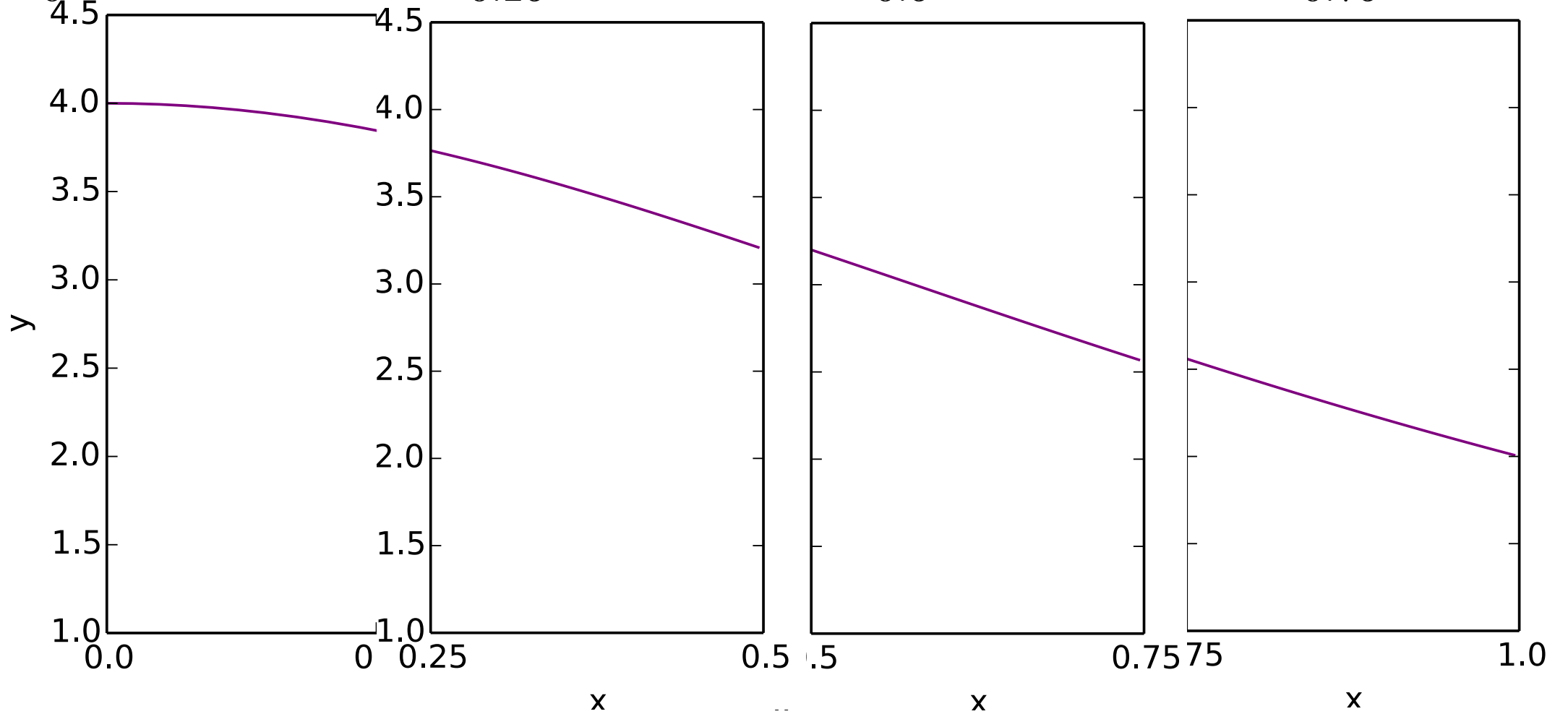


Parallelization strategy, SPMD version



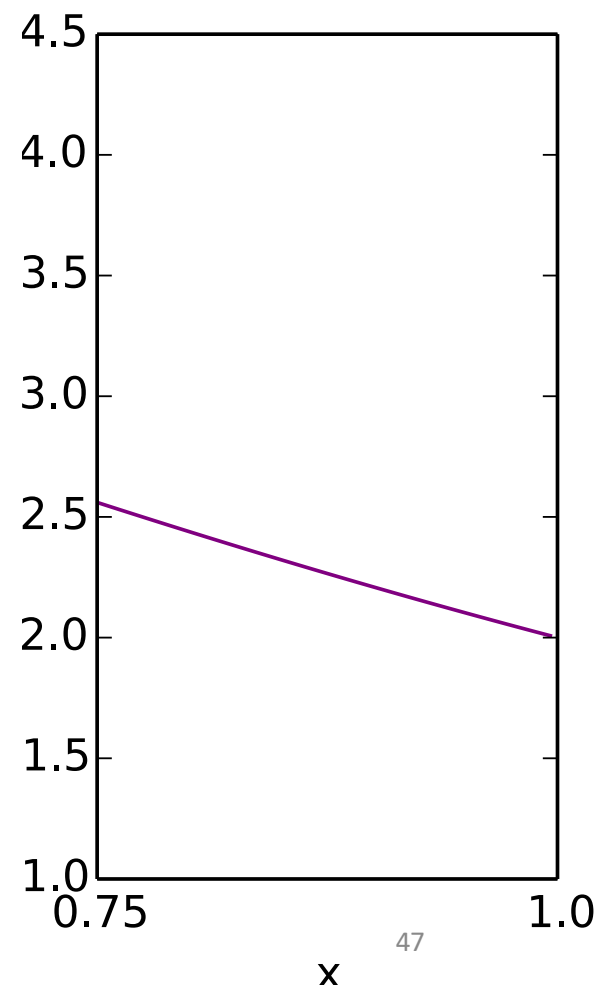
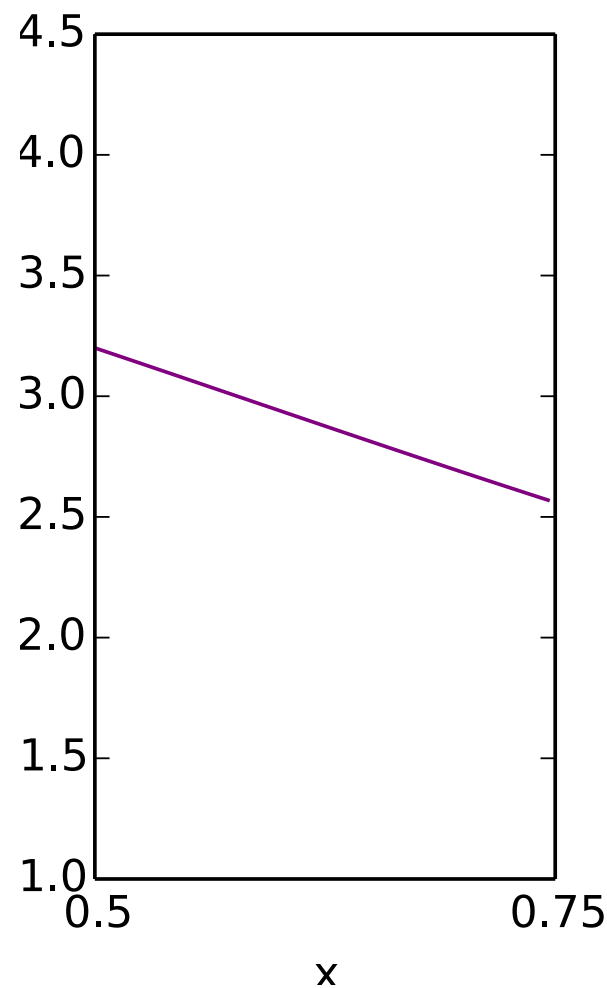
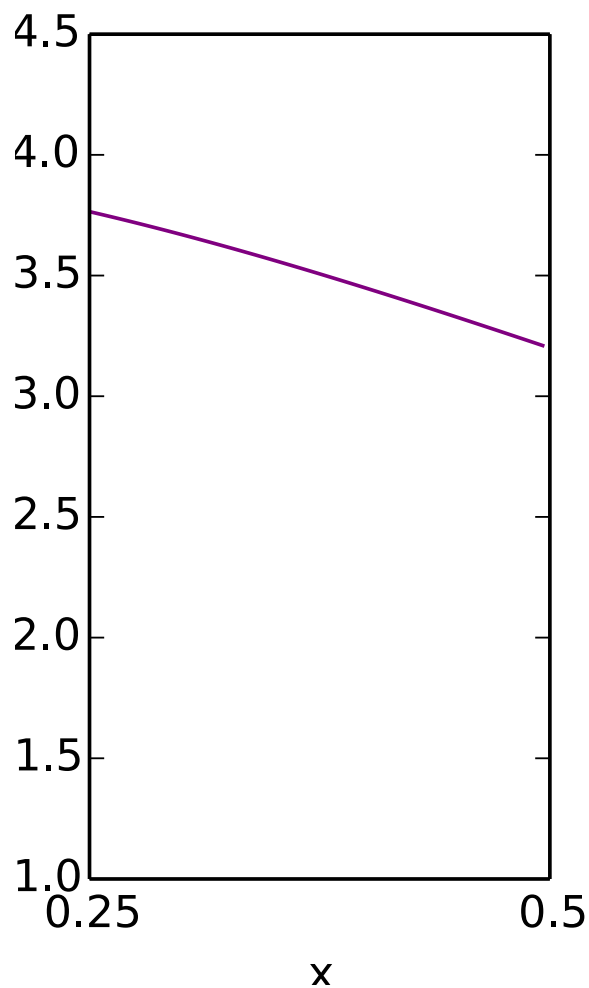
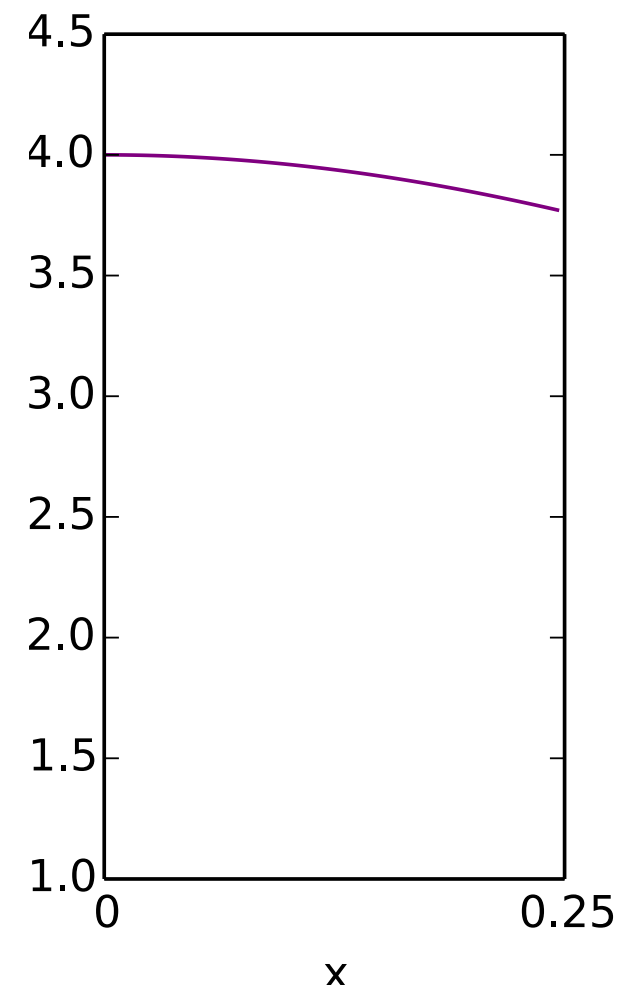
Finding Concurrency

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



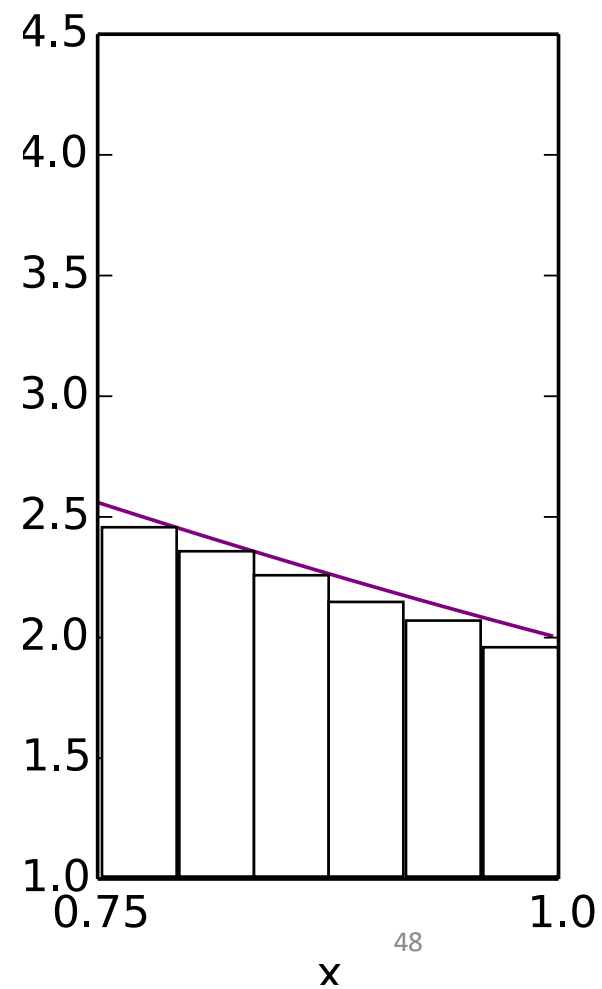
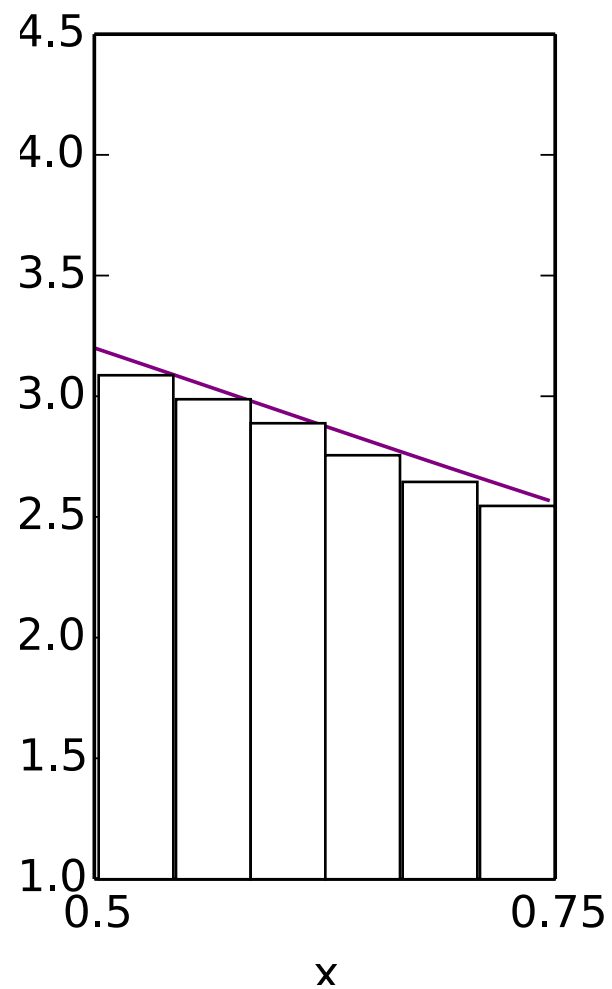
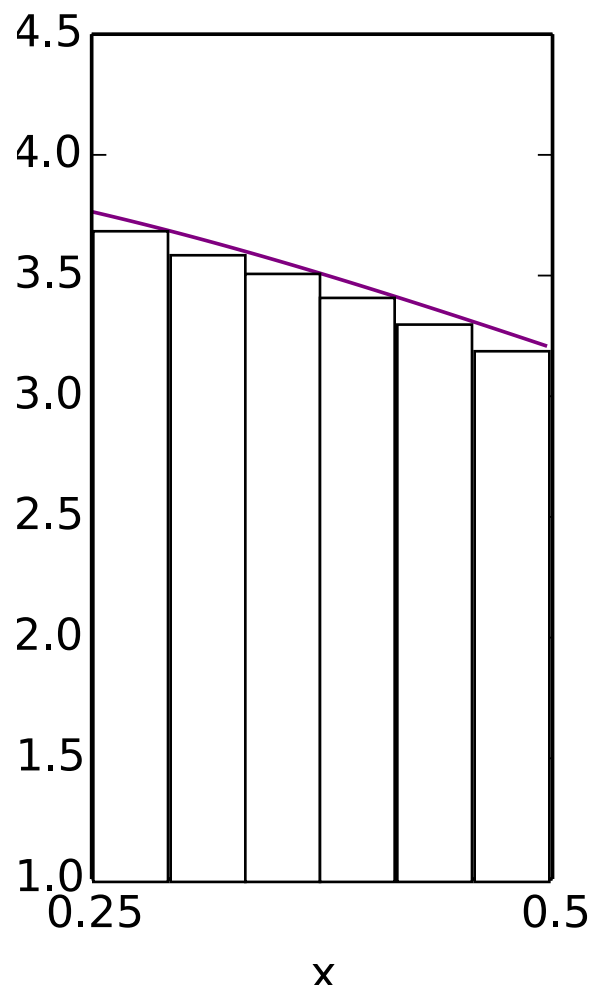
Finding Concurrency

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



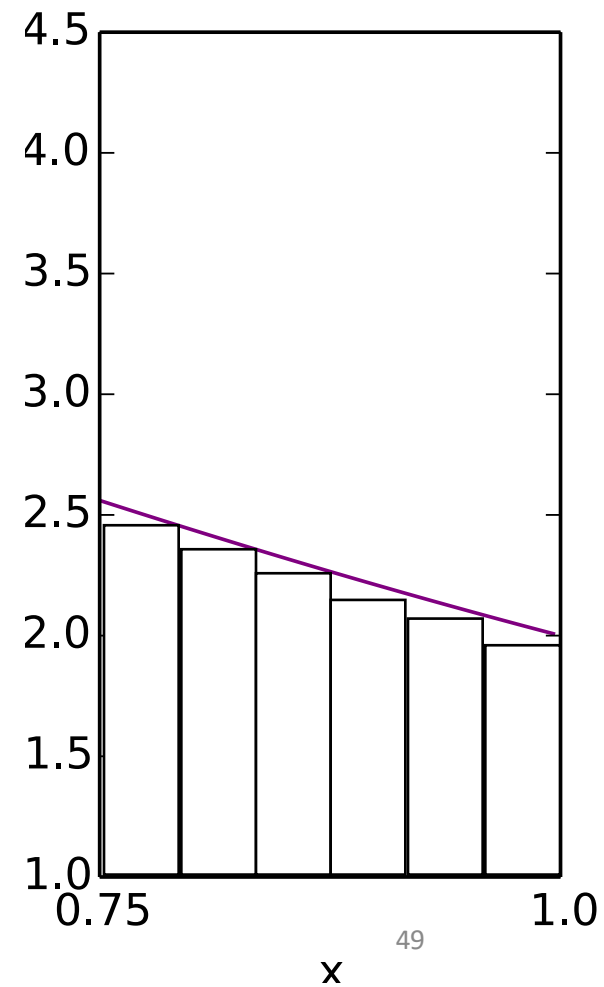
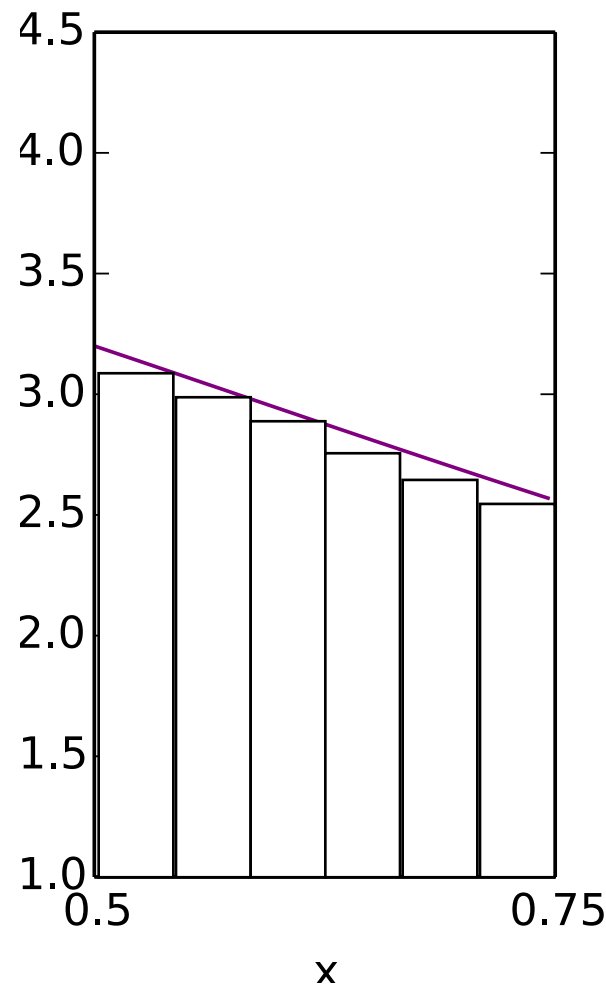
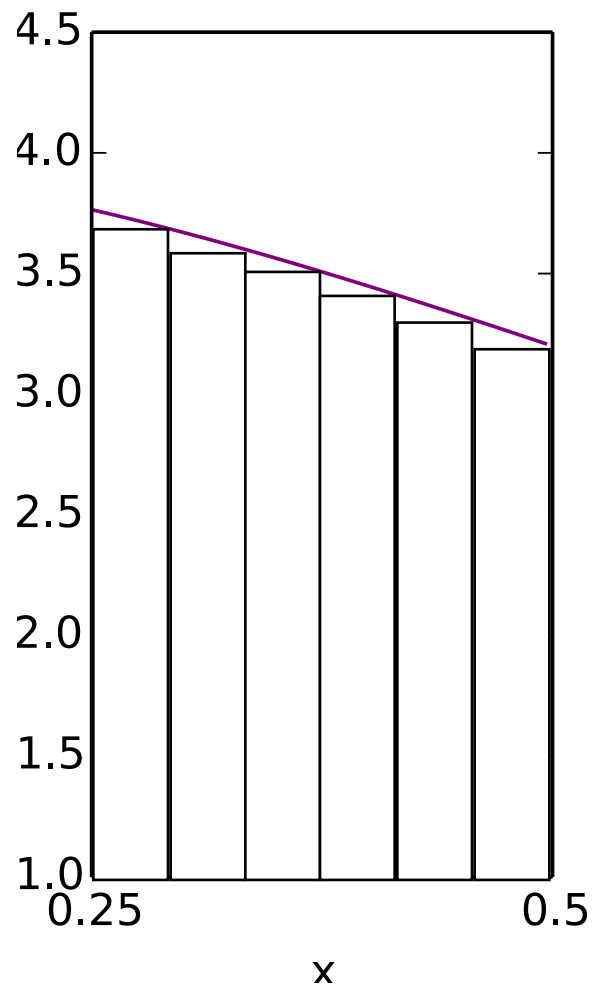
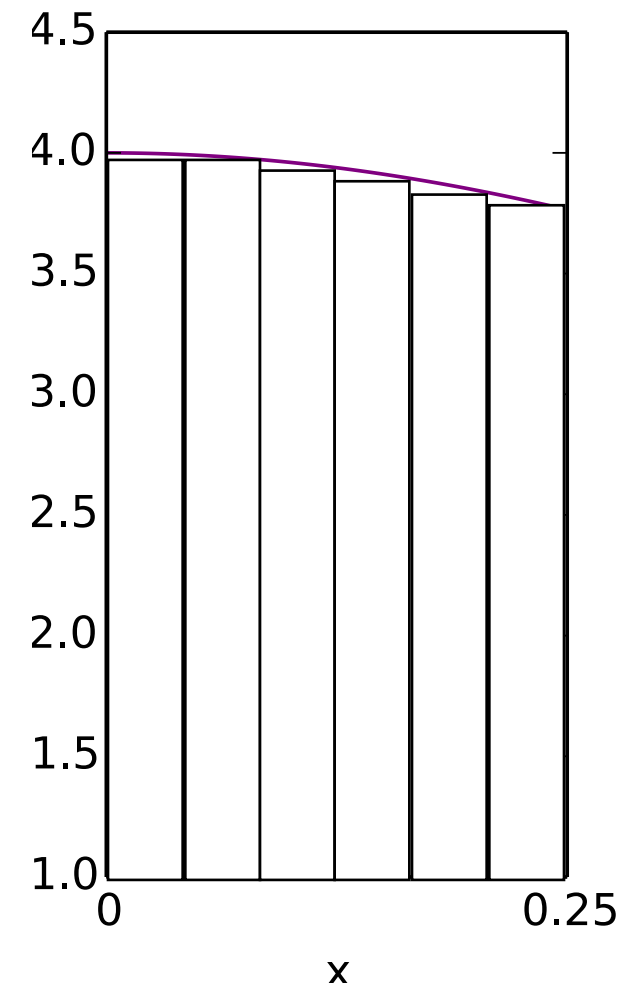
Finding Concurrency

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



Finding Concurrency

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



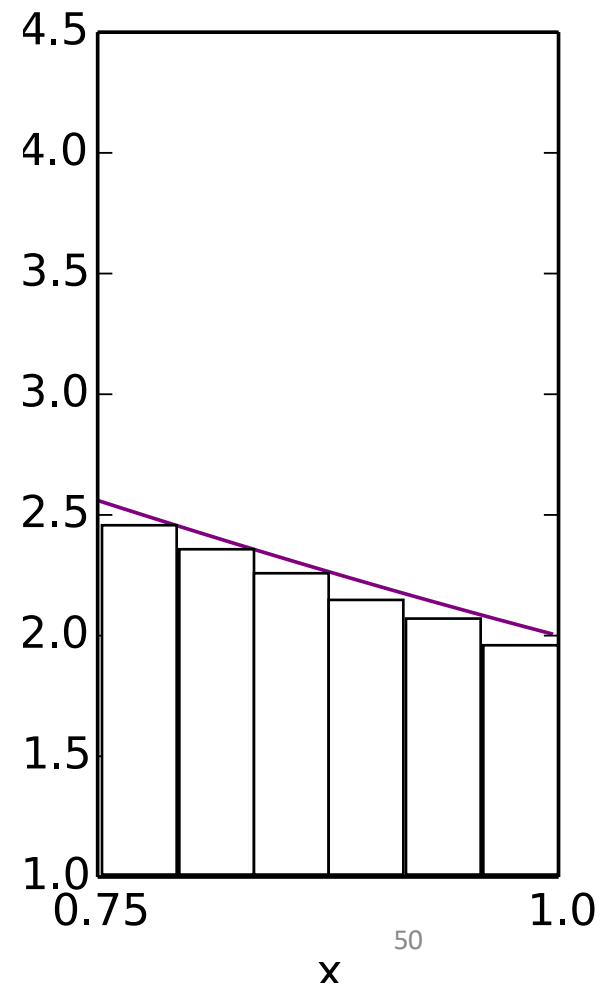
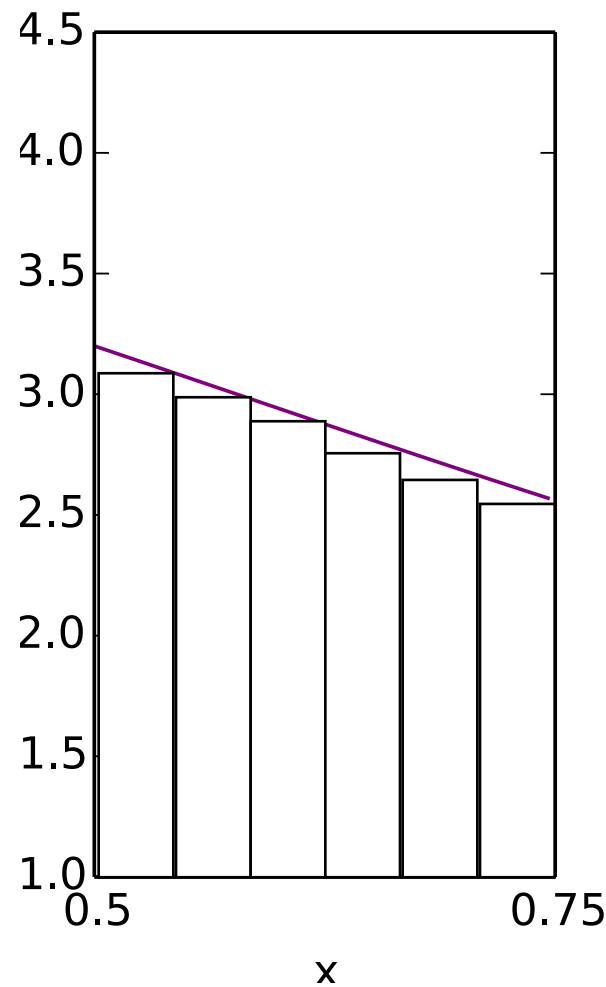
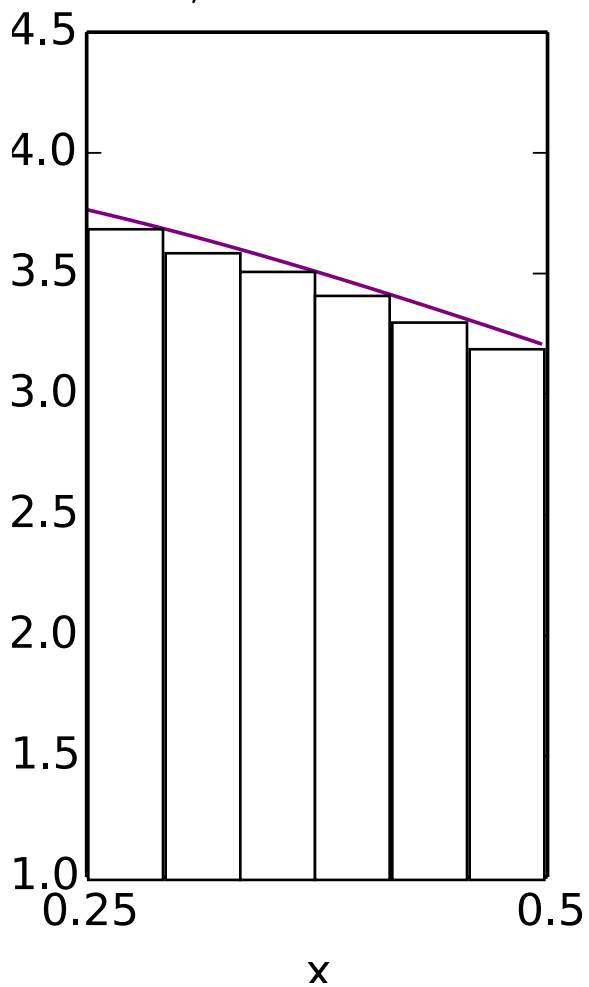
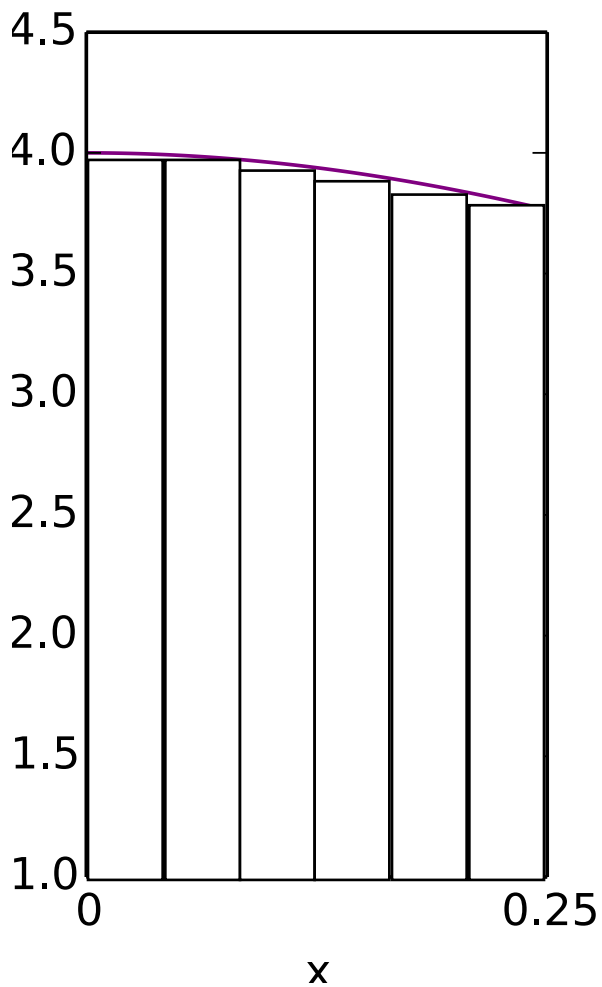
Finding Concurrency

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

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

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

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



Finding Concurrency

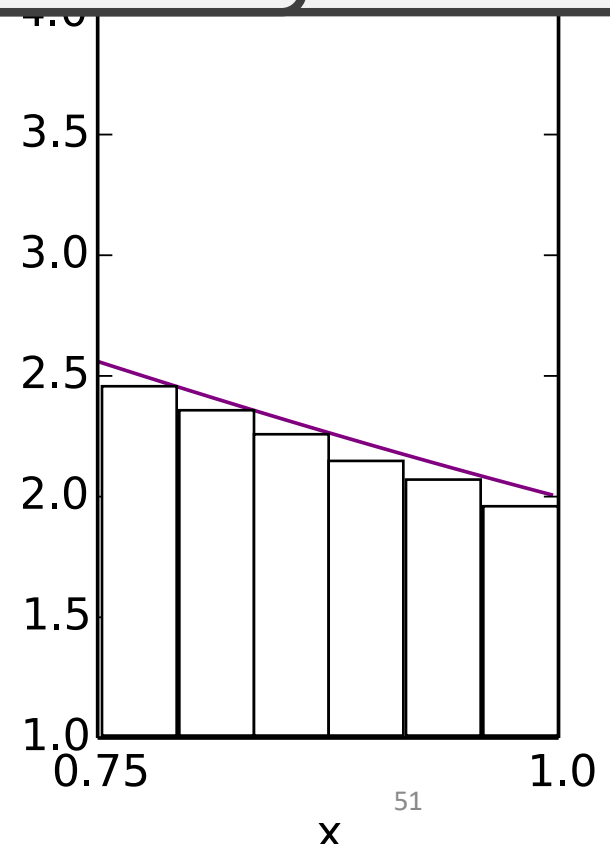
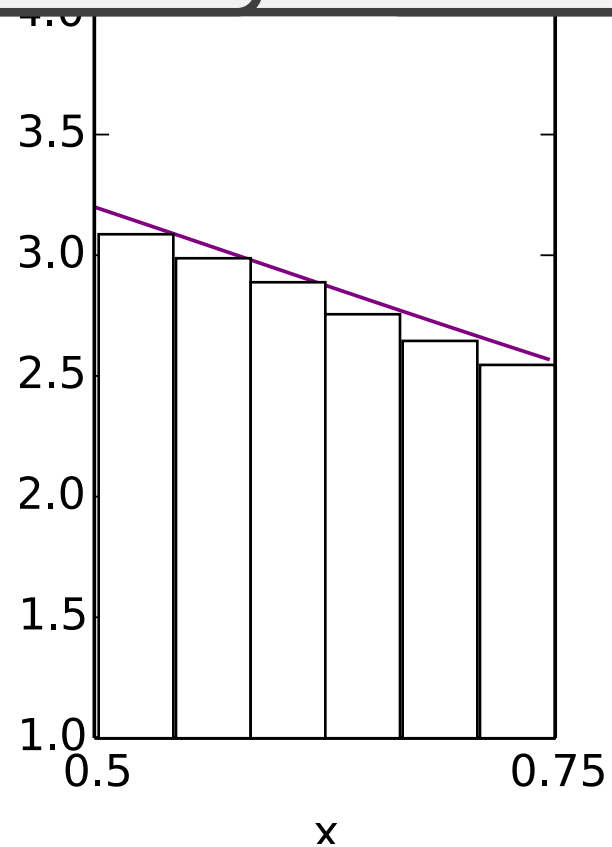
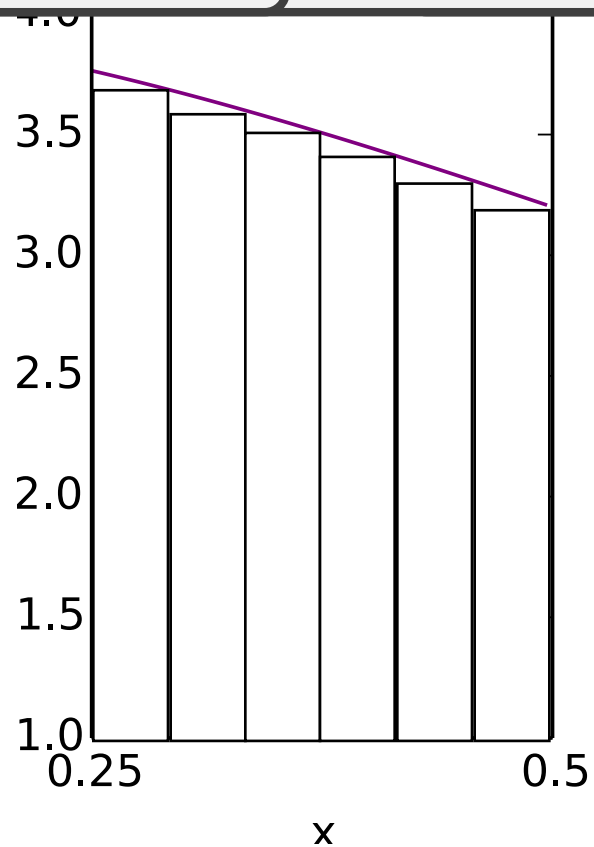
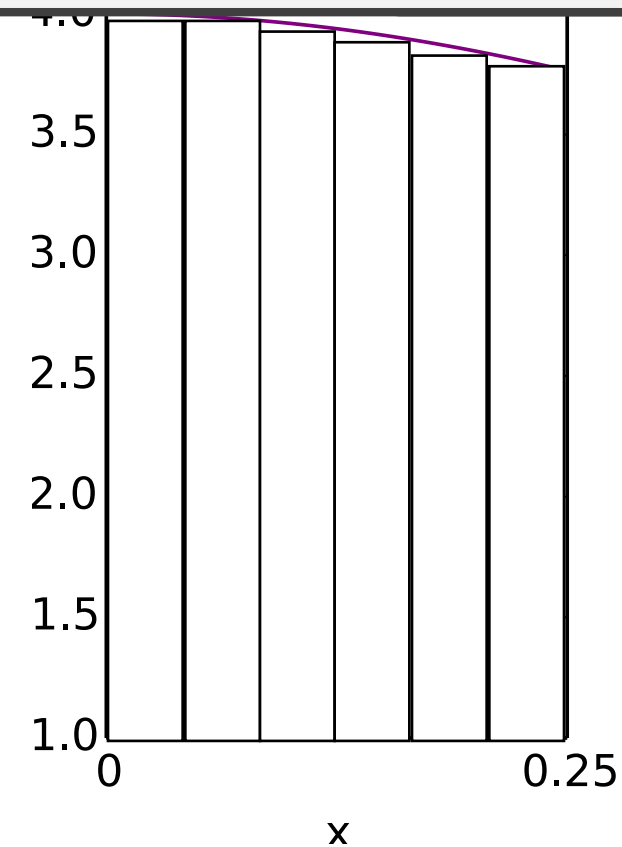
```
for (int i = begin; i < end; ++i) {  
    pi += h * 4.0 / (1 + i*h*i*h);  
}
```

```
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);
```

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

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



Finding Concurrency

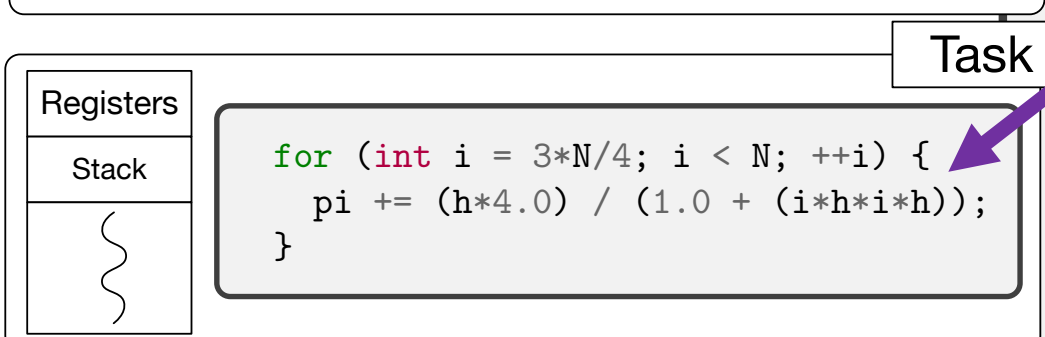
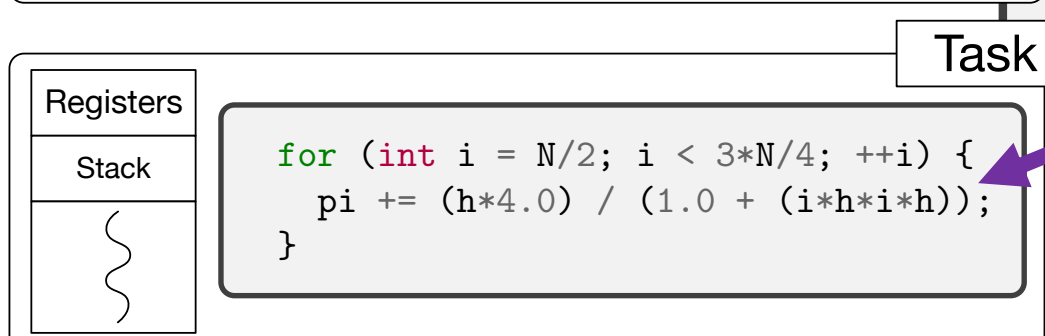
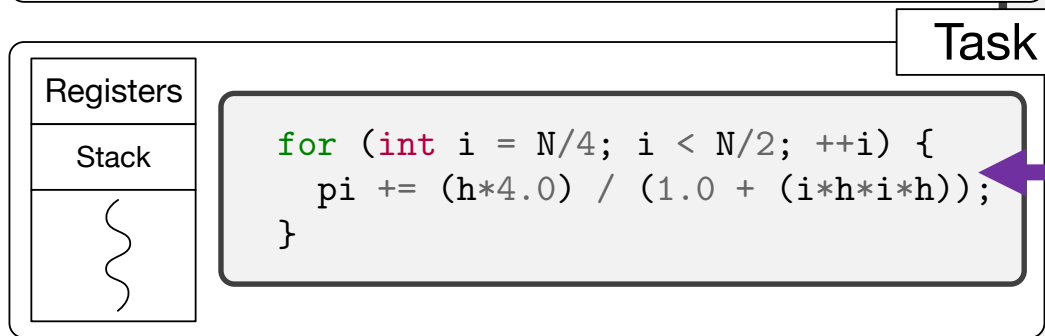
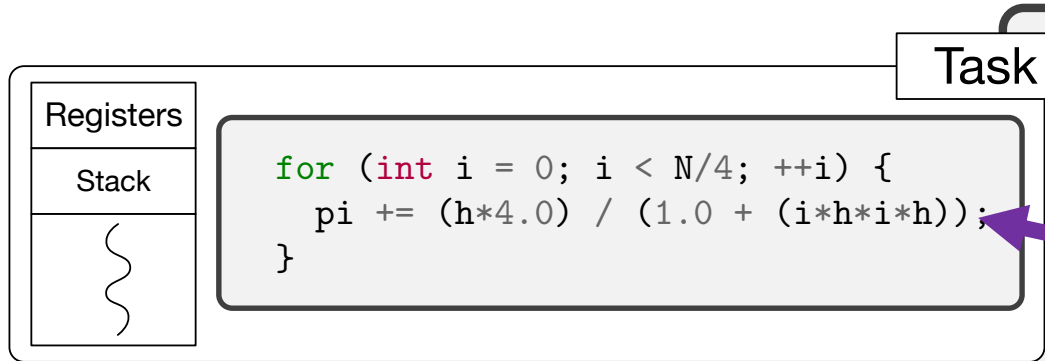
$$h \sum_{i=0}^{N/4-1} \frac{4}{1 + (ih)^2}$$

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

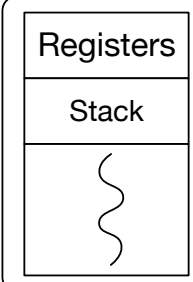
$$h \sum_{i=N/2}^{3N/4-1} \frac{4}{1 + (ih)^2}$$

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

```
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;  
}
```

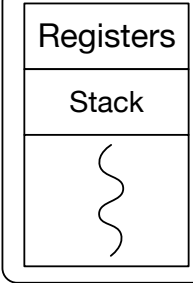


```
int main() {  
    double pi = 0.0;    int N = 1024*1024;  
  
    for (int i = 0; i < N/4; ++i)  
        pi += (h*4.0) / (1.0 + (i*h*i*h));  
  
    for (int i = N/4; i < N/2; ++i)  
        pi += (h*4.0) / (1.0 + (i*h*i*h));  
  
    for (int i = N/2; i < 3*N/4; ++i)  
        pi += (h*4.0) / (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;  
}
```



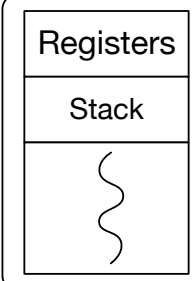
Task

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



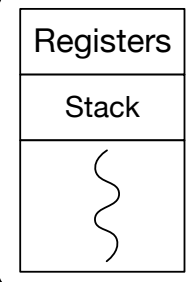
Task

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



Task

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



Task

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

```
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

Registers
Stack
⋮

Task

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

Registers
Stack
⋮

Task

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

Registers
Stack
⋮

Task

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

Registers
Stack
⋮

Task

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

```
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 << std::endl;  
}
```

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 << std::endl;  
}
```

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 << std::endl;  
}
```

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;  
}
```

Process

```
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

```
#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;
}
```

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>

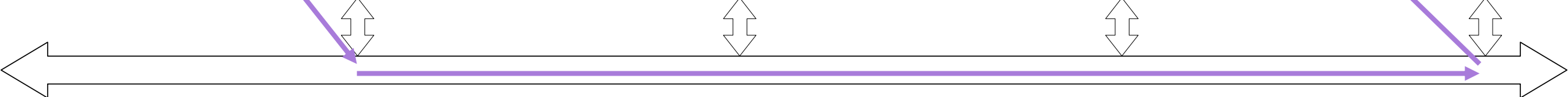
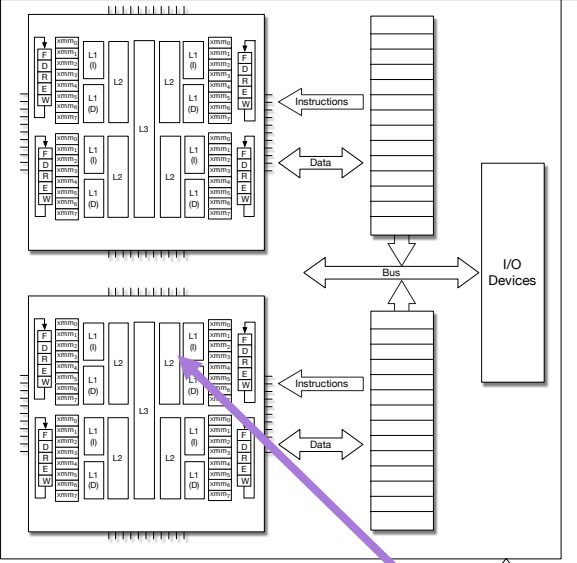
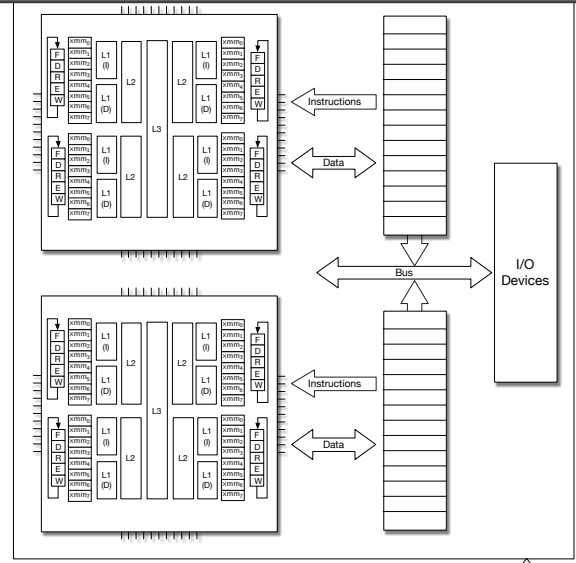
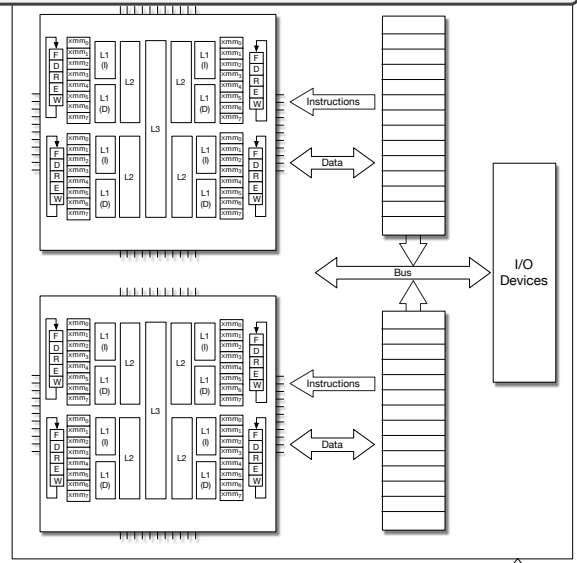
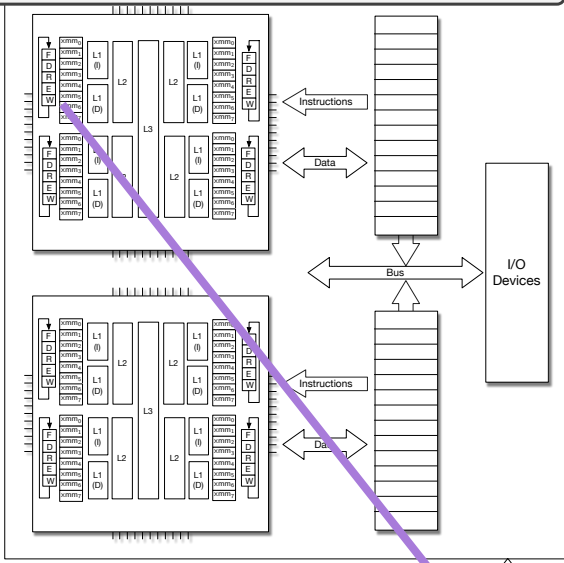
int main() {
    double pi = 0.0; double h = 1./((double) N);
    for (size_t i = N/2; i < 3*N/4; ++i)
        pi += (h * 4.0) / (1.0 + (i * h * i * h));
    std::cout << "pi is ~ " << pi << std::endl;

    return 0;
}
```

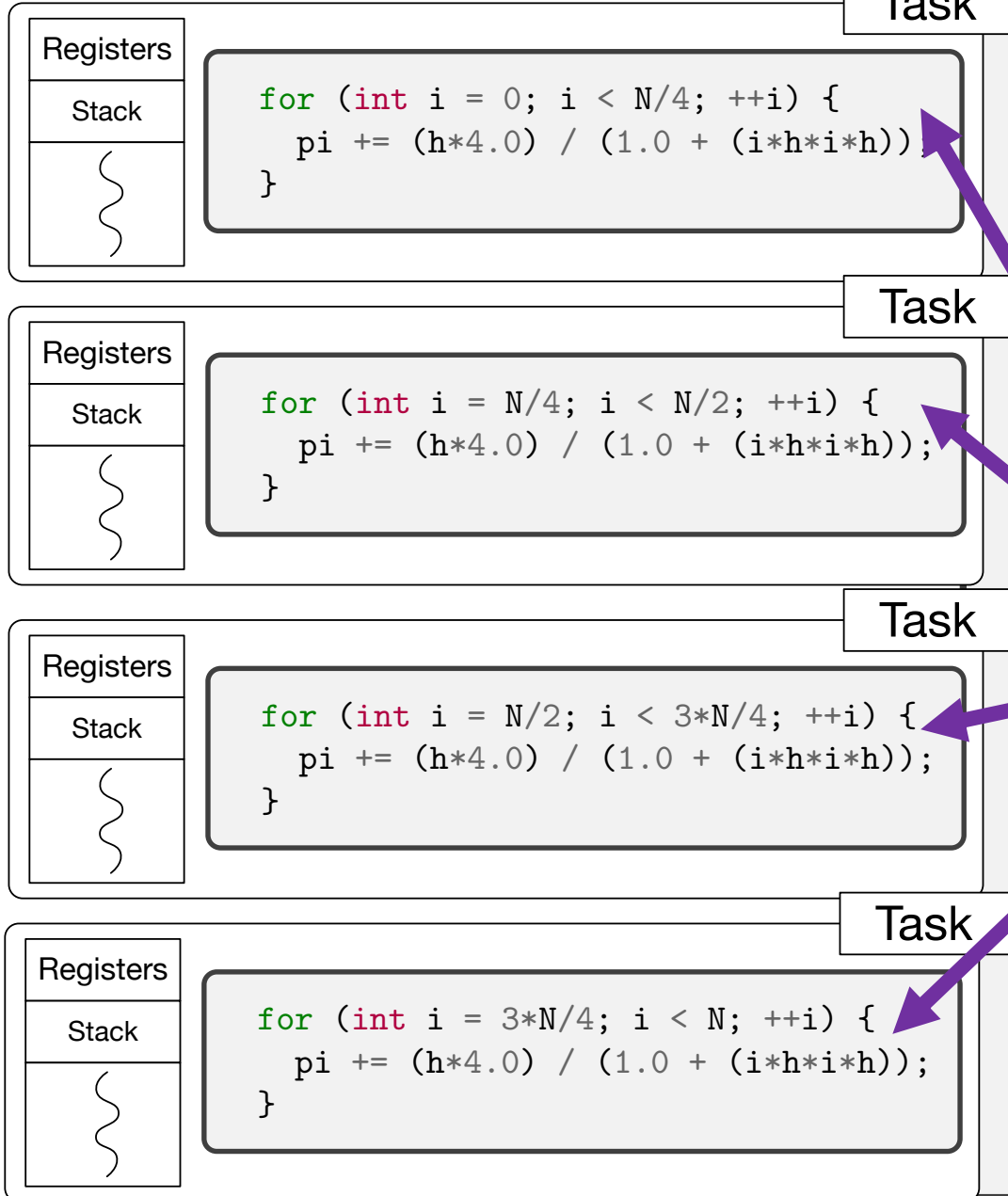
```
#include <iostream>

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;
}
```

Very Important Slide!!

Process

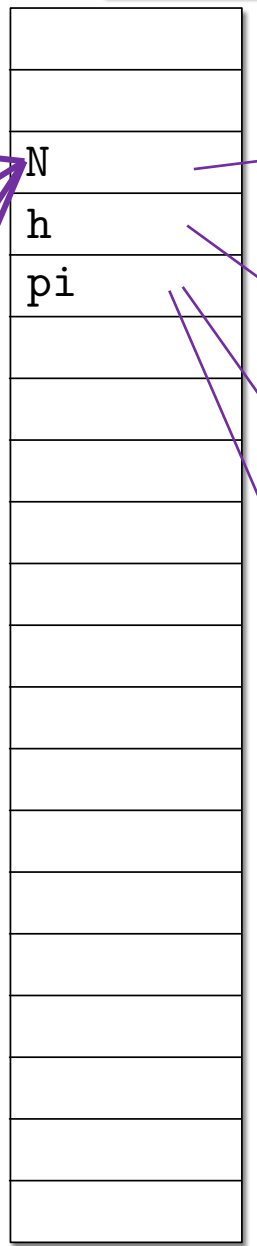


```
Task
for (int i = 0; i < N/4; ++i) {
  pi += (h*4.0) / (1.0 + (i*h*i*h));
}
Task
```

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

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

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



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 *the same* N

Similarly h

Similarly pi

At least for reading

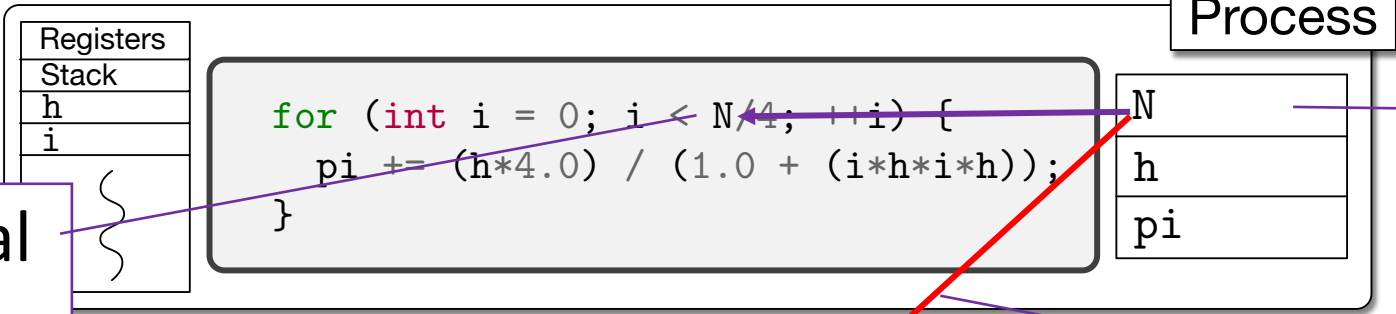
(Have to deal with race when writing)

$\{P_1\}S_1\{Q_1\}, \{P_2\}S_2\{Q_2\}$ interference free

$\{P_1\} \wedge \{P_2\}$ cobegin $S_1||S_2$ coend $\{Q_1 \wedge Q_2\}$

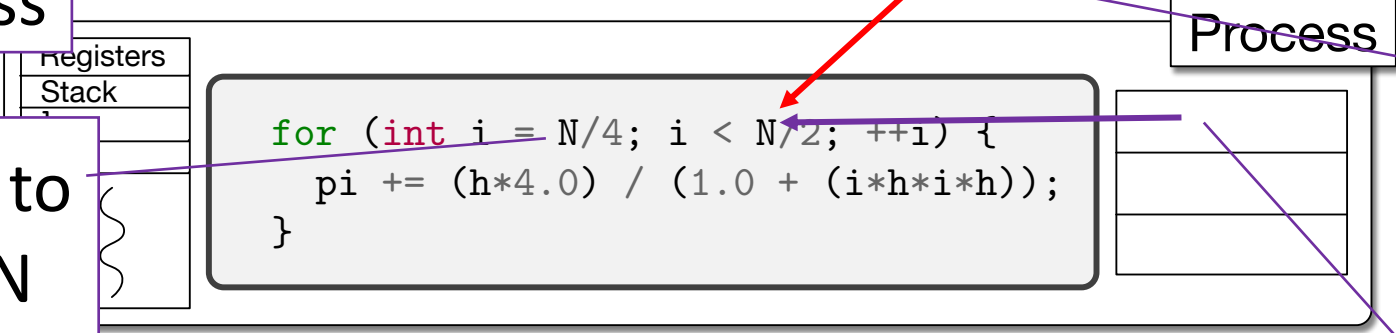
Distributed

This N is local to this process



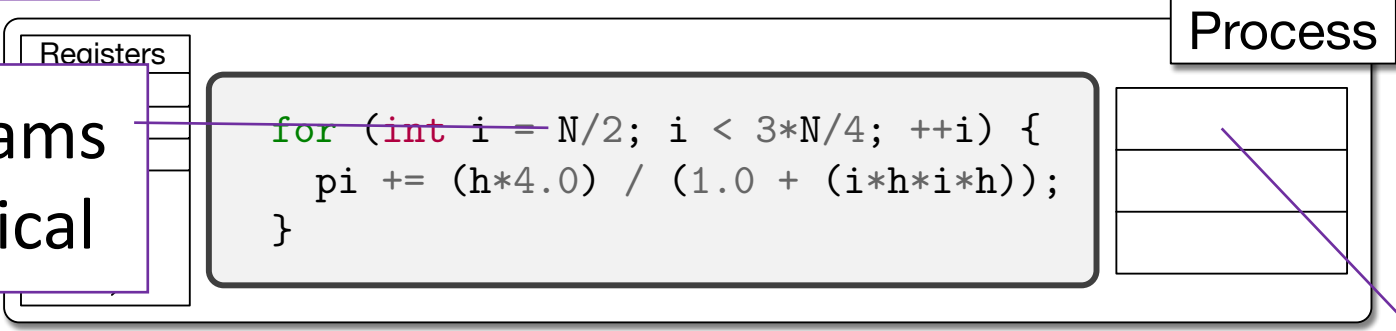
This N is local to this process

But we need to read *some* N



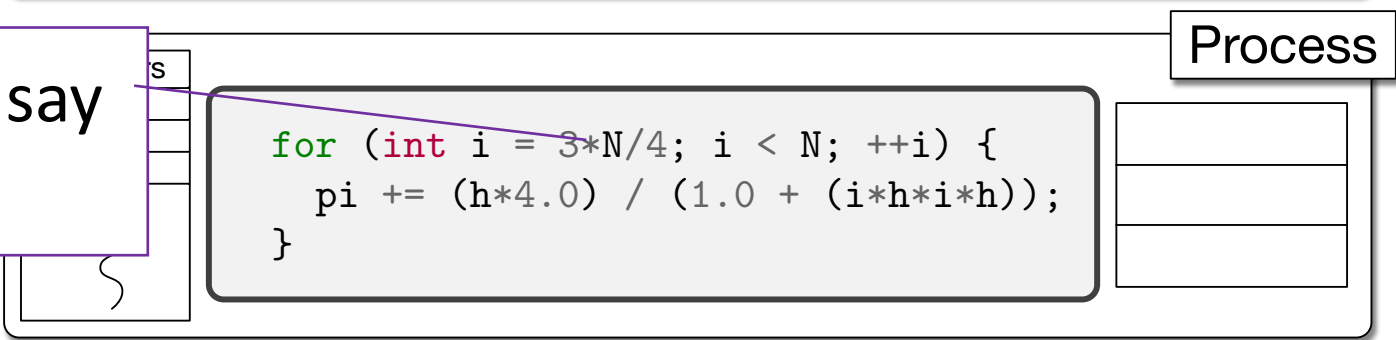
Can *not* read from another process memory

These programs are all identical



(In some sense there is an N here)

And they all say "read N"



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

Distributed

To read the "right" N

Registers		<pre>for (int i = 0; i < N/4; ++i) { pi += (h*4.0) / (1.0 + (i*h*i*h)); }</pre>	Process
Stack			N
h			h
i			pi

Registers		<pre>for (int i = N/4; i < N/2; ++i) { pi += (h*4.0) / (1.0 + (i*h*i*h)); }</pre>	Process
Stack			N
h			
i			

Registers		<pre>for (int i = N/2; i < 3*N/4; ++i) { pi += (h*4.0) / (1.0 + (i*h*i*h)); }</pre>	Process
Stack			N
h			
i			

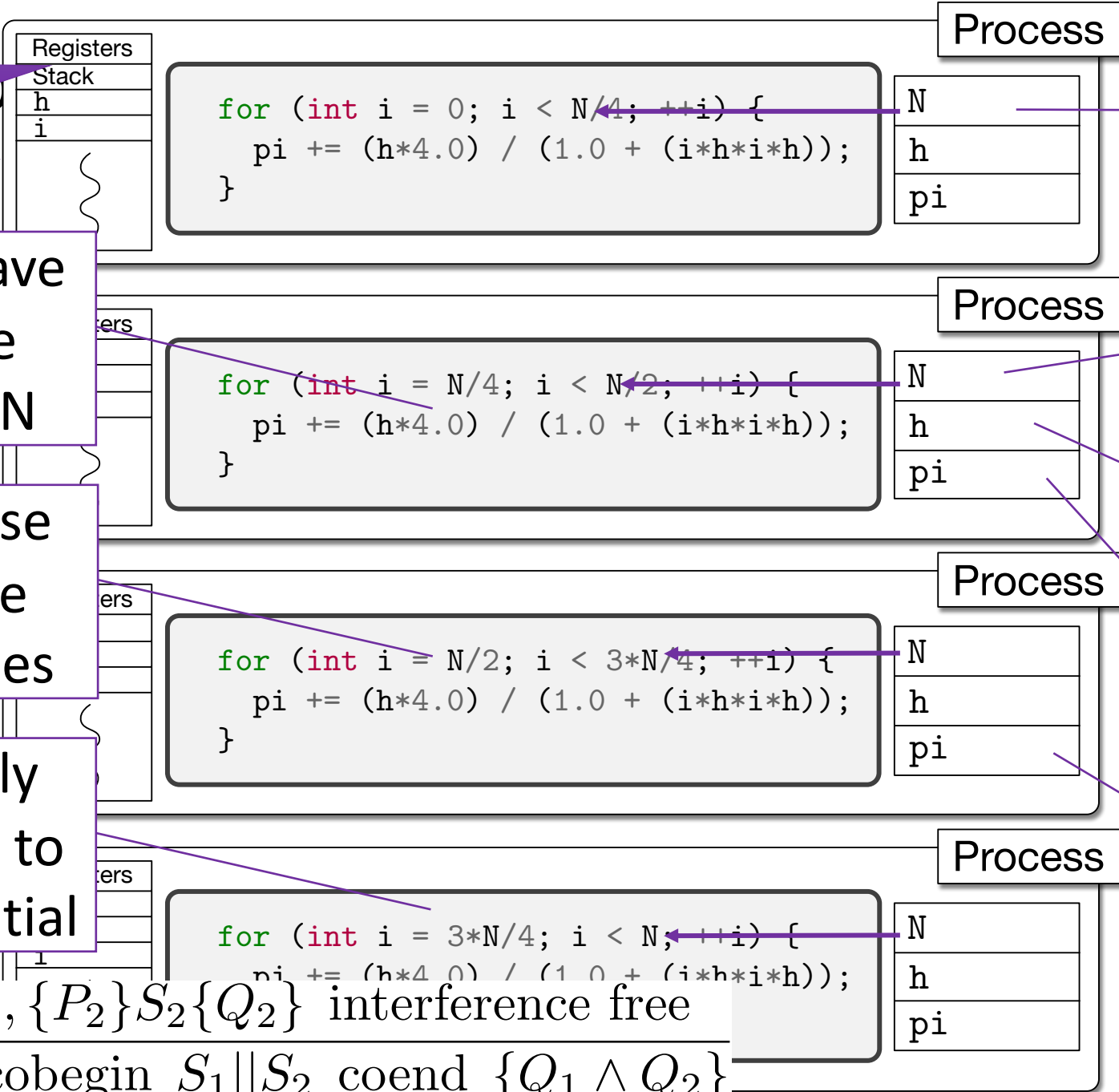
Registers		<pre>for (int i = 3*N/4; i < N; ++i) { pi += (h*4.0) / (1.0 + (i*h*i*h)); }</pre>	Process
Stack			N
h			
i			

Copy to each process

Copy to each process

Copy to each process

Very Important Slide!!



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

Must make consistent when writing to maintain *as if*

SPMD? Single program multiple data?

Multiple data
(different limits)

```
#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 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 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;  
}
```

```
#include <iostream>
```

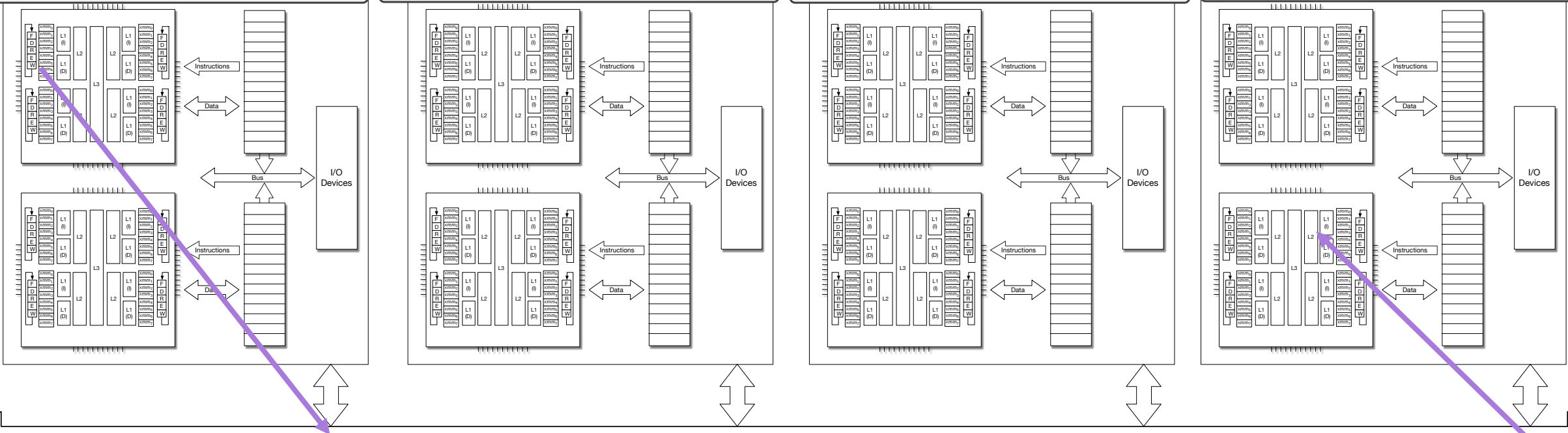
```
int main() {  
    double pi = 0.0; double h = 1./((double) N);  
    for (size_t i = 0; i < N/4; ++i)  
        pi += (h * 4.0) / (1.0 + (i * h * i * h));  
    std::cout << "pi is ~ " << pi << std::endl  
  
    return 0;  
}
```

```
#include <iostream>
```

```
int main() {  
    double pi = 0.0; double h = 1./((double) N);  
    for (size_t i = 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 = 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)

How do we do these two things?

```
#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?

We need 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 the same program

Each program computes same thing

But a different begin and end everywhere

Single Program Multiple Data

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

```
int main(size_t argc, char* argv[]) {  
    size_t N = atoi(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;  
}
```

Single Program Multiple Data

This node reads from the command line

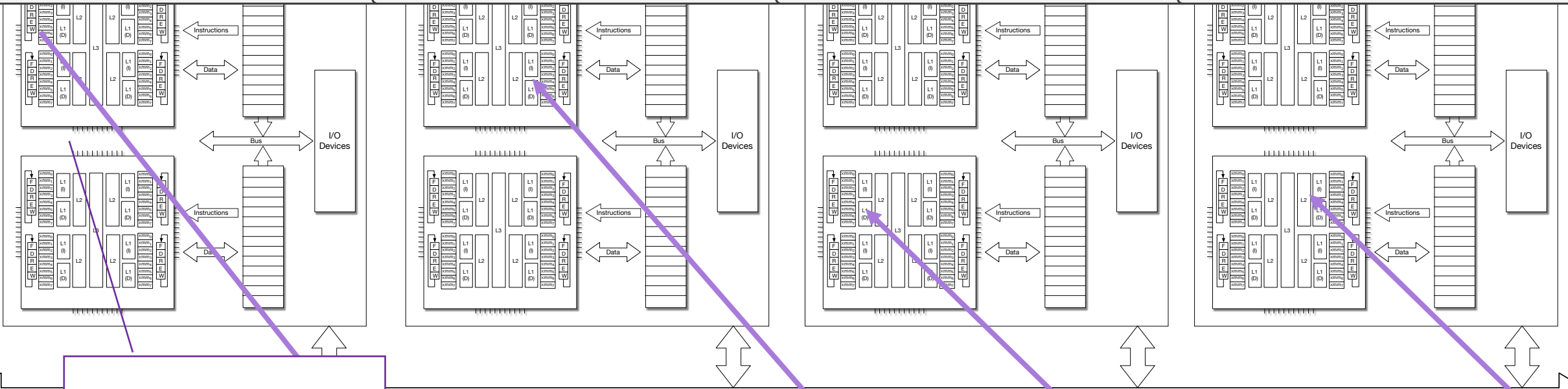
Single program: all still read from the command line

```
int main(size_t argc, char* argv[]) {  
    size_t N = atoi(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;  
}
```

```
int main(s  
size_t N  
double h  
double p  
  
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;  
}
```

```
main(size_t argc,  
size_t N = atoi(a  
double h = 1.0 /  
double pi = 0.0;  
  
for (size_t i = begi  
    pi_i += (h * 4.0)  
  
std::cout << "pi is ~ " << pi << std::endl;  
  
return 0;  
}
```

```
char* argv[]) {  
    argv[1]);  
    double) N;  
  
    ; i < end; ++i)  
        (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::endl;

    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?

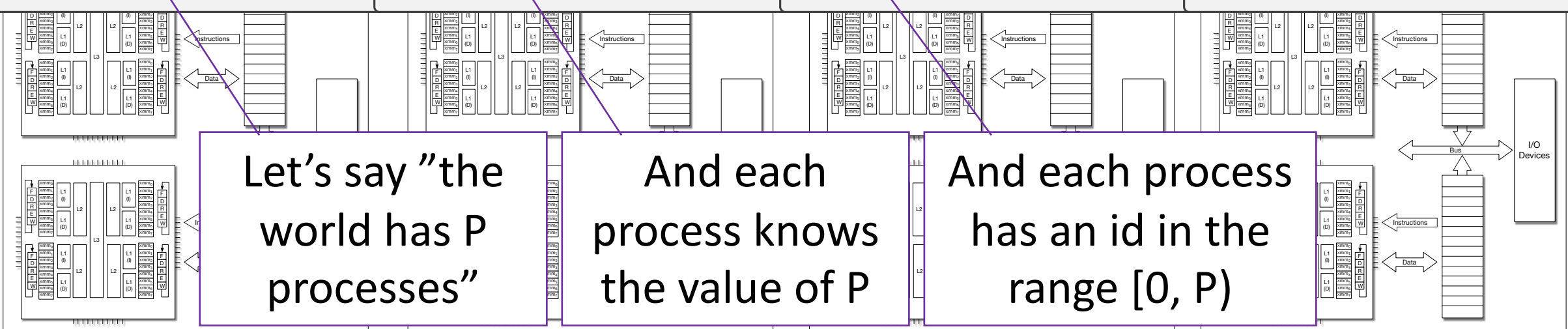
Single Program Multiple Data

```
int main(size_t argc, char* argv[]) {  
    size_t N = atoi(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;  
}
```

```
int main(size_t argc, char* argv[]) {  
    size_t N = atoi(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;  
}
```

```
int main(size_t argc, char* argv[]) {  
    size_t N = atoi(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;  
}
```

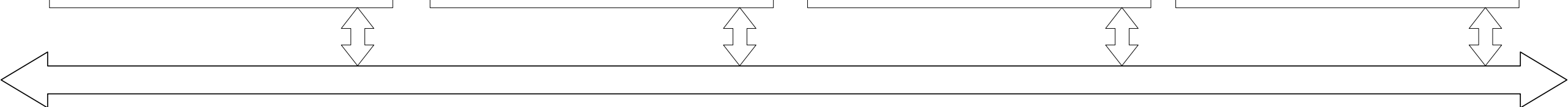
```
int main(size_t argc, char* argv[]) {  
    size_t N = atoi(argv[1]);  
    double h = 1.0 / (double) N;  
    double pi = 0.0;  
  
    for (size_t i = begin; i < end; ++i)  
        pi_i += (h * 4.0) / (1.0 + (i * h * i * h));  
  
    std::cout << "pi is ~ " << pi << std::endl;  
  
    return 0;  
}
```



Let's say "the world has P processes"

And each process knows the value of P

And each process has an id in the range [0, P)



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

Compute begin
and end

Only one node
prints pi

Is that going
to be correct?

Is that going
to be correct?

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?

Distinguished Replicated Processes

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

```
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;
}
```

Not here

```
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;
}
```

Not here

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

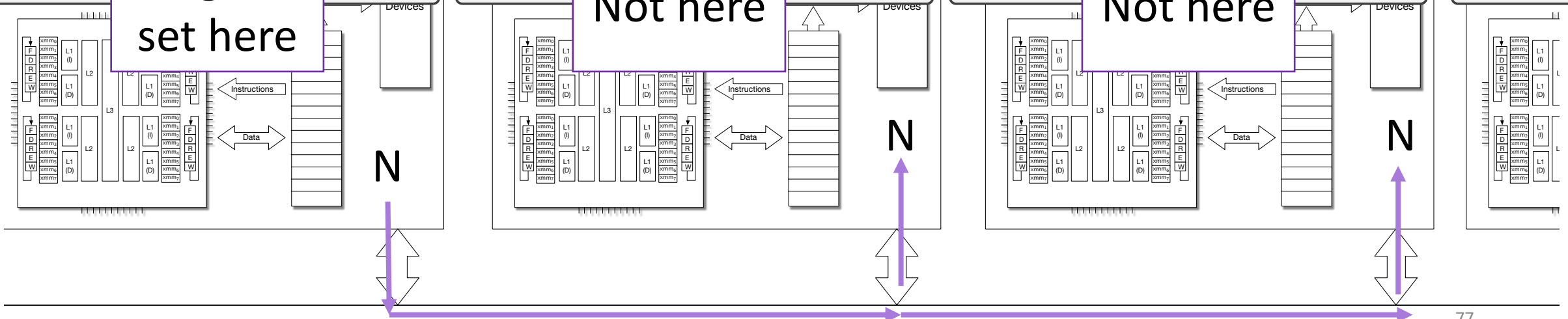
    size_t N = -1;
    if (0 == my_id) {
        N = atol(argv[1]);
    }

    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;
}
```



Finally

Get our id and number of other nodes

This pattern is ubiquitous

id 0 gets N

id shares N

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

```
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);

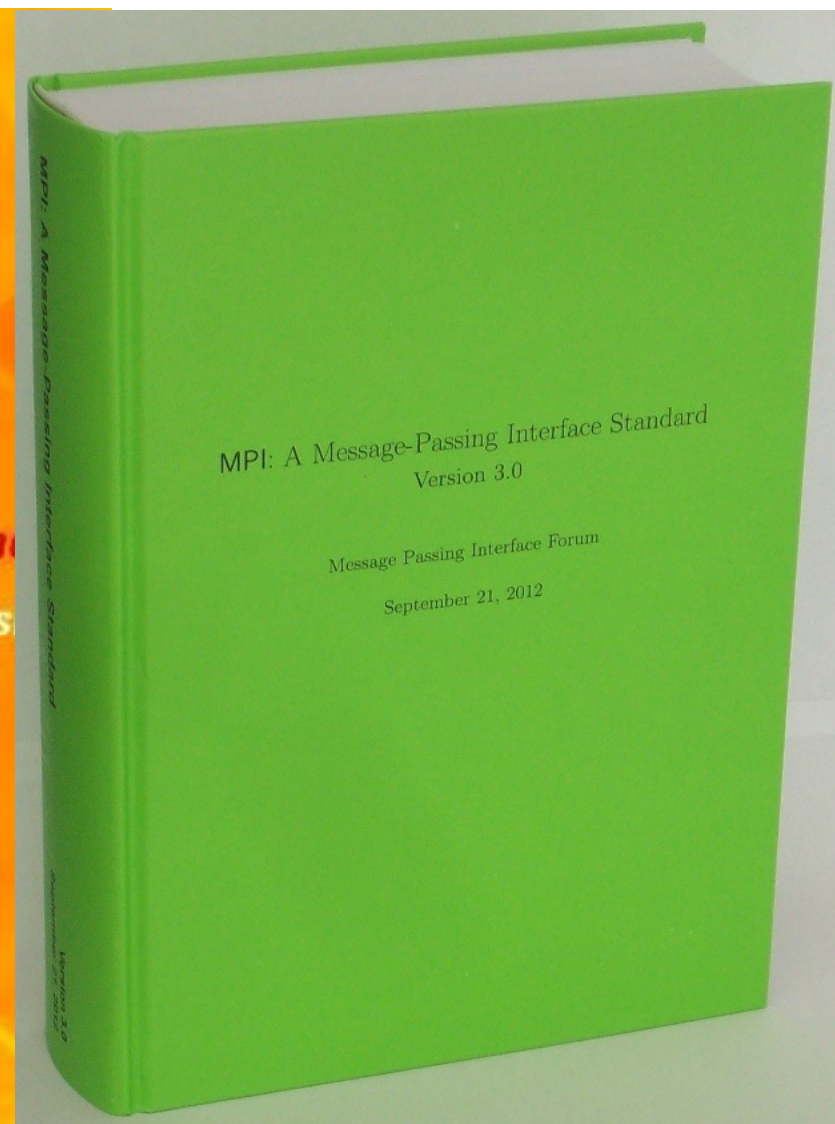
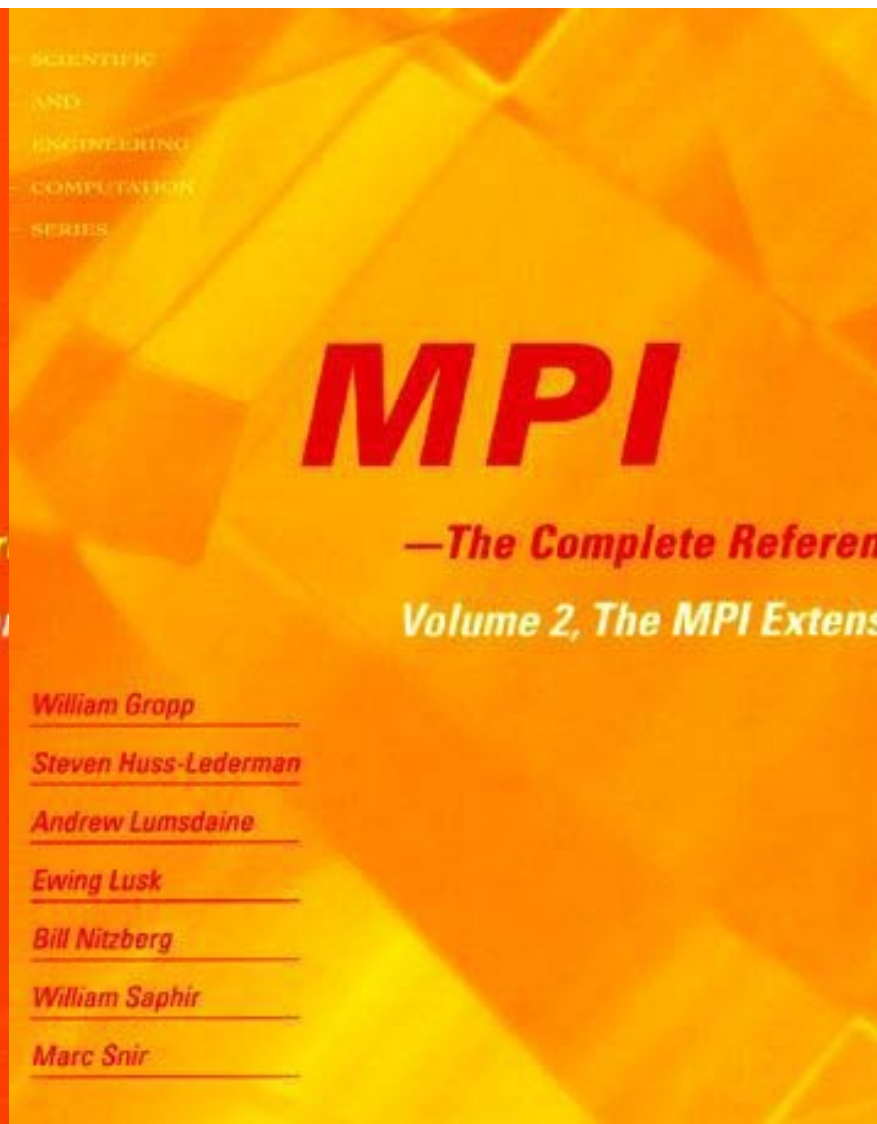
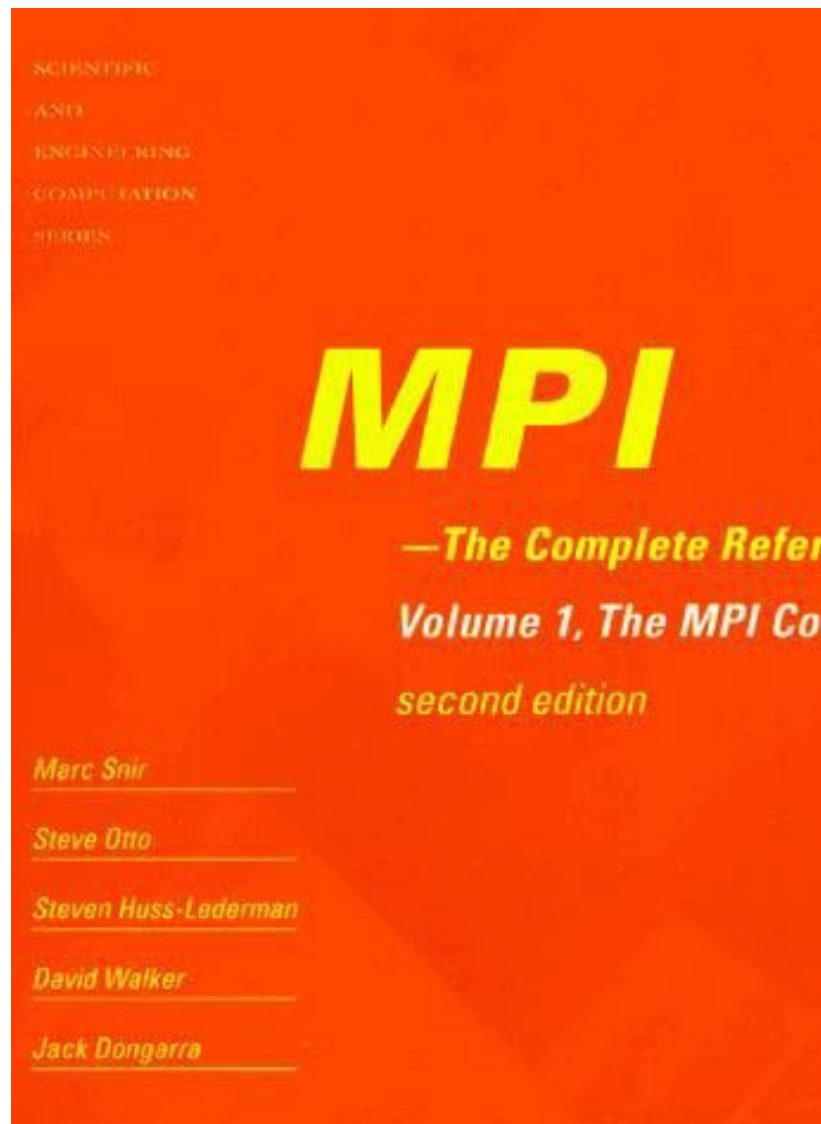
    if (0 == myrank) {
        std::cout << "pi is approximately " << pi << std::endl;
    }

    MPI::Finalize();

    return 0;
}
```

id 0 is root

The Message Passing Interface (MPI)



Thank You!

Creative Commons BY-NC-SA 4.0 License



© Andrew Lumsdaine, 2017-2022

Except where otherwise noted, this work is licensed under

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

