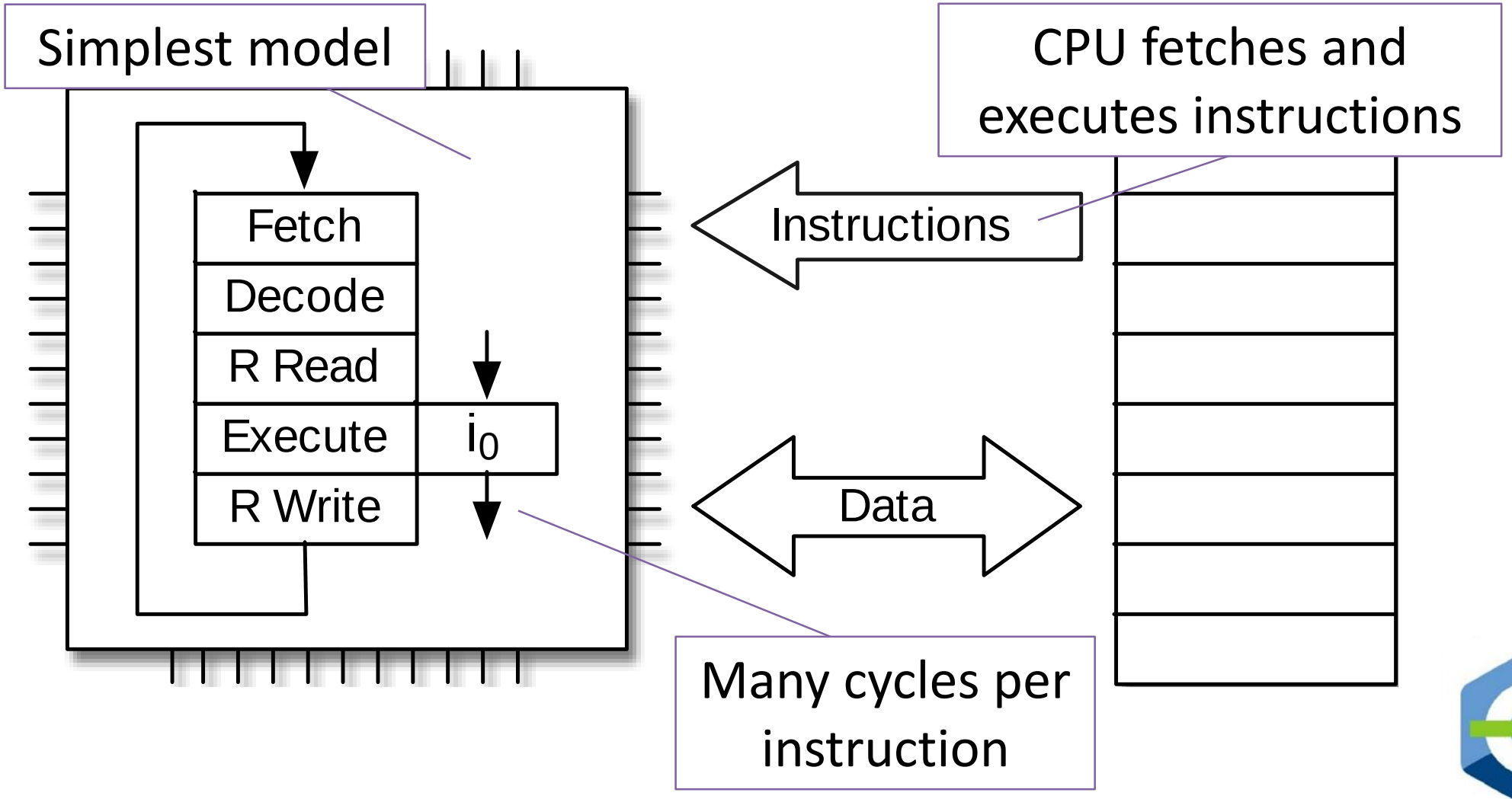
Introduction to Distributed Parallelism

Lecture 16

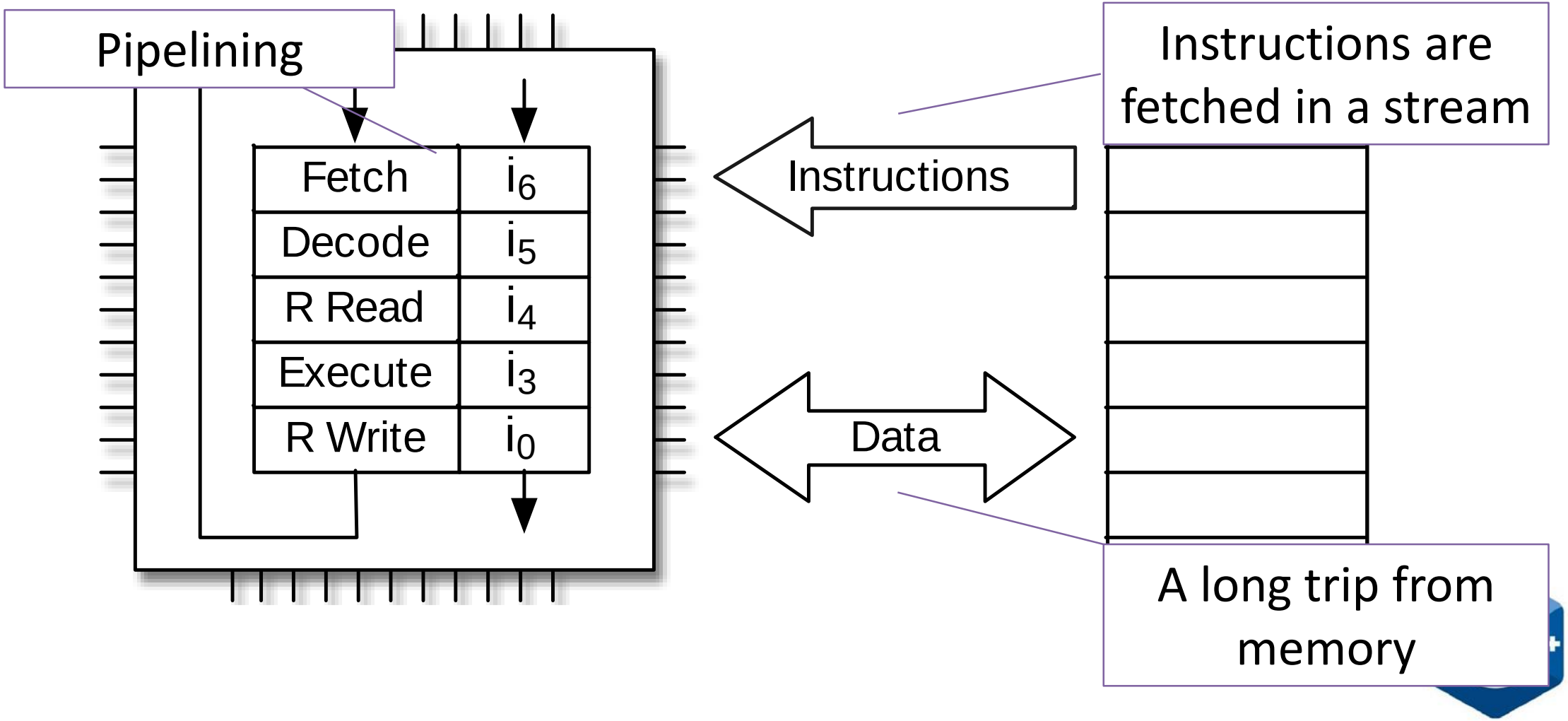
Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4700/

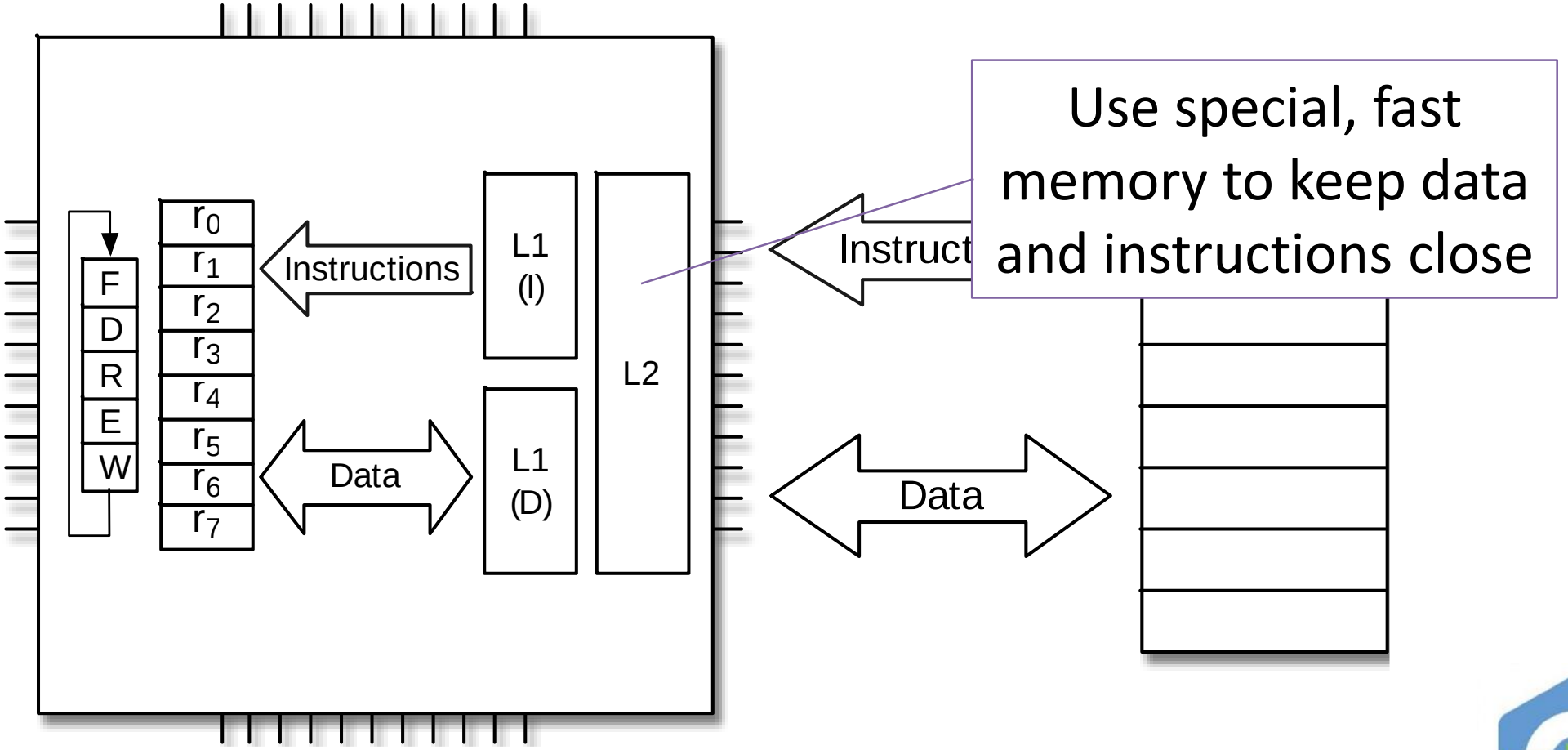
Scaling Progression of CPUs



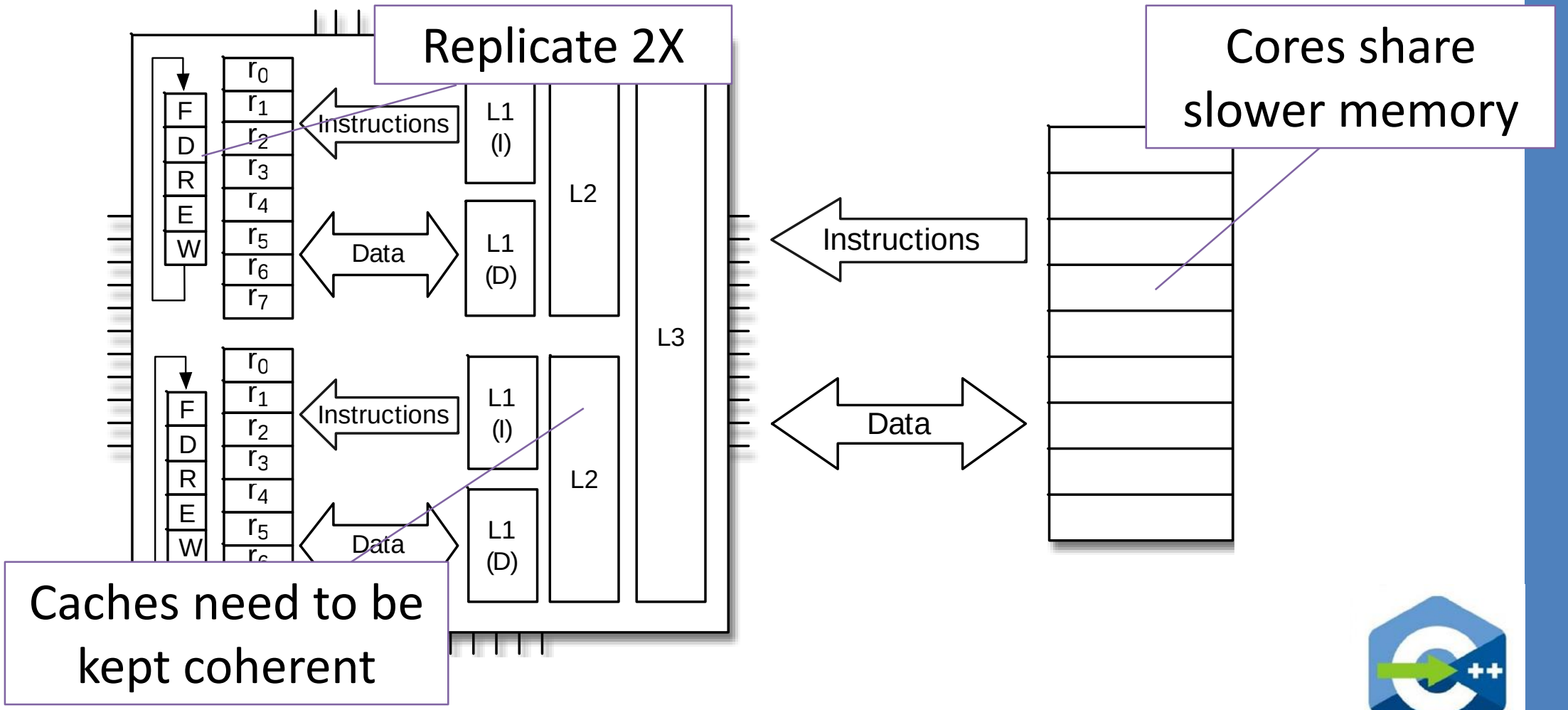
Pipelining


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

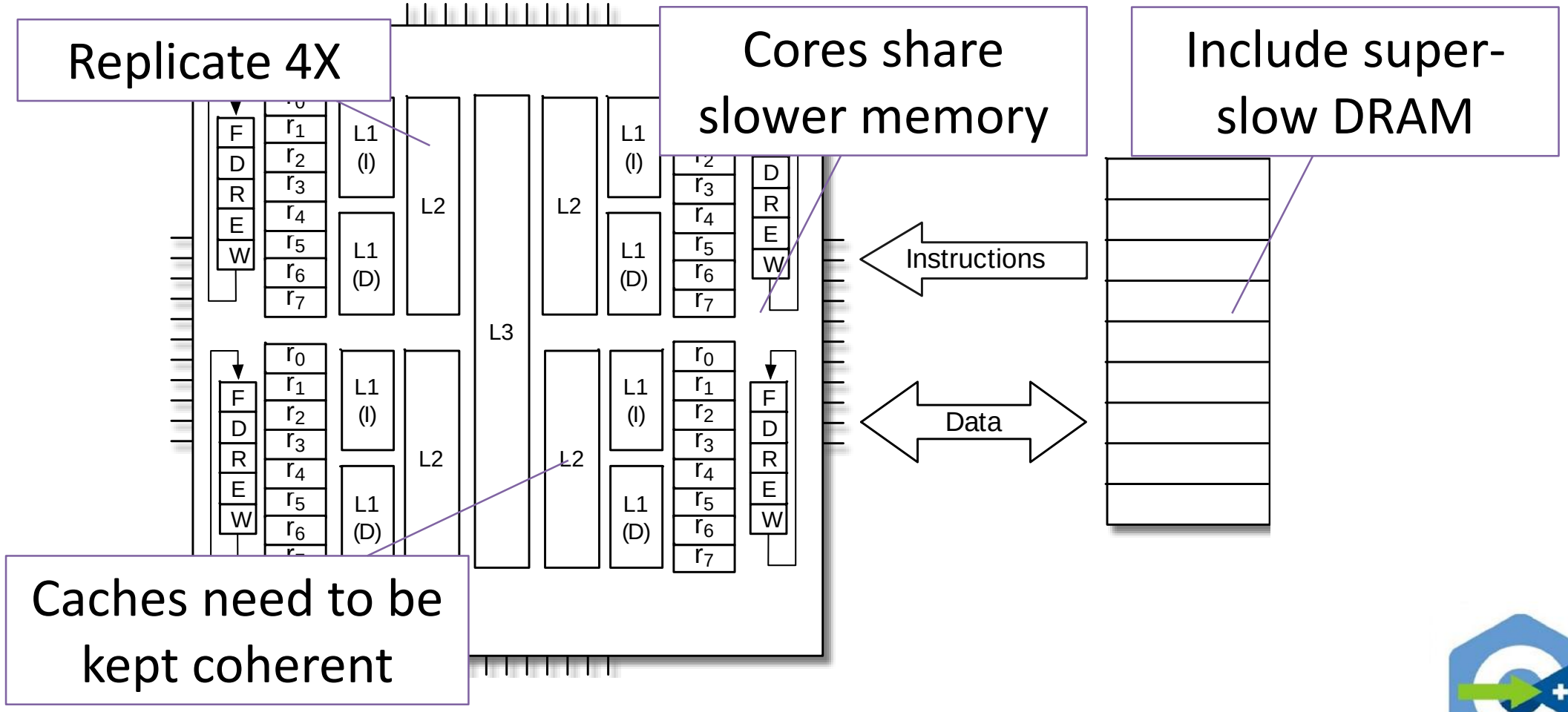
Hierarchical Memory



Multicore CPUs



Even more Cores

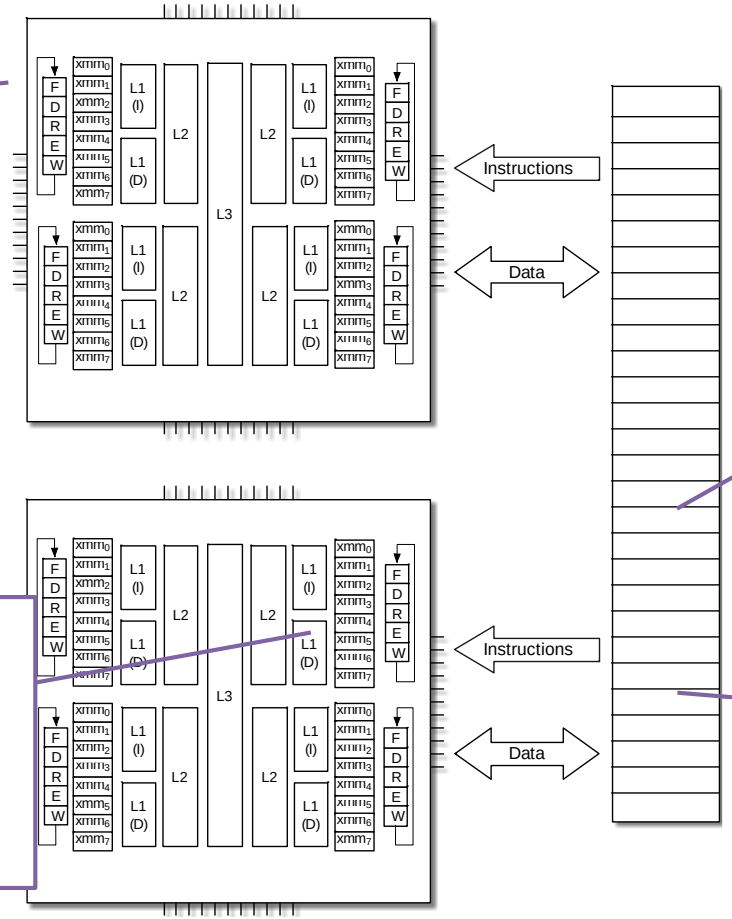


Symmetric Multi-Processor (SMP)

Multiple CPU chips

AKA “sockets”

Caches still need to be kept (somewhat) coherent



Memory may be uniformly shared among sockets

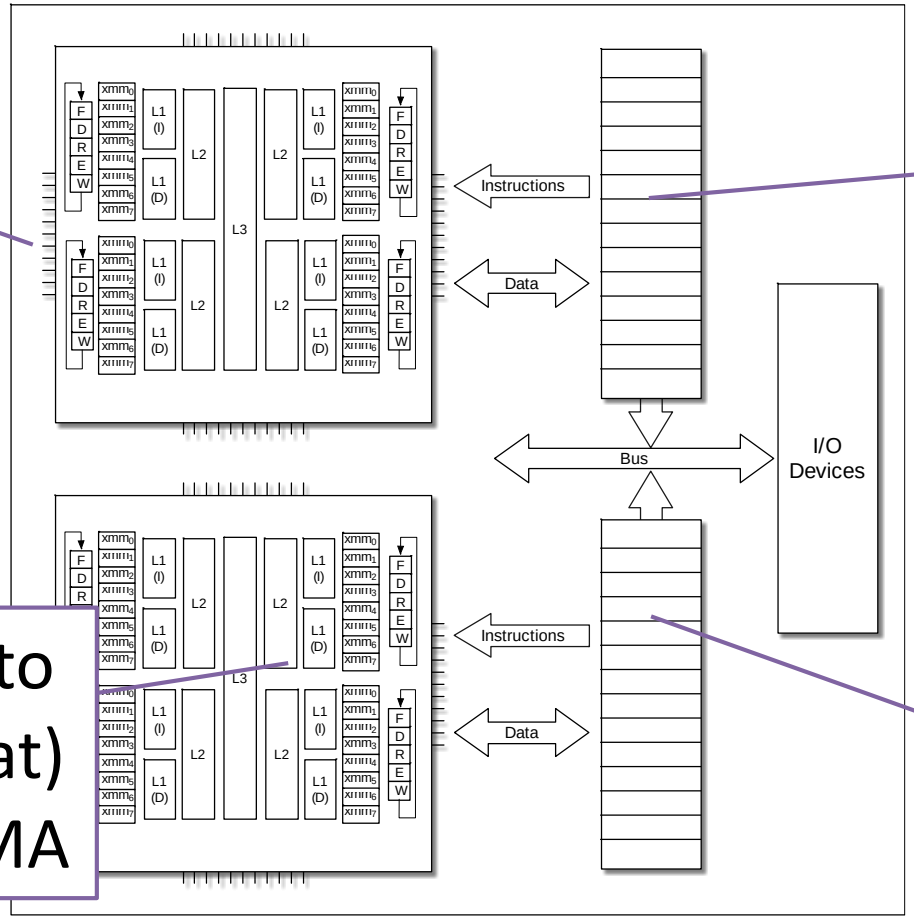
Uniform memory access (UMA)

Asymmetric Multi-Processor

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

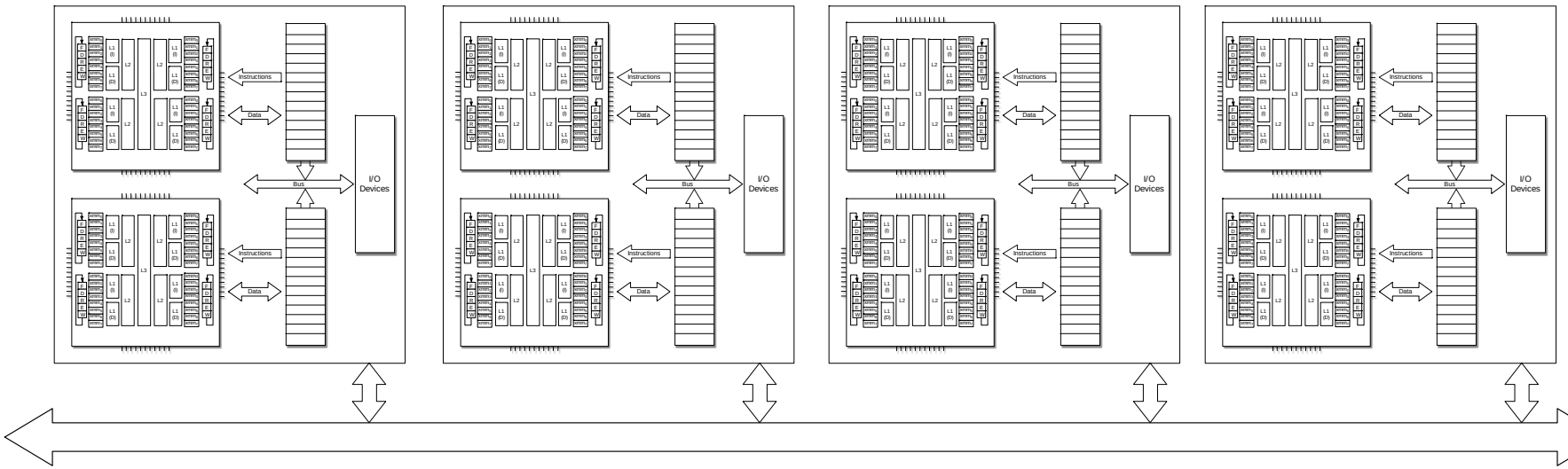
Put sockets on a blade

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!

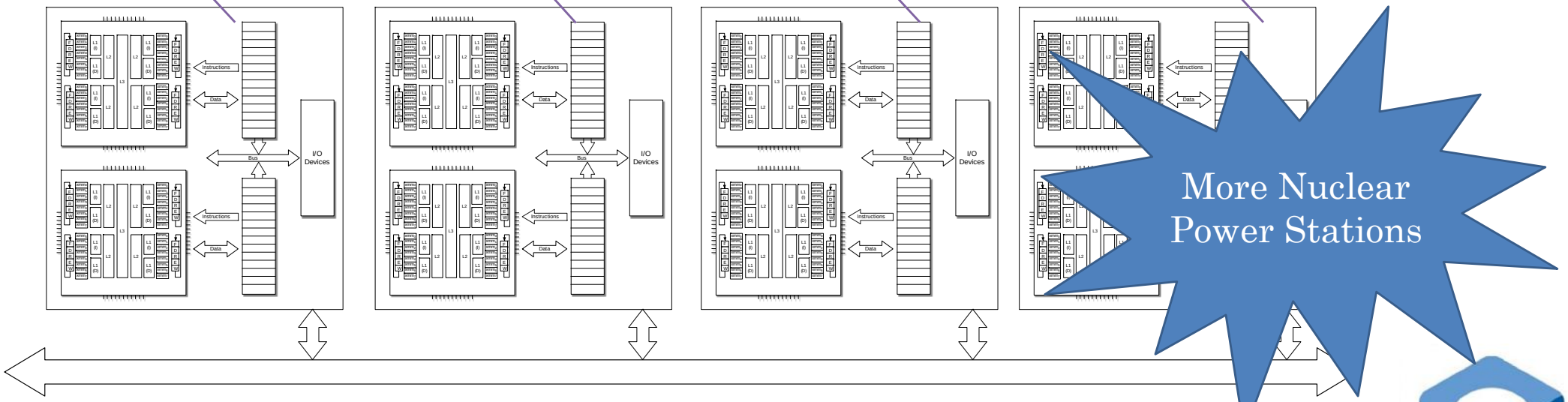
More cores!

More blades!

More chassis!

More racks!

More centers!



More Nuclear Power Stations

Top500 as of Nov 2024 (top500.org)

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,039,616	1,742.00	2,746.38	29,581
2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
4	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	

11M cores

9M cores

9.2M cores

2M cores

The HPC Canon (as of 2025)

Technology	Paradigm	Hammer
CPU (single core)	Sequential	C++
SIMD/Vector (single co	Data parallel	
Multicore	Threads	
NUMA shared memory	Threads	
GPU	GPU	
Clusters	Message passing	

This semester

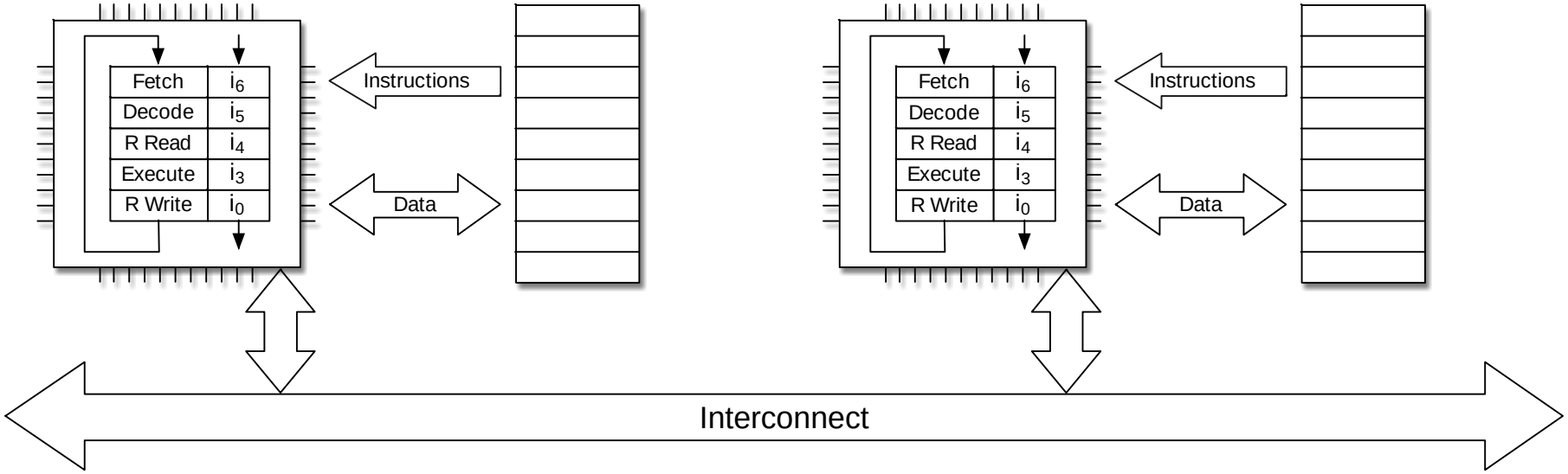
There are no “parallel” Computers

- …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)

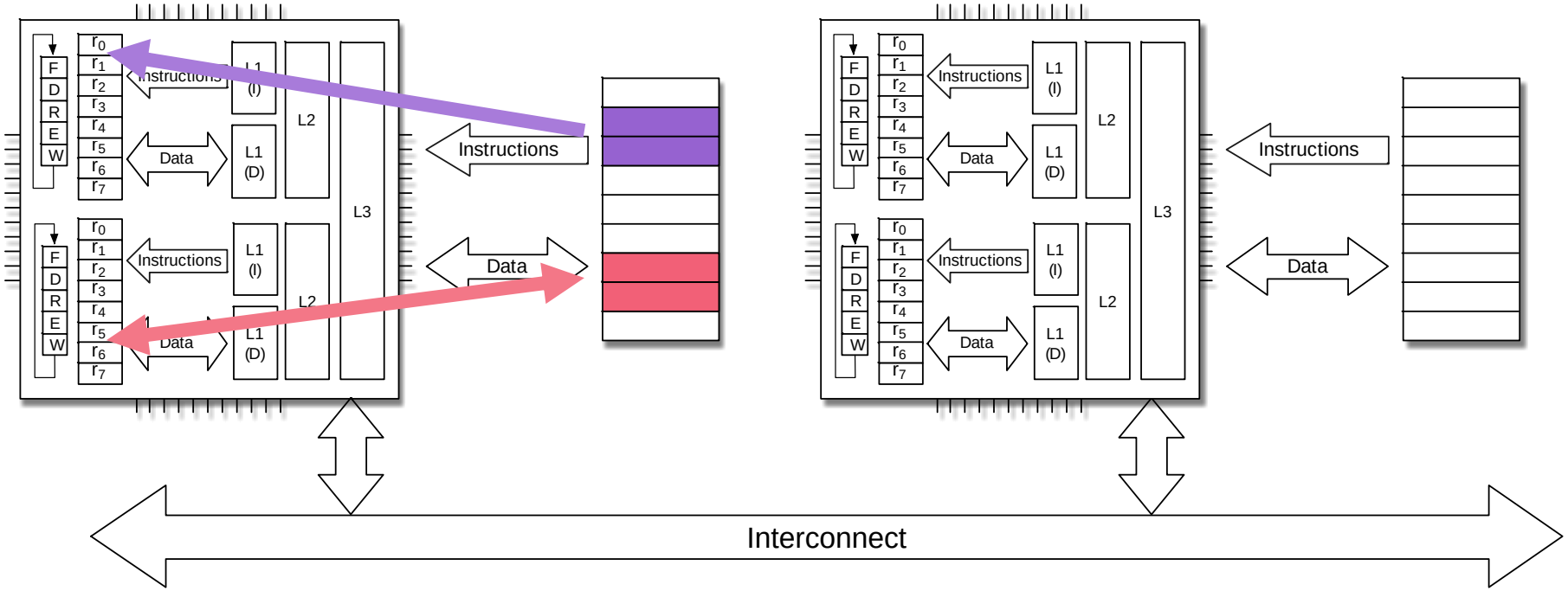
There are no “parallel” Programs

- …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

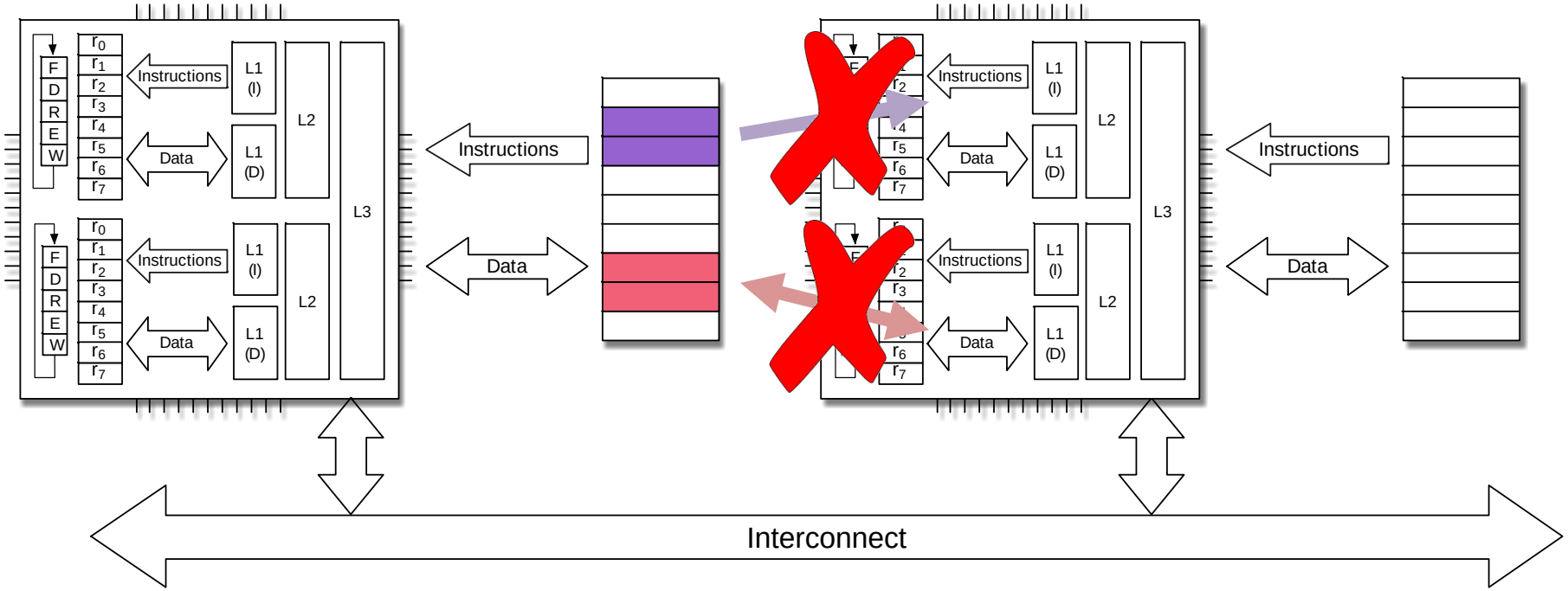
Distributed Memory



Distributed Memory

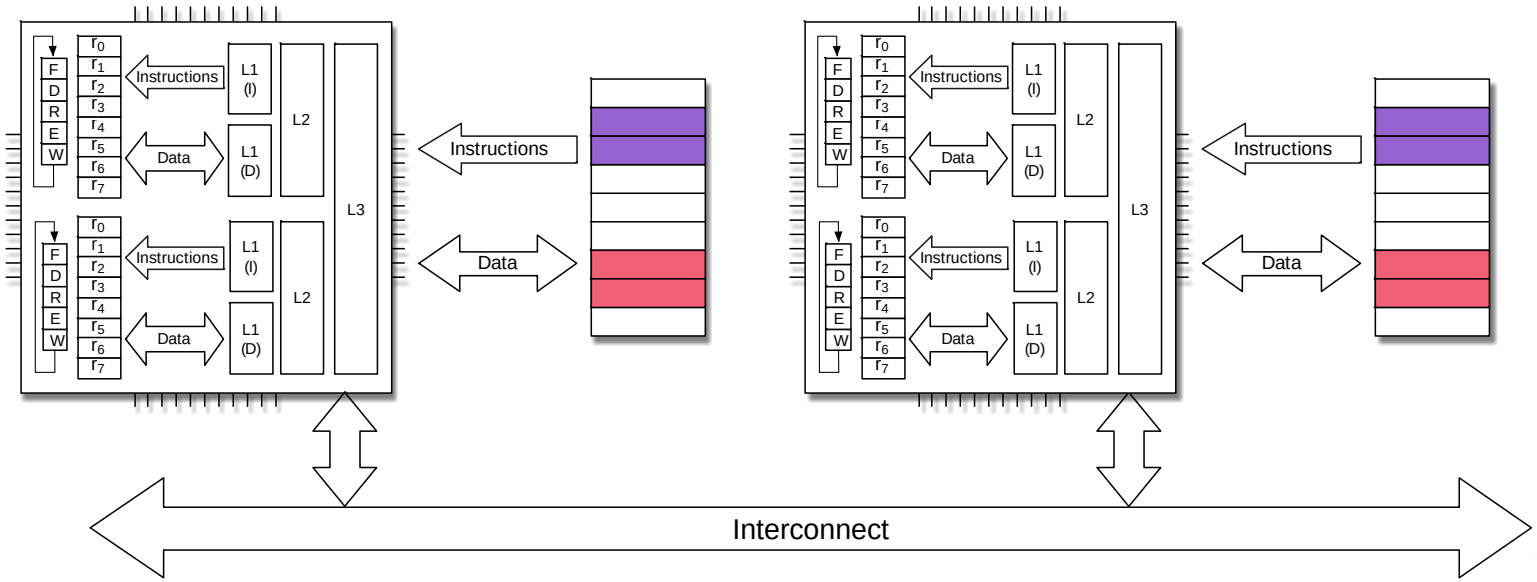


Distributed Memory

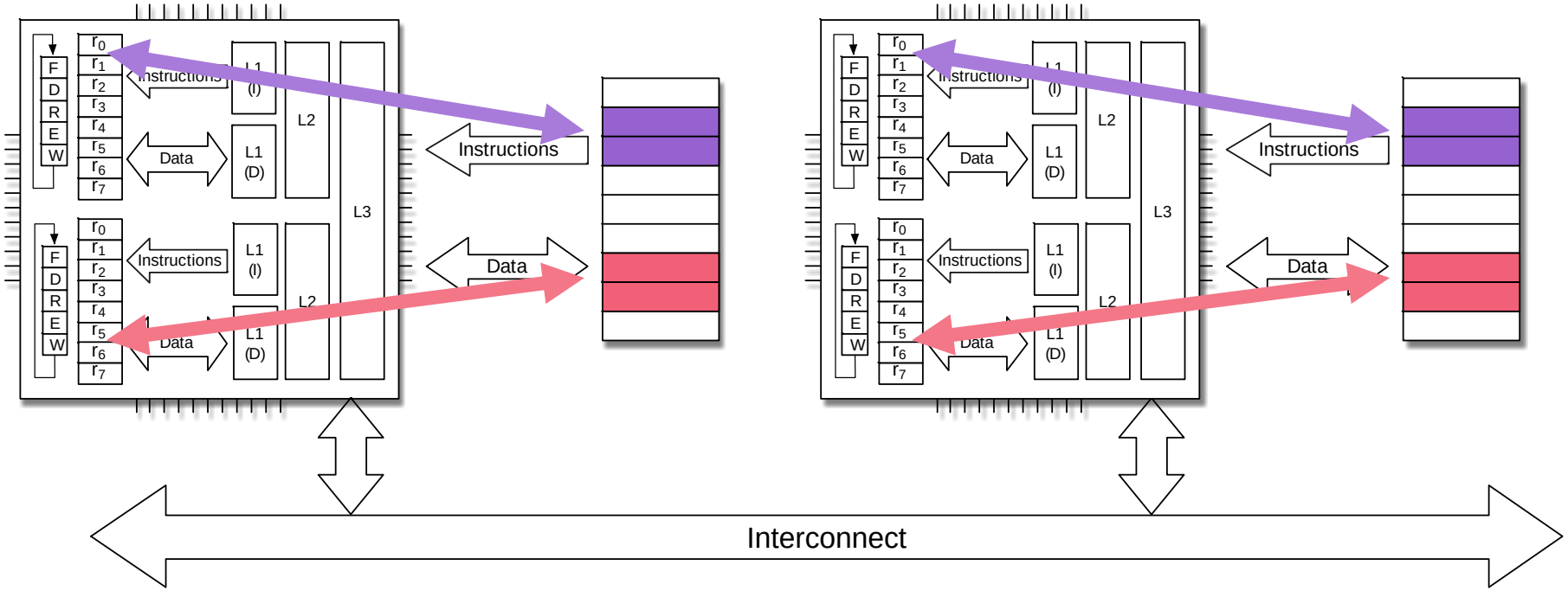


Distributed Memory

- Remote memory is not directly accessible
- Data needs to be explicitly copied



Distributed Memory



Distributed Memory

- This begs new questions:
  - Should all nodes do exactly the same thing?
  - Will there be speedup if we do?
- 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
- But. Do we need all the data on every node?
  - Hint: No
- What do we keep? What do we not keep?
  - Every node has some of the data, however, the union of all should be the whole problem
  - "Collectively exhaustive"

Distributed Memory

- What about the program?
  - Does it grow with problem size?
  - Hint: No
- We probably need all of it everywhere anyways

Single Program Multiple Data Model (SPMD)

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

- Most widely used model in distributed memory programming

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

https://doi.org/10.1016/0167-8191(88)90094-4

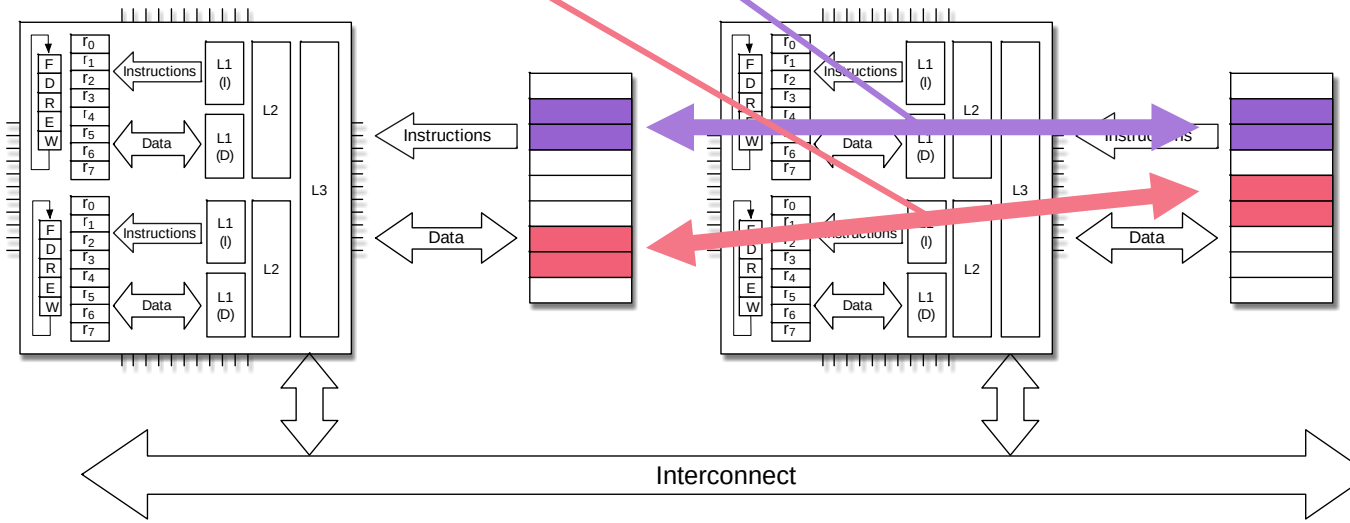
Get rights and content

Abstract

We present a single-program-multiple-data computational model which we have implemented in the EPEX system to run in parallel mode FORTRAN scientific application programs. The 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 of 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.

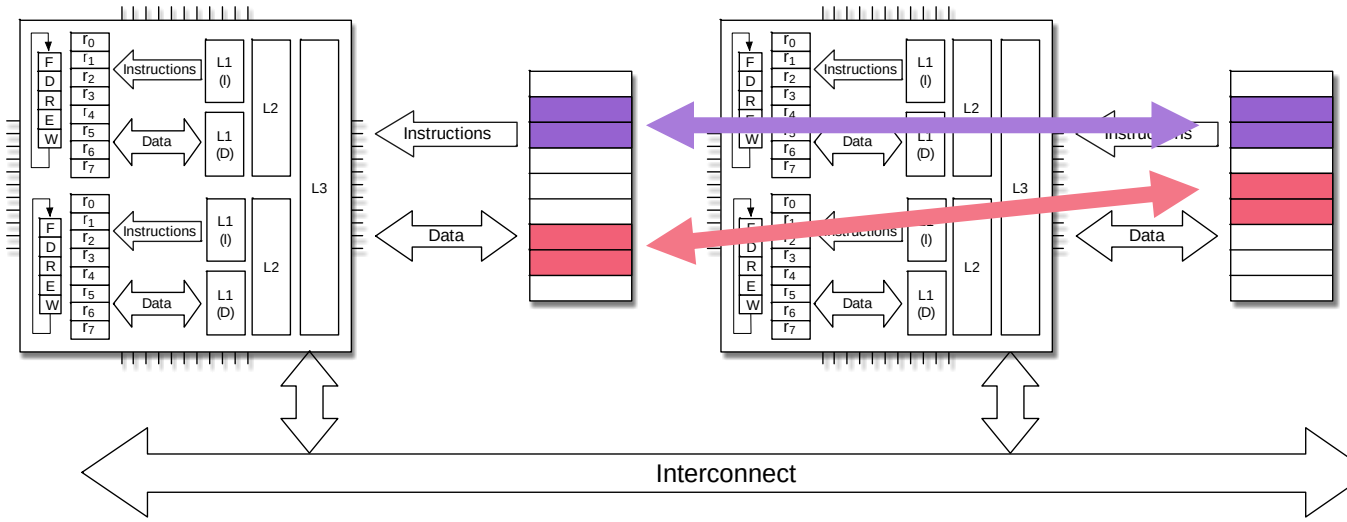
SPMD

- Single program
  - This is the same on all nodes
- Multiple Data
  - This is not the same on all nodes, but collectively exhaustive



SPMD

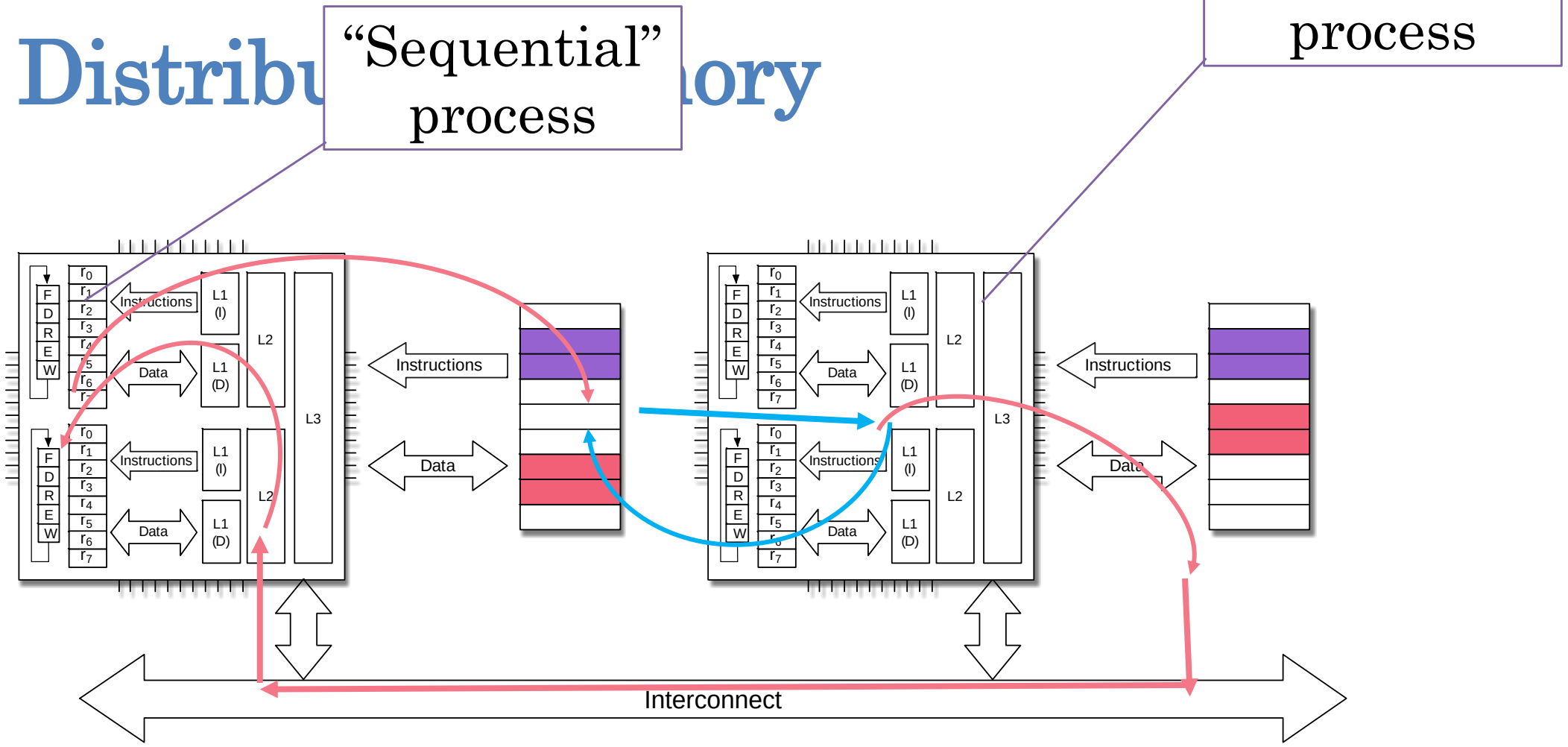
- Single program multiple data model (SPMD)
  - How do you pronounce “SPMD”?
- Recall Flynn: SIMD, MIMD
  - “SPMD” is pronounced “spim dee”
- Most widely used model in distributed memory programming
  - Better model for today’s practice than Flynn’s



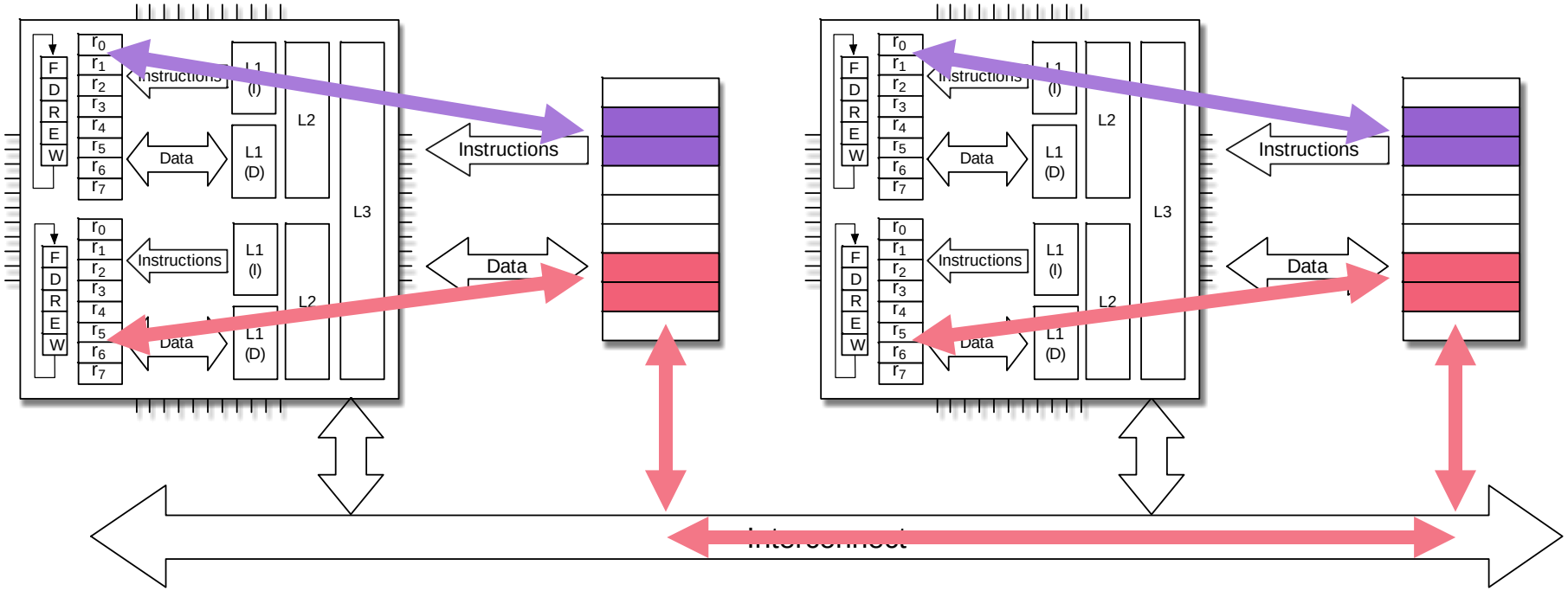
Distributed Memory

- Every node runs a sequential process (today often: parallel process)
  - All of the code is replicated
  - All nodes run the same executable
- Data is distributed using resource allocation mechanisms
  - However, data dependencies are probably not disjoint
  - Data has cross-node dependencies
  - Data may be needed by another node, but can't be accessed directly
  - Data are partitioned
  - The union of the partitions should be the whole problem

Distribu ory

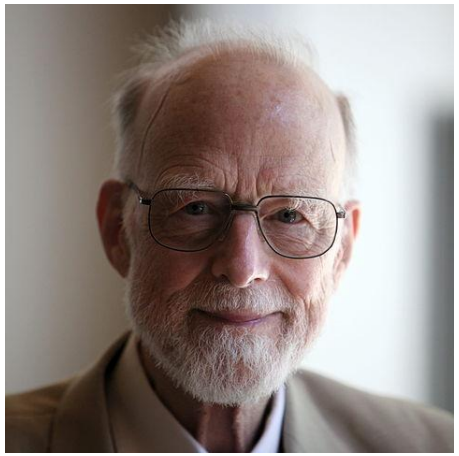


Distributed Memory



Communicating Sequential Processes

- 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 explore the logical foun
tions of computer programming by use of techniques wl
were first applied in the study of geometry and have l
been extended to other branches of mathematics. This
volves the elucidation of sets of axioms and rules of infere
which can be used in proofs of the properties of compu
programs. Examples are given of such axioms and rules,
a formal proof of a simple theorem is displayed. Finally,
argued that important advantages, both theoretical and pr
tical, may follow from a pursuance of these topics.

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

- Communicating Sequential Processes "CSP" (pronounced "see ess pea")

Example

Computing π

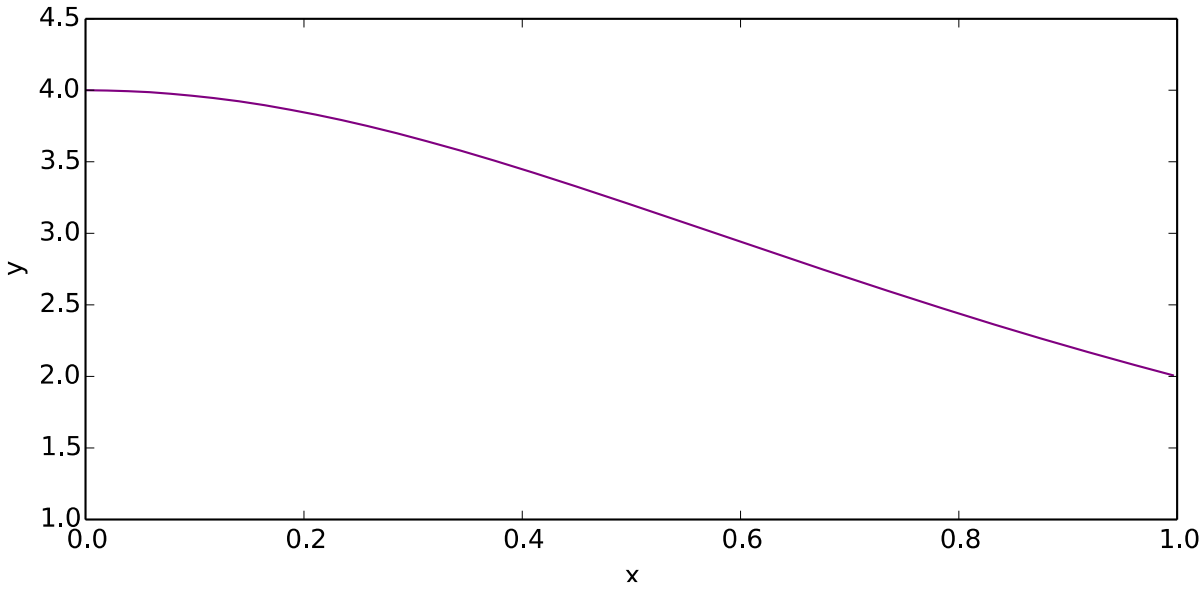
Back to our trusty Example (one of them)

- Find the value of

$$\pi$$

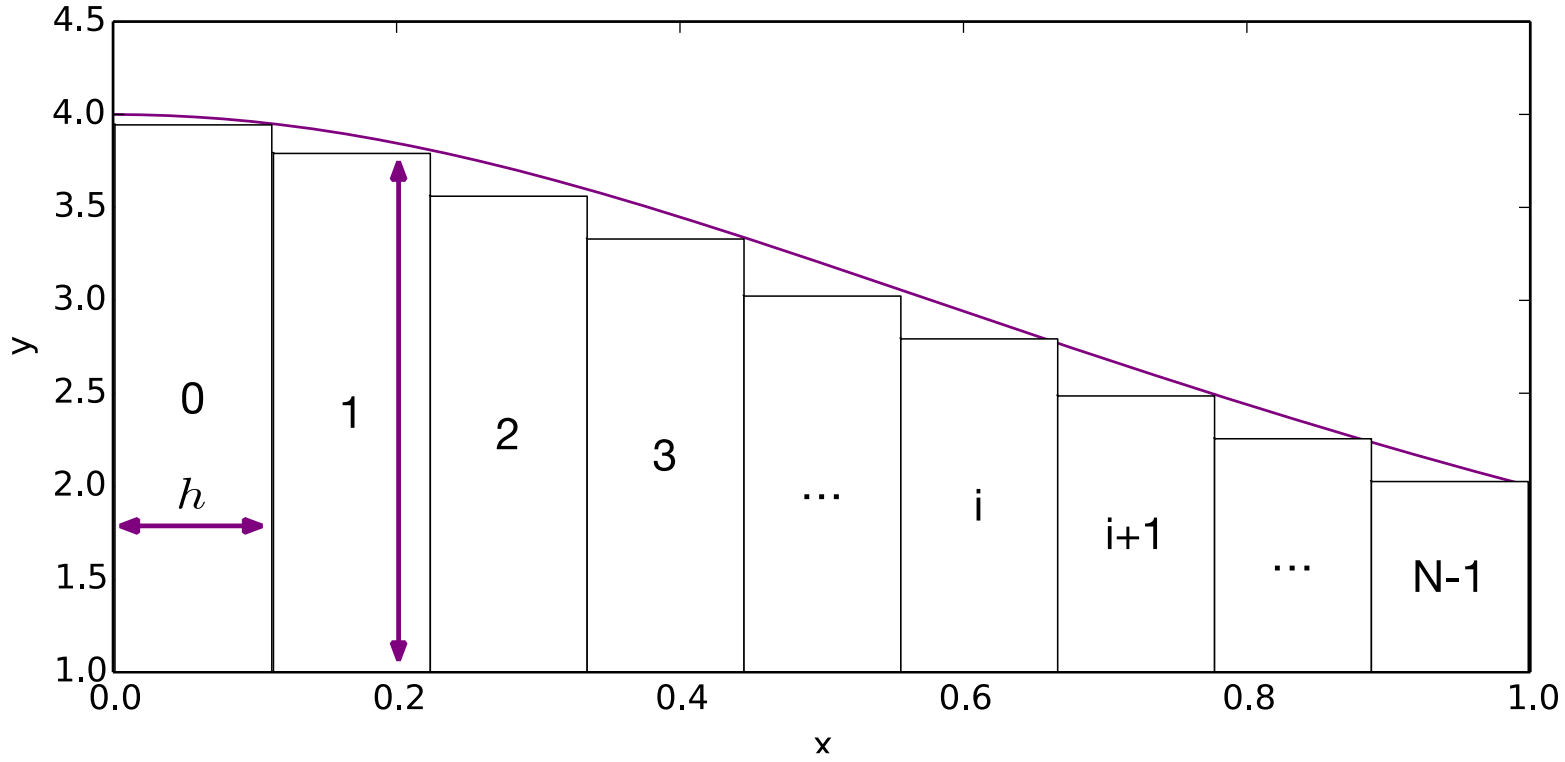
- Using formula

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



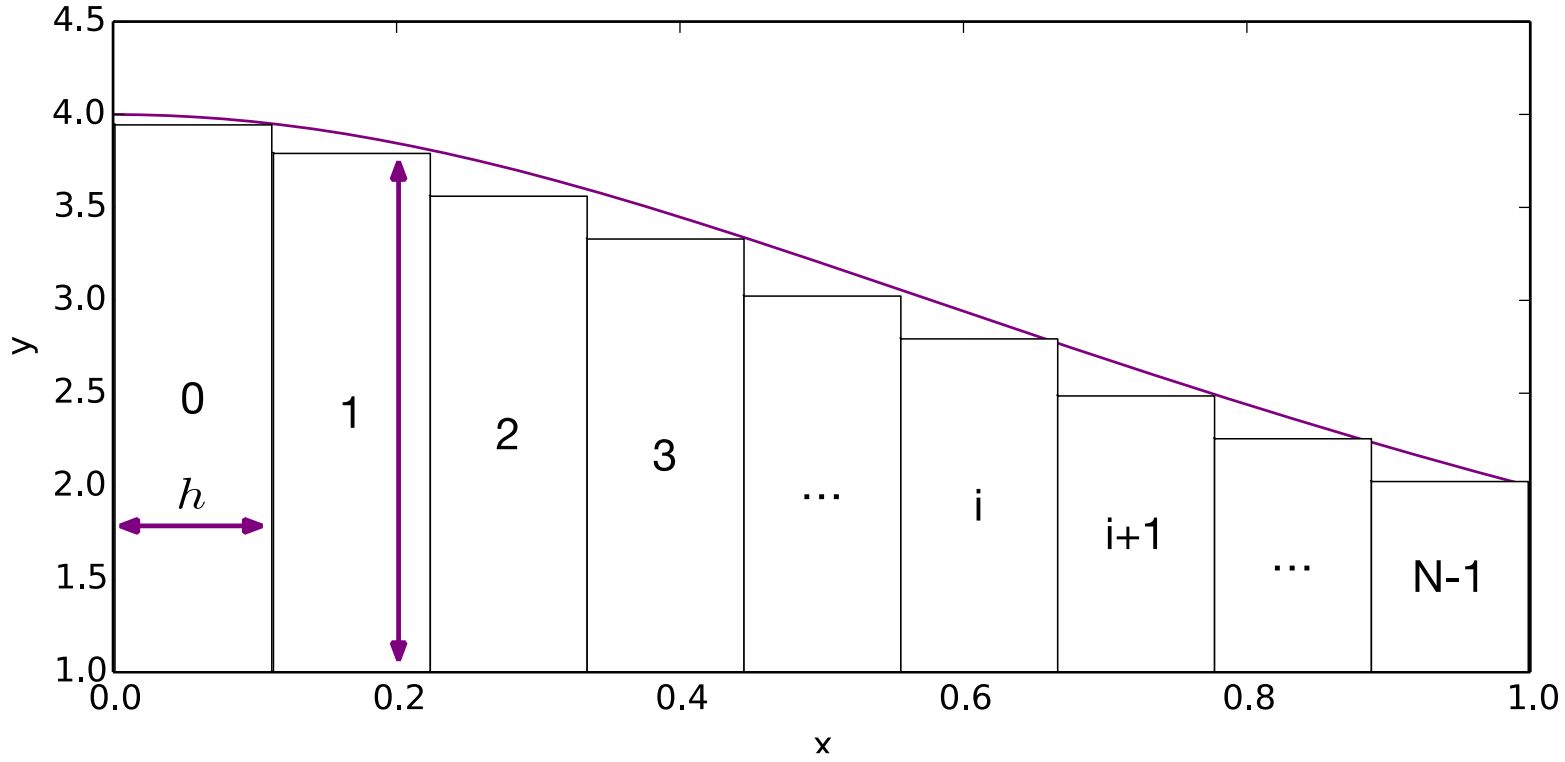
Numerical Integration

$$\frac{4}{1+x(i)^2} = \frac{4}{1+(ih)^2} \qquad A = h\frac{4}{1+(ih)^2}$$



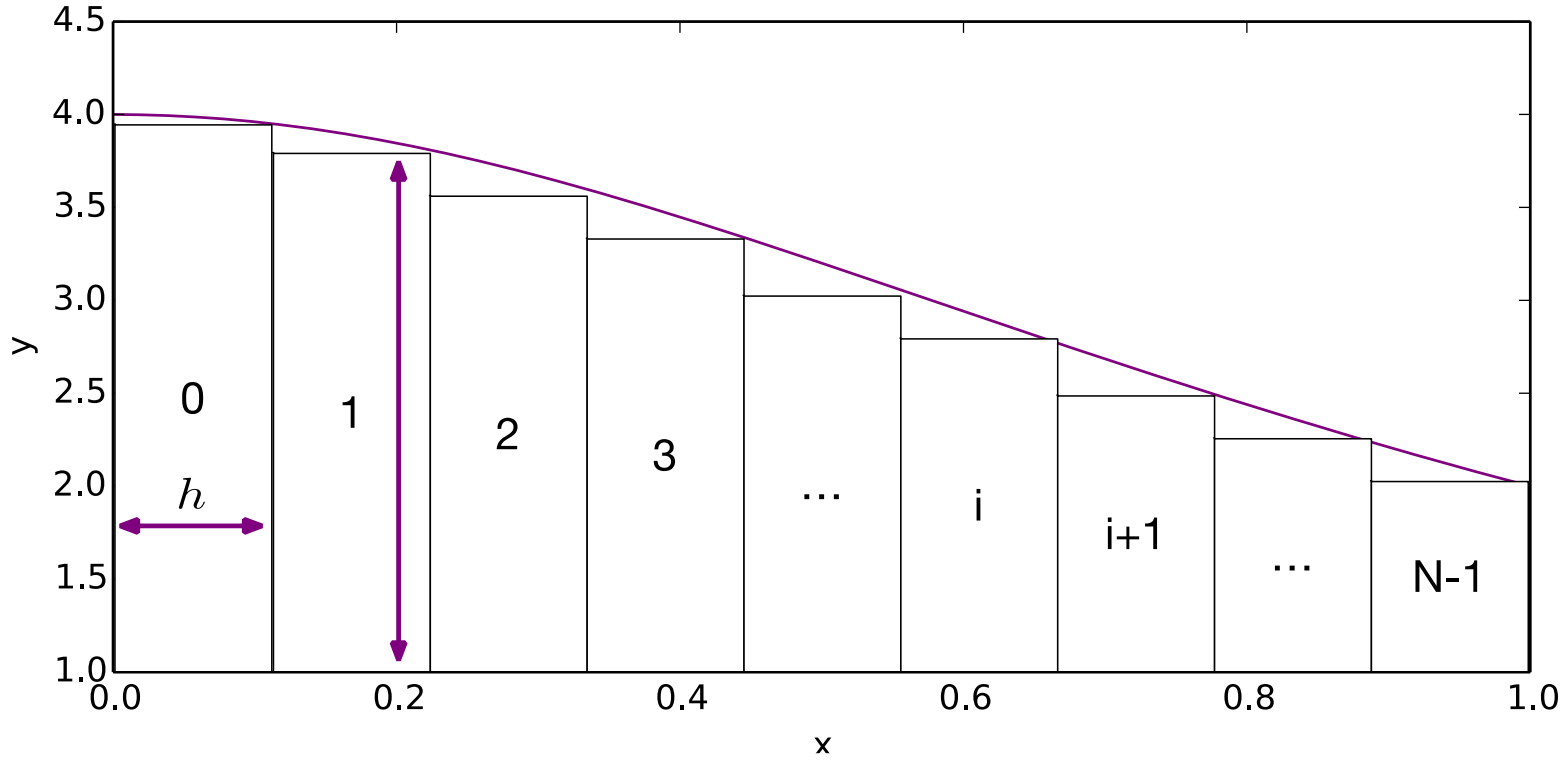
Numerical Integration

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



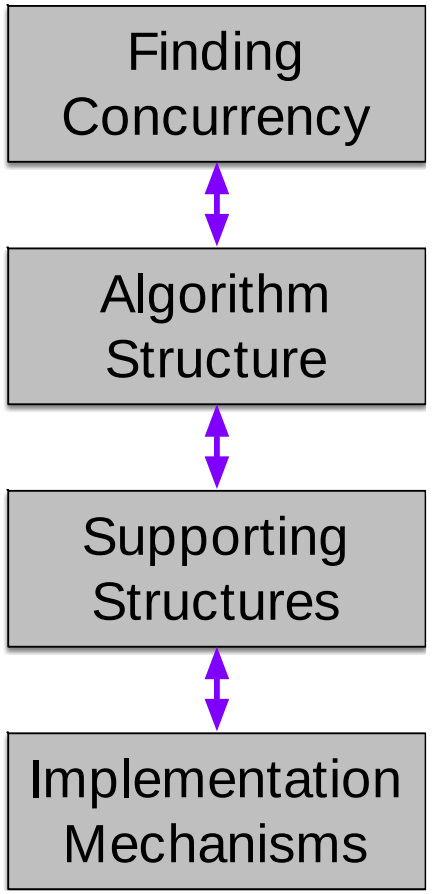
Numerical Integration (Sequential)

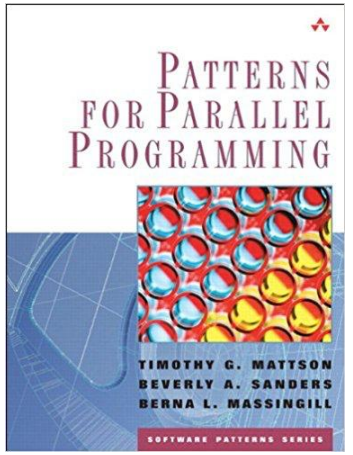
```
double pi = 0;
for (int i = 0; i != N; ++i)
    pi += h * 4.0 / (1 + sqr(i * h));
```



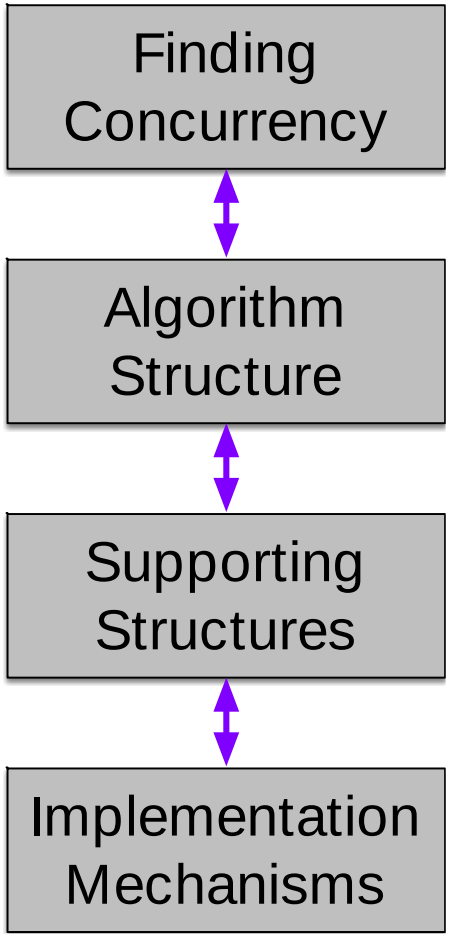
Parallelization Strategy

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





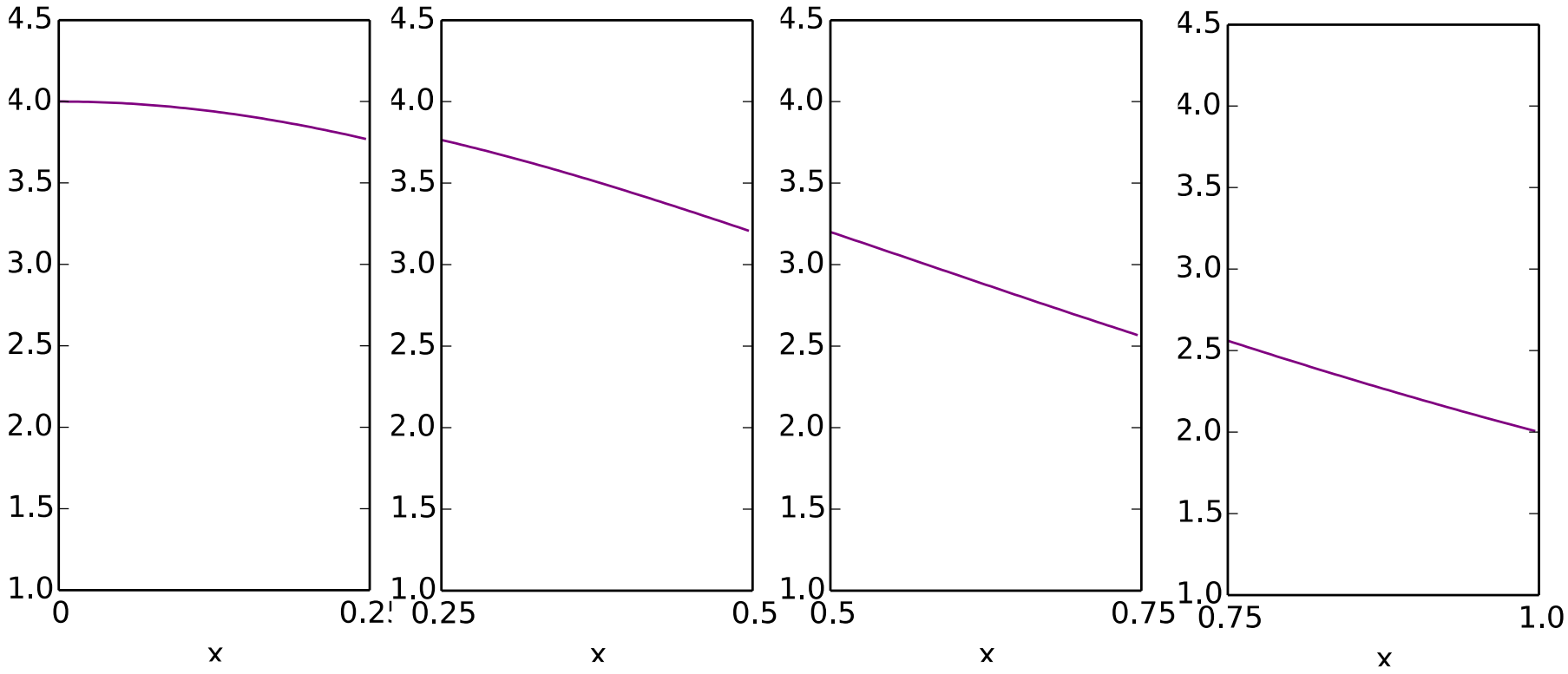
Parallelization Strategy, SPMD version



- Decompose problem into pieces that can execute in SPMD
  - By task or by data
  - Manage sharing (communication)
- Fundamental organizing principle
  - Around data decomposition?
  - Around tasks?
  - Around data flow?
- Programming paradigms and data structures: SPMD
- Manage processes, communication
  - Collective operations

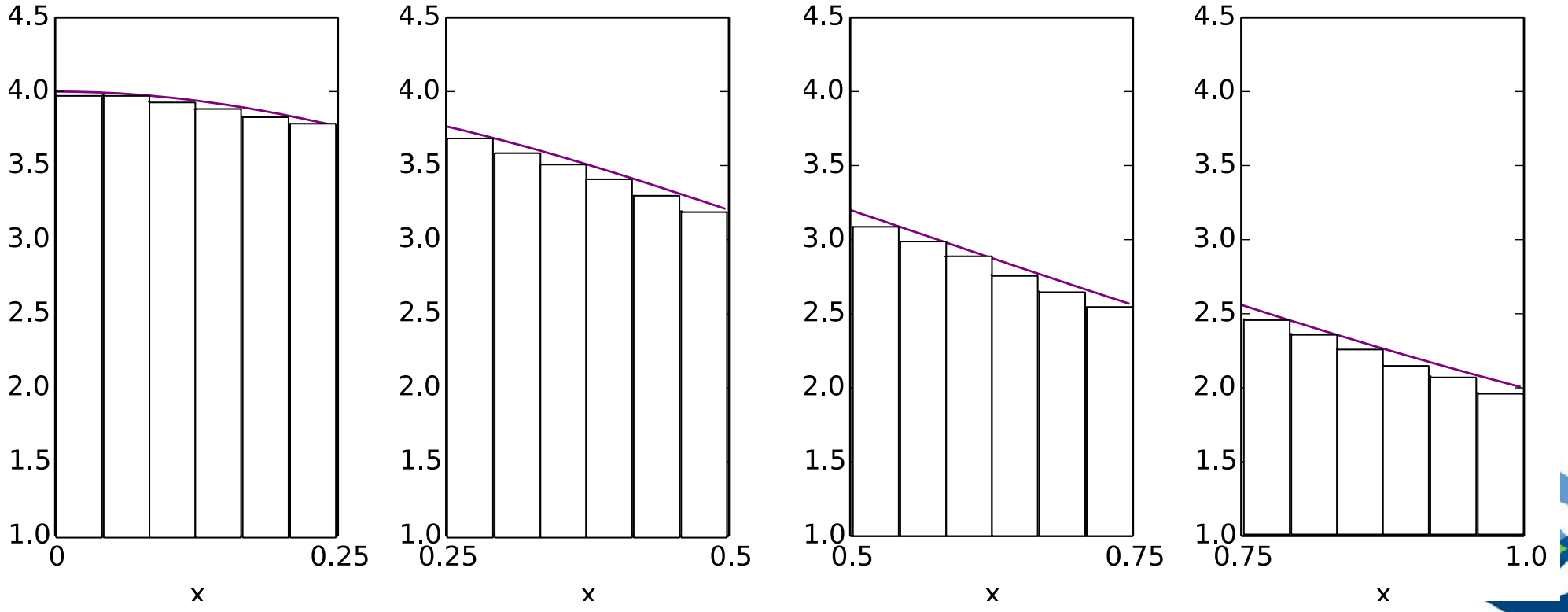
Finding Concurrency

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



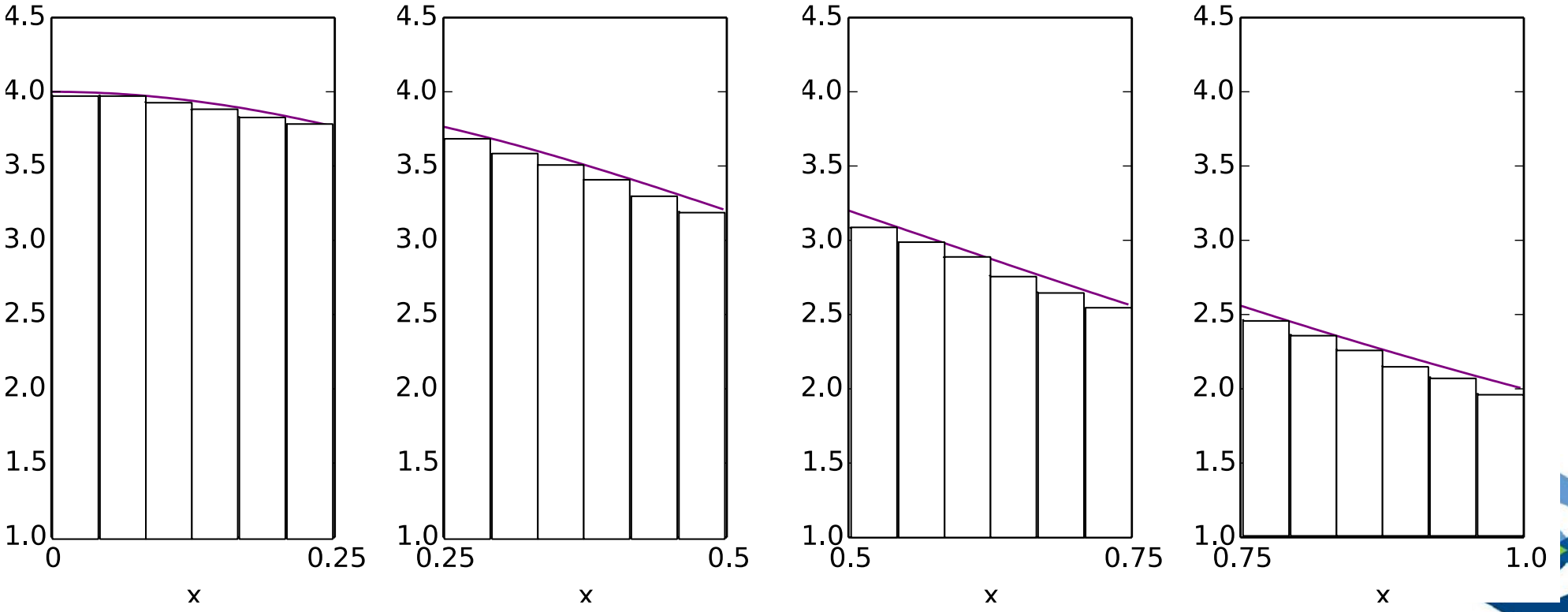
Finding Concurrency

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



Finding Concurrency

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



Finding Concurrency

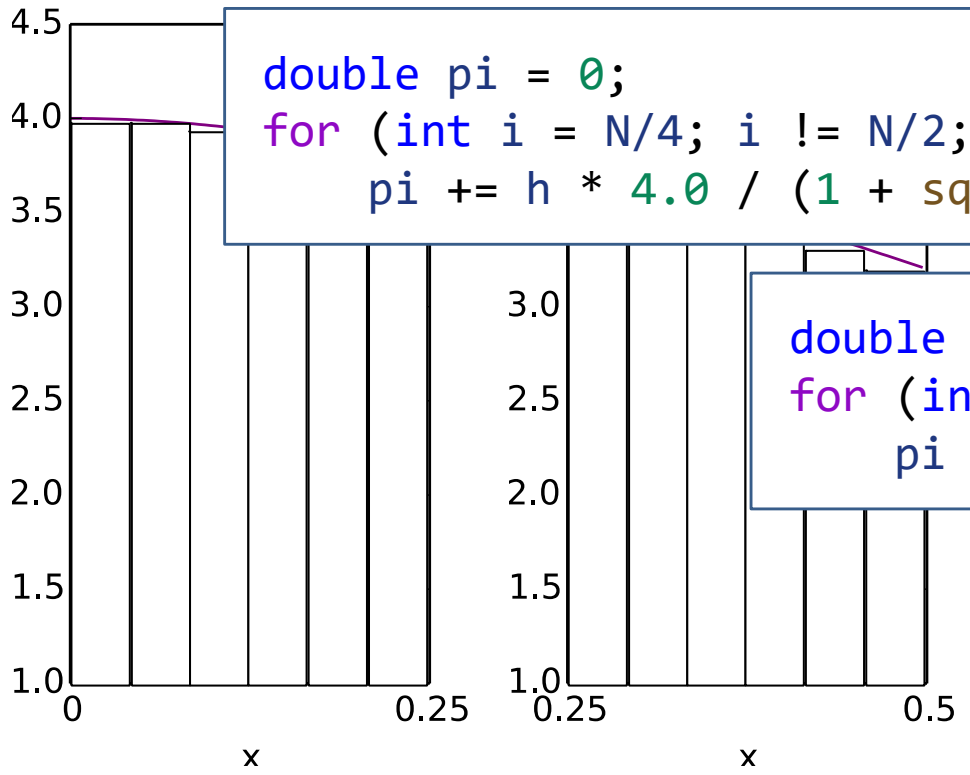
```
double pi = 0;
for (int i = 0; i != N/4; ++i)
    pi += h * 4.0 / (1 + sqr(i * h));
```

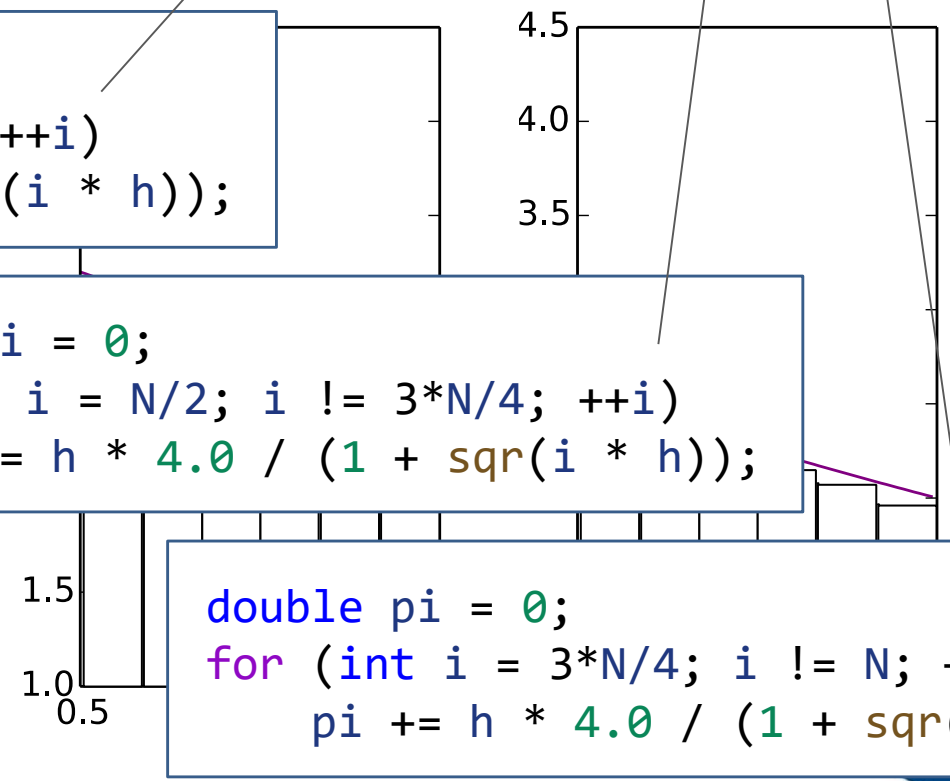
Same Program!
Different Data!
SPMD?

```
double pi = 0;
for (int i = N/4; i != N/2; ++i)
    pi += h * 4.0 / (1 + sqr(i * h));
```

```
double pi = 0;
for (int i = N/2; i != 3*N/4; ++i)
    pi += h * 4.0 / (1 + sqr(i * h));
```

```
double pi = 0;
for (int i = 3*N/4; i != N; ++i)
    pi += h * 4.0 / (1 + sqr(i * h));
```





Finding Concurrency

```cpp
int main()
{
    int const N = 1'000'000;
    double pi = 0;

    for (int i = 0; i != N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = N/4; i != N/2; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = N/2; i != 3*N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = 3*N/4; i != N; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    std::println("pi: {}", pi);
}
```

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

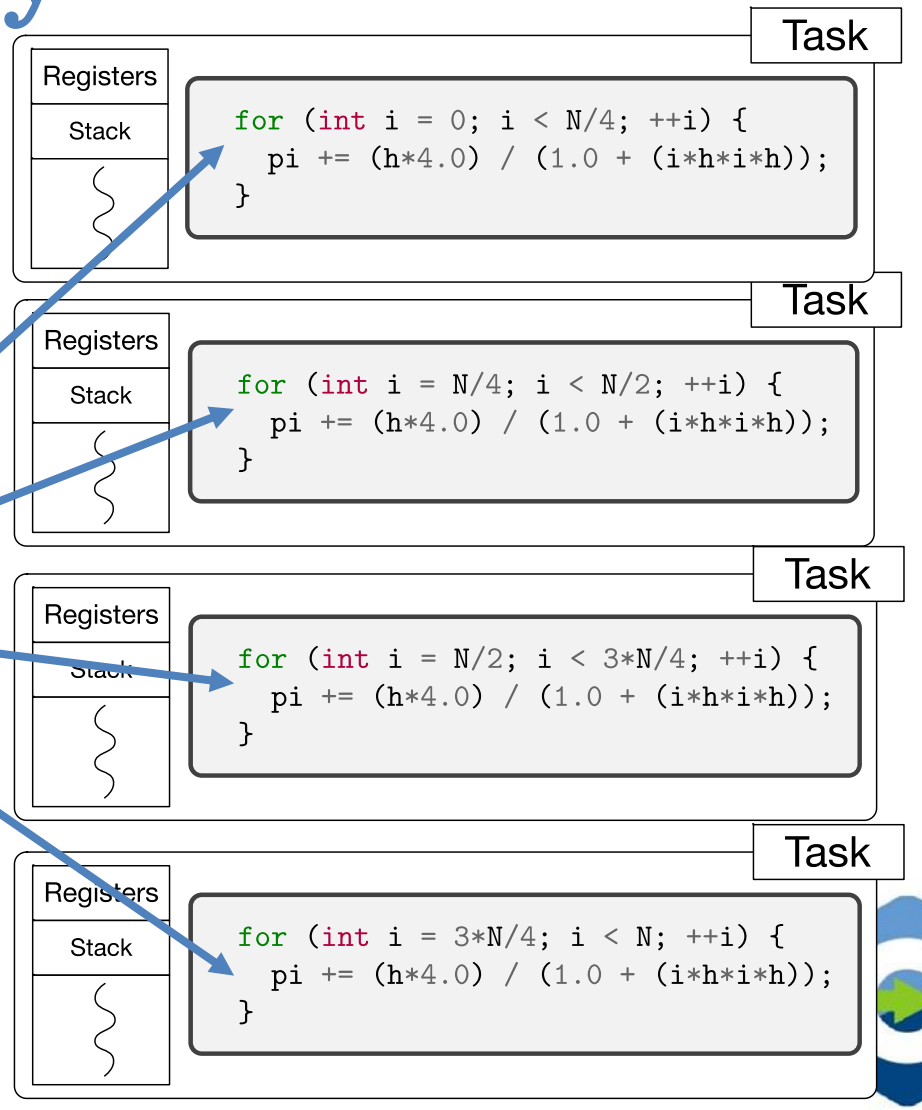
$$h \sum_{i=N/4}^{N/2-1} \frac{4}{1+(ih)^2}$$

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

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

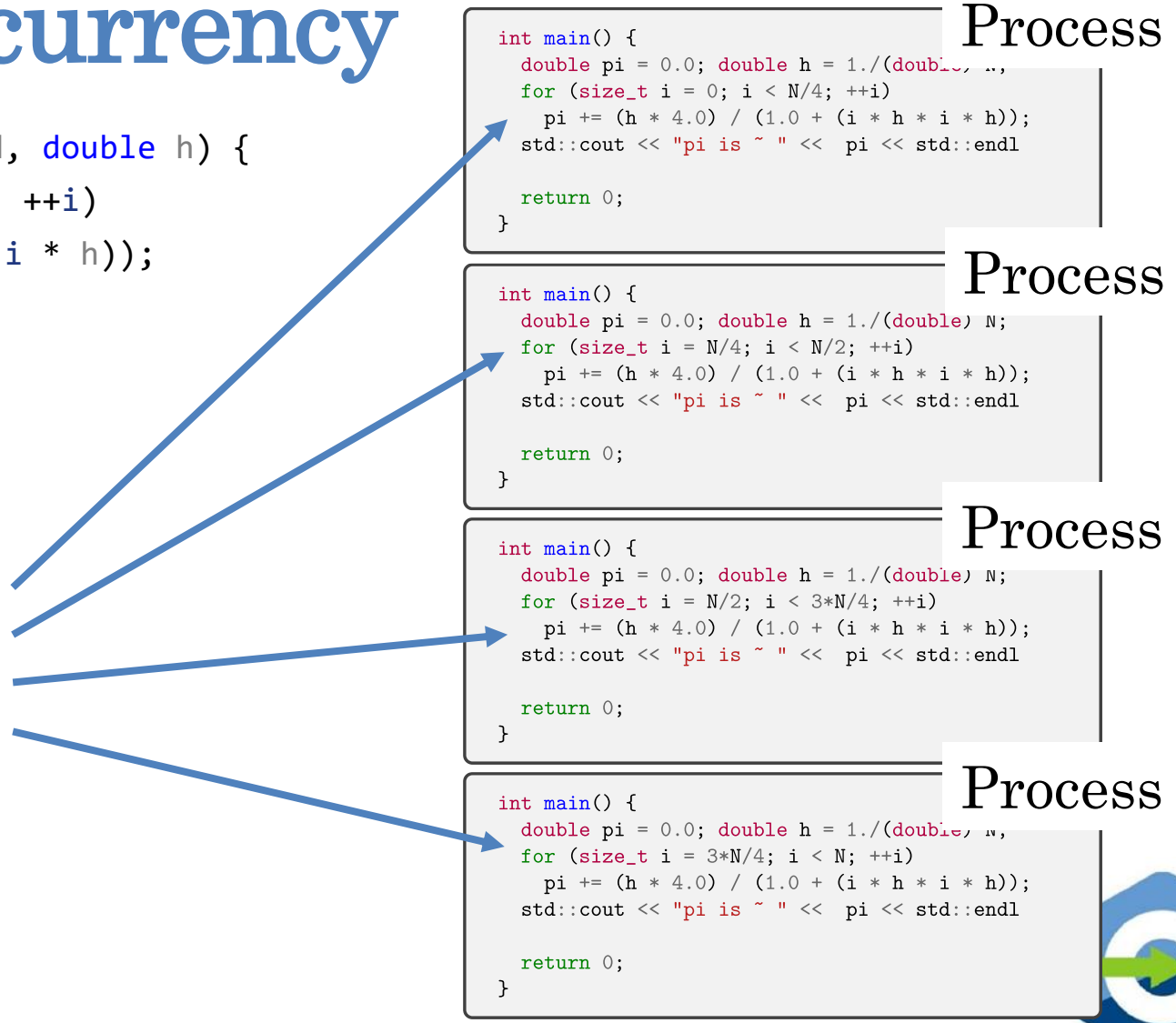
Finding Concurrency

```
void pi_helper(int begin, int end, double h) {
    for (int i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));
}

int main() {
    int const N = 1'000'000;
    double h = 1.0 / 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.joint(); t1.join();
    t2.join(); t3.join();

    std::println("pi: {}", pi);
}
```



Finding Concurrency

```
void pi_helper(int begin, int end, double h) {
    for (int i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));
}

int main() {
    int const N = 1'000'000;
    double h = 1.0 / N;




    std::println("pi: {}", pi);
}
```



Communicating sequential Processes / SPMD

```cpp
int main(int argc, char* argv[]) {
    double h = 1.0 / N;
    double pi = 0.0;
    for (long i = 0; i != N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    std::println("pi: {}", pi);
}
```

```cpp
int main(int argc, char* argv[]) {
    double h = 1.0 / N;
    double pi = 0.0;
    for (long i = N/4; i != N/2; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    std::println("pi: {}", pi);
}
```

```cpp
int main(int argc, char* argv[]) {
    double h = 1.0 / N;
    double pi = 0.0;
    for (long i = N/2; i != 3*N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    std::println("pi: {}", pi);
}
```

```cpp
int main(int argc, char* argv[]) {
    double h = 1.0 / N;
    double pi = 0.0;
    for (long i = 3*N/4; i != N / 4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    std::println("pi: {}", pi);
}
```

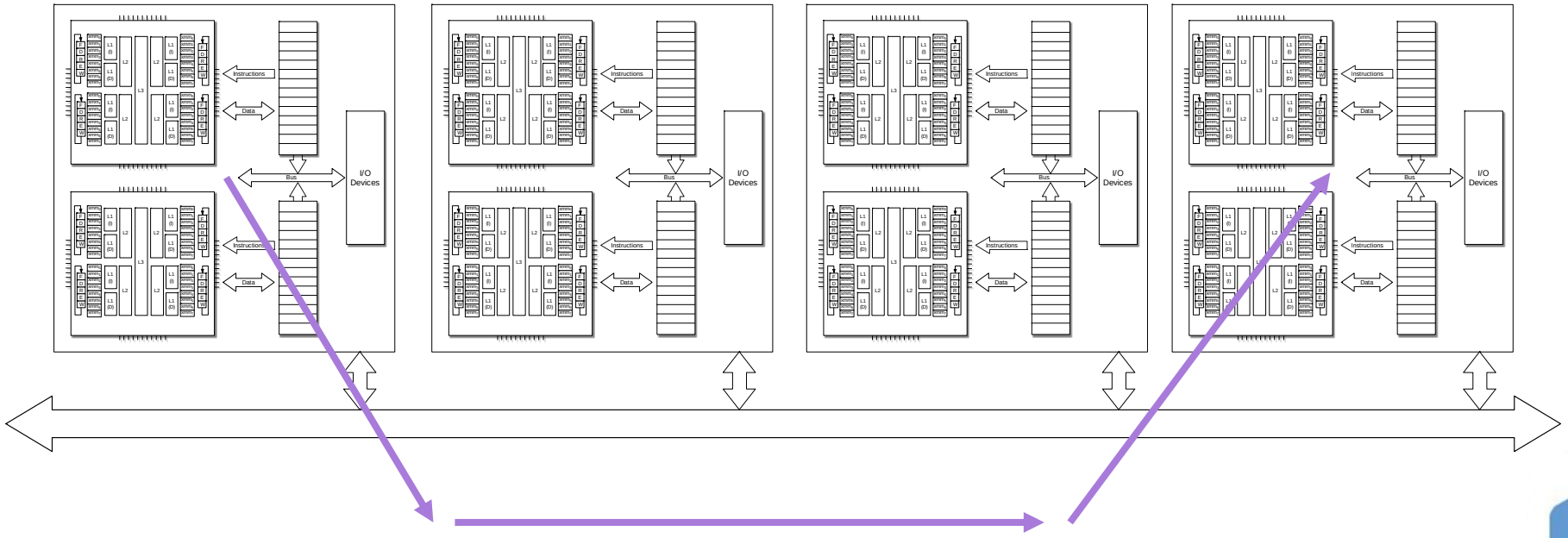
Communicating sequential processes / SPMD

```cpp
int main(int argc, char* argv[]) {
    double h = 1.0 / N;
    double pi = 0.0;
    for (long i = 0; i != N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    std::println("pi: {}", pi);
}
```

```cpp
int main(int argc, char* argv[]) {
    double h = 1.0 / N;
    double pi = 0.0;
    for (long i = N/4; i != N/2; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    std::println("pi: {}", pi);
}
```

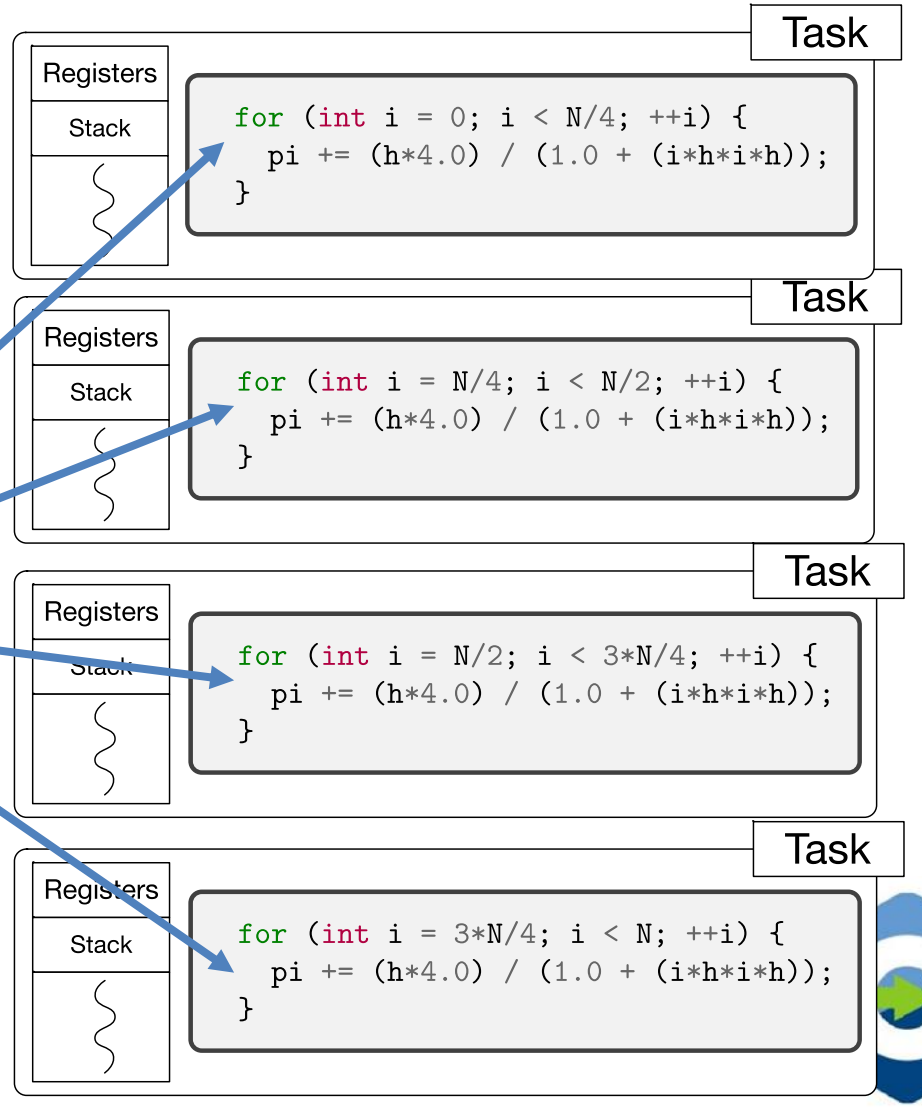
```cpp
int main(int argc, char* argv[]) {
    double h = 1.0 / N;
    double pi = 0.0;
    for (long i = N/2; i != 3*N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    std::println("pi: {}", pi);
}
```

```cpp
int main(int argc, char* argv[]) {
    double h = 1.0 / N;
    double pi = 0.0;
    for (long i = 3*N/4; i != N / 4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    std::println("pi: {}", pi);
}
```



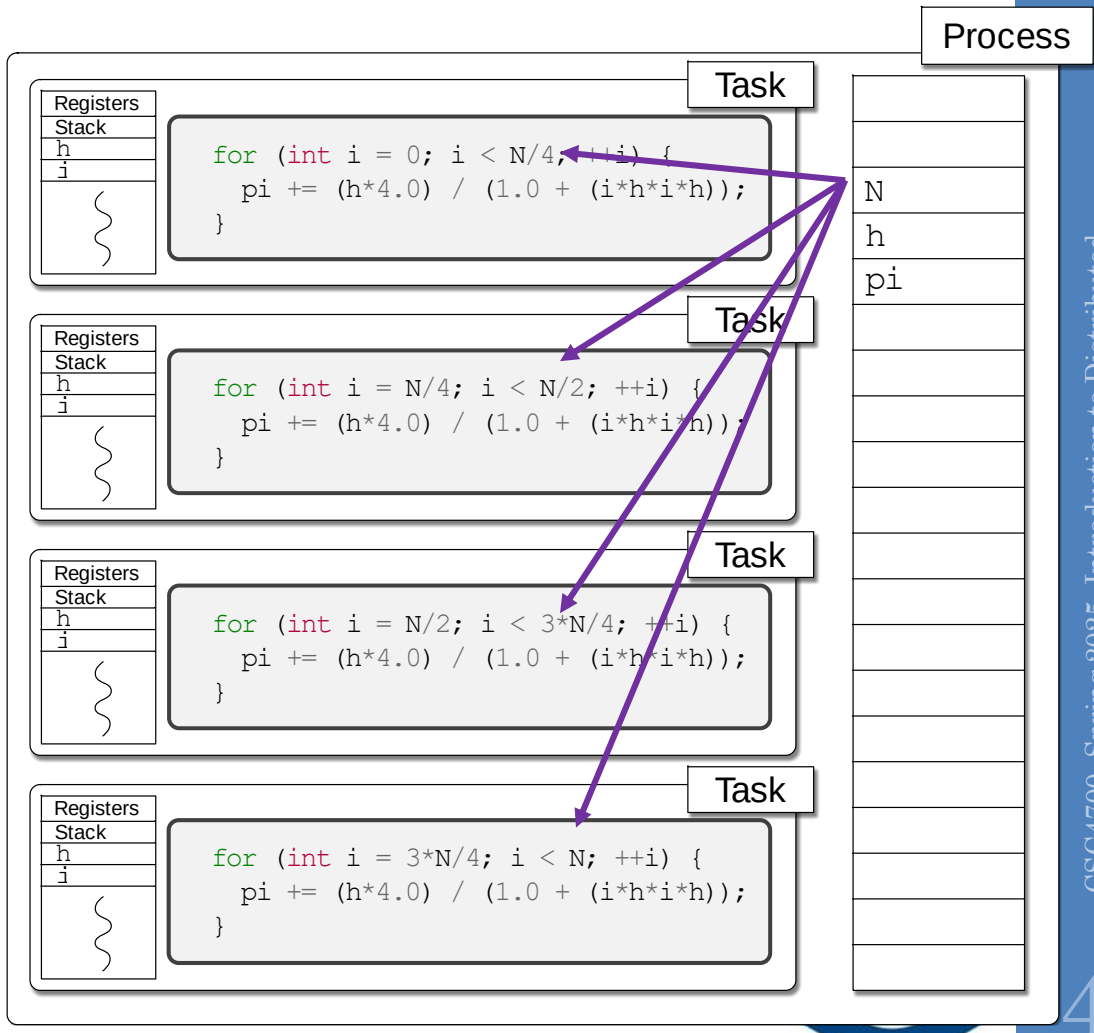
Threads

```
void pi_helper(int begin, int end, double h) {
    for (int i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));
}

int main() {
    int const N = 1'000'000;
    double h = 1.0 / 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.joint(); t1.join();
    t2.join(); t3.join();

    std::println("pi: {}", pi);
}
```



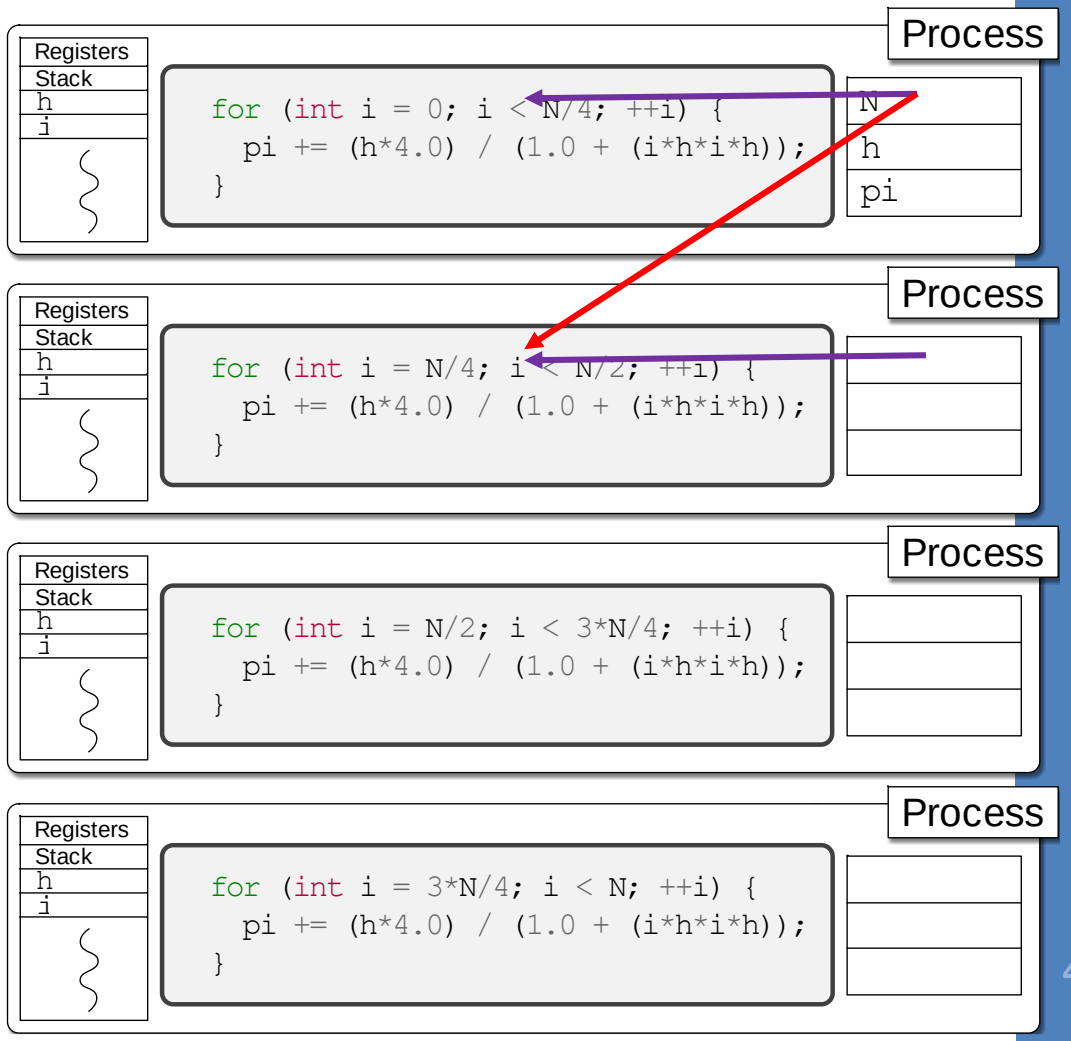
Shared Memory Parallelism

- `N` is local to this process
- All threads read the same value `N`
- `N` is exactly equivalent to the sequential `N`
- Because `N` is at the same memory location for all
  - Race conditions aside, there is no additional work needed to make this happen
- Same is true for `h` and `pi`



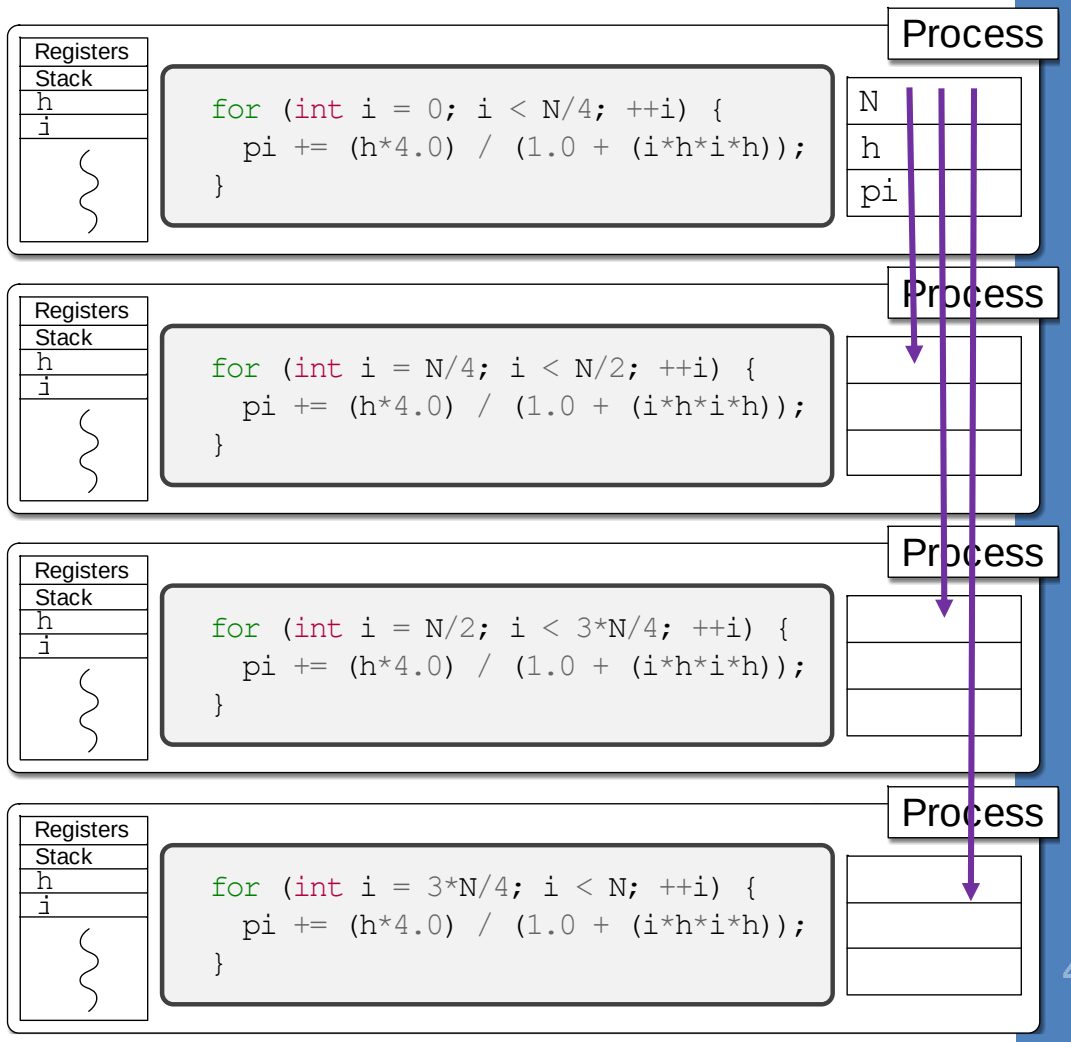
Distributed Memory Parallelism

- Every `N` is local to one process
- While all programs are identical they need to use the same value for `N`
- No process can directly read (or write) the `N` from another process
- How do we get the right value for `N` to all processes?



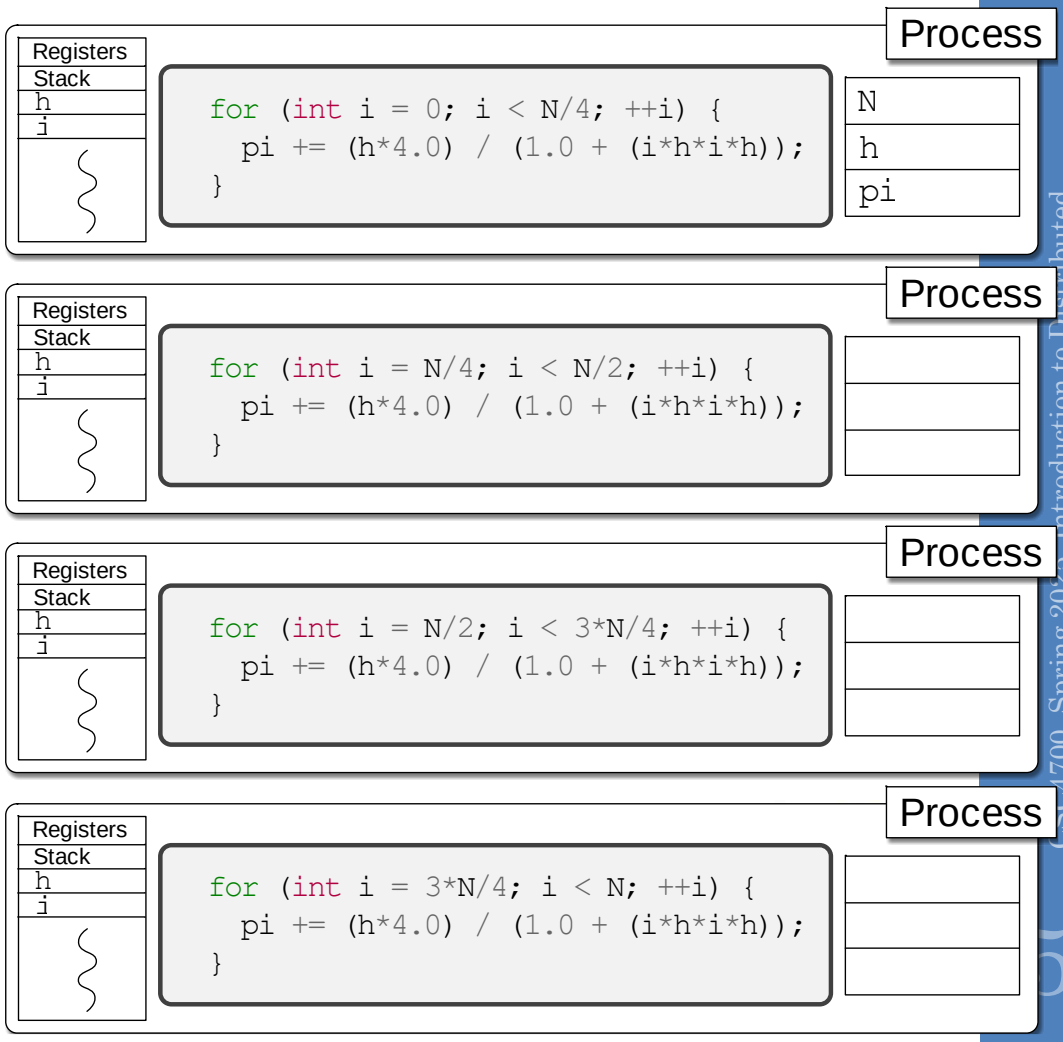
Distributed Memory Parallelism

- For each process to read the "right" N
- We need to copy it to all of them



Distributed Memory Parallelism

- Threads have the same value for N
  - The overall result is exactly equivalent to the sequential
- Because processes are reading copies of N
  - It is as if they were the same N
  - Similarly for `h` and `pi`
- Must make N consistent across processes when writing to maintain as if



SPMD? Single Program Multiple Data?

```
int main(int argc, char* argv[])
{
    double h = 1.0 / N;
    double pi = 0.0;
    for (long i = 0; i != N / 4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));
    std::println("pi: {}", pi);
}
```

```
int main(int argc, char* argv[])
{
    double h = 1.0 / N;
    double pi = 0.0;
    for (long i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));
    std::println("pi: {}", pi);
}
```

- Single Program Multiple Data?
- Only if each process has a different begin, end

SPMD? Single Program Multiple Data?

- The code for each process is the same, but
  - We need the same N in every process
  - We need different begin and end in each of the processes

- How do we achieve this?

Single Program Multiple Data

```
int main(int argc, char* argv[])
{
    long N = 1'000'000'000;

    if (argc > 1)
        N = std::stol(argv[1]);

    double h = 1.0 / N;
    double pi = 0.0;
    for (long i = 0; i != N / 4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    std::println("pi: {}", pi);
}
```

- We can get N from the command line
- But from every node?
  - That's a lot of typing
- Better to get it at just one node and send it around
- But we want to have a single program
  - So we let every process attempt to get N from the command line
  - And send N from one of the processes to all others

Single Program Multiple Data

- How do we get the same program to do different things?
- While keeping them the same?
  - Hint: multiple data
  - Likely use an `if` statement
- So we need means to distinguish the process that executes the code

- Let's say "the world has `P` processes"
- And each process knows the value of its own id
- And each process has an id in the range `[0, P)`

Single HPX (SPMD) Program

```
int main(int argc, char* argv[]) {
    long N = 1'000'000'000;
    std::uint32_t num_localities = hpx::get_num_localities(hpx::launch::sync);
    std::uint32_t locality_id = hpx::get_locality_id();

    if (locality_id == 0 && argc > 1) N = std::stol(argv[1]);

    // send ('broadcast') N to all nodes if locality_id == 0

    std::size_t blocksize = N / num_localities;
    std::size_t begin = blocksize * locality_id, end = blocksize * (locality_id + 1);
    double h = 1.0 / N, pi = 0.0;

    for (long i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    // collect ('reduce') results from all localities

    if (locality_id == 0)
        std::println("pi: {}", pi);
}
```

A better Name than MIMD or SPMD

- Andrew Lumsdaine

- Distinguished replicated processes, distributed data (DRPDD)
  - Pronounced “drop dee”

HPX Collective Operations

- We assume that process with `locality_id == 0` receives `N` from the command line
- This process sends `N` to all others
  - This is an operation that is called `broadcast`

```
hpx::collectives::broadcast(
    hpx::collectives::get_world_communicator(), N);
```

- Every process calculates its own `begin` and `end` based on its `locality_id`
- Every process now computes part of the overall solution
- In the end, every process needs to provide its partial result, all of which need to be consolidated
  - This is an operation that is called `reduce`

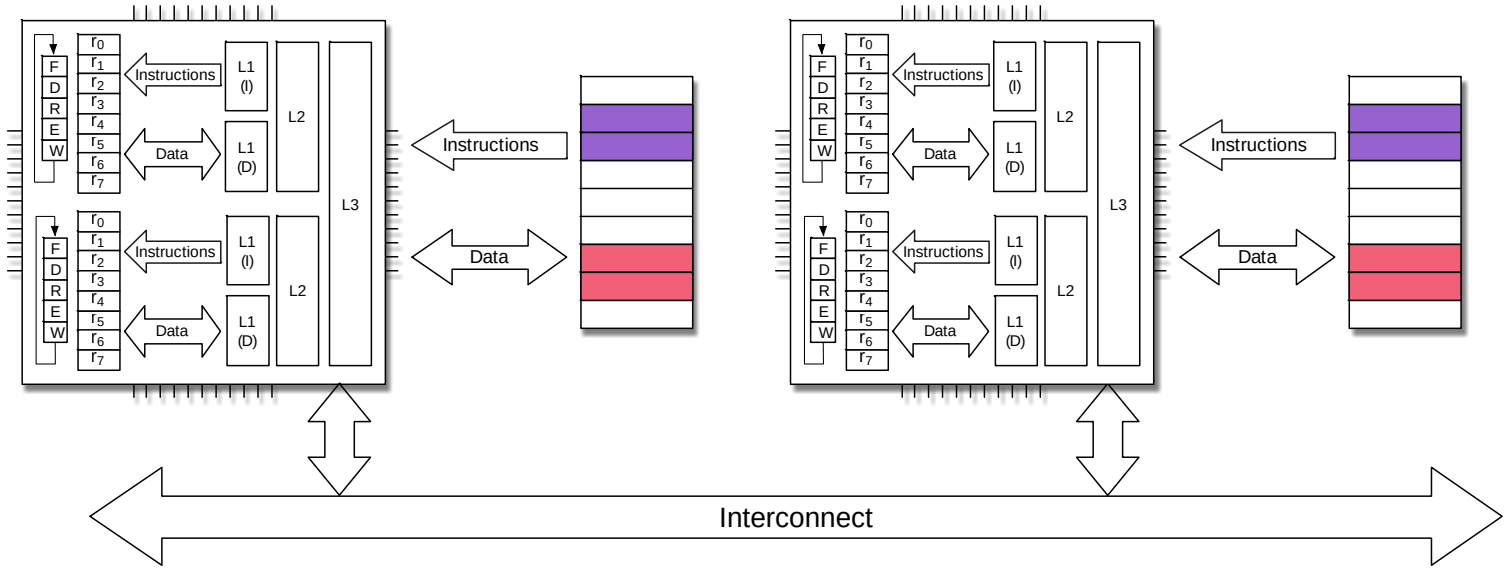
```
hpx::collectives::reduce(
    hpx::collectives::get_world_communicator(),
    pi, std::plus{});
```

Single Program Multiple Data

```
int main(int argc, char* argv[]) {
    long N = 1'000'000'000;
    std::uint32_t num_localities = hpx::get_num_localities(hpx::launch::sync);
    std::uint32_t locality_id = hpx::get_locality_id();

    if (locality_id == 0 && argc > 1) N = std::stol(argv[1]);

    hpx::collectives::broadcast(hpx::collectives::get_world_communicator(), N);

    std::size_t blocksize = N / num_localities;
    std::size_t begin = blocksize * locality_id, end = blocksize * (locality_id + 1);
    double h = 1.0 / N, pi = 0.0;

    for (long i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    hpx::collectives::reduce(hpx::collectives::get_world_communicator(), pi, std::plus{});

    if (locality_id == 0)
        std::println("pi: {}", pi);
}
```

Distributed Memory

- Every process:
  - Independent memory space
  - Code is replicated
- Data are partitioned
  - The union of the partitions should be the whole problem



Finding Concurrency (Decomposition)

```cpp
int main()
{
    int const N = 1'000'000;
    double pi = 0;

    for (int i = 0; i != N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = N/4; i != N/2; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = N/2; i != 3*N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = 3*N/4; i != N; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    std::println("pi: {}", pi);
}
```

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

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

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

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

Threads to Processes

```
void pi_helper(int begin, int end, double h, double
    for (int i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));
}

int main() {
    int const N = 1'000'000;
    double h = 1.0 / N;

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

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

    std::println("pi: {}", pi);
}
```

Process

```
int main(int argc, char* argv[])
{
    double pi = 0.0; double h = 1.0 / N;
    for (long i = 0; i != N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));
    std::println("pi: {}", pi);
}
```

Process

```
int main(int argc, char* argv[])
{
    double pi = 0.0; double h = 1.0 / N;
    for (long i = N/4; i != N/2; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));
    std::println("pi: {}", pi);
}
```

Process

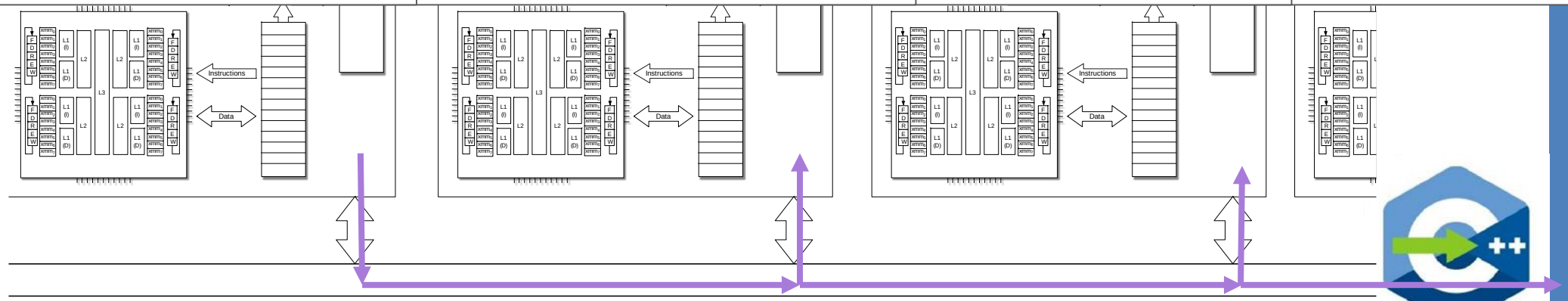
```
int main(int argc, char* argv[])
{
    double pi = 0.0; double h = 1.0 / N;
    for (long i = N/2; i != 3*N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));
    std::println("pi: {}", pi);
}
```

Process

```
int main(int argc, char* argv[])
{
    double pi = 0.0; double h = 1.0 / N;
    for (long i = 3*N/4; i != N; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));
    std::println("pi: {}", pi);
}
```

Distinguished Replicated Processes

```
int main(int argc, char* argv[])
{
    long N = 1'000'000'000;
    std::uint32_t num_localities =
        hpx::get_num_localities(hpx::launch::sync);
    std::uint32_t locality = hpx::get_locality_id();

    if (locality == 0 && argc > 1) N = stol(argv[1]);

    std::size_t blocksize = N / num_localities;
    std::size_t begin = blocksize * locality_id;
    std::size_t end = blocksize * (locality_id + 1);
    double h = 1.0 / N;

    double pi = 0.0;
    for (long i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    if (locality == 0) println("pi: {}", pi);
}
```



‘DropDee’ using HPX

```
int main(int argc, char* argv[]) {
    long N = 1'000'000'000;
    std::uint32_t num_localities =
        hpx::get_num_localities(hpx::launch::sync);
    std::uint32_t locality = hpx::get_locality_id();

    if (locality == 0 && argc > 1) N = stol(argv[1]);

    hpx::collectives::broadcast(
        hpx::collectives::get_world_communicator(), N)

    std::size_t blocksize = N / num_localities;
    std::size_t begin = blocksize * locality_id,
    std::size_t begin = end = blocksize * (locality_id + 1);
    double h = 1.0 / N, pi = 0.0;
    for (long i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    hpx::collectives::reduce(
        hpx::collectives::get_world_communicator(), pi);

    if (locality == 0) println("pi: {}", pi);
}
```

- Get our locality id and number of other nodes
- locality 0 gets N, shares N
- Everyone computes their own partial result
- locality 0 collects all partials, adds them, and prints

This pattern is ubiquitous

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP