Lecture 4:

Parallelizing Code: The Programming Thought Process

Parallel Computing
Stanford CS149, Fall 2025

Today's topics: case study on writing an optimizing a parallel program

- More in ISPC semantics (finishing off lecture 3 material)
  - Key focus: abstraction vs. implementation

- Case study on thought process of writing and optimizing a parallel program
  - Demonstrated in two programming models
    - data parallel
    - shared address space

Last time: our sinx() example in ISPC

C++ code: `main.cpp`

```
#include “sinx_ispc.h”

int main(int argc, void** argv) {
  int N = 1024;
  int terms = 5;
  float* x = new float[N];
  float* result = new float[N];

  // initialize x here

  // execute ISPC code
  ispc_sinx(N, terms, x, result);
  return 0;
}
```

ISPC code: `sinx.ispc`

```
export void ispc_sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    // assume N % programCount = 0
    for (uniform int i=0; i<N; i+=programCount)
    {
        int idx = i + programIndex;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6;  // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```

SPMD programming abstraction:

Call to ISPC function spawns “gang” of ISPC “program instances”

All instances run ISPC code concurrently

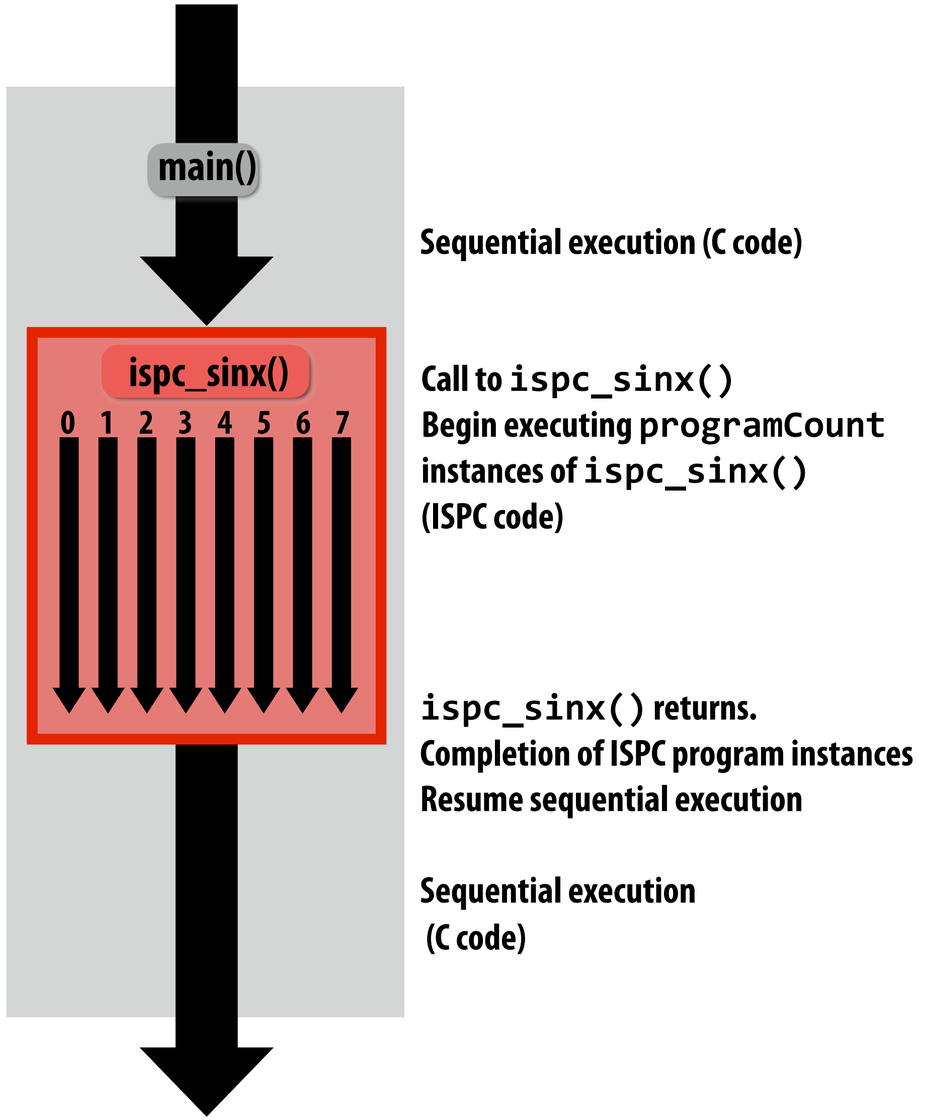
Each instance has its own copy of local variables (blue variables in code, we’ll talk about “uniform” later)

Upon return, all instances have completed

Invoking sinx() in ISPC

C++ code: `main.cpp`

```
#include “sinx_ispc.h”

int main(int argc, void** argv) {
  int N = 1024;
  int terms = 5;
  float* x = new float[N];
  float* result = new float[N];

  // initialize x here

  // execute ISPC code
  ispc_sinx(N, terms, x, result);
  return 0;
}
```



SPMD programming abstraction:

Call to ISPC function spawns “gang” of ISPC “program instances”

All instances run ISPC code concurrently

Each instance has its own copy of local variables

Upon return, all instances have completed

In this illustration `programCount` = 8

sinx() in ISPC

"Interleaved" assignment of array elements to program instances

C++ code: `main.cpp`

```
#include "sinx_ispc.h"

int main(int argc, void** argv) {
  int N = 1024;
  int terms = 5;
  float* x = new float[N];
  float* result = new float[N];

  // initialize x here

  // execute ISPC code
  ispc_sinx(N, terms, x, result);
  return 0;
}
```

ISPC code: `sinx.ispc`

```
export void ispc_sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    // assumes N % programCount = 0
    for (uniform int i=0; i<N; i+=programCount)
    {
        int idx = i + programIndex;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6;  // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```

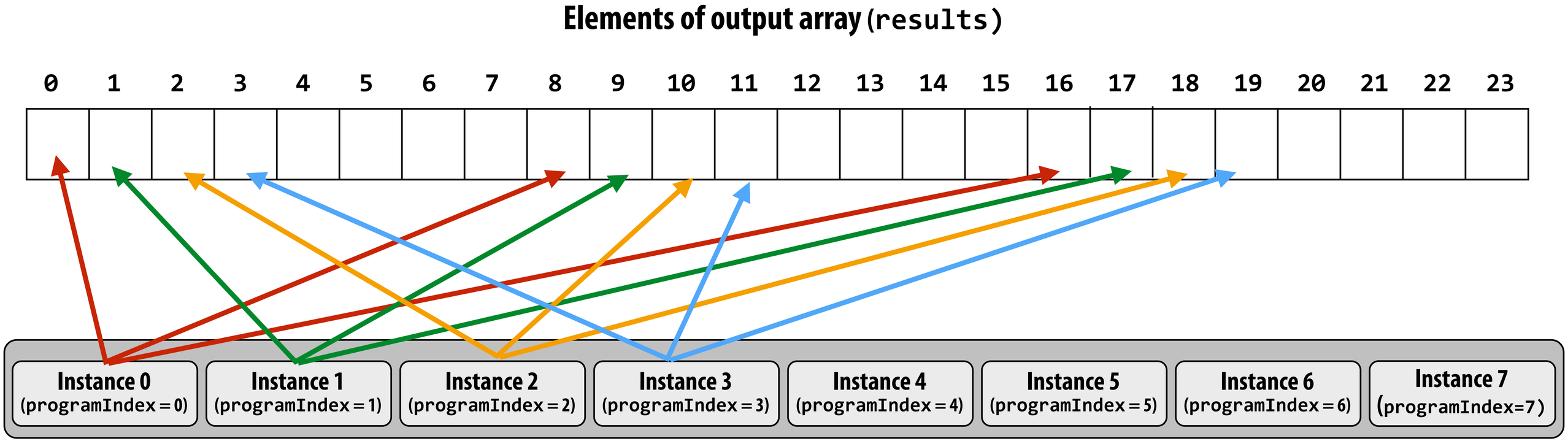
ISPC language keywords:

`programCount`: number of simultaneously executing instances in the gang (uniform value)

`programIndex`: id of the current instance in the gang. (a non-uniform value: "varying")

`uniform`: A type modifier. All instances have the same value for this variable. Its use is purely an optimization. Not needed for correctness.

Interleaved assignment of program instances to loop iterations



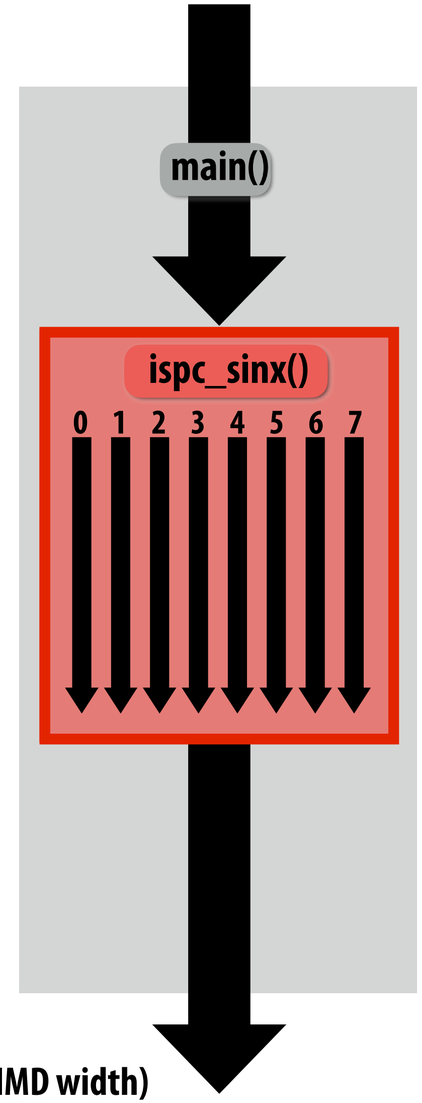
"Gang" of ISPC program instances

In this illustration: gang contains eight instances: `programCount` = 8

ISPC implements the gang abstraction using SIMD instructions

C++ code: `main.cpp`

```
#include “sinx_ispc.h”

int main(int argc, void** argv) {
  int N = 1024;
  int terms = 5;
  float* x = new float[N];
  float* result = new float[N];

  // initialize x here

  // execute ISPC code
  ispc_sinx(N, terms, x, result);
  return 0;
}
```



Sequential execution (C code)

Call to `ispc_sinx()`
Begin executing `programCount` instances of `ispc_sinx()` (ISPC code)

`ispc_sinx()` returns.
Completion of ISPC program instances
Resume sequential execution

Sequential execution (C code)

SPMD programming abstraction:

Call to ISPC function spawns “gang” of ISPC “program instances”
All instances run ISPC code simultaneously
Upon return, all instances have completed

ISPC compiler generates SIMD implementation:

Number of instances in a gang is the SIMD width of the hardware (or a small multiple of SIMD width)
ISPC compiler generates a C++ function binary (.o) whose body contains SIMD instructions
C++ code links against generated object file as usual

sinx() in ISPC: version 2

"Blocked" assignment of array elements to program instances

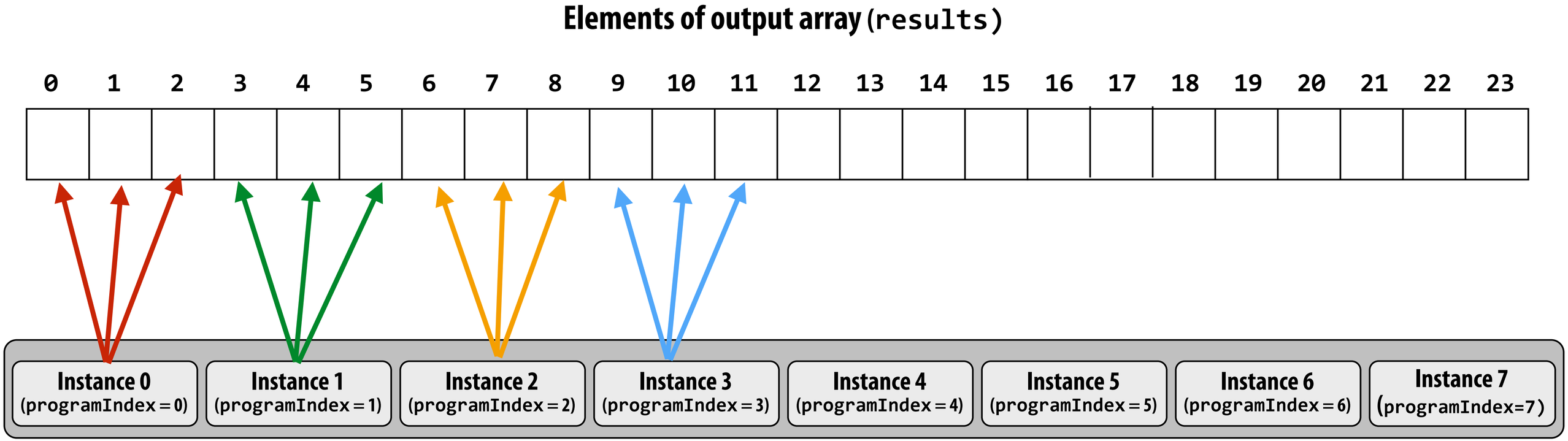
C++ code: `main.cpp`

```
#include "sinx_ispc.h"

int main(int argc, void** argv) {
  int N = 1024;
  int terms = 5;
  float* x = new float[N];
  float* result = new float[N];

  // initialize x here

  // execute ISPC code
  ispc_sinx_v2(N, terms, x, result);
  return 0;
}
```

ISPC code: `sinx.ispc`

```
export void ispc_sinx_v2(
   uniform int N,
   uniform int terms,
   uniform float* x,
   uniform float* result)
{
   // assume N % programCount = 0
   uniform int count = N / programCount;
   int start = programIndex * count;
   for (uniform int i=0; i<count; i++)
   {
      int idx = start + i;
      float value = x[idx];
      float numer = x[idx] * x[idx] * x[idx];
      uniform int denom = 6;  // 3!
      uniform int sign = -1;

      for (uniform int j=1; j<=terms; j++)
      {
         value += sign * numer / denom
         numer *= x[idx] * x[idx];
         denom *= (j+3) * (j+4);
         sign *= -1;
      }
      result[idx] = value;
   }
}
```

Blocked assignment of program instances to loop iterations



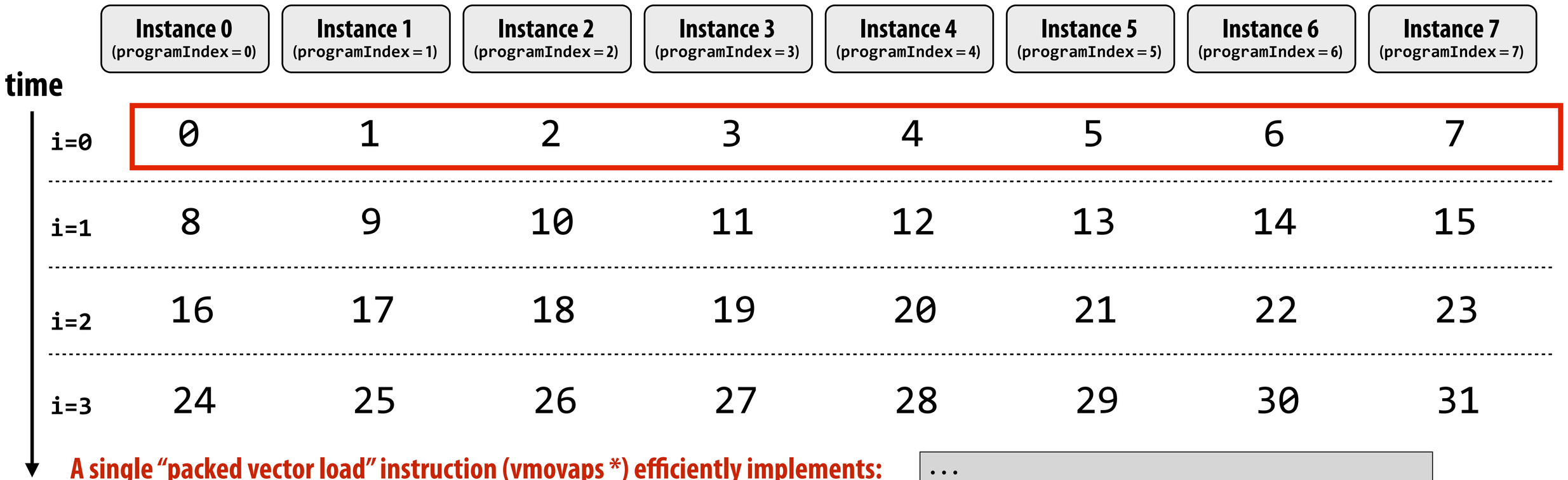
"Gang" of ISPC program instances

In this illustration: gang contains eight instances: `programCount` = 8

Schedule: interleaved assignment

"Gang" of ISPC program instances

Gang contains four instances: `programCount` = 8



A single "packed vector load" instruction (vmovaps *) efficiently implements:

```
float value = x[idx];
```

for all program instances, since the eight values are contiguous in memory

```
...
// assumes N % programCount = 0
for (uniform int i=0; i<N; i+=programCount)
   {
      int idx = i + programIndex;
      float value = x[idx];
...
```

* see _mm256_load_ps() intrinsic function

Schedule: blocked assignment

"Gang" of ISPC program instances

Gang contains four instances: `programCount` = 8

time	Instance 0 (programIndex = 0)	Instance 1 (programIndex = 1)	Instance 2 (programIndex = 2)	Instance 3 (programIndex = 3)	Instance 4 (programIndex = 4)	Instance 5 (programIndex = 5)	Instance 6 (programIndex = 6)	Instance 7 (programIndex = 7)
i=0	0	8	16	24	32	40	48	56
i=1	1	9	17	25	33	41	49	57
i=2	2	10	18	26	34	42	50	58
i=3	3	11	19	27	35	43	51	59

```
float value = x[idx];
```

For all program instances now touches eight non-contiguous values in memory. Need "gather" instruction (vgatherdps *) to implement (gather is a more complex, and more costly SIMD instruction…)

```
uniform int count = N / programCount;
int start = programIndex * count;
for (uniform int i=0; i<count; i++) {
   int idx = start + i;
   float value = x[idx];
...
```

* see _mm256_i32gather_ps() intrinsic function

Raising level of abstraction with foreach

C++ code: `main.cpp`

```
#include “sinx_ispc.h”

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```

ISPC code: `sinx.ispc`

```
export void ispc_sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    foreach (i = 0 ... N)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        uniform int denom = 6;  // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[i] = value;
    }
}
```

`foreach`: key ISPC language construct

- `foreach` declares parallel loop iterations
  - Programmer says: these are the iterations the entire gang (not each instance) must perform
- ISPC implementation takes responsibility for assigning iterations to program instances in the gang

How might foreach be implemented?

Code written using `foreach` abstraction:

```
foreach (i = 0 ... N)
{
   // do work for iteration i here...
}
```

Implementation 1: program instance 0 executes all iterations

```
if (programCount == 0) {
   for (int i=0; i<N; i++) {
     // do work for iteration i here…
  }
}
```

Implementation 2: interleave iterations onto program instances

```
// assume N % programCount = 0
for (uniform int loop_i=0; loop_i<N; loop_i+=programCount)
{
  int i = loop_i + programIndex;
  // do work for iteration i here...
}
```

Implementation 3: block iterations onto program instances

```
// assume N % programCount = 0
uniform int count = N / programCount;
int start = programIndex * count;
for (uniform int loop_i=0; loop_i<count; loop_i++)
{
    int i = start + loop_i;
    // do work for iteration i here...
}
```

Implementation 4: dynamic assignment of iterations to instances

```
uniform int nextIter;
if (programCount == 0)
  nextIter = 0;

int i = atomic_add_local(&nextIter, 1);
while (i < N) {

  // do work for iteration i here...

  i = atomic_add_local(&nextIter, 1);
}
```

Thinking about iterations, not parallel execution

In many simple cases, using `foreach` allows the programmer to express their program almost as if it was a sequential program

```
export void ispc_function(
    uniform int    N,
    uniform float* x,
    uniform float* y)
{
    foreach (i = 0 ... N)
    {
      float val = x[I];
      float result;

      // do work here to compute
      // result from val

      y[i] = result;
    }
}
```

What does this program do?

```
// main C++ code:
const int N = 1024;
float* x = new float[N/2];
float* y = new float[N];

// initialize N/2 elements of x here

// call ISPC function
absolute_repeat(N/2, x, y);
```

```
// ISPC code:
export void absolute_repeat(
    uniform int N,
    uniform float* x,
    uniform float* y)
{
    foreach (i = 0 ... N)
    {
        if (x[i] < 0)
            y[2*i] = -x[i];
        else
            y[2*i] = x[i];
        y[2*i+1] = y[2*i];
    }
}
```

This ISPC program computes the absolute value of elements of x, then repeats it twice in the output array y

What does this program do?

```
// main C++ code:
const int N = 1024;
float* x = new float[N];
float* y = new float[N];

// initialize N elements of x

// call ISPC function
shift_negative(N, x, y);
```

```
// ISPC code:
export void shift_negative(
    uniform int N,
    uniform float* x,
    uniform float* y)
{
    foreach (i = 0 ... N)
    {
        if (i >= 1 && x[i] < 0)
          y[i-1] = x[i];
        else
          y[i] = x[i];
    }
}
```

The output of this program is undefined!

Possible for multiple iterations of the loop body to write to same memory location

Computing the sum of all elements in an array (incorrectly)

What's the error in this program?

```
export uniform float sum_incorrect_1(
    uniform int N,
    uniform float* x)
{
    float sum = 0.0f;
    foreach (i = 0 ... N)
    {
        sum += x[i];
    }

    return sum;
}
```

`sum` is of type `float`
(different variable for all program instances)

Cannot return many copies of a varianble to the calling C code, which expects one return value of type float
Result: compile-time type error

What's the error in this program?

```
export uniform float sum_incorrect_2(
    uniform int N,
    uniform float* x)
{
    uniform float sum = 0.0f;
    foreach (i = 0 ... N)
    {
        sum += x[i];
    }

    return sum;
}
```

`sum` is of type `uniform float`
(one copy of variable for all program instances)

`x[i]` has a different value for each program instance
So what gets copied into sum?
Result: compile-time type error

Computing the sum of all elements in an array (correctly)

```
export uniform float sum_array(
    uniform int N,
    uniform float* x)
{
    uniform float sum;
    float partial = 0.0f;
    foreach (i = 0 ... N)
    {
        partial += x[i];
    }

    // reduce_add() is part of ISPC's cross
    // program instance standard library
    sum = reduce_add(partial);

    return sum;
}
```

Each instance accumulates a private partial sum (no communication)

Partial sums are added together using the `reduce_add()` cross-instance communication primitive. The result is the same total sum for all program instances (`reduce_add()` returns a uniform float)

The ISPC code at left will execute in a manner similar to the C code with AVX intrinsics implemented below. *

```
float sum_summary_AVX(int N, float* x) {

  float tmp[8];  // assume 16-byte alignment
  __mm256 partial = _mm256_broadcast_ss(0.0f);

  for (int i=0; i<N; i+=8)
    partial = _mm256_add_ps(partial, _mm256_load_ps(&x[i]));

  _mm256_store_ps(tmp, partial);

  float sum = 0.f;
  for (int i=0; i<8; i++)
    sum += tmp[i];

  return sum;
}
```

* Self-test: If you understand why this implementation correctly implements the semantics of the ISPC gang abstraction, then you've got a good command of ISPC

ISPC's cross program instance operations

Compute sum of a variable's value in all program instances in a gang:

```
uniform int64 reduce_add(int32 x);
```

Compute the min of all values in a gang:

```
uniform int32 reduce_min(int32 a);
```

Broadcast a value from one instance to all instances in a gang:

```
int32 broadcast(int32 value, uniform int index);
```

For all `i`, pass value from instance `i` to the instance `i+offset % programCount`:

```
int32 rotate(int32 value, uniform int offset);
```

ISPC: abstraction vs. implementation

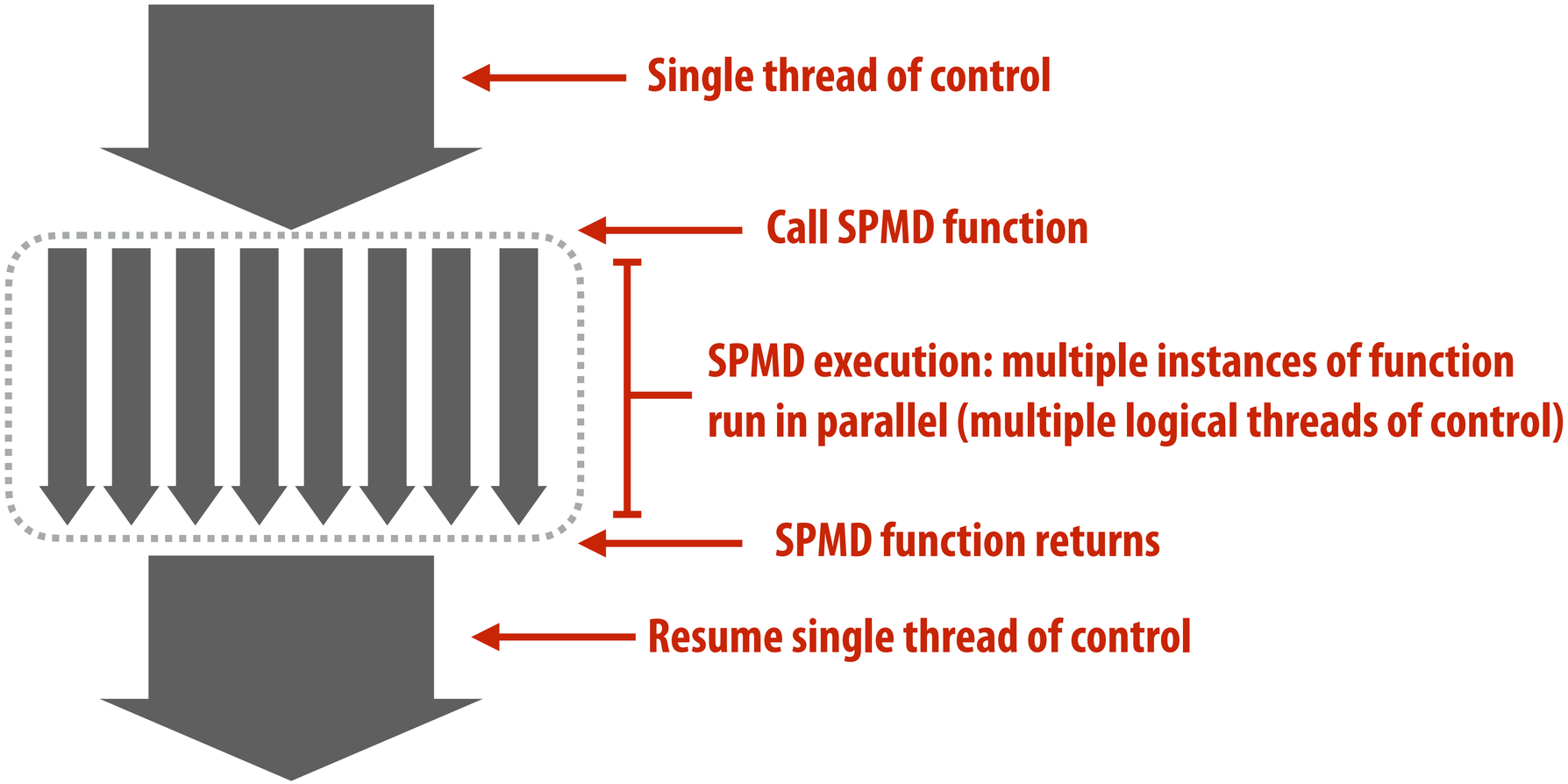
- **Single program, multiple data (SPMD) programming model**
  - Programmer "thinks": running a gang is spawning `programCount` logical instruction streams (each with a different value of `programIndex`)
  - This is the programming abstraction
  - Program is written in terms of this abstraction

- **Single instruction, multiple data (SIMD) implementation**
  - ISPC compiler emits vector instructions (e.g., AVX2, ARM NEON) that carry out the logic performed by a ISPC gang
  - ISPC compiler handles mapping of conditional control flow to vector instructions (by masking vector lanes, etc. like you do manually in assignment 1)

- **Semantics of ISPC can be tricky**
  - SPMD abstraction + uniform values
    (allows implementation details to peek through abstraction a bit)

SPMD programming model summary

- SPMD = "single program, multiple data"
- Define one function, run multiple instances of that function in parallel on different input arguments



ISPC tasks

- The ISPC gang abstraction is implemented by SIMD instructions that execute within on thread running on one x86 core of a CPU.

- So all the code I've shown you in the previous slides would have executed on only one of the four cores of the myth machines.

- ISPC contains another abstraction: a "task" that is used to achieve multi-core execution. I'll let you read up about that as you do assignment 1.

Thinking about operating on data in parallel?

- In many simple cases, using ISPC foreach allows the programmer to express their program almost as if it was a sequential program
  - Almost want to explain code as: "independently, for each element in the input array… do this…"

- Exceptions:
  - Uniform variables
  - Cross-instance operations (in standard library, like reduceAdd)

- But ISPC is a low-level programming language: by exposing programIndex and programCount, it allows programmer to define what work each program instance does and what data each instance accesses
  - Can implement programs with undefined output
  - Can implement programs that are correct only for a specific programCount

```
export void ispc_function(
    uniform int    N,
    uniform float* x,
    uniform float* y)
{
    foreach (i = 0 ... N)
    {
      float val = x[i];
      float result;

      // do work here to compute
      // result from val

      y[i] = result;
    }
}
```

But can express very advanced cooperation

Here's a program that computes the product of all elements of an array in lg(8) = 3 steps

```
// compute the product of all eight elements in the
// input array. Assumes the gang size is 8.
export void vec8product(
   uniform float* x,
   uniform float* result)
{
   float val1  = x[programIndex];
   float val2 = shift(val1, 1);

   if (programIndex % 2 == 0)
     val1 = val1 * val2;

   val2 = shift(val1, 2);
   if (programIndex % 4 == 0)
      val1 = val1 * val2;
   }

   val2 = shift(val1, 4);
   if (programIndex % 8 == 0) {
      *result = val1 * val2
   }
}
```

But what if ISPC was not trying to be a low-level language?

- Example: change language so there is no access to programIndex, programCount
- Expect programmer to just use foreach

- Now there's very little need to think about program instances at all.
  - Everything outside a foreach must be uniform values and uniform logic. Why?

```
export void ispc_function(
    int    N,
    float* x,
    float* y)
{

    int twoN = 2 * N;

    foreach (i = 0 ... twoN)
    {
      float val = x[i];
      float result;

      // do work here to compute
      // result from val

      y[i] = result;
    }
}
```

Another alternative

- Don't even allow array indexing!
- Invoke computation once per element of a "collection" data structure
- Programmer writes no loops, performs no data indexing

```
float dowork(float x) {
  // do work here to compute
  // result from x
}

Collection x;  // data structure of N

// invoke doWork for all elements of x,
// placing results in collection y
Collection y = map(doWork, x, y);
```

- This model should be very family to NumPy, PyTorch, etc. programmers, right?

```
import numpy as np

def addOne(i):
    return i+1
mapAddOne = np.vectorize(addOne);

X = np.arange(15) # create numPy array [0, 1, 2, 3, ...]
Y = np.arange(15) # create numPy array [0, 1, 2, 3, ...]

Z = X + Y;               # Z = [0, 2, 4, 6, … ]
Zplus1 = mapAddOne(Z);   # Zplus1 = [1, 3, 5, 7, …]
```

- Much more on this to come

Summary

- Programming models provide a way to think about the organization of parallel programs.

- They provide abstractions that permit multiple valid implementations.

- *I want you to always be thinking about abstraction vs. implementation for the remainder of this course.*

Thought process of writing and optimizing a parallel program

Creating a parallel program

- Your thought process:
  1. Identify work that can be performed in parallel
  2. Partition work (and also data associated with the work)
  3. Manage data access, communication, and synchronization

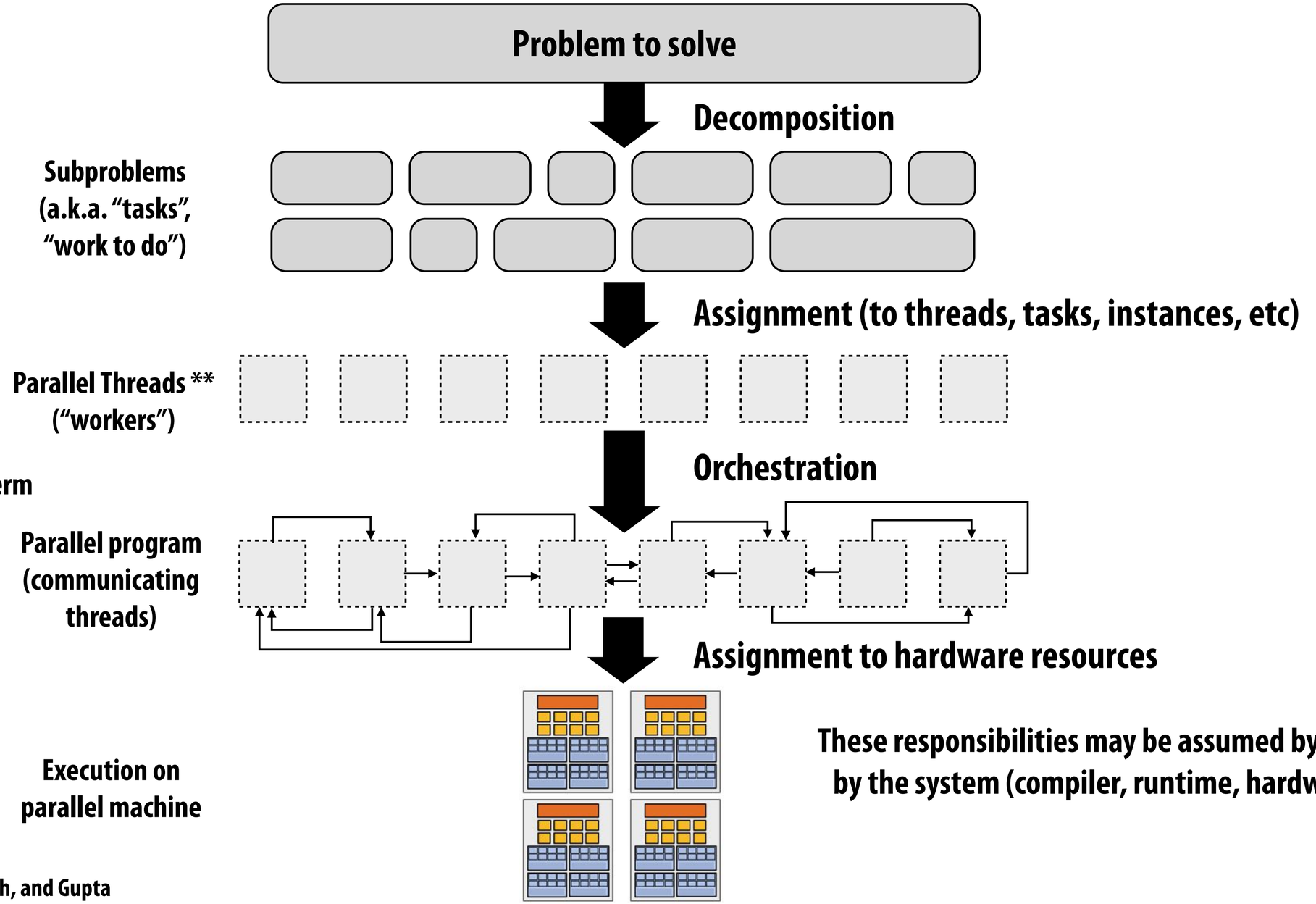
- A common goal is maximizing speedup *

For a fixed computation:

$$\text{Speedup(P processors)} = \frac{\text{Time (1 processor)}}{\text{Time (P processors)}}$$

* Other goals include achieving high efficiency (cost, area, power, etc.) or working on bigger problems than can fit on one machine

Creating a parallel program



** I had to pick a term

These responsibilities may be assumed by the programmer, by the system (compiler, runtime, hardware), or by both!

Adopted from: Culler, Singh, and Gupta

Problem decomposition

- Break up problem into tasks that can be carried out in parallel
- In general: create at least enough tasks to keep all execution units on a machine busy

Key challenge of decomposition:
identifying dependencies
(or... a lack of dependencies)

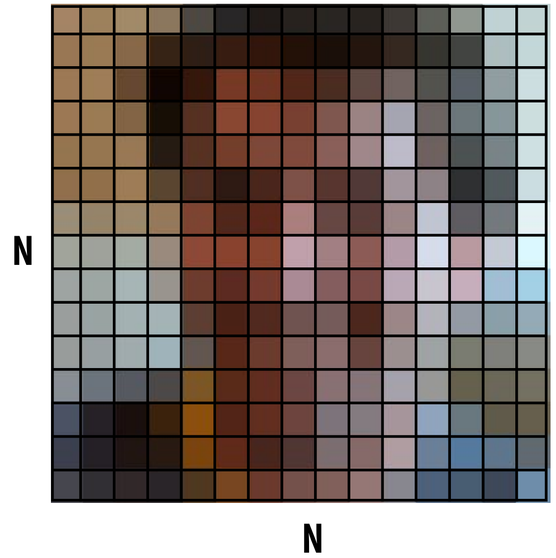
Amdahl's Law: dependencies limit maximum speedup due to parallelism

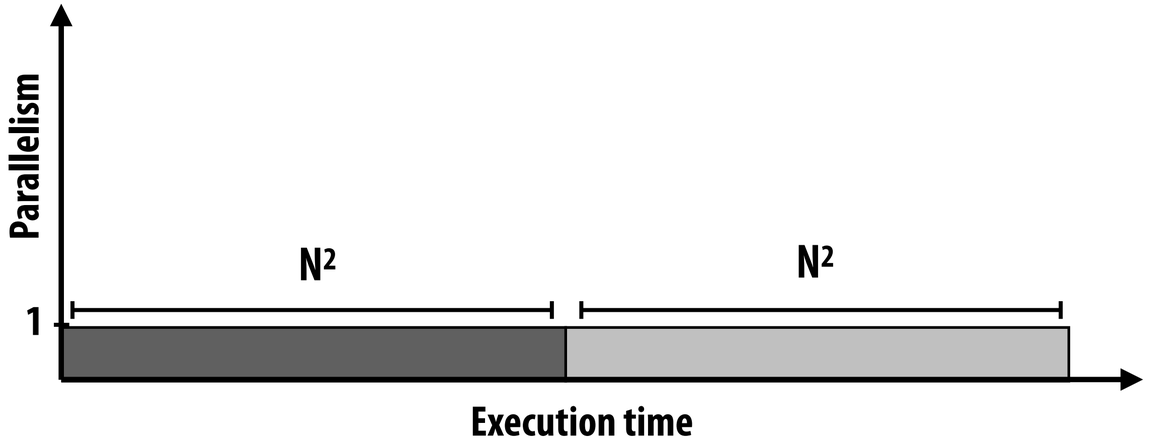
- You run your favorite sequential program...
- Let $S =$ the fraction of sequential execution that is inherently sequential (dependencies prevent parallel execution)
- Then maximum speedup due to parallel execution $\leq {}^{1}/_{S}$

A simple example

- Consider a two-step computation on a N x N image
  - Step 1: multiply brightness of all pixels by two (independent computation on each pixel)
  - Step 2: compute average of all pixel values

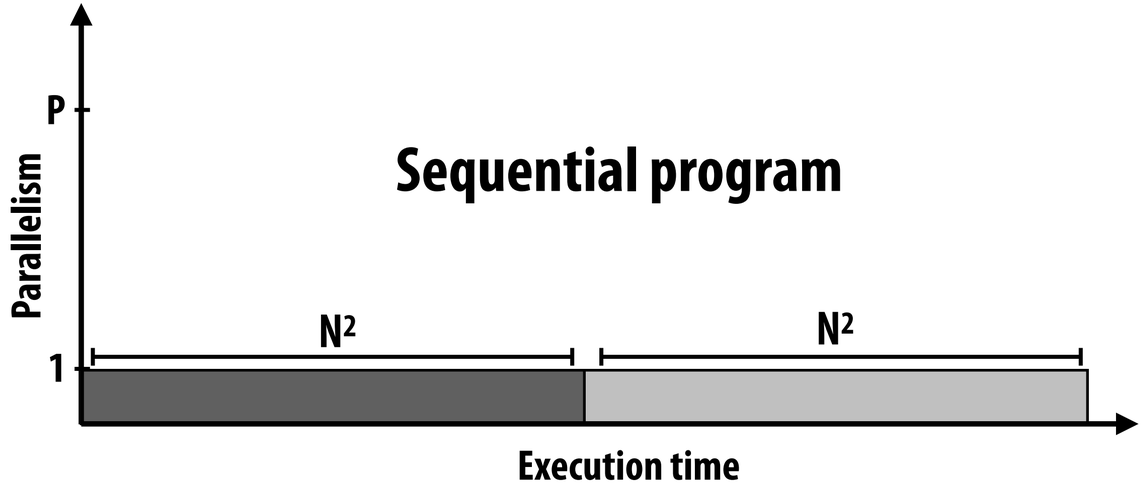
- Sequential implementation of program
  - Both steps take ~ N^2 time, so total time is ~ $2N^2$





First attempt at parallelism (P processors)

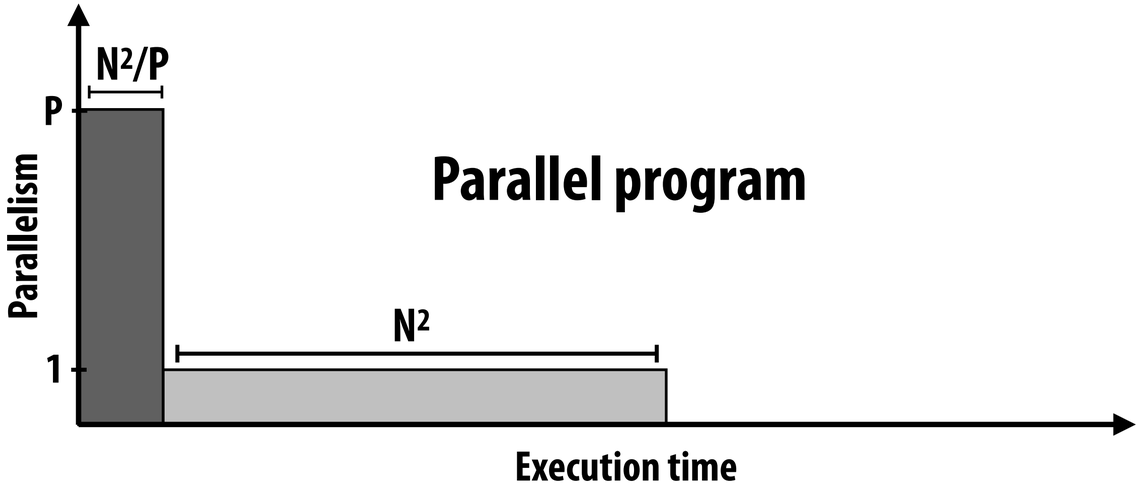
- **Strategy:**
  - Step 1: execute in parallel
    - time for phase 1: N^2/P
  - Step 2: execute serially
    - time for phase 2: N^2



- **Overall performance:**

$$\text{Speedup} \leq \frac{2n^2}{\frac{n^2}{p} + n^2}$$

Speedup ≤ 2

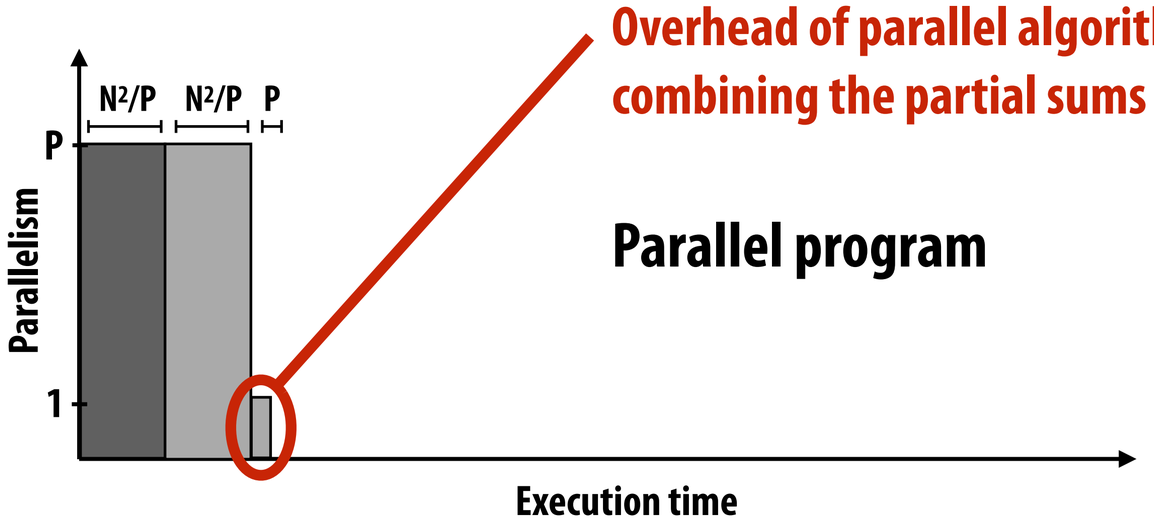


Parallelizing step 2

- **Strategy:**
  - Step 1: execute in parallel
    - time for phase 1: N^2/P
  - Step 2: compute partial sums in parallel, combine results serially
    - time for phase 2: $N^2/P + P$

- **Overall performance:**
  - Speedup $\leq \dfrac{2n^2}{\frac{2n^2}{p} + p}$

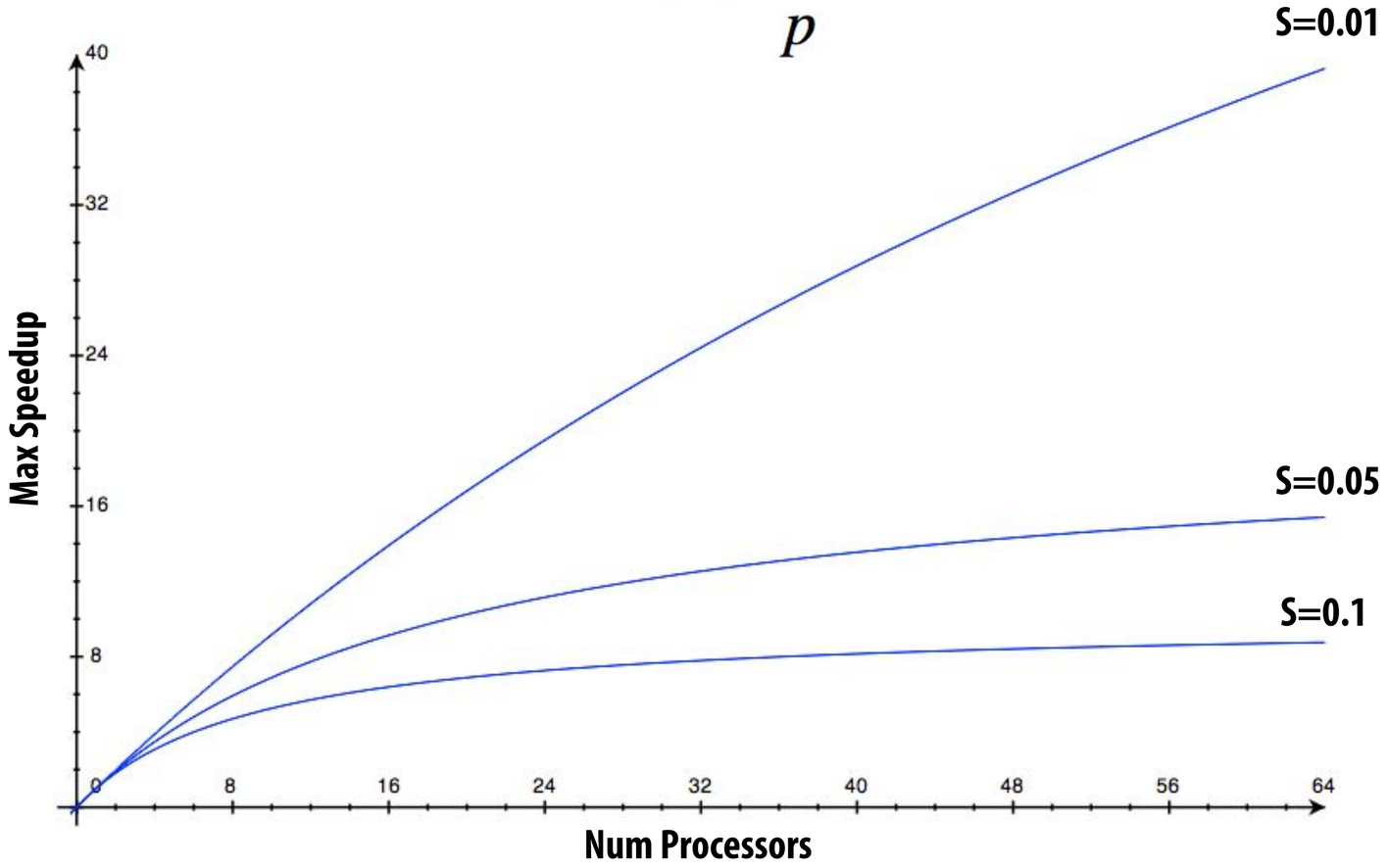
Note: speedup → P when N >> P



Amdahl's law

- Let S = the fraction of total work that is inherently sequential
- Max speedup on P processors given by:

$$\text{speedup} \leq \frac{1}{s + \frac{1-s}{p}}$$



A small serial region can limit speedup on a large parallel machine

Summit supercomputer: 27,648 GPUs x (5,376 ALUs/GPU) = 148,635,648 ALUs

Machine can perform 148 million single precision operations in parallel

What is max speedup if 0.1% of application is serial?

Decomposition

- **Who is responsible for decomposing a program into independent tasks?**
  - In most cases: the programmer

- **Automatic decomposition of sequential programs continues to be a challenging research problem (very difficult in the general case)**
  - Compiler must analyze program, identify dependencies
    - What if dependencies are data dependent (not known at compile time)?
  - Researchers have had modest success with simple loop nests
  - The "magic parallelizing compiler" for complex, general-purpose code has not yet been achieved

Assignment

- Assigning tasks to workers
  - Think of "tasks" as things to do
  - What are "workers"? (Might be threads, program instances, vector lanes, etc.)
- Goals: achieve good workload balance, reduce communication costs
- Can be performed statically (before application is run), or dynamically as program executes
- Although programmer is often responsible for decomposition, many languages/runtimes take responsibility for assignment.

Example: static assignment using C++11 threads

```
void my_thread_start(int N, int terms, float* x, float* results) {
  sinx(N, terms, x, result); // do work
}



void parallel_sinx(int N, int terms, float* x, float* result) {

    int half = N/2.

    // launch thread to do work on first half of array
    std::thread t1(my_thread_start, half, terms, x, result);

    // do work on second half of array in main thread
    sinx(N - half, terms, x + half, result + half);

    t1.join();
}
```

Decomposition of work by loop iteration

Assignment of work to C++ threads is performed by the programmer.

This program is written such that loop iterations are assigned to threads in a blocked fashion (first half of array assigned to the spawned thread, second half assigned to main thread)

Two assignment examples in ISPC

```
export void ispc_sinx_interleaved(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    // assumes N % programCount = 0
    for (uniform int i=0; i<N; i+=programCount)
    {
        int idx = i + programIndex;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6;  // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[idx] * x[idx];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[i] = value;
    }
}
```

Decomposition of work by loop iteration

Programmer-managed assignment:

Static assignment

Assign iterations to ISPC program instances in interleaved fashion

```
export void ispc_sinx_foreach(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    foreach (i = 0 ... N)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        uniform int denom = 6;  // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[i] = value;
    }
}
```

Decomposition of work by loop iteration

`foreach` construct exposes independent work to system

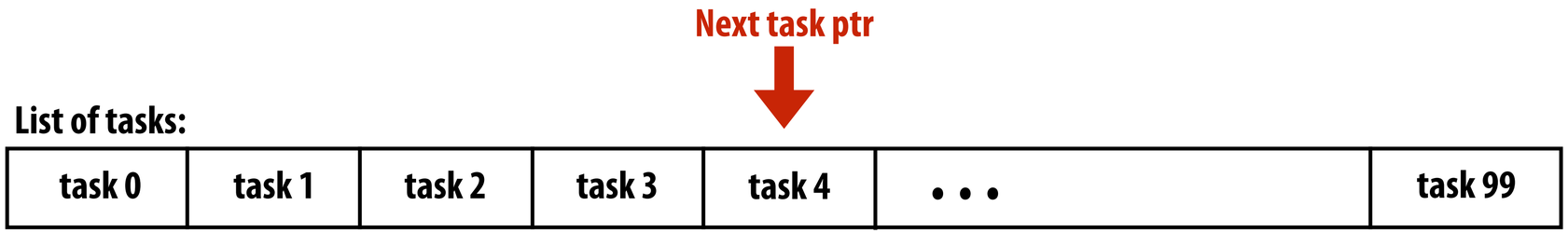
System-manages assignment of iterations (work) to ISPC program instances

(abstraction leaves room for dynamic assignment, but current ISPC implementation is a static scheme just like the code on the left)

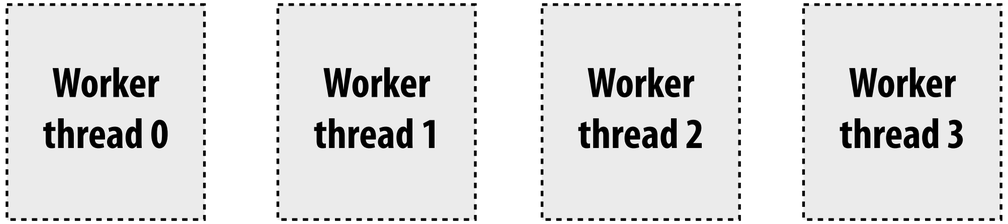
Dynamic assignment using ISPC tasks

```
void foo(uniform float* input,
         uniform float* output,
         uniform int N)
{
  // create a bunch of tasks
  launch[100] my_ispc_task(input, output, N);
}
```

ISPC runtime (invisible to the programmer) assigns tasks to worker threads in a thread pool



Implementation of task assignment to threads: after completing current task, worker thread inspects list and assigns itself the next uncompleted task.



Orchestration

- Involves:
  - Structuring communication
  - Adding synchronization to preserve dependencies if necessary
  - Organizing data structures in memory
  - Scheduling tasks

- Goals: reduce costs of communication/sync, preserve locality of data reference, reduce overhead, etc.

- Machine details impact many of these decisions
  - If synchronization is expensive, programmer might use it more sparsely

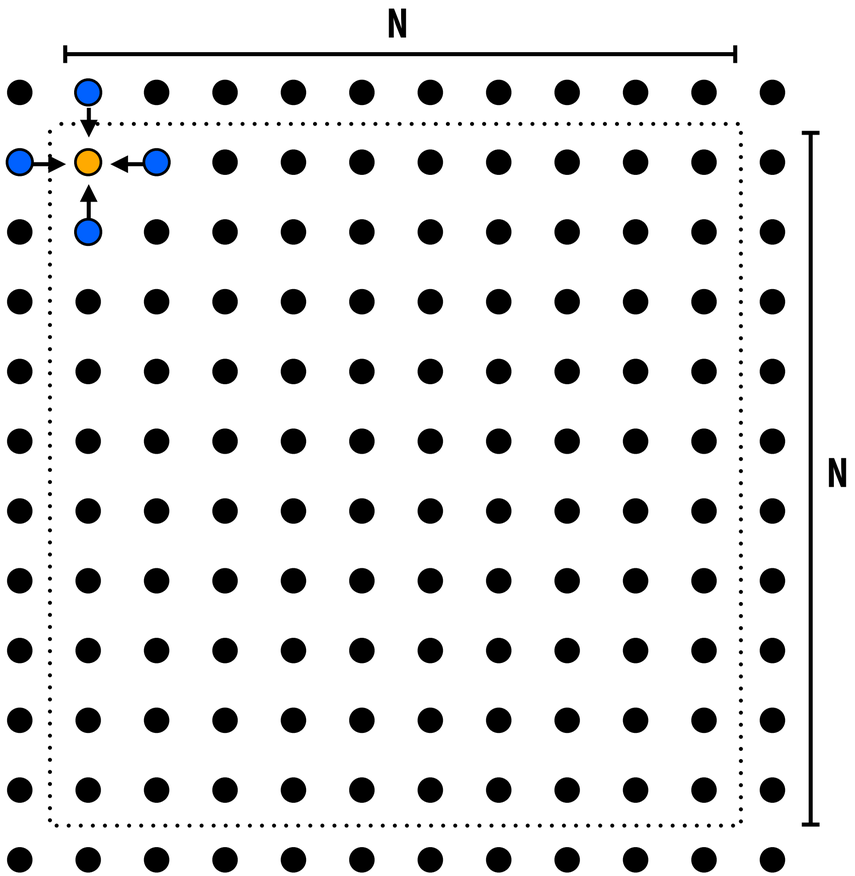
Assignment to hardware

- Assign "threads" ("workers") to hardware execution units
- Example 1: assignment to hardware by the operating system
  - e.g., map a thread to HW execution context on a CPU core
- Example 2: assignment to hardware by the compiler
  - e.g., Map ISPC program instances to vector instruction lanes
- Example 3: assignment to hardware by the hardware
  - e.g., Map CUDA thread blocks to GPU cores (discussed in a future lecture)
- Many interesting decisions:
  - Place related threads (cooperating threads) on the same core
    (maximize locality, data sharing, minimize costs of comm/sync)
  - Place unrelated threads on the same core (one might be bandwidth limited and another might be compute limited) to use machine more efficiently

A parallel programming example

A 2D-grid based solver

- Problem: solve partial differential equation (PDE) on (N+2) x (N+2) grid
- Solution uses iterative algorithm:
  - Perform Gauss-Seidel sweeps over grid until convergence



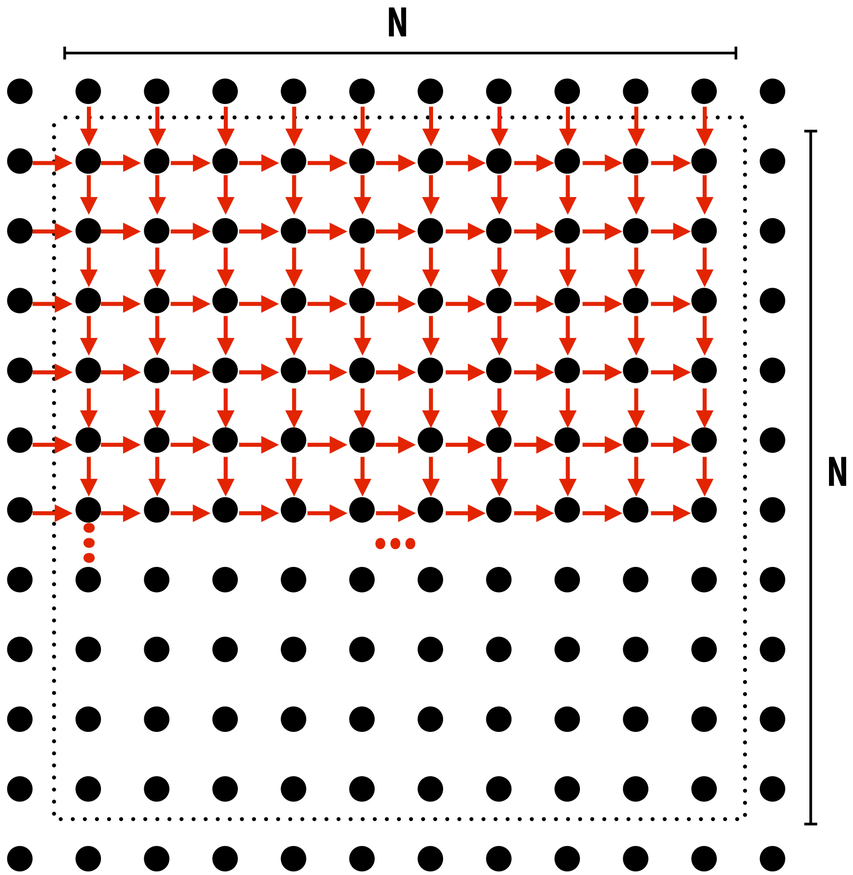

```
A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j]
                       + A[i,j+1] + A[i+1,j]);
```

Grid solver algorithm: find the dependencies

Pseudocode for sequential algorithm is provided below

```
const int n;
float* A;                       // assume allocated for grid of N+2 x N+2 elements

void solve(float* A) {

  float diff, prev;
  bool done = false;

  while (!done) {                           // outermost loop: iterations
    diff = 0.f;
    for (int i=1; i<n i++) {                // iterate over non-border points of grid
      for (int j=1; j<n; j++) {
        prev = A[i,j];
        A[i,j] = 0.2f * (A[i,j] + A[i,j-1] + A[i-1,j] +
                                  A[i,j+1] + A[i+1,j]);
        diff += fabs(A[i,j] - prev);    // compute amount of change
      }
    }

    if (diff/(n*n) < TOLERANCE)         // quit if converged
      done = true;
  }
}
```

Step 1: identify dependencies (problem decomposition phase)

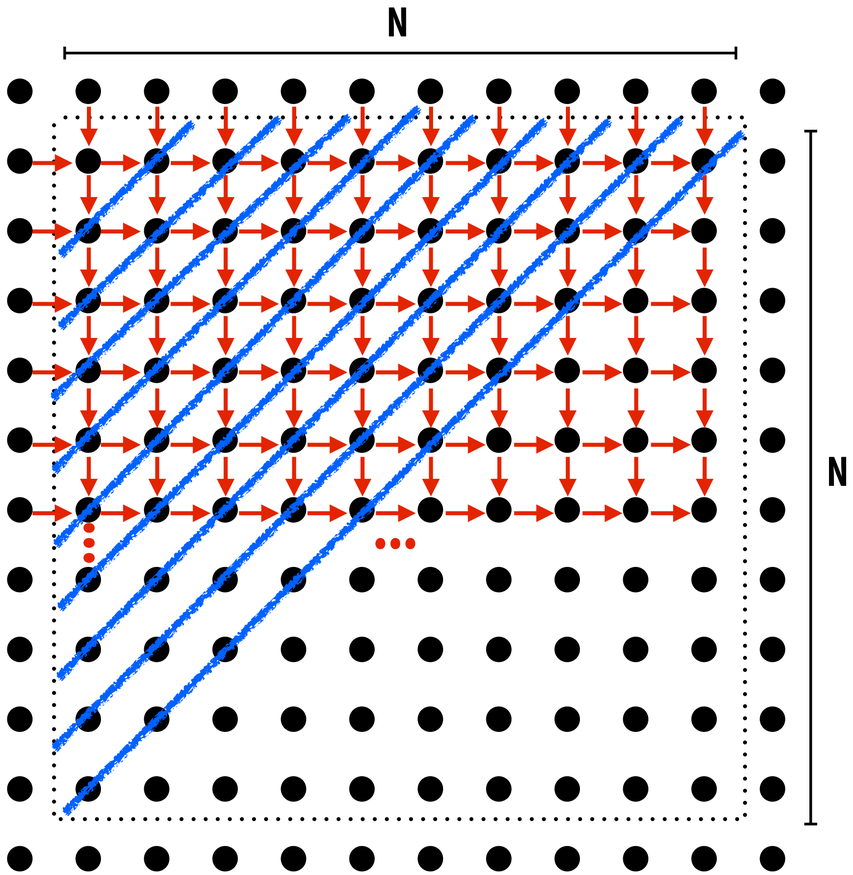


Each row element depends on element to left.

Each row depends on previous row.

Note: the dependencies illustrated on this slide are grid element data dependencies in one iteration of the solver (in one iteration of the "while not done" loop)

Step 1: identify dependencies (problem decomposition phase)



There is independent work along the diagonals!

Good: parallelism exists!

Possible implementation strategy:

1. **Partition grid cells on a diagonal into tasks**
2. **Update values in parallel**
3. **When complete, move to next diagonal**

Bad: independent work is hard to exploit

Not much parallelism at beginning and end of computation.

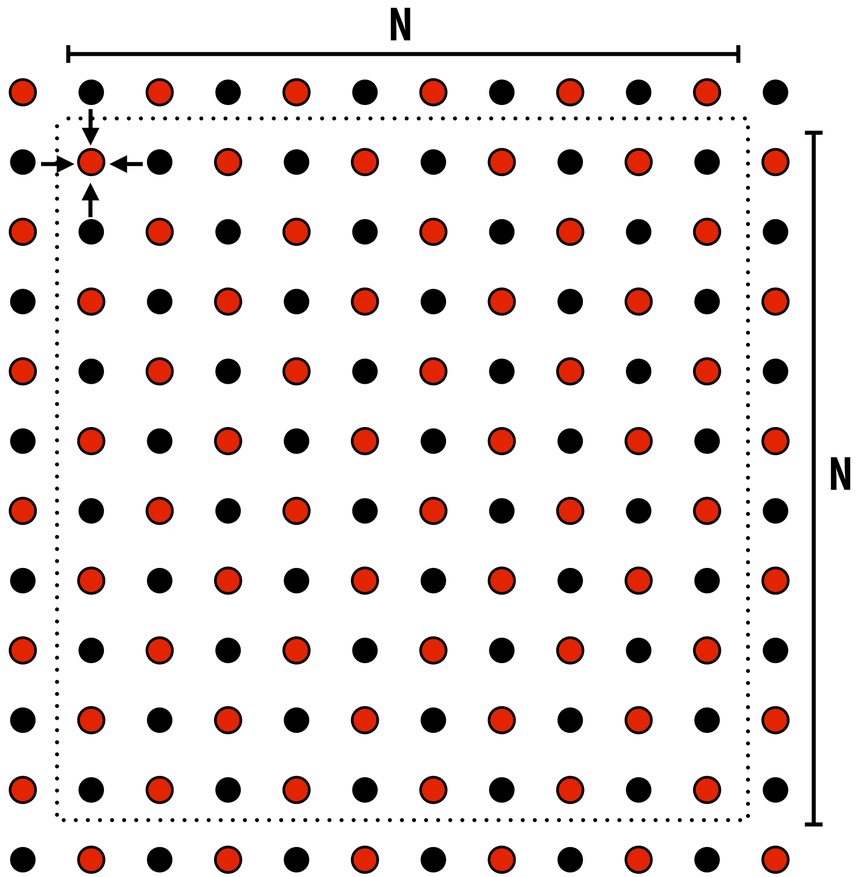
Frequent synchronization (after completing each diagonal)

Let's make life easier on ourselves

- Idea: improve performance by changing the algorithm to one that is more amenable to parallelism
  - Change the order that grid cell cells are updated
  - New algorithm iterates to same solution (approximately), but converges to solution differently
    - Note: floating-point values computed are different, but solution still converges to within error threshold
  - Yes, we needed domain knowledge of the Gauss-Seidel method to realize this change is permissible
    - But this is a common technique in parallel programming

New approach: reorder grid cell update via red-black coloring

Reorder grid traversal: red-black coloring



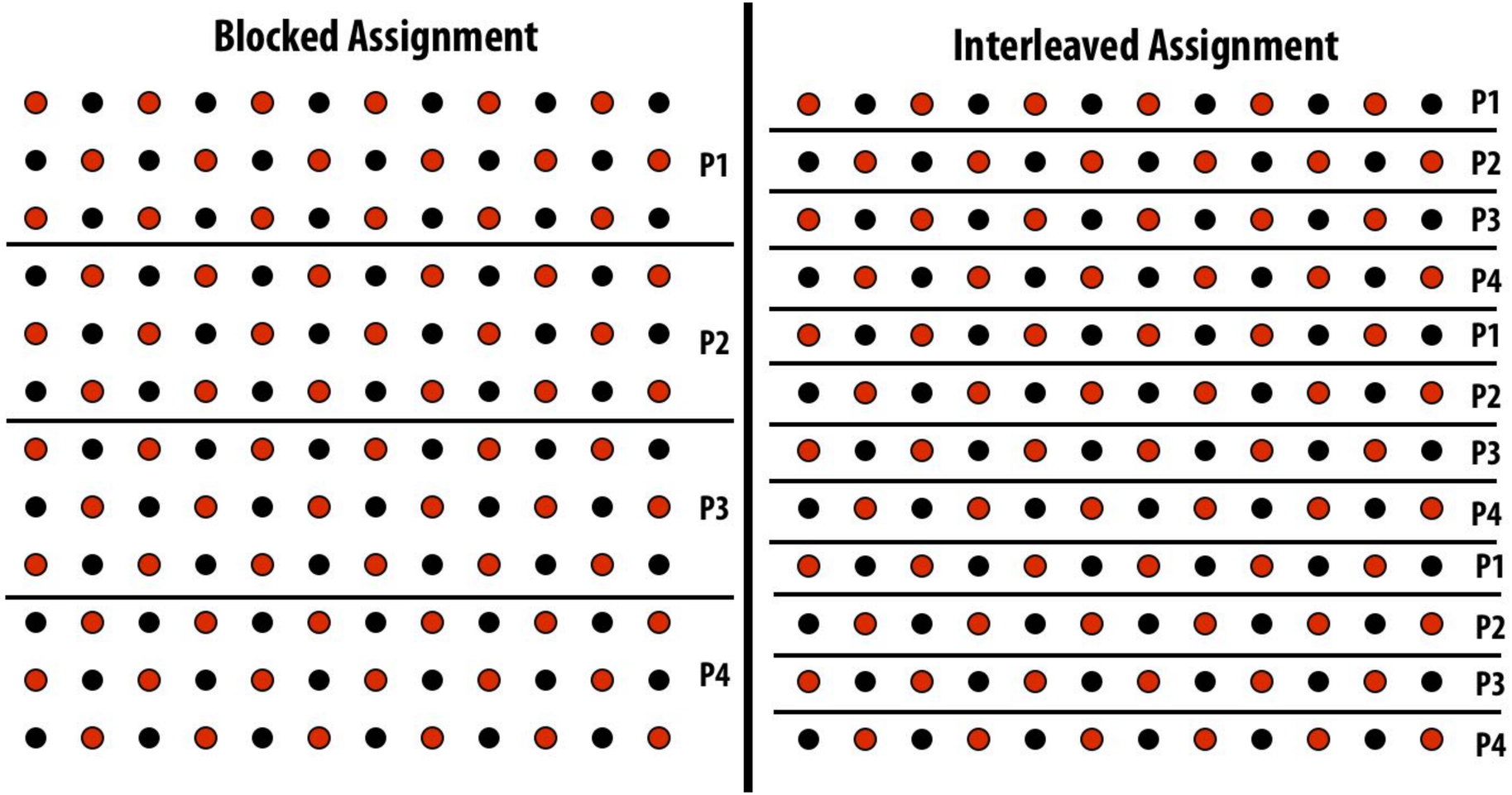
Update all red cells in parallel

When done updating red cells , update all black cells in parallel (respect dependency on red cells)

Repeat until convergence

Possible assignments of work to processors

Reorder grid traversal: red-black coloring

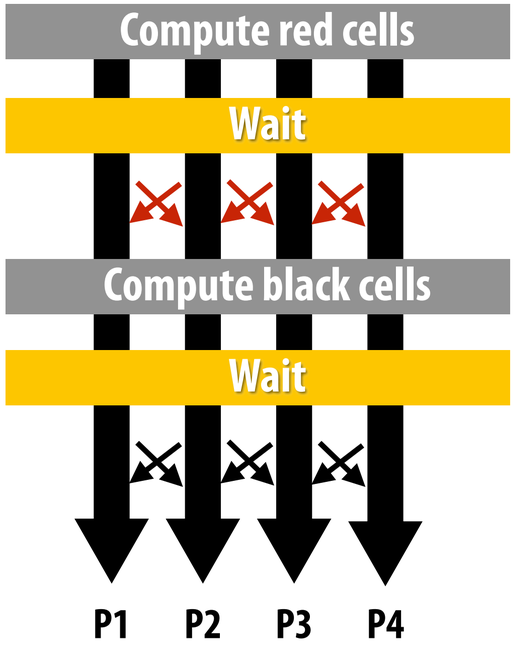


Question: Which is better? Does it matter?

Answer: it depends on the system this program is running on

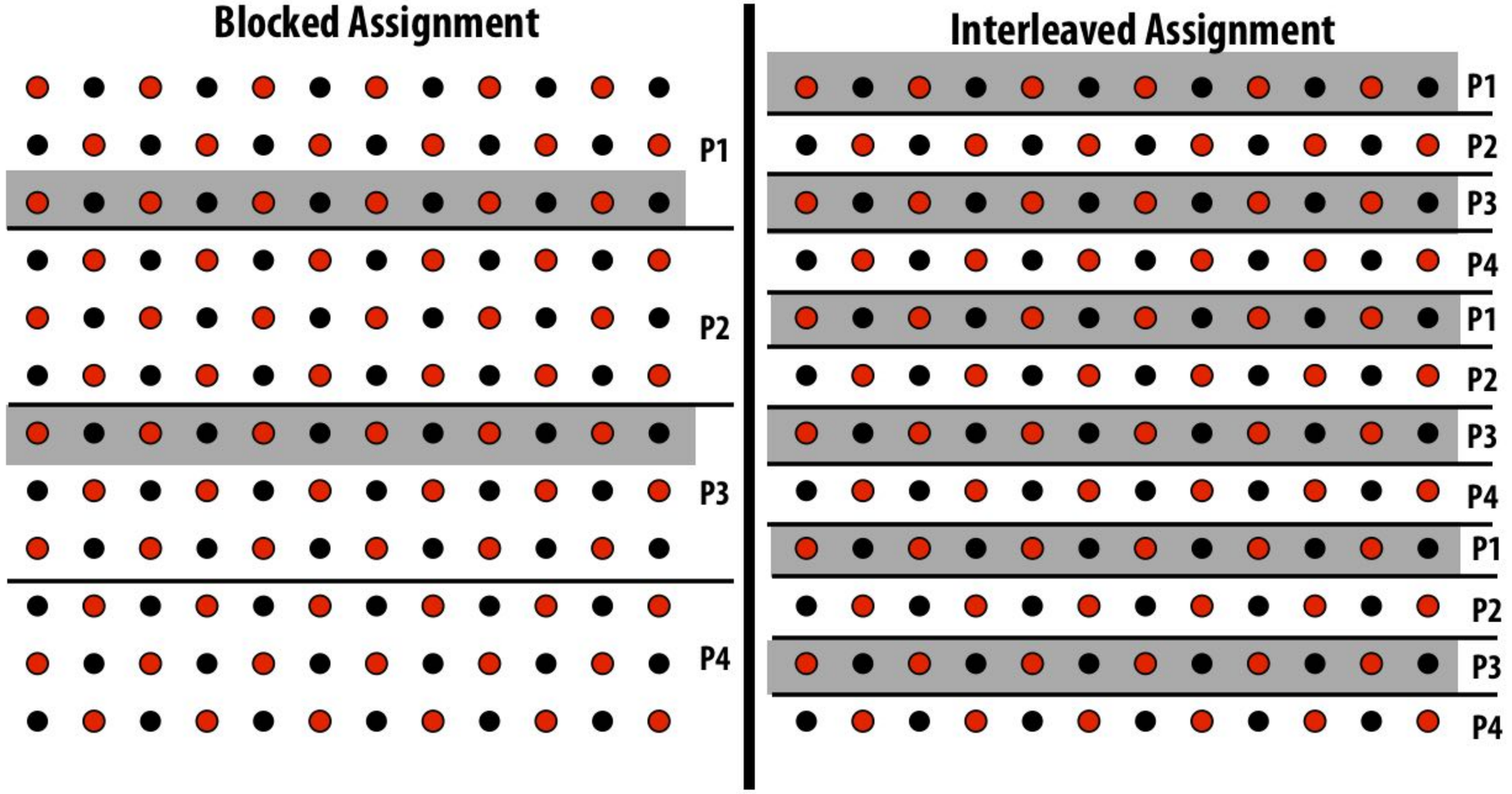
Consider dependencies in the program

1. Perform red cell update in parallel
2. Wait until all processors done with update
3. Communicate updated red cells to other processors
4. Perform black cell update in parallel
5. Wait until all processors done with update
6. Communicate updated black cells to other processors
7. Repeat



Communication resulting from assignment

Reorder grid tra



= data that must be sent to P2 each iteration

Blocked assignment requires less data to be communicated between processors

Two ways to think about writing this program

- Data parallel thinking

- SPMD / shared address space

Data-parallel expression of solver

Data-parallel expression of grid solver

Note: to simplify pseudocode: just showing red-cell update

```
const int n;
float* A = allocate(n+2, n+2));   // allocate grid

void solve(float* A) {

   bool done = false;
   float diff = 0.f;
   while (!done) {
     for_all (red cells (i,j)) {
       float prev = A[i,j];
       A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] +
                        A[i+1,j] + A[i,j+1]);
       reduceAdd(diff, abs(A[i,j] - prev));
     }

     if (diff/(n*n) < TOLERANCE)
       done = true;
   }
}
```

Assignment: ???

Decomposition:
processing individual grid elements
constitutes independent work

Orchestration: handled by system
(builtin communication primitive: reduceAdd)

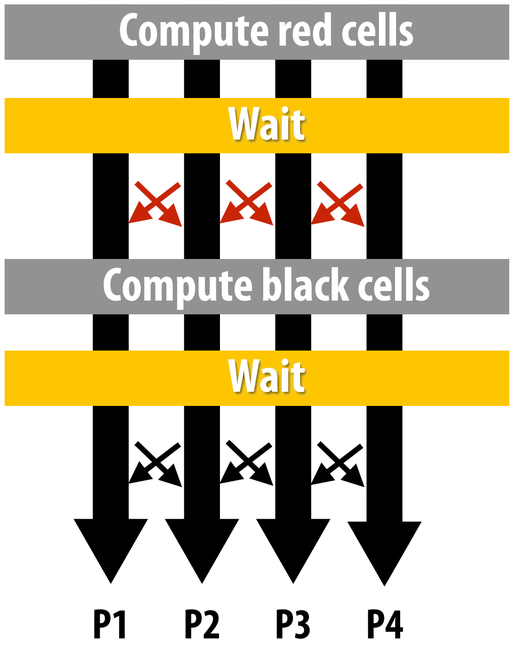
Orchestration: handled by system
(End of for_all block is implicit wait for all workers
before returning to sequential control)

Shared address space (with SPMD threads) expression of solver

Shared address space expression of solver

SPMD execution model

- Programmer is responsible for synchronization
- Common synchronization primitives:
  - Locks (provide mutual exclusion): only one thread in the critical region at a time
  - Barriers: wait for threads to reach this point



Shared address space solver (pseudocode in SPMD execution model)

```
int     n;                  // grid size
bool    done = false;
float   diff = 0.0;
LOCK    myLock;
BARRIER myBarrier;

// allocate grid
float* A = allocate(n+2, n+2);

void solve(float* A) {
   float myDiff;
   int threadId = getThreadId();
   int myMin = 1 + (threadId * n / NUM_PROCESSORS);
   int myMax = myMin + (n / NUM_PROCESSORS)

   while (!done) {
     float myDiff = 0.f;
     diff = 0.f;
     barrier(myBarrier, NUM_PROCESSORS);
     for (j=myMin to myMax) {
        for (i = red cells in this row) {
           float prev = A[i,j];
           A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] + A[i+1,j], A[i,j+1]);
           myDiff += abs(A[i,j] - prev));
     }
     lock(myLock);
     diff += myDiff;
     unlock(myLock);
     barrier(myBarrier, NUM_PROCESSORS);
     if (diff/(n*n) < TOLERANCE)   // check convergence, all threads get same answer
         done = true;
     barrier(myBarrier, NUM_PROCESSORS);
   }
}
```

Assume these are global variables (accessible to all threads)

Assume solve() function is executed by all threads. (SPMD-style)

Value of threadId is different for each SPMD instance: use value to compute region of grid to work on

Each thread computes the rows it is responsible for updating

What's this lock doing here ?????

And these barriers?

Synchronization in a shared address space

Shared address space model (abstraction)

Threads communicate by reading/writing to locations in a shared address space (shared variables)

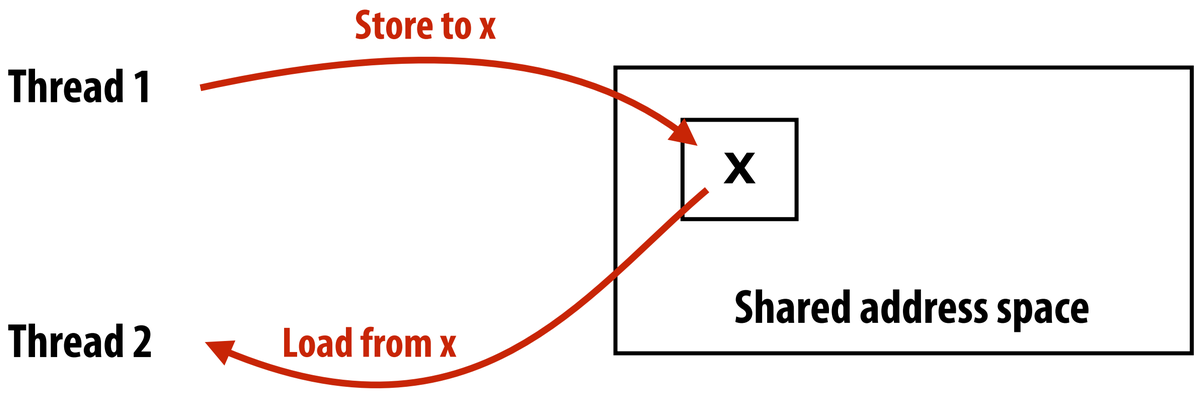
Assume x=0 when threads are launched

Thread 1:

```
// Do work here…

// write to address holding
// contents of variable x
x = 1;
```

Thread 2:

```
void foo(int* x) {

  // read from addr storing
  // contents of variable x
  while (x == 0) {}
  print x;
}
```



(Communication operations shown in red)

(Pseudocode provided in a fake C-like language for brevity.)

A common metaphor:
A shared address space is like a bulletin board

(Everyone can read/write)

Image credit:
https://thetab.com/us/stanford/2016/07/28/honest-packing-list-freshman-stanford-1278

Coordinating access to shared variables with synchronization

Shared (among all threads) variables:

```
int x = 0;
Lock my_lock;
```

Thread 1:

```
mylock.lock();
x++;
mylock.unlock();

print(x);
```

Thread 2:

```
my_lock.lock();
x++;
my_lock.unlock();

print(x);
```

Review: why do we need mutual exclusion?

- Each thread executes:
  - Load the value of variable x from a location in memory into register r1
    (this stores a copy of the value in memory in the register)
  - Add the contents of register r2 to register r1
  - Store the value of register r1 into the address storing the program variable x
- One possible interleaving: (let starting value of x=0, r2=1)

T1	T2	
		T1 reads value 0
r1 ← x		
	r1 ← x	T2 reads value 0
r1 ← r1 + r2		T1 sets value of its r1 to 1
	r1 ← r1 + r2	T2 sets value of its r1 to 1
X ← r1		T1 stores 1 to address of x
	X ← r1	T2 stores 1 to address of x

- Need this set of three instructions must be "atomic"

Example mechanisms for preserving atomicity

- Lock/unlock mutex around a critical section

```
mylock.lock();
// critical section
mylock.unlock();
```

- Some languages have first-class support for atomicity of code blocks

```
atomic {
  // critical section
}
```

- Intrinsics for hardware-supported atomic read-modify-write operations

```
atomicAdd(x, 10);
```

Summary: shared address space model

- Threads communicate by:
  - Reading/writing to shared variables in a shared address space
    - Communication between threads is implicit in memory loads/stores
  - Manipulating synchronization primitives
    - e.g., ensuring mutual exclusion via use of locks

- This is a natural extension of sequential programming
  - In fact, all our discussions in class have assumed a shared address space so far!

Shared address space solver (pseudocode in SPMD execution model)

```
int     n;               // grid size
bool    done = false;
float   diff = 0.0;
LOCK    myLock;
BARRIER myBarrier;

// allocate grid
float* A = allocate(n+2, n+2);

void solve(float* A) {
   float myDiff;
   int threadId = getThreadId();
   int myMin = 1 + (threadId * n / NUM_PROCESSORS);
   int myMax = myMin + (n / NUM_PROCESSORS)

   while (!done) {
     float myDiff = 0.f;
     diff = 0.f;
     barrier(myBarrier, NUM_PROCESSORS);
     for (j=myMin to myMax) {
        for (i = red cells in this row) {
           float prev = A[i,j];
           A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] + A[i+1,j], A[i,j+1]);
           LOCK(myLock);
           diff += abs(A[i,j] - prev));
           UNLOCK(myLock):
     }
     barrier(myBarrier, NUM_PROCESSORS);
     if (diff/(n*n) < TOLERANCE)            // check convergence, all threads get same answer
         done = true;
     barrier(myBarrier, NUM_PROCESSORS);
   }
}
```

Value of threadId is different for each SPMD instance: use value to compute region of grid to work on

Each thread computes the rows it is responsible for updating

Hint

Do you see a potential performance problem with this implementation?

Shared address space solver

(pseudocode in SPMD execution model)

```
int     n;                // grid size
bool    done = false;
float   diff = 0.0;
LOCK    myLock;
BARRIER myBarrier;

// allocate grid
float* A = allocate(n+2, n+2);

void solve(float* A) {
   float myDiff;
   int threadId = getThreadId();
   int myMin = 1 + (threadId * n / NUM_PROCESSORS);
   int myMax = myMin + (n / NUM_PROCESSORS)

   while (!done) {
     float myDiff = 0.f;
     diff = 0.f;
     barrier(myBarrier, NUM_PROCESSORS);
     for (j=myMin to myMax) {
        for (i = red cells in this row) {
           float prev = A[i,j];
           A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] + A[i+1,j], A[i,j+1]);
           myDiff += abs(A[i,j] - prev));
     }
     lock(myLock);
     diff += myDiff;
     unlock(myLock);
     barrier(myBarrier, NUM_PROCESSORS);
     if (diff/(n*n) < TOLERANCE)            // check convergence, all threads get same answer
         done = true;
     barrier(myBarrier, NUM_PROCESSORS);
   }
}
```

Improve performance by accumulating into partial sum locally, then complete global reduction at the end of the iteration.

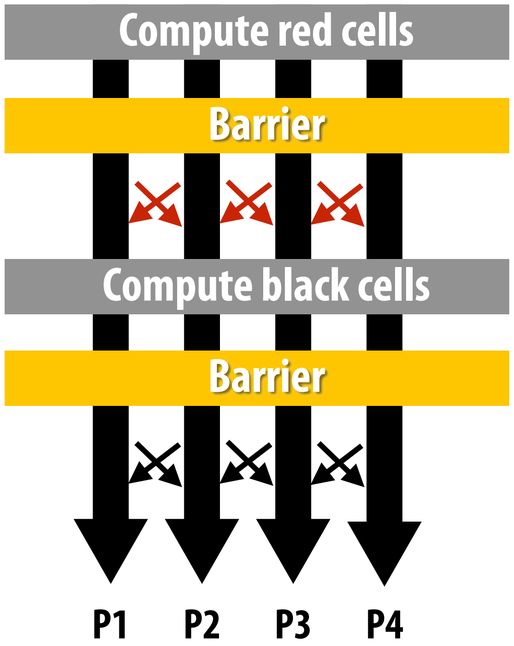
Compute partial sum per worker

Now only only lock once per thread, not once per (i,j) loop iteration!

Grid solver example from: Culler, Singh, and Gupta

Barrier synchronization primitive

- `barrier(num_threads)`
- Barriers are a conservative way to express dependencies
- Barriers divide computation into phases
- All computation by all threads before the barrier complete before any computation in any thread after the barrier begins
  - In other words, all computations after the barrier are assumed to depend on all computations before the barrier



Shared address space solver

```
int     n;               // grid size
bool    done = false;
float   diff = 0.0;
LOCK    myLock;
BARRIER myBarrier;

// allocate grid
float* A = allocate(n+2, n+2);

void solve(float* A) {
   float myDiff;
   int threadId = getThreadId();
   int myMin = 1 + (threadId * n / NUM_PROCESSORS);
   int myMax = myMin + (n / NUM_PROCESSORS)

   while (!done) {
     float myDiff = 0.f;
     diff = 0.f;
     barrier(myBarrier, NUM_PROCESSORS);
     for (j=myMin to myMax) {
        for (i = red cells in this row) {
           float prev = A[i,j];
           A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] + A[i+1,j], A[i,j+1]);
           myDiff += abs(A[i,j] - prev));
     }
     lock(myLock);
     diff += myDiff;
     unlock(myLock);
     barrier(myBarrier, NUM_PROCESSORS);
     if (diff/(n*n) < TOLERANCE)            // check convergence, all threads get same answer
        done = true;
     barrier(myBarrier, NUM_PROCESSORS);
   }
}
```

Why are there three barriers?

Shared address space solver: one barrier

```
int     n;               // grid size
bool    done = false;
LOCK    myLock;
BARRIER myBarrier;
float diff[3];  // global diff, but now 3 copies

float *A = allocate(n+2, n+2);


void solve(float* A) {
  float myDiff;   // thread local variable
  int index = 0;  // thread local variable

  diff[0] = 0.0f;
  barrier(myBarrier, NUM_PROCESSORS);  // one-time only: just for init

  while (!done) {
    myDiff = 0.0f;
    //
    // perform computation (accumulate locally into myDiff)
    //
    lock(myLock);
    diff[index] += myDiff;    // atomically update global diff
    unlock(myLock);
    diff[(index+1) % 3] = 0.0f;
    barrier(myBarrier, NUM_PROCESSORS);
    if (diff[index]/(n*n) < TOLERANCE)
      break;
    index = (index + 1) % 3;
  }
}
```

Idea:

Remove dependencies by using different `diff` variables in successive loop iterations

Trade off footprint for removing dependencies! (a common parallel programming technique)

Grid solver implementation in two programming models

- **Data-parallel programming model**
  - Synchronization:
    - Single logical thread of control, but iterations of `forall` loop may be parallelized by the system (implicit barrier at end of `forall` loop body)
  - Communication
    - Implicit in loads and stores (like shared address space)
    - Special built-in primitives for more complex communication patterns: e.g., reduce

- **Shared address space**
  - Synchronization:
    - Mutual exclusion required for shared variables (e.g., via locks)
    - Barriers used to express dependencies (between phases of computation)
  - Communication
    - Implicit in loads/stores to shared variables

Summary

- **Amdahl's Law**
  - Overall maximum speedup from parallelism is limited by amount of serial execution in a program
- **Aspects of creating a parallel program**
  - Decomposition to create independent work, assignment of work to workers, orchestration (to coordinate processing of work by workers), mapping to hardware
  - We'll talk a lot about making good decisions in each of these phases in the coming lectures
- **Focus today: identifying dependencies**
- **Focus soon: identifying locality, reducing synchronization**