Lecture 5:
Parallel Programming Basics

Parallel Computer Architecture and Programming
CMU / 清华大学, Summer 2017
Prof. Kayvon in China
Checking your understanding

This is an ISPC function.

It contains a loop nest.

Which iterations of the loop(s) are parallelized by ISPC? Which are not?

Answer: none of them
Parallel program instances where created when the `sinx()` ispc function was called

Compute $\sin(x)$ using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + ...$

```c
#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);
```

Sequential execution (C code)

Call to `sinx()`

Begin executing `programCount` instances of `sinx()` (ISPC code)

`sinx()` returns.

Completion of ISPC program instances.

Resume sequential execution

Sequential execution (C code)

Each instance will run the code in the `ispc` function serially.
(parallelism exists because there are multiple program instances, not the code that defines an `ispc` function)
Today’s topic: the process of parallelizing a problem
Creating a parallel program

Thought process:
1. Identify work that can be performed in parallel
2. Partition problem into pieces of work that can be performed in parallel (and also data associated with the work)
3. Manage data access, communication, and synchronization between the pieces of work

Recall one of our main goals is achieving speedup *

For a given computation:

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

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

Problem to solve

Decomposition

Subproblems (a.k.a. “tasks”, “work to do”)

Parallel Threads ** ("workers")

Assignment

Parallel program (communicating threads)

Orchestration

Mapping

Execution on parallel machine

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

Adopted from: Culler, Singh, and Gupta

** I had to pick a term

CMU / 清华大学, Summer 2017
Problem decomposition

- Break up problem into tasks that can be carried out in parallel

- Main idea: 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 $\leq \frac{1}{S}$
A simple example

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

- Sequential implementation of program
  - Both steps take \( \sim N^2 \) time, so total time is \( \sim 2N^2 \)
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:**

  \[
  \text{Speedup} \leq \frac{2n^2}{n^2 + n^2/P}
  \]

  \[
  \text{Speedup} \leq 2
  \]
Parallelizing step 2

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

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

Note: speedup approaches $P$ when $N >> P$
Amdahl’s law

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

$$\text{speedup} \leq \frac{1}{S + \frac{1-S}{P}}$$

<table>
<thead>
<tr>
<th>Processors</th>
<th>Max Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
</tr>
</tbody>
</table>
Decomposition

- Who is responsible for performing problem decomposition?
  - In most cases: the programmer

- Automatic decomposition of sequential programs continues to be a challenging research problem (very difficult in 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

Problem to solve

Decomposition

Subproblems
(a.k.a. “tasks”,
“work to do”)

Assignment

Parallel Threads**
(“workers”)

Parallel program
(communicating
threads)

Orchestration

Mapping

Execution on
parallel machine

** I had to pick a term
Assignment

- Assigning tasks to threads **
  - Think of “tasks” as things to do
  - Think of threads as “workers”

- Goals: balance workload, reduce communication costs

- Although programmer is often responsible for decomposition, many languages/runtimes take responsibility for assignment.

- Assignment be performed statically, or dynamically during execution

** I had to pick a term (will explain in a second)
Example 1: Assignment in ISPC

```c
export void 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[i] = value;
    }
}
```

Decomposition of work by loop iteration

Programmer-managed assignment:

**Static assignment**

Code assigns iterations of loop to ISPC program instances in interleaved fashion

System-manages assignment of iterations (work) to ISPC program instances (abstraction leaves room for dynamic assignment, but current ISPC implementation is static)
Example 2: static assignment using C++11 threads

```c++
void my_thread_start(int N, int terms, float* x, float* results) {
    sinx(N, terms, x, results); // 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

Programmer-managed static assignment
This program assigns iterations to threads in a blocked fashion
(first half of array assigned to the spawned thread, second half assigned to main thread)
Dynamic assignment using ISPC tasks

```c
void foo(uniform float input[],
  uniform float output[],
  uniform int N)
{
// create many tasks
launch[100] my_ispc_task(input, output, N);
}
```

ISPC runtime assign tasks to worker threads

List of tasks:

| task 0 | task 1 | task 2 | task 3 | task 4 | ... | task 99 |

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

Problem to solve

Decomposition

Subproblems
(a.k.a. “tasks”,
“work to do”)

Assignment

Parallel Threads**
(“workers”)

Parallel program
(communicating
threads)

Orchestration

Mapping

Execution on
parallel machine

** I had to pick a term
Orchestration

- **Involves:**
  - Communicating between workers
  - 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, might use it more sparsely
Mapping to hardware

Problem to solve

Decomposition

Subproblems (a.k.a. “tasks”, “work to do”)

Assignment

Parallel Threads **
(“workers”)

Parallel program (communicating threads)

Orchestration

Parallel Threads ** ("workers")

Execution on parallel machine

** I had to pick a term
Mapping to hardware

- Mapping “threads” (“workers”) to hardware execution units

- Example 1: mapping by the operating system
  - e.g., map thread to hardware execution context on a CPU core

- Example 2: mapping by the compiler
  - Map ISPC program instances to vector instruction lanes

- Example 3: mapping by the hardware
  - Map CUDA thread blocks to GPU cores (we will discuss in a future lecture)

- Some interesting mapping decisions:
  - Place related threads (cooperating threads) on the same processor
    (maximize locality, data sharing, minimize costs of comm/sync)
  - Place unrelated threads on the same processor (one might be bandwidth limited and another might be compute limited) to use machine more efficiently
Example: last class I asked you a question about mapping

- Consider an application that creates **two** threads
- The application runs on the processor shown below
  - Two cores, two-execution contexts per core, up to instructions per clock, one instruction is an 8-wide SIMD instruction.
- Question: “who” is responsible for mapping the applications’s pthreads to the processor’s thread execution contexts?
  
  **Answer:** the operating system

- Question: If you were implementing the OS, how would to map the two threads to the four execution contexts?

- Another question: How would you map threads to execution contexts if your C program spawned **five** threads?
A parallel programming example
A 2D-grid based solver

- Goal: solve partial differential equation (PDE) on \((N+2) \times (N+2)\) grid

- Iterative solution
  - Perform Gauss-Seidel sweeps over grid until convergence

\[
\]
Grid solver algorithm

Pseudocode for sequential algorithm is provided below

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

void solve(float* A) {
    float diff, prev;
    bool done = false;
    while (!done) {
        // outermost loop: iterations
        diff = 0.0f;
        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];
                diff += fabs(A[i,j] - prev); // compute amount of change
            }
            // compute amount of change
        }
        if (diff/(n*n) < TOLERANCE) // quit if simulation has converged
            done = true;
    }
}
```

Grid solver example from: Culler, Singh, and Gupta
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 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 the grid 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 Gauss-Seidel method for solving a linear system to realize this change is permissible for the application
New approach: reorder grid cell update via 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

Question: Which is better? Does it matter?
Answer: It depends on the system this program is running on.
Consider dependencies (data flow)

1. Perform red update in parallel
2. Wait until all processors done with update
3. **Communicate updated red cells to other processors**
4. Perform black 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

Blocked Assignment

Interleaved Assignment

= data that must be sent to P2 each iteration

Blocked assignment requires less data to be communicated between processors
Three ways to think about writing this program

- Data parallel
- SPMD / shared address space
- Message passing (will wait until a future class)
Data-parallel
But first, let’s talk about “thinking” in a data parallel way
Data-parallel model

- Express programs with a very rigid structure
  - Perform same operation on each element of an array

- Programming in matlab or numPy is a good example: \( C = A + B \)
  (A, B, and C are vectors of same length)

- Often takes form of SPMD programming
  - map(function, collection)
  - Where function is applied to each element of collection independently
  - function may be a complicated sequence of logic (e.g., a loop body)
  - map returns when function has been applied to all elements of collection
Data parallelism in ISPC via foreach

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

// initialize N elements of x here
absolute_value(N, x, y);
```

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

Think of loop body as function (from the previous slide)
foreach construct is a map

Given this program, it is reasonable to think of the program as mapping the loop body onto each element of the arrays X and Y.

But if we want to be more precise: the collection is not a first-class ISPC concept. It is implicitly defined by the array indexing logic in the code.

(There is no operation in ISPC with the semantics: “map this code over all elements of this array”)

CMU / 清华大学, Summer 2017
Data parallelism in ISPC

// 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
absolute_repeat(N/2, x, y);

Think of loop body as function
foreach construct is a map
Collection is implicitly defined by array indexing logic

// 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 is also a valid ISPC program!

It takes the absolute value of elements of x, then repeats it twice in the output array y

(Less obvious how to think of this code as mapping the loop body onto existing collections.)
Data parallelism in ISPC

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

// initialize N elements of x
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];
    }
}

Think of loop body as function
foreach construct is a map
Collection is implicitly defined by array indexing logic

The output of this program is undefined!

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

Data-parallel model (foreach) provides no specification of order in which iterations occur

Model provides no primitives for fine-grained mutual exclusion/synchronization). It is not intended to help programmers write programs with that structure
Data parallelism: a “pure” approach

Note: this is not ISPC syntax (more of Kayvon’s made up syntax)

Main program:

```cpp
const int N = 1024;

stream<float> x(N);  // sequence (a “stream”)
stream<float> y(N);  // sequence (a “stream”)

// initialize N elements of x here...

// map function absolute_value onto streams
absolute_value(x, y);
```

Data-parallelism expressed in this functional form is sometimes referred to as the stream programming model

Streams: sequences of elements. Elements in a stream can be processed independently

Kernels: side-effect-free functions. Operate element-wise on collections

Think of the inputs, outputs, and temporaries for each kernel invocation as forming a private per-invocation address space

“Kernel” definition:

```cpp
void absolute_value(float x, float y) {
    if (x < 0)
        y = -x;
    else
        y = x;
}
```
A data-parallel program with two kernels

const int N = 1024;
stream<float> input(N);
stream<float> output(N);
stream<float> tmp(N);

foo(input, tmp);
bar(tmp, output);

parallel_for(int i=0; i<N; i++)
{
    output[i] = bar(foo(input[x]));
}

Key program dependencies are known by compiler (enables compiler to perform optimizations):

- Independent processing on elements, kernel functions are side-effect free:
  - Optimization: parallelize kernel execution
  - Application cannot write a program that is non-deterministic under parallel execution
Stream programming drawbacks

// “pure” stream programming pseudocode
const int N = 1024;
stream<float> input(N/2);
stream<float> tmp(N);
stream<float> output(N);

// double length of stream by replicating
// all elements 2x
streamRepeat(2, input, tmp);

absolute_value(tmp, output);

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

Need library of stream data access operators to describe complex data flows (see use of repeat operator at left to obtain same behavior as indexing code at right)
Gather/scatter: two key data-parallel communication primitives

Map absolute_value onto stream produced by gather:

```cpp
const int N = 1024;
stream<float> input(N);
stream<int> indices;
stream<float> tmp_input(N);
stream<float> output(N);

stream_gather(input, indices, tmp_input);
absolute_value(tmp_input, output);
```

Map absolute_value onto stream, scatter results:

```cpp
const int N = 1024;
stream<float> input(N);
stream<int> indices;
stream<float> tmp_output(N);
stream<float> output(N);

absolute_value(input, tmp_output);
stream_scatter(tmp_output, indices, output);
```

ISPC equivalent:

```cpp
export void absolute_value(
    uniform float N,
    uniform float* input,
    uniform float* output,
    uniform int* indices)
{
    foreach (i = 0 ... n)
    {
        float tmp = input[indices[i]];
        if (tmp < 0)
            output[i] = -tmp;
        else
            output[i] = tmp;
    }
}
```

ISPC equivalent:

```cpp
export void absolute_value(
    uniform float N,
    uniform float* input,
    uniform float* output,
    uniform int* indices)
{
    foreach (i = 0 ... n)
    {
        if (input[i] < 0)
            output[indices[i]] = -input[i];
        else
            output[indices[i]] = input[i];
    }
}
```
Gather instruction

gather(R1, R0, mem_base);  “Gather from buffer mem_base into R1 according to indices specified by R0.”

Array in memory with (base address = mem_base)

Gather supported with AVX2 in 2013
But AVX2 does not support SIMD scatter (must implement as scalar loop)
Scatter instruction exists in AVX512

Hardware supported gather/scatter does exist on GPUs.
(still an expensive operation compared to load/store of contiguous vector)
Back to the grid solver example
Data-parallel expression of grid solver

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

```c
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];
            reduceAdd(diff, abs(A[i,j] - prev));
        }

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

Assignment: ???

decomposition: individual grid elements constitute 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)

Grid solver example from: Culler, Singh, and Gupta
Shared address space (with SPMD threads) expression of solver
But first, let’s talk about “thinking” about coordination and communication in a shared address space.
Shared address space model (abstraction)

- Programs may create multiple threads
- Threads communicate by reading/writing to shared variables
- Shared variables are like a big bulletin board
  - Any thread can read or write to shared variables

Thread 1:
```c
int x = 0;
spawn_thread(foo, &x);
x = 1;
```

Thread 2:
```c
void foo(int* x) {
    while (x == 0) {}
    print x;
}
```

(Pseudocode provided in a fake C-like language for brevity.)
Shared address space model (abstraction)

- Threads communicate by:
  - Reading/writing to shared variables
    - Inter-thread communication is implicit in memory operations
    - Thread 1 stores to X
    - Later, thread 2 reads X (and observes update of value by thread 1)
  - Access to shared variables can be coordinated using primitives like locks (future lecture)

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

- Helpful analogy: shared variables are like a big bulletin board
  - Any thread can read or write to shared variables
HW implementation of a shared address space

Key idea: hardware allows any processor to **directly** access any memory location using load and store instructions

"Symmetric (shared-memory) multi-processor" (SMP):

- Cost of accessing an uncached memory address is the same for all processors
Shared address space HW architectures
Commodity x86 examples

Intel Core i7 (quad core)
(interconnect is a ring)

AMD Phenom II (six core)
SUN Niagara 2 (UltraSPARC T2)

Note area of crossbar (CCX): about same area as area of one core

Eight cores
Non-uniform memory access (NUMA)

All processors can access any memory location, but... the cost of memory access (latency and/or bandwidth) is different for different processors

Example: latency to access address $x$ is higher from cores 5-8 than cores 1-4

Example: modern dual-socket configuration

- Memory
  - Memory Controller
    - Core 1
    - Core 2
    - Core 3
    - Core 4
  - Memory Controller
    - Core 5
    - Core 6
    - Core 7
    - Core 8

On chip network

AMD Hyper-transport / Intel QuickPath (QPI)
Back to the grid solver example
Recall the red-black grid solver algorithm

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
Shared address space expression of solver

Using the SPMD execution model

- Need synchronization primitives to coordinate all the threads (programmer must handle synchronization):

  We’ll use two primitives in this example:
  - Locks (provide mutual exclusion): only one thread in the critical region at a time
  - Barriers: wait for all threads to reach this point
Shared address space solver (pseudocode in SPMD execution model)

```c
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) {
    int threadId = getThreadId();
    int myMin = 1 + (threadId * n / NUM_PROCESSORS);
    int myMax = myMin + (n / NUM_PROCESSORS);

    while (!done) {
        diff = 0.0f;
        barrier(myBarrier, NUM_PROCESSORS);
        for (j=myMin to myMax) {
            for (i = red cells in this row) {
                float prev = A[i,j];

                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;
    }
}
```

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

Each thread computes the rows it is responsible for updating

Assume these are global variables (accessible to all threads)

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

Grid solver example from: Culler, Singh, and Gupta
Why do we need mutual exclusion?

- Each thread executes
  - Load the value of `diff` from shared memory into register `r1`
  - Add the register `r2` to register `r1`
  - Store the value of register `r1` into `diff`

- One possible interleaving: (let starting value of `diff`=0, `r2`=1)

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r1 ← diff</code></td>
<td><code>T0 reads value 0</code></td>
</tr>
<tr>
<td><code>r1 ← r1 + r2</code></td>
<td><code>T1 reads value 0</code></td>
</tr>
<tr>
<td><code>diff ← r1</code></td>
<td><code>T0 sets value of its r1 to 1</code></td>
</tr>
<tr>
<td></td>
<td><code>r1 ← r1 + r2</code></td>
</tr>
<tr>
<td></td>
<td><code>T1 sets value of its r1 to 1</code></td>
</tr>
<tr>
<td></td>
<td><code>diff ← r1</code></td>
</tr>
<tr>
<td></td>
<td><code>T0 stores 1 to diff</code></td>
</tr>
<tr>
<td></td>
<td><code>T1 stores 1 to diff</code></td>
</tr>
</tbody>
</table>

- This set of three instructions must be “atomic”
Mechanisms for preserving atomicity

- Lock/unlock mutex around a critical section
  
  LOCK(mylock);
  
  // critical section
  UNLOCK(mylock);

- 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);
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) {
    int threadId = getThreadId();
    int myMin = 1 + (threadId * n / NUM_PROCESSORS);
    int myMax = myMin + (n / NUM_PROCESSORS);

    while (!done) {
        diff = 0.f;
        barrier(myBarrier, NUM_PROCESSORS);
        for (j=myMin to myMax) {
            for (i = red cells in this row) {
                float prev = A[i,j];
                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);
    }
}

Grid solver example from: Culler, Singh, and Gupta

Do you see a potential performance problem with this implementation?
Shared address space solver (SPMD execution model)

```c
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];
                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);
    }
}
```
Barrier synchronization primitive

- `barrier(num_threads)`
- Barriers are a conservative way to express dependencies
- Barriers divide computation into phases
- All computations 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
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+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);
    }
}

Grid solver example from: Culler, Singh, and Gupta
Shared address space solver: one barrier

```c
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 example from: Culler, Singh, and Gupta
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
Message passing model of communication
Message passing model (abstraction)

- Threads operate within their own private address spaces
- Threads communicate by sending/receiving messages
  - send: specifies recipient, buffer to be transmitted, and optional message identifier ("tag")
  - receive: sender, specifies buffer to store data, and optional message identifier
  - Sending messages is the only way to exchange data between threads 1 and 2

Illustration adopted from Culler, Singh, Gupta
I will wait to discuss the message passing version of the solver program in a later class.
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 (in practice, they are very inter-related)

- **Focus today: identifying dependencies**

- **Focus soon: identifying locality, reducing synchronization**