Shared Memory Parallel Programming
Pthread/ OpenMP Examples

Kunle Olukotun
Stanford University

(C) 2011 Kunle Olukotun

Units of Measure

- High Performance Computing (HPC) units are:
  - Flop: floating point operation, usually double precision unless noted
  - Flop/s: floating point operations per second
  - Bytes: size of data (a double precision floating point number is 8)

- Typical sizes are millions, billions, trillions…

– Mega	Mflop/s = 106 flop/sec	Mbyte = 220 = 1048576 ~ 106 bytes
– Giga	Gflop/s = 109 flop/sec	Gbyte = 230 ~ 109 bytes
– Tera	Tflop/s = 1012 flop/sec	Tbyte = 240 ~ 1012 bytes
– Peta	Pflop/s = 1015 flop/sec	Pbyte = 250 ~ 1015 bytes
– Exa	Eflop/s = 1018 flop/sec	Ebyte = 260 ~ 1018 bytes
– Zetta	Zflop/s = 1021 flop/sec	Zbyte = 270 ~ 1021 bytes
– Yotta	Yflop/s = 1024 flop/sec	Ybyte = 280 ~ 1024 bytes

- Current fastest (public) machine ~ 4.7 Pflop/s
  - Up-to-date list at www.top500.org

(C) 2011 Kunle Olukotun

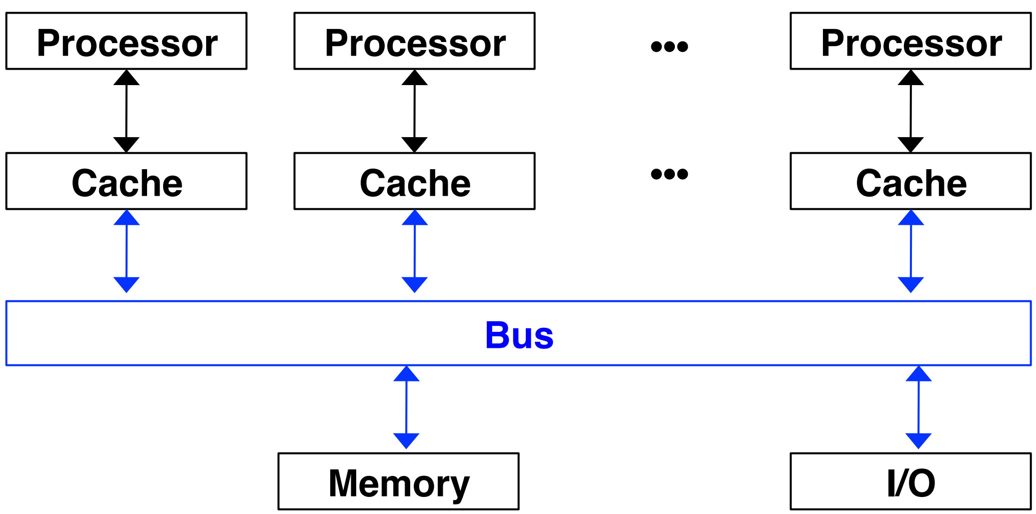
2

Shared Memory Architecture

- Architecture
  - Shared address space
  - Synchronization
- Thread Programming APIs
  - Pthreads
  - Open MP

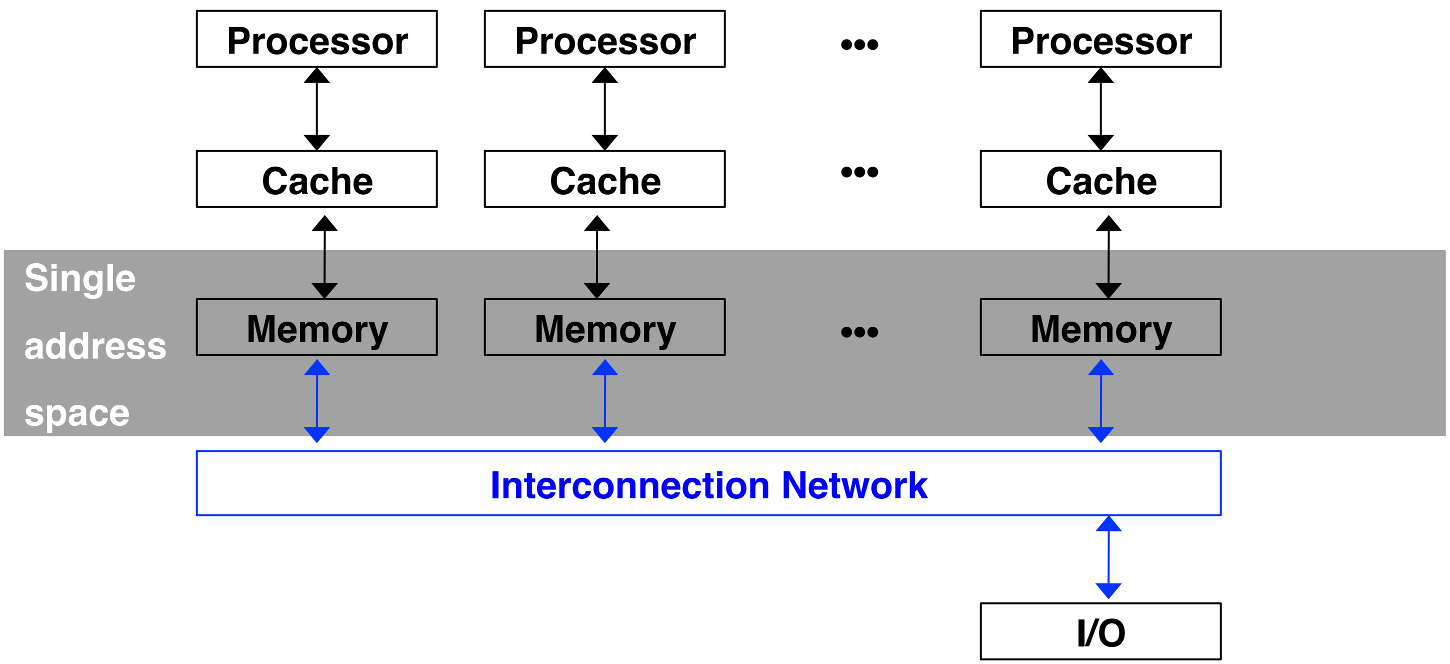
Shared Memory

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks
  - Memory access time
    - UMA (uniform) vs. NUMA (nonuniform)



Shared Memory

- Memory access time
  - UMA (uniform) vs. NUMA (nonuniform)
  - Local vs. remote
  - Why build NUMA?



What Makes Parallel Programming Difficult?

- Finding independent tasks
- Mapping tasks to parallel execution units
- Implementing synchronization
  - Races, livelocks, deadlocks, …
- Composing parallel tasks
- Recovering from HW & SW errors
- Optimizing locality and communication
- Predictable performance & scalability
- … and all the sequential programming issues

(C) 2011 Kunle Olukotun

Focus on Two Steps

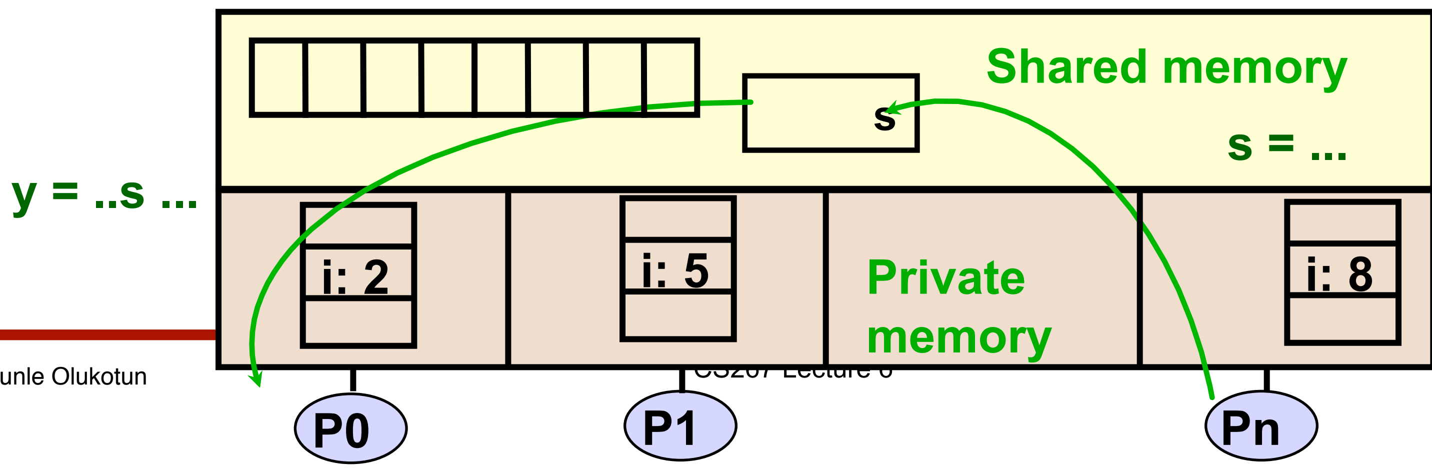
- Finding independent tasks
- Mapping tasks to parallel execution units
- Implementing synchronization
  - Races, livelocks, deadlocks, …
- Composing parallel tasks
- Recovering from HW & SW errors
- Optimizing locality and communication
- Predictable performance & scalability
- … and all the sequential programming issues

Rest of Today's Outline

- Threads: Mapping tasks to parallel execution units
  - What is a thread?
  - How can we use them for parallel programming?
- Synchronization: How we control access to shared memory
  - Protecting critical variable accesses
  - Variations on basic locks
  - Barriers
- Mechanics of pthreads and OpenMP usage

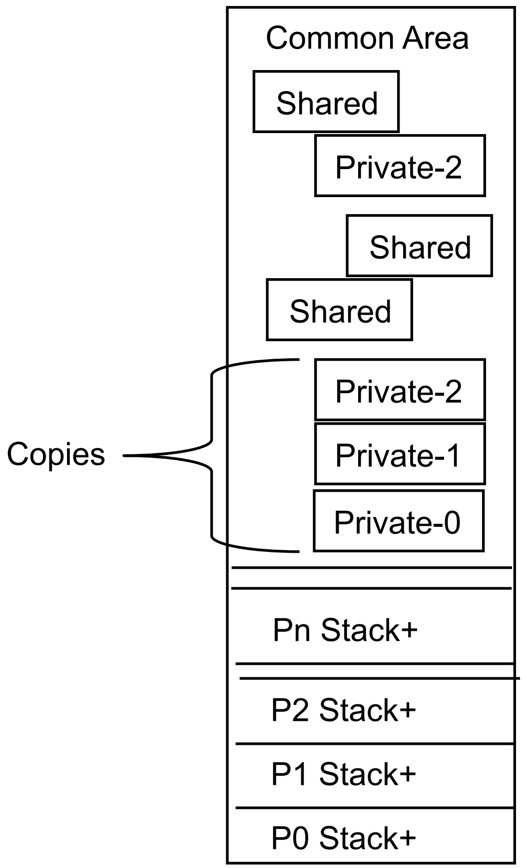
Shared Memory Programming Model

- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables



“Lightweight” Thread Model

Common **Virtual** Address Space



- Each thread is just a PC, registers, and stack
  - Often made with `pthread_create()`
- Usually *all* memory shared
  - Same page table, so no separation
  - Globals are completely shared
- **Pros:**
  - Easier sharing
    - No need for shared memory segment calls
    - Now pointer usage controls sharing
  - *Much* less OS overhead (factor 10–100x)
- **Con:** Non-shared data just by copying vars
  - Pointer errors may be able to corrupt other processors’ data (ouch!)

(C) 2011 Kunle Olukotun

Shared Memory Programming

- Several Thread Libraries/systems
- PTHREADS is the POSIX Standard
  - Relatively low level
  - Portable but possibly slow
- OpenMP standard for application level programming
  - Support for scientific programming on shared memory
  - http://www.openMP.org
- TBB: Thread Building Blocks
  - Intel
- CILK: Language of the C "ilk"
  - Lightweight threads embedded into C
- Java threads
  - Built on top of POSIX threads
  - Object within Java language

Common Notions of Thread Creation

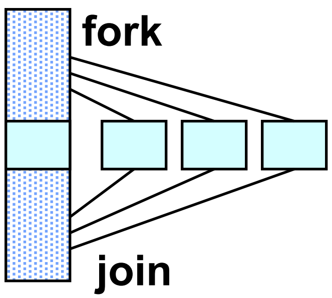
- cobegin/coend
  - cobegin
  - job1(a1);
  - job2(a2);
  - coend
- fork/join
  - tid1 = fork(job1, a1);
  - job2(a2);
  - join tid1;

> - Statements in block may run in parallel
> - cobegins may be nested
> - Scoped, so you cannot have a missing coend

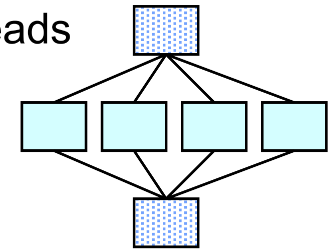
> - Forked procedure runs in parallel
> - Wait at join point if it' s not finished

So How Do We Use Threads?

- First, figure out where there is parallel work in an application
  - Main topic of the next two lectures



- Next, choose a programming model
  - **Pthreads**: Low-level threading *library*
    - Uses fork-join model, like processes
    - Allows arbitrary code division
  - **OpenMP**: *Compiler directives* for parallel programming
    - Uses “parallel region” model to simplify threads
    - Divide up iterations of data parallel loops among threads
    - Is often much easier to use, but not as general
  - *Others:* Many other choices are available
    - System-specific threads: Solaris, Windows, etc.
    - System-specific directives: Solaris compilers have them
    - Parallel languages: Java, HP Fortran, UPC, Cilk, Titanium, etc.



Overview of POSIX Threads

- POSIX: Portable Operating System Interface for UNIX
  - Interface to Operating System utilities
- PThreads: The POSIX threading interface
  - System calls to create and synchronize threads
  - Should be relatively uniform across UNIX-like OS platforms
- PThreads contain support for
  - Creating parallelism
  - Synchronizing
  - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread

Forking Posix Threads

Signature:

```
int pthread_create(pthread_t *,
                   const pthread_attr_t *,
                   void * (*)(void *),
                   void *);
```

Example call:

```
errcode = pthread_create(&thread_id; &thread_attribute
                         &thread_fun; &fun_arg);
```

- thread_id is the thread id or handle (used to halt, etc.)
- thread_attribute various attributes
  - Standard default values obtained by passing a NULL pointer
    attribute: minimum stack size
- thread_fun the function to be run (takes and returns void*)
- fun_arg an argument can be passed to thread_fun when it starts
- errorcode will be set nonzero if the create operation fails

Simple Threading Example

Compile using gcc –lpthread

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```


(C) 2011 Kunle Olukotun

Loop Level Parallelism

- Many scientific application have parallelism in loops
  - With threads:

```
… my_stuff [n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    … pthread_create (update_cell[i][j], …,
                      my_stuff[i][j]);
```

- But overhead of thread creation is nontrivial
  - update_cell should have a significant amount of work
  - 1/pth if possible

Some More Pthread Functions

- **`pthread_yield();`**
  - Informs the scheduler that the thread is willing to yield its quantum, requires no arguments.
- **`pthread_exit(void *value);`**
  - Exit thread and pass value to joining thread (if exists)
- **`pthread_join(pthread_t *thread, void **result);`**
  - Wait for specified thread to finish. Place exit value into *result.

Others:

- **`pthread_t me; me = pthread_self();`**
  - Allows a pthread to obtain its own identifier pthread_t thread;
- **`pthread_detach(thread);`**
  - Informs the library that the threads exit status will not be needed by subsequent pthread_join calls resulting in better threads performance. For more information consult the library or the man pages, e.g., man -k pthread..

(C) 2011 Kunle Olukotun

Shared Data and Threads

- Variables declared outside of main are shared
- Object allocated on the heap may be shared (if pointer is passed)
- Variables on the stack are private: passing pointer to these around to other threads can cause problems

- Often done by creating a large "thread data" struct
  - Passed into all threads as argument
  - Simple example:

```
char *message = "Hello World!\n";

pthread_create( &thread1,
      NULL,
      (void*)&print_fun,
      (void*) message);
```

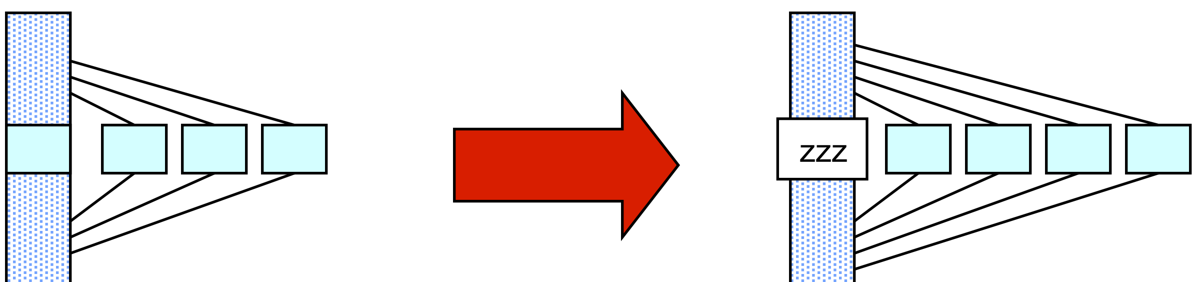
(C) 2011 Kunle Olukotun

Pthreads Example Implementation

```
#include <pthread.h>
main(){
    pthread_t p_threads[NUM_PROCS];
    input_buffer_struct_t input[NUM_PROCS]; /* Minimally, a processor ID */
    output_buffer_struct_t *output_bufptr;
    . . .
    for (i=1; i < NUM_PROCS; i++)
         pthread_create(&p_threads[i], &attr, Parallel_Work,
                  (void *) &input[i]);
    Parallel_Work(&(input[0])); /* Optional */
    for (i=1; i < NUM_PROCS; i++)
         pthread_join(p_threads[i], &output_bufptr);
    . . .
}
void *Parallel_Work(void *input_buffer) {
    /* Parallel Work Here */
    return &output_buffer;
}
```

Subtle Pthreads Details I

- Number of threads ≠ number of processors
  - You *can* have a different number of threads and processors
  - Don't really need for simple parallel loops . . .
    - Just wastes memory & causes overhead
  - But useful in some cases:
    - Arbitrary forked-off parallel tasks (like database queries)
    - "Sleeping" master thread
      - Master is "special," so allows all parallel workers to be *identical*



- A library, so just link to it (may require extra compiler flag)

Subtle Pthreads Details II

- You *can* pass data in/out through I/O pointers
  - Good for “processor ID,” at least
  - Other values can be passed through shared variables easier

- Thread characteristics are controlled through an attributes struct
  - Similar to a C++ object
  - Set your preferences before creation

- Threads normally terminate when the parallel function returns
  - But you can end earlier with `pthread_exit()` (for self-kill) or `pthread_cancel()` (for “killing” other threads)
    - Just like UNIX `exit()` and `kill()` calls

(C) 2011 Kunle Olukotun

Setting Attribute Values

- Once an initialized attribute object exists, changes can be made. For example:
  - To change the stack size for a thread to 8192 (before calling pthread_create), do this:
    - pthread_attr_setstacksize(&my_attributes, (size_t)8192);
  - To get the stack size, do this:
    - size_t my_stack_size;
      pthread_attr_getstacksize(&my_attributes, &my_stack_size);
- Other attributes:
  - Detached state – set if no other thread will use pthread_join to wait for this thread (improves efficiency)
  - Guard size – use to protect against stack overfow
  - Inherit scheduling attributes (from creating thread) – or not
  - Scheduling parameter(s) – in particular, thread priority
  - Scheduling policy – FIFO or Round Robin
  - Contention scope – with what threads does this thread compete for a CPU
  - Stack address – explicitly dictate where the stack is located
  - Lazy stack allocation – allocate on demand (lazy) or all at once, "up front"

Introduction to OpenMP

- What is OpenMP?
  - Open specification for Multi-Processing
  - “Standard” API for defining multi-threaded shared-memory programs
  - openmp.org – Talks, examples, forums, etc.

- High-level API
  - Preprocessor (compiler) directives (~ 80%)
  - Library Calls (~ 19%)
  - Environment Variables (~ 1%)

A Programmer’s View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming *specification* with “light” syntax
  - Exact behavior depends on OpenMP *implementation*!
  - Requires compiler support (C or Fortran)

- OpenMP will:
  - Allow a programmer to separate a program into *serial regions* and *parallel regions,* rather than T concurrently-executing threads*.*
  - Hide stack management
  - Provide synchronization constructs

- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races

Motivation – OpenMP

```
int main() {



  // Do this part in parallel

  printf( "Hello, World!\n" );

  return 0;
}
```

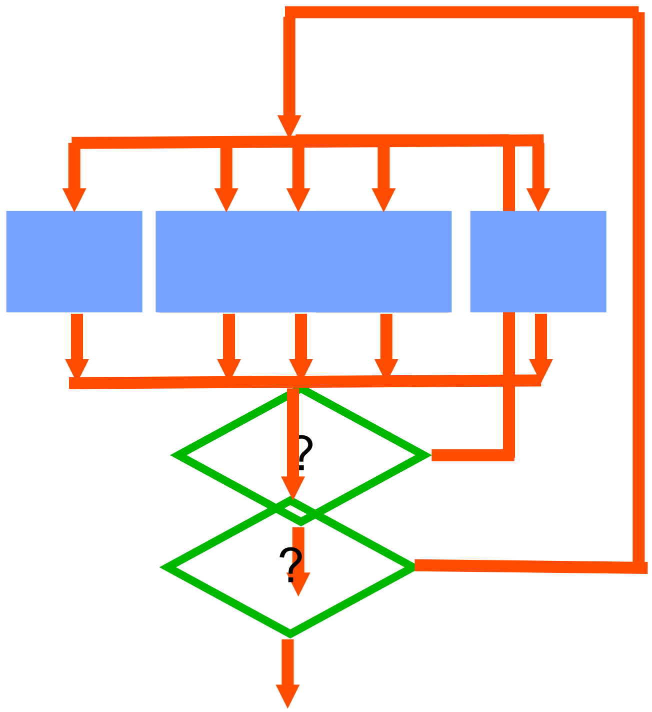
Motivation – OpenMP

```
int main() {

  omp_set_num_threads(16);

  // Do this part in parallel
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```

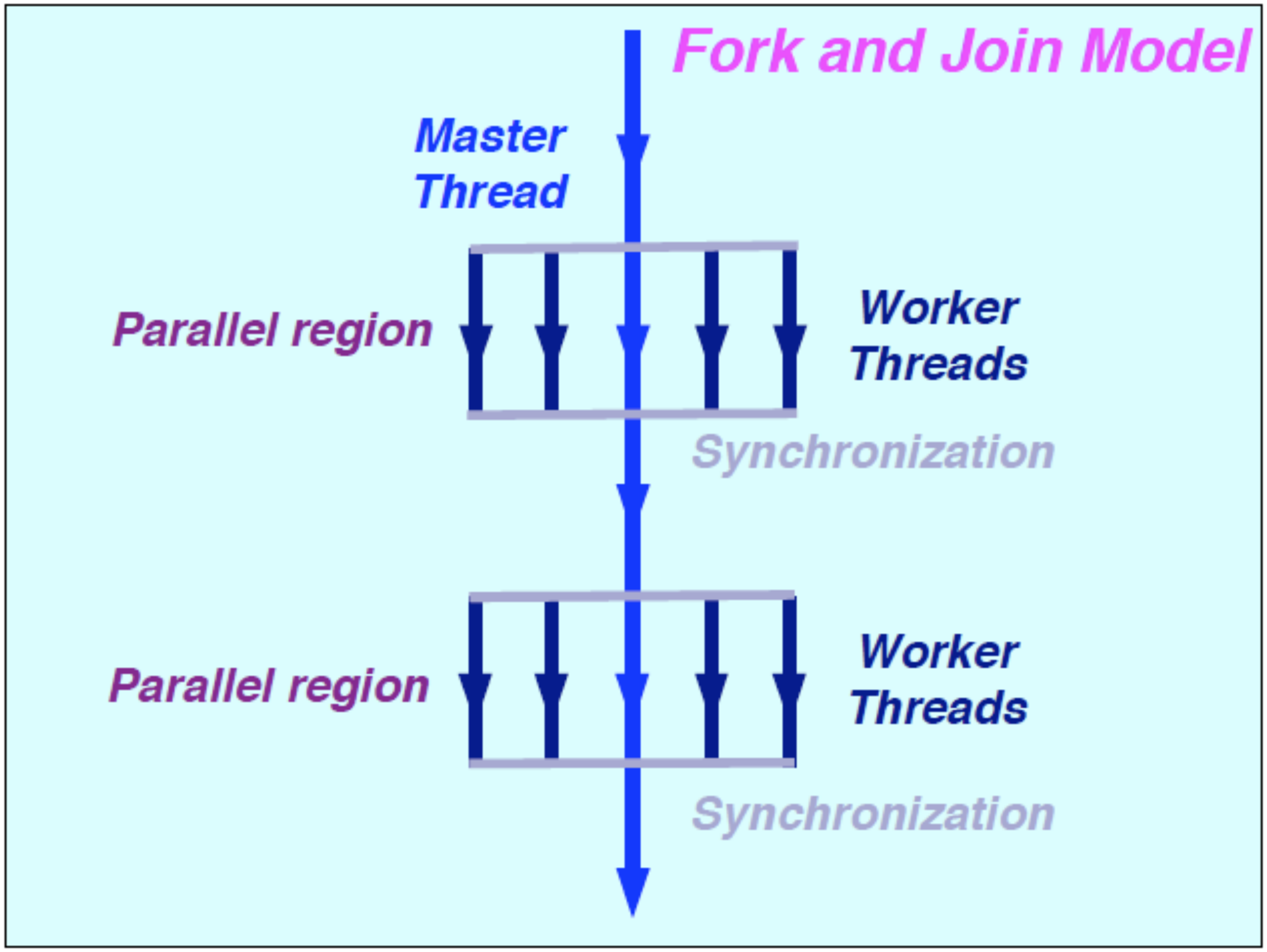
Programming Model – Concurrent Loops

- OpenMP easily parallelizes loops
  - Requires: No data dependencies (reads/write or write/write pairs) between iterations!
- Preprocessor calculates loop bounds for each thread directly from *serial* source

```
#pragma omp parallel for

for( i=0; i < 25; i++ ) {

  printf("Foo");

}
```



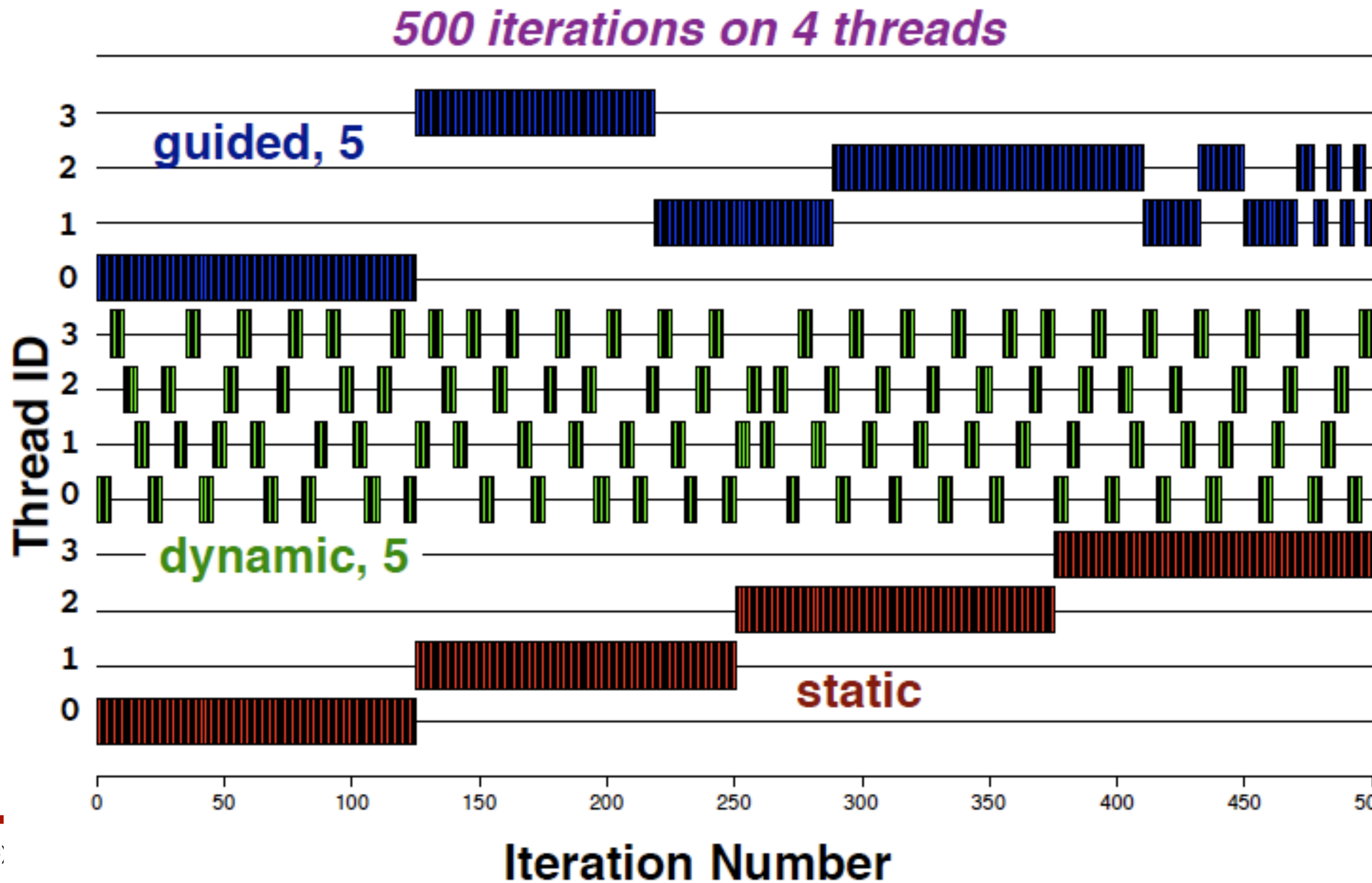
OpenMP Execution Model



Programming Model – Loop Scheduling

- `schedule` clause determines how loop iterations are divided among the thread team
  - **`static([chunk])`** divides iterations statically between threads
    - Each thread receives `[chunk]` iterations, rounding as necessary to account for all iterations
    - Default **`[chunk]`** is **`ceil( # iterations / # threads )`**
  - **`dynamic([chunk])`** allocates **`[chunk]`** iterations per thread, allocating an additional **`[chunk]`** iterations when a thread finishes
    - Forms a logical work queue, consisting of all loop iterations
    - Default **`[chunk]`** is 1
  - **`guided([chunk])`** allocates dynamically, but **`[chunk]`** is exponentially reduced with each allocation

OpenMP Scheduling



Programming Model – Data Sharing

- Parallel programs often employ two types of data
  - Shared data, visible to all threads, similarly named
  - Private data, visible to a single thread (often stack-
- PThreads:
  - Global-scoped variables are shared
  - Stack-allocated variables are private
- OpenMP:
  - **shared** variables are shared
  - **private** variables are private

```
// shared, globals
int bigdata[1024];

void* foo(void* bar) {
  int tid;

  #pragma omp parallel \
   shared ( bigdata ) \
   private ( tid )
  {
    /* Calc. here */
  }
}
```

(C) 2011 Kunle Olukotun

OpenMP Directives

- C: directives are case sensitive
  - Syntax: #pragma omp directive [clause [clause] ...]
- Continuation: use \ in pragma

OpenMP Example Implementation

```
#include <omp.h>
main(){
   . . .
  #pragma omp parallel for default(private) \
      num_threads(NUM_PROCS) . . . << var info >> . . .
  for (i=0; i < NUM_PROCS; i++)
  {
      /* Parallel Work Here */
  }
   . . .
}
```

Important!!

- Simpler than pthreads for this basic `for` example
  - But harder for less structured parallelism (like webservers)
  - Just “attaches” to the following `for` loop & runs it in parallel
  - Be careful: These are *preprocessor directives!*

Privatizing Variables

- Critical to performance!
- Simple in pthreads: Just use different variables!
  - Easy concept, but can sometimes complicate code
    - May require many `variable[processor_id]`-like accesses
- More work in OpenMP pragmas:
  - Designed to make parallelizing sequential code easier
  - Makes copies of "private" variables *automatically*
    - And performs some automatic initialization, too
  - Must specify shared/private per-variable in `parallel`
    - `private`: Uninitialized private data
    - `first/lastprivate`: Private, initialize@input & output@end
    - `shared`: All-shared data
    - `threadprivate`: "static" private for use across several `parallel` regions

(C) 2011 Kunle Olukotun

Firstprivate/Lastprivate Clauses

- firstprivate (list)
  - All variables in the list are initialized with the value the original object had before entering the parallel region

- lastprivate(list)
  - The thread that executes the last iteration or section in sequential order updates the value of the objects in the list

Example Private Variables

```
main()
{
  A = 10;

 #pragma omp parallel
 {
   #pragma omp for private(i) firstprivate(A) lastprivate(B)...
   for (i=0; i<n; i++)
   {
        ....
        B = A + i;         /*-- A undefined, unless declared
                                firstprivate --*/

        ....
   }

   C = B;                  /*-- B undefined, unless declared
                                lastprivate --*/


 } /*-- End of OpenMP parallel region --*/
}
```

Sections Example

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
       for (i=0; i<n-1; i++)
           b[i] = (a[i] + a[i+1])/2;

      #pragma omp section
       for (i=0; i<n; i++)
           d[i] = 1.0/c[i];

    } /*-- End of sections --*/

  } /*-- End of parallel region --*/
```

(C) 2011 Kunle Olukotun

Loop Level Parallelism with OMP

- Consider the single precision vector add-multiply operation
  Y=a**X**+**Y** ("SAXPY")

```
for (i=0;i<n;++i) {
  Y[i]+=a*X[i];
}
```

```
#pragma omp parallel for \
     private(i) shared(X,Y,n,a)
for (i=0;i<n;++i) {
  Y[i]+=a*X[i];
}
```


(C) 2011 Kunle Olukotun

OpenMP Sections

- Parallel threads can also do different things with `sections`
  - Use instead of `for` in the `pragma`, and no attached loop
  - Contains several `section` blocks, one per thread
- You can also have a "multi-part" parallel region
  - Allows easy alternation of serial & parallel parts
  - Doesn't require re-specifying # of threads, etc.

```
#pragma omp parallel . . .
{
 #pragma omp for
 . . . Loop here . . .
 #pragma omp single
 . . . Serial portion here . . .
 #pragma omp sections
 . . . Sections here . . .
}
```

```
#pragma omp sections
{
    #pragma omp section
    { taskA(); }
    #pragma omp section
    { taskB(); }
}
```

Data Race Example

```
static int s = 0;
```

```
Thread 1
  for i = 0, n/2-1
      s = s + f(A[i])
```

```
Thread 2
  for i = n/2, n-1
      s = s + f(A[i])
```

- Problem is a race condition on variable s in the program
- A race condition or data race occurs when:
  - two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

Race Conditions: A Concurrency Problem

- We must be able to *control* access to *shared* memory
  - Unpredictable results called races can happen if we don't
  - (e.g. `x++`)

Thread 1	Thread 2	
ld r1, x	**. . .**	**x = 0**
add r1, r1, 1	**ld r1, x**	
—	**add r1, r1, 1**	
—	**st r1, x**	
st r1, x	**. . .**	**what's the value of x?**

Dealing with Race Conditions

- Need mechanism to ensure updates to single variables occur within a *critical section*
- Any thread entering a critical section blocks all others
- Critical sections can be established by using:
  – Lock variables (single bit variables)
  – Semaphores (Dijkstra 1968)
  – Monitor procedures (Hoare 1974, used in Java)

(C) 2011 Kunle Olukotun

Coordinating Access to Shared Data: Locks

Thread 1	Thread 2
ld r1, x	. . .
add r1, r1, 1	ld r1, x
—	add r1, r1, 1
—	st r1, x
st r1, x	. . .

Thread 1	Thread 2
LOCK X	. . .
ld r1, x	LOCK X
add r1, r1, 1	stall
st r1, x	stall
UNLOCK X →	unstall
	ld r1, x
	etc.

- Locks are a simple primitive to assert control
  - Put lock/unlock (acquire/release) pair *around* each critical region
  - Basis of all more complex variable control & synchronization
    - Semaphores, monitors, condition variables

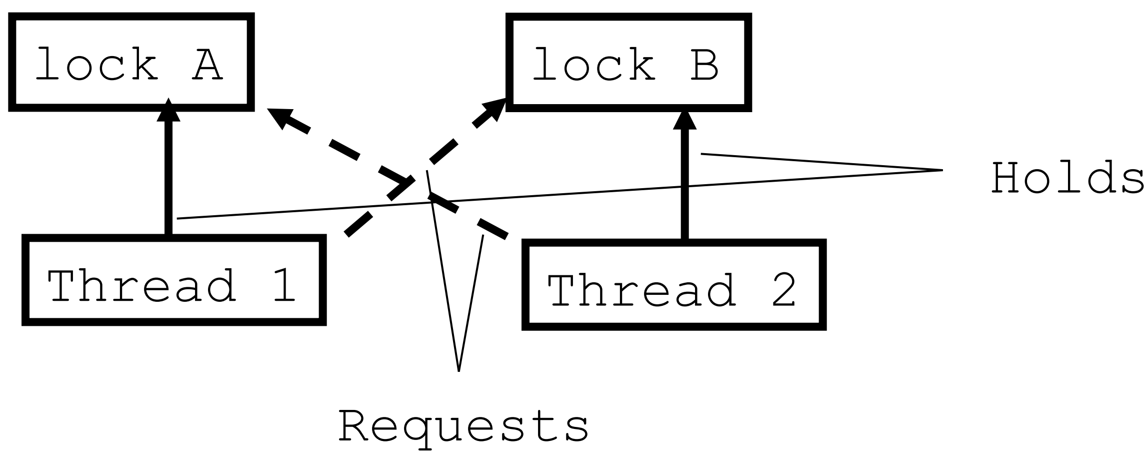
(C) 2011 Kunle Olukotun

"Fun" with Locking

- Basic idea of locks is simple:
  - Assign a lock to each shared variable (or variable groups)
  - *Initialize* the lock before you use it
  - *Always* use the lock when you access shared state
- But details can get tricky:
  - Need to minimize the time processors spend stalled
  - Need to carefully select groupings of variables
    - Want to minimize # of locks to reduce overhead
    - But want to maximize available parallelism
  - Must be careful to always nest lock acquires correctly
    - Can cause **deadlock** if you're not careful!
  - *Moral:* Privatize as much as possible to avoid locking!

Deadlocks: The pitfall of locking

- Must ensure a situation is not created where requests in possession create a deadlock:



- Nested locks are a classic example of this
- Can also create problem with multiple processes - `deadly embrace'

(C) 2011 Kunle Olukotun

Locks: Performance vs. Correctness

- Few locks
  - Coarse grain locking
  - Easy to write parallel program
  - Processors spend a lot of time stalled waiting to acquire locks
  - Poor performance
- Many locks
  - Fine grain locking
  - Difficult to write parallel program
  - Higher chance of incorrect program (deadlock)
  - Good performance? (higher lock overheads)
- Parallel programming difficulty
  - How do you know what level of lock granularity to use?
  - Will discuss further in upcoming lectures . . . .

Non-blocking Locks

- Structuring parallel programs correctly will be our main weapon against lock stall overhead
- But another one is *non-blocking* locks
  - Try to grab the lock, if possible
  - Do other, non-critical work if you can't get it

```
while (nonblocking_lock(&lock) != GOT_LOCK) {
 /* Do something else non-critical */
}
/* critical region here */
unlock(&lock);
```

- Performance is limited by availability of non-critical work

Locks in Pthreads and OpenMP

- Both have *equivalent* lock APIs:

Lock Task	Pthreads Version	OpenMP Version
Lock Object Type	`pthread_mutex_t`	`omp_lock_t`
Initialize New Lock	`pthread_mutex_init`	`omp_init_lock`
Destroy Lock	`pthread_mutex_destroy`	`omp_destroy_lock`
Blocking Lock Acquire	`pthread_mutex_lock`	`omp_set_lock`
Lock Release	`pthread_mutex_unlock`	`omp_unset_lock`
Non-blocking Lock Acquire	`pthread_mutex_trylock`	`omp_test_lock`

- For programming assignments, can define some macros and switch between the two with a `#define`

Other Types of Synchronization

- We often want to control sequencing of parts of threads:
  - To impose a sequential order on a code block
    - Threads must execute code in sequential order
  - To control producer-consumer access to data
    - Producer signals consumer when output is ready
    - Consumer signals producer when it needs more input
  - To globally get all processors to the same point in the program
    - Divides a program into easily-understood phases
    - Generally called a barrier

(C) 2011 Kunle Olukotun

Simple Problem

```
for i = 1 to N
        A[i] = (A[i] + B[i]) * C[i]
        sum = sum + A[i]
```

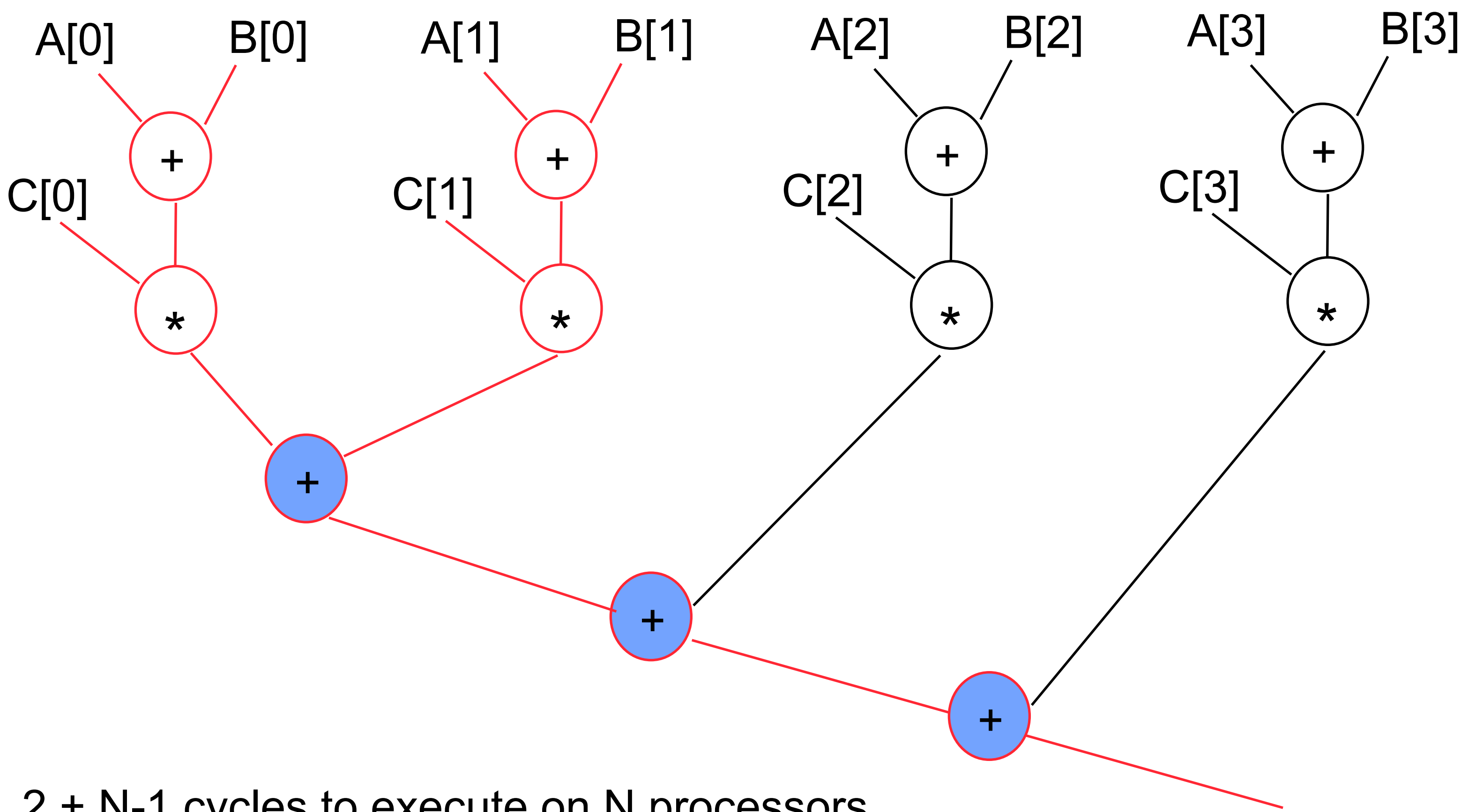
- Split the loops
  - Independent iterations

```
for i = 1 to N
        A[i] = (A[i] + B[i]) * C[i]
for i = 1 to N
        sum = sum + A[i]
```

- Data flow graph?

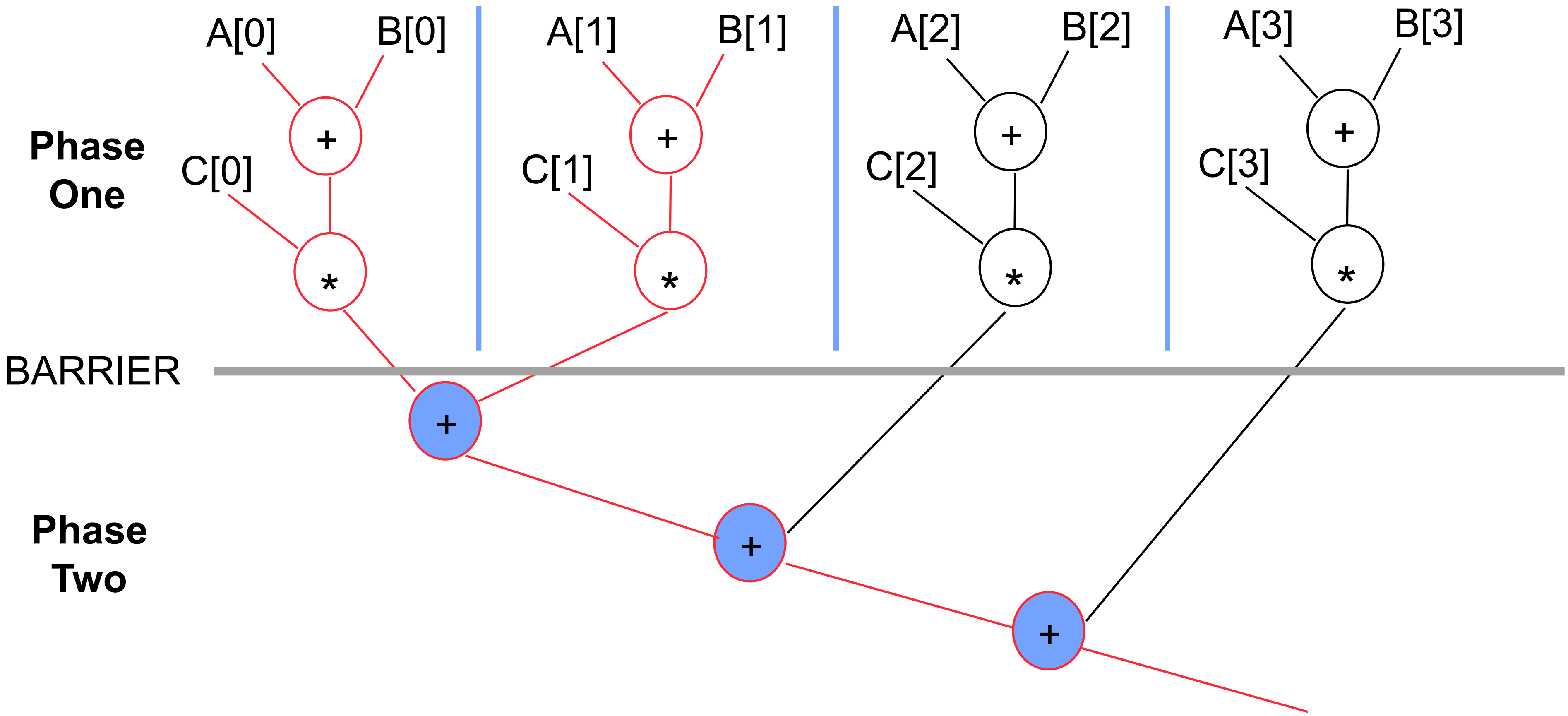
(C) 2011 Kunle Olukotun

Data Flow Graph



2 + N-1 cycles to execute on N processors
what assumptions?

Partitioning of Data Flow Graph



Barriers: Pros & Cons

- **Pro:** Program phases *ease debugging*
  - Eliminates cases of processors in different code regions
    - Otherwise we may have to consider nasty race conditions!
  - Generally easier to reason about
- **Pro:** Program phases *reduce the need for locks*
  - Only need to use the strongest type of lock *for that phase*
    - Normal or full R/W when many/everyone is modifying
    - Switch to single writer or read-only when possible
    - *Example:* A[i] array is *read-only* in phase 2 of example!
  - Can eliminate most of lock overhead for large structures
- **Con:** OVERHEAD
  - "Fast" processors are stalled waiting at the barrier
  - Barrier code itself can be expensive

Programming Model - Synchronization

- OpenMP Synchronization
  - OpenMP Critical Sections
    - Named or unnamed
    - No *explicit* locks / mutexes

```
#pragma omp critical
{
  /* Critical code here */
}
```

  - Barrier directives

```
#pragma omp barrier
```

  - Explicit Lock functions
    - When all else fails – may require `flush` directive

```
omp_set_lock( lock l );
/* Code goes here */
omp_unset_lock( lock l );
```

  - Single-thread regions *within* parallel regions
    - **`master, single`** directives

```
#pragma omp single
{
  /* Only executed once */
}
```

OpenMP Synchronization

- OpenMP provides for a few useful “common cases”
- `barrier` implements an arbitrary barrier
  - A barrier anyplace in one line!
  - Note that many other primitives *implicitly* add barriers, too
- `ordered` locks *and* sequences a block
  - Acts like a lock around a code block
  - Forces loop iterations to run block in “loop iteration” order
  - Only one allowed per loop
  - Good for handling reductions manually, when necessary
    - `sum[i] = sum[i-1];`
- `single`/`master` force only *one* thread to execute a block
  - Acts like a lock
  - Only allows one thread to run the critical code
  - Good for computing a common, global value or handling I/O

Locks in Pthreads and OpenMP

- Both have *equivalent* lock APIs:

Lock Task	Pthreads Version	OpenMP Version
Lock Object Type	`pthread_mutex_t`	`omp_lock_t`
Initialize New Lock	`pthread_mutex_init`	`omp_init_lock`
Destroy Lock	`pthread_mutex_destroy`	`omp_destroy_lock`
Blocking Lock Acquire	`pthread_mutex_lock`	`omp_set_lock`
Lock Release	`pthread_mutex_unlock`	`omp_unset_lock`
Non-blocking Lock Acquire	`pthread_mutex_trylock`	`omp_test_lock`

- For programming assignments, can define some macros and switch between the two with a `#define`
  - Avoid the OpenMP `critical` directive unless you're OpenMP-only (just a lock, but completely different syntax)

Reductions

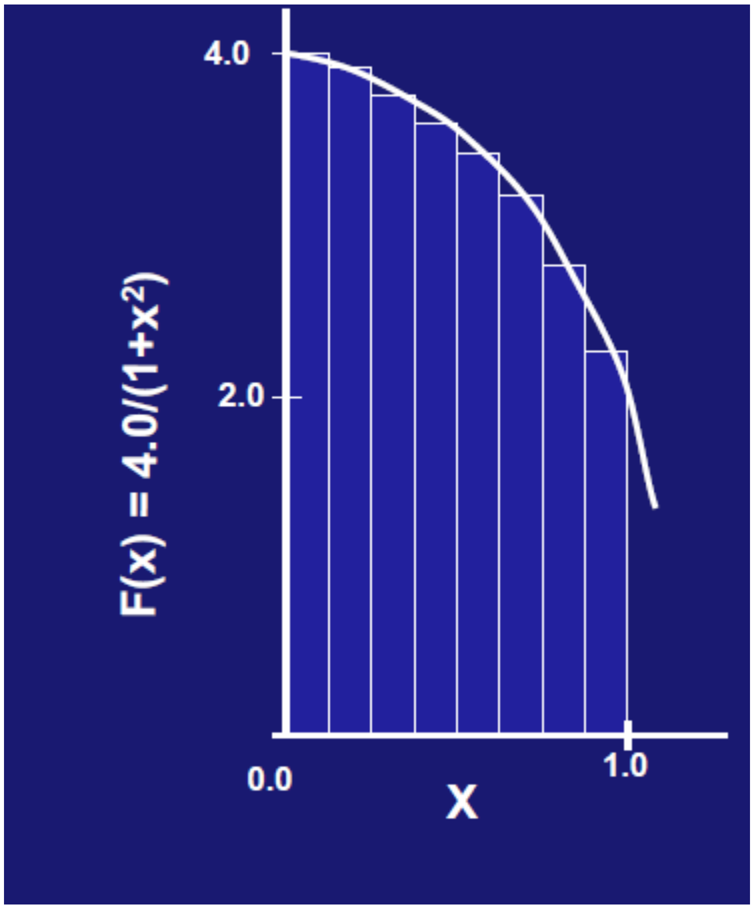
- One of the banes of parallelism is a *reduction* in dimensionality
  - Go from N dimensions to N-1, N-2, . . . 0
  - Dot products are the most common example
    - a[i] = a[i] + b[j] x c[j]
- Single output, *associative* reduction
  - Combine to P elements
    - Do as much of the reduction in parallel as possible
  - Do final step in serial (small P) or in a parallel tree (large P)
- Single output, *non-associative* reduction
  - It's serial, so try to overlap *parts* of tasks
  - Good place to apply dataflow/pipeline parallelism!

(C) 2011 Kunle Olukotun

Reductions in OpenMP

- Reductions are so common that OpenMP provides support for them
- May add reduction clause to parallel for pragma
- Specify reduction operation and reduction variable
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop
- The reduction clause has this syntax:
  reduction (<op> :<variable>)
- Operators
  - + Sum
  - * Product
  - &, |, ^ Bitwise and, or , exclusive or
  - &&, || Logical and, or

Example: Numerical Integration



- We know mathematically that

$$\pi = \int_0^1 \frac{4.0}{(1+x^2)}\, dx$$

- We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Sequential Pi Computation

```
static long num_steps = 100000;
double step;

void main () {
  int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  for (i=0;i< num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

Open MP Parallelized Pi Computation

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2

void main (){
  int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for private(x) reduction(+:sum)
  for (i=0;i< num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

- Notice that we haven't changed any lines of code, only added 4 lines

Microbenchmark: Structured Grid

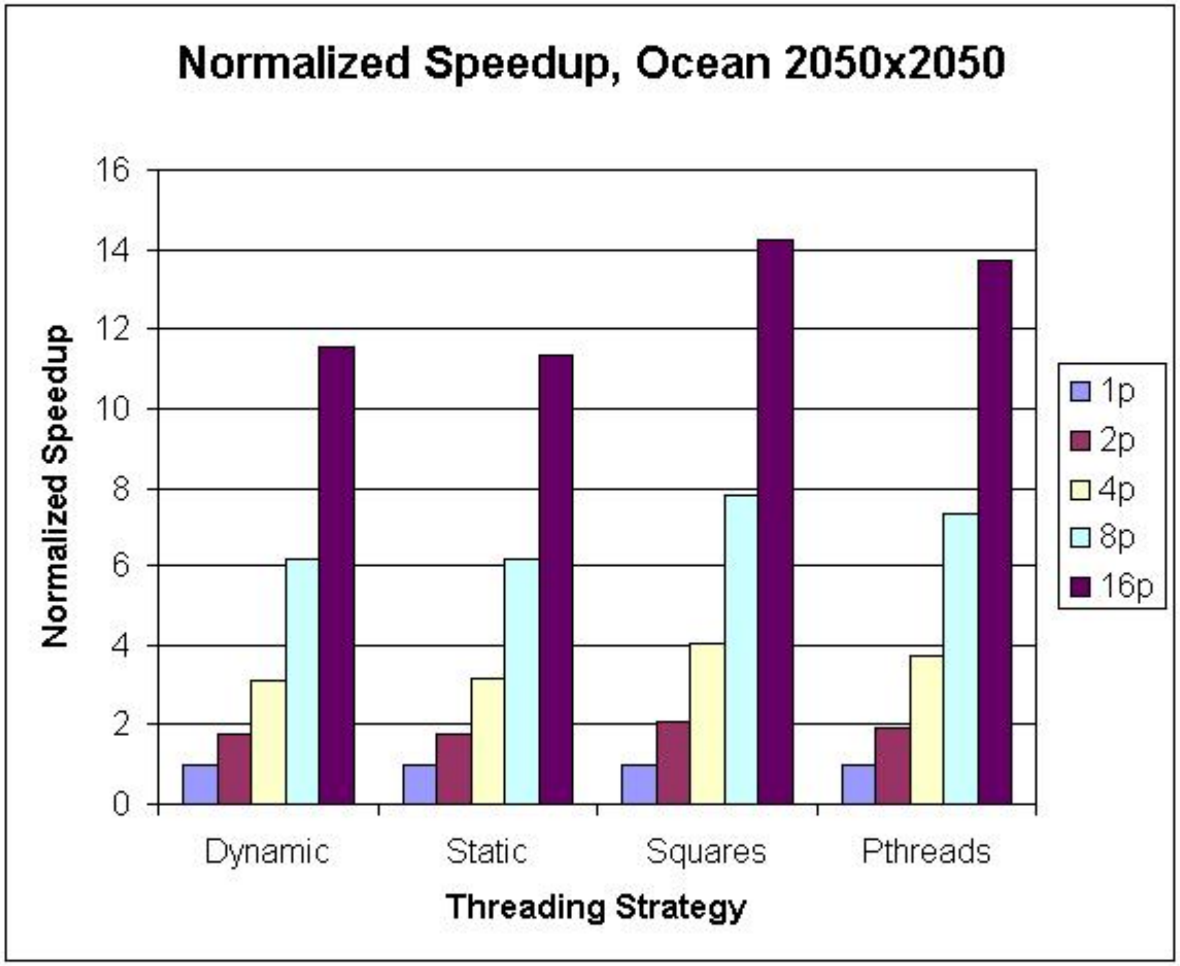
- **`ocean_dynamic`** – Traverses entire ocean, row-by-row, assigning row iterations to threads with `dynamic` scheduling.

- **`ocean_static`** – Traverses entire ocean, row-by-row, assigning row iterations to threads with `static` scheduling.

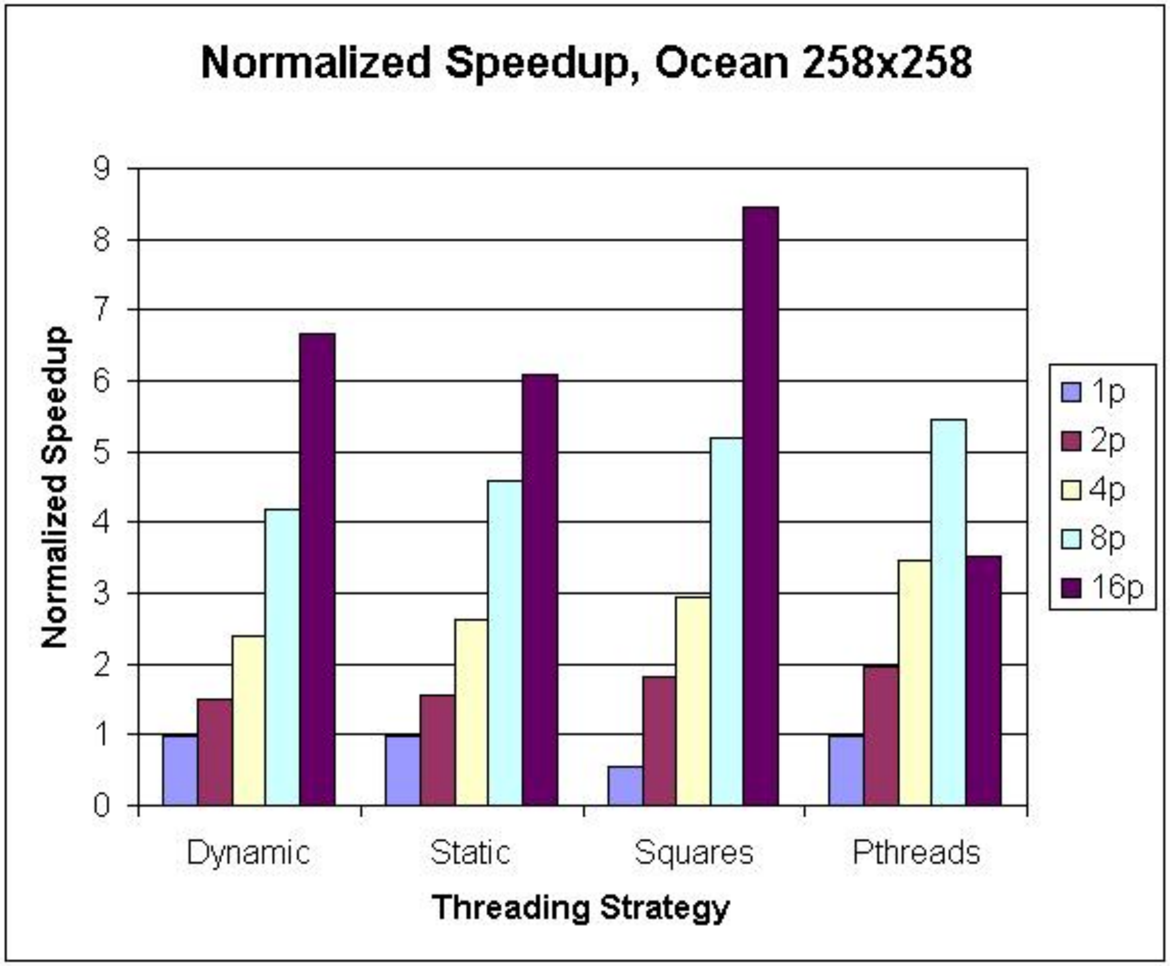
- **`ocean_squares`** – Each thread traverses a square-shaped section of the ocean. Loop-level scheduling not used—loop bounds for each thread are determined explicitly.

- **`ocean_pthreads`** – Each thread traverses a square-shaped section of the ocean. Loop bounds for each thread are determined explicitly.

Microbenchmark: Ocean



Microbenchmark: Ocean



Evaluation

- OpenMP scales to 16-processor systems
  - Was overhead too high?
    - In some cases, yes
  - Did compiler-generated code compare to hand-written code?
    - Yes!
  - How did the loop scheduling options affect performance?
    - dynamic or guided scheduling helps loops with variable iteration runtimes
    - static or predicated scheduling more appropriate for shorter loops

- OpenMP is a good tool to parallelize (at least some!) applications

More Information

- openmp.org
  - OpenMP official site
- www.llnl.gov/computing/tutorials/openMP/
  - A handy OpenMP tutorial
- www.nersc.gov/nusers/help/tutorials/openmp/
  - Another OpenMP tutorial and reference

So How Do We Use Threads?

- Review
  - **Pthreads**: Low-level threading *library*
    - Uses fork-join model, like processes
    - Allows arbitrary code division
  - **OpenMP**: *Compiler directives* for parallel programming
    - Uses "parallel region" model to simplify threads
    - Divide up iterations of data parallel loops among threads
    - Is often much easier to use, but not as general
  - *Others:* Many other choices are available
    - System-specific threads: Solaris, Windows, etc.
    - System-specific directives: Solaris compilers have them
    - Parallel languages: Java, HP Fortran, UPC, Cilk, Titanium, etc.

(C) 2011 Kunle Olukotun

Simple Example: Computing the Inner Product

```
int inner_product (int * A, int * B, int length){
   int i;
   int sum = 0 ;
   for (i = 0; i < length; i++){
        sum += A[i] * B[i];
   }
   return sum;
}
```

Computing the Inner Product Pthreads (first attempt)

```
struct ip_input{
    int * A;
    int * B;
    int length;
    int my_proc;
    int * sum;
}

void * pthread_inner_product (void * v_input){
    struct ip_input my_input = (ip_input) v_input;
    for (int i = 0; i < my_input->length; i++){
         *(my_input->sum) += my_input->A[i] * my_input->B[i];
    return NULL;
}
```

How much work does each thread do?

Simple Example: Computing the Inner Product Creating Pthreads

```
int inner_product (int * A, int * B, int length){
   pthread_t pthreads[num_procs];
   struct ip_input ip_inputs[num_procs];
   void * ip_outputs[num_procs];
   int sum = 0;
   for (int proc = 0; proc < num_procs; proc++){
        int res = pthread_attr_init(&thread_attr[proc]);
        ip_inputs[proc].A = A; ip_inputs[proc].B = B;       ip_inputs
   [proc].length = length;
        ip_inputs[proc].my_proc = proc;
        ip_inputs[proc].sum = &sum;
        pthread_create(&pthreads[proc], &thread_attr[proc],pthread_ip,
                 (void*)&ip_inputs[proc]);
  }
  for ( proc = 0; proc < _num_procs; proc++){
    pthread_join(pthreads[proc], &ip_outputs[proc]);
  }
   return sum;
}
```

(C) 2011 Kunle Olukotun

Computing the Inner Product
Pthreads Second Attempt

```
void * pthread_inner_product (void * v_input){
   struct ip_input my_input = (ip_input) v_input;
   int my_start, my_end;
   int chunk_size = my_input->length / num_procs;
   my_start = my_input->my_proc * chunk_size;
   if (my_input->my_proc == num_procs-1)
        my_end = my_input->length;
   else
        my_end = (my_input->my_proc + 1) * chunk_size;
   for (int i = my_start; i < my_end; i++)
        *(ip_input->sum) += ip_input->A[i] * ip_input->B[i];
   return NULL;
}
```

Will `sum` have the correct value at the end?

Computing the Inner Product Pthreads Third Attempt

```
void * pthread_inner_product (void * v_input){
    struct ip_input my_input = (ip_input) v_input;
    int my_start, my_end;
    int chunk_size = my_input->length/num_procs;
    my_start = my_input->my_proc * chunk_size;
    if (my_input->my_proc == num_procs-1)
         my_end = my_input->length;
    else
         my_end = (my_input->my_proc + 1) * chunk_size;
    for (int i = my_start; i < my_end; i++){
         pthreads_mutex_acquire(my_input->sum_lock);
         *(ip_input->sum) += my_input->A[i] * my_input->B[i];
         pthreads_mutex_release(my_input->sum_lock);
    }
    return NULL;
}
```

Simple Example: Computing the Inner Product Synchronizing Pthreads

```
int inner_product (int * A, int * B, int length){
   pthread_t pthreads[num_procs];
   struct ip_input ip_inputs[num_procs];
   void * ip_outputs[num_procs];
   pthread_mutex_t sum_lock;
   pthread_mutex_init(&sum_lock);
   for (int proc = 0; proc < num_procs; proc++){
        int res = pthread_attr_init(&thread_attr[proc]);
        ip_inputs[proc].A = A; ip_inputs[proc].B = B; ip_inputs
   [proc].length = length;
        ip_inputs[proc].my_proc = proc;
        ip_inputs[proc].sum = &sum;
        ip_inputs[proc].sum_lock = &sum_lock
        pthread_create(&pthreads[proc], &thread_attr[proc],pthread_ip,
                (void*)&ip_inputs[proc]);
  }
  for ( proc = 0; proc < _num_procs; proc++){
    pthread_join(pthreads[proc], &ip_outputs[proc]);
  }
   return sum;
}
```

How well will this code parallelize?

(C) 2011 Kunle Olukotun

Simple Example: Computing the Inner Product Removing Synchronization - Pthreads

```
int inner_product (int * A, int * B, int length){
   int sum = 0;
   pthread_t pthreads[num_procs];
   struct ip_input ip_inputs[num_procs];
   int ip_sums[num_procs];
   for (int proc = 0; proc < num_procs; proc++){
        int res = pthread_attr_init(&thread_attr[proc]);
        ip_inputs[proc].A = A; ip_inputs[proc].B = B;       ip_inputs
   [proc].length = length;
        ip_inputs[proc].my_proc = proc;
        ip_inputs[proc].sum = &ip_sums[proc];
        pthread_create(&pthreads[proc], &thread_attr[proc],pthread_ip,
                (void*)&ip_inputs[proc]);
  }
  for ( proc = 0; proc < _num_procs; proc++){
    pthread_join(pthreads[proc], &ip_outputs[proc]);
    sum += ip_sums[proc];
  }
  return sum;
}
```

Computing the Inner Product
Pthreads Final Attempt

```
void * pthread_inner_product (void * v_input){
   struct ip_input my_input = (ip_input) v_input;
   int my_start, my_end;
   int chunk_size
   *(my_input->sum) = 0;
   my_start = my_input->my_proc * chunk_size;
   if (my_input->my_proc == num_procs-1)
        my_end = my_input->length;
   else
        my_end = (my_input->my_proc + 1) * chunk_size;

   for (int i = my_start; i < my_end; i++){
        *(my_input->sum) += my_input->A[i] * my_input->B[i];
   }
   return NULL;
}
```

Simple Example: Computing the Inner Product OpenMP – first attempt

```
int inner_product (int * A, int * B, int length){
    int i;
    int sum = 0;
    omp_set_num_threads(num_procs);
    #pragma omp parallel private (i)
    {
        #pragma omp for
        for (i = 0; i < length; i++){
            sum += A[i] * B[i];
        }
    }
    return sum;
}
```

Do not need to explicitly break up the work, unlike pthreads.
BUT will `sum` have the correct answer?

(C) 2011 Kunle Olukotun

Simple Example: Computing the Inner Product OpenMP – second attempt

```
int inner_product (int * A, int * B, int length){
   int i;
   int sum = 0;
   omp_lock_t sum_lock;
   omp_init_lock(sum_lock, NULL);
   omp_set_num_threads(num_procs);
   #pragma omp parallel private (i)
   {
        #pragma omp for
        for (i = 0; i < length; i++){
                omp_set_lock (&sum_lock);
                sum += A[i] * B[i];
                omp_unset_lock (&sum_lock);
        }
   }
   return sum;
}
```

How well will this code parallelize?

(C) 2011 Kunle Olukotun

Simple Example: Computing the Inner Product OpenMP – Removing synchronization

```
int inner_product (int * A, int * B, int length){
    int sum = 0;
    int i;
    int my_proc, * my_sum;
    int sums[num_procs];
    omp_set_num_threads(num_procs);
    #pragma omp parallel private (i, my_sum, my_proc)
    {
          my_proc = omp_get_thread_num();
          my_sum = &sums[my_proc];
          *my_sum = 0;
          #pragma omp for
          for (i = 0; i < length; i++){
                    *my_sum += A[i] * B[i];
          }
    }
    for (int proc = 0; proc < num_procs; proc++){
          sum += sums[proc];
    }
    return sum;
}
```