Shared Memory
Parallel Programming
Shared Memory

- **Shared address space**
  - Process based models – private by default
  - Lightweight process & thread model – shared …
  - Directive based model – create & synch. threads

- **Parallelization focus**
  - expressing concurrency
  - synchronization
  - minimizing overheads

- **Paradigm variations**
  - data sharing
  - concurrency model
  - synchronization support
Quiz 2.5.1

• Will every parallelization paradigm work equally well for every application?

No – consider the difference between the traveling salesman problem and weather prediction.
Processes

Threads


Lecture 2.5 – Programming with Shared Memory
Quiz 2.5.2

• Would all threads share the same instruction pointer or would they each have their own?

Each thread will have its own IP so that threads can operate independently (but normally in a cooperative fashion).
Threads

- A *thread* is a single stream of control in the flow of a program.

- Consider the code excerpt

  ```
  for (row=0; row<n; row++)
      for (col=0; col<n; col++)
          c[row][col]=
                      dot(get_row(a,row),get_col(b,col));
  ```

- and the ability to create threads

  ```
  for (row=0; row<n; row++)
      for (col=0; col<n; col++)
          c[row][col]=
              create_thread(dot(get_row(a,row),
                              get_col(b,col)));
  ```
Quiz 2.5.3a

• How many threads does this code create?

```c
for (row=0; row<n; row++)
    for (col=0; col<n; col++)
        c[row][col]=
            create_thread(dot(get_row(a,row),
                              get_col(b,col));

n*n threads
```
Quiz 2.5.3b

• If you want to use two different programs to cooperatively solve a problem, would you use threads or processes? Explain why.

You should use processes as the code is different for the two different programs and you will need to have independent spaces.
Threads

- Parallel Model

logical machine model

reality – distributed shared address space

Origin 2000
Quiz 2.5.4

• How does the difference between the logical view and the reality of parallel architectures affect parallelization?

Performance
Threads

- Advantages over message passing
  - portable from serial to parallel
  - latency hiding
  - scheduling & load balancing
  - ease of use - arguable
  - widespread - POSIX
Quiz 2.5.5

• Do all of the advantages of the shared memory parallelization make it the only choice for parallel implementation? Why or why not?

No, because it does not scale on current architectures.
POSIX

- IEEE standard 1003.1c-1995
  - aka Pthreads
- Compatible with most APIs
  - NT threads
  - Solaris threads
  - Java threads
Thread Basics

- Thread creation
  - `pthread_create`

```c
#include <pthread.h>
int pthread_create(
    pthread_t *thread_handle,
    const pthread_attr_t *attribute,
    void * (*thread_function) (void *),
    void *arg);
```

```c
int pthread_join(pthread_t thread, void **ptr);
```

http://java.icmc.sc.usp.br/library/books/ibm_pthreads/document.htm
main() {
  Computing $\pi$
  pthread_t p_threads[MAX_THREADS];
  …
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
  sample_points_per_thread=sample_points/num_threads;
  for (i=0; i<num_threads; i++) {
    hits[i] = i;
    pthread_create(&p_threads[i], &attr, compute_pi,
                   (void *) &hits[i]);
  }
  for (i=0; i<num_threads; i++) {
    pthread_join(p_threads[i], NULL);
    total_hits += hits[i];
  }
  …
void *compute_pi (void *s) {
    int seed, I, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;
    hit_pointer = (int *) x;
    seed = *hit_pointer;
    local_hits = 0;
    for (i=0; i<sample_points_per_thread; i++) {
        rand_no_x = (double) (rand_r(&seed))/ (double)((2<<14)-1);
        rand_no_y = (double) (rand_r(&seed))/ (double)((2<<14)-1);
        if (((rand_no_x–0.5)*(rand_no_x–0.5) +
             (rand_no_y–0.5)*(rand_no_y–0.5)) < 0.25)
            local_hits++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
Quiz 2.5.6

• What prevents a routine from being reentrant?

Use of globals that are changed without protection, e.g., if we updated the total_hits in the threads computing \( \pi \) without ensuring only one process could access the value at a time.
Computing $\pi$: Performance
Quiz 2.5.7

- Does the experiment shown push the spacing to the cache line size? Explain.
Synchronization in Pthreads

- When multiple threads attempt to manipulate the same data, the results can become incoherent.

```c
if (my_cost < best_cost)
    best_cost = my_cost;
```

- Let’s look at what can happen.
Synchronization in Pthreads

```c
if (my_cost < best_cost)
    best_cost = my_cost;
```

<table>
<thead>
<tr>
<th></th>
<th>global</th>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>best_cost</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>my_cost</td>
<td></td>
<td>50</td>
<td>75</td>
</tr>
</tbody>
</table>

What can happen here?
### Synchronization in Pthreads

<table>
<thead>
<tr>
<th></th>
<th>global</th>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>best_cost</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>my_cost</td>
<td>50</td>
<td>75</td>
<td></td>
</tr>
</tbody>
</table>

- thread 1
  - if (my_cost < best_cost)
  - best_cost = my_cost;

- thread 2
  - if (my_cost < best_cost)
  - best_cost = my_cost;

result: **best_cost = 50**
Synchronization in Pthreads

<table>
<thead>
<tr>
<th></th>
<th>global</th>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>best_cost</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>my_cost</td>
<td>50</td>
<td>75</td>
<td></td>
</tr>
</tbody>
</table>

thread 1

if (my_cost < best_cost)
best_cost = my_cost;

thread 2

if (my_cost < best_cost)
best_cost = my_cost;

result: best_cost = 50
**Synchronization in Pthreads**

<table>
<thead>
<tr>
<th></th>
<th>global</th>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>best_cost</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>my_cost</td>
<td>50</td>
<td>75</td>
<td></td>
</tr>
</tbody>
</table>

```c
if (my_cost < best_cost)
    best_cost = my_cost;
```

Result: `best_cost = 50`

```
thread 1
    if (my_cost < best_cost)
        best_cost = my_cost;

thread 2
    if (my_cost < best_cost)
        best_cost = my_cost;
```

Lecture 2.5 – Programming with Shared Memory
Synchronization in Pthreads

<table>
<thead>
<tr>
<th></th>
<th>global</th>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>best_cost</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>my_cost</td>
<td></td>
<td>50</td>
<td>75</td>
</tr>
</tbody>
</table>

if (my_cost < best_cost)
best_cost = my_cost;

result: best_cost = 75
Quiz 2.5.8

• How many assembly instructions does the following C instruction take?

```c
global_count += 5;
```

4 instructions.
Quiz 2.5.9

• Given the previous quiz on how many assembly instructions a C instruction can take, should you ever assume that standard C instructions are atomic?

No – unless the instruction guarantees atomicity.
Synchronization

- **mutex locks**

```c
int pthread_mutex_lock
(pthread_mutex_t *mutex_lock);
```

```c
int pthread_mutex_unlock
(pthread_mutex_t *mutex_lock);
```

```c
int pthread_mutex_init
(pthread_mutex_t *mutex_lock,
const pthread_mutexattr_t *lock_attr);
```
Synchronization

• Reducing Lock Overheads

   int pthread_mutex_trylock
       (pthread_mutex_t *mutex_lock);
   – EBUSY
   – 0
Quiz 2.5.10

• What limitations can you see on speeding up execution with the use of `pthread_mutex_trylock`?

It will only speed things up if you have something else to do between the calls.
Condition Variables & Synchronization

- Condition variable
  - allows thread to block on variable until the data reaches a predefined state
  - has an associated mutex

```c
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```
Controlling Threads

- Attributes
  - associated with threads & synch. variables
  - scheduling (round robin, prioritized, ...)
  - stack size, ...

- attribute objects
  - data structure
  - describes entity properties
    - thread
    - mutex
    - condition variables

http://docs.hp.com/en/B9106-90010/pthread.3T.html
Thread Attributes

- Attribute setup and destruction
  ```c
  int pthread_attr_init(pthread_attr_t *attr);
  int pthread_attr_destroy(pthread_attr_t *attr);
  ```

- Change the default attributes
  ```c
  pthread_attr_setdetachstate
  pthread_attr_setguardsize_np
  pthread_attr_setstacksize
  pthread_attr_setinheritsched
  pthread_attr_setschedpolicy
  pthread_attr_setschedparam
  ```
Mutex Attributes

• Attribute setup and destruction

```c
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_settype_np(
    pthread_mutexattr_t *attr, int type);
```

– where types are one of

```c
PTHREAD_MUTEX_NORMAL_NP
PTHREAD_MUTEX_RECURSIVE_NP
PTHREAD_MUTEX_ERRORCHECK_NP
```

http://java.icmc.sc.usp.br/library/books/ibm_pthreads/document.htm
Thread Cancellation

• To terminate threads:

  int pthread_cancel(pthread_t thread);

• Notes
  – thread may cancel itself or others
  – no guarantees
    • receive or act
  – threads can protect themselves
Composite Synchronization

• Read-Write Locks
  – multiple threads may have read locks at the same time
  – to write the value, a thread must have an exclusive write lock

• Barriers
  – used to hold threads until all other threads participating read the thread
Asynchronous Programming

• Don’t assume any order of execution of threads.

• Rules of Thumb
  – set up all requirements for a thread before you create the thread
  – make sure producers place data before it can be consumed and that buffers can not overflow
  – ensure that all consumers can access data before it is removed or deleted
  – define and use group synchronizations and data replication where appropriate
Quiz 2.5.11

• Why would it make sense to set up all of the requirements for a thread before the thread is created?

So that parameters for initializing a thread can be used in creation of the thread.
Quiz 2.5.12

• What would you expect to happen if you allow producer / consumer buffers to overflow?

An error – may cause termination or at best incorrect behavior.
OpenMP

- Pragma based shared memory parallel
  - `#pragma omp directive [clause list]`
- Execute serially until the parallel directive.
- Parallel directive creates a group of threads.
- Main thread becomes master
  - thread id = 0
- Threads execute structured block indicated by the parallel directive.
Quiz 2.5.13

- Is serial by default the same in Pthreads and OpenMP?

  Yes.
OpenMP

• Structured blocks
  – conditional parallelization
    • if (scalar expression)
  – degree of concurrency
    • num_threads (integer expression)
  – data handling
    • private (variable list)
OpenMP

```c
int a, b;
main () { 
    //serial segment
    #pragma omp parallel num_threads (8) \ 
        private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```
int a, b;
main () {
    //serial segment
    for (i=0; i<8; i++)
        pthreads_create(...,\
                        internal_thread_fn_name, ...);
    // parallel segment
    for (i=0; i<8; i++)
        pthreads_join(...);
    // rest of serial segment
}
Quiz 2.5.14

• Does it appear that OpenMP compiles to equivalent Pthreads code?

Yes.
OpenMP

```c
int a, b;
main () {  
    //serial segment
    #pragma omp parallel if (is_parallel == 1) \  
        num_threads (8) private (a) \  
        shared (b) firstprivate(c)  
    {  
        // structured block  
    }  
    // rest of serial segment  
}
```
OpenMP

• Sharing
  – explicit
    • shared (a)
    • private (b)
    • firstprivate (c)
    • lastprivate (d)
    • threadprivate (e)
    • copyin (f)
  – default (shared)
  – default (none)
OpenMP

• **parallel reduction**

```c
#pragma omp parallel reduction (+: sum) \ 
  num_threads(8)
{
    // compute local sums here
}

// sum here contains sum of all locals
// in master thread only
```
OpenMP

#pragma omp parallel default(private) \ shared (npoints) reduction (+:sum) \ num_threads (8)

    { num_thread = omp_get_num_threads();
      sample_points_per_thread = npoints /
                                  num_threads;
      sum = 0;
      for (i=0; i<sample_point_per_thread; i++) {
        // compute random locations
        if (((rand_no_x - 0.5)*(rand_no_x - 0.5)
            +(rand_no_y -0.5)*(rand_no_y -0.5))
            < 0.25)   sum++;
      } // end parallel section
OpenMP

• Specifying concurrent tasks
  – for directive
    • `#pragma omp for [clause list]`
  – clauses
    • `private`
    • `firstprivate`
    • `lastprivate`
    • `reduction`
    • `nowait`
    • `ordered`
    • `schedule`
OpenMP

- Assigning iterations to threads
  - schedule clause
    - `schedule (scheduling_class \ [, parameter]`
  - classes
    - `static [, chunk-size]`
    - `dynamic [, chunk-size]`
    - `guided [, chunk-size]`
    - `runtime`
Quiz 2.5.15

• If you are using a heterogeneous machine for a homogeneous problem, what scheduling classes might make sense?

dynamic or guided.
OpenMP

• Synchronization across multiple fors

```c
#pragma omp parallel
{
    #pragma omp for nowait
    for (i=0; i<nmax; i++) {
        if (isEqual(name,current_list[i])
            processCurrentName(name); }

    #pragma omp for
    for (i=0; i<nmax; i++) {
        if (isEqual(name,past_list[i])
            processPastName(name); }
} // end parallel section
```

Note the implied barrier at the end of each loop
OpenMP

• Nesting parallel Directives

```
pragma omp sections [clause list]
{
    #pragma omp section
    // structured block
    #pragma omp section
    // structured block
}   // end parallel section
```
OpenMP

- **Nested parallelism**

```c
#pragma omp parallel for default(private) \ 
shared (a, b, c, dim) \ 
num_threads (2)
for (i=0; i<dim; i++) {
    #pragma omp parallel for default(private \ 
shared (a, b, c, dim) \ 
num_threads (2)
    for (j=0; j<dim; j++) {
        c(i,j) = 0;
        #pragma omp parallel for \ 
        default(private \ 
shared (a, b, c, dim) \ 
num_threads (2)
        for (k=0; k<dim; k++) {
            c(i,j) += a(i,k)*b(k,j); }
```

Note: need a critical section
OpenMP

- Processors
  - The number of physical processors actually hosting the threads at any given time is implementation-defined.
  - The functions here affect and monitor threads, processors, and the parallel environment:

  - `omp_set_num_threads`
  - `omp_get_num_threads`
  - `omp_get_max_threads`
  - `omp_get_thread_num`
  - `omp_get_num_procs`
  - `omp_in_parallel`
  - `omp_set_dynamic`
  - `omp_get_dynamic`
  - `omp_set_nested`
  - `omp_get_nested`
OpenMP

• Global Variables
  – Sharing attribute clauses apply only to variables in the lexical extent of the directive on which the clause appears.
  – If a variable is visible when a parallel or work-sharing construct is encountered, and the variable is not specified in a sharing attribute clause or threadprivate directive, then the variable is shared.
OpenMP

• Global Variables

```c
int a;

void f(int n) {
    a = 0;

    #pragma omp parallel for private(a)
    for (int i=1; i<n; i++) {
        a = i;
        g(i, n);
        d(a); // Private copy of "a"
        ...
    }

    ...
}

void g(int k, int n) {
    h(k, a); // The global "a", not the private "a" in f
}
```

Beware accesses to a will be to the global version.
OpenMP

- Synchronization
  - barrier
    #pragma omp barrier
  - single
    #pragma omp single [clause list]
    structured block
  - master
    #pragma omp master
    structured block
Quiz 2.5.16

• Can you think of any reason why you would want a serial section that had to be executed by the master process?
OpenMP

• Critical Sections
  – critical
    
    #pragma omp critical [(name)]
    structured block
  – atomic
    
    #pragma omp atomic [clause list]
    statement

• where statement is one of
  
    x binary_operation = expr
    x++
    x--
    ++x
    --x
OpenMP

- In order execution
  - ordered
    
    #pragma omp ordered
    
    structured block

- Memory consistency
  - flush
    
    #pragma omp flush [(list)]
OpenMP

- **Library functions**
  ```c
  void omp_set_num_threads(int num_threads);
  int omp_get_num_threads();
  int omp_get_max_threads();
  int omp_get_thread_num();
  int omp_get_num_procs();
  int omp_in_parallel();
  ```
OpenMP

• Library thread functions

```c
void omp_set_dynamic(int dynamic_threads);
int omp_get_dynamic();
void omp_set_nested(int nested);
int omp_get_nested();
```
OpenMP

- Library mutual exclusion functions

  ```c
  void omp_init_lock (omp_lock_t *lock);
  void omp_destroy_lock (omp_lock_t *lock);
  void omp_set_lock (omp_lock_t *lock);
  void omp_unset_lock (omp_lock_t *lock);
  int omp_test_lock (omp_lock_t *lock);
  ```
OpenMP

• Library nestable lock functions
  
  ```c
  void omp_init_nest_lock
    (omp_nest_lock_t *lock);
  void omp_destroy_nest_lock
    (omp_nest_lock_t *lock);
  void omp_set_nest_lock
    (omp_nest_lock_t *lock);
  void omp_unset_nest_lock
    (omp_nest_lock_t *lock);
  int omp_test_nest_lock
    (omp_nest_lock_t *lock);
  ```
OpenMP

- environment variables
  - OMP_NUM_THREADS
  - OMP_DYNAMIC
  - OMP_NESTED
  - OMP_SCHEDULE
Explicit Threads vs OpenMP

- OpenMP is simpler – less initialization.
  - minimal overhead
- Explicit threads makes alleviating some overheads more apparent:
  - data movement
  - false sharing
  - contention
- Pthreads more widely available than OpenMP.