

CS 292

Introduction to Parallel Computing

Spring 2008

VANDERBILT



School of Engineering

Announcements

- Read Sections 7.9-7.11
- Programming assignment #1 has been posted to Oak
 - Two programs using Pthreads
 - Due Friday, 2/29/08
- Questions??

Chapter 7 Topic Overview

Shared memory programming:

- Thread Basics ✓
- The POSIX Thread API ✓
- Synchronization Primitives in Pthreads ✓
- Controlling Thread and Synchronization Attributes ✓
- Composite Synchronization Constructs
- OpenMP: a Standard for Directive Based Parallel Programming

Read-Write Locks

- Let's rewrite our `best_cost` computation to use read-write locks which allows multiple threads to simultaneously read the `best_cost` computed so far:

```
void *find_best_cost(void *list_ptr) {
    ....
    mylib_rwlock_rlock(&read_write_lock);
    if (my_cost < best_cost) {
        mylib_rwlock_unlock(&read_write_lock);
        mylib_rwlock_wlock(&read_write_lock);
        best_cost = my_cost;  /* RACE CONDITION?? */
    }
    mylib_rwlock_unlock(&read_write_lock);
    ....
}
```

Read-Write Locks

- Notes on the `best_cost` computation that uses read-write locks:
 - Performance is influenced by the number of writes compared to the number of reads
 - If there are few writes, then we should get better performance since the reads can be performed simultaneously
 - If in the worst case that each thread must perform a write, then the read locks are superfluous and only add overhead to the program

Barriers

- A *barrier* holds a thread until all threads participating in the barrier have reached it.
- Barriers can be implemented using a counter, a mutex and a condition variable.
- A single integer is used to keep track of the number of threads that have reached the barrier.
- If the count is less than the total number of threads, the threads execute a condition wait.
- The last thread entering the barrier (and setting the count to the number of threads) wakes up all the threads using a condition broadcast.

Barriers

```
typedef struct {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed;
    int count;
} mylib_barrier_t;

void mylib_init_barrier(mylib_barrier_t *b) {
    pthread_mutex_init(&(b->count_lock), NULL);
    pthread_cond_init(&(b->ok_to_proceed), NULL);
    b->count = 0;
}
```

Barriers

```
void mylib_barrier (mylib_barrier_t *b, int num_threads)
{
    pthread_mutex_lock(&(b->count_lock));
    b->count++;
    if (b->count == num_threads) {
        b->count = 0;
        pthread_cond_broadcast(&(b->ok_to_proceed));
    }
    else {
        while (b->count != 0)    /* slightly different than text */
            pthread_cond_wait(&(b->ok_to_proceed),
                              &(b->count_lock));
    }
    pthread_mutex_unlock(&(b->count_lock));
}
```

Must use a different barrier object at each barrier point in your program. Why?

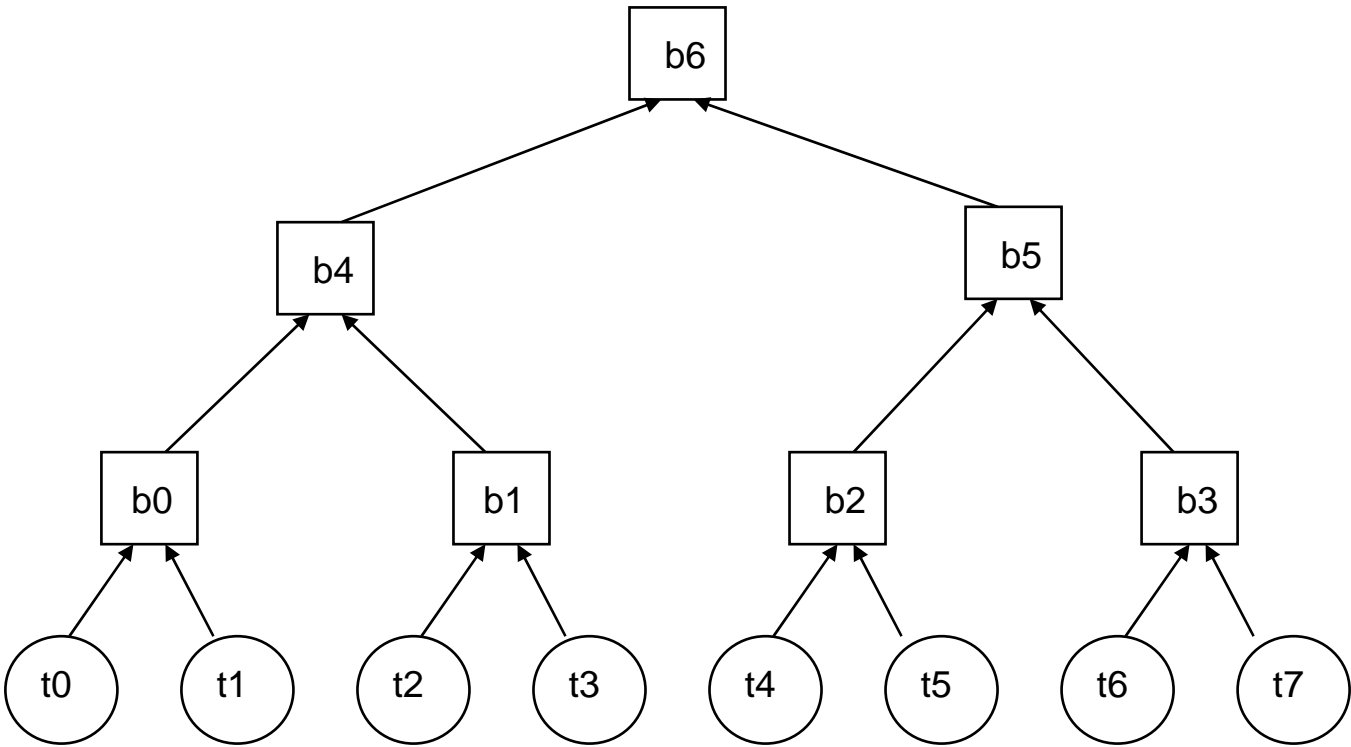
Barriers

- When the broadcast is issued to wake all the waiting threads, each thread must be granted the mutex before it can continue execution.
- This serializes the wake-up process.
- The barrier described above is called a *linear barrier*.
- The trivial lower bound on execution time of this function is therefore $O(n)$ for n threads.

Barriers

- This implementation of a barrier can be sped up using multiple barrier variables organized in a tree.
- We use $n/2$ condition variable-mutex pairs for implementing a barrier for n threads.
- At the lowest level, threads are paired up and each pair of threads shares a single condition variable-mutex pair.
- Once both threads of a pair arrive, one of the two moves on, the other one waits.
 - Need $n/4$ condition variable-mutex pairs at the next level
- This process repeats up the tree to ensure all threads have reached the barrier, and then reverses and goes back down to signal.
- This is also called a log barrier and its runtime grows as $O(\log p)$.

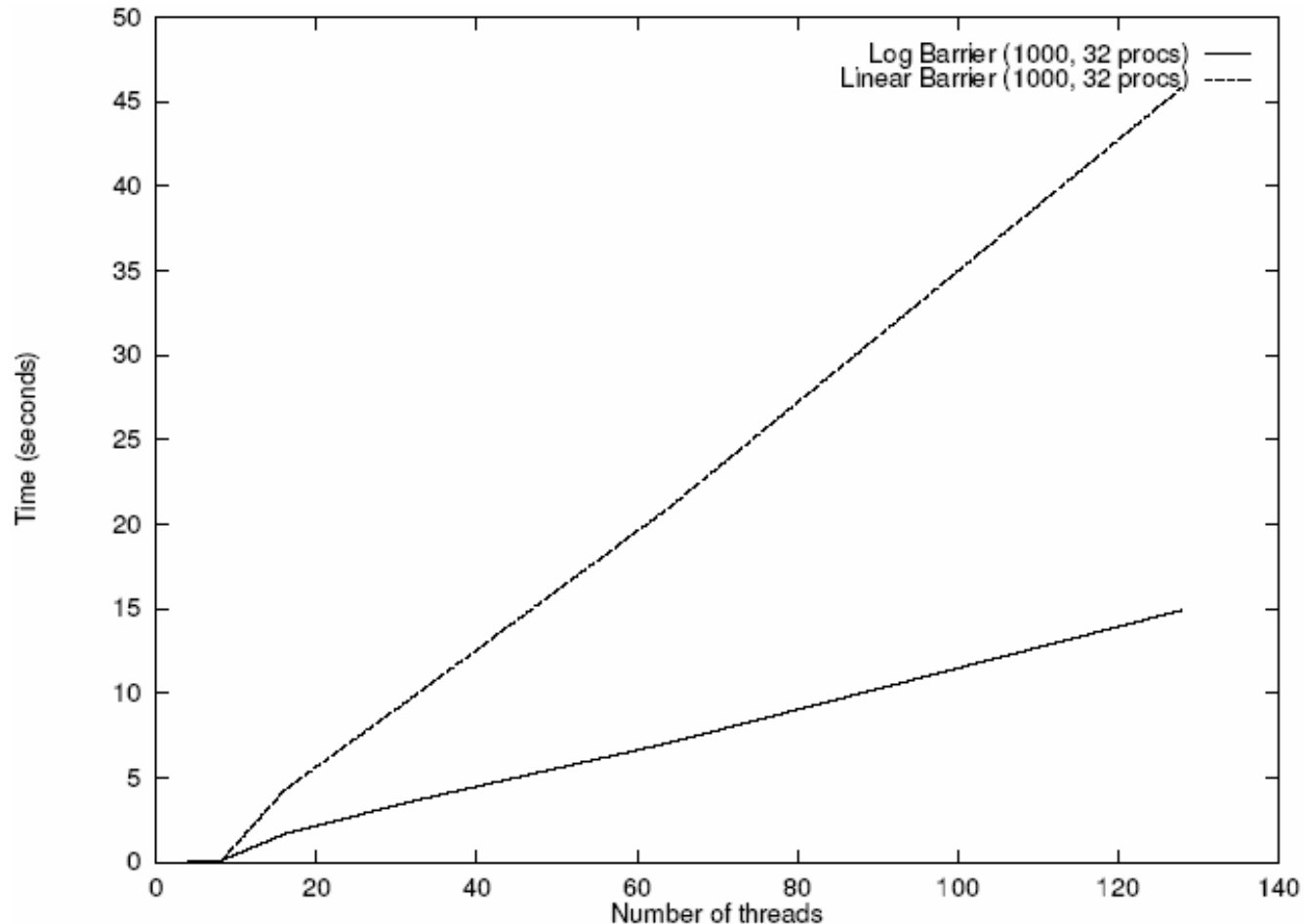
Log Barrier



b0	b1	b2	b3	b4	b5	b6	b7
----	----	----	----	----	----	----	----

Array of barriers

Barrier



- Execution time of 1000 sequential and logarithmic barriers as a function of number of threads on a 32 processor SGI Origin 2000.

Tips for Designing Asynchronous Programs

- Never rely on scheduling assumptions when exchanging data.
 - I.e., always make sure data is ready for other threads when you create them.
- Never rely on liveness of data resulting from assumptions on scheduling.
 - I.e., never communicate via private (stack) data since it may no longer exist when the other thread attempts to access it.
- Do not rely on system scheduling as a means of synchronization.
 - I.e., don't rely on thread priority levels
- Where possible, define and use group synchronizations and data replication.

OpenMP

- Pthreads API is a rather low-level programming model. Not only must the programmer identify potential parallelism, but the programmer must do all the work to exploit that parallelism (which can be error prone).
- Why can't compilers do this for us; i.e., convert serial code to parallel code automatically??
- This has been an area of active research for over thirty years. Many successes, but in general this is a very hard problem.
- Happy medium: have the programmer provide additional information, then the compiler can handle the low-level details. *OpenMP* is one such strategy.

OpenMP: a Standard for Directive Based Parallel Programming

- OpenMP is a directive-based API that can be used with FORTRAN, C, and C++ for programming shared address space machines.
- OpenMP directives provide support for concurrency, synchronization, and data handling while obviating the need for explicitly setting up mutexes, condition variables, data scope, and initialization.
 - Let the compiler take care of the details!!
- *Directives* are a method of adding platform-specific information to your program while maintaining portability
 - The directives are simply ignored when not applicable

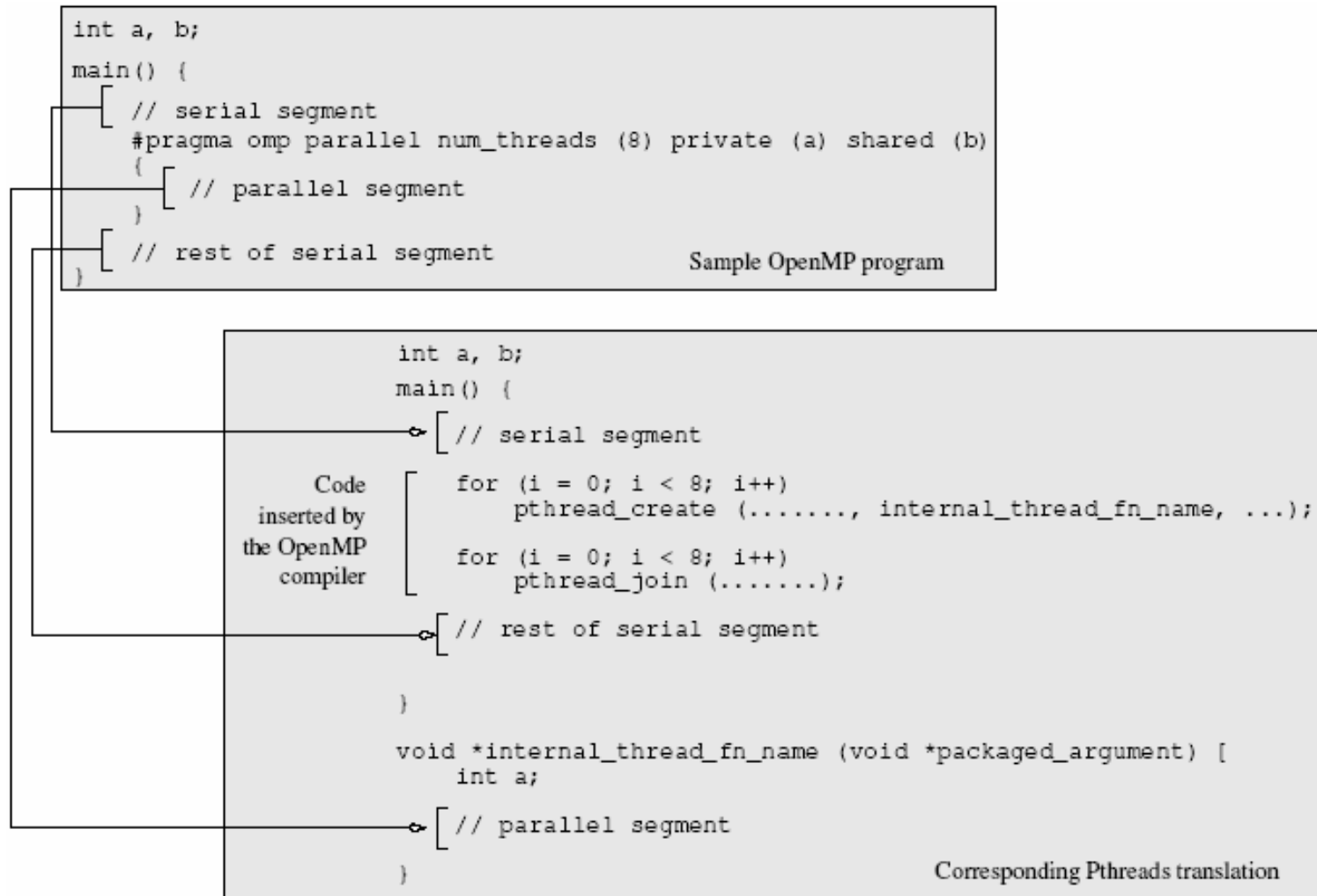
OpenMP Programming Model

- OpenMP directives in C and C++ are based on the `#pragma` compiler directives.
- A directive consists of a directive name followed by clauses.

```
#pragma omp directive [clause list]
```
- OpenMP programs execute serially until they encounter the `parallel` directive, which creates a group of threads.

```
#pragma omp parallel [clause list]  
/* structured block */
```
- Each thread that is created executes the code in the structured block.

OpenMP Programming Model



- A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

OpenMP Programming Model

- The clause list is used to specify conditional parallelization, number of threads, and data handling.
 - **Conditional Parallelization:** The clause `if (scalar expression)` determines whether the parallel construct results in creation of threads.
 - **Degree of Concurrency:** The clause `num_threads(integer expression)` specifies the number of threads that are created.
 - **Data Handling:**
 - The clause `private (variable list)` indicates variables local to each thread.
 - The clause `firstprivate (variable list)` is similar to the `private`, except values of variables are initialized to corresponding values before the parallel directive.
 - The clause `shared (variable list)` indicates that variables are shared across all the threads.

OpenMP Programming Model

```
#pragma omp parallel if (is_parallel==1) num_threads(8) \  
    private (a) shared (b) firstprivate(c)  
{  
    /* structured block */  
}
```

- If the value of the variable `is_parallel` equals one, eight threads are created.
- Each of these threads gets private copies of variables `a` and `c`, and shares a single value of variable `b`.
- The value of each copy of `c` is initialized to the value of `c` before the parallel directive.
- The default state of a variable is specified by the clause `default (shared)` **Or** `default (none)`.

Reduction Clause in OpenMP

- The `reduction` clause specifies how multiple local copies of a variable from different threads are combined into a single copy at the master when threads exit.
- The usage of the `reduction` clause is
`reduction (operator: variable list).`
- The variables in the list are implicitly specified as being private to threads.
- The operator can be one of `+`, `*`, `-`, `&`, `|`, `^`, `&&`, and `||`.

```
#pragma omp parallel reduction(+: sum) num_threads(8)
{
    /* compute local sums here */
}
/*sum here contains sum of all local instances of sums */
```

OpenMP Programming: Example

```
/* *****  
An OpenMP version of a threaded program to compute PI.  
***** */  
#pragma omp parallel \  
private(num_threads, sample_points_per_thread, i, rand_no_x, rand_no_y) \  
shared(npoints) reduction(+: sum) num_threads(8)  
{  
    num_threads = omp_get_num_threads();  
    sample_points_per_thread = npoints / num_threads;  
    sum = 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)  
            sum++;  
    }  
}
```

Note: default(private) is not supported in C/C++. Red font indicates a difference between lecture notes and text.