Parallelism: Practical Considerations, pthreads

CSC 2/458
Instructor: Sandhya Dwarkadas

Performance Metrics

- Speedup \( p \): \( s(p) = \frac{T(1)}{T(p)} \)
- Parallel efficiency \( p \): \( e(p) = s(p)/p = \frac{T(1)}{pT(p)} \)
- Experimentally determined serial fraction

Programming in Parallel

- Explicitly concurrent languages – e.g., Occam, SR, Java, Ada, UPC, C++11
- Compiler-supported extensions – e.g., HPF, Cilk
- Library packages outside the language proper – e.g., pthreads, MPI
### Pthreads: A Shared Memory Programming Model

- POSIX standard shared-memory multithreading interface
- Not just for parallel programming, but for general multithreaded programming
- Provides primitives for process management and synchronization

### What does the user have to do?

- Decide how to decompose the computation into parallel parts
- Create (and destroy) processes to support that decomposition
- Add synchronization to make sure dependences are covered

### General Thread Structure

- Typically, a thread is a concurrent execution of a function or a procedure
- So, your program needs to be restructured such that parallel parts form separate procedures or functions

### Thread Creation

```c
int pthread_create
(pthread_t *new_id,
 const pthread_attr_t *attr,
 void *(*func)(void *),
 void *arg)
```

- `new_id`: thread’s unique identifier
- `attr`: ignore for now
- `func`: function to be run in parallel
- `arg`: arguments for function `func`
Example of Thread Creation

```c
void *func(void *arg) {
    int *I=arg;
    ....
}

void main() {
    int X;  pthread_t id;
    ....
    pthread_create(&id, NULL, func, &X);
    ...
}
```

Pthread Termination

```c
void pthread_exit(void *status)
```

- Terminates the currently running thread.
- Is implicit when the function called in pthread_create returns.

Thread Joining

```c
int pthread_join( pthread_t new_id, void **status)
```

- Waits for the thread with identifier new_id to terminate, either by returning or by calling pthread_exit().
- Status receives the return value or the value given as argument to pthread_exit().

Example of Thread Creation

```c
main()
    pthread_create(func)
    pthread_join(id)
    pthread_exit()
```
Matrix Multiply

for (i=0; i<n; i++)
for (j=0; j<n; j++) {
    c[i][j] = 0.0;
    for (k=0; k<n; k++)
        c[i][j] += a[i][k]*b[k][j];
}

Parallel Matrix Multiply

• All i- or j-iterations can be run in parallel
• If we have p processors, n/p rows to each processor
• Corresponds to partitioning i-loop

Matrix Multiply: Parallel Part

void mmult(void* s)
{
    int slice = (int) s;
    int from = (slice*n)/p;
    int to = ((slice+1)*n)/p;
    for (i=from; i<to; i++)
        for (j=0; j<n; j++)
            c[i][j] = 0.0;
            for (k=0; k<n; k++)
                c[i][j] += a[i][k]*b[k][j];
}

Matrix Multiply: Main

int main()
{
    pthread_t thrd[p];
    for (i=0; i<p; i++)
        pthread_create(&thrd[i], NULL, mmult, (void*) i);
    for (i=0; i<p; i++)
        pthread_join(thrd[i], NULL);
}
General Program Structure

- Encapsulate parallel parts in functions.
- Use function arguments to parametrize what a particular thread does.
- Call pthread_create() with the function and arguments, save thread identifier returned.
- Call pthread_join() with that thread identifier.

Pthreads Synchronization

- Create/exit/join
  - provide some form of synchronization
  - at a very coarse level
  - requires thread creation/destruction
- Need for finer-grain synchronization
  - mutex locks, reader-writer locks, condition variables, semaphores

Synchronization Primitives in Pthreads

- Mutexes
- Reader-writer locks
- Condition variables
- Semaphores
- Barriers

Mutex Locks: Creation and Destruction

```c
pthread_mutex_init(
    pthread_mutex_t * mutex,
    const pthread_mutex_attr *attr);
```

- Creates a new mutex lock

```c
pthread_mutex_destroy( 
    pthread_mutex_t *mutex);
```

- Destroys the mutex specified by mutex
Mutex Locks: Lock

pthread_mutex_lock(
    pthread_mutex_t *mutex);

• Tries to acquire the lock specified by mutex.
• If mutex is already locked, then calling thread blocks until mutex is unlocked.

Mutex Locks: UnLock

pthread_mutex_unlock(
    pthread_mutex_t *mutex);

• If calling thread has mutex currently locked, this will unlock the mutex.
• If other threads are blocked waiting on this mutex, one will unblock and acquire mutex
  • Which one is determined by the scheduler

Condition variables: Creation and Destruction

pthread_cond_init(
    pthread_cond_t *cond, 
    pthread_cond_attr *attr)

• Creates a new condition variable cond

pthread_cond_destroy( 
    pthread_cond_t *cond)

• Destroys the condition variable cond.

Condition Variables: Wait

pthread_cond_wait(
    pthread_cond_t *cond, 
    pthread_mutex_t *mutex)

• Blocks the calling thread, waiting on cond
• Unlocks the mutex
• Re-acquires the mutex when unblocked
**Condition Variables: Signal**

```
pthread_cond_signal(
    pthread_cond_t *cond)
```

- Unblocks one thread waiting on cond.
- Which one is determined by scheduler.
- If no thread waiting, then signal is a no-op.

**Condition Variables: Broadcast**

```
pthread_cond_broadcast(
    pthread_cond_t *cond)
```

- Unblocks all threads waiting on cond.
- If no thread waiting, then broadcast is a no-op.

**Use of Condition Variables**

- **IMPORTANT NOTE:** A signal is “forgotten” if there is no corresponding wait that has already occurred.
- Use semaphores (or construct a semaphore) if you want the signal to be remembered

**Semaphores**

```
sem_wait(sem_t *sem)
```

- Blocks until the semaphore value is non-zero.
- Decrements the semaphore value on return.

```
sem_post(sem_t *sem)
```

- Unlocks the semaphore and unblocks one waiting thread.
- Increments the semaphore value otherwise.
**PIPE with Pthreads**

```c
P1: for (i=0; i<num_pics; i++) {
    int_pic_1[i] = trans1(in_pic);
    sem_post(event_1_2[i]);
}

P2: for (i=0; i<num_pics; i++) {
    sem_wait(event_1_2[i]);
    int_pic_2[i] = trans2(int_pic_1[i]);
    sem_post(event_2_3[i]);
}
```

**Parallel TSP**

```c
process i:
while((p = de_queue()) != NULL) {
    for each expansion by one city {
        q = add_city(p);
        if complete(q) { update_best(q); }
        else en_queue(q);
    }
}
```

**Parallel TSP: Mutual Exclusion**

```c
en_queue() / de_queue() {
    pthread_mutex_lock(&queue);
    ...
    pthread_mutex_unlock(&queue);
}

update_best() {
    pthread_mutex_lock(&best);
    ...
    pthread_mutex_unlock(&best);
}
```

**Parallel TSP: Mutual Exclusion**

- Need critical section
  - in `update_best`,
  - in `en_queue/de_queue`.
- In `de_queue`
  - wait if q is empty,
  - terminate if all processes are waiting.
- In `en_queue`
  - signal q is no longer empty.
Parallel TSP: Condition Synchronization

de_queue() {
    pthread_mutex_lock(&queue);
    while( (q is empty) and (not done) ) {
        waiting++;
        if( waiting == p ) {
            done = true;
            pthread_cond_broadcast(&empty);
        } else {
            pthread_cond_wait(&empty, &queue);
            waiting--;
        }
    }
    if( done )
        return null;
    else
        remove and return head of the queue;
    pthread_mutex_unlock(&queue);
}

Sequential SOR

for some number of timesteps/iterations {
    for (i=0; i<n; i++)
        for( j=1, j<n, j++ )
            temp[i][j] = 0.25 * ( grid[i-1][j] + grid[i+1][j]
                                    + grid[i][j-1] + grid[i][j+1] );
    for( i=0; i<n; i++)
        for( j=1; j<n; j++ )
            grid[i][j] = temp[i][j];
}

Parallel SOR

• First (i,j) loop nest can be parallelized
• Second (i,j) loop nest can be parallelized
• Must wait until all processors have finished first loop nest before starting second
• Must wait until all processors have finished second loop nest of previous iteration before starting first loop nest of next iteration
• Give n/p rows to each processor

Pthreads SOR: Parallel parts (1)

void* sor_1(void *s)
{
    int slice = (int) s;
    int from = (slice*n)/p;
    int to = ((slice+1)*n)/p;
    for(i=from;i<to;i++)
        for( j=0; j<n; j++ )
            temp[i][j] = 0.25*(grid[i-1][j] + grid[i+1][j]
                                + grid[i][j-1] + grid[i][j+1]);
}
Pthreads SOR: Parallel parts (2)

```c
void* sor_2(void *s)
{
    int slice = (int) s;
    int from = (slice*n)/p;
    int to = ((slice+1)*n)/p;

    for(i=from; i<to; i++)
        for(j=0; j<n; j++)
            grid[i][j] = temp[i][j];
}
```

Pthreads SOR: main

```c
for some number of timesteps {
    for( i=0; i<p; i++ )
        pthread_create(&thrd[i], NULL, sor_1, (void *)i);
    for( i=0; i<p; i++ )
        pthread_join(thrd[i], NULL);
    for( i=0; i<p; i++ )
        pthread_create(&thrd[i], NULL, sor_2, (void *)i);
    for( i=0; i<p; i++ )
        pthread_join(thrd[i], NULL);
}
```

Barrier Synchronization

- A wait at a barrier causes a thread to wait until all threads have performed a wait at the barrier.
- At that point, they all proceed
- Use instead of creating and destroying threads multiple times to achieve the same global synchronization with lower overhead

Implementing Barriers in Pthreads

- Count the number of arrivals at the barrier.
- Wait if this is not the last arrival.
- Make everyone unblock if this is the last arrival.
- Since the arrival count is a shared variable, enclose the whole operation in a mutex lock-unlock.
Implementing Barriers in Pthreads

```c
void barrier()
{
    pthread_mutex_lock(&mutex_arr);
    arrived++;
    if (arrived<N) {
        pthread_cond_wait(&cond, &mutex_arr);
    } else {
        pthread_cond_broadcast(&cond);
        arrived=0; /* be prepared for next barrier */
    }
    pthread_mutex_unlock(&mutex_arr);
}
```

Parallel SOR with Barriers (1 of 2)

```c
void* sor (void* arg)
{
    int slice = (int)arg;
    int from = (slice * (n-1))/p + 1;
    int to = ((slice+1) * (n-1))/p + 1;
    for some number of iterations { … } }
```

Parallel SOR with Barriers (2 of 2)

```c
for (i=from; i<to; i++)
    for (j=1; j<n; j++)
        temp[i][j] = 0.25 * (grid[i-1][j] + grid[i+1][j] + grid[i][j-1] + grid[i][j+1]);
barrier();
```

Parallel SOR with Barriers: main

```c
int main(int argc, char *argv[])
{
    pthread_t *thrds[p];
    /* Initialize mutex and condition variables */
    for (i=0; i<p; i++)
        pthread_create(&thrds[i], &attr, sor, (void*)i);
    for (i=0; i<p; i++)
        pthread_join(thrds[i], NULL);
    /* Destroy mutex and condition variables */
}```
Busy Waiting

- Not an explicit part of the API
- Available in any general shared memory programming environment

Busy Waiting

initially: \( \text{flag} = 0; \)

P1: produce data;
\( \text{flag} = 1; \)

P2: while( !\text{flag} ) ;
consume data;

Use of Busy Waiting

- On the surface, simple and efficient
- In general, not a recommended practice
- Often leads to messy and unreadable code (blurs data/synchronization distinction)
- On some architectures, may be inefficient or may not even work as intended (depending on consistency model)

Private Data in Pthreads

- To make a variable private in pthreads, you need to make an array out of it
- Index the array by thread identifier, which you can get by the pthread_self() call
- An alternative is to declare the variable on the stack
- Not very elegant or efficient
Other Primitives in Pthreads

• Set the attributes of a thread
• Set the attributes of a mutex lock
• Set scheduling parameters

Acknowledgements

Slides include content from Willy Zwaenepoel that accompany his corresponding course/textbook and that have been adapted to suit the content of this course.