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

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.
Synchronization Primitives in Pthreads

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

Sequential SOR

```c
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];
}
```

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

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);
}

Monitor

- Synchronization construct that provides mutual exclusion and the ability to wait on one or more conditions
  - Hoare style: signaler must leave/give up the monitor to the thread being woken up
    - E.g., Hansen’s Concurrent Pascal
  - Mesa style: thread being woken up may return to the monitor at a later time
    - E.g., Java and C#

Implications: 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);
}

Parallel SOR with Barriers (1 of 2)

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)

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();
for (i=from; i<to; i++)
for (j=1; j<n; j++)
    grid[i][j]=temp[i][j];
barrier();

Parallel SOR with Barriers: main

int main(int argc, char *argv[])
{
    pthread_t *thrd[p];
    /* Initialize mutex and condition variables */
    for (i=0; i<p; i++)
        pthread_create (&thrd[i], &attr, sor, (void*)i);
    for (i=0; i<p; i++)
        pthread_join (thrd[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: flag = 0;

P1: produce data;
    flag = 1;

P2: while(!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
Thread Creation Syntax

- Properly nested (can share context)
  - Co-Begin (Algol 68, Occam, SR)
  - Parallel loops (HPF, Occam, Fortran90, SR)
  - Launch-at-Elaboration (Ada, SR)
- Fork/Join (pthreads, Ada, Modula-3, Java, SR)
- Implicit Receipt (RPC systems, SR)
- Early Reply (SR)

Co-Begin (e.g., Algol 68)

```plaintext
par begin
  p (a, b, c),
  begin
    d := q(e, f);
    r(d, g, h)
  end,
  s(i, j)
end
```

- 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
Co-Begin in OpenMP

```c
#pragma omp sections
{  
    #pragma omp section
    { printf("thread 1 here\n");}
    #pragma omp section
    { printf("thread 2 here\n");
  }
```

Parallel Loops

- Examples:
  - OpenMP: `# pragma omp parallel for`
  - C#: Task Parallel Library

OpenMP

For-loop parallelized using an OpenMP pragma

```c
#pragma omp parallel for
for (int i=0; i<n; i++)
    c[i] = a[i] + b[i];
```

% `cc -xopenmp source.c`
% `setenv OMP_NUM_THREADS 5`
% `a.out`

Parallelism in Loops (Fortran)

- FOR: sequential
- FORALL: each statement executed completely and in parallel; old values for current and subsequent statements; new values for previous statement
- DOPAR: each iteration executed in parallel; statements within each iteration executed sequentially; old values for other iterations
  – DOALL: DOPAR with no conflicts
- DOACROSS: sequential loop with dependences
- DOSINGLE: each variable assigned once, new value always used
**Example Loop**

1. Loop \( i=1:4 \)
2. \( a[i] = a[i-1]+1 \)
3. \( b[i] = b[i+1] + a[i-1] \)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

---

**Launch-at-Elaboration (e.g., Ada)**

```plaintext
procedure P is
task T is
  ...
end T;
begin – P
  ...
end P;
```

---

**Explicit Thread Creation Syntax**

- Fork/Join (pthreads, Ada, Modula-3, Java, SR)
- Implicit Receipt (RPC systems, SR)
- Early Reply (SR – fork a thread on a local procedure, reply early, and continue execution after returning/replying to caller)

---

**CILK**

```plaintext
cilk int fib (int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib (n-1);
    y = spawn fib (n-2);
    sync;
    return (x+y);
  }
}
```
Nested vs. General Parallelism

Sequential SOR

• for (k = 0; k < 100; k++) {
  • for (j = 1; j < M-1; j++)
    • for (i = 1; i < M-1; i++)
      • a[j][i] = (b[j][i-1] + b[j][i+1] + b[j-1][i] + b[j+1][i])/4;
  • for (j = 1; j < M-1; j++)
    • for (i = 1; i < M-1; i++)
      • b[j][i] = a[j][i];
  • }

Shared Memory Version

• for (k = 0; k < 100; k++) {
  • for (j = begin; j < end; j++)
    • for (i = 1; i < M-1; i++)
      • a[j][i] = (b[j][i-1] + b[j][i+1] + b[j-1][i] + b[j+1][i])/4;
    • barrier();
  • for (j = begin; j < end; j++)
    • for (i = 1; i < M-1; i++)
      • b[j][i] = a[j][i];
    • barrier();
  • }
Data Parallel Version of SOR (OpenMP)

```
for (k = 0; k < 100; k++) {
    #pragma parallel shared(a, b) private(i, j)
    {
        #pragma omp parallel for
        for (j = 1; j < M-1; j++)
            for (i = 1; i < M-1; i++)
                a[j][i] = (b[j-1][i] + b[j][i+1] + b[j-1][i+1] + b[j+1][i]) / 4;
    }
    #pragma parallel shared(a, b) private(i, j)
    {
        #pragma omp parallel for
        for (j = 1; j < M-1; j++)
            for (i = 1; i < M-1; i++)
                b[j][i] = a[j][i];
    }
}
```

SOR in HPF (Fortran D)

```
real a (1000, 1000), b(1000, 1000)
C decomposition d(1000, 1000)
C align a, b with d
C distribute d(:, block)
do k = 1, 1000
do j = 2, 999
do I = 2, 999
  a(i,j) = F(b(i-1,j),b(i+1,j),b(i,j-1),b(i,j+1))
enddo
enddo
second loop (b(i,j) = a(i,j))
Compiler applies owner-computes rule
```

Thread Implementation
Processes or Threads

- A process or thread is a potentially-active execution context
- Processes/threads can come from
  - Multiple CPUs
  - Kernel-level multiplexing of single physical CPU (kernel-level threads or processes)
  - Language or library-level multiplexing of kernel-level abstraction (user-level threads)
- Threads can run
  - Truly in parallel (on multiple CPUs)
  - Unpredictably interleaved (on a single CPU)
  - Run-until-block (coroutine-style)

Processes Vs. Threads

- Process
  - Single address space
  - Single thread of control for executing program
  - State information
    - Page tables, swap images, file descriptors, queued I/O requests, saved registers
- Threads
  - Separate notion of execution from the rest of the definition of a process
  - Other parts potentially shared with other threads
  - Program counter, stack of activation records, control block (e.g., saved registers/state info for thread management)
  - Kernel-level (lightweight process) handled by the system scheduler
  - User-level handled in user mode

Thread Implementation Requirements

- Data structures
  - Program counter
  - Stack
  - Control block (state for thread management)
  - Run queue

Thread Scheduling: Transferring Context Blocks

Coroutines

\[
\text{transfer}(\text{other})
\]
\[
\begin{align*}
\text{save all callee-saves registers on stack, including ra and fp} \\
*\text{current} & := \text{sp} \\
\text{current} & := \text{other} \\
\text{sp} & := *\text{current} \\
\text{pop all callee-saves registers (including ra, but NOT \text{sp})} \\
\text{return (into different coroutine!)}
\end{align*}
\]
Uniprocessor Scheduling

- Use Ready List to reschedule voluntarily (cooperative threading)

reschedule:
  - $t : cb := \text{dequeue(ready\_list)}$
  - transfer($t$)

yield:
  - enqueue(ready\_list, current)
  - reschedule

sleep\_on($q$):
  - enqueue($q$, current)
  - reschedule

Preemption

- Use timer interrupts or signals to trigger involuntary yields
- Protect scheduler data structures by disabling/re enabling prior to/after rescheduling

yield:
  - disable\_signals
  - enqueue(ready\_list, current)
  - reschedule
  - re-enable\_signals

Multiprocessor Scheduling

- Disabling signals not sufficient
- Acquire scheduler lock when accessing any scheduler data structure, e.g.,

yield:
  - disable\_signals
  - acquire(scheduler\_lock) // spin lock
  - enqueue(ready\_list, current)
  - reschedule
  - release(scheduler\_lock)
  - re-enable\_signals

Thread Management

- Creation
- Startup
- Block
- Signal
- Resume
- Finish
Performance Measures

- Latency
  - Cost of thread management under the best case assumption of no contention for locks
- Throughput
  - Rate at which threads can be created, started, and finished when there is contention

Thread Management Data Structures?

- Control blocks
- Stacks
- Ready queue

Anderson et al.

- Raises issues of
  - Locality (per-processor data structures)
  - Granularity of scheduling tasks
  - Lock overhead
  - Tradeoff between throughput and latency
    - Large critical sections are good for best-case latency (low locking overhead) but bad for throughput (low parallelism)

Optimizations

- Allocate stacks lazily
- Store deallocated control blocks and stacks in free lists
- Create per-processor ready lists
- Create local free lists for locality
- Queue of idle processors (in addition to queue of waiting threads)
Ready List Management

- Single lock for all data structures
- Multiple locks, one per data structure
- Local freelists for control blocks and stacks, single shared locked ready list
- Queue of idle processors with preallocated control block and stack waiting for work
- Local ready list per processor, each with its own lock