Programming in Parallel

- Explicitly concurrent languages – e.g., Occam, SR, Java, Ada, UPC
- Compiler-supported extensions – e.g., HPF, Cilk
- Library packages outside the language proper – e.g., pthreads, MPI

Data Parallelism: Parallel Loops

- 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

Programming Models

- Standard models of parallelism
  - shared memory (Pthreads)
  - message passing (MPI)
  - data parallel (Fortran 90 and HPF)
  - shared memory + data parallel (OpenMP)
  - Remote procedure call
  - Global address space (UPC)

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>Initially</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>4</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>

<table>
<thead>
<tr>
<th></th>
<th>(2) a[i-1]</th>
<th>(3) a[i-1]</th>
<th>(3) b[i+1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>for</td>
<td>New</td>
<td>New</td>
<td>Old</td>
</tr>
<tr>
<td>forall</td>
<td>Old</td>
<td>New</td>
<td>Old</td>
</tr>
<tr>
<td>dopar</td>
<td>Old</td>
<td>Old</td>
<td>Old</td>
</tr>
<tr>
<td>dosingle</td>
<td>New</td>
<td>New</td>
<td>New</td>
</tr>
</tbody>
</table>

Final values using for, forall, dopar, dosingle?
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

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)

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

Co-Begin in OpenMP

```c
#pragma omp sections
{
#pragma omp section
    { printf("thread 1 here\n");}
#pragma omp section
    { printf("thread 2 here\n");
}
```
Launch-at-Elaboration (e.g., Ada)

```ada
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

```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);
  }
}
```
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

```plaintext
transfer(other)
    save all callee-saves registers on stack, including ra and fp
    *current := sp
    current := other
    sp := *current
    pop all callee-saves registers (including ra, but NOT sp!)
    return (into different coroutine!)
```

Uniprocessor Scheduling

- Use Ready List to reschedule voluntarily (cooperative threading)
- **reschedule:**
  - `t : cb := 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/reenabling 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`
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)

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

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