OpenMP 3

Victor Eijkhout & Cyrus Proctor

PCSE 2015

Synchronization

Barriers

- Let threads wait for each other in a parallel region
- No need to break up the team

```
#pragma omp parallel shared(x)
{
  // some parallel computation of x
  .....
#pragma omp barrier
#pragma omp for
  for (i=0; i<N; i++) {
    ....... x ...... // use x
  }
}
```

Without the barrier a thread could start using `x` before it's computed.

Note: barriers need to be encountered by all threads in a team; therefore can not be in a worksharing construct.

Barriers on workshare

- No barrier at the start
- Implicit barrier at the end
- No barrier with `nowait`

`nowait` example 1

```
#pragma omp parallel
{
  x = local_computation()
#pragma omp for nowait
  for (i=0; i<N; i++) {
    f(i)
  }
#pragma omp for schedule(dynamic,n)
  for (i=0; i<N; i++) {
    x[i] = ...
  }
}
```

the `nowait` clause cancels the barrier:
threads can start the second loop early

`nowait` example 2

```
#pragma omp parallel
{
  x = local_computation()
#pragma omp for nowait
  for (i=0; i<N; i++) {
    x[i] = ...
  }
#pragma omp for
  for (i=0; i<N; i++) {
    y[i] = ... x[i] ...
  }
}
```

Both loops have a static schedule:
it is safe to start the second

Critical / atomic

- Atomic operations: can only be executed by one thread at a time
- The `s = s+...` update of a reduction is atomic operation
- Critical section: OpenMP construct for atomic operations
- `critical` directive is general; `atomic` limited, but can use hardware support
- Critical sections can be very expensive: require operating system support

critical section

```
  double s = 0;
#pragma omp parallel for
  for (i=0; i<N; i++) {
    double t = f(i);
#pragma omp critical
    s += t;
  }
```

Critical sections can be named.

Locks

- Locks and critical sections both give exclusive execution
- Subtle difference: critical limits access to section of *code*
- lock limits access to item of *data*
- Example: writing to database

Locks

Create/destroy:

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

Set and release:

```
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
```

Locks example

```
omp_lock_t lockvar;
void omp_init_lock(&lockvar);

omp_set_lock(&lockvar);
var = var+update
omp_unset_lock(&lockvar);

void omp_destroy_lock(&lockvar);
```

Not tied to parallel regions!

Tasks

What about fork/join?

- OpenMP is based on fork/join
- Task is a work unit: there can be more tasks active than threads
- Task dependencies: action after the join depends on the tasks that are forked

Explicit task syntax

```
 x = f();
#pragma omp task
 y = g(x);
#pragma omp task
 z = h(x);
#pragam omp taskwait
 a = y+x;
```

How does this relate to sections? Loops?

Tasks and thread teams

Strange idiom:

- Create parallel region
- Task generation done by a single thread

```
 x = f();
#pragma omp parallel
#pragma omp single
  {
#pragma omp task
   y = g(x);
#pragma omp task
   z = h(x);
#pragam omp taskwait
   a = y+x;
  }
```

Tasks and loops

```
  for (i=0; i<N; i++) {
#pragma omp task
      x[i] = f(i);
  }
#pragma omp taskwait
  y = g(x);
```

Task example: breadth-first search

```
void search_node( Node *n ) {
  if (n->is_leaf()) {
    return something;
  } else {
    int left,right;
#pragma omp task
    left = search_node( n->left_child() );
#pragma omp task
    right = search_node( n->right_child() );
#pragma omp taskwait
    return f(left,right);
  }
```

Simultaneous search in the whole tree.

Task example: Fibonacci

In a tree each node is visited once.
What if we have a general Directed Acyclic Graph?

```
void search_node( Node *n ) {
  for ( p in n->predecessors() ) {
#pragma omp task
    search_node( n->predecessors[p] );
  }
#pragma omp taskwait
  return f( p->predecessors );
}
```

Task example: Fibonacci

$F(0) = 1, \qquad F(1) = 1, \qquad F(n) = F(n-1) + F(n-2)$ for $n \geq 2$.

```
int main() {
  value = new int[nmax+1];
  value[0] = 1; value[1] = 1;
  fib(10);
}

int fib(int n) {
  int i, j, result;
  if (n>=2) {
    i=fib(n-1); j=fib(n-2);
    value[n] = i+j;
  }
  return value[n];
}
```

Task example: Fibonacci

Prevent recomputation:

```
  done = new int[nmax+1];
  for (i=0; i<=nmax; i++)
    done[i] = 0;
  done[0] = 1; done[1] = 1;
  ...
int fib(int n) {
  int i, j;
  if (!done[n]) {
    i = fib(n-1); j = fib(n-2);
    value[n] = i+j; done[n] = 1;
  }
  return value[n];
}
```

Task example: Fibonacci

Use tasks for basic parallelism:

```
int fib(int n) {
  int i, j;
  if (n>=2) {
#pragma omp task shared(i) firstprivate(n)
    i=fib(n-1);
#pragma omp task shared(j) firstprivate(n)
    j=fib(n-2);
#pragma omp taskwait
    value[n] = i+j;
  }
  return value[n];
}
```

Task example: Fibonacci

Again prevent recomputation:

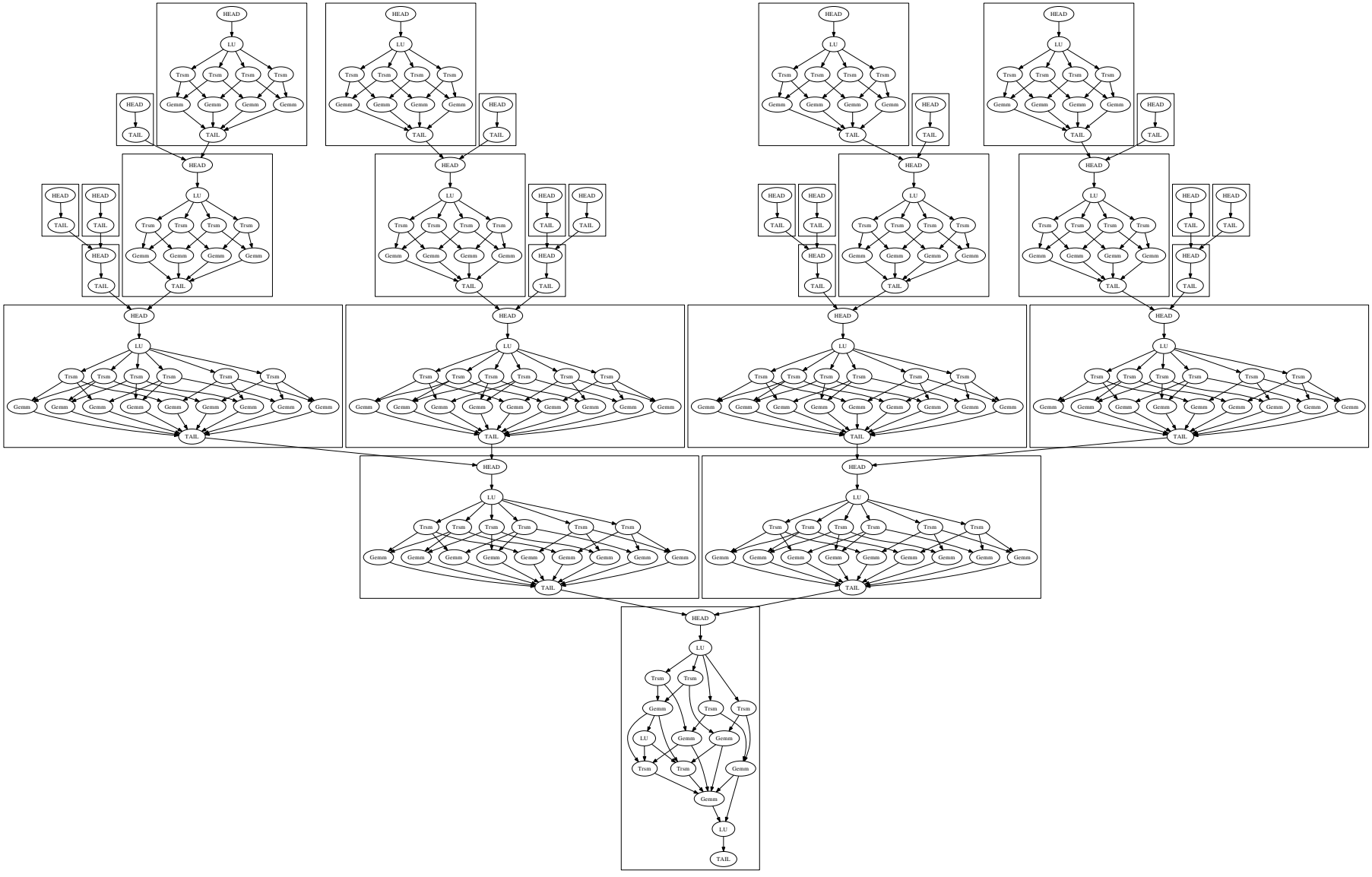
```
int fib(int n) {
  int i, j, result;
  if (!done[n]) {
#pragma omp task shared(i) firstprivate(n)
    i=fib(n-1);
#pragma omp task shared(i) firstprivate(n)
    j=fib(n-2);
#pragma omp taskwait
    value[n] = i+j;
    done[n] = 1;
  }
  return value[n];
}
```

Aside

[H]umans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations. (Sutter and Larus 2005)

Task example: Fibonacci

```
int fib(int n)
{
  int i, j;
  omp_set_lock( &(dolock[n]) );
  if (!done[n]) {
#pragma omp task shared(i) firstprivate(n)
    i = fib(n-1);
#pragma omp task shared(j) firstprivate(n)
    j = fib(n-2);
#pragma omp taskwait
    value[n] = i+j;
    done[n] = 1;
  }
  omp_unset_lock( &(dolock[n]) );
  return value[n];
}
```


HEAD
LU
Trsm
Gemm
TAIL

OMP to infinity and beyond

- Memory model: `flush` directive
- OpenMP version 4: `simd`, `target`, et cetera