CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Shared Parallel Programming Using OpenMP

Instructor: Haidar M. Harmanani

Spring 2019

Synchronization

Synchronization

- Synchronization is used to impose order constraints and to protect access to shared data
- High level synchronization:
  - critical
  - atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

OMP Synchronization Overview

- OpenMP Synchronization
  - OpenMP Critical Sections
    - Defines a critical region on a structured code block
    - Named or unnamed
    - No explicit locks
  - Barrier directives
  - Explicit Lock functions
    - When all else fails – may require flush directive
  - Single-thread regions within parallel regions
    - master, single directives

```
#pragma omp critical [(lock_name)]
{
  /* Critical code here */
}
```

```
#pragma omp barrier
```

```
omp_set_lock( lock_l );
/* Code goes here */
omp_unset_lock( lock_l );
```

```
#pragma omp single
{
  /* Only executed once */
}
```

Barrier Construct

- Explicit barrier synchronization
  - Each thread waits until all threads arrive
  - We will talk about the shared construct later

```
#pragma omp parallel shared (A, B, C)
{
    DoSomeWork(A,B); // Processed A into B

    #pragma omp barrier

     DoSomeWork(B,C); // Processed B into C
}
```


Explicit Barrier

- Several OpenMP constructs have implicit barriers
  - Parallel – necessary barrier – cannot be removed
  - for
  - single
- Unnecessary barriers hurt performance and can be removed with the `nowait` clause

Avoiding Overhead: `nowait` Clause

- Use when threads unnecessarily *wait* between independent computations

```
#pragma omp for nowait
   for(...)
     {...};
```

```
#pragma single nowait
{ [...] }
```

```
#pragma omp for schedule(dynamic,1) nowait
 for(int i=0; i<n; i++)
   a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
 for(int j=0; j<m; j++)
   b[j] = bigFunc2(j);
```


Explicit Barrier: Example

```
#pragma omp parallel shared (A, B, C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
#pragma omp barrier

#pragma omp for
    for(i=0;i<N;i++){
       C[i]=big_calc3(i,A);
    }
#pragma omp for nowait
    for(i=0;i<N;i++){
      B[i]=big_calc2(C,  i);
    }
    A[id] = big_calc4(id);
}
```

implicit barrier at the end of a for work sharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

Avoiding Overhead: `if` clause

- The if clause is an integral expression that, if evaluates to true (nonzero), causes the code in the parallel region to execute in parallel
  - Used for optimization, e.g. avoid going parallel

```
#pragma omp parallel if(expr)
```


Avoiding Overhead: `if` clause

```
#include <stdio.h>
#include <omp.h>

void test(int val)
{
    #pragma omp parallel if (val)
    if (omp_in_parallel())
    {
        #pragma omp single
        printf_s("val = %d, parallelized with %d threads\n",
                 val, omp_get_num_threads());
    }
    else
        printf_s("val = %d, serialized\n", val);
}
```

```
int main( )
{
    omp_set_num_threads(2);
    test(0);
    test(2);
}
```

OpenMP `critical`: Example

- Threads wait their turn – only one at a time calls `consume()` thereby protecting `RES` from race conditions

```
• Naming the critical construct
  RES_lock is optional
• Good Practice – Name all
  critical sections
```

```
float RES;
#pragma omp parallel
{
float B;
#pragma omp for
  for(int i=0; i < niters; i++)
  {
    B = big_job(i);
    #pragma omp critical (RES_lock)
       consume (B, RES);
  }
}
```


Synchronization: `atomic`

- Very similar to the `critical` directive
  - Difference is that `atomic` is only used for the update of a memory location
  - `atomic` is referred to as a mini critical section with a block of one statement

```
#pragma omp parallel
{
  double tmp, B;
  B =  DoIt();
  tmp = big_ugly(B);
  #pragma omp atomic
    x +=  tmp;
}
```

```
Atomic only protects
the read/update of X
```

Synchronization: `atomic` Example

```
#pragma omp parallel for shared(x, y, index, n)
   for (i = 0; i < n; i++) {
      #pragma omp atomic
        x[index[i]] += work1(i);
      y[i] += work2(i);
   }
```


Synchronization: Lock Functions

A lock implies a memory fence of all thread visible variables

- Simple Lock routines:
  - A simple lock is available if it is unset.
    - `omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()`
- Nested Locks
  - A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
    - `omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()`

Synchronization: Lock Functions

- Protect resources with locks.

```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
     id = omp_get_thread_num();
     tmp = do_lots_of_work(id);
     omp_set_lock(&lck);
        printf("%d %d", id, tmp);
     omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

Wait here for your turn.

Release the lock so the next thread gets a turn.

Free-up storage when done.

single Construct

- The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
  - First thread to arrive is chosen
- A barrier is implied at the end of the single block (can remove the barrier with a **nowait** clause).

```
#pragma omp parallel
{
   DoManyThings();
   #pragma omp single
   {
     exchange_boundaries();
   }  // threads wait here for single
   do_many_more_things();
}
```

`master` Construct

- A `master` construct denotes block of code to be executed only by the master thread
  - The other threads just skip it (no synchronization is implied).
  - Identical to the `omp single`, except that the master thread is the thread chosen to do the work

```
#pragma omp parallel
{
   DoManyThings();
   #pragma omp master // if not master skip to next stmt
   {
     ExchangeBoundaries();
   }
   DoManyMoreThings();
}
```


Synchronization: ordered

- Specifies that code under a parallelized for loop should be executed like a sequential loop.

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)

   for (i=0;i < n;i++){
      tmp = Neat_Stuff(i);
     #pragma ordered
      res += consum(tmp);
   }
```

Avoiding Overhead: collapse Clause

- Fuse or collapse perfectly nested loops to exploit a larger iteration space for the parallelization
  - Increase the total number of iterations that will be partitioned across the available number of OMP threads by reducing the granularity of work to be done by each thread
  - If the amount of work to be done by each thread is non-trivial (after collapsing is applied), this may improve the parallel scalability of the OMP application

Avoiding Overhead: collapse Clause

```
#pragma omp for collapse(2)
for(i = 1; i < N; i++)
   for(j = 1; j < M; j++)
      for(k = 1; k < K; k++)
         foo(i, j, k);
```

Iteration space from i-loop and j-loop is collapsed into a single one, if loops are perfectly nested and form a rectangular iteration space.

Worksharing in OMP

SPMD vs. Worksharing

- A parallel construct by itself creates an SPMD or "Single Program Multiple Data" program … i.e., each thread redundantly executes the same code.
- How do you split up pathways through the code between threads within a team?
  - This is called worksharing
    - Loop construct
    - Task construct
    - Sections/section constructs
    - Single construct

Worksharing

- Worksharing is the general term used in OpenMP to describe distribution of work across threads.
- Three examples of worksharing in OpenMP are:
  - `omp for` construct
  - `omp sections` construct
  - `omp task` construct

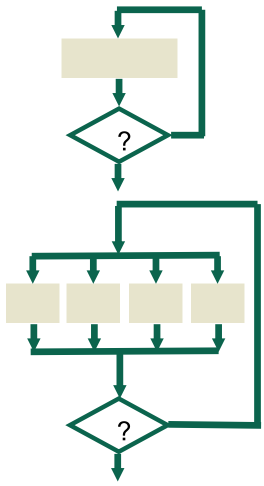
Automatically divides work among threads

OpenMP: Concurrent Loops

- OpenMP easily parallelizes loops
  - No data dependencies between iterations!

```
#pragma omp parallel for
for( i=0; i < 25; i++ ) {
   printf("Foo");
}
```

- Preprocessor calculates loop bounds for each thread directly from serial source



Definition: Loop-Carried Dependencies

- A dependency that exists across iterations
  - if the loop is removed, the dependency *no longer exists.*

```
for(i=1; i<n; i++) {
   S1: a[i] = a[i-1] + 1;
   S2: b[i] = a[i];
}
```

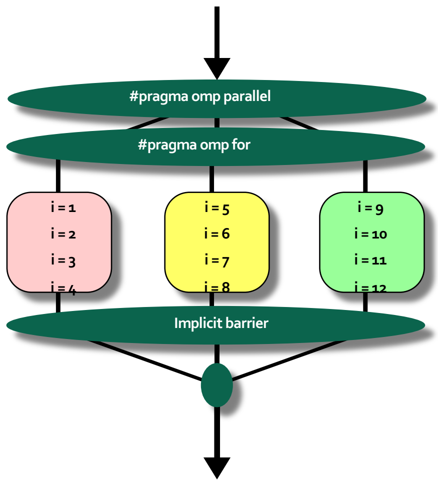
```
S1[i] → T S1[i+1]: loop-carried
S1[i] → T S2[i]: loop-independent
```

```
for(i=1; i<n; i++)
   for(j=1; j<n; j++)
      S3: a[i][j] = a[i][j-1] + 1;
```

```
S3[i,j] → T S3[i,j+1]:
• loop-carried on for j loop
• no loop-carried dependence in
  for i loop
```


OpenMP: Concurrent Loops

```
// assume N=12
#pragma omp parallel for
for(i = 1; i < N+1; i++)
      c[i] = a[i] + b[i];
```



Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent so they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```
int i, j, A[MAX];
 j = 5;
 for (i=0;i< MAX; i++) {
     j +=2;
     A[i] = big(j);
}
```

Note: loop index "i" is private by default

```
int i,  A[MAX];
#pragma omp parallel for
 for (i=0;i< MAX; i++) {
     int j = 5 + 2*(i+1);
      A[i] = big(j);
}
```

Remove loop carried dependence

Working with Loops: Subtle Details

- Dynamic mode (the default mode)
  - The number of threads used in a parallel region can vary from one parallel region to another.
  - Setting the number of threads only sets the maximum number of threads - you could get less.
- Static mode
  - The number of threads is fixed and controlled by the programmer.
- Although you can nest parallel loops in OpenMP, the compiler can choose to serialize the nested parallel region

OpenMP `schedule` Clause

- Determine how loop iterations are divided among the thread team
  - `static([chunk])` divides iterations statically between threads
    - Each thread receives `[chunk]` iterations, rounding as necessary to account for all iterations
    - Default `[chunk]` is `ceil( # iterations / # threads )`
  - `dynamic([chunk])` allocates `[chunk]` iterations per thread, allocating an additional `[chunk]` iterations when a thread finishes
    - Forms a logical work queue, consisting of all loop iterations
    - Default `[chunk]` is 1
  - `guided([chunk])` allocates dynamically, but `[chunk]` is exponentially reduced with each allocation

Loop Work-Sharing: The schedule clause

- `static`
  - Divide the loop into equal-sized chunks or as equal as possible if the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size.
  - By default, chunk size is `loop_count/number_of_threads`.
  - Set chunk to 1 to interleave the iterations.
  - Least work at runtime : scheduling done at compile-time

Loop Work-Sharing: The schedule clause

- `dynamic`
  - Use the internal work queue to give a chunk-sized block of loop iterations to each thread.
  - When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue.
  - By default, the chunk size is 1.
  - Be careful when using this scheduling type because of the extra overhead involved.
  - Least work at runtime : scheduling done at compile-time

Loop Work-Sharing: The schedule clause

- `guided`
  - Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations.
  - The optional chunk parameter specifies them minimum size chunk to use.
  - By default the chunk size is approximately loop_count/number_of_threads.

Loop Work-Sharing: The schedule clause

- `auto`
  - When schedule (auto) is specified, the decision regarding scheduling is delegated to the compiler.
  - The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads in the team.

Loop Work-Sharing: The schedule clause

- `runtime`
  - Uses the OMP_schedule environment variable to specify which one of the three loop-scheduling types should be used.
  - OMP_SCHEDULE is a string formatted exactly the same as would appear on the parallel construct.

OpenMP: Loop Scheduling

```
#pragma omp parallel for schedule(static)

for( i=0; i<16; i++ )
{
  doIteration(i);
}
```

```
// static scheduling

int chunk = 16/T;
int base = tid * chunk;
int bound = (tid+1)*chunk;

for( i=base; i<bound; i++ )
{
  doIteration(i);
}

Barrier();
```


OpenMP: Loop Scheduling

```
#pragma omp parallel for \
 schedule(dynamic)

for( i=0; i<16; i++ )
{
  doIteration(i);
}
```

```
// Dynamic Scheduling

int current_i;

while( workLeftToDo() )
{
  current_i = getNextIter();
  doIteration(i);
}

Barrier();
```

Schedule Clause Example

- Iterations are divided into chunks of 8
  - If start = 3, then first chunk is
  - i={3,5,7,9,11,13,15,17}

```
#pragma omp parallel for schedule (static, 8)
    for( int i = start; i <= end; i += 2 )
    {
       if ( TestForPrime(i) )
        gPrimesFound++;
    }
```


Example

- Parallelize the following:

```
for (i=0; i < NumElements; i++) {
   array[i] = StartVal;   StartVal++;
}
```

Example

- Parallelize the following:

```
for (i=0; i < NumElements; i++) {
   array[i] = StartVal;   StartVal++;
}
```

- Impossible to parallelize due to data dependency
- Fix?

Example: Fix

```
#pragma omp parallel for
for (i=0; i < NumElements; i++) {
    array[i] = StartVal + i;
}

StartVal += NumElements;
```

Sections worksharing Construct

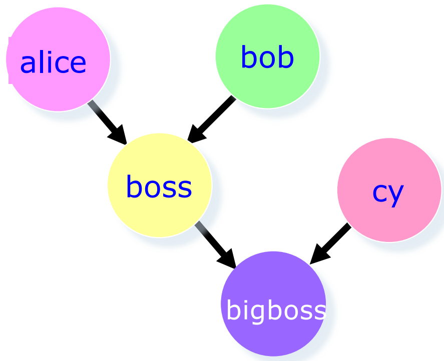
- OpenMP supports non-iterative parallel task assignment using the sections directive.
  - `#pragma omp sections`
    - Must be inside a parallel region
    - Precedes a code block containing of N blocks of code that may be executed concurrently by N threads
    - Encompasses each omp section
  - `#pragma omp section`
    - Precedes each block of code within the encompassing block described above
    - May be omitted for first parallel section after the parallel sections pragma
    - Enclosed program segments are distributed for parallel execution among available threads

Sections Worksharing Construct

- The `omp sections` directive supports the following OpenMP clauses:
  - shared(list)
  - private(list) firstprivate(list) lastprivate(list)
  - default(shared | none)
  - nowait
  - reduction

Decomposition

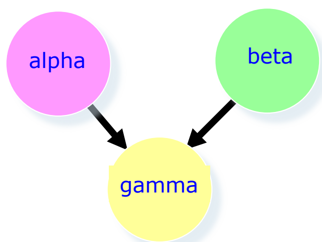
```
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n", bigboss(s,c));
```



Alice ,bob, and cy can be computed in parallel

Sections work sharing Construct

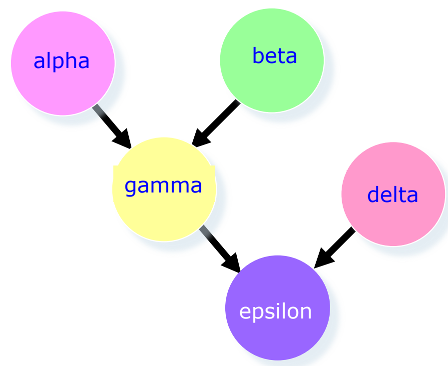
```
#pragma omp parallel sections
{
#pragma omp section  /* Optional */
   a = alpha();
#pragma omp section
   b = beta();
}

printf ("%6.2f\n", gamma(a, b) );
```



By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.

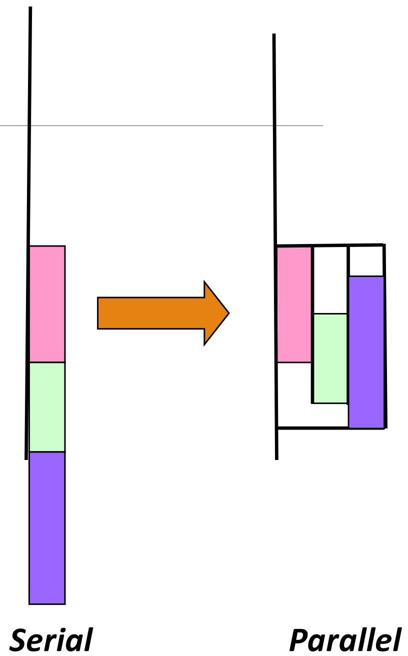
Sections work sharing Construct

```
#pragma omp parallel sections
{
#pragma omp section  /* Optional */
   a = alpha();
#pragma omp section
   b = beta();
}
#pragma omp parallel sections
{
#pragma omp section  /* Optional */
   c = delta();
#pragma omp section
   s = gamma(a, b);
}
printf ("%6.2f\n", epsilon(s,c));
```



Tasks

- Tasks are independent units of work
- Threads are assigned to perform the work of each task
  - Tasks may be deferred or executed immediately
  - The system determines at runtime which case of the above
- Tasks are composed of:
  - code to execute
  - data environment
  - internal control variables (ICV)



OpenMP `task` Worksharing Construct

- The OpenMP tasking model enables the parallelization of a large range of applications.
- The `task` pragma can be used to explicitly define a task.
  - Used to identify a block of code to be executed in parallel with the code outside the task region
  - The task pragma can be useful for parallelizing irregular algorithms such as pointer chasing or recursive algorithms.

OpenMP `task` Worksharing Construct

- The omp task pragma has the following syntax:

```
#pragma omp task [clause[[,] clause] ...] new-line structured-block
```

  - Where a clause is one of the following:
    - if(scalar-expression)
    - final (scalar expression)
    - Untied
    - default(shared | none)
    - Mergeable
    - private(list)
    - firstprivate(list)
    - shared(list)

OpenMP `task` Worksharing Construct

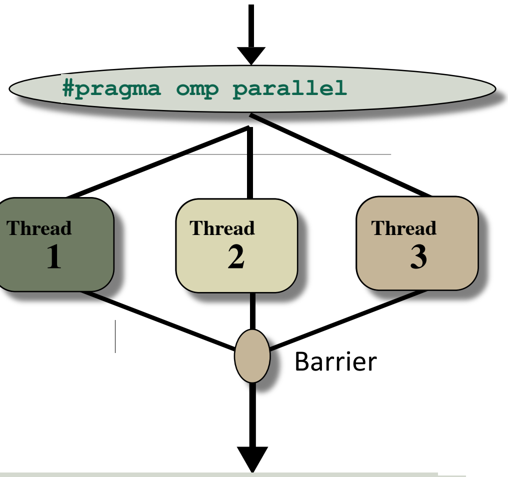
- Developers use a pragma to specify where the tasks are using the assumption that all tasks can be executed independently
- OpenMP Run Time System
  - When a thread encounters a task construct, a new task is generated
  - The moment of execution of the task is up to the runtime system
  - Execution can either be immediate or delayed
  - Completion of a task can be enforced through task synchronization

Tasks versus Sections

- In contrast to tasks, sections are enclosed within the sections construct and (unless the nowait clause was specified) threads will not leave it until all sections have been executed
- Tasks are queued and executed whenever possible at the so-called task scheduling points

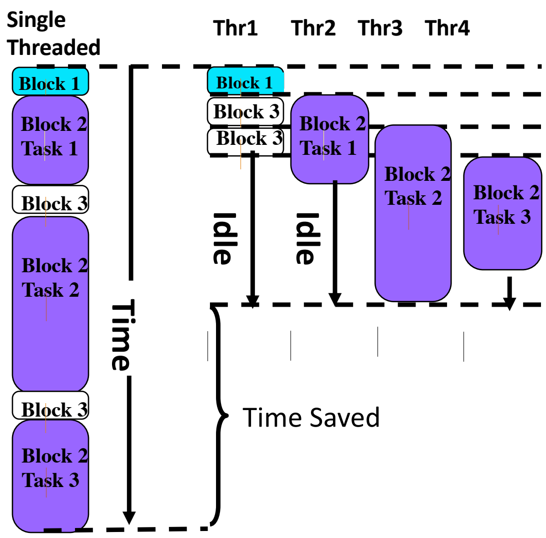
Parallel Construct Implicit Task View

- Tasks are created in OpenMP even without an explicit `task` directive.
  - Let's look at how tasks are created implicitly for the code snippet below
    - Thread encountering parallel construct packages up a set of implicit tasks
    - Team of threads is created.
    - Each thread in team is assigned to one of the tasks (and tied to it).
    - Barrier holds original master thread until all implicit tasks are finished.




```
#pragma omp parallel
    {
        int mydata;
        code…
    }
```


Why are tasks useful?



When are tasks guaranteed to be complete?

- Tasks are guaranteed to be complete at thread or task barriers
  - At the directive: #pragma omp barrier
  - At the directive: #pragma omp taskwait
- Task barrier: `taskwait`
  - Encountering task is suspended until children tasks are complete
  - Applies only to direct children, not descendants!

Task Completion Example

```
#pragma omp parallel
{
   #pragma omp task
   foo();
   #pragma omp barrier
   #pragma omp single
   {
      #pragma omp task
      bar();
   }
}
```

Multiple foo tasks created here – one for each thread

All foo tasks guaranteed to be completed here

One bar task created here

bar task guaranteed to be completed here

Simple Example: || Linked List

```
#pragma omp parallel
// assume 8 threads
{
  #pragma omp single private(p)
  {
    …
    while (p) {
    #pragma omp task
      {
        processwork(p);
      }
      p = p->next;
    }
  }
}
```

A pool of 8 threads is created here

One thread gets to execute the while loop

The single “while loop” thread creates a task for each instance of processwork()

Example: Linked List using Tasks

```
#pragma omp parallel
{
 #pragma omp single nowait
   {
     while (p != NULL) {
       #pragma omp task firstprivate(p)
       {
         processwork(p);
       }
       p = p->next;
     }
   }
 }
```

List Traversal

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
   #pragma omp task
      process(e);
}
```

What's wrong here?

Possible data race !
Shared variable e
updated by multiple tasks

List Traversal

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
#pragma omp task firstprivate(e)
      process(e);
}
```

Good solution – e is firstprivate

List Traversal

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single private(e)
{
   for(e=ml->first;e;e=e->next)
#pragma omp task
      process(e);
}
```

Good solution – e is private

List Traversal

```
List ml; //my_list
#pragma omp parallel
{
   Element *e;
   for(e=ml->first;e;e=e->next)
#pragma omp task
      process(e);
}
```

Good solution – e is private

Task synchronization

```
#pragma omp parallel num_threads(np)
{
#pragma omp task
   function_A();
#pragma omp barrier
#pragma omp single
   {
#pragma omp task
      function_B();
   }
}
```

np Tasks created here, one for each thread

All Tasks guaranteed to be completed here

1 Task created here

B-Task guaranteed to be completed here

Avoiding Overhead: `final` Clause

```
#pragma omp task final(expr)
```

- `final` clause is useful for recursive problems that perform task decomposition
- Stop task creation at a certain depth in order to expose enough parallelism and reduces the overhead.
- The generated task will be a final one if the `expr` evaluates to nonzero value
- All task constructs encountered inside a final task create final and included tasks

Avoiding Overhead: `final` Clause

```
void foo(int arg)
{
  int i = 3;

  #pragma omp task final(arg < 10) firstprivate(i)
   i++;

   printf("%d\n", i); // will print 3 or 4 depending on arg
}
```


A couple of Notes…

- A task is untied if the code can be executed by more than one thread, so that different threads execute different parts of the code.
  - By default, tasks are tied

Avoiding Overhead: `taskyield` Clause

- The **`taskyield`** directive specifies that the current task can be suspended in favor of execution of a different task.
  - Hint to the runtime for optimization and/or deadlock prevention

```
#pragma omp taskyield
```


Avoiding Overhead: `taskyield` Clause

```
#include <omp.h>
void something_useful();
void something_critical();
void foo(omp_lock_t * lock, int n)
{
 for(int i = 0; i < n; i++)
 #pragma omp task
 {
   something_useful();
   while( !omp_test_lock(lock) ) {
   #pragma omp taskyield
 }
 something_critical();
 omp_unset_lock(lock);
 }
}
```

The waiting task may be suspended here and allow the executing thread to perform other work. This may also avoid deadlock situations