

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

Shared Parallel Programming Using OpenMP

Instructor: Haidar M. Harmanani

Spring 2021



More on Sharing and Synchronizing Variables in OpenMP

firstprivate Example

- Variables initialized from shared variable

```
incr = 0;
#pragma omp parallel for firstprivate(incr)

for (i=0;i <= Max; i++) {
    if ((i%2)==0) incr++;
    A(i)= incr;
}
```

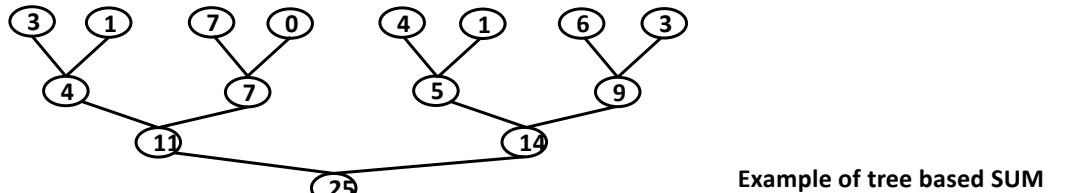
lastprivate Example

- Variables update shared variable using value from last iteration
- C++ objects are updated as if by assignment

```
void sq2(int n, double *lastterm)
{
    double x; int i;
#pragma omp parallel
#pragma omp for lastprivate(x)
    for (i = 0; i < n; i++){
        x = a[i]*a[i] + b[i]*b[i];
        b[i] = sqrt(x);
    }
    lastterm = x;
}
```

Reduction

- Perform a reduction of the data before transferring to the CPU
- Tree based reduction approach used within each thread block



- Reduction decomposed into multiple kernels to reduce number of threads issued in the later stages of tree based reduction

Reduction

- OpenMP reduction clause:
 - reduction (op : list)
- Inside a parallel or a work-sharing construct:
 - A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for "+").
 - Updates occur on the local copy.
 - Local copies are reduced into a single value and combined with the original global value.
- The variables in "list" must be shared in the enclosing parallel region.

```
double ave=0.0, A[MAX];
int i;

#pragma omp parallel for reduction (+:ave)
for (i=0;i< MAX; i++) {
    ave += A[i];
}
ave = ave/MAX;
```

C/C++ Reduction Operations

- A range of associative operands can be used with reduction
- Initial values are the ones that make sense

Operand	Initial Value
+	0
*	1
-	0
^	0

Operand	Initial Value
&	~ 0
	0
&&	1
	0

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)

Synchronization

- 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
 - More about this one later
 - Single-thread regions within parallel regions
 - master, single directives

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

```
#pragma omp barrier  
  
omp_set_lock( lock_1 );  
/* Code goes here */  
omp_unset_lock( lock_1 );  
  
#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

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

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: 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);
}
```

Avoiding Overhead: if clause

- At times it maybe useful to identify conditions when a parallel region should be executed by a single thread or using parallel threads

```
double ave=0.0, A[MAX];
int i;

#pragma omp parallel for reduction (+:ave) if (MAX > 10000)
for (i=0;i< MAX; i++) {
    ave += A[i];
}
ave = ave/MAX;
```

Controlling Threads Execution

Spring 2021

Parallel Programming for Multicore and Cluster Systems

17 | LAU
Lebanese American University

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();
}
```

Spring 2021

Parallel Programming for Multicore and Cluster Systems

18 | LAU
Lebanese American University

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();
}
```

Worksharing

SPMD vs. Worksharing

- A parallel construct by itself creates “Single Program Multiple Data (SPMD)” program
 - Each thread redundantly executes the same code.
- Worksharing
 - Split up pathways through the code between threads within a team
- OpenMP Constructs for Worksharing
 - Loop construct
 - Task construct
 - Sections/section constructs
 - Single construct

Worksharing

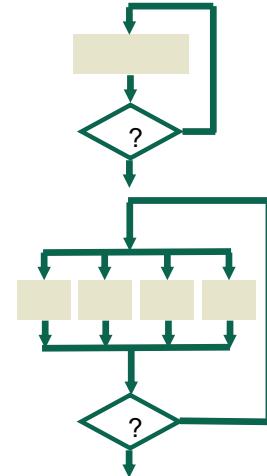
- Worksharing is the general term used in OpenMP to describe distribution of work across threads.
- Three examples of worksharing in OpenMP are:
 - `omp for` construct
 - `omp sections` construct
 - `omp task` construct

Automatically divides work among threads

OpenMP: Concurrent Loops

- 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

```
for( i=0; i < 25; i++ ) {  
    BigTask(i);  
}
```

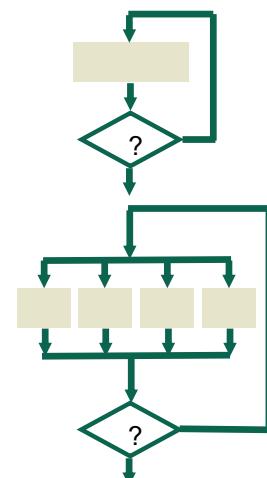


OpenMP: Concurrent Loops

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

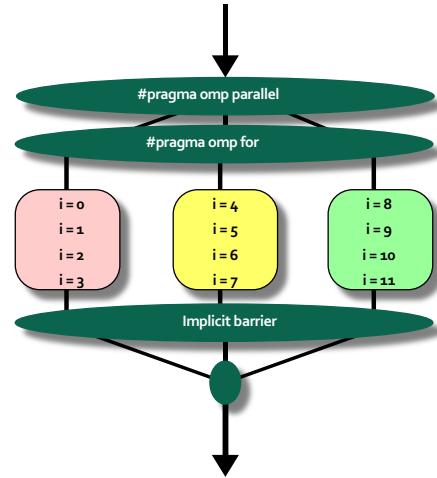
```
#pragma omp parallel for  
for( i=0; i < 25; i++ ) {  
    printf("Foo");  
}
```

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



OpenMP: Concurrent Loops

```
// assume N=12
#pragma omp parallel for
for(i = 0; i < N; i++)
    c[i] = a[i] + b[i];
```



Working with Loops: `schedule` Clause

- Can control how loop iterations are divided among the thread team using the `schedule` clause
 - Static
 - Dynamic
 - Guided
- Although you can nest parallel loops in OpenMP, the compiler can choose to serialize the nested parallel region

Working with Loops: schedule Clause

- Static or schedule(static, chunk-size)
 - 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

OpenMP: Loop Scheduling

```
// static scheduling

#pragma omp parallel for schedule(static)
for( i=0; i<16; i++ )
{
    doIteration(i);
}

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

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

```
schedule(static):
*****
*****
*****
*****
*****
*****
*****
*****
```

```
schedule(static, 4):
****      ****      ****      ****
****      ****      ****      ****
****      ****      ****      ****
****      ****      ****      ****
```

```
schedule(static, 8):
*****      *****
*****      *****
*****      *****
*****      *****
```

Working with Loops: 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

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(dynamic):
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
schedule(dynamic, 1):
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
schedule(dynamic, 4):
***** ***** ***** *****
**** **** **** **** ****
**** **** **** **** ****
**** **** **** **** ****
```

```
schedule(dynamic, 8):
***** ***** ***** *****
***** ***** ***** *****
***** ***** ***** *****
***** *****
```

Working with Loops: `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`.

Working with Loops: *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.

Working with Loops: *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.

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);
```

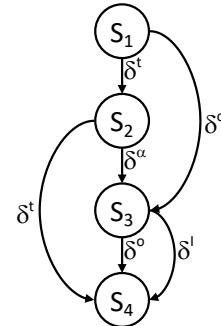
```
#pragma omp for schedule(dynamic,1)
for(int j=0; j<m; j++)
    b[j] = bigFunc2(j);
```

Loop Dependence

Data Dependence

- Data dependence in a program may be represented using a **dependence graph** $G=(V,E)$, where the nodes V represent statements in the program and the directed edges E represent dependence relations.

$S_1 : A = 1.0$
 $S_2 : B = A + 2.0$
 $S_3 : A = C - D$
 :
 $S_4 : A = B/C$



True Dependence and Anti-Dependence

- Given statements S_1 and S_2 ,
 - $S_1;$
 - $S_2;$
- S_2 has a **true (flow) dependence** on S_1 if and only if S_2 reads a value written by S_1
- S_2 has an **anti-dependence** on S_1 if and only if S_2 writes a value read by S_1

$$\begin{array}{c} x = \\ \vdots \\ = x \end{array} \quad \lceil \delta$$

$$\begin{array}{c} = x \\ \vdots \\ x = \end{array} \quad \lceil \delta^{-1}$$

Output Dependence

- Given statements S1 and S2,
S1;
S2;
- S2 has an **output dependence** on S1 if and only if S2 writes a variable written by S1
- Anti- and output dependences are “name” dependencies
 - Are they “true” dependences?
- How can you get rid of output dependences?
 - Are there cases where you can not?

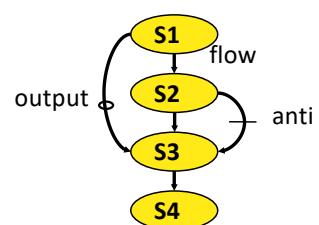
$$\begin{matrix} X = & : & \square \\ & & \leftarrow \\ X = & & \delta^0 \end{matrix}$$

Statement Dependency Graphs

- Can use graphs to show dependence relationships

- Example

S1: $a=1;$
S2: $b=a;$
S3: $a=b+1;$
S4: $c=a;$



- $S_2 \delta S_3$: S_3 is flow-dependent on S_2
- $S_1 \delta^0 S_3$: S_3 is output-dependent on S_1
- $S_2 \delta^{-1} S_3$: S_3 is anti-dependent on S_2

When can two statements execute in parallel?

- Statements S1 and S2 can execute in parallel if and only if there are **no dependences** between S1 and S2
 - True dependences
 - Anti-dependences
 - Output dependences
- Some dependences can be removed by modifying the program
 - Rearranging statements
 - Eliminating statements

How do you determine dependencies?

- Data dependence relations can be found by comparing the IN and OUT sets of each node
- The IN and OUT sets of a statement S are defined as:
 - $\text{IN}(S)$: set of memory locations (variables) that may be used in S
 - $\text{OUT}(S)$: set of memory locations (variables) that may be modified by S
- Note that these sets include all memory locations that may be fetched or modified
 - As such, the sets can be conservatively large

IN / OUT Sets and Computing Dependence

- Assuming that there is a path from S_1 to S_2 , the following shows how to intersect the IN and OUT sets to test for data dependence

$out(S_1) \cap in(S_2) \neq \emptyset \quad S_1 \delta \quad S_2 \quad$ flow dependence

$in(S_1) \cap out(S_2) \neq \emptyset \quad S_1 \delta^{-1} \quad S_2 \quad$ anti - dependence

$out(S_1) \cap out(S_2) \neq \emptyset \quad S_1 \delta^0 \quad S_2 \quad$ output dependence

Loop-Level Parallelism

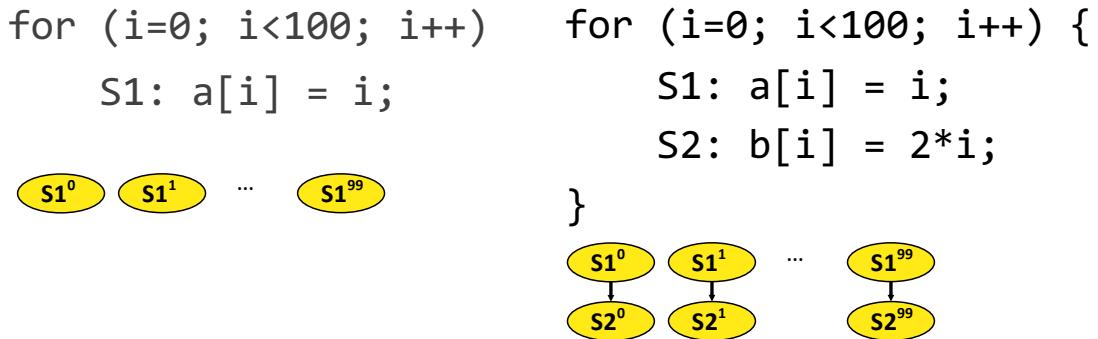
- Significant parallelism can be identified within loops

```
for (i=0; i<100; i++)      for (i=0; i<100; i++) {  
    S1: a[i] = i;          S1: a[i] = i;  
                           S2: b[i] = 2*i;  
}
```

- Dependencies? What about i , the loop index?
- #pragma omp parallel for**
 - All iterations are independent of each other
 - All statements be executed in parallel at the same time
 - Is this really true?

Iteration Space

- Unroll loop into separate statements / iterations
- Show dependences between iterations



Examples

Example 1

- S1: a=1;
- S2: b=1;

Statements are independent

Example 2

- S1: a=1;
- S2: b=a;

Dependent (true (flow) dependence)

- Second is dependent on first
- Can you remove dependency?

Example 3

- S1: a=f(x);
- S2: a=b;

Dependent (output dependence)

- Second is dependent on first
- Can you remove dependency? How?

Example 4

- S1: a=b;
- S2: b=1;

Dependent (anti-dependence)

- First is dependent on second
- Can you remove dependency? How?

Example: 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];  
}
```

$S1[i] \rightarrow T S1[i+1]$: loop-carried
 $S1[i] \rightarrow T S2[i]$: loop-independent

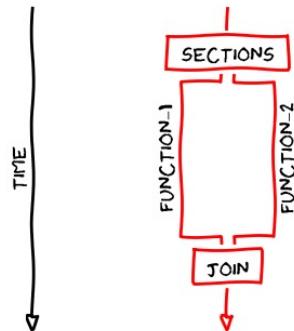
```
for(i=1; i<n; i++)  
    for(j=1; j<n; j++)  
        S3: a[i][j] = a[i][j-1] + 1;
```

$S3[i,j] \rightarrow T S3[i,j+1]$:
• loop-carried on **for j loop**
• no loop-carried dependence in
for i loop

Sections and Tasks

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

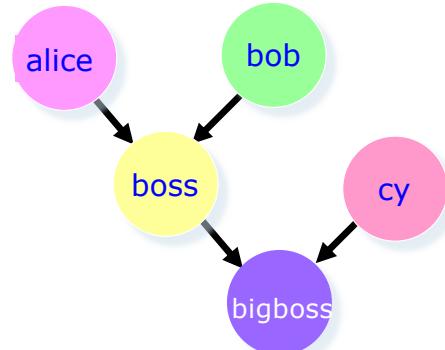


Sections Worksharing Construct

- 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));
```

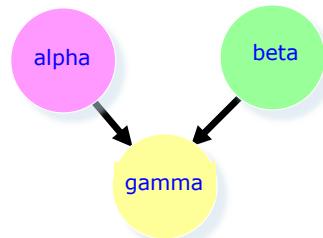


Alice ,bob, and cy can be computed in parallel

Sections work sharing Construct

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

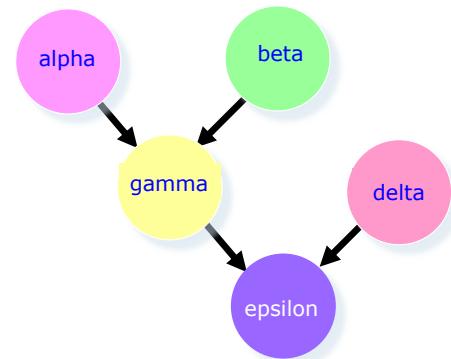
printf ("%6.2f\n", gamma(a, b) );
```



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

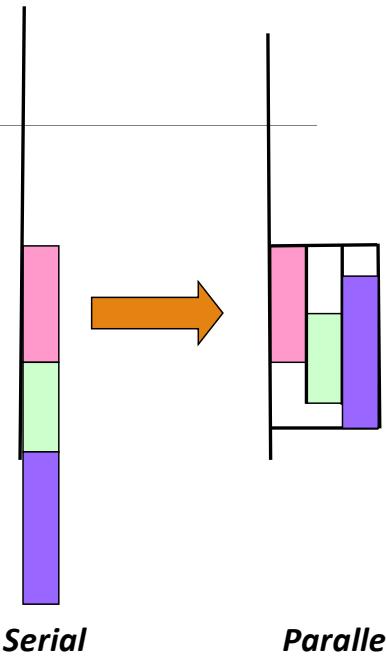
Sections work sharing Construct

```
#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

- 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

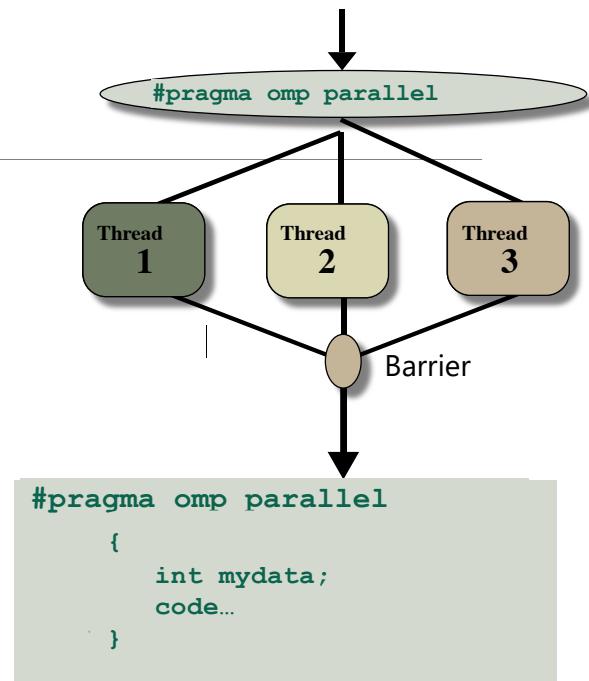
Task synchronization

```
#pragma omp parallel num_threads(np)
{
    #pragma omp task           np Tasks created here, one for each thread
    function_A();
    #pragma omp barrier        All Tasks guaranteed to be completed here
    #pragma omp single
    {
        #pragma omp task         1 Task created here
        function_B();
    }
}
```

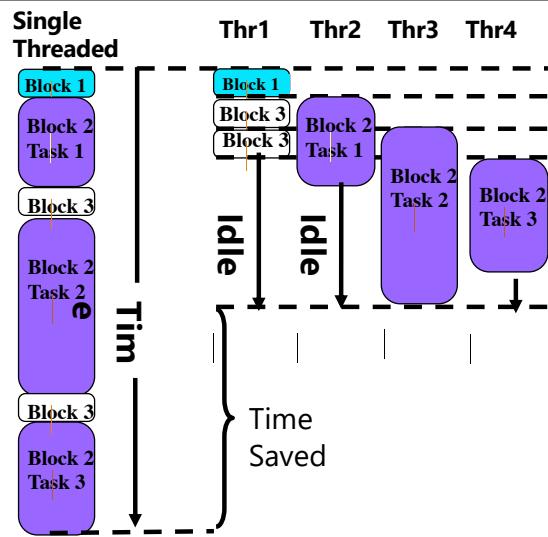
B-Task guaranteed to be completed here

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.



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!

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

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

Closing Comments: Explicit Threads Versus Directive Based Programming

- Directives layered on top of threads facilitate a variety of thread-related tasks.
- A programmer is rid of the tasks of initializing attributes objects, setting up arguments to threads, partitioning iteration spaces, etc.
- There are some drawbacks to using directives as well.
- An artifact of explicit threading is that data exchange is more apparent. This helps in alleviating some of the overheads from data movement, false sharing, and contention.
- Explicit threading also provides a richer API in the form of condition waits, locks of different types, and increased flexibility for building composite synchronization operations.
- Finally, since explicit threading is used more widely than OpenMP, tools and support for Pthreads programs are easier to find.