Introduction to OpenMP

Amitava Datta

University of Western Australia
Compiling OpenMP programs

- OpenMP programs written in C are compiled by (for example): gcc -fopenmp -o prog1 prog1.c
- We have assumed the name of the C file is prog1.c and the name of the executable is prog1
- The compiler will look for OpenMP directives in your program for generating code.
- No action is taken if there are no OpenMP directives in your program.
pragma directive

If you want the compiler to generate code using OpenMP, you have to use the pragma directive

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

int main()
{
    #pragma omp parallel
    {
        printf("The parallel region is executed by thread %d\n", omp_get_thread_num());
    }
}
```
#pragma parallel

- When the compiler encounters the `parallel` directive, it generates multi-threaded code.
- How many threads will execute the code will depend on how many threads are specified (more later).
- The default is number of threads equal to number of cores.
- The parallel region is executed by thread 4
  - The parallel region is executed by thread 3
  - The parallel region is executed by thread 7
  - The parallel region is executed by thread 2
  - The parallel region is executed by thread 5
  - The parallel region is executed by thread 1
  - The parallel region is executed by thread 6
  - The parallel region is executed by thread 0
- But I have only 4 cores in my machine.
Hyperthreading

- Hyperthreading is an Intel technology that treats each physical core as two logical cores.
- Two threads are executed at the same time (logically) on the same core.
- Processors (or cores) do not execute instructions in every clock cycle.
- There is an opportunity to execute another instruction from another thread when the core is idle.
- Hyperthreading schedules two threads to every core.
- So, my processor has 4 physical cores and 8 logical cores.
Hyperthreading

- The purpose of hyperthreading is to improve the throughput (processing more per unit time).
- This may or may not happen. In fact hyperthreading may actually have slower performance.
- Your process will run slower when hyperthreading is turned on.
- It all depends on how well the L1 cache is shared.
- It is possible to turn hyperthreading off through the BIOS (more on lab sheet).
Threads run independently

- There is only one thread until the `parallel` directive is encountered.
- 7 other threads are launched at that point.
- Thread 0 is usually the master thread (that spawns the other threads).
- The `parallel` region is enclosed in curly brackets.
- There is an implied barrier at the end of the parallel region.
What is a barrier?

- A barrier is a place in the process where all threads must reach before further processing occurs.
- Threads may run away without barriers and it is necessary many times to have barriers at different places in a process.
- Barriers are sometime implicit (like here), barriers sometime can be removed (more later).
- Barriers are expensive in terms of run time performance. A typical barrier may take hundreds of clock cycles to ensure that all threads have reached the barrier.
- It is better to remove barriers, but this is fraught with danger.
A variation of our code

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

int main()
{
    #pragma omp parallel
    {
        if (omp_get_thread_num()==3) sleep(1);
        printf("The parallel region is executed by thread %d\n",omp
    }
}
```
Output

The parallel region is executed by thread 4
The parallel region is executed by thread 7
The parallel region is executed by thread 1
The parallel region is executed by thread 2
The parallel region is executed by thread 5
The parallel region is executed by thread 6
The parallel region is executed by thread 0
The parallel region is executed by thread 3

- Thread 3 is now suspended for 1 second, so all other threads complete before thread 3.
Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- Schedule your for and sections
- Memory model
- OpenMP 3.0 and Tasks
OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 20 years of SMP practice
OpenMP BasicDefs: Solution Stack

User layer
- End User

Application

Prog. Layer
- Directives, Compiler
- OpenMP library
- Environment variables

System layer
- OpenMP Runtime library
- OS/system support for shared memory and threading

HW
- Proc1
- Proc2
- Proc3
- ProcN
- Shared Address Space
OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.
  
  ```
  #pragma omp construct [clause [clause]...]
  ```

  Example
  ```
  #pragma omp parallel num_threads(4)
  ```

- Function prototypes and types in the file:
  ```
  #include <omp.h>
  ```

- Most OpenMP* constructs apply to a “structured block”.
  
  Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.

  It’s OK to have an exit() within the structured block.
Exercise 1, Part A: Hello world

Verify that your environment works

- Write a program that prints “hello world”.

```c
void main()
{
    int ID = 0;
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
}
```
Exercise 1, Part B: Hello world
Verify that your OpenMP environment works
● Write a multithreaded program that prints “hello world”.

```c
#include “omp.h”
void main()
{
    #pragma omp parallel
    {
        int ID = 0;
        printf(“ hello(%d )”, ID);
        printf(“ world(%d )\n”, ID);
    }
}
```

Switches for compiling and linking
- fopenmp gcc
- mp pgi
/ Qopenmp intel
Exercise 1: Solution
A multi-threaded “Hello world” program

• Write a multithreaded program where each thread prints “hello world”.

```c
#include "omp.h"
void main()
{
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    printf("hello(%d) ", ID);
    printf("world(%d) 
", ID);
}
}
```

Sample Output:

```
hello(1) hello(0) world(1)
world(0)
hello(3) hello(2) world(3)
world(2)
```

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Runtime library function to return a thread ID.
OpenMP Overview: How do threads interact?

- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.

- Unintended sharing of data causes race conditions:
  - race condition: when the program’s outcome changes as the threads are scheduled differently.

- To control race conditions:
  - Use synchronization to protect data conflicts.

- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.
Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- Schedule your for and sections
- Memory model
- OpenMP 3.0 and Tasks
OpenMP Programming Model:

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program.
Thread Creation: Parallel Regions

- You create threads in OpenMP* with the parallel construct.

- For example, To create a 4 thread Parallel region:

```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

- Each thread calls `pooh(ID,A)` for `ID = 0` to `3`.

* The name “OpenMP” is the property of the OpenMP Architecture Review Board
Thread Creation: Parallel Regions

- You create threads in OpenMP* with the parallel construct.
- For example, To create a 4 thread Parallel region:

```c
double A[1000];
#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

Each thread executes a copy of the code within the structured block.

- Each thread calls `pooh(ID,A)` for `ID = 0 to 3`
Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```

A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*).

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.
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
    - Sections/section constructs
    - Single construct
    - Task construct .... Coming in OpenMP 3.0

Discussed later
The loop worksharing Constructs

- The loop worksharing construct splits up loop iterations among the threads in a team.

```c
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i<N; i++){
        NEAT_STUFF(i);
    }
}
```

**Loop construct name:**
- C/ C++: for
- Fortran: do

The variable `i` is made “private” to each thread by default. You could do this explicitly with a “private(i)” clause.
Loop worksharing Constructs
A motivating example

Sequential code

```c
for(i=0; i<N; i++) { a[i] = a[i] + b[i];}
```

OpenMP parallel region

```c
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    if (id == Nthrds-1) iend = N;
    for(i=istart; i<iend; i++) { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a worksharing for construct

```c
#pragma omp parallel
#pragma omp for
for(i=0; i<N; i++) { a[i] = a[i] + b[i];}
```
Combined parallel/worksharing construct

- **OpenMP shortcut**: Put the “parallel” and the worksharing directive on the same line.

```c
double res[MAX]; int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i< MAX; i++) {
        res[i] = huge();
    }
}
```

```c
double res[MAX]; int i;
#pragma omp parallel for
for (i=0; i< MAX; i++) {
    res[i] = huge();
}
```

These are equivalent.
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

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

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

**Note:** loop index “i” is private by default

**Remove loop carried dependence**
Reduction

- How do we handle this case?

```c
double ave=0.0, A[MAX];  int i;
for (i=0;i< MAX; i++) {
    ave += A[i];
}
ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can’t be trivially removed

- This is a very common situation … it is called a “reduction”.

- Support for reduction operations is included in most parallel programming environments.
Reduction

- OpenMP reduction clause:
  
  \[
  \text{reduction (op : list)}
  \]

- 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 “+”).
  - Compiler finds standard reduction expressions containing “op” and uses them to update 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.

```c
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;
```
OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C/C++ only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
</tr>
<tr>
<td>&amp;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>^</td>
</tr>
<tr>
<td>&amp;&amp;</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fortran Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
</tr>
<tr>
<td>.AND.</td>
</tr>
<tr>
<td>.OR.</td>
</tr>
<tr>
<td>.NEQV.</td>
</tr>
<tr>
<td>.IEOR.</td>
</tr>
<tr>
<td>.IOR.</td>
</tr>
<tr>
<td>.IAND.</td>
</tr>
<tr>
<td>.EQV.</td>
</tr>
<tr>
<td>MIN*</td>
</tr>
<tr>
<td>MAX*</td>
</tr>
</tbody>
</table>
Data environment: Default storage attributes

- **Shared Memory programming model:**
  - Most variables are shared by default

- **Global variables are SHARED among threads:**
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
  - Both: dynamically allocated memory (ALLOCATE, malloc, new)

- **But not everything is shared…**
  - Stack variables in subprograms (Fortran) or functions (C) called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.
double A[10];
int main() {
    int index[10];
    #pragma omp parallel
        work(index);
    printf("%d\n", index[0]);
}

eextern double A[10];
void work(int *index) {
    double temp[10];
    static int count;
    ...
}

A, index, count

A, index, count

temp

temp

temp

A, index, count

A, index and count are shared by all threads.

temp is local to each thread

* Third party trademarks and names are the property of their respective owner.
Data sharing: Changing storage attributes

● One can selectively change storage attributes for constructs using the following clauses*
  – SHARED
  – PRIVATE
  – FIRSTPRIVATE

● The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
  – LASTPRIVATE

● The default attributes can be overridden with:
  – DEFAULT (PRIVATE | SHARED | NONE)

  DEFAULT(PRIVATE) is Fortran only

All data clauses apply to parallel constructs and worksharing constructs except “shared” which only applies to parallel constructs.
Data Sharing: Private Clause

- **private(var)** creates a new local copy of var for each thread.
  - The value is uninitialized
  - In OpenMP 2.5 the value of the shared variable is undefined after the region

```c
void wrong() {
    int tmp = 0;
    #pragma omp for private(tmp)
    for (int j = 0; j < 1000; ++j)
        tmp += j;
    printf("%d\n", tmp);
}
```

`tmp` was not initialized

`tmp`: 0 in 3.0, unspecified in 2.5
Data Sharing: Private Clause
When is the original variable valid?

- The original variable’s value is unspecified in OpenMP 2.5.
- In OpenMP 3.0, if it is referenced outside of the construct
  - Implementations may reference the original variable or a copy ....
    A dangerous programming practice!

```
int tmp;
void danger() {
    tmp = 0;
    #pragma omp parallel private(tmp)
    work();
    printf("%d\n", tmp);
}
```

```
extern int tmp;
void work() {
    tmp = 5;
}
```

tmp has unspecified value

unspecified which copy of tmp
Data Sharing: Firstprivate Clause

- Firstprivate is a special case of private.
  - Initializes each private copy with the corresponding value from the master thread.

```c
void useless() {
    int tmp = 0;
    #pragma omp for firstprivate(tmp)
    for (int j = 0; j < 1000; ++j)
        tmp += j;
    printf("%d\n", tmp);
}
```

Each thread gets its own tmp with an initial value of 0

tmp: 0 in 3.0, unspecified in 2.5
Data sharing: Lastprivate Clause

- Lastprivate passes the value of a private from the last iteration to a global variable.

```c
void closer() {
    int tmp = 0;
    #pragma omp parallel for firstprivate(tmp) \ 
    lastprivate(tmp)
    for (int j = 0; j < 1000; ++j)
        tmp += j;
    printf("%d\n", tmp);
}
```

Each thread gets its own tmp with an initial value of 0

tmp is defined as its value at the “last sequential” iteration (i.e., for j=999)
Data Sharing: A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

  variables A, B, and C = 1
  #pragma omp parallel private(B) firstprivate(C)

- Are A, B, C local to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

Inside this parallel region ...

- “A” is shared by all threads; equals 1
- “B” and “C” are local to each thread.
  - B’s initial value is undefined
  - C’s initial value equals 1

Outside this parallel region ...

- The values of “B” and “C” are unspecified in OpenMP 2.5, and in OpenMP 3.0 if referenced in the region but outside the construct.
Data Sharing: Default Clause

- Note that the default storage attribute is `DEFAULT(SHARED)` (so no need to use it)
  - Exception: `#pragma omp task`
- To change default: `DEFAULT(PRIVATE)`
  - *each* variable in the construct is made private as if specified in a private clause
  - mostly saves typing
- `DEFAULT(NONE)`: *no* default for variables in static extent. Must list storage attribute for each variable in static extent. Good programming practice!

Only the Fortran API supports default(private).

C/C++ only has default(shared) or default(none).
Data Sharing: Default Clause Example

\[
\text{itotal} = 1000 \\
\text{C$\text{OMP PARALLEL PRIVATE}(np, each) \\
np = \text{omp}\_\text{get}\_\text{num}\_\text{threads}() \\
each = \text{itotal}/np \\
\ldots……..}\\n\text{C$\text{OMP END PARALLEL}}
\]

These two code fragments are equivalent

\[
\text{itotal} = 1000 \\
\text{C$\text{OMP PARALLEL DEFAULT(PRIVATE) SHARED(itotal) \\
np = \text{omp}\_\text{get}\_\text{num}\_\text{threads}() \\
each = \text{itotal}/np \\
\ldots……..}\\n\text{C$\text{OMP END PARALLEL}}
\]
Data Sharing: tasks (OpenMP 3.0)

- The default for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope).
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared, because the barrier guarantees task completion.

```c
#pragma omp parallel shared(A) private(B)
{
    ...
    #pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```

- A is shared
- B is firstprivate
- C is private
Data sharing: Threadprivate

- Makes global data private to a thread
  - Fortran: COMMON blocks
  - C: File scope and static variables, static class members
- Different from making them PRIVATE
  - with PRIVATE global variables are masked.
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using COPYIN or at time of definition (using language-defined initialization capabilities).
A threadprivate example (C)

Use threadprivate to create a counter for each thread.

```c
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return (counter);
}
```
Data Copying: Copyin

You initialize threadprivate data using a copyin clause.

```
parameter (N=1000)
common/buf/A(N)
!$OMP THREADPRIVATE(/buf/)

C Initialize the A array
   call init_data(N,A)

!$OMP PARALLEL COPYIN(A)

   … Now each thread sees threadprivate array A initialized
   … to the global value set in the subroutine init_data()

!$OMP END PARALLEL
```

end
Data Copying: Copyprivate

Used with a single region to broadcast values of privates from one member of a team to the rest of the team.

```c
#include <omp.h>
void input_parameters (int, int); // fetch values of input parameters
void do_work(int, int);

void main()
{
    int Nsize, choice;

    #pragma omp parallel private (Nsize, choice)
    {
        #pragma omp single copyprivate (Nsize, choice)
        input_parameters (Nsize, choice);

        do_work(Nsize, choice);
    }
}
```
Synchronization: Barrier

- **Barrier**: Each thread waits until all threads arrive.

```c
#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 parallel region.
- Implicit barrier at the end of a parallel region.
- No implicit barrier due to nowait.
Master Construct

- The **master** construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

```c
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
    { exchange_boundaries(); }
    #pragma omp barrier
    do_many_other_things();
}
```
Single worksharing Construct

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

```c
#pragma omp parallel
{
    do_many_things();
#pragma omp single
    {   exchange_boundaries();   }
    do_many_other_things();
}
```
Synchronization: ordered

- The **ordered** region executes in the sequential order.

```c
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)
for (l=0;l<N;l++){
    tmp = NEAT_STUFF(l);
    #pragma ordered
    res += consum(tmp);
}
```
The *Sections* worksharing construct gives a different structured block to each thread.

```c
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    X_calculation();
    #pragma omp section
    y_calculation();
    #pragma omp section
    z_calculation();
  }
}
```

By default, there is a barrier at the end of the “omp sections” clause. Use the “nowait” clause to turn off the barrier.
loop worksharing constructs: The schedule clause

- The schedule clause affects how loop iterations are mapped onto threads
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - schedule(dynamic [,chunk])
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
  - schedule(guided [,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library … for OpenMP 3.0).
## Loop work-sharing constructs: The schedule clause

<table>
<thead>
<tr>
<th>Schedule Clause</th>
<th>When To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STATIC</strong></td>
<td>Pre-determined and predictable by the programmer</td>
</tr>
<tr>
<td><strong>DYNAMIC</strong></td>
<td>Unpredictable, highly variable work per iteration</td>
</tr>
<tr>
<td><strong>GUIDED</strong></td>
<td>Special case of dynamic to reduce scheduling overhead</td>
</tr>
</tbody>
</table>

**Least work at runtime:** scheduling done at compile-time

**Most work at runtime:** complex scheduling logic used at run-time
Tasks

- Adding tasking is the biggest addition for 3.0

- Worked on by a separate subcommittee
  - led by Jay Hoeflinger at Intel

- Re-examined issue from ground up
  - quite different from Intel taskq’s
General task characteristics

- A task has
  - Code to execute
  - A data environment (it *owns* its data)
  - An assigned thread that executes the code and uses the data

- Two activities: packaging and execution
  - Each encountering thread packages a new instance of a task (code and data)
  - Some thread in the team executes the task at some later time
Definitions

- **Task construct** – task directive plus structured block
- **Task** – the package of code and instructions for allocating data created when a thread encounters a task construct
- **Task region** – the dynamic sequence of instructions produced by the execution of a task by a thread
Tasks and OpenMP

- Tasks have been fully integrated into OpenMP
- Key concept: OpenMP has always had tasks, we just never called them that.
  - Thread encountering parallel construct packages up a set of implicit tasks, one per thread.
  - 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.
- We have simply added a way to create a task explicitly for the team to execute.
- Every part of an OpenMP program is part of one task or another!
task Construct

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

where clause can be one of:

- `if (expression)`
- `untied`
- `shared (list)`
- `private (list)`
- `firstprivate (list)`
- `default( shared | none )`
The `if` clause

- When the `if` clause argument is false
  - The task is executed immediately by the encountering thread.
  - The data environment is still local to the new task...
  - ...and it’s still a different task with respect to synchronization.

- It’s a user directed optimization
  - when the cost of deferring the task is too great compared to the cost of executing the task code
  - to control cache and memory affinity
When/where are tasks complete?

- **At thread barriers, explicit or implicit**
  - applies to all tasks generated in the current parallel region up to the barrier
  - matches user expectation

- **At task barriers**
  - i.e. Wait until all tasks defined in the current task have completed.
    - `#pragma omp taskwait`
  - Note: applies only to tasks generated in the current task, not to “descendants”.
Example – parallel pointer chasing using tasks

```c
#pragma omp parallel
{
    #pragma omp single private(p)
    {
        p = listhead ;
        while (p) {
            #pragma omp task
            process (p)
            p=next (p) ;
        }
    }
}
```

*p* is firstprivate inside this task
Example – parallel pointer chasing on multiple lists using tasks

```c
#pragma omp parallel
{
    #pragma omp for private(p)
    for ( int i = 0; i < numlists ; i++) {
        p = listheads[i];
        while (p) {
            #pragma omp task
            process (p)
            p = next (p );
        }
    }
}
```
Example: postorder tree traversal

```c
void postorder(node *p) {
    if (p->left)
        #pragma omp task
        postorder(p->left);
    if (p->right)
        #pragma omp task
        postorder(p->right);
    #pragma omp taskwait // wait for descendants
    process(p->data);
}
```

- Parent task suspended until children tasks complete
Task switching

- Certain constructs have task scheduling points at defined locations within them.
- When a thread encounters a task scheduling point, it is allowed to suspend the current task and execute another (called task switching).
- It can then return to the original task and resume.
Task switching example

```c
#pragma omp single
{
    for (i=0; i<ONEZILLION; i++)
        #pragma omp task
        process(item[i]);
}
```

- Too many tasks generated in an eye-blink
- Generating task will have to suspend for a while
- With task switching, the executing thread can:
  - execute an already generated task (draining the "task pool")
  - dive into the encountered task (could be very cache-friendly)
Thread switching

```c
#pragma omp single
{
    #pragma omp task untied
    for (i=0; i<ONEZILLION; i++)
        #pragma omp task
        process(item[i]);
}
```

- Eventually, too many tasks are generated
- Generating task is suspended and executing thread switches to a long and boring task
- Other threads get rid of all already generated tasks, and start starving…

- With thread switching, the generating task can be resumed by a different thread, and starvation is over
- Too strange to be the default: the programmer is responsible!
Dealing with taskprivate data

- The Taskprivate directive was removed from OpenMP 3.0
  - Too expensive to implement
- Restrictions on task scheduling allow threadprivate data to be used
  - User can avoid thread switching with tied tasks
  - Task scheduling points are well defined
Conclusions on tasks

- Enormous amount of work by many people
- Tightly integrated into 3.0 spec
- Flexible model for irregular parallelism
- Provides balanced solution despite often conflicting goals
- Appears that performance can be reasonable