CS529 Lecture 05: OpenMP

Dimitrios S. Nikolopoulos

University of Crete and FORTH-ICS

March 15, 2011

Outline

Introduction
- OpenMP

OpenMP
- API
- Scheduling
- Library API

Memory consistency

Tasks

Sources of material

- OpenMP 3.0 specification
  - http://www.openmp.org/mp-documents/spec30.pdf
- OpenMP tutorial, Lawrence Livermore National Lab
  https://computing.llnl.gov/tutorials/openMP/
- Comp 422 course notes, Department of Computer Science, Rice University

What is OpenMP

- De-facto standard API for writing parallel applications based on the abstraction of a shared address space, in C, C++, and Fortran
- Consists of:
  - Compiler directives
  - Run time routines
  - Environment variables
- Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)

First OpenMP example

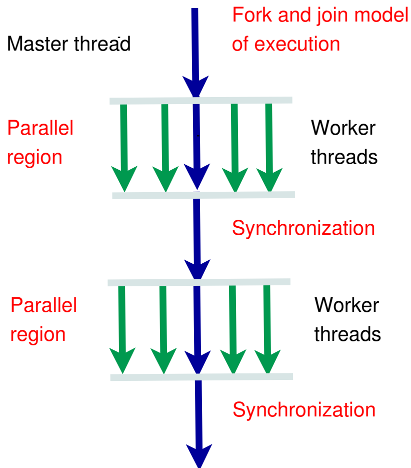
For-loop with independent iterations

```
for (i = 0; i < n; i++)
  c[i] = a[i] + b[i];
```

For-loop parallelized with OpenMP pragma

```
#pragma omp parallel for \
  shared(n, a, b, c) \
  private(i)
for (i = 0; i < n; i++)
  c[i] = a[i] + b[i];
```

OpenMP execution model



OpenMP Terminology

- OpenMP team = Master + Workers
- A parallel region is a block of code executed by all threads simultaneously
  - The master thread always has ID 0
  - Thread adjustment (if enabled) is only done before entering a parallel region
  - Parallel regions can be nested, but support for this is implementation dependent
  - An "if" clause can be used to parallelize optionally and execute serially if condition is not met
- A work-sharing construct divides the execution of the enclosed code region among the members of the team; in other words: they split the work

Components of OpenMP

Directives

- Parallel regions
- Work sharing
- Synchronization
- Data-sharing attributes
  - private
  - firstprivate
  - lastprivate
  - shared
  - reduction
  - threadprivate

Environment variables

- Number of threads
- Scheduling algorithm
- Dynamic thread adjustment
- Nested parallelism

Runtime environment

- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Timers
- API for locks, barriers

Outline

Introduction
- OpenMP

OpenMP
- API
- Scheduling
- Library API

Memory consistency

Tasks

OpenMP directives and clauses

- C-directives are case-sensitive

```
#pragma omp directive [clause[clause]...]
```

- Continuation: use ’ín pragma
- Conditional compilation: `_OPENMP` macro is set
- `if (scalar expression)`
  - Only execute in parallel if expression evaluates to true
  - Otherwise execute serially

```
#pragma omp parallel if (n > threshold) \
  shared (n,x,y) private(i)
  {
    #pragma omp for
      for (i=0; i<n; i++)
        x[i] += y[i];
   } /* -- End of parallel region -- */
```

OpenMP directives and clauses

- `private (list)`
  - No storage association with original object
  - All references are to the local object
  - Values are undefined on entry and exit
- `shared (list)`
  - Data is accessible by all threads in the team
  - All threads access the same address space
- `firstprivate (list)`
  - All variables in the list are initialized to the value the original object had before entering the parallel region
- `lastprivate (list)`
  - The thread that executes the sequentially last iteration updates the values of the objects in the list

Parallel region

```
#pragma omp parallel [clause[[,] clause] ...]
{
 "this is executed in parallel"
} // (implied barrier)
```

A parallel region is a block of code executed by multiple threads simultaneously and supports the following clauses

if	(scalar expression)	
private	(list)	
shared	(list)	
default	(none — shared)	(C/C++)
default	(none — shared — private)	Fortran
reduction	(operator:list)	
copyin	(list)	
firstprivate	(list)	
num_threads	(scalar_int_expression)	

OpenMP programming model

- The clause list is used to specify conditional parallelization, number of threads and data handling
  - Conditional parallelization: The clause `if (scalar expression)` determines whether the parallel construct results in creation of threads
  - Degree of concurrency: The clause `num_threads (integer expression)` specifies the number of threads that are created
  - Data handling: The clause `private (list)` indicates variables local to each thread. The clause `firstprivate (list)` is similar to `private` except values of variables are initialized to corresponding values before the parallel directive. The clause `shared (variable list)` indicates that variables are shared across all threads.

Worksharing constructs in parallel regions

```
#pragma omp for
{
   ...
}
```

```
#pragma omp sections
{
   ...
}
```

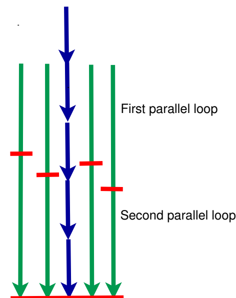
```
#pragma omp single
{
   ...
}
```

- Work is distributed between threads
- Must be enclosed in a parallel region
- Must be encountered by all threads in a team or none at all
- No implied barrier on entry; implied barrier on exit (unless nowait is specified)
- A work-sharing construct does not launch new threads
- Shorthand syntax supported for parallel region with single work sharing construct

```
#pragma omp parallel for
for (...)
```

Example of worksharing OpenMP for loop

```
#pragma omp parallel default (none) \
  shared (n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
      b[i] = (a[i] + a[i+1]) / 2;
    #pragma omp for nowait
    for (i=0; i<n; i++)
      d[i] = 1.0/c[i];
  } /*-- End of parallel region -- */
```



OpenMP reductions

- The `reduction` clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit
- The usage of the reduction clause is `reduction (operator: variable list)`.
- The variables in the list are implicitly specified as being private to threads.
- The operator can be one of: `+, *, -, &, |, ^, &&, ||`

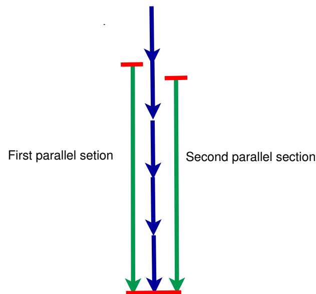
```
#pragma omp parallel reduction (+: sum) num_threads(8) {
  /* compute local sums here */
}
/* sum here contains sum of all instances of sums */
```

OpenMP programming example

```
/* ****************************************************
An OpenMP version of a threaded program to compute PI.
**************************************************** */
#pragma omp parallel default(private) shared (npoints) \
reduction(+: sum) num_threads(8)
{
  num_threads = omp_get_num_threads();
  sample_points_per_thread = npoints / num_threads;
  sum = 0;
  for (i = 0; i < sample_points_per_thread; i++) {
    rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
    rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
    if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
      (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
    sum ++;
  }
}
```

Worksharing sections example

```
#pragma omp parallel default (none) \
  shared (n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
      for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1]) / 2;
      #pragma omp section
      for (i=0; i<n; i++)
        d[i] = 1.0/c[i];
    } /*-- End of sections -- */
  } /*-- End of parallel region -- */
```



single and master constructs in parallel region

only one thread in the team executes enclosed code

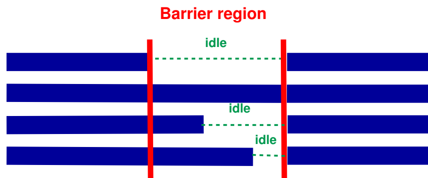
```
#pragma omp single [clause[[,] clause] ...]
{
  <code block>
}
```

only the master thread in the team executes enclosed code

```
#pragma omp master [clause[[,] clause] ...]
{
  <code block>
}
```

- Single and master are useful for computations that are intended for single-processor execution, such as I/O
- No implied barrier on entry or exit of single or master construct

Implicit barriers




```
#pragma omp for
  for (i=0; i<N; i++)
    a[i] = c[i] + b[i];
  /* --- Implicit barrier --- */
#pragma omp for
  for (i=0; i<N; i++)
    d[i] = a[i] + b[i];
```

Barrier is redundant if there is a guarantee that the mapping of iterations is identical on both threads!

nowait clause and explicit barrier

```
#pragma omp for nowait
{
  ...
}
```

```
#pragma omp barrier
```

- To minimize synchronization, some OpenMP directives support an optional nowait clause
- If present, threads do not synchronize/wait at the end of that particular construct
- An explicit barrier can force synchronization at only the desired program points

Second example

```
#pragma omp parallel if (n>limit) default (none) \
  shared (n,a,b,c,x,y,z) private (f,i,scale)
{
  f = 1.0 /* --- Executed by all threads ---*/
#pragma omp for nowait
  for (i=0; i<n; i++)  /* ---Parallel loop, work distributed---*/
    z[i] = x[i] + y[i];
#pragma omp for nowait
  for (i=0; i<n; i++) /* ---Parallel loop, work distributed---*/
    a[i] = b[i] + c[i];
#pragma omp barrier /*---Synchronization---*/
  ...
  scale = sum (a,0,n) + sum (z,0,n) + f; /* --- Executed by all threads --- */
} /*---End of parallel region---*/
```

schedule clause for parallel loops

```
schedule (static | dynamic | guided [, chunk])
schedule (runtime)
static [,chunk]
```

- Distribute iterations in blocks of size ”chunk” over the threads in a round-robin fashion
- In absence of ”chunk” each thread executes $\lceil \frac{N}{P} \rceil$ or $\lfloor \frac{N}{P} \rfloor$ iterations

Example, 4 threads, 16 iterations

TID	0	1	2	3
no chunk	1–4	5–8	9–12	13–16
chunk=2	1–2 9–10	3–4 11–12	5–6 13–14	7–8 15–16

schedule clause for parallel loops

```
dynamic [, chunk]
```

- Fixed portions of work; size is controlled by the value of chunk
- When a thread finishes it starts on the next available portion of work
- Assignment of iterations to threads is non-deterministic (can vary across executions)

```
guided [, chunk]
```

- Same behavior as dynamic but with automatically controlled chunk size
- Chunk size reduced exponentially over time $\lceil \frac{N}{P} \rceil, \frac{N - \lceil \frac{N}{P} \rceil}{P}, \ldots$

schedule clause for parallel loops

runtime

- Scheduling defined by runtime system
- Can be set with `OMP_SCHEDULE` environment variable
- Can be used to implement sophisticated alternatives (e.g. work stealing)

Assigning iterations to threads example

```
/* static scheduling of matrix multiplication loops */
#pragma omp parallel default(private) shared (a, b, c, dim) \
  num_threads(4)
  #pragma omp for schedule(static)
  for (i = 0; i < dim; i++) {
    for (j = 0; j < dim; j++) {
      c(i,j) = 0;
      for (k = 0; k < dim; k++) {
        c(i,j) += a(i, k) * b(k, j);
      }
    }
  }
```

Nesting `parallel` directives

- Nested parallelism can be enabled using the `OMP_NESTED` environment variable
- In this case, each parallel directive creates a new team of threads
- Advantages:
  - Nested parallelism can increase concurrency (availability of parallel work for more cores)
  - Provides more flexibility to the scheduler, opportunities for load balancing
- Disadvantages:
  - May increase overheads and degrade performance if #threads of all teams exceeds #cores
  - Implementation-dependent, no standard behavior

Orphan directives

- OpenMP does not restrict worksharing and synchronization directives (omp for, omp single, omp barrier, omp critical) to be within the lexical extent of a parallel region. If not, directives are orphaned

```
(void) dowork; /*---Sequential for---*/
#pragma omp parallel
{
  (void) dowork; /*---Parallel for---*/
}
```

```
(void) dowork {
#pragma omp for
  for (i=0;...)
  {
    ...
  }
}
```

When an orphaned directive is encountered in the sequential part of the program (outside parallel region) only the master thread executes it (effectively, directive is ignored)

OpenMP library functions

- In addition to directives OpenMP also supports a number of functions that allow a programmer to control the execution of threaded programs

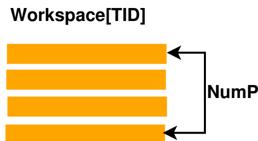
```
/* thread and processor count */
void omp_set_num_threads (int num_threads);
int omp_get_num_threads ();
int omp_get_max_threads ();
int omp_get_thread_num ();
int omp_get_num_procs ();
int omp_in_parallel();
```

Example

```
#pragma omp parallel single (...)
  NumP = omp_get_num_threads();

allocate WorkSpace[NumP][N];

#pragma omp parallel for (...)
for (i=0; i<N; i++)
{
  TID = omp_get_thread_num();
  ...
  WorkSpace[TID][i] = ... ;
  ...
  ... = WorkSpace[TID][i];
  ...
}
```



OpenMP locks

- Simple locks: may not be locked if already in a locked state
- Nestable locks: may be locked multiple times by the same thread before being unlocked
- The API for functions dealing with nested and simple locks is similar

Simple locks

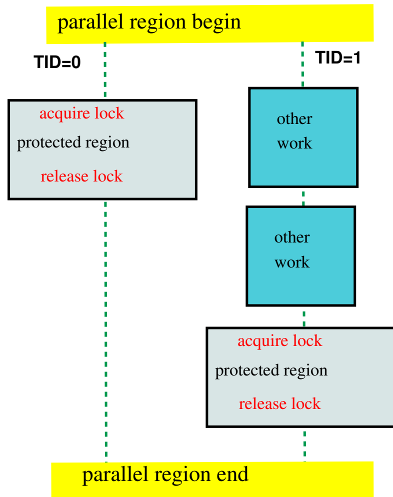
```
omp_init_lock
omp_destroy_lock
omp_set_lock
omp_unset_lock
omp_test_lock
```

Nested locks

```
omp_init_nest_lock
omp_destroy_nest_lock
omp_set_nest_lock
omp_unset_nest_lock
omp_test_nest_lock
```

Critical and atomic constructs can be used instead of locks

Locking example



- Protected region updates shared variables
- One thread acquires the lock and performs the update
- Meanwhile, other thread can perform other work
- When lock is released, other thread performs update on shared variables

Environment variables in OpenMP

- OMP_NUM_THREADS: Specifies the default number of threads created upon entering the following parallel region
- OMP_SET_DYNAMIC: Determines if the number of threads can be dynamically changed
- OMP_NESTED: Turns on nested parallelism
- OMP_SCHEDULE: Scheduling of for loops if the schedule clause specifies that runtime scheduling is used

Shared data in OpenMP

- Global data in OpenMP is shared by default
- Problems arise if multiple threads access shared data simultaneously
  - Read-only data is not a problem
  - Updates need to be checked for race conditions
- It is the programmer's responsibility to deal with this problem in OpenMP
- Solution:
  - Split the global data into a part that is accessed in serial parts only and a part that is accessed in parallel
  - Manually create thread private copies of the latter
  - Use the thread ID to access these private copies
- Alternative: Use OpenMP's `threadprivate` directive

threadprivate directive

```
#pragma omp threadprivate (list)
```

- Creates private copies of designated global variables
- Several restrictions and rules apply:
  - The number of threads has to remain the same for all the parallel regions (i.e. no dynamic threads)
  - Initial data values are undefined, unless `copyin` is used

OpenMP performance optimization hints

- Parallelizing at the outermost level
  - Outer loop preferred over inner loop
    - If it is sufficiently long in execution time
- Parallel regions
  - Use as few parallel regions as possible
    - Each region creation carries significant overhead
    - Enclose as many parallel loops as possible in same region
    - Avoid parallel regions in innermost loops
- Reduce barriers to the bare minimum
  - Use nowait whenever possible
  - Removing waits needs care to avoid data races

OpenMP performance optimization hints

- Minimize the size of critical regions
- Avoid ordered loops
  - Slow due to point-to-point synchronization
- Avoid or minimize false sharing
  - Use private data
  - Experiment with different values of chunk size in loops
  - Try non-static scheduling schemes
- Experimentation:
  - Using master versus single can improve performance
  - Read-only data privatization or sharing

Outline

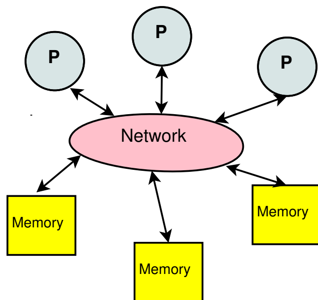
Introduction
- OpenMP

OpenMP
- API
- Scheduling
- Library API

Memory consistency

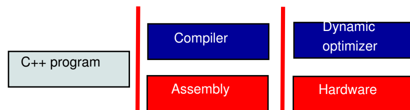
Tasks

Memory consistency models



- Set of rules governing how the memory system will process memory operations from multiple processors
- Contract between the programmer and the system
- Determines what optimizations can be performed for correct programs

What is a memory model

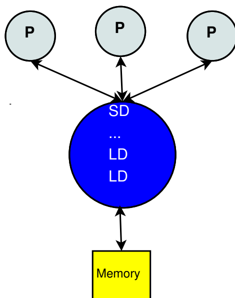


- Interface between program and transformers of the program
  - Determines what values a read returns
- Language level model has implications for hardware and vice versa
- Weakest possible model exposed to the programmer
- Programmer enforces synchronization based on memory model

Uniprocessor memory model

- Model of memory behavior
  - Memory operations occur in program order, read returns the value of the last write in program order
- Semantics defined by sequential program order
  - Simple to reason about but too constrained
    - What really needs to be enforced is control and data dependencies
  - Independent operations can execute in parallel
  - Optimizations preserve these semantics

Sequential consistency



[Lamport]: A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program

Sequential consistency

- SC constrains all memory operations
  - Write $\rightarrow$ Read
  - Write $\rightarrow$ Write
  - Read $\rightarrow$ Read, Write
- SC is a simple model for reasoning about parallel programs
- But, intuitively, unreasonable ordering of memory operations in a uniprocessor may violate sequential consistency model
  - Modern processors reorder memory operations to obtain performance using write buffers, overlapped writes, non-blocking reads
  - Optimizing compilers perform transformations that have the effect of reordering memory operations e.g. scalar replacement, register allocation, instruction scheduling, etc.
  - Programmers may perform similar transformations for software engineering reasons, without realizing that they are changing the program's semantics

SC violation: write buffer

```
//Dekker's algorithm for critical sections
Flag1 = Flag2 = 0;
```

```
P1
Flag1 = 1; //W(Flag1)
if (Flag2 == 0) // R(Flag2)
// critical section
...
```

```
P2
Flag2 = 1; W(Flag2)
if (Flag1 == 0) R(Flag1)
// critical section
...
```

- Relaxed consistency: assume processor with write buffer
  - Example breaks sequential consistency
  - Each processor can read other's flag as 0 while write is pending
  - Violates Dekker's algorithm's correctness

Weak ordering

- Divide memory operations into data operations and synchronization operations
- Synchronization acts as a fence
  - All data operations before synchronization in program order must complete before synch is executed
  - All data operations after synchronization in program order must wait for synchronization to complete
  - All synchronizations are performed in program order
- Hardware implementation of fence: processor has counter that is incremented when data operation is issued and decremented when data operation is completed.

Release consistency

- Further relaxation of weak consistency
- Synchronization accesses are divide into:
  - Acquires: operate like a lock
  - Releases: operate like an unlock
- Semantics of acquire:
  - Acquire must complete before all following memory accesses
- Semantics of release:
  - All memory operations before release are complete
  - but accesses after release in program order do not have to wait for release to complete
  - operations that follow release and depend on release must be protected by an acquire

Data races

- Data races are defined only for SC executions (total order)
- Two memory accesses form a race if:
  - Executed from different threads to same memory location and one is a write
  - May execute consecutively in a SC global total order, i.e. may execute “in parallel”
- Data race free program: no data race in any SC execution

Thread 1	Thread 2
write A, 10	
write B, 20	
	read Flag, 0
write Flag, 1	
	read Flag, 1
	read B, 20

Data races

- Different data operations have different semantics
- Flag is a synchronization variable (wait for flag to update data)
- A, B are data variables
- Separation of synchronization variables from data variables yields best programming practices and optimizations given a consistency model

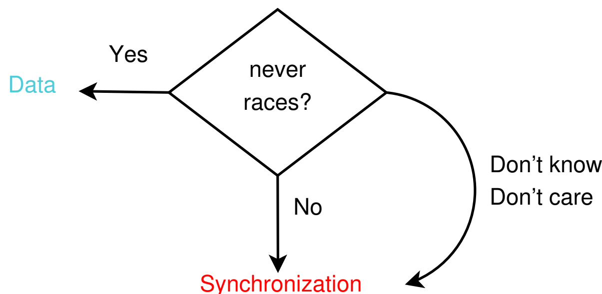
Thread 1	Thread 2
write A, 10	
write B, 20	
	read Flag, 0
write Flag, 1	
	read Flag, 1
	read B, 20

Data-race-free-0 (DRF-0) program

- Data-Race-Free-0 Program
  - All accesses distinguished as either synchronization or data
  - All races distinguished as synchronization (in any SC execution)
- Data-Race-Free-0 Model
  - Guarantees SC to data-race-free-0 programs
  - (For others, reads return value of some write to the location)

Programming with data-race-free-0

- Information required:
  - This operation never races (in any SC execution)
- Write program assuming SC
- For every memory operation specified in the program do:



Distinguish data from synchronization in programming model

- Option 1: annotations of statements

Thread 1	Thread 2
data = ON	synchronization = ON
A=10	while (Flag != 1) {;}
B=20	data=ON
synchronization=ON	... = B;
Flag=1;	... = A;

- Declarations:
  - synch int: Flag
  - data int: A, B
- Option 3: use flush operations or other memory barriers in lieu of synchronization = ON, OFF

OpenMP memory model

- All shared and private variables have original variables
- Shared access to a variable:
  - Within a structured block, references to the variable all refer to the original variable
- Private access to a variable:
  - A variable of the same type and size as the original variable is provided for each thread

OpenMP memory model

- A form of weak ordering
- All accesses to variables may be reordered unless the programmer annotates code with memory barriers
- Memory barriers implemented with flush directive in OpenMP
  - Different memory barriers must be sequentially consistent
  - No reordering across memory barriers
    - W $\rightarrow$ flush, R $\rightarrow$ flush, flush $\rightarrow$ R, flush $\rightarrow$ W, flush $\rightarrow$ flush
- A flush construct has an optional list of variables that must be flushed
  - If pointer present in the list then only pointer is flushed not pointed to data
- If no list is given entire thread state flushed
  - May be necessary due to hardware implementation limitations

Memory latency overlap with flush

```
a = . . .;  /* a can be committed here... */
<other computation>
#pragma omp flush(a) /* or as late as here... */
```

Reordering example

```
a = ...;  //(1)
b = ...;  //(2)
c = ...;  //(3)
#pragma omp flush(c)    //(4)
#pragma omp flush(a,b)  //(5)
. . . a . . . b . . .;  //(6)
. . . c . . .;          //(7)
```

- (1) and (2) may not be moved after (5)
- (6) may not be moved before (5)
- (4) and (5) may be interchanged at will.

Moving data between threads

- To move the value of a shared variable from thread 1 to thread 2:
  - Write variable on thread 1
  - Flush variable on thread 1
  - Flush variable on thread 2
  - Read variable on thread 2

Using the flush directive

```
/*  Announce that I am done with my work. The first flush
 *  ensures that my work is made visible before synch.
 *  The second flush ensures that synch is made visible.
 */
 #pragma omp flush(work,synch)
  synch[iam] = 1;
 #pragma omp flush(synch)
     /* Wait for neighbor. The first flush ensures that synch is read
      * from memory, rather than from the temporary view of memory.
      * The second flush ensures that work is read from memory, and
      * is done so after the while loop exits.
      */
  neighbor = (iam>0 ? iam : omp_get_num_threads()) - 1;
  while (synch[neighbor] == 0) {
    #pragma omp flush(synch)
  }
 #pragma omp flush(work,synch)
 /* Read neighbor's values of work array */
 result[iam] = fn(work[neighbor], work[iam]);
```

Implicit flushes

- In barriers
- At entry to and exit from
  - Parallel, parallel worksharing, critical, ordered regions
- At exit of worksharing regions (unless nowait specified)
- In `omp_set_lock`, `omp_set_nest_lock`, `omp_unset_lock`, `omp_unset_nest_lock`
- In `omp_test_lock`, `omp_test_nest_lock`, if lock is acquired
- At entry to and exit from `atomic`, flush set is the atomically updated variable

OpenMP ordering vs. weak ordering

- OpenMP reordering restrictions are similar to weak ordering with ”flush” defined as a ”synch” operation
- Actually weaker than weak ordering:
  - Synchronization operations on disjoint variables are not ordered with respect to each other
- Relaxed memory model enables OpenMP implementations on distributed-memory architectures, such as clusters and accelerators.

Use of list in flush directives

Incorrect example:

```
                a = b = 0

thread 1                      thread 2

b = 1                         a = 1
flush(b)                      flush(a)
flush(a)                      flush(b)
if (a == 0) then              if (b == 0) then
   critical section              critical section
end if                        end if
```

Correct example:

```
                a = b = 0

thread 1                      thread 2

b = 1                         a = 1
flush(a,b)                    flush(a,b)
if (a == 0) then              if (b == 0) then
   critical section              critical section
end if                        end if
```

Outline

Introduction
- OpenMP

OpenMP
- API
- Scheduling
- Library API

Memory consistency

Tasks

OpenMP tasks (OpenMP v3.0 onwards)

- A task is:
  - A piece of code to execute
  - A data environment (task owns the data)
  - An assigned thread that executes the code and accesses the data
- Two activities: packaging (instantiation) and executing
  - Each encountering thread packages a new instance of a task (code and data)
  - Some thread in the team executes the task at a 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 the task construct
- Task region: the dynamic sequence of instructions produced by the execution of a task by a thread

Task construct

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

clause can be one of:

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

The `if` clause

- When `if` clause argument is false
  - The task is executed immediately by the encountering thread
  - The data environment is still local to the new task
  - The task is still an independent task with respect to synchronization
- User-directed optimization
  - when the cost of deferring the task for parallel execution is too high compared to executing the task locally
  - controls cache and memory affinity

When and where are tasks completed?

- At thread barriers, implicit or explicit
  - applies to all tasks generated in the current parallel region up to the barrier
  - matches user expectation
- At task barriers
  - applies only to children tasks generated in the current task , not to "descendants"
  - `#pragma omp taskwait`

Pointer chaining using tasks

```
#pragma omp parallel
{
  #pragma omp single private(p)
  {
    p = listhead ;
    while (p) {
       #pragma omp task  /* spawn call to process(p) */
         process (p)
       p=next (p) ;
     }
   }
} /* Implicit taskwait */
```

Pointer chaining on multiple lists using tasks

```
#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) ;
       }
   }
}
```

Postorder tree traversal

```
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 child tasks
  process(p->data);
}
```

Task switching

- Certain constructs have task scheduling points at defined locations
- When a thread encounters a task scheduling point it can suspend the current task and switch to execute another task
- Following the other task execution, the thread can return to the suspended task

Task switching example

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

- Too many tasks created in very short time
- Generating task does not have chance to execute computation until it creates all other tasks
- With task switching, the generating thread can:
  - execute already generated tasks (draining the task queue)
  - execute the first encountered task (cache-friendly)

Task switching example

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

- Too many tasks created in very short time
- Generating task does not have chance to execute computation until it creates all other tasks
- With task switching, the generating thread can:
  - execute already generated tasks (draining the task queue)
  - execute the first encountered task (cache-friendly)
- Must be provided by programmer!