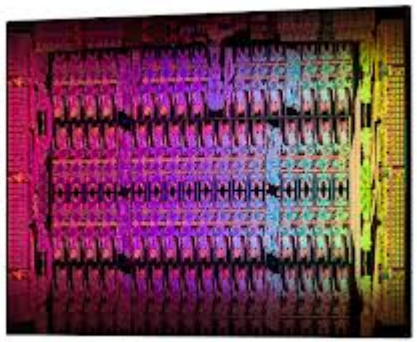
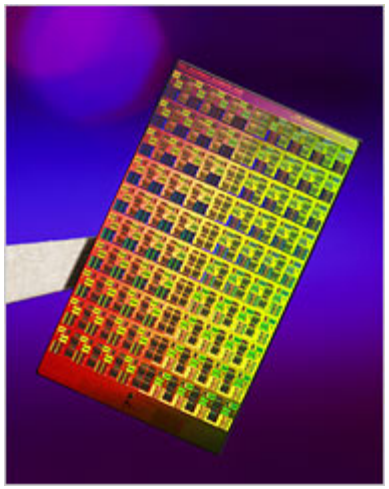
Parallel Computing for Science & Engineering



Intel Xeon Phi,

02/07/2018

Instructors:
Lars Koesterke, TACC
Charlie Dey, TACC



Intel Teraflop chip, 2007

OpenMP Setting Number of Threads

1. ***Before run: by Environment Variable, OMP_NUM_THREADS***
2. ***In an OpenMP API Function***
3. ***In a Clause:***

1. **% setenv OMP_NUM_THREADS #**
   % export OMP_NUM_THREADS=#

Env. Var.

precedence over implementation defined.

2. **omp_set_num_threads(#)**

API

precedence over 1.

3. **!$omp parallel num_threads(*int. expression*)**
   #pragma omp parallel num_threads(*int expression*)

Clause

precedence over 1 and 2.

OpenMP Setting Number of Threads

0. OMP_THREAD_LIMIT (Env.Var.) Sets limit over all methods. Default value = implementation defined (e.g 2^32 on intel)

1. **% setenv OMP_NUM_THREADS #**
   % export OMP_NUM_THREADS=#

2. **omp_set_num_threads(#)**

   omp_set_num_threads(omp_get_num_procs()) *

3. **!$omp parallel num_threads(*int. expression*)**
   #pragma omp parallel num_threads(*int expression*)

```
omp_get_num_threads()        Reports # of threads in team.
omp_get_max_threads()        Reports # of threads by 1, 2 **.
export OMP_THREAD_LIMIT=#    If 1,2 or 3 exceeds limit#, use limit #
omp_get_thread_limit()       Reports OMP_THREAD_LIMIT.
```

TACC

* If hyperthreading (2-SMT) is on : procs=cores*2

** "Max Threads is # of threads that could be used by a parallel region without a num_threads clause."

OpenMP Setting Number of Threads

```c
#include <stdio.h>
#include   <omp.h>
//env. Var. OMP_NUM_THREADS =12
int main(){
int nt, N=1000;
printf("PROCS = %d\n",\
omp_get_num_procs());

omp_set_num_threads(4);

#pragma omp parallel num_threads(N/100)
   {

     printf("thrds %d max %d\n",\
       omp_get_num_threads(),\
       omp_get_max_threads());
    }
}
```

```fortran
program nthreads

! Env. var. OMP_NUM_THREADS=12
use omp_lib
integer, parameter :: N=1000
print*, "PROCS = ", &
        omp_get_num_procs()

call omp_set_num_threads(4)

!$omp parallel num_threads(N/100)


 print*,  &
 "thrds",omp_get_num_threads(),&
 "max",  omp_get_max_threads()
!$omp end parallel
end program
```

TACC

lscpu command reports →

Thread(s) per core: 2
Core(s) per socket: 12
Socket(s): 2

OpenMP Thread and Memory Location

Where do threads/processes and memory allocations go?

Default: Decided by policy when process exec'd or thread forked, and on write to memory . Processes and threads can be rescheduled to different sockets and cores.

Ways Process Affinity and Memory Policy can be changed:

1.) Dynamically on a running process (knowing process id)
2.) At process execution (with wrapper command)
3.) Within program through F90/C API

Barrier Example

- Barrier: Each thread waits until all threads arrive

```
#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,N);
   }                                          Implicit barrier
   #pragma omp for nowait
   for(i=0;i<N;i++){
           B[i]=big_calc2(C, I, N);
   }                                          No Implicit barrier due to
   A[id] = big_calc4(id);                     nowait
}                                             Implicit barrier
```

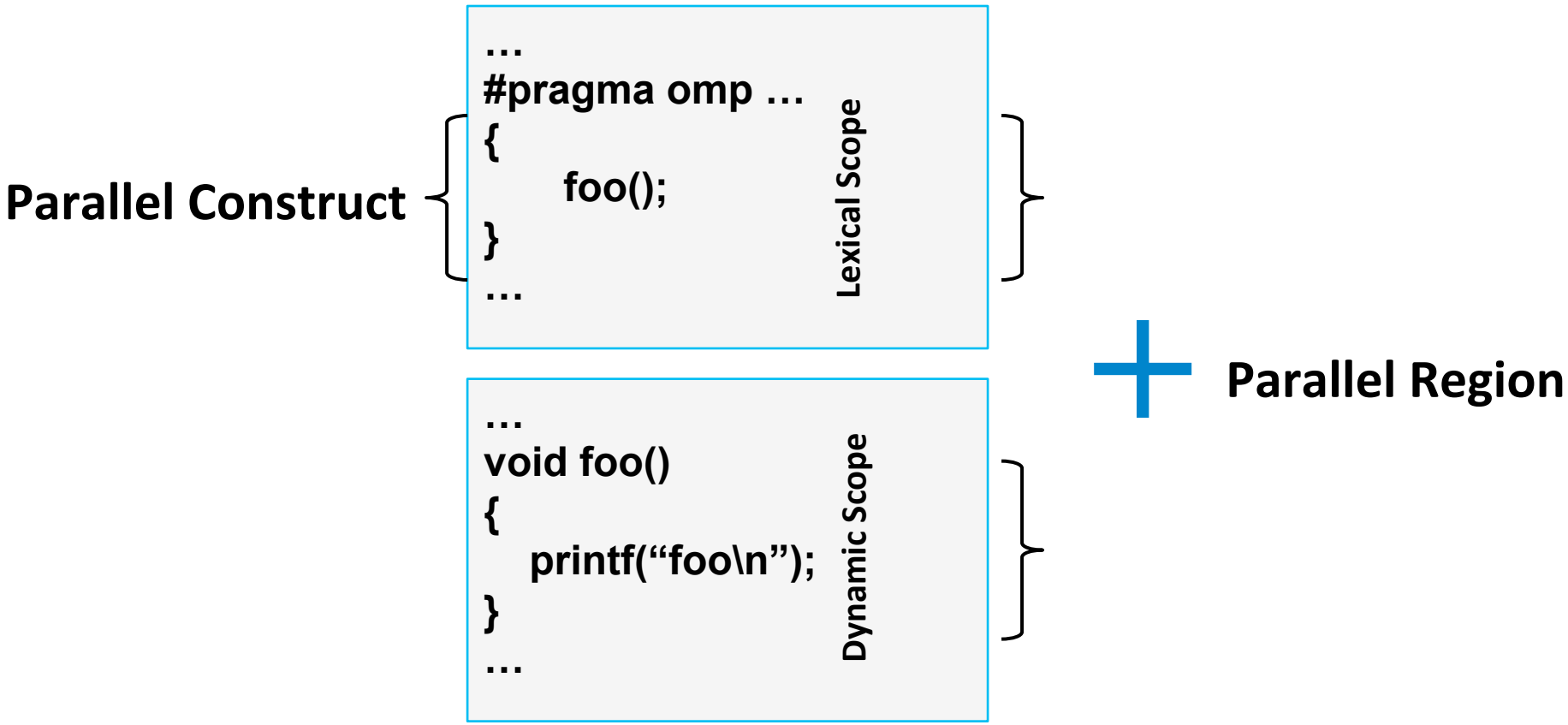
Barrier Example

- Barrier: Each thread waits until all threads arrive

```
!$omp parallel shared (A, B, C) private(id)

   id=omp_get_thread_num()
   A(id) = big_calc1(id)
   !$omp barrier
   !$omp do
   do i=0,N
          C(i)=big_calc3(i,A,N)
   end do <-------------------------- Implicit barrier
   !$omp do
   do i=1,N
       B(i)=big_calc2(C, I, N)
   end do nowait <------------------- No implicit barrier due to
   A(id} = big_calc4(id)                nowait
!$omp end parallel <---------------- Implicit barrier
```

Concepts


Parallel Construct
…
#pragma omp …
{
foo();
}
…
Lexical Scope
+
Parallel Region
…
void foo()
{
printf("foo\n");
}
…
Dynamic Scope

OpenMP Orphaned Work-sharing Constructs

- Work-sharing directives that appear outside the lexical extent of a parallel directive are called orphaned work sharing constructs.
  (lexical extent: also called the “parallel construct”)

- When in the dynamic extent (i.e, within a called function or subroutine in a parallel region), the work-sharing construct behavior is identical (almost) to a work-sharing construct within the parallel region.

- When encountered from outside a parallel region (i.e. called from a serial portion of code) the master thread is the “team of threads”. It is safely invoked as serial code.

OpenMP Orphaned Work-sharing Constructs

```
!$omp parallel
  call work(n,a,b,c)
!$omp end parallel

  call work(n,a,b,c)
!-----
subroutine work(n,a,b,c)
use omp_lib
integer n, id, i
real*8, dimension(n):: a,b,c

   print*,omp_get_num_threads()

!$omp do
   do i = 1,n
      a(i) = b(i) + c(i)
   end do

end subroutine
```

```
#pragma omp parallel
  work(n,a,b,c);


  work(n,a,b,c);
//----
int work(int n, double *a,…){

int id, i;


printf("%d\n",omp_get_num_threads());

#pragma omp for
   for(i=0; i<n; i++){
      a[i] = b[i] + c[i];
   }

}
```

These variables take on the share/private attributes of the callee.

modified

OpenMP if clause

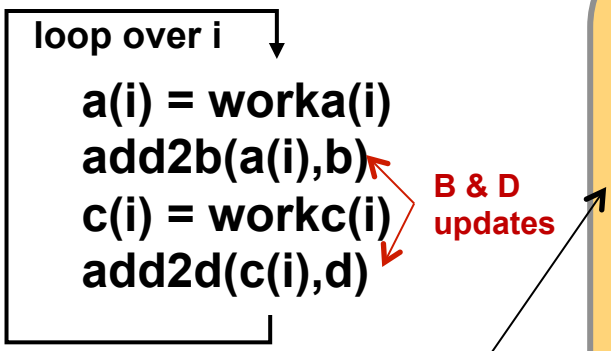
- Syntax of clause:
  … **`parallel if(`*`logical expression`*`)`**
- Executes region as a parallel region if true, otherwise executed serially as a team of 1 thread.

```
!$omp           parallel if( N > 10000)
#pragma omp parallel if( N > 10000)
```

modified

OpenMP critical region

- One thread at a time–
  - Can exist in parallel section, and in orphaned or serial code
  - Critical namespace is global. Same-named critical sections share a single lock.
- No guarantee for entry fairness, but
- Guaranteed forward processing
- Named critical region are independent **see below:



As separately named critical sections, B and D may have threads executing simultaneously.

```
!$omp   parallel  do
do i = 1,n
      a(i) = worka(i)
!$omp critical (B)
      call add2b(a(i),b)
!$omp end critical (B)
      c(i) = workc(i)
!$omp critical (D)
      call add2d(c(i),d)
!$omp end critical (D)
end do
```

```
#pragma omp parallel  for
{   for( i=0; i<n; i++)
      a[i] = worka(i);
#pragma omp critical (B)
      add2b(&a[i],&b);

      c[i] = workc(i);
#pragma omp critical (D)
      add2d(&c[i],&d);

}
```

OpenMP flush directive

- Syntax: **!$omp flush [*list( )*]**
  #pragma omp flush [*list( )*]

- A memory fence that inhibits movement of memory operations across the synchronization point.
- Point where executing thread has a consistent view of memory (of shared variables)
  - All memory operations (read/write) before synch. pt. must be performed before synch. pt. (no store later)
  - Likewise, all memory operations after synch. pt. must occur after synch. pt. (e.g. no prefetching across fence)
- Implicit flush (all shared variables, no list of variables)
  - Barrier, Entry/Exit Parallel, Critical & Ordered regions
  - Exit of worksharing region (except nowait), locks/tasks

OpenMP directives & clauses

Directive

Clause	parallel	do/for	sections	single	parallel do/for	parallel sections
if	X				X	X
default	X				X	X
shared	X				X	X
private	X	X	X	X	X	X
firstprivate	X	X	X	X	X	X
lastprivate		X	X		X	X
reduction	X	X	X		X	X
schedule		X			X	
nowait		X	X	X		

These don't accept clauses:

master
critical
barrier
atomic
flush

Table: Acceptable clauses for directives.

OpenMP runtime

Name	Type	Chunk	Chunk Size	Number of Chunks	Static or Dynamic	Computer Overhead
Simple Static	**simple**	no	N/P	P	static	lowest
Interleaved	simple	yes	C	N/C	static	low
Simple dynamic	dynamic	optional	C	N/C	dynamic	medium
Guided	guided	optional	decreasing from N/P	fewer than N/C	dynamic	high
Runtime	runtime	no	varies	varies	varies	varies

Guided (p=#of threads, n=# of iterations): starts at n/p, next thread get #unassigned/p, etc. and decrements until chunk size is reached.