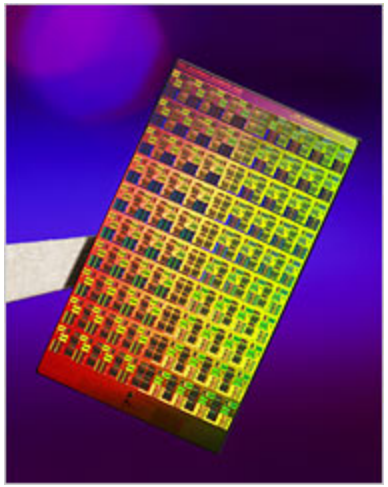
Parallel Computing for Science & Engineering
CS395T

2/12/08



Instructors:

Dr. Bill Barth, Research Associate, TACC

Dr. Kent Milfeld, Research Associate, TACC

http://www.nytimes.com/2007/02/12/technology/12chip.html?_r=2&oref=slogin&oref=slogin

OpenMP Setting Number of Threads

1. ***Before run: by Environment Variable, OMP_NUM_THREADS***
2. ***Within Application, as directed, or determined from available processors***
3. ***By OS, using dynamic threads (implementation dependent)***

1. **% setenv OMP_NUM_THREADS #**

Set number of threads (#) In shell before executing a.out

2. **omp_set_num_threads(#)**

   omp_set_num_threads(omp_get_num_procs())

Set number of threads (#) In serial region of code. Or Set number to number of processors

3. **% setenv OMP_DYNAMIC TRUE**

   omp_set dynamic(*true/false*)

Set number of threads to processors available to the application at run time– use either environment variable or API call inside code.

OpenMP Setting Number of Threads

User:
Requested Threads = R#
Environment OMP_NUM_THREADS
API opm_set_num_threads()

Operating System:
Available CPUs = A#
Processor Affinity

Threads may be bound to specific processors.

CPUs Dedicated (batch system)	**R# <= A#**	**1-to-1 map between threads and CPUs**
	R# > A#	**Many-to-1 map between threads and CPUs** **Execution may be unbalanced**
CPUs Shared	**R# <= A#** **R# > A#**	**Time slicing & OS thread scheduling** **Longer wall-clock time** **Unbalanced Execution**

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 when memory allocated. Processes and threads can be rescheduled to different sockets and cores.

Scheduling Affinity and Memory Policy can be changed on Linux systems within code with:

sched_get/setaffinity
get/set_memory_policy,

they can be set/changed outside of code with:

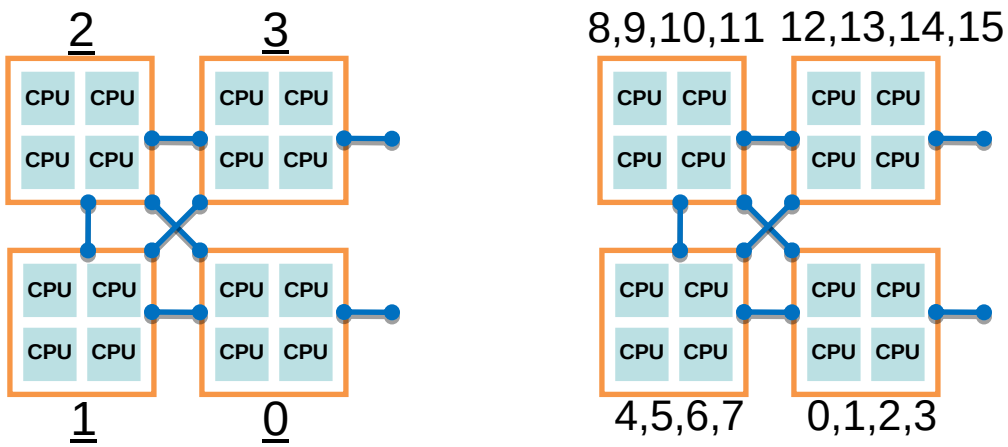
numactl

NUMA Operations (cont. 1)

- Affinity and Policy can be changed externally through numactl at the socket and core level.

```
Command:    numactl  <options>  ./a.out
```

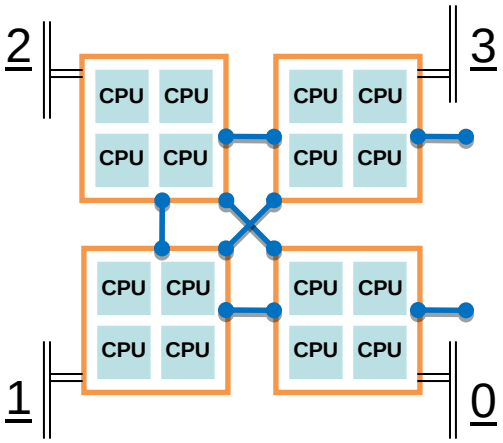
E.G. Ranger Node



Socket References

Core References

NUMA Operations (cont. 2)

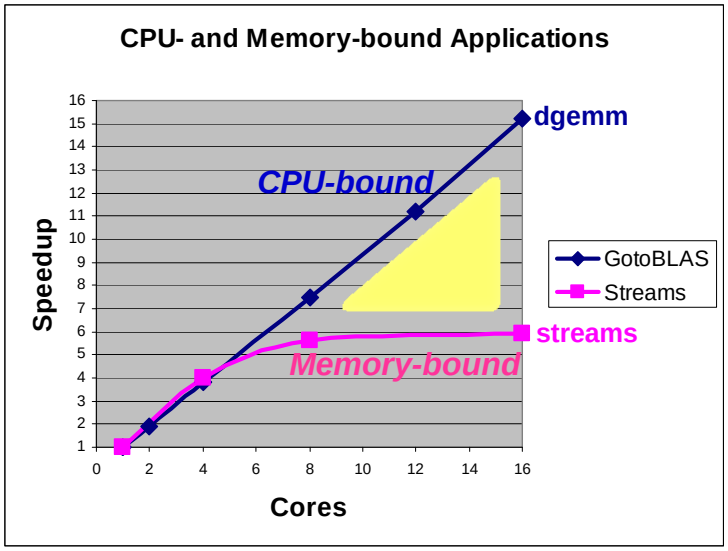


Memory: Socket References

- MPI – local is best
- SMP – Interleave best for large, completely shared arrays
- SMP – local best for private arrays
- Once allocated, a memory structure's is fixed

NUMA Operations (cont. 3)

- Load Balancing
- Reduce Memory Traffic
- Concurrent Communication/Work



NUMA Operations (cont. 4)

	cmd	option	arguments	description
Socket Affinity	numactl	-N	{0,1,2,3}	Only execute process on cores of this (these) socket(s).
Memory Policy	numactl	-l	{no argument}	Allocate on current socket.
Memory Policy	numactl	-i	{0,1,2,3}	Allocate round robin (interleave) on these sockets.
Memory Policy	numactl	--preferred=	{0,1,2,3} select only one	Allocate on this socket; fallback to any other if full .
Memory Policy	numactl	-m	{0,1,2,3}	Only allocate on this (these) socket(s).
Core Affinity	numactl	-C	{0,1,2,3, 4,5,6,7, 8,9,10,11, 12,13,14,15}	Only execute process on this (these) Core(s).

Examples launches on Ranger

hybrid: Ibrun numactl -N $socket -m $socket ./a.out

omp-only: numactl –i all ./a.out

OpenMP Runtime Work-Sharing Scheduling

- Syntax:
  … parallel do/for **shedule(runtime)**
- Allows scheduling to be set at runtime. The ***chunk_size*** parameter must not appear here.
- The schedule type is determined from ***OMP_SCHEDULE*** environment variable.

In code:
… parallel do/for schedule (runtime)

{ C-type Shell }

```
% setenv OMP_SCHEDULE "guided, 100“
% setenv OMP_NUM_THREADS 4
% ./a.out
```

{ Bourne-type Shell }

```
% export OMP_SCHEDULE=“guided,100”
% export OMP_NUM_THREADS=4
% ./a.out
```

OpenMP Orphaned Work-sharing Constructs

- Work-sharing directives that appear outside the lexical extent of a parallel region are called orphaned work sharing constructs.
- When in the dynamic extent (i.e, within 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

subroutine work(n,a,b,c)
integer omp_get_num_threads
integer n, id, i
real*8, dimension(n):: a,b,c

  id=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);
}

int work(int n, double * a,…){

int id, i;

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

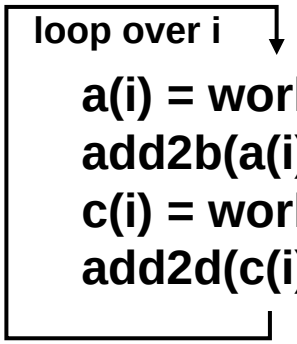
OpenMP if clause

- 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( work_to_do > run_as_serial_limit );
#pragma omp parallel if( work_to_do > run_as_serial_limit );
```

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 guaranteed fairness for entry, but
- Guaranteed forward processing
- Named critical region are independent **see below:



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

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

```
#pragma omp parallel  for
{   for( i=0; i<n; i++)
      a[i] = worka(i);
!$critical (B)
      add2b(&a[i],&b);
!$end critical
      c[i] = workc(i);
!$critical (D)
      add2d(&c[i],&d);
!$end critical
}
```

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)

OpenMP Ordered clause

- Can apply to portion of loop – portion instance is executed in “serial loop” order.
  - Ordered appears in clause AND as directive
- Directive may be orphaned.

```
!$OMP parallel do ordered
  do i = 1,n
    a(i) =  work(i)
!$OMP ordered
    print*,a(i)
!$OMP end ordered
  end do
```

```
#pragma omp parallel for ordered
  for(i=0; i<n; i++){
    a[i] =  work(i);
#pragma ordered
    { printf(“%lf\n”,a[i]); }

  }
```

OpenMP Master clause

- Syntax:
  - !$omp master … !$omp end master
  - #pragma omp master { … }
- Executed only by master thread (in || region)
- No implicit barrier; other team members not required to reach; similar to if(id=0)…
- Access to master's copy of threadprivate
- May be more efficient than single (which has a barrier).

OpenMP threadprivate clause

- **Syntax:**

```
  declaration global_var
  !$omp       threadprivate global_var
 #pragma omp threadprivate global_var
```

- **Makes private variables of threads have global file scope (persist across multiple parallel regions).**
- **For variables (C/C++) or common blocks (Fortran)**
- **Cannot change number of threads for parallel regions when using threadprivate directive.**

Unlike private, which is local to parallel region!!!

http://www.llnl.gov/computing/tutorials/openMP/#THREADPRIVATE

OpenMP copyin clause

- Syntax
  … parallel copyin (*list*)
- Used to assign the same value to THREADPRIVATE variables for all threads in the team (F90, can use common blocks).
- A *copyout* is not needed – values persist throughout entire program.

```
integer :: n,m
!$omp threadprivate (n,m)
read(*) n
!$omp   parallel copyin(n),
!$omp&             private(id,nt)
   nt = omp_num_threads()
   id = omp_thread_num()
   m = n/(nthrds-1)
   if(id=(nthrds-1)) m= n+m(1-nthrds)
   call work(id,m)
!$omp end parallel
```

```
int n,m;
#pragma omp threadprivate (n,m)
  scanf("%d",&n);
#pragma omp parallel copyin(n), \
                                private(id,nt)
 {  nt = omp_num_threads();
    id = omp_thread_num();
    m = n/(nthrds-1);
    if(id==(nthrds-1)) m= n+m(1-nthrds);
    work(id,m);
}
```

OpenMP directives & clauses

Directive

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

These don't accept clauses:

master
critical
barrier
atomic
flush
ordered
threadprivate

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

Copied from ??

OpenMP Work-sharing control

- If any thread reaches a worksharing construct, then all team members must reach that construct.
- Remember, static scheduling maps the sequence of threads to the sequence of work-sharing sections in order. (That is, thread 0 is mapped to the first work-sharing region, thread 1 is mapped to the second, etc.)

```
!$omp parallel
…
if( refine_it ) then
!$omp do
   do i=1,n; call work(i); end do
else
   call nowork(id)
endif
!$omp parallel end
```

What's new? -- OpenMP 2.0

- Wallclock timers
- Workshare directive (Fortran)
- Reduction on array variables
- NUM_THREAD clause

OpenMP Wallclock Timers

```
Real*8 :: omp_get_wtime, omp_get_wtick()
```
(Fortran)

```
double omp_get_wtime(), omp_get_wtick();
```
(C)

```
#include <omp.h>
int main(int argc, char *argv[]){
  double t0, t1, dt, res;


  t0=omp_get_wtime();
  system("sleep 3 ");
  t1=omp_get_wtime();

  dt=t1-t0; res=omp_get_wtick();
  printf("Time:%lf Resol:=%lf\n",dt,res);
}
```

```
Time:3.007201 Resol:=0.010000
```

```
program timer
  real*8 omp_get_wtime  , &
         omp_get_wtick ;
  real*8 t0, t1, dt, res;

  t0=omp_get_wtime();
  call system("sleep 3 ");
  t1=omp_get_wtime();

  dt=t1-t0; res=omp_get_wtick();
  print*,"Time:",dt," Resol:",re
```

```
Time: 3.0063  Resol: 0.100E-01
```

Workshare directive

- WORKSHARE directive enables parallelization of Fortran 90 array expressions and FORALL constructs

```
integer, parameter :: n=1000
real*8             :: a(n,n), b(n,n), c(n,n)
!$omp workshare
       a=b+c       !This is Array Syntax
!$omp end workshare
```

- Enclosed code is separated into units of work
- All threads in a team share the work – it is a worksharing construct
- A work unit may be assigned to any thread

Reduction on array variables

- Array variables may now appear in the REDUCTION clause

```
Real*8 :: A(N), B(M,N)
Integer :: i, j
…
!$OMP Parallel Do Reduction(+:A)
do i=1,n
          do j=1,m
            A(i)=A(i)+B(j,i)
          end do
end do
!$OMP End Parallel Do
```

- Exceptions are assumed size and deferred shape arrays
- Variable must be shared in the enclosing context

NUM_THREADS clause

- Use the NUM_THREADS clause to specify the number of threads to execute a parallel region

  Usage:

```
!$OMP PARALLEL NUM_THREADS(scalar integer expression)
   <code block>
!$OMP End PARALLEL
```

  where ***scalar integer expression*** must evaluate to a positive integer

- NUM_THREADS supersedes the number of threads specified by the **OMP_NUM_THREADS** environment variable or that set by the **OMP_SET_NUM_THREADS** function

References

- Some material identical to: http://www.ascc.net/compsrv/sysmgnt/euler/workshop/IBM-Compilers.pdf
- This one is a real tutorial and even has test modules:http://webct.ncsa.uiuc.edu:8900/public/OPENMP/
- The sites
  http://www.llnl.gov/computing/tutorials/openMP/
  http://www.nersc.gov/nusers/help/tutorials/openmp
  have good reference/tutorial pages for OpenMP.