Shared Memory programming with OpenMP

Ashwanth Srinivasan, Dilip Sarkar, Mohamed Iskandarani and CCS HPC Staff

Center for Computational Science
&
MPO Division
Rosenstiel School of Marine and Atmospheric Science
&
Computer Science Department
University of Miami
Miami, Florida
Outline

- Shared Memory Systems
- Introduction to OpenMP
- Parallel Regions
- Worksharing directives
- Synchronization
- Data Sharing Environment
- Additional features
- OpenMP 3.0
In a shared memory system,

- multiple processing units share a single global address space (memory)
- processing units coordinate via shared variables to solve a problem.
- common types of shared memory systems:
  - Multi-Processor Systems - SMP’s and CC-NUMA machines
  - Multi-Core Systems - e.g., dual core Intel and AMD processor based computers
  - Multi-Threaded Systems - e.g., intel hyperthreading, IBM SMT etc
  - gpu cards, cell processors etc.
Shared Memory Systems

True shared memory

Distributed shared memory

Interconnect

Memory

Interconnect

Memory
SMPs as building blocks for large clusters

Most large systems are clustered systems of Shared memory machines

- Ares is a 16 node cluster. Each node has 16 cpus sharing 32 GB of memory
Multicore Systems

Dual core desktops, laptops etc.

HPC Workshop: Shared Memory programming with OpenMP, July 15th, 2009 – p.6/56
Multithreaded Systems

Hyperthreading, SMT etc.

[Diagram showing physical and logical processors with resource allocation and throughput]
In a shared memory architecture,

- the threaded programming model is commonly used
- threads are lightweight processes, that exist within a single operating system process
- the application can transparently access any memory location
- a single global address space is shared between threads
- communication and data exchange between the threads takes place through shared memory.
- pthreads, java multi-threading, OpenMP etc. can all be used to write multi-threaded programs.
What is OpenMP?

- Open specifications for Multi Processing (OpenMP)
- is an API to produce multi-threaded code for shared memory machines
- consists of 3 components
  - compiler directives
  - runtime libraries
  - environment variables
- Latest Version: OpenMP 3.0 released summer 2008
Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism is added incrementally: i.e. the sequential program evolves into a parallel program.
OpenMP Usage

- user inserts directives into the Fortran and C/C++ source codes
- user compiles with OpenMP flag enabled
- compiler produces threaded code to run on multiple cores
- behaviour controlled via environment variables.
OpenMP Syntax

Some syntax details to get us started

- most of the constructs in OpenMP are compiler directives or pragmas.
- for C and C++, the pragmas take the form:
  
  ```
  #pragma omp construct [clause]
  ```
- For Fortran, the directives take one of the forms:
  
  ```
  C$OMP construct [clause]
  !$OMP construct [clause]
  !$OMP construct [clause]
  ```
- Since the constructs are directives, an OpenMP program can be compiled by compilers that don't support OpenMP
- OpenMP is essentially the same in both Fortran and C/C++
PROGRAM HELLO_WORLD
   INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM

C Fork a team of threads giving them their own copies of variables
!$OMP PARALLEL PRIVATE(TID)

C Obtain and print thread id
   TID = OMP_GET_THREAD_NUM()
   PRINT *, 'Hello World from thread = ', TID

C Only master thread does this
   IF (TID .EQ. 0) THEN
      NTHREADS = OMP_GET_NUM_THREADS()
      PRINT *, 'Number of threads = ', NTHREADS
   END IF

C All threads join master thread and disband
!$OMP END PARALLEL
OpenMP: C Example

```c
#include <omp.h>
#include <stdio.h>
main () {
    int nthreads, tid;

    /* Fork a team of threads giving them their own copies of variables */
    #pragma omp parallel private(tid)
    {
        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    } /* All threads join master thread and terminate */
```
Example OpenMP Run

[ashwanth@kronos ~]$ icc -openmp hello.c
hello.c(9): (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED.

[ashwanth@kronos ~]$ export OMP_NUM_THREADS=2

[ashwanth@kronos ~]$ ./a.out
Hello World from thread = 0
Number of threads = 2
Hello World from thread = 1

[ashwanth@kronos ~]$ export OMP_NUM_THREADS=4
[ashwanth@kronos ~]$ ./a.out
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 3
Hello World from thread = 2
Hello World from thread = 1
[ashwanth@kronos ~]$
OpenMP constructs fall into 5 categories:

- Parallel Regions
- Work Sharing
- Data Environment
- Synchronization
- Runtime functions and environment variables
Parallel regions

- create threads in OpenMP with the omp parallel directive.
- code block within a parallel region is executed by all threads
- implied barrier at the end of the region
- syntax:

  **Fortran:**
  ```fortran
  !$OMP PARALLEL [CLAUSE...]

  code block

  !$OMP END PARALLEL
  ```

  **C/C++:**
  ```c
  #pragma omp parallel [clause ...]
  {
    block
  }
  ```
example:

```c
    call sub1()
 !$OMP PARALLEL
    call sub2()
 !$OMP END PARALLEL
    call sub3()
```
Parallel Regions Example

- log on to kronos
- `cp -r /nethome/ashwanth/wkshp02`
- `cd to wkshp02`
- test your setup by compiling the `omp_hello_world.f`
- `ifort -openmp -o hello_world hello_world.f`
- open up and examine the `compress.f` program
- this program gzips and unzips files in the current directory
- use openMP directives to parallelize this program
- solutions are in `parallel_compress.f`
Clauses

Specify additional information in the parallel region directive through clauses:

Fortran: !$OMP PARALLEL \[clauses\]
C/C++: #pragma omp parallel \[clauses\]
\[clauses\]

if (scalar_expression)
private (list)
shared (list)
default (shared | none)
firstprivate (list)
reduction (operator: list)
copyin (list)
num_threads (integer-expression)

Clauses are comma or space separated in Fortran, space separated in C/C++. 
How many threads?

- Dynamic mode:
  - The number of threads used in a parallel region can vary from one parallel region to another.
  - Setting the number of threads only sets the maximum number of threads - you could get less.
  - Set OMP_DYNAMIC environment variable to FALSE to turn off dynamic threads.

- Static mode:
  - The number of threads is fixed and controlled by the programmer.
How many threads?

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  - Evaluation of the IF clause
  - Setting of the NUM_THREADS clause
  - Use of the omp_set_num_threads() library function
  - Setting of the OMP_NUM_THREADS environment variable
  - Implementation default - usually the number of CPUs on a node, though it could be dynamic.

- Threads are numbered from 0 (master thread) to N-1
Work Sharing Constructs

- Loops are the most common source of parallelism in most codes.
- Parallel loop directives are therefore very important!
- do/for work sharing construct splits up loop iterations among the threads in a team

   Fortran: !$OMP DO [clause ...]
            do loop
            !$OMP END DO [ NOWAIT ]

   c/c++:  #pragma omp for [<clause> [<clause>] ... ]
          <for-loop statement>

- By default, there is a barrier at the end of the omp do/for. Use the nowait clause to turn off the barrier
Parallel do/for

Again, this construct is so common that there is a shorthand form which combines parallel region and DO/FOR directives:

Fortran:

    !$OMP PARALLEL DO [clauses]
    do loop
    [ !$OMP END PARALLEL DO ]

C/C++:

    #pragma omp parallel for [clauses]
    for loop
Parallel do Fortran Example

! Some initializations
N=1000
DO I = 1, N
A(I) = I * 1.0
B(I) = A(I)
ENDDO
CHUNK = 10

!$OMP PARALLEL DO
!$OMP& SHARED(A,B,C,CHUNK) PRIVATE(I)
!$OMP& SCHEDULE(STATIC,CHUNK)

DO I = 1, N
C(I) = A(I) + B(I)
ENDDO

!$OMP END PARALLEL DO
Work Sharing Example

- cd to wkshp02
- open up and examine the matmul.c program
- this program multiplies two matrices
- use openMP directives to parallelize this program
- solutions are in omp_matmul.c
The schedule clause effects 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

```sh
export OMP_SCHEDULE=guided,4
```

HPC Workshop: Shared Memory programming with OpenMP, July 15th, 2009 – p.27/56
Which Schedule to choose?

- STATIC best for load balanced loops - least overhead.
- STATIC,n good for loops with mild or smooth load imbalance, but can induce false sharing (see later).
- DYNAMIC useful if iterations have widely varying loads, but ruins data locality.
- GUIDED often less expensive than DYNAMIC, but beware of loops where the first iterations are the most expensive!
- Use RUNTIME for convenient experimentation.
How can you tell if a loop is parallel?

Useful test: if the loop gives the same answers if it is run in reverse order, then it is almost certainly parallel. Jumps out of the loop are not permitted.

A non parallel loop example:

```fortran
    do i=2,n
        a(i)=2*a(i-1)
    end do
```

A parallel loop example:

```fortran
    do i=2,n
        b(i) = (a(i)-a(i-1))*0.5
    end do
```
Work Sharing Construct - Parallel Sections

- Allows separate blocks of code to be executed in parallel (e.g. several independent subroutines)
- Not scalable: the source code determines the amount of parallelism available.
- not used as often as Do/for, except with nested parallelism

```c
 !$OMP SECTIONS [clauses]
  [ !$OMP SECTION ]
  block

  [ !$OMP SECTION
    block ]
  . . .

  !$OMP END SECTIONS
```
Fortran Parallel Sections Example

!$OMP PARALLEL SHARED(A,B,C,D), PRIVATE(I)

!$OMP SECTIONS

!$OMP SECTION
  DO I = 1, N
    C(I) = A(I) + B(I)
  ENDDO

!$OMP SECTION
  DO I = 1, N
    D(I) = A(I) * B(I)
  ENDDO

!$OMP END SECTIONS NOWAIT

!$OMP END PARALLEL
PI Calculation by numerical Integration

\[ \pi = \int_{0}^{1} \frac{4.0 \, dx}{1 + x^2} \]

We can approximate this integral using Simpson’s rule

- divide the domain into n partitions
- evaluate the function at each partition
- multiply the function evaluation times the width of the function to obtain differential area
- add the areas together and output the result
PI Example

- cd to wkshp02
- open up and examine the pi.c or the pi.f program
- use OpenMP directives to parallelize this program
- possible solutions are in omp_pi.c and omp_pi.f
- run these programs and check the answer
- what is wrong?
OpenMP provides the following constructs to support synchronization:

- atomic
- critical section
- barrier
- flush
- ordered
- single ==> really a work sharing construct
- master ==> again a work sharing construct
Synchronization - Critical Directive

Critical Sections: Only one thread at a time can enter a critical section. It is illegal to branch into or out of a CRITICAL block.

```fortran
PROGRAM CRITICAL

INTEGER X
X = 0

!$OMP PARALLEL SHARED(X)

!$OMP CRITICAL
X = X + 1
!$OMP END CRITICAL

!$OMP END PARALLEL

END
```
PI Example - OMP Critical

- cd to wkshp02
- open up and examine the pi.c or the pi.f program
- add the critical directive to these programs
- run these programs and check the answer
Synchronization - Atomic

Atomic is a special case of a critical section that can be used for certain simple statements. It applies only to a single statement update of a memory location (the update of X in the following example):

```c
!$OMP PARALLEL PRIVATE(B)
    B = init(I)
!$OMP ATOMIC
    X = X + B
!$OMP END PARALLEL
```
Synchronization - Barrier

- Each thread waits until all threads arrive
- Note that there is an implicit barrier at the end of DO/FOR, SECTIONS and SINGLE directives.
- Either all threads or none must encounter the barrier: otherwise DEADLOCK!!
- costly in terms of performance

```c
!$OMP PARALLEL PRIVATE(I,MYID,NEIGHB)
    myid = omp_get_thread_num()
    neighb = myid - 1

    if (myid.eq.0) neighb = omp_get_num_threads()-1
    ...
    a(myid) = a(myid)*3.5

!$OMP BARRIER

b(myid) = a(neighb) + c
...!

!$OMP END PARALLEL
```
Ordered Directive

- Can specify code within a loop which must be done in the order it would be done if executed sequentially.
- Can only appear inside a DO/FOR directive which has the ORDERED clause specified

```
!$OMP PARALLEL DO ORDERED
  do j=1,n
    . . .
  !$OMP ORDERED
    write(*,*) j, count(j)
  !$OMP END ORDERED
  . . .
  end do
!$OMP END PARALLEL DO
```
Flush Directive

- The FLUSH directive ensures that a variable is written to/read from main memory.
- A FLUSH directive is implied by a BARRIER, at entry and exit to CRITICAL and ORDERED sections, and at the end of PARALLEL, DO/FOR, SECTIONS and SINGLE directives (except when a NOWAIT clause is present).

```c
!$OMP PARALLEL PRIVATE(MYID,I, NEIGHB)
  . . .
  do j = 1, niter
    do i = lb(myid), ub(myid)
      a(i) = (a(i-1) + a(i))*0.5
    end do
    ndone (myid) = ndone (myid) + 1
  !$OMP FLUSH (NDONE)
  do while (ndone(neighb).lt. ndone(myid))
  !$OMP FLUSH (NDONE)
  end do
```

HPC Workshop: Shared Memory programming with OpenMP, July 15th, 2009 – p.40/56
The master construct denotes a structured block that is only executed by the master thread.

The other threads just skip it no barrier and flush

```c
#pragma omp parallel private (tmp)
{
    do_many_things();
}

#pragma omp master

    { exchange_boundaries(); } 

#pragma barrier
    do_many_other_things();
```
Synchronization - Single Construct

- The single construct denotes a block of code that is executed by only one thread.
- A barrier and a flush are implied at the end of the single block.

```c
#pragma omp parallel private (tmp)
{
    do_many_things();

#pragma omp single
{
    exchange_boundaries();
}
```
Data Environment

!$OMP PARALLEL SHARED(A,B,C,D), PRIVATE(I)

!$OMP SECTIONS

!$OMP SECTION
  DO I = 1, N
    C(I) = A(I) + B(I)
  ENDDO

!$OMP SECTION
  DO I = 1, N
    D(I) = A(I) * B(I)
  ENDDO

!$OMP END SECTIONS NOWAIT

!$OMP END PARALLEL
Data Environment

- Global variables are SHARED among threads
- Fortran: COMMON blocks, SAVE variables, MODULE variables
- C: File scope variables, static

- But not everything is shared...
  - Stack variables in sub-programs called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.
Inside a parallel region, variables can be either shared (all threads see same copy) or private (each thread has its own copy). Specified by shared, private and default clauses

**Fortran:**
- `SHARED(list)`
- `PRIVATE(list)`
- `DEFAULT(SHARED|PRIVATE|NONE)`
- `FIRSTPRIVATE(list)`
- `LASTPRIVATE(list)`

**C/C++:**
- `shared(list)`
- `private(list)`
- `default(shared|none)`
- `firstprivate(list)`
- `lastprivate(list)`

- the value of a private inside a parallel loop can be initialized by `FIRSTPRIVATE`
- the value of a private inside a parallel loop can be transmitted to a global value outside the loop with: `LASTPRIVATE`
Private and Shared variables cont

Example: each thread initializes its own column of a shared array:

```c
!$OMP PARALLEL DEFAULT(NONE),PRIVATE(I,MYID),
!$OMP& SHARED(A,N)
    myid = omp_get_thread_num() + 1
    do i = 1,n
        a(i,myid) = 1.0
    end do
!$OMP END PARALLEL
```
Private and Shared variables cont

Private variables are uninitialized at the start of the parallel region. If we wish to initialize them, we use the FIRSTPRIVATE clause:

Example:

```c
x = 20.0;
y=0.0;
...
```

```c
#pragma omp parallel firstprivate(x), lastprivate(y) private(i,myid)
{
    myid = omp_get_thread_num();
    for (i=0; i<n; i++){
        x += c[myid][i];
        y=x;
    }
}
```

Private copy is not storage associated with the original. Use lastprivate to pass to global variable
How do we decide which variables should be shared and which private?

- Most variables are shared
- Loop indices are private
- Loop temporaries are private
- Read-only variables - shared
- Main arrays - shared
- Write-before-read scalars - usually private
Data Sharing - Threadprivate

Threadprivate makes global data private to a thread

- Fortran: COMMON blocks
- C: File scope and static variables
- 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 by using DATA statements.
A threadprivate example: Consider two different routines called within a parallel region

```fortran
subroutine HEE
parameter (N=1000)
common/buf/A(N),B(N)
C$OMP THREADPRIVATE(/buf/)
do i=1, N
  B(i) = const* A(i)
end do
return
end

subroutine HAA
parameter (N=1000)
common/buf/A(N),B(N)
C$OMP THREADPRIVATE(/buf/)
do i=1, N
  A(i) = sqrt(B(i))
end do
return
end
```
Another clause that effects the way variables are shared:
- specified as reduction (op : list)

The variables in list must be shared in the enclosing parallel region.

Inside a parallel or a worksharing construct:
- A local copy of each list variable is made and initialized depending on the op (e.g. 0 for +)
- pairwise op is updated on the local value
- Local copies are reduced into a single global copy at the end of the construct

Variables in the list must be named scalar variables

Reduction operations may not be associative for real numbers.
Some initializations

N = 1000
DO I = 1, N
   A(I) = I * 1.0
   B(I) = I * 2.0
ENDDO
RESULT = 0.0
CHUNK = 10

!$OMP PARALLEL DO
!$OMP& DEFAULT(SHARED) PRIVATE(I)
!$OMP& SCHEDULE(STATIC, CHUNK)
!$OMP& REDUCTION(:RESULT)

DO I = 1, N
   RESULT = RESULT + (A(I) * B(I))
ENDDO

!$OMP END PARALLEL DO NOWAIT
PI Example - With Reduction

- cd to wkshp02
- rewrite the pi.c or the pi.f program to use reduction
- solution is in pi_reduction.[c,f]
- run these programs and check the answer
Low level routines

- provides more flexibility than CRITICAL and ATOMIC directions.
- A lock is a special variable that may be set by a thread.
- No other thread may set the lock until the thread which set the lock has unset it.
- Setting a lock can either be blocking or non-blocking.
- A lock must be initialized before it is used, and may be destroyed when it is not longer required.
- Lock variables should not be used for any other purpose.
Utility Functions

Often useful to find out number of threads being used

Fortran:

    INTEGER FUNCTION OMP_GET_NUM_THREADS()

C/C++:

    #include <omp.h>

    int omp_get_num_threads(void);

Important note: returns 1 if called outside parallel region!
more Utility functions

Also useful to find out number of the executing thread

Fortran:

```
INTEGER FUNCTION OMP_GET_THREAD_NUM()
```

C/C++:

```
#include <omp.h>
int omp_get_thread_num(void)
```

Takes values between 0 and OMP_GET_NUM_THREADS() - 1