Introduction to OpenMP

Modified by Prof. Carlos Bazilio
Fluminense Federal University, Brazil
Outline

1. What is OpenMP
2. Getting started with OpenMP
3. Exploiting loop-level parallelism
4. Beyond loop-level parallelism: Parallel Regions
5. Synchronization
6. Performance
1. What is OpenMP?
OpenMP

• OpenMP (*open specifications for Multi Processing*) is an
  – Application Programming Interface (API),
  – that provides a portable model for developers of
    shared memory parallel applications.

• Available on most single address space machines.
  – shared memory systems, including cc-NUMA, multi-core systems.
What is a Shared Memory Architecture?

FEATURES:
1. All CPUs share memory - One address space.
2. CPUs access memory using the interconnection network.
Hamilton

- hamilton.ichec.ie is a shared memory machine.
- 32 x Itanium 1.5Ghz processors.
- 256GB memory.
What OpenMP is not?

OpenMP is not a new computer language;
– Rather it works in conjunction with existing base languages such as standard Fortran or C/C++.

OpenMP is not meant for distributed memory parallel systems (by itself).
What OpenMP is?

• Comprised of three primary API components:
  – Compiler directives
  – Runtime library routines
  – Environment variables

• OpenMPs Strong points:
  – Incremental parallelization
  – Portability
  – Ease of use
  – Standardised
Program Flow

- Thread-based Parallelism
- Explicit Parallelism
- Fork-Join Model
- Compiler Directive Based
- Dynamic Threads

*Source: http://www.llnl.gov/computing/tutorials/openMP/#ProgrammingModel*
Writing a Parallel Application

- Decompose the problem into tasks
  - Ideally, these tasks can be worked on independently of the others.

- Map tasks onto “threads of execution” (processors)

- Threads have shared and local data
  - Shared: used by more than one thread
  - Local: private to each thread

- Write source code using a parallel programming environment.

- Choices may depend on (among many things)
  - The hardware platform to be run on
  - The level of performance needed
  - The nature of the problem
The Big Merge

In future there will be just one specification for all languages
What Is OpenMP 2.5?

Merges Fortran and C/C++ specifications

- Text and terms same wherever possible
  - Helped resolve inconsistencies.

- Reorganized with new material, but no new features
  - Internal control variables.
  - Memory model.

- More accurate specs for compiler writers.

- Language specific sections are identified graphically.
Life is Short, Remember?

Arguably, OpenMP is easier to use than MPI!
OpenMP vs. POSIX Threads

- POSIX threads is the other widely used shared programming API.
- Fairly widely available, usually quite simple to implement on top of OS kernel threads.
- Lower level of abstraction than OpenMP:
  - library routines only, no directives.
  - more flexible, but harder to implement and maintain.
  - OpenMP can be implemented on top of POSIX threads.
- Not much difference in availability:
  - not that many OpenMP C++ implementations.
  - no standard Fortran interface for POSIX threads.
2. Getting Started with OpenMP.
• A thread is a (lightweight) process - an instance of a program + its data.
• Each thread can follow its own flow of control through a program.
• Threads can share data with other threads, but also have private data.
• Threads communicate with each other via the shared data.
• A thread team is a set of threads which co-operate on a task.
• The master thread is responsible for co-ordinating the team.
Thread 1

Thread 2

Thread 3

PC  Private data

PC  Private data

PC  Private data

Shared data
Getting Started with OpenMP

• OpenMP’s constructs fall into 5 categories:
  – Parallel Regions.
  – Work sharing.
  – Data Environment.
  – Synchronization.
  – Runtime functions/environment variables.

• OpenMP is essentially the same for both Fortran and C/C++. 
Directives Format

• A directive is a special line of source code with meaning only to certain compilers.

• A directive is distinguished by a sentinel at the start of the line.

• OpenMP sentinels are:
  – Fortran: !$OMP (or C$OMP or *$OMP)
  – C/C++: #pragma omp
Fortran Directives Format

Fortran 90+ implementations provide a *OpenMP module* for inclusion in source code.

```fortran
use omp_lib
```

Provides interfaces to OpenMP runtime library routines e.g.

```fortran
omp_get_thread_num()
```

In C/C++ use:

```c
<omp.h>
```
• The *parallel region* is the basic parallel construct in OpenMP.
• A parallel region defines a section of a program.
• Program begins execution with the initial thread (sometimes called the master thread).
• When the first parallel region is encountered, the master thread creates a team of threads (fork/join model).
• *Every* thread executes the statements which are inside the parallel region
• At the end of the parallel region, the master thread waits for the other threads to finish (*synchronises*), and continues executing the next statements
OpenMP: Parallel Regions

- For example, to create a 4-thread parallel region:
  - each thread calls `foo(ID,A)` for \( ID = 0 \) to \( 3 \)

```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    foo(ID, A);
}
printf("All Done\n");
```

Each thread redundantly executes the code within the structured block.
OpenMP Parallel Regions

```c
double A[1000];
omp_set_num_threads(4);
foo(0,A);
foo(1,A);
foo(2,A);
foo(3,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. barrier).
PROGRAM FRED

!$OMP PARALLEL

!$OMP END PARALLEL

!$OMP PARALLEL

!$OMP END PARALLEL
How many threads?

The number of threads in a parallel region is determined by the following factors:

1. Use of NUM THREADS clause
2. Use of theomp_set_num_threads() library function.
3. Setting of the OMP_NUM_THREADS environment variable.
4. The implementation default.

Threads are numbered from 0 (master thread) to N-1.
OpenMP runtime library

**OMP_GET_NUM_THREADS()** – returns the current number of threads.

**OMP_GET_THREAD_NUM()** - returns the id of this thread.

**OMP_SET_NUM_THREADS(n)** – set the desired number of threads.

**OMP_IN_PARALLEL()** – returns .true. if inside parallel region.

**OMP_GET_MAX_THREADS()** - returns the number of possible threads.

etc...
Note that updates to shared variables:

\[(\text{e.g. } a = a + 1)\]

are *not* atomic!

If two threads try to do this at the same time, one of the updates may get overwritten.
Thread 1

Program

load a
add a l
store a

Private data

11

Shared data

11

Thread 2

load a
add a l
store a

11
Simple Fortran OpenMP Program

program hello

use omp_lib

print *, 'Starting off in the sequential world.'

!$omp parallel
   print *, 'Hello from thread number =', omp_get_thread_num()
!$omp end parallel

print *, 'Back to the sequential world.'

end program hello
Simple C OpenMP Program

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

int main ( ) {
    printf("Starting off in the sequential world.\n");
    #pragma omp parallel
    {
        printf("Hello from thread number %d\n", omp_get_thread_num() );
    }
    printf("Back to the sequential world.\n");
}
```
3. Exploiting Loop Level Parallelism
Exploiting Loop Level Parallelism

Loop level Parallelism: parallelize only loops

- Easy to implement
- Highly readable code
- Less than optimal performance (sometimes)
- Most often used
Parallel Loop Directives

- Fortran do loop directive
  - !$omp do
- C\C++ for loop directive
  - #pragma omp for
- These directives do not create a team of threads but assume there has already been a team forked.
- If not inside a parallel region shortcuts can be used.
  - !$omp parallel do
  - #pragma omp parallel for
Parallel Loop Directives continued

- These are equivalent to a parallel construct followed immediately by a worksharing construct.

\[
\begin{align*}
\texttt{!$omp parallel do} & \quad \text{Same as} \quad \texttt{!$omp parallel} \texttt{!$omp do} \\
\texttt{#pragma omp parallel for} & \quad \text{Same as} \quad \texttt{#pragma omp parallel} \texttt{#pragma omp for}
\end{align*}
\]
How is OpenMP Typically Used?

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

**Split-up this loop between multiple threads**

```c
void main()
{
    double Res[1000];
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

Sequential program

```c
void main()
{
    double Res[1000];
    #pragma omp parallel for
data    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

Parallel program
Work-Sharing Constructs

• Divides the execution of the enclosed code region among the members of the team that encounter it.

• Work-sharing constructs **do not** launch new threads.

• No implied barrier upon entry to a work sharing construct.

• However, there is an implied barrier at the end of the work sharing construct (unless `nowait` is used).
DO / END DO
Fortran syntax for the *parallel do* directive

```fortran
!$omp parallel do [clause [,] [clause...]]
  
do index = first, last [, stride]
  body of the loop
  enddo

[!$omp end parallel do]  optional
```
C/C++ syntax for the \textit{parallel for} directive

\begin{verbatim}
#pragma omp parallel for [clause [,] [clause...]]

for ( index = first; index <= last; index++ ){
    body of the loop
}
\end{verbatim}
A Simple Example - Parallel Loop

!$OMP PARALLEL DO
  do i=1,1234567890
    b(i) = a(i) + c(i)
  enddo
!$OMP END PARALLEL DO  (note - END PARALLEL DO optional)

The first directive specifies that the loop immediately following should be executed in parallel.

The second (optional) directive specifies the end of the parallel section.
OpenMP: Work-Sharing Constructs

- The “for” Work-Sharing construct splits up loop iterations among the threads in a team.

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

By default, there is a barrier at the end of the “omp for”. Use the “nowait” clause to turn off the barrier.
Work Sharing Constructs - 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;
    for(i=istart;I<iend;i++) {a[i]=a[i]+b[i];}
}
```

OpenMP Parallel Region and a work-sharing for construct

```c
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0;I<N;i++) { a[i]=a[i]+b[i];}
```
DO/for Directive Clauses

- SCHEDULE (type [,chunk])
- NO WAIT
The Schedule Clause

- The schedule clause effects how loop iterations are mapped onto threads

\[
schedule(\text{static [,chunk]})
\]
  - Deal-out blocks of iterations of size “chunk” to each thread.

\[
schedule(\text{dynamic[,chunk]})
\]
  - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.

\[
schedule(\text{guided[,chunk]})
\]
  - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.

\[
schedule(\text{runtime})
\]
  - Schedule and chunk size taken from the OMP_SCHEDULE environment variable.
SCHEDULE(STATIC)

• Iterations are divided evenly among threads

• If chunk is specified, divides the work into chunk sized parcels

• If there are N threads, each thread does every N\textsuperscript{th} chunk of work.
SCHEDULE(STATIC)

!$OMP PARALLEL DO &
!$OMP SCHEDULE(STATIC,3)

DO J = 1, 36
   Work (j)
END DO

!$OMP END DO
schedule(dynamic[,chunk])

• Divides the workload into chunk sized parcels.

• As a thread finishes one chunk, it grabs the next available chunk.

• Default value for chunk is one.

• More overhead, but potentially better load balancing.
schedule(dynamic[,chunk])

!$OMP PARALLEL DO & !
$OMPSCHEDULE(DYNAMIC,1)

DO J = 1, 36
    Work (j)
END DO

!$OMP END DO
schedule(guided[,chunk])

The iterations of the loop are divided into chunks such that the size of each successive chunk is exponentially decreasing.

- `chunk` specifies the size of the smallest chunk size, except possibly the last.

- If you do not specify a value for `chunk`, the default value is 1.
schedule(guided[,chunk])

!$OMP PARALLEL DO &
!$OMP SCHEDULE(GUIDED,1)

DO J = 1, 36
   Work (j)
END DO
!$OMP END DO
No Wait Clauses

- No wait: if specified then threads do not synchronise at the end of the parallel loop.

- For Fortran, the END DO directive is optional with NO WAIT being the default.

- Note that the nowait clause is incompatible with a simple parallel region meaning that using the composite directives will not allow you to use the nowait clause.
Timing with OpenMP

**OMP_GET_WTIME**
- Provides a portable wall clock timing routine
- Returns a double-precision floating point value equal to the number of elapsed seconds since some point in the past.

*Fortran*

```fortran
DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
```

*C/C++*

```c
#include <omp.h>
double omp_get_wtime(void)
```
OMP_GET_WTIME Fortran Example

program time

    use omp_lib
    implicit none
    double precision :: start, finish

    start = omp_get_wtime()
    CALL sleep(10)
    finish = omp_get_wtime()

    print *, 'Start: ', start
    print *, 'Finish: ', finish
    print *, 'Time: ', finish - start

end program time
#include "omp.h"
#include <stdio.h>

int main() {
    double start = omp_get_wtime( );
    sleep(10);
    double end = omp_get_wtime( );

    printf("start: % .16g \n end: % .16g \n Time: % .16g \n", start, end, end - start);

    return 0;
}
Practical Session

1) Write a simple vector-add program for which:

- Arrays A, B, C and variable N will be shared by all threads.
- The iterations of the loop will be distributed dynamically in CHUNK sized pieces.
- Threads will not synchronize upon completing their individual pieces of work (NOWAIT).

- Add calls to `omp_get_wtime` before and after the loop to get timing information. Get timing information for different numbers of processors.

- Experiment with modifying the chunk size, array size and using a static distribution. See what effect this has on the speed of the program. Create a simple plot with number of processors on x-axis and time on y-axis. Plot graphs in different colours for dynamic and static scheduling and different CHUNK and array sizes.

- Rewrite the same code with a PARALLEL REGION only
4. Beyond Loop Level Parallelism
Data Dependencies

• In order for a loop to parallelize, the work done in one loop iteration cannot depend on the work done in any other iteration.

• The order of the loop iterations must be irrelevant -> data independent

How can you check for loop independence?
Data scope

**SHARED** - variable is shared by all processors

**PRIVATE** - each processor has a private copy of a variable

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

```
!$OMP PARALLEL DEFAULT(NONE),PRIVATE(I,MYID),
!$OMP& SHARED(A,N)

  myid = omp_get_thread_num()
  do i = 1,n
    a(i,myid) = 1.0
  end do

!$OMP END PARALLEL
```
Data Environment: Changing Storage Attributes

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

• The value of a private inside a parallel loop can be transmitted to a global value outside the loop with:
  – LASTPRIVATE

• The default status can be modified with:
  – DEFAULT (PRIVATE | SHARED | NONE)

* All data clauses apply to parallel regions and worksharing constructs except “shared” which only applies to parallel regions.
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

- But not everything is shared...
  - Stack variables in sub-programs called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.
• 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
  – Sometimes either is semantically OK, but there may be performance implications in making the choice.

• N.B. can have private arrays as well as scalars
Private Clause

- `private(var)` creates a local copy of `var` for each thread.
  - The value is uninitialized.
  - Private copy is *not* storage associated with the original.

```c
void wrong(){
    int IS = 0;
    #pragma parallel for private(IS)
    for(int J=1;J<1000;J++)
        IS = IS + J;
    printf("%i", IS);
}
```
PRIVATE Data Example

Each processor needs its own private copy of the variable TEMP.

If TEMP were shared, the result would be unpredictable since multiple processors would be writing to the same memory location.

!$OMP PARALLEL DO SHARED(A,B,C,N) PRIVATE(I,TEMP)

do I=1,N
    TEMP = A(I)/B(I)
    C(I) = TEMP + SQRT(TEMP)
enddo

!$OMP END PARALLEL DO
FIRSTPRIVATE / LASTPRIVATE

**FIRSTPRIVATE**: Private copies of a variable are initialized from the original global object. Use can lead to better performing code.

**LASTPRIVATE**: On exiting the parallel region or loop, variable has the value that it would have had in the case of serial execution.

```
A = 2.0; X=0
! On entry, each thread has A equal to 2.0

!$OMP PARALLEL DO FIRSTPRIVATE(A),LASTPRIVATE(X)
  DO I=1,N
    Z(I) = A*X(I) + Y(I)
    X = I
  ENDDO
  c On exit, I is set to N
```
Another Example:

\[
b = 23.0; \\
. . . . .
\]

```
#pragma omp parallel firstprivate(b), private(i,myid)
{
    myid = omp_get_thread_num();
    for (i=0; i<n; i++){
        b += c[myid][i];
    }
    c[myid][n] = b;
}
```
OpenMP: 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 “+”)
  
  - pair wise “op” is updated on the local value
  
  - Local copies are reduced into a single global copy at the end of the construct.
OpenMP: A Reduction Example

#include <omp.h>
define NUM_THREADS 2
void main ()
{
    int i;
    double ZZ, func(), sum=0.0;

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for reduction(+:sum) private(ZZ)

    for (i=0; i< 1000; i++){
        ZZ = func(i);
        sum = sum + ZZ;
    }
}
IF CLAUSE

We can make the parallel region directive itself conditional.

- Can be useful if there is not always enough work to make parallelism worthwhile.

Fortran: IF (scalar logical expression)

C/C++: if (scalar expression)

```c
#pragma omp parallel if (tasks > 1000)
{
    while(tasks > 0) donexttask();
}
```
SECTIONS Directive

- A non-iterative work-sharing construct.

- Specifies that the enclosed section(s) of code are to be divided among the threads in the team.

- An implied barrier at the end of a SECTIONS directive, unless the nowait (C/C++) or NOWAIT (Fortran) clause is added.
SECTIONS Directive

\[\text{\$OMP SECTIONS } [\text{\^{\textit{clause}}}[,\text{\^{\textit{clause}}}]\ldots] [\text{\$OMP SECTION}] \text{ block} [\text{\$OMP SECTION } \text{ block}]\ldots\]

\text{\$OMP END SECTIONS [NOWAIT]}

\text{Thread 1} \quad \text{Thread 2} \quad \text{Thread 3}

\text{SECTIONS} \quad \text{END}
SECTIONS Directive

- The Sections work-sharing construct gives a different structured block to each thread.

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

Note: SECTION Not Scalable

By default, there is a barrier at the end of the “omp sections”. Use the “nowait” clause to turn off the barrier.
Example

INTEGER, PARAMETER:: N=12
REAL, DIMENSION(N):: A,B,C,D
INTEGER:: I
REAL:: SUMMED

!$OMP PARALLEL SHARED(A,B,C,D) PRIVATE(I)
  !***** Reading files fort.10, fort.11, fort.12 in parallel
!$OMP SECTIONS
!$OMP SECTION
  READ(10,*) (A(I),I=1,N)
!$OMP SECTION
  READ(11,*) (B(I),I=1,N)
!$OMP SECTION
  READ(12,*) (C(I),I=1,N)
!$OMP END SECTIONS
!$OMP END PARALLEL
Runtime Libraries

- OpenMP standard defines an API for library calls that perform a variety of functions
  - Query the number of threads/processors, set number of threads to use
  - General purpose locking routines
  - Portable all clock timing routines
  - Set execution environment functions: nested parallelism, dynamic adjustment of threads
OpenMP: Library routines

- Lock routines
  - `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`

- Runtime environment routines:
  - Modify/Check the number of threads
    - `omp_set_num_threads()`, `omp_get_num_threads()`,
      `omp_get_thread_num()`, `omp_get_max_threads()`
  - Turn on/off nesting and dynamic mode
    - `omp_set_nested()`, `omp_set_dynamic()`, `omp_get_nested()`,
      `omp_get_dynamic()`
  - Are we in a parallel region?
    - `omp_in_parallel()`
  - How many processors in the system?
    - `omp_num_procs()`
Calculating π Parallel Solutions

- Parallel strategy:
  - break the loop into portions which can be executed by the processors.

- For the task of approximating PI:
  - each processor executes its portion of loop N times.
  - each processor does work without requiring any information from the other processors
  - there are no data dependencies
  - no communications except at end
  - this is known as Embarassingly Parallel (EP)

- Embarassingly parallel
  - Computationally intensive
  - Minimal communication
  - Minimal I/O

Source: Maui Supercomputing Center
Calculating π : Serial Version – Fortran Code

program calc_pi
implicit none
integer n,i
double precision w,x,sum,pi,f,a
double precision start, finish, timef
f(a) = 4.0 / (1.0 + a*a)

n=100000000

start=timef()

w=1.0/n
sum=0.0

do i=1,n
   x = w * (i - 0.5)
   sum = sum + f(x)
end do

pi = w * sum
finish=timef()
print*,"value of pi, time taken:"
print*,pi,finish-start
end program calc_pi
Calculating π : Serial Version – C Code

#include <stdio.h>

#define F(x) (4.0 / (1.0 + x * x ) )

int main(){
    long n , i ;
    double  w,x,sum,pi,f,a;

    n = 100000000;
    w = 1.0/n;
    sum = 0.0;

    for ( i = 1 ; i < n ; i++ ) {
        x = w * (i - 0.5);
        sum = sum + F(x);
    }
    pi = w * sum ;
    printf("Value of pi\n");
    printf("%.16g\n", pi);

    return 0;
}
Practical

• Create a parallel version of the above serial code using OpenMP.

• Add in timing using OpenMP timing functions.

• Graph timings for $n = 100$ million on 1, 2, 4 and 8 cpus.

• Suggest a reason why approximations for pi differ.

• Experiment with static, dynamic and guided scheduling schemes and chunk sizes.
Running the Code

- To run the program on $N$ processors we enter either of the following, depending on the login shell:

  - `export OMP_NUM_THREADS=N ./pi.out`

  Or for csh shell

  - `setenv OMP_NUM_THREADS N ./pi.out`

- On $N$ processors, you should get a speed up close to $N$, and thus an efficiency close to 100%.
Speed-Up

- The graph below illustrates the results we obtained - the timings taken, in milliseconds, to calculate pi using 100 million iterations.
5. Synchronization
Synchronisation in OpenMP

- OpenMP has the following constructs to support synchronization:
  - barrier
  - critical section
  - atomic
  - flush
  - ordered
  - single
  - master
OpenMP: How do Threads Interact?

- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.

- Unintended sharing of data can lead to 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 stored to minimize the need for synchronization.
Barrier

- Fortran - !$OMP BARRIER
- C/C++ - #pragma omp barrier

- This directive synchronises the threads in a team by causing them to wait until all of the other threads have reached this point in the code.
- Implicit barriers exist after work sharing constructs. The nowait clause can be used to prevent this behaviour.
Critical

- Only one thread at a time can enter a critical section.

Example: pushing and popping a task stack

```c
 !$OMP PARALLEL SHARED(STACK),PRIVATE(INEXT,INEW)
 ...  
 !$OMP CRITICAL (STACKPROT)
     inext = getnext(stack)
 !$OMP END CRITICAL (STACKPROT)
     call work(inext, inew)
 !$OMP CRITICAL (STACKPROT)
     if (inew .gt. 0) call putnew(inew, stack)
 !$OMP END CRITICAL (STACKPROT)
 ...  
 !$OMP END PARALLEL
```
Atomic

- Atomic is a special case of a critical section that can be used for certain simple statements

Fortran: !$OMP ATOMIC

statement

where statement must have one of these forms:

\[ x = x \ op \ expr, \quad x = expr \ op \ x, \quad x = \text{intr} \ (x, \ expr) \] or
\[ x = \text{intr} \ (expr, \ x) \]

op is one of \(+, *, -, /, .\ and., .\ or., .\ eqv., or .\ neqv.\)
intr is one of \MAX, \MIN, \ IAND, \ IOR \ or \ IEOR\)
C/C++: `#pragma omp atomic` statement

where `statement` must have one of the forms:

\[ x \text{ binop} = \text{ expr, } x++, \text{ ++}x, x--, \text{ or } --x \]

and `binop` is one of `+, \ast, -, /, \&, ^, <, <<, \text{ or } >>`
Example (compute degree of each vertex in a graph):

```c
#pragma omp parallel for
for (j=0; j<nedges; j++){
  #pragma omp atomic
  degree[edge[j].vertex1]++;
  #pragma omp atomic
  degree[edge[j].vertex2]++;
}
```
Flush Directive

- Identifies a synchronization point at which the implementation must provide a consistent view of memory.
- Thread visible variables are written back to memory at this point.
- Implicit flush is often present unless nowait clause is present.

Fortran

```fortran
 !$OMP FLUSH (list)
```

C/C++

```c
 #pragma omp flush (list) newline
```
• 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).
Ordered Directive

- The code enclosed is executed in the order in which iterations would be executed in a sequential execution of the loop.
- The ORDERED directive can only appear in the context of a DO or PARALLEL DO directive. It is illegal to branch into or out of an ORDERED block.

```omp
$OMP ORDERED
block
$OMP END ORDERED
```
Master directive

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

```c
#pragma omp parallel private (tmp)
{
    do_many_things();
    #pragma omp master
    { exchange_boundaries(); }
    #pragma barrier
    do_many_other_things();
}
```
Single directive

- 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(); }
    do_many_other_things();
}
```
Locks

- Occasionally we may require more flexibility than is provided by 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 initialised before it is used, and may be destroyed when it is not longer required.
- Lock variables should not be used for any other purpose.
Fortran:

```
SUBROUTINE OMP_INIT_LOCK(var)
SUBROUTINE OMP_SET_LOCK(var)
LOGICAL FUNCTION OMP_TEST_LOCK(var)
SUBROUTINE OMP_UNSET_LOCK(var)
SUBROUTINE OMP_DESTROY_LOCK(var)
```

`var` should be an INTEGER of the same size as addresses (e.g. INTEGER*8 on a 64-bit machine)
C/C++:

```c
#include <omp.h>

void omp_init_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

There are also nestable lock routines which allow the same thread to set a lock multiple times before unsetting it the same number of times.
Example:

```fortran
    call omp_init_lock( ilock )
    !$OMP PARALLEL SHARED( ilock )
    ...
    do while ( .not. omp_test_lock( ilock ) )
        call do_something_else()
    end do
    call work()
    call omp_unset_lock( ilock )
    ...
    !$OMP END PARALLEL
```
• As a rough guide, use ATOMIC directives if possible, as these allow most optimisation.

• If this is not possible, use CRITICAL directives. Make sure you use different names wherever possible.

• As a last resort you may need to use the lock routines, but this should be quite a rare occurrence.
Additional Features
Nested Parallelism

• Unlike most previous directive systems, nested parallelism is permitted in OpenMP.

• This is enabled with the **OMP_NESTED** environment variable or the **OMP_SET_NESTED** routine.

• If a PARALLEL directive is encountered within another PARALLEL directive, a new team of threads will be created.

• The new team will contain only one thread unless nested parallelism is enabled.
Example:

```c
 !$OMP PARALLEL
 !$OMP SECTIONS
 !$OMP SECTION
 !$OMP PARALLEL DO
   do i = 1,n
     x(i) = 1.0
   end do
 !$OMP SECTION
 !$OMP PARALLEL DO
   do j = 1,n
     y(j) = 2.0
   end do
 !$OMP END SECTIONS
 !$OMP END PARALLEL
```
Orphaned Directives

- Directives are active in the *dynamic* scope of a parallel region, not just its *lexical* scope.

```c
!$OMP PARALLEL
   call fred()
!$OMP END PARALLEL

subroutine fred()
!$OMP DO
   do i = 1,n
      a(i) = a(i) + 23.5
   end do
return
do
end
```
• This is very useful, as it allows a modular programming style.

• But it can also be rather confusing if the call tree is complicated (what happens if fred is also called from outside a parallel region?)

• There are some extra rules about data scope attributes.
When we call a subroutine from inside a parallel region:

- Variables in the argument list inherit their data scope attribute from the calling subroutine.
- Global variables and COMMON blocks are shared, unless declared THREADPRIVATE (see later).
- **static** local variables in C/C++ and **SAVE** variables in Fortran are shared.
- All other local variables are private.
We can find out if we are in a parallel region or not with the OMP_IN_PARALLEL function:

**Fortran:**

```fortran
LOGICAL FUNCTION OMP_IN_PARALLEL()
```

**C/C++:**

```c
#include <omp.h>

int omp_in_parallel(void);
```
There could be ambiguity about which parallel region directives refer to, so we need some rules.

- **DO/FOR, SECTIONS, SINGLE, MASTER and BARRIER** directives always bind to the nearest enclosing PARALLEL directive.

- **ORDERED** directive binds to nearest enclosing **DO** directive.
Dynamic Parallelism

• It is possible to let the system choose how many threads execute each parallel region, to let it optimise resource allocation.

• The number of threads will be equal to or less than that set by the user, and remains fixed for the duration of each parallel region.

• Can be set by `OMP_SET_DYNAMIC` routine or by the `OMP_DYNAMIC` environment variable.

• Its default value is implementation dependent: if your code relies on using a certain number of threads (not recommended) you should disable dynamic parallelism.
Conditional Compilation

- Allows source lines to be recognised by an OpenMP compiler and ignored (treated as comments) by other compilers.
- In C/C++ this is done in the traditional way with the preprocessor macro \_OPENMP
- In Fortran, in addition to this macro, any line beginning with the sentinels !$, C$ or *$ (latter two only in fixed source form), is conditionally compiled.
- The sentinel is replaced with two spaces.
Example (read value of OMP_NUM_THREADS):

```
nthreads = 1
!$OMP PARALLEL
!$OMP MASTER
!$    nthreads = omp_get_num_threads()
!$OMP END MASTER
!$OMP END PARALLEL
!$OMP END PARALLEL
   print *, "No. of threads = ", nthreads
```
Input/Output

• Should assume that I/O is not thread-safe.

• Need to synchronise multiple threads writing to or reading from the same file.
  – Note that there is no way for multiple threads to have private file positions.

• OK to have multiple threads reading/writing to different files.
Workshare Directive

• A worksharing directive (!) which allows parallelisation of Fortran 90 array operations, WHERE and FORALL constructs.

• Syntax:

  !$OMP WORKSHARE
  block
  !$OMP END WORKSHARE [NOWAIT]
Simple example

```c
REAL A(100,200), B(100,200), C(100,200)
...
!$OMP PARALLEL
!$OMP WORKSHARE
    A=B+C
!$OMP END WORKSHARE
!$OMP END PARALLEL
```

- N.B. No schedule clause: distribution of work units to threads is entirely up to the compiler!
6. Performance
Designing Parallel Programs in OpenMP

1) **Partition**
   In OpenMP, look for any independent operations (loop parallel, task parallel)

2) **Communicate**
   In OpenMP, look for synch points and dependencies

3) **Agglomerate**
   In OpenMP, mark parallel loops and/or parallel sections

4) **Map**
   In OpenMP, implicit or explicit scheduling. Data mapping goes outside the standard
Basic Strategies

- If a parallelized loop does not perform well, check for/consider
  - Parallel start up costs
  - Small loops
  - Load imbalances
  - Many references to shared variables
  - Low cache affinity
  - Unnecessary synchronization
  - Costly remote memory references (in NUMA machines)
  - Try to avoid false sharing
Debugging OpenMP code

• Shared memory parallel programming opens up a range of new programming errors arising from unanticipated conflicts between shared resources.

• **Race conditions**
  – When the outcome of a program depends on the detailed timing of the threads in the team.

• **Deadlock**
  – When threads hang while waiting on a locked resource that will never become available.
Race Condition

C$OMP PARALLEL SECTIONS
   A = B+C
C$OMP SECTION
   B=A+C
C$OMP SECTION
   C=B+A
C$OMP END PARALLEL SECTIONS

The result varies unpredictably based on the detailed order of execution for each section.

Wrong answers produced without warning.
ICHEC
Irish Centre for High-End Computing

PURO
Polo Universitario de Rio das Ostras

ICHEC - Introduction to OpenMP

Deadlock

CALL OMP_INIT_LOCK (LCKA)
C$OMP PARALLEL SECTIONS
C$OMP SECTION
CALL OMP_SET_LOCK(LCKA)
IVAL = DOWORK()
IF (IVAL .EQ. TOL) THEN
    CALL OMP_UNSET_LOCK (LCKA)
ELSE
    CALL ERROR (IVAL)
ENDIF
C$OMP SECTION
CALL OMP_SET_LOCK(LCKA)
CALL USE_B_and_A (RES)
CALL OMP_UNSET_LOCK(LCKA)
C$OMP END SECTIONS

- This shows a race condition and a deadlock.
- If A is locked in the first section and the if statement branches around the unset lock, threads running the other sections deadlock waiting for the lock to be released.
- Make sure you release your locks.

5. Correct Programming 9
Other Dangers

• Are the libraries you are using thread-safe?
  – Standard libraries should be OK
• I/O inside a parallel region can interleave unpredictably.
• Private variables can mask global.
• Understand when shared memory is coherent.
  – When in doubt, use FLUSH
• NOWAIT removes implicit barriers.
Performance considerations

Main obstacles to scaling of OpenMP codes are:

1. Serial code (code outside of loops, also CRITICAL regions).
2. Load imbalance.
3. Directives overhead.
4. False sharing (when several processors update the same cache line).

Number 3 and 4 is the price of using OpenMP as opposed to MPI. The advantage is the absence of communication overheads.
OpenMP directives overhead

![Graph showing overhead of OpenMP directives with varying number of threads.](image-url)
General performance recommendations

• Be aware of the Amdahl’s law
  – Minimize serial code
  – Remove dependencies among iterations

• Balance the load
  – Experiment with using SCHEDULE clause

• Reduce false sharing
  – Use private variables

• Try task level parallelism
General performance recommendations

• Be aware of directives cost
  – Parallelize outer loops
  – Minimize the number of directives
  – Minimize synchronization – minimize the use of BARRIER, CRITICAL, ORDERED
  – Consider using NOWAIT clause of OMP DO when enclosing several loops inside one PARALLEL region.
  – Merge loops to reduce synchronization cost
OpenMP Partners

The OpenMP Architecture Review Board was comprised of the following organizations.

- Compaq (Digital)
- Hewlett-Packard Company
- Intel Corporation
- International Business Machines (IBM)
- Kuck & Associates, Inc. (KAI)
- Silicon Graphics, Inc.
- Sun Microsystems, Inc.
- U.S. Department of Energy ASCI program
OpenMP

Pros:
- Incremental parallelism -- can parallelize existing serial codes one bit at a time.
- Quite simple set of directives.
- Shared data!
- Partitioning operations on arrays is very simple.

Cons:
- Requires proprietary compilers. - no longer true with OpenMP support in gcc.
- Requires shared memory multiprocessors.
- Shared data!
- Having to think about what data is shared and what data is private.
- Cannot handle models like master/slave work allocation (yet).
- Generally not as scalable (more synchronization points).
- Not well-suited for non-trivial data structures like linked lists, trees etc.
Summary

- Use OpenMP directives to help compiler parallelize your code
- Loop level parallelism is the easiest (just use PARALLEL DO)
- Pay attention to variable scope (PRIVATE/SHARED)
- Avoid race conditions – use REDUCTION, CRITICAL, ATOMIC, MASTER etc
Conclusion

- OpenMP is successful in small-to-medium SMP systems.
- Multiple cores/CPUs dominate the future computer architectures; OpenMP would be the major parallel programming language in these architectures.
- Simple: everybody can learn it in 2 weeks.
- Not so simple: Don’t stop learning! keep learning it for better performance.
Questions?
References:

http://www.msi.umn.edu/tutorial/scicomp/general/openMP/content/

www.openmp.org

http://www.osc.edu/hpc/training/openmp/