An Introduction to OpenMP

Alexey A. Romanenko
arom@ccfit.nsu.ru

Based on Sun Microsystems' presentation

What is this section about?

- OpenMP overview
- OpenMP program compilation
- OpenMP execution environment
- Ways to make parallel program with OpenMP
- etc.

Agenda

- The OpenMP programming model
- OpenMP guide tour
- OpenMP overview
  - Clauses
  - Worksharing constructs
  - Synchronization constructs
  - Environment variables
  - Global Data
  - Runtime functions

OpenMP Origins

- In the early 90's, vendors of SMP supplied similar, directive-based, Fortran programming extensions:
  - The user would augment a serial Fortran program with directives specifying which loops were to be parallelized
  - The compiler would be responsible for automatically parallelizing such loops across the SMP processors
- Implementations were all functionally similar, but were diverging
- First attempt at a standard was the draft for ANSI X3H5 in 1994. It was never adopted, largely due to waning interest as distributed memory machines became popular.
- The OpenMP standard specification started in the spring of 1997, taking over where ANSI X3H5 had left off, as newer shared memory machine architectures started to become prevalent.

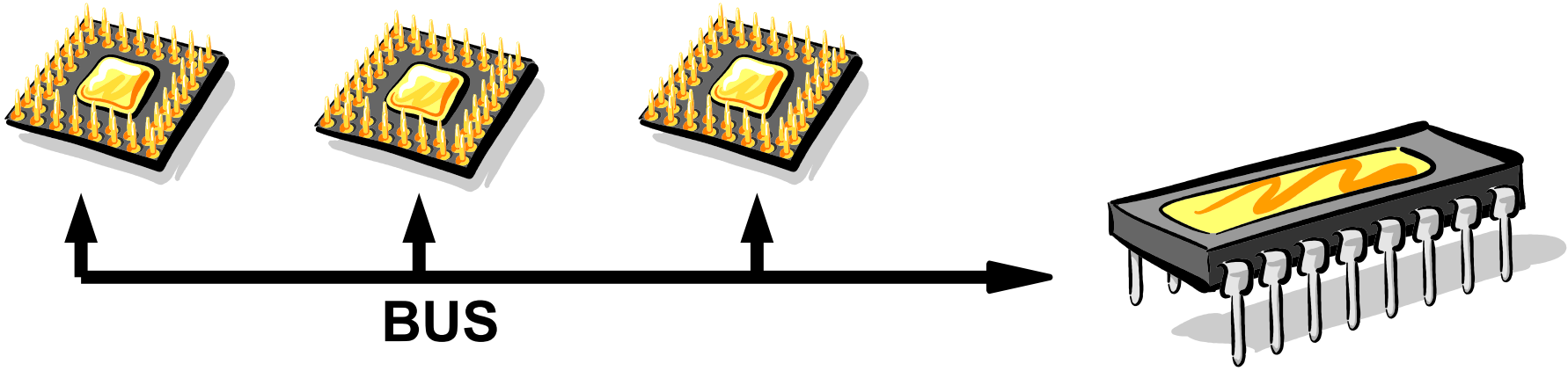
OpenMP today

- The OpenMP programming model is a powerful, yet compact, de-facto standard for Shared Memory Programming
- Current release of the standard: 3.0
- Specifications released May 2008

Goals of OpenMP

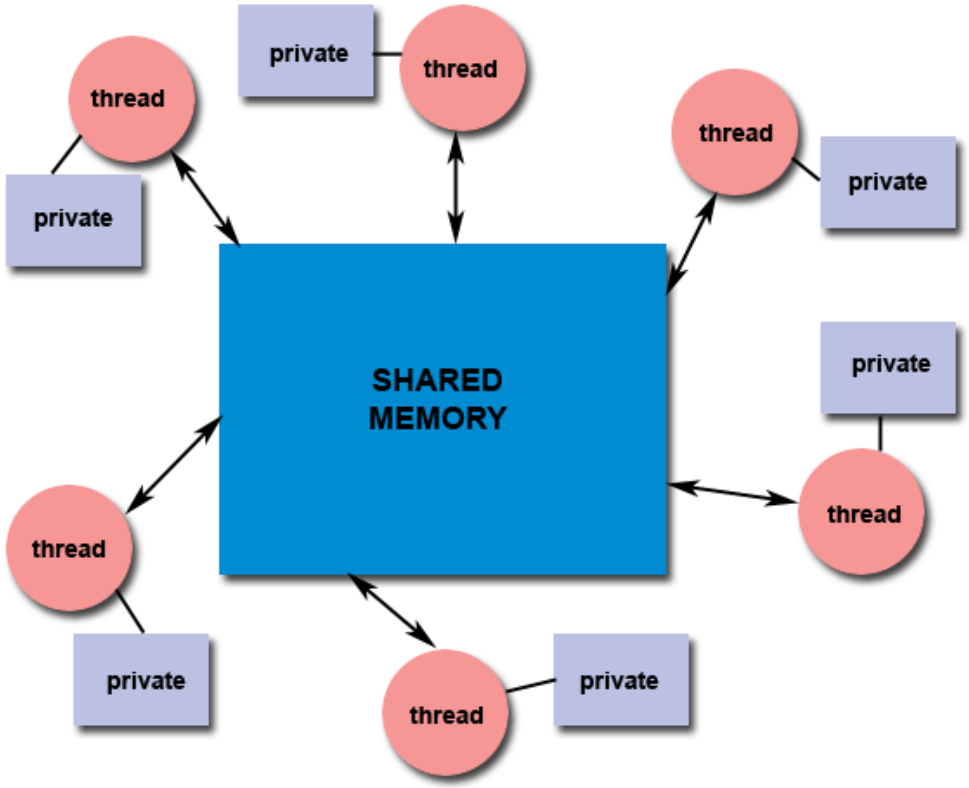
- Provide a standard among a variety of shared memory architectures/platforms
- Establish a simple and limited set of directives for programming shared memory machines. Most of the work is done by just 3 or 4 directives.
- Provide capability to incrementally parallelize a program, unlike message-passing libraries which typically require an all or nothing approach.
- Provide the capability to implement both coarse-grain and fine-grain parallelism
- Support Fortran (77, 90, and 95), C, and C++

The SMP systems



Shared Memory Model

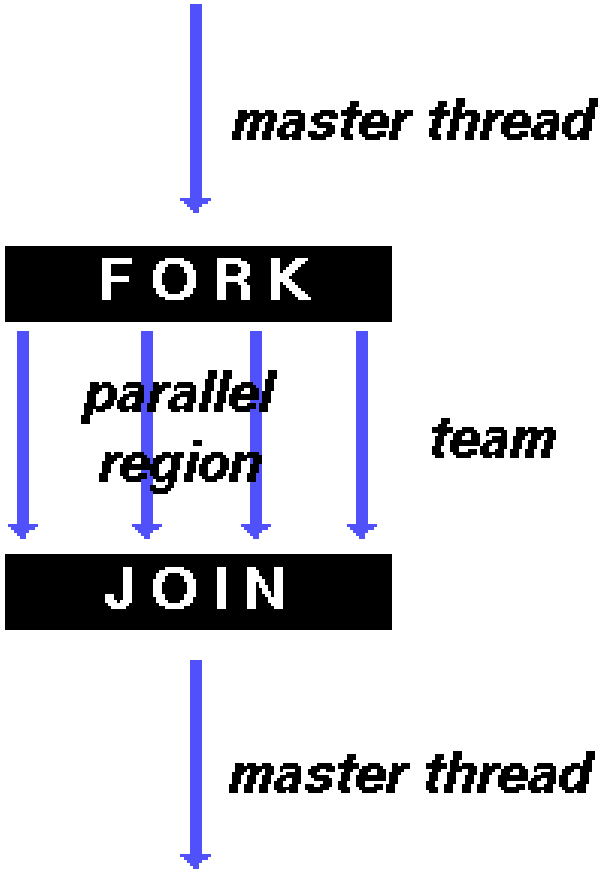
- All threads have access to the same, globally shared, memory
- Data can be shared or private
- Shared data is accessible by all threads
- Private data can be accessed only by the threads that owns
- Data transfer is transparent to the programmer
- Synchronization takes place, but it is mostly implicit



About Data

- In a shared memory parallel program variables have a "label" attached to them:
  - Labelled "Private" ⇨ Visible to one thread only
    - Change made in local data, is not seen by others
    - Example - Local variables in a function that is executed in parallel
  - Labelled "Shared" ⇨ Visible to all threads
    - Change made in global data, is seen by all others
    - Example - Global data

The OpenMP execution model



Example

Sequential code

```
void main(){
  double x[1000];
  for(i=0; i<1000; i++){
    calc_smth(&x[i]);
  }
}
```

Parallel code

```
void main(){
  double x[1000];
#pragma omp parallel for ...
  for(i=0; i<1000; i++){
    calc_smth(&x[i]);
  }
}
```

OpenMP Guided Tour

http://www.openmp.org

When to consider using OpenMP?

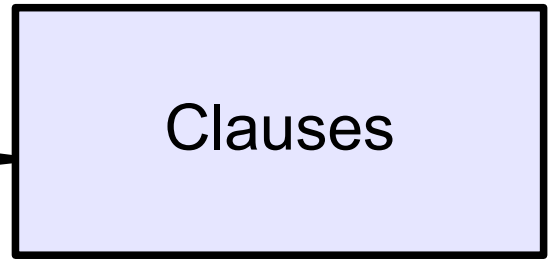
- The compiler may not be able to do the parallelization in the way you like to see it:
- A loop is not parallelized
  - The data dependency analysis is not able to determine whether it is safe to parallelize or not
  - The granularity is not high enough
  - The compiler lacks information to parallelize at the highest possible level
- This is when explicit parallelization through OpenMP directives and functions comes into the picture

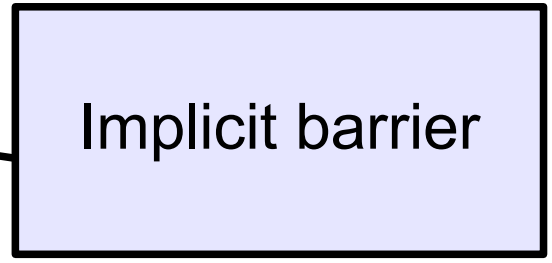
Terminology

- OpenMP Team := Master + Workers
- A Parallel Region is a block of code executed by all threads simultaneously
  - The master thread always has thread 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 guard the parallel region; in case the condition evaluates to "false", the code is executed serially
- 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

A loop parallelized with OpenMP

```
#pragma omp parallel shared(a,b)
{
#pragma omp for private(i)
  for(i=0; i<10000; i++)
    a[i] = a[i] + b[i];
}
```





Components of OpenMP

Directive format

- C: directives are case sensitive
  - Syntax: #pragma omp directive [clause [clause] ...]
- Continuation: use \ in pragma
- Conditional compilation: _OPENMP macro is set

Example

```
#ifdef _OPENMP
printf("Caution: The program was compiled with "
       "OpenMP and can consume all CPU resources "
       " of your PC!\n");
#endif
...
#omp parallel for private(i,j) \
   shared(a,b,c)
{
   for(i=0; i<100; i++)
      for(j=0; j<100; j++)
         a[i] = b[i][j]*c[j];
}
```

Some OpenMP Clauses

About OpenMP clauses

- Many OpenMP directives support clauses
- These clauses are used to specify additional information with the directive
- For example, **private(a)** is a clause to the for directive:
  - **#pragma omp for *private(a)***
- Before we present an overview of all the directives, we discuss several of the OpenMP clauses first
- The specific clause(s) that can be used, depends on the directive

The if/private/shared clauses

- if (scalar expression)
  - Only execute in parallel if expression is true
  - Otherwise, execute serially
- 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

Example

```
#omp parallel for private(i,j) \
    shared(a,b,c) if(M>100)
{
    for(i=0; i<M; i++)
        for(j=0; j<100; j++)
            a[i] = b[i][j]*c[j];
}
```

About storage association

- Private variables are undefined on entry and exit of the parallel region
- The value of the original variable (before the parallel region) is undefined after the parallel region!
- A private variable within a parallel region has no storage association with the same variable outside of the region
- Use the first/last private clause to override this behaviour
- We will illustrate these concepts with an example

The first/last private clauses

- firstprivate (list)
  - All variables in the list are initialized with the value the original object had before entering the parallel construct
- lastprivate (list)
  - The thread that executes the sequentially last iteration or section updates the value of the objects in the list

Example

```
#pragma omp parallel
{
#pragma omp for private(i) lastprivate(k)
  for(i=0; i<10; i++)
      k = i*i;
}
printf("k = %d\n", k); // k == 81
```

Example

```
int myid, a;

a = 10;
#pragma omp parallel default(private) \
                     firstprivate(a)
{
   myid = omp_get_thread_num();
   printf("Thread%d: a = %d\n", myid, a);
   a = myid;
   printf("Thread%d: a = %d\n", myid, a);
}
```

```
Thread1: a = 10
Thread1: a = 1
Thread2: a = 10
Thread0: a = 10
Thread3: a = 10
Thread3: a = 3
Thread2: a = 2
Thread0: a = 0
```

The default clause

- default (none | shared)
- none
  - No implicit defaults
  - Have to scope all variables explicitly
- shared
  - All variables are shared
  - The default in absence of an explicit "default" clause

The reduction clause - example

- Example:

```
#pragma omp parallel
   {
   #pragma for shared(x, sum) private(i)
      for(i=0; i<10000; i++)
         sum = sum + x[i];
   }
```

- Care needs to be taken when updating shared variable SUM
- With the reduction clause, the OpenMP compiler generates code such that a race condition is avoided

The reduction clause

- reduction (operator : list)
  - Reduction variable(s) must be shared variables
  - Note that the value of a reduction variable is undefined from the moment the first thread reaches the clause till the operation has completed
  - The reduction can be hidden in a function call

```
#pragma omp parallel
   {
   #pragma for shared(x) private(i) reduction(+:sum)
       for(i=0; i<10000; i++)
          sum += x[i];
   }

#pragma omp parallel
   {
   #pragma for shared(x) private(i) reduction(min:gmin)
      for(i=0; i<10000; i++)
          gmin = min(gmin, x[i]);
   }
```

The nowait clause

- To minimize synchronization, some OpenMP directives/pragmas support the optional nowait clause
- If present, threads will not synchronize/wait at the end of that particular construct

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

The parallel region

- A parallel region is a block of code executed by multiple threads simultaneously

```
#pragma omp parallel [clause[[,] clause] ...]
{
    "this will be executed in parallel"
} //implied barrier
```

The parallel region - clauses

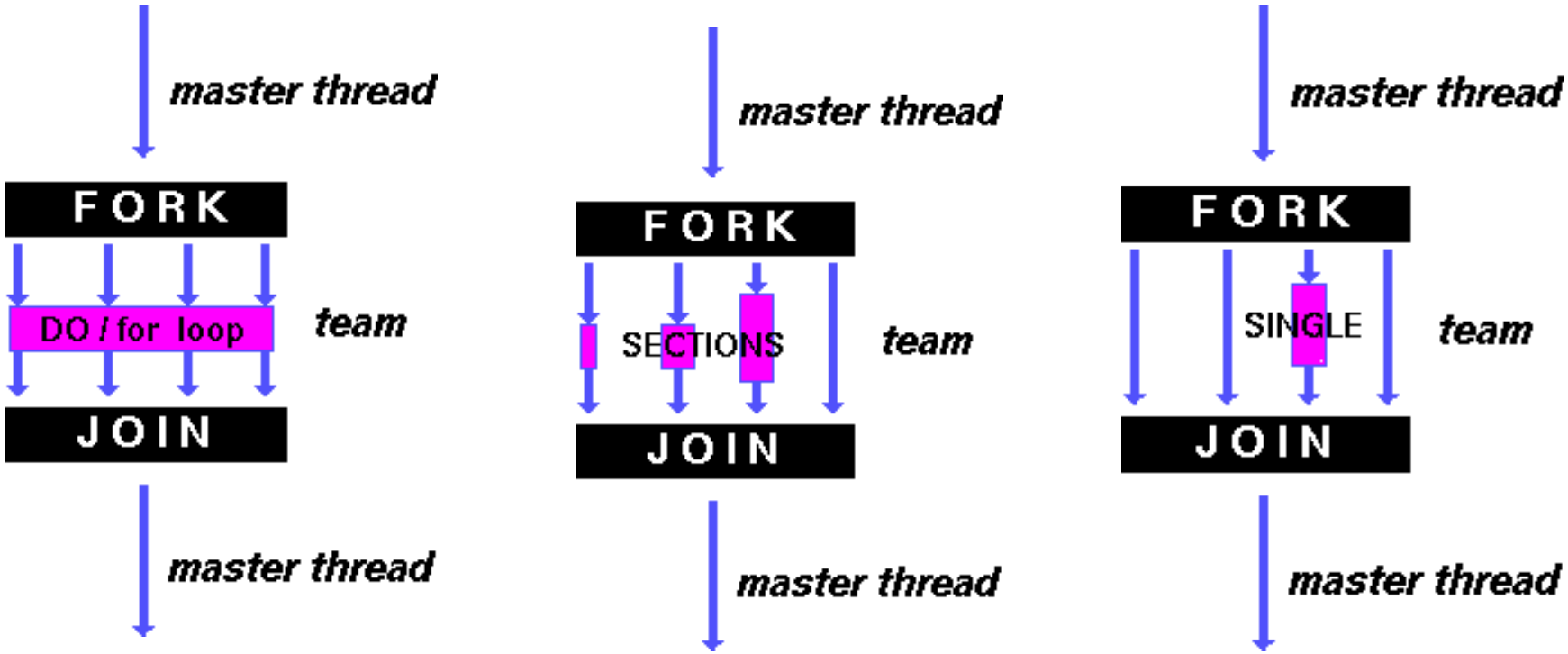
- A parallel region supports the following clauses:
  - if (scalar expression)
  - private (list)
  - shared (list)
  - default (none|shared)
  - reduction (operator: list)
  - copyin (list)
  - firstprivate (list)
  - num_threads (scalar_int_expr)

Worksharing Directives

Work-sharing constructs

- for, section, single
  - The work is distributed over the threads
  - Must be enclosed in a parallel region
  - Must be encountered by all threads in the 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 any new threads

Work-sharing constructs



The omp for directive

- The iterations of the loop are distributed over the threads

```
#pragma omp for [clause[[,] clause] ...]
        <original for-loop>
```

- Clauses supported:
  - private
  - firstprivate
  - lastprivate
  - reduction
  - ordered
  - schedule
  - nowait

Load balancing

- Load balancing is an important aspect of performance
- For regular operations (e.g. a vector addition), load balancing is not an issue
- For less regular workloads, care needs to be taken in distributing the work over the threads
- Examples of irregular workloads:
  - Transposing a matrix
  - Multiplication of triangular matrices
  - Parallel searches in a linked list
- For these irregular situations, the schedule clause supports various iteration scheduling algorithms

The schedule clause

- schedule (static | dynamic | guided [, chunk] | 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 approx. N/P chunks for a loop of length N and P threads

The schedule clause

- dynamic [, chunk]
  - Fixed portions of work; size is controlled by the value of chunk
  - When a thread finishes, it starts on the next portion of work
- guided [, chunk]
  - Same dynamic behaviour as "dynamic", but size of the portion of work decreases exponentially
- runtime
  - Iteration scheduling scheme is set at runtime through environment variable OMP_SCHEDULE

The SECTIONS directive

- The individual code blocks are distributed over the threads
- Clauses supported:
  - private
  - firstprivate
  - lastprivate
  - reduction
  - nowait

Synchronization Controls

Barrier

- Suppose we run each of these two loops in parallel over i:

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

- This may give us a wrong answer
- We need to have updated all of a[ ] first, before using a[ ]

Barrier

- Each thread waits until all others have reached this point:
  - #pragma omp barrier

When to use barriers?

- When data is updated asynchronously and the data integrity is at risk
- Examples:
  - Between parts in the code that read and write the same section of memory
  - After one timestep/iteration in a solver
- Unfortunately, barriers tend to be expensive and also may not scale to a large number of processors
- Therefore, use them with care

Critical region

- If sum is a shared variable, this loop can not be run in parallel

```
for (i=0; i < N; i++){
        .....
      sum += a[i];
        .....
}
```

- We can use a critical region for this:

```
for (i=0; i < N; i++){
        .....
 //one at a time can proceed
     sum += a[i];
 //next in line, please
        .....
}
```

Critical region

- Useful to avoid a race condition, or to perform I/O (but which still will have random order)
- Be aware that your parallel computation may be serialized and so this could introduce a scalability bottleneck (Amdahl's law)
- All threads execute the code, but only one at a time:
  - #pragma omp critical [(name)]
{<code-block>}
  - #pragma omp atomic
<statement>

SINGLE and MASTER construct

- Only one thread in the team executes the code enclosed

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

- Only the master thread executes the code block:

  #pragma omp master
  {<code-block>}

OpenMP Environment Variables

OpenMP environment variables

- OMP_NUM_THREADS n
- OMP_SCHEDULE “schedule,[chunk]”
- OMP_DYNAMIC { TRUE | FALSE }
- OMP_NESTED { TRUE | FALSE }

OpenMP Runtime Functions

OpenMP runtime environment

- OpenMP provides various user-callable functions
  - To control and query the parallel environment
  - General purpose semaphore/lock routines
    - Nested locks are supported, but will not be covered here
- The runtime functions take precedence over the corresponding environment variables
- Recommended to use under control of an #ifdef for _OPENMP (C/C++)
- C/C++ programs need to include <omp.h>

Runtime library overview

omp_set_num_threads	Set number of threads
omp_get_num_threads	Return number of threads in team
omp_get_max_threads	Return maximum number of threads
omp_get_thread_num	Get thread ID
omp_get_num_procs	Return maximum number of processors
omp_in_parallel	Check whether in parallel region
omp_set_dynamic	*Activate dynamic thread adjustment*
omp_get_dynamic	Check for dynamic thread adjustment
omp_set_nested	*Activate nested parallelism*
omp_get_nested	Check for nested parallelism
omp_get_wtime	Returns wall clock time
omp_get_wtick	Number of seconds between clock ticks

OpenMP locking routines

- Locks provide greater flexibility over critical sections and atomic updates:
  - Possible to implement asynchronous behaviour
  - Not block structured
- The so-called lock variable, is a special variable:
  - C/C++: type omp_lock_t and omp_nest_lock_t for nested locks
- Lock variables should be manipulated through the API only
- It is illegal, and behaviour is undefined, in case a lock variable is used without the appropriate initialization

Nested locking

- 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 interface for functions dealing with nested locks is similar (but using nestable lock variables):

Simple locks	**Nestable locks**
omp_init_lock	omp_init_nest_lock
omp_destroy_lock	omp_destroy_nest_lock
omp_set_lock	omp_set_nest_lock
omp_unset_lock	omp_unset_nest_lock
omp_test_lock	omp_test_nest_lock

OpenMP and Compilers

- OpenMP v2.5
  - Visual C++ 2005 (Professional and Team System editions)
  - Intel Parallel Studio
  - Sun Studio
  - Portland Group compilers
  - GCC since version 4.2.
- OpenMP v3.0
  - GCC 4.3.1
  - Nanos compiler
  - Intel Fortran and C/C++ versions 11.0 and 11.1 Compilers, and Intel Parallel Studio.
  - IBM XL C/C++ Compiler
  - Sun Studio 12 update 1

Program compilation

- `gcc -fopenmp -o test test.c`
- `icc -openmp -o test test.c`

Summary

- OpenMP provides for a compact, but yet powerful, programming model for shared memory programming
- OpenMP supports Fortran, C and C++
- OpenMP programs are portable to a wide range of systems
- An OpenMP program can be written such that the sequential version is still “built-in”