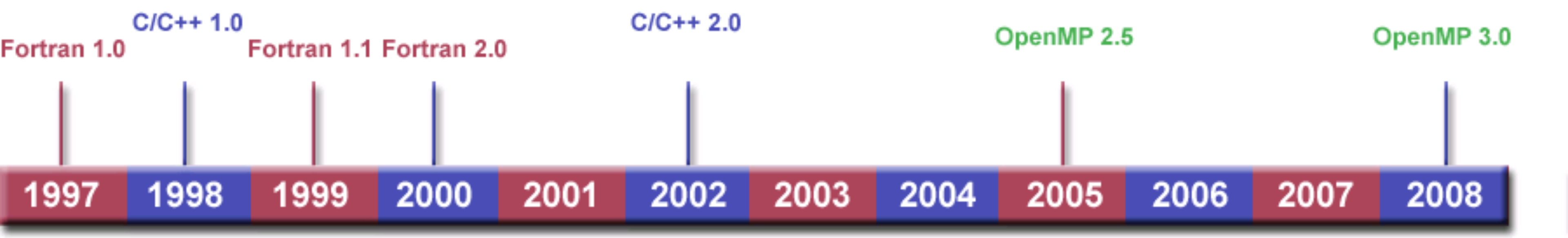
OpenMP

Alexandre David
1.2.05
adavid@cs.aau.dk

Release History



Some pictures from
https://computing.llnl.gov/tutorials/openMP/

Goals

- Standardization:
  - Standard among a variety of shared memory architectures/platforms
- Lean and Mean:
  - Simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives.
- Ease of Use:
  - Capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach.
  - Capability to implement both coarse-grain and fine-grain parallelism.
- Portability:
  - Supports Fortran (77, 90, and 95), C, and C++.
  - Public forum for API and membership.

Introduction

- Idea: Augment sequential program in minor ways to gain parallelism.
  - Directive based – using *#pragma*.
  - Simple.
  - More restrictive.
- C compiler that understands OpenMP will generate multi-threaded code automatically.
  - Other compilers ignore the directives.

Example

```
int count3s()
{
   int i, count_p;
   count=0;
   #pragma omp parallel shared(array, count, length)\
     private(count_p)
   {
     count_p=0;
     #pragma omp parallel for private(i)
     for(i=0; i<length; i++)
     {
       if(array[i]==3)
       {
         count_p++;
       }
     }
     #pragma omp critical
     {
       count+=count_p;
     }
   }
   return count;
}
```

Run this in parallel with shared & private var.

Example

```
int count3s()
{
   int i, count_p;
   count=0;
   #pragma omp parallel shared(array, count, length)\
      private(count_p)
   {
      count_p=0;
      #pragma omp parallel for private(i)
      for(i=0; i<length; i++)
      {
         if(array[i]==3)
         {
            count_p++;
         }
      }
      #pragma omp critical
      {
         count+=count_p;
      }
   }
   return count;
}
```

Iterate in parallel in any order.

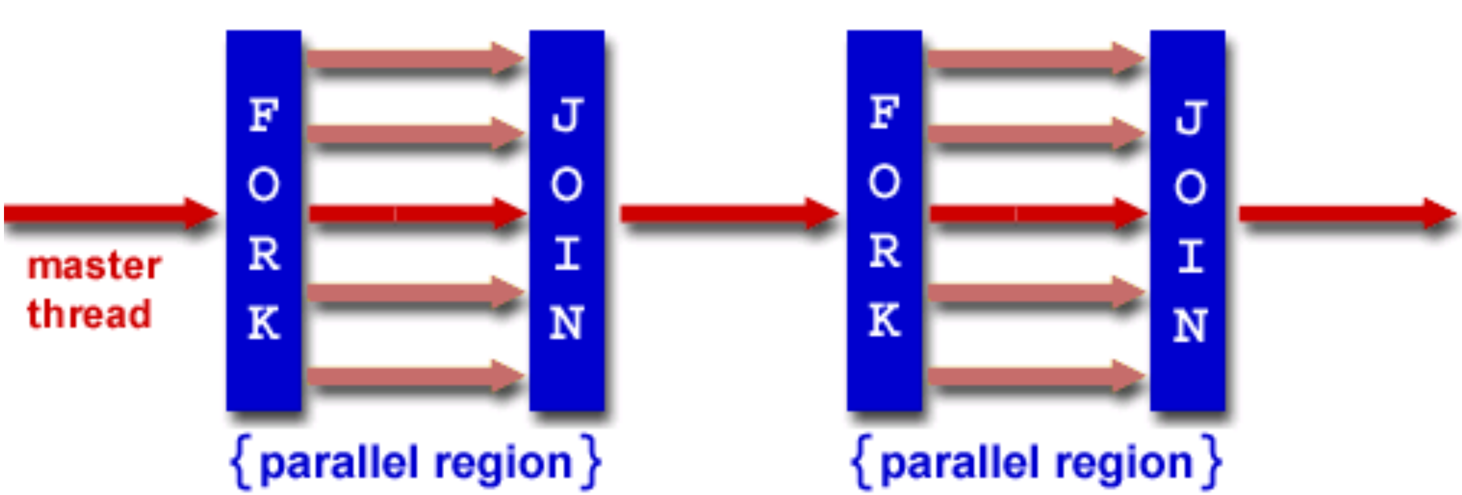
Example

```
int count3s()
{
   int i, count_p;
   count=0;
   #pragma omp parallel shared(array, count, length)\
      private(count_p)
   {
      count_p=0;
      #pragma omp parallel for private(i)
      for(i=0; i<length; i++)
      {
         if(array[i]==3)
         {
            count_p++;
         }
      }
      #pragma omp critical
      {
         count+=count_p;
      }
   }
   return count;
}
```

Locked access.

Programming Model

- Shared memory, thread based parallelism.
- Explicit parallelism.
- Fork-join model



Programming Model

- Fork-join
  - All OpenMP programs begin as a single process: the **master thread**.
    The master thread executes sequentially until the first **parallel region** construct is encountered.
  - **FORK:** the master thread then creates a ***team*** of parallel threads.
    - The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads
  - **JOIN:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

Programming Model

- Compiler directive based.
  - Nested parallelism.
  - Dynamic threads.
  - No support for I/O.
- Memory model: relaxed consistency, flush to maintain consistency.

Peril-L Concepts

- Parallelism – *parallel for*
  - independent iterations
  - certain types of for-loops only
- Reductions – *reduction(op,var)*
  - split iterations of a loop and accumulate the result automatically

```
count=0;
#pragma omp parallel for reduction(+,count)
for(i=0; i<length; i++)
{
  count +=(array[i]==3)?:1:0
}
```

Parallel For

parallel for

```
#pragma omp parallel for
  for(<var>=<expr1>; <var> <relop> <expr2>; <var>=<expr3>)(<body> }
```

Conditions:

- <*var*> must be a signed integer variable and the same in each instance.
- <*relop*> must be one of <, <=, =>, >.
- <*expr2*>, <*expr3*> must be a loop-invariant integer expression.
- if <*relop*> is < or <=, <*expr3*> must increment each iteration; if <relop> is >, >=, <*expr3*> must decrement each iteration.
- <*body*> must be a basic block, that is, it has no other entries or exits.

Notes:

- Optional specifications on the pragma line include `private` and `nowait`.
- A set of threads created for a `parallel for` will join at completion, implying a barrier synchronization.

Reduction

reduction

```
reduction(<op>:<list>)
```

Conditions:

- <*op*> is one of the operators in the accompanying table; its identity is the value that is used as the left operand for the first step of the reduce operation.
- <*list*> is a set of variables into which the reduce accumulates; for example, `count` in the Count 3s example.

Notes:

Fortran has several more <*op*> choices, including `min` and `max`.

<table>
<tr><th><<i>op</i>></th><th>Identity</th></tr>
<tr><td>+</td><td>0</td></tr>
<tr><td>*</td><td>1</td></tr>
<tr><td>-</td><td>0</td></tr>
<tr><td>&</td><td>~0</td></tr>
<tr><td>|</td><td>0</td></tr>
<tr><td>^</td><td>0</td></tr>
<tr><td>&&</td><td>1</td></tr>
<tr><td>||</td><td>0</td></tr>
</table>

Threads

- Threads are created upon “parallel for”
  - pthread_create
- Threads are joined at the end of the block – implicit barrier
  - pthread_join
  - Can be avoided by
    #pragma omp parallel for nowait
    (useful if followed by another parallel for)
- Atomicity
  - #pragma omp atomic

Atomicity

atomic

```
#pragma omp atomic
  <var> <op> <expr> | <expr>++ | <expr>-- | ++<expr> | --<expr>
```

Result:

The statement following the pragma becomes uninterruptible.

Conditions:

- *<var>* is a program variable.
- *<op>* is one of the operations: `+=`, `-=`, `*=`, `/=`, `<<=`, `>>=`, `&=`, `|=`, `^=`.
- *<expr>* is any legal expression.

Notes:

Use of atomic in a loop can have serious performance implications.

Restricted operations.
Reason: They correspond to special assembly instructions.

Critical Sections

- #pragma omp exclusive(section_name)
```
{
    …
}
```
- The name identifies the critical section.
- Corresponds to locking/unlocking a given mutex.
  - pthread_mutex_lock/pthread_mutex_unlock

Sections

- Sections specify task parallelism – independent tasks.
  - ```
    #pragma omp sections
    {
      #pragma omp section
      {
        Task_A();
      }
      #pragma omp section
      {
        Task_B();
      }
    }
    ```
    ```

Matrix Multiplication

```
void mult(const int *a, const int *b, int c, int n)
{
    int i;
    #pragma parallel for shared(a,b,c,n) private(i)
    for(i=0; i<n; ++i)
    {
        …loops on j & k
    }
}
```

Other Synchronization Primitives

- Barrier - #pragma omp barrier.
- Tasks – creation (omp task) & wait for completion (taskwait).

Access to the OMP Runtime

```
#include <omp.h>

void omp_set_num_threads(int);
int omp_get_num_threads();
int omp_get_thread_num();
int omp_get_num_procs();
...
```

Compiler

- gcc 4.3.2 with –fopenmp option
  - installed on the system
- Try yourself, best way to learn.
  - You will get some exercises on it.
  - Tutorials on www.openmp.org