University of Minho
Informatics Department

Parallel Computing Paradigms

OpenMP 4.0 (what is new?)

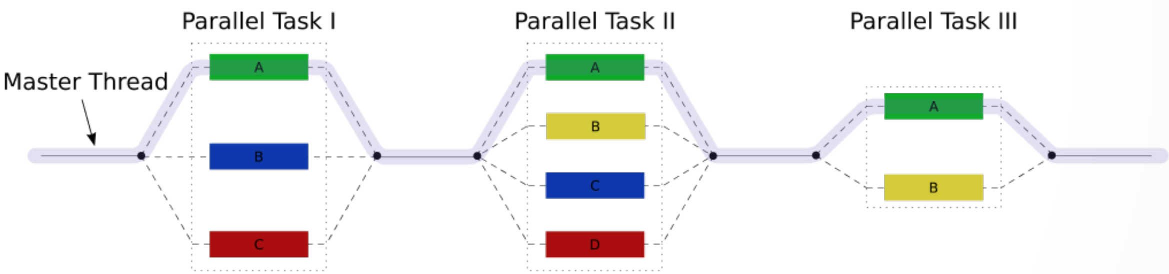
OpenMP 4.0

João Luis Sobral
Bruno Medeiros

Introduction to OpenMP (review)

- OpenMP is a standard to Shared Memory (SM) parallel programming (e.g., on multi-core machines).

- Execution model is based on the **fork-join** model of parallel execution.



- The creation/managing of threads are delegated to the compiler and OpenMP runtime
  - Easier to parallelize an application
  - Less controlover threads' behavior

Review: OpenMP constructs (1)

- OpenMP directives format for C/C++ applications:

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
block of code // group of statements separated by semicolons, enclosed in braces
```

- **Parallel Construct**
  - *#pragma omp parallel* Creates a team of threads.

- **Work-sharing Constructs**
  - *#pragma omp for* Assignment of loop iterations to threads.
  - #pragma omp sections Assignment of blocks of code (section) to threads.
  - #pragma omp single Restricts a code of block to be executed by a single thread.

Review: OpenMP constructs (2)

- **Tasking Constructs**
  - #pragma omp task Creation of a pool of tasks to be executed by threads.

- **Master & Synchronization Constructs**
  - #pragma omp master Restricts a block of code to be executed only the master thread.
  - #pragma omp critical Restricts the execution of a block of code to a single thread at a time.
  - #pragma omp barrier Makes all threads in a team to wait for the remaining.
  - #pragma omp taskwait wait for the completion of the current task child's.
  - #pragma omp atomic Ensures that a specific storage location is managed atomically.
  - #pragma omp flush Makes a thread's temporary view of memory consistent with memory.
  - #pragma omp ordered Specifies a block of code in a loop region that will be executed in the order of the loop iterations.

What is new in OpenMP 4.0 (vs 3.0)?

- Places and threads affinity
- Array sections
- Taskyield, taskgroup and dependent task
- User defined reductions
- Construct cancellation
- SIMD Construct
- Device Construct
- Teams

Parallel Region

- When a thread encounters a parallel construct, a team of threads is created (**FORK**);
- The thread which encounters the parallel region becomes the **master** of the new team;
- **All threads** in the team (including the master) execute the region;
- At end of parallel region, all threads synchronize, and join master thread (**JOIN**).

Parallel region syntax

```
#pragma omp parallel [clauses]
{
        code_block
}
```

Where clause can be:

```
if (scalar-expression)
num_threads (integer-expression)
private (list)
firstprivate (list)
shared (list)
reduction (operator: list)
```

(new OpenMP 4.0)

```
proc_bind (master | close | spread)
```

Controlling OpenMP Thread Affinity

- Since many system are now **NUMA**, placement of threads on the hardware can have a big effect on performance

- **proc_bind (master | close | spread)**
  - The **master** thread affinity policy instructs the execution environment to assign every thread in the team to the same place as the master thread.
  - The **close** thread affinity policy instructs the execution environment to assign the threads to places close to the place of the parent thread.
  - The purpose of the **spread** thread affinity policy is to create a sparse distribution for a team of *T* threads among the *P* places of the parent's place partition.

Thread Affinity Example

- Two sockets, each one with a quad-core processor and configured to execute two hardware threads simultaneously on each core

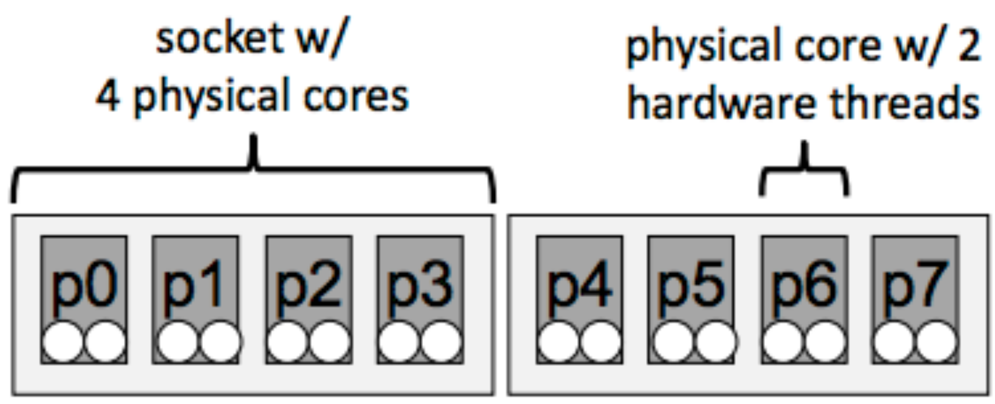


Image Reference : http://openmp.org/mp-documents/OpenMP_Examples_4.0.1.pdf

  - Composed by 8 places (which are designated as p0 to p7).

Affinity with 4 threads (master on p0)

Spread - as widespread as possible

- Thread 0 on p0
- Thread 1 on p2
- Thread 2 on p4
- Thread 4 on p6

Close - as close as possible

- Thread 0 on p0
- Thread 1 on p1
- Thread 2 on p2
- Thread 4 on p3

Master

- Thread 0 to 3 on p0

Array Section

- An array section designates a subset of the elements in an array. An array section can appear only in clauses where it is explicitly allowed
  - **[** *lower-bound* **:** *length* **]**
  - **[** *lower-bound* **:]**
  - **[:***length***]**
  - **[:]**
  - When the length is absent it defaults to the size of the array dimension minus the lower-bound
  - When the lower-bound is absent it defaults to 0

Task construct

- When a thread encounters a task construct a new task is generated

- The task can be immediately executed or it can be executed later by any thread in the team

- OpenMP creates a pool of tasks to be executed by the active threads in the team

- OpenMP 4.0 allows **task dependencies specifications**

Task constructor:

```
int fib(int n)
{
  int i, j;
  if (n<2) return n;
  else
    {
       #pragma omp task shared(i) firstprivate(n)
       i=fib(n-1);

       #pragma omp task shared(j) firstprivate(n)
       j=fib(n-2);

       #pragma omp taskwait
       return i+j;
    }
}

int main()
{
  int n = 10;

  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
}
```

- The **taskwait** directive ensures that the tasks generated are completed before the return statements.
- Although, only one thread executes the **single** directive and hence the call to fib(n), **all four threads** will participate in executing the tasks generated.

Task dependencies

- The **depend** clause enforces additional constraints on the scheduling of tasks sharing the same parent
  - establishes dependences only between **sibling tasks**
- #pragma omp task **depend** (**type** : list)
  - Where **type** is

    in - the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an **out** or **inout** clause

    out or **inout -** the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in **in**, **out** or **inout** clause

    and **list** is list of variables that may contain sub arrays

Flow dependence (RAW)

- The program will always print "x = 2” because the depend clauses enforce the ordering of the tasks
- If the depend clauses are omitted, then the tasks can execute in any order and the program will have a race condition

```
void flowDependence()
{
     int x = 1;
     #pragma omp parallel
     {
          #pragma omp single
          {
            #pragma omp task shared (x) depend (out: x)
               x = 2;
            #pragma omp task shared (x) depend (in : x)
               printf(“x = %d\n”, x);
          }
     }
}
```

Anti-dependence (WAR)

- The program will always print "x = 1" because the depend clauses enforce the ordering of the tasks
- If the depend clauses are omitted, then the tasks can execute in any order and the program will have a race condition

```
void antiDependence()
{
     int x = 1;
     #pragma omp parallel
     {
          #pragma omp single
          {
             #pragma omp task shared (x) depend (in: x)
                printf("x = %d\n", x);
             #pragma omp task shared (x) depend (out: x)
                x = 2;
          }
     }
}
```

Output dependence (WAW)

- The program will always print "x = 2" because the depend clauses enforce the ordering of the tasks
- If the depend clauses are omitted, then the tasks can execute in any order and the program will have a race condition

```
void outDependence(){
	int x;
	#pragma omp parallel
	{
		#pragma omp single
		{
		   #pragma omp task shared (x) depend (out: x)
		      x = 1;
		   #pragma omp task shared (x) depend (out: x)
		      x = 2;
		   #pragma omp taskwait
		      printf("x = %d\n", x);
		}
	}
}
```

Matrix Multiplication with blocks

- This example shows a task-based blocked matrix multiplication. Matrices are of NxN elements and the multiplication is implemented using blocks of BSxBS elements

```
// Assume BS divides N perfectly
void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float C[N][N])
{
   int i, j, k, ii, jj, kk;
   for(i = 0; i < N; i += BS)
     for(j = 0; j < N; j += BS)
      for(k = 0; k < N; k += BS)
      {
         #pragma omp task depend (in: A[i:BS][k:BS], B[k:BS][j:BS]) \
                          depend (inout: C[i:BS][j:BS])
                for(ii = i; ii < i + BS; ii++)
                 for(jj = j; jj < j + BS; jj++)
                  for(kk = k; kk < k + BS; kk++)
                    C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
      }
}
```

Taskyield construct

- The taskyield construct specifies that the current task can be suspended in favor of execution of a different task

- The taskyield construct is a stand-alone directive

```
void foo(omp_lock_t *lock, int N)
{
  for(int i = 0; i < N; i++)
   #pragma omp task
   {
      something_useful();
      while(!omp_test_lock(lock))
      {
         #pragma omp taskyield
      }
      something_critical();
    omp_unset_lock(lock);
   }
 }
```

- The task computes something_useful() and then do some necessary computation in a critical region

- By using taskyield when a task cannot get access to the critical region the implementation can suspend the current task and schedule some other task that can do something useful

User defined reductions

- The declared reduction directive allows to define new reduction operators, that can be used in a reduction clause
- #pragma omp declare reduction **(reduction-identifier: typename-list : combiner) [initialize-clause] new– line**
- reduction **–** identifier: can be the name of the user-defined operator or one of the following operators: +, -, *, &, | , ^, && and | |
- typename – list is a list of types to which it applies
- combiner – expression specifies how to combine the values

Construct cancellation

- The cancel construct activates cancellation of the innermost enclosing region of the type specified

#pragma omp cancel **construct [if (expr)]**

- The cancel construct can be
  - parallel
  - sections
  - for
  - taskgroup

Construct cancellation example

```
void testCancel()
{
 int t = 1;
 #pragma omp parallel firstprivate(t)
 {
    #pragma omp for
    for(int i = 0; i < 100; i++)
    {
      t = test();
      #pragma omp cancel for if (t == 5)
    }
    #pragma omp cancel parallel if (t == 5)
    #pragma omp barrier
    printf("Thread %d \n", omp_get_thread_num());
  }
}
```

- The first thread to which `t == 5` is true will cancel the `parallel for` and `parallel region` and exit
- Other threads exit next time they hit the `cancel` directive
- `cancellation` is disabled by default, to enable it is necessary to set the environment variable `OMP_CANCELLATION` to `true`
  - E.g. `$OMP_CANCELLATION=true ./test`

SIMD Constructs

- **simd construct**
  - The **simd** construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop
- **declare simd construct**
  - The **declare simd** construct can be applied to a function (C, C++ and Fortran) to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop.
- **Loop SIMD construct**
  - The loop SIMD construct specifies a loop that can be executed concurrently using SIMD instructions and that those iterations will also be executed in parallel by threads in the team.

Simd Construct

- enables the execution of multiple iterations of the associated loops concurrently by means of SIMD instructions
  - Can enable vectorization of both sequential and parallel for loops
    - #pragma omp for simd [clauses]
  - Can also indicate to OpenMP to create versions of functions that can be invoked across SIMD lanes
    - #pragma omp declare simd [clauses]
  - If the **safelen** clause is used then no two iterations executed concurrently with SIMD instructions can have a greater distance in the logical iteration space than its value.
  - The **aligned** clause declares that the object to which each list item points is aligned to the number of bytes expressed in the optional parameter of the **aligned** clause.

syntax

```
#pragma omp simd[clauses]
{
        for-loop
}
```

Where clause can be:

safelen(*length*)

linear(*list[:linear-step]*)

aligned(*list[:alignment]*)

private(*list*)

lastprivate(*list*)

reduction(*reduction-identifier:list*)

collapse(*n*)

Declare simd Construct

- enables the creation of SIMD versions of the associated function that can be used to process multiple arguments from a single invocation from a SIMD loop concurrently.
  - the number of concurrent arguments for the function is determined by the **simdlen** clause
  - The **uniform** clause declares one or more arguments to have an invariant value for all concurrent invocations of the function
  - The **inbranch** clause specifies that the function will always be called from inside a conditional statement of a SIMD loop. The **notinbranch** clause specifies that the function will never be called from inside a conditional statement of a SIMD loop.

syntax

```
#pragma omp declare simd[clauses]
{
        function
}
```

Where clause can be:

simdlen(*length*)

linear(*argument-list[:constant-linear-s*

aligned(*argument-list[:alignment]*)

uniform(*argument-list*)

inbranch

notinbranch

Simd Construct example

```
#pragma omp declare simd
double inc (int i)
{
        return i + 1;
}
int main()
{
  int d1 = 0, N = 100;
  double a[N], b[N], d2 = 0.0;

  #pragma omp simd reduction(+:d1)
  for(int i = 0; i < N; i++)
     d1 += i * inc(i);

  #pragma omp parallel for simd reduction(+:d2)
  for(int i = 0; i < N; i++)
     d2 += a[i] * b[i];
}
```

Device Constructs

- **target data Construct**
  - Create a device data environment for the extent of the region.
- **target Construct**
  - Create a device data environment and execute the construct on the same device.
- **target update Construct**
  - makes the corresponding list items in the device data environment consistent with their original list items, according to the specified motion clauses.
- **declare target Directive**
  - specifies that variables, functions (C, C++) are mapped to a device.

Target Construct

- Creates a device data environment and execute the construct on the same device
  - The encountering task waits for the device to complete the target region.
  - Provides a superset of the functionality and restrictions provided by the **target data** directive. The functionality added to the **target** directive is the inclusion of an executable region to be executed by a device.
  - **#pragma omp target update** *clause[[,] clause],...] new-line*
    - where *motion-clause* is one of the following: **to(** *list* **)** **from(** *list* **)**
  - makes the corresponding list items in the device data environment consistent with their original list items

syntax

#pragma omp target[clauses]

```
{
    block
}
```

Where clause can be:

`device` (*integer-expression*)

`map` (*[map-type* **:** *] list*)

`if` (*scalar-expression*)

Teams

- **teams Construct**
  - creates a league of thread teams and the master thread of each team executes the region.
  - When a thread encounters a **teams** construct, a league of thread teams is created and the master thread of each thread team executes the **teams** region.
- **distribute Construct**
  - the iterations of one or more loops will be executed by the thread teams in the context of their implicit tasks.
- **distribute simd Construct**
  - The **distribute simd** construct specifies a loop that will be distributed across the master threads of the **teams** region and executed concurrently using SIMD instructions.
- **Distribute Parallel Loop Construct**
- **Distribute Parallel Loop SIMD Construct**

Teams Construct

- A league of thread teams is created and the master thread of each thread team executes the **teams** region
    - The threads other than the master thread do not begin execution until the master thread encounters a **parallel** region.

- **#pragma omp distribute** *[clause[[,] clause],...] new-line*
- *for-loops*
    - The **distribute** construct specifies that the iterations of one or more loops will be executed by the thread teams.
    - The iterations are distributed across the master threads of all teams that execute the **teams** region.

syntax

```
#pragma omp teams [clauses]
{
    block
}
```

Where clause can be:

num_teams(*integer-expression* **)**

thread_limit(*integer-expression* **)**

default(shared | none)

private(*list* **)**

firstprivate(*list* **)**

shared(*list* **)**

reduction(*reduction-identifier* **:** *list* **)**

Version 3.1 to 4.0 Differences

- Various changes throughout the specification were made to provide initial support of Fortran 2003 (see Section 1.6 on page 22).
- C/C++ array syntax was extended to support array sections (see Section 2.4 on page 42).
- The **proc_bind** clause (see Section 2.5.2 on page 49), the **OMP_PLACES** environment variable (see Section 4.5 on page 241), and the **omp_get_proc_bind** runtime routine (see Section 3.2.22 on page 216) were added to support thread affinity policies.
- SIMD constructs were added to support SIMD parallelism (see Section 2.8 on page 68).
- Device constructs (see Section 2.9 on page 77), the **OMP_DEFAULT_DEVICE** environment variable (see Section 4.13 on page 248), the **omp_set_default_device**, **omp_get_default_device**, **omp_get_num_devices**, **omp_get_num_teams**, **omp_get_team_num**, and **omp_is_initial_device** routines were added to support execution on devices.
- Implementation defined task scheduling points for untied tasks were removed (see Section 2.11.3 on page 118).
- The **depend** clause (see Section 2.11.1.1 on page 116) was added to support task dependences.
- The **taskgroup** construct (see Section 2.12.5 on page 126) was added to support more flexible deep task synchronization.
- The **reduction** clause (see Section 2.14.3.6 on page 167) was extended and the **declare reduction** construct (see Section 2.15 on page 180) was added to support user defined reductions.
- The **atomic** construct (see Section 2.12.6 on page 127) was extended to support atomic swap with the **capture** clause, to allow new atomic update and capture forms, and to support sequentially consistent atomic operations with a new **seq_cst** clause.
- The **cancel** construct (see Section 2.13.1 on page 140), the **cancellation point** construct (see Section 2.13.2 on page 143), the **omp_get_cancellation** runtime routine (see Section 3.2.9 on page 199) and the **OMP_CANCELLATION** environment variable (see Section 4.11 on page 246) were added to support the concept of cancellation.