OpenMP Execution Model

Execution Model

- OpenMP API uses the fork-join model of parallel execution.
- Multiple threads of execution perform tasks defined implicitly or explicitly by openMP directives
- It is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full openMP support library) and as sequential programs (directives ignored and a simple openMP stubs library).

Execution Model(cont...)

- An OpenMP program begins as a single thread of execution, called an initial thread.
- An initial thread executes sequentially, as if enclosed in an implicit task region, called an initial task region that is defined by an implicit **inactive parallel region** that surrounds the entire target region.
- When a target construct is encountered, the target region is executed by the implicit device task.
- The task that encounters the target construct **waits** at the end of the construct until execution of the region completes.
    - If a target device doesn't exist, or the target device is not supported by the implementation, or the target device cannot execute the target construct then the target region is executed by the host device.

Execution Model(cont...)

- The teams construct creates a ***league of thread teams*** where the **masters thread** of each team executes the region.
  - Each of these master threads is an initial thread, and executes sequentially, as if enclosed in an implicit task region that is defined by an **implicit parallel region** that surrounds the entire teams region.
- When any thread encounters a parallel construct, the thread creates a team of itself and zero or more additional threads and becomes the master of the new team.
  - A **set of implicit tasks**, one per thread, is generated.
  - The code for each task is defined by the code **inside the parallel construct.**

Execution Model(cont...)

- Each task is assigned to a different thread in the team and becomes tied, ie., it is always executed by the <u>thread</u> to which it is <u>initially assigned.</u>
  - The task region of the task being executed by the encountering thread is suspended, and each member of the new team executes its implicit task.
  - There is an <u>**implicit barrier**</u> at the end of the parallel construct.
  - Only the master thread resumes execution beyond the end of the parallel construct, resuming the task region that was suspended upon encountering the parallel constuct.
  - Any no. of parallel constructs can be specified in a single program.

Execution Model : nested parallelism

- Parallel regions may be arbitrarily nested inside each other.
- If nested parallelism is disabled, or is not supported by the OpenMP implementation, then the new team that is created by a thread encountering a parallel construct inside a parallel region will consist only of the encountering thread. (ie. Encountering thread alone will be present there and executes the implementation).
- However, if nested parallelism is supported and enabled, then the new team can consist of more than one thread.
    - A parallel construct may include a **proc_bind** clause to specify the places to use for the threads in the team within the parallel region.

Execution Model : Work Sharing Constructs

- When any team encounters a work sharing construct, the work inside the construct is divided among the members of the team, and executed cooperatively instead of being executed by every thread.
- There is a default barrier at the end of each work sharing construct
- Redundant execution of code by every thread in the team resumes after the end of the work sharing construct.
- When any thread encounters a task construct, a new explicit task is generated.
- Execution of explicitly generated tasks is assigned to one of the threads in the current team, subject to the thread's availability to execute work.

Execution Model : Work Sharing Constructs (cont...)

- Thus, execution of the new task could be immediate, or deferred unitl later according to task scheduling constraints and thread availability.
- Threads are allowed to suspend the current task region at a task scheduling point in order to execute a different task.
- If the suspended task region is for tied task, the initially assigned thread resumes execution of the suspended task region.
- If the suspended task region is for untied task, then any thread may resume its execution.

Execution Model : Work Sharing Constructs (cont...)

- Completion of all explicit tasks bound to a given parallel region is guaranteed before the master thread leaves the implicit barrier at the end of the region.
- Completion of a subset of all explicit tasks bound to a given parallel region may be specified through the use of task synchronization constructs.
- Completion of all explicit tasks bound to a implicit parallel region is guaranteed by the time the program exits.

Execution Model : construct-type clause

- When any thread encounters a SIMD construct, the iterations of the loop associated with the construct may be executed concurrently using the SIMD lanes that are available to the thread.
- The cancel construct can alter the previously described flow of execution in an OpenMP region.
- Effect of the cancel construct depends on its construct-type clause.
- If a task encounters a cancel construct with a task group construct-type clause, then the task activates cancellation and continues execution at the end of its task region, which implies completion of that task.

Parallel region construct

- A block of code that will be executed by multiple threads.

  #pragma omp parallel ***[clause …]***

  {

  ……

  } (implied barrier)

  Clauses*: if (expression),* private *(list)*, shared *(list),* default (shared | none), reduction *(operator: list),* firstprivate*(list),* lastprivate*(list)*

  - if (expression): only in parallel if expression evaluates to true
  - private(list): everything private and local (no relation with variables outside the block).
  - shared(list): data accessed by all threads
  - default (none|shared)

- The reduction clause:

```
Sum = 0.0;
#pragma parallel default(none) shared (n, x) private (I) reduction(+ : sum)
{
  For(I=0; I<n; I++) sum = sum + x(I);
}
```

  - Updating sum must avoid racing condition
  - With the reduction clause, OpenMP generates code such that the race condition is avoided.
- Firstprivate(list): variables are initialized with the value before entering the block
- Lastprivate(list): variables are updated going out of the block.

Work-sharing constructs

- #pragma omp for [clause …]
- #pragma omp section [clause …]
- #pragma omp single [clause …]

- The work is distributed over the threads
- Must be enclosed in parallel region
- No implied barrier on entry, implied barrier on exit (unless specified otherwise)

The omp for directive: example

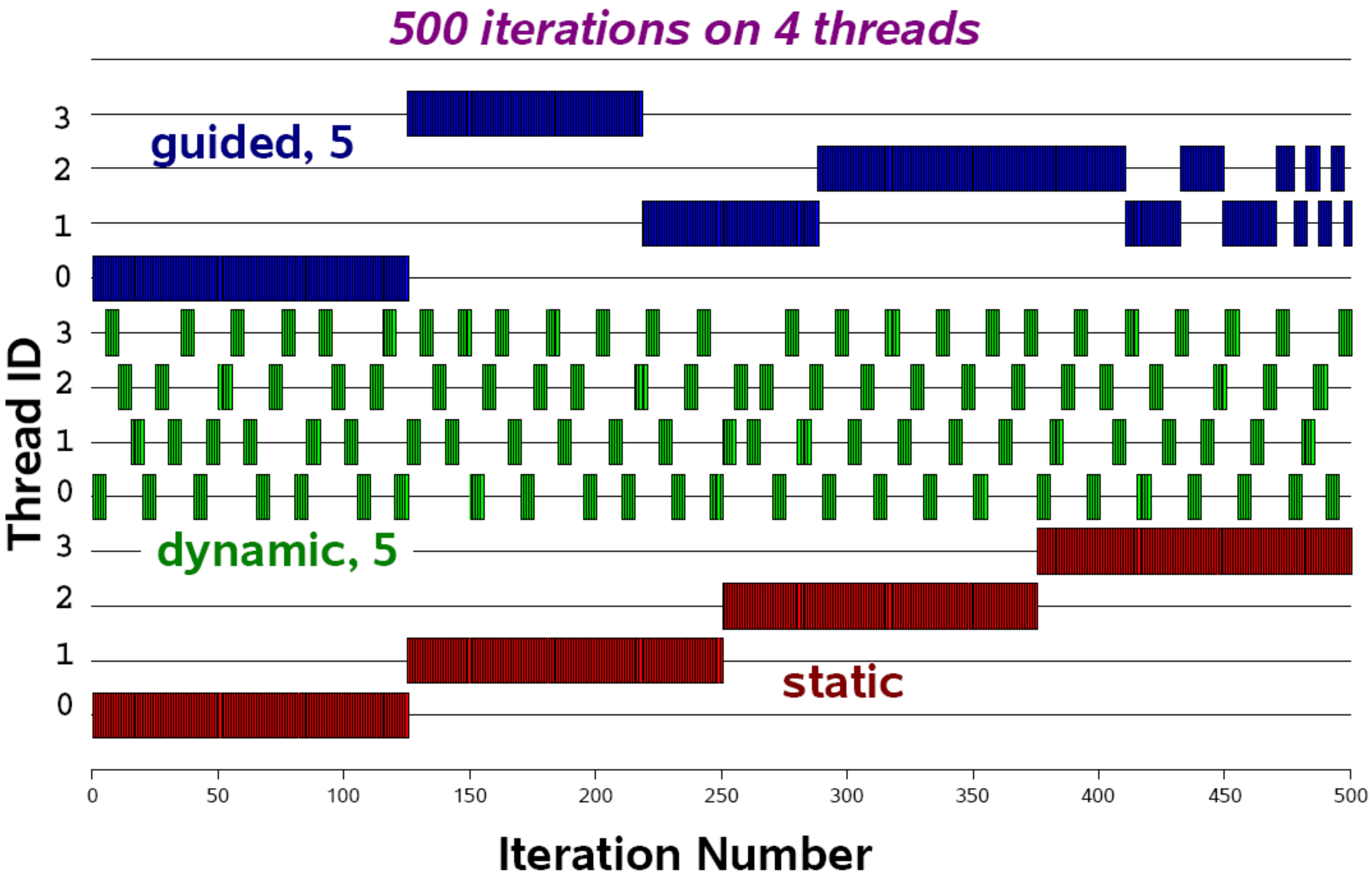
```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
     for (i=0; i<n-1; i++)
         b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
     for (i=0; i<n; i++)
         d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
```

(implied barrier)

- Schedule clause (decide how the iterations are executed in parallel):

schedule (static | dynamic | guided [, chunk])



The omp session clause - example

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
       for (i=0; i<n-1; i++)
           b[i] = (a[i] + a[i+1])/2;

      #pragma omp section
       for (i=0; i<n; i++)
           d[i] = 1.0/c[i];

    } /*-- End of sections --*/

  } /*-- End of parallel region --*/
```

```
#pragma omp parallel
#pragma omp for
   for (...)
```

```
#pragma omp parallel for
for (....)
```

Single PARALLEL loop

```
#pragma omp parallel
#pragma omp sections
{ ...}
```

```
#pragma omp parallel sections
{ ... }
```

Single PARALLEL sections

Synchronization: barrier

```
For(I=0; I<N; I++)
   a[I] = b[I] + c[I];

For(I=0; I<N; I++)
   d[I] = a[I] + b[I]
```

Both loops are in parallel region
With no synchronization in between.
What is the problem?

Fix:

```
For(I=0; I<N; I++)
   a[I] = b[I] + c[I];

#pragma omp barrier

For(I=0; I<N; I++)
   d[I] = a[I] + b[I]
```

Critical session

```
For(I=0; I<N; I++) {
   ……
   sum += A[I];
   ……
}
```

Cannot be parallelized if sum is shared.

Fix:

```
For(I=0; I<N; I++) {
   ……
   #pragma omp critical
  {
    sum += A[I];
   }
   ……
}
```

OpenMP environment variables

- OMP_NUM_THREADS
- OMP_SCHEDULE

OpenMP runtime environment

- omp_get_num_threads
- omp_get_thread_num
- omp_in_parallel
- Routines related to locks
- ……

OpenMP example

- See pi.c

Sequential Matrix Multiply

```
For (I=0; I<n; I++)
    for (j=0; j<n; j++)
        c[I][j] = 0;
        for (k=0; k<n; k++)
            c[I][j] = c[I][j] + a[I][k] * b[k][j];
```

OpenMP Matrix Multiply

```
#pragma omp parallel for private(j, k)
For (I=0; I<n; I++)
    for (j=0; j<n; j++)
        c[I][j] = 0;
        for (k=0; k<n; k++)
            c[I][j] = c[I][j] + a[I][k] * b[k][j];
```

Travelling Salesman Problem(TSP)

- The map is represented as a graph with nodes representing cities and edges representing the distances between cities.
- A special node (cities) is the starting point of the tour.
- Travelling salesman problem is to find the circle (starting point) that covers all nodes with the smallest distance.
- This is a well known NP-complete problem.

Sequential TSP

```
Init_q(); init_best();
While ((p = dequeue()) != NULL) {
    for each expansion by one city {
        q = addcity (p);
        if (complete(q)) {update_best(q);}
        else enqueue(q);
    }
}
```

OpenMP TSP

```
Do_work() {
  While ((p = dequeue()) != NULL) {
    for each expansion by one city {
        q = addcity (p);
        if (complete(q)) {update_best(q);}
        else enqueue(q);
    }
  }
}

main() {
   init_q(); init_best();
   #pragma omp parallel for
   for (i=0; I < NPROCS; i++)
      do_work();
}
```

Sequential SOR

```
for some number of timesteps/iterations {
   for (i=0; i<n; i++ )
           for( j=1, j<n, j++ )
                       temp[i][j] = 0.25 *
                                  ( grid[i-1][j] + grid[i+1][j]
                                  grid[i][j-1] + grid[i][j+1] );
   for( i=0; i<n; i++ )
           for( j=1; j<n; j++ )
                       grid[i][j] = temp[i][j];
}
```

- OpenMP version?

- Summary:
  - OpenMP provides a compact, yet powerful programming model for shared memory programming
    - It is very easy to use OpenMP to create parallel programs.
  - OpenMP preserves the sequential version of the program
  - Developing an OpenMP program:
    - Start from a sequential program
    - Identify the code segment that takes most of the time.
    - Determine whether the important loops can be parallelized
      - The loops may have critical sections, reduction variables, etc
    - Determine the shared and private variables.
    - Add directives

OpenMP discussion

- Ease of use
  - OpenMP takes cares of the thread maintenance.
    - Big improvement over pthread.
  - Synchronization
    - Much higher constructs (critical section, barrier).
    - Big improvement over pthread.

- OpenMP is easy to use!!

OpenMP discussion

- Expressiveness
  - Data parallelism:
    - MM and SOR
    - Fits nicely in the paradigm
  - Task parallelism:
    - TSP
    - Somewhat awkward. Use OpenMP constructs to create threads. OpenMP is not much different from pthread.

OpenMP discussion

- Exposing architecture features (performance):
  - Not much, similar to the pthread approach
    - Assumption: dividing job into threads = improved performance.
    - How valid is this assumption in reality?
      - Overheads, contentions, synchronizations, etc
  - This is one weak point for OpenMP: the performance of an OpenMP program is somewhat hard to understand.

OpenMP final thoughts

- Main issues with OpenMP: performance
  - Is there any obvious way to solve this?
    - Exposing more architecture features?
  - Is the performance issue more related to the fundamantal way that we write parallel program?
    - OpenMP programs begin with sequential programs.
    - May need to find a new way to write efficient parallel programs in order to really solve the problem.

References

1.www.cs.fsu.edu/~xyuan/cda5125/lect9_openmp.ppt
2.http://eclass.uoa.gr/modules/document/file.php/D186/%CE%8E%CE%BB%CE%B7%202011-12/PachecoChapter_5.pdf
3. www.cs.utah.edu/~mhall/cs4961f11/CS4961-L5.ppt
4.sddsd