National Institute of Technology Karnataka Surathkal
Department of Information Technology

IT 301 Parallel Computing

Shared Memory Programming Technique (7)

OpenMP : *Task, taskwait*

Dr. Geetha V

Assistant Professor

Dept of Information Technology

NITK Surathkal

Index

Course Outline

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

Course Outline

Part B: OpenMP/MPI/CUDA

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if(), threadprivate, copyin, copyprivate*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

Practical:

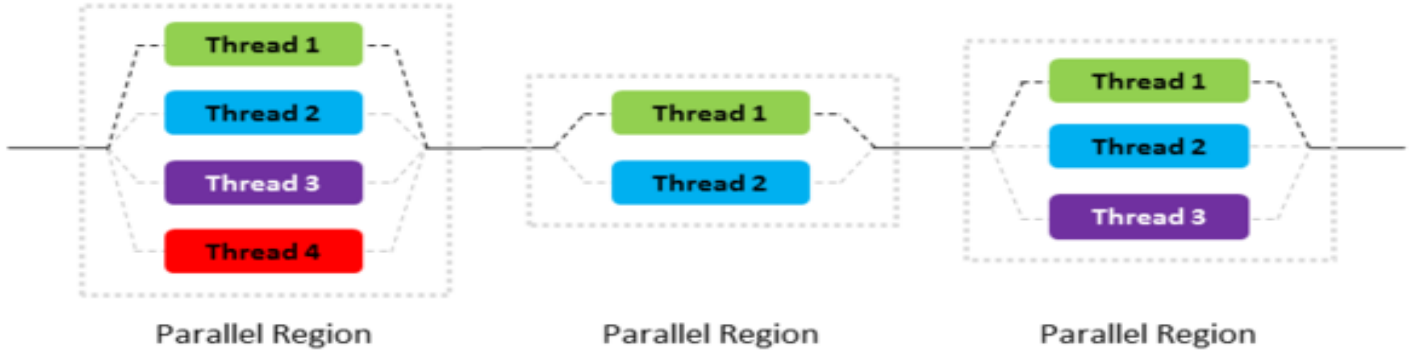
Implementation of parallel programs using OpenMP/MPI/CUDA.

Assignment: Performance evaluation of parallel algorithms (in group of 2 or 3 members)

1. OpenMP

FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
  - Create a group of threads by FORK.
  - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.

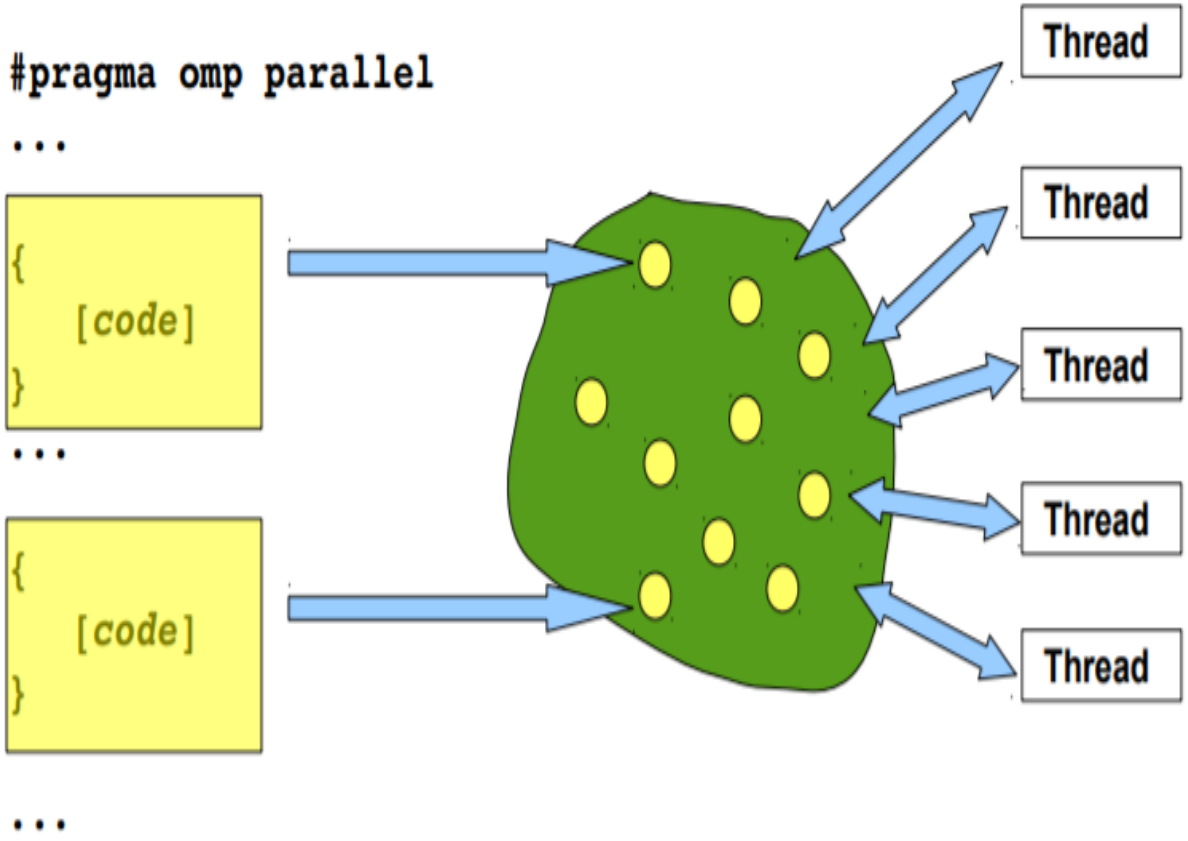


2. OpenMP Programming: Task

- Tasks are available starting with OpenMP 3.0
- A task is composed of
  - Code to be executed
  - Data environment (inputs to be used and outputs to be generated)
  - A Location where the task will be executed (thread)
- The tasks were initially implicit in OpenMP
  - A parallel construct constructs implicit tasks, one per thread
  - Teams of threads are created (or declared)
  - Threads in teams are assigned to each task
  - They synchronize with the master thread using barrier once all tasks completed
- Allowing the application to explicitly create tasks provide support for different execution models
  - More flexible
  - Requires scheduling of task
  - Moves away from the original fork/join model of OpenMP construct

2. OpenMP Programming: Task

- Assumption here is that tasks are independent. Ex : tree traversal




```
void preorder(node *p) {
    process(p->data);
    if (p->left)
    #pragma omp task
        preorder(p->left);
    if (p->right)
    #pragma omp task
        preorder(p->right);
}
```

2. OpenMP programming - Tasks

Task – Technical Notion

- When a thread encounters a task construct, it may choose to execute the task immediately or defer its execution until a later time. If deferred, the task is placed in a conceptual pool of tasks associated with the current parallel region. All team threads will take tasks out of the pool and execute them until the pool is empty. A thread that executes a task might be different from the thread that originally encountered it.
- The code associated with a task construct will be executed only once. A task is **tied** if the code is executed by the same thread from beginning to end. Otherwise, the task is **untied** (the code can be executed by more than one thread).

2. OpenMP programming - Tasks

Types of Task

- **Undeferred:** the execution is not deferred with respect to its generating task region, and the generating task region is suspended until execution of the **undeferred** task is completed (such as the tasks created with the if clause)
- **Included:** execution is sequentially included in the generating task region (such as a result from a final clause)
- **Subtle difference:** for **undeferred** task, the generating task region is suspended until execution of the **undeferred** task is completed, even if the **undeferred** task is not executed immediately.
  - The **undeferred** task may be placed in the conceptual pool and executed at a later time by the encountering thread or by some other thread; in the meantime, the generating task is suspended. Once the execution of the **undeferred** task is completed, the generating task can resume.
- A merged task is a task whose data environment is the same as that of its generating task region.

2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

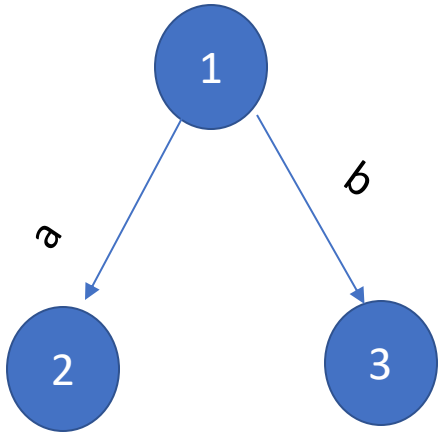
firstprivate(list)

shared(list)

priority(value)

Depend :

- It enforces additional constraints between tasks
- The list in depend contains storage locations (memory addresses) on which the dependency will be tracked
- The dependency is mentioned with **in, out,inout**




```
#pragma omp task shared(a,b) depend(out a,b)
task1(……)     //T1
#pragma omp shared(a) task depend(in a)
task2(……)    //T2
#pragma omp task shared(b) depend(in b)
task3(……)   //T3
```

T1 must be executed first. T2 and T3 can be executed in parallel

2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

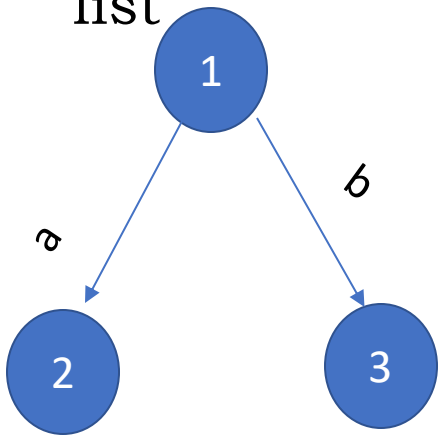
firstprivate(list)

shared(list)

priority(value)

Depend :

- In: The generated task will be dependent of all previously generated sibling tasks that reference at least one of the list items in an **out** or **inout** dependence type.
- Out and **inout**: The generated task will be dependent of all previously generated sibling tasks that reference at least one of the list items in an **in,out** or **inout** dependence type list




```
#pragma omp task shared(a,b) depend(out a,b)
task1(……)     //T1
#pragma omp shared(a) task depend(in a)
task2(……)    //T2
#pragma omp task shared(b) depend(in b)
task3(……)   //T3
```

T1 must be executed first. T2 and T3 can be executed in parallel

2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

If (expression):

- When the if expression argument evaluate to false, an undeferred task is generated, and the encountering thread must suspend the current task region, for which execution cannot be resumed until execution of the strucured block that is associated with the generated task is completed.
- The use of variable in an if clause expression of a task construct causes an implicit reference to the variable in all enclosing constructs

2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

If (expression):

```
#pragma omp task if(0) // This is undeferred
{
  #pragma omp task // This is a regular task
     for( i = 0; i < 3; i++ ) {
       #pragma omp task // This is a regular task
         bar();
     }
}
```

2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

final (expression):

- When the final clause argument is true
  - The generated task will be a final task
  - All tasks encountered during execution of a final task will generate included tasks
    - An included task is a task for which execution is sequentially included in the generating task region; that is undeferred and executed immediately by the encountering threads
- It is another user directed optimization
- Omp_in_final() returns true if the enclosing task region is final. Otherwise, it returns false.

2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

final (expression):

```
#pragma omp task final(1) // This is a regular task
{
  #pragma omp task // This is included
    for(i=0;i<3;i++){
      #pragma omp task // This is also included
        bar();
    }
}
```

2. OpenMP Programming: Task

```
#include <stdio.h>>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
  printf("A ");
  printf("Race ");
  printf("Car ");
    printf("\n");
return 0;
}
```

2. OpenMP Programming: Task

```
#include <stdio.h>>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
  printf("A ");
  printf("Race ");
  printf("Car ");
    printf("\n");
return 0;
}
```

- The output of the program is

A Race Car

2. OpenMP Programming: Task

```
#include <stdio.h>>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
#pragma omp parallel {
  printf("A ");
  printf("Race ");
  printf("Car ");
}
    printf("\n");
return 0;
}
```

- The output of the program is

A Race Car A Race Car A Race Car

(Assume three threads)

2. OpenMP Programming: Collapse

```
#include <stdio.h>>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
#pragma omp parallel {
#pragma omp single{
  printf("A ");
  printf("Race ");
  printf("Car ");
} }
    printf("\n");
return 0;
}
```

- The output of the program is

A Race Car

(Assume three threads)

2. OpenMP Programming: Task

```
#include <stdio.h>>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
#pragma omp parallel {
#pragma omp single{
  printf("A ");
#pragma omp task
   { printf("Race "); }
#pragma omp task
  { printf("Car "); }
} }
    printf("\n");
return 0;
}
```

- The output of the program is

A Race Car

Or

A Car Race

(Assume three threads)

2. OpenMP Programming: Task

```
#include <stdio.h>>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
#pragma omp parallel {
#pragma omp single{
  printf("A ");
#pragma omp task
   { printf("Race "); }
#pragma omp task
  { printf("Car "); }
printf("is fun to watch");
} }
    printf("\n");
return 0;
```

- The output of the program is

A is fun to watch Race Car

Or

A is fun to watch Car Race

(Assume two threads)

2. OpenMP Programming: Task

```
int main(void)
{
#pragma omp parallel {
#pragma omp single{
  printf("A ");
#pragma omp task
   { printf("Race "); }
#pragma omp task
  { printf("Car "); }
#pragma omp taskwait
printf("is fun to watch");
} }
    printf("\n");
return 0;
}
```

- The output of the program is

A Race Car is fun to watch

Or

A Car Race is fun to watch

(Assume two threads)

2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
Clauses
depend(list)
if(expression)
final(expression)
untied
mergeable
default(shared|firstprivate|none)
private(list)
firstprivate(list)
shared(list)
priority(value)
```

untied

- Different parts of the task can be executed by different threads. Implies the tasks will yield, allowing the executing thread to switch context and execute another task instead.
- If the task is tied, it is guaranteed that the same thread will execute all the parts of the task, even if the task execution has been temporarily suspended
- An **untied** task generator can be moved from thread to thread allowing the tasks to be generated by different entities

2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

meargeable

- A merged task is a task whose data environment is the same as that of its generating task region.
- When a mergeable clause is present on a task construct, then the implementation may choose to generate a merged task instead.
- If a merged task is generated, then the behavior is as though there was no task directive at all

2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

Default, private, firstprivate

- Default defines the data-sharing attributes of variables that are referenced
- **firstprivate:** each construct has a copy of the data item, and it is initialized from the upper construct before the call
- **shared:** All references to a list item within a task refer to the storage area of the original variable
- **private:** each task receive a new item

2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

Priority(n)

- Priority is a hint for the scheduler. A non-negative numerical value, that recommend a task with a high priority to be executed before a task with lower priority

Index

Reference

Text Books and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999
3. I.Foster, ”Designing and building parallel programs”, 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

Reference

Acknowledgements

1. Introduction to OpenMP https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf

2. Introduction to parallel programming for shared memory Machines https://www.youtube.com/watch?v=LL3TAHpxOig

3. OpenMP Application Program Interface Version 2.5 May 2005

4. OpenMP Application Program Interface Version 5.0 November 2018

5 Introduction to openmp tasks

http://www.icl.utk.edu/~luszczek/teaching/courses/fall2016/cosc462/pdf/W45L1%20-%20OpenMP%20Tasks.pdf

Thank You