National Institute of Technology Karnataka Surathkal
Department of Information Technology

IT 301 Parallel Computing

Shared Memory Programming Technique (3)

OpenMP :*parallel, for –reduction, master*

Dr. Geetha V

Assistant Professor

Dept of Information Technology

NITK Surathkal

Index

- OpenMP
  - Program Structure
  - Directives : master
  - Clauses
    - Reduction
    - lastprivate
- References

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().*

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:

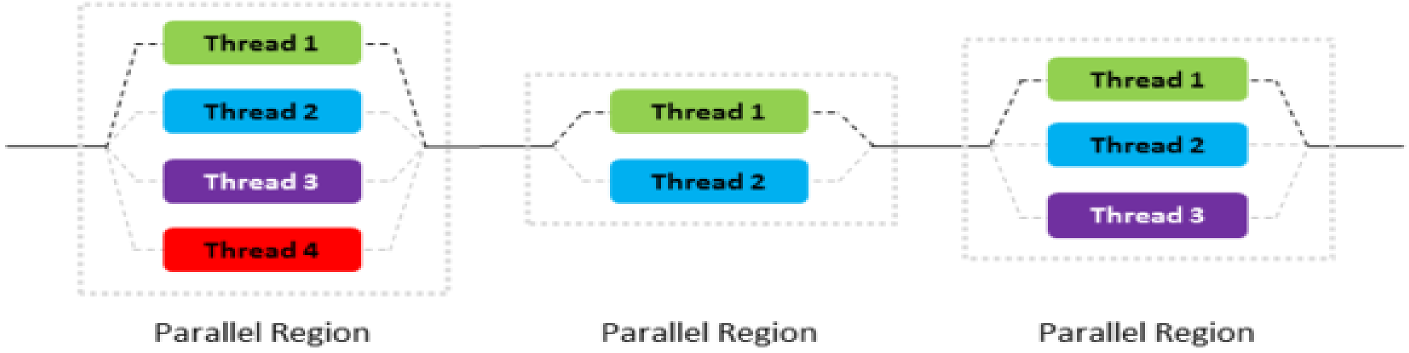
Implementation of parallel programs using OpenMP/MPI/CUDA.

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: Directives

#pragma omp parallel *[clause[,]clause…] new-line*

Structured-block

Clause: **if**(*scalar-expression*)

num_threads(*integer-expression*)

default(*shared*|*none*)

private(*list*)

firstprivate(*list)*

shared(*list)*

copyin(*list)*

reduction(*operator:list)*

2. OpenMP Programming: Clauses

#pragma omp parallel *[clause[,]clause...] new-line*

Structured-block

Clause: **if**(*scalar-expression*)

num_threads(*integer-expression*)

default(*shared|none*)

private(*list*)

firstprivate(*list*)

shared(*list*)

copyin(*list*)

reduction(*operator:list*)

Default(shared/none) , shared(list) Clause

- **if(scalar_expression):** if true execute in parallel
- **num_threads(int):** set number of threads
- **default (shared)** clause causes all variables referenced in the construct which have implicitly determined sharing attributes to be shared.
- **default(none)** clause requires that each variable which is referenced in the construct, and that does not have a predetermined sharing attribute, must have its sharing attribute explicitly determined by being listed in a data sharing attribute clause
- **shared(list)** : One or more list items must be shared among all the threads in a team.
- **private (list)** clause declares one or more list items must be private to a thread.
- **firstprivate (list)** clause declares one or more list items to be private to a thread and initializes each of them with that the corresponding original item has when the construct is encountered.

2. OpenMP Programming: Directives - Shared

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel  num_threads(4) shared(x) private(tid)
    {

    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    x=15;
    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    x=x+1;
    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
return 0;
}
```

```
x value ouside parallel:10

 1. Thread [2] value of x is 10

 2. Thread [2] value of x is 15

 3. Thread [2] value of x is 16

 1. Thread [3] value of x is 10

 2. Thread [3] value of x is 15

 3. Thread [3] value of x is 16

 1. Thread [1] value of x is 10

 2. Thread [1] value of x is 15

 3. Thread [1] value of x is 16

 1. Thread [0] value of x is 10

 2. Thread [0] value of x is 15

 3. Thread [0] value of x is 16
```

X is shared. X is assigned value 15 inside parallel region.
Each thread is assigning value as 15. And updating it with x=x+1; But update is not getting reflected in other threads.

2. OpenMP Programming: Directives - Shared

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel  num_threads(4) shared(x) private(tid)
    {

    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    //x=15;

    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    x=x+1;

    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
return 0;
}
```

```
x value ouside parallel:10

 1. Thread [1] value of x is 10

 2. Thread [1] value of x is 10

 3. Thread [1] value of x is 11

 1. Thread [3] value of x is 10

 2. Thread [3] value of x is 11

 3. Thread [3] value of x is 12

 1. Thread [0] value of x is 10

 2. Thread [0] value of x is 12

 3. Thread [0] value of x is 13

 1. Thread [2] value of x is 10

 2. Thread [2] value of x is 13

 3. Thread [2] value of x is 14
```

x is shared. No assignment in the parallel region.
Update is getting reflected in all other threads.
Synchronization is job of programmer.

2. OpenMP Programming: Directives - master

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel  num_threads(4) shared(x) private(tid)
    {

    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    #pragma omp master
    {
        x=15;
    }

    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    x=x+1;

    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
return 0;
}
```

```
x value ouside parallel:10

 1. Thread [0] value of x is 10

 2. Thread [0] value of x is 15

 3. Thread [0] value of x is 16

 1. Thread [3] value of x is 10

 2. Thread [3] value of x is 16

 3. Thread [3] value of x is 17

 1. Thread [1] value of x is 10

 2. Thread [1] value of x is 17

 3. Thread [1] value of x is 18

 1. Thread [2] value of x is 10

 2. Thread [2] value of x is 18

 3. Thread [2] value of x is 19
```

x is shared. Assignment statement is done only by master.

2. OpenMP Programming: Directives - master

The master construct specifies a structured block that is executed by the master thread of the team.

#pragma omp master *new-line*

Structured block

- A master region binds to the innermost enclosing parallel region.
- Only the master thread executes the structured block
- There is no implied barrier on entry or exit, for master construct. So other threads need not fork or join.

```
#pragma omp master
  {
       Structured block
}
```

Consider a program for adding sum of elements in an array.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int tid,p,a[50],sum[20],dsum,finalsum;
int i, low, high;
int n=20;

//initialise
for(i=0;i<20;i++)
{
  a[i]=i;
  dsum=dsum+i;
}
printf("\n 1. dsum=%d \n",dsum);
#pragma omp parallel  num_threads(4) default(shared) private(tid,low,high,i)
   {
   p=omp_get_num_threads();
   int tid=omp_get_thread_num();
   //assign the iterations to threads
   low=n*tid/p;
   high=n*(tid+1)/p;
   printf("\n 2. Thread [%d] low=%d high=%d \n",tid,low,high);

   //find partial sum
   sum[tid]=0;
   for(i=low;i<high;i++)
       sum[tid]=sum[tid]+a[i];

  printf("3: partial sum [%d] = %d \n",tid,sum[tid]);

} //close parallel
```

```c
finalsum=0;
   //add all partial sum
   for(i=0;i<p;i++)
      finalsum=finalsum+sum[i];

    printf("\n 4. finalsum= %d \n",finalsum);

    return 0;
}
```

The parallel region assigns iterations to each thread .
Partial sum is calculated in each thread
Partial sum is stored in an array
Master thread computes final sum

Consider a program for adding sum of elements in an array.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int tid,p,a[50],sum[20],dsum,finalsum;
int i, low, high;
int n=20;

//initialise
for(i=0;i<20;i++)
{
  a[i]=i;
  dsum=dsum+i;
}
printf("\n 1. dsum=%d \n",dsum);
#pragma omp parallel  num_threads(4) default(shared) private(tid,low,high,i)
   {
   p=omp_get_num_threads();
   int tid=omp_get_thread_num();
   //assign the iterations to threads
   low=n*tid/p;
   high=n*(tid+1)/p;
   printf("\n 2. Thread [%d] low=%d high=%d \n",tid,low,high);

   //find partial sum
   sum[tid]=0;
   for(i=low;i<high;i++)
       sum[tid]=sum[tid]+a[i];

  printf("3: partial sum [%d] = %d \n",tid,sum[tid]);

} //close parallel
```

```c
finalsum=0;
  //add all partial sum
  for(i=0;i<p;i++)
     finalsum=finalsum+sum[i];

   printf("\n 4. finalsum= %d \n",finalsum);

   return 0;
}
```

```
 1. dsum=190

 2. Thread [2] low=10 high=15

 2. Thread [3] low=15 high=20
3: partial sum [3] = 85
3: partial sum [2] = 60

 2. Thread [1] low=5 high=10
3: partial sum [1] = 35

 2. Thread [0] low=0 high=5
3: partial sum [0] = 10

 4. finalsum= 190

--------------------------------
Process exited after 0.05116 seconds
```

2. OpenMP Programming: Clauses

#pragma omp parallel *[clause[,]clause...] new-line*

Structured-block

Clause: **if**(*scalar-expression*)

num_threads(*integer-expression*)

default(*shared|none*)

private(*list*)

firstprivate(*list)*

shared(*list)*

copyin(*list)*

reduction(*operator:list)*

- Reduction(operator: list)
- The **reduction** clause specifies an operator and one or more list items.
- For each list item, a private copy is created on each thread, and is initialized appropriately for the operator.
- After the end of the region, the original list item is updated with the values of the private copies using the specified operator.
- Initialization value depend on data type of the **reduction** variable.

2. OpenMP Programming: Clauses

#pragma omp parallel *[clause[,]clause…] new-line*

Structured-block

Clause: **if**(*scalar-expression*)

num_threads(*integer-expression*)

default(*shared|none*)

private(*list*)

firstprivate(*list*)

shared(*list*)

copyin(*list*)

reduction(*operator:list*)

- Reduction(operator: list)
- Initialization value depend on data type of the reduction variable.

Operator	Initialization value
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

Consider a program for adding sum of elements in an array: with reduction

```c
#include <stdlib.h>
#include <omp.h>
int main (void) {
int tid,p,a[50],sum=0,dsum,finalsum;
int i, low, high;
int n=20;

//initialise
for(i=0;i<20;i++)
{
  a[i]=i;
  dsum=dsum+i;
}
printf("\n 1. dsum=%d \n",dsum);
#pragma omp parallel  num_threads(4) default(shared) private(tid,low,high,i) reduction(+:sum)
   {
   p=omp_get_num_threads();
   int tid=omp_get_thread_num();
   //assign the iterations to threads
   low=n*tid/p;
   high=n*(tid+1)/p;
   printf("\n 2. Thread [%d] low=%d high=%d \n",tid,low,high);

   //find partial sum
   for(i=low;i<high;i++)
      sum=sum+a[i];
} //close parallel

 printf("\n 4. finalsum= %d \n",sum);
   return 0;
}
```

```
 1. dsum=190

 2. Thread [2] low=10 high=15

 2. Thread [3] low=15 high=20
3: partial sum [3] = 85
3: partial sum [2] = 60

 2. Thread [1] low=5 high=10
3: partial sum [1] = 35

 2. Thread [0] low=0 high=5
3: partial sum [0] = 10

 4. finalsum= 190
```

2. OpenMP Programming: Clauses

#pragma omp parallel *[clause[,]clause…] new-line*

Structured-block

Clause: **if**(*scalar-expression*)

num_threads(*integer-expression*)

default(*shared|none*)

private(*list*)

firstprivate(*list)*

shared(*list)*

copyin(*list)*

reduction(*operator:list)*

- Reduction(operator: list)
- Used for some form of recurrence calculations
- The type of a list item that appears in a reduction clause must be valid for the **reduction** operator.
- Aggregate types(including arrays), pointers types and reference types may not appear in a **reduction** clause
- A variable must appear in a **reduction** clause must not be const-qualified
- The operator specified in a **reduction** clause cannot be overloaded with respect to the variables that appear in that clause.

2. OpenMP Programming: Clauses

#pragma omp for *[clause[,]clause…] new-line*

for-loops

Clause: **private(*list*)**

firstprivate(*list*)

lastprivate(*list*)

reduction(*operator:list*)

schedule(*kind[,chunk_size]*)

collapse(*n*)

ordered

nowait

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int n=20, dsum=0, tid,i,a[20],sum=0;

for(i=0;i<n;i++)
{
a[i]=i;
dsum=dsum+i;
}
printf("dsum=%d\n",dsum);
#pragma omp parallel  num_threads(4)
    {
    int tid=omp_get_thread_num();
    #pragma omp for private(i) schedule(static, 5) reduction(+:sum)
    for(i=0;i<n;i++)
     sum=sum+a[i];
}
    printf("\n sum= %d \n",sum);
return 0;
}
```

2. OpenMP Programming: Clauses

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int n=20, dsum=0, tid,i,a[20],sum=0;

for(i=0;i<n;i++)
{
a[i]=i;
dsum=dsum+i;
}
printf("dsum=%d\n",dsum);
#pragma omp parallel  num_threads(4)
    {
    int tid=omp_get_thread_num();
    #pragma omp for private(i) schedule(static, 5) reduction(+:sum)
    for(i=0;i<n;i++)
     sum=sum+a[i];
}
    printf("\n sum= %d \n",sum);
return 0;
}
```

```
dsum=190

 sum= 190

--------------------------------

Process exited after 0.02466 seconds
```

2. OpenMP Programming: Clauses

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int n=10, dsum=0, tid,i,a[10],sum=0, count=0;

for(i=0;i<n;i++)
{
a[i]=i;
}
#pragma omp parallel  num_threads(2) default(shared)
    {
    int tid=omp_get_thread_num();
    count=0;
    #pragma omp for schedule(static, 5) private(count)
    for(i=0;i<n;i++)
     {
     a[i]=a[i]+5;
     if(a[i]%2==0) count++;
     printf("tid[%d] a[%d]=%d count %d\n",tid,i,a[i],count);
}
}
return 0;
}
```

```
tid[0] a[0]=5 count 4199876
tid[0] a[1]=6 count 4199877
tid[0] a[2]=7 count 4199877
tid[0] a[3]=8 count 4199878
tid[0] a[4]=9 count 4199878
tid[1] a[5]=10 count 1790492427
tid[1] a[6]=11 count 1790492427
tid[1] a[7]=12 count 1790492428
tid[1] a[8]=13 count 1790492428
tid[1] a[9]=14 count 1790492429
```

2. OpenMP Programming: Clauses

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int n=10, dsum=0, tid,i,a[10],sum=0, count=0;

for(i=0;i<n;i++)
{
a[i]=i;
}
#pragma omp parallel  num_threads(2) default(shared)
    {
    int tid=omp_get_thread_num();
    count=0;
    #pragma omp for schedule(static, 5) firstprivate(count)
    for(i=0;i<n;i++)
     {
     a[i]=a[i]+5;
     if(a[i]%2==0) count++;
     printf("tid[%d] a[%d]=%d count %d\n",tid,i,a[i],count);
}
}
return 0;
}
```

```
tid[0] a[0]=5 count 0
tid[0] a[1]=6 count 1
tid[0] a[2]=7 count 1
tid[0] a[3]=8 count 2
tid[0] a[4]=9 count 2
tid[1] a[5]=10 count 1
tid[1] a[6]=11 count 1
tid[1] a[7]=12 count 2
tid[1] a[8]=13 count 2
tid[1] a[9]=14 count 3
```

2. OpenMP Programming: Clauses

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
int x;
x=100;
printf("X value before parallel region %d\n",x);
#pragma omp parallel for num_threads(2) private(x)
for(i=0;i<=10;i++){
x=x+i;
printf("Thread number: %d     x: %d\n",omp_get_thread_num(),x);
}
printf("x is %d\n", x);

return 0;
}
```

```
X value before parallel region 100
Thread number: 0     x: 8
Thread number: 0     x: 9
Thread number: 1     x: 2067342
Thread number: 1     x: 2067349
Thread number: 1     x: 2067357
Thread number: 1     x: 2067366
Thread number: 1     x: 2067376
Thread number: 0     x: 11
Thread number: 0     x: 14
Thread number: 0     x: 18
Thread number: 0     x: 23
x is 100
```

2. OpenMP Programming: Clauses

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
int x;
x=100;
printf("X value before parallel region %d\n",x);
#pragma omp parallel for num_threads(2) firstprivate(x)
for(i=0;i<=10;i++){
x=x+i;
printf("Thread number: %d     x: %d\n",omp_get_thread_num(),x);
}
printf("x is %d\n", x);

return 0;
}
```

```
X value before parallel region 100
Thread number: 0     x: 100
Thread number: 0     x: 101
Thread number: 0     x: 103
Thread number: 0     x: 106
Thread number: 0     x: 110
Thread number: 0     x: 115
Thread number: 1     x: 106
Thread number: 1     x: 113
Thread number: 1     x: 121
Thread number: 1     x: 130
Thread number: 1     x: 140
x is 100
```

2. OpenMP Programming: Clauses

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
int x;
x=100;
printf("X value before parallel region %d\n",x);
#pragma omp parallel for num_threads(2) firstprivate(x) lastprivate(x)
for(i=0;i<=10;i++){
x=x+i;
printf("Thread number: %d     x: %d\n",omp_get_thread_num(),x);
}
printf("x is %d\n", x);

return 0;
}
```

```
X value before parallel region 100
Thread number: 0     x: 100
Thread number: 1     x: 106
Thread number: 1     x: 113
Thread number: 0     x: 101
Thread number: 1     x: 121
Thread number: 1     x: 130
Thread number: 1     x: 140
Thread number: 0     x: 103
Thread number: 0     x: 106
Thread number: 0     x: 110
Thread number: 0     x: 115
x is 140
```

Index

- OpenMP
  - Program Structure
  - Directives : master
  - Clauses
    - Reduction
    - Lastprivate
- References

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

Thank You