Multicore Programming: Synchronization:Semaphores

Dhananjay Brahme
Parallelization and Optimization Group
TATA Consultancy Services, SahyadriPark Pune, India

April 30, 2013

 ©All rights reserved

Topics

1. Synchronization Problems

 ©All rights reserved

Topics

1. Synchronization Problems
2. Semaphore Definition

©All rights reserved

Topics

1. Synchronization Problems
2. Semaphore Definition
3. Exploration of Problem Solutions using Semaphores

Reference: The Little Book of Semaphores, Allen Downey Version 2.1.5

 ©All rights reserved

Synchrhonization Constraints: Serialization

Algorithm

Ram:
Eats breakfast
Writes Software module B
Signals Done
Eats Lunch

Algorithm

Laxman:
Eats breakfast
Tests Software module A
Eats Lunch
Receives Signal to Start
Tests Software module B

©All rights reserved

Syncrhonization Constraints: Mutual Exclusion

Algorithm

Ram:

Eats breakfast

Signals Begining Modify

Modifies Software module A

Signals Done Modifying

Algorithm

Laxman:

Eats breakfast

Signals Begining Modify

Modifies Software module A

Signals Done Modifying

©All rights reserved

Semaphore: Definition

Semaphore is an integer with the following differences:

1. Semaphore is created and initialized to any value.
2. When a thread decrements its value, if it is negative it waits, i.e., it gets blocked.
3. When a thread increments its value, if there are waiting threads, then one thread gets unblocked.

©All rights reserved

Serialization Problem:

A writes Software Module A

B tests Software Module A

©All rights reserved

Signaling with Semaphores:

Create done(0)

A writes Software Module A
done.signal

done.wait
B tests Software Module A

©All rights reserved

Rendezvous:

A writes Software Module A
A tests Software Module B

B writes Software Module B
B tests Software Module A

©All rights reserved

Rendezvous: Solution with Semaphores:

Thread 1:
Create doneA(0)
A writes Software Module A
doneA.signal
waitB.signal
A tests Software Module B

Thread 2:
Create doneB(0)
B writes Software Module B
doneB.signal
waitA.signal
B tests Software Module A

Potential Problems?

©All rights reserved

Mutual Exclusion:

Thread 1:
count = count++;

Thread 2:
count = count++;

©All rights reserved

Mutual Exclusion: Solution with Semaphore

Create update(1)

Thread 1:

```
update.wait()
count = count++;
update.signal()
```

Thread 2:

```
update.wait()
count = count++;
update.signal()
```


©All rights reserved

Barrier: Rendezvous With n threads

Thread i:

1. rendezvous
2. critical section

©All rights reserved

Barrier: Solution 1 with Semaphores

Thread 1: create b(-n)
Thread i:

1. b.signal
2. b.wait
3. barrier point

Problem ??

©All rights reserved

Barrier: Solution 1 with Semaphores

Thread 1: count = 0
create update_count(1)
create b(0)
if (count == n) b.signal
Thread i:

1. update_count.wait
2. count++;
3. update_count.signal
4. if (count == n) b.signal
5. b.wait
6. barrier point

Problem ??

Is there always a problem

©All rights reserved

Implement Barrier using Semaphore and OpenMP

1. Implement using “semaphore.h” and openMP “omp.h”.
2. *YOUR SOLUTION HERE.*

©All rights reserved

Barrier using semaphores

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <omp.h>

int main(int argc, char **argv){
    sem_t b;
    int count=0;
    sem_init(&b, 1, 0);

#pragma omp parallel num_threads(3)
  {
    sleep(omp_get_thread_num());
#pragma omp critical
    {
      count++;
    }
    if (count == 3){
      sem_post(&b);
      printf("thread=%d arrived at barrier\n", omp_get_thread_num());
    } else if (count < 3){
      printf("Wating In thread %d\n", omp_get_thread_num());
      sem_wait(&b);
      printf("Awakened thread=%d\n", omp_get_thread_num());
      sem_post(&b);
    }
  }
  return 0;
}
```


©All rights reserved

Multicore Programming

Parallelization and Optimization Group
TATA Consultancy Services, Sahyadri Park Pune, India

May 3, 2013

 ©All rights reserved

Agenda

1. Introduction to Multithreading
2. About Pthreads
3. Pthreads - An Example
4. Introduction to OpenMP
5. Programming Model
6. OpenMP Example
7. Data Parallelism
8. Task Parallelism
9. Synchronization Issues
10. Explicit Asynchronous/Synchronous Execution
11. Load balancing in Loops

 ©All rights reserved

Introduction to Multithreading

1. Multiple threads exists within the context of a single process.
2. Can share resources such as memory, not applicable with processes.
3. Single-core system: More Context switching time.
4. Multi-core system: Less switching, truly concurrent.

©All rights reserved

About Pthreads

1. Standard C library with functions for using threads.
2. Available across different platforms.

©All rights reserved

About Pthreads

1. Standard C library with functions for using threads.
2. Available across different platforms.

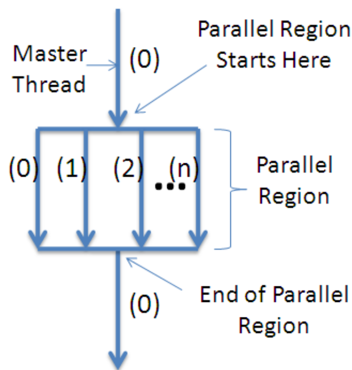


Figure: Creation of Multi-threads

 ©All rights reserved

Example : Vector Addition

In main():

```
pthread_t callThread[numT];

for(i=0; i<numT; i++)
    pthread_create(&callThread[i],NULL, vectAdd,(void*)i);

for(i=0; i<numT; i++)
    pthread_join(&callThread[i], NULL);
```


©All rights reserved

Example : Vector Addition

In vectAdd() :

```
start = tid * (N/numT);
end = (tid+1) * (N/numT);

for(i=start; i< end; i++)
    C[i]=  A[i] + B[i];
```

 ©All rights reserved

Compilation :

1. Include pthread.h in the main file

©All rights reserved

Compilation :

1. Include pthread.h in the main file
2. Compile program with

©All rights reserved

Compilation :

1. Include pthread.h in the main file
2. Compile program with -lpthread

©All rights reserved

Compilation :

1. Include pthread.h in the main file
2. Compile program with -lpthread
   - gcc -o test test.c -lpthread

©All rights reserved

About OpenMP

1. Abbreviation: **Open** specifications for **Multi-Processing**.
2. API to exhibit *multi-threaded shared memory parallelism*.
3. The API is specified for C/C++ and Fortran
4. Three distinct components. As of version 3.1:
    4.1 Compiler Directives (20)
    4.2 Runtime Library Routines (32)
    4.3 Environment Variables (9)

©All rights reserved

OpenMP Programming Model

```
#include <omp.h>
.
.

#pragma omp parallel
    {


        //Parallel Region



    }
```

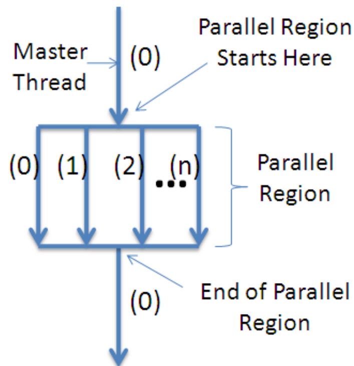


Figure: Creation of Multi-threads

©All rights reserved

OpenMP Example

```c
#include <stdio.h>
#include <omp.h>

int main()
{

#pragma omp parallel num_threads(4)

    {
        printf("Hello! My Thread Id is :: %d\n", omp_get_thread_num());
    }

    return 0;
}
```


©All rights reserved

OpenMP Example

```
#include <stdio.h>
#include <omp.h>

int main()
{

#pragma omp parallel num_threads(4)

    {
        printf("Hello! My Thread Id is :: %d\n", omp_get_thread_num());
    }

    return 0;
}
```

Compile (GNU C Compiler): gcc -fopenmp hello.c -o hello.out
Run: ./hello.out

©All rights reserved

OpenMP Example

Output:
Hello! My Thread Id is :: 0
Hello! My Thread Id is :: 3
Hello! My Thread Id is :: 1
Hello! My Thread Id is :: 2

 ©All rights reserved

Setting the Number of Threads

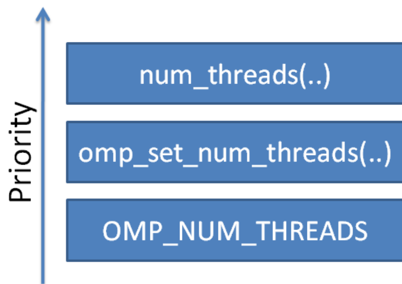


Figure: Ways to Set Threads

©All rights reserved

Bigger Data

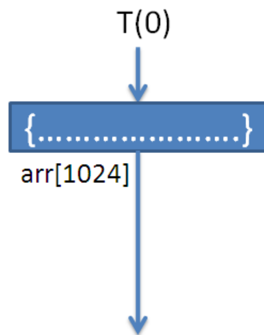


Figure: Single Thread Operation (Sequential)

Will take longer execution time.

©All rights reserved

Chunk your Data, Share work among threads

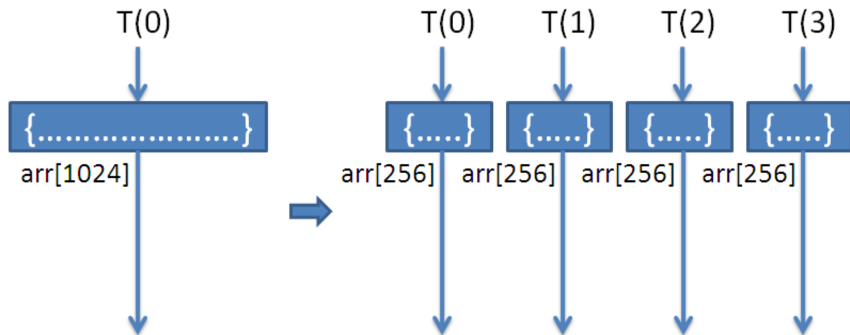


Figure: MultiThread Operation (Parallel)

Will take less execution time.

©All rights reserved

OpenMP For directive

```
#pragma omp parallel num_threads(4)
  #pragma omp for
    for(i=0; i<4; i++)
    {
        printf("Iteration %d, ThreadId %d\n", i, omp_get_thread_num());
    }
```


©All rights reserved

OpenMP For directive

```
#pragma omp parallel num_threads(4)
  #pragma omp for
    for(i=0; i<4; i++)
    {
        printf("Iteration %d, ThreadId %d\n", i, omp_get_thread_num());
    }
```

Output:

Iteration 0, ThreadId 0

Iteration 2, ThreadId 2

Iteration 3, ThreadId 3

Iteration 1, ThreadId 1

©All rights reserved

Shared Data

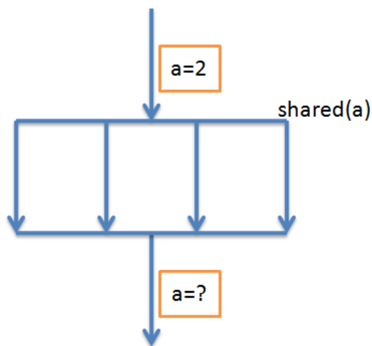


Figure: Globally accessible Data

Simultaneous Write Operations can do hazards.

©All rights reserved

Make your Data Private

Creates private copy of variable for each thread.

```
    int a = 2;

#pragma omp parallel private(a)
    {
        printf("Value of a :: %d\n", a);
    }
```

 ©All rights reserved

Make your Data Private

Creates private copy of variable for each thread.

```
    int a = 2;

#pragma omp parallel private(a)
    {
        printf("Value of a :: %d\n", a);
    }
```

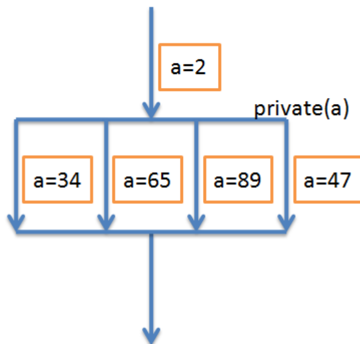


Figure: Private Clause

Problem: Variables uninitialized for each thread

©All rights reserved

Firstprivate Clause

Creates private copy of variable for each thread with automatic initialization.

```
    int a = 2;

#pragma omp parallel firstprivate(a)
    {
        printf("Value of a :: %d\n", a);
    }
```

 ©All rights reserved

Firstprivate Clause

Creates private copy of variable for each thread with automatic initialization.

```
int a = 2;

#pragma omp parallel firstprivate(a)
{
    printf("Value of a :: %d\n", a);
}
```

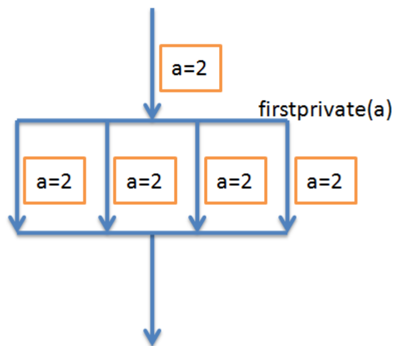


Figure: Firstprivate Clause

©All rights reserved

Lastprivate Clause

Private + Copy back value from last loop iteration to the original variable.

```
    int i = 0,a = 2;

#pragma omp parallel firstprivate(a) lastprivate(a)
#pragma omp for
        for(i=0; i<4; i++)
        {
            a += omp_get_thread_num();
        }

        printf("Value of a :: %d\n", a);
```

 ©All rights reserved

Lastprivate Clause

Private + Copy back value from last loop iteration to the original variable.

```
    int i = 0,a = 2;

#pragma omp parallel firstprivate(a)  lastprivate(a)
#pragma omp for
        for(i=0; i<4; i++)
        {
            a += omp_get_thread_num();
        }

        printf("Value of a :: %d\n", a);
```

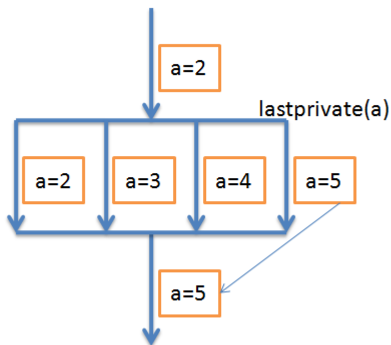


Figure: Lastprivate Clause

©All rights reserved

Reduction Clause

1. A private copy for each list variable is created for each thread.
2. Reduction variable is applied to all private copies of the shared variable.
3. Final result is written to the global shared variable.

```
int i, sum=0;
int a[2] = {1,1}, b[2] = {2,2};

#pragma omp parallel reduction(+:sum) num_threads(2)
#pragma omp for
    for(i=0; i<2; i++)
    {
        sum = a[i]+b[i];
    }

    printf("Sum :: %d\n", sum);
```

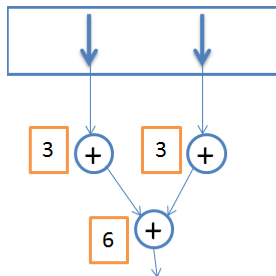


Figure: Reduction Clause

 ©All rights reserved

Master Directive

Code is to be executed by only Master thread.

```
int a[4], i=0;

#pragma omp parallel
{
    // Computation
    #pragma omp for
    for (i = 0; i < 4; i++)
    {
        a[i] = i * i;
    }

    // Print Results
    #pragma omp master
    {
        for (i = 0; i < 4; i++)
            printf("a[%d] = %d\n", i, a[i]);
    }
}
```


©All rights reserved

Master Directive

Code is to be executed by only Master thread.

```
int a[4], i=0;

#pragma omp parallel
{
    // Computation
    #pragma omp for
    for (i = 0; i < 4; i++)
    {
        a[i] = i * i;
    }

    // Print Results
    #pragma omp master
    {
        for (i = 0; i < 4; i++)
            printf("a[%d] = %d\n", i, a[i]);
    }
}
```

Output:

a[0] = 0

a[1] = 1

a[2] = 4

a[3] = 9

©All rights reserved

Single Directive

Code is to be executed by only one thread in the team.

```
int a[4], i=0;

#pragma omp parallel
{
    // Computation
    #pragma omp for
    for (i = 0; i < 4; i++)
    {
        a[i] = i * i;
    }

    // Print Results
    #pragma omp single
    {
        for (i = 0; i < 4; i++)
            printf("a[%d] = %d\n", i, a[i]);
    }
}
```

 ©All rights reserved

Single Directive

Code is to be executed by only one thread in the team.

```
int a[4], i=0;

#pragma omp parallel
{
    // Computation
    #pragma omp for
    for (i = 0; i < 4; i++)
    {
        a[i] = i * i;
    }

    // Print Results
    #pragma omp single
    {
        for (i = 0; i < 4; i++)
            printf("a[%d] = %d\n", i, a[i]);
    }
}
```

Output:
$a[0] = 0$
$a[1] = 1$
$a[2] = 4$
$a[3] = 9$

©All rights reserved

Independent tasks

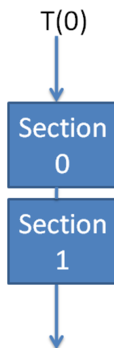


Figure: Single Thread Operation (Sequential)

Will take Longer Execution Time.

©All rights reserved

Independent Tasks can run in parallel

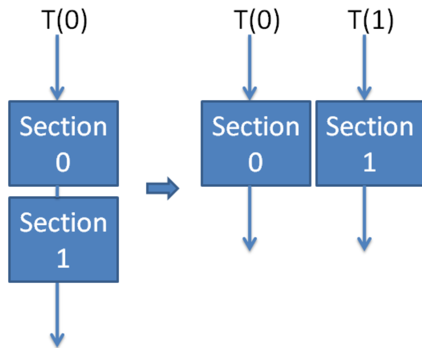


Figure: MultiThread Operation (Parallel)

Will take Less Execution Time.

©All rights reserved

OpenMP Sections Directive

```
#pragma omp parallel sections num_threads(2)
    {

#pragma omp section
        {
            printf("A. My ThreadId :: %d\n", omp_get_thread_num());
        }
#pragma omp section
        {
            printf("B. My ThreadId :: %d\n", omp_get_thread_num());
        }

    }
```

Output:
A. My ThreadId :: 0
B. My ThreadId :: 1

©All rights reserved

Synchronization Issues

What will be the output.. ??

```
    int max = 11;
    int a[2] = {22,33};
    int tid = 0;

#pragma omp parallel
    {
        tid = omp_get_thread_num();

        if(a[tid] > max)
            max = a[tid];
    }

    printf("Value of Max :: %d\n", max);
```


©All rights reserved

Synchronization Issues

What will be the output.. ??

```
int max = 11;
int a[2] = {22,33};
int tid = 0;

#pragma omp parallel
{
    tid = omp_get_thread_num();

    if(a[tid] > max)
        max = a[tid];
}

printf("Value of Max :: %d\n", max);
```

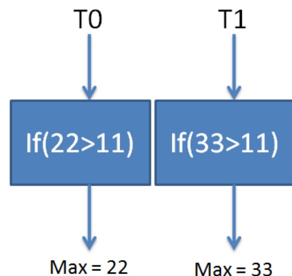


Figure: Race Condition

Output:
Value of Max :: 33 **OR** Value of Max :: 22

©All rights reserved

Use Critical Directive

```
    int max = 11;
    int a[2] = {22,33};
    int tid = 0;

#pragma omp parallel
    {

#pragma omp critical
        {
            tid = omp_get_thread_num();
            if(a[tid] > max)
                max = a[tid];
        }

    }

    printf("Value of Max :: %d\n", max);
```


©All rights reserved

Use Critical Directive

```
    int max = 11;
    int a[2] = {22,33};
    int tid = 0;

#pragma omp parallel
    {

#pragma omp critical
        {
            tid = omp_get_thread_num();
            if(a[tid] > max)
                max = a[tid];
        }

    }

    printf("Value of Max :: %d\n", max);
```

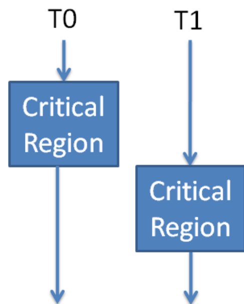


Figure: Critical Region

Output: Value of Max :: 33

 ©All rights reserved

Synchronization Issue

What will be the output.. ??

```
#pragma omp parallel
{
    #pragma omp for // Computation 1
    for (i = 0; i < 4; i++)
        a[i] = i * i;

    #pragma omp master //Print Intermediate Results.
        for (i = 0; i < 4; i++)
            printf("Partial a[%d] = %d\n", i, a[i]);

    #pragma omp for //Computation 2
    for (i = 0; i < 4; i++)
        a[i] += i;
}

for (i = 0; i < 4; i++)
    printf("Final a[%d] = %d\n", i, a[i]);
```

 ©All rights reserved

Synchronization Issue

Expected Output:
Partial a[0] = 0
Partial a[1] = 1
Partial a[2] = 4
Partial a[3] = 9
Final a[0] = 0
Final a[1] = 2
Final a[2] = 6
Final a[3] = 12

©All rights reserved

Synchronization Issue

Expected Output:
Partial a[0] = 0
Partial a[1] = 1
Partial a[2] = 4
Partial a[3] = 9
Final a[0] = 0
Final a[1] = 2
Final a[2] = 6
Final a[3] = 12

Actual Output:
Partial a[0] = 0
Partial a[1] = 2
Partial a[2] = 6
Partial a[3] = 12
Final a[0] = 0
Final a[1] = 2
Final a[2] = 6
Final a[3] = 12

©All rights reserved

Explicit Synchronous Execution

Use OpenMP Barrier Directive:

```
#pragma omp parallel
{
    #pragma omp for // Computation 1
    for (i = 0; i < 4; i++)
        a[i] = i * i;

    #pragma omp master //Print Intermediate Results.
        for (i = 0; i < 4; i++)
            printf("Partial a[%d] = %d\n", i, a[i]);

    #pragma omp barrier

    #pragma omp for //Computation 2
    for (i = 0; i < 4; i++)
        a[i] += i;
}

for (i = 0; i < 4; i++)
    printf("Final a[%d] = %d\n", i, a[i]);
```

 ©All rights reserved

Explicit Synchronous Execution

Expected Output:
Partial a[0] = 0
Partial a[1] = 1
Partial a[2] = 4
Partial a[3] = 9
Final a[0] = 0
Final a[1] = 2
Final a[2] = 6
Final a[3] = 12

©All rights reserved

Explicit Synchronous Execution

Expected Output:
Partial a[0] = 0
Partial a[1] = 1
Partial a[2] = 4
Partial a[3] = 9
Final a[0] = 0
Final a[1] = 2
Final a[2] = 6
Final a[3] = 12

Actual Output:
Partial a[0] = 0
Partial a[1] = 1
Partial a[2] = 4
Partial a[3] = 9
Final a[0] = 0
Final a[1] = 2
Final a[2] = 6
Final a[3] = 12

©All rights reserved

Explicit Asynchronous Execution

Is someone sitting Ideal.. ??

```
#pragma omp parallel private(i)
 {
```

```
#pragma omp for
      for(i=0; i<N; i++)
          a[i] += b[i];
```

```
#pragma omp for
      for(i=0; i<N; i++)
          c[i] += d[i];
```

```
 }
```

Independent Calculations

Figure: Two Independent Parallel Regions

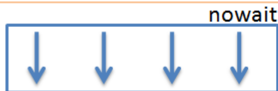
©All rights reserved

Explicit Asynchronous Execution

Can execute asynchronously.

```
#pragma omp parallel private(i)
{
```

```
#pragma omp for nowait
    for(i=0; i<N; i++)
        a[i] += b[i];
```




```
#pragma omp for nowait
    for(i=0; i<N; i++)
        c[i] += d[i];
```

```
}
```

Figure: Use "nowait"

 ©All rights reserved

Load balancing in Loops

1. Division of loop iterations among the threads in the team.
2. By default, schedule is implementation dependent.
3. Use OpenMP schedule clause with for construct.

schedule (static [, chunk]): Static load balancing by iterations.

```
#pragma omp parallel for schedule(static)
#pragma omp parallel for schedule(static, 2)
```

schedule (dynamic [, chunk]): Dynamic load balancing.

```
#pragma omp parallel for schedule(dynamic)
#pragma omp parallel for schedule(dynamic, 2)
```

 ©All rights reserved

Thank You

©All rights reserved