CSCE 5160 Parallel Processing

Homework #2: (Due sept 22)

Write OpenMP and MPI programs for Dot product (multiplying a matrix with a vector resulting in a vector) A*b = y
Make sure each thread/task is assigned more than one array element

```
for (i=0; i<n; i++)
  {y[i] = 0.0;
   for (j=0; j<n; j++)
              y[i] = y[i] + A[i][j] * b[j];
   }
```

Review
- Threads
- OpenMP pragma parallel for and parallel sections
  - private and shared
  - critical sections, reduce
  - Barriers and no wait

CSCE 5160 Parallel Processing

OMP inserts barriers at parallel constructs (for or sections).
But you can explicitly state that you don't want barriers
The Nowait clause

```
#prgam omp for nowait
  for (i=0; i<n; i++)
          {….}
```

One of more powerful concept in OMP is on how you want to distribute work among threads, known as the schedule construct.

You can either assign

- fixed amounts, determined statically using chunk size (static)
- fixed amounts, assigned dynamically as threads complete work (dynamic)
- similar to dynamic but varying number of iterations assigned (guided)
- dependent on runtime environmental variable OMP_SCHEDULE (runtime)

CSCE 5160 Parallel Processing

Consider the following example

```
#pragma omp parallel for default (none) schedule (runtime) private (j) shared (n)

    for (i=0; i<n; i++)
      {
        printf ("Iteration %d executed by theard %d\n", i,
                    omp_get_thread_num(j));
       for (j=0; j<i; j++)
               system ("sleep 1");
      }
```

Note that we are parallelizing outer loop – but because of the inner loop the amount of work for each ith iteration is different

If we use static or even dynamic with fixed chunks, load is not balanced

CSCE 5160 Parallel Processing

Let us now discuss the shared memory programming in more general terms
When we are dealing with shared data, we need to use critical sections which rely on
Mutual exclusion – one thread at a time in the critical section

Mutual Exclusion Example--- pseudo syntax

```
thread Producer
Buf: Character;

Begin
 Loop
     Lock Mutex (B);
     Check_Free_Buf;
     If No_Buffer   Then
        UnLock Mutex (B);
     Else
        Read_Input_Data (Buf);
              Create_Full_Buf(Buf);
        UnLock Mutex (B);
     End If;
   Exit When No_more_data;
 End Loop;
End Producer;
```

```
thread Consumer
 F_Buf : Character;

Begin
   Loop
      Lock Mutex (B);
       Check_Full_Buf;
       If No_Full_Buf Then
              UnLock Mutex (B);
      Else
              Print_Data(Buf);
              Release_Empty_Buf(Buf);
              UnLock Mutex(B);
     End If;
     Exit When ?
   End Loop;
End Consumer;
```

CSCE 5160 Parallel Processing

OpenMP achieves this using critical construct

```
#prgrma omp parallel shared (n, a, sum) private (TID, sumLocal)

    {   TID = op_get_thread_num();
        sumLocal = 0;
        #pragma omp for
                for (i=0; i<n; o++)  sumLocal += a[i];
        #pragma omp critical (update_sum)
                {sum += sumLocal;
                 printf("TID = %d; sumLocal=%d sum=%d\n", Tid, sumLocal,sum);
                }
    }
```

A few things to note here: The first pragma can be combined with the #pragma_omp for but we wanted the TID to be declared as a shared variable

TID is thread number similar to RANK in MPI
We can give a name to the critical section.

CSCE 5160 Parallel Processing

See how you can use the critical for producer and consumer threads

You need to view producer and consumer as "sections" to be executed

Then create **critical sections** for checking to see if free buffer or full buffer is available.

Instead of critical sections – which can include several statements, OpenMP allows protected single statements using atomic construct

```
int ic, i, n
ic=0;
#pragma omp parallel shared(n,ic) private(i)
        for (i=0; i++; i<n
        {       #pragma omp atomic          /* only one statement in atomic
                ic =ic+1;  }
        printf ("counter = %d\n", ic);
```

Note: you should be careful with atomic statements.

The variable protected by atomic should always be updated using atomic only

CSCE 5160 Parallel Processing

Critical sections and atomic statements are implemented using mutual exclusions locks
Can be very expensive in term of overhead in most thread implementations.

Better performance if locks are implemented using hardware level instructions such as
- test and set,
- fetch and add (Concurrent Write using "sum of all values to be written"),
- or transactional memories (can be in hardware or software)

Dealing with Locks directly is difficult to deal
- You need to remember to initialize, and delete at the end
- Avoid locking the variable multiple times
- avoid deadlocks, and be careful about "nesting" of locks

Hence the creation of critical and atomic features of OpenMP

OpenMP also allows you to create explicit locks and barriers

CSCE 5160 Parallel Processing

We need to declare lock variables using omp_loc_t

For example
omp_loc_t my_lock;

Then you should initialize the locks using
omp_init_lock (*my_lock);
Lock (acquire) the variable using
omp_set_lock (*my_lock);
Or omp_test_lock(*my_lock);

Release (unlock) using

omp_unset_lock(*my_lock);

Delete or the lock variable using
omp_destroy_lock(*my_lock);

```
….
If (omp_test_lock(*my_lock)
            --- critical section
Else
            -- try again or do
            -- something else
```

If you use set_lock you will not return until the lock is available

CSCE 5160 Parallel Processing

An explicit barrier can be placed using
#pragma omp barrier

A couple more useful OMP constructs

Single construct indicates that a piece of code should be executed only once
Example

```
#pragma omp parallel shared (a,b) private (i)
{ #pragma omp single
    {  a= 10;                                   /* executed only once

       printf("Single construct executed by thread %d\n", omp_get_thread_num());

    }                               /* barrier is automatically inserted here
    #pragma omp for
            for (i=0; i<n; i++) b[i] = a;
}
```

CSCE 5160 Parallel Processing

Master construct designates one threads as the master
and Master performs the single constructs

Example

```
#pragma omp parallel shared (a,b) private (i)
{ #pragma omp master
    {  a= 10;                                    /* executed only once

       printf("Master construct executed by thread %d\n", omp_get_thread_num());

    }
   #pragma omp barrier                           /*not automatically inserted

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

CSCE 5160 Parallel Processing

We have seen how to use critical sections, mutual exclusion, atomic statements.
Let us explore synchronization issues in shared memory systems further

Using Mutual Exclusion locks

```
void Producer (){
int Buf;

 loop {
  pthread_lock (&Mutex_B);
      Check_Free_Buf;
      if No_Buffer   Then
          pthread_UnLock (&Mutex_B);
      else   {
           Read_Input_Data (Buf);
            Create_Full_Buf(Buf);
           pthread_UnLock (&Mutex_B);
               }
    exit When No_more_data;
                   }
}
```

```
void Consumer {
int  F_Buf;

 loop {
   pthread_lock (&Mutex_B);
       Check_Full_Buf;
       if No_Full_Buf Then
           pthread_UnLock (&Mutex_B);
      else{
              Print_Data(Buf);
              Release_Empty_Buf(Buf);
             pthread_UnLock (&Mutex_B);
              }
exit When ?
        }
 }
```

CSCE 5160 Parallel Processing

Other way of synchronizing access to Data.

Ada's Rendezvous

```
Procedure XYZ Is

task Producer  is
     entry Send_Data
          (Ch: out Character);
end Producer ;

task body Producer is

Buf: Character;

Begin
  Loop
     Consumer.Free_Buffer (Buf);

     Read_Input_Data (Buf);


     Accept Send_Data
          ( Ch: out Character) do
          Ch : Buf;
     end Send_Data;

   Exit When No_more_data;

  End Loop;

End Producer;
```

```
task  Consumer is
        entry Free_Buffer
             (Buf: out Character);
 end Consumer;

task body Consumer is

 F_Buf : Character;

 Begin
  Loop
     Accept Free_Buffer
            (Buf : out Character) do
          Buf := F_Buf;
     end Free_Buffer;


  Producer.Send_Data(F_Buf);


     Print_Data(F_Buf);
     Exit When ?
      -- need to eventually terminate
  End Loop;

 End Consumer;
```

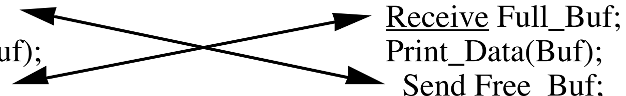
```
Begin    ---- the body XYZ
         Null;
 End XYZ;
```

CSCE 5160 Parallel Processing

Other way of synchronizing access to Data.

Message Passing

```
Procedure XYZ Is

task body Producer is

Buf: Character;

Begin
  Loop
        _ Receive Free_Buf;
         Read_Input_Data (Buf);
         Send Full_Buf;
         Exit When No_more_data;
   End Loop;
End Producer;

task body Consumer is

F_Buf : Character;

Begin
   Loop
       Receive Full_Buf;
       Print_Data(Buf);
     _  Send Free_Buf;
       Exit When ?
     End Loop;
   End Consumer;
```



CSCE 5160 Parallel Processing

Other way of synchronizing access to Data.

Using Monitors (like Java synchronized methods)

```
Package Body Monitor is
Procedure Request_Free_Buffer
Request_Full_Buffer
        (Buf : Out Character);
Begin
        If No Free_Buf Then
                Wait On Free_Buf;
        Else -- return an empty buffer
        End if;
End  Request_Free_Buffer;


Procedure Release_Free_Buffer
Release_Full_Buffer
        (Buf : In Character);
Begin
        -- Save the Free buffer

        Signal Free_Buf;
End Release_Free_Buffer;
```

```
Procedure

        (Buf : Out Character);
Begin
If No_Full_Buf Then
Wait on  Full_Buffer;
Else -- return a full buffer
End If;
End  Request_Full_Buffer;


Procedure

        (Buf : In Character);
Begin
--- Copy Buf into a
--local variable
    Signal Full_Buffer;
End Release_Full_Buffer;
```

```
task body Producer is
Buf: Character;

Begin
 Loop
    Request_Free_Buffer (Buf):
        Read_Input_Data (Buf);
    Release_Full_Buffer (Buf):
   Exit When No_more_data;
  End Loop;
End Producer;

task body Consumer is
F_Buf : Character;
Begin
Loop
  Request_Full_Buffer (Buf):
        Print_Data(Buf);
 Release_Free_Buffer (Buf);
  Exit When ?
End Loop;
End Consumer
```

CSCE 5160 Parallel Processing

Note – to implement monitors, we used waiting for something to happen
We can use condition variables with wait and signal in Pthreads

For example, a thread can sleep until some condition is met -- say, when the value of a variable exceeds some predefined value.

Say when there are no new items produced by the producer, the consumer thread can sleep until a new item is produced.

We can view this somewhat similar to Unix signals. The main difference being, a thread waiting for a condition is expressing a willingness to handle the condition, while a Unix signal interrupts the process and forces handling of the signal.
OpenMP does not provide this capability

Pthreads provide greater control for you.

A brief introduction to Pthreads

CSCE 5160 Parallel Processing

Creating A Thread.

An Example.

```
#include <pthread.h>
------
main()
{  pthread_t   thread1;
 pthread_create(&thread1,  NULL, (void *) funct_1 , (void *) &r1);
  ……..
  pthread_join(thread1, NULL);        /* WHY
-----
}
```

Note if main (or parent) thread returns before other threads complete, they will be aborted.

So do not return -- use Pthread_exit(); In this case, the parent lingers until all created threads complete.

CSCE 5160 Parallel Processing

Termination Of A Thread

A thread terminates normally after completing the execution of the start_routine. You can also explicitly terminate a thread using either the Pthread_exit or Pthread_Cancel

void **pthread_exit** (void *status);

This function terminates the thread that is making the call, and returns the value passed via the status to any waiting threads. Note that the variable containing the value to be returned should not be an automatic variable, since the value may be needed even after the thread quits.

int **pthread_cancel** (pthread_t thread);

This function cancels (or kills) a thread specified by the thread id.
The cancellation may not take effect immediately because

holding locks
performing clean-up
the specified thread set state to
PTHREAD_CANCEL_DEFERRED

A thread can also disable cancellation.

CSCE 5160 Parallel Processing

An example

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

void fibonacci (void *arg) {  //argument must be of void type
   int i1 = 0, i2 = 1, m;
   m = (int) arg;            //recover the desired type
   printf("%d  %d  ", i1, i2);
   while (i1 <= m && i2 <= m){
      printf("%d  ", i1 = i1 + i2);
      printf("%d  ", i2 = i2 + i1);
   }
   printf("\n");
}

main()
{

   int n = 500;
   pthread_t thread;

   if ( pthread_create(&thread, NULL, (void *)fibonacci, (void *) n ) != 0 ) {
   // default attrributes(2nd param) set, one void argument(4th param) passed
      printf("Failed to create pthread!\n");
   }

   // do other things ...

}
```