Electrical and Computer Engineering C97-20

SPECIAL TOPICS IN ECE

INTRODUCTION TO PARALLEL COMPUTING

Lecture 7

Shared Memory Programming - IV

Outline

- Programming with PTHREADS
- Example parallel computation: Matrix vector multiplication
- Programming with parallelization directives
- Example parallel computation: LU Factorization
- READING:
  - POSIX PTHREADS Programming Manual.
  - B. Bauer, "Practical Parallel Programming", chap 7, 12,

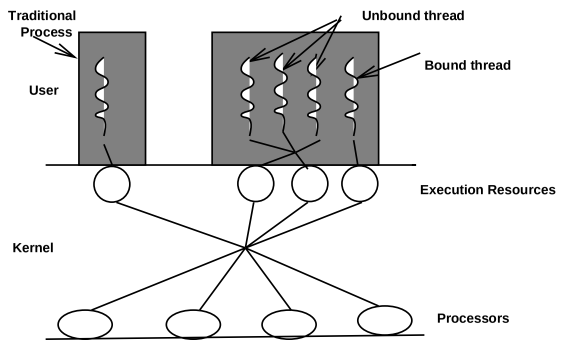
Introduction to Multithreaded Programming

- All programmers are used to programming a single thread of control
- A thread is a single sequence of execution steps performed by a program.
- A traditional UNIX process is a program with a single thread of control; it has sole possession of its address space.
- In traditional symmetric multiprocessing operating systems such as IRIX, when a user makes call to *m-fork()*, a new process is created.
- Such processes can share data through the *mmap()* UNIX call, or through the use of *shared* directives.
- Multithreaded libraries allow multiple threads to share an address space.
- Cost of creating a thread is very small
- Hence threads are called "lightweight processes"

Introduction to PTHREADS

- Each parallel computer vendor has its own way of supporting shared memory parallel programming
  - Sequent , Encore, SGI, SUN, HP, IBM, Convex
- IEEE had a POSIX 1003 group that defined a UNFORM interface to multithreaded programming
- This is called PTHREADS, and is similar to Solaris Threads from Sun

General Architecture of Threads Interface



Synchronization Primitives

- PTHREADS provide a variety of synchronization facilities for threads to cooperate in accessing shared resources.
  - Mutual exclusion locks (mutex locks)
  - Condition variables
  - Counting semaphores
  - Multiple readers, single writer locks

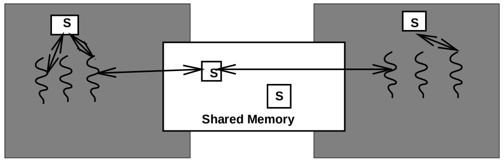
Introduction to PTHREADS Programming

- A multithreaded process is not a thread of control itself. It contains one or more threads of control.
- Some system calls affect the process as a whole
  - Example: if one thread calls the EXIT() call, all the threads in the process are destroyed.
- Most process resources are equally accessible to all the threads in the process.
- All process virtual memory is shared; a change in shared data by one thread is available to the other threads in the process.
- Each thread sees the same open files. If two threads read the same file sequentially, they advance through the file as though one thread were reading it.

Per-Thread Resources

- Each thread has:
  - A thread identifier
  - A set of registers (including program counter and stack pointer)
  - The signal mask
  - Execution priority
  - Stack
  - Thread-specific data

Use of Synchonization Variables in Shared Memory



- Synchronization variables S used by various threads in two different processes.

Thread Creation

```
int pthread_create(pthread_t *thread,
        const pthread_attr_t *attr = NULL,
        void *(*entry) (void *), void *arg);
```

- The function creates and starts a new thread
- The new thread executes the function specified in the call with the optional arguments.

Other Thread Control

- To determine the thread ID of the calling thread

  `pthread_t pthread_self(void);`

- To terminate a calling thread

  `void pthread_exit(void *status = NULL);`

Use of Mutex Locks

- The following function acquires the lock or blocks the calling thread if the lock is already held.

  `int pthread_mutex_lock( pthread_mutex_t *mutex);`

- The following function enables asynchronous polling of lock. It acquires the lock if free, and does not block (returns error) if the lock is held.

  `int pthread_mutex_trylock( pthread_mutex_t *mutex);`

- The following function unlocks the mutex lock. If other threads are blocked waiting for the lock, thread at head of queue is unblocked.

  `int pthread_mutex_unlock( pthread_mutex_t *mutex);`

Example of Use of Locks

```
mutex_t count_mutex;
int count;

increment_count()
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
}

int get_count()
{
    int c;

    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return(c);
}
```


Condition Variables

- Condition Variables are used to wait until a particular condition is true.
- A condition variable must be used in conjunction with a mutex lock.
- The following function blocks the calling thread until the condition is signaled. It atomically releases the associated mutex lock before blocking, and atomically reacquires it before returning.

  `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
- The following function signals one of the threads blocked in cond_wait().

  `int pthread_cond_signal( pthread_cond_t *cond);`

Condition Variable Usage

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned int count;

decrement_count();
{
   pthread_mutex_lock(&count_lock);
   while (count == 0)
       pthread_cond_wait(&count_nonzero, &count_lock);
   count = count - 1;
   pthread_mutex_unlock(&count_lock);
}
increment_count();
{
   pthread_mutex_lock(&count_lock);
   if (count == 0)
       pthread_cond_signal(&count_nonzero);
   count = count + 1;
   pthread_mutex_unlock(&count_lock);
}
```


Semaphores

- A semaphore is a non-negative integer count that is incremented and decremented.
- It is not as efficient as a mutex lock. They need not be acquired and released by the same thread, so they may be used in asynchronous event notification.
- The following function blocks the thread until the semaphore becomes greater than zero, then decrements it.

  `pthread_sema_wait(pthread_sema_t *sem);`
- The following function increments the semaphore, potentially unblocking a waiting thread.

  `pthread_sema_post(pthread_sema_t *sem);`

Use of Semaphores (Producer-Consumer)

```
int rdprt = 0;  /* pointers to a circular buffer */
int wrptr = 0;  /* of size BUFSIZE */
data_t buf[BUFSIZE];
pthread_sema_t sem;

THREAD 1:

while (work_to_do) {
    buf[wrptr] = produce();
    wrptr = (wrptr + 1) % BUFSIZE;
    pthread_sema_post(&sem);
}

THREAD 2:

while (work_to_do) {
    pthread_sema_wait(&sem);
    consume(buf[rdptr]);
    rdptr = (rdptr + 1) % BUFSIZE;
}
```


Example: Matrix Vector Multiplication

```
float a[max][max], b[max], c[max];
int m,n;
pthread_mutex_t vec_lock;

main()

{
   int i,j,nproc;
   void readdata(),matvec(),printdata();

   /* input m and n */

   printf("Enter number of rows and columns");
   scanf("%d %d",&m,&n);
   printf("Enter number of processors\n");
   scanf("%d",&nproc);
   /* read data */
   readdata();
```


Shared Memory Matrix-Vector Multiplication

```
    /* set desired level of concurrency */
    pthread_setconcurrency(&nprocs);

   /* create parallel threads */
   for (i=0; i < nprocs; i++)
       pthread_create(NULL, NULL, matvec, NULL);

   /* print output */
   printdata();
}
```


Shared Memory Matrix Vector Multiplication

```
void matvec()
{
   int i,j,nprocs,myid;
   float tmp[max];
   for (i=0; i < m; i++) {
      tmp[i] = 0;
      c[i] = 0;
   }
   myid = pthread_self();
   for (j = myid; j < n; j = j + nprocs) {
      for (i=0; i < m; i++)
         tmp[i] = tmp[i] + a[i][j] * b[j];
   }
   pthread_lock(&vec_lock);
   for (i=0; i < m; i++)
        c[i] = c[i] + tmp[i];
   pthread_unlock(&vec_lock);
}
```

Programming Environment on the SGI Challenge

- We will look at parallelization directives on the SGI Challenge systems, other vendors have similar directives, e.g. Convex Exemplar, Cray Fortran.
- PowerFORTRAN programming - explicit parallel directives DOACROSS
- PowerFORTRAN Analyzer - automatic parallelization
- Pro MPF - A Visual interface to Power FORTRAN
- Pro Dev Workshop - A suite of tools for debugging and analyzing programs

Parallelization Directives

- Directive used before a loop tells compiler how to parallelize the code in the loop:

```
C$DOACROSS [options]
  do i=1,n
    ...
  enddo
```

```
#pragma parallel
[options]
  #pragma pfor iterate(i=0;n;1)
  [options]
    for (i=0;i<n;i++) {
      ...
    }
```


How to Program with PTHREADS

- To program a parallel application with PTHREADS, need to add this statement to the source file:

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

- Compile a C program using

```
% cc -lpthread input.c
```


Parallelization Directives

- Looked at parallelizing loops for various commonly occuring code fragments
- Created parallel program using:
  - Creation and management of processes
  - Implementation of scheduling scheme
- Observations:
  - Procedure tedious and routine
  - Making changes can be difficult – scheduling scheme for instance
- Difficult part of parallel programming is enabling parallelization – can use a compiler for all the "grunt" work
- Directives are used to inform the compiler about parallelism

C$DOACROSS Options

- **mp_schedtype=type**:
  - Sets the scheduling scheme for the loop
  - Types can be:
    - *simple* - static block
    - *interleaved* - static interleaved
    - *dynamic* - self scheduling
    - *gss* - guided self scheduling
    - *runtime* - one of the above selected at runtime by examining a environment variable (MP_SCHEDTYPE)
- **chunk=num**:
  - Sets the chunksize for dynamic and interleaved scheduling schemes

#pragma parallel Options

- **#pragma shared(list)**: same as the **share(list)** option
- **#pragma local(list)**: same as the **local(list)** option
- **#pragma if(expression)**: same as the **if(expression)** option
- **#pragma byvalue(list)**:
  - Variables in list are treated as read only variables
  - The compiler makes copies of these variables for each process
- **#pragma numthreads(num)**:
  - Sets the number of processes on which the loop is executed to num
  - Default value is the number of processors available

C$DOACROSS Options

- **local(list)**:
  - Variables in the list are local to each process
  - The value of the local variables is undefined beyond the loop
- **lastlocal(list)**:
  - Variables in the list are local to each process
  - The values of these local variables beyond the loop is guaranteed to be that obtained in the last iteration
  - All loop indices default to lastlocal
- **share(list)**:
  - Variables in the list are shared among processes
  - All variables except the loop indices default to shared if not explicitly declared otherwise

C$DOACROSS Options

- **reduction(list)**:
  - Variables in the list are presumed involved in a reduction operation
  - Only single elements are allowed; arrays are not permitted. However, single array elements may be included.
- **if(expression)**:
  - Provides a run-time selection between parallel and serial execution of the loop
  - If the expression evaluates to $.true.$, the loop executes in parallel

#pragma pfor Options

- **iterate(var=start;num_iters;step)**:
  - Assigns the index variable and sets the initial value, total number of iterations and stepsize
- **schedtype(type)**: same as the **mp_schedtype=type** option
- **chunksize(num)**: same as the **chunk=num** option

Local Variables

```
do i=1,n
  x=a(i)**3
  b(i)=x*b(i)
enddo


C$DOACROSS local(x,i),
C$& share(a,b,n)

  do i=1,n
    x=a(i)**3
    b(i)=x*b(i)
  enddo
```

```
for (i=0;i<n;i++) {
  x=a[i]**3;
  b[i]=x*b[i];
}

#pragma parallel local (x,i)
#pragma shared(a,b)
#pragma byvalue(n)
  {
  #pragma pfor iterate(i=0;n;1)
  for (i=0;i<n;i++) {
    x=a[i]**3;
    b[i]=x*b[i];
  }
  }
```


Local Variables

```
do i=1,n
  x=a(i)**3
  b(i)=x*b(i)
enddo
val=foo(x)


C$DOACROSS lastlocal(x),
C$& local (i),
C$& share(a,b,n)

  do i=1,n
    x=a(i)**3
    b(i)=x*b(i)
  enddo

  val=foo(x)
```

```
for (i=0;i<n;i++) {
  x=a[i]**3;
  b[i]=x*b[i];
}
val=foo(x);

#pragma parallel local (local_x,i)
#pragma shared(a,b,x)
#pragma byvalue(n)
  {
  #pragma pfor iterate(i=0;n;1)
  for (i=0;i<n;i++) {
    local_x=a[i]**3;
    b[i]=local_x*b[i];
    if (i==n-1) x=local_x;
  }
  }
val=foo(x);
```


Loop-Carried Values

```
indx=0
do i=1,n
  indx=indx+i
  a(i)=b(indx)
enddo


C$DOACROSS local(indx,i),
C$& share(a,b)

  do i=1,n
    indx=(i*(i+1))/2
    a(i)=b(indx)
  enddo
```

```
indx=0;
for (i=0;i<n;i++) {
  indx+=i;
  a[i]=b[indx];
}

#pragma parallel local(indx,i)
#pragma shared(a,b)
  {
  #pragma pfor iterate(i=0;n;1)
  #pragma schedtype(simple)

    for (i=0;i<n;i++) {
      indx=(i*(i+1))/2;
      a[i]=b[indx];
    }
  }
```

Sum Reduction

```
total=0.0
do i=1,n
  total=total+a(i)
enddo
```

```
total=0.0
C$DOACROSS local(i),
C$& share(a),
C$& reduction(total)

  do i=1,n
    total=total+a(i)
  enddo
```

```
total=0.0
for(i=0;i<n;i++) {
  total+=a[i];
}

total=0.0;
#pragma parallel local(sub_total,i)
#pragma shared(a,total)
  {
  sub_total=0.0;
  #pragma pfor iterate(i=0;n;1)
  for (i=0;i<n;i++)
    {
    sub_total+=a[i];
    }
  #pragma critical
    {
    total+=sub_total;
    }
  }
```

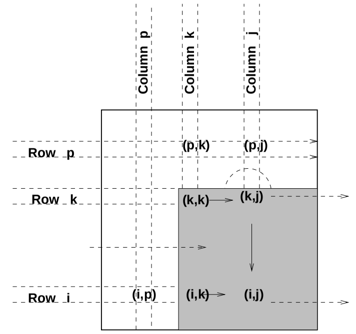

LU Factorization

- Given an n X n matrix A, decompose it as

$$A = L.U$$

- where L is a lower triangular matrix, and U is an upper triangular matrix.

Illustration of LU Factorization



Main Steps

Initialize u(k,j) = a(k,j)
Update u(k,j) = u(k,j)-l(k,p)*u(p,j)

Initialize l(i,k) = a(i,k)
Update l(i,k) = l(i,k)-l(i,p)*u(p,k)

Normalize l(i,k) = l(i,k)/u(k,k)

LU Decomposition: Serial Program

```
do k=1,n
  do j=1,k-1
    u(k,j)=0
    l(j,k)=0
  enddo
  l(k,k)=1
  do j=k,n
    u(k,j)=a(k,j)
    do p=1,k-1
      u(k,j)=u(k,j)-l(k,p)*u(p,j)
    enddo
  enddo
  if (k.lt.n) then
    do i=k+1,n
      l(i,k)=a(i,k)
      do p=1,k-1
        l(i,k)=l(i,k)-l(i,p)*u(p,k)
      enddo
      l(i,k)=l(i,k)/u(k,k)
    enddo
  endif
enddo
```

```
for (k=0;k<n;k++) {
  for (j=0;j<k;j++) {
      u[k][j]=0;
      l[j][k]=0;
  }
  l[k][k]=1;
  for (j=k;j<n;j++) {
    u[k][j]=a[k][j];
    for (p=0;p<=k-1;p++) {
      u[k][j]-=l[k][p]*u[p][j];
    }
  }
  if (k<n-1) {
    for (i=k+1;i<n;i++) {
      l[i][k]=a[i][k];
      for (p=0;p<=k-1;p++) {
        l[i][k]-=l[i][p]*u[p][k];
      }
      l[i][k]=l[i][k]/u[k][k];
    }
  }
}
```

LU Decomposition: Parallel Program

```
C$mp_schedtype=dynamic
C$doacross local(j), share(l,u,k)
    do j=1,k-1
      u(k,j)=0
      l(j,k)=0
    enddo
    l(k,k)=1
C$doacross local(j,p), share(l,u,a,k,n)
    do j=k,n
      u(k,j)=a(k,j)
      do p=1,k-1
        u(k,j)=u(k,j)-l(k,p)*u(p,j)
      enddo
    enddo
C$doacross local(i,p), share(l,u,a,k,n)
    do i=k+1,n
      l(i,k)=a(i,k)
      do p=1,k-1
        l(i,k)=l(i,k)-l(i,p)*u(p,k)
      enddo
      l(i,k)=l(i,k)/u(k,k)
    enddo
```

```
#pragma parallel local(i,j,p)
#pragma byvalue(k,n) shared(l,u,a)
  #pragma one processor
    for (j=0;j<k;j++)
        u[k][j]=l[j][k]=0.0;
    l[k][k]=1.0;
  #pragma pfor iterate(j=k;n-k;1)
  #pragma schedtype(dynamic)
    for (j=k;j<n;j++) {
      u[k][j]=a[k][j];
      for (p=0;p<=k-1;p++)
        u[k][j]-=l[k][p]*u[p][j];
    }
  #pragma synchronize
  #pragma pfor iterate(i=k+1;n-k-1;1)
  #pragma schedtype(dynamic)
    for (i=k+1;i<n;i++) {
      l[i][k]=a[i][k];
      for (p=0;p<=k-1;p++)
        l[i][k]-=l[i][p]*u[p][k];
      l[i][k]=l[i][k]/u[k][k];
    }
```


Summary

- Programming with PTHREADS
- Example parallel computation: Matrix vector multiplication
- Programming with parallelization directives
- Example parallel computation: LU Factorization
- NEXT LECTURE: Distributed Memory Parallel Architectures.
- READING: V. Kumar et al, "Introduction to Parallel Computing," Chapter 2.