

# CSCE 5160 Parallel Processing

**Homework #1: 2.2, 2.3, 2.12 2.13**  
**Due Sept 15, 2009**

Review

MPI

Rank, size and MPI communication groups (MPI\_COMM\_WORLD)

MPI\_Send, MPI\_Recv

MPI\_Bcast

**MPI\_Scatter** and **MPI\_Gather**

**MPI\_Reduce** and **MPI\_Allreduce**

Example program –serial version first

float **dot\_product** (float x[], float y[], int n)

```
{ int i;  
float sum = 0.0;  
for (i=0; i< n; i++) sum = sum + x[i]*y[i];  
return sum; }
```

# CSCE 5160 Parallel Processing

Let us assume  $p < n$  processes.

Each process computes a sum of  $n/p$  elements. Then we combine all these sums. Consider the following code fragment.

```
float parallel_version ( float local_x[], local_y[], int my_n)
{
    float local_sum;
    float total_sum = 0.0;
    float dot_product (float x[], float y[], int n);
    local_sum = dot_product (local_x, local_y, my_n);

    MPI_Reduce(&local_sum, &total_sum, 1, MPI_FLOAT, MPI_SUM,
              0, MPI_COMM_WORLD);
    return total_sum; }
```

# CSCE 5160 Parallel Processing

Use Scatter to distribute elements of X and Y elements

```
...    my_n = n/p;
        MPI_Scatter (&x, /* data source
                    my_n, /* count of items to be sent
                    MPI_FLOAT, /* data type sent
                    &local_x, /* storage for received data
                    my_n, /* count of items to be received
                    MPI_FLOAT, /*data type received
                    root, /* sender id
                    MPI_COMM_WORLD);

....    /* similarly, scatter Y elements.
```

All processes execute the MPI\_Reduce, however, only the process with rank=0 will accumulate the total\_sum value.

All other nodes will only have total\_sum=0.0

We can also use All\_reduce

# CSCE 5160 Parallel Processing

Another Example

Sieve of Eratosthenes: find all prime numbers up to a specified integer  $n$

Think of the numbers  $2..n$  stored in an array.

$k=2$ ;

    repeat

        mark all multiples of  $k$  between  $k^2$  and  $n$

        find the smallest unmarked number greater than  $k$  and set to that value as  $k$

    until  $k^2 > n$

How do we parallelize this?

Distribute the numbers to processors and repeat the algorithm in parallel

We can send the numbers to processors either cyclically or in blocks

Cyclical is better for load balancing purposes

# CSCE 5160 Parallel Processing

Need to balance load

if  $n$  is not a multiple of  $p$ , we cannot distribute equal number of values

The one way to do this is

$$r = n \bmod p$$

For the first  $r$  processors ( $0..r-1$ ) distribute rounded-up ( $n/p$ ) and the remaining  $p-r$  processors receive rounded-down ( $n/p$ )

The next problem is how do we know the indexes of the array elements assigned to a processor  $j$

The first element =  $j * \text{rounded-down}(n/p) + \min(j,r)$

The last element is one less than the first number assigned to  $j+1$

# CSCE 5160 Parallel Processing

## Communication

For each loop iteration, we can perform marking in parallel  
but we need to determine the next k collectively  
we can do gather to collect the k values from each processor  
and find the min of these  
or **we can use reduce** at process 0  
then broadcast the next k  
or we can do all reduce

Can you write an MPI program for Sieve of Eratosthenes?

Handout contains the code

# CSCE 5160 Parallel Processing

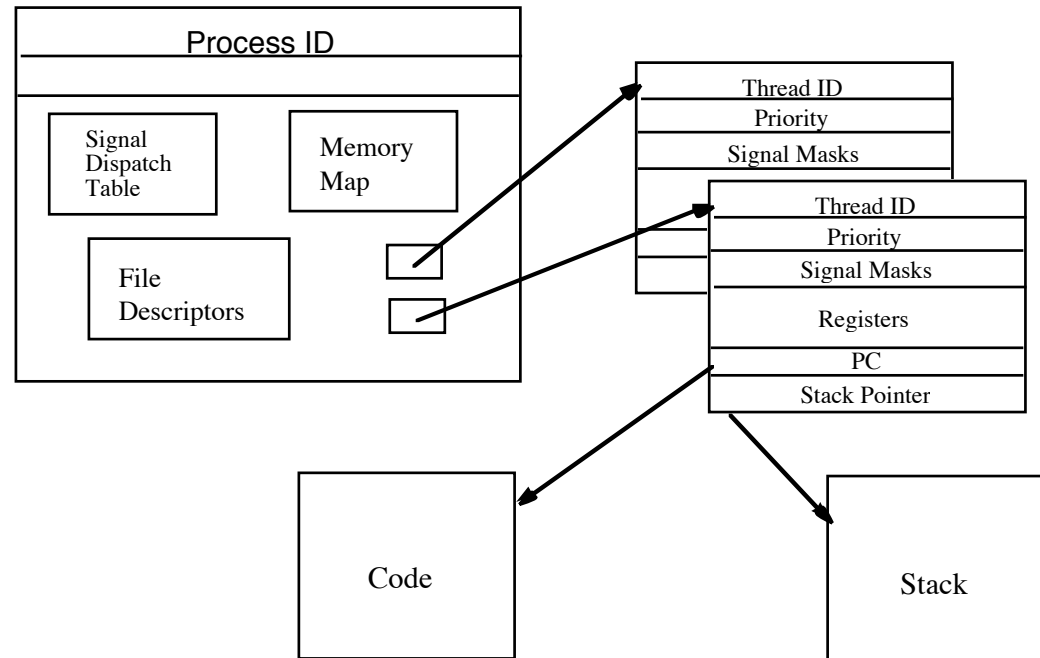
Let us now see how to program for shared memory system.  
We will use multithreading. We can use Pthreads or OpenMP

## Threads vs Processes (or Tasks)

Each thread will have a smaller “context” associated with it.

Context switching between threads is more efficient

Multiple Hardware contexts makes it even better



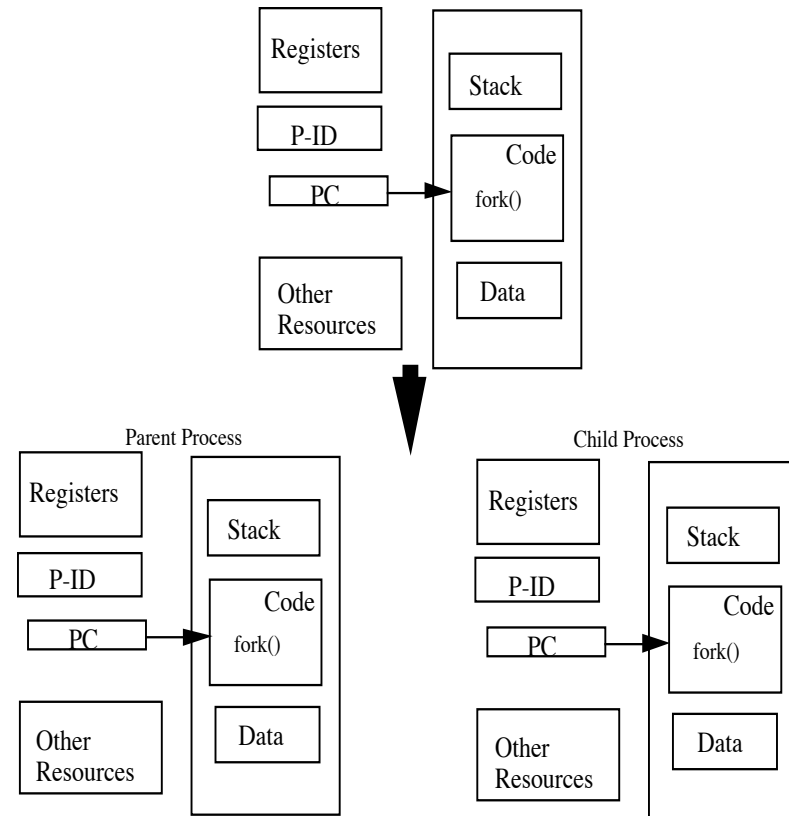
# CSCE 5160 Parallel Processing

## Threads vs Processes (or Tasks)

The fork command creates a child process that is identical to its parent process, but the child will be given a separate Process Id.

Since the child looks identical to the Parent, we need to include checks to see if the process is the parent or the child, and describe which function the child executes and which function the parent executes.

Fork returns different “return” values to the parent and the child -- and this distinguishes them.



# CSCE 5160 Parallel Processing

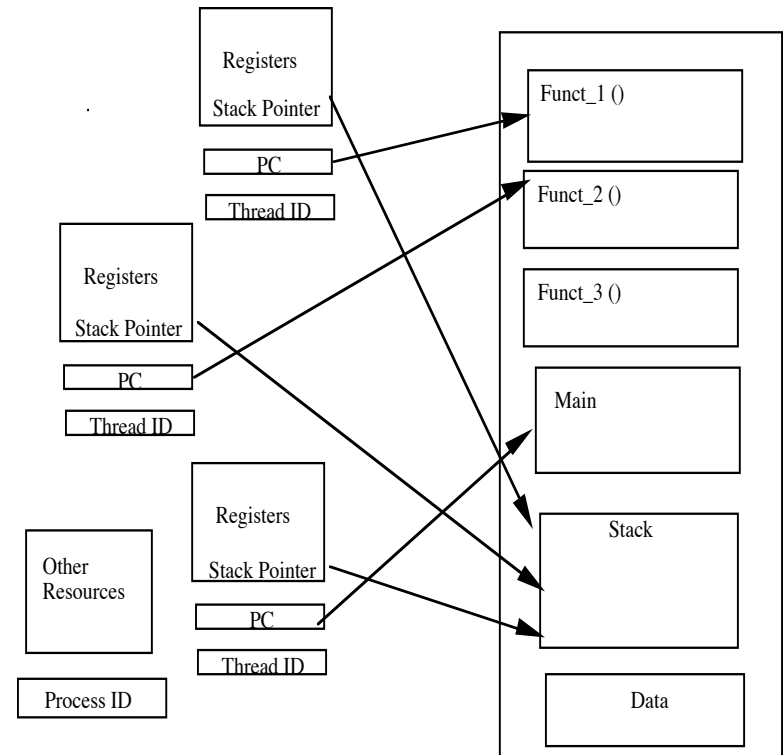
## Threads vs Processes (or Tasks)

Threads are light weight -- less expensive to create

When a thread is created, the process is not replicated -- created thread knows the function it executes.

Some cases, the threads can be created in user space -- no system calls

Communication and synchronization between threads of the same process can be done by sharing variables -- not so easily with processes



# CSCE 5160 Parallel Processing

## Differences between message passing and shared memory programming styles

Consider an application like Jacobi linear equation solver

A set of n equations containing n unknown variables:

$$\begin{aligned} a_{n-1,0} x_0 + a_{n-1,1} x_1 + a_{n-1,2} x_2 + \dots + a_{n-1,n-1} x_{n-2} &= b_{n-1} \\ a_{n-2,0} x_0 + a_{n-2,1} x_1 + a_{n-2,2} x_2 + \dots + a_{n-2,n-1} x_{n-2} &= b_{n-2} \\ \dots\dots\dots \\ a_{1,0} x_0 + a_{1,1} x_1 + a_{1,2} x_2 + \dots + a_{1,n-1} x_{n-2} &= b_1 \\ a_{0,0} x_0 + a_{0,1} x_1 + a_{0,2} x_2 + \dots + a_{0,n-1} x_{n-2} &= b_0 \end{aligned}$$

This can also be written as  $A \cdot X = B$

Where A is a nxn matrix, X and B are 1xn vectors

Let us consider the ith equation:

$$a_{i,0} x_0 + a_{i,1} x_1 + a_{i,2} x_2 + \dots + a_{i,i} x_i + \dots + a_{i,n-1} x_{n-2} = b_i$$

This can be written as

$$x_i = \frac{1}{a_{i,i}} \left[ \sum_{j \neq i}^n a_{i,j} x_j + b_i \right]$$

# CSCE 5160 Parallel Processing

Note that this expression shows each unknown as a function of other unknowns. So, in order to solve this numerically, we will devise an iterative solution. That is we will start with some assumed values for each  $x_i$ , and then update it using corrections. For example, we can initially set  $x_i = b_i$ . Then we will update each  $x_i$  using the above equation.

Consider a simple program to do this:

```
for (i =0; i < n; i ++ ) x[i] = b[i];

for (repeat = 0; repeat < limit; repeat++)
{
    for (i =0; i < n; i ++ ) {
        sum = 0;
        for (j =0; j < n; j ++ ) if ( i !=j) sum = sum + a[i][j]*x[j];
        new_x[i] = (b[i] - sum)/a[i][ i];
    }
    for (i =0; i < n; i ++ ) x[i] = new_x[i];
}
```

# CSCE 5160 Parallel Processing

How do we code this in a shared memory system (say using Pthreads)?

We will assume that we create a separate thread to compute each  $x[i]$ .

What we have to do is coordinate the shared data in array  $x$ .

Two steps that we must be careful about:

The new value of  $x[i]$  **cannot be updated** until all threads have used old values

New values of  $x[i]$  **cannot be read** by threads until all  $x[i]$ 's are updated

So,

After each thread computes its  $new\_x[i]$ ,

it must wait for all threads to complete computing their  $new\_x$ 's

Then each thread must update the  $x[i]$  using the  $new\_x[i]$

and

must wait until all threads completed the updates

before reading  $x[i]$  values

# CSCE 5160 Parallel Processing

How do we code this in a message passing (say using MPI)?

Again, let us assume that each  $x[i]$  is computed by a different process.

After each thread computes a new  $x[i]$  value, it must broadcast its data to all other processes or threads.

Each thread must also receive messages from all other threads.

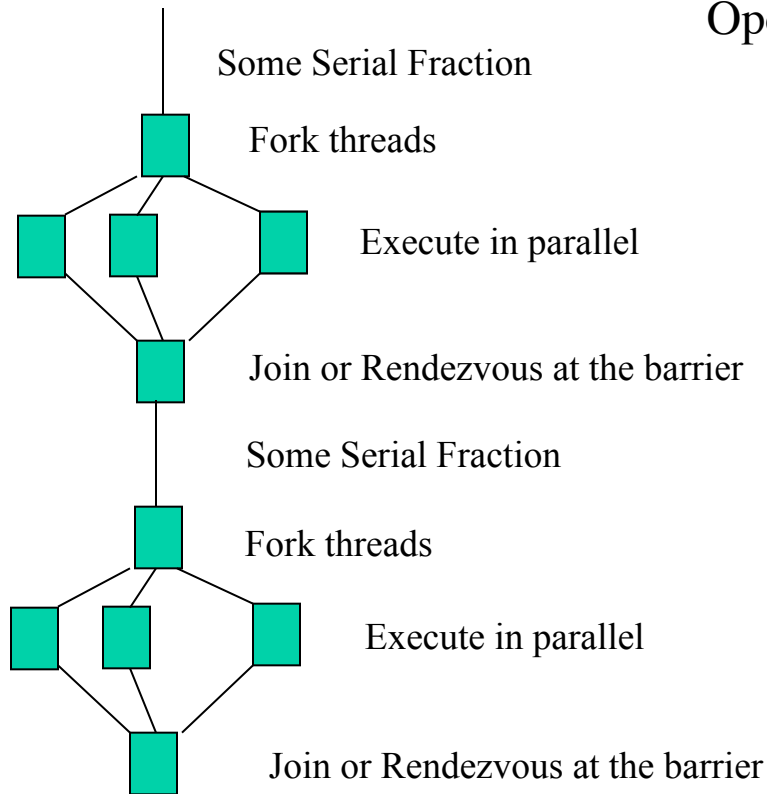
Need to decide on order for the messages to be broadcast and receive

This type of coordination (either using messages or thread) is called **barrier synchronization**

**(also known as fork and join)**

# CSCE 5160 Parallel Processing

Open MP uses “fork-join” model of computation



Parallel for loops lead to forking.  
And the end of the loop means “join”

If you use Pthreads, or other multithreading programming languages, you will have greater control on spawning, and joining

# CSCE 5160 Parallel Processing

In addition, in multithreading (i.e., OpenMP, Pthreads), when we want to control access to shared data

We will use **mutual exclusion** (or locks)

-- only one thread at a time is allowed access to shared data

OpenMP is based on Pthreads

```
for (i = 0; i < n; i++) {  
  #pragma omp parallel for  
    for (j = 0; j < n; j++) {  
      C[i,j] = 0;  
      for (k = 0; k < n; k++) C[i,j] = C[i,j] + A[i,k]*B[k,j];  
    }  
}
```

Automatic loop level parallelism works only for “canonical loops”

Limited tests (<, <=, >, >=, =)

Increment must be a constant integer

# CSCE 5160 Parallel Processing

Of course we can use this type of parallelism only if the computations are independent  
results of one iteration are not used by another (loop carried dependency)  
shared variables are handled using mutual exclusion

## Private and shared variables

private variables = a separate memory location for each thread  
shared = same address for all threads

OpenMP default is shared

To create private use `private(<variable-list>)`

```
#pragma omp parallel for private (j) (not needed for i)  
    for (i=0; i< BLOCK_SIZE(id,p,n); i++)  
        for(j=0; j<n;j++)  
            a[i][j] = MIN(a[i][j], a[i][k]+t,mp[j]);
```

# CSCE 5160 Parallel Processing

A few more constructs

`firstprivate` clause

All private variables “inherit” an initial value

Consider the following loop

```
x[0] = some_complex_function();
for (i=0;i<n; i++) {
  for (j=1; j<4; j++)
    x[j] = g(I, x[j=1]);
  answer[i] = x[i]- x[3];
}
```

If we want to parallelize the OUTER loop, we want `x[j]` to be private to each iteration/thread, but we do not want to initialize `x[0]` in each thread.

# CSCE 5160 Parallel Processing

```
x[0] = some_complex_function();  
##pragma omp parallel for private (j) firstprivate (x)  
for (i=0;i<n; i++) {  
    for (j=1; j<4; j++) x[j] = g(I, x[j=1]);  
    answer[i] = x[i]- x[3];  
}
```

Likewise, if we want the final value(s) of all parallel iterations to be shared, we can use **lastprivate** clause

For example consider

```
for (i=0; i<n; i++) {  
    x[0] =1.0;  
    for (j=1; j<4; j++) x[j] = x[j-1]*(i+1);  
    sum_of_powers[i] = x[0]+x[1]+x[2]+x[3];  
}  
n_cubed =x[3];
```

# CSCE 5160 Parallel Processing

```
#pragma omp parallel for private(j) lastprivate(x)
for (i=0; i<n; i++) {
    x[0] =1.0;
    for (j=1; j<4; j++) x[j] = x[j-1]*(i+1);
    sum_of_powers[i] = x[0]+x[1]+x[2]+x[3];
}
n_cubed =x[3];
```

Now the final x[3] is not private but shared value

# CSCE 5160 Parallel Processing

We need to create “critical sections” so that updates to shared data is correctly operated  
critical sections imply mutual exclusion – only one thread can be  
executing the code inside the critical section

Consider the following example for computing  $\pi$

```
area =0.0;
for (i=0; i<n; i++){
    x= (i+0.5)/n;
    area = area+4.0/(1.0+x*x);
}
pi = area/n;
```

If we parallelize the loop, area is a shared variable, and we should only update it  
one thread at a time.

# CSCE 5160 Parallel Processing

Consider the following example for computing  $\pi$

```
area =0.0;
#pragma omp parallel for private (x)

for (i=0; i<n; i++){
    x= (i+0.5)/n;
    #pragma omp critical
        area = area+4.0/(1.0+x*x);
    }
pi = area/n;
```

Of course we could also use MPI reduction like operations for this

# CSCE 5160 Parallel Processing

Consider the following example for computing  $\pi$

```
area =0.0;
#pragma omp parallel for private (x) reduction (+:area)

for (i=0; i<n; i++){
    x= (i+0.5)/n;
    area = area+4.0/(1.0+x*x);
}

pi = area/n;
```

Of course we could also use MPI like reduction for this

# CSCE 5160 Parallel Processing

In addition to parallelism at loop level, OpenMP allows other types of parallelism consider creating task level parallelism using SECTIONS

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {...some work}
    #pragma omp section
    {...some other work}
    #pragma omp section
    {...entirely different work}

  } /* end of sections block
} /* end of parallel block
```

# 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

```
#pragma 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**)