Electrical Engineering and Computer Science

EECS 358 - INTRODUCTION TO PARALLEL COMPUTING

Lecture 6

# Shared Memory Programming - III

## Outline

- Example parallel computation: PI
- Example parallel computation: matrix-vector multiplication
- Example parallel computation: matrix-matrix multiplication

## Example Application 1: Computation of Pi

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

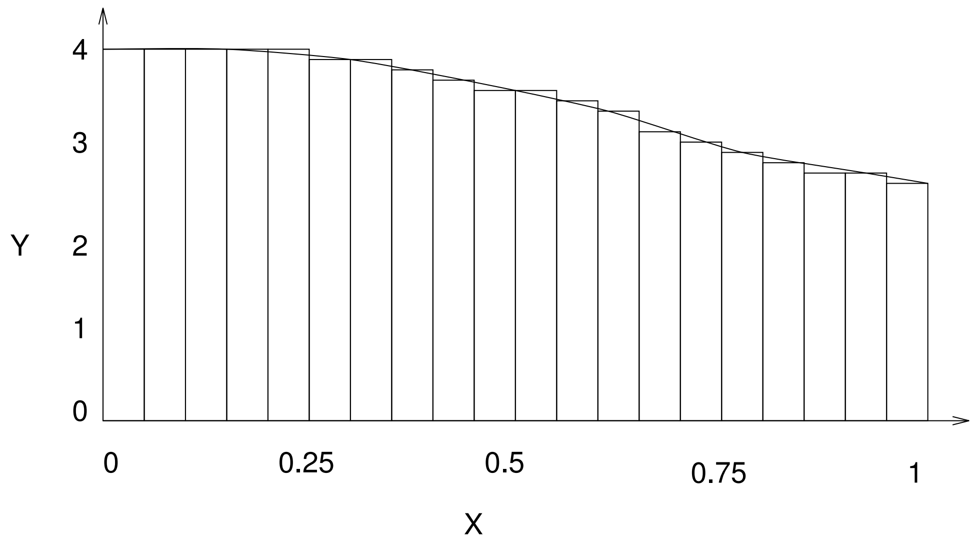
- This integration can be evaluated by computing the area under the curve for $f(x) = \frac{4}{1\,+\,x^2}$ from 0 to 1.

- With numerical integration using the rectangle rule for decomposition, one divides the region $x$ from 0 to 1 into $n$ points.

- The value of the function $f(x) = \frac{4}{1\,+\,x^2}$ is evaluated at the midpoint of each interval

- The values are summed up and multiplied by the width of one interval.

## Illustration of PI Computation



## Serial PI Program

```
/*  serial pi program  */

float h,sum,x,pi;
int i,n;

main()
{
   printf("Enter the number of intervals\n");
   scanf("%d",&n);
  /* call procedure to compute pi */
   computepi(n);

   printf("Value of pi is %f\n",pi);
}

computepi()
{
```

## Serial PI Program

```
   h = 1.0 / n;
   sum = 0.0;
   for (i=0; i < n; i++) {
     x = h * (i + 0.5);
     sum = sum + 4.0 / (1 + x * x);
   }
   pi = h * sum;
}
```

## Shared Memory Parallel Implementation

- We now present a shared memory MIMD parallel implementation of the pi computation.
- The algorithm uses a dynamic scheduling of the $i$ index loop to get the next value of $i$ stored in the *global_i* variable, which is incremented under locks.
- Each processor performs a local computation of $\pi$ and stores it in the *localpi* variable.
- Finally, the individual values of *localpi* are added up to the global value of $\pi$, named the $pi$ variable under locks.

## Shared Memory Parallel Implementation

```
/*  parallel pi program  */

shared int n, global_i;
shared float h, pi;
main()
{
   int nprocs;
   void computepi();

   printf("Enter the number of intervals");
   scanf("%d",&n);
   printf("Enter the number of processors");
   scanf("%d",&nprocs);
   /* initialize global index, pi, h */
   global_i = 0;
   pi = 0;
   h = 1.0 / n;
```

## Shared Memory Parallel Implementation

```
  /* create nprocs parallel threads */
   m_set_procs(nprocs);

  /* compute pi in parallel */
   m_fork(computepi);

  /* wait for all threads to complete */
   m_kill_procs();

   printf("Value of pi is %f",pi);
}

void computepi()

{
   int i;
   float sum, localpi, x;
```

## Shared Memory Parallel Implementation

```
sum = 0.0;

while (i < n) {

  m_lock();
    i = global_i;
    global_i = global_i + 1;
  m_unlock();

  x = h * (i + 0.5);
  sum = sum + 4.0 / (1 + x * x);
}
```

## Shared Memory Parallel Implementation

```
     localpi = h * sum;
     m_lock();
       pi = pi + localpi;
     m_unlock();
}
```

## Example Application 2: Matrix Vector Multiplication

- We will develop a parallel application for multiplying a matrix of $m$ rows and $n$ columns times a vector of $n$ elements to produce a vector of $m$ elements.

- Note that the multiplication can be performed two ways

- The first way [called the *DOTPRODUCT or (i,j) method*] is to compute each element of the result output vector as a inner product of a row vector of the input matrix $a$ to the vector $b$.

- The second method [called the *SAXPY or (j,i) method*] proceeds as follows. The algorithm proceeds by computing the scalar product of each element of the $b$ vector with a column vector of the $a$ matrix and summing up the intermediate result vectors to form the output vector.

- We will use the second method for our parallel implementation.

## Illustration of Matrix Vector Multiplication

$c = A . b$

$$\begin{bmatrix} 14 \\ 32 \\ 50 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} X \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Dot product method

```
for (i=0,i <m; i++)
  for(j=0; j<n; j++)
    c[i]  = c[i] + a[i,j] * b[j]
```

$14 = 1 * 1 + 2 * 2 + 3 * 3$ p0

$32 = 4 * 1 + 5 * 2 + 6 * 3$ p1

$50 = 7 * 1 + 8 * 2 + 9 * 3$ p2

SAXPY method

```
for(j=0; j<n; j++)
  for (i=0,i <m; i++)
    c[i]  = c[i] + a[i,j] * b[j]
```

$$\begin{bmatrix} 14 \\ 32 \\ 50 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 \\ 4 \\ 7 \end{bmatrix} * 1}_{p0} + \underbrace{\begin{bmatrix} 2 \\ 5 \\ 8 \end{bmatrix} *2}_{p1} + \underbrace{\begin{bmatrix} 3 \\ 6 \\ 9 \end{bmatrix} *3}_{p2}$$

## Serial Matrix-Vector Multiplication

```
/* serial matrix vector multiplication program */

float a[max][max], b[max], c[max];
int i,j,m,n;

main()
{
   /* input m and n */

   printf("Enter number of rows and columns\n");
   scanf("%d %d",&m,&n);

   /* read data */
   readdata();

   /* perform matrix vector multiplication */
   matvec();
}
```

## Serial Matrix-Vector Multiplication

```
matvec()
{
   int i,j;

   for (j=0; j < n; j++) {
      for (i=0; i < m; i++) {
         c[i] = c[i] + a[i][j] * b[j];
      }
   }
}
```

## Shared Memory Parallel Algorithm

- We will partition the problem by allowing each processor to perform the multiplication of a set of columns of the input matrix times the corresponding sets of elements of the input vector to produce an intermediate result vector.
- The computations of each intermediate vector will be performed in parallel among all the processors.
- These intermediate vectors are going to be accumulated among the processors in a sequential step.
- We will use a static interleaved scheduling algorithm for distributing the $j$ index iterations of the matrix vector multiplication loop.
- A processor $i$ picks the iterations $i$, $i+p$, $i+2p$, and so on, where $p$ is the number of processors, determined at runtime.

## Shared Memory Parallel Implementation

```
/* parallel matrix vector multiplication program */

shared float a[max][max], b[max], c[max];
shared int m,n;

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

```
   /* create nprocs parallel threads */
   m_set_procs(nproc);

  /* perform parallel multiplication */
   m_fork(matvec);

   /* wait for all threads to complete */
   m_sync();

   /* 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;
   }
```

## Shared Memory Matrix Vector Multiplication

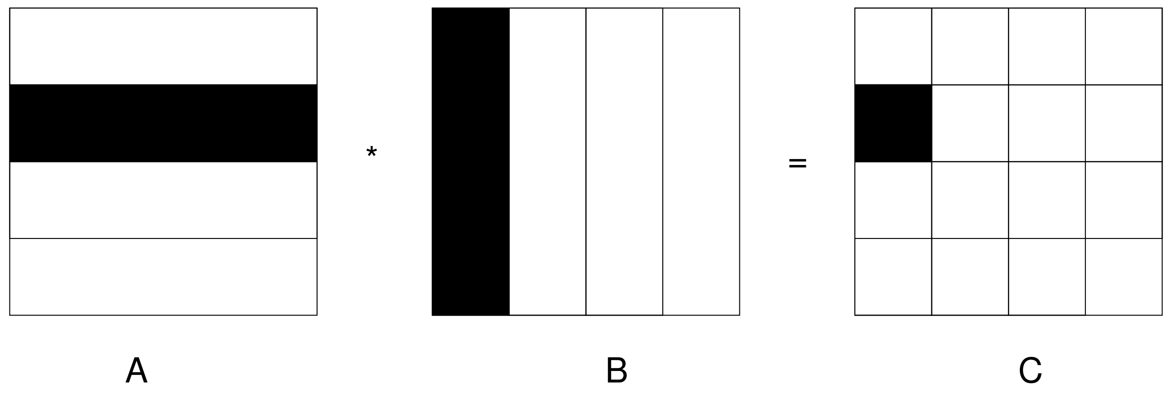
```
   nprocs = m_get_numprocs();
   myid = m_get_myid();

   for (j = myid; j < n; j = j + nprocs) {
      for (i=0; i < m; i++)
         tmp[i] = tmp[i] + a[i][j] * b[j];
   }
   m_lock();
      for (i=0; i < m; i++)
        c[i] = c[i] + tmp[i];
   m_unlock();
}
```

## Example Application 3: Matrix-Matrix Multiplication

- There are many ways of performing matrix multiplication, which multiplies matrices $a$ and $b$ of sizes $n \times n$ and produces a result matrix $c$.
- We show one way in the following program
- Each result matrix element is obtained as an inner product of a row of the $a$ matrix (containing $n$ elements) times a column of the $b$ matrix (containing $n$ elements).
- The program first reads the value of the size of the matrices, then reads the data for the two matrices, an then performs the matrix computation and prints out the results.

## Matrix Multiplication



## Serial Implementation

```
/* serial matrix matrix multiplication */

#define max 100

float a[max][max], b[max][max], c[max][max];
int n;
main()
{
   /* input n the size of matrices */

    printf("Enter n, the size of matrices\n");
    scanf("%d",&n);
    /* reads matrix a and b */
    readdata();
```

## Serial Implementation

```
   /* call procedure to multiply matrices */
    matmul();
   /* prints matrix c */
    printdata();
}
```

## Serial Implementation

```
matmul()
{
  int i, j, k;

  for (i = 0; i < n; i++) {
     for (j = 0; j < n;  j++) {
        c[i][j] = 0.0;
        for (k = 0 ; k < n; k++) {
          c[i][j] = c[i][j] + a[i][k] * b [k][j];
        }
     }
  }
}
```

## Parallel Implementation

- The preceding application is parallelized on a shared memory MIMD multi-processor using static block-wise scheduling
- The blocking is done in *both dimensions of the matrix*.
- For a $P$ processor system, the processors are logically arranged as a $\sqrt{P} \times \sqrt{P}$ array on which the result $c$ matrix is logically mapped.
- Each processor is assigned the computations of a rectangular subblock of the $c$ matrix.
- The computation of each subblock of the $c$ matrix involves accessing a row subblock of the $a$ matrix and a column subblock of the $b$ matrix.

## Parallel Implementation

```
#define max 100
shared float a[max][max], b[max][max], c[max][max];
shared int n;

main()

{
    int nprocs;
    void readdata(),matmul(),printdata();

   /* input n the size of matrices */

    printf("Enter the size of matrices\n");
    scanf("%d",&n);
    printf("Enter the number of processors\n");
    scanf("%d",&nprocs);

    /* reads matrix a and b */
    readdata();
```

## Parallel Implementation

```
   /* set number of processes */
    m_set_procs(nprocs);

   /* execute parallel loop */
    m_fork(matmul);

    /* kill child processes */
    m_kill_procs();
   /* prints matrix c */
    printdata();
}
```

## Parallel Implementation

```
void matmul()
{
  int i, j, k;
  int nprocs, iprocs, jprocs;
  int my_id, i_id,j_id,ilb,iub,jlb,jub;
 /* number of processors */
  nprocs = m_get_numprocs();
  /* number of processors in i direction */
  iprocs = (int) sqrt((double) nprocs);
```

## Parallel Implementation

```
/* number of processors in j direction */
jprocs = nprocs / iprocs;

my_id = m_get_myid();
/* get processor ID in i and j dimensions */
i_id = my_id / iprocs;
j_id = my_id % jprocs;
/* find lower and upper bounds of i loop */
ilb = i_id * n / iprocs;
iub = (i_id + 1) * (n / iprocs);

/* find lower and upper bounds of j loop */
jlb = j_id * n / jprocs;
jub = (j_id + 1) * (n / jprocs);
```

## Parallel Implementation

```
  for (i = ilb; i < iub; i++)
     for (j = jlb; j < jub;  j++) {
        c[i][j] = 0.0;
        for (k = 0 ; k < n; k++)
          c[i][j] = c[i][j] + a[i][k] * b [k][j];
     }
}
```

## Summary

- Example parallel computation: PI
- Example parallel computation: matrix-vector multiplication
- Example parallel computation: matrix-matrix multiplication
- READING: Chapter 7