HW #5 Due March 4
Implement Matrix Vector Product A*b = c using MPI
Use 1-D partitioning → row striping A
and i) broadcast b to all nodes
ii) scatter n/p b elements to each node followed by all-to-all
Try with 4 nodes, 16*16 matrix
then increase matrix (and b) size to 32*32…
Submit code, screen shots showing output and timing information

Cannon’s Algorithms for 2-D (checkerboard) partitioning for matrix multiplication
A*B =C
shift left row wise for A and up column wise for B

We need two all-to-all broadcasts -- blocks from rows of A and blocks from columns of B
Each broadcast is for (n/p^{1/2}) * (n/p^{1/2}) = n^2/p words of data
Communication cost in hypercube = 2[t, log(p^{1/2}) +t_e(m)*((p^{1/2} -1)) = t, log p + 2t_e (n^2/p^{1/2})].
Each processor also computes (n/p^{1/2}) * (n/p^{1/2}) = n^2/p element of C.
Each element of C requires n multiplications and n additions
So each processor has a computation time of O(n^3/p)
Total Execution Time T_p = (n^3/p ) + t, log p + 2t_e (n^2/p^{1/2})
T_o = p*[t, log p + 2t_e (n^2/p^{1/2})]
The iso-efficiency function with respect to t_e, W = O(p^{1/2})
CSCE 5160 Parallel Processing

What about memory overhead? Each processor must contain $p^{1/2}$ sub-matrices of $A$ and $p^{1/2}$ sub-matrices of $B = 2(n^2/p)$ or $O(n^2/p)$ as compared to $O(n^2)$ for a sequential solution.

For 1-D initially each processor receives $n/p$ rows of $A$ and $B (=n^2/p)$
But each processor will then receive all of $B$ for a total memory of $n^2 + n^2/p$

Creating groups of processes or splitting processes along rows and columns

```c
int MPI_Comm_Split (MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

An Example

```c
int MPI_Comm_Split (MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

Processes 0, 2, 3 will use color = 0; process 0 will use key=0, process 2, key=1, process 3, key=2
Likewise, processes 1, 5, 6, 7 will use color=1; but the keys are p5=0, p7=1, p1=2, p6=3

More specific topologies – like 2D and 3D grids

MPI has built-in functions to do this.
CSCE 5160 Parallel Processing

int MPI_Cart_create
  
  ( 
    MPI_Comm old_comm, /*original communication group */
    int dims, /* # dimension in the grid */
    int *size, /* size of each dimension */
    int *periodic, /* for each dimension, wrap-around or not */
    int reorder, /*renumber processes in each group? */
    MPI_Comm *cart_comm /*new groups in grid */
  )

This function creates process group to look like a 2-D grid of processes

dimensions =2;
dims[0] = dims[1] = sqrt(npes);
periods[0]=periods[1]= 1;

MPI_Cart_create(MPI_Comm_World, dimension, dims, periods, 1,&comm_2d);

MPI_Comm_rank(comm_2d, &my2drank);
MPI_Cart_coords(comm_2d, my2drank, dimension, mycoords);

CSCE5160 February27, 2019

If we can divide processes into a square, it is easier. If not?
Say we have 8 processors, what is the best way to divide them into a 2-D grid?

int MPI_Dims_create (int nodes, int dims, int *size)

This function takes #nodes, and #dims, and tries to create a grid that is as close to being a square
as possible. size[0], size[1],....size[dims-1] contain the number of processes in each dimension

We can use this in our call to MPI_Cart_create

To move data along rows or columns we will use another MPI function
This function finds your source and destination processors (neighbors) based in dimension of
shift, if it is left or right shift

int MPI_Cart_shift
  
  ( 
    MPI_Comm comm_cart, /* Cartesian groups */
    int shft_dim, /* which dimension to shift -- row, column */
    int direction, /* direction, >0 up/left; <0 down/right */
    int disp, /* by how much? */
    int *rank_source, /* source */
    int *rank_dest /* destination */
  );
CSCE 5160 Parallel Processing

MPI_Cart_shift(comm_2d, 0, -mycoords[1], &shiftsource, &shiftdest);

/* shiftsource and shiftdest will be used to send and receive
/* We need to execute a similar function for other dimensions

this combines send and receive, and also uses the same buffer for sending and receiving

int MPI_Sendrecv_replace ( 
    void* buff, /*buffer for sending and receiving 
    int count, /* number of items involved 
    MPI_Datatype datatype,/* data type 
    int dest, /* destination for your sending 
    int sendtag, /* tag of message sent 
    int source, /* source for your receiving 
    int rcvtag, /* tag of message received 
    MPI_Comm comm, /* communication group involved 
    MPI_Status, *status /*status of the operation 
) 

An example
MPI_Sendrecv_replace(b, nlocal, MPI_INT, shiftdest, 1, shiftsource, 1, comm_2d, &status);

Other useful functions. Send and receive at the same time

int MPI_Sendrecv( 
    void *sendbuf, 
    int sendcount, 
    MPI_Datatype sendtype, 
    int dest, 
    int sendtag, 
    void *recvbuf, 
    int recvcnt, 
    MPI_Datatype recvtype, 
    int source, 
    int recvtag, 
    MPI_Comm comm, 
    MPI_Status *status 
) 

CSCE 5160 Parallel Processing

We already have seen OpenMP version of matrix multiplication

```c
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];
}
```

Note this is similar to “row striping” in MPI

Is there any reason to think of something like Cannon’s algorithm with OpenMP?
To reduce the amount of memory needed at at each thread
And improve cache performance

---

CSCE 5160 Parallel Processing

Blocking to improve cache accesses (reduce misses)

Original Matrix multiplication

```c
for (i = 0; i < n; i = i+1)
    for (j = 0; j < n; j = j+1)
    {
        x[i][j] = 0;
        for (k =0; k < n; k = k+1)
            x[i][j] = x[i][j]  + y[i][k]*z[k][j];
    }
```

And rewrite the code as

```c
for (jj=0; jj < n; jj = jj + block_size)
    for (kk = 0; kk < n; kk = kk + block_size)
        for (i = 0; i < n; i = i+1)
            for (j = jj; j < min (jj+block_size, n); j = j+1)
                x[i][j] = 0;
            for (k =kk; k < min(kk+block_size, n); k = k+1)
                x[i][j] = x[i][j]  + y[i][k]*z[k][j];
```

CSCE 5160 Parallel Processing

Solving system of linear equations

can be dense or sparse

One of the most common need for matrix related algorithms stems from the need to solve a system of linear equations in n-unknowns.

Typically we have a set of n equations as follows:

\[ a_{00} x_0 + a_{01} x_1 + \ldots + a_{0n-1} x_{n-1} = b_0 \]
\[ a_{10} x_0 + a_{11} x_1 + \ldots + a_{1n-1} x_{n-1} = b_1 \]
\[ \vdots \]
\[ a_{n-10} x_0 + a_{n-11} x_1 + \ldots + a_{n-1n-1} x_{n-1} = b_{n-1} \]

We can write this as \( A \times X = B \) where \( A \) is a \( n \times n \) matrix, \( X \) is a \( 1 \times n \) vector, and \( B \) is a \( 1 \times n \) vector.

One of the most common technique used to solve the above problem is called Gaussian Elimination technique.

CSCE 5160 Parallel Processing

Gaussian elimination technique

Convert \( A \) into Upper Triangular fashion

This involves two main steps:

**Division step:** during this step the next row makes its diagonal element =1

This row is called **Pivot Row**

Divide all the coefficients of the row by the coefficient of the diagonal element

The diagonal element is called **Pivot Element**

**Elimination step**

All subsequent rows (rows below) take a multiples of the above row and subtract from its coefficients

making first column element in each row = 0

Repeat this process going to the next row.
Gaussian Elimination method

Numerical Stability Issues

• If pivot element is close to zero, significant round-off errors can result
• Gaussian elimination with partial pivoting eliminates this problem
• In step \( i \) we search rows \( i \) through \( n-1 \) for the row whose column \( i \) element has the largest absolute value
• Swap (pivot) this row with row \( i \)

Sequential Code

```c
for (k=0; k < n-1; k++)
{
    for (j=k+1; j< n-1; j++)
    {
        A[k][j] = A[k][j]/A[k][k]; /* division step */
    }
    /* division step */
    b[k] = b[k]/A[k][k]; /* need to divide the right side of the equations */
    A[k][k] =1;

    for (i=k+1; i<n-1; i++)
    {
        for (j = k+1; j< n-1; j++)
        {
        }
        /* i loop */
        b[i] = b[i] - A[i][k]*b[k]; /* need to subtract from the right side also */
        A[i][k] = 0;
    } /* k loop */
} /* k loop */
```
CSCE 5160 Parallel Processing

So what is the execution complexity of this algorithm on a single processor?

We have a total of 4 loops

The division step is repeated $O(n^2/2)$ times.

- In iteration $i$, pivot row $= i$, all elements in columns $0, 1, \ldots, i-1$ are zeros
  - So we divide only elements in columns $i+1, \ldots, n-1$ or $(n-1)$ divisions

We repeat this with each value of $i$: So we have

$$\sum_{i=0}^{n-1} (n-1-i)$$

The subtraction step (line 12) is repeated $O(n^3/3 - n^2/2)$ times.

- In iteration $i$, all rows below $(i+1, \ldots, n-1)$ perform multiplication plus subtraction (elimination)
  - And each row need to subtract starting with column $i$ through $n-1$

$$\sum_{i=0}^{n-1} (n-1-i)^* (n-1-i)$$

Summarizing

The division step is repeated $O(n^2/2)$ times.

The subtraction step (line 12) is repeated $O(n^3/3 - n^2/2)$ times.

If we assume that each arithmetic operation takes one time unit, the subtraction and multiplication step takes 2 time units and the complexity due to these operations is

$$O(2^* (n^3/3 - n^2/2))$$

Total execution time

$$O(n^2/2) + O(n^3/3 - n^2/2) + O(2^* (n^3/3 - n^2/2))$$

So the total complexity is dominated by $O(2^*n^3/3)$

And the problem size $W$ is given by $2^*n^{3/3}$

How can we parallelize this?
Parallelizing Gaussian Elimination method

There are 4 nested loops in the algorithm

1. outmost loop with k (one per row)
2. the j loop that performs division step
3 and 4. i and j loops that perform elimination step

Which of these loops can we parallelize?

Loop 1 cannot be executed in parallel because of the dependencies (division steps)
Loop 2 must be completed before loops 3
Loop 3 can be executed in parallel (elimination step)
Loop 4 can also be executed in parallel, depending on how data is distributed.

Let us consider column striping as our data distribution method (each processor uses a different j value)

Page 356 shows the communication and Computations resulting from row striping.
We will assume n processors.

We are now at P3 with a pivot row
Division done by P3
Send the P3 row to P4…P7
Processors P4…P7 perform Elimination step in parallel
CSCE 5160 Parallel Processing

Processor k will perform the division step
Computation cost = n-k-1 divisions

Then processor k will broadcast its row to all processors i > k.
This can be achieved using one-to-all broadcast and the message size is (n-k-1) words

Once all processors receive updated row k, they can compute the elimination step in parallel
Computation cost = (n-k-1) multiplications and (n-k-1) subtractions

If we use Hypercube, the communication cost for processor k one-to-all broadcast
= (t_s + t_w(n-k-1)) log n

So the total communication cost, we need to sum these costs for all k

$$\sum_{k=0}^{n-1} (t_s + t_w (n-k-1))\log(n) = t_s \log n + n^* (n-1)/2 \cdot t_w \log n$$

Let us look at the computation cost -- we have a total of 3 * (n-k-1) units for each k value.

$$\sum_{k=0}^{n-1} 3^* (n-k-1) = 3^* n^* (n-1) / 2$$

The total cost (communication + computation)

$$T_p = 3^*n^*(n-1)/2 + t_s \log n + n^*(n-1)/2 \cdot t_w \log n$$

If we set t_w = 1, and ignore the n^* log n we have the execution cost as O(n^2 + n^*n log n)

Since we are using n processors, the cost is O(n^3 log n)

Note that the sequential implementation cost is O(n^3).
So the parallel implementation is not cost-optimal

We can improve the solution if we can pipeline communication and computations.

Consider the following pipelining.

Processor k sends its row to processor k+1.
Processor k+1 receives the row, and initiates the forwarding of the row to processor k+2.
At the same time processor can start execution of the “elimination” step.
Processor k+2 receives updated row k, forwards it to k+3 but starts its own elimination step.

If processor k+1 completed its elimination step, it can start the division step for row k+1 and then start sending the modified row k+1 to processor k+2.

Textbook assumes this is true since they assume that you send one word at a time.

See the figure on page 358 -- and follow this step by step on how the pipelining works.

The elimination step is completed by the time data is forwarded to the next node.

Let us consider some specific examples. In part h of the figure, processor P2 is performing the elimination step while processor P3 and P4 are performing the communication step.

![Diagram of processor operations](image-url)
CSCE 5160 Parallel Processing

Consider part i. Here processor P2 is performing the division step on row 2 while processor P3 is performing the elimination step on row 3 while processor P4 just received the modified row 1 (and gets ready to perform elimination step).

Note unlike the previous parallel implementation, we are NOT using broadcast communication. At each step, *we only have one-to-one communication*

At step k, we have at most n-k-1 communications active simultaneously.

The total communication cost is $O(n^2)$
Likewise the total computation cost is $O(n^2)$

The cost (since we have n processors) is $O(n^3)$ which is the same as the sequential implementation – COST OPTIMAL

*(if we can overlap computations and communications as needed)*

---

How about using column striping?
Need communication during division step but only one element is communicated
No communication for elimination step

How about using checkerboard striping?
Now we need communication during both division and elimination steps

---

Row striping $\rightarrow$ chunking
Cyclical? This is very suitable for pipelining

---

(a) Block 1-D mapping

(b) Cyclic 1-D mapping

Row striping $\rightarrow$ chunking
Cyclical? This is very suitable for pipelining

---

How about using column striping?
Need communication during division step but only one element is communicated
No communication for elimination step

How about using checkerboard striping?
Now we need communication during both division and elimination steps
Consider an example

\[
\begin{align*}
4x_0 + 6x_1 + 2x_2 - 2x_3 &= 8 \\
2x_0 + 5x_2 - 2x_3 &= 4 \\
-4x_0 - 3x_1 - 5x_2 + 4x_3 &= 1 \\
8x_0 + 18x_1 - 2x_2 + 3x_3 &= 40
\end{align*}
\]

Assume 4 processors

We have to decide on how to distribute \( b \)

Consider the division step.
All elements of row \( k \) must be divided by \( a_{k,k} \)
But these elements are in different processors – so we need to broadcast \( a_{k,k} \) to processes in the same row.

This communication was not needed in row striping.

Consider the elimination step. We need to subtract some multiple of row \( k \) from all rows \( k+1 \) to \( n \). We need to send \( k+1 \)th column elements to processors in that column broadcast to \((p^{1/2})\) column processors

The asymptotic complexity is the same using the pipelined communication since the only difference is due to the division step and this computation can be pipelined.

Read the sections the textbook for more detailed investigation of these computations.

Pivoting. In all the above cases we are assuming that the diagonal elements are non zero. If not, our division step causes a problem.

So, what we may want to do is re-arrange the columns of \( A \) such that the diagonal element of any row is the largest element (as compared to the elements on the right hand side columns).

In iteration \( k \), we examine row \( k \), for columns \( k \) through \( n-1 \). Find the largest element (say in column \( i \)). Then we swap columns \( k \) and \( i \).

Once the pivoting is complete, we can use our previous techniques for generating Upper diagonal matrix.
How do we parallelize Pivoting step? Row striping is the best, since only one processor is involved in each pivoting iteration.

If we use checkerboard, we need to have \((p^{1/2}) \cdot k\) processors to search and find the largest element in row \(k\).

Read the textbook carefully

How to parallelize Gaussian Elimination using shared memory (OpenMP)?

```c
for (k=0; k < n-1; k++)
    { for (j=k+1; j< n-1; j++)
        A[k][j] = A[k][j]/A[k][k]; /* division step */
        y[k] = y[k]/A[k][k];
        A[k][k] = 1;
        for (i=k+1; i<n-1; i++)
        { for (j = k+1; j<n-1; j++)
            y[i] = y[i] - A[i][k]*y[k];
            A[i][k] = 0;
        } /* i loop */
    } /* k loop */
```

Does this work? We are spawning threads repeatedly; first \(n-1\), then \(n-2\), \(\ldots\), \(2\), \(1\). Can be expensive