CSCE 5160 Parallel Processing

Since our MPI cluster has only 5 nodes, it does not make much sense to use it for Cannon's algorithm. So for Homework #5 we will write an MPI program for Matrix*Vector product. Due on March 4, 2019

Matrix-Vector product $A*b = c$

Here our data consists of a n*n matrix and a n*1 vector.
We need to distribute the data (both matrix A and vector b) to processors
We can again use row striping for the matrix (n/p rows to each node)
What about the vector?
  i). n/p vector elements per processor?
  ii) broadcast entire b to all processors?

Use 4 nodes in MPI. Make sure your matrix is at least 16*16
You can create the matrix and vector using random numbers
*You should try with larger matrices and plot efficiency and iso-efficiency*

Review:
Dense Matrix algorithms
Decomposing input matrices
row striping, column striping or Checkerboard

Different distributions/decomposition require different types of communication
and different amount of memory

Matrix Multiplication
Let us consider 1-D partitioning: row stripe A and column stripe B

We need all of B at each processor
We need *all to all* broadcast
p processors and each messages contains
n^2/p elements

CSCE5160 February25, 2019
CSCE 5160 Parallel Processing

Computation cost: \( n^3/p \)
Communication cost: all to all broadcast
\( t_s \log(p) + t_w \times (n^2/p)(p-1) \)

\[
T_p = n^3/p + t_s \log(p) + t_w \times n^2 \quad \text{and} \quad T_0 = p^s t_s \log(p) + p^s t_w \times n^2
\]

We can define iso-efficiency with respect to \( t_s \) or \( t_w \)
\( W = K \times p^s t_s \log(p) \)
\( W = K \times p^s t_w \times n^2 \) but \( W = n^3 \) so \( W = K^3 t_w^3 p^3 \)

CSCE 5160 Parallel Processing

Now let us consider Matrix Multiplication using checker-board partitioning.

Need all to all broadcast of A sub-matrices in each row
Need all to all broadcast of B sub-matrices in each column

**We need two all-to-all broadcasts** -- blocks from rows of A and blocks from columns of B

Each broadcast is for \((n/p)^{1/2}) \times (n/p)^{1/2} = n^2/p\) words of data

Communication cost in hypercube = \(2[t_s \log(p^{1/2}) + t_w(m)(p^{1/2} - 1)] = t_s \log(p) + 2t_w \times (n^2/p^{1/2})\)

Each processor also computes \((n/p)^{1/2}) \times (n/p)^{1/2} = n^2/p\) element of C.
CSCE 5160 Parallel Processing

Each processor also computes \((n/p^{1/2}) \times (n/p^{1/2}) = n^2/p\) element of \(C\).

Each requires \(n\) multiplications and \(n\) additions, for a total of \((n^3/p)\) multiplications and additions

Total Execution Time \(T_p = (n^3/p) + t_s \log p + 2t_s (n^2/p^{1/2})\)

\[ T_s = p^*[t_s \log p + 2t_s (n^2/p^{1/2})] \]

The iso-efficiency function with respect to \(t_s\) \(W = O(p^{3/2})\)

What about memory overhead? Each processor must contain \(p^{1/2}\) sub-matrices of \(A\) and \(p^{1/2}\) sub-matrices of \(B = 2(n^2/p^{1/2})\)

or \(O(n^2p^{1/2})\) as compared to \(O(n^2)\) for a sequential solutions

This algorithm requires lot of memory at each node

Each processor must hold all of \(B\) and \(n^2/p\) elements of \(A\)

Can we do better in terms of memory?

CSCE5160 February25, 2019

---

CSCE 5160 Parallel Processing

**Cannon’s Algorithm**

At any time during the computation, we need only one sub-matrix each of \(A\) and \(B\) for a memory size of \((n/p^{1/2}) \times (n/p^{1/2}) = n^2/p\)

To make this algorithm work, we need to carefully circulate the sub-matrices to all processors. Let us look at page 347 to see how this is achieved.

In each step, we shift the \(A\) sub-matrices around so that each processor can perform local computations based on the \(B\) sub-matrices originally allocated to that processor.

For example, processor \(p_{10}\) has the \(A_{10}\) and \(B_{10}\) sub-matrix.

We can multiply these submatrices to get some partial results

Once we complete the initial local computations, we shift both \(A\) and \(B\) sub-matrices.

Here we only shift \(A\) submatrices left by one position (wraparound) and each \(B\) submatrix is shifted up by one position (wraparound).

We again compute using local sub-matrices, and add these results to previous partial results

CSCE5160 February25, 2019
CSCE 5160 Parallel Processing

\[
\begin{array}{cccc}
A_{00} & A_{01} & A_{02} & A_{03} \\
A_{10} & A_{11} & A_{12} & A_{13} \\
A_{20} & A_{21} & A_{22} & A_{23} \\
A_{30} & A_{31} & A_{32} & A_{33} \\
\end{array}
\hspace{2cm}
\begin{array}{cccc}
B_{00} & B_{01} & B_{02} & B_{03} \\
B_{10} & B_{11} & B_{12} & B_{13} \\
B_{20} & B_{21} & B_{22} & B_{23} \\
B_{30} & B_{31} & B_{32} & B_{33} \\
\end{array}
\]

\[C_{00} = A_{00} \times B_{00}\]

\[
\begin{array}{cccc}
A_{00} & A_{01} & A_{02} & A_{03} \\
A_{10} & A_{11} & A_{12} & A_{13} \\
A_{20} & A_{21} & A_{22} & A_{23} \\
A_{30} & A_{31} & A_{32} & A_{33} \\
\end{array}
\hspace{2cm}
\begin{array}{cccc}
B_{00} & B_{01} & B_{02} & B_{03} \\
B_{10} & B_{11} & B_{12} & B_{13} \\
B_{20} & B_{21} & B_{22} & B_{23} \\
B_{30} & B_{31} & B_{32} & B_{33} \\
\end{array}
\]

\[C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10}\]

\[
\begin{array}{cccc}
A_{00} & A_{01} & A_{02} & A_{03} \\
A_{10} & A_{11} & A_{12} & A_{13} \\
A_{20} & A_{21} & A_{22} & A_{23} \\
A_{30} & A_{31} & A_{32} & A_{33} \\
\end{array}
\hspace{2cm}
\begin{array}{cccc}
B_{00} & B_{01} & B_{02} & B_{03} \\
B_{10} & B_{11} & B_{12} & B_{13} \\
B_{20} & B_{21} & B_{22} & B_{23} \\
B_{30} & B_{31} & B_{32} & B_{33} \\
\end{array}
\]

\[C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10} + A_{02} \times B_{20}\]
CSCE 5160 Parallel Processing

Execution time: Note we have been ignoring the time needed to get original data to processors, either using 1-D or 2-D partitioning

Repeat: We compute partial products
We then do the circular shifts of A’s and B’s
Each shift transfer n²/p elements (of A or B)
  * let us assume each shift is to a neighbor at a distance of 1 hop
Cost of communication for shift = 2*(tₚ + tₑ * n²/p)

How many time do we repeat these steps (computation followed by circular shift)?

How many times do we repeat these steps (computation followed by circular shift)? p⁻¹/₂

\[ T_p = \frac{n^3}{p} + (p^{1/2})*[2*(tₚ + tₑ * (n²/p))] \]

\[ T₀ = 2^p p^{1/2} tₑ + 2^p * n²p^{1/2} \]

Iso-efficiency with respect to tₑ → W = K*[2^p tₑ * n²p^{1/2}]

\[ W = n^3 = 8*K²tₑ²p^{1/2} \text{ or } O(p^{3/2}) \]

No difference in complexity compared regular checkerboard implementation.
What we improved is the memory requirements.

CSCE5160 February 25, 2019

---

CSCE 5160 Parallel Processing

Farm’s algorithm

There is a slightly different way of achieving the same memory complexity as given by Cannon’s algorithm.

Here, one processor in each row will broadcast their A sub matrices to everyone in its row. Multiply using local B sub-matrices. Then shift B’ matrices up.

Then select another processor in each row, broadcast the data to all processors in its row. Multiply with new B matrices and so on.

<table>
<thead>
<tr>
<th>A₀₀</th>
<th>A₀₁</th>
<th>A₀₂</th>
<th>A₀₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁₁</td>
<td>A₁₂</td>
<td>A₁₃</td>
<td>A₁₄</td>
</tr>
<tr>
<td>A₂₂</td>
<td>A₂₃</td>
<td>A₂₄</td>
<td>A₂₅</td>
</tr>
<tr>
<td>A₃₃</td>
<td>A₃₄</td>
<td>A₃₅</td>
<td>A₃₆</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B₀₀</th>
<th>B₀₁</th>
<th>B₀₂</th>
<th>B₀₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₁₀</td>
<td>B₁₁</td>
<td>B₁₂</td>
<td>B₁₃</td>
</tr>
<tr>
<td>B₂₀</td>
<td>B₂₁</td>
<td>B₂₂</td>
<td>B₂₃</td>
</tr>
<tr>
<td>B₃₀</td>
<td>B₃₁</td>
<td>B₃₂</td>
<td>B₃₃</td>
</tr>
</tbody>
</table>
Execution time is:

Computation time at each processor is the same as Cannon’s $s = (n^3/p)$
The communication cost is more since we are using broadcast one processor
in each row broadcast to all others.

Using hypercube, the time for communication in Fox’s approach is

$$(1/2) t_s (p^{1/2}) \log p + t_s (n^2/p^{1/2}) \log p$$

*Note that this communication cost is more than that of Cannon’s algorithm*

To implement Cannon’s algorithm, we need to split our processes along rows and columns
– only need to rotate among processes in your row or column

**Creating new groups** Create virtual topologies: grids, hypercubes and other topologies

A general method:

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

This is a group communication – all processes in `comm` group must execute the same
function. The array `newcomm` will contain all newly created groups.

---

Although each process executes the same `MPI_Comm_Split` they use different color and key

Each new group is given a different "color" and the original communication group is
divided into groups. And each group will re-number processes.

The parameter `key` determine the "rank" order of processes within the same color group.
If all processes in a group have the same key, their rank is determined by the rank in the
original communication group.

See an example on page 273.

A slightly different example

```
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Original process rank

Color

Key
CSCE 5160 Parallel Processing

A slightly different example

Original process rank

New process rank

Original process rank

Color

Key: rank ordered by Key and original rank

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.

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_Cart_create *cart_comm /*new groups in grid */

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

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

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

MPI_Comm_rank(comm_2d, &my2drank);

MPI_Cart_coords(comm_2d, my2drank, dimension, mycoords);
CSCE 5160 Parallel Processing

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?

```c
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

```c
MPI_Cart_create(comm, dimension, dims, periods, 1, &comm_2d);
```

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

To move data along rows or columns we will use another MPI function

```c
int MPI_Cart_shift

MPI_Comm comm_cart,
int shit_dim,
int direction,
int disp,
int *rank_source,
int *rank_dest

/* Cartesian groups */
/* which dimension to shift – row, column */
/* direction, >0 up/left; <0 down/right */
/* by how much? */
/* source */
/* destination */
);
```

For each node which executes this function (need to execute this for each dimensions)
we receive the ranks of your neighbors left/right or up/down
This information is needed when we shift actual data

```c
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 */

```c
MPI_Sendrecv_replace(b, nlocal * nlocal, MPI_INT, shiftdest, 1, shiftsource, 1, comm_2d, &status);
```

/* 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);

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

Other useful functions. Send and receive at the same time

int MPI_Sendrecv(void *sendbuf, /*send buffer*/
    int sendcount, /* number of items involved*/
    MPI_Datatype sendtype, /* send data type*/
    int dest, /* destination for your sending*/
    int sendtag, /* tag of message sent*/
    void *recvbuf, /* receive buffer*/
    int recvcount, /* number of items involved*/
    MPI_Datatype recvtype, /* receive data type*/
    int source, /* source for your receiving*/
    int recvtag, /* tag of message received*/
    MPI_Comm comm, /* communication group involved*/
    MPI_Status *status /*status of the operation*/)  

We already have seen OpenMP version of matrix multiplication

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

Is there any reason to think of something like Cannon’s algorithm with OpenMP?

Improve cache memory accesses
CSCE 5160 Parallel Processing

Once Thomas installs MPIH on arch we can do cannon’s MPI version on arch (there is no physical communication since arch is a shared memory system)

We can also implement OpenMP version of cannon.

In each iteration (note the number of iterations = \( p^{1/2} \))
   - Each processor (or thread) needs to its starting index for rows of A
   - ending index for rows of A
   - starting and ending indexes for columns of B

After completing one iteration
   - We “rotate” A matrix row wise ((current index + \( p^{1/2} \)) modulo n)
   - We “rotate” up ((current index-\( p^{1/2} \)) modulo n)

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
\[
\begin{align*}
    a_{00} x_0 + a_{01} x_1 + \ldots + a_{0n-1} x_{n-1} &= b_0 \\
    a_{10} x_0 + a_{11} x_1 + \ldots + a_{1n-1} x_{n-1} &= b_1 \\
    \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \\
    a_{n-10} x_0 + a_{n-11} x_1 + \ldots + a_{n-1n-1} x_{n-1} &= b_{n-1}
\end{align*}
\]

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

One of the most common technique used to solve the above problem is called Gaussian Elimination technique
CSCE 5160 Parallel Processing

Before we talk about Gaussian technique, here are some special matrices to know:

- Symmetrically banded
- Upper triangular
- Lower triangular
- Diagonally dominant
- Symmetric

If width = w, then $a_{ij} = 0$ for all $i-j > w$ and $j-i > w$.

Symmetrically banded with width 2

Upper triangular

<table>
<thead>
<tr>
<th>4</th>
<th>2</th>
<th>-1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>-4</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>-2</td>
<td>0</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Lower triangular

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>-2</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>-3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
CSCE 5160 Parallel Processing

Diagonally dominant
The absolute value of the diagonal element is greater than the sum of other elements in the row

\[ |a_{ii}| > \sum_{j \neq i} |a_{ij}| \]

Symmetric

\[
\begin{array}{cccccc}
19 & 0 & 2 & 2 & 0 & 6 \\
0 & -15 & 2 & 0 & -3 & 0 \\
5 & 4 & 22 & -1 & 0 & 4 \\
2 & 3 & 2 & 13 & 0 & -5 \\
5 & -2 & 0 & 1 & 16 & 0 \\
-3 & 5 & 5 & 3 & 5 & -32 \\
\end{array}
\]

\[
\begin{array}{cccccc}
3 & 0 & 2 & 2 & 0 & 6 \\
0 & 7 & 4 & 3 & -3 & 5 \\
2 & 4 & 0 & -1 & 0 & 4 \\
2 & 3 & -1 & 9 & 0 & -5 \\
0 & -3 & 0 & 0 & 5 & 5 \\
6 & 5 & 4 & -5 & 5 & -3 \\
\end{array}
\]

CSCE 5160 Parallel Processing

Key observations.
We can substitute any equation with a multiple of the coefficients
We can add or subtract a multiple of coefficients of one equation from coefficients of other equations
(we can apply these to matrices – each equation is a row)

Consider a simple example

\[
\begin{align*}
4x_0 + 6x_1 + 2x_2 - 2x_3 &= 8 & \text{Multiply first equation by 0.5 and subtract from second equation} \\
2x_0 + 5x_2 - 2x_3 &= 4 & \text{Multiply first equation by -1 and subtract from third} \\
-4x_0 - 3x_1 - 5x_2 + 4x_3 &= 1 & \text{Multiply first equation by 2 and subtract from 4th equation} \\
8x_0 + 18x_1 - 2x_2 + 3x_3 &= 40 \\
\end{align*}
\]
CSCE 5160 Parallel Processing

\[\begin{align*}
4x_0 + 6x_1 + 2x_2 - 2x_3 &= 8 \\
-3x_1 + 4x_2 - 1x_3 &= 0 \\
+3x_1 - 3x_2 + 2x_3 &= 9 \\
+6x_1 - 6x_2 + 7x_3 &= 24
\end{align*}\]

Multiply second equation by -1 and subtract from third equation

Multiply second equation by -2 and subtract from fourth

Multiply third equation by 2 and subtract from fourth

Now we can solve for \(x_3\) from last equation

\[x_3 = 2\]

Substituting in third

\[x_2 + 2 = 9 \text{ or } x_2 = 7\]

\[-3x_1 + 28 - 2 = 0 \text{ or } x_1 = 26/3\]

\[4x_0 + 52 + 14 - 4 = 8 \text{ or } x_0 = -54/4\]

This is the basis for Gaussian elimination technique

The only difference is that, generally we want the leading coefficient = 1

We divide the row by the leading coefficient

Division Step

Then we take multiples of the row and subtract from rows below

Elimination step

We start with first row and repeat it