CSCE 5160 Parallel Processing

Chapter 8: Dense Matrix Algorithms
We have already seen a few algorithms
- matrix * vector product
- matrix * matrix multiplication
We have talked about row-wise and column-wise “striping” (1D partition)
  We can also do a checker-board distribution (2D partition)
We will elaborate on these and other algorithms.

Let us start with a simple problem of transposing a matrix.
Do we really need to do this in parallel?

First, we need to distribute data to various Processors
We can assume either row or column striped or checker-board distribution.
We can scatter different rows (or sub matrices) to processors
And gather the columns (or sub-matrices ) in different order
You need to decide how many words to send, and also make sure the data being sent is in consecutive locations.

Row major storage makes it easy to scatter rows.

You need to make sure all processors must first read their partitions before anyone is allowed to write. Use barrier synchronization.
#pragma omp parallel for
for (i=0, i<number_of_threads) {
    processor[i].read_partition[i];
    barrier;
    processor[i].write_partition[j];
    barrier;
}

What is the execution time?
Each processor reads $n^2/p$ elements and writes $n^2/p$ elements

If each read and write takes one time unit (and no bus conflicts) $= 2(n^2/p)$

Each barrier will take $O(p)$ operations.

So, total execution time $= 2p + 2(n^2/p)$

MPI version: We have one scatter $n^2/p$ elements and gather of $n^2/p$ element
Each has a complexity of $t_s \log p + t_w (n^2/p)(p-1)$

---

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 or checker-board partitioning for the matrix

What about the vector?
$n/p$ vector elements per processor?
or broadcast entire b to all processors?

Consider broadcasting all n elements of b to all processors in a hypercube.

This will take $[t_s + n*t_s] \log p$

Each processor performs $O((n/p)*n)$ computations

$$T_s = (n^2/p) + [t_s + n*t_s] \log p$$
$$T_w = [W + T_s(W, p)]/p$$
$$T_b = [t_s + W*t_s] \log p$$
CSCE 5160 Parallel Processing

Suppose we distributed \( \frac{n}{p} \) elements of \( b \) to each processor. Then each processor broadcasts \( \frac{n}{p} \) elements to other processors using all-to-all communication.

Communication cost for all-to-all (each sending \( \frac{n}{p} \) elements):

\[
T_p = \left( \frac{n^2}{p} \right) + \left( t_s \log p + t_w \cdot n \right) \quad \text{and} \quad T_o = p \cdot \left( t_s \log p + t_w \cdot n \right)
\]

But

\[
W = \frac{(E/(1-E)) \cdot T_o(W,p)}{}
\]

We can derive \( W \) in terms of \( t_s \) first

\[
W = K \cdot t_s \cdot p \cdot \log p
\]

We can then derive \( W \) in terms of \( t_w \)

But \( W = n^2 = K \cdot t_w \cdot n \cdot p \)

Or

\[
W = n^2 = K^2 \cdot t_w \cdot p^2
\]

So depending on which of the two times are larger, we can estimate \( W \) needed to keep the same efficiency.

CSCE 5160 Parallel Processing

Matrix-Vector product \( A \cdot b = c \)

Now consider 2-D (checkerboard) partitioning of \( A \)

\[
\begin{array}{cc|cc}
A_{00} & A_{01} & A_{02} & A_{03} \\
A_{10} & A_{11} & A_{12} & A_{13} \\
A_{20} & A_{21} & A_{22} & A_{23} \\
A_{30} & A_{31} & A_{32} & A_{33}
\end{array}
\]

Which \( b \) elements are needed where?

\( b_0 \) are needed in the first column processors

\( b_1 \) are needed in the second column processors

……

How do we achieve this?

What type of communication is needed?

Step 1: First send “\( b_i \)” elements to the diagonal processors

Step 2: Then use one to all broadcast to get \( b_i \) elements to processors in that column

Step 3: Now we can perform products locally, and then

Step 4: we need to “reduce” the terms along each row (use all to one reduction)
CSCE 5160 Parallel Processing

Step 1: First get “b,” elements to the diagonal processors
    one to one communication: \( t_s + t_w \times (n/p^{1/2}) \times (p^{1/2}) = t_s + t_w \times n \)
    (n/p^{1/2}) is the number of elements of b sent to diagonal processor
    and p^{1/2} is the number of hops

    Text book (page 343) assumes these messages travel only one hop
    \( t_s + t_w \times (n/p^{1/2}) \)

Step 2: Then use one to all broadcast to get b, elements to processors in that column
    one to all broadcast, involving p^{1/2} processors, and (n/p^{1/2}) elements
    \( [t_s + t_w \times (n/p^{1/2})] \log (p^{1/2}) \) – assuming hypercube

Step 3: Now we can perform products locally
    \( n^2/p \)

Step 4: we need to “reduce” the terms along each row
    all to one reduction along rows involving p^{1/2} processors and n/p^{1/2} values
    \( [t_s + t_w \times (n/p^{1/2})] \log (p^{1/2}) \)

\[ \begin{align*}
    \text{Total } T_p &= \frac{n^2}{p} + t_s + t_w \times n + 2^* \times t_s \times (n/p^{1/2}) \log (p^{1/2}) \\
    &= \frac{n^2}{p} + t_s + t_w \times n + t_s \times \log (p) + t_w \times (n/p^{1/2}) \log (p) \\
    \text{or} \quad T_p &= \frac{n^2}{p} + t_s + t_w \times (n/p^{1/2}) + t_s \times \log (p) + t_w \times (n/p^{1/2}) \log (p)
\end{align*} \]

the terms in green are ignored

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

We can define iso-efficiency either in terms of \( t_s \) or in terms of \( t_w \).
Now let us talk about **Matrix Multiplication**

We already have seen how to distribute matrices using striping rows of A and columns of B.

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

![Matrix Multiplication Diagram]

We need all of B at each processor

We need *all to all broadcast* p processors and each messages contains \( n^2/p \) elements

Computation cost: \( n^3/p \)

Communication cost: all to all broadcast

\[ t_s \log(p) + t_w * (n^2/p)(p-1) \]

\[ T_p = n^3/p + t_s \log(p) + t_w * n^2 \]

and \( T_0 = p^* t_s \log(p) + p^*t_w * n^2 \)

Consider Checkerboard partitioning

---

\[ T_p = n^3/p + t_s \log(p) + t_w * n^2 \]

We can define iso-efficiency with respect to \( t_s \) or \( t_w \)

\[ W = K^* p^* t_s \log(p) \]

\[ W = K^* p^*t_w * n^2 \]

but \( W = n^3 \) so \( W = K^* t_w * p^3 \)

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

![Checkerboard Partitioning Diagram]

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 \((n/p^{1/2}) \times (n/p^{1/2})\) words of data.

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

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

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

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

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

The previous algorithm using checker-board partitioning is the best we can do in terms of execution time, but requires a lot of memory at each processor.

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

Can we do better in terms of memory?

**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.

We continue this cyclical left and up shifts of A and B submatrices until all computations can be completed.

\[
\begin{array}{cccc}
A_{00} & A_{01} & A_{02} & A_{03} \\
A_{10} & A_{11} & A_{12} & A_{13} \\
A_{20} & A_{21} & A_{22} & A_{23} \\
A_{30} & A_{31} & A_{32} & A_{33} \\
\end{array}
\quad
\begin{array}{cccc}
B_{00} & B_{01} & B_{02} & B_{03} \\
B_{10} & B_{11} & B_{12} & B_{13} \\
B_{20} & B_{21} & B_{22} & B_{23} \\
B_{30} & B_{31} & B_{32} & B_{33} \\
\end{array}
\quad
C_{00} = A_{00} \cdot B_{00}
\]
CSCE 5160 Parallel Processing

\[
C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20} + A_{03}B_{30}
\]

\[
C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21} + A_{03}B_{31}
\]

\[
C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22} + A_{03}B_{32}
\]

\[
C_{03} = A_{00}B_{03} + A_{01}B_{13} + A_{02}B_{23} + A_{03}B_{33}
\]

\[
C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20} + A_{13}B_{30}
\]

\[
C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}
\]

\[
C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32}
\]

\[
C_{13} = A_{10}B_{03} + A_{11}B_{13} + A_{12}B_{23} + A_{13}B_{33}
\]