Review: Cannon’s Algorithm using Checkerboard (or 2D) partitioning
Shift A submatrices along row processors
Shift B submatrices along column processors

In terms of Shared memory using OpenMP
We can view Cannon’s algorithm as blocking or tiling

Solving Linear Equations
Gaussian elimination technique
Convert A into Upper Triangular fashion

Division Step (ith row: Divide all coefficients to the right by diagonal element
Elimination Step (i+1th on, subtract a multiple of ith row to create zeros to the
left of diagona

When we parallelize Gaussian algorithm for MPI
Row striping

processor assigned ith row completes division and broadcasts ith row
Processors assigned i+1th row onwards, perform elimination step
We move to i+th row and repeat

Instead of broadcasting ith row (after division), overlap communication

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 checker board 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

Division step: processors in the same row need to communicate diagonal elements
Elimination step: processors in the same column must receive row elements

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
```
CSCE 5160 Parallel Processing

Consider parallelizing “i” loop

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] = b[k]/A[k][k];
        A[k][k] = 1;
        #pragma omp parallel for
        for (i=k+1; i < n-1; i++)
            { for (j=k+1; j < n-1; j++)  A[i][j] = A[i][j] - A[i][k]*A[k][j]; /* elimination step */
                b[i] = b[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, …, 2, 1
Can be expensive

CSCE5160 March 04, 2019

CSCE 5160 Parallel Processing

A different version of Gaussian elimination in shared memory (parallelize k loop)

#pragma omp parallel for private (i,j)
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] = b[k]/a[k][k];
        a[k][k] = 1;
        #pragma omp barrier  #need a barrier, no default barrier here
        for (i=k+1; i < n-1; i++)
            { for (j=k+1; j < n-1; j++)  a[i][j] = a[i][j] - a[i][k]*a[k][j]; /* elimination step */
                b[i] = b[i] - a[i][k]*y[k];
                a[i][k] = 0;
            } #pragma omp barrier  #need a barrier, no default barrier here
    }

This version is similar to our MPI version
Frist thread does less work than last thread
We can explore “cyclical” assignment of loop iterations

CSCE5160 March 04, 2019
Schedule clause

We talked about assigning consecutive rows of A to a process (in row striping) or cyclically assigning row to processors.

We can do the same thing in OpenMP using Schedule clause to specify how iterations of a loop are assigned to threads.

```plaintext
#pragma omp for [clause[, clause] ... for (int i = 0; i < 100; i++)
```

The clause can be one or more of the following: private(...), firstprivate(...), lastprivate(...), linear(...), reduction(...), schedule(...), collapse(...), ordered[...], nowait, allocate(...)

A Schedule for an OpenMP loop

```plaintext
#pragma omp parallel for schedule([modifier [modifier]:]kind[,chunk_size])
```

A specification of how iterations of loops are divided into contiguous non-empty subsets. Each of the contiguous non-empty subsets a chunk. And how these chunks are distributed to threads of the team. The size of a chunk, denoted as chunk_size must be a positive integer.

Most OpenMP systems support the following schedule clauses

Static: loop iterations are assigned at the very beginning of program execution. If chunk is specified, each thread receives that many consecutive iterations. You can specify chunk size of 1 to achieve cyclical

```plaintext
#pragma omp parallel for schedule(static,1)
```

If you omit chunk size, the system divides the number of iterations by # of threads

Dynamic: iterations are assigned either in chunks or one at a time, only when a thread completes assigned iterations. For example suppose we have 100 iterations and 5 threads. We can initially assign each thread 5 iterations. And assign additional 5 threads when a thread completes

good for load balancing
CSCE 5160 Parallel Processing

Most OpenMP systems support the following schedule clauses

Guided: Similar to dynamic, but the system assigns \( \frac{1}{2} \) of the work to threads first
Then one half of the remaining work (1/4) and so on

Auto: The run time system decides how to schedule iterations
We found that thread zero tends to be assigned more work

User: More involved but you can write your own scheduling methods

#pragma omp parallel for schedule(user, staggered:chunkSize: dynamicFraction:ltr)
For (int i = 0; i < n; i++) c[i] = a[i] * b[i]

Here we are specifying “user” scheduling
Staggered is like dynamic/guided -> not allocate all iterations at once
dynamic fraction indicates how the work is distributed

In order for this to work, user need to provide additional libraries (compiled and included)

CSCE 5160 Parallel Processing

Implementation of User-defined Schedule

When a user specifies a schedule kind user and a strategy named X (eg. Staggered)

Need to link a library that defines functions:

\( X\_\text{loopStart}() \)
\( X\_\text{loopNext}() \)
\( \text{and } X\_\text{init}() \)

\( X\_\text{init}() \) allows a user-level scheduler to allocate and initialize its data structures that are to be used commonly across parallel loops that use X.
The functions \( X\_\text{loopStart}() \) and \( X\_\text{loopNext}() \) determine a loop’s indices that a thread should work on based on the parameter values for the scheduling strategy and of the loop.

As long as you are allowed to define these functions, one can implement a user-defined scheduler. Every thread should call \( X\_\text{loopNext}() \) repeatedly to obtain next assignment

Every thread executes when starting a new loop:
\( \ldots X\_\text{loopStart}(); \text{do } X\_\text{loopNext}(); \text{ until done; } \) // done flag is set by the user-defined scheduler functions

CSCE 5160 March 04, 2019 9

CSCE 5160 Parallel Processing

CSCE5160 March 04, 2019 10
Another reason to worry about Scheduling

Consider how array/matrix elements are laid out
- Row Major vs Column Major
  - All element of a row are in consecutive locations
  - Spatial locality → good for caches

So, if an algorithm is going access elements along rows, use “Chunk” scheduling

On the other hand, if an algorithm accesses columns, then elements are not in consecutive location
  → may consider cyclical scheduling

Homework #6

Implement Gaussian Elimination using OpenMP and Static, Cyclical distribution of iterations
Need to generate input data to be used (you may be able find example input matrices online)

Or create randomly: make sure that the diagonal element is dominant (or at least largest value)
For example you can make all elements of a row “1” and the diagonal element as the sum of all the other elements

Midterm Review

Covers Through Chapter 8

Chapter 8: Dense Matrix Algorithms
  - Vector * Vector multiplication
  - Matrix * Vector
  - Matrix * Matrix
    - Cannon’s algorithm
    - Fox’s algorithm

Solving Linear equations
  - Gaussian Elimination
  - MPI implementation and OpenMP implementation

Chapter 7: Open MP

Chapter 6: MPI
CSCE 5160 Parallel Processing

Chapter 5: Performance Measurement
Execution Time
Speed UP
Efficiency
Amdhal’s Law
Scalability
Gustavson’s law
Iso-Efficiency
Scaled speedup (keep constant time, scale work with # processors or amount of memory)

Chapter 4: Communication Operations
One-to-one – costs on different networks
One-to-all (broadcast)
One-to-all-different-messages (scatter)
All-to-all
All-to-one (reduction or gather)

Chapter 3
Decomposing a problem
Recursive, data decomposition (input data and output data)
Exploratory (dividing search space, AI algorithms)
Speculative decomposition
Task graphs
Task interactions and dependencies
Critical paths
Mapping and load balancing
Assigning tasks to threads/processes

Chapter 2: SIMD vs MIMD
Cache and Memory limiting performance
Latency and bandwidth
Performance using multithreading and prefetching
to hide memory latency
Data parallel and Task Parallel
Interconnection networks and topologies
CSCE 5160 Parallel Processing

Problems from Chapter 8

8.2. Matrix vector multiplication but matrix is column striped

Each processor needs to broadcast its column values to other processors

We have all to all broadcast of n/p*n elements (n/p columns to each processor)

From chapter 4 (page 165) for hypercube

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

Computation cost is n^2/p multiplications and additions

8.4. Here we are looking at Matrix *vector using 2D partitioning (checkerboard)

\[ T_0 = t_s \log(p) + t_w n \left( p^{1/2} \right) \log(p) \]

Equation 5.14 (page 215): \( W = K \cdot T_0 \cdot (W, p) \)

\[ W = K t_s \log(p) + t_w n \left( p^{1/2} \right) \log(p) \]

Solving for \( W = O(n^2) \)

\[ W = \frac{K t_s p \left( \log(p) \right)^2}{2} + K t_w p \log(p) + \frac{K t_s \log(p) \left( \log(p) \right)^2}{2} + 4 \left( t_s, \log(p) \right) \]

Note the first term \( K t_s \log(p) \left( \log(p) \right)^2 \) is greater than the isoefficiency function \( O(p \left( \log(p) \right)^2) \)

for practical values of \( t_s, K \) \( \Leftrightarrow \) \( E/(1-E) \) and \( p \)

CSCE5160 March 04, 2019

---

CSCE 5160 Parallel Processing

8.7. This is related Gaussian Elimination using pipelined communication

We are trying find out the distance between two successive divisions

Let the division operation on the \( i^{th} \) row (0 ≤ i < n) be performed during time step \( t_i \).

Then in time step \( t_i + 1 \), the active part of the \( i^{th} \) row after division is sent to the processor storing the \( (i + 1)^{st} \) row.

In time step \( t_i + 2 \), the processor storing the \( (i + 1)^{st} \) row sends this data to the processor storing the \( (i + 2)^{nd} \) row.

In time step \( t_i + 3 \), the elimination operation is performed on the \( (i + 1)^{st} \) row.

And in time step \( t_i + 4 \), the division operation on the \( (i + 1)^{st} \) row.

In other words, \( four \) \( time \) \( steps \) separate the division operations on any two consecutives rows.

Hence, in \( 4n \) steps, all the \( n \) division operations can be performed and Gaussian elimination can be completed on an \( n \times n \) matrix.

For the last row we do not have any communication

Thus, the actual number of steps \( 4(n - 1) \).

CSCE5160 March 04, 2019
8.12. Here we want to find out the idle time of processors.

Note after first step, processor P_0 does not participate.
Likewise after step 2, P_1 stops. Also P_i has one less division to perform and processor P_i has only n-i divisions and n-1 subtractions.

If you add all the times when a processor is busy and not busy we will find that the idle times is O(n^3).

We can also derive this from execution times:

Sequential = O(2n^2/3)
Parallel = O(n^3/p) --- so the total Processor*time = O(n^3)

Problems from Chapter 5.

5.1. Amdahl’s law – serial portion limits the speedup

\[
T_p = W_s + (W-W_s)/p \\
T_s = W
\]

Speedup = \(T_s/T_p = W/[W_s+(W-W_s)/p] = (W/[W_s+(W-W_s)/p]\)

As p tends to infinity, the above equation becomes equal to W/W_s.

5.2. We discussed this in class regarding superlinear speed up when using exploratory decomposition.

5.10 Prefix sum. What is it?

Sum of all "previous" numbers (page 168)

For k = 0, 1, … p-1 for p processors

\[
S_k = \sum_{i=0}^{k} x_i
\]

Note this is slightly different than adding n numbers

We need to add numbers sequentially (not like a tree)

So, we have n-1 steps of communication plus addition

Execution time = (1+10)*(n-1)

Sequential execution is = n-1

Not cost optimal

---
CSCE 5160 Parallel Processing

5.11 We can use our summation of n numbers.

First, each processor performs a local prefix sum of its \( n/p \) numbers in \((n/p - 1)t_{add}\) time.

In the second step, the \( p \) processors compute prefix sums of \( p \) numbers by using the last prefix sum resulting from the local computation on each processor.

Note that this second step is like adding \( n \) numbers (here adding prefix sum values)

This step takes \( t_{add} + t_s + t_w \) \( \log p \).

Finally, the result of the parallel prefix sums operation of the second step is added to all the \( n/p \) prefix sums of the first step at each processor in \( t_{add}\)\(n/p\) time. Therefore,

\[
T_p = (2^*(n/p - 1)*t_{add} + (t_{add}+t_s+t_w) \log p
\]

Now we set \( t_{add} \) to 1 and \( t_s=10 \) (ignore \( t_i \))

CSCE 5160 March 04, 2019

---

CSCE 5160 Parallel Processing

4.17. we are asked to show that if a message \( m \) is divided into \( m/p \) messages, the all-to-one reduction can be performed in \( 2(t_s \log p + t_e m (p-1)/p) \)

Note we have \( p \) message each of size \( m/p \) being sent.

What is cost of all-to-all communication if each message size = \( k \)

On page 165, equation 4.4. states that all-to-all reduction will cost \( t_s \log p + t_e k^*(p-1) \)

Replacing \( k \) with \( m/p \), we get \( t_s \log p + t_e m^*(p-1)/p \)

To complete the all-to-one reduction, we need to “gather” the result of all-to-all reduction

The cost of scatter (and gather) is given on page 169, equation 4.6 as

\( t_s \log p + t_e k (p-1) \)

And substituting \( m/p \) for \( k \) we get \( t_s \log p + t_e m^*(p-1)/p \)

The total cost of all-to-one reduction = \( 2(t_s \log p + t_e m^*(p-1)/p) \)

We can view this as all-to-all reduction followed by gather operation
Now to problem 4.19
Consider the discussion on page 185. One to all broadcast using \( p \) messages of size \( m/p \) is
given as
\[
2*(t_s \log p + t_w m)
\]
This is achieved by treating the one to all broadcast as “scatter” to scatter \( p \) messages of
size \( m/p \) followed by all-to-all broadcast (each sends its \( m/p \) message to others).

All to one (reduction) is a dual of one-to-all (just reverse the direction of the messages).
So the cost is still the same \( 2*(t_s \log p + t_w m) \)
(or we can view this as a all-to-all broadcast followed by scatter as mentioned in the hint)

4.19 continued
Consider another option
Instead of dividing the message into \( p \) messages of \( m/p \); we will divide them into \( \log p \)
messages of \( m/ \log p \).

Consider one to one communication
We will not use the same route for each message. Each message is sent on a different
dimension.

The slowest message takes a path of length \( \log p \)
But we will be sending messages in every dimension – so we will have a total of \( 2^\log p \) steps of sending messages.

So we have \( 2^\log p *(t_s + t_w m/\log p) = 2^\log p *(t_s + t_w m) \)
But this algorithm will be the same as “one-to-all” broadcast.
CSCE 5160 Parallel Processing

Chapter 3

3.3. Consider the matrix multiplication using checkerboard decomposition

\[
\begin{array}{ccc}
A_{1,1} & A_{1,2} & B_{1,1} & B_{1,2} \\
A_{2,1} & A_{2,2} & B_{2,1} & B_{2,2} \\
\end{array} + \begin{array}{ccc}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2} \\
\end{array} = \begin{array}{ccc}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2} \\
\end{array}
\]

If we are looking at shared memory – where both A and B matrices are shared
There is no difference in terms of critical paths and concurrency for the two decompositions on page 99. Both task graphs look like the figure below and have a critical path of 2 and concurrency of 4

This task graph shows dependencies including communication cost.

For decomposition 1 assume checkerboard distribution and 4 processors – 2 tasks per processor

P0: Task1: \( C_{11} = A_{11} \ast B_{11} \)
P0: Task 2: \( C_{12} = C_{11} + A_{12} \ast B_{21} \)
P1: Task3: \( C_{12} = A_{11} \ast B_{12} \)
P1: Task 4: \( C_{12} = C_{12} + A_{12} \ast B_{22} \)
P2: Task5: \( C_{22} = A_{22} \ast B_{22} \)
P2: Task 6: \( C_{22} = C_{22} + A_{22} \ast B_{22} \)
P3: Task7: \( C_{22} = A_{22} \ast B_{12} \)
P4: Task 8: \( C_{22} = C_{22} + A_{22} \ast B_{22} \)

This task graph shows dependencies including communication cost.
Now consider decomposition 2, same checkerboard distribution and 4 processors

For decomposition 2

Task 1: \( C_{11} = A_{11} \times B_{11} \)
Task 2: \( C_{21} = C_{11} + A_{12} \times B_{21} \)
Task 3: \( C_{12} = A_{12} \times B_{22} \)
Task 4: \( C_{22} = C_{12} + A_{11} \times B_{22} \)
Task 5: \( C_{21} = A_{22} \times B_{21} \)
Task 6: \( C_{12} = C_{21} + A_{21} \times B_{11} \)
Task 7: \( C_{22} = A_{22} \times B_{12} \)
Task 8: \( C_{22} = C_{22} + A_{22} \times B_{22} \)

Now the communication is slightly different

If you distribute A and B differently (say row striping of A and column striping of B), you may have to wait for different data items