CSCE 5160 Parallel Processing

HW #9  10.2, 10.3, 10.7  Due April 17

Review
Completing Graph Algorithms
  Maximal Independent Set
  Johnson’s shortest path algorithm using adjacency lists

\{
  Q= V;
  for all v in Q
    l[v] = infinity;  #source node
  l[s] = 0;
  while Q is not empty
    u= extract_smallest_from_Q;  #find a node u such l[u] is min
    for v in Adjacency_list of u {z
      if v is in Q and l[u]+w(u,v) < l[v] then
        l[v] = l[u]+w(u,v);  }
\}

We keep a priority queue Q such that u where l[u] is the smallest is at the head of the queue.

CSCE 5160 Parallel Processing

Other MPI functions of interest – non blocking send and receive

MPI_Isend and MPI_Irecv are non blocking send and receive

int MPI_Isend (void *buffer, int cnt, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Request *handle)

int MPI_Irecv (void *buffer, int cnt, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Request *handle)

Pointer to object that identifies communication operation

The first 6 parameters are the same. We have a new argument called MPI_Request
The handle object will receive the status of the receive (or send)
Receiver cannot immediately access buffer. Must first check the status
You need to use MPI_Wait before accessing the receive buffer

\[ \text{int MPI\_Wait (MPI\_Request } *\text{handle, MPI\_Status } *\text{status)} \]

Note the process executing MPI_Wait blocks until the message communication is complete.

One more function

\[ \text{int MPI\_Probe (int src, int tag, MPI\_Comm comm, MPI\_Status } *\text{status)} \]

This function allows you check if a message with “tag” from sender “src” is received (or will be blocked) – you can use Probe or Wait.

You can use MPI\_ANY\_SOURCE and MPI\_ANY\_TAG.

For our purpose

Manager:

repeat (until no more work)

\{ . Lock 
  . Check if a request for work exists 
  . If not, Unlock 
  . Check if work remains 
  . Set work 
  . If not, “kill worker threads” or flag that no more work 
  . Unlock \}

Worker

repeat (no more work)

\{ . Lock 
  . set request 
  . Unlock 
  . Lock 
  . Get work if exists and Unlock 
  . If no work, unlock and try again 
  . If work, do work 
  \}

An generic example

\text{omp\_lock } *\text{my\_lock}; \text{omp\_int\_lock } (&\text{my\_lock});

\text{#pragma omp parallel private(tmp, id);} 
\{ 
  \text{id = omp\_get\_thread\_num();} 
  \text{tmp = result\_of\_lots\_of\_work(id);} 
  \text{omp\_set\_lock}(&\text{my\_lock}); 
  \text{print("%d %d", id, tmp);} 
  \text{omp\_unset\_lock}(&\text{my\_lock}); 
\} 
\text{omp\_destroy\_lock}(&\text{my\_lock});
Yet another model for parallelism

Pipelined model of parallelism

\( P_0 \) performs some computation and sends results to \( P_1 \)
\( P_i \) computes its computation and passes results to \( P_{i+1} \)

We have seen one example for sorting

\( P_0 \) reads next input number, keeps small value
\( \text{(min or new value and previously stored value)} \)
and sends larger value to \( P_1 \)
\( P_1 \) likewise keeps the smaller value and sends larger to \( P_2 \)

Another Example of Pipelined parallelism can be applied to find prime numbers

Let us consider the algorithm known as the Sieve Eratosthenes

1. Create list of unmarked natural numbers 2, 3, \( \ldots \), \( n \)
2. \( k = 2 \) is the first in the list
3. Repeat
   (a) Mark all multiples of \( k \) between \( k^2 \) and \( n \)
   (b) set \( k \) to the smallest unmarked number > \( k \)
until \( k^2 \) > \( n \)
4. The unmarked numbers are primes

Iterative solutions

In iterative methods, we use start with a guess, and then iteratively updated the guess until a reasonably acceptable solution is obtained. Consider our set of linear equations.

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

Assuming that the diagonal element of the \( A \) matrix are non-zero, we can solve the equations for \( x_i \) as follows

\[
x_i = \left( \frac{1}{a_{ii}} \right) [b_i - \sum_{j \neq i} \left( a_{ij} * x_j \right)]
\]

We need an initial estimate of \( x_i \) and we then continuously improve the estimate.
Jacobi Iterative Solver

What should be our initial guesses?
We can start with all zeroes or we can start with $x_i = b_i$.

for ($i=0; i < n; i++)$ $x[i] = b[i];$

repeat /*until termination condition is met
  for ($i=0; i < n; i++)$
    {
      sum = 0.0;
      for ($j=0; j < n; j++)$
        {if ($i != j)$ sum = sum + $a[i][j] * x[j];$}
      new_x[i] = $b[i] - sum/x[i];$
      for ($i=0; i < n; i++)$
        { $x[i] = new_x[i];$ }
  until /* test for termination condition

We can eliminate the test in the inner loop with a minor change to sum

for ($i=0; i < n; i++)$ $x[i] = b[i];$

repeat /*until termination condition is met
  for ($i=0; i < n; i++)$
    {
      $sum = -a[i][i] * x[i];$
      for ($j=0; j < n; j++)$
        { $sum = sum + a[i][j] * x[j];$}
      new_x[i] = $b[i] - sum/x[i];$
      for ($i=0; i < n; i++)$
        { $x[i] = new_x[i];$ }
  until /* test for termination condition

How can we parallelize this?
Let us start with shared memory version first.
Suppose we have n threads. We can assign the responsibility of computing an \( x[i] \) to each processor \( p_i \).

1. **Initialization step can proceed in parallel,** \( \{ \text{for } (i=0; i < n; i++) \ x[i] = b[i]; \} \)
2. **Repeat**
   a. All threads can compute new \( x[i] \) in parallel.
      We must wait until all threads complete their computations using a barrier.
   b. All threads update their \( x[i] \)’s.
      Again we wait until all threads complete their updates using a barrier.

Let us consider how a MPI version works?

We can assign the computation of an \( x_i \) to a different processor
Now we need to broadcast new \( x_i \) values to all processes we have a *all to all broadcast*

---

**Will the algorithm converge?**

Only if the matrix \( A \) is diagonally dominant

\[
\| a_i \| = \sum_{j \neq i} |a_{ij}| \quad 10 \leq i < n
\]

But we do not know how many steps it takes to converge.

Consider a variation — **Gauss-Seidel Approach**

Consider the key step in Jacob algorithm

\[
x[i] = \frac{1}{A[i,i]} (b[i] - \sum_{j \neq i} A[i,j] \times x_{k-1}[j])
\]

If you are executing this equation sequentially, by the time you are dealing with variable "i", you already computed \( x_k[0], x_k[1], \ldots x_k[i-1] \)

Why not use these new values instead of values from \( k-1 \)th iteration?
How do we parallelize this idea?

We need to wait to get updated values of \(x_k[j], j=0, 1, \ldots i-1\)

This does not lead to very efficient implementation

But we can consider computing the second summation for \(j=i+1, \ldots n-1\) while waiting

Also, may want to pipeline computing the first summation as you receive \(x_k[j]\)

If we have sparse matrices, then we only have to wait for those \(x[j<i]\) values where \(A[i,j]\) is Non-zero.

\[
x_k[i] = \frac{1}{A[i,i]} (b[i]) - \sum_{j=0}^{j=i-1} A[i,j] \cdot x_k[j] - \sum_{j=i+1}^{j=n-1} A[i,j] \cdot x_{k-1}[j]
\]
CSCE 5160 Parallel Processing

\[ h^2 f = x[i+1,j] + x[i-1,j] + x[i,j+1] + x[i,j-1] - 4 * x[i,j] \]

In general we can have other coefficients in the equation such as

\[ f_{ij} = a_{ij} x[i+1,j] + b_{ij} x[i-1,j] + c_{ij} x[i,j+1] + d_{ij} x[i,j-1] - e_{ij} x[i,j] \]

Consider this as one equation per grid point.
If we have \( n \) grid points we will have \( n \) equations.

Consider a matrix describing the \( n \) equations.
Each row of the matrix will have only 5 non zero “coefficients” describing the equation relating the 4 neighboring points and the grid point being computed.
Such matrices are called block-tri-diagonal since they look as blocks with diagonal elements containing non zero value.

Note that the off-the-main diagonal elements are at a distance dependent on how the grid points are numbered.
It is not always possible to get such structure to sparse matrices.
We will discuss other types of sparse matrices later

Let us consider a numbering scheme called RED-BLACK
CSCE 5160 Parallel Processing

Consider the red-black ordering (or coloring) of grid points

The grid points are first colored as red and black points -- alternating both in row and column coloring. All red points are numbered in natural order first from 0 to \((n/2)-1\). Then all black points are numbered in natural order from \((n/2)\) to \(n-1\).

The first \(n/2\) rows have no elements below the principal diagonal and the second half of the rows have no elements above the principal diagonal.

The Gauss-Seidel method executes each iteration (to compute \(x_k[i]\)) in two steps. Compute for its red points using the values from previous iteration for black points. Each processor sends the \(x_{k-1}[i]\) values for its black points to its neighbors (according the grid position not according to matrix rows).

After computing \(x_k[i]\) values for red points, we compute \(x_k[i]\) values for black points. Each processor sends \(x_k[i]\) values of its red points to its neighbors.

How many red or black points are at the boundaries? \(\lfloor n/p \rfloor^{1/2}/2\)

Each communication in a mesh network (to send values of red or black points) takes \(4*t_s+4*t_w\) \(\lfloor (n/p)^{1/2} \rfloor /2\)

The 4 in the above equation is because of 4 neighbors that a processor may have.

Total communication for the 2 phases is given by \(8*t_s+4*t_w\) \(\lfloor (n/p)^{1/2} \rfloor\)
CSCE 5160 Parallel Processing

What is the computation time?
Each processor has n/p rows, and in each row we have at most 5 non zero elements -- 4 off diagonal elements.

So assuming $t_c$ as the time for one multiplication and addition, the computation time is equal to $4*t_c*(n/p)$ (because we will have 4 Mult+Add)

Total execution time = $4*t_c*(n/p)+ 8*t_s + 8 *t_w ([((n/p)^{1/2} )/2 )$

What if we use more than 2 colors?

**Once again our goal is to minimize communication.**

If we represent the grids as a graph, and if the graph is planar we know that at most 4 neighbors (or colors) are possible.

If not a planar we may need more colors.

Once we have a graph representation of tasks, we can map these tasks to processors and minimize communication.

---

**Conjugate Gradient method**

A bit complicated in terms of mathematics. I will only outline the approach.

If you are interested in the details of the theory behind it, check this site

www.cs.cmu.edu/~jrs/jrspapers.html

In simple terms, if we are trying to solve $A*x = b$
Consider the following equation $q = (1/2) x^T * A * x - x^T * b + c$

$C$ is some normalizing constants. For our purpose we will set $c=0$

**Note A is a matrix, b and x are vectors ($x^T$ is the transpose of x).**

If $A$ is symmetric and positive definite (for any vector $x$, $x^T * A * x > 0$)

the minimum value of $q$ is the solution to $x$ in $A*x = b$
$q$ is known as a quadratic form.
Conjugate Gradient method
We will try to find the minimum value of \( \phi \) iteratively

In order to do this, CG uses
a direction vector – in which (vector) direction should we move in the next step
a step vector – how large is the next step to move towards the minimization

Consider one example with two unknowns
(x has 2 elements)
The figure shows a plot of
\[ q = \frac{1}{2} x^T A x - x^T b + c \]
(this is called a quadratic function and c =0)
\( q \) is minimum at \((2, -2)\) which is the solution
for \( x1=2, x2 = -2 \)

This graph plots the contours of \( q \).
The minimum value is shown by the dot
And at the minimum point, we will note
that the solution for \( x = [2,-2] \)

This figure shows the “direction” of the gradient for each contour in previous figure
But this direction actually moves you away from
minimum (size of arrow will be step size)
So we need to move in the opposite direction
Conjugate Gradient method

\[ x(t) = x(t-1) + s(t)*d(t) \]

In other words, the new value of \( x \) is a function of the current value of \( x \), a step value \( s \) and a direction vector \( d \) (we use \( t \) to indicate time step or next updated value).

Each iteration involves 4 steps (here \( i \) is associated with \( x_i \)).

But first let us initialize the following values:

- \( x(t)[i] = 0 \) for all \( i \) (in first iteration)
- \( d(t)[i] = 0 \) direction vector
- \( g(t)[i] = -b[i] \) gradient

In each iteration we perform the following steps:

**Step 1:** new \( g(t) = A*x(t-1) - b \)

Note \( A \) is a \( n \times n \) matrix, \( b \), \( d \), \( x \) and \( g \) are \( 1 \times n \) vectors.

**Step 2:** compute new direction vector

\[ newd = -g + \frac{(newg^T) * (newg)}{(oldg)^T * (oldg)} * oldd \]

Note \( g^T \) means \( g \) transpose and \( g^T * g \) mean the inner product of the vectors.

Also we need to keep three sets of values for each \( g \) (vector) element:

- Old: from previous iteration
- g: current value
- Newg: newly computed value

**Step 3:** Compute new step size

\[ news = -\frac{(newd)^T * (newg)}{(newd)^T * A * (newd)} \]

**Step 4:** compute new value for \( x_i \)'s

\[ new \ x = old \ x + news * new \ d \]
Pretty complicated – but efficient since we know that the solution converges in \( n \) or fewer steps, where \( n \) is the number of unknowns

How to parallelize?

We already know how to parallelize Matrix * Vector product

And we are dealing with sparse matrices

(I will talk about matrix * vector products for sparse matrices)

And the primary computations involve Matrix * Vector products

For the class project you can consider writing parallel implementations of CG – both OpenMP and MPI versions.

You can get source code for sequential versions from Matlab or online.

And see if you find information on cache misses, communication overhead etc.

Also, I believe there are test matrices available online or methods to create randomly large matrices that are symmetric and positive definite
How to represent the sparse matrix?

1. Coordinate form

```
 1  0  0  2  0  3
 4  5  0  0  0  0
 0  6  7  0  0  8
 9  0  0 10 11 12
 0 13  0  0 14  0
 0  0  0  0  0 15
```

\(\text{VAL} \quad 8 \quad 6 \quad 12 \quad 1 \quad 15 \quad 14 \quad 9 \quad 2 \quad 3 \quad 5 \quad 13 \quad 4 \quad 11 \quad 7 \quad 10\)

\(\text{I} \quad 2 \quad 2 \quad 3 \quad 0 \quad 5 \quad 4 \quad 3 \quad 0 \quad 0 \quad 1 \quad 4 \quad 1 \quad 3 \quad 2 \quad 3\)

\(\text{J} \quad 5 \quad 1 \quad 5 \quad 0 \quad 3 \quad 4 \quad 0 \quad 3 \quad 5 \quad 1 \quad 1 \quad 0 \quad 4 \quad 2 \quad 3\)

Here we have 3 arrays. The first array VAL contains only the non zero elements. The other two arrays I and J define the I,J subscripts for the values. Note: we can store the values in any order.

2. Compressed Sparse row format

```
 1  0  0  2  0  3
 4  5  0  0  0  0
 0  6  7  0  0  8
 9  0  0 10 11 12
 0 13  0  0 14  0
 0  0  0  0  0 15
```

\(\text{VAL} \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15\)

\(\text{I} \quad 0 \quad 3 \quad 5 \quad 0 \quad 1 \quad 1 \quad 2 \quad 5 \quad 0 \quad 3 \quad 4 \quad 5 \quad 1 \quad 4 \quad 5\)

\(\text{J} \quad 0 \quad 3 \quad 5 \quad 8 \quad 12 \quad 14\)

Here too we have 3 arrays, VAL, I and J. The non zero elements in VAL are ordered by rows. We first store all non zero values from row 0, Then all non zero values from row 1, etc.
3. Diagonal Storage format

Here we will assume that any row has no more than \( d \) element around the diagonal.
We store all non-zero elements in \( n \times d \) matrix.
-- only non-zero elements from each row are stored.
The OFFSET array \((d+1)\) stores the offsets for the diagonal elements.

Offset -3 shows that the first non zero values in VAL are from a diagonal which is
3 positions to the left of the principal diagonal.

\[
\begin{array}{cccccc}
1 & 2 & 0 & 0 & 0 & 0 \\
3 & 4 & 5 & 0 & 0 & 0 \\
0 & 6 & 7 & 0 & 0 & 0 \\
8 & 0 & 9 & 10 & 11 & 0 \\
0 & 13 & 0 & 0 & 14 & 15 \\
0 & 0 & 16 & 0 & 17 & 18 \\
\end{array}
\quad
\begin{array}{cccc}
- & - & 1 & 2 \\
- & 3 & 4 & 5 \\
- & 6 & 7 & 0 \\
8 & 9 & 10 & 11 \\
13 & 0 & 14 & 15 \\
16 & 17 & 18 & - \\
\end{array}
\quad
\begin{array}{cccc}
\text{OFFSET} \\
-3 & -1 & 0 & 1 \\
\end{array}
\]

4. Ellpack-Itpack format

This is similar to Diagonal Storage format but more general.
Good if no row has more than \( m \) non zero elements
and the average number of non-zero elements per row is close to \( m \).

We use two \( n \times m \) matrices VAL and J.

VAL stores all non zero elements of the original matrix.
J indicates the column position for the non zero values.
-1 in J matrix indicates the end of non-zero elements in that row.

\[
\begin{array}{cccccc}
1 & 2 & 0 & 0 & 0 & 0 \\
3 & 4 & 5 & 0 & 0 & 0 \\
0 & 6 & 7 & 0 & 0 & 0 \\
8 & 0 & 9 & 10 & 11 & 0 \\
0 & 13 & 0 & 0 & 14 & 15 \\
0 & 0 & 16 & 0 & 17 & 18 \\
\end{array}
\quad
\begin{array}{cccc}
1 & 2 & - & -
3 & 4 & 5 & -
6 & 7 & - & -
8 & 9 & 10 & 11
13 & 14 & 15 & -
16 & 17 & 18 & -
\end{array}
\quad
\begin{array}{cccc}
0 & 1 & -1 & 0
0 & 1 & -1 & -
1 & 2 & -1 & -
0 & 2 & 3 & 4
1 & 4 & 5 & -1
2 & 4 & 5 & -1
\end{array}
\]

Note: the trade-off we are looking at is: the memory needed for storing the matrix versus the
amount of time needed to access the elements.
We have a block around the principle diagonal (with max of 3 elements per row) and we also have two other blocks (each with a principle diagonal only).

Consider Matrix – Vector Product
Sparse Matrix (say in Diagonal representation) is distributed using row striping
Vector is also distributed

For this partitioning, we can use the Diagonal storage form

\[
\begin{array}{cccccccc}
\times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times \\
\end{array}
\]

Rows of VAL are distributed using stripping -- there is no need to distribute OFFSET since the location of the diagonal elements is fixed and known to all processors.

Now let us see if and what type of communication is needed between processors.

Note each processor will also receive "stripped" elements of the vector b in A*b

The computations along the principle diagonal, all values are local
CSCE 5160 Parallel Processing

To compute the necessary multiplications and additions, each processor needs one vector element from its neighbor above and one vector element from its neighbor below. This communication costs $2 \times (t_s + t_w)$ assuming one word communication.

Now we need to consider the computations involving the “outer” diagonal elements—the blocks above and below the principle diagonal blocks.

We need to get the vector elements to the processors containing these sparse matrix elements.

The communication cost for these steps depends on how far the outer diagonal elements are, and the value of $p$ in relationship to this value.

Let us assume that we will use $n^{1/2}$ as the distance—because of 2-D shape of grid points—\( n^{1/2} \) above and \( n^{1/2} \) below.

Let us estimate communication cost using this value.
This depends on how many processors we have.
CSCE 5160 Parallel Processing

If \( n/p \) is greater than \( n^{1/2} \) or \( p < n^{1/2} \)

- you communicate values with your two neighbors
- and you need to exchange \( n^{1/2} \) elements with your neighbor

\[
\text{Communication cost} = 2 \times (t_s + (n^{1/2}) \times t_w)
\]

\[
\text{Total cost} = 5 \times t_c \times (n/p) + 2 \times (t_s + t_w) + 2 \times (t_s + (n^{1/2}) \times t_w)
\]

If \( n/p \) is less than \( n^{1/2} \) or \( p > n^{1/2} \)

- processor \( p_j \) needs vector elements from processor \( j + or - p/ (n^{1/2}) \)
- Thus each processor must exchange all its \( n/p \) vector elements with processors located at a distance of \( p/ (n^{1/2}) \)
- This is like a shift (not circular) operation exchanging data among processors
  - Actually two shift operations, one in each direction.

\[
\text{Communication cost} = 2 \times (t_s + (n/p) \times t_w)
\]

\[
\text{Total cost} = 5 \times t_c \times (n/p) + 2 \times (t_s + t_w) + 2 \times (t_s + (n/p) \times t_w)
\]

4 processors (\( p=4 \)); 64\(^*\)64 matrix
\( n^{1/2} = 8 \)
- That is \( p < n^{1/2} \)
- For off diagonal, you need vector elements from your neighbor
- Need only \( n^{1/2} \) vector elements form neighbors

16 processors (\( p=16 \)); 64\(^*\)64 matrix
\( n^{1/2} = 8 \) that is \( p > n^{1/2} \)
- For off diagonal, you need vector elements from a node which is not a neighbor.
- And you need all \( n/p \) vector elements