

CSCE 5160 Parallel Processing

Homework #1: 2.2, 2.3, 2.12 2.13

Due Sept 10, 2009

Grader: Shu Chen

shuchen@my.unt.edu

Office Hours: Thursday 8-12

Project idea

Learn SequenceL

Simple tutorial on class website

More information and an Interpreter is available – see me if interested

Review

Interconnection networks

Mesh, multistage, Hypercube

Properties of networks

Impact on execution performance

Designing parallel programs

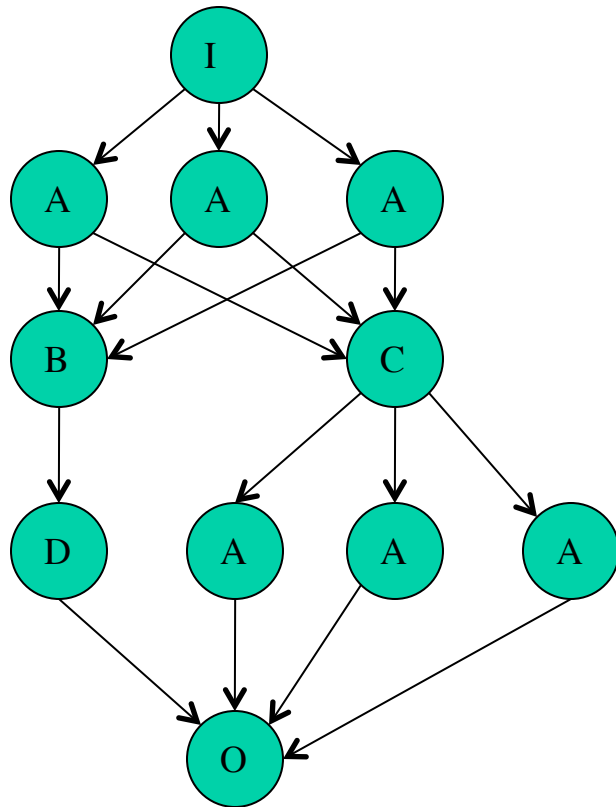
Task graphs to identify

data and task parallelism

communication patterns

CSCE 5160 Parallel Processing

An example



Data parallelism

All A's after I and after C

Task parallelism

B and C can be executed in parallel

D and A's can be executed in parallel

Task parallelism can be identified using

“leveling” algorithms

Communication needs

depends on how tasks are assigned to processors

CSCE 5160 Parallel Processing

Mapping algorithms to networks

If we assume that an algorithm can be defined as a graph representing communication requirements among the various sub-computations, each computation on a different processing node, then we can see how well this graph can be mapped onto a network
-- the network itself is represented as a graph.

In general we are considering mapping a graph $G(V,E)$ -- algorithm
onto another graph $G'(V',E')$ -- communication

It is possible that one link of E may be mapped to more than one link in E' (called **dilation factor**-- leads to delays due to more links that must be traversed);

or several links in E may be mapped to a single link of E' (called **congestion factor** and leads to delays due to congestion).

Likewise several nodes of V may be mapped to a single node in V' (called **expansion**, and this implies that one processor is executing several sub-computations).

CSCE 5160 Parallel Processing

Our goal is to take any algorithm which assumes one network of interconnection and map the algorithm to another physical interconnections

Let us see how we can map Hypercubes to a linear array.

Linear array nodes will be numbered 0, 1, 2, ... n-1

Hypercube nodes use as binary numbers

But one property of linear array is that two consecutive numbers are connected (are neighbors).

If we use straight binary coding, this will not happen.

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Let us look at nodes 1 and 2 (001 and 010) --

According to our hypercube routing -- Hamming distance
there is a distance of 2 between these nodes -- they are not neighbors.

Likewise between 3 (011) and 4 (100) -- in the hypercube
they are not neighbors -- but they are in the linear array.

CSCE 5160 Parallel Processing

So what do we do?

We can use **Gray codes** for this purpose -- actually called reflected Gray codes

0	000
1	001
2	011
3	010
4	110
5	111
6	101
7	100

How about mapping Hypercubes to Meshes -- say 2-D mesh?

Each dimension in the mesh will be treated as a linear array and we use reflected Gray code mapping.

Mapping Hypercubes to Trees?

We will digress from the textbook to learn how to use MPI and OpenMP

CSCE 5160 Parallel Processing

Introduction to MPI

```
#include "mpi.h"
#include <stdio.h>

int main( argc, argv )
int argc;
char **argv;
{
int rank, size;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
printf( "Hello world! I'm %d of %d\n",
        rank, size );
MPI_Finalize();
return 0;
}
```

CSCE 5160 Parallel Processing

Note that we concentrated only on the following MPI functions

```
int MPI_Init (int * argc, char ** argv[]);
int MPI_Finalize ();
int MPI_Comm_Size (MPI_Comm comm, int * number_of_processes);
int MPI_Comm_rank( MPI_Comm comm, int * my_process_id);
```

Let us look at two most basic communication functions in MPI

Blocking receive: receiver waits until the message is received but sending is non-blocking

In general you can have either blocking or non-blocking send and receive

```
int MPI_Send (
    void *      message_data,
    int         count,
    MPI_Datatype datatype,
    int         destination,
    int         tag,
    MPI_Comm   communicator);

int MPI_Recv (
    void *      message_data,
    int         count,
    MPI_Datatype datatype,
    int         source,
    int         tag,
    MPI_Comm   communicator,
    MPI_Status * status);
```

CSCE 5160 Parallel Processing

MPI provides a function

```
int MPI_Get_count(  
    MPI_Status *      status,  
    MPI_Datatype      datatype,  
    int *             count_ptr)
```

The count is returned in count_ptr

MPI_Status returns a structure containing

```
Status → MPI_SOURCE,  
Status → MPI_TAG  
Status → MPI_ERROR
```

CSCE 5160 Parallel Processing

MPI data types

MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_BYTE
MPI_PACKED

You can build your own data types like structures.
But when sending and receiving, the data will be send as a string of bytes.

MPI functions help in converting data structures into string of bytes and then reconstruct the data structure from string of bytes

How to measure performance
computation time
Communication Time

CSCE 5160 Parallel Processing

Two functions come in handy

MPI_Wtime
MPI_Wtick

MPI_Wtick tells about the resolution of
the clock

```
.....  
double start,end;  
...  
start = MPI_Wtime;  
....code being timed  
end =MPI_Wtime;
```

We can find the elapsed time from $(end) - (start)$.

Note you can only use these times on a single processor – not compare start time on one processor with end time on a different processor

CSCE 5160 Parallel Processing

Other Communication Functions

```
int MPI_Bcast (void *      message,  
              int        count,  
              MPI_Datatype data_type,  
              int        root,    /*sending process  
MPI_Comm      comm.    /*receivers  
              )
```

Note the difference between send and broadcast:

Instead of destination, we specify the sender (root).

There is no tag with message.

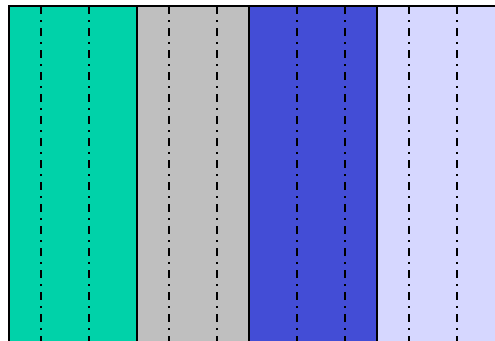
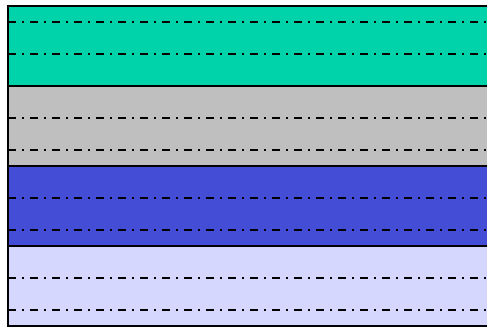
Each receiver will invoke the same MPI_Bcast message

The system looks at the “root” information and the process’s rank to determine if we are sending or receiving.

Since there is no tag, you should be careful in invoking the broadcast (and receive) in correct order.

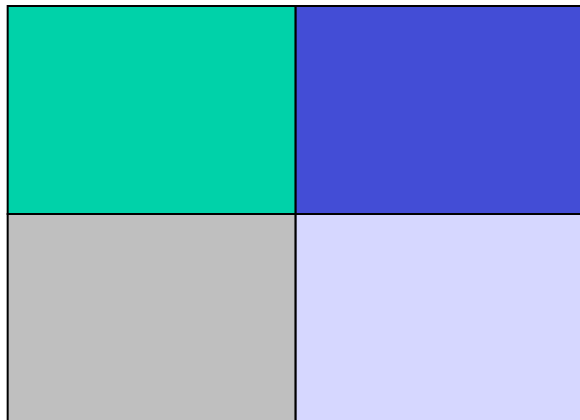
CSCE 5160 Parallel Processing

Data distribution. Consider distributing an array or matrix to various processes or processors.



Row/Column Striping

Checkerboard



Cyclical (for both striping and checkerboard)

Blocking (both striping and checkerboard)

CSCE 5160 Parallel Processing

Instead of

```
for (i=0; i<number_of_processors; i++) MPI_Send(.....);
```

We can use

```
int MPI_Scatter (void *      send_data,      /* data source
                  int        send_count,     /* count of items to be sent
                  MPI_Datatype send_type,    /* data type sent
                  void*      recv_data,      /* storage for received data
                  int        recv_count,     /* count of items to be received
                  MPI_Datatype recv_type,    /*data type received
                  int        sender,        /* sender or root
                  MPI_Comm   comm          /* communication group);
```

Both sender and **receivers** will execute the same function `send_data` is meaningless with receivers

Both `send_count` and `recv_count` is the number of elements sent to or received by one process

CSCE 5160 Parallel Processing

Likewise instead of

```
for (i=0; i<number_of_processors; i++) MPI_Recv(.....);
```

We can use

```
int MPI_Gather (void *      send_data,      /* data source
                 int        send_count,     /* count of items to be sent
                 MPI_Datatype send_type,    /* data type sent
                 void*      recv_data,     /* storage for received data
                 int        recv_count,    /* count of items to be received
                 MPI_Datatype recv_type,   /* data type received
                 int        receiver,     /* receiver or root
                 MPI_Comm   comm         /* communication group);
```

Executed by the **senders** and receiver (root)

Both send_count and recv_count is the number of elements sent to or received by one process.

recv_data must be large enough to hold all received data (n*recv_count)

CSCE 5160 Parallel Processing

Another very useful function -- reduce

We receive ONE data item from each receiver like Gather but we also specify an **operation** on the received data

For example, we want to find the sum of all the values received or find the minimum of all values received

We can use MPI_Reduce and specify an operation like MPI_SUM, or MPI_MIN, etc.

```
int MPI_Reduce (void *      recv_data,      /* data source
                 void*     accumulated_data, /* result
                 int       count,           /* per process count
                 MPI_Datatype send_type,    /* data type sent
                 MPI_Op   operation,      /* operation like, sum, min
                 int       receiver,       /* receiver or root
                 MPI_Comm  comm           /* communication group);
```

CSCE 5160 Parallel Processing

A few things to remember about Reduce

All processes execute the same command

The result (or accumulated_data) is only at the root

Count refers to amount sent by each node

Result stays with only root

Operations with MPI_Reduce

We can view MPI_Reduce as

```
for (j=1;....) {  
    MPI_Recv();  
    perform_accumulation;}  
}
```

MPI_MAX, MPI_MIN
MPI_MAXLOC
MPI_MINLOC
MPI_SUM, MPI_PROD
MPI_LAND (logical AND)
MPI_BAND (Bitwise AND)
MPI_LOR, MPI BOR
MPI_LXOR, MPI_BXOR

CSCE 5160 Parallel Processing

Yet another useful MPI function: ALL REDUCE

We can view this as MPI_Reduce followed by MPI_Bcast

Or that every node performs MPI_Reduce

```
int MPI_Allreduce (void *      send_data,          /* data source
                    void*     accumulated_data,    /* result
                    int       count,              /* per process count
                    MPI_Datatype send_type,       /* data type sent
                    MPI_Op     operation,         /* operation like, sum, min
                    int       receiver,          /* receiver or root
                    MPI_Comm   comm             /* communication group);
```

Now we need to allocate space for accumulated_data (result) at each node in addition to the data being sent.

CSCE 5160 Parallel Processing

Consider an example of using reduce. dot product

$$X * Y = b$$

Where X and Y are vectors (single dimensional arrays).

$$b = x_0 * y_0 + x_1 * y_1 + x_2 * y_2 + \dots + x_{n-1} * y_{n-1} \quad .$$

Simple serial version:

```
float dot_product (float x[], float y[], int n)
```

```
{  
  int i;  
  float sum = 0.0;  
  for (i=0; i< n; i++)  
    sum = sum + x[i]*y[i];  
  return sum;  
}
```

So how do we parallelize this?

Let us assume $p < n$ processes

CSCE 5160 Parallel Processing

Let us assume $p < n$ processes.

Each process computes a sum of n/p elements. Then we combine all these sums. Consider the following code fragment.

```
float parallel_version ( float local_x[], local_y[], int my_n)
{
    float local_sum;
    float total_sum = 0.0;
    float dot_product (float x[], float y[], int n);

    local_sum = dot_product (local_x, local_y, my_n);

    MPI_Reduce(&local_sum, &total_sum, 1, MPI_FLOAT, MPI_SUM,
              0, MPI_COMM_WORLD):
    return total_sum;
}
```

CSCE 5160 Parallel Processing

What assumptions are we making?

The p processes are assumed to receive a portion of the X and Y vectors

using blocking method; p_0 receives $x_0, \dots, x_{(n/p)-1}$,

p_i receives, $x_{i*(n/p)}, x_{(i*(n/p)+1)}, \dots, x_{(i*(n/p)-1)}$

This distribution is done by root (or process with rank=0) using either Scatter or simply Send messages in a loop

All processes execute the `MPI_Reduce`, however, only the process with rank=0 will accumulate the `total_sum` value.

All other nodes will only have `total_sum=0.0`

Note that we could have defined any process rank (instead of zero) as the place for returning the result

CSCE 5160 Parallel Processing

Consider an example use of Scatter.

In our dot matrix we assumed that each process receives n/p elements. We execute the following call on all processes

```
...      my_n = n/p;
          MPI_Scatter (&x,                               /* data source
                    my_n,                               /* count of items to be sent
                    MPI_FLOAT,                         /* data type sent
                    &local_x,                         /* storage for received data
                    my_n,                               /* count of items to be received
                    MPI_FLOAT,                         /*data type received
                    root,                               /* sender id
                    MPI_COMM_WORLD);

....      /* similarly, scatter Y elements.
```

Note all processes execute this call. Root contains the complete vectors X and Y. All processes (including root) will receive n/p elements.