Week 1 Discussion

UCSB CS140, Winter 2026

Table of Content

Useful gcc Options

- Include: -I<path>
- Define: -D<identifier>
- Optimization: -O<level>
- Library: -l<libraryname>

Example:

gcc -DDEBUG -O2 -I/usr/include example.c -o example -lm

```
void func(int nums[], int m) {
#ifdef DEBUG
    printf("input m is %d: ", m);
#endif
}
```

Macro Preprocessor pitfalls

- **Example: the "min" function**

```
int min(int a, int b)
 { if (a < b) return a; else return b; }
#define min(a,b) ((a) < (b) ? (a) : (b))
```

- **Different when evaluating expression has side-effect for min(a++,b)**
  - min function increments a once
  - min macro may increment a twice if a < b
- **Text substitution can expose unexpected groupings**

```
#define mult(a,b) a*b
mult(5+3,2+4)
```

- Expands to `5 + 3 * 2 + 4`
- To fix this, enclose each macro argument in parenthesis:

```
#define mult(a,b) (a)*(b)
```

How to avoid defining things twice?

- Example: include the same .h file twice

#include “myheader.h”

#include “file.h”

- Convention: surround each header (.h) file with a conditional:

```
/*File myheader.h*/
#ifndef __MYHEADER_H__
#define __MYHEADER_H__
/* Declarations */
#endif
```

Makefile for Exercise 1

```
CC=gcc
CFLAGS=-Wall –O2
CPPFLAGS=-DDEBUG
OBJECTS= exercise.o exercise_test.o minunit.o
TARGET=exercise

all: $(TARGET)
%.o : %.c exercise.h
	$(CC) -c $(CFLAGS) $(CPPFLAGS) $<

#Pattern rule $@ means the target file
$(TARGET): $(OBJECTS)
	$(CC) $(CFLAGS) $(OBJECTS) -o $@

run:
	./$(TARGET)
clean:
	rm $(OBJECTS)
	rm $(TARGET)
```

Usage:
make
make run
make clean

Pattern rule $< means the source file.

Minimum unit test for C

```
/*minunit.h  defines a statement*/
#define mu_assert(message, condition) do { \
        if (!(condition)) return message;  \
        } while (0)
```

do {...} while(0) just means {…} to group C statements

```
/*minunit.c*/
int  _mu_tests_run=0;   /*count #tests*/
int  _mu_tests_failed=0; /*count #tests failed */
char*  mu_run_test(char * (*test_fun)()){
        char *message = (*test_fun)();
        _mu_tests_run++;
        if (message){
              _mu_tests_failed++;
            }
        return message;
}
```

Run a test based on this function pointer

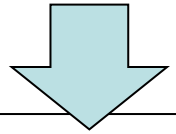
Non empty message means failed

Combine C statements in Macro with do{ …} while(0)

```
#define mu_assert(condition,msg)  { \
        if (!(condition)) return msg;  \
        }
```

Let's remove "do" "while(0)"

```
if (x>0)
   mu_assert("error 1", x>1);
else
   mu_assert("error 2", x<-1);
```



```
If (x>0)
   {if (!(x>1)) return "error 1";};
else
   {if (!(x<1)) return "error 2";};
```

gcc compilation error!

Use of mu_assert in HW1

```
struct key_action {
    char *cmd;
    int (*func)();
};
```

```
void set_key_action(
  struct key_action *rec,
  char *cmd, int (*f)()){
       if(rec!=NULL) {
              rec->cmd=cmd;
              rec->func=f;
             }
}
```

```
int del1(int x){
        return x-1;
}
/*Make a call and if result is unexpected, return
an error message. Otherwise return NULL */

char * set_key_action_test(void){
   struct key_action *rec= (struct key_action*)
            malloc(sizeof(struct key_action));
   char *key="del1";
   set_key_action(rec, key, del1);

   mu_assert("Error in set_key",
                    strcmp(key, rec->cmd)==0);
   mu_assert("Error in set_action",
                    rec->func == del1);
   /*All checkups  are valid so far*/
   return NULL;
}
```

How to fix the memory leak in this code?

Mistake in using mu_assert

```
struct key_action {
    char *cmd;
    int (*func)();
};
```

```
void set_key_action(
  struct key_action *rec,
  char *cmd, int (*f)()){
       if(rec!=NULL) {
              rec->cmd=cmd;
              rec->func=f;
             }
}
```

```
int del1(int x){
        return x-1;
}

char * set_key_action_test(void){
   struct key_action rec;
   char *key="del1";
   set_key_action(&rec, key, del1);

   char* msg=mu_assert("Error in set_key",
                       strcmp(key, rec.cmd)==0);
   return msg;
}
```

mu_assert is NOT a function. It is a macro!

HW1: Matrix-vector multiplication: y= A * x

$$\overset{A}{\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}} * \overset{x}{\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}} = \begin{pmatrix} 1*1+2*2+3*3 \\ 4*1+5*2+6*3 \\ 7*1+8*2+9*3 \end{pmatrix} = \begin{pmatrix} 14 \\ 32 \\ 50 \end{pmatrix}$$

```
/*exercise.c*/
void mat_vect_mult(
   double  A[], double  x[], double  y[],  int  m, int   n  ) {
   int i, j;
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++) {
         y[i] +=A[i*n+j]*x[j];
           }
    }}
```

Unit test for matrix vector multiplication

If the result is not expected. return a message. If successful, return NULL

```
char* matrix_vect_test(int m, int n) {
    double *A = malloc(m*n*sizeof(double));
    double *x = malloc(n*sizeof(double));
    double *y = malloc(m*sizeof(double));
      /*… initialize A, x*/
    mat_vect_mult(A, x, y, m, n);
     char *msg=test_vect(y, m, n);
     free(A); free(x); free(y);
    return msg;
}
char *test1(void){
      return  matrix_vect_test(2,4);
}
```

```
char*  test_vect(double y[], int m,
int n){
      int i;
      double expected= n*(n-1)/2;
      for (i = 0; i < m; i++){
            mu_assert("One error",
            y[i] ==expected);
            }
      return NULL;
}
```

```
void run_all_tests(void){
    mu_run_test(test1);
}
```

Valgrind debugging tool

- **Goal: detect memory errors**
  - Accesses outside of memory bounds
  - Memory leaks
- **Great for finding errors that would only show during harsh test cases**
- **Can you find two errors in this program?**

```
#include <stdlib.h>
void f(void) {
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;
}

int main(void) {
    f();
    return 0;
}
```

1. Invalid memory access

2. Memory never free()'d

Running Example in Valgrind

- **Running valgrind with the program:**
  - valgrind --leak-check=yes myprog arg1 arg2

- **Invalid access output (error 1):**

Process ID

==19182== **Invalid write** of size 4
==19182== at 0x804838F: f (**example.c:6**)
==19182== by 0x80483AB: main (example.c:11)

Where the error occurs

- **Memory leak output (error 2):**

==19182== **40 bytes in 1 blocks are definitely lost** in loss record 1 of 1
==19182== at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182== by 0x8048385: f (a.c:5)
==19182== by 0x80483AB: main (a.c:11)

Size of the leak

GDB, the GNU Debugger

- **Command-based. Compile with –g and invoked with:**
  gdb <executable file>
- **Example to reverse a string**

$ cc -g reversefirst.c -o reverse

$ gdb reverse

(gdb) help

<…>

Type "help" followed by a class name for a list of commands in that class.

Type "help" followed by command name for full documentation.

Command name abbreviations are allowed if unambiguous.

(gdb)

Prof. Andrzej (AJ) Bieszczad
Email: andrzej@csun.edu
Phone: 818-677-4954

Basic GDB Commands

- General Commands:

  file [<file>] selects <file> as the program to debug

  run [<args>] runs selected program with arguments <args>

  attach <pid> attach gdb to a running process <pid>

  kill kills the process being debugged

  quit quits the gdb program

  help [<topic>] accesses the internal help documentation

- Stepping and Continuing:

  c[ontinue] continue execution (after a stop)

  s[tep] step one line, entering called functions

  n[ext] step one line, without entering functions

  finish finish the function and print the return value

Prof. Andrzej (AJ) Bieszczad
Email: andrzej@csun.edu
Phone: 818-677-495416

GDB Breakpoints

- Useful breakpoint commands:

b[reak] [<where>]	sets breakpoints. <where> can be a number of things, including a hex address, a function name, a line number, or a relative line offset
[r]watch <expr>	sets a watchpoint, which will break when <expr> is written to [or read]
info break[points]	prints out a listing of all breakpoints
clear [<where>]	clears a breakpoint at <where>
d[elete] [<nums>]	deletes breakpoints by number

Prof. Andrzej (AJ) Bieszczad
Email: andrzej@csun.edu
Phone: 818-677-495417

Playing with Data in GDB

- **Commands for looking around:**

list [<where>]	prints out source code at <where>
search <regexp>	searches source code for <regexp>
backtrace [<n>]	prints a backtrace <n> levels deep
info [<what>]	prints out info on <what> (like local variables or function args)
p[rint] [<expr>]	prints out the evaluation of <expr>

- **Commands for altering data and control path:**

set <name> <expr>	sets variables or arguments
return [<expr>]	returns <expr> from current function
jump <where>	jumps execution to <where>

Prof. Andrzej (AJ) Bieszczad
Email: andrzej@csun.edu
Phone: 818-677-495418

gdb - Example

```
(gdb) list 1                                  ---> can use “l” for “list”
1       /* REVERSE.C */
2
3       #include <stdio.h>
4
5       /* Function Prototype */
6       void reverse ();
7
8       /*****************************************************************/
9
10       main ()
(gdb) l                                       ---> same as “list”; continues from the previous
11
12        {
13   char str [100]; /* Buffer to hold reversed string */
14
15   reverse ("cat", str); /* Reverse the string "cat" */
16   printf ("reverse (\"cat\") = %s\n", str); /*         Display */
17   reverse ("noon", str); /* Reverse the string "noon" */
18   printf ("reverse (\"noon\") = %s\n", str); /* Display */
19      }
```

Prof. Andrzej (AJ) Bieszczad
Email: andrzej@csun.edu
Phone: 818-677-495419

gdb - Example

```
(gdb) break main                    ---> set a breakpoint in function main
Breakpoint 1 at 0x29f4: file reversefirst.c, line 15.
(gdb) break 16                      ---> set a breakpoint in line 16 (current source)
Breakpoint 2 at 0x2a0c: file reversefirst.c, line 16.
(gdb) break reverse                 ---> set a breakpoint in function reverse
Breakpoint 3 at 0x2a80: file reversefirst.c, line 33.
(gdb) info break                    ---> display information on breakpoints
Num Type         Disp Enb Address    What
1   breakpoint   keep y   0x000029f4 in main at reversefirst.c:15
2   breakpoint   keep y   0x00002a0c in main at reversefirst.c:16
3   breakpoint   keep y   0x00002a80 in reverse at reversefirst.c:33
(gdb) run
Starting program: /home/usersNN/userID/reverse/reverse

Breakpoint 1, main () at reversefirst.c:15
15  reverse ("cat", str); /* Reverse the string "cat" */
(gdb) continue                      ---> you can use "c" as well

Breakpoint 3, reverse (before=0x40001028 "cat", after=0x7f7f0958 "")
    at reversefirst.c:33
33  len = strlen (before);
(gdb) _
```

Prof. Andrzej (AJ) Bieszczad
Email: andrzej@csun.edu
Phone: 818-677-495420

gdb - Example

```
(gdb) backtrace                                    ---> show the execution stack
#0  reverse (before=0x40001028 "cat", after=0x7f7f0958 "") at reversefirst.c:33
#1  0x2a0c in main () at reversefirst.c:15
(gdb) l
28 {
29  int i;
30  int j;
31  int len;
32
33  len = strlen (before);
34
35  for (j = len - 1, i = 0; j >= 0; j--, i++) /* Reverse loop */
36          after[i] = before[j];
37
(gdb) next                                         ---> execute next line
35  for (j = len - 1, i = 0; j >= 0; j--, i++) /* Reverse loop */
(gdb) n                                            ---> same as “next”
36          after[i] = before[j];
(gdb) _
```

Prof. Andrzej (AJ) Bieszczad
Email: andrzej@csun.edu
Phone: 818-677-495421

gdb - Example

```
(gdb) print after[i]                                    ---> display data (expression)
$1 = 0 '\000'
(gdb) p before[j]                                       ---> same as “print”
$2 = 116 't'
(gdb) _
(gdb) n
35  for (j = len - 1, i = 0; j >= 0; j--, i++) /* Reverse loop */
(gdb) p after                                           ---> print
$4 = 0x7f7f0958 "t"
(gdb) p before
$5 = 0x40001028 "cat"
(gdb) c
Continuing.
Breakpoint 2, main () at reversefirst.c:16
16  printf ("reverse (\"cat\") = %s\n", str); /*        Display */
17  (gdb) _
```

Prof. Andrzej (AJ) Bieszczad
Email: andrzej@csun.edu
Phone: 818-677-495422

gdb - Example

```
(gdb) n
reverse ("cat") = tac
17  reverse ("noon", str); /* Reverse the string "noon" */
(gdb) s
Breakpoint 3, reverse (before=0x40001030 "noon", after=0x7f7f0958 "tac")
    at reversefirst.c:33
33  len = strlen (before);
(gdb) return 0
Make reverse return now? (y or n) y
#0  main () at reversefirst.c:18
18  printf ("reverse (\"noon\") = %s\n", str); /* Display */
(gdb) p str
$4 = "tac", '\000' <repeats 96 times>
(gdb) n
reverse ("noon") = tac                                  ---> this is the output from
     the program
19      }
(gdb) quit
$ _
```

Prof. Andrzej (AJ) Bieszczad
Email: andrzej@csun.edu
Phone: 818-677-495423

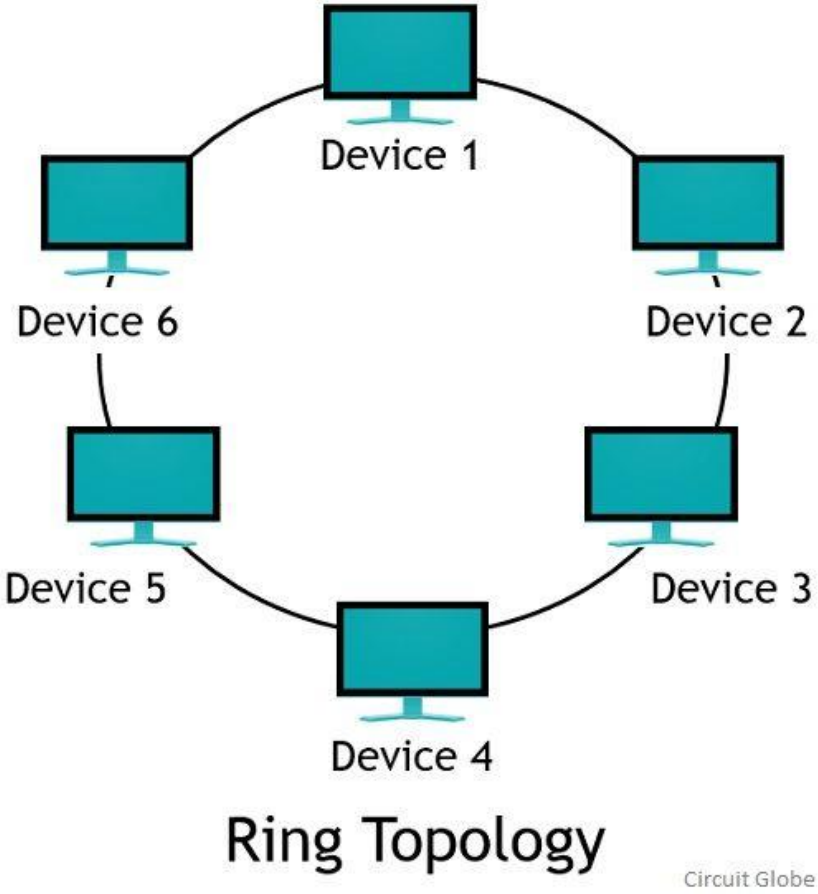
Parallel Architecture

Discussion

Bisection width vs Bisection bandwidth

Assume each link of this ring architecture carries 1GB/sec.

What is the bisection width and bandwidth?



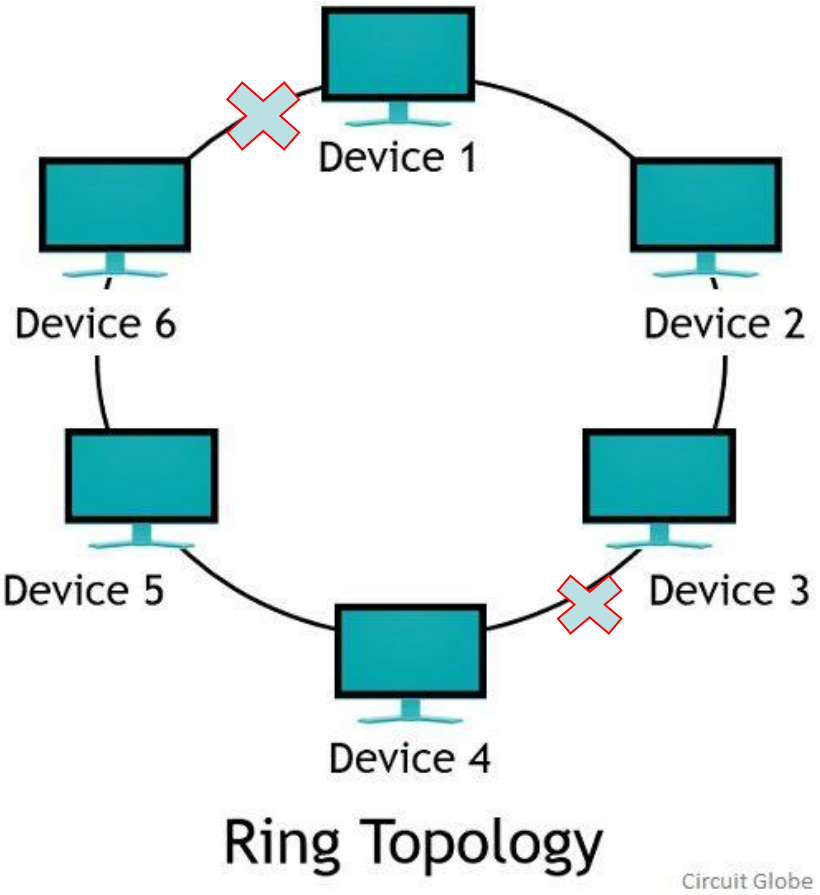
Bisection width vs Bisection bandwidth

What is the bisection width and bandwidth?

Partition the network with all possible even cuts.

- For each cut, compute # links that connect two subnets and their aggregated bandwidth.

Choose the minimum in each setting



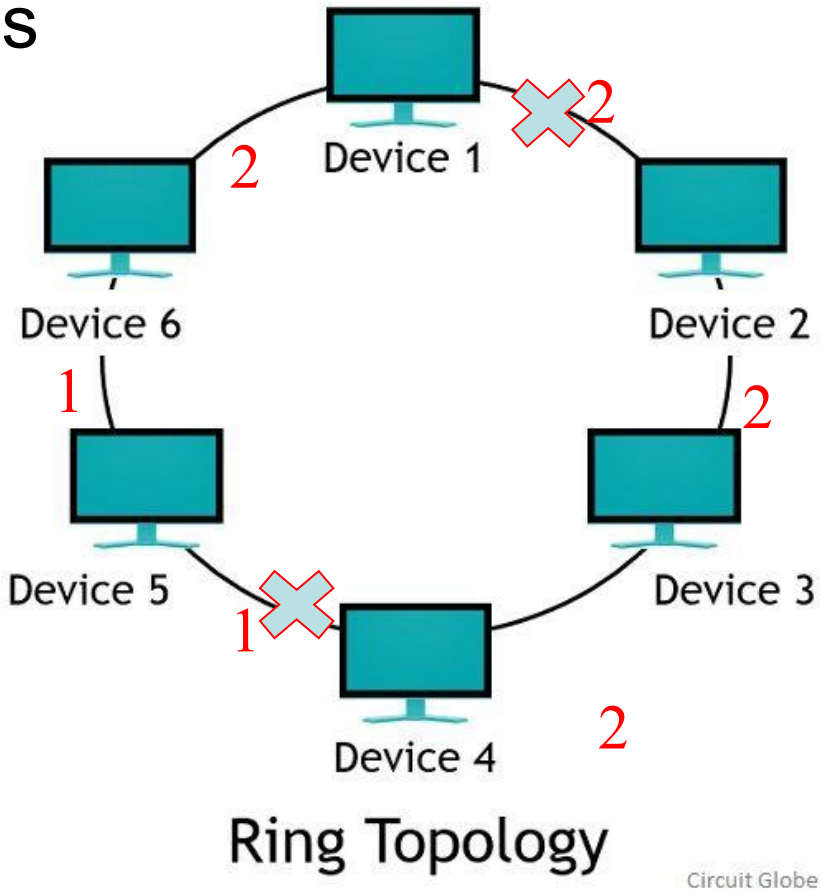
Bisection width 2

Bisection bandwidth 2 GB/s

Bisection width vs Bisection bandwidth

Links carry 1GB/s or 2GB/s as marked.

What is the bisection width and bandwidth?

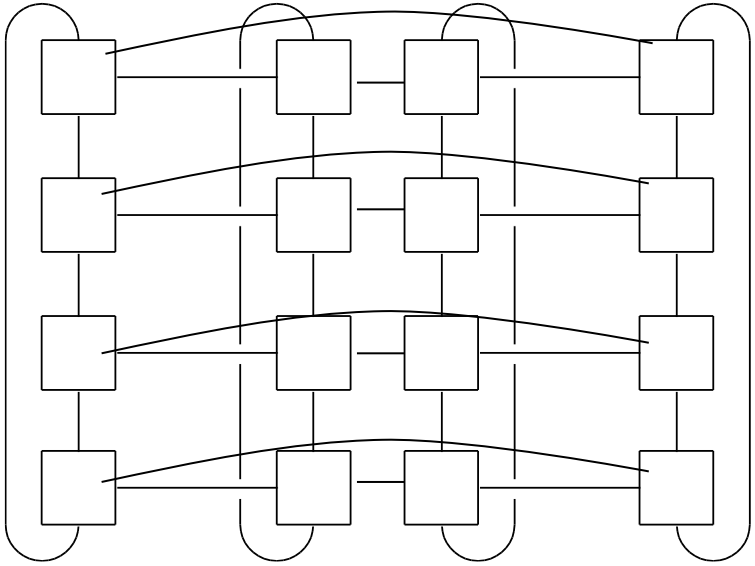


Bisection width 2

Bisection bandwidth 3 GB/s

Bisection width and bandwidth

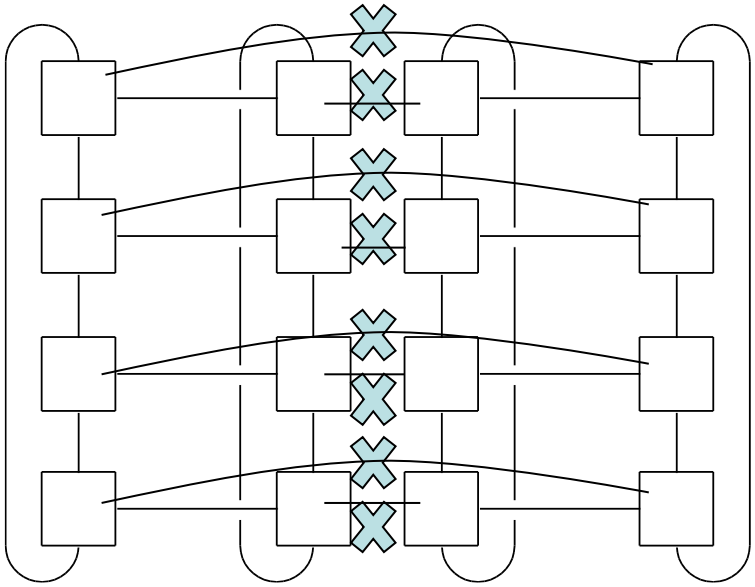
What is bisection width of a 2D torus with q^2 nodes: ?



What is the bisection bandwidth, assume each link carries 1GBytes/sec

Bisection width and bandwidth

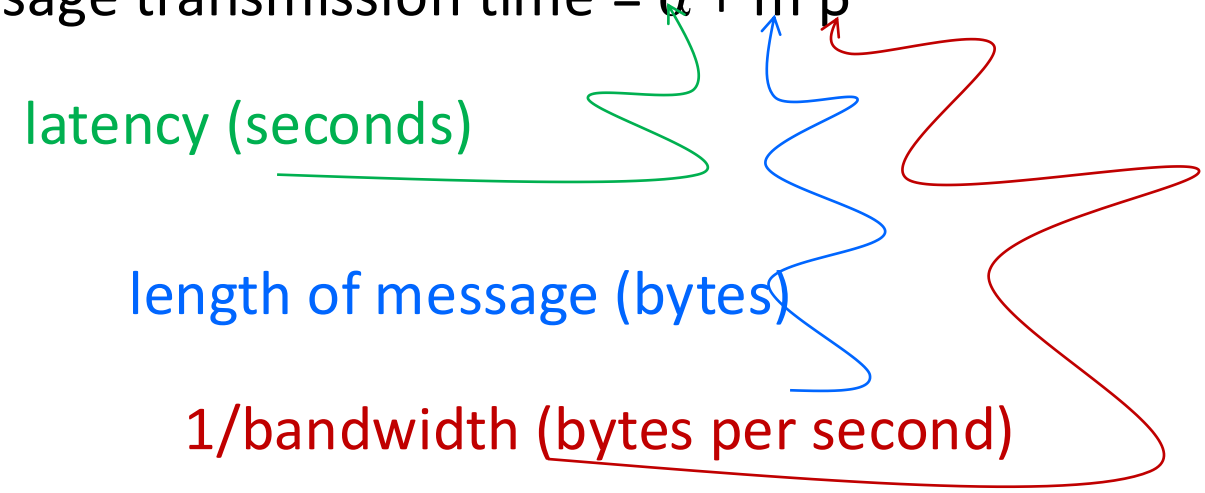
Bisection width: 2q



Bisection bandwidth: 2q GB/s

Network transmission cost

Message transmission time = $\alpha + m\,\beta$



Assume latency between two CSIL machines: 1 millisecond
Average bandwidth between them is 1GB/second

How long does it to transmit a message of 1KB bytes?

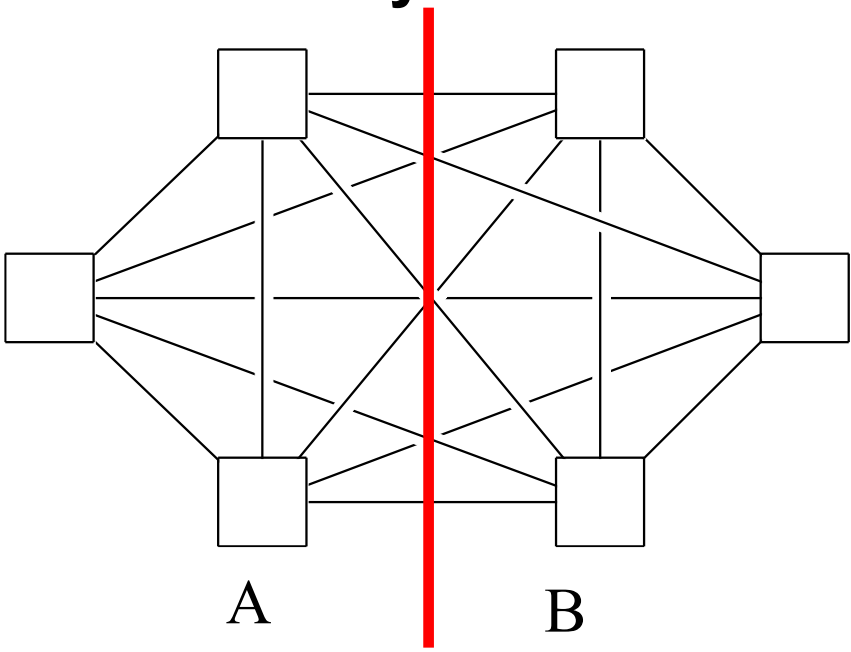
About $0.001s + 1000/(10^9) = 0.001001s = 1.001ms$

How long does it to transmit a message of 1MB bytes?

About $0.001s + 10^6/(10^9) = 0.002\ s = 2ms$

Fully connected network

- **Each node is directly connected to every other node.**



Assume p= 2k

bisection width = $p^2/4=k^2$

Why?

Divide two partitions A and B of size k

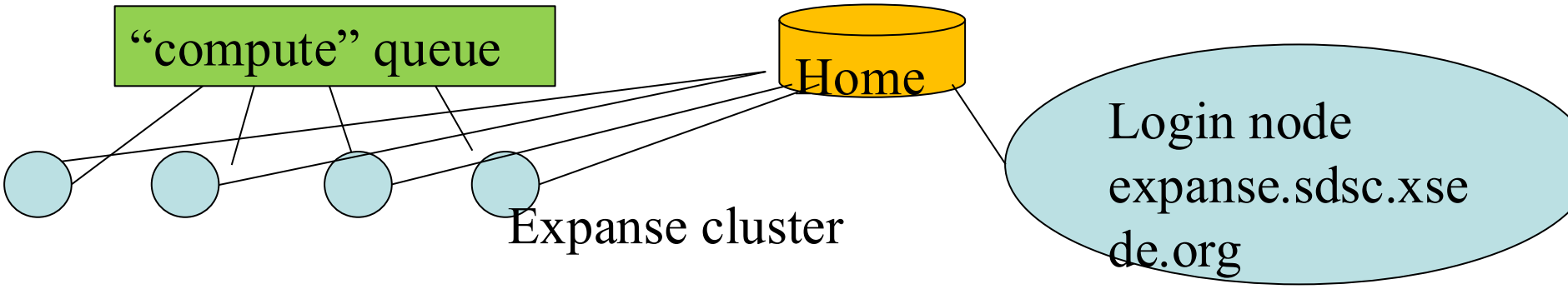
#Cross edges between A and B are:

1 node in partition A has k edges connected with k nodes in partition B.

k nodes in Partition A connect k^2 edges in Partition B

How to Run a Job at Expanse Cluster

https://sites.cs.ucsb.edu/~tyang_class/140w26/expanse140.html



- Expanse cluster managed by XSEDE has 728 standard CPU nodes, 54 GPU nodes and 4 large-memory nodes.
- ssh username@login.expanse.sdsc.edu
  - Login node can only be used for light activities
  - For example, can compile C code, but not Java code
- Sample example at Expanse under /home/tyang/cs140sample/mpi/
- Submit a job to run code at "compute" partition
  - To submit a job that runs a hello binary
  - sbatch run-hello.sh

Job shell script that runs a sequential hello program at Expanse

```
#!/bin/bash
#SBATCH --job-name="hello"
#SBATCH --output="job_hello.%j.out"
#SBATCH --partition=shared
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --export=ALL
#SBATCH --account=csb175
#SBATCH -t 00:01:00

../hello
```

Job name you can observe when querying status

Job id in the submitted queue

Which cluster to run this program

This job runs with 1 nodes, 1 core per node under CS140 account csb175

Set a time limit that this job runs at most 1 minute.

Run the hello program

```
main(void){
        printf("Hello\n");
}
```

Other useful information regarding Expanse

Command	Functionality
expanse-client user -p	Check computing resource balance
sbatch run-hello.sh	Submit job
squeue –u username	Check the job queues
sacct -u username	Check status of your jobs
scancel JobID	Cancel job

To get an account

Step 1: Open your account in https://access-ci.org

Step 2: Fill your account name to a google sheet that will be announced in Piazza

Step 3: TAs will add your user name to the class resource allocation

Step 4: ssh username@login.expanse.sdsc.edu