CISC 372: Parallel Computing
Exam 1 Review

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

Ghost cells

- when do you need ghost cells?
  - an algorithm requires data on its nearest neighbors
- the typical scenario
  - block distributed array `a`
  - the update function for `a[i]` requires the left and/or right neighbor
    - `a[i] = ... a[i-1] ... a[i+1] ...`
  - on each proc, the left neighbor of the left-most cell is on another proc
  - ditto for the right
  - solution: create ghost cells to mirror those neighbors
  - just before updating, exchange ghost cells

Nearest neighbor communication and Pascal's triangle

Usual representation:

				1				
			1		1			
		1		2		1		
	1		3		3		1	
1		4		6		4		1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

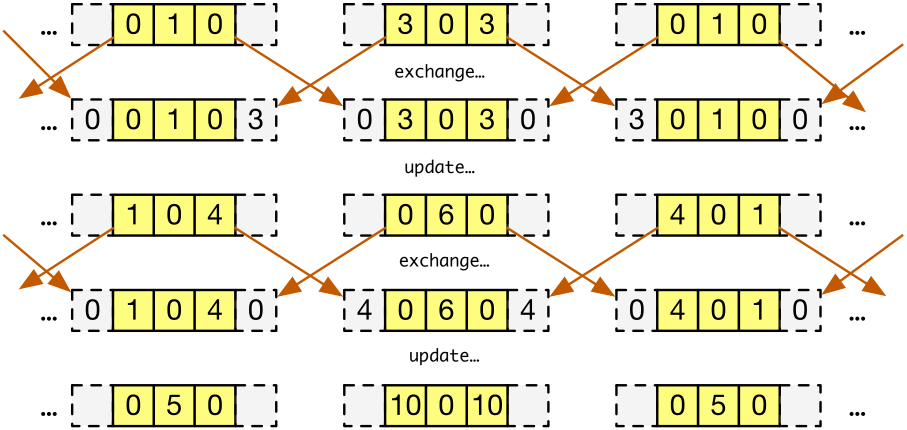
Computer representation:

0	0	0	0	1	0	0	0	0
0	0	0	1	0	1	0	0	0
0	0	1	0	2	0	1	0	0
0	1	0	3	0	3	0	1	0
1	0	4	0	6	0	4	0	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Pascal: sequential implementation

- see `pascal.c`
- two arrays are used
  - one always holds the current value
  - the other holds the previous value
- note use of pointer swapping
- note the update function has nearest neighbor dependency
  - `for (int j=1; j<2*N; j++) q[j] = p[j-1]+p[j+1]`

Pascal: ghost cell exchange



- the length of the array on proc r is `NUM_OWNED`$(r) + 2$
- indexes are shifted up by 1; see `pascal_mpi.c`
- `diffusion1d.c`: same issue: `u_new[i] = u[i] + k*(u[i+1] + u[i-1] - 2*u[i])`

2-d Diffusion

- a metal unit square
  - initially 100°
  - temperature on perimeter kept at 0°
- $u = u(x, y, t)$ temperature function
- 2d diffusion equation

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

- discretization

```
u_new[i][j] =  u[i][j]
  + k*(u[i+1][j] + u[i-1][j]
    + u[i][j+1] + u[i][j-1] - 4*u[i][j]);
```

Parallelization of diffusion2d

- how to distribute the 2d spatial domain?

Parallelization of diffusion2d

- how to distribute the 2d spatial domain?
- "striped" decompositions

Parallelization of diffusion2d

- how to distribute the 2d spatial domain?
- "striped" decompositions
  - apply the Standard Block Distribution Scheme to the columns

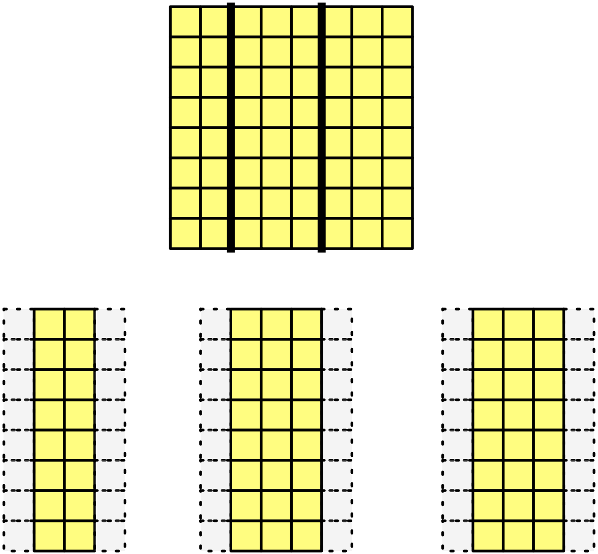
Parallelization of diffusion2d

- how to distribute the 2d spatial domain?
- "striped" decompositions
  - apply the Standard Block Distribution Scheme to the columns
    - "column distribution"
    - each process gets a certain number of x values
    - a ghost cell column on the left and on the right
    - exchange ghost columns after each time step
    - the entire column should be sent as one message
    - this is much more efficient than sending each cell in its own message

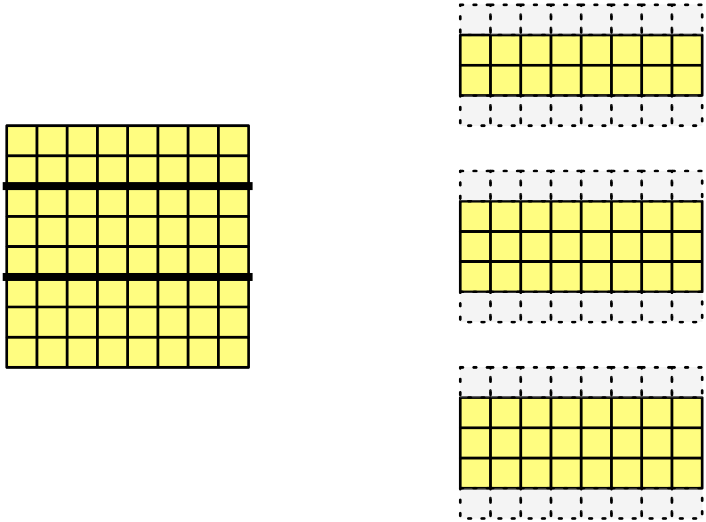
Parallelization of diffusion2d

- how to distribute the 2d spatial domain?
- "striped" decompositions
  - apply the Standard Block Distribution Scheme to the columns
    - "column distribution"
    - each process gets a certain number of x values
    - a ghost cell column on the left and on the right
    - exchange ghost columns after each time step
    - the entire column should be sent as one message
    - this is much more efficient than sending each cell in its own message
  - apply the Standard Block Distribution Scheme to the rows
    - "row distribution"
    - ...

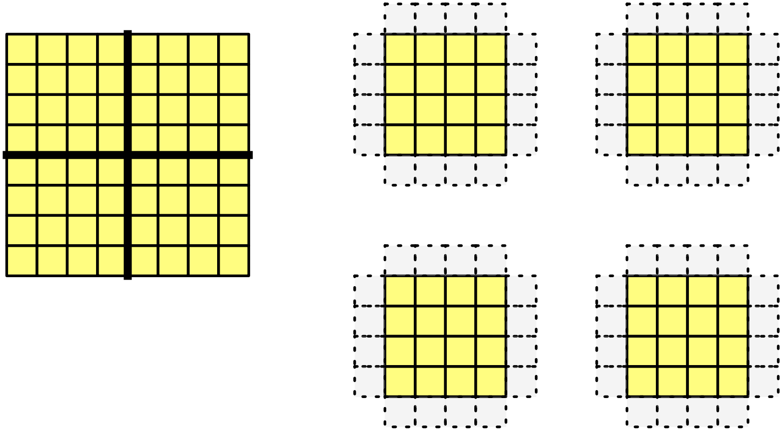
2d Diffusion: column distribution



2d Diffusion: row distribution



2d Diffusion: checkerboard decomposition



- 4 ghost regions for each process
- 4 exchanges: up, down, left, right

Outline

1. introduction
   - Moore's law, N/UMA, clusters vs. multicore, power
   - message-passing vs. shared-memory models
2. UNIX basics
   - file system, `ls`, `pwd`, `mkdir`, `make`, ...
3. C
   - preprocessor, compiler, linker
   - types, declarations, function definitions, pointers, `malloc`/`free`, multi-dimensional arrays
4. MPI
   - startup, shutdown, communicators, rank, size
   - point-to-point: send, receive, wildcards, semantics, deadlock
   - collectives
5. distribution strategies: cyclic, block, manager-worker
6. applications: SAT, Pascal, diffusion1d/2d

C: Pointers

- a pointer is the address of a memory location
- pointers are first-class objects in C
- there are pointer types
- a pointer can be assigned using `=`
- a pointer can be passed as an argument in a function call
- a pointer can be returned by a function
- there are operations which consume pointers and return pointers
- a pointer is just like any other kind of data

Pointer types

- declaration
  - if $T(x)$ declares x to have type T
  - then $T(*\texttt{p})$ declares `p` to have type *pointer-to-T*
- declaration examples
  - `double *p`
    - $T(x) =$ `double` x
    - $T(*\texttt{p}) =$ `double *p`
    - `p` has type *pointer-to-double*
  - `unsigned long int *p`
    - $T(x) =$ `unsigned long int` x
    - $T(*\texttt{p}) =$ `unsigned long int *p`
    - `p` has type *pointer-to-unsigned-long-int*

Pointer operations

There are two basic operations on pointers:

- address-of (`&`)
  - given a variable, returns the address of that variable
  - if `x` has type *T* then `&x` has type pointer-to-*T*
  - example
    - ```
      int x;
      int *p = &x; // address of x
      ```
- dereference (`*`)
  - given a pointer, returns the value stored at that address
  - if `p` has type pointer-to-*T* then `*p` has type *T*
  - example
    - ```
      int x = 5;
      int *p = &x;
      int y = 2 * (*p); // 10
      ```

Pointer operations, cont

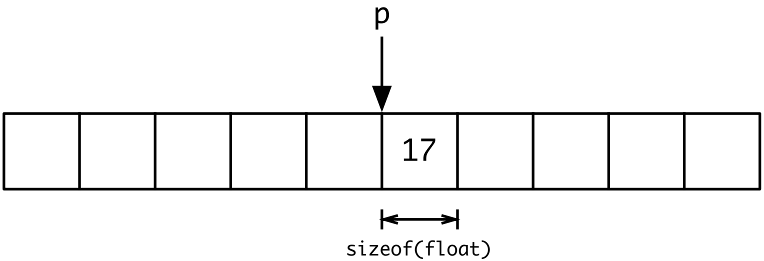
- *p can also be used on the left-hand side of an assignment

```
double x = 3.1415;
double *p = &x;
*p = 2.71828;
printf("%lf", x); // 2.71828
```

Pointers into arrays

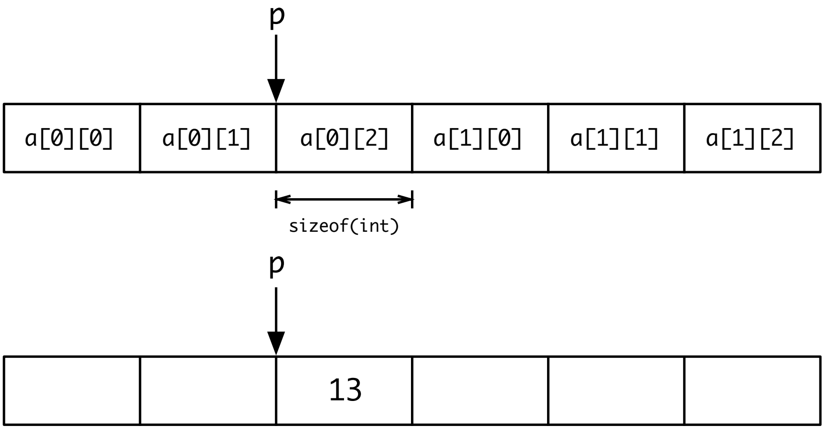
- you can also take the address of array elements

```
float a[10];
float *p = &a[5];
*p = 17;
```



Pointer into 2d-array

```
int a[2][3];
int *p = &a[0][2];
*p = 13;
```



Pointer arithmetic

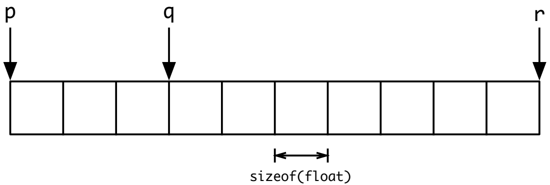
if all of the following hold

- p is an expression of type pointer-to-T
- i is an expression of integer type
- T is a complete type (size of T is known!!)

then

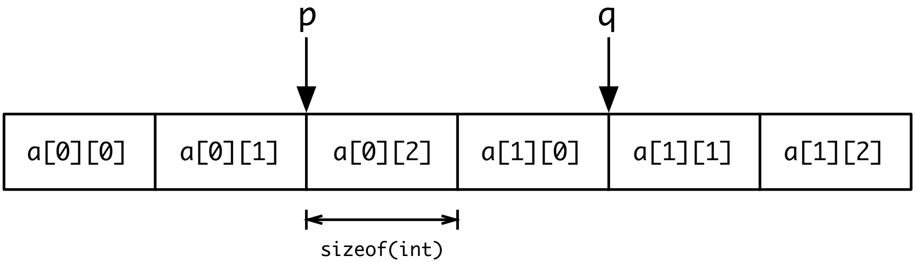
- `p+i` is an expression of type pointer-to-T
- it points to the address that is i T's past p
- if `sizeof(T)` is n bytes, then `p+i` is $i * n$ bytes after p

```
float a[10];
float *p = &a[0], *q = p+3, *r = q+7;
```



Pointer arithmetic within a 2d-array

```
int a[2][3];
int *p = &a[0][2];
int *q = p+2; // q == &a[1][1]
```



The real meaning of the index operator [..]

The meaning of `x[y]`:

- `x[y]` is syntactic sugar for `*(x+y)`
- if `p` is a pointer-to-T, then p[i] means `*(p+i)`
  - recall: this can be used to read or write to location `p+i`

Example: index operator and pointers

```c
#include <stdio.h>

/* assigns val to p[i], ..., p[i+n-1] */
void set_range(int *p, int n, int val) {
  for (int i=0; i<n; i++) p[i] = val;
}

/* prints p[0], ..., p[n-1] */
void print(int *p, int n) {
  for (int i=0; i<n; i++) printf("%d ", p[i]);
  printf("\n");
}

int main() {
  int a[10];
  set_range(&a[0], 10, 0); // a[0..9]=0
  print(&a[0], 10);
  set_range(&a[3],  5, 8); // a[3..7]=8
  print(&a[0], 10);
}
```

```
basie:c siegel$ cc ptr1.c
basie:c siegel$ ./a.out
0 0 0 0 0 0 0 0 0 0
0 0 0 8 8 8 8 8 0 0
basie:c
```

C's array-pointer "pun"

In most contexts:

- any expression of type *array-of-T* is automatically converted to an expression of type *pointer-to-T*
- pointing to the first (i.e., 0-th) element of the array
- i.e. `a` and `&a[0]` denote the same thing
  - the pointer to element 0 of array `a`

```
#include <assert.h>
int main() {
  int a[10];
  int *p;
  p = a; // same as p=&a[0]
  assert(a[3] == *(p+3));
  assert(a[3] == *(a+3));
}
```

Exceptions: `sizeof` and a few other places

C's array pointer pun, cont.

- any formal parameter in a function header of type *array-of-T* is converted to type *pointer-to-T*
- example: the following all mean exactly the same thing:
  - `int f(double *a);`
  - `int f(double a[]);`
  - `int f(double a[1000]);`
    - the `1000` is simply ignored
    - no reason to do this, unless it is as documentation
- one difference: an array can not occur on left side of `=`
  - ```
    int a[10];
    int b[10];
    int *p;
    p = a; // yes
    p = b; // yes
    a = p; // no!
    a = b; // no!
    ```
    ```

Allocating sequences of data

Multiple ways:

1. `double a[10];`
   - in the file scope
   - allocates an array that persists for the entire life of the program
   - can be accessed in any scope
   - length must be a constant expression
   - cannot be used if length is unknown at compile time
2. `double a[n];`
   - in a local scope
   - allocates an array that persists until the end of that scope is reached
   - can be accessed in that scope and sub-scopes, and through pointers
   - length can be any integer expression
3. `malloc` and `free`
   - dynamic memory allocation
   - memory allocated in the heap
   - programmer controls when allocation and deallocation occur
   - all accesses through pointers

Heap allocation: `malloc` and `free`

- `malloc` and `free` are functions defined in `stdlib`
- `malloc`
  - consumes argument of integer type
    - the number of bytes to allocate
  - allocates that many bytes in the heap
  - returns `void*`
    - address of first byte allocated
    - typically, this is converted immediately into a non-void pointer type
  - example
    - `int *p = (int*)malloc(10*sizeof(int));`
    - allocates space for 10 `int`s and returns pointer to beginning of that region
- `free`
  - consumes a `void*` pointer previously produced by `malloc`
  - deallocates the object

Heap allocation: example

```
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
void print(int *p, int n) {
  for (int i=0; i<n; i++) printf("%d ", p[i]);
  printf("\n");
}
int main(int argc, char * argv[]) {
  int n = atoi(argv[1]); // converts first command-line arg to int
  int * p = (int*)malloc(n*sizeof(int));
  assert(p); // check that malloc succeeded
  for (int i=0; i<n; i++) p[i] = i;
  print(p, n);
  free(p);
}
```

```
basie:c siegel$ cc malloc1.c
basie:c siegel$ ./a.out 10
0 1 2 3 4 5 6 7 8 9
basie:c siegel$
```

Structures

The following defines a new type named `struct Show`:

```
struct Show {
  int channel; // this is an int field
  char * name; // this is a string field
  double cost; // this is a double field
};
```

```
struct Show show;
show.channel = 10;
show.name = "The 372 Show";
show.cost = 100000.00;
```

- `struct Show` is a type just like any other type
- can be used to declare variables, as function parameter type, can be returned by a function, ...

Structures, cont.

It may be convenient to give the new type a shorter name:

```
typedef struct _show {
  int channel; // this is an int field
  char * name; // this is a string field
  double cost; // this is a double field
} Show;
```

- now you can just use `Show` instead of `struct _show`
- note: you can use the same name for the struct and the new type
  - `typedef struct X { ...} X;`

Structures and pointers

- structures are often manipulated using pointers
- functions consuming a structure typically consume a pointer to the structure
- functions returning structures typically return a pointer to a structure

```
int getChannel(Show * show) {
  return (*show).channel;
}

void setChannel(Show * show, int c) {
  (*show).channel = c;
}
```

- this pattern is so popular that C provides a shortcut
  - `s->x` is syntactic sugar for `(*s).x`

Structures and pointers, cont.

OK:

```
int getChannel(Show * show) {
  return (*show).channel;
}

void setChannel(Show * show, int c) {
  (*show).channel = c;
}
```

Better:

```
int getChannel(Show * show) {
  return show->channel;
}

void setChannel(Show * show, int c) {
  show->channel = c;
}
```

Arrays of structures

- one can create an array of structures, or
- one can create an array of pointers to structures.

Each has advantages (and disadvantages).

```
Show *shows[n]; // array of pointer to Show
for (int i=0; i<n; i++) {
  Show * s = (Show*)malloc(sizeof(Show));
  s->channel = i;
  shows[i] = s;
}
```

MPI Program Model

- an MPI program consists of multiple processes
- each process has its own memory (no shared memory)
- think of each process as a program running on its own computer
- the computers can have different architectures
- the programs do not even have to be written in the same language
  - MPI officially supports C and Fortran
- however, in most cases:
  - programmer writes one generic program
  - compiles this
  - at run-time, specifies number of processes
  - run-time system
    - instantiates that number of processes
    - distributes them where they need to go
  - a process can obtain its unique ID ("rank")
    - by branching on rank, each process can execute different code

Cyclic Distribution

Generalize

Given any number of tasks.

Given p processes.

Distribute the tasks cyclically:

- proc 0: $0, p, 2p, \ldots$
- proc 1: $1, p+1, 2p+1, \ldots$
- proc 2: $2, p+2, 2p+2, \ldots$
- etc.

I.e., proc i gets tasks t, where $t\%p = i$.

See `sat1.c`, `Makefile`.

- good for most embarrassingly parallel problems
- generally effective when longer tasks tend to occur next to each other
- for problems that require nearest-neighbor communicaton: don't use this

MPI_Send

`MPI_Send(buf, count, datatype, dest, tag, comm)`

`buf`	address of send buffer (`void*`)
`count`	number of elements in buffer (`int`)
`datatype`	data type of elements in buffer (`MPI_Datatype`)
`dest`	rank of destination process (`int`)
`tag`	integer to attach to message envelope (`int`)
`comm`	communicator (`MPI_Comm`)

- message envelope
  - source rank
  - destination rank
  - tag
  - communicator
- tag can be used by receiver to select which message to receive

MPI_Recv

```
MPI_Recv(buf, count, datatype, source, tag, comm, status)
```

buf	address of send buffer (void*)
count	number of elements in buffer (int)
datatype	data type of elements in buffer (MPI_Datatype)
source	rank of source process (int)
tag	tag of message to receive (int)
comm	communicator (MPI_Comm)
status	pointer to status object (MPI_Status*)

- count must be at least as large as count of incoming message
- status: object to store envelope information on received message
  - source, tag, count
- why would you need to know source and tag?

Using "wildcards" in `MPI_Recv`

```
MPI_Recv(buf, count, datatype, source, tag, comm, status)
```

- `source` argument can be `MPI_ANY_SOURCE`
  - special constant defined by MPI
  - means "receive message from any source"
  - use with care
  - introduce nondeterminism into the parallel program
  - program can produce different results on different executions
  - sometimes this is necessary (dynamic load-balancing)
  - do not use unless necessary for algorithm
- `tag` argument can be `MPI_ANY_TAG`
  - "receive message with any tag"
  - this one does not introduce nondeterminism
- can use neither, either, or both in one receive operation

Getting the status

`status` is a C `struct`

Getting the status

`status` is a C `struct`

- getting the rank of the source
  - `status.MPI_SOURCE`

Getting the status

`status` is a C `struct`

- getting the rank of the source
  - `status.MPI_SOURCE`
- getting the tag of the message
  - `status.MPI_TAG`

Getting the status

`status` is a C `struct`

- getting the rank of the source
  - `status.MPI_SOURCE`
- getting the tag of the message
  - `status.MPI_TAG`
- getting the error code
  - `status.MPI_ERROR`

Getting the status

`status` is a C `struct`

- getting the rank of the source
  - `status.MPI_SOURCE`
- getting the tag of the message
  - `status.MPI_TAG`
- getting the error code
  - `status.MPI_ERROR`
- getting the size ("count") of the message
  - not simply a field in the struct
  - need to use function `MPI_Get_count`

Example: status.c

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

int main() {
  char message[100];
  int rank;
  MPI_Status status;

  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    strcpy(message,"Hello, from proc 0!");
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  } else if (rank == 1)  {
    MPI_Recv(message, 100, MPI_CHAR, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    printf("Proc 1 received: \"%s\"\n", message);
    printf("source=%d tag=%d \n", status.MPI_SOURCE, status.MPI_TAG);
  }
  MPI_Finalize();
}
```

`status.c` output

Note that in C, a string is a sequence of `char` ending with the "null terminating char" `'\0'`. The number of characters in the string is therefore `strlen(message) + 1` $= 19 + 1 = 20$.

```
> mpiexec status.exec
Proc 1 received: "Hello, from proc 0!"
source=0 tag=99
```

MPI_Get_count

`MPI_Get_count(status, datatype, count)`

`status`	pointer to status object (`MPI_Status*`)
`datatype`	data type of elements received (`MPI_Datatype`)
`count`	pointer to variable in which to return result (`int*`)

MPI_Get_count

`MPI_Get_count(status, datatype, count)`

`status`	pointer to status object (`MPI_Status*`)
`datatype`	data type of elements received (`MPI_Datatype`)
`count`	pointer to variable in which to return result (`int*`)

- should only be called after `status` has been filled in by receive

MPI_Get_count

`MPI_Get_count(status, datatype, count)`

`status`	pointer to status object (`MPI_Status*`)
`datatype`	data type of elements received (`MPI_Datatype`)
`count`	pointer to variable in which to return result (`int*`)

- should only be called after `status` has been filled in by receive
- `datatype` should be same as used in receive

MPI_Get_count

`MPI_Get_count(status, datatype, count)`

`status`	pointer to status object (`MPI_Status*`)
`datatype`	data type of elements received (`MPI_Datatype`)
`count`	pointer to variable in which to return result (`int*`)

- should only be called after `status` has been filled in by receive
- `datatype` should be same as used in receive
- sets `count` to the number of elements received

MPI_Get_count

`MPI_Get_count(status, datatype, count)`

`status`	pointer to status object (`MPI_Status*`)
`datatype`	data type of elements received (`MPI_Datatype`)
`count`	pointer to variable in which to return result (`int*`)

- should only be called after `status` has been filled in by receive
- `datatype` should be same as used in receive
- sets `count` to the number of elements received
- note
  - `count` specified in receive statement and message `count` can differ
  - receive buffer must be big enough to hold incoming message
  - memory in receive buffer after message count will not be altered

Example: getting the count: count.c

The following lines are added to proc 1:

```
int count;
MPI_Get_count(&status, MPI_CHAR, &count);
printf("source=%d tag=%d count=%d\n",
       status.MPI_SOURCE, status.MPI_TAG, count);
```

Example: getting the count: count.c

The following lines are added to proc 1:

```
int count;
MPI_Get_count(&status, MPI_CHAR, &count);
printf("source=%d tag=%d count=%d\n",
       status.MPI_SOURCE, status.MPI_TAG, count);
```

This sets count to the actual number of characters (MPI_CHAR) received.

Example: getting the count: count.c

The following lines are added to proc 1:

```
int count;
MPI_Get_count(&status, MPI_CHAR, &count);
printf("source=%d tag=%d count=%d\n",
       status.MPI_SOURCE, status.MPI_TAG, count);
```

This sets count to the actual number of characters (MPI_CHAR) received.

```
> mpiexec -n 4 ./count.exec
Proc 1 received: "Hello, from proc 0!"
source=0 tag=99 count=20
```

Note the null terminating character is counted.

Point-to-point

`MPI_STATUS_IGNORE` is

A. a type
B. a function
C. a constant
D. a variable
E. a type qualifier

Point-to-point semantics

Each of the following program fragments attempts to have two processes exchange data. In each case, state which of the following is true:

A. the fragment will definitely deadlock

B. the fragment will definitely not deadlock

C. the fragment may or may not deadlock

```
if (rank == 0) {
  MPI_Send(&myNumber, 1, MPI_INT, 1, 9, comm);
  MPI_Recv(&otherNumber, 1, MPI_INT, 1, 9, comm, &status);
} else if (rank == 1) {
  MPI_Send(&myNumber, 1, MPI_INT, 0, 9, comm);
  MPI_Recv(&otherNumber, 1, MPI_INT, 0, 9, comm, &status);
}
```

Point-to-point semantics

```
if (rank == 0) {
  MPI_Recv(&otherNumber, 1, MPI_INT, 1, 9, comm, &status);
  MPI_Send(&myNumber, 1, MPI_INT, 1, 9, comm);
} else if (rank == 1)  {
  MPI_Recv(&otherNumber, 1, MPI_INT, 0, 9, comm, &status);
  MPI_Send(&myNumber, 1, MPI_INT, 0, 9, comm);
}
```

A. the fragment will definitely deadlock

B. the fragment will definitely not deadlock

C. the fragment may or may not deadlock

Point-to-point semantics

```
if (rank == 0) {
  MPI_Send(&myNumber, 1, MPI_INT, 1, 9, comm);
  MPI_Recv(&otherNumber, 1, MPI_INT, 1, 9, comm, &status);
} else if (rank == 1)  {
  MPI_Recv(&otherNumber, 1, MPI_INT, 0, 9, comm, &status);
  MPI_Send(&myNumber, 1, MPI_INT, 0, 9, comm);
}
```

A. the fragment will definitely deadlock
B. the fragment will definitely not deadlock
C. the fragment may or may not deadlock

Point-to-point semantics

```
if (rank == 0) {
  MPI_Sendrecv(&myNumber, 1, MPI_INT, 1, 9,
               &otherNumber, 1, MPI_INT, 1, 9, comm, &status);
} else if (rank == 1)  {
  MPI_Sendrecv(&myNumber, 1, MPI_INT, 0, 9,
               &otherNumber, 1, MPI_INT, 0, 9, comm, &status);
}
```

A. the fragment will definitely deadlock
B. the fragment will definitely not deadlock
C. the fragment may or may not deadlock

Point-to-point semantics

```
if (rank == 0) {
  MPI_Send(&myNumber, 1, MPI_INT, 1, 3, comm);
  MPI_Recv(&otherNumber, 1, MPI_INT, 1, 4, comm, &status);
} else if (rank == 1) {
  MPI_Send(&myNumber, 1, MPI_INT, 0, 4, comm);
  MPI_Recv(&otherNumber, 1, MPI_INT, 0, 3, comm, &status);
}
```

A. the fragment will definitely deadlock

B. the fragment will definitely not deadlock

C. the fragment may or may not deadlock

Point-to-point semantics

In a correct MPI program, the length of the receive buffer used in an `MPI_Recv`...

A. must be exactly equal to the length of the incoming message

B. must be greater than or equal to the length of the incoming message

C. may be any number; if the length of the buffer is less than the length of the incoming message, the message will be truncated

D. must be at least one greater than the length of the incoming message

E. must be less than or equal to the length of the incoming message

```
int main(int argc, char **argv) {
  int rank, x, y;  MPI_Init( &argc, &argv );  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    x = 10; y = 11;
    MPI_Send(&x, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
    MPI_Send(&y, 1, MPI_INT, 2, 9, MPI_COMM_WORLD);
  } else if (rank == 1)  {
    MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&y, 1, MPI_INT, MPI_ANY_SOURCE, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("%d %d\n", x, y); fflush(stdout);
  } else if (rank == 2) {
    x=20;
    MPI_Recv(&y, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&x, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
  }
  MPI_Finalize();
}
```

A. the program will never deadlock and the output will always be 10 20

B. the program will never deadlock and the output may be 10 20 or 20 10

C. it is possible for the program to deadlock; when it does not deadlock it will output 10 20

D. the program will always deadlock

E. it is possible for the program to deadlock, it is possible for it to output 10 20, and it is possible for it to output 20 10

Collective operations

- example: want to print the total number of solutions found
- each process can count its solutions
- then we need to add up these numbers across all processes
- this obviously requires communication
- an example of a collective operation
  - a communication operation involving all processes in a communicator
- to carry out a collective operation in MPI:
  - each process calls the same function
  - some arguments will be the same for all processes
  - some will differ
- non-interference: collective communication and p2p communication in two separate universes
- synchronization: no synchronization implied by collectives except what is logically necessary

Collectives

A program contains a call to `MPI_Bcast` with data type `MPI_DOUBLE` used on every process. Which of the following must be true if the program is correct:

A. the `count` argument used on a non-root process must be exactly equal to the `count` on the root

B. the `count` arguments used on non-root processes can differ, as long as they are all greater than or equal to the `count` on the root

C. the `count` on all the non-root processes must be the same number, but that number may be larger than the `count` used on the root

D. the `count` on all the non-root processes must be the same number, but that number may be at least one larger than the `count` used on the root

E. the `count` values on the non-root processes can be any numbers; if they are smaller than the count on the root, the message will just be truncated.

```c
int main(int argc, char **argv) {
  int rank, x, y;  MPI_Init( &argc, &argv ); MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    x = 10; y = 11;
    MPI_Send(&x, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
    MPI_Send(&y, 1, MPI_INT, 2, 9, MPI_COMM_WORLD);
  } else if (rank == 1)  {
    MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, 11, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&y, 1, MPI_INT, MPI_ANY_SOURCE, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("%d %d\n", x, y); fflush(stdout);
  } else if (rank == 2) {
    x = 20;
    MPI_Recv(&y, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&x, 1, MPI_INT, 1, 11, MPI_COMM_WORLD);
  }
  MPI_Finalize();
}
```

A. the program will never deadlock and the output will always be 20 10

B. the program will never deadlock and the output may be 10 20 or 20 10

C. it is possible for the program to deadlock, but when it does not, it will output 20 10

D. the program will always deadlock

E. it is possible for the program to deadlock, it is possible for it to output 10 20, and it is possible for it to output 20 10

Applications

Consider a `diffusion1d` program. Suppose that the length of the global temperature array is 100 (including boundary values), and the program is executed with 10 processes. What is the *maximum* number of ghost cells stored on any one process?

A. 0

B. 1

C. 2

D. 10

E. 100

Applications

Consider a row-distributed `diffusion2d` program. Suppose that the dimensions of the global temperature matrix is 100×100 (including boundaries), and the program is executed with 10 processes. What is the *maximum* number of ghost cells stored on any one process?

A. 1

B. 2

C. 10

D. 100

E. 200

Collectives

Suppose every process in a communicator calls `MPI_Allreduce` (correctly) with `MPI_SUM` as the reduction operation. Does this necessarily induce a barrier? (Y/N)

A. Yes

B. No

Collectives

Suppose every process in a communicator calls `MPI_Bcast` (correctly). Does this necessarily induce a barrier? (Y/N)

A. Yes

B. No

```
int main(int argc, char **argv) {
  int rank, x, y;  MPI_Init( &argc, &argv );  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    x = 10; y = 11;
    MPI_Send(&x, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
    MPI_Send(&y, 1, MPI_INT, 2, 9, MPI_COMM_WORLD);
  } else if (rank == 1)  {
    MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&y, 1, MPI_INT, MPI_ANY_SOURCE, 11, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("%d %d\n", x, y); fflush(stdout);
  } else if (rank == 2) {
    x = 20;
    MPI_Recv(&y, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&x, 1, MPI_INT, 1, 11, MPI_COMM_WORLD);
  }
  MPI_Finalize();
}
```

A. the program will never deadlock and the output will always be 10 20

B. the program will never deadlock and the output may be 10 20 or 20 10

C. it is possible for the program to deadlock, but when it does not, it will output 10 20

D. the program will always deadlock

E. it is possible for the program to deadlock, it is possible for it to output 10 20, and it is possible for it to output 20 10

Distribution

For the following, suppose an array of length n (indexed from 0 to $n-1$) is block-distributed over p processes (with ranks $0, \ldots, p-1$) using the standard block-distribution scheme.

Distribution

For the following, suppose an array of length n (indexed from 0 to $n-1$) is block-distributed over p processes (with ranks $0, \ldots, p-1$) using the standard block-distribution scheme.

What is the formula $\mathit{first}(i)$ for the global index of the first element on process i?

Distribution

For the following, suppose an array of length n (indexed from 0 to $n-1$) is block-distributed over p processes (with ranks $0, \ldots, p-1$) using the standard block-distribution scheme.

What is the formula $\mathit{first}(i)$ for the global index of the first element on process i?

$$\text{first}(i) = \text{floor}(in/p)$$

Distribution

For the following, suppose an array of length n (indexed from 0 to $n-1$) is block-distributed over p processes (with ranks $0, \ldots, p-1$) using the standard block-distribution scheme.

What is the formula *first*(i) for the global index of the first element on process i?

$$\text{first}(i) = \text{floor}(in/p)$$

What is the formula for the rank i of the process controlling the element with global index j?

Distribution

For the following, suppose an array of length n (indexed from 0 to $n-1$) is block-distributed over p processes (with ranks $0, \ldots, p-1$) using the standard block-distribution scheme.

What is the formula *first*(i) for the global index of the first element on process i?

$$\text{first}(i) = \text{floor}(in/p)$$

What is the formula for the rank i of the process controlling the element with global index j?

$$\text{owner}(j) = \text{floor}((p(j+1)-1)/n)$$

Distribution

For the following, suppose an array of length n (indexed from 0 to $n-1$) is block-distributed over p processes (with ranks $0, \ldots, p-1$) using the standard block-distribution scheme.

What is the formula $\mathit{first}(i)$ for the global index of the first element on process i?

$$\text{first}(i) = \text{floor}(in/p)$$

What is the formula for the rank i of the process controlling the element with global index j?

$$\text{owner}(j) = \text{floor}((p(j+1)-1)/n)$$

What is the formula for the local index k of the element with global index j?

Distribution

For the following, suppose an array of length n (indexed from 0 to $n-1$) is block-distributed over p processes (with ranks $0, \ldots, p-1$) using the standard block-distribution scheme.

What is the formula *first*(i) for the global index of the first element on process i?

$$\text{first}(i) = \text{floor}(in/p)$$

What is the formula for the rank i of the process controlling the element with global index j?

$$\text{owner}(j) = \text{floor}((p(j+1)-1)/n)$$

What is the formula for the local index k of the element with global index j?

$$j - \text{first}(\text{owner}(j))$$

```
#include<stdio.h>
#include<mpi.h>
int main(int argc, char **argv) {
  int rank, x=1;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    MPI_Send(&x, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
  } else if (rank == 1) {
    MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Message received.\n");
  }
}
```

A. When run with 2 or more processes, the program will never deadlock and will output "`Message received.`".

B. When run with more than 2 processes, the program may or may not deadlock; if it does not deadlock it will output "`Message received.`".

C. When run with more than 2 processes, the program will deadlock.

D. When run with 1 process, the program will always terminate normally without printing anything.

E. The program is incorrect.