• Algorithm Design
• Collective Communication
• Communication Modes
Foster's Design Methodology

- Task/Channel Model
- Partitioning
- Communication
- Agglomeration
- Mapping
Foster's Methodology

Problem → Partitioning → Communication

Mapping → Agglomeration

Figure Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Task/Channel Model

• Program consists of a number of tasks
  – each task does computation on some data
  – each task has ports where data goes in and out
  – a task may or may not be an asynchronous process or thread (do not confuse with MPI tasks)

• Channels connect the input port of one task to the output port of another
  – define the flow of data between tasks
  – define dependencies between tasks
Task Model Example

Figure Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Partitioning

- Partitioning divides the data and/or computation into tasks
  - Domain decomposition is dividing the data
  - Functional decomposition is dividing the computation
- Try to extract as much parallelism as possible
  - at least an order of magnitude more tasks than processors
  - minimize redundant computations and data
  - tasks roughly the same size
  - number of tasks increases with problem size
Partitioning Example
Functional Decomposition Example

- Acquire Patient Images
  - Register Images
    - Determine Image Locations
      - Track Position of Instruments
        - Display Image

Figure Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Communication

• Determine communication between tasks
  – local communication involves few tasks
  – global communication involves many tasks

• Communication is overhead
  – communication should be balanced
  – tasks communicate with small number of locals
  – tasks can communicate concurrently
  – tasks can compute concurrently
Agglomeration

- Group tasks together to simply programming or improve efficiency
  - increase locality / reduce communication
  - combine messages
  - maintain scalability
  - increase reuse of sequential code
- Agglomeration should
  - increase locality
  - replicated tasks take less time that comm
  - replicated data small enough to allow scaling
  - tasks have similar comm and comp time
  - num tasks increase with problem size
  - num tasks small but large as P
  - trade-off between agglomeration and mods to code is reasonable
Agglomeration Example
Mapping

- Assign tasks to processors
- Maximize processor utilization
  - balance computation on processors
  - balance communication on processors
- Minimize interprocessor communication
  - mapping tasks to same processor eliminates communication
  - but ... also reduces parallelism
  - and reduces processor utilization
- Balance conflicting goals
  - NP complete problem
  - rely on heuristics
Mapping Example

(a)

(b)

Figure Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Mapping Decision Tree

• Static task allocation
  – Structured communication pattern
    • Constant computation per task
      – Agglomerate to one task per processor
    • Computation per task varies
      – Map tasks cyclically to balance load
  – Unstructured communication pattern
    – Use static load balancing algorithm

• Dynamic task allocation
  – Frequent communication
    – Use dynamic load balancing algorithm
  – Many short tasks, no communication
    – Use run-time task-scheduling algorithm
Mapping Goals

• Consider both one task per processor and multiple tasks per processor
• Consider both static and dynamic allocation of tasks
• For dynamic allocation, be sure manager is not bottleneck
• For static allocation, tasks per processor should be at least 10:1
Boundary Value Problem

Ice water  Rod  Insulation

Figure Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Results of Finite Different Solution

[Graph showing temperature variation over time with labels t = 0 and t = 3]

Figure Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Finite Difference Formulation

\[ u_{i,j+1} = ru_{i-1,j} + (1-2r)u_{i,j} + ru_{i+1,j} \]
\[ r = \frac{k}{h^2} \]
Partitioning and Agglomeration

Figure Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Analysis

• Sequential run time
  – m time steps
  – n-1 items to compute each step
  – each computation takes $T_{\text{comp}}$ time

\[ T(1) = m(n-1)T_{\text{comp}} \]

• Parallel run time
  – m time steps
  – n-1/p items to compute each step
  – 2 items to communicate each step
  – each communication takes $T_{\text{comm}}$

\[ T(p) = m[T_{\text{comp}}[(n-1)/p] + 2T_{\text{comm}}] \]
Global Sum - Simple

\[(n-1)(T_{\text{comm}} + T_{\text{comp}})\]
Global Sum - On 2 Processors

\[(n/2)(T_{comm} + T_{comp})\]
Global Sum On 4 Processors

\[(n/4)(T_{\text{comm}} + T_{\text{comp}})\]
Binomial Tree - Step 1

4
-3
8
-4

2
5
1
4

0
-6
2
6

7
-3
3
-1
Binomial Trees - Step 2

1 \rightarrow 7 \quad -6 \rightarrow 4

4 \rightarrow 5 \quad 8 \rightarrow 2
Binomial Trees - Step 3

\[
\begin{array}{cccc}
\text{8} & \text{-2} \\
\text{9} & \text{10}
\end{array}
\]
Binomial Trees - Step 4

17 → 8
Global Sum Partitioning

![Binomial Tree](image.png)

$$[\log n](T_{\text{comm}} + T_{\text{comp}})$$
Agglomeration
Agglomeration
Analysis

- initial summation
  \((\lceil n/p \rceil - 1)T_{comp}\)

- Each reduction step
  \(T_{comp} + T_{comm}\)

- Overall execution
  \(T(p) = (\lceil n/p \rceil - 1)T_{comp} + \lceil \log p \rceil (T_{comp} + T_{comm})\)
N-Body Problem

\[ f = G \times m_1 \times m_2 / d^2 \]

\( G = 6.674 \times 10^{-11} \text{ N} \)

\( f \) in Newtons

\( m \) in Kgram

\( d \) in meters
Decompose force vectors
\[ f_x = f \times \frac{d_x}{d} \]
\[ f_y = f \times \frac{d_y}{d} \]

Calculate acceleration, velocity, position
\[ a = \frac{f}{m} \]
\[ v = v_{-1} + a \times \Delta t \]
\[ p = p_{-1} + v \times \Delta t \]
Gather
Gather All
Naive Gather All
Hypercube Based Gather All
Simple IO

Input

Output
Scatter
Logarithmic Scatter
Collective Communication

• A communication pattern that involves all processes in a communicator.

• MPI contains a rich set of collective communication operations:
  
  - Broadcast, Reduce, Allreduce
  - Scatter, Gather, Allgather, Alltoall
  - Scan, Barrier, Scatter_reduced
Broadcast

- Broadcast to all tasks of a communicator
  ```c
  int root; // rank of the source
  MPI_Bcast(buffer, count, datatype, root, MPI_COMM_WORLD);
  ```
- Must be called by all tasks with the same arguments
Barrier

• Barrier synchronizes all tasks of a communicator

```
MPI_Barrier(MPI_COMM_WORLD);
```

• Each task calling MPI_Barrier will stop until all tasks in the communicator have called MPI_Barrier
Reduction

- Perform collective operations on a group of tasks

```c
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, Mpi_comm comm);
```

```
<table>
<thead>
<tr>
<th></th>
<th>A0</th>
<th>B0</th>
<th>C0</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>T2</td>
<td>A2</td>
<td>B2</td>
<td>C2</td>
<td>D2</td>
</tr>
<tr>
<td>T3</td>
<td>A3</td>
<td>B3</td>
<td>C3</td>
<td>D3</td>
</tr>
</tbody>
</table>
```

Reduce

```
<table>
<thead>
<tr>
<th></th>
<th>ΣA</th>
<th>ΣB</th>
<th>ΣC</th>
<th>ΣD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

copyright © 2003, Walt Ligon and Dan Stanzione, all rights reserved.
Reduction Operators

- The MPI_Op argument can be user-defined or one of a number of predefined operation.
  - The operation is performed on each node on the send buffer and stored in the destination node in the receive buffer
  - Predefined Operations include:
    - Sum, Product
    - Max, Min
    - Logical and bitwise AND, OR, XOR
- User defined functions are done with MPI_OP_CREATE and MPI_OP_FREE
- Many other collective ops exist in MPI and MPI-2; wait for the advanced course!
Example

- Find PI by numerically integrating
  \[ f(x) = \frac{4}{1 + x^2} \]
  - over range 0-1
- Implement with broadcast and reduce
- Number of intervals is broadcast
- Processors each pick a range of intervals to compute in parallel
- Reduction is used to produce a global sum
- Implemented in C, Fortran, and C++
```c
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (1) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    }
```
if (n == 0)
    break;
else {
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);
    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\\n", 
               pi, fabs(pi - PI25DT));
}
}

MPI_Finalize();
return 0;
An Example - Integration with the Trapezoidal Rule

```c
main (int argc, char **argv) {
    MPI_Init(&argc, &argv);
    /* Get my process rank and size*/
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    h = (b-a)/n;   /* h same for all processes */
    local_n = n/p; /* So is number of trapezoids */

    /* Length of each process' interval of integration = local_n*h. So my interval */
    /* starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a,local_b,local_n,h);

    /* Add up integrals calc'd by each process */
    if (my_rank == 0) {
        total = integral;
        for (source = 1; source < p; source++) {
            MPI_Recv(&integral, 1, MPI_FLOAT,
                      source, tag,
                      MPI_COMM_WORLD, &status);
            total = total + integral;
        }
    } else {
        MPI_Send(&integral, 1, MPI_FLOAT, dest,
                  tag, MPI_COMM_WORLD);
    }

    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n",
                n);
        printf("of the integral from %f to %f = %f\n",
                a, b, total);
    }

    /* Shut down MPI */
    MPI_Finalize();
} /* main */
```
An Example - Integration with the Trapeziodal Rule

float Trap(
    float local_a /* in */,
    float local_b /* in */,
    int local_n /* in */,
    float h /* in */) {
    float integral; /* Store result in integral */
    float x;
    int i;

    float f(float x); /* function we're integrating */

    integral = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (i = 1; i <= local_n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;
    return integral;
} /* Trap */

float f(float x) {
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
    return_val = x*x;
    return return_val;
} /* f */
void Get_data(float* a_ptr, float* b_ptr, 
    int* n_ptr, int my_rank, int p) {

    int source = 0; /* All local variables used by */
    int dest;       /* MPI_Send and MPI_Recv */
    int tag;
    MPI_Status status;

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
        for (dest = 1; dest < p; dest++) {
            tag = 0;
            MPI_Send(a_ptr, 1, MPI_FLOAT, dest, tag, 
                MPI_COMM_WORLD);
            tag = 1;
            MPI_Send(b_ptr, 1, MPI_FLOAT, dest, tag, 
                MPI_COMM_WORLD);
            tag = 2;
            MPI_Send(n_ptr, 1, MPI_INT, dest, tag, 
                MPI_COMM_WORLD);
        }
    } else {
        tag = 0;
        MPI_Recv(a_ptr, 1, MPI_FLOAT, source, tag, 
            MPI_COMM_WORLD, &status);
        tag = 1;
        MPI_Recv(b_ptr, 1, MPI_FLOAT, source, tag, 
            MPI_COMM_WORLD, &status);
        tag = 2;
        MPI_Recv(n_ptr, 1, MPI_INT, source, tag, 
            MPI_COMM_WORLD, &status);
    }
} /* Get_data */
Communication Pattern in Trapezoid Program

For initial send, assuming 8 nodes

Last worker starts after process 0 sends 7 messages!
Tree structured Communication

Last worker starts after process 0 sends 3 messages!
Modified Trapezoid

- Adapt to add tree-based computation using send and receive.

```c
void Get_data1(...

    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
    }

    for (stage = 0; stage < Ceiling_log2(p); stage++)
        if (I_receive(stage, my_rank, &source))
            Receive(a_ptr, b_ptr, n_ptr, source);
        else if (I_send(stage, my_rank, p, &dest))
            Send(*a_ptr, *b_ptr, *n_ptr, dest);
    } /* Get_data1*/
```

- Creates a log(p) stage tree to exchange data
Modified Trapezoid

- Adapt to add tree-based computation using send and receive.

```c
int I_receive(int stage, int my_rank,
             int* source_ptr /* out */) {
    int power_2.stage;

    // 2^stage = 1 << stage
    power_2.stage = 1 << stage;
    if ((power_2.stage <= my_rank) &&
        (my_rank < 2*power_2.stage)) {
        *source_ptr = my_rank - power_2.stage;
        return 1;
    } else return 0;
} /* I_receive */

int I_send(int stage, int my_rank,
           int p, int* dest_ptr ) {
    int power_2.stage;

    // 2^stage = 1 << stage
    power_2.stage = 1 << stage;
    if (my_rank < power_2.stage) {
        *dest_ptr = my_rank + power_2.stage;
        if (*dest_ptr >= p) return 0;
        else return 1;
    } else return 0;
} /* I_send */
```

void Receive(float* a_ptr,
             float* b_ptr, int* n_ptr, int source ) {
    MPI_Status status;
    MPI_Recv(a_ptr, 1, MPI_FLOAT, source, 0,
             MPI_COMM_WORLD, &status);
    MPI_Recv(b_ptr, 1, MPI_FLOAT, source, 1,
             MPI_COMM_WORLD, &status);
    MPI_Recv(n_ptr, 1, MPI_INT, source, 2,
             MPI_COMM_WORLD, &status);
} /* Receive */
Tree Communication

• OK, that **should** go faster
• Wouldn't it be nice if MPI created the tree for us?
• But on, certain systems, it may not be faster....
Tree Communication

Nodes 0-3

Nodes 4-7

Slow link

0

2

16

34

57

163475

0

2

1

3

0

4

2

6

1

5

3

7

copyright © 2003, Walt Ligon and Dan Stanzione, all rights reserved.
Tree Communication

This tree minimizes the number of messages across the slow link!
Collective Operations

- MPI collective operations protect programmer from details of cluster topology
- MPI definition does not specify how broadcast is implemented
- MPICH implements broadcast as a spanning tree as shown
- New versions embed topology aware operations to deal with "slow link" problem
  - Programmer calls broadcast, MPI Implementation deals with details
- Our trapezoid integration program can be modified to use broadcast as follows:
Trapezoid with Broadcast

```c
void Get_data2(float* a_ptr, float* b_ptr, int* n_ptr, int my_rank) {
    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
    }
    MPI_Bcast(a_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_ptr, 1, MPI_INT, 0, MPI_COMM_WORLD);
} /* Get_data2 */
```

- Cleaner, simpler program
- More portable between clusters
- Note, process 0 has broadcast the same as all other ranks...
Reduction Revisited

- Perform collective operations on a group of tasks

```c
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root, Mpi_comm comm);
```

sendbuf
recvbuf
count
datatype
op
root
comm
Reduction Operators

- **MPI_Op** argument can be user-defined or predefined operation
  - Values in send buffer on each node combined using operator
  - Result stored on root in receive buffer
  - Predefined Operations:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical And</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical Or</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise Or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical Exclusive OR</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Logical Bitwise OR</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and it's location</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum and it's location</td>
</tr>
</tbody>
</table>
Dot-Products with Reduction

float Serial_dot(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;
} /* Serial_dot */

float Parallel_dot(float local_x[], float local_y[], int n_bar) {

    float local_dot;
    float dot = 0.0;
    float Serial_dot(float x[], float y[], int m);

    local_dot = Serial_dot(local_x, local_y, n_bar);
    MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT,
               MPI_SUM, 0, MPI_COMM_WORLD);
    return dot;
} /* Parallel_dot */

Note: you may not make the first and second arguments to reduce the same!
Creating Reduction Operators

- Reduce supports user defined “plug in” functions for reduce (and scan):

  ```c
  int MPI_Op_create(MPI_User_function *function,
                    int commute, MPI_Op *op);
  ```

- Function – user supplied routine meeting specification
- Commute – is the operation commutative (T/F) ?
- Op – handle (pointer) to give to reduce
Creating Reduction Operators

- User-supplied functions must have 4 arguments as described below:

  ```
typedef void MPI_User_function(void *invec, void *
inoutvec, int *len, MPI_Datatype *datatype)
  ```

- Datatype is the type to perform the operation on
- len is the length of both invec and inoutvec
- The operation is performed on each element of invec with the corresponding element of inoutvec
- Results are returned in inoutvec
Scatter and Gather

- **Scatter** takes data stored on a single node and sends a piece of it to every node in a communicator.
- **Gather** merges a data set scattered across multiple tasks and collects it on a single task.
- Extremely useful in any kind of matrix operations.

![Diagram showing Scatter and Gather process]

<table>
<thead>
<tr>
<th>T0</th>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scatter

Gather
Scatter

- Distinct data sent from root to each task

```c
int MPI_Scatter(void* sndbuf, int sndcount, MPI_Datatype sndtype,
                 void* rcvbuf, int rcvcount, MPI_Datatype rcvtype, int root,
                 MPI_Comm comm);
```
Gather

- Data is sent from all tasks to the root
- Each task's data is stored in rank order

\[
\text{MPI\_Gather(} \text{void* sndbuf, int sndcount, MPI\_Datatype sndtype, void* rcvbuf, int rcvcount, MPI\_Datatype rcvtype, int root, MPI\_Comm comm)}\; \text{;} 
\]
Example - Dot product

• Take the dot product of all rows of matrix A with vector x
• Assume A and x are stored in a block-row distribution on tasks.
• Need to **gather** all elements of x onto a single task

```c
float local_x[]; /* Storage for local part of vector */
float global_x[]; /* Storage for global vector */
/* Assumes n is divisible by p */
MPI_Gather(local_x, n/p, MPI_FLOAT, global_x, n/p,
          MPI_FLOAT, 0, MPI_COMM_WORLD);
```

• X stored on task with rank 0 in communicator comm_world
• Scatter vector read from user across tasks to begin with:

```c
void Read_vector(
    char* prompt    /* in */,
    float local_x[] /* out */,
    int local_n    /* in */,
    int my_rank    /* in */,
    int p          /* in */) {

    int i;
    float temp[MAX_ORDER];
    if (my_rank == 0) {
        printf("%s\n", prompt);
        for (i = 0; i < p*local_n; i++)
            scanf("%f", &temp[i]);
    }
    MPI_Scatter(temp, local_n, MPI_FLOAT, local_x, local_n,
                MPI_FLOAT, 0, MPI_COMM_WORLD);
} /* Read_vector */
```
Example, continued...

- Gather matrix to print results:

```c
void Print_matrix(
    char*            title       /* in */,
    LOCAL_MATRIX_T   local_A     /* in */,
    int              local_m     /* in */,
    int              n           /* in */,
    int              my_rank     /* in */,
    int              p           /* in */) {

    int    i, j;
    float  temp[MAX_ORDER][MAX_ORDER];

    MPI_Gather(local_A, local_m*MAX_ORDER, MPI_FLOAT, temp,
               local_m*MAX_ORDER, MPI_FLOAT, 0, MPI_COMM_WORLD);

    if (my_rank == 0) {
        printf("%s\n", title);
        for (i = 0; i < p*local_m; i++) {
            for (j = 0; j < n; j++)
                printf("%4.1f ", temp[i][j]);
        printf("\n");
    }
}
} /* Print_matrix */
```
Gather to All

- Gather, but all tasks get result
  
  \[
  \text{MPI\_Allgather(}\text{void}\ast \text{sndbuf,}\\
  \text{int sndcount, MPI\_Datatype sndtype,}\\
  \text{void}\ast \text{rcvbuf,}\\
  \text{int rcvcount, MPI\_Datatype rcvtype,}\\
  \text{int root, MPI\_Comm comm});
  \]

<table>
<thead>
<tr>
<th>T0</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>A1</td>
</tr>
<tr>
<td>T2</td>
<td>A2</td>
</tr>
<tr>
<td>T3</td>
<td>A3</td>
</tr>
</tbody>
</table>

Allgather

<table>
<thead>
<tr>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
</tr>
<tr>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
</tr>
<tr>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
</tr>
</tbody>
</table>

• Scatter combined with gather-to-all

\[
\text{MPI\_Alltoall}(\text{void}\* \text{sndbuf}, \\
\text{int} \text{sndcount}, \text{MPI\_Datatype} \text{sndtype}, \\
\text{void}\* \text{rcvbuf}, \\
\text{int} \text{rcvcount}, \text{MPI\_Datatype} \text{rcvtype}, \\
\text{MPI\_Comm} \text{comm})
\]
GatherV

• Like gather, but with a stride:
  
  \[
  \text{int } \text{MPI}_\text{Gatherv}( \text{void }* \text{sendbuf}, \text{int } \text{sendcount}, \\
  \text{MPI}_\text{Datatype } \text{sendtype}, \text{void }* \text{recvbuf}, \text{int}
  \\
  *\text{recvcounts}, \text{int }*\text{displs}, \text{recvtype}, \text{root}, \text{comm});
  \]

• Allows a varying amount of data to be gathered from each process
  – recvcounts now an array

• Allows spacing between the blocks:
  – the “displs” argument lists displacements between start of received blocks
## All Reduce

- Reduce, but all tasks receive result

```c
MPI_Allreduce(void* sendbuf, 
void* recvbuf, int count, 
MPI_Datatype datatype, 
MPI_Op op, MPI_Comm comm)
```

<table>
<thead>
<tr>
<th></th>
<th>A0</th>
<th>B0</th>
<th>C0</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A1</th>
<th>B1</th>
<th>C1</th>
<th>D1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A2</th>
<th>B2</th>
<th>C2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A3</th>
<th>B3</th>
<th>C3</th>
<th>D3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Allreduce

<table>
<thead>
<tr>
<th></th>
<th>ΣA</th>
<th>ΣB</th>
<th>ΣC</th>
<th>ΣD</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ΣA</th>
<th>ΣB</th>
<th>ΣC</th>
<th>ΣD</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ΣA</th>
<th>ΣB</th>
<th>ΣC</th>
<th>ΣD</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ΣA</th>
<th>ΣB</th>
<th>ΣC</th>
<th>ΣD</th>
</tr>
</thead>
<tbody>
<tr>
<td>T3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
AllReduce

- Same syntax as reduce; however, result is left on all tasks not just root task
- Equivalent result to calling `reduce` then `broadcast`
- On a switched network, should be same performance as reduce
Reduce Scatter

• Reduce, results scattered to tasks

```c
MPI_Reduce_scatter(void* sndbuf,
                    void* rcvbuf, int *rcvcounts,
                    MPI_Datatype datatype,
                    MPI_Op op, MPI_Comm comm);
```

```
\begin{tabular}{|c|c|c|c|}
\hline
T0 & A0 & B0 & C0 & D0 \\
\hline
T1 & A1 & B1 & C1 & D1 \\
\hline
T2 & A2 & B2 & C2 & D2 \\
\hline
T3 & A3 & B3 & C3 & D3 \\
\hline
\end{tabular}
```

\begin{align*}
\Sigma A \\
\Sigma B \\
\Sigma C \\
\Sigma D
\end{align*}
Scan

- Performs prefix reduction
  
  \[
  \text{MPI\_Scan}(\text{void* sndbuf,} \\
  \text{void* rcvbuf, int count,} \\
  \text{MPI\_Datatype datatype,} \\
  \text{MPI\_Op op, MPI\_Comm comm})
  \]

- User defined functions are the same as with reduce

- \text{MPI\_IN\_PLACE} option as second argument leaves result in the input buffer

- Inclusive scan, exclusive scan is available with \text{MPI\_Exscan()}
Communication Modes

- Convered on next pages ...
  - Standard Mode
  - Buffered Mode
    \[\text{MPI} \_ \text{Bsend}(\text{buf}, \text{count}, \text{datatype}, \text{dest},\]
    \[\text{tag}, \text{MPI} \_ \text{COMM} \_ \text{WORLD})];
  - Synchronous Mode
    \[\text{MPI} \_ \text{Ssend}(\text{buf}, \text{count}, \text{datatype}, \text{dest},\]
    \[\text{tag}, \text{MPI} \_ \text{COMM} \_ \text{WORLD})];
  - Ready Mode
    \[\text{MPI} \_ \text{Rsend}(\text{buf}, \text{count}, \text{datatype}, \text{dest},\]
    \[\text{tag}, \text{MPI} \_ \text{COMM} \_ \text{WORLD})];
Standard Mode

- Message is buffered if system space is available
- Otherwise call will block until message delivered or system space becomes available
- Completion of send does not imply message delivered
Buffered Mode

- Program provides buffer space for messages
  
  `MPI_Buffer_attach(buffer, size);`
  
  `MPI_Buffer_detach(buffer_addr, size);`

- Send returns as soon as message is copied into buffer

- Completion of send does not imply message delivered
Synchronous Mode

- Send blocks until matching receive is posted and data either buffered by system or delivered
- Completion of send does not imply completion of receive, but DOES imply start of receive
Ready Mode

• Send can only be correctly called if the matching receive has already been posted.
• Send completes when data is either received or copied to system buffer.
• Used primarily on systems where data flow can be optimized if receiver is ready for message.
Non-blocking Send andRecv

• Each form of send and recv have corresponding non-blocking versions:
  – MPI_Isend, MPI_Ibsend, MPI_Irsend, MPI_Issend, MPI_Irecv
• Non blocking versions
  – return immediately
  – do not complete until some time later
• A Request structure is returned
  – keeps track of each outstanding operation
  – can be used to test for or wait for completion
Non-blocking Send/Recv Details

- Added argument for MPI_Request structure
  ```c
  MPI_Request request;
  MPI_Isend(buffer, count, datatype, src, tag, MPI_COMM_WORLD, &request);
  ```
- Added argument for MPI_Request structure
  ```c
  MPI_Request request;
  MPI_Irecv(buffer, count, datatype, dest, tag, MPI_COMM_WORLD, &request);
  ```
Ending Non-blocking Requests

• Test for completion
  
  MPI_Status status;
  int flag;
  MPI_Test(&request, &flag, &status);

• Wait for completion
  
  MPI_Wait(&request, &status);
Checking for Multiple Completions

- **MPI_Waitany** or **MPI_Testany** will return if any one of the requests is complete.
  ```
  int MPI_Waitany(int count, MPI_Request *requests, int *index, MPI_Status *status)
  int MPI_Testany(int count, MPI_Request *requests, int *index, int *flag, MPI_Status *status)
  ```

- **MPI_Waitall** or **MPI_Testall** will return if ALL of the requests are complete
  ```
  int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses)
  int MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses)
  ```

- **MPI_Waitsome** or **MPI_Testsome** will return after completing as many requests as possible without blocking
  ```
  int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount, int *indices, MPI_Status *statuses)
  int MPI_Testsome(int incount, MPI_Request *requests, int *outcount, int *indices, MPI_Status *statuses)
  ```
Probe, Iprobe, and Cancel

- **MPI_Probe** waits for a message that will match a specific `recv`
  ```c
  int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
  ```
- **MPI_Iprobe** tests for a message that will match a `recv`
  ```c
  int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
  ```
- **MPI_Cancel** effectively undoes a pending operation
  ```c
  int MPI_Cancel(MPI_Request *request)
  ```