MPI: Beyond the Basics

David McCaughan

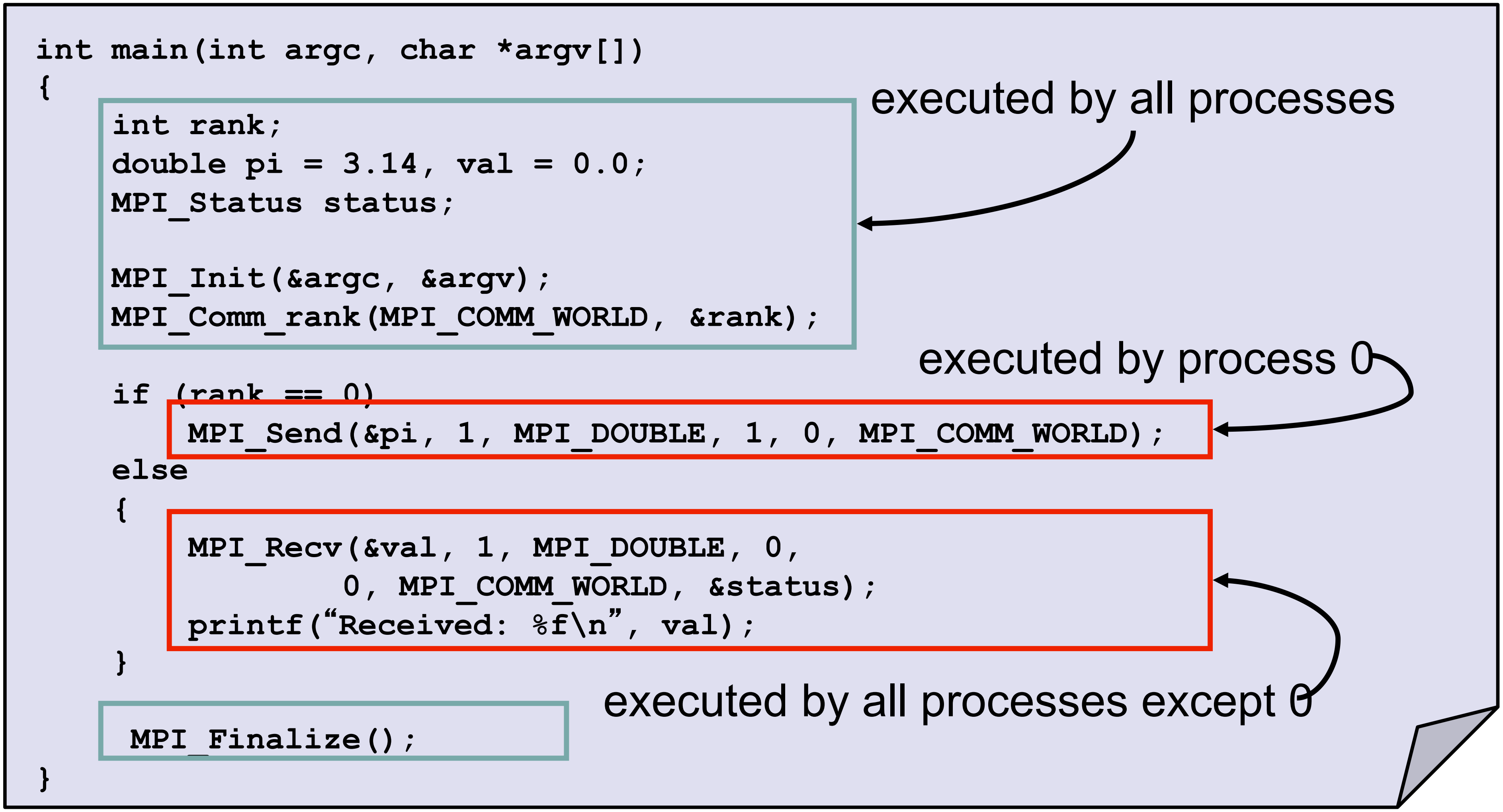
HPC Analyst, SHARCNET

dbm@sharcnet.ca

Review: “The Basics”

- `MPI_Init()`
- `MPI_Finalize()`
- `MPI_Comm_rank()`
- `MPI_Comm_size()`
- `MPI_Send()`
- `MPI_Recv()`

Review: sending/receiving



```
int main(int argc, char *argv[])
{
    int rank;
    double pi = 3.14, val = 0.0;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        MPI_Send(&pi, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
    else
    {
        MPI_Recv(&val, 1, MPI_DOUBLE, 0,
                 0, MPI_COMM_WORLD, &status);
        printf("Received: %f\n", val);
    }

     MPI_Finalize();
}
```

executed by all processes

executed by process 0

executed by all processes except 0

Understanding parallelism: Euclidian Inner Product

- Compute a weighted sum

$$s = \sum_i a_i b_i$$

Sequential Algorithm:

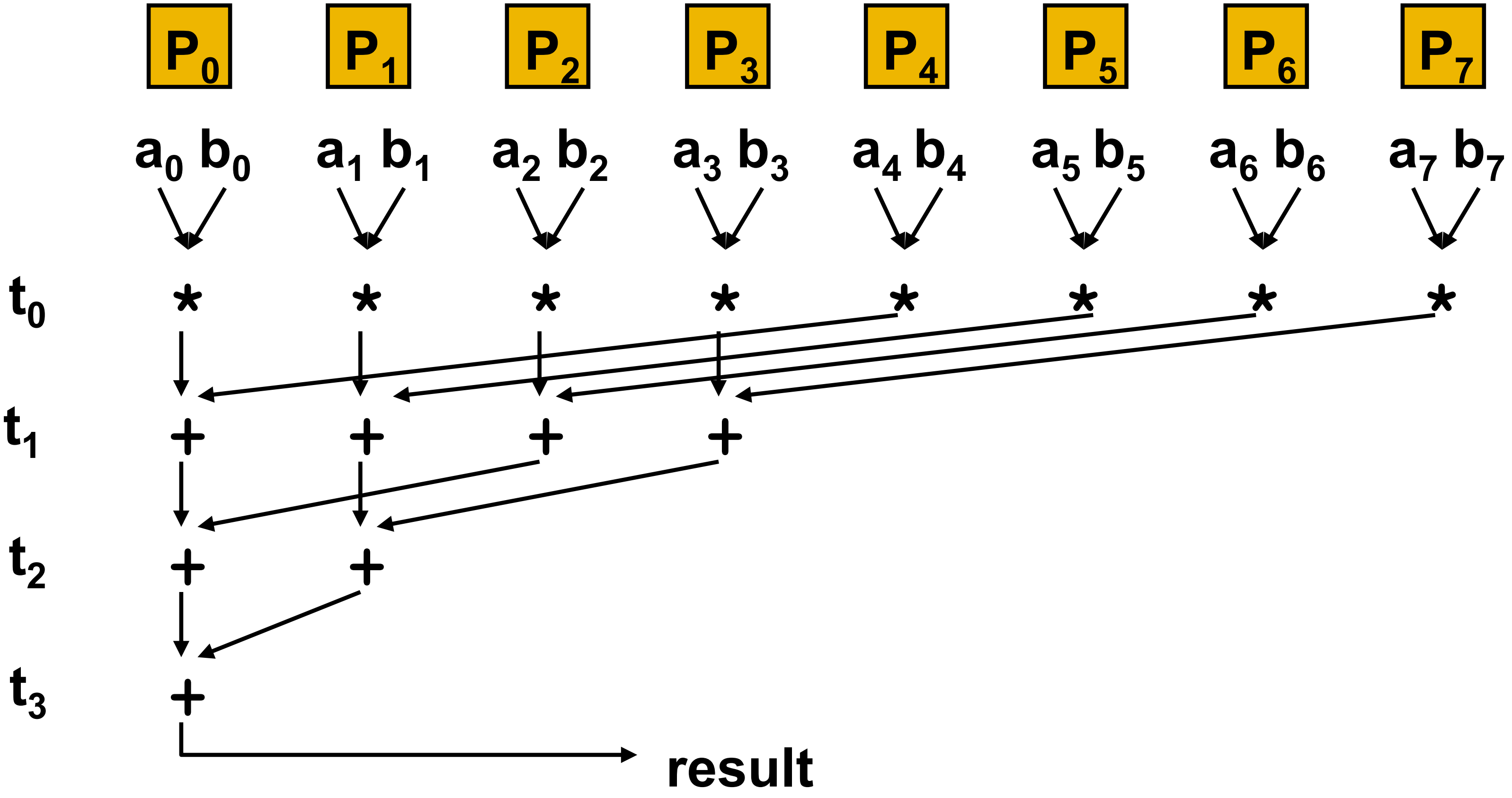
given arrays a, b of size N

```
s := 0
do i := 1,N
     s := s + (a[i] * b[i])
```

- run-time proportional to N

Thinking in Parallel

- Assume N = 2^x processors (x an integer)



Parallel Inner Product Algorithm

- All processors executing same algorithm

```
x := a[k] * b[k]

do t := ($\log_2 N$-1), 0, -1
     if $2^t$ <= k < $2^{t+1}$
          send x to $P_{k-2^t}$
      else if k < $2^t$
          receive y from $P_{k+2^t}$
          x := x + y

if k = 0
     output x
```

- run-time proportional to $\log_2 N$

Parallel Euclidian Inner Product Implementation

```
...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &k);
    MPI_Comm_size(MPI_COMM_WORLD, &N);

    op1 = a[k] * b[k];
    v2_t = pow(2,loga(N,2)-1);

    while(v2_t > 0)
    {
        if ((k >= v2_t) && (k < (v2_t << 1)))
            MPI_Send(&op1, 1, MPI_DOUBLE, (k-v2_t), 0, MPI_COMM_WORLD);
        else if (k < v2_t)
        {
            MPI_Recv(&op2, 1, MPI_DOUBLE,
                    (k+v2_t), 0, MPI_COMM_WORLD, &status);
            op1 = op1 + op2;
        }
        v2_t = v2+t >> 1;
    }
```

Parallel Euclidian Inner Product Implementation (cont.)

```
    if (k == 0)
        printf(“result = %f\n”, op1);

    MPI_Finalize();
...
```

- Note:
  - this example illustrates basic parallelism and communication well
  - not the best example of how MPI might really be used
    - parallelism here is quite fine-grained
  - alternatives?

Safe vs. Unsafe MPI

- The MPI standard does not require buffering of data communications
  - many MPI implementations do provide it however
  - should you assume that there is buffering available or not?

- If there is no buffering
  - a process will block on `MPI_Send` until a `MPI_Recv` is called to allow for delivery of the message
  - a process will block `on MPI_Recv` until a `MPI_Send` is executed delivering the message

- A ***safe*** MPI program is one that does not rely on a buffered underlying implementation in order to function correctly

Safe vs Unsafe MPI (cont.)

```
if (rank == 0)
{
    MPI_Recv(from 1);
    MPI_Send(to 1);
}
else if (rank == 1)
{
    MPI_Recv(from 0);
    MPI_Send(to 0);
}
```

deadlock
receives executed on both processes before matching send;
buffering irrelevant

```
if (rank == 0)
{
    MPI_Send(to 1);
    MPI_Recv(from 1);
}
else if (rank == 1)
{
    MPI_Send(to 0);
    MPI_Recv(from 0);
}
```

unsafe
may work if buffers are larger than messages; if unbuffered, or messages are sufficiently large this code will fail

```
if (rank == 0)
{
    MPI_Send(to 1);
    MPI_Recv(from 1);
}
else if (rank == 1)
{
    MPI_Recv(from 0);
    MPI_Send(to 0);
}
```

safe
sends are paired with corresponding receives between processes;
no assumptions made about buffering

Ensuring a Program is Safe

- Must work the same using `MPI_Send` and `MPI_Ssend`
  - `MPI_Ssend` is synchronous mode send

- Strategies for avoiding deadlock
  - pay attention to order of send/receive in communication operations
  - use synchronous or buffered mode communication
  - use `MPI_Sendrecv`
  - use non-blocking communication

MPI_Sendrecv

```
int MPI_Sendrecv
(
    void *s_msg,
    int s_count,
    MPI_Datatype s_datatype,
    int dest,
    int s_tag,
    void *r_msg,
    int r_count,
    MPI_Datatype r_datatype,
    int source,
    int r_tag,
    MPI_Comm comm,
    MPI_Status *status
);
```

- Combined send and receive operation
  - allows MPI to deal with order of calls to reduce potential for deadlock
  - does not imply pairwise send/receive
- Can be satisfied by regular sends/receives on other processes
  - does not remove all potential for deadlock

Example (sendrecv.c)

```
...
    if (rank == 0)
    {
        int in, out = 5;

        MPI_Sendrecv(&out, 1, MPI_INT, 1, 0,
                &in, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);

        printf("P0 Received: %d\n",in);
    }
    else
    {
        int in, out = 25;

        MPI_Recv(&in, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        MPI_Send(&out, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

        printf("P1 Received: %d\n", in);
    }
...
```

Non-Blocking Communication

- Standard MPI sends and receives don’t return until their arguments can be safely modified by the calling process
  - sending: message envelope created, data sent/buffered
  - receiving: message received, data copied to provided buffer

- If this activity can take place concurrently with computation we aren’t fully utilizing available resources
  - assuming separate computing resource for communication

Non-Blocking Communication (cont.)

- Non-blocking communication avoids deadlock and allows computation to be interleaved with these activities
  - call to send/receive "posts" the operation
  - must subsequently explicitly complete the operation
  - NOTE: non-blocking send/recv can be matched by standard recv/send

MPI_Isend
MPI_Irecv

```
int MPI_Isend
(
    void *message,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm,
    MPI_Request *request
);
```

```
int MPI_Irecv
(
    void *message,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Request *request
);
```

- Post non-blocking send

- Post non-blocking receive

MPI_Wait

```
int MPI_Wait
(
    MPI_Request *request,
    MPI_Status *status
);
```

- Complete non-blocking send/ receive

- `request`
  - a handle for your operation provided by the system
  - identifies the operation when you subsequently complete it

Non-blocking Communication Example

```
...
    if (rank == 0)
    {
        MPI_Isend(&val1, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &req1);
        MPI_Irecv(&val2, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &req2);
        /* do some computational work here */
        MPI_Wait(&req1, &status);
        MPI_Wait(&req2, &status);
        val1 = tmp;
     }
    else
    {
        MPI_Irecv(&val2, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &req2);
        MPI_Isend(&val1, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &req1);
        /* do some computational work here */
        MPI_Wait(&req2, &status);
        MPI_Wait(&req1, &status);
        val1 = tmp;
     }
...
```

Communication Modes

- Standard
  - buffering of receives is system dependent (this is the default)
- Synchronous
  - processes will block on send until a corresponding receive has been initiated (requires no buffering - otherwise identical to standard)
- Buffered
  - user allocated buffer used for sends
- Ready
  - sends are only valid once corresponding receive has been initiated (may be faster on some systems - we will not look at this)

Communication Patterns

- It is uncommon for communication between processes to be completely arbitrary

- Tends to follow specific patterns
  - one sends to all (*broadcast*)
  - all send to one (*reduction*)
  - all send to all (*all broadcast/reduction*)
  - subset sends to subset
    - modularized by data and/or task

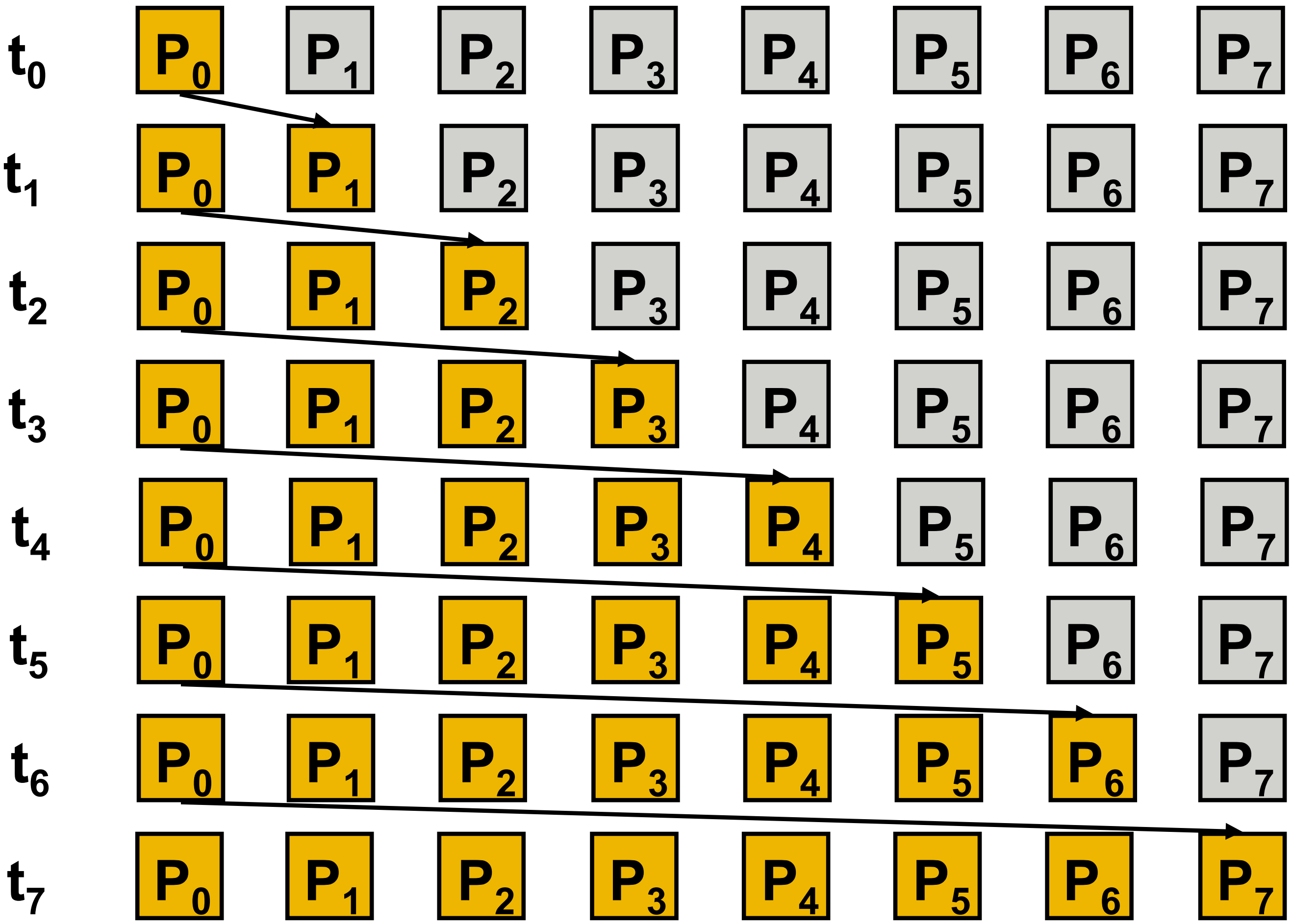
Broadcasting

- A process has a value that needs to be sent to all other processes

- Serial method:
  - run a loop that sends value to each other process in turn
  - N-1 iterations to send value to all other processes
  - number of iterations proportional to number of processes
    - O(n)

```
do i := 1, N-1
     send data from $P_0$ to $P_i$
```

Serial Broadcast


t_0 P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
t_1 P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
t_2 P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
t_3 P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
t_4 P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
t_5 P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
t_6 P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
t_7 P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7

Parallel Broadcast

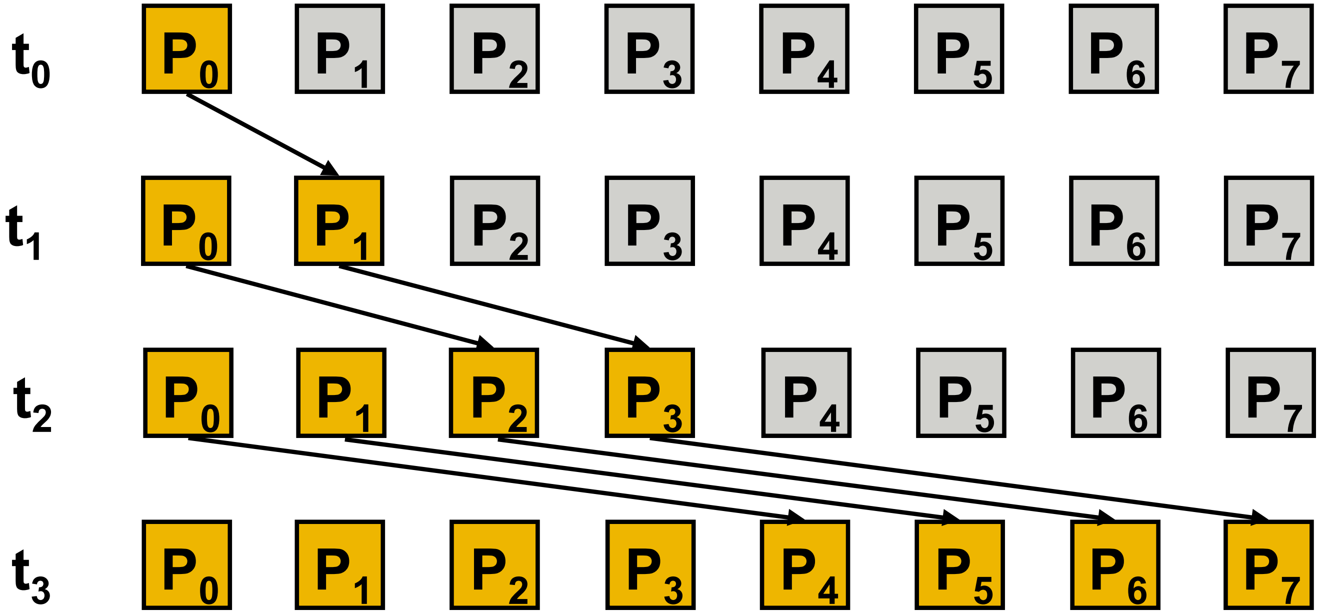
- Broadcasting can take advantage of available parallelism

- Consider:
  - step 1: P_0 sends to P_1
  - step 2: P_0 sends to P_2, P_1 sends to P_3
  - step 3: $P_0 \rightarrow P_4$, $P_1 \rightarrow P_5$, $P_2 \rightarrow P_6$, $P_3 \rightarrow P_7$ ...
  - *note: there is no canonical order for communication*

- $\lceil \log_2 N \rceil$ steps
  - $O(\log_2 N)$

- Must consider issues of knowing sender/receiver in broadcast implementation

Parallel Broadcast (cont.)



- This is a common parallel technique called *recursive doubling*
  - progress toward completion of process doubled (or halved) at each step by doubling (halving) number of processes involved at each step

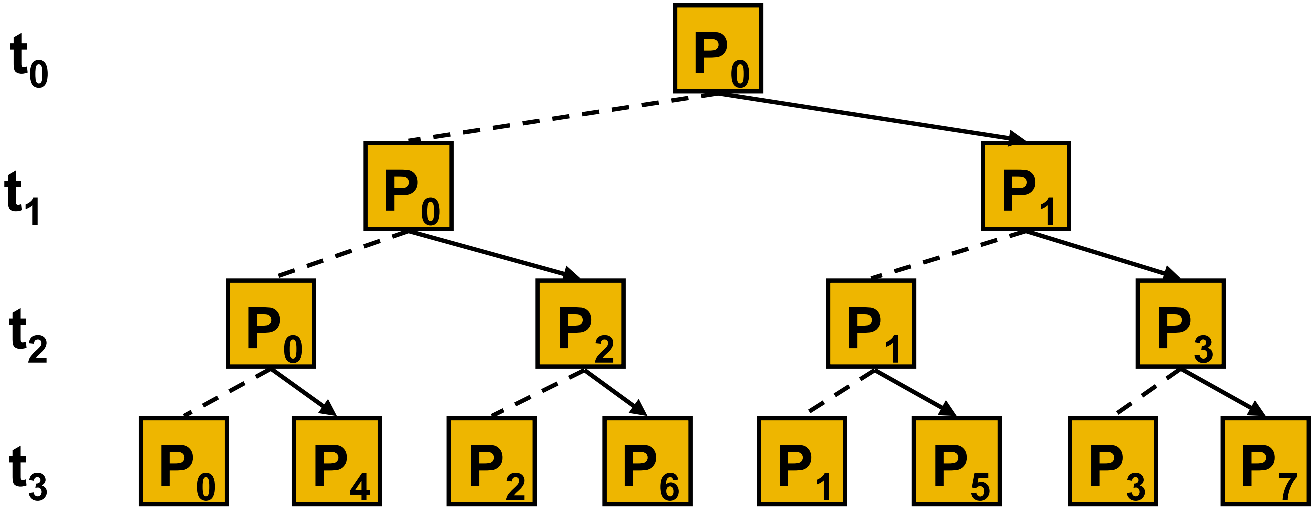
Broadcast Implementation

```
...
    int step, v2_t, rank, size;
    double value = 0.0;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_SIZE, &size);
    ...
    for (step = 0; step < ceil(loga(size,2)); step++)
    {
        v2_t = pow(2,step);

        if ((rank < v2_t) && (v2_t + rank < size))
            MPI_Send(&value, 1, MPI_DOUBLE,
                    (v2_t + rank), 0, MPI_COMM_WORLD);
        else if (rank < (v2_t << 1))
            MPI_Recv(&value, 1, MPI_DOUBLE,
                    (rank - v2_t), 0, MPI_COMM_WORLD, &status);
    }
...
```

Tree-Based Communication

- The pattern by which processes join the computation is tree-based
  - note: depth of balanced binary tree with N nodes is $\lceil \log_2 N \rceil$

- Tree-based communication is very common when values are being distributed or collected as it makes optimal use of communication resources



Adapted from *Parallel Programming with MPI*, Peter S. Pacheco, 1997.

Reduction

- Partial results held by multiple processes need to be combined and sent to a single process

- Serial reduction (O(N) steps)

```
do i = N-2, 0, -1
  $result_i$ := combine($result_i$, $result_{i+1}$)
```

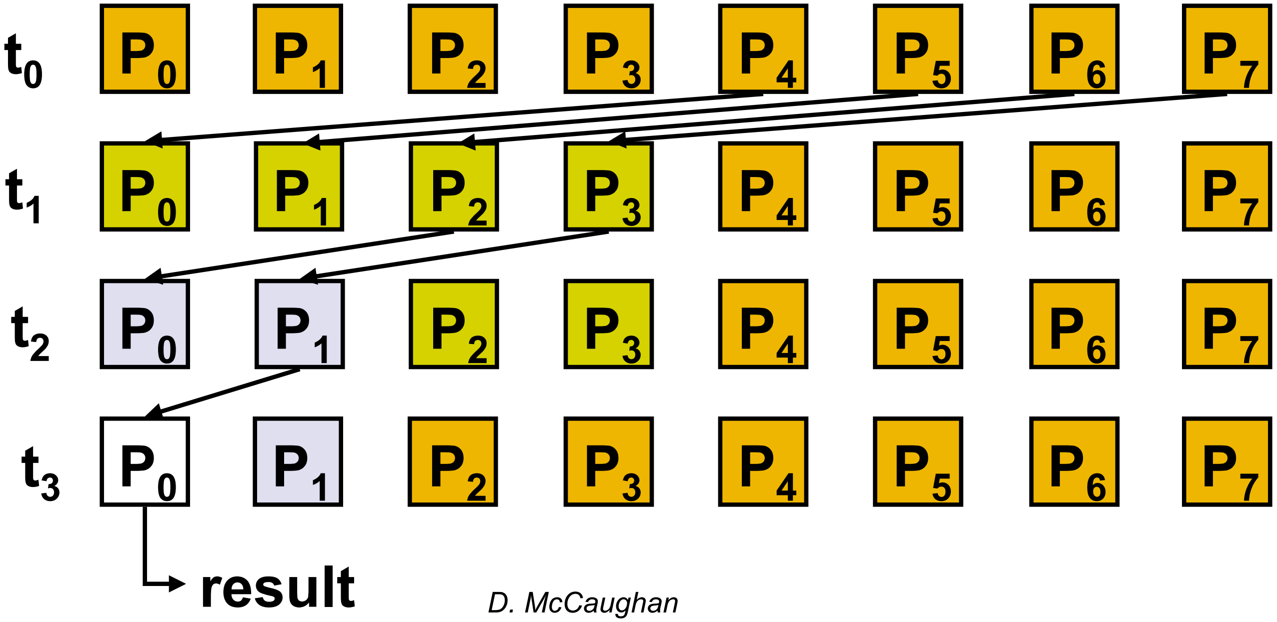
Reduction (cont.)

- Parallel reduction
  - *there is no canonical order for communication*

```
in parallel do (for each processor i)
  do t := ceil($\log_2$N)-1, 0, -1
    if i < $2^t$ AND k+$2^t$ < N
      $result_i$ := combine($result_i$, $result_{i+2^t}$)
final result on $P_0$
```

Parallel Reduction

- Parallel reduction is essentially the reverse of the tree-based broadcast with added operations at each step to combine partial results where necessary.
  - recall: parallel inner product --- result computed by reduction of partial sums
  - inverted binary tree, depth = $\lceil \log_2 N \rceil$
    - $O(\log_2 N)$


t_0
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
t_1
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
t_2
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
t_3
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
result

Common Reduction Operations

- Simple arithmetic operations
  - sum, difference, product, quotient
- Properties
  - maximum/minimum value, location of max/min value
- Logical operations
  - AND, OR, XOR
- Bitwise operations
  - AND, OR, XOR
- Sorting*

Reduction Implementation

```
...
    int local, received;
    MPI_Comm_rank(MPI_COMM_WORLD, &k);
    MPI_Comm_size(MPI_COMM_WORLD, &N);

    for (v2_t = pow(2,ceil(loga(N,2)-1); v2_t > 0; v2_t >>=  1;
    {
        if ((k >= v2_t) && (k < (v2_t << 1)))
            MPI_Send(&local, 1, MPI_INT, (k-v2_t), 0, MPI_COMM_WORLD);
        else if ((k < v2_t) && (k + v2_t) < N)
        {
            MPI_Recv(&received, 1, MPI_INT,
                    (k+v2_t), 0, MPI_COMM_WORLD, &status);
            /*
             * update “local” partial result based on whatever
             * operation is desired (e.g. max, sum, etc.)
             */
            local = operation(local,recieved);
        }
    }
...
```

Reduction to All Processes

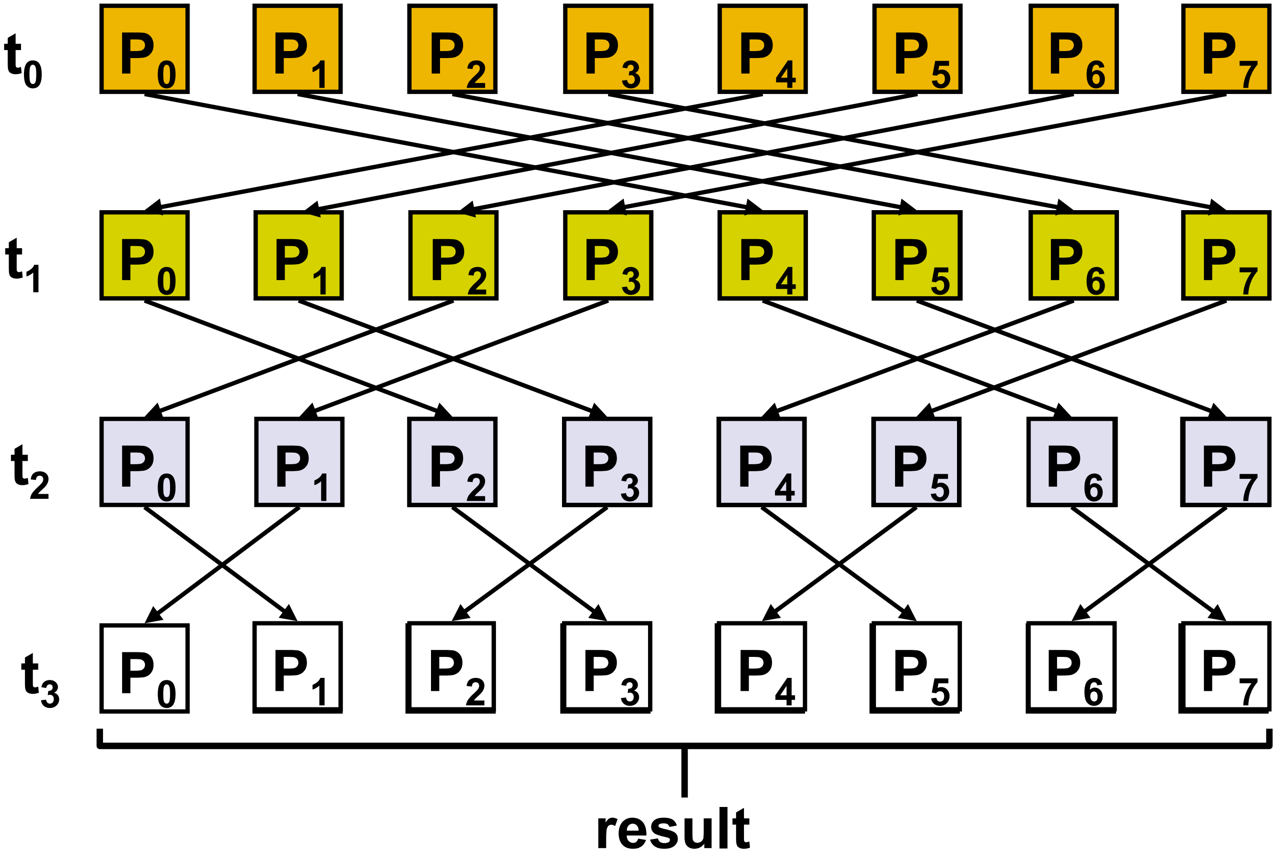
- What happens if we want a reduced result on all processes?

- *Option 1:*
  - standard reduction followed by broadcast
  - can we do this more efficiently?

Reduction to All Processes (cont.)

- *Option 2:*
  - note that at each step in a reduction the processes that send their values no longer contribute to computation
  - introduce communication between both halves of the subset of processors used at each step, e.g.
    - 4 → 0, 5 → 1, 6 → 2, 7 → 3, while at the same time
    - 0 → 4, 1 → 5, 2 → 6, 3 → 7
    - etc.
  - result ends up computed on all processes
- This is a *Butterfly* reduction

Butterfly Reduction



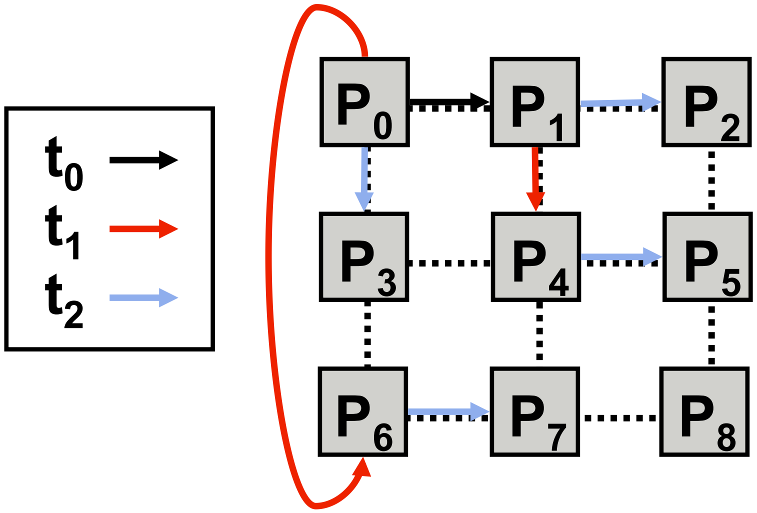
A Word About Messages…

- How does MPI distinguish messages from one another?
  - who it is from
  - who it is for
  - a user defined “label” used to mark messages
  - the group of processes in which communication is taking place
  - other information (*attributes*)

- The ***message envelope*** contains at least these pieces of information (possibly more depending on implementation)
  - sender rank
  - receiver rank
  - tag
  - communicator

Issues of Cluster Topology

- If we are interested in efficiency, we would like to ensure that our communication pattern is optimal given cluster topology
  - problems:
    - irregular communication patterns complicate programming
    - may not even be aware of the topology
  - e.g. broadcast in a toroid (all communications are 1 step)



Desirable pattern can be mathematically irregular

t_0 :

$0 \rightarrow 1$

t_1 :

$0 \rightarrow 6$, $1 \rightarrow 4$

t_2 :

$0 \rightarrow 3$, $1 \rightarrow 2$, $6 \rightarrow 7$, $4 \rightarrow 5$

Collective Communication

- We think of sends and receives as distinct events even though they are abstractly paired on sender/receiver
- MPI provides a number of collective communication operations
  - all processes participate in an operation together
  - library implementations provide convenience over handling these details manually, and an opportunity for efficiency where it is possible
- Collective data movement
  - `MPI_Bcast`, `MPI_Gather`, `MPI_Allgather`, `MPI_Scatter`
- Collective computation
  - `MPI_Reduce`, `MPI_Allreduce`, `MPI_Barrier`

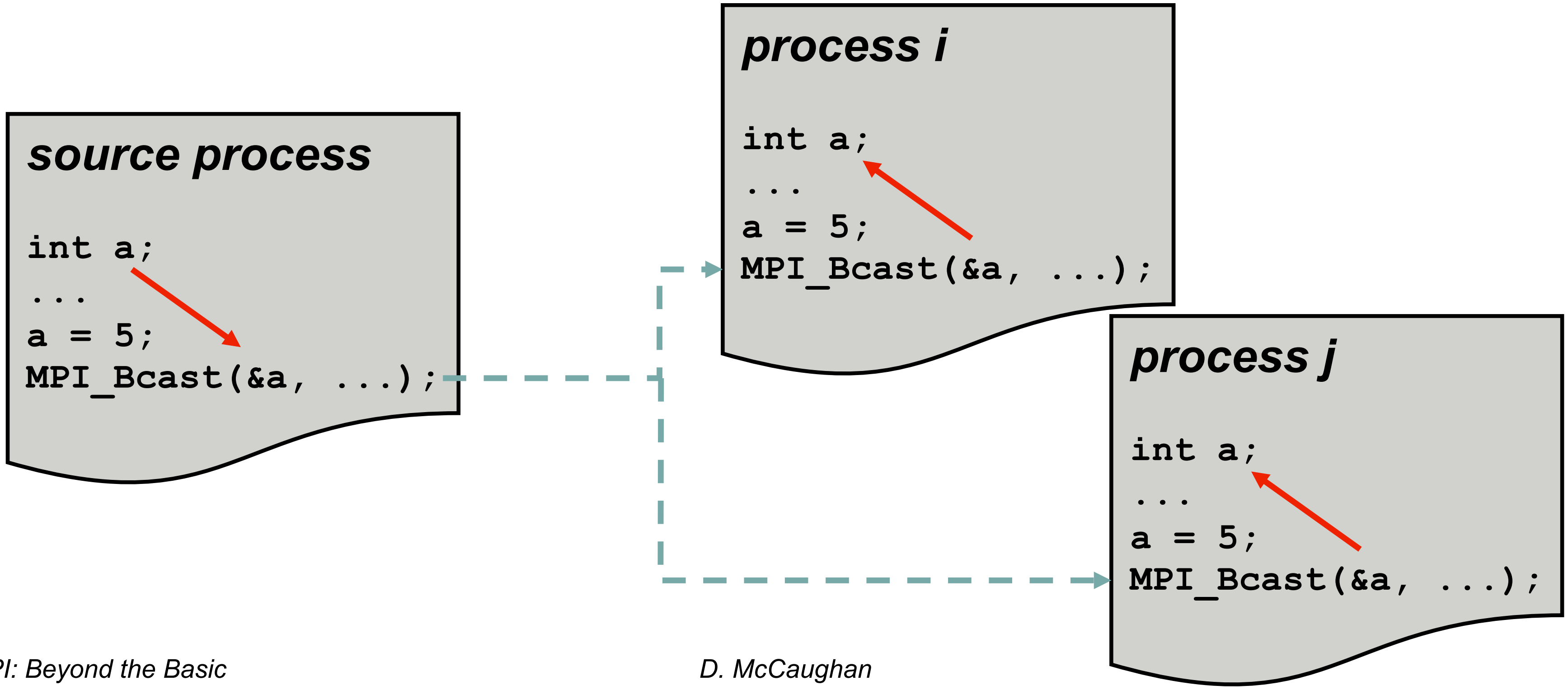
MPI_Bcast

```
int MPI_Bcast
(
    void *buffer,
    int count,
    MPI_Datatype datatype,
    int source,
    MPI_Comm comm
);
```

- Collective data movement operation
  - abstracts mechanics of the broadcast
  - improved portability
  - allows vendor or site to optimize communication in library function
- `buffer`
  - input/output buffer (depending on processes)
- `source`
  - rank of sending process

MPI Broadcast Operation

- All processes call `MPI_Bcast`
  - contents of `buffer` on process `sender` copied to `buffer` on all other processes in communicator (point of synchronization)



source process

```
int a;
...
a = 5;
MPI_Bcast(&a, ...);
```

process i

```
int a;
...
a = 5;
MPI_Bcast(&a, ...);
```

process j

```
int a;
...
a = 5;
MPI_Bcast(&a, ...);
```

Example (bcast.c)

```
...
    if (rank == 0)
    {
        nargs = argc-1;
        if (!(args = calloc(sizeof(int),nargs)))
        ...

        MPI_Bcast(&nargs, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(args, nargs, MPI_INT, 0, MPI_COMM_WORLD);
    }
    else
    {
        MPI_Bcast(&nargs, 1, MPI_INT, 0, MPI_COMM_WORLD);

        if (!(args = calloc(sizeof(int),nargs)))
        ...

        MPI_Bcast(args, nargs, MPI_INT, 0, MPI_COMM_WORLD);
    }
...
```

MPI_Gather

```
int MPI_Gather
(
    void *outbuf,
    int n_out,
    MPI_Datatype out_type,
    void *inbuf,
    int n_in,
    MPI_Datatype in_type,
    int dest,
    MPI_Comm comm
);
```

- Collective data movement operation
  - collects data subsets from all processes; store them on one
- `dest` and `comm` must be the same on all processes in the communicator
  - gathers data from *all* processes
- Storage occurs in rank order and is contiguous

MPI_Scatter

```
int MPI_Scatter
(
    void *outbuf,
    int n_out,
    MPI_Datatype out_type,
    void *inbuf,
    int n_in,
    MPI_Datatype in_type,
    int source,
    MPI_Comm comm
);
```

- Collective data movement operation
  - segments contiguous data on one process and sends each segment to another process in the communicator
  - dispersal is contiguous and in rank order
- `source` and `comm` must be the same on all processes in the communicator
  - scatters data to *all* processes

MPI_Allgather

```
int MPI_Allgather
(
    void *outbuf,
    int n_out,
    MPI_Datatype out_type,
    void *inbuf,
    int n_in,
    MPI_Datatype in_type,
    MPI_Comm comm
);
```

- Collective data movement operation
  - uses butterfly communication to collect data segments from all processes and store them on all
  - storage occurs in rank order and is contiguous
- `comm` must be the same on all processes in the communicator
  - gathers data from and to *all* processes

MPI_Reduce

```
int MPI_Reduce
(
    void *operand,
    void *result,
    int count,
    MPI_Datatype datatype,
    MPI_Op operator,
    int dest,
    MPI_Comm comm
);
```

- Collective computation operation
  - abstracts mechanics of reduction
  - all processes contribute data to be used in a binary operation
- `operand`
  - data on each process to be reduced
- `operator`
  - operation to be performed during reduction
- `dest`
  - rank of process to receive result

MPI Reduction Operations

- `MPI_Op`
  - MPI predefines several common reduction operations
  - it is also possible to define your own

MPI_MAX	MPI_LAND
MPI_MIN	MPI_LOR
MPI_MAXLOC	MPI_LXOR
MPI_MINLOC	MPI_BAND
MPI_SUM	MPI_BOR
MPI_PRODUCT	MPI_BXOR

Example: Simplified Euclidean Inner Product

```
...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &k);

    op1 = a[k] * b[k];
    v2_t = pow(2,ceil(loga(N,2)-1);

    /*
     * summation of all op values - result to process 0
     */
    MPI_Reduce(&op, &result, 1,
            MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (k == 0)
        printf(“result = %f\n”, result);
...
```

MPI_Allreduce

```
int MPI_Allreduce
(
    void *operand,
    void *result,
    int count,
    MPI_Datatype datatype,
    MPI_Op operator,
    MPI_Comm comm
);
```

- Collective computation operation
  - recall: butterfly reduction
  - no `dest` process --- result stored on all processes
- e.g. inner product with result to all:

```
/* summation of all op values - result to all processes */

MPI_Allreduce(&op, &result, 1,
       MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

MPI_Barrier

```
int MPI_Barrier
(
    MPI_Comm comm
);
```

- Collective operation synchronizing all processes
  - ensures all processes have reached the same point in processing

- All processes in communicator block until every process calls `MPI_Barrier`
  - this must be using sparingly and only where appropriate
  - e.g. timing for task completion

- Too often used as a programming crutch to force synchronization
  - note that entire job becomes slaved to these barrier points
  - time to barrier completion now dictated by the slowest process
  - this is a design issue --- are you using it appropriately?

Special Purpose Communication

- While there are hundreds of functions we have not considered, there are a few worth pointing out
  - `MPI_Sendrecv_replace`
    - use same buffer for a combined send/receive operation
  - `MPI_Gatherv`
    - gather variable sized data from all processes
  - `MPI_Scatterv`
    - distribute variable sized data to all processes
  - `MPI_Allgatherv`, `MPI_Alltoall`, `MPI_Alltoallv`, `MPI_Reduce_scatter`

Single Buffer Send/Receive

```
int MPI_Sendrecv_replace
(
    void *mesg,
    int count,
    MPI_Datatype datatype,
    int dest,
    int s_tag,
    int source,
    int r_tag,
    MPI_Comm comm,
    MPI_Status *status
);
```

- Generally, it is unsafe to use the same buffer for both output and input in the same function call
- `mesg`
  - data referenced by `mesg` is first sent to `dest`, then data is received from `source` into the same buffer
- note that the same amount and type of data must be both sent and received

Gather Variable Sized Data

```
int MPI_Gatherv
(
    void *outbuf,
    int n_out,
    MPI_Datatype outtype,
    void *inbuf,
    int n_in[],
    int offsets[],
    MPI_Datatype intype,
    int dest,
    MPI_Comm comm
);
```

- MPI_Gather required the size of the data on each process be the same

- `n_in`
  - `n_in[i]` contains number of values to receive from process `i`

- `offsets`
  - `offsets[i]` contains offset into `inbuf` at which to begin storing values recieved from process `i`

Scatter Variable Sized Data

```
int MPI_Scatterv
(
    void *outbuf,
    int n_out[],
    int offsets[],
    MPI_Datatype outtype,
    void *inbuf,
    int n_in,
    MPI_Datatype intype,
    int source,
     MPI_Comm comm
);
```

- MPI_Scatter similarly required distribution of data be uniform in size
- `n_out`
  - `n_out[i]` contains number of values to send to process `i`
- `offsets`
  - `offsets[i]` contains offset into outbuf at which to begin sending values to process `i`

Other Data Distribution Functions

- `MPI_Allgatherv`
  - gather variable sized data to all processes

- `MPI_Alltoall` / `MPI_Alltoallv`
  - each process sends a different set of data to all other processes
  - effect the same as a series of scatters by all processes

- `MPI_Reduce_scatter`
  - perform a reduction operation with results scattered to all processes

Programming Issues

- Buffer dependencies
  - consider our “hello, world” example where all processes send to their left while receiving from their right (this will apply to our next example as well)
  - if the sends block, the receives will be staggered starting from the right
    - the one process that didn’t send receives, allowing the process to its right to unblock and complete its send
    - this process can now initiate its receive allowing the process to its right to unblock and complete its send, etc.
  - this virtually negates any benefit from parallel communication
  - SOLUTION:
    - pair sends and receives, operation specific to process
    - e.g. odd processes send, even processes receive

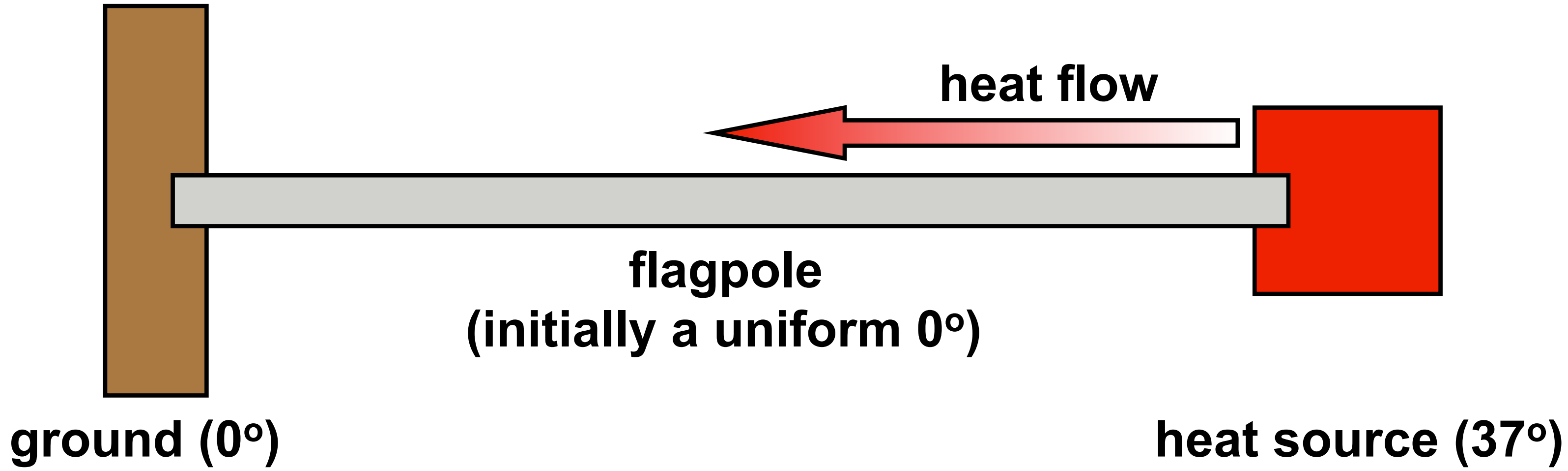
Example: Heat Flow

- This example is adapted from one in *High Performance Computing*, Kevin Dowd & Charles Severance, O' Reilly & Associates, 1998.

- Heat flow is a classic problem in scalable parallel processing
  - in a single dimension:
    - a rod at a constant temperature
    - one end of the rod is exposed to a heat source
    - simulate the flow of heat from one end to the other and determine its steady-state
  - in two dimensions:
    - a metal sheet at a constant temperature
    - heat source(s) placed beneath the sheet
    - simulate flow of heat out from heat sources
  - etc.

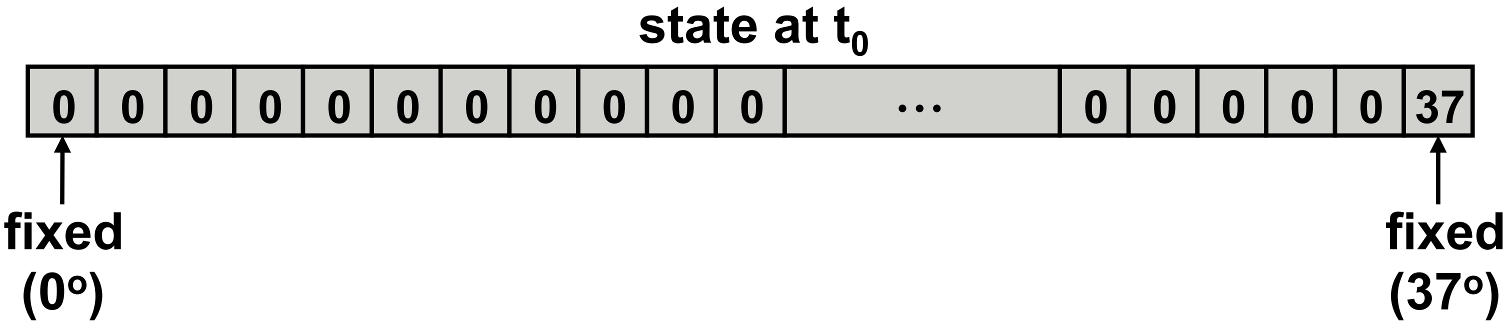
Example: Heat Flow (cont.)

- In our example, we will consider the following set-up:
  - we wish to find the steady state of heat distribution in a flagpole
  - assume that over night the flagpole and the ground achieved a uniform temperature of 0 degrees Celsius (we will also ignore any ambient air effects)
  - first thing in the morning, the top of the flag pole is embedded in an object which has a temperature of 37 degrees Celsius



Example: Heat Flow (cont.)

- Problem set-up
  - note: data parallelism
  - we’ll treat the system in a discrete fashion
    - divide pole into discrete segments with uniform temperature throughout
      - fix temperatures of leftmost/rightmost cell to 0 and 37 respectively
    - divide time into discrete intervals at which to compute temperature
  - compute current temperature as mean of surrounding temperatures



state at t_0

0	0	0	0	0	0	0	0	0	0	0	…	0	0	0	0	0	37

fixed (0°)

fixed (37°)

Heat Flow Design

- Design:
  - divide pole into N segments, each given to one of N processors
    - can divide it up more finely and give multiple segments to each processor if desired
  - each processors computes update for its segment(s) at each time step:
    - $T_s(t) = T_{s-1}(t-1) + T_{s+1}(t-1) / 2$
  - to compute T_s a processor needs T_{s-1} and T_{s+1}
    - have an extra segment at each end fixed to the desired temperature
      - $T_0 = 0$, $T_{n+1} = 37$ (remaining T_s, s = 1..N are the segments of the pole)
    - only process T_s computation for $0 < s <= n$

Heat Flow Design (cont.)

- Issues
  - consider the sequential procedure:

```
for (i = 1; i <= n; i++)
    segment[i]=(segment[i-1]+segment[i+1])/2;
```

  - note that segment[i-1] will actually be the temperature at the *next* time step rather than this one (as segment[i] was computed first)

Heat Flow Design (cont.)

- Solution?
  - red-black arrays
  - alternate between arrays of segments using one as the source of values from the previous time step while writing computed values to the other
- How do we account for this in our parallel program (or is it even relevant)?

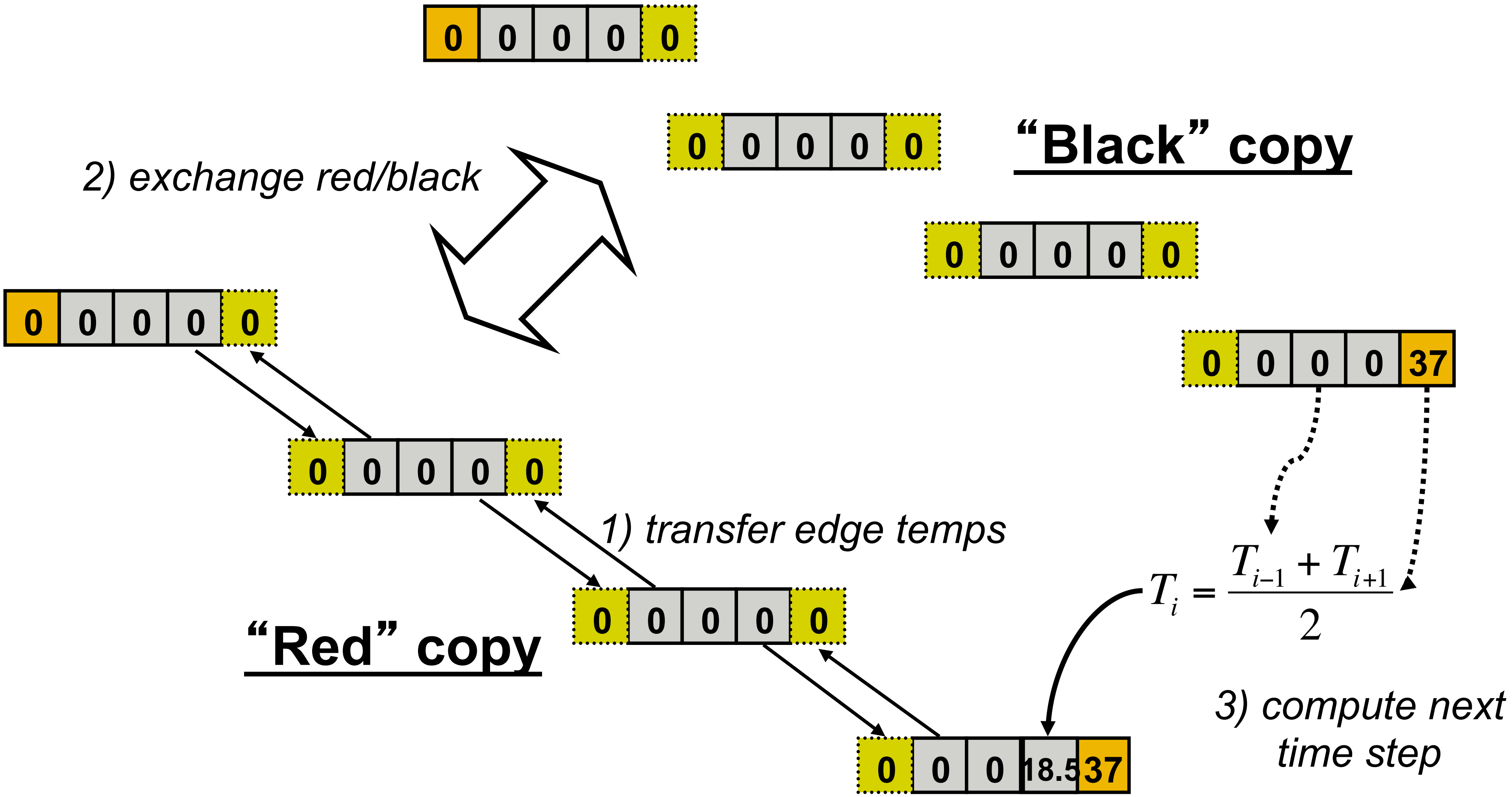
Heat Flow Design (cont.)

- Issues
  - consider communication needs
    - only adjacent segments on different processors need to communicate their values
    - note that there is communication flowing both ways
    - must have this value before computing the temperature at the current time step for at least the adjacent segment

Heat Flow Design (cont.)

- Solution?
  - have the array of segments in each process be two larger than necessary (one on each end --- array runs *0..M+1*)
  - receive the adjacent value we need into this location and local loop on only values *1..M*
  - is order of communication important?
    - adopt some convention, e.g. receive from left, send to right, receive from right, send to left
    - is this safe? are there other options?

Heat Flow Implementation



2) exchange red/black

“Black” copy

1) transfer edge temps

“Red” copy

$$T_i = \frac{T_{i-1} + T_{i+1}}{2}$$

3) compute next time step

Example (heatflow.c et al.)

- Things to note:
  - order of sends/receives (what other options are there?)
  - red/black arrays are references so swapping is fast
    - alternative is to explicitly code the two sequential iterations one reading black->red, the next red->black
    - copying the array contents is a bad idea (why?)

Example (heatflow.c et al.)

- Things to note (cont.):
  - data distribution
    - this method spreads segment count as evenly as possible
    - the alternative is to have the first N-1 processors have equal sized segments, while the last processor handles whatever is left over
      - this option is better if you are going to use gather/scatter
  - only minimal information is broadcast from P_0
    - there is no need to broadcast the entire initial pole state here (why?)

Food For Thought - 2-D Heat Flow

- How would you model a 2-D heat flow problem
  - e.g. a uniform temperature metal plate with a heat source placed under it in the middle (or multiple heat sources at different locations)

- Same basic idea
  - block the data and set up communication patterns
  - at each iteration edge processors (around entire plate) communicate with adjacent processors
  - for each time step compute temp at a given point as average of the 4 or 8 points surrounding it
  - same issues

- What about 3-D heat flow?

Derived Data Types

- Recall we always provide a datatype argument to MPI functions
  - how can we define our own (derived) data types?
  - note that the built-in MPI data types are actually variables, not types

- MPI provides functions to create new instances of `MPI_Datatype` variables for data types we define ourselves
  - MPI_Datatype structures define the structure of data to be sent, it does not encapsulate that data (see example)

Derived Data Types (cont.)

- Building derived data types is relatively expensive
  - if we are going to bother with the overhead there must be some expected benefit
  - we should be expecting to use a derived data type extensively to warrant creating one

- Derived types in MPI are internally represented as sequences of *`(datatype, offset)`* pairs
  - *`datatype`* is a known data type
  - *`offset`* is the displacement in bytes from the start of the message where the value is located
  - the sequence of data types is the *type signature*

Derived Data Types (cont.)

- This is an important concept in understanding how we can define our own derived types
  - e.g. an array of 5 doubles might have the following type signature:

    ```
    {(MPI_DOUBLE, 0), (MPI_DOUBLE, 8),
      (MPI_DOUBLE, 16),   (MPI_DOUBLE, 24),
      (MPI_DOUBLE, 32) }
    ```

  - MPI allows us to define our own types by constructing type signatures (internally) matching the definitions we provide

MPI Type Equivalence

- Types are considered compatible if the complete sequence of types (i.e. the type signature) in the derived data type in the receiver matches the initial sequence of types from the sender
  - i.e. sender’s type signature can be longer than the receivers as long as the initial sequences matches
  - displacements are not considered in type matching

- Example: `compatibility.c`
  - modification of indexed type example (to follow)
  - the main diagonal of a matrix is sent, and received into the reverse diagonal of a matrix
  - can play a similar game with columns sent and received into rows, etc.

Committing Data Types: *MPI_Type_commit*

- Before we can use a derived data type we must “commit” the type to MPI
  - allows the library to optimize representation of the type for communication (where possible)
  - recall: type signature

```
int MPI_Type_commit
(
    MPI_Datatype *new
);
```

- `new`
  - an initialized derived data type to be committed

Contiguous Type: *MPI_Type_contiguous*

```
int MPI_Type_contiguous
(
    int count,
    MPI_Datatype type,
    MPI_Datatype *new
);
```

- Sending contiguous data
  - note that in C this is typically unnecessary
    - arrays are allocated contiguously anyway
  - type is provided for portability

- `count`
  - number of contiguous elements in the derived type

- `type`
  - an existing data type identifying the elements in the derived type

- `new`
  - a data type object to be initialized by this call

Contiguous Type Example

```
...
    double vals[SIZE];          /* contiguous array of doubles */
    MPI_Datatype dbl_array;
    ...

    MPI_Type_contiguous(SIZE, MPI_DOUBLE, &dbl_array);
    MPI_Type_commit(&dbl_array);

    if (rank == 0)
    {
        /*
         * send 1 instance of the dbl_array type (vs. 10 of MPI_DOUBLE)
         */
        MPI_Send(vals,1,dbl_array,1,0,MPI_COMM_WORLD);
    }
    else
    {
        MPI_Recv(vals, 1, dbl_array, 0, 0, MPI_COMM_WORLD, &status);
    }
...
```

Vector Type: *MPI_Type_vector*

```
int MPI_Type_vector
(
    int count,
    int length,
    int stride,
    MPI_Datatype type,
    MPI_Datatype *new
);
```

- Sending non-adjacent data stored in contiguous memory with constant stride

- `length`
  - number of entries in each element (i.e. can be an array)

- `stride`
  - number of elements of *`type`* between elements of *`new`*

- e.g. C stores matrices in row-major order --- the values in a column of the matrix are row-length elements apart

Vector Type Example

```
...
    double vals[SIZE][SIZE];       /* contiguous array of doubles */
    MPI_Datatype dbl_column;
    ...
    MPI_Type_vector(SIZE, 1, SIZE, MPI_DOUBLE, &dbl_column);
    MPI_Type_commit(&dbl_column);

    if (rank == 0)
    {
        /* note we can send any column of a SIZExSIZE array of
         * doubles in this same way */
        MPI_Send(&vals[0][0],1,dbl_column,1,0,MPI_COMM_WORLD);
    }
    else
    {
        /* we'll receive the column sent into a different
         * column of the array in this process */
        MPI_Recv(&vals[0][3],1,dbl_column,0,0,MPI_COMM_WORLD, &status);
    }
...
```

Indexed Type: *MPI_Type_index*

```
int MPI_Type_index
(
    int count,
    int lengths[],
    int offsets[],
    MPI_Datatype type,
    MPI_Datatype *new
);
```

- Sending non-adjacent data stored in contiguous memory with variable stride

- `lengths`
  - `length[i]` gives number of elements in the ith entry of the new type

- `offsets`
  - `offsets[i]` gives number elements of *`type`* offset from the beginning of type for the ith entry of the new type

- e.g. main diagonal of a matrix (since lengths can be variable we can send sub-matrices this way as well)

Indexed Type Example

```
...
    double vals[SIZE][SIZE];  /* contiguous array of doubles */
    MPI_Datatype dbl_diag;    /* main diagonal doubles from 2-D array */
    int lengths[SIZE];        /* entry lengths */
    int offsets[SIZE];        /* entry offsets */
    ...

    for (i = 0; i < SIZE; i++)
    {
        lengths[i] = 1;
        offsets[i] = (SIZE+1) * i;
    }
    MPI_Type_indexed(SIZE, lengths, offsets, MPI_DOUBLE, &dbl_diag);
    MPI_Type_commit(&dbl_diag);

    if (rank == 0)
        MPI_Send(vals[0],1,dbl_diag,1,0,MPI_COMM_WORLD);
    else
        MPI_Recv(vals[0], 1, dbl_diag, 0, 0, MPI_COMM_WORLD, &status);
...
```

Structure Type: *MPI_Type_struct*

```
int MPI_Type_struct
(
    int count,
    int lengths[],
    MPI_Aint offsets[],
    MPI_Datatype types[],
    MPI_Datatype *new
);
```

- MPI allows us to combine arbitrary types into a structure type
  - i.e. create arbitrary type signatures

- `offsets`
  - offsets[i] gives offset from the beginning of the type for the ith component of the new type
    - note that these can be completely arbitrarily located in memory, all offsets are simply relative to the beginning of the type (i.e. the buffer provided to send/recv)
  - note: `MPI_Aint` is a MPI defined type for addresses (allows for addresses too large to represent with an integer)

Structure Type (cont.): *MPI_Address*

- Offset values are expressed as addresses rather than counts
  - consecutive values are not necessarily the same type, and may not occur contiguously in storage (structure types are completely arbitrary)
  - in C we could just use the address-of operator (&); this function ensures portability

```
int MPI_Address
(
    void *ref,
    MPI_Aint *address
);
```

- `ref`
  - reference to storage containing value
- `address`
  - initialized with address of provided reference upon return

Structure Type Example

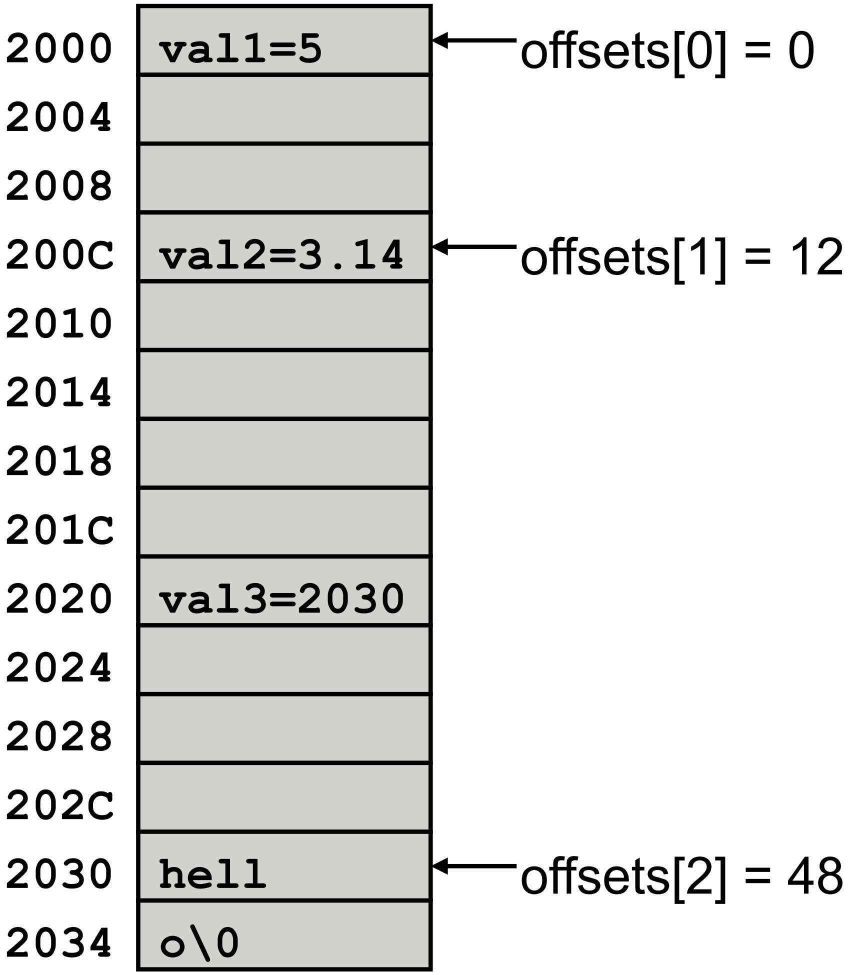
```
...
    int val1 = 0; double val2 = 0.0; char val3[STR_LEN];

    MPI_Datatype int_dbl_str;   /* type containing int/double/string */
    int lengths[3];             /* entry lengths */
    MPI_Aint offsets[3];        /* entry offsets */
    MPI_Datatype types[3];      /* entry types */
    MPI_Aint base, off;
    ...

    lengths[0] = lengths[1] = 1;
    lengths[2] = STR_LEN;

    /* all addresses relative to val1 (&val1 is the “buffer” on send) */
    MPI_Address(&val1, &base);
    offsets[0] = 0;
    MPI_Address(&val2, &off);
    offsets[1] = off - base;
    MPI_Address(&val3, &off);
    offsets[2] = off - base;
```

Structure Type Example (cont.)

```
    types[0] = MPI_INT;
    types[1] = MPI_DOUBLE;
    types[2] = MPI_CHAR;

    MPI_Type_struct(3, lengths, offsets, types, &int_dbl_str);
    MPI_Type_commit(&int_dbl_str);

    if (rank == 0)
    {
        val1 = 5;
        val2 = 3.14;
        strcpy(val3,"hello");

        MPI_Send(&val1, 1, int_dbl_str, 1, 0, MPI_COMM_WORLD);
    }
    else
    {
        MPI_Recv(&val1, 1, int_dbl_str, 0, 0, MPI_COMM_WORLD, &status);
    }
...
```

Notes on Derived Types

- Set-up is relatively expensive
  - only use when data being transmitted forms a regular part of communication
- Reasonably efficient once set-up is complete
  - permit more natural data representation in code
- CAUTION re: Structure types
  - offsets in structure types reflect actual data layout in memory
  - only reference addresses that will exist at the time of transmission



2000	val1=5	offsets[0] = 0
2004		
2008		
200C	val2=3.14	offsets[1] = 12
2010		
2014		
2018		
201C		
2020	val3=2030	
2024		
2028		
202C		
2030	hell	offsets[2] = 48
2034	o\0	

Packing Data: *MPI_Pack*

```
int MPI_Pack
(
    void *data,
    int count,
    MPI_Datatype datatype,
    void *buffer,
    int size,
    int *offset,
    MPI_Comm comm
);
```

- Packing allows us to explicitly store non-contiguous data in contiguous memory locations

- `data`
  - reference to data to be packed
- `buffer`
  - buffer into which we are packing the data
- `offset`
  - offset into buffer where data should be packed
  - when the function returns *`offset`* has been changed to refer to the first location in *`buffer`* after the data that was just packed

Unpacking Data : *MPI_Unpack*

```
int MPI_Unpack
(
    void *buffer,
    int size,
    int *offset,
    void *data,
    int count,
    MPI_Datatype datatype,
    MPI_Comm comm
);
```

- Unpacking data is exactly the opposite process

- `buffer`
  - buffer from which we are unpacking the data
- `offset`
  - references the starting position of the data to be unpacked within buffer
  - after return *`offset`* is changed to reference the first position in `buffer` after the data that was just unpacked
- `data`
  - reference to location into which data is to be unpacked

Notes on Packing Data

- The type of contiguous data sent in this manner is MPI_PACKED

- This is completely explicit data packing/unpacking
  - there is no part of this where you define a type that can be reused; you are simply jamming data into a contiguous representation for transmission

- There is significant ongoing overhead with packing data in this manner
  - if it must be done repetitively you are probably better off with a contiguous or structured derived type
  - if you are transmitting highly variable messages, it becomes an empirical question whether you'll see better performance from packing data as compared to numerous individual sends

Packing Data Example

```
...
    int val1 = 0;
    double val2 = 0.0;
    char val3[STR_LEN];
    int offset = 0;
    char buf[BUF_SIZE];
    ...

    if (rank == 0)
    {
        val1 = 5; val2 = 3.14; strcpy(val3,"hello");

        /*
         * pack data to be sent into buffer
         */
        MPI_Pack(&val1,1,MPI_INT,buf,BUF_SIZE,&offset,MPI_COMM_WORLD);
        MPI_Pack(&val2, 1, MPI_DOUBLE,
                buf, BUF_SIZE, &offset, MPI_COMM_WORLD);
        MPI_Pack(&val3, STR_LEN, MPI_CHAR,
                buf, BUF_SIZE, &offset, MPI_COMM_WORLD)
```

Packing Data Example (cont.)

```
        /*
         * send the packed data buffer
         */
        MPI_Send(buf, BUF_SIZE, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
    }
    else
    {
        MPI_Recv(buf,BUF_SIZE,MPI_PACKED,0,0,MPI_COMM_WORLD,&status);

        /*
         * unpack received data from packed data buffer
         */
        MPI_Unpack(buf, BUF_SIZE, &offset,
                &val1, 1, MPI_INT,  MPI_COMM_WORLD);
        MPI_Unpack(buf, BUF_SIZE, &offset,
                &val2, 1, MPI_DOUBLE, MPI_COMM_WORLD);
        MPI_Unpack(buf, BUF_SIZE, &offset,
                &val3, STR_LEN, MPI_CHAR, MPI_COMM_WORLD);
...
```

Data Summary

- If data to be sent is already stored in contiguous memory
  - use standard methods for transmission (or contiguous type depending on language)
  - use vector derived type if values are not adjacent, but are uniform in relative location
  - use indexed derived type if values are non adjacent and stride between values not a constant

- If data is not stored contiguously
  - use structure derived type if arbitrary storage locations are used repeatedly
  - use pack/unpack only where communication needs are highly non-uniform and there is measurable benefit from doing it

Where do you go from here?

- Other communication options
  - buffered
  - non-blocking pipelines
- Communicators
- Topologies
- MPI-2
  - one-sided communication

Exercise 1: safe MPI

1) The `phello.c` file in `~dbm/public/exercises/intro` is a copy of the one used in the earlier example

2) Modify this program so that each process *i* sends it's text string to process *(i + 1) % n*, where *n* is the number of processes
   - before a process sends its message it should output what it is doing making reference to the process ranks involved (e.g. "Process 1 sending to process 2")
   - similarly, after a process receives a message it should output what it did and what the message was (e.g "Process 2 received 'Hello, world! from process 1' from process 1").
   - note that the receiving process should be making use of the message it received in the output

3) Draw a diagram illustrating the order of events implied by your code

Exercise 1: safe MPI (cont.)

- Compile and submit this job to
  - 2 processes
  - 4 processes
  - 8 processes
- Answer the following questions:
  - *is the order in which processes perform their send/receives significant?*
  - *what would happen if you reversed them?*
  - *what happens if you run this on a single processor?*
  - *are there any problems with the order of events in you code?*

Exercise 2: parallel communication

1) The parallel inner product code presented in these notes appears in file `piprod.c` in `~dbm/pub/exercises/intro`
   - you can compile with -DTRACE to activate the code that traces the sends and receives to/from all processes if you'd like to see it
   - note that as written, all processes read two vectors from a file provided as a command-line argument

2) Modify this program to accommodate the following:
   - an arbitrary number of processors can be involved (not necessarily a power of 2)
   - vectors can be of arbitrary size (assume vector size can be much larger than the number of processors)

Exercise 2: parallel communication (cont.)

3) Note that it is not necessary to strictly do the recursive doubling approach that we used for the example
   - think about how you are going to distribute the work and the data and ensure that process 0 outputs the result

- Answer the following questions:
  - *how did you choose to parallelize the work and data?*
  - *what sort of speed-up would you expect from the approach you have taken?*
  - *what would you have to take into account if you were going to distribute the data from process 0 (rather than have all processes read it)?*

Exercise 3: non-blocking communication

1) The file `nonblocking.c` in `~dbm/pub/exercises/communication` is code that implements both standard and non-blocking sends and receives between two processes
   - use “-n” command-line option to enable use of non-blocking calls
   - note that as configured, several hundred messages of exactly 1MB are sent between processes in this test, and no work is done between them

2) Add some computational work in the work() function
   - this function is called once per send/receive between processes, but occurs between the `MPI_Isend`/ `MPI_Irecv` and the `MPI_Wait` calls
   - there are several elements of the work you can adjust
     - size of messages (adjusts time spent in communication)
     - number of messages (amplifies any timing differences)
     - volume of work done (adjusts time spend in computation)
   - ensure processes run on different nodes when you submit these jobs (why?)

Exercise 3: non-blocking communication (cont.)

- Answer the following questions:
  - *if the interconnect does not have a communication co-processor, what is your expectation of run time for the standard/non-blocking cases?*
  - *what sort of speed-up do you observe when the message size is small vs. large? what about when the amount of computation is small vs. large?*
  - *what problem characteristics would need to be present for you to take advantage of non-blocking communication?*
  - *is it possible for non-blocking communication to eliminate issues of communication overhead under certain circumstance? is this important?*

Exercise 4: 2-D Heatflow

1) Using the one-dimensional heat flow implementation provided in `~dbm/pub/exercises/heatflow` as a guide to design issues, implement a two-dimensional parallel version of the heat flow simulation (alternatively, implement a parallel cellular automata such as Conway's Game of Life).

2) Note that there are options with respect to distributing the data
   - blocking it into sub-grids and assigning those to the running processes
   - break it up by groups of complete rows (or columns) and assign those to the running processes

3) Answer the following questions:
   - *which of the provided options makes more sense to you outside of implementing it in code?*
   - *which do you think is more effective in terms of a MPI parallel implementation?*