MPI, continued

Outline

- Finish MPI discussion
  - Review blocking and non-blocking communication
  - Introduce one-sided communication
- Sources for this lecture:
  - http://mpi.deino.net/mpi_functions/

Today's MPI Focus – Communication Primitives

- Collective communication
  - Reductions, Broadcast, Scatter, Gather
- Blocking communication
  - Overhead
  - Deadlock?
- Non-blocking
- One-sided communication

Quick MPI Review

- Six most common MPI Commands (aka, Six Command MPI)
  - **MPI_Init**
  - **MPI_Finalize**
  - **MPI_Comm_size**
  - **MPI_Comm_rank**
  - **MPI_Send**
  - **MPI_Recv**
- Send and Receive refer to "point-to-point" communication
- Last time we also showed collective communication
  - **Reduce**

Deadlock?

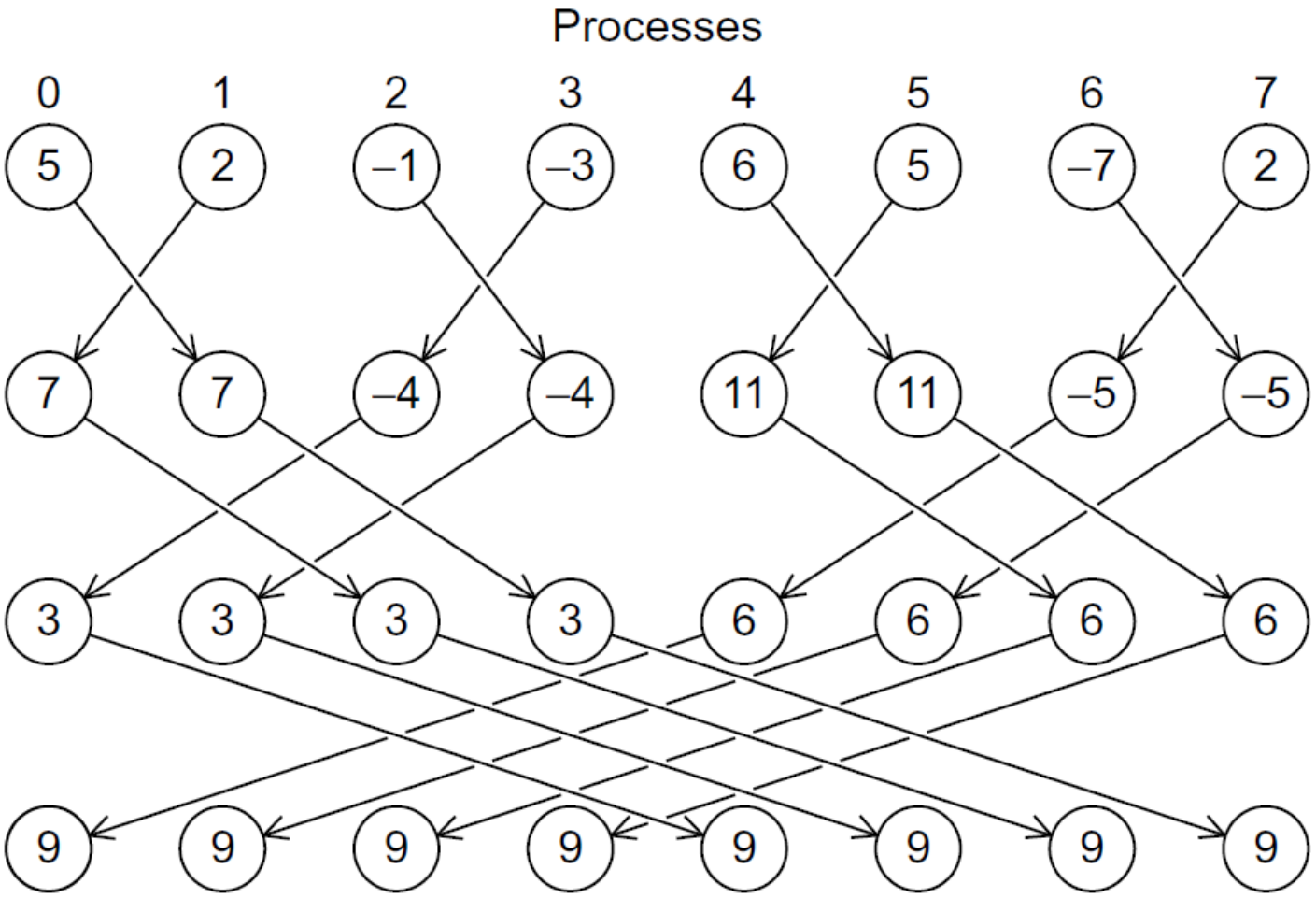
```
int a[10], b[10], myrank;
MPI_Status status; ...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD); }

else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

Deadlock?

Consider the following piece of code:

```
int a[10], b[10], npes, myrank;
MPI_Status status; ...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
  MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
  MPI_COMM_WORLD); ...
```

Global Sum using Butterfly



Example: MPI Code for Butterfly Allreduce

- This is a class time exercise!

MPI_Allreduce

Combines values from all processes and distributes the result back to all processes

Synopsis

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

Input Parameters

sendbuf
starting address of send buffer (choice)

count
number of elements in send buffer (integer)

datatype
data type of elements of send buffer (handle)

op
operation (handle)

comm
communicator (handle)

Output Parameters

recvbuf
starting address of receive buffer (choice)

Global Sum using Allreduce

- Write the same program!

Matrix-Vector Multiplication

a_{00}	a_{01}	$\cdots$	$a_{0,n-1}$
a_{10}	a_{11}	$\cdots$	$a_{1,n-1}$
$\vdots$	$\vdots$		$\vdots$
a_{i0}	a_{i1}	$\cdots$	$a_{i,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$

x_0
x_1
$\vdots$
x_{n-1}

$=$

y_0
y_1
$\vdots$
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
$\vdots$
y_{m-1}

Serial Algorithm

```
 1  void Mat_vect_mult(
 2          double  A[]  /* in  */,
 3          double  x[]  /* in  */,
 4          double  y[]  /* out */,
 5          int     m    /* in  */,
 6          int     n    /* in  */) {
 7      int i, j;
 8
 9      for (i = 0; i < m; i++) {
10          y[i] = 0.0;
11          for (j = 0; j < n; j++)
12              y[i] += A[i*n+j]*x[j];
13      }
14  }   /* Mat_vect_mult */
```

MPI Version

```
1  void Mat_vect_mult(
2        double    local_A[]  /* in  */,
3        double    local_x[]  /* in  */,
4        double    local_y[]  /* out */,
5        int       local_m    /* in  */,
6        int       n          /* in  */,
7        int       local_n    /* in  */,
8        MPI_Comm  comm       /* in  */) {
9     double* x;
10    int local_i, j;
11    int local_ok = 1;
12
13    x = malloc(n*sizeof(double));
14    MPI_Allgather(local_x, local_n, MPI_DOUBLE,
15          x, local_n, MPI_DOUBLE, comm);
16
17    for (local_i = 0; local_i < local_m; local_i++) {
18       local_y[local_i] = 0.0;
19       for (j = 0; j < n; j++)
20          local_y[local_i] += local_A[local_i*n+j]*x[j];
21    }
22    free(x);
23 }  /* Mat_vect_mult */
```

Homework

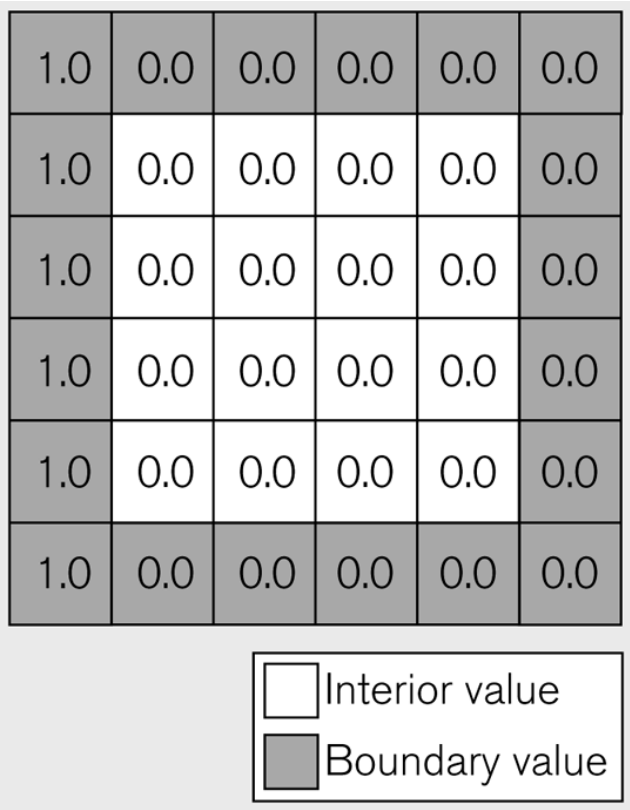
- Include the parallel and serial algorithms of Matrix Mupltiply and compare their timing on a parallel machine. Use a 1000x1000 double matrix and a 1000 element vector.

More difficult p2p example: 2D relaxation

Replaces each interior value by the average of its four nearest neighbors.

Sequential code:

```
for (i=1; i<n-1; i++)
   for (j=1; j<n-1; j++)
      b[i,j] = (a[i-1][j]+a[i][j-1]+
            a[i+1][j]+a[i][j+1])/4.0;
```



MPI code, main loop of 2D SOR computation

```
  1   #define Top      0
  2   #define Left     0
  3   #define Right    (Cols-1)
  4   #define Bottom   (Rows-1)
  5
  6   #define NorthPE(i)        ((i)-Cols)
  7   #define SouthPE(i)        ((i)+Cols)
  8   #define EastPE(i)         ((i)+1)
  9   #define WestPE(i)         ((i)-1)
...
101   do
102   { /*
103     * Send data to four neighbors
104     */
105     if(row !=Top)                                    /* Send North */
106     {
107       MPI_Send(&val[1][1], Width-2, MPI_FLOAT,
108                NorthPE(myID), tag, MPI_COMM_WORLD);
109     }
110
111     if(col !=Right)                                  /* Send East */
112     {
113       for(i=1; i<Height-1; i++)
114       {
115         buffer[i-1]=val[i][Width-2];
116       }
117       MPI_Send(buffer, Height-2, MPI_FLOAT,
118                EastPE(myID), tag, MPI_COMM_WORLD);
119     }
120
121     if(row !=Bottom)                                 /* Send South */
122     {
123       MPI_Send(&val[Height-2][1], Width-2, MPI_FLOAT,
124                SouthPE(myID), tag, MPI_COMM_WORLD);
125     }
126
127     if(col !=Left)                                   /* Send West */
128     {
```

MPI code, main loop of 2D SOR computation, cont.

```
129         for(i=1; i<Height-1; i++)
130         {
131           buffer[i-1]=val[i][1];
132         }
133         MPI_Send(buffer, Height-2, MPI_FLOAT,
134                  WestPE(myID), tag, MPI_COMM_WORLD);
135       }
136
137       /*
138        * Receive messages
139        */
140       if(row !=Top)                          /* Receive from North */
141       {
142         MPI_Recv(&val[0][1], Width-2, MPI_FLOAT,
143                  NorthPE(myID), tag, MPI_COMM_WORLD, &status);
144       }
145
146       if(col !=Right)                        /* Receive from East */
147       {
148         MPI_Recv(&buffer, Height-2, MPI_FLOAT,
149                  EastPE(myID), tag, MPI_COMM_WORLD, &status);
150         for(i=1; i<Height-1; i++)
151         {
152           val[i][Width-1]=buffer[i-1];
153         }
154       }
155
156       if(row !=Bottom)                       /* Receive from South */
157       {
158         MPI_Recv(&val[0][Height-1], Width-2, MPI_FLOAT,
159                  SouthPE(myID), tag, MPI_COMM_WORLD, &status);
160       }
161
162       if(col !=Left)                         /* Receive from West */
163       {
164         MPI_Recv(&buffer, Height-2, MPI_FLOAT,
165                  WestPE(myID), tag, MPI_COMM_WORLD, &status);
166         for(i=1; i<Height-1; i++)
167         {
168           val[i][0]=buffer[i-1];
169         }
170       }
171
172       delta=0.0; /* Calculate average, delta for all points */
173       for(i=1; i<Height-1; i++)
174       {
175         for(j=1; j<Width-1; j++)
176         {
```

MPI code, main loop of 2D SOR computation, cont.

```
177          average=(val[i-1][j]+val[i][j+1]+
178                   val[i+1][j]+val[i][j-1])/4;
179          delta=Max(delta, Abs(average-val[i][j]));
180          new[i][j]=average;
181        }
182      }
183
184      /* Find maximum diff */
185      MPI_Reduce(&delta, &globalDelta, 1, MPI_FLOAT, MPI_MIN,
186                 RootProcess, MPI_COMM_WORLD);
187      Swap(val, new);
188    } while(globalDelta < THRESHOLD);
```

MPI_Scatter()

MPI_Scatter()

```
int MPI_Scatter(                        // Scatter routine
  void          *sendbuffer,            // Address of the data to send
  int           sendcount,              // Number of data elements to send
  MPI_Datatype  sendtype,               // Type of data elements to send
  int           destbuffer,             // Address of buffer to receive data
  int           destcount,              // Number of data elements to receive
  MPI_Datatype  desttype,               // Type of data elements to receive
  int           root,                   // Rank of the root process
  MPI_Comm      *comm                   // An MPI communicator
);
```

Arguments:

- The first three arguments specify the address, size, and type of the data elements to send to each process. These arguments only have meaning for the root process.
- The second three arguments specify the address, size, and type of the data elements for each receiving process. The size and type of the sending data and the receiving data may differ as a means of converting data types.
- The seventh argument specifies the root process that is the source of the data.
- The eighth argument specifies the MPI communicator to use.

Notes:

This routine distributes data from the root process to all other processes, including the root. A more sophisticated version of the routine, `MPI_Scatterv()`, allows the root process to send different amounts of data to the various processes. Details can be found in the MPI standard.

Return value:

An MPI error code.

MPI_Scatter Example

```
1  void Read_vector(
2        double    local_a[]   /* out */,
3        int       local_n     /* in  */,
4        int       n           /* in  */,
5        char      vec_name[]  /* in  */,
6        int       my_rank     /* in  */,
7        MPI_Comm  comm        /* in  */) {
8
9     double* a = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       a = malloc(n*sizeof(double));
14       printf("Enter the vector %s\n", vec_name);
15       for (i = 0; i < n; i++)
16          scanf("%lf", &a[i]);
17       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
18             MPI_DOUBLE, 0, comm);
19       free(a);
20    } else {
21       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
22             MPI_DOUBLE, 0, comm);
23    }
24 }  /* Read_vector */
```

Distribute Data from input using a scatter operation

```
16  length_per_process=length/size;
17  myArray=(int *) malloc(length_per_process*sizeof(int));
18
19  array=(int *) malloc(length*sizeof(int));
20
21  /* Read the data, distribute it among the various processes */
22  if(myID==RootProcess)
23  {
24    if((fp=fopen(*argv, "r"))==NULL)
25    {
26      printf("fopen failed on %s\n", filename);
27      exit(0);
28    }
29    fscanf(fp,"%d", &length);        /* read input size */
30
31    for(i=0; i<length-1; i++)        /* read entire input file */
32    {
33      fscanf(fp,"%d", myArray+i);
34    }
35  }
36
37  MPI_Scatter(Array, length_per_process, MPI_INT,
38              myArray, length_per_process, MPI_INT,
39              RootProcess, MPI_COMM_WORLD);
```

MPI Gather

MPI_Gather

Gathers together values from a group of processes

Synopsis

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

Input Parameters

sendbuf
starting address of send buffer (choice)

sendcount
number of elements in send buffer (integer)

sendtype
data type of send buffer elements (handle)

recvcount
number of elements for any single receive (integer, significant only at root)

recvtype
data type of recv buffer elements (significant only at root) (handle)

root
rank of receiving process (integer)

comm
communicator (handle)

Output Parameters

recvbuf
address of receive buffer (choice, significant only at `root`)

MPI_Gather Example

```
1  void Print_vector(
2        double    local_b[]  /* in */,
3        int       local_n    /* in */,
4        int       n          /* in */,
5        char      title[]    /* in */,
6        int       my_rank    /* in */,
7        MPI_Comm  comm       /* in */) {
8
9     double* b = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       b = malloc(n*sizeof(double));
14       MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
15             MPI_DOUBLE, 0, comm);
16       printf("%s\n", title);
17       for (i = 0; i < n; i++)
18          printf("%f ", b[i]);
19       printf("\n");
20       free(b);
21    } else {
22       MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
23             MPI_DOUBLE, 0, comm);
24    }
25 }  /* Print_vector */
```

Non-Blocking Communication

- The programmer must ensure semantics of the send and receive.
- This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so.
- Non-blocking operations are generally accompanied by a check-status operation.
- When used correctly, these primitives are capable of overlapping communication overheads with useful computations.
- Message passing libraries typically provide both blocking and non-blocking primitives.

Non-Blocking Communication

- To overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations ("I" stands for "Immediate"):

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

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

  These operations return before the operations have been completed.

- Function MPI_Test tests whether or not the non- blocking send or receive operation identified by its request has finished.

  int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

- MPI_Wait waits for the operation to complete.

  int MPI_Wait(MPI_Request *request, MPI_Status *status)

Improving SOR with Non-Blocking Communication

```
if (row != Top) {
   MPI_Isend(&val[1][1], Width-
   2,MPI_FLOAT,NorthPE(myID),tag,MPI_COMM_WORLD, &requests[0]);
}
// analogous for South, East and West
…
if (row!=Top) {
   MPI_Irecv(&val[0][1],Width-2,MPI_FLOAT,NorthPE(myID),
                        tag, MPI_COMM_WORLD, &requests[4]);
}
…
// Perform interior computation on local data
…
//Now wait for Recvs to complete
MPI_Waitall(8,requests,status);

//Then, perform computation on boundaries
```

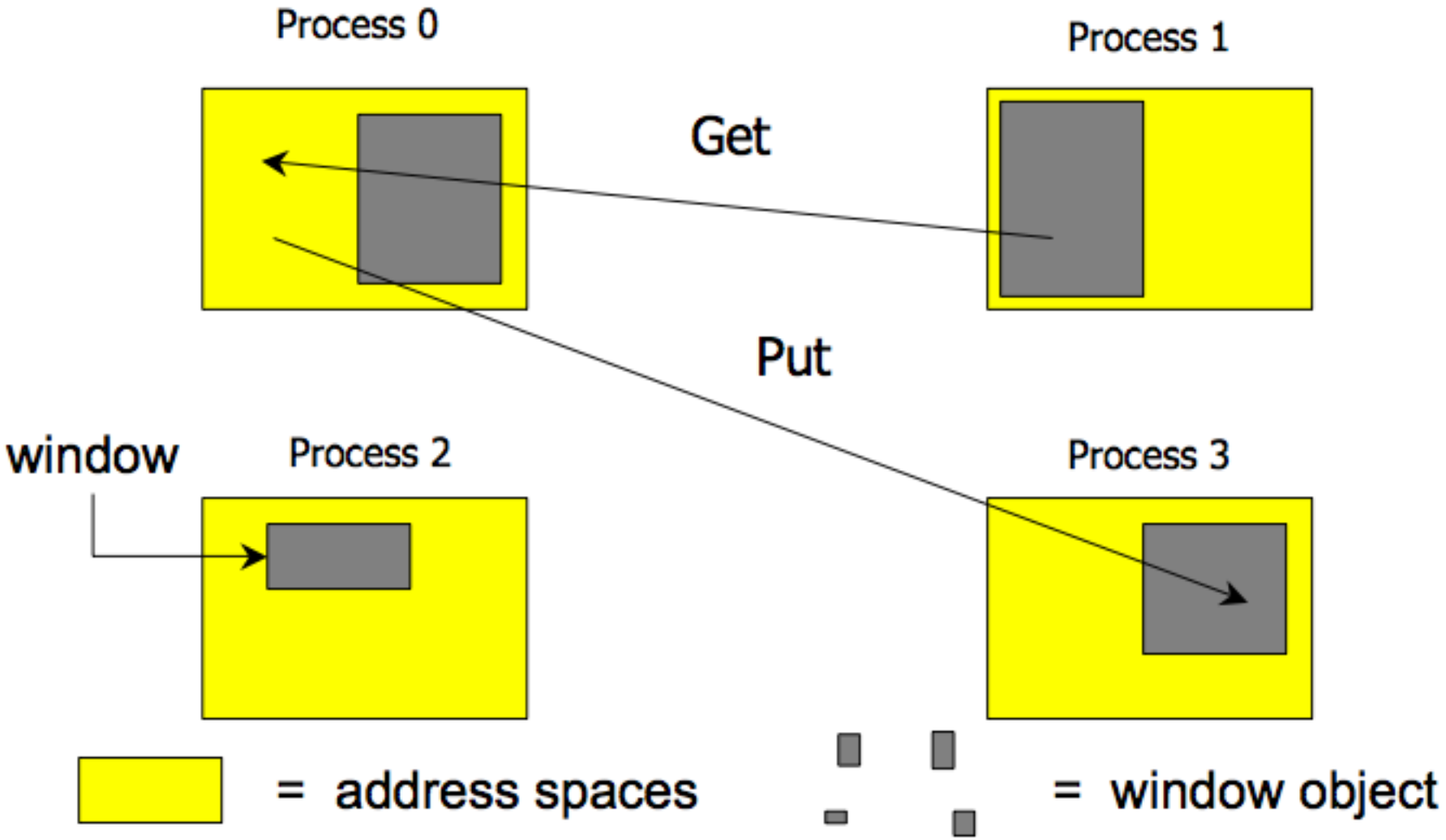
Taking Time in MPI

- MPI_Wtime

```
start = MPI_Wtime();
/* Code to be timed */
. . .
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n"
       my_rank, finish-start);
```

```
double local_start, local_finish, local_elapsed, elapsed;
. . .
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */
. . .
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
   MPI_MAX, 0, comm);

if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", elapsed);
```

One-Sided Communication



MPI Constructs supporting One-Sided Communication (RMA)

- MPI_Win_create exposes local memory to RMA operation by other processes in a communicator
  - Collective operation
  - Creates window object
- MPI_Win_free deallocates window object

- MPI_Put moves data from local memory to remote memory
- MPI_Get retrieves data from remote memory into local memory
- MPI_Accumulate updates remote memory using local values

MPI_Put and MPI_Get

```
int MPI_Put( void *origin_addr, int origin_count,
  MPI_Datatype origin_datatype, int target_rank,
  MPI_Aint target_disp, int target_count,
  MPI_Datatype target_datatype, MPI_Win win);
```

```
int MPI_Get( void *origin_addr, int origin_count,
  MPI_Datatype origin_datatype, int target_rank,
  MPI_Aint target_disp, int target_count,
  MPI_Datatype target_datatype, MPI_Win win);
```

Specify address, count, datatype for origin and target, rank for target and MPI_win for 1-sided communication.

MPI Critique (Snyder)

- Message passing is a very simple model
- Extremely low level; heavy weight
  - Expense comes from λ and lots of local code
  - Communication code is often more than half
  - Tough to make adaptable and flexible
  - Tough to get right and know it
  - Tough to make perform in some (Snyder says most) cases
- Programming model of choice for scalability
- Widespread adoption due to portability, although not completely true in practice