More MPI

Bryan Mills, PhD

Spring 2017

MPI So Far

- Communicators
- Blocking Point-to-Point
  - MPI_Send
  - MPI_Recv
- Collective Communications
  - MPI_Bcast
  - MPI_Barrier
  - MPI_Reduce
  - MPI_Allreduce

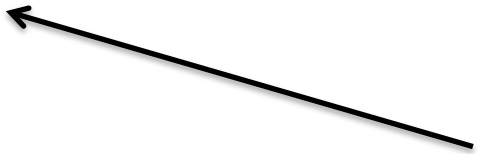
Non-blocking Send

```
int MPI_Isend(
      void*          msg_buf  /* input */,
      int            msg_size /* input */,
      MPI_Datatype   msg_type /* input */,
      int            dest     /* input */,
      int            tag      /* input */,
      MPI_Comm       comm     /* input */,
      MPI_Request    &request /* output */)
```

Identical to MPI_Send but added argument for getting handle to MPI_Request struct.

Non-blocking Receive

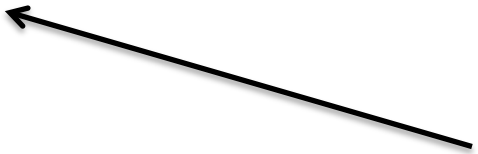
```
int MPI_Irecv(
    void*        msg_buf  /* output */,
    int          msg_size /* input */,
    MPI_Datatype msg_type /* input */,
    int          source   /* input */,
    int          tag      /* input */,
    MPI_Comm     comm     /* input */,
    MPI_Request  &request /* output */)
```



Identical to MPI_Read but replaces MPI_Status argument with MPI_Request struct. Status is now retrieved in the MPI_Wait.

Blocking Wait

```
int MPI_Wait(
   MPI_Request  request /* input */,
   MPI_Status   &status /* output */)
```



Same status that would have been returned from MPI_Recv, not as useful in MPI_Send.

```
int MPI_Waitall(
   int              count  /* input */,
   MPI_Request**    requests /* input */,
   MPI_Status**     &statuses /* output */)
```

Non-blocking Test

```
int MPI_Test(
   MPI_Request  request /* input */,
   int          &flag   /* output */
   MPI_Status   &status /* output */)
```

Flag returns 1 if completed, otherwise 0.

```
MPI_Testall(incount, requests,
   flag, statuses)
MPI_Testsome(incount, requests,
   outcount, indices, statuses)
MPI_Testany(incount, requests,
   index, flag, status)
```

Serial implementation of vector addition

```
void Vector_sum(double x[], double y[], double z[], int n) {
   int i;

   for (i = 0; i < n; i++)
      z[i] = x[i] + y[i];
}  /* Vector_sum */
```

Copyright © 2010, Elsevier Inc. All rights Reserved

Parallel implementation of vector addition

```
void Parallel_vector_sum(
      double  local_x[]  /* in  */,
      double  local_y[]  /* in  */,
      double  local_z[]  /* out */,
      int     local_n    /* in  */) {
   int local_i;

   for (local_i = 0; local_i < local_n; local_i++)
      local_z[local_i] = local_x[local_i] + local_y[local_i];
}  /* Parallel_vector_sum */
```

Copyright © 2010, Elsevier Inc. All rights Reserved

Scatter

- MPI_Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.

```
int MPI_Scatter(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            src_proc    /* in  */,
      MPI_Comm       comm        /* in  */);
```

Copyright © 2010, Elsevier Inc. All rights Reserved

Reading and distributing a vector

```
void Read_vector(
      double    local_a[]   /* out */,
      int       local_n     /* in  */,
      int       n           /* in  */,
      char      vec_name[]  /* in  */,
      int       my_rank     /* in  */,
      MPI_Comm  comm        /* in  */) {

   double* a = NULL;
   int i;

   if (my_rank == 0) {
      a = malloc(n*sizeof(double));
      printf("Enter the vector %s\n", vec_name);
      for (i = 0; i < n; i++)
         scanf("%lf", &a[i]);
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
      free(a);
   } else {
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
   }
}  /* Read_vector */
```

Gather

- Collect all of the components of the vector onto process 0, and then process 0 can process all of the components.

```
int MPI_Gather(
      void*         send_buf_p  /* in  */,
      int           send_count  /* in  */,
      MPI_Datatype  send_type   /* in  */,
      void*         recv_buf_p  /* out */,
      int           recv_count  /* in  */,
      MPI_Datatype  recv_type   /* in  */,
      int           dest_proc   /* in  */,
      MPI_Comm      comm        /* in  */);
```

Copyright © 2010, Elsevier Inc. All rights Reserved

Print a distributed vector (1)

```
void Print_vector(
      double     local_b[]  /* in */,
      int        local_n    /* in */,
      int        n          /* in */,
      char       title[]    /* in */,
      int        my_rank    /* in */,
      MPI_Comm   comm       /* in */) {

   double* b = NULL;
   int i;
```

Copyright © 2010, Elsevier Inc. All rights Reserved

Print a distributed vector (2)

```
   if (my_rank == 0) {
      b = malloc(n*sizeof(double));
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
            0, comm);
      printf("%s\n", title);
      for (i = 0; i < n; i++)
         printf("%f ", b[i]);
      printf("\n");
      free(b);
   } else {
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
            0, comm);
   }
}  /* Print_vector */
```

Copyright © 2010, Elsevier
Inc. All rights Reserved

Allgather

- Concatenates the contents of each process' send_buf_p and stores this in each process' recv_buf_p.
- As usual, recv_count is the amount of data being received from each process.

```
int MPI_Allgather(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      MPI_Comm       comm        /* in  */);
```

Copyright © 2010, Elsevier Inc. All rights Reserved

Matrix-vector multiplication

$A = (a_{ij})$ is an $m \times n$ matrix

$\mathbf{x}$ is a vector with n components

$\mathbf{y} = A\mathbf{x}$ is a vector with m components

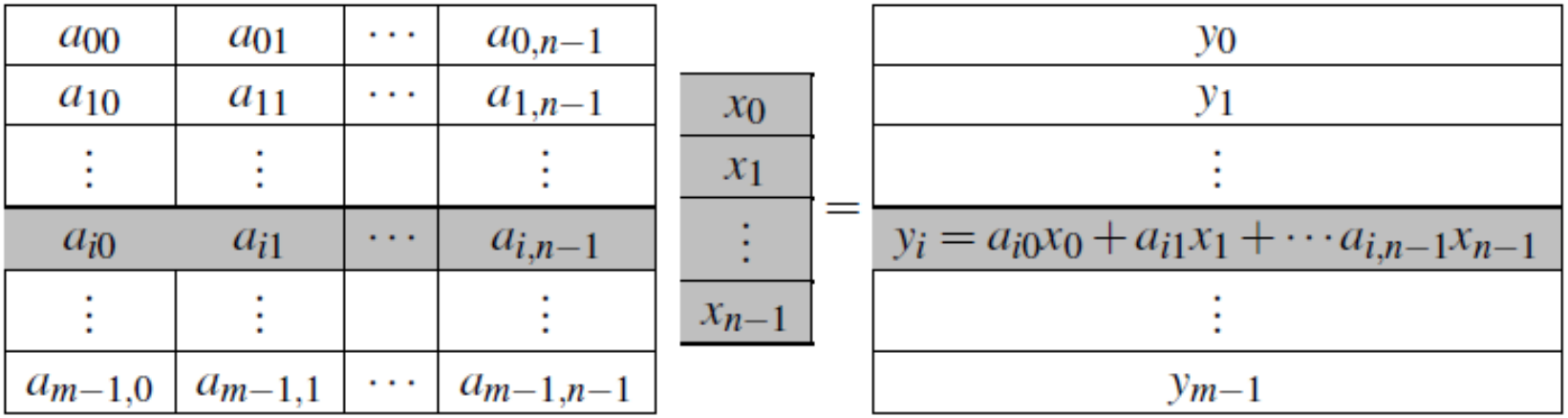
$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}$$

i-th component of y

Dot product of the ith row of A with x.

Copyright © 2010, Elsevier Inc. All rights Reserved

Matrix-vector multiplication



Copyright © 2010, Elsevier Inc. All rights Reserved

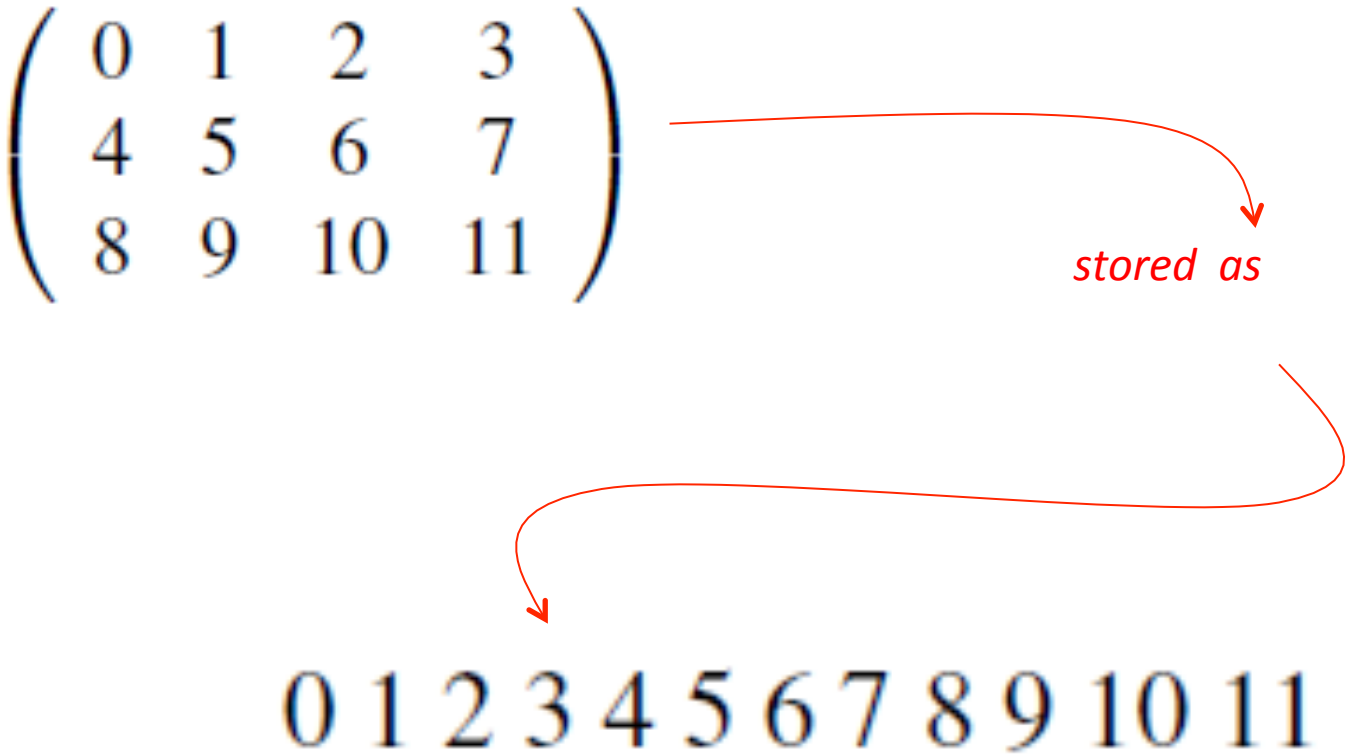
Multiply a matrix by a vector

```
/* For each row of A */
for (i = 0; i < m; i++) {
    /* Form dot product of ith row with x */
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```

Serial pseudo-code

Copyright © 2010, Elsevier Inc. All rights Reserved

C style arrays



Copyright © 2010, Elsevier Inc. All rights Reserved

Serial matrix-vector multiplication

```
void Mat_vect_mult(
      double  A[]  /* in  */,
      double  x[]  /* in  */,
      double  y[]  /* out */,
      int     m    /* in  */,
      int     n    /* in  */) {
   int i, j;

   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i*n+j]*x[j];
   }
}  /* Mat_vect_mult */
```

Copyright © 2010, Elsevier Inc. All rights Reserved

An MPI matrix-vector multiplication function (1)

```
void Mat_vect_mult(
      double     local_A[]  /* in  */,
      double     local_x[]  /* in  */,
      double     local_y[]  /* out */,
      int        local_m    /* in  */,
      int        n          /* in  */,
      int        local_n    /* in  */,
      MPI_Comm   comm       /* in  */) {
   double* x;
   int local_i, j;
   int local_ok = 1;
```

Copyright © 2010, Elsevier Inc. All rights Reserved

An MPI matrix-vector multiplication function (2)

```
   x = malloc(n*sizeof(double));
   MPI_Allgather(local_x, local_n, MPI_DOUBLE,
         x, local_n, MPI_DOUBLE, comm);

   for (local_i = 0; local_i < local_m; local_i++) {
      local_y[local_i] = 0.0;
      for (j = 0; j < n; j++)
         local_y[local_i] += local_A[local_i*n+j]*x[j];
   }
   free(x);
}  /* Mat_vect_mult */
```

Copyright © 2010, Elsevier Inc. All rights Reserved

PERFORMANCE EVALUATION

Copyright © 2010, Elsevier
Inc. All rights Reserved

Elapsed parallel time

- Returns the number of seconds that have elapsed since some time in the past.

```
double MPI_Wtime(void);
```

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

Copyright © 2010, Elsevier Inc. All rights Reserved

Run-times of serial and parallel matrix-vector multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

(Seconds)

Copyright © 2010, Elsevier Inc. All rights Reserved

Speedups of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Copyright © 2010, Elsevier Inc. All rights Reserved

Efficiencies of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Copyright © 2010, Elsevier Inc. All rights Reserved

Scalability

- A program is scalable if the problem size can be increased at a rate so that the efficiency doesn't decrease as the number of processes increase.

- Programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be strongly scalable.

- Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be weakly scalable.

Copyright © 2010, Elsevier
Inc. All rights Reserved

MPI Derived Datatypes

- Allows transmission of more complex data types. In C this is struct.

```
struct Point {
  float x;
  float y;
  int color[3];
};
```

MPI Derived Datatypes

- To create such a type you will need to create a custom MPI data type

```
int MPI_Type_create_struct(
                        int count,
                        int array_of_blocklengths[],
                        MPI_Aint array_of_displacements[],
                        MPI_Datatype array_of_types[],
                        MPI_Datatype *newtype)
```

MPI Derived Datatypes

```
struct Point {
    float x;
    float y;
    int color[3];
};
```

- Count – How many elements in struct
  - Example: 3
- Block lengths – How many elements in elements
  - ex: {1, 1, 3}
- Displacements – Where are these elements in relation to the base pointer value.
  - Ex: {0, 4, 8}
- Data Types – Mapping elements to base MPI types.
  - Ex: {MPI_DOUBLE, MPI_DOUBLE, MPI_INT}

MPI Derived Datatypes

```
struct Point points[N];
MPI_Datatype PointStruct;
MPI_Datatype types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT}
Int blocklens[3] = {1, 1, 3};

MPI_Aint     displacements[3];
MPI_Get_address(points,              displacements);
MPI_Get_address(&points [0].y,       displacements+1);
MPI_Get_address(&points [0].color,   displacements+2);
MPI_Aint     base;
base = displacements[0];
displacements[0] = 0;
displacements[1] = displacements[1] – base;
displacements[2] = displacements[2] – base;

MPI_Type_create_struct(3, blocklens, displacements,
    types, &PointStruct);
MPI_Type_commit(&PointStruct);
```

Derived Datatypes - Optimized

- Once you have created a data type one can smash it together (sometimes). Tools to do that Pack/Unpack and create resized data type.

```
int MPI_Type_create_resized(
  MPI_Datatype oldtype,
  MPI_Aint lb,
  MPI_Aint extent,
  MPI_Datatype *newtype
);
```

Graveyard

Different partitions of a 12-component vector among 3 processes

Process	Components											
	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Copyright © 2010, Elsevier Inc. All rights Reserved

Partitioning options

- Block partitioning
  - Assign blocks of consecutive components to each process.
- Cyclic partitioning
  - Assign components in a round robin fashion.
- Block-cyclic partitioning
  - Use a cyclic distribution of blocks of components.

Copyright © 2010, Elsevier Inc. All rights Reserved