CME213/ME339
Lecture 20

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2012

Example: row-wise matrix-vector product

```
RowMatrixVectorMultiply(int n, double *a, double *b, double *x,
                        MPI_Comm comm) {
/* a stores the rows of the matrix that this process
   is going to compute with.
   b stores only the part of b owned by this process */
int i, j;
int nlocal; /* Number of locally stored rows of A */
double *fb; /* Will store the entire vector b */
int nprocs, myrank;
MPI_Status status;

/* Get information about the communicator */
MPI_Comm_size(comm, &nprocs);
MPI_Comm_rank(comm, &myrank);

/* Allocate the memory used to store the entire b */
fb = (double *)malloc(n*sizeof(double));
nlocal = n/nprocs; // nprocs must divide n
/* Gather entire vector b on each processor using Allgather */
MPI_Allgather(b, nlocal, MPI_DOUBLE, fb, nlocal, MPI_DOUBLE,
              comm);

/* Perform the matrix-vector multiplication involving the
   locally stored submatrix. */
for (i=0; i<nlocal; i++) {
  x[i] = 0.0;
  for (j=0; j<n; j++)
    x[i] += a[i*n+j]*fb[j];
}
/* Done! */
free(fb);
}
```

Example: column-wise matrix-vector product

```c
ColMatrixVectorMultiply(int n, double *a, double *b, double *x,
                        MPI_Comm comm) {
int i, j;
int nlocal;
double *px;
double *fx;
int nprocs, myrank;
MPI_Status status;
/* Get identity and size information from the communicator */
MPI_Comm_size(comm, &nprocs);
MPI_Comm_rank(comm, &myrank);

nlocal = n/nprocs;

/* Allocate memory for arrays storing intermediate results. */
px = (double *)malloc(n*sizeof(double));
fx = (double *)malloc(n*sizeof(double));
/* Compute the partial-dot products that correspond to the
   local columns of A.*/
for (i=0; i<n; i++) {
  px[i] = 0.0;
  for (j=0; j<nlocal; j++)
    px[i] += a[i*nlocal+j]*b[j];
}

/* Sum-up the results by performing an element-wise
   reduction operation. Result is stored at 0. */
MPI_Reduce(px, fx, n, MPI_DOUBLE, MPI_SUM, 0, comm);

/* Redistribute fx in a fashion similar to that of vector b.
   Data is sent from 0 to all other processes. */
MPI_Scatter(fx, nlocal, MPI_DOUBLE, x, nlocal, MPI_DOUBLE, 0,
            comm);
free(px); free(fx);
}
```

Virtual topologies

Many problems are naturally mapped to certain topologies such as grids. This is the case for example for matrices, or for 2D or 3D structured grids.

The two main types of topologies supported by MPI are Cartesian grids and graphs.

MPI topologies are virtual — there may be no relation between the physical structure of the parallel machine and the process topology.

Why?

Convenience: virtual topologies may be useful for applications with specific communication patterns — patterns that match an MPI topology structure.

Communication efficiency: a particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine. For example nodes that are nearby on the grid (East/West/North/South neighbors) will be as close as possible in the network (lowest communication time).

The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation.

MPI functions for virtual topologies

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
                    int *periods, int reorder,
                    MPI_Comm *comm_cart)
```

The topology information is attached to a new communicator `comm_cart` that is created by `MPI_Cart_create`.

The array `periods` is used to specify whether or not the topology has wraparound connections. In particular, if `periods[i]` is non-zero, then the topology has wraparound connections along dimension `i`, otherwise it does not.

The argument `reorder` is used to determine if the processes in the new group (i.e., communicator) are to be reordered or not.

If reorder is false, then the rank of each process in the new group is identical to its rank in the old group.

Otherwise, `MPI_Cart_create` may reorder the processes if that leads to a better embedding of the virtual topology onto the parallel computer.

It will result in an error if the total number of processes specified by `dims` is greater than the number of processes in the `comm_old` communicator.

Other useful functions

```c
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)
int MPI_Cart_coords(MPI_Comm comm_cart, int rank, int maxdims,
                    int *coords)
```

`coords` are the Cartesian coordinates of a process. Its size is the number of dimensions.

```
int MPI_Cart_shift(MPI_Comm comm_cart, int dir, int s_step,
                   int *rank_source, int *rank_dest)
```

`dir`: direction; `s_step`: length shift.

`rank_dest` contains the group rank of the neighboring process in the specified dimension and distance. The rank of the process for which the calling process is the neighboring process in the specified dimension and distance is returned `rank_source`.

Thus, the group ranks returned in rank dest and rank source can be used as parameters for `MPI_Sendrecv()`.

Groups and communicators

This is a more advanced topic. I will only cover it superficially. More information can be found online.

The main point is that collective communications assume that we have identified a certain group of processes that want to communicate.

By default this group contains all the processes.

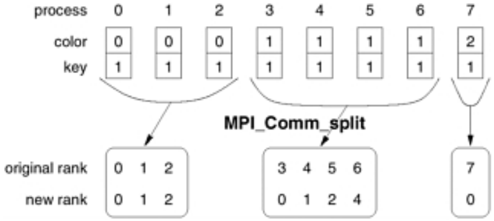
This can be changed using groups. For example we may want to communicate along the rows of a matrix or the columns.

Split

The simplest method to create groups:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
```

Each subgroup contains all processes that have supplied the same value for the `color` parameter. Within each subgroup, the processes are ranked in the order defined by the value of the `key` parameter, with ties broken according to their rank in the old communicator (i.e., `comm`). A new communicator for each subgroup is returned in the `newcomm` parameter.


process
0
1
2
3
4
5
6
7
color
0
0
0
1
1
1
1
2
key
1
1
1
1
1
1
1
1
MPI_Comm_split
original rank
0 1 2
3 4 5 6
7
new rank
0 1 2
0 1 2 4
0

Splitting a Cartesian topology

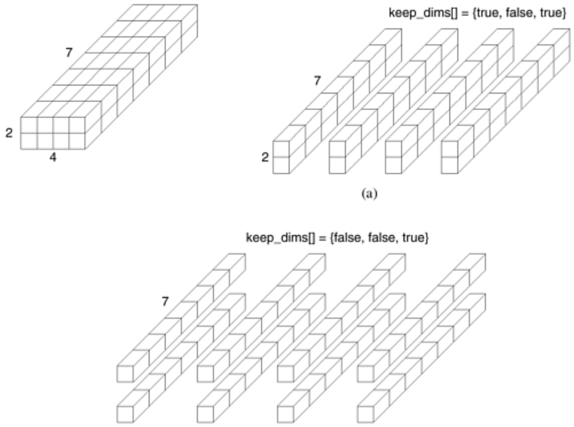
It is very common that one wants to split a Cartesian topology along certain dimensions.

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,
                 MPI_Comm *comm_subcart)
```

`keep_dims`: boolean flag that determines whether that dimension is retained in the new communicators or split, e.g., if false then a split occurs.

Example

The grid has size $2 \times 4 \times 7$.


7
2
4
keep_dims[] = {true, false, true}
7
2
keep_dims[] = {false, false, true}
7

(a)

(b)

Matrix-vector multiplication: 2D

```
MatrixVectorMultiply_2D(int n, double *a, double *b, double *x,
    MPI_Comm comm) {
int ROW=0, COL=1; /* Improve readability */
int i, j, nlocal;
double *px; /* Will store partial dot products */
int nprocs, dims[2], periods[2], keep_dims[2];
int myrank, my2drank, mycoords[2];
int drank, col0rank, coords[2];
MPI_Status status;
MPI_Comm comm_2d, comm_row, comm_col;

/* Get information about the communicator */
MPI_Comm_size(comm, &nprocs); MPI_Comm_rank(comm, &myrank);

/* Compute the size of the square grid.
   We assume that nprocs is a square and that the matrix size
   is a multiple of sqrt(nprocs). */
dims[ROW] = dims[COL] = sqrt(nprocs);
nlocal    = n/dims[ROW];
...
```

```c
/* Allocate memory for the array that will hold
   the partial dot-products */
px = malloc(nlocal*sizeof(double));

/* Set up the Cartesian topology and get the rank &
   coordinates of the process in this topology */
periods[ROW] = periods[COL] = 1;
/* Periods for wrap-around connections */

MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_2d);

/* Get my rank in the new topology */
MPI_Comm_rank(comm_2d, &my2drank);

/* Get my coordinates */
MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);
...
```

```
/* Create the row-based sub-topology */
keep_dims[ROW] = 0;
keep_dims[COL] = 1;
MPI_Cart_sub(comm_2d, keep_dims, &comm_row);

/* Create the column-based sub-topology */
keep_dims[ROW] = 1;
keep_dims[COL] = 0;
MPI_Cart_sub(comm_2d, keep_dims, &comm_col);
...
```

```c
/* Redistribute the b vector. */
/* Step 1. The processors along the 0th column
   send their data to the diagonal processor. */
if (mycoords[COL] == 0 && mycoords[ROW] != 0) {
  /* I'm in the first column */
  coords[ROW] = mycoords[ROW];
  coords[COL] = mycoords[ROW]; // coords of diagonal block
  MPI_Cart_rank(comm_2d, coords, &drank);
  /* Send data to the diagonal block */
  MPI_Send(b, nlocal, MPI_DOUBLE, drank, 1, comm_2d);
}

if (mycoords[ROW] == mycoords[COL] && mycoords[ROW] != 0) {
  /* I am a diagonal block */
  coords[ROW] = mycoords[ROW];
  coords[COL] = 0; // Receiving from column 0
  MPI_Cart_rank(comm_2d, coords, &col0rank);
  MPI_Recv(b, nlocal, MPI_DOUBLE, col0rank, 1, comm_2d,
      &status);
}
...
```

```
/* Step 2. The diagonal processors perform a
           column-wise broadcast */
coords[0] = mycoords[COL];
/* Note the 1D rank in the column sub-topology */
MPI_Cart_rank(comm_col, coords, &drank);
MPI_Bcast(b, nlocal, MPI_DOUBLE, drank, comm_col);

/* Get into the main computational loop */
for (i=0; i<nlocal; i++) {
  px[i] = 0.0;
  for (j=0; j<nlocal; j++)
    px[i] += a[i*nlocal+j]*b[j];
}
...
```

```
/* Perform the sum-reduction along the rows to add up
   the partial dot-products */
coords[0] = 0;
/* Note the 1D rank in the row sub-topology */
MPI_Cart_rank(comm_row, coords, &col0rank);
MPI_Reduce(px, x, nlocal, MPI_DOUBLE, MPI_SUM, col0rank,
           comm_row);

/* Free up communicators */
MPI_Comm_free(&comm_2d);
MPI_Comm_free(&comm_row);
MPI_Comm_free(&comm_col);
free(px);
}
```

Derived data types

We do not have time to cover that topic in detail.

This can be used to communicate data that is not simply contiguous in memory.

The most useful are:

```
int MPI_Type_vector(
        int count, int blocklen, int stride,
        MPI_Datatype old_type, MPI_Datatype *newtype)

int MPI_Type_indexed(
        int count, int blocklens[], int indices[],
        MPI_Datatype old_type, MPI_Datatype *newtype)
```

Short example

```
...
float a[SIZE][SIZE] =
  {1.0, 2.0, 3.0, 4.0,
   5.0, 6.0, 7.0, 8.0,
   9.0, 10.0, 11.0, 12.0,
  13.0, 14.0, 15.0, 16.0};
...
MPI_Datatype columntype;
MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
MPI_Type_commit(&columntype);
...
MPI_Send(&a[0][i], 1, columntype, i, tag, MPI_COMM_WORLD);
...
```

Performance metrics

There are a few key concepts that help understand the running time of parallel applications.

There are several components in the total run time $T_p(n)$:

- Local computations
- Data exchange and communication
- Waiting time because of load imbalance

Cost

The cost $C_p(n)$ of a parallel program is:

$$C_p(n) = p\ T_p(n)$$

This is a measure of the total amount of work done for all processors.

This cost is optimal if

$$C_p(n) = T^\star(n),$$

the optimal runtime of the sequential algorithm.

Speedup

Defined as:

$$S_p(n) = \frac{T^\star(n)}{T_p(n)}$$

This represents the savings obtained from running in parallel.

In some rare cases superlinear effect can be observed, that is $S_p(n) > p$. This is in most cases due to cache effect. With p processes we have effectively access to more cache space than on a single processor, resulting in some cases in better performance.

Efficiency

It makes sense to compare the speed-up against p. Ideally the speed-up scales like p. To better illustrate this dependence, it is common to define the efficiency of a parallel code:

$$E_p(n) = \frac{T^\star(n)}{C_p(n)} = \frac{S_p(n)}{p} = \frac{T^\star(n)}{p T_p(n)}$$

The ideal speed-up corresponds to $E_p = 1$.

Amdahl's law

Although this is a crude model, it can provide a general sense of what speed-up can be achieved. This is a good measure to understand how much of the code one should try to parallelize.

Assume that a fraction f of the code is executed sequentially. Then the speed-up is given by:

$$S_p(n) = \frac{T^\star(n)}{f\,T^\star(n) + ((1-f)/p)\,T^\star(n)} = \frac{1}{f + (1-f)/p} \le \frac{1}{f}$$

For example, if $f = 20\%$, the maximum speed-up that can be achieved is 5, no matter how good our MPI implementation is, or how many processors we can employ.

This sounds a bit depressing. How small can f be?

The good news is that f typically depends on n and goes to 0 as n increases.

This means that to observe a certain speed-up on p processes we may need to increase the problem size n sufficiently.

Increasing n often also allows reducing the overhead in the calculation from communication and idle times.

Gustafson's law

This model assumes that the sequential part has a constant execution time τ_f irrespective of n. Then if the parallel part can be perfectly parallelized:

$$S_p(n) = \frac{T^\star(n)}{\tau_f + (T^\star(n) - \tau_f)/p}$$

Therefore in this model we always have

$$\lim_{n\to\infty} S_p(n) = p$$

Short example: dot product

Goal: calculate a dot product and store the result on one node.

α: time to perform a multiplication or addition (we assume these times are equal for simplicity).
β: time to communicate a float to a neighboring process.

Two steps algorithm:

1. Calculate local dot product: $\sum_j a_j\ b_j$
2. Use a spanning tree: $\ln_2 p$ passes, each with one addition and one communication

Total serial run time: $T^\star = 2\alpha n$
Total parallel run time: $T_p = 2\alpha n/p + (\alpha + \beta) \ln_2 p$

Efficiency and iso-efficiency

Efficiency:

$$E_p(n) = \frac{2\alpha n}{2\alpha n + (\alpha + \beta)\, p \ln_2 p} = \frac{1}{1 + ((\alpha + \beta)/2\alpha)\, p \ln_2 p / n}$$

Efficiency is maintained (iso-efficiency) as long as:

$$p \ln_2 p = \Theta(n) \quad \text{or} \quad p = \Theta(n / \ln_2 n)$$