CSCI 5451: Introduction to Parallel Computing

Lecture 13: MPI Examples

computer science
& engineering

University of Minnesota
Driven to Discover®

Announcements (10/15)

- HW1 → Due later today (Canvas)
- HW 2 released on Monday, Oct 20
- Group Formation due Sunday, Oct 19 (Canvas)

Lecture Overview

- Leftover MPI Functions from (10/13)
  - MPI_Alltoallv
  - MPI_Comm_split & MPI_Cart_sub
- MPI Examples (Whiteboard walkthrough)
  - Row-Wise Matrix-Vector Multiplication
  - Column-Wise Matrix-Vector Multiplication
  - 2-D Matrix-Vector Multiplication
  - Djikstra’s Single Source Shortest Path
  - Sample Sort

Lecture Overview

- **Leftover MPI Functions from (10/13)**
  - **MPI_Alltoallv**
  - MPI_Comm_split & MPI_Cart_sub
- MPI Examples (Whiteboard walkthrough)
  - Row-Wise Matrix-Vector Multiplication
  - Column-Wise Matrix-Vector Multiplication
  - 2-D Matrix-Vector Multiplication
  - Djikstra’s Single Source Shortest Path
  - Sample Sort

MPI_Alltoallv

```
int MPI_Alltoallv(const void *sendbuf,
        const int *sendcounts, const int *sdispls,
        MPI_Datatype sendtype, void *recvbuf,
        const int *recvcounts, const int *rdispls,
        MPI_Datatype recvtype,
        MPI_Comm comm);
```

Each process sends variable-sized data to all others and receives variable-sized data from all others.

- sendbuf: Starting address of send buffer.
- sendcounts: Array specifying number of elements sent to each rank.
- sdispls: Array of offsets in sendbuf.
- sendtype: Type of send data.
- recvbuf: Starting address of receive buffer.
- recvcounts: Array specifying number of elements received from each rank.
- rdispls: Array of offsets in recvbuf.
- recvtype: Type of receive data.
- comm: Communicator handle.

MPI_Alltoallv

```
int MPI_Alltoallv(const void *sendbuf,
        const int *sendcounts, const int *sdispls,
        MPI_Datatype sendtype, void *recvbuf,
        const int *recvcounts, const int *rdispls,
        MPI_Datatype recvtype,
        MPI_Comm comm);
```

Each process sends variable-sized data to all others and receives variable-sized data from all others.

```
int sendbuf[10];
for (int i=0; i<10; i++) sendbuf[i] = rank*10 + i;
int sendcounts[4] = {1,2,3,4};
int sdispls[4]    = {0,1,3,6};

int recvcounts[4] = {rank+1, rank+1, rank+1, rank+1};
int rdispls[4]    = {0, (rank+1), (rank+1)*2, (rank+1)*3};
int recvbuf[16];

MPI_Alltoallv(sendbuf, sendcounts, sdispls, MPI_INT, recvbuf,
                    recvcounts, rdispls, MPI_INT, MPI_COMM_WORLD);

printf("Rank %d received:", rank);
for (int i=0; i<(rank+1)*4; i++)
    printf(" %d", recvbuf[i]);
printf("\n");
```

MPI_Alltoallv

```
int MPI_Alltoallv(const void *sendbuf,
        const int *sendcounts, const int *sdispls,
        MPI_Datatype sendtype, void *recvbuf,
        const int *recvcounts, const int *rdispls,
        MPI_Datatype recvtype,
        MPI_Comm comm);
```

Each process sends variable-sized data to all others and receives variable-sized data from all others.

```
int sendbuf[10];
for (int i=0; i<10; i++) sendbuf[i] = rank*10 + i;
int sendcounts[4] = {1,2,3,4};
int sdispls[4]   = {0,1,3,6};

int recvcounts[4] = {rank+1, rank+1, rank+1, rank+1};
int rdispls[4]    = {0, (rank+1), (rank+1)*2, (rank+1)*3};
int recvbuf[16];

MPI_Alltoallv(sendbuf, sendcounts, sdispls, MPI_INT, recvbuf,
              recvcounts, rdispls, MPI_INT, MPI_COMM_WORLD);

printf("Rank %d received:", rank);
for (int i=0; i<(rank+1)*4; i++)
    printf(" %d", recvbuf[i]);
printf("\n");
```

```
Rank 0 received: 0 10 20 30
Rank 1 received: 1 2 11 12 21 22 31 32
Rank 2 received: 3 4 5 13 14 15 23 24 25 33 34 35
Rank 3 received: 6 7 8 9 16 17 18 19 26 27 28 29 36 37 38 39
```

Lecture Overview

- **Leftover MPI Functions from (10/13)**
  - MPI_Alltoallv
  - **MPI_Comm_split & MPI_Cart_sub**
- MPI Examples (Whiteboard walkthrough)
  - Row-Wise Matrix-Vector Multiplication
  - Column-Wise Matrix-Vector Multiplication
  - 2-D Matrix-Vector Multiplication
  - Djikstra's Single Source Shortest Path
  - Sample Sort

MPI_Comm_split

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

Creates new communicator by splitting an existing communicator into subgroups based on color.

Processes with the same color are grouped together; within each group, ranks are ordered by key

MPI_Comm_split

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

Creates new communicator by splitting an existing communicator into subgroups based on color.

Processes with the same color are grouped together; within each group, ranks are ordered by key

- comm: Existing communicator to be split (e.g., MPI_COMM_WORLD).
- color: Integer identifier for subgroup membership.
  - All processes with the same color form a new communicator.
  - A process can set color = MPI_UNDEFINED to be excluded entirely.
- key: Determines ordering of ranks within the new communicator.
  - Lower keys get lower ranks.
  - Equal keys are ordered according to ordering in previous communicator
- newcomm: Output handle to the newly created communicator.

MPI_Comm_split

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

```
int key = 1;
if (rank <= 2)      color = 0;
else if (rank <= 6) color = 1;
else                color = 2;

MPI_Comm newcomm;
MPI_Comm_split(MPI_COMM_WORLD,
        color, key, &newcomm);
```

MPI_Comm_split

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

```
int key = 1;
if (rank <= 2)      color = 0;
else if (rank <= 6) color = 1;
else                color = 2;

MPI_Comm newcomm;
MPI_Comm_split(MPI_COMM_WORLD,
        color, key, &newcomm);
```

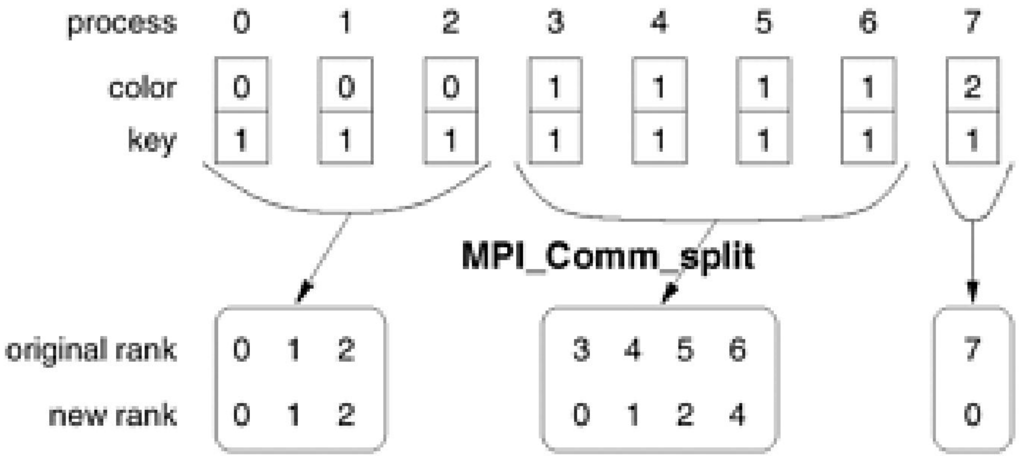
process	0	1	2	3	4	5	6	7
color	0	0	0	1	1	1	1	2
key	1	1	1	1	1	1	1	1

Which processes will be grouped together?

MPI_Comm_split

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

```
int key = 1;
if (rank <= 2)      color = 0;
else if (rank <= 6) color = 1;
else                color = 2;

MPI_Comm newcomm;
MPI_Comm_split(MPI_COMM_WORLD,
        color, key, &newcomm);
```



MPI_Cart_sub

```
int MPI_Cart_sub(MPI_Comm comm,
        const int *remain_dims,
        MPI_Comm *newcomm);
```

Creates a lower-dimensional subcommunicator from an existing Cartesian topology communicator by selecting which dimensions to keep.

MPI_Cart_sub

```
int MPI_Cart_sub(MPI_Comm comm,
        const int *remain_dims,
        MPI_Comm *newcomm);
```

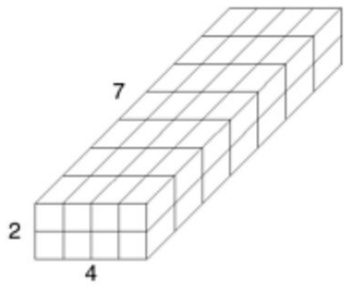
Creates a lower-dimensional subcommunicator from an existing Cartesian topology communicator by selecting which dimensions to keep.

- comm: Input communicator with Cartesian topology (created via MPI_Cart_create).
- remain_dims: Logical array of length equal to the number of dimensions.
  - remain_dims[i] = 1 → keep this dimension.
  - remain_dims[i] = 0 → collapse (drop) this dimension.
- newcomm: Output communicator corresponding to the subgrid of the original topology.

MPI_Cart_sub

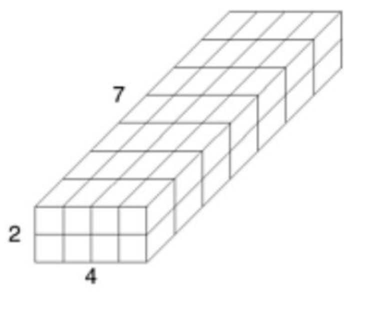
```
int MPI_Cart_sub(MPI_Comm comm,
        const int *remain_dims,
        MPI_Comm *newcomm);
```

```
int dims[3]    = {2, 4, 7};      // grid dimensions
int periods[3] = {0, 0, 0};      // non-periodic in all dimensions
int reorder    = 0;              // keep original ranks

MPI_Comm cart_comm;
MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods,
        reorder, &cart_comm);

int keep_dims[3] = {1, 0, 1};
MPI_Comm sub_comm;
MPI_Cart_sub(cart_comm, keep_dims, &sub_comm);
```



MPI_Cart_sub

```
int MPI_Cart_sub(MPI_Comm comm,
        const int *remain_dims,
        MPI_Comm *newcomm);
```

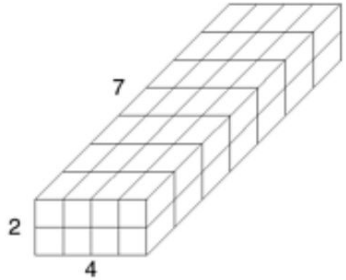


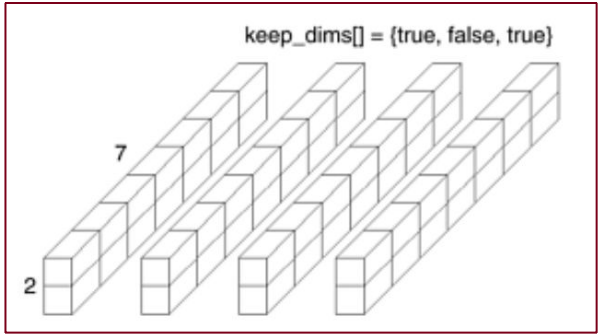

```
int dims[3]    = {2, 4, 7};       // grid dimensions
int periods[3] = {0, 0, 0};       // non-periodic in all dimensions
int reorder    = 0;               // keep original ranks

MPI_Comm cart_comm;
MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods,
        reorder, &cart_comm);

int keep_dims[3] = {1, 0, 1};
MPI_Comm sub_comm;
MPI_Cart_sub(cart_comm, keep_dims, &sub_comm);
```

MPI_Cart_sub

```
int MPI_Cart_sub(MPI_Comm comm,
        const int *remain_dims,
        MPI_Comm *newcomm);
```

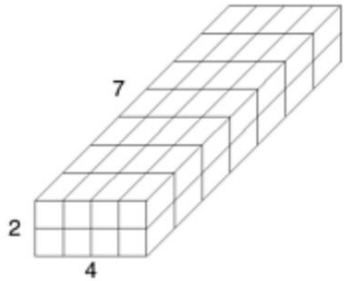
```
int dims[3]    = {2, 4, 7};       // grid dimensions
int periods[3] = {0, 0, 0};       // non-periodic in all dimensions
int reorder    = 0;               // keep original ranks

MPI_Comm cart_comm;
MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods,
        reorder, &cart_comm);

int keep_dims[3] = {1, 0, 1};
MPI_Comm sub_comm;
MPI_Cart_sub(cart_comm, keep_dims, &sub_comm);
```





MPI_Cart_sub

```
int MPI_Cart_sub(MPI_Comm comm,
        const int *remain_dims,
        MPI_Comm *newcomm);
```

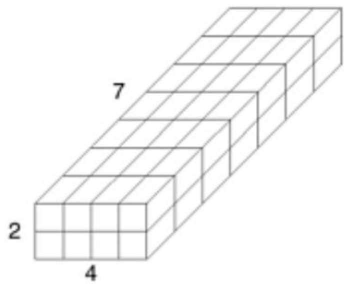


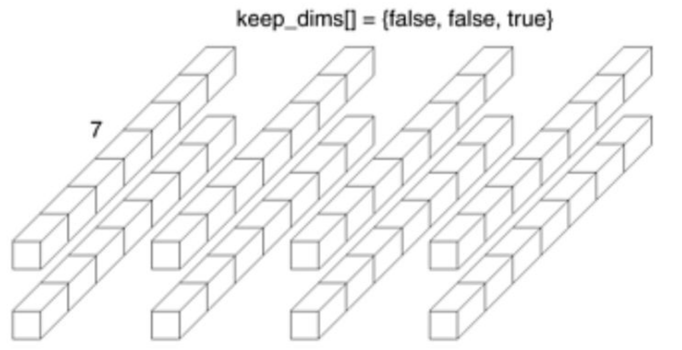

```
int dims[3]    = {2, 4, 7};       // grid dimensions
int periods[3] = {0, 0, 0};       // non-periodic in all dimensions
int reorder    = 0;               // keep original ranks

MPI_Comm cart_comm;
MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods,
        reorder, &cart_comm);

int keep_dims[3] = {0, 0, 1};
MPI_Comm sub_comm;
MPI_Cart_sub(cart_comm, keep_dims, &sub_comm);
```

MPI_Cart_sub

```
int MPI_Cart_sub(MPI_Comm comm,
        const int *remain_dims,
        MPI_Comm *newcomm);
```

```
int dims[3]    = {2, 4, 7};        // grid dimensions
int periods[3] = {0, 0, 0};        // non-periodic in all dimensions
int reorder    = 0;                // keep original ranks

MPI_Comm cart_comm;
MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods,
        reorder, &cart_comm);

int keep_dims[3] = {0, 0, 1};
MPI_Comm sub_comm;
MPI_Cart_sub(cart_comm, keep_dims, &sub_comm);
```





Lecture Overview

- Leftover MPI Functions from (10/13)
  - MPI_Alltoallv
  - MPI_Comm_split & MPI_Cart_sub
- **MPI Examples (Whiteboard walkthrough)**
  - **Row-Wise Matrix-Vector Multiplication**
  - Column-Wise Matrix-Vector Multiplication
  - 2-D Matrix-Vector Multiplication
  - Djikstra’s Single Source Shortest Path
  - Sample Sort

Row-Wise Matrix-Vector Multiplication

```
void RowMatrixVectorMultiply(int n, double *a, double *b, double *x, MPI_Comm comm)
{
    int i, j;
    int nlocal;   /* Number of locally stored rows of A */
    double *fb;   /* Buffer that stores the entire vector b */
    int npes, myrank;
    MPI_Status status;

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

    /* Allocate memory to store the entire vector b */
    fb = (double *)malloc(n * sizeof(double));

    nlocal = n / npes;
```

Row-Wise Matrix-Vector Multiplication

```
/* Gather the entire vector b on each processor using MPI's ALLGATHER operation */
    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];
    }

    free(fb);
}
```

Lecture Overview

- Leftover MPI Functions from (10/13)
  - MPI_Alltoallv
  - MPI_Comm_split & MPI_Cart_sub
- MPI Examples (Whiteboard walkthrough)
  - Row-Wise Matrix-Vector Multiplication
  - **Column-Wise Matrix-Vector Multiplication**
  - 2-D Matrix-Vector Multiplication
  - Djikstra's Single Source Shortest Path
  - Sample Sort

Column-Wise Matrix-Vector Multiplication

```
void ColMatrixVectorMultiply(int n, double *a, double *b, double *x, MPI_Comm comm)
{
    int i, j;
    int nlocal;
    double *px;
    double *fx;
    int npes, myrank;
    MPI_Status status;

    /* Get identity and size information from the communicator */
    MPI_Comm_size(comm, &npes);
    MPI_Comm_rank(comm, &myrank);

    nlocal = n / npes;
```

Column-Wise Matrix-Vector Multiplication

```
/* 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 */
    MPI_Reduce(px, fx, n, MPI_DOUBLE, MPI_SUM, 0, comm);

    /* Redistribute fx in a fashion similar to that of vector b */
    MPI_Scatter(fx, nlocal, MPI_DOUBLE, x, nlocal, MPI_DOUBLE, 0, comm);

    free(px);
    free(fx);
}
```

Lecture Overview

- Leftover MPI Functions from (10/13)
  - MPI_Alltoallv
  - MPI_Comm_split & MPI_Cart_sub
- MPI Examples (Whiteboard walkthrough)
  - Row-Wise Matrix-Vector Multiplication
  - Column-Wise Matrix-Vector Multiplication
  - **2-D Matrix-Vector Multiplication**
  - Djikstra’s Single Source Shortest Path
  - Sample Sort

2-D Matrix-Vector Multiplication

```
void MatrixVectorMultiply_2D(int n, double *a, double *b, double *x, MPI_Comm comm)
{
    int ROW = 0, COL = 1;
    int i, j, nlocal;
    double *px;
    int npes, dims[2], periods[2], keep_dims[2];
    int myrank, my2drank, mycoords[2];
    int other_rank, coords[2];
    MPI_Status status;
    MPI_Comm comm_2d, comm_row, comm_col;

    MPI_Comm_size(comm, &npes);
    MPI_Comm_rank(comm, &myrank);

    dims[ROW] = dims[COL] = sqrt(npes);
    nlocal = n / dims[ROW];
    px = malloc(nlocal * sizeof(double));
```

```
periods[ROW] = periods[COL] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_2d);
MPI_Comm_rank(comm_2d, &my2drank);
MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

keep_dims[ROW] = 0; keep_dims[COL] = 1;
MPI_Cart_sub(comm_2d, keep_dims, &comm_row);

keep_dims[ROW] = 1; keep_dims[COL] = 0;
MPI_Cart_sub(comm_2d, keep_dims, &comm_col);

if (mycoords[COL] == 0 && mycoords[ROW] != 0) {
    coords[ROW] = mycoords[ROW];
    coords[COL] = mycoords[ROW];
    MPI_Cart_rank(comm_2d, coords, &other_rank);
    MPI_Send(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d);
}

if (mycoords[ROW] == mycoords[COL] && mycoords[ROW] != 0) {
    coords[ROW] = mycoords[ROW];
    coords[COL] = 0;
    MPI_Cart_rank(comm_2d, coords, &other_rank);
    MPI_Recv(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d, &status);
}
```

2-D Matrix-Vector Multiplication

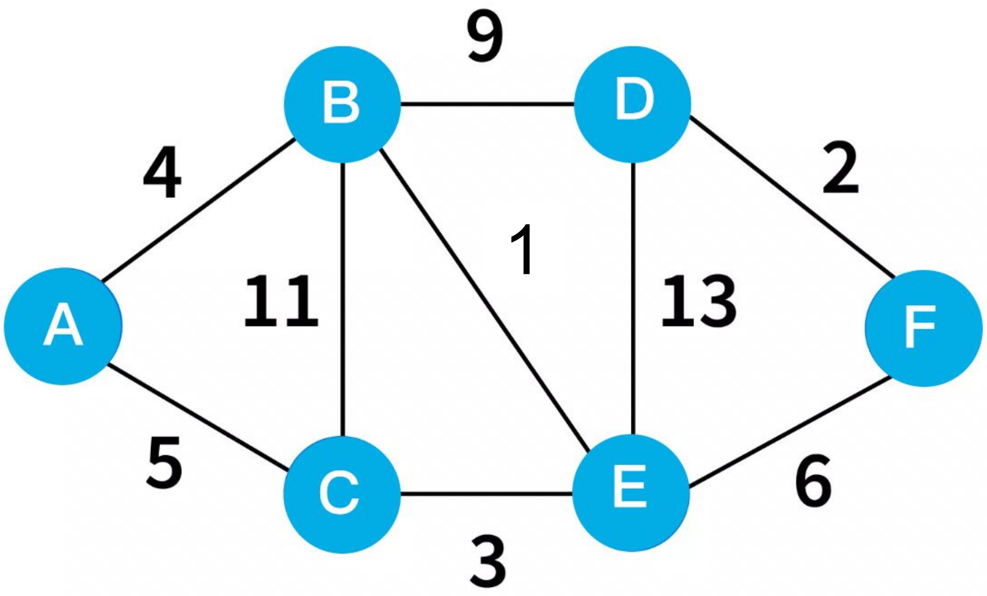
```
    coords[0] = mycoords[COL];
    MPI_Cart_rank(comm_col, coords, &other_rank);
    MPI_Bcast(b, nlocal, MPI_DOUBLE, other_rank, comm_col);

    for (i = 0; i < nlocal; i++) {
        px[i] = 0.0;
        for (j = 0; j < nlocal; j++)
            px[i] += a[i * nlocal + j] * b[j];
    }

    coords[0] = 0;
    MPI_Cart_rank(comm_row, coords, &other_rank);
    MPI_Reduce(px, x, nlocal, MPI_DOUBLE, MPI_SUM, other_rank, comm_row);

    MPI_Comm_free(&comm_2d);
    MPI_Comm_free(&comm_row);
    MPI_Comm_free(&comm_col);
    free(px);
}
```

Lecture Overview

- Leftover MPI Functions from (10/13)
  - MPI_Alltoallv
  - MPI_Comm_split & MPI_Cart_sub
- MPI Examples (Whiteboard walkthrough)
  - Row-Wise Matrix-Vector Multiplication
  - Column-Wise Matrix-Vector Multiplication
  - 2-D Matrix-Vector Multiplication
  - **Djikstra's Single Source Shortest Path**
  - Sample Sort

Djikstra's Single-Source Shortest Path (Review)

- Find the shortest path from a single node in a graph to all other nodes in a graph
- Serial Example Walkthrough



Djikstra’s Single-Source Shortest Path

```
void SingleSource(int n, int source, int *wgt, int *lengths, MPI_Comm comm)
{
    int i, j;
    int nlocal;      /* The number of vertices stored locally */
    int *marker;     /* Used to mark the vertices belonging to Vo */
    int firstvtx;    /* The index number of the first vertex that is stored locally */
    int lastvtx;     /* The index number of the last vertex that is stored locally */
    int u, udist;
    int lminpair[2], gminpair[2];
    int npes, myrank;
    MPI_Status status;

    MPI_Comm_size(comm, &npes);
    MPI_Comm_rank(comm, &myrank);

    nlocal = n / npes;
    firstvtx = myrank * nlocal;
    lastvtx = firstvtx + nlocal - 1;
```

Djikstra's Single-Source Shortest Path

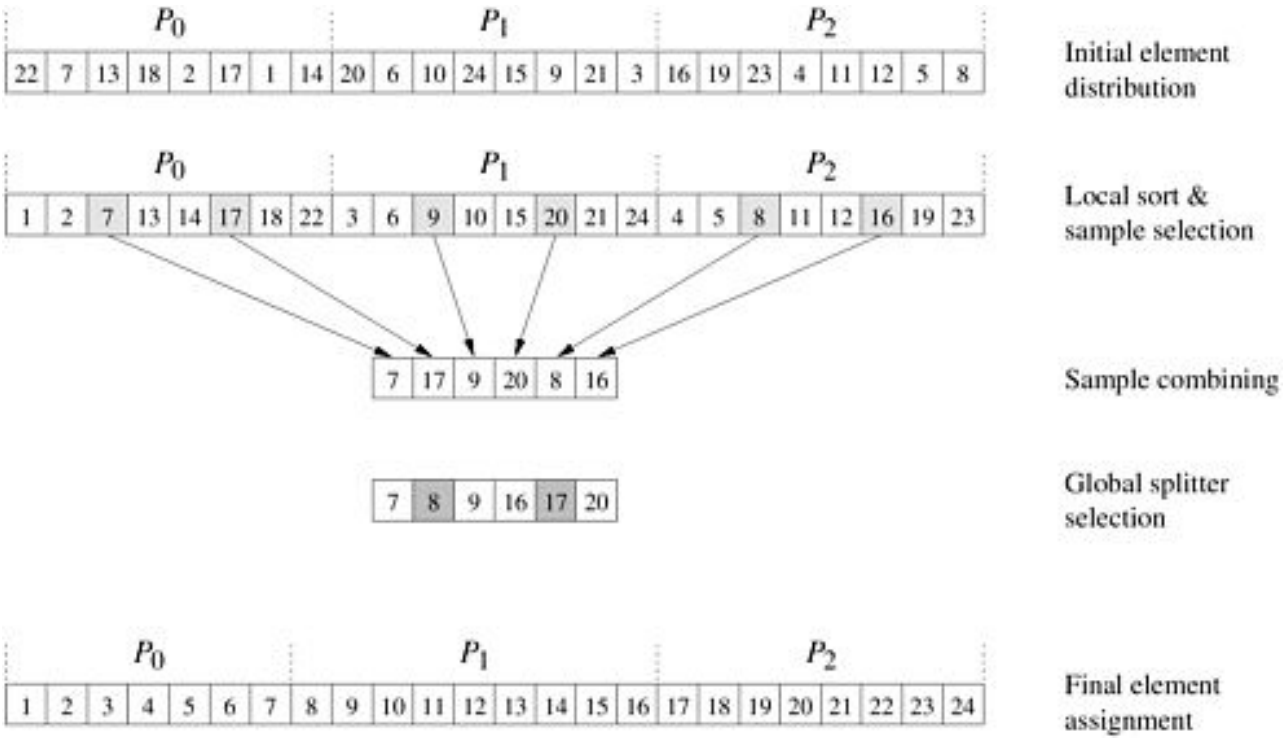
```
/* Set the initial distances from source to all the other vertices */
for (j = 0; j < nlocal; j++)
    lengths[j] = wgt[source * nlocal + j];

/* This array is used to indicate if the shortest path to a vertex has been found */
marker = (int *)malloc(nlocal * sizeof(int));
for (j = 0; j < nlocal; j++)
    marker[j] = 1;

/* The process that stores the source vertex marks it as seen */
if (source >= firstvtx && source <= lastvtx)
    marker[source - firstvtx] = 0;
```

```
    /* The main loop of Dijkstra's algorithm */
    for (i = 1; i < n; i++) {
        /* Step 1: Find the local vertex at the smallest distance from source */
        lminpair[0] = MAXINT; /* architecture dependent large number */
        lminpair[1] = -1;

        for (j = 0; j < nlocal; j++) {
            if (marker[j] && lengths[j] < lminpair[0]) {
                lminpair[0] = lengths[j];
                lminpair[1] = firstvtx + j;
            }
        }

        /* Step 2: Compute the global minimum vertex and insert it into Vc */
        MPI_Allreduce(lminpair, gminpair, 1, MPI_2INT, MPI_MINLOC, comm);
        udist = gminpair[0];
        u = gminpair[1];

        /* The process that stores the minimum vertex marks it as seen */
        if (u == lminpair[1])
            marker[u - firstvtx] = 0;

        /* Step 3: Update the distances given that u got inserted */
        for (j = 0; j < nlocal; j++) {
            if (marker[j] && udist + wgt[u * nlocal + j] < lengths[j])
                lengths[j] = udist + wgt[u * nlocal + j];
        }
    }
    free(marker);
}
```

Lecture Overview

- Leftover MPI Functions from (10/13)
  - MPI_Alltoallv
  - MPI_Comm_split & MPI_Cart_sub
- MPI Examples (Whiteboard walkthrough)
  - Row-Wise Matrix-Vector Multiplication
  - Column-Wise Matrix-Vector Multiplication
  - 2-D Matrix-Vector Multiplication
  - Djikstra's Single Source Shortest Path
  - **Sample Sort**

Sample Sort



P_0 P_1 P_2
22 7 13 18 2 17 1 14 20 6 10 24 15 9 21 3 16 19 23 4 11 12 5 8
Initial element distribution
1 2 7 13 14 17 18 22 3 6 9 10 15 20 21 24 4 5 8 11 12 16 19 23
Local sort & sample selection
7 17 9 20 8 16
Sample combining
7 8 9 16 17 20
Global splitter selection
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Final element assignment

Sample Sort

```
int *SampleSort(int n, int *elmnts, int *nsorted, MPI_Comm comm)
{
    int i, j, nlocal, npes, myrank;
    int *sorted_elmnts, *splitters, *allpicks;
    int *scounts, *sdispls, *rcounts, *rdispls;

    /* Get communicator-related information */
    MPI_Comm_size(comm, &npes);
    MPI_Comm_rank(comm, &myrank);

    nlocal = n / npes;

    /* Allocate memory for the arrays that will store the splitters */
    splitters = (int *)malloc(npes * sizeof(int));
    allpicks = (int *)malloc(npes * (npes - 1) * sizeof(int));

    /* Sort local array */
    qsort(elmnts, nlocal, sizeof(int), IncOrder);
```

Sample Sort

```
/* Select local npes-1 equally spaced elements */
for (i = 1; i < npes; i++)
    splitters[i - 1] = elmnts[i * nlocal / npes];

/* Gather the samples in the processors */
MPI_Allgather(splitters, npes - 1, MPI_INT, allpicks, npes - 1, MPI_INT, comm);

/* Sort these samples */
qsort(allpicks, npes * (npes - 1), sizeof(int), IncOrder);

/* Select splitters */
for (i = 1; i < npes; i++)
    splitters[i - 1] = allpicks[i * npes - (int)ceil((double)npes/2)];
splitters[npes - 1] = MAXINT;

/* Compute the number of elements that belong to each bucket */
scounts = (int *)malloc(npes * sizeof(int));
for (i = 0; i < npes; i++)
    scounts[i] = 0;
for (j = i = 0; i < nlocal; i++) {
    if (elmnts[i] < splitters[j])
        scounts[j]++;
    else
        scounts[++j]++;
}
```

Sample Sort

```
/* Determine starting locations of each bucket's elements */
   sdispls = (int *)malloc(npes * sizeof(int));
   sdispls[0] = 0;
   for (i = 1; i < npes; i++)
      sdispls[i] = sdispls[i - 1] + scounts[i - 1];

   /* Inform all processes about receive counts */
   rcounts = (int *)malloc(npes * sizeof(int));
   MPI_Alltoall(scounts, 1, MPI_INT, rcounts, 1, MPI_INT, comm);

   /* Compute receive displacements */
   rdispls = (int *)malloc(npes * sizeof(int));
   rdispls[0] = 0;
   for (i = 1; i < npes; i++)
      rdispls[i] = rdispls[i - 1] + rcounts[i - 1];

   *nsorted = rdispls[npes - 1] + rcounts[npes - 1];
   sorted_elmnts = (int *)malloc((*nsorted) * sizeof(int));

   /* Exchange elements between processors */
   MPI_Alltoallv(elmnts, scounts, sdispls, MPI_INT,
              sorted_elmnts, rcounts, rdispls, MPI_INT, comm);

   /* Final local sort */
   qsort(sorted_elmnts, *nsorted, sizeof(int), IncOrder);
```

Sample Sort

```
    free(splitters);
    free(allpicks);
    free(scounts);
    free(sdispls);
    free(rcounts);
    free(rdispls);

    return sorted_elmnts;
}
```