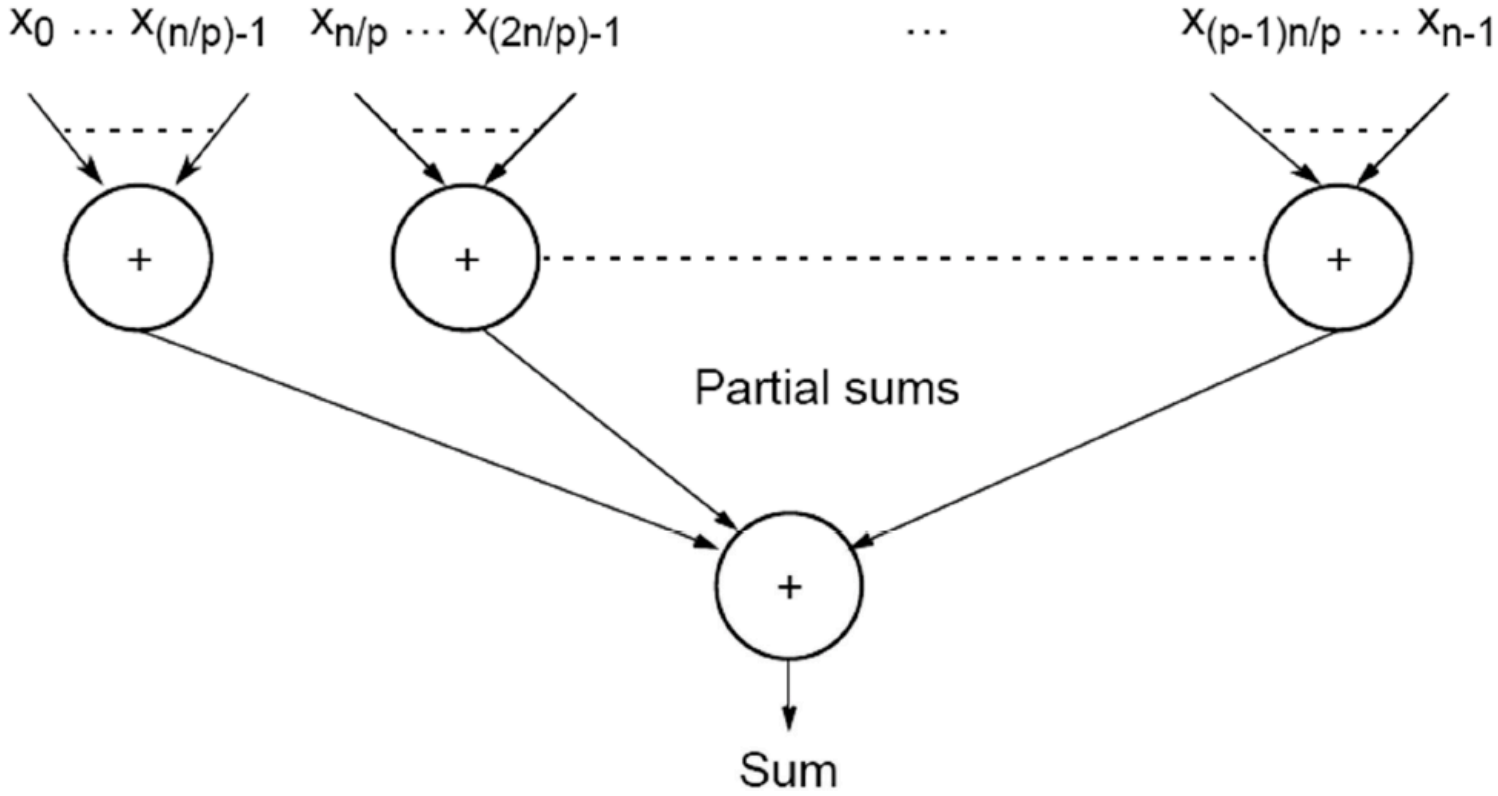
# CPSC 4770/6770

# Distributed and Cluster Computing

## Lecture 6: Partitioning and Divide and Conquer Strategies

# Partitioning

- Partitioning simply divides the problem into parts and then compute the parts and combine results

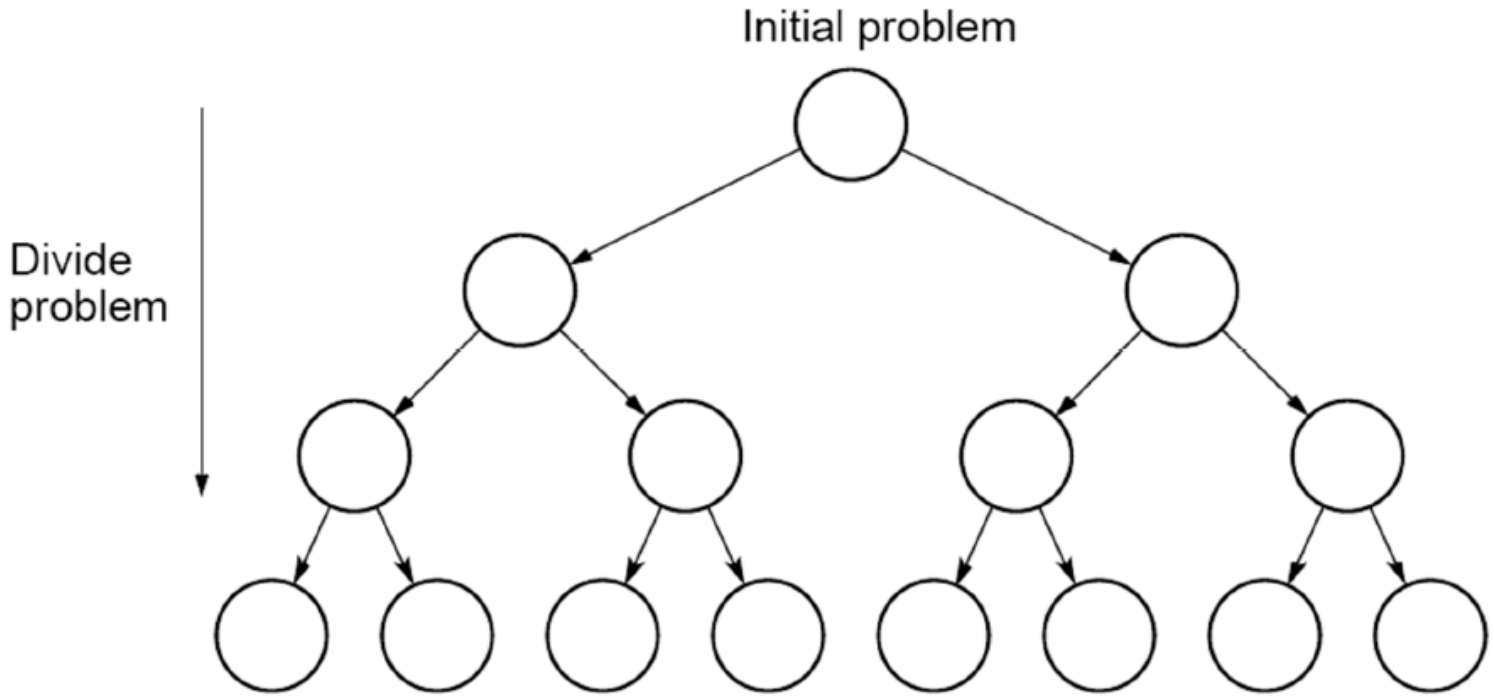



Partitioning a sequence of numbers into parts and adding the parts

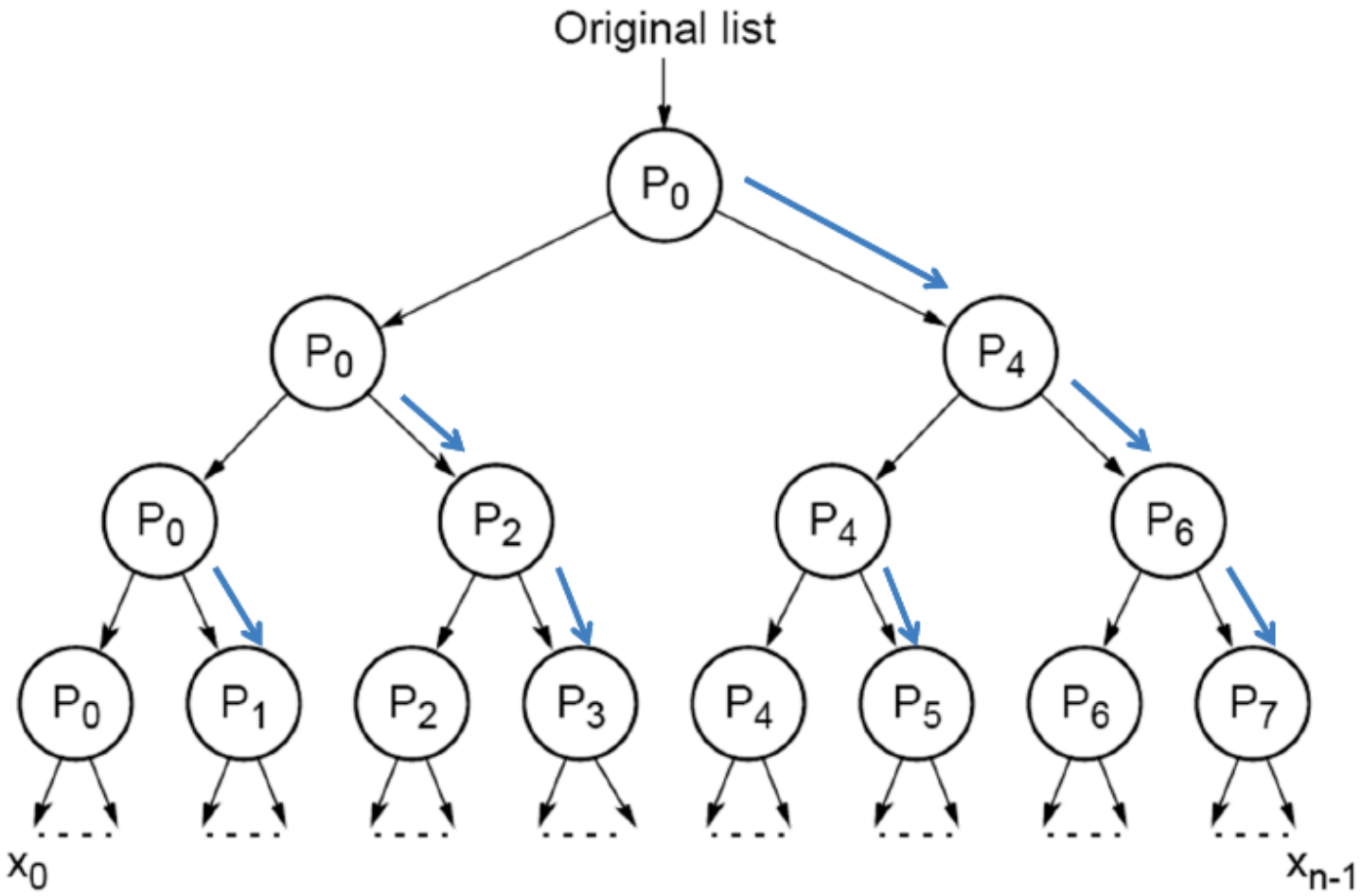
# Divide and Conquer

- Characterized by dividing problem into sub-problems of same form as larger problem. Further divisions into still smaller sub-problems, usually done by recursion
- Recursive divide and conquer amenable to parallelization because separate processes can be used for divided pairs. Also usually data is naturally localized

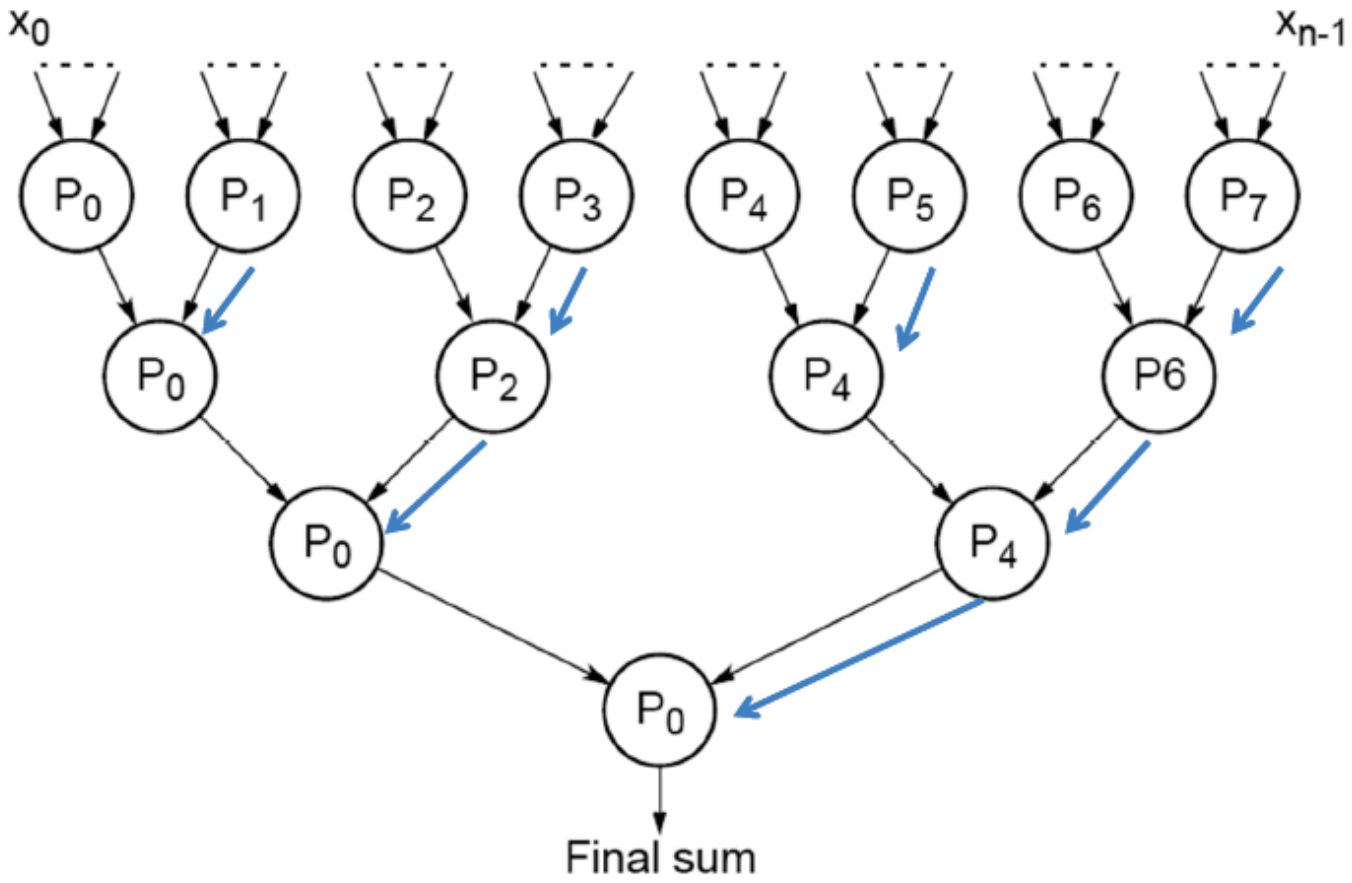
# Tree Construction



# Dividing a List into Parts



# Partial Summation



# divide.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

int main(int argc, char * argv[] ) {
  int rank;     /* rank of each MPI process */
  int size;     /* total number of MPI processes */
  int i;        /* counter */
  int distance; /* distance between sender and receiver */

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

  /* Am I sender or receiver? */
  /* Who am I sending/receiving to/from */
  distance = 1;
  i = 1;
  while (distance <= size / 2){
    if (rank < distance) {
      printf ("At time step %d, sender %d sends to %d\n", i, rank, rank + distance);
    }
    if ((rank >= distance) && (rank < distance * 2)){
      printf ("At time step %d, receiver %d receives from %d\n", i, rank, rank - distance);
    }
    printf ("Process %d has distance value %d and time step %d\n", rank, distance, i);
    distance = distance * 2;
    i += 1;
  }

  MPI_Finalize();
  return 0;
}
```

```
[jin6@node0329 07-divide-and-conquer]$ mpirun -np 8 divide
Process 2 has distance value 1 and time step 1
At time step 2, receiver 2 receives from 0
Process 2 has distance value 2 and time step 2
At time step 3, sender 2 sends to 6
Process 2 has distance value 4 and time step 3
Process 3 has distance value 1 and time step 1
At time step 2, receiver 3 receives from 1
Process 3 has distance value 2 and time step 2
At time step 3, sender 3 sends to 7
Process 3 has distance value 4 and time step 3
Process 4 has distance value 1 and time step 1
Process 4 has distance value 2 and time step 2
At time step 3, receiver 4 receives from 0
Process 4 has distance value 4 and time step 3
Process 5 has distance value 1 and time step 1
Process 5 has distance value 2 and time step 2
At time step 3, receiver 5 receives from 1
Process 5 has distance value 4 and time step 3
Process 6 has distance value 1 and time step 1
Process 6 has distance value 2 and time step 2
At time step 3, receiver 6 receives from 2
Process 6 has distance value 4 and time step 3
Process 7 has distance value 1 and time step 1
Process 7 has distance value 2 and time step 2
At time step 3, receiver 7 receives from 3
Process 7 has distance value 4 and time step 3
At time step 1, sender 0 sends to 1
Process 0 has distance value 1 and time step 1
At time step 2, sender 0 sends to 2
Process 0 has distance value 2 and time step 2
At time step 3, sender 0 sends to 4
Process 0 has distance value 4 and time step 3
At time step 1, receiver 1 receives from 0
Process 1 has distance value 1 and time step 1
At time step 2, sender 1 sends to 3
Process 1 has distance value 2 and time step 2
At time step 3, sender 1 sends to 5
Process 1 has distance value 4 and time step 3
```

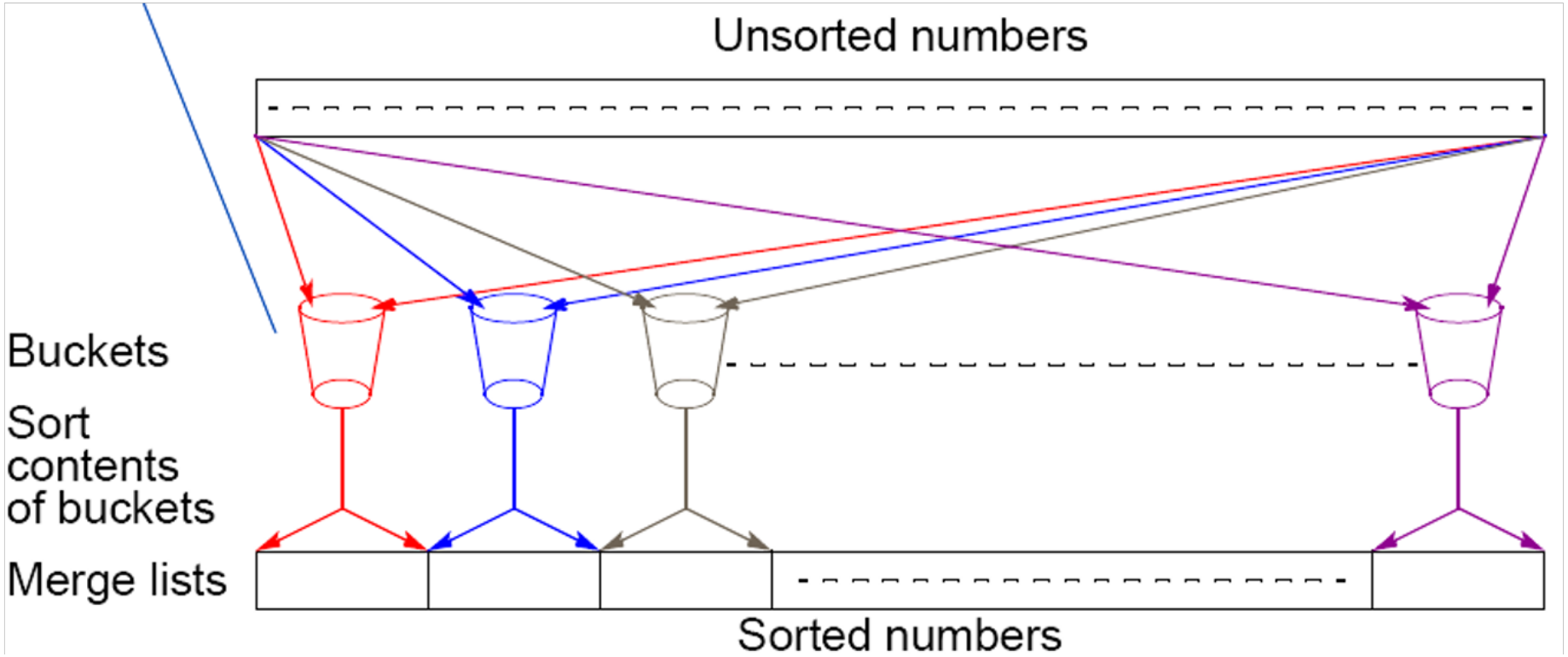
# conquer.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

int main(int argc, char * argv[] ) {
  int rank;     /* rank of each MPI process */
  int size;     /* total number of MPI processes */
  int i;        /* counter */
  int distance; /* distance between sender and receiver */

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

  /* Am I sender or receiver? */
  /* Who am I sending/receiving to/from */
  distance = (int)(size / 2);

  i = 1;
  while (distance >= 1){
    if ((rank >= distance) && (rank < distance * 2)){
      printf ("At time step %d, sender %d sends to %d\n", i, rank, rank - distance);
    }
    if (rank < distance) {
      printf ("At time step %d, receiver %d receives from %d\n", i, rank, rank + distance);
    }
    distance = distance / 2;
    i += 1;
  }

  MPI_Finalize();
  return 0;
}
```

```
[jin6@node0329 07-divide-and-conquer]$ mpirun -np 8 conquer
At time step 1, receiver 2 receives from 6
At time step 2, sender 2 sends to 0
At time step 1, sender 4 sends to 0
At time step 1, sender 5 sends to 1
At time step 1, sender 6 sends to 2
At time step 1, sender 7 sends to 3
At time step 1, receiver 0 receives from 4
At time step 2, receiver 0 receives from 2
At time step 3, receiver 0 receives from 1
At time step 1, receiver 1 receives from 5
At time step 2, receiver 1 receives from 3
At time step 3, sender 1 sends to 0
At time step 1, receiver 3 receives from 7
At time step 2, sender 3 sends to 1
```

# Example: Bucket Sort

One "bucket" assigned to hold numbers that fall within each region.
Numbers in each bucket sorted using a sequential sorting algorithm.

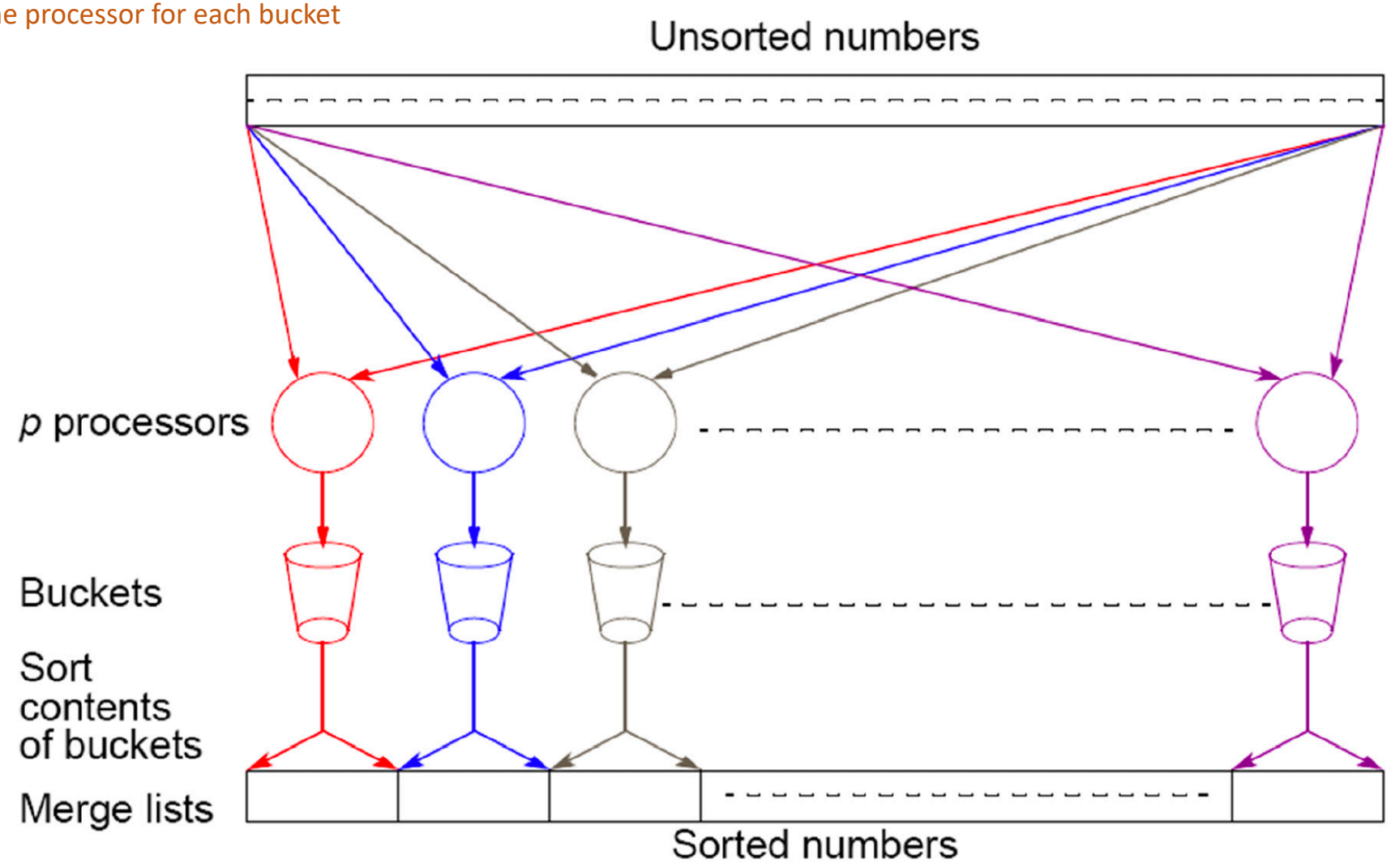



Sequential sorting time complexity: O(nlog(n/m))
Works well if the original numbers uniformly distributed across a known interval, say 0 to a-1.

Source: Barry Wilkinson and Michael Allen, Parallel Programming, 2nd Edition

# A Simple Parallel Version of Bucket Sort

Assign one processor for each bucket

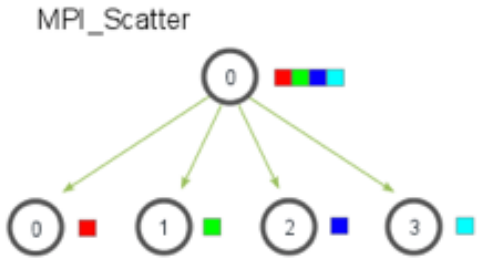


# Simple Parallel Version Implementation

- Phase 1: Broadcast data
- Phase 2: Sort only those elements that fit in local interval bucket (determined by rank)
- Phase 3: Gather sorted bucket

# MPI_Scatterv

- int **MPI_Scatter** (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm );




- int **MPI_Scatterv** (void *sendbuf, int *sendcnts, int *displs, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm );

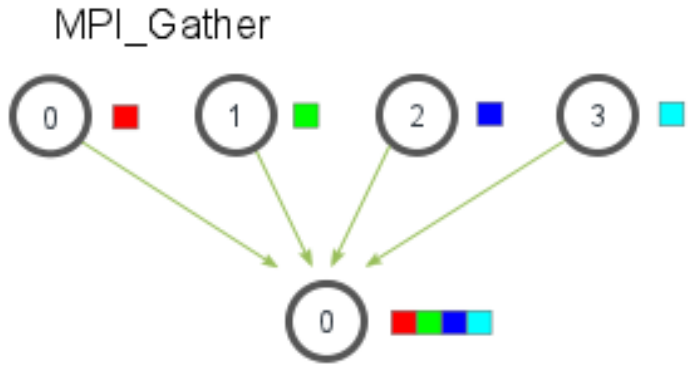
# scatterv.c

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int rank, size;
    int i;
    int sendcounts[4] = {1,2,3,4}; /* each process will receive its rank plus 1 numbers from the sendbuf array */
    int displs[4] = {0,0,0,0}; /* array describing the displacements where each segment begins and is initialized to all 0s */
    int sendbuf[10] = {2,13,4,3,5,1,0,12,10,8}; /* the buffer to be sent */
    int *recvbuf; /* array at each process to receive data. To be initialized based on process rank */

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

    /* initializes recvbuf to contain exactly rank plus 1 numbers */
    recvbuf = malloc(sizeof(int)* (rank + 1));

    // calculate displacements
    for (i = 1; i < 4; i++) {
        displs[i] = displs[i-1] + sendcounts[i-1];
    }

    // divide the data among processes as described by sendcounts and displs
    MPI_Scatterv(sendbuf, sendcounts, displs, MPI_INT, recvbuf, (rank + 1), MPI_INT, 0, MPI_COMM_WORLD);

    // print what each process received
    printf("%d: ", rank);
    for (i = 0; i < sendcounts[rank]; i++) {
        printf("%d ", recvbuf[i]);
    }
    printf("\n");

    free(recvbuf);

    MPI_Finalize();
    return 0;
}
```

```
[jin6@node0087 07-divide-and-conquer]$ mpiexec -np 4 scatterv
0: 2
1: 13 4
3: 0 12 10 8
2: 3 5 1
```

# MPI_Gatherv

- int **MPI_Gather** ( void *sendbuf, int sendcnt, MPI_Datatype sendtype,void *recvbuf, int recvcnt, MPI_Datatype recvtype,int root, MPI_Comm comm );




- int **MPI_Gatherv** ( void *sendbuf, int sendcnt, MPI_Datatype sendtype,void *recvbuf, int *recvcnts, int *displs, MPI_Datatype recvtype,int root, MPI_Comm comm );

# gatherv.c

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int rank, size;
    int i;
    int recvcounts[4] = {1,2,3,4}; /* process 0 will receive from each process that process rank */
                                   /* plus 1 numbers */
    int displs[4] = {0,0,0,0}; /* array describing the displacements where each segment begins */
                               /* and is initialized to all 0s */
    int *sendbuf; /* the buffer to be sent. will be initialized individually at each process */
    int *recvbuf; /* arrayto receive data. will only be initialized at process 0*/

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

    /* initializes recvbuf to receive 10 numbers */
    if (rank == 0){
      recvbuf = malloc(sizeof(int) * (10));

      for (i = 0; i < 10; i ++)
        recvbuf[i] = -1;
    }

    /* initializes sendbuf to receive 10 numbers */
    sendbuf = malloc(sizeof(int) * (rank + 1));
    for (i = 0; i < (rank + 1); i++){
        sendbuf[i] = rank;
    }

    // calculate displacements
    for (i = 1; i < 4; i++) {
        displs[i] = displs[i-1] + recvcounts[i-1];
    }

    // divide the data among processes as described by sendcounts and displs
    MPI_Gatherv(sendbuf, rank + 1, MPI_INT, recvbuf, recvcounts, displs, MPI_INT, 0, MPI_COMM_WORLD);

    // print what process has at the end
    if (rank == 0){
      for (i = 0; i < 10; i++) {
        printf("%d ", recvbuf[i]);
      }
      printf("\n");
      free(recvbuf);
    }
    MPI_Finalize();
    return 0;
}
```

```
[jin6@node0281 07-divide-and-conquer]$ mpirun -np 4 gatherv
0 1 1 2 2 2 3 3 3 3
```

# bucket1.c

- Phase 1: Broadcast data

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define N 64

int main(int argc, char* argv[]){

  int rawNum[N];
  int sortNum[N];
  int* local_bucket;
  int rank,size;
  int* proc_count;
  int* disp;
  MPI_Status status;
  int i,j,counter;
  int local_min,local_max;
  int tmp;

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

  if (rank == 0){
    /* Initialize a random array with N integers whose values range between 0 and N */
    for (i = 0; i < N; i++){
      rawNum[i] = rand() % N;
    }
  }

  /* Broadcast contents of rawNum from 0 to all other processes */
  MPI_Bcast(rawNum, N, MPI_INT, 0, MPI_COMM_WORLD);
```

# bucket1.c (Cont.)

- Phase 2: Sort only those elements that fit in local interval bucket (determined by rank)

```
34  /* Each process only works with numbers within their assigned interval */
35  counter = 0;
36  local_min = rank * (N/size);
37  local_max = (rank + 1) * (N/size);
38  for (i = 0; i < N; i++){
39    if ((rawNum[i] >= local_min) && (rawNum[i] < local_max)){
40      counter += 1;
41    }
42  }
43
44  printf("For rank %d, max is %d, min is %d, and there are %d elements in rawNum that falls within max and min \n",
45         rank,local_max,local_min,counter);
46
47  /* Each process creates its own bucket containing values that fall within its interval */
48  local_bucket = malloc(counter * sizeof(int));
49  counter = 0;
50  for (i = 0; i < N; i++){
51    if ((rawNum[i] >= local_min) && (rawNum[i] < local_max)){
52      local_bucket[counter] = rawNum[i];
53      counter += 1;
54    }
55  }
56
57  /* Insertion sort */
58  for (i = 0; i < counter; i++){
59    for (j = i+1; j < counter; j++){
60      if (local_bucket[i] > local_bucket[j]){
61        tmp = local_bucket[i];
62        local_bucket[i] = local_bucket[j];
63        local_bucket[j] = tmp;
64      }
65    }
66  }
67
68  for (i = 0; i < counter; i++){
69    printf("%d %d \n",rank,local_bucket[i]);
70  }
```

```
[jin6@node0087 07-divide-and-conquer]$ mpiexec -np 8 bucket1
For rank 2, max is 24, min is 16, and there are 6 elements in rawNum that falls within max and min
For rank 3, max is 32, min is 24, and there are 11 elements in rawNum that falls within max and min
For rank 4, max is 40, min is 32, and there are 10 elements in rawNum that falls within max and min
For rank 5, max is 48, min is 40, and there are 5 elements in rawNum that falls within max and min
For rank 6, max is 56, min is 48, and there are 10 elements in rawNum that falls within max and min
For rank 7, max is 64, min is 56, and there are 8 elements in rawNum that falls within max and min
For rank 0, max is 8, min is 0, and there are 9 elements in rawNum that falls within max and min
```

```
0 1   1 9    2 17   3 24   4 33   5 40   6 48   7 56
0 1   1 13   2 20   3 24   4 33   5 41   6 49   7 59
0 1   1 13   2 20   3 26   4 33   5 41   6 50   7 60
0 2   1 13   2 20   3 26   4 35   5 43   6 50   7 60
0 2   1 14   2 23   3 26   4 35   5 46   6 50   7 61
0 5          2 23   3 27   4 35          6 51   7 62
0 6                 3 27   4 37          6 52   7 62
0 6                 3 28   4 38          6 52   7 62
0 7                 3 29   4 39          6 54
                    3 30   4 39          6 55
                    3 31
```

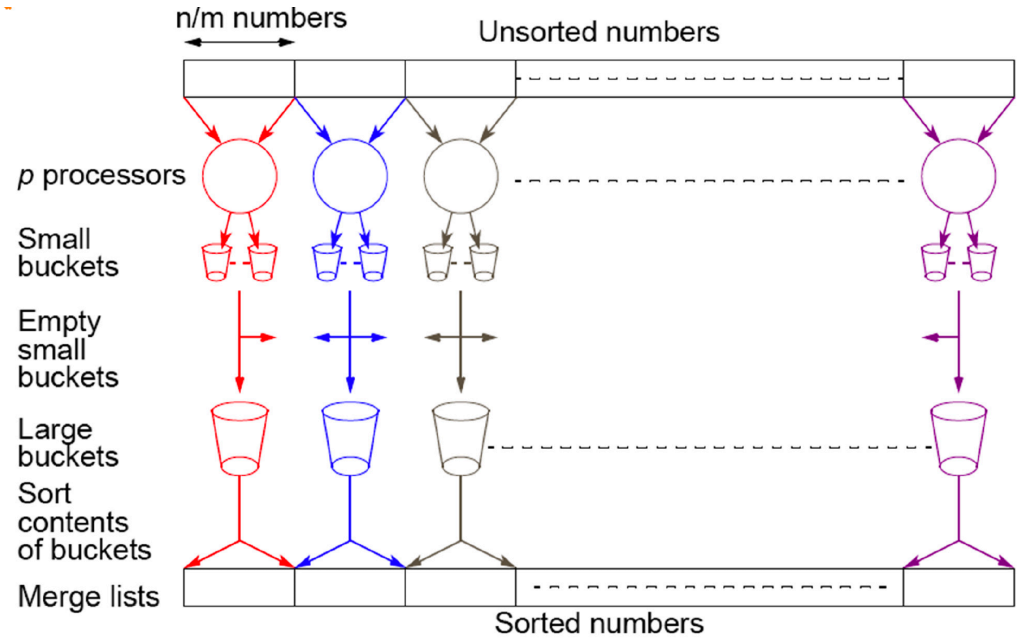
# bucket1.c (Cont.)

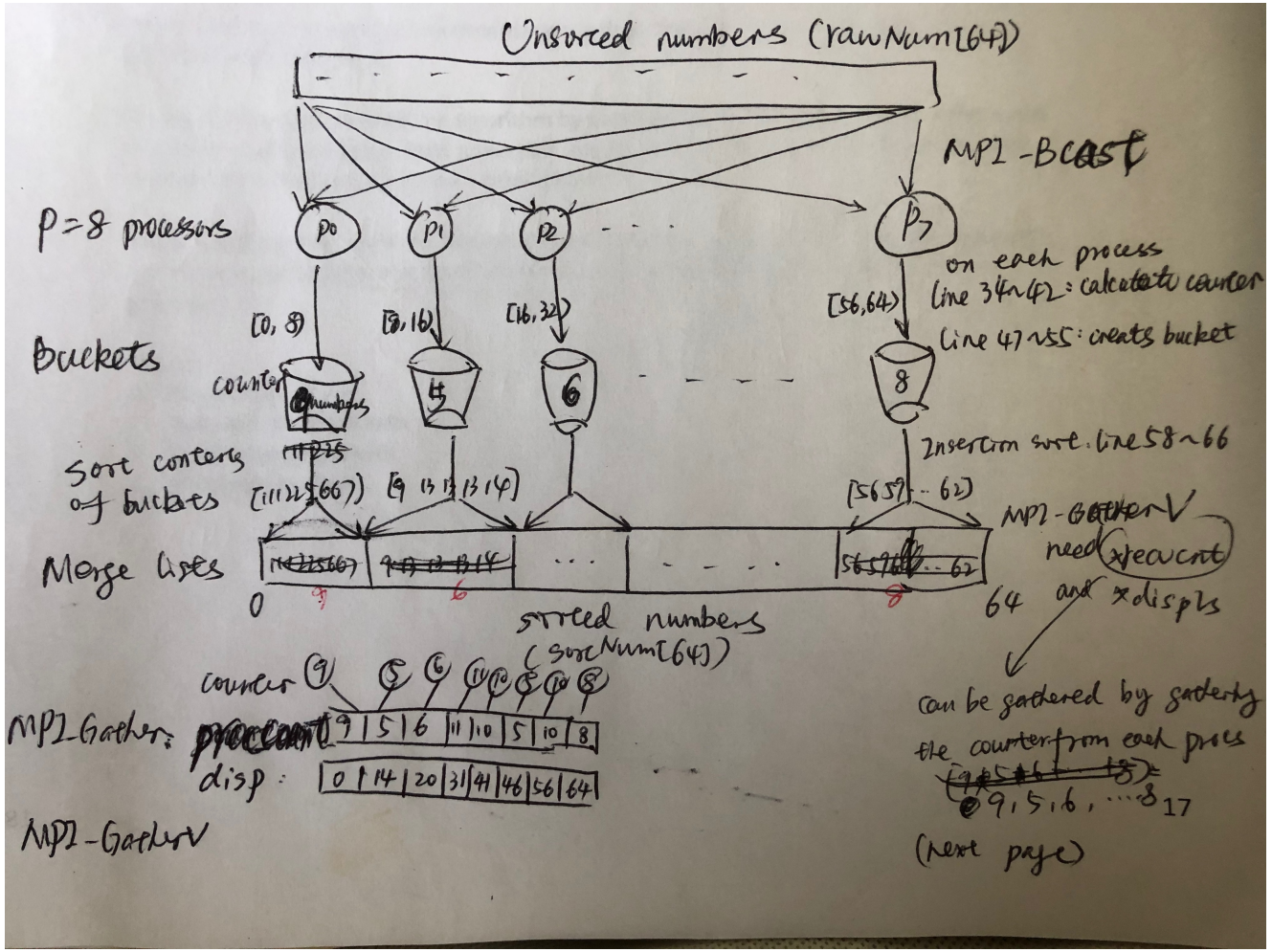
- Phase 3: Gather sorted bucket

```
72   /* set up root process */
73   if (rank == 0){
74     proc_count = malloc(size * sizeof(int));
75     disp = malloc(size * sizeof(int));
76   }
77
78   /* populate proc_count */
79   MPI_Gather(&counter,1,MPI_INT,proc_count,1,MPI_INT,0,MPI_COMM_WORLD);
80
81   if (rank == 0){
82     disp[0] = 0;
83     for (i = 0; i < size-1; i++){
84       disp[i+1] = disp[i] + proc_count[i];
85     }
86   }
87
88   // receive final result
89   MPI_Gatherv(local_bucket,counter,MPI_INT,sortNum,proc_count,disp,MPI_INT,0,MPI_COMM_WORLD);
90
91   if (rank == 0){
92     printf("Before sort: \n");
93     for (i = 0; i < N; i++) printf("%d ",rawNum[i]);
94     printf("\nAfter sort: \n");
95     for (i = 0; i < N; i++) printf("%d ",sortNum[i]);
96   }
97
98   MPI_Finalize();
99   return 0;
100 }
```

```
Before sort:
50 59 35 6 60 2 20 56 27 40 39 13 54 26 46 35 51 31 9 26 38 50 13 55 49 24 35 26 37 29 5 23
 24 41 30 20 43 50 13 6 27 52 20 17 14 2 52 1 33 61 28 7 48 41 62 33 1 33 60 39 62 1 62 23
After sort:
1 1 1 2 2 5 6 6 7 9 13 13 13 14 17 20 20 20 23 23 24 24 26 26 26 27 27 28 29 30 31 33 33 33
 35 35 35 37 38 39 39 40 41 41 43 46 48 49 50 50 50 51 52 52 54 55 56 59 60 60 61 62 62 62
```

# Further Parallel Version of Bucket Sort



# Further Parallel Version Implementation

Phase 1: Partition numbers

Phase 2: Sort into small buckets

Phase 3: Send to large buckets

Phase 4: Sort large buckets

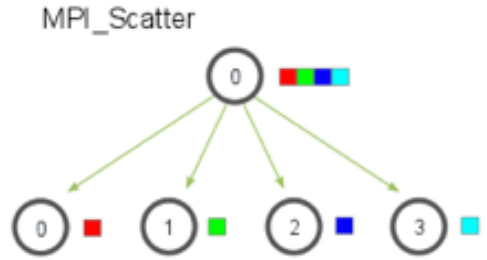
# Further Parallel Version Implementation

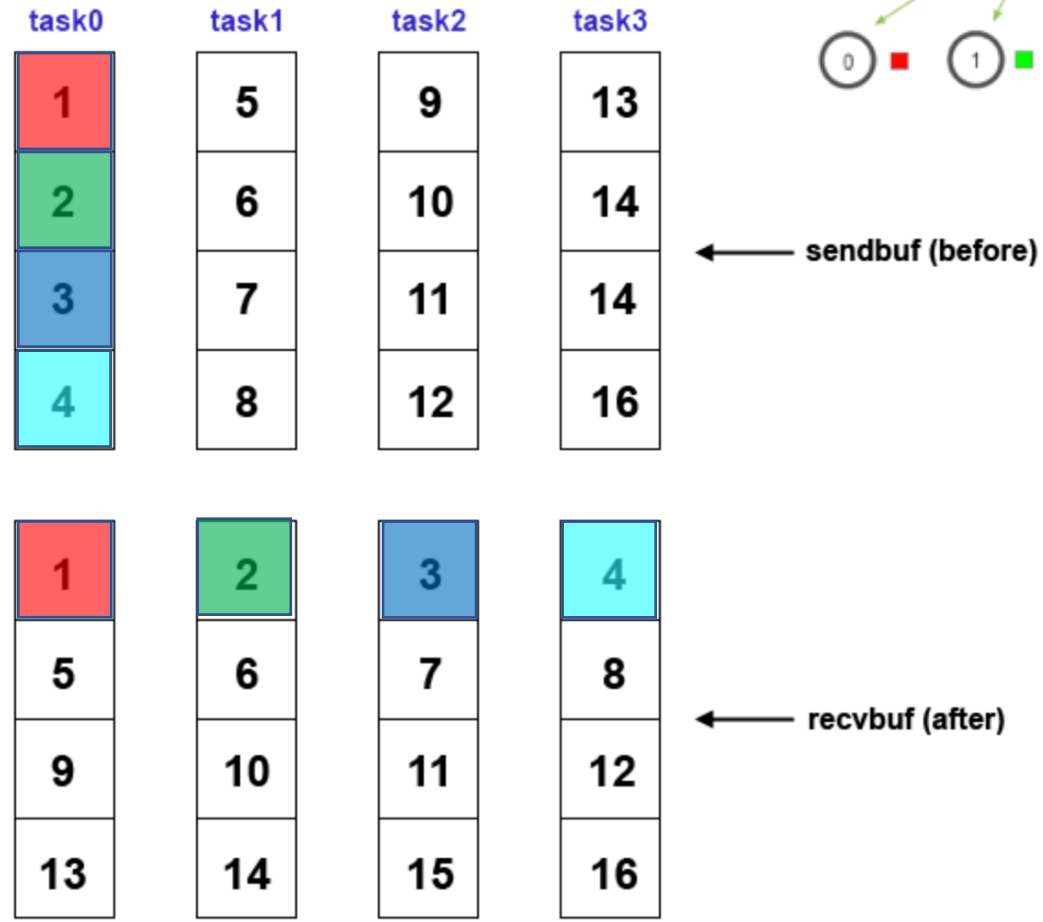
- Partition sequence into m regions
- Each processor assigned one region (Hence number of processors, p, equals m)
- Each processor maintains one “big” bucket for its region
- Each processor maintains m “small” buckets, one for each region
- Each processor separates numbers in its region into its own small buckets
- All small buckets emptied into p big buckets
- Each big bucket sorted by its processor

# MPI_Alltoall

- int **MPI_Alltoall**( void *sendbuf, int sendcnt, MPI_Datatype sendtype,void *recvbuf, int recvcnt, MPI_Datatype recvtype,MPI_Comm comm ): Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index

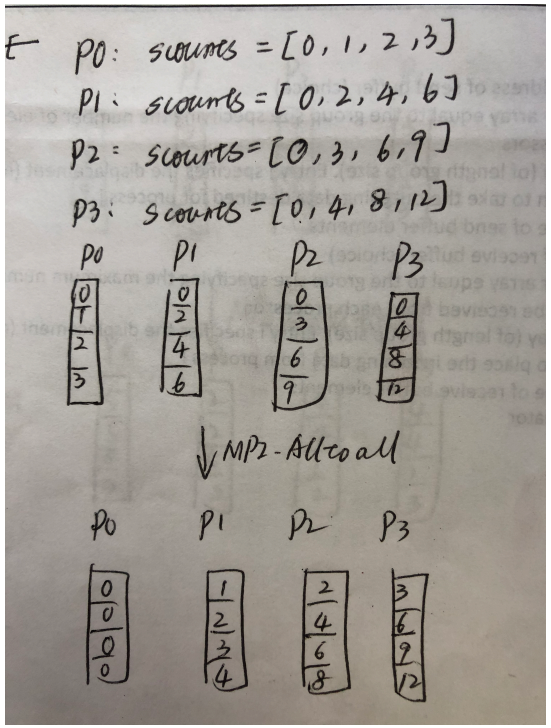
```
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT
             recvbuf, recvcnt, MPI_INT
             MPI_COMM_WORLD);
```



# alltoall.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc,char *argv[]){
  int rank, size;
  int *sray,*rray;
  int *sdisp,*scounts,*rdisp,*rcounts;
  int ssize,rsize,i,k,j;
  float z;

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

  scounts=(int*)malloc(sizeof(int)*size);
  rcounts=(int*)malloc(sizeof(int)*size);
  sdisp=(int*)malloc(sizeof(int)*size);
  rdisp=(int*)malloc(sizeof(int)*size);

  z = (float) rand() / RAND_MAX;

  for(i=0; i < size; i++){
    scounts[i]= rank * i + i;
  }

  printf("myid = %d scounts = ",rank);
  for(i=0;i<size;i++)
    printf("%d ",scounts[i]);
  printf("\n");

  /* send the data */
  MPI_Alltoall(scounts,1,MPI_INT,rcounts,1,MPI_INT,MPI_COMM_WORLD);
  printf("myid = %d rcounts = ",rank);
  for(i=0;i<size;i++)
    printf("%d ",rcounts[i]);
  printf("\n");
  MPI_Finalize();
}
```



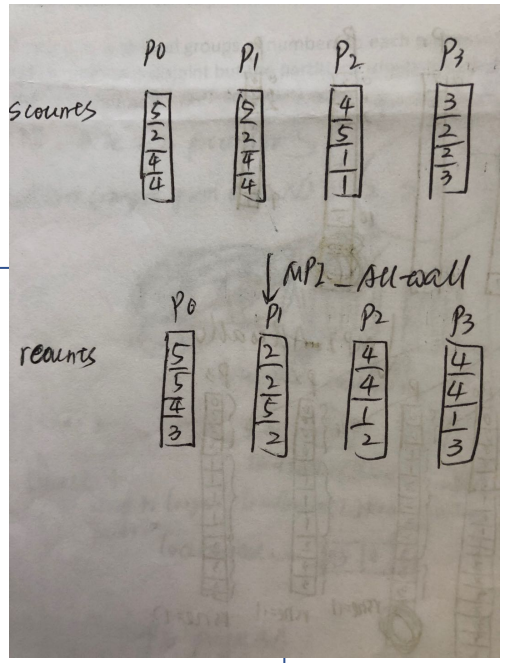

```
[jin6@node0087 07-divide-and-conquer]$ mpirun -np 4 alltoall
myid = 0 scounts = 0 1 2 3
myid = 0 rcounts = 0 0 0 0
myid = 1 scounts = 0 2 4 6
myid = 1 rcounts = 1 2 3 4
myid = 2 scounts = 0 3 6 9
myid = 2 rcounts = 2 4 6 8
myid = 3 scounts = 0 4 8 12
myid = 3 rcounts = 3 6 9 12
```

# MPI_Alltoallv

- int **MPI_Alltoallv**( void *sendbuf, int *sendcnts, int *sdispls, MPI_Datatype sendtype,void *recvbuf, int *recvcnts, int *rdispls, MPI_Datatype recvtype,MPI_Comm comm ): Sends data from all to all processes; each process may send a different amount of data and provide displacements for the input and output data
- Additional reference: alltoallv.ppt (Lori Pollock, University of Delaware)

# alltoallv.c

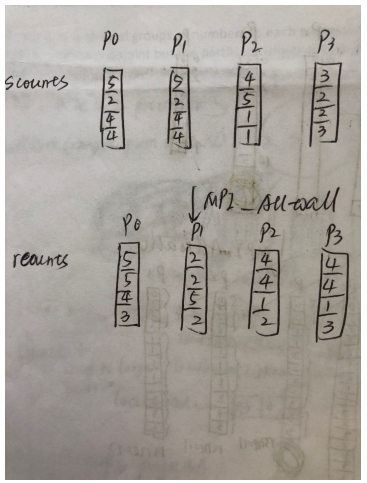
```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc,char *argv[]){
  int size, rank;
  int *sray,*rray;
  int *sdisp,*scounts,*rdisp,*rcounts;
  int ssize,rsize,i,k,j;
  float z;

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

  scounts=(int*)malloc(sizeof(int)*size);
  rcounts=(int*)malloc(sizeof(int)*size);
  sdisp=(int*)malloc(sizeof(int)*size);
  rdisp=(int*)malloc(sizeof(int)*size);

  /* find out how much data to send */
  srand((unsigned int) rank);
  for(i=0;i<size;i++){
    z = (float) rand()/RAND_MAX;
    scounts[i]=(int)(5.0 * z) + 1;
  }

  printf("rank= %d scounts= %d %d %d %d\n",rank,scounts[0],scounts[1],scounts[2],scounts[3]);

  /* tell the other processors how much data is coming */
  MPI_Alltoall(scounts,1,MPI_INT,rcounts,1,MPI_INT,MPI_COMM_WORLD);
```

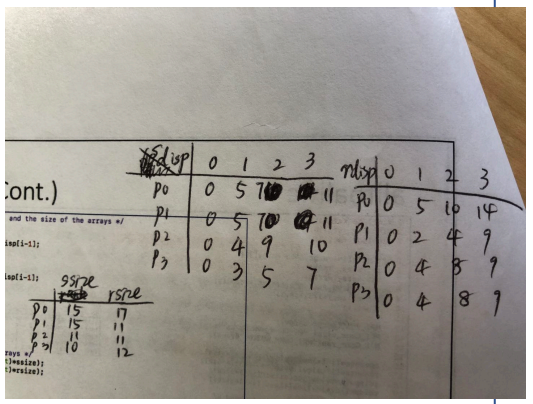


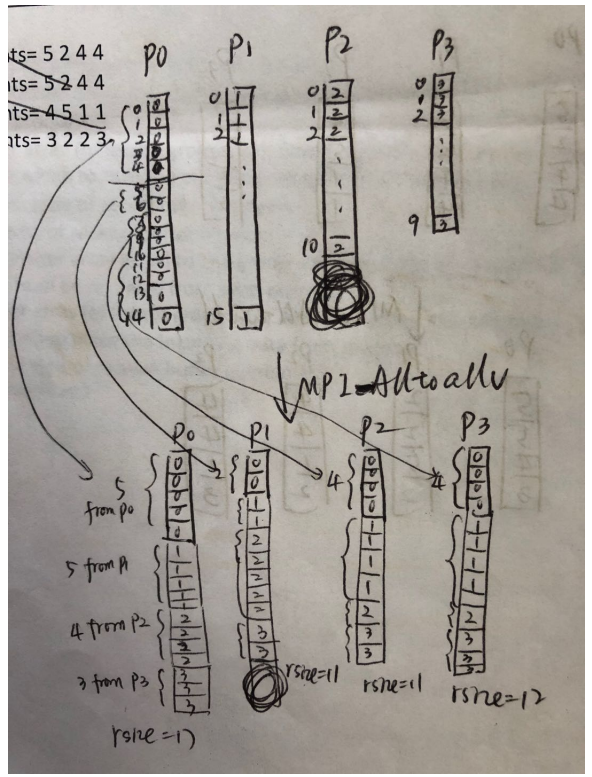

```
[jin6@node0087 07-divide-and-conquer]$ mpirun -np 4 alltoallv
rank= 0 scounts= 5 2 4 4
rank= 1 scounts= 5 2 4 4
rank= 2 scounts= 4 5 1 1
rank= 3 scounts= 3 2 2 3
```

# alltoallv.c (Cont.)

```
33   /* calculate displacements and the size of the arrays */
34   sdisp[0]=0;
35   for(i=1;i<size;i++){
36     sdisp[i]=scounts[i-1]+sdisp[i-1];
37   }
38   rdisp[0]=0;
39   for(i=1;i<size;i++){
40     rdisp[i]=rcounts[i-1]+rdisp[i-1];
41   }
42   ssize=0;
43   rsize=0;
44   for(i=0;i<size;i++){
45     ssize=ssize+scounts[i];
46     rsize=rsize+rcounts[i];
47   }
48
49   /* allocate send and rec arrays */
50   sray=(int*)malloc(sizeof(int)*ssize);
51   rray=(int*)malloc(sizeof(int)*rsize);
52   for(i=0;i<ssize;i++)
53     sray[i]=rank;
54
55   /* send/rec different amounts of data to/from each processor */
56   MPI_Alltoallv( sray,scounts,sdisp,MPI_INT,rray,rcounts,rdisp,MPI_INT,MPI_COMM_WORLD);
57
58   printf("rank= %d rray=",rank);
59   for(i=0;i<rsize;i++)
60     printf("%d ",rray[i]);
61   printf("\n");
62   MPI_Finalize();
63 }
```



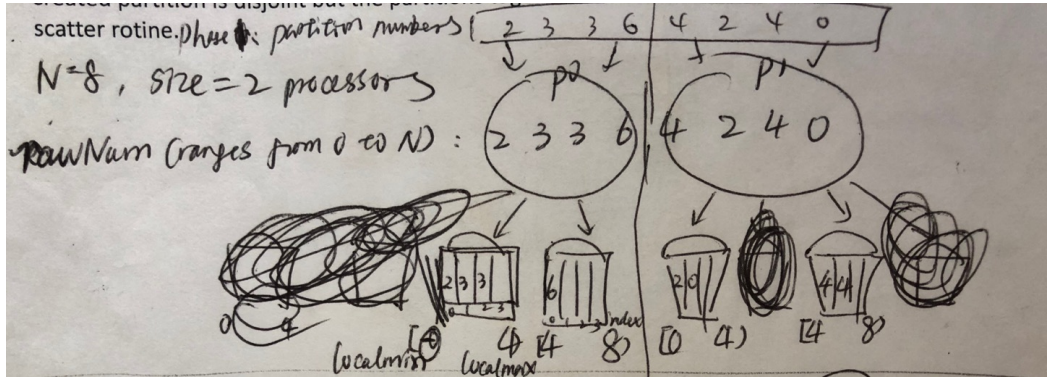






```
rank= 0 rray=0 0 0 0 0 1 1 1 1 1 2 2 2 2 3 3 3
rank= 1 rray=0 0 1 1 2 2 2 2 2 3 3
rank= 2 rray=0 0 0 0 1 1 1 1 2 3 3
rank= 3 rray=0 0 0 0 1 1 1 1 2 3 3 3
```

# bucket2.c

- Phase 1: Partition numbers

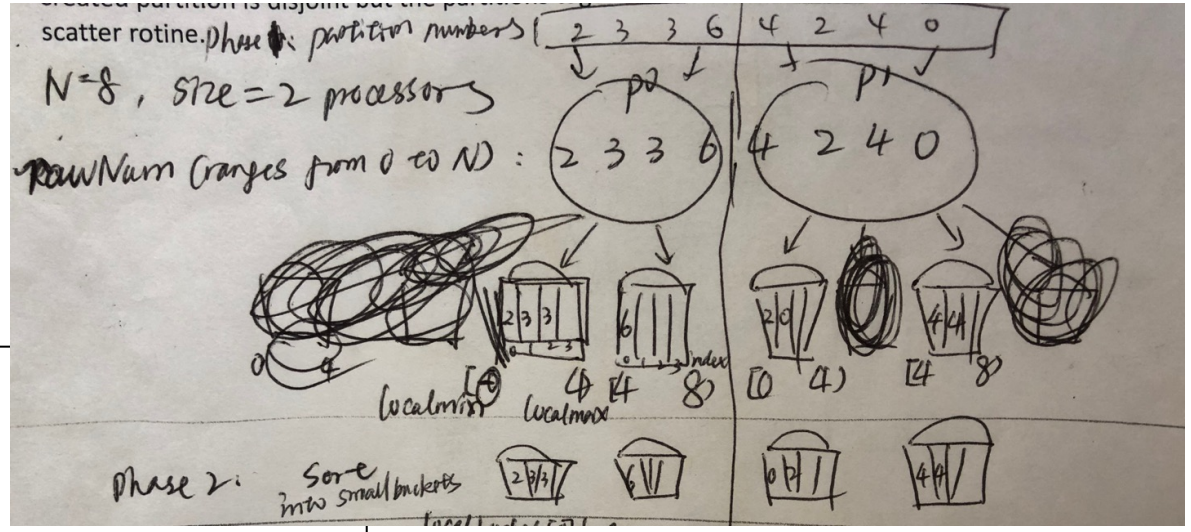



```c
/* Phase 1: Partition numbers */

  // prepare the local container, then distribute equal portions of the
  // unsorted array to all the processes from process 0
  local_array = malloc((N/size) * sizeof(int));
  MPI_Scatter(rawNum,(N/size),MPI_INT,local_array,(N/size),MPI_INT,0,MPI_COMM_WORLD);

  // initialize the local bucket matrix
  int local_bucket[size][N/size];
  for (i = 0; i < size; i++){
    for (j = 0; j < N/size; j++){
      local_bucket[i][j] = RAND_MAX;
    }
  }

  int counter = 0;
  int local_min,local_max;
  // populate the bucket matrix
  for (i = 0; i < size; i++){
    counter = 0;
    for (j = 0; j < N/size; j++){
      local_min = i * N/size;
      local_max = (i + 1) * N / size;
      if ((local_array[j] >= local_min)&&(local_array[j] < local_max)){
        local_bucket[i][counter] = local_array[j];
        printf("rank %d: local_bucket[%d][%d] = %d\n", rank, i, counter, local_array[j]);
        counter += 1;
      }
    }
  }
```

```
[jin6@node0087 07-divide-and-conquer]$ mpiexec -np 2 bucket2
Rank 0: local_bucket[0][0] = 2
Rank 0: local_bucket[0][1] = 3
Rank 0: local_bucket[0][2] = 3
Rank 0: local_bucket[1][0] = 6


Rank 1: local_bucket[0][0] = 2
Rank 1: local_bucket[0][1] = 0
Rank 1: local_bucket[1][0] = 4
Rank 1: local_bucket[1][1] = 4
```

# bucket2.c (Cont.)

## Phase 2: Sort into small buckets

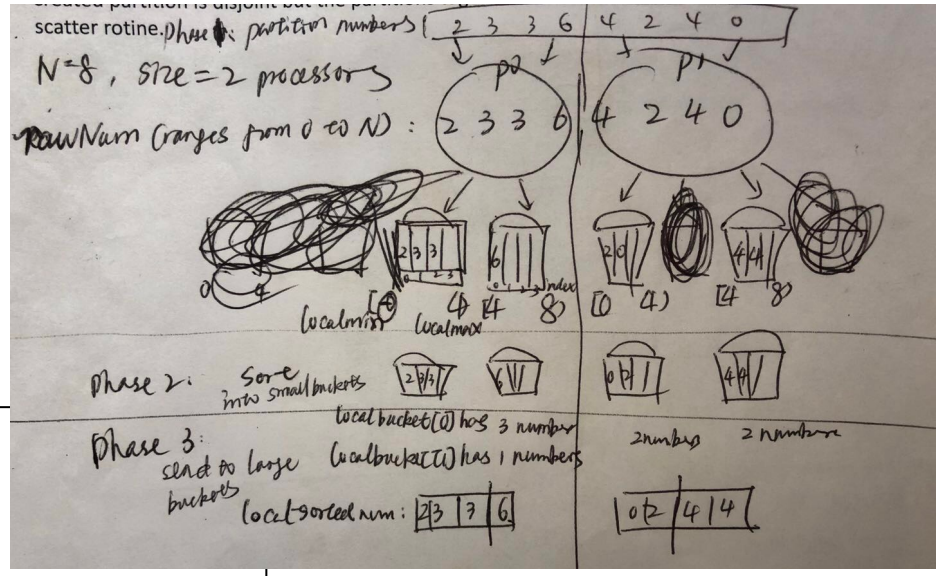



```
60 /* Phase 2: Sort into small buckets */
61
62   // sort the bucket matrix.
63   int tmp = 0;
64   for (i = 0; i < size; i++){
65     for (j = 0; j < N/size; j++){
66       for (counter = j; counter < N/size; counter++){
67         if (local_bucket[i][j] > local_bucket[i][counter]){
68           tmp = local_bucket[i][j];
69           local_bucket[i][j] = local_bucket[i][counter];
70           local_bucket[i][counter] = tmp;
71         }
72       }
73     }
74   }
```

# bucket2.c (Cont.)

## Phase 3: Send to large buckets

```
76 /* Phase 3: Send to large buckets */
77
78   // placing the number from the buckets back into the main array
79   counter = 0;
80   int array_counter[size];
81   for (i = 0; i < size; i++){
82     for (j = 0; j < N/size; j++){
83       if (local_bucket[i][j] != RAND_MAX){
84         local_array[counter] = local_bucket[i][j];
85         counter += 1;
86       }
87       else {
88         array_counter[i] = j;
89         printf("Rank %d: local_bucket[%d]'s counter = %d \n",rank,i,array_counter[i]);
90         break;
91       }
92     }
93   }
94
95   printf("Rank %d: local-sorted num: ",rank);
96   for (i = 0; i < N/size; i++){
97     printf("%d ", local_array[i]);
98   }
99   printf("\n");
100
101  MPI_Barrier(MPI_COMM_WORLD);
```

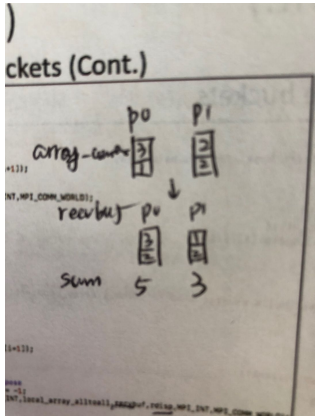


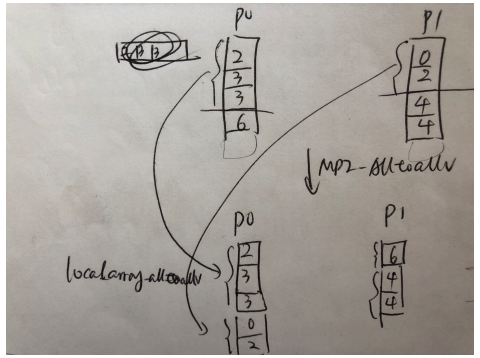

```
Rank 0: local_bucket[0]'s counter = 3
Rank 0: local_bucket[1]'s counter = 1
Rank 0: local-sorted num: 2 3 3 6


Rank 1: local_bucket[0]'s counter = 2
Rank 1: local_bucket[1]'s counter = 2
Rank 1: local-sorted num: 0 2 4 4
```

# bucket2.c (Cont.)

## Phase 3: Send to large buckets (Cont.)

```
103   // preparation for bucket gathering
104   int recvbuf[size];
105   int rdisp[size];
106   int sdisp[size];
107
108   sdisp[0] = 0;
109   for (i = 0; i < size - 1; i++){
110     sdisp[i+1] = sdisp[i] + array_counter[i];
111     printf("Rank %d: send displace %d \n",rank,sdisp[i+1]);
112   }
113
114   MPI_Alltoall(array_counter,1,MPI_INT,recvbuf,1,MPI_INT,MPI_COMM_WORLD);
115
116   MPI_Barrier(MPI_COMM_WORLD);
117
118   int sum = 0;
119   for (i = 0; i < size; i++){
120     sum += recvbuf[i];
121     printf("Rank %d: recvbuf %d \n",rank,recvbuf[i]);
122   }
123
124   printf("Rank %d: total recv buf %d \n", rank,sum);
125
126   MPI_Barrier(MPI_COMM_WORLD);
127
128   rdisp[0] = 0;
129   for (i = 0; i < size - 1; i++){
130     rdisp[i+1] = rdisp[i] + recvbuf[i];
131     printf("Rank %d: recv displace %d \n",rank,rdisp[i+1]);
132   }
133
134   int local_array_alltoall[sum];
135   // initialize local_array_alltoall for testing purpose
136   for (i = 0; i < sum; i++) local_array_alltoall[i] = -1;
137   MPI_Alltoallv(local_array,array_counter,sdisp,MPI_INT,local_array_alltoall,recvbuf,rdisp,MPI_INT,MPI_COMM_WORLD);
138
139   printf("Rank %d: semi-sorted num: ",rank);
140   for (i = 0; i < sum; i++){
141     printf("%d ", local_array_alltoall[i]);
142   }
143   printf("\n");
```

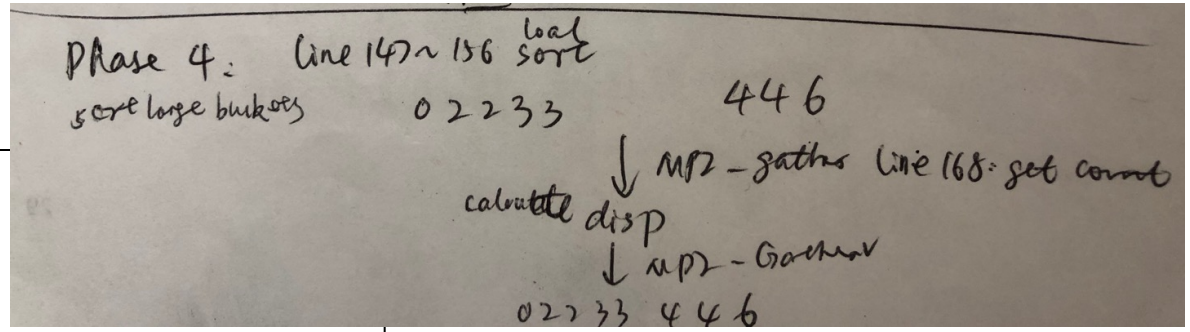



```
Rank 0: send displace 3
Rank 0: recvbuf 3
Rank 0: recvbuf 2
Rank 0: total recv buf 5
Rank 0: recv displace 3
Rank 0: semi-sorted num: 2 3 3 0 2


Rank 1: send displace 2
Rank 1: recvbuf 1
Rank 1: recvbuf 2
Rank 1: total recv buf 3
Rank 1: recv displace 1
Rank 1: semi-sorted num: 6 4 4
```

# bucket2.c (Cont.)

## Phase 4: Sort large buckets




```
Rank 0: sorted num: 0 2 2 3 3
Rank 1: sorted num: 4 4 6



Before sort:
2 3 3 6 4 2 4 0
After sort:
0 2 2 3 3 4 4 6
```

```
145 /* Phase 4: Sort large buckets */
146
147   // local sort on big bucket one more time
148   for (i = 0; i < sum; i++){
149     for (j = i; j < sum; j++){
150       if (local_array_alltoall[i] > local_array_alltoall[j]){
151         tmp = local_array_alltoall[i];
152         local_array_alltoall[i] = local_array_alltoall[j];
153         local_array_alltoall[j] = tmp;
154       }
155     }
156   }
157
158   printf("Rank %d: sorted num: ", rank);
159   for (i = 0; i < sum; i++){
160     printf("%d ", local_array_alltoall[i]);
161   }
162   printf("\n");
163
164   // preparation for the final gathering
165   int proc_count[size];
166   int disp[size];
167
168   MPI_Gather(&sum,1,MPI_INT,proc_count,1,MPI_INT,0,MPI_COMM_WORLD);
169
170   if (rank == 0){
171     disp[0] = 0;
172     for (i = 0; i < size-1; i++){
173       disp[i+1] = disp[i] + proc_count[i];
174     }
175   }
176
177   MPI_Gatherv(local_array_alltoall,sum,MPI_INT,sortNum,proc_count,disp,MPI_INT,0,MPI_COMM_WORLD);
178
179   if (rank == 0){
180     printf("Before sort: \n");
181     for (i = 0; i < N; i++) printf("%d ",rawNum[i]);
182     printf("\nAfter sort: \n");
183     for (i = 0; i < N; i++) printf("%d ",sortNum[i]);
184   }
```

# N-Body Problem

- Fundamental settings for most, if not all, of computational simulation problems:
  - Given a space
  - Given a group of entities whose activities are (often) bounded within this space
  - Given a set of equation that governs how these entities react to one another and to attributes of the containing space
  - Simulate how these reactions impact all entities and the entire space overtime

# Gravitational N-Body Problem

- Finding positions and movements of bodies in space subject to gravitational forces from other bodies, using Newtonian laws of physics

Gravitational force between two bodies of masses $m_a$ and $m_b$ is:

$$F = \frac{Gm_a m_b}{r^2}$$

$G$ is the gravitational constant and $r$ the distance between the bodies. Subject to forces, body accelerates according to Newton's 2nd law:

$$F = ma$$

$m$ is mass of the body, $F$ is force it experiences, and $a$ the resultant acceleration.

Let the time interval be $t$. For a body of mass $m$, the force is:

$$F = \frac{m(v^{t+1} - v^t)}{\Delta t}$$

New velocity is:

$$v^{t+1} = v^t + \frac{F\Delta t}{m}$$

where $v^{t+1}$ is the velocity at time $t + 1$ and $v^t$ is the velocity at time $t$.

Over time interval $\Delta t$, position changes by

$$x^{t+1} - x^t = v\Delta t$$

where $x^t$ is its position at time $t$.

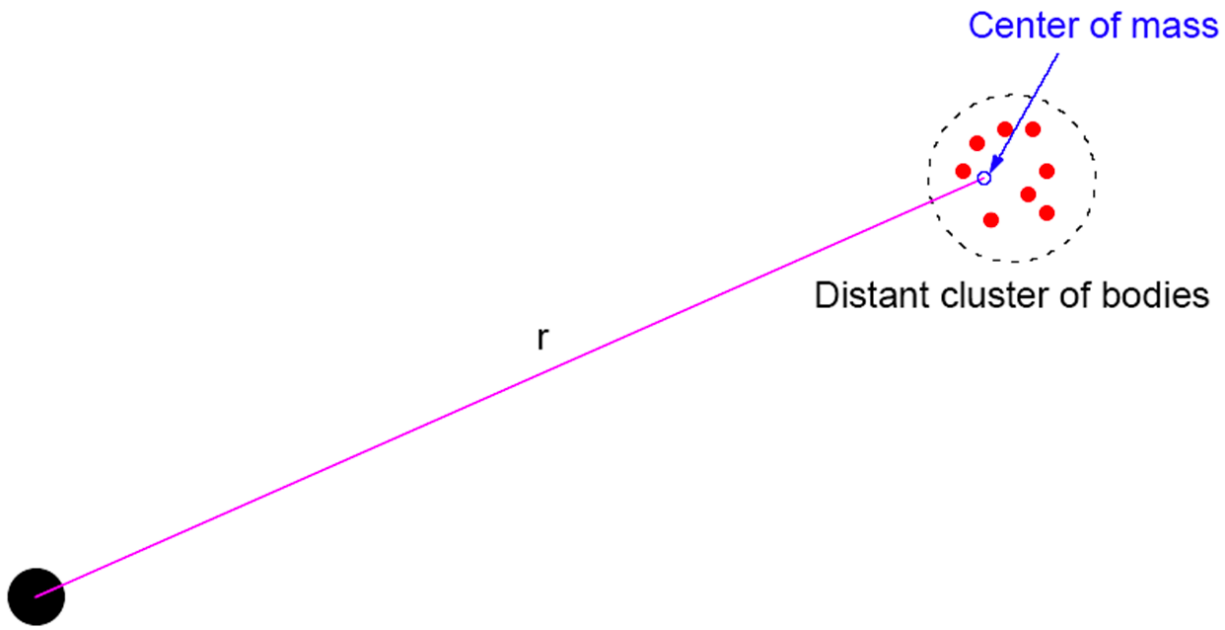
# Sequential Code

Overall gravitational *N*-body computation can be described by:

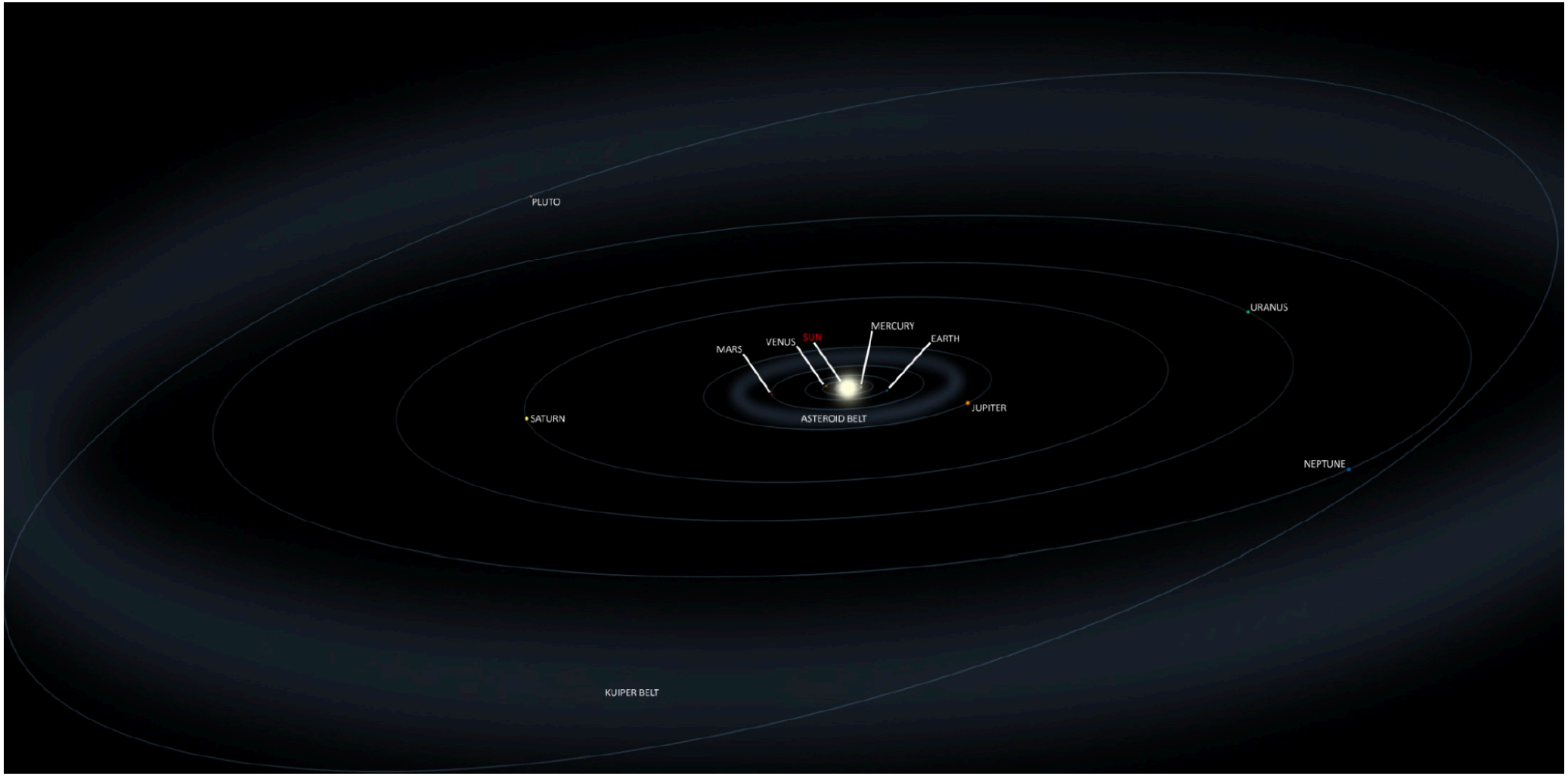
```
for (t = 0; t < tmax; t++)          /* for each time period */
    for (i = 0; i < N; i++) {         /* for each body */
        F = Force_routine(i);                  /* compute force on ith body */
      v[i]new = v[i] + F * dt / m;     /* compute new velocity */
       x[i]new = x[i] + v[i]new * dt;   /* and new position */
}
for (i = 0; i < nmax; i++) {        /* for each body */
    x[i] = x[i]new;                   /* update velocity & position*/
    v[i] = v[i]new;
}
```

# Parallel Code

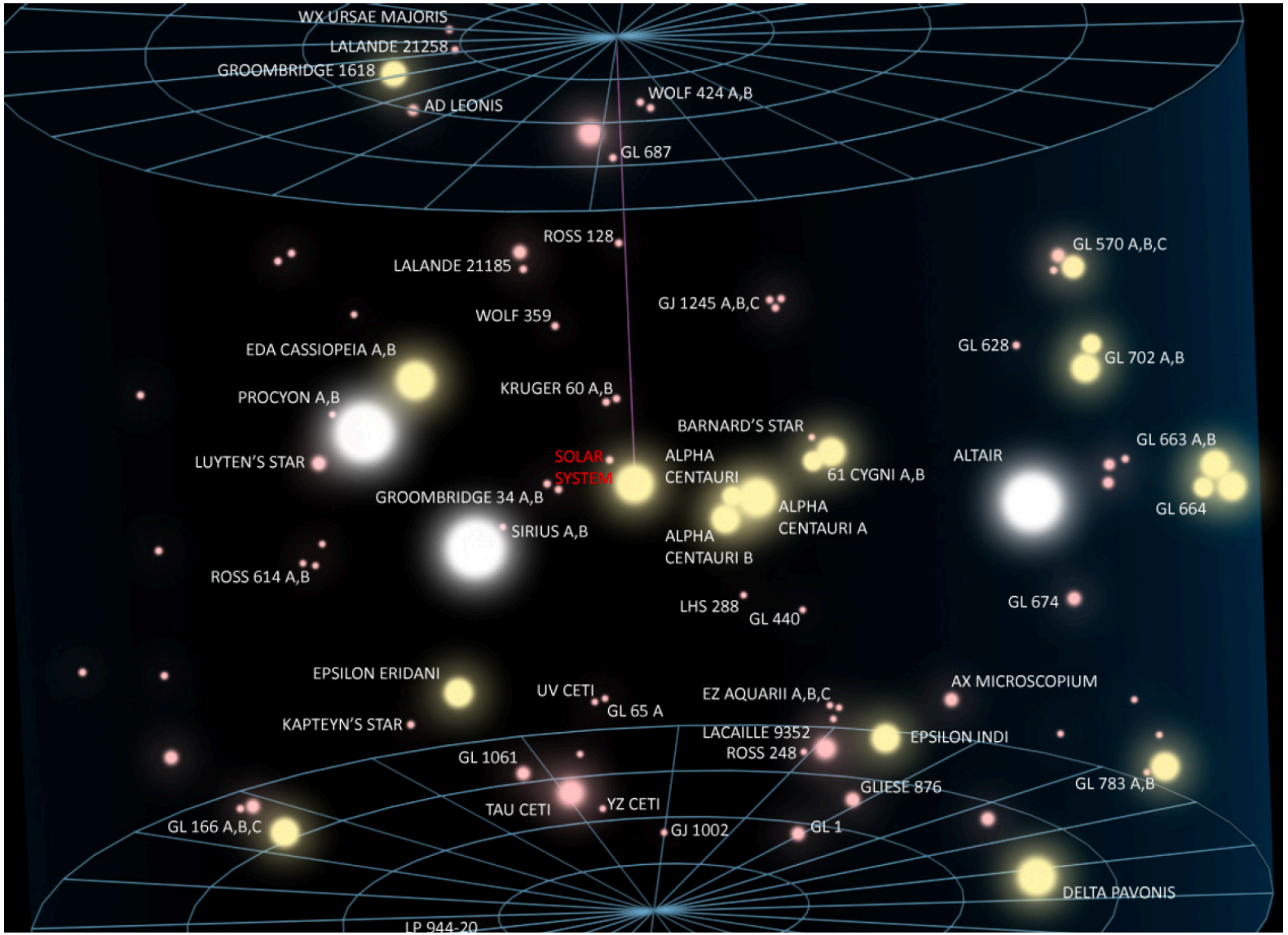
- Time complexity can be reduced approximating a cluster of distant bodies as a single distant body with mass sited at the center of mass of the cluster:



# Parallel Code

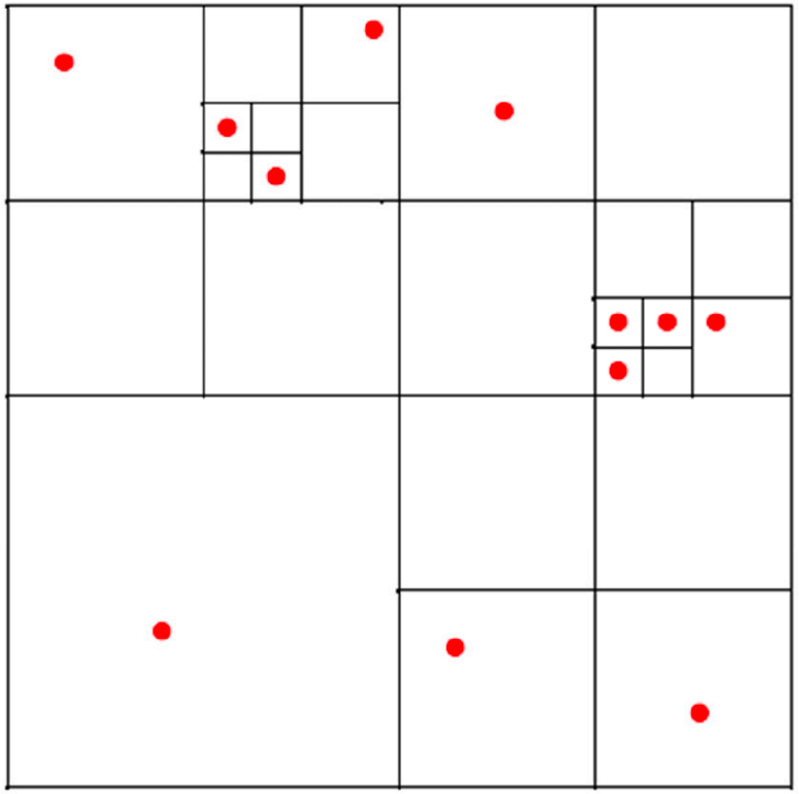


# Parallel Code



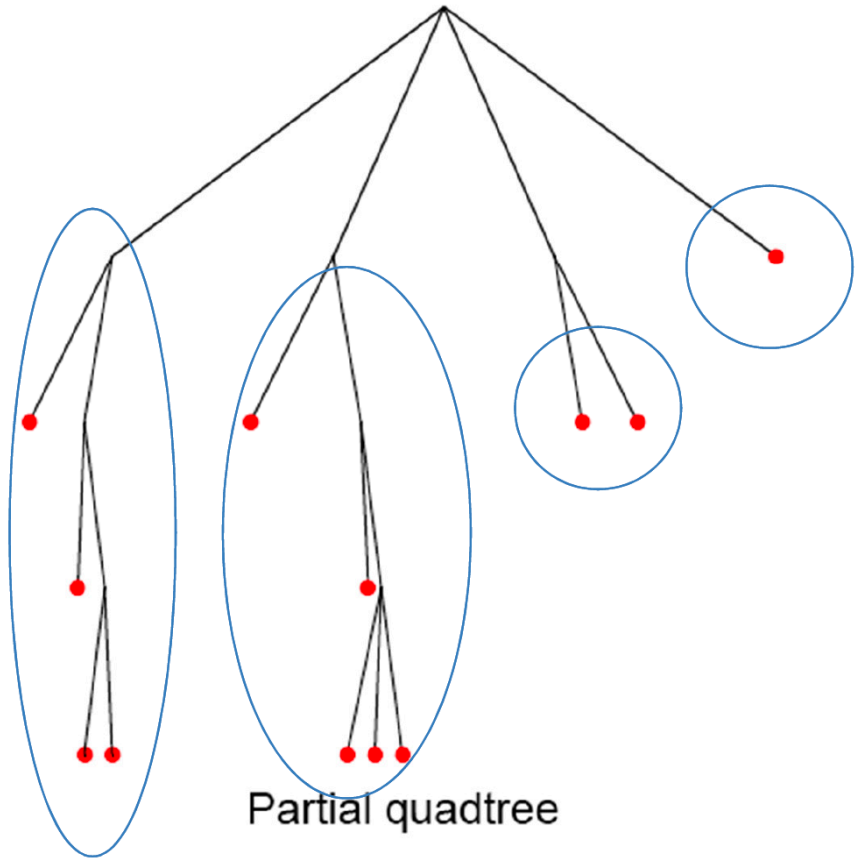
# Barnes-Hut Algorithm (2-D)

- Start with whole region in which one square contains the bodies (or particles)
  - First, this cube is divided into four subregions
  - If a subregion contains no particles, it is deleted from further consideration
  - If a subregion contains one body, it is retained
  - If a subregion contains more than one body, it is recursively divided until every subregion contains one body

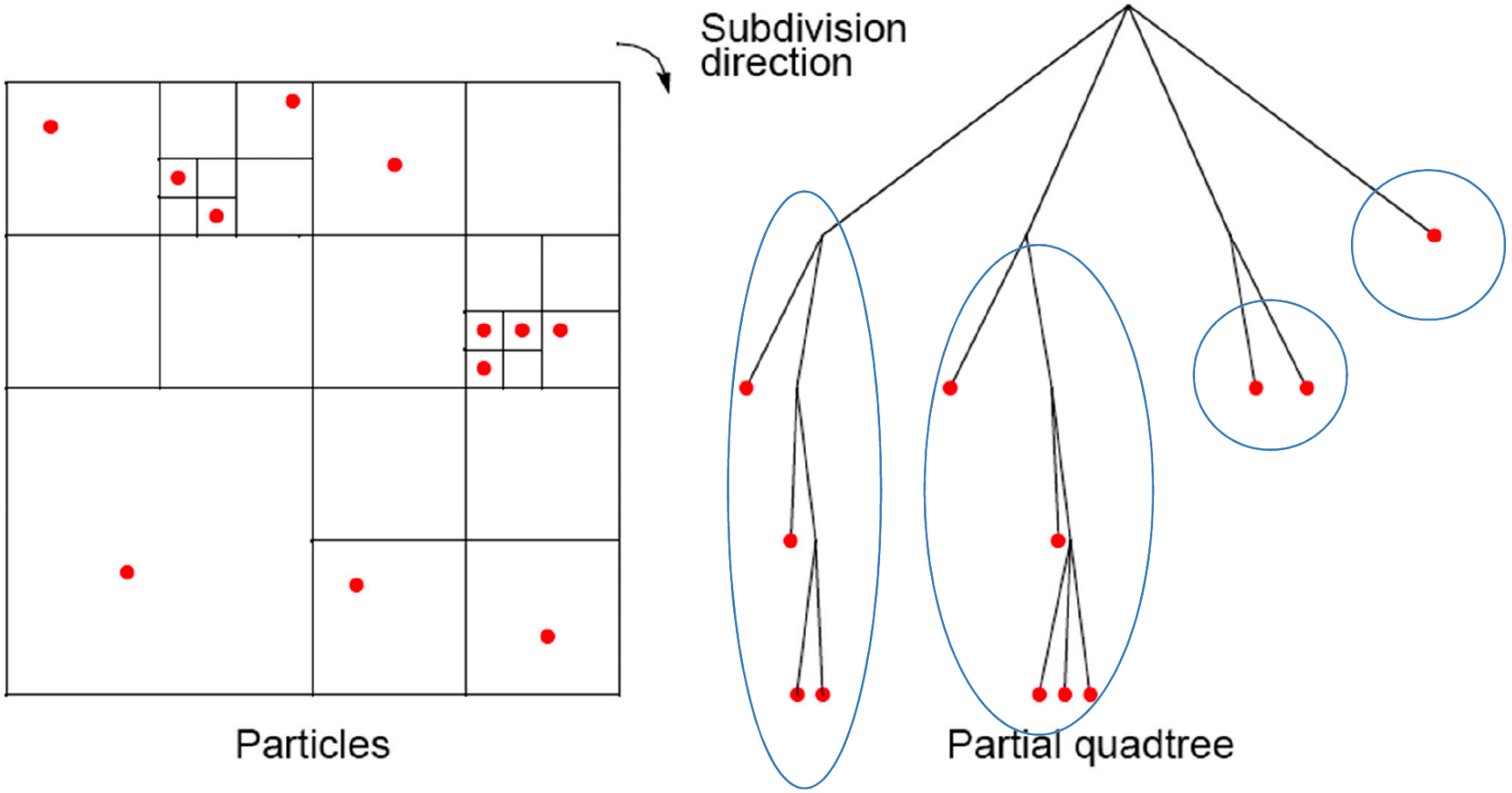


# Construct a Quadtree

- Create a quadtree – a tree with up to four edges from each node
- The leaves represent cells each containing one body
- After the tree has been constructed, the total mass and center of mass of the subregion is stored at each node



# Recursive Division of 2-D Space



# Orthogonal Recursive Bisection

- First, a vertical line found that divides area into two areas each with equal number of bodies
- For each area, a horizontal line found that divides it into two areas, each with equal number of bodies
- Repeated as required