Lecture 16

Parallel Sorting
MPI Datatypes

Today's lecture

- MPI Derived Datatypes
- Parallel Sorting

MPI Datatypes

Data types

- MPI messages sources need not be contiguous 1-dimensional arrays
- The element types need to be restricted to built in types like `float`, `int`, `char`
- MPI provides a data type mechanism to enable us to work with a richer set of types
  - structs
  - "every k^{th} element of a vector"

Data types in MPI

- MPI encodes the meaning of user-defined types with a special set of functions
- The type system is limited
- No support for
  - pointer-based data structures
  - callbacks

The basics of MPI data types

- ***Create*** an **MPI_Datatype** object

```
MPI_Datatype new_type_t
MPI_Type_vector(nblks, blkLen, stride, elt_t,
                &new_type_t)
```

- ***Commit*** the data type, allowing MPI to take some internal actions that may improve performance

```
MPI_Type_commit(&new_type_t)
```

- ***Communicate*** the data using the committed type

```
MPI_Send(ptr, n, new_type_t, dest, tag, comm)
```

Derived types

- MPI provides derived data types, e.g. **struct**
- We need to describe
  - The number of elements in the **struct**
  - The type of each element
- From this information we can determine the displacement from the start of the **struct**, where each element begins
  $\{(\tau_0, d_0), (\tau_1, d_1), \ldots, (\tau_{n-1}, d_{n-1})\}$
- Members may be built-in or previously defined MPI types, but not pointers

An example of a derived datatype

- Consider

  struct x { float a; float b; int n; }

- There are three members
  - The first member (a) has type MPI_FLOAT
  - The second member (b) has type MPI_FLOAT
  - The third member (c) has type MPI_INT
- There are two block
  - The first block contains 2 * MPI_FLOAT
  - The second block contains 1 * MPI_INT

The API

```
MPI_Type_struct( int           count,
                 int           block_lengths[ ],
                 MPI_Aint      displacements[ ],
                 MPI_Datatype  typelist[ ],
                 MPI_Datatype* new_mpi_t);
```

count: number of members in the struct

block_lengths[ ]: number of entries in each member

displacements[ ]: byte displacement of each block

typelist[ ]: type of each member

Building a struct type

```
MPI_Type_struct( int              count,
                 int              block_lengths[ ],
                 MPI_Aint         displacements[ ],
                 MPI_Datatype     typelist[ ],
                 MPI_Datatype*    new_mpi_t);
```

Consider **struct x { float a; float b; int n; }**

Count = 2

block_lengths[0] = { 2 , 1 }

typelist[ ] = {**MPI_FLOAT**, **MPI_INT**}

displacements[0] = 0

displacements [1] = ?

Computing the displacements

- Since component types may be previously defined types, we need to have a way of computing a block's size

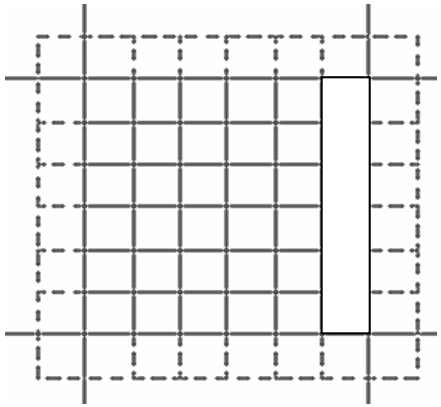
```
int MPI_Type_extent(
   MPI_Datatype datatype, MPI_Aint *extent )

typedef struct { float a; float b; int n} Ts;

MPI_Type_extent(MPI_FLOAT,&extent);
 displacements[0] = 0
 displacements[1] =  2 * extent;
```

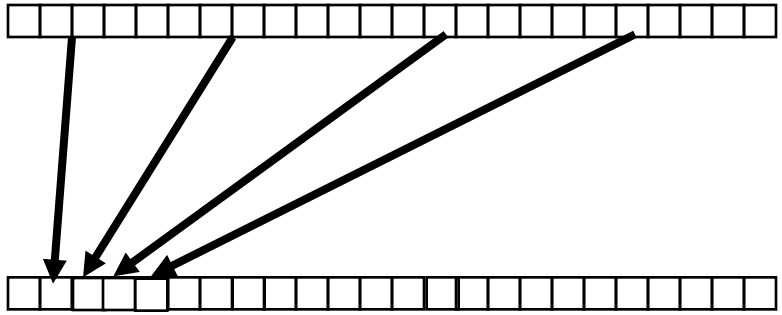
An alternative to types

- We can copy the data into a buffer
- Packing a heterogeneous struct ourselves can lead to surprises if we are moving across machine boundaries

```
struct {int x; float y; double z;}
```

- MPI provides functions to support more elaborate types, and to support message packing and unpacking, but won't discuss these

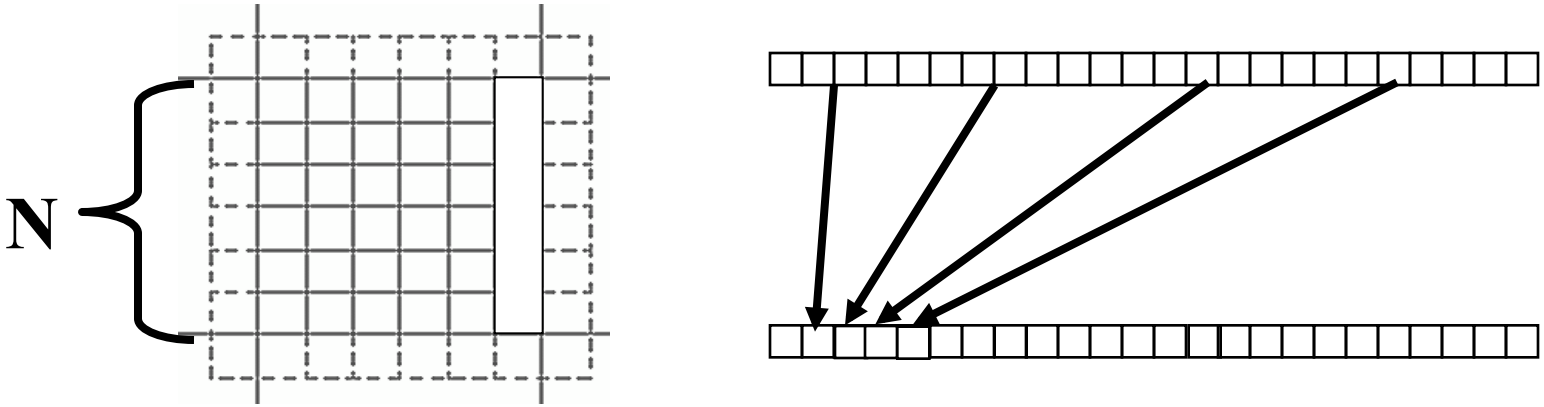




Specifying a Vector type

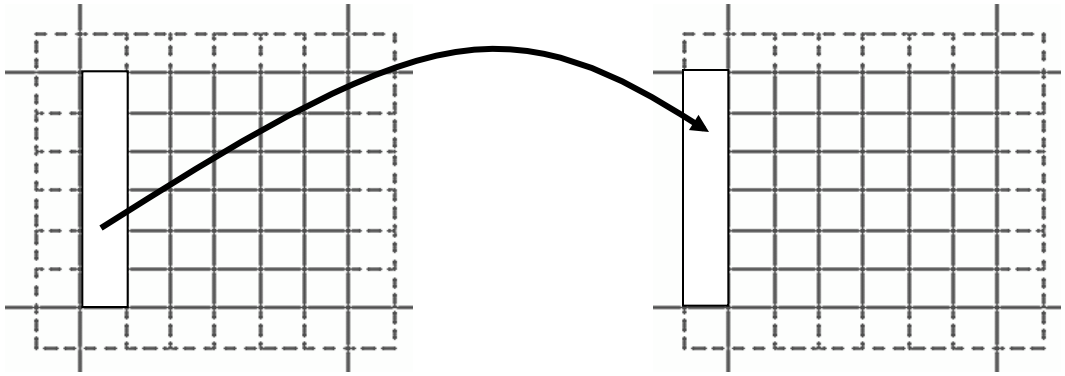
- The addresses in a column of a 2D array are not contiguous

```
MPI_Type_vector(
    int count=N,int blockLen=1,
    int stride=N,
    MPI_Datatype  MPI_INT, MPI_Datatype &vec_t);
```



Copying a column

```
if (myid ==0)
    MPI_Send (&A[0][1], 1,vec_t,1,0,MPI_COMM_WORLD);
else
    MPI_Recv( &A,
                  1,
                  vec_t,
                  0,
                  0,
                  MPI_COMM_WORLD,
                  &status);
```



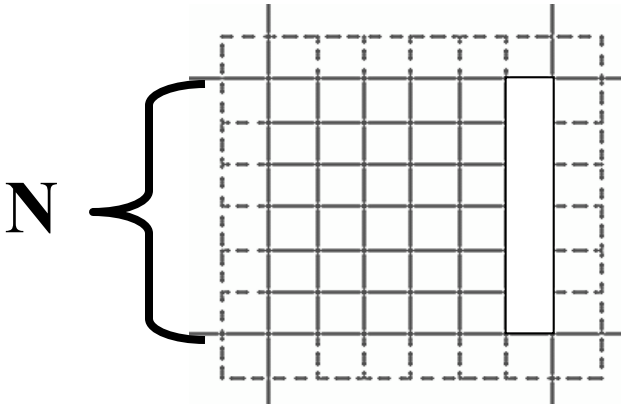
Structural equivalence

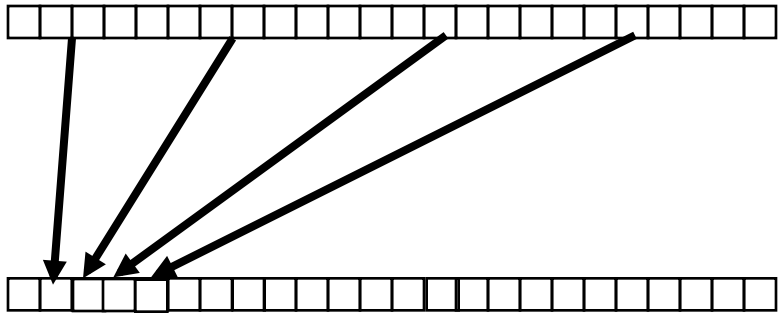
- The recipient has flexibility to store incoming values according to a locally defined rule
- Process A can send a block of data in row major order to Process B, which can receive the data in column major order into a local data structure
- The only constraint is that that the number of sent and received elements are the same

Receiving data contiguously

- Takes advantage of structural equivalence

```
If (myid ==0)
    MPI_Send(A,1,vec_t,1,0,MPI_COMM_WORLD);
else
    MPI_Recv(A,N,MPI_INT,0,0,MPI_COMM_WORLD,
                &status);
```





An example using structural equivalence

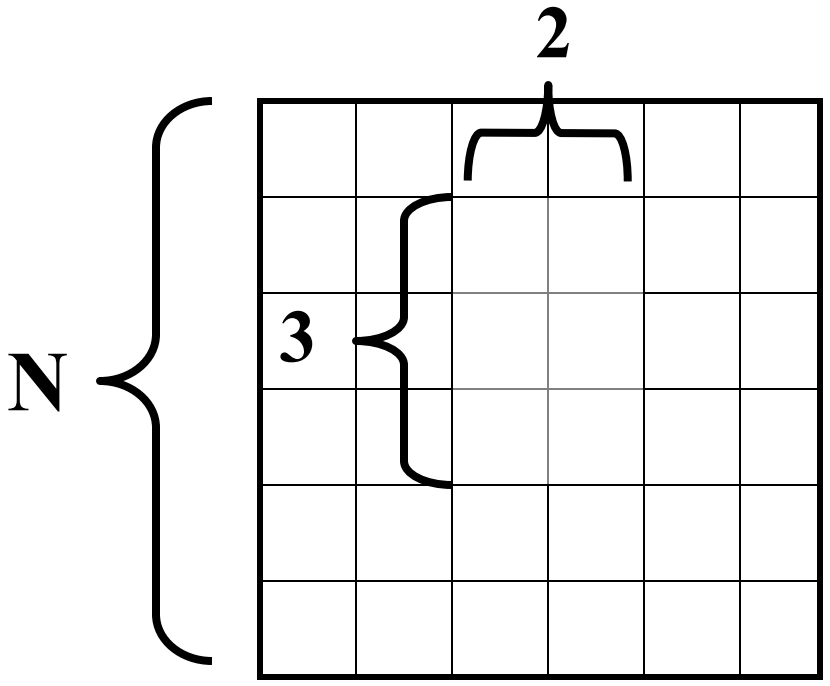
- Copy a horizontal plane of a 3D array into a 2D buffer

```
MPI_Type_vector(n, n, n*n, MPI_INT, &horiz_t);
…

MPI_Send(buff, 1,     horiz_t,  dest, tag, comm)

MPI_Recv(buff,  n*n, MPI_INT, src, t ag,
                comm, &status);
```

A more elaborate example

Transmit a 3 x 2 subblock of an N x N array

```
MPI_Type_vector(
   int count=2,
   int blockLen=3,
   int stride=N,
   MPI_Datatype
   MPI_DOUBLE,
   MPI_Datatype
   &vec_t);
```



An example datatype problem

- Reminiscent of block cyclic distributions
- Let's collect a block of 2 elements, that skips 4 elements between each block

```
MPI_Type_vector(N,          // blocks
                2,          // blockLen
                6,          // stride
                MPI_INT,
                &horiz_t);
```

Another example datatype problem

- What does this sequence of calls perform?

```
MPI_Type_vector(n,                  // blocks
                n,                  // blockLen
                n*n,                // stride
                MPI_INT,  &horiz_t);
…

MPI_Send(buff, 1,     horiz_t,  dest, tag, comm)
MPI_Recv(buff,  n*n, MPI_INT, src, t ag,
                comm, &status);

MPI_Type_vector(
   int count, int blockLen,,int stride,
   MPI_Datatype  TYPE,
   MPI_Datatype &my_type);
```

Parallel Sorting

- Sorting is fundamental algorithm in data processing
  - Given an unordered set of keys $x_0, x_1, \ldots, x_{N-1}$
  - Return the keys in sorted order
- The keys may be strings, numbers, or any object for which the relations >, <, and = hold
- We'll consider in-memory sorting of integer keys
  - Bucket sort
  - Sample sort
  - Bitonic sort
- In practice, we sort on external media, i.e. disk
  - See: http://sortbenchmark.org
  - **100 Terabytes in 173 minutes** (O'Malley and Murthy)
    3452 x (2 Quadcore Xeons, 8 GB memory, 4 SATA)

Rank sorting

- Compute the rank of each input value
- Move each value in sorted position according to its rank
- Not practical on real hardware

```
forall  i=0:n-1, j=0:n-1
    if ( x[i] > x[j] ) then rank[i] += 1 end if
forall  i=0:n-1
    y[rank[i]] = x[i]
```

In search of a fast and practical sort

- Rank sorting is impractical on real hardware
- Let's borrow the concept: compute the processor owner for each key
- Communicate data in sorted order in one step
- But how do we know which processor is the owner?
- Depends on the distribution of keys

Bucket sort

- Divide key space into equal subranges and associate a bucket with each subrange
- Unsorted input data distributed evenly over processors
- Each processor maintains p local buckets
  - Assigns each key to a local bucket: $\lfloor p \times key/(K_{max}-1) \rfloor$
  - Routes the buckets to the correct owner (each local bucket has ~ n/p^2 elements)
  - Sorts all incoming data into a single bucket



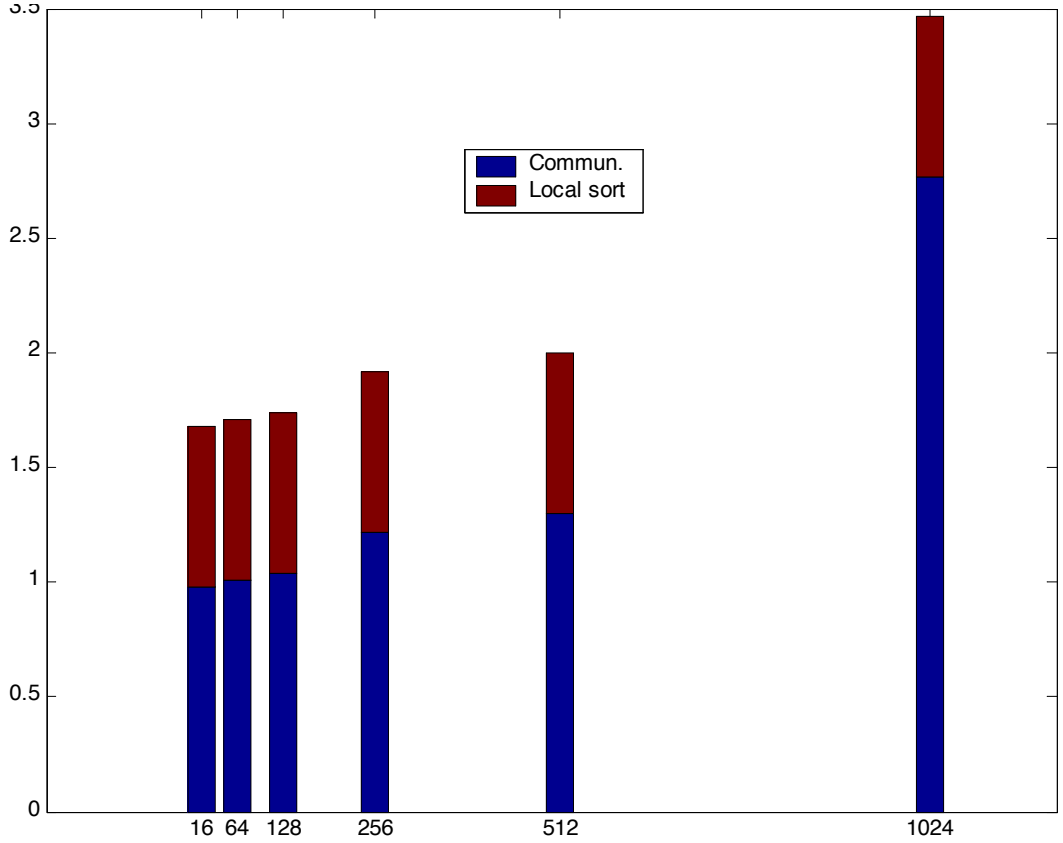
Wikipedia

Running time

- Assume that the keys are distributed uniformly over 0 to K_{max}-1
- Local bucket assignment: O(n/p)
- Route each local bucket to the correct owner
  All to all: O(n)
- Local sorting : O(n/p)
  - Radix sort
  - www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Radix

Scaling study

- IBM SP3 system: 16-way servers w/ Power 3 CPUs
- Weak scaling : 1M points per processor



Local sort: quicksort
O(n/p log(n/p))

All-to-allv
O(n)

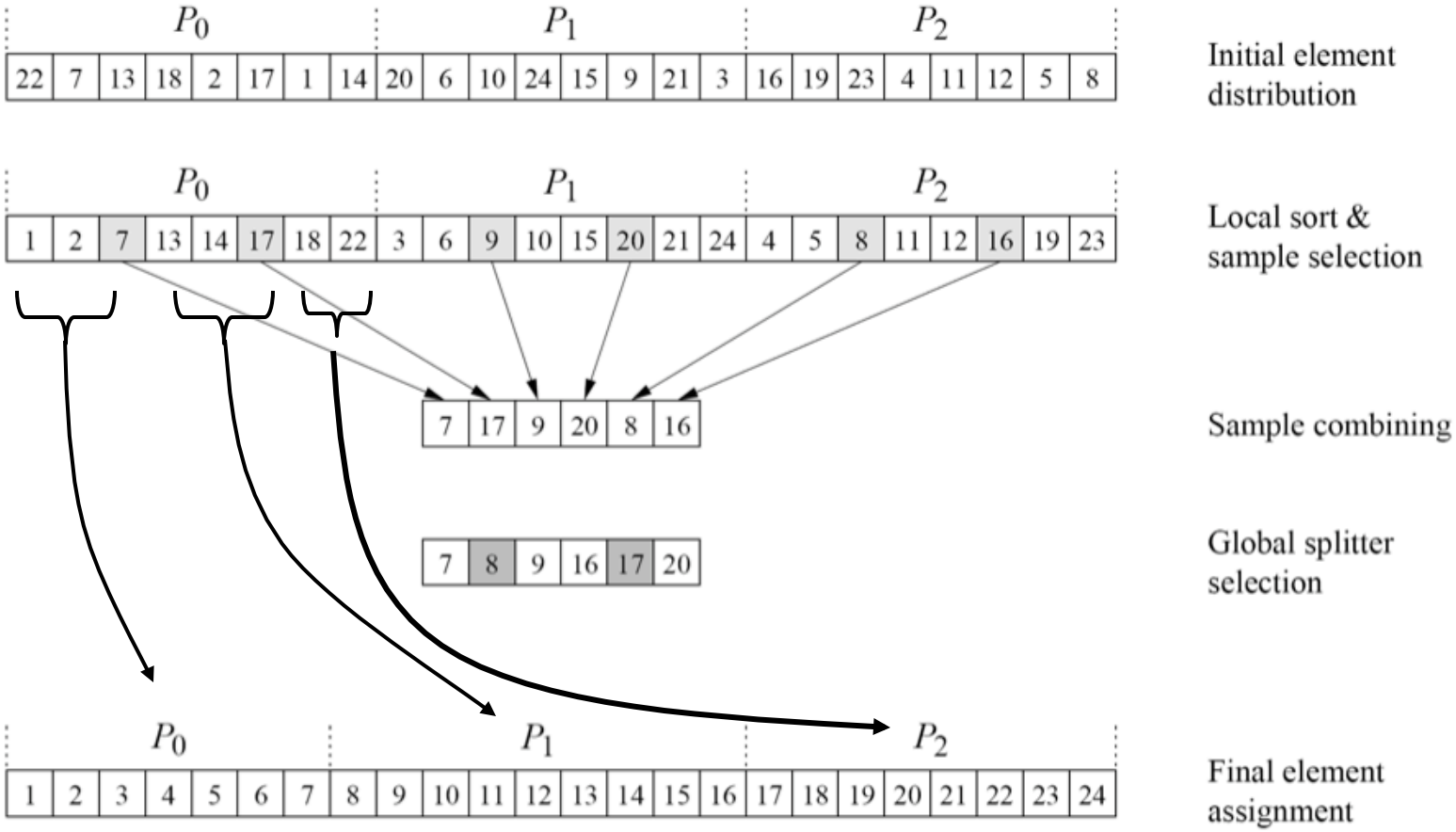
Worst case behavior

- What is the worst case?
- Mapping of keys to processors based on knowledge of K_{max}
- If keys are in range [0,Q-1] …
  … processor *k* has keys in the range [k*Q/P : (k+1)*Q/P]
- For $Q=2^{30}$, P=64, each processor gets $2^{24} = 16$ M elements
- What if keys $\in [0, 2^{24}-1] \subset [0, 2^{30}-1]$?
- For a non-uniform distribution, we need more information to balance keys (and communication) over the processors
- Sample sort is an algorithm that collects such informatio and improves worst case behavior

The idea behind sample sort

- Estimate the distribution of the global key range over the *p* processors
- Sample the keys to determine a set of *p-1* **splitters** that partition the key space into *p* disjoint intervals
  [sample size parameter: s]
- Each interval is assigned a unique processor
- Once each processor knows the splitters, it can distribute its keys to the others accordingly
- Processors sort incoming keys

Alltoallv used in sample sort



Introduction to Parallel Computing, 2nd Ed,, A.Grama, A.l Gupta, G. Karypis, and V. Kumar, Addison-Wesley, 2003.

Performance

- Assuming $n \geq p^3$ …
- $\mathbf{T_P = O((n/p) \lg n)}$
- If $\boldsymbol{s = p}$, each processor will will merge not more than
  $\boldsymbol{2n/p + n/s - p}$ elements
- If s > p, each processor will will merge not more than
  $\boldsymbol{(3/2)(n/p) - (n/(ps)) + 1 + d}$ elements
- Duplicates ***d*** do not impact performance unless $\boldsymbol{d = O(n/p)}$
- Tradeoff: increasing ***s*** …
  - Spreads the final distribution more evenly over the processors
  - Increases the cost of determining the splitters