```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "NestStep.h"

/** p-way randomized Combine-CRCW BSP-Quicksort
 *  variant by Gerbessiotis/Valiant JPDC 22(1994)
 *  implemented in NestStep-C.
 */
int N=10;               // default value

/** findloc(): find largest index j in [0..n-1] with
 *  a[j]<=key<a[j+1] (where a[n] = +infty).    C's
 *  bsearch() can't be used, it requires a[j]==key.
 */
int findloc( void *key, void *a, int n, int size,
             int (*cmp)(const void *, const void *) )
{
 int j;    // for the first, use naive linear search:
 for (j=0; j<n; j++)
   if ( cmp( key, a+j*size ) <= 0)  return j;
 return n-1;
}

int flcmp( const void *a, const void *b ) // compare
{                                          // 2 floats
 if (*(float *)a < *(float *)b) return -1;
 if (*(float *)a > *(float *)b) return 1;
 return 0;
}

int main( int argc, char *argv[] )
{
  // shared variables:
  sh float *A</>, *B</>;   // block-wise distr. arrays
  sh float *size, *pivs;   // replicated shared arrays
  // private variables such as T used as destination of
  // remote read are automatically treated in a special
  // way (wrapper objects) by the compiler:
  float *T, **myslice;
  int *mysize, *Ssize, *prefsize;
  // local variables:
  int i, j, p, ndp, nmp, myndp, pivi, psumsize;
  double startwtime, endwtime;

  NestStep_init( &argc, &argv );   // sets # and $
  // N may be passed as a parameter:
  if (argc > 1) sscanf( argv[1], "%d", &N );
  p = #;
  ndp = N / p;
  nmp = N % p;
  if ($ < nmp) myndp = ndp+1;
  else         myndp = ndp;
  A = new_DArray( N, Type_float ); // dyn.alloc. A[0:N-1]
  B = new_DArray( N, Type_float ); // dyn.alloc. B[0:N-1]
  mysize = new_Array( p, Type_int );
  Ssize = new_Array( p, Type_int );
  size = new_Array( p, Type_int );
  prefsize = new_Array( p, Type_int );
  pivs = new_Array( p, Type_int );

  forall (i, A)      // useful macro supplied by NestStep
     A[i]=(double)(rand()%1000000); // create some values
                                    //       to be sorted
  myndp = DArray_Length(A); // size of my local partition

  startwtime=NestStep_time(); //here starts the algorithm

  // STAGE 1: each processor randomly selects a pivot el.
  //          and writes (EREW) it to pivs[$].
  // Pivot intervals are defined as follows:
  // I(P_0)     = ] -\infty, pivs[0] ]
  // I(P_i)     = ] pivs[i-1], pivs[i] ],  0<i<p-2
  // I(P_{p-1}) = ] pivs[p-2], \infty [

  step { /*1*/
    int pivi = DArray_index( rand()%myndp );
    pivs[$] = A[pivi];  // random index in my partition
  }
  // now the pivots are in pivs[0..p-1]

  // STAGES 2, 3, 4 form a single superstep:

  step { /*2*/
   // STAGE 2: locally sort pivots. This computation is
   // replicated, thus not work-optimal, but faster than
   // sorting on one processor only and communicating
   // the result to the others.
```

```
   qsort( pivs, p, sizeof(float), flcmp );

   // STAGE 3: local p-way partitioning in local arrays
   // A->array. I have no in-place algorithm, so I over-
   // allocate O(p*ndp) space, only npd elements of
   // which will be actually filled by the partitioning:

   myslice = (float **)malloc( p * sizeof(float *));
   myslice[0] = (float *)malloc( p * ndp *sizeof(float));
   for (i=1; i<p; i++) myslice[i] = myslice[0] + i*ndp;
   for (i=0; i<p; i++) size[i] = 0;
   forall (i, A) {  // insert my own elements in slices:
     j=findloc( &(A[i]), pivs, p, sizeof(float),flcmp);
     myslice[j][size[j]++] = A[i];
   }
   // keep the local sizes for later:
   for (i=0; i<p; i++) mysize[i] = size[i];

   // STAGE 4: Globally compute the array size of global
   // slice sizes and the prefixes of the individual
   // processor contributions. This is just done as a
   // side-effect of combining:

  } combine ( size[:]<+:prefsize[:]> ); //end superstep 2

  // This array ^^^^ notation in the optional combine
  // annotation is a hint for the compiler that the
  // shared array "size" needs not be packed elementwise,
  // thus avoiding smaller combine items.   Here, the
  // effect is marginal, as "size" is usually small.

  // After combining, size[i] holds the global size
  // of pivot interval I(i), and prefsize[i] holds the
  // accumulated length of contributions of the
  // processors P0..P{$-1} from pivot interval I(i),
  // that is, the prefix sum of all size[i] contribu-
  // tions made by processors 0...$-1.

  // STAGE 5: Write the local slices to the
  // corresponding parts in array B.

  step { /*3*/
   psumsize = 0;
   for (i=0; i<p; i++) {
    // remote write to distributed shared array B:
    B[ psumsize + prefsize[i] :
       psumsize + prefsize[i] + mysize[i] - 1 ]
      = myslice[i][ 0: mysize[i]-1 ];
    Ssize[i] = psumsize; //prefixsum size[0]+...+size[i-1]
    psumsize += size[i]; //     by replicated computation
   }
#pragma mirror( B [ Ssize[$] : Ssize[$]+size[$]-1 ] )
  } // combining includes the updates to B and prefetches

  // STAGE 6:  Allocate space for all elements in my pivot
  // interval I($), and bulk-read the myslice slices
  // from the partitioned array.

  step { /*4*/
    T = new_Array( size[$], Type_float );
    // remote read from distributed array B:
    T[ 0 : size[$]-1 ]
      = B [ Ssize[$] : Ssize[$]+size[$]-1 ];
  }

  // Stages 7 + 8 can be combined into a single superstep,
  // because Stage 7 contains only local computation.

  step { /*5*/
   // STAGE 7: Now each processor sorts its array T.

   qsort( T, /* this is recognized as a NestStep
              * array by the compiler */
          size[$], sizeof(float), flcmp );

   // STAGE 8: Now write T back to B.
   // bulk remote write to B:
   B [ Ssize[$] : Ssize[$]+size[$]-1 ]
     = T [ 0 : size[$] - 1 ]
  }

  endwtime = NestStep_Wtime();

  ... // print distributed result array and time.
  NestStep_finalize();
} // main
```

Figure 3: NestStep implementation of BSP p-way Quicksort