Table 7.8 Run-times (in seconds) of CUDA basic and CUDA shared memory n-body solvers.

Blocks	Particles	Basic	Shared Mem	Speedup
1	1024	1.93e-3	1.59e-3	1.21
2	2048	2.89e-3	2.17e-3	1.33
32	32,768	7.34e-2	4.17e-2	1.76
64	65,536	2.94e-1	1.98e-1	1.48
256	262,144	2.24e+0	1.96e+0	1.14
1024	1,048,576	4.81e+1	2.91e+1	1.65

The actual performance of the basic and shared-memory solvers is shown in Table 7.8. We ran the shared-memory solver on the same system that we ran the basic solver on: an Nvidia Pascal-based system (CUDA capability 6.1) with 20 SMs, 128 SPs per SM, and a 1.73 GHz clock. Also, as with the basic solver, we always used 1024 threads per block. The shared memory implementation is always a good deal faster than the basic implementation, and on average, the speedup is 1.43.

7.2 Sample sort

7.2.1 Sample sort and bucket sort

Sample sort is a generalization of an algorithm known as *bucket sort*. In bucket sort, we're given two values $c < d$, and the keys are assumed to have a uniform distribution over the interval $[c, d)$. This means that if we take two subintervals $[p, q)$ and $[r, s)$ of the same length, then the number of keys in $[p, q)$ is approximately equal to the number of keys in $[r, s)$. So suppose we have n keys (often **float**s or **int**s), and we divide $[c, d)$ into b "buckets" or "bins" of equal length. Finally suppose that $h = (d - c)/b$. Then there should be approximately n/b keys in each of the buckets:

$$[c, c + h), [c + h, c + 2h), \ldots, [c + (b - 1)h, b).$$

In bucket sort we first assign each key to its bucket, and then sort the keys in each bucket.

For example, suppose $c = 1$, $d = 10$, the keys are

$$9, 8, 2, 1, 4, 6, 3, 7, 2,$$

and we're supposed to use 3 buckets. Then $h = (10 - 1)/3 = 3$, and the buckets are

$$[1, 4),\ [4, 7),\ [7, 10).$$

So the contents of the three buckets should be

$$\{2, 1, 3, 2\},\ \{4, 6\},\ \text{and}\ \{9, 8, 7\},$$

respectively.

In sample sort, we don't know the distribution of the keys. So we take a "sample" and use this sample to estimate the distribution. The algorithm then proceeds in the same way as bucket sort. For example, suppose we have b buckets and we choose our sample by picking s keys from the n keys. Here $b \leq s \leq n$. Suppose that after sorting the sample, its keys are

$$a_0 \leq a_1 \leq \ldots \leq a_{s-1},$$

and we choose the keys in slots that are separated by s/b sample elements as "splitters." If the splitters are

$$e_1, e_2, \ldots, e_{b-1}$$

then we can use

$$[\min, e_1), [e_1, e_2), \ldots, [e_{b-1}, \max)$$

as our buckets.[3] Here min is any value less than or equal to the smallest element in the list, and max is a value greater than the largest element in the list.

For example, suppose that after being sorted our sample consists of the following 12 keys:

$$2, 10, 23, 25, 29, 38, 40, 49, 53, 60, 62, 67.$$

Also suppose we want $b = 4$ buckets. Then the first three keys in the sample should go in the first bucket, the next three into the second bucket, etc. So we can choose as the "splitters"

$$e_1 = (23 + 25)/2 = 24,\ e_2 = (38 + 40)/2 = 39,\ e_3 = (53 + 60)/2 = 57.$$

Note that we're using the convention that if the splitter is not a whole number, we round up.

After determining the splitters, we assign each of the keys in the list to one of the buckets

$$[\min, 24), [24, 39), [39, 57), [57, \max).$$

Thus, we can use the pseudocode in Program 7.4 as the basis for an implementation of serial sample sort. Note that we're distinguishing between "sublists" of the input list and "buckets" in the sorted list. The sublists partition the input list into b sublists, each of which has n/b elements. The buckets partition the list into b subsets of elements with the property that all the keys in one bucket lie between consecutive splitters. So after each of the buckets has been sorted, we can concatenate them to get a sorted list.

[3] If there are many duplicates in the list, we may want to allow the buckets to be closed intervals, and if duplicate elements are equal to a splitter, we can divide the duplicates between a pair of consecutive buckets. We won't consider this possibility in the programs we develop.

Clearly, to convert this pseudocode into a working program we must implement the sample generation function and the mapping function and any associated data structures. There are many possibilities. Let's take a look at some of them.

```
1  /* Input:
2   *     list[n]: the input list
3   *           n: number of elements in list (evenly
4   *              divisible by s and b)
5   *           s: number of elements in the sample (evenly
6   *              divisible by b)
7   *           b: number of buckets
8   * Output:
9   *     list[n]: the original input data in sorted order
10  */
11
12    /* Take the sample */
13    Gen_sample(list, n, s, b, sample);
14
15    Sort(sample, s);
16
17    Find_splitters(sample, s, splitters, b);
18
19    /* Assign each element of each sublist to
20     * its destination bucket in mapping, a data
21     * structure that can store all n elements.
22     */
23    Map(list, n, splitters, b, mapping);
24
25    /* Sort each bucket and copy it back to the original list */
26    Sort_buckets(mapping, n, b, list);
```

Program 7.4: Pseudocode for a serial sample sort.

7.2.2 Choosing the sample

An obvious method for generating the sample is to use a random number generator. For example,

```
// Seed the random number generator
srandom(...);
chosen = EMPTY_SET;  // subscripts of chosen elements
for (i = 0; i < s; i++) {
   // Get a random number between 0 and n
   sub = random() % n;
   // Don't choose the same subscript twice
   while (Is_an_element(sub, chosen))
```

```
        sub = random() % n;
        // Add sub to chosen elements
        Add(chosen, sub);
        sample[i] = list[sub]
    }
```

The variable `chosen` is a set abstract data type that's used to store the subscripts of elements that have been added to the sample. The **while** loop is used to ensure that we don't choose the same subscript twice. For example, suppose `random() % n` is 5 when i = 2 and when i = 4. Then without the **while** loop the sample will include element `list`[5] twice. Checking whether `sub` has already been chosen can be expensive, but if the sample size, s, is small relative to n, the size of the list, then it's unlikely that we'll choose the same subscript twice, and we'll only need to check `chosen` once for each element in the sample.

An alternative takes a "deterministic" sample.

```
    for each sublist {
        Sort(sublist, n/b);

        Choose s/b equally spaced elements
            from the sorted sublist and add
            these to the sample;
    }
```

For example, suppose that after being sorted a sublist consists of the elements {21, 27, 49, 62, 86, 92} and we're choosing two elements from each sublist for the sample. Then we could choose elements 49 and 92 from this sublist. Even if the sample size is large, the main cost of this method will be the calls to `Sort`. So this method will work best if there is a large number of relatively small sublists.

7.2.3 A simple implementation of the `Map` function

Perhaps the simplest implementation of the `Map` function allocates b arrays, each capable of storing at least n/b elements—for example, $2n/b$ elements. This is the `mapping` data structure in the pseudocode in Program 7.4. These arrays will be the buckets. After allocating the buckets, we can simply iterate through the elements of `list` determining for each element which bucket it should be mapped to and copying the element to the appropriate bucket.

```
Allocate storage for b buckets;
for each element x of list {
    Search splitters for x's bucket;
    if (bucket is full)
        Allocate additional storage for
            bucket b;
    Append x to bucket b;
}
```

The catch is that, in general, a bucket may need to store more than the initially allocated number of elements, and we'll need to increase its size. We can do this using the C library function `realloc`:

```
void* realloc(void* ptr, size_t size);
```

For us, `ptr` will be one of the buckets, and `size` is the amount of storage, in bytes, that we want for the resized bucket. If there is insufficient space immediately following the memory that has already been allocated, `realloc` will try to find another block that is large enough and copy the contents of the old block into the new block. When this happens, the pointer to the new block is returned by the call. If this isn't necessary, `realloc` returns the old pointer. If there isn't enough memory on the heap for the new allocation, `realloc` returns `NULL`. Note that any memory referred to by `ptr` before the call to `realloc` *must* have been allocated on the heap.

A final note here: `realloc` can be an expensive call. So in general, we don't allocate storage for just one additional element, since there's a good chance that we'll need to allocate more memory for the same bucket. Of course, in this setting, we don't know how much more memory we will ultimately need. If you'll be using the sample sort on a particular type of data, it may be useful to run some tests on data that is similar to the data encountered in the application that will be using the sort. However, for a fully general sort, we won't have this option. So we might try allocating another n/b elements when a bucket is full.

7.2.4 An alternative implementation of `Map`

An alternative implementation of `Map` uses a single n-element array, `temp`, for the buckets, but it builds some additional data structures that provide information about the buckets. These data structures might store the following information:

1. How many elements will be moved from each sublist to each bucket, `from_to_cts`;
2. The subscript of the first element in each sublist that goes to each bucket `bkt_starts_in_slists`;
3. The subscript of the first element in each bucket that comes from each sublist `slist_starts_in_bkts`, and the total number of elements in each bucket;
4. The location of the first element of each bucket in `temp`, `bkt_starts`.

The first two data structures are $b \times b$ arrays with rows corresponding to sublists and columns corresponding to buckets. The third data structure is a $(b+1) \times b$ array with rows $0, 1, \ldots, b-1$ corresponding to sublists, columns corresponding to buckets, and the last row storing the total number of elements in each bucket. The fourth data structure is a b-element array.

Note that we may not need all of these data structures. For example, in our serial implementation, we only use the first b rows of `slist_starts_in_bkts` to compute the last row. So we could simply sum each column of `from_to_cts` to get the final row and store it in a one-dimensional array.

We can build these data structures using the following pseudocode:

```
Sort each sublist;
for each sublist
   for each bucket
      Count elements in sublist
         going to bucket and store
         in from_to_cts[sublist,bucket];
for each sublist
   Build bkt_starts_in_slists
      by forming exclusive prefix sums
      of row sublist of from_to_cts;
for each bucket
   Build sublist_starts_in_bkts
      by forming exclusive prefix sums
      of column bucket of from_to_cts;
Build bkt_starts by forming
   exclusive prefix sums of last row
   of sublist_starts_in_bkts;
```

In spite of the apparent complexity of this code, it only uses **for** loops and three basic algorithms:

1. A sort for the sublists,
2. Counting elements of a sublist that fall in a range determined by the splitters, and
3. Exclusive prefix sums of a row or column of a two-dimensional array. (See paragraph 'Prefix sums' on p. 404.)

For the sort, we should probably choose a sorting algorithm that has relatively little overhead, so that it will be quite fast on short lists. So let's look at some details of the second and third algorithms.

To illustrate the second algorithm, we need to specify the details of the organization of the array of splitters. We'll assume that there are $b + 1$ elements in the array `splitters` and they're stored as shown in Table 7.9. The constants `MINUS_INFTY` and `INFTY` have been chosen so that they are, respectively, less than or equal to and greater than any possible element of the input list.

Since the sublists have been sorted, we can implement the second item by iterating sequentially through a sublist:

```
int curr_row[b]; /* row of from_to_cts      */
dest_bkt = 0;    /* subscript in splitters */

Assign(curr_row, 0);  /* Set elements to 0 */
for (j = 0; j < n/b; j++) {
   while (sublist[j] >= splitters[dest_bkt+1])
      dest_bkt++;
   curr_row[dest_bkt]++;
}
```

Table 7.9 The splitters in sample sort.

Bucket	Elements
0	MINUS_INFTY = splitters[0] ≤ key < splitters[1]
1	splitters[1] ≤ key < splitters[2]
⋮	⋮
i	splitters[i] ≤ key < splitters[i+1]
⋮	⋮
b-1	splitters[b-1] ≤ key < splitters[b] = INFTY

So we iterate through the elements of the current sublist in the **for** statement. For each of these elements, we iterate through the remaining splitters until we find a splitter that is strictly greater than the current element.

Suppose, for example, that $n = 24$, $b = 4$, and the splitters are $\{-\infty, 45, 75, 91, \infty\}$. Now suppose one of the sublists contains the elements $\{21, 27, 49, 62, 86, 92\}$. Then both 21 and 27 are less than 45. So the first two elements will go to bucket 0. Similarly, 49 and 62 are both less than 75 (and greater than 45). So they should go to bucket 1. However $86 > 75$, but less than 91; so it should go to bucket 2. Finally, $92 > 91$, but less than ∞; and it should go to bucket 3. This gives us two elements going to bucket 0, two elements going to bucket 1, one element going to bucket 2, and one element going to bucket 3.

Note that since the splitters are sorted, the linear search in the while loop could be replaced with binary search. Alternatively, since the sublists are sorted, we could use a binary search to find the location of each splitter in the sublist. (See Exercise 7.20.)

Prefix sums

A **prefix sum** is an algorithm that is applied to the elements of an n-element array of numbers, and its output is an n-element array that's obtained by adding up the elements "preceding" the current element. For example, suppose our array contains the elements

$$5, 1, 6, 2, 4$$

Then an *inclusive* prefix sum of this array is

$$5, 6, 12, 14, 18$$

So

```
incl_pref_sum[0] = array[0];
```

and for $i > 0$

```
incl_pref_sum[i] = array[i] + incl_pref_sum[i-1];
```

An *exclusive* prefix sum of our array is

$$0, 5, 6, 12, 14$$

So in an exclusive prefix sum, element 0 is 0, and the other elements are obtained by adding the *preceding* element of the array to the most recently computed element of the prefix sum array:

```
excl_pref_sum[0] = 0;
```

and for $i > 0$

```
excl_pref_sum[i] = array[i-1] + excl_pref_sum[i-1];
```

Observe that since the sublists have been sorted, the exclusive prefix sums of the rows of `from_to_cts` tell us which element of a sublist should be the first element from the sublist in each bucket. In our preceding example, we have $b = 4$ buckets and for the sublist $\{21, 27, 49, 62, 86, 92\}$ the row corresponding to it in `from_to_cts` is

$$2, 2, 1, 1.$$

The exclusive prefix sums of this array are

$$0, 2, 4, 5$$

So the subscripts of the first element in the sublist going to each of the four buckets are shown in the following table:

Bucket	Subscript of First Element Going to Bucket	First Element Going to Bucket
0	0	21
1	2	49
2	4	86
3	5	92

In fact, the elements of each of the remaining data structures can be computed using exclusive prefix sums. Here's a complete example with $n = 24$, $b = 4$, $s = 8$, and the following list:

```
Sublist 0:  15, 35, 77, 83, 86, 93
Sublist 1:  21, 27, 49, 62, 86, 92
Sublist 2:  26, 26, 40, 59, 63, 90
Sublist 3:  11, 29, 36, 67, 68, 72
```

As before, the splitters are $\{-\infty, 45, 75, 91, \infty\}$. Using this information, the various data structures are

from_to_cts				
	bucket			
sublist	0	1	2	3
0	2	0	3	1
1	2	2	1	1
2	3	2	1	0
3	3	3	0	0

bkt_starts_in_slists				
sublist	bucket			
0	0	2	2	5
1	0	2	4	5
2	0	3	5	6
3	0	3	6	6

slist_starts_in_bkts				
sublist	bucket			
0	0	0	0	0
1	2	0	3	1
2	4	2	4	2
3	7	4	5	2
bkt sizes	10	7	5	2

bkt_starts				
bucket	0	1	2	3
1st elt	0	10	17	22

Note that the last row of `slist_starts_in_bkts` is the number of elements in each bucket, and it is obtained by continuing the prefix sums down the columns of `from_to_cts`. Finally, note that `bkt_starts` is obtained by taking the exclusive prefix sums of the last row of `slist_starts_in_bkts`, and it tells us where each bucket should start in the array `temp`. So we don't need to do any reallocating in this version, and there's no wasted space.

Finishing up

Now we can assign elements of `list` to `temp`, the array that will store the buckets. Since `bkt_starts` tells us where the first element of each bucket is (e.g., the first element in bucket 1 should go to `temp[10]`), we can finish by iterating through the sublists, and in each sublist iterating through the buckets:

```
1     for each slist of list {
2        sublist = pointer to start of next block
3                of n/b elements in list;
```

```
4       for each bkt in temp {
5          bucket = pointer to start of bkt in temp;
6          dest = 0;
7
8          /* Now iterate through the elements of sublist */
9          first_src_sub = bkt_starts_in_slist[slist, bkt];
10         if (bkt < b)
11            last_src_sub = bkt_starts_in_slist[slist, bkt+1];
12         else
13            last_src_sub = n/b;
14         for (src = first_src_sub; src < last_src_sub; src++)
15            bucket[dest++] = sublist[src];
16      }
17   }
```

The assignments in Lines 2 and 4 are just defining pointers to the beginning of the current sublist and the beginning of the current bucket. The innermost loop iterates through the elements of the current sublist that go to the current bucket. In general, the limits of this iteration can be obtained from `bkt_starts_in_slist`. However, since there are only b columns in this array, the last element has to be treated as a special case—hence the else clause.

7.2.5 Parallelizing sample sort

Serial sample sort can be divided into a sequence of well-defined phases:

1. Take a sample from the input list.
2. Find the splitters from the sample.
3. Build any needed data structures from the list and the splitters.
4. Map keys in the list into buckets.
5. Sort the buckets.

There are other possible phases—e.g., sorts at various points in the algorithm—and, as we've already seen, there are a variety of possible implementations of each phase.

Serial sample sort can be naturally parallelized for MIMD systems by assigning sublists and buckets to processes/threads. For example, if each process/thread is assigned a sublist, then Steps 1 and 2 can have embarrassingly parallel implementations. First each process/thread chooses a "subsample" from its sublist, and then the splitters are chosen by each process/thread from its subsample.

Since CUDA thread blocks can operate independently of each other, in CUDA it is natural to assign sublists and buckets to thread blocks instead of individual threads.

We'll go into detail for the various APIs below, but we'll assume that each MIMD process/thread is assigned a sublist and a bucket. For the CUDA implementations, we'll look at two different assignments. The first, basic, implementation will also assign a bucket and a sublist to each thread. We'll use a more natural, but more complex, approach in the second implementation, where we'll assign buckets and sublists to thread blocks.

Note also that in our serial implementations, we used a generic sort function several times. So for the MIMD parallel implementations, we'll assume that the sorts are either single-threaded or that we have a parallel sort function that performs well on a relatively small list. For the basic CUDA implementation, we'll use single-threaded sorts on the host and each thread. For the second implementation, we'll assume that we have a parallel sort function that performs well when implemented with a single thread block and one that works well with multiple thread blocks on relatively small lists.

Finally, for our "generic" parallelizations, we'll focus on the MIMD APIs, and we'll address the details of CUDA implementations in Section 7.2.9.

Basic MIMD implementation

The first implementation uses many of the ideas from our first serial implementation:

1. Use a random number generator to choose a sample from the input list.
2. Sort the sample.
3. Choose splitters from the sample.
4. Use the splitters to map keys to buckets.
5. Sort buckets.

As we noted earlier, Step 1 can be embarrassingly parallel if we assign a sublist to each MIMD process/thread. For Step 2 we can either have a single process/thread carry out the sort, or we can use a parallel sort that is reasonably efficient on short lists (e.g., parallel odd-even transposition sort).

If the sample is distributed among the processes/threads, then Step 3 can be implemented by having each process/thread choose a single splitter. So if we used a single process/thread for Step 2, we can first distribute the sample among the processes/threads. Finally, to determine which elements of the list go to which buckets, each process/thread will need access to all of the splitters. So in a shared memory system, the splitters should be shared among the threads, and in a distributed memory system, the splitters should be gathered to each process.

For Step 4, one task might be to determine which bucket an element of the list should be mapped to. A second might be to assign the element to its destination bucket. Clearly these two tasks can be aggregated, since the second can't proceed until the first is completed. The obvious problem here is that we need some sort of synchronization among tasks with the same destination bucket. For example, on a shared memory system, we might use a temporary array to record for each element its destination bucket. Then a process/thread can iterate through the list, and when it finds an element going to its assigned bucket, it can copy from the element its destination bucket. This will require an extra array to record destinations, but if the list isn't huge, this should be possible.

For distributed memory, this approach could be very expensive: we might gather both the original list and the list of destination buckets to each process. A less costly alternative would be to construct some of the data structures outlined in the second serial implementation and use these with an MPI collective communication called

`MPI_Alltoallv` to redistribute the elements of the list from each process to its correct destination. We'll go into detail in the MPI section (Section 7.2.8), but the idea is that each process has a block of elements going to every other process. For example, process q has some elements that should go to process 0, some that should go to process 1, etc., and this is the case for each process. So we would like for each process to scatter its original elements to the correct processes, *and* we would like for each process to gather its "final" elements from every process.

In either the shared- or the distributed-memory implementation, we can finish by having each process/thread sort its elements using a serial sort.

A second implementation

As you might have guessed our second implementation uses many of the ideas from the second serial implementation.

1. Sort the elements of each sublist.
2. Choose a random or a "deterministic" sample as discussed in our serial implementation.
3. Sort the sample.
4. Choose equally spaced splitters from the sample.
5. Use the input list and the splitters to build the data structures needed to implement the assignment of elements to buckets.
6. Assign elements of the list to the correct buckets.
7. Sort each bucket.

As with the first implementation, we'll assign a single sublist and a single bucket to each thread/process.

As we noted near the beginning of this section, for steps 1, 3, and 7 we assume each sort is either done by a single thread/process, or that we already have a good parallel sort for "small" lists.

Step 2 is embarrassingly parallel for both the random and deterministic samples. For the deterministic sample, a task is choosing an element of the sample, and since they're equally spaced, a single thread/process can choose a subset of the sample elements from a single sublist.

Step 4 is the same as choosing the splitters in the first parallel implementation.

For Step 5, there are two basic parts. First we need to count the number of elements going from each sublist to each bucket. So as before, one task might be determining for an element of the list which bucket it should go to. Now a second task could be incrementing the appropriate element of the data structure. Clearly these tasks can be aggregated. Since the number of processes/threads is the same as the number of buckets, we can map the aggregate tasks for the elements of a sublist to a single process/thread.

The second part of Step 5 is implementing one or more prefix sums. Since the number of processes/threads is the same as the number of buckets, one process/thread can be responsible for each set of prefix sums. See below and, for example, [27].

Table 7.13 Run-times of the second Pthreads implementation of sample sort (times are in seconds).

Threads	Run-time
1	2.65
2	1.36
4	1.08
8	0.35
16	0.20
32	0.14
64	0.12
qsort	2.59

We can observe that performance is very similar for both the OpenMP and Pthreads approaches. This is not surprising; apart from some minor syntactical differences, their implementations are also quite similar.

7.2.8 Implementing sample sort with MPI

For the MPI implementation, we assume that the initial list is distributed among the processes, but the final, sorted list is gathered to process 0. We also assume that both the total number of elements n and the sample size s are evenly divisible by the number of processes p. So each process will start with a sublist of n/p elements, and each process can choose a "subsample" of s/p elements.

First implementation

Each process can choose a sample of its sublist using the method outlined above for the first serial algorithm. Then these can be gathered onto process 0. Process 0 will sort the sample, choose the splitters, and broadcast the splitters to all the processes. Since there will be p buckets, there will be $p + 1$ splitters: the smallest less than or equal to any element in the list (`MINUS_INFTY`), and the largest greater than any element in the list (`INFTY`).

Then each process can determine to which bucket each of its elements belongs, and we can use the MPI function `MPI_Alltoallv` to distribute the contents of each sublist among the buckets. The syntax is

```
int MPI_Alltoallv(
      const void     *send_data      /* in  */,
      const int      *send_counts    /* in  */,
      const int      *send_displs    /* in  */,
      MPI_Datatype   send_type       /* in  */,
      void           *recv_data      /* out */,
      const int      *recv_counts    /* in  */,
      const int      *recv_displs    /* in  */,
```

```
MPI_Datatype  recv_type      /* in  */,
MPI_Comm      comm           /* in  */);
```

In our setting, `send_data` is the calling process' sublist, and `recv_data` is the calling process' bucket. Furthermore, the function assumes that the elements going to any one process occupy contiguous locations in `send_data`, and the elements coming from any one process should occupy contiguous locations in `recv_data`.[4] The communicator is `comm`, which for us will be `MPI_COMM_WORLD`. For us, both `send_type` and `recv_type` are `MPI_INT`. The arrays `send_counts` and `send_displs` specify where the elements of the sublist should go, and the arrays `recv_counts` and `recv_displs` specify where the elements of the bucket should come from.

As an example, suppose we have three processes, the splitters are $-\infty, 4, 10, \infty$ and the sublists and buckets are shown in the following table:

	Processes		
	0	1	2
sublists	3, 5, 6, 9	1, 2, 10, 11	4, 7, 8, 12
buckets	3, 1, 2	5, 6, 9, 4, 7, 8	10, 11, 12

Then we see that processes should send the following numbers of elements to each process:

From Process	To Process		
	0	1	2
0	1	3	0
1	2	0	2
2	0	3	1

So `send_counts` on each process should be

Process 0 : {1, 3, 0}
Process 1 : {2, 0, 2}
Process 2 : {0, 3, 1}

and `recv_counts` should be

Process 0 : {1, 2, 0}
Process 1 : {3, 0, 3}
Process 2 : {0, 2, 1}

So `recv_counts` can be obtained by taking the *columns* of the matrix whose rows are the `send_counts`.

[4] These assumptions apply when the datatype arguments occupy contiguous locations in memory.

The array `send_counts` tells `MPI_Alltoallv` how many **ints** should be sent from a given process to any other process, and the `recv_counts` array tells the function how many elements come from each process to a given process. However, to determine where the elements of a sublist going to a given process begin, `MPI_Alltoallv` also needs an offset or **displacement** for the first element that goes from the process to each process, and the first element that comes to a process from each process. In our setting, these displacements can be obtained from the exclusive prefix sums of `send_counts`. Thus, in our example, the `send_displs` arrays on the different processes are

$$\text{Process } 0 : \{0, 1, 4\}$$
$$\text{Process } 1 : \{0, 2, 2\}$$
$$\text{Process } 2 : \{0, 0, 3\}$$

and the `recv_displs` arrays are

$$\text{Process } 0 : \{0, 1, 3\}$$
$$\text{Process } 1 : \{0, 3, 3\}$$
$$\text{Process } 2 : \{0, 0, 2\}$$

Note also that by adding in the final elements of `recv_counts` to the corresponding final elements `recv_displs` we get the total number of elements needed for `recv_data`:

$$\text{Process } 0 : \texttt{recv\_data[]},\ 3 + 0 \textbf{ ints}$$
$$\text{Process } 1 : \texttt{recv\_data[]},\ 3 + 3 \textbf{ ints}$$
$$\text{Process } 2 : \texttt{recv\_data[]},\ 2 + 1 \textbf{ ints}$$

We'll just use a *large* n-element array for `recv_data`. So we won't need to worry about allocating the exact amount of needed storage for each process' `recv_data`. After the call to `MPI_Alltoallv`, each process sorts its local list.

Note that, in general, we'll need to first get the *numbers* of elements coming from each process. We could use `MPI_Alltoallv` to do this, but since each process is sending a single int to every process, we can use a somewhat simpler function: `MPI_Alltoall`:

```
int MPI_Alltoall(
      const void    *send_data     /* in  */,
      int           send_count     /* in  */,
      MPI_Datatype  send_type      /* in  */,
      void          *recv_data     /* out */,
      int           recv_count     /* in  */,
      MPI_Datatype  recv_type      /* in  */,
      MPI_Comm      comm           /* in  */);
```

For us, each process should put in `send_data` the number of elements going to each process; `send_count` and `recv_count` should be 1, and the types should both be `MPI_INT`.

At this point our convention is that we need to gather the entire list onto process 0. At first, it might seem that we can simply use `MPI_Gather` to collect the distributed list onto process 0. However, `MPI_Gather` expects each process to contribute the same number of elements to the global list, and, in general, this won't be the case. We encountered another version of `MPI_Gather`, `MPI_Gatherv`, back in Chapter 3. (See Exercise 3.13.) This has the following syntax:

```
int MPI_Gatherv(
      void*          loc_list          /* in  */,
      int            loc_list_count    /* in  */,
      MPI_Datatype   send_elt_type     /* in  */,
      void*          list              /* out */,
      const int*     recv_counts       /* in  */,
      const int*     recv_displs       /* in  */,
      MPI_Datatype   recv_elt_type     /* in  */,
      int            root              /* in  */,
      MPI_Comm       comm              /* in  */);
```

This is very similar to the syntax for `MPI_Gather`:

```
int MPI_Gather(
      void*          loc_list          /* in  */,
      int            loc_list_count    /* in  */,
      MPI_Datatype   send_elt_type     /* in  */,
      void*          list              /* out */,
      const int      recv_count        /* in  */,
      MPI_Datatype   recv_elt_type     /* in  */,
      int            root              /* in  */,
      MPI_Comm       comm              /* in  */);
```

The only differences are that in `MPI_Gatherv`, there is an *array* of `recv_counts` instead of a single int, and there is an array of displacements. These differences are completely natural: each process is contributing a collection `loc_list` of `loc_list_count`, each of which has type `send_elt_type`. In `MPI_Gather`, each process contributes the same number of elements, so `loc_list_count` is the same for each process. In `MPI_Gatherv`, the processes contributed different numbers of elements. So instead of a single `recv_count`, we need an array of `recv_counts`, one for each process. Also, by analogy with `MPI_Alltoallv`, we need an array of *displacements*, `recv_displs`, that indicates where in `list` each process' contribution begins.

So we need to know how many elements each process is contributing, and we need to know the offset in `list` of the first element of each process' contribution.

Now recall that each process has built two arrays that are used in the call to `MPI_Alltoallv`. The first one specifies the number of elements the process receives from every process, and the second one specifies the offset (or displacement) of the first element received from each process. If we call these arrays `my_fr_counts` and `my_fr_offsets`, respectively, we can get the total number of elements received from each process by adding the last elements

```
my_new_count = my_fr_counts[p-1] + my_fr_offsets[p-1];
```

We can "gather" these counts onto process 0 to get the elements of `recv_counts`, and we can form the exclusive prefix sums of `recv_counts` to get `recv_displs`.

To summarize, then, Program 7.5 shows pseudocode for this version of sample sort.

```
/* s = global sample size */
loc_s = s/p;
Gen_sample(my_rank, loc_list, loc_n, loc_samp, loc_s);
Gather_to_0(loc_samp, global_sample);
if (my_rank == 0) Find_splitters(global_sample, splitters);
Broadcast_from_0(splitters);

/* Each process has all the splitters */
Sort(loc_list, loc_n);
/* my_to_counts has storage for p ints */
Count_elts_going_to_procs(
      loc_list, splitters, my_to_counts);

/* Get number of elements coming from each process
 *       in my_fr_counts */
MPI_Alltoall(my_to_counts, 1, MPI_INT,
      my_fr_counts, 1, MPI_INT, comm);

/* Construct displs or offset arrays for each process */
Excl_prefix_sums(my_to_counts, my_to_offsets, p);
Excl_prefix_sums(my_fr_counts, my_fr_offsets, p);

/* Redistribute the elements of the list          */
/* Each process receives its elements into tlist */
MPI_Alltoallv(
      loc_list, my_to_counts, my_to_offsets, MPI_INT,
      tlist, my_fr_counts, my_fr_offsets, MPI_INT, comm);
my_new_count = my_fr_offsets[p-1] + my_fr_counts[p-1];

Sort(tlist, my_new_count);

/* Gather tlists to process 0 */
MPI_Gather(&my_new_count, 1, MPI_INT, bkt_counts, 1,
      MPI_INT, 0, comm);
if (my_rank == 0)
      Excl_prefix_sums(bkt_counts, bkt_offsets, p);
MPI_Gatherv(tlist, my_new_count, MPI_INT, list, bkt_counts,
      bkt_offsets, MPI_INT, 0, comm);
```

Program 7.5: Pseudocode for first MPI implementation of sample sort.

Table 7.14 Run-times of first MPI implementation of sample sort (times are in seconds).

Processes	Run-time
1	9.94e-1
2	4.77e-1
4	2.53e-1
8	1.32e-1
16	7.04e-2
32	5.98e-2
`qsort`	7.39e-1

Table 7.14 shows run-times of this implementation in seconds. The input list has $2^{22} = 4,194,304$ **int**s, and the sample has 16,384 **int**s. The last row of the table shows the run-time of the C library `qsort` function. We're using a system running Linux with two Xeon Silvers 4116 running at 2.1 GHz. Each of the Xeons has 24 cores, and the MPI implementation is MPICH 3.0.4. Compiler optimization made little difference in overall run-times. So we're reporting run-times with no optimization.

Second implementation

Our first MPI implementation seems to be fairly good. Except for the 32 core run, the efficiency was at least 0.88, and with 2 or more cores, it is always faster than the C library `qsort` function. So we would like to improve the efficiency when we're using 32 cores.

There are several modifications we can make to the code that may improve performance:

1. Replace the serial sort of the sample with a parallel sort.
2. Use a parallel algorithm to find the splitters.
3. Write our own implementation of the `Alltoallv` function.

The first two modifications are straightforward. For the first, we can replace the gather of the sample to process zero and the serial sort of the sample, by a parallel odd-even transposition sort. (See Section 3.7.)

After the parallel sort of the sample, the sample will be distributed among the processes. So there's no need to scatter the sample, and the keys on process $q-1$ are all $\leq$ the keys on process q, for $q = 1, 2, \ldots, p-1$. Thus, since there are p splitters, we can compute `splitter[i]` by sending the maximum element of the sample on each process to the next higher-ranked process (if it exists), and the next higher-ranked process can average this value with its minimum value. (See Fig. 7.9.)

So let's take a look at implementing `Alltoallv`. When p is a power of 2, we can implement this with a butterfly structure (Fig. 3.9). First we'll use a small example to

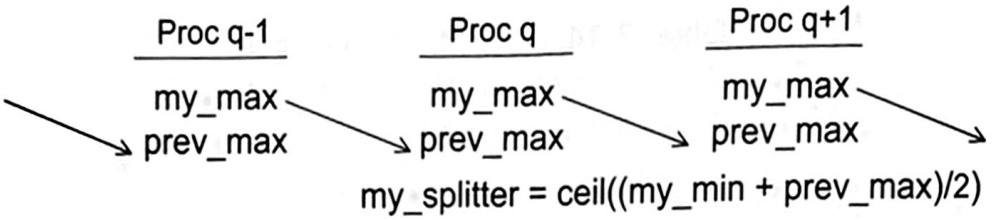

FIGURE 7.9

Finding the splitters.

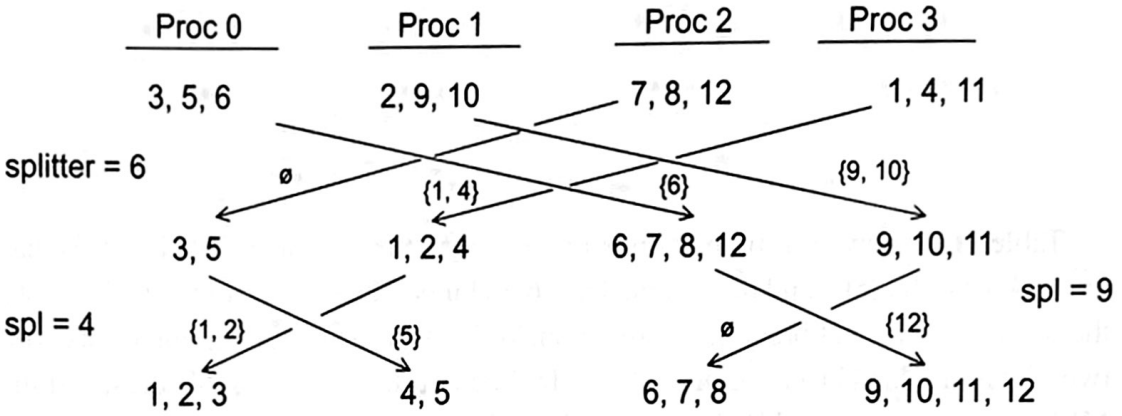

FIGURE 7.10

Butterfly implementation of `Alltoallv`.

illustrate how this is done. We'll then talk about the general case. So suppose we have four processes, and the splitters are $\{-\infty, 4, 6, 9, \infty\}$. Also suppose that the sublists are shown in the following table:

	Processes			
	0	1	2	3
sublists	3, 5, 6	2, 9, 10	7, 8, 12	1, 4, 11

Then a butterfly implementation of `Alltoallv` can be illustrated as shown in Fig. 7.10.

The algorithm begins with the sorted sublists distributed among the processors. The first stage uses the "middle" splitter, `splitter[2]` = 6 to divide the keys. Keys that are less than 6 will go to the first half of the distributed list, and keys greater than or equal to 6 will go to the second half of the distributed list. Processes 0 and 2 exchange keys, and processes 1 and 3 exchange keys. After the exchanges take place what remains of the old sublist is merged with the received keys.

Now that the distributed list has been split into two halves, there is no need for further communication between the processes that are less than $p/2$ and the processes that are greater than or equal to $p/2$. So we can now split the first half of the list between processes 0 and 1, and we can split the second half of the list between processes 2 and 3.

To split the first half of the list among processes 0 and 1, we'll use `splitter[1] = 4`, and to split the second half, we'll use `splitter[3] = 9`. As in the first "phase," the paired processes (0–1 and 2–3) will exchange keys, and each process will merge its remaining keys with its new keys.

This gives us a distributed, sorted list, and we can use `MPI_Gather` and `MPI_Gatherv` to gather the list to process 0: this is the same as the final step in the algorithm we used for the first MPI implementation.

So let's take a look at some of the details in the implementation of `Alltoallv`. (See Program 7.6.)

```
1   unsigned bitmask = which_splitter = p >> 1;
2
3   while (bitmask >= 1) {
4      partner = my_rank ^ bitmask;
5      if (my_rank < partner) {
6         Get_send_args(loc_list, loc_n, &offset, &count,
7            splitter[which_splitter], SEND_TO_UPPER);
8         new_loc_n = offset
9         bitmask >>= 1;
10        which_splitter -= bitmask /* splitter for next pass */
11     } else {
12        Get_send_args(loc_list, loc_n, &offset, &count,
13           splitter[which_splitter], SEND_TO_LOWER);
14        new_loc_n = loc_n - count;
15        bitmask >>= 1;
16        which_splitter += bitmask;
17     }
18     MPI_Sendrecv(loc_list + offset, count, MPI_INT, partner,
19           0, rcv_buf, n, MPI_INT, partner, 0, comm, &status);
20     MPI_Get_count(&status, MPI_INT, &rcv_count);
21
22     loc_n = new_loc_n;
23     if (my_rank < partner) {
24        offset = 0;
25     } else {
26        offset = count;
27     }
28
29     /* Merges loc_list and rcv_buf into tmp_buf and swaps
30      *     loc_list and tmp_buf */
31     Merge(&loc_list, &loc_n, offset,
32           rcv_buf, rcv_count, &tmp_buf);
33  } /* while bitmask >= 1 */
```

Program 7.6: Pseudocode for the butterfly algorithm in the second MPI sample sort.

The key to understanding how it works is to look at the binary or base 2 representations of the various scalar values. The variable `bitmask` determines which processes are paired: process `my_rank` is paired with the process with rank

```
partner = my_rank ^ bitmask;
```

(Recall that ^ takes the bitwise exclusive or of its operands.) The initial value of bitmask is `p >> 1`, and after each phase of the butterfly, it's updated by a right shift `bitmask >= 1`. When it's equal to 0, the iterations stop.

In our example, when $p = 4 = 100_2$, `bitmask` will take on the values 10_2, 01_2, and 00_2. The following table shows which processes are paired:

	Processes			
bitmask	$0 = 00_2$	$1 = 01_2$	$2 = 10_2$	$3 = 11_2$
10_2	$2 = 10_2$	$3 = 11_2$	$0 = 00_2$	$0 = 01_2$
01_2	$1 = 01_2$	$0 = 00_2$	$3 = 11_2$	$2 = 10_2$

Note that all the processes will have the same value of `bitmask`.

The variable `which_splitter` is the subscript of the element in the splitter array that's used to determine which elements should be sent from one process to its partner. So, in general, the processes will have different values of `which_splitter`. It is also initialized to `p >> 1`, but it is updated using its value in the previous iteration and the updated bitmask. If the process' current partner has a higher rank, then, in the next iteration, it should choose the splitter

```
which_splitter = which_splitter - bitmask
```

If the partner has a lower rank, it should choose the splitter

```
which_splitter = which_splitter + bitmask
```

The following table shows the values of `which_splitter` for each process when $p = 8$. (`bitm` is an abbreviation of `bitmask`.)

New	Processes							
bitm	$0 = 000_2$	$1 = 001_2$	$2 = 010_2$	$3 = 011_2$	$4 = 100_2$	$5 = 101_2$	$6 = 110_2$	$7 = 111_2$
XXX	$4 = 100_2$	$4 = 100_2$	$4 = 100_2$	$4 = 100_2$	$4 = 100_2$	$4 = 100_2$	$4 = 100_2$	$4 = 100_2$
010_2	$2 = 010_2$	$2 = 010_2$	$2 = 010_2$	$2 = 010_2$	$6 = 110_2$	$6 = 110_2$	$6 = 110_2$	$6 = 110_2$
001_2	$1 = 001_2$	$1 = 001_2$	$3 = 011_2$	$3 = 011_2$	$5 = 101_2$	$5 = 101_2$	$7 = 111_2$	$7 = 111_2$

The function `Get_send_args` determines the arguments `offset` and `count` that are used in the call to `MPI_Sendrecv`. The variable `offset` gives the subscript of the first element in `loc_list` that will be sent to the partner, and the variable `count` is the number of elements that will be sent. They are determined by a search of the local list for the location of the splitter. This can be carried out with a linear or a binary search. We used linear search.

After the call to `MPI_Sendrecv`, we use the `status` argument and a call to `MPI_Get_count` to determine the number of elements that were received, and then

call a merge function that merges the old local list and the received list into `tmp_buf`. At the end of the merge, the new list is swapped with the old `loc_list`.

An important difference between the implementation that uses `MPI_Alltoallv` and this implementation is that to use `MPI_Alltoallv`, we must determine *before* the call

- How many elements go *from* each process *to* every other process, and
- The offsets or displacements of the first element going from each process to every other process.

As we've already seen, this requires communication, so it's relatively time consuming. With the hand-coded implementation, we can determine these values as the algorithm proceeds, and the *amount* of the data is, effectively, included in the communication of the actual data. We can access the amount by using the status and the (local) MPI function `MPI_Get_count` after receiving the data.

In the approach outlined above for implementing the data exchange, we use `MPI_Sendrecv` and a receive buffer that's large enough to receive any list (e.g., an n-element buffer). A more memory-efficient implementation can split the send and the receive, and before the receive is executed, it can call `MPI_Probe`:

```
int MPI_Probe(
      int           source     /* in  */,
      int           tag        /* in  */,
      MPI_Comm      comm       /* in  */,
      MPI_Status*   status_p   /* out */);
```

This function will block until it is notified that there is a message from process `source` with the tag `tag` that has been sent with communicator `comm`. When the function returns, the fields referred to by `status_p` will be initialized, and the calling process can use `MPI_Get_count` to determine the size of the incoming message. With this information, the process can, if necessary, call `realloc` to enlarge the receive buffer before calling `MPI_Recv`.

To ensure safety in the MPI sense (see Section 3.7.3) of this pair of communications, we can use a **nonblocking** send instead of the standard send. The syntax is

```
int MPI_Isend(
      void*          buf        /* in  */,
      int            count      /* in  */,
      MPI_Datatype   datatype   /* in  */,
      int            dest       /* in  */,
      int            tag        /* in  */,
      MPI_Comm       comm       /* in  */,
      MPI_Request*   request_p  /* out */);
```

The first six arguments are the same as the arguments to `MPI_Send`. However, unlike `MPI_Send`, `MPI_Isend` only *starts* the communication. In other words, it notifies the system that it should send a message to `dest`. After notifying the system, `MPI_Isend`

Table 7.15 Run-times of the two MPI implementation of sample sort (times are in seconds).

Processes	Version 1 Run-time	Version 2 Run-time
1	9.94e-1	8.39e-1
2	4.77e-1	4.03e-1
4	2.53e-1	2.10e-1
8	1.32e-1	1.10e-1
16	7.04e-2	6.36e-2
32	5.98e-2	4.67e-2
qsort	7.39e-1	7.39e-1

returns: the "I" stands for "immediate," since the function returns (more or less) immediately. Because of this, the user code *cannot* modify the send buffer `buf` until after the communication has been explicitly *completed.*

There are several MPI functions that can be used for completing a nonblocking operation, but the only one we'll need is `MPI_Wait`:

```
int MPI_Wait(
      MPI_Request*    request_p    /* in  */,
      MPI_Status*     status_p     /* out */);
```

The `request_p` argument is the argument that was returned by the call to `MPI_Isend`. It is an opaque object that is used by the system to identify the operation. When our program calls `MPI_Wait`, the program blocks until the send is completed. For this reason, nonblocking communications are sometimes called "split" or "two-phase" communications. We won't make use of the `status_p` argument, so we can pass in `MPI_STATUS_IGNORE`.

To summarize, if we want to minimize memory use, we can replace the call to `MPI_Sendrecv` in Program 7.6 with the function shown in Program 7.7. The function

```
int MPI_Abort(
      MPI_Comm     comm          /* in  */,
      int          errorcode     /* in  */);
```

will terminate all the processes in `comm` and return `errorcode` to the invoking environment. So we need to define the constant `REALLOC_FAILED`. Note that current implementations of MPI will terminate *all* processes, not just those in `comm`.

Let's take a look at the performance of our the version of sample sort that uses a hand-coded butterfly implementation of `Alltoallv` and `MPI_Sendrecv` (with large buffers). Table 7.15 shows run-times of this version and our original version of sample sort that uses `MPI_Alltoallv`. All the times were taken on the same system (see page 423). Both versions used no compiler optimization, and the input data was identical. We see that the second version is always faster than the first, and it is usually more efficient. The sole exception is that with $p = 16$, the first version has an ef-

```
1  void Send_recv(int snd_buf[], int count, int** rcv_buf_p,
2        int* rcv_buf_sz_p, int partner, MPI_Comm comm) {
3     int rcv_count, *rcv_ptr;
4     MPI_Status status;
5     MPI_Request req;
6
7     /* Start the send */
8     MPI_Isend(snd_buf, count, MPI_INT, partner, 0, comm, &req);
9
10    /* Wait for info on storage needed for rcv_buf */
11    MPI_Probe(partner, 0, comm, &status);
12    MPI_Get_count(&status, MPI_INT, &rcv_count);
13    if (rcv_count > *rcv_buf_sz_p) {
14       rcv_ptr = realloc(*rcv_buf_p, rcv_count);
15       if (rcv_ptr == NULL)
16          /* Call to realloc failed, quit */
17          MPI_Abort(comm, REALLOC_FAILED);
18       else
19          *rcv_buf_p = rcv_ptr;
20    }
21
22    /* Now do the receive */
23    *rcv_buf_sz = rcv_count;
24    MPI_Recv(*rcv_buf_p, rcv_count, MPI_INT,
25          partner, 0, comm, &status);
26
27    /* Complete the send */
28    MPI_Wait(&req, MPI_STATUS_IGNORE);
29 } /* Send_recv */
```

Program 7.7: `Send_recv` function for minimizing memory requirements.

ficiency of 0.88, while the second has an efficiency of 0.82. This is because the run-time of the first version is so much greater. Note that neither version has a very high efficiency with $p = 32$: the first version has an efficiency of 0.52 and the second 0.56. So we'll look at a couple of further improvements to the second version in Exercise 7.22.

7.2.9 Implementing sample sort with CUDA

As usual, we'll look at two implementations. The first splits the implementation between the host and the device. The second has the same basic structure as our second serial implementation. Unlike the first implementation, however, the entire algorithm, except for synchronization, is executed on the device.