Parallel FFT and Sorting

1. The Fast Fourier Transform in Parallel
   - the Fastest Fourier Transform in the West (FFTW)
   - running the OpenMP version of FFTW
2. Bucket Sort for Distributed Memory
   - bucket sort in parallel
   - communication versus computation
3. Quicksort for Shared Memory
   - partitioning numbers
   - quicksort with OpenMP
4. Radix Sort for GPU acceleration
   - parallel radix sort

MCS 572 Lecture 34
Introduction to Supercomputing
Jan Verschelde, 13 November 2024

Parallel FFT and Sorting

1. The Fast Fourier Transform in Parallel
   - the Fastest Fourier Transform in the West (FFTW)
   - running the OpenMP version of FFTW
2. Bucket Sort for Distributed Memory
   - bucket sort in parallel
   - communication versus computation
3. Quicksort for Shared Memory
   - partitioning numbers
   - quicksort with OpenMP
4. Radix Sort for GPU acceleration
   - parallel radix sort

the Discrete Fourier Transform (DFT)

A periodic function $f(t)$ can be written as a series of sinusoidal waveforms of various frequencies and amplitudes: the Fourier series.

The Fourier transform maps f from the time to the frequency domain.

In discrete form:

$$F_k = \frac{1}{n} \sum_{j=0}^{n-1} f_j e^{-2\pi i(jk/n)}, \quad k = 0, 1, \ldots, n-1, f_k = f(x_k).$$

The Discrete Fourier Transform (DFT) maps a convolution into a componentwise product.

The Fast Fourier Transform is an algorithm that reduces the cost of the DFT from $O(n^2)$ to $O(n \log(n))$, for length n.

Many applications, for example: signal and image processing.

the Fastest Fourier Transform in the West (FFTW)

FFTW is a library for the Discrete Fourier Transform (DFT), developed at MIT by Matteo Frigo and Steven G. Johnson available under the GNU GPL license at `http://www.fftw.org`.

FFTW 3.3 supports MPI and comes with multithreaded versions: with Cilk, Pthreads and OpenMP are supported.

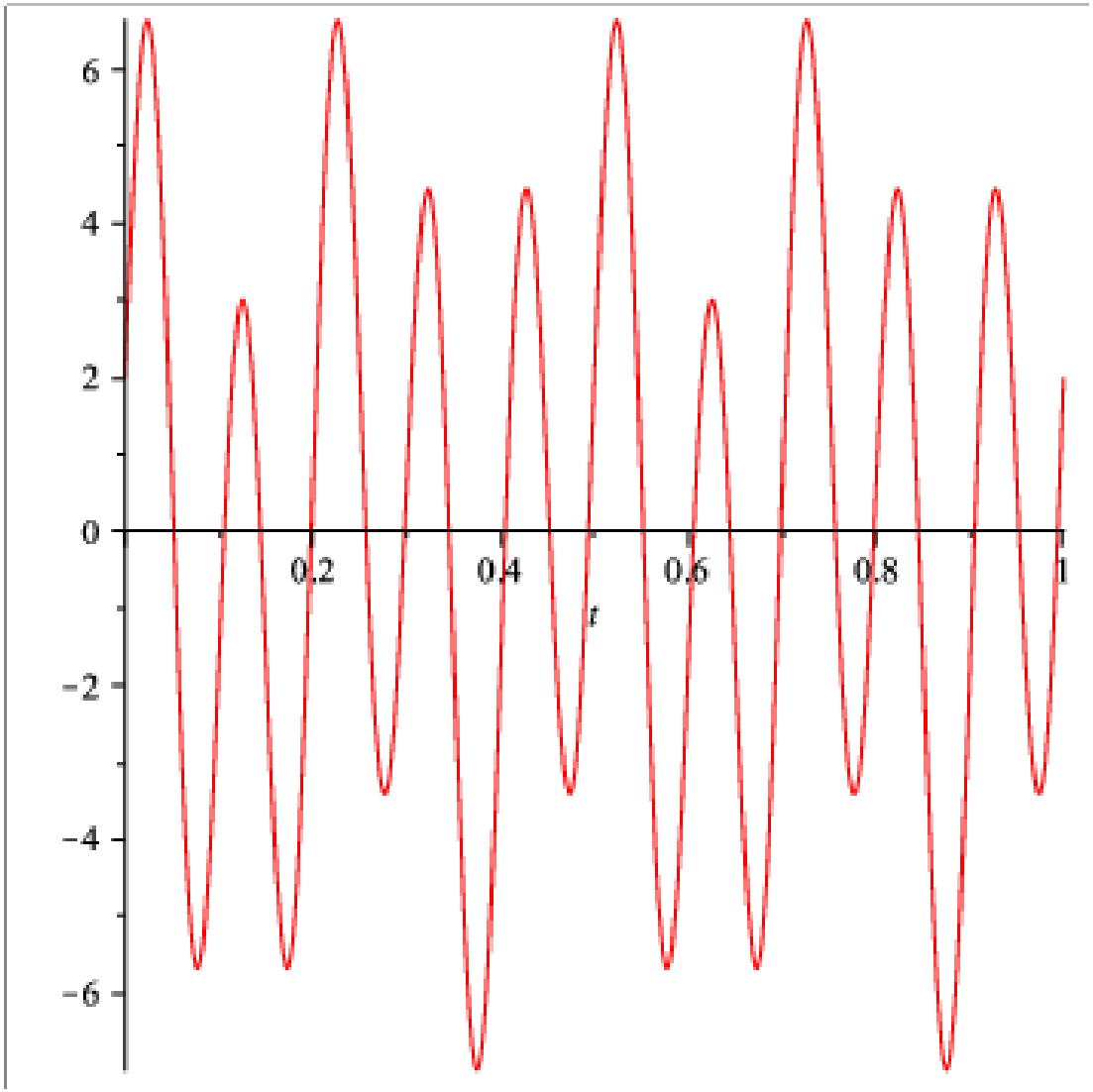
Before `make install`, do

```
./configure --enable-mpi --enable-openmp --enable-threads
```

FFTW received the 1999 J. H. Wilkinson Prize for Numerical Software.

a test signal

Consider $f(t) = 2\cos(4 \times 2\pi t) + 5\sin(10 \times 2\pi t)$, for $t \in [0, 1]$.



processing a test signal

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <fftw3.h>

double my_signal ( double t );
/*
 * Defines a signal composed of
 * a cosine of frequency 4 and amplitude 2,
 * a sine of frequency 10 and amplitude 5. */

void sample_signal
 ( double f ( double t ), int m,
   double *x, double *y );
/*
 * Takes m = 2^k samples of the signal f,
 * returns abscisses in x and samples in y. */
```

calling FFTW

```
int main ( int argc, char *argv[] )
{
   const int n = 256;
   double *x, *y;
   x = (double*)calloc(n,sizeof(double));
   y = (double*)calloc(n,sizeof(double));
   sample_signal(my_signal,n,x,y);

   int m = n/2+1;
   fftw_complex *out;
   out
   = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*m);

   fftw_plan p;
   p = fftw_plan_dft_r2c_1d(n,y,out,FFTW_ESTIMATE);
   fftw_execute(p);
```

processing the output

```
   printf("scanning through the output of FFTW...\n");
   double tol = 1.0e-8;  /* threshold on noise */
   int i;
   for(i=0; i<m; i++)
   {
      double v = fabs(out[i][0])/(m-1)
               + fabs(out[i][1])/(m-1);
      if(v > tol)
      {
         printf("at %d : (%.5e,%.5e)\n",
                i,out[i][0],out[i][1]);
         printf("=> frequency %d and amplitude %.3e\n",
                i,v);
      }
   }
   return 0;
}
```

compiling and running

```
$ make fftw_use
gcc fftw_use.c -o /tmp/fftw_use -lfftw3 -lm

$ /tmp/fftw_use
scanning through the output of FFTW...
at 4 : (2.56000e+02,-9.63913e-14)
=> frequency 4 and amplitude 2.000e+00
at 10 : (-7.99482e-13,-6.40000e+02)
=> frequency 10 and amplitude 5.000e+00
$
```

We recovered the frequencies and amplitudes
in the components of our test signal.

Parallel FFT and Sorting

1. The Fast Fourier Transform in Parallel
   - the Fastest Fourier Transform in the West (FFTW)
   - running the OpenMP version of FFTW
2. Bucket Sort for Distributed Memory
   - bucket sort in parallel
   - communication versus computation
3. Quicksort for Shared Memory
   - partitioning numbers
   - quicksort with OpenMP
4. Radix Sort for GPU acceleration
   - parallel radix sort

timing the FFTW – the setup

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <fftw3.h>

void random_sequence ( int n, double *re, double *im );
/*
 * Returns a random sequence of n complex numbers
 * with real parts in re and imaginary parts in im. */

int main ( int argc, char *argv[] )
{
   int n,m;
   if(argc > 1)
      n = atoi(argv[1]);
   else
      printf("please provide dimension on command line\n");
   m = (argc > 2) ? atoi(argv[2]) : 1;
```

computing m times for dimension n

```
double *x, *y;
x = (double*)calloc(n,sizeof(double));
y = (double*)calloc(n,sizeof(double));
random_sequence(n,x,y);

fftw_complex *in,*out;
in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*n);
out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*n);

fftw_plan p;
p = fftw_plan_dft_1d(n,in,out,FFTW_FORWARD,FFTW_ESTIMATE);
clock_t tstart,tstop;
tstart = clock();
int i;
for(i=0; i<m; i++) fftw_execute(p);
tstop = clock();
printf("%d iterations took %.3f seconds\n",
       m,(tstop-tstart)/((double) CLOCKS_PER_SEC));
```

using OpenMP

```
#include <omp.h>
#include <fftw3.h>

int main ( int argc, char *argv[] )
{
   int t;  /* number of threads */

   t = (argc > 3) ? atoi(argv[3]) : 1;

   int okay = fftw_init_threads();
   if(okay == 0)
      printf("error in thread initialization\n");
   omp_set_num_threads(t);
   fftw_plan_with_nthreads(t);
```

collecting running times

```
$ time /tmp/fftw_timing_omp 100000 1000 p
```

Computing 1,000 DFTs for $n = 100,000$, with p threads:

p	real	user	sys	speed up
1	2.443s	2.436s	0.004s	1
2	1.340s	2.655s	0.010s	1.823
4	0.774s	2.929s	0.008s	3.156
8	0.460s	3.593s	0.017s	5.311
16	0.290s	4.447s	0.023s	8.424

where real = wall clock time, user = cpu time, sys = system time, obtained with `time` on 2 8-core processors at 2.60 GHz.

collecting more running times

```
$ time /tmp/fftw_timing_omp 1000000 1000 p
```

Computing 1,000 DFTs for $n = 1,000,000$, with p threads:

p	real	user	sys	speed up
1	44.806s	44.700s	0.014s	1
2	22.151s	44.157s	0.020s	2.023
4	10.633s	42.336s	0.019s	4.214
8	6.998s	55.630s	0.036s	6.403
16	3.942s	62.250s	0.134s	11.366

where real = wall clock time, user = cpu time, sys = system time, obtained with `time` on 2 8-core processors at 2.60 GHz.

As n increases ten fold, the speedups improve.

how FFTW3 uses SIMD

SIMD = Single Instruction Multiple Data.
SIMD instructions perform the same operation in parallel
on a data vector.

- The planner computes an executable data structure,
  tuned towards the specific microprocessor capabilities.
- FFTW calls a special-purpose compiler
  to generate optimized code.
- The standard distribution contains a set of 150 pre-generated
  "codelets" that cover the most common uses.

After generating the plan, it is executed as many times as requested.

Parallel FFT and Sorting

1. The Fast Fourier Transform in Parallel
   - the Fastest Fourier Transform in the West (FFTW)
   - running the OpenMP version of FFTW
2. Bucket Sort for Distributed Memory
   - bucket sort in parallel
   - communication versus computation
3. Quicksort for Shared Memory
   - partitioning numbers
   - quicksort with OpenMP
4. Radix Sort for GPU acceleration
   - parallel radix sort

bucket sort

Given are n numbers, suppose all are in $[0, 1]$.

The algorithm using p buckets proceeds in two steps:

- Partition numbers x into p buckets:
  $x \in [i/p, (i+1)/p[\quad \Rightarrow \quad x \in (i+1)$th bucket.
- Sort all p buckets.

The cost to partition the numbers into p buckets is $O(n \log_2(p))$.
Note: radix sort uses most significant bits to partition.

In the best case: every bucket contains n/p numbers.
The cost of Quicksort is $O(n/p \log_2(n/p))$ per bucket.
Sorting p buckets takes $O(n \log_2(n/p))$.

Total cost is $O(n(\log_2(p) + \log_2(n/p)))$.

parallel bucket sort

On p processors, all nodes sort:

1. Root node distributes numbers: processor i gets ith bucket.
2. Processor i sorts ith bucket.
3. Root node collects sorted buckets from processors.

Is it worth it? Recall: serial cost is $n(\log_2(p) + \log_2(n/p))$.
Cost of parallel algorithm:

- $n \log_2(p)$ to place numbers into buckets,
- $n/p \log_2(n/p)$ to sort buckets.

$$\begin{aligned} \text{speedup} &= \frac{n(\log_2(p) + \log_2(n/p))}{n(\log_2(p) + \log_2(n/p)/p)} \\ &= \frac{1+L}{1+L/p} = \frac{1+L}{(p+L)/p} = \frac{p}{p+L}(1+L), \quad L = \frac{\log_2(n/p)}{\log_2(p)}. \end{aligned}$$

comparing to quicksort

$$\begin{aligned}
\text{speedup} &= \frac{n\log_2(n)}{n(\log_2(p) + n/p\log_2(n/p))} \\
&= \frac{\log_2(n)}{\log_2(p) + 1/p(\log_2(n) - \log_2(p))} \\
&= \frac{\log_2(n)}{1/p(\log_2(n) + (1 - 1/p)\log_2(p))}
\end{aligned}$$

Example: $n = 2^{20}$, $\log_2(n) = 20$, $p = 2^2$, $\log_2(p) = 2$,

$$\begin{aligned}
\text{speedup} &= \frac{20}{1/4(20) + (1 - 1/4)2} \\
&= \frac{20}{5 + 3/2} = \frac{40}{13} \approx 3.08.
\end{aligned}$$

Parallel FFT and Sorting

1. The Fast Fourier Transform in Parallel
   - the Fastest Fourier Transform in the West (FFTW)
   - running the OpenMP version of FFTW
2. Bucket Sort for Distributed Memory
   - bucket sort in parallel
   - communication versus computation
3. Quicksort for Shared Memory
   - partitioning numbers
   - quicksort with OpenMP
4. Radix Sort for GPU acceleration
   - parallel radix sort

communication and computation

The scatter of n data elements costs $t_{\text{start up}} + nt_{\text{data}}$, where t_{data} is the cost of sending 1 data element.

For distributing and collecting of all buckets, the total communication time is $2p\left(t_{\text{start up}} + \frac{n}{p}t_{\text{data}}\right)$.

The computation/communication ratio is

$$\frac{(n\log_2(p) + n/p\log_2(n/p))t_{\text{compare}}}{2p\left(t_{\text{start up}} + \frac{n}{p}t_{\text{data}}\right)}$$

where t_{compare} is the cost for one comparison.

the computation/communication ratio

The computation/communication ratio is

$$\frac{(n\log_2(p) + n/p\log_2(n/p))t_{\text{compare}}}{2p\left(t_{\text{start up}} + \frac{n}{p}t_{\text{data}}\right)}$$

where t_{compare} is the cost for one comparison.

We view this ratio for $n \gg p$, for fixed p, so:

$$\frac{n}{p}\log_2\left(\frac{n}{p}\right) = \frac{n}{p}(\log_2(n) - \log_2(p)) \approx \frac{n}{p}\log_2(n).$$

The ratio then becomes $\frac{n}{p}\log_2(n)t_{\text{compare}} \gg 2nt_{\text{data}}$.

Thus $\log_2(n)$ must be sufficiently high...

Parallel FFT and Sorting

1. The Fast Fourier Transform in Parallel
   - the Fastest Fourier Transform in the West (FFTW)
   - running the OpenMP version of FFTW
2. Bucket Sort for Distributed Memory
   - bucket sort in parallel
   - communication versus computation
3. Quicksort for Shared Memory
   - partitioning numbers
   - quicksort with OpenMP
4. Radix Sort for GPU acceleration
   - parallel radix sort

a recursive algorithm

```
void quicksort ( double *v, int start, int end ) {
   if(start < end) {
      int pivot;
      partition(v,start,end,&pivot);
      quicksort(v,start,pivot-1);
      quicksort(v,pivot+1,end);
   }
}
```

where `partition` has the prototype:

```
void partition
 ( double *v, int lower, int upper, int *pivot );
/* precondition: upper - lower > 0
 * takes v[lower] as pivot and interchanges elements:
 * v[i] <= v[pivot] for all i < pivot, and
 * v[i] > v[pivot] for all i > pivot,
 * where lower <= pivot <= upper. */
```

a `partition` function

```
void partition
  ( double *v, int lower, int upper, int *pivot )
{
   double x = v[lower];
   int up = lower+1;   /* index will go up */
   int down = upper;   /* index will go down */
   while(up < down)
   {
      while((up < down) && (v[up] <= x)) up++;
      while((up < down) && (v[down] > x)) down--;
      if(up == down) break;
      double tmp = v[up];
      v[up] = v[down]; v[down] = tmp;
   }
   if(v[up] > x) up--;
   v[lower] = v[up]; v[up] = x;
   *pivot = up;
}
```

partition and qsort in main()

```
int lower = 0;
int upper = n-1;
int pivot = 0;
if(n > 1) partition(v,lower,upper,&pivot);

if(pivot != 0)
   qsort((void*)v,(size_t)pivot,
         sizeof(double),compare);

if(pivot != n)
   qsort((void*)&v[pivot+1],(size_t)(n-pivot-1),
         sizeof(double),compare);
```

Parallel FFT and Sorting

1. The Fast Fourier Transform in Parallel
   - the Fastest Fourier Transform in the West (FFTW)
   - running the OpenMP version of FFTW
2. Bucket Sort for Distributed Memory
   - bucket sort in parallel
   - communication versus computation
3. Quicksort for Shared Memory
   - partitioning numbers
   - quicksort with OpenMP
4. Radix Sort for GPU acceleration
   - parallel radix sort

a parallel region in `main()`

```
omp_set_num_threads(2);
#pragma omp parallel
{
   if(pivot != 0)
      qsort((void*)v,(size_t)pivot,
            sizeof(double),compare);
   if(pivot != n)
      qsort((void*)&v[pivot+1],(size_t)(n-pivot-1),
            sizeof(double),compare);
}
```

on dual core Mac OS X at 2.26 GHz

```
$ time /tmp/time_qsort 10000000 0
time elapsed : 4.0575 seconds

real    0m4.299s
user    0m4.229s
sys     0m0.068s

$ time /tmp/part_qsort_omp 10000000 0
pivot = 4721964
-> sorting the first half : 4721964 numbers
-> sorting the second half : 5278035 numbers

real    0m3.794s
user    0m7.117s
sys     0m0.066s
```

Speed up: 4.299/3.794 = 1.133, or 13.3% faster with one extra core.

Parallel FFT and Sorting

1. The Fast Fourier Transform in Parallel
   - the Fastest Fourier Transform in the West (FFTW)
   - running the OpenMP version of FFTW
2. Bucket Sort for Distributed Memory
   - bucket sort in parallel
   - communication versus computation
3. Quicksort for Shared Memory
   - partitioning numbers
   - quicksort with OpenMP
4. Radix Sort for GPU acceleration
   - parallel radix sort

radix sort on 16 4-bit numbers

Step 1:

12	3	6	9	15	8	5	10	9	6	11	13	4	10	7	0
1100	0011	0110	1001	1111	1000	0101	1010	1001	0110	1011	1101	0100	1010	0111	0000
1100		0110			1000		1010		0110			0100	1010		0000
	0011		1001	1111		0101		1001		1011	1101			0111	
1100	0110	1000	1010	0110	0100	1010	0000	0011	1001	1111	0101	1001	1011	1101	0111

Place the numbers in two buckets, along the last bit.
Within each bucket, the original order is preserved.

radix sort on 16 4-bit numbers

Step 2:

12	3	6	9	15	8	5	10	9	6	11	13	4	10	7	0
1100	0011	0110	1001	1111	1000	0101	1010	1001	0110	1011	1101	0100	1010	0111	0000
1100		0110			1000		1010		0110			0100	1010		0000
	0011		1001	1111		0101		1001		1011	1101			0111	
1100	**0110**	**1000**	**1010**	**0110**	**0100**	**1010**	**0000**	**0011**	**1001**	**1111**	**0101**	**1001**	**1011**	**1101**	**0111**
1100		**1000**			**0100**		**0000**		**1001**		**0101**	**1001**		**1101**	
	0110		**1010**	**0110**		**1010**		**0011**		**1111**			**1011**		**0111**
1100	**1000**	**0100**	**0000**	**1001**	**0101**	**1001**	**1101**	**0110**	**1010**	**0110**	**1010**	**0011**	**1111**	**1011**	**0111**

Place the numbers in two buckets, along the second-to-last bit.
Within each bucket, the original order is preserved.

radix sort on 16 4-bit numbers

Step 3:

12	3	6	9	15	8	5	10	9	6	11	13	4	10	7	0
1100	0011	0110	1001	1111	1000	0101	1010	1001	0110	1011	1101	0100	1010	0111	0000
1100		0110			1000		1010		0110			0100	1010		0000
	0011		1001	1111		0101		1001		1011	1101			0111	
1100	0110	1000	1010	0110	0100	1010	0000	0011	1001	1111	0101	1001	1011	1101	0111
1100		1000			0100		0000		1001		0101	1001		1101	
	0110		1010	0110		1010		0011		1111			1011		0111
1100	**1000**	**0100**	**0000**	**1001**	**0101**	**1001**	**1101**	**0110**	**1010**	**0110**	**1010**	**0011**	**1111**	**1011**	**0111**
	1000		**0000**	**1001**		**1001**			**1010**		**1010**	**0011**		**1011**	
1100		**0100**			**0101**		**1101**	**0110**		**0110**			**1111**		**0111**
1000	**0000**	**1001**	**1001**	**1010**	**1010**	**0011**	**1011**	**1100**	**0100**	**0101**	**1101**	**0110**	**0110**	**1111**	**0111**

Place the numbers in two buckets, along the third-to-last bit.
Within each bucket, the original order is preserved.

radix sort on 16 4-bit numbers

Step 4:

12	3	6	9	15	8	5	10	9	6	11	13	4	10	7	0
1100	0011	0110	1001	1111	1000	0101	1010	1001	0110	1011	1101	0100	1010	0111	0000
1100		0110			1000		1010		0110			0100	1010		0000
	0011		1001	1111		0101		1001		1011	1101			0111	
1100	0110	1000	1010	0110	0100	1010	0000	0011	1001	1111	0101	1001	1011	1101	0111
1100		1000			0100		0000		1001		0101	1001		1101	
	0110		1010	0110		1010		0011		1111			1011		0111
1100	1000	0100	0000	1001	0101	1001	1101	0110	1010	0110	1010	0011	1111	1011	0111
	1000		0000	1001		1001			1010		1010	0011		1011	
1100		0100			0101		1101	0110		0110			1111		0111
1000	**0000**	**1001**	**1001**	**1010**	**1010**	**0011**	**1011**	**1100**	**0100**	**0101**	**1101**	**0110**	**0110**	**1111**	**0111**
	0000					**0011**			**0100**	**0101**		**0110**	**0110**		**0111**
1000		**1001**	**1001**	**1010**	**1010**		**1011**	**1100**			**1101**			**1111**	
0000	**0011**	**0100**	**0101**	**0110**	**0110**	**0111**	**1000**	**1001**	**1001**	**1010**	**1010**	**1011**	**1100**	**1101**	**1111**
0	3	4	5	6	6	7	8	9	9	10	10	11	12	13	15

By the invariant: *within each bucket, the original order is preserved*, the list is sorted.

parallel radix sort

On 16 numbers, work with 16 threads, where the *i*-th thread computes the position of the *i*-th input number in the output.

(0) Mapping to a zero bucket:

$$\begin{aligned} \text{destination of a zero} &= \#\text{zeros before}, \\ &= \#\text{keys before} - \#\text{ones before}, \\ &= \text{key index} - \#\text{ones before}. \end{aligned}$$

(1) Mapping to a one bucket:

$$\begin{aligned} \text{destination of a one} &= \#\text{zeros in total} + \#\text{ones before}, \\ &= \#\text{keys in total} - \#\text{ones in total} + \#\text{ones before}, \\ &= \text{input size} - \#\text{ones in total} + \#\text{ones before}. \end{aligned}$$

The nontrivial computation is counting the number of keys before it that map to the 1 bucket. This is done by an *exclusive scan*.

an exclusive scan

The exclusive scan counts the number of ones before.

12	3	6	9	15	8	5	10	9	6	11	13	4	10	7	0
1100	0011	0110	1001	1111	1000	0101	1010	1001	0110	1011	1101	0100	1010	0111	0000
1100		0110			1000		1010		0110			0100	1010		0000
	0011		1001	1111		0101		1001		1011	1101			0111	
0	1	0	1	1	0	1	0	1	0	1	1	0	0	1	0

output of the exclusive scan :

0	0	1	1	2	3	3	4	4	5	5	6	7	7	7	8

destination of a zero $=$ key index $-$ #ones before

destination of a one $=$ input size $-$ #ones in total $+$ #ones before

0	8	1	9	10	2	11	3	12	4	13	14	5	6	15	7

after step 1:

1100	0110	1000	1010	0110	0100	1010	0000	0011	1001	1111	0101	1001	1011	1101	0111

Memory Coalescing and Thread Coarsening

The exclusive scan is similar to the prefix sum scan.

One major inefficiency is the writing to the output,
which has an access pattern that cannot be memory coalesced.
A local sort in shared memory improves memory coalescing.

Memory coalescing is also improved by thread coarsening,
when each thread is assigned to multiple keys instead of one.

GPU accelerated sorting is provided by Thrust, see:
N. Bell and J. Hoberock: **Thrust: a productivity-oriented library for CUDA.** In *GPU Computing Gems Jade Edition.* Pages 359-371, Morgan Kaufmann, 2012.

Chapter 13 in the fourth edition of *Programming Massively Parallel Programming* is entirely devoted to sorting.

recommended reading

- Matteo Frigo and Steven G. Johnson: **The Design and Implementation of FFTW3**. *Proc. IEEE* 93(2): 216–231, 2005.
- Edgar Solomonik and Laxmikant V. Kale: **Highly Scalable Parallel Sorting.** In the proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010.
- Mirko Rahn, Peter Sanders, and Johannes Singler: **Scalable Distributed-Memory External Sorting**. In the proceedings of the 26th IEEE International Conference on Data Engineering (ICDE), pages 685-688, IEEE, 2010.
- Davide Pasetto and Albert Akhriev: **A Comparative Study of Parallel Sort Algorithms**. In SPLASH'11, the proceedings of the ACM international conference companion on object oriented programming systems languages and applications, pages 203-204, ACM 2011.

Summary + Exercises

In the book of Wilkinson and Allen, bucket sort is described in §4.2.1 and chapter 10 is entirely devoted to sorting algorithms.

Exercises:

1. Consider the fan out scatter and fan in gather operations and investigate how these operations will reduce the communication cost and improve the computation/communication ratio in bucket sort of n numbers on p processors.
2. Use Julia to implement radix sort in parallel with tasking.
3. Implement parallel radix sort on a GPU using Julia.