Recitation 4:

OpenMP Programming

15-418 Parallel Computer Architecture and Programming

CMU 15-418/15-618, Spring 2019

Goals for today

- Learn to use Open MP

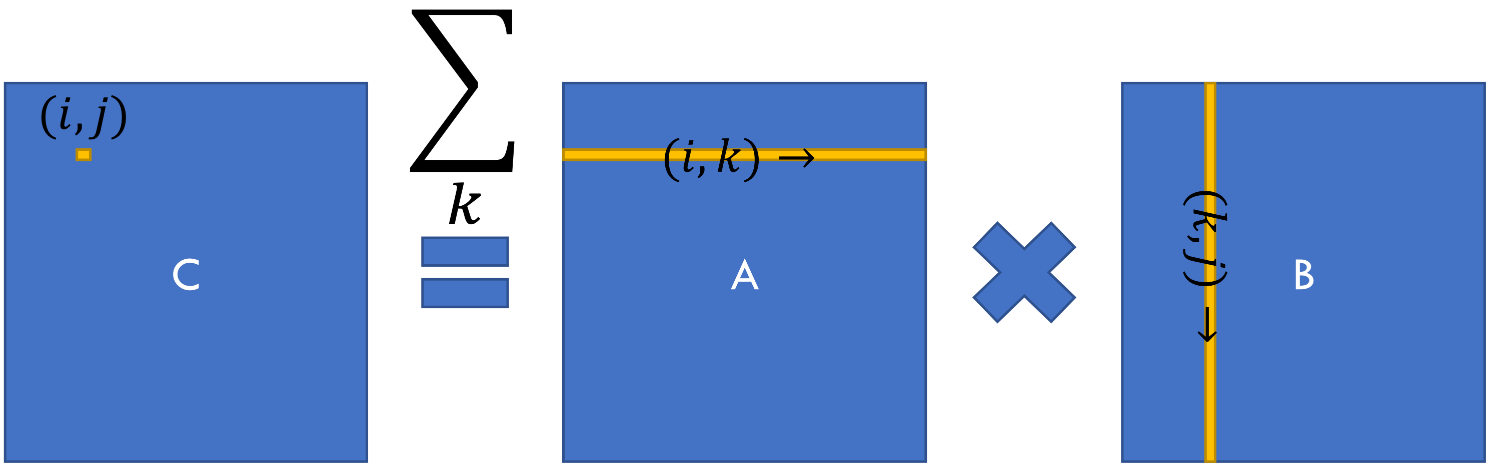
1. Sparse matrix-vector code
   - Understand “CSR” sparse matrix format
   - Simplest OpenMP features
2. Parallelize array sum via OpenMP reductions
3. Parallelize radix sort
   - More complex OpenMP example

- Most of all,

<u>ANSWER YOUR QUESTIONS!</u>

Matrix-matrix multiplication

- Recall from recitation 2…



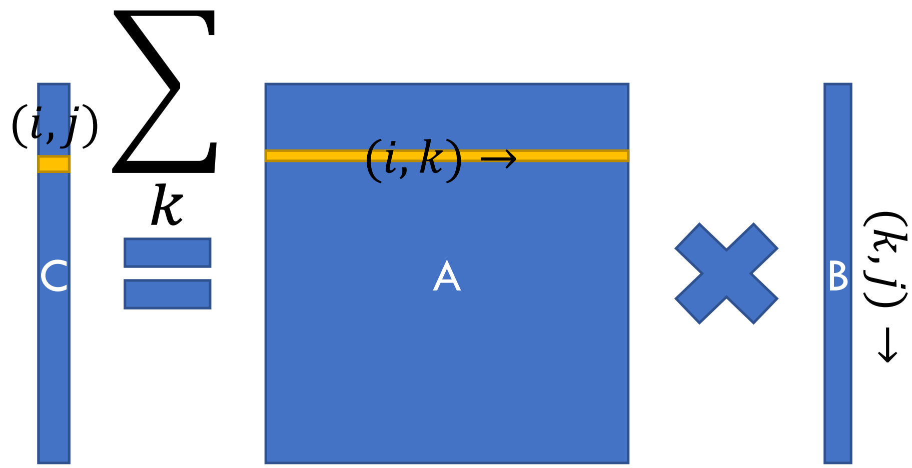
Matrix-matrix multiplication (matmul)

- Simple C++ implementation:

```
/* Find element based on row-major ordering */
#define RM(r, c, width) ((r) * (width) + (c))

// Standard multiplication
void multMatrixSimple(int N, float *matA, float *matB, float *matC) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            float sum = 0.0;
            for (int k = 0; k < N; k++)
                sum += matA[RM(i,k,N)] * matB[RM(k,j,N)];
            matC[RM(i,j,N)] = sum;
        }
}
```

Today: Matrix-vector multiplication



- $(n \times n) \times (n \times 1) \Rightarrow (n \times 1)$ output vector
- Output = dot-products of rows from A and the vector B

Matrix-vector multiplication

- Simple C++ implementation (discards *j* loop from matmul):

```
/* Find element based on row-major ordering */
#define RM(r, c, width) ((r) * (width) + (c))

void matrixVectorProduct(int N, float *matA, float *vecB, float *vecC) {
    for (int i = 0; i < N; i++)
        float sum = 0.0;
        for (int k = 0; k < N; k++)
            sum += matA[RM(i,k,N)] * vecB[k];
        vecC[i] = sum;
    }
}
```

Matrix-vector multiplication

- Our code is slightly refactored:

```
float rvp_dense_seq(dense_t *m, vec_t *x, index_t r) {
    index_t nrow = m->nrow;
    index_t idx = r*nrow;
    float val = 0.0;
    for (index_t c = 0; c < nrow; c++)
        val += x->value[c] * m->value[idx++];
    return val;
}
```

Row dot product (the inner loop over k in original code)

```
void mvp_dense_seq(dense_t *m, vec_t *x, vec_t *y, rvp_dense_t rp_fun) {
    index_t nrow = m->nrow;
    for (index_t r = 0; r < nrow; r++) {
        y->value[r] = rp_fun(m, x, r);
    }
}
```

The inner loop over rows (over i in original code)

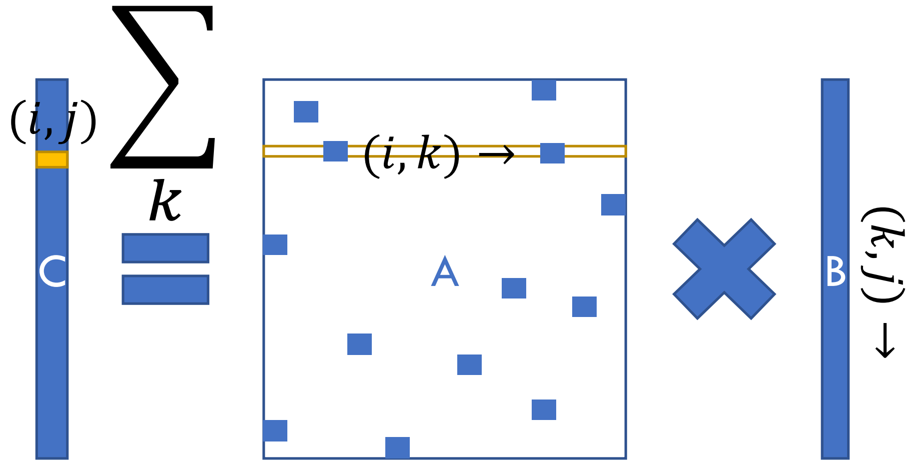
Benchmarking dense mat-vec

```
$ ./mrun -l 3 -d 10 -t r -D
Dense    3    10   r
     MVP  RVP  GF
     seq  seq  0.12
```

- 0.12 GFLops … machine capable of 6.4 Gflops
  ➔ This is bad performance
- Why? We are only counting non-zero entries of matrix

Sparse matrix-vector multiplication

- What if A is mostly zeroes? (This is common)



- Idea: We should only compute on non-zeros in A
- ➔ Need new <u>sparse</u> matrix representation

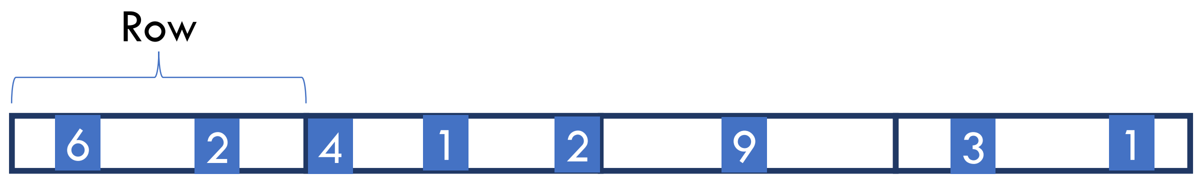
Compressed sparse-row (CSR) matrix format

- Dense matrix:

- CSR matrix:

Compressed sparse-row (CSR) matrix format

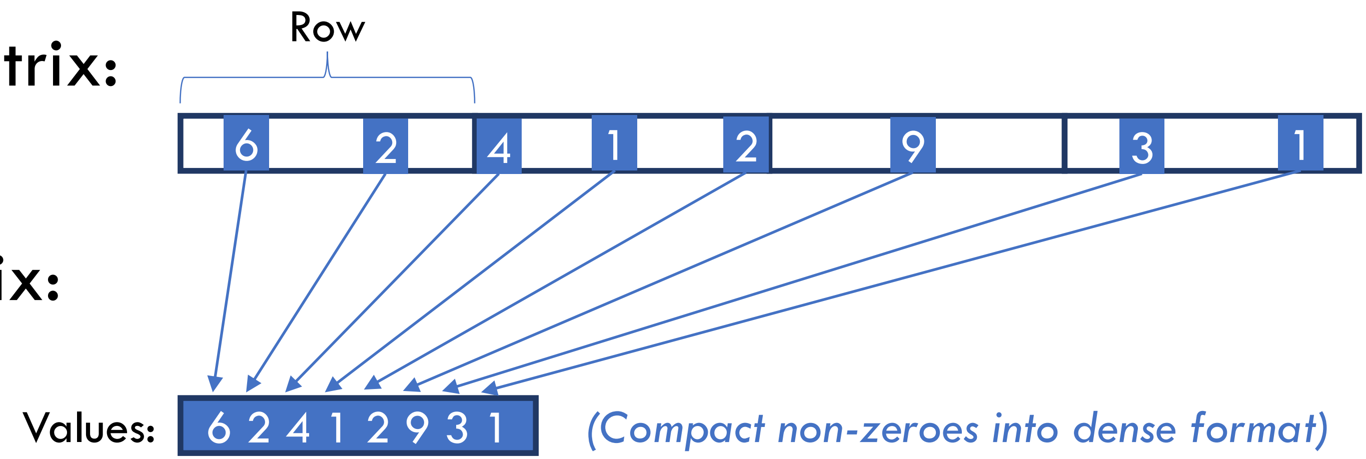
- Dense matrix:



- CSR matrix:

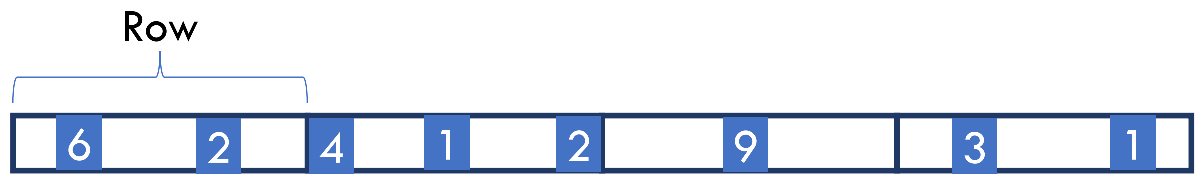
Compressed sparse-row (CSR) matrix format

- Dense matrix:
- CSR matrix:

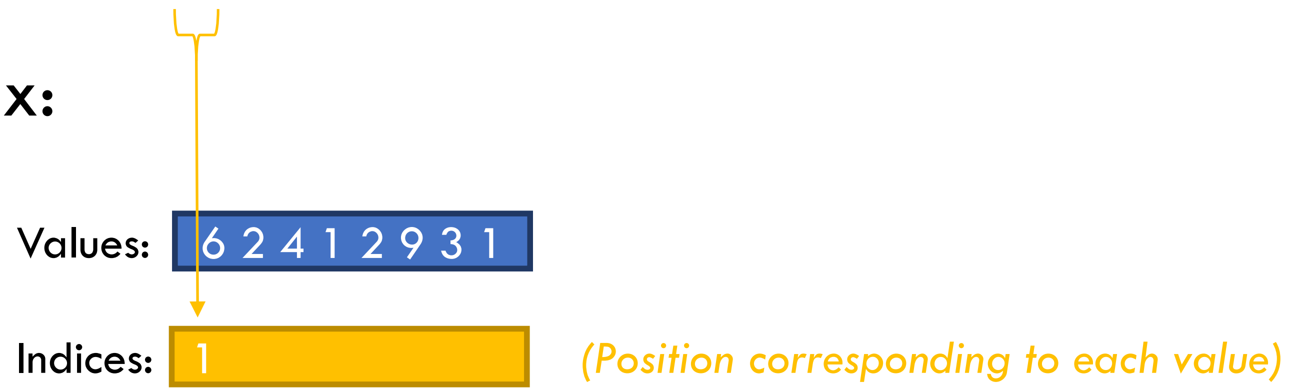


Compressed sparse-row (CSR) matrix format

- Dense matrix:



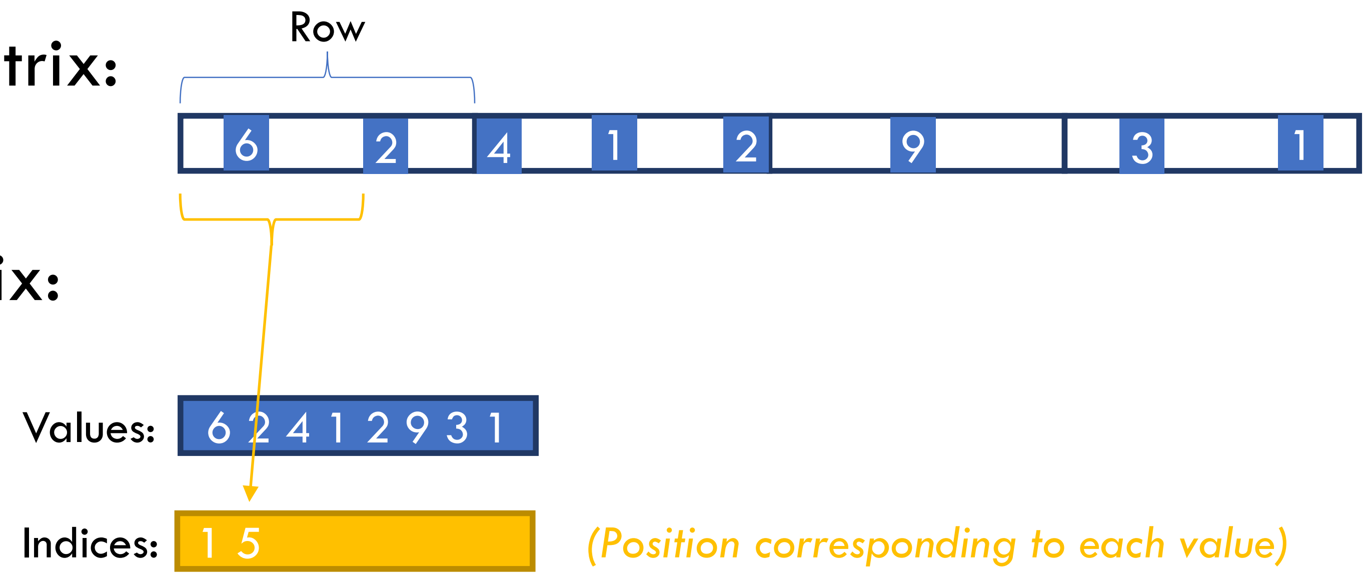
- CSR matrix:



Compressed sparse-row (CSR) matrix format

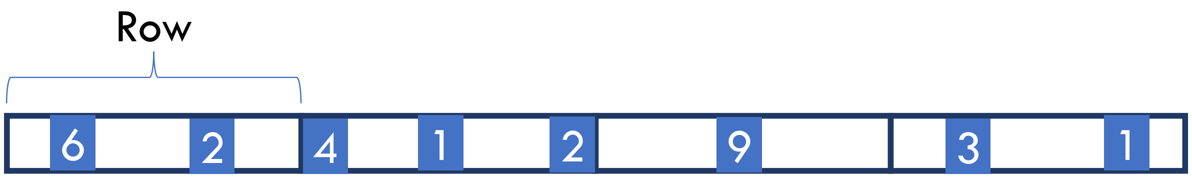
- Dense matrix:

- CSR matrix:

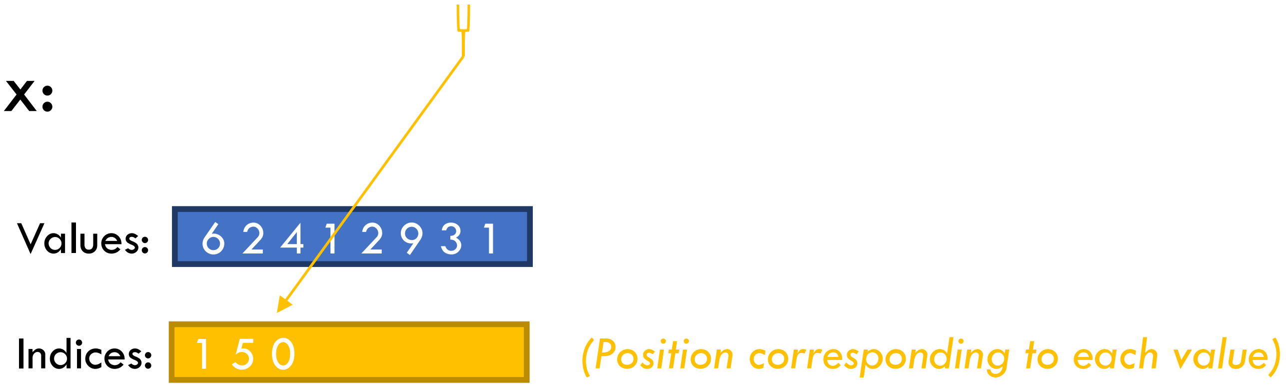


Compressed sparse-row (CSR) matrix format

- Dense matrix:

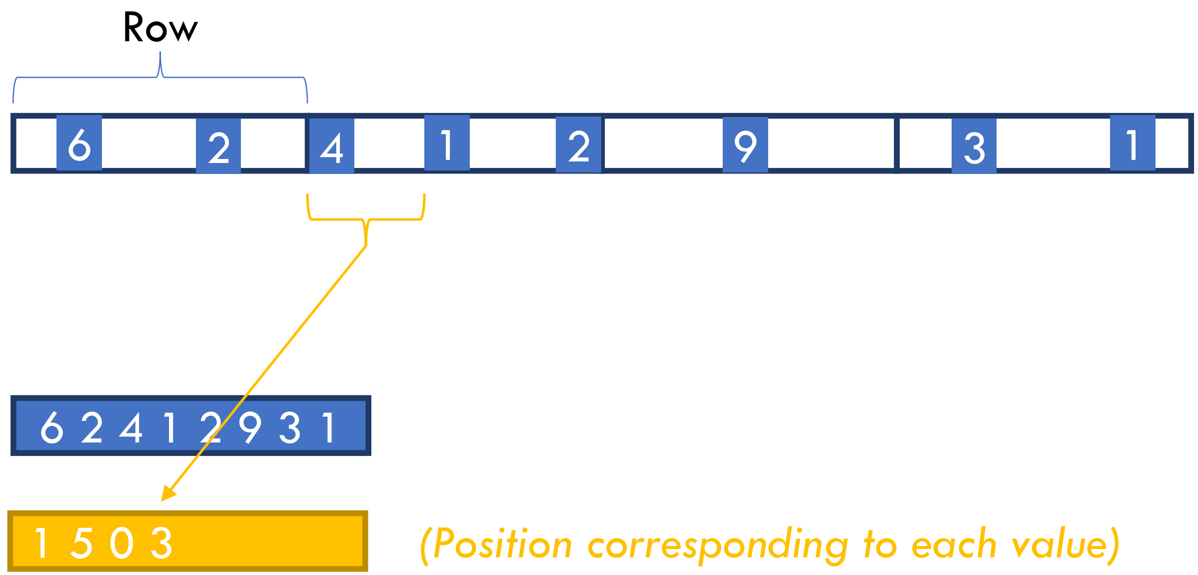


- CSR matrix:



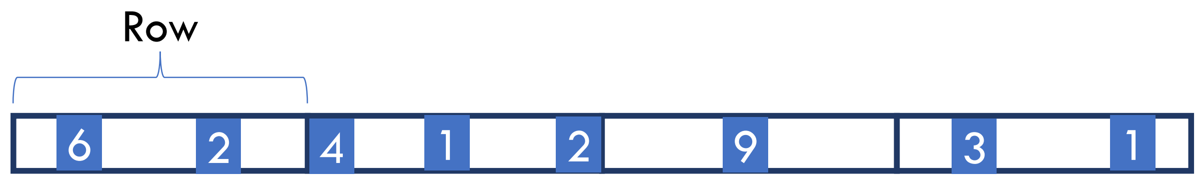
Compressed sparse-row (CSR) matrix format

- Dense matrix:
- CSR matrix:



Compressed sparse-row (CSR) matrix format

- Dense matrix:



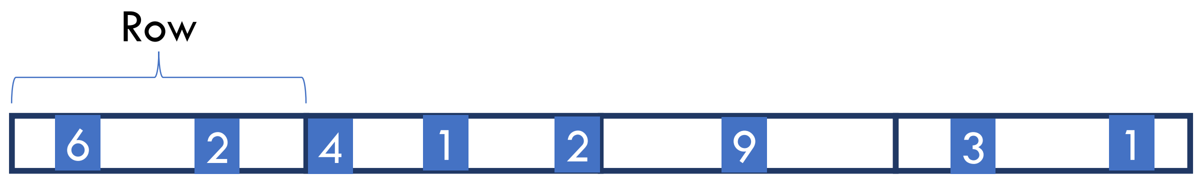
- CSR matrix:

Values: 6 2 4 1 2 9 3 1

Indices: 1 5 0 3 7 4 1 6 *(Position corresponding to each value)*

Compressed sparse-row (CSR) matrix format

- Dense matrix:



- CSR matrix:

Values: 6 2 4 1 2 9 3 1

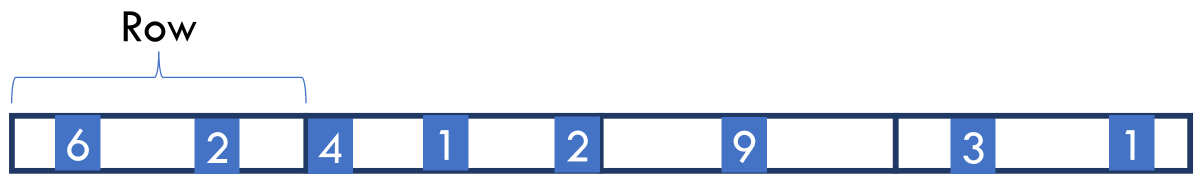
Indices: 1 5 0 3 7 4 1 6

Offsets:

(Where each row starts)

Compressed sparse-row (CSR) matrix format

- Dense matrix:



- CSR matrix:

Values: 6 2 4 1 2 9 3 1

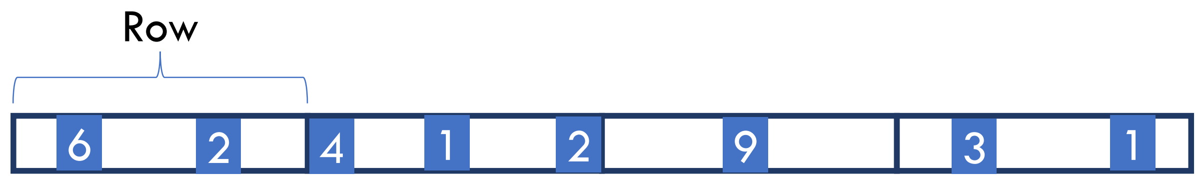
Indices: 1 5 0 3 7 4 1 6

Offsets: 0

(Where each row starts)

Compressed sparse-row (CSR) matrix format

- Dense matrix:



- CSR matrix:

Values: 6 2 4 1 2 9 3 1

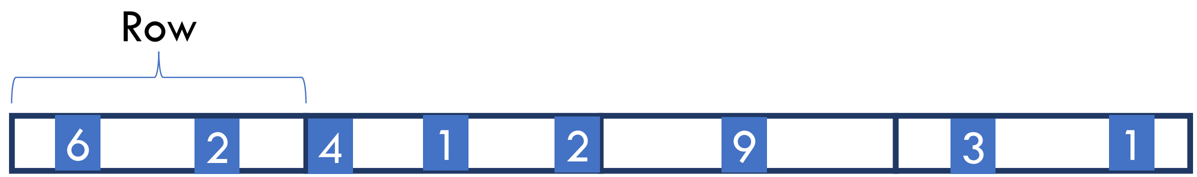
Indices: 1 5 0 3 7 4 1 6

Offsets: 0 2

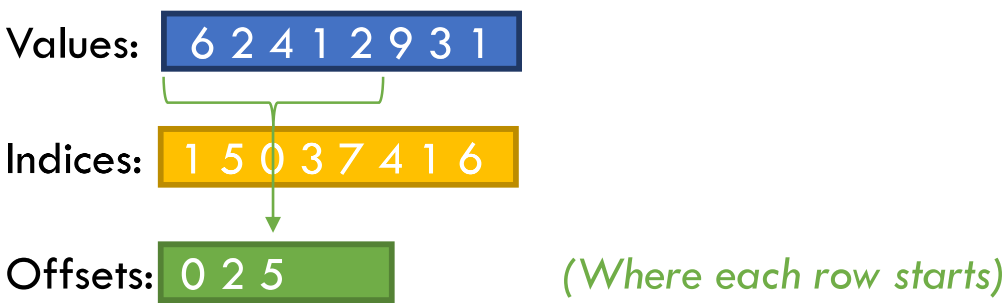
(Where each row starts)

Compressed sparse-row (CSR) matrix format

- Dense matrix:

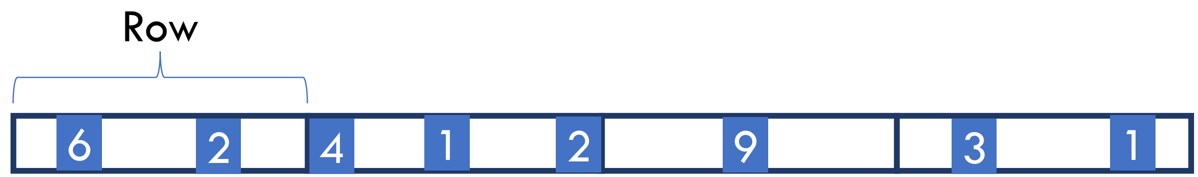


- CSR matrix:



Compressed sparse-row (CSR) matrix format

- Dense matrix:



- CSR matrix:

Values: 6 2 4 1 2 9 3 1

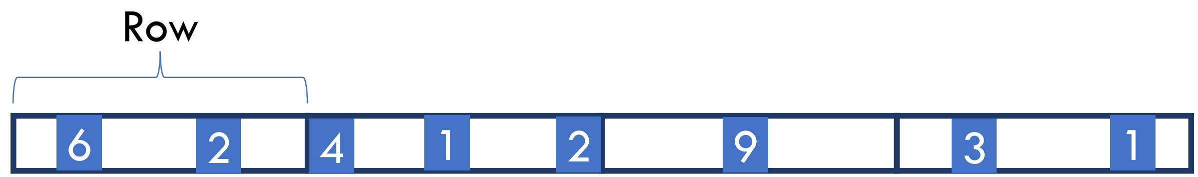
Indices: 1 5 0 3 7 4 1 6

Offsets: 0 2 5 6

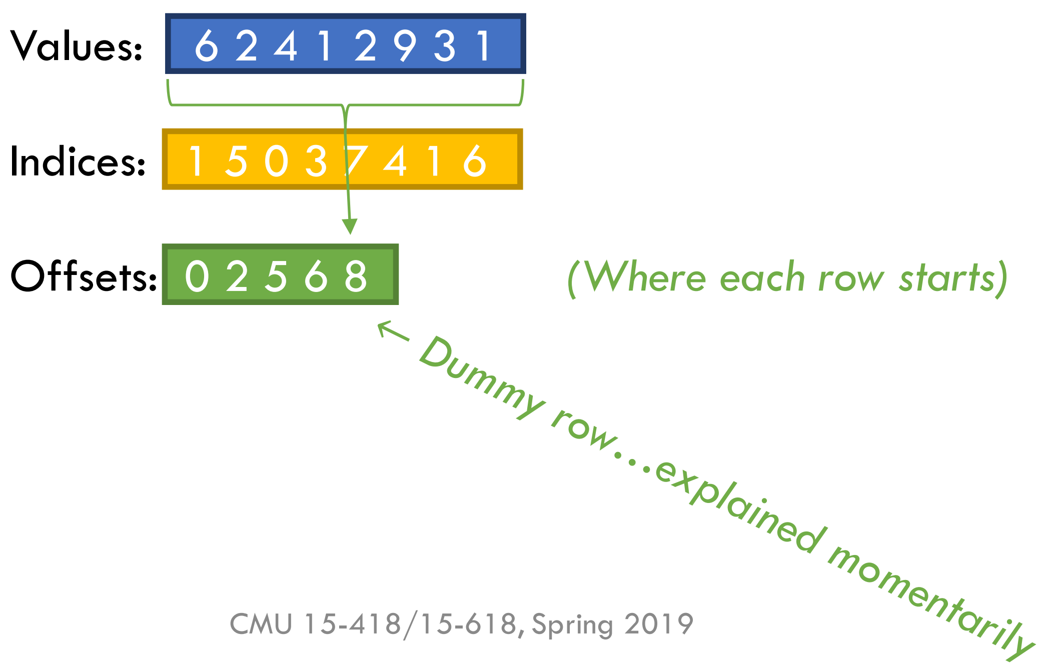
(Where each row starts)

Compressed sparse-row (CSR) matrix format

- Dense matrix:

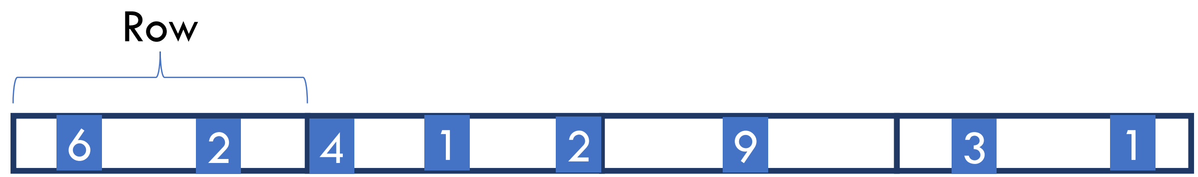


- CSR matrix:



Compressed sparse-row (CSR) matrix format

- Dense matrix:



- CSR matrix:

Values: 6 2 4 1 2 9 3 1 *(Compact non-zeroes into dense format)*

Indices: 1 5 0 3 7 4 1 6 *(Position corresponding to each value)*

Offsets: 0 2 5 6 8 *(Where each row starts)*

Sparse matrix-vector multiplication (spmv)

```
float rvp_csr_seq(csr_t *m, vec_t *x, index_t r) {
    index_t idxmin = m->rowstart[r];    /* 1. get indices from offsets */
    index_t idxmax = m->rowstart[r+1];

    float val = 0.0;
    for (index_t idx = idxmin; idx < idxmax; idx++) {  /* 2. iterate over row */
        index_t c = m->cindex[idx];     /* 3. find corresponding index in vector */
        data_t mval = m->value[idx];    /* read matrix (using idx) */
        data_t xval = x->value[c];      /* read vector (using c) */
        val += mval * xval;
    }
    return val;
}


/* the outer loop (across rows) doesn't change */
void mvp_csr_seq(csr_t *m, vec_t *x, vec_t *y, rvp_csr_t rp_fun) {
    index_t nrow = m->nrow;
    for (index_t r = 0; r < nrow; r++) {
        y->value[r] = rp_fun(m, x, r);
    }
}
```

Benchmarking dense mat-vec

```
$ ./mrun -l 3 -d 10 -t r –D
Dense    3   10  r
     MVP RVP GF
     seq seq 0.12
```

Benchmarking spmv

```
$ ./mrun -l 3 -d 10 -t r
Sparse   3   10   r
     MVP  RVP  GF
     seq  seq  1.15
```

- This matrix is 10% dense (-d parameter)
- CSR gives 9.6× speedup!

Let’s optimize this code!

Exploit parallelism at multiple levels

- ILP
- SIMD
- Threads

To properly measure parallel speedup, need to start from an optimized baseline

What's the ILP of spmv?

- GHC machines: 2x LD @ 1 cycle & 2x FMA @ 3 cycle
- Q: How many operations per iteration?
- A: 3 loads, 1 FMA
- Iteration takes at least: 3 cycles (latency bound)
- Expected: $\frac{3.2\ GHz}{3\ cycles\ per\ iteration} = 1.07\ GFlops$
  - Pretty close to what we observe
- Machine utilization low: FMA $\approx$ 17%, LD $\approx$ 50%

Improving spmv ILP

Note: gcc with –O3 will unroll these loops for us

```
#define UNROLL 3
float rvp_csr_par(csr_t *m, vec_t *x, index_t r) {
    index_t idxmin = m->rowstart[r], idxmax = m->rowstart[r+1];
    index_t idx;
    float uval[UNROLL];
    float val = 0.0;
    for (index_t i = 0; i < UNROLL; i++)
        uval[i] = 0.0;
    for (idx = idxmin; idx+UNROLL <= idxmax; idx+=UNROLL) {
        for (index_t i = 0; i < UNROLL; i++) {
            index_t c = m->cindex[idx+i];
            data_t mval = m->value[idx+i];
            data_t xval = x->value[c];
            uval[i] += mval * xval;
        }
    }
    for (; idx < idxmax; idx ++) {
        index_t c = m->cindex[idx];
        data_t mval = m->value[idx];
        data_t xval = x->value[c];
        val += mval * xval;
    }

    for (index_t i = 0; i < UNROLL; i++)
        val += uval[i];
    return val;
}
```

Accumulate elements mod UNROLL in uval array

Return sum of uval array

Benchmarking spmv

```
$ ./mrun -l 4 -d 10 -t r
Sparse   4   10   r
     MVP RVP GF
     seq seq 1.40
```

Benchmarking unrolled spmv

```
$ ./mrun -l 4 -d 10 -t r
Sparse   4   10   r
     MVP  RVP  GF
     seq  seq  1.40
     seq  par  2.72
```

- 3× unrolling gives ≈ 2 × speedup
- More unrolling doesn't help (3 slightly better than 2)
- Why? *With 100% hits, LD utilization = 100% when unrolled twice*
  Unrolling thrice helps a bit, probably by hiding misses on sparse vector loads

Let’s optimize this code!

Exploit parallelism at multiple levels

- ILP ✓
- SIMD
- Threads

Thread parallelism with OpenMP

- OpenMP is supported by gcc

- “Decorate” your code with `#pragmas`

- We will cover only a few of OpenMP’s features

Parallelizing spmv with OpenMP

- Focus on the outer loop

```
void mvp_csr_seq(csr_t *m, vec_t *x, vec_t *y,
                 rvp_csr_t rp_fun) {
    index_t nrow = m->nrow;

    for (index_t r = 0; r < nrow; r++) {
        y->value[r] = rp_fun(m, x, r);
    }
}
```

Parallelizing spmv with OpenMP

- Focus on the outer loop

```
void mvp_csr_mps(csr_t *m, vec_t *x, vec_t *y,
                 rvp_csr_t rp_fun) {
    index_t nrow = m->nrow;
#pragma omp parallel for schedule(static)
    for (index_t r = 0; r < nrow; r++) {
        y->value[r] = rp_fun(m, x, r);
    }
}
```

- `schedule(static)` divides loop statically across system cores (including hyperthreads)

Benchmarking threaded spmv

```
$ ./mrun -l 4 -d 10 -t r
Sparse   4   10   r
    MVP  RVP  GF
    seq  seq  1.40
    seq  par  2.72
...
    mps  seq  9.92
    mps  par  10.44
```

- Threading gives 7× for naive, 4× for unrolled
- We have 8x2-hyperthread = 16 cores

Analyzing poor threading performance

- Sources of poor parallel speedup:

- Communication/synchronization? *No, static work division is ≈zero overhead*

- Throughput limitations? *Not execution units… 'mps/seq' has poor FMA utilization, and 'mps/par' only gets 4× speedup from 8 cores*

- Load imbalance? *Maybe…how many elements are there per row? A: its random*

Benchmarking threaded spmv

```
$ ./mrun -l 4 -d 10 -t s
Sparse    4    10   s
     MVP  RVP  GF
     seq  seq  1.40
     seq  par  3.30
...
     mps  seq  2.10
     mps  par  4.24
```

- Increasing skew (all non-zero elements are in first 10% of rows) reduces speedup significantly
- Single-thread is faster (why?), OpenMP is slower (why?)

Benchmarking threaded spmv

```
$ ./mrun -l 4 -d 10 -t u
Sparse    4    10   u
     MVP  RVP  GF
     seq  seq  1.45
     seq  par  2.63
...
     mps  seq  10.08
     mps  par  10.51
```

- …But forcing all rows to have the same number of non-zero elements doesn't change much vs. random
- Hypothesis: Memory bandwidth saturated?

Parallelizing spmv with OpenMP

- Dealing with skewed workloads

```
void mvp_csr_mps(csr_t *m, vec_t *x, vec_t *y,
                 rvp_csr_t rp_fun) {
    index_t nrow = m->nrow;
#pragma omp parallel for schedule(static)
    for (index_t r = 0; r < nrow; r++) {
        y->value[r] = rp_fun(m, x, r);
    }
}
```

Parallelizing spmv with OpenMP

- Dealing with skewed workloads

```
void mvp_csr_mps(csr_t *m, vec_t *x, vec_t *y,
                 rvp_csr_t rp_fun) {
    index_t nrow = m->nrow;
#pragma omp parallel for schedule(dynamic)
    for (index_t r = 0; r < nrow; r++) {
        y->value[r] = rp_fun(m, x, r);
    }
}
```

- `schedule(dynamic)` divides loop into many small chunks and load balances them across threads

Benchmarking threaded spmv

```
$ ./mrun -l 4 -d 10 -t s
Sparse   4   10   s
     MVP  RVP  GF
     seq  seq  1.36
     seq  par  3.17
...
     mpd  seq  8.05
     mpd  par  8.24
```

- Dynamic scheduling increases performance on heavily skewed workloads

Benchmarking threaded spmv

```
$ ./mrun -l 4 -d 10 -t r
Sparse   4   10   r
     MVP  RVP  GF
     seq  seq  1.36
     seq  par  3.17
...
     mpd  seq  10.52
     mpd  par  11.24
```

- …And some on slightly skewed workloads

Let's optimize this code!

<u>Exploit parallelism at multiple levels</u>

- ILP ✓
- SIMD
- Threads ✓

There's code for this in the tar,
but I'm not going to talk about it

Example: Summing an array

```
double sum_array(double *d, int n) {
    int i;
    double sum = 0.0;
    for (i = 0; i < n ; i++) {
        double val = d[i];
        sum += val;
    }
    return sum;
}
```

Parallelizing array sum w/ OpenMP (Attempt #1)

```
double sum_array(double *d, int n) {
    int i;
    double sum = 0.0;
#pragma omp parallel for schedule(static)
    for (i = 0; i < n ; i++) {
        double val = d[i];
        sum += val;
    }
    return sum;
}
```

- Q: Is this OK?
- A: No, concurrent updates to sum

Parallelizing array sum w/ OpenMP (Attempt #2)

```
double sum_array(double *d, int n) {
    int i;
    double sum = 0.0;
#pragma omp parallel for schedule(static)
    for (i = 0; i < n ; i++) {
        double val = d[i];
#pragma omp critical
        sum += val;
    }
    return sum;
}
```

- Q: Is this OK?
- A: Its correct, but won't speedup (threads serialize)

Parallelizing array sum w/ OpenMP (Attempt #3)

```
double sum_array(double *d, int n) {
    int i;
    double sum = 0.0;
#pragma omp parallel for schedule(static)
    for (i = 0; i < n ; i++) {
        double val = d[i];
        __sync_fetch_and_add(&sum, val);
    }
    return sum;
}
```

- GCC __sync_* intrinsics compile to x86 atomic instructions (vs. locks for #pragma omp critical)
- Faster, but threads still serialize!

Parallelizing array sum w/ OpenMP (Attempt #4)

```
double sum_array(double *d, int n) {
    int i;
    double sum = 0.0;
#pragma omp parallel for schedule(static) \
     reduction (+:sum)
    for (i = 0; i < n ; i++) {
        double val = d[i];
        sum += val;
    }
    return sum;
}
```

- OpenMP has specialized support for this pattern
  - Syntax: (*operand*:*variable*)

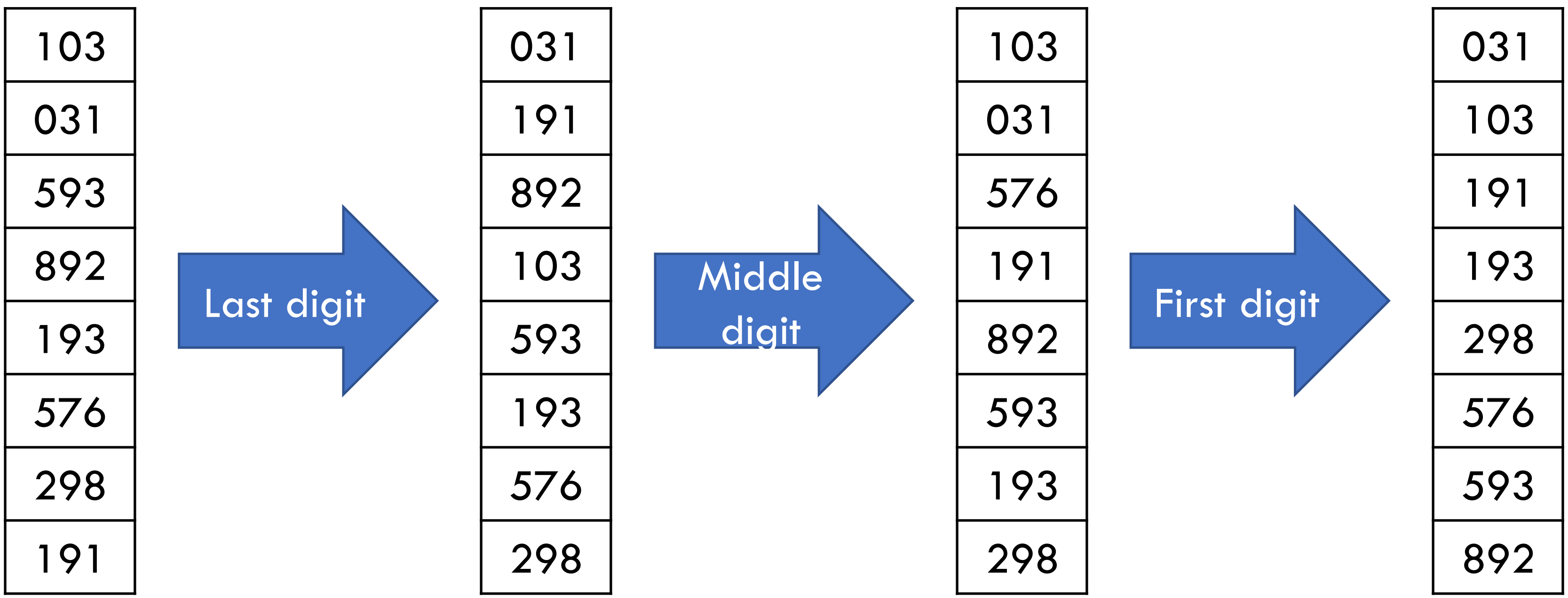
Benchmarking array sum

```
$ ./bigsum
Function        …     16
OMP critical    …     0.005
GCC atomic      …     0.037
…
OMP reduce      …     12.828
```

- Atomics are much faster than locks (7.4× speedup)
- Reduction avoids serialization (346× speedup)

More complex example: Radix sort

- Sort numbers digit-by-digit



More complex example: Radix sort

```
void seq_radix_sort(index_t N, data_t *indata,
                    data_t *outdata, data_t *scratchdata) {
    data_t *src = indata;
    data_t *dest = scratchdata;
    index_t count[BASE];
    index_t offset[BASE];
    index_t total_digits = sizeof(data_t) * 8;

    for (index_t shift = 0; shift < total_digits; shift += BASE_BITS) {

        // Accumulate count for each key value
        memset(count, 0, BASE*sizeof(index_t));
        for (index_t i = 0; i < N; i++) {
            data_t key = DIGITS(src[i], shift);
            count[key]++;
        }

        // Compute offsets
        offset[0] = 0;
        for (index_t b = 1; b < BASE; b++)
            offset[b] = offset[b-1] + count[b-1];

        // Distribute data
        for (index_t i = 0; i < N; i++) {
            data_t key = DIGITS(src[i], shift);
            index_t pos = offset[key]++;
            dest[pos] = src[i];
        }

        // Swap src/dest (only keep two buffers)
        src = dest;
        dest = (dest == outdata) ? scratchdata : outdata;
    }
}
```

Translating CUDA → OpenMP

CUDA	OpenMP
__global__ function (kernel)	#pragma omp parallel
if (threadId == 0) { … }	#pragma omp single
__sync_threads()	#pragma omp barrier
Spin lock using ‘atomicExch’	#pragma omp critical
for-loop (with iterations computed from threadId)	#pragma omp for schedule(static/dynamic)

Radix sort in OpenMP (credit: Haichuan Wang, UIUC)

```
void full_par_radix_sort(index_t N, data_t *indata,
           data_t *outdata, data_t *scratchdata) {
    data_t *src = indata, *dest = scratchdata;
    index_t total_digits = sizeof(data_t) * 8;
    index_t count[BASE], offset[BASE];

    for(index_t shift = 0; shift < total_digits;
        shift+=BASE_BITS) {
        memset(count, 0, BASE*sizeof(index_t));
        #pragma omp parallel
        {
            index_t local_count[BASE] = {0};
            index_t local_offset[BASE];
            #pragma omp for schedule(static) nowait
            for(index_t i = 0; i < N; i++){
                data_t key = DIGITS(src[i], shift);
                local_count[key]++;
            }
            #pragma omp critical
            for(index_t b = 0; b < BASE; b++) {
                count[b] += local_count[b];
            }
            #pragma omp barrier
            #pragma omp single
            {
                offset[0] = 0;
                for (index_t b = 1; b < BASE; b++)
                    offset[b]=count[b-1]+offset[b-1];
            }
            int nthreads = omp_get_num_threads();
            int tid = omp_get_thread_num();
            for (int t = 0; t < nthreads; t++) {
                if(t == tid) {
                    for(index_t b = 0; b < BASE; b++){
                        local_offset[b] = offset[b];
                        offset[b] += local_count[b];
                    }
                }
                #pragma omp barrier
            }
            #pragma omp for schedule(static)
            for(index_t i = 0; i < N; i++) {
                data_t key = DIGITS(src[i], shift);
                index_t pos = local_offset[key]++;
                dest[pos] = src[i];
            }
        }
        src = dest;
        dest = (dest == outdata) ?
            scratchdata : outdata;
    }
}
```

Run this code in parallel in each thread

Split loop across threads, don't join at the end

Critical section

Barrier

Only one thread runs this code

Figure out which thread we are

Another barrier

Another loop, with an implicit barrier on exit

Radix sort in OpenMP (credit: Haichuan Wang, UIUC)

```
void full_par_radix_sort(index_t N, data_t *indata,
           data_t *outdata, data_t *scratchdata) {
    data_t *src = indata, *dest = scratchdata;
    index_t total_digits = sizeof(data_t) * 8;
    index_t count[BASE], offset[BASE];

    for(index_t shift = 0; shift < total_digits;
        shift+=BASE_BITS) {
        memset(count, 0, BASE*sizeof(index_t));
        #pragma omp parallel
        {
            index_t local_count[BASE] = {0};
            index_t local_offset[BASE];
            #pragma omp for schedule(static) nowait
            for(index_t i = 0; i < N; i++){
                data_t key = DIGITS(src[i], shift);
                local_count[key]++;
            }
            #pragma omp critical
            for(index_t b = 0; b < BASE; b++) {
                count[b] += local_count[b];
            }
            #pragma omp barrier
            #pragma omp single
            {
                offset[0] = 0;
                for (index_t b = 1; b < BASE; b++)
                    offset[b]=count[b-1]+offset[b-1];
            }
            int nthreads = omp_get_num_threads();
            int tid = omp_get_thread_num();
            for (int t = 0; t < nthreads; t++) {
                if(t == tid) {
                    for(index_t b = 0; b < BASE; b++){
                        local_offset[b] = offset[b];
                        offset[b] += local_count[b];
                    }
                }
                #pragma omp barrier
            }
            #pragma omp for schedule(static)
            for(index_t i = 0; i < N; i++) {
                data_t key = DIGITS(src[i], shift);
                index_t pos = local_offset[key]++;
                dest[pos] = src[i];
            }
        }
        src = dest;
        dest = (dest == outdata) ?
            scratchdata : outdata;
    }
}
```

Each thread counts digits locally

Threads add their local counts into the global count

A single thread computes the offsets

Threads claim parts of the output array in turn

Finally, threads distribute their values

Benchmarking radix sort

- ./rsort -n 10000000 -r 10
- Standard library (Quicksort): 28.83
- Sequential radix sort: 34.50
- OpenMP radix sort: 176.4