Recitation 4:

OpenMP Programming

15-418 Parallel Computer Architecture and Programming

CMU 15-418/15-618, Spring 2020

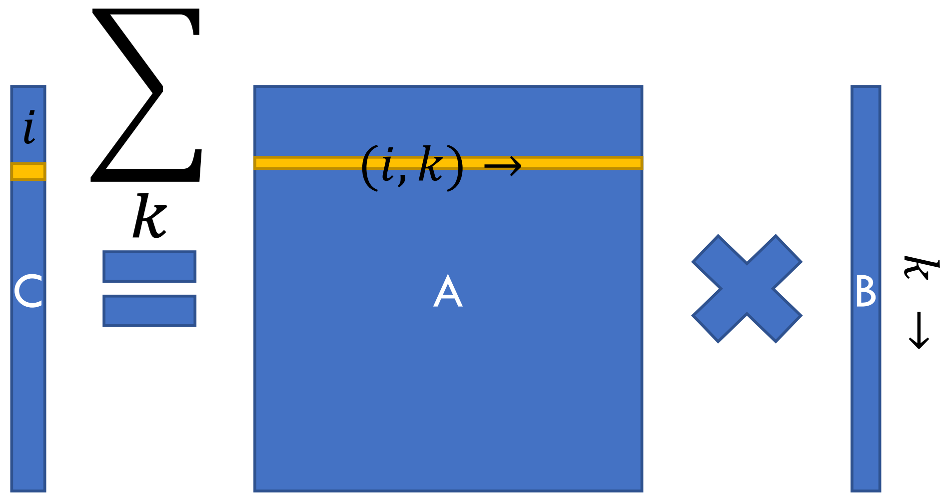
Goals for today

- Learn to use Open MP

1. Sparse matrix-vector code
   - Understand "CSR" sparse matrix format
   - Simplest OpenMP features
2. Compare different parallel computing strategies
   - Find ways that work for irregular matrices
3. Code available in:

   /afs/cs.cmu.edu/academic/class/15418-s20/www/code/rec04/mvmul

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:

```
/* 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:

```
typedef float data_t;

typedef unsigned index_t;

float rvp_dense_seq(dense_t *m, vec_t *x, index_t r) {
    index_t nrow = m->nrow;
    index_t rstart = r*nrow;
    data_t val = 0.0;
    for (index_t c = 0; c < nrow; c++)
        val += x->value[c] * m->value[rstart+c];
    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 outer loop over rows (over i in original code)

Thread parallelism with OpenMP

- OpenMP is supported by gcc
- Write standard C/C++ code
- "Decorate" your code with `#pragmas`
- We will cover only some of OpenMP's features

Parallel Outer Loop

```
void mvp_dense_mps(dense_t *m, vec_t *x, vec_t *y, rvp_dense_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);
    }
}
```

- Recruit multiple threads
- Have each do subrange of row indices

Understanding Parallel Outer Loop

```
void mvp_dense_mps_impl(dense_t *m, vec_t *x, vec_t *y, rvp_dense_t rp_fun)
{
    index_t nrow = m->nrow;
    #pragma omp parallel
    {
        // Following code executed by each thread
        index_t t = omp_get_thread_num();
        index_t tcount = omp_get_num_threads();
        index_t delta = (nrow+tcount-1)/tcount;
        index_t rstart = t * delta;
        index_t rend = (t+1) * delta;
        if (rend > nrow) rend = nrow;
        for (index_t r = rstart; r < rend; r++) {
            y->value[r] = rp_fun(m, x, r);
        }
    }
}
```

Activate tcount threads

Partition range into blocks of size delta

Assign separate block to each thread

- Each thread t does its range of rows

Parallel Inner Loop

```
data_t rvp_dense_mpr(dense_t *m, vec_t *x, index_t r) {
    index_t nrow = m->nrow;
    index_t rstart = r*nrow;
    data_t val = 0.0;

    #pragma omp parallel for reduction(+:val)

    for (index_t c = 0; c < nrow; c++) {
        data_t mval = m->value[rstart+c];
        data_t xval = x->value[c];
        val += mval * xval;
    }
    return val;
}
```

Partition range into blocks of size delta

Each thread accumulates its subrange of values

Combine values across threads

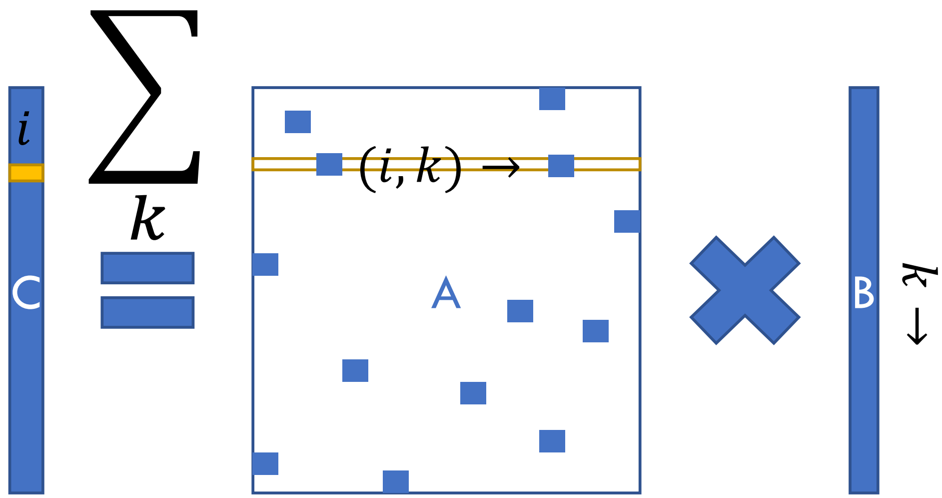
- Recruit multiple threads
- Accumulate separate copies of val and combine

Benchmarking dense mat-vec

- Matrix: 256 x 256 (65,536 entries)
  - Sequential: 2.48 GF
  - Parallel Rows: 15.43 GF (6.22 X)
  - Parallel Columns: 4.90 GF (1.98 X)
    - Tasks are too fine-grained

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 sparse matrix representation

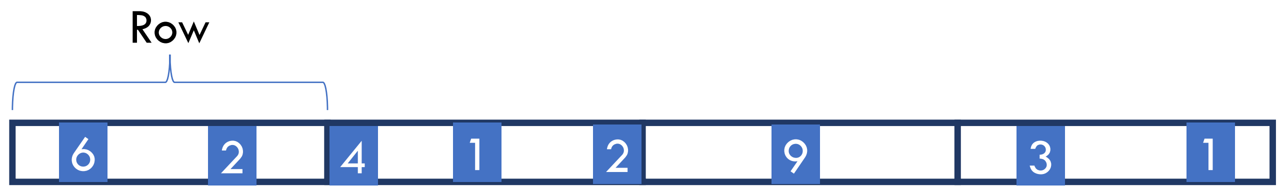
Compressed sparse-row (CSR) matrix format

- Dense matrix:

- CSR matrix:

Compressed sparse-row (CSR) matrix format

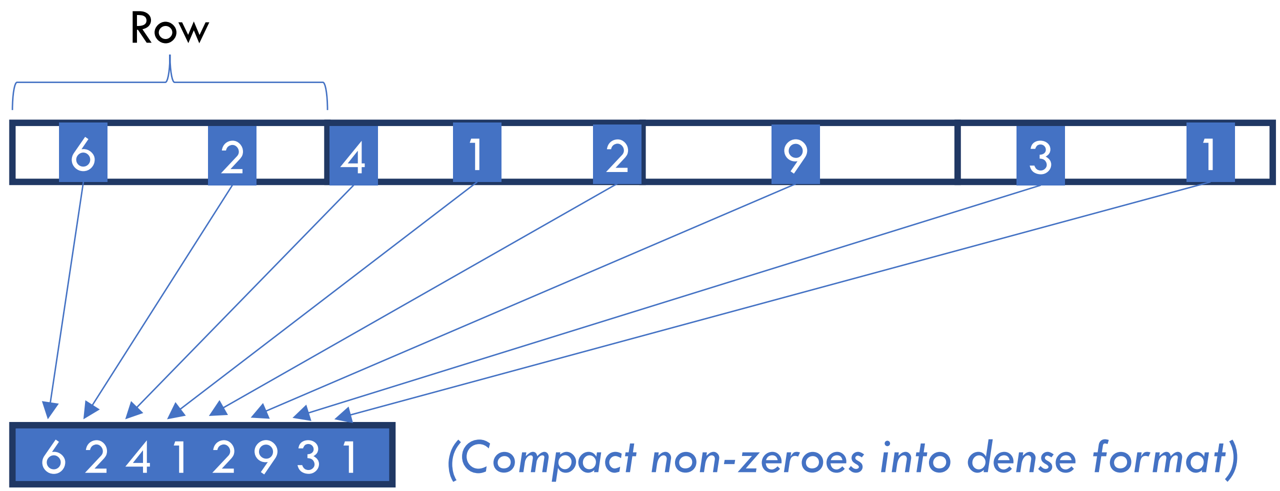
- Dense matrix:



- CSR matrix:

Compressed sparse-row (CSR) matrix format

- Dense matrix:
- CSR matrix:

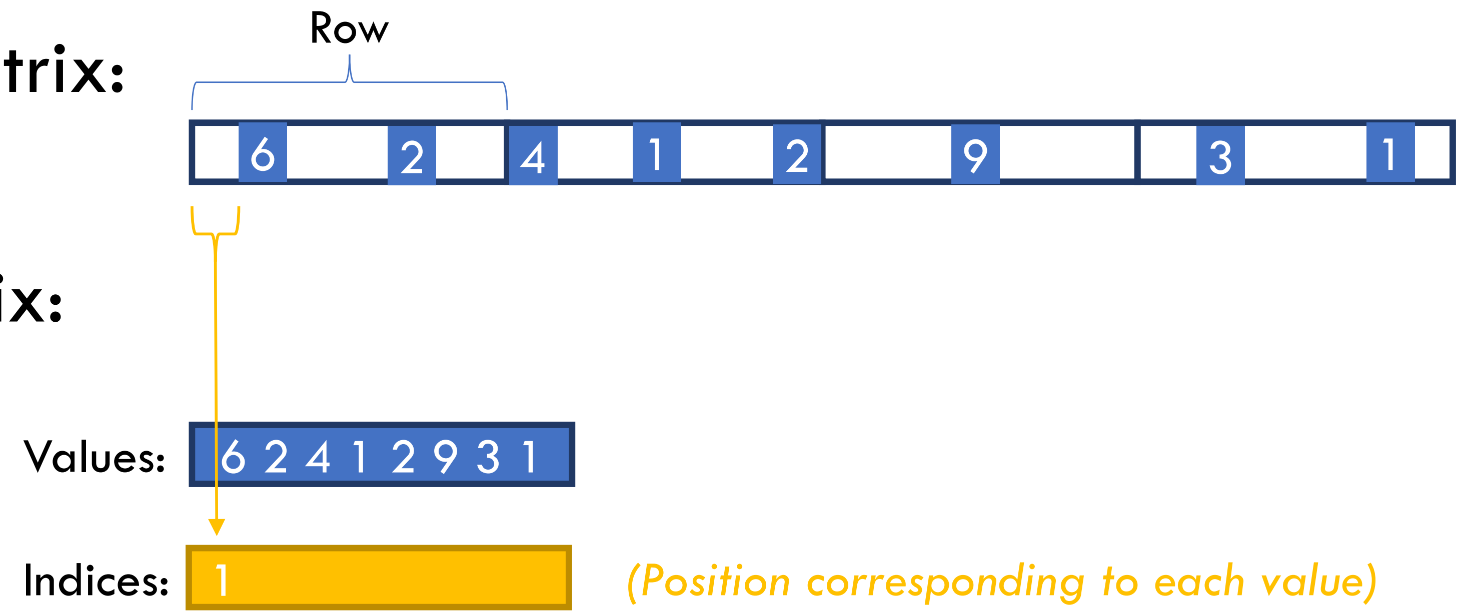


Values: *(Compact non-zeroes into dense format)*

Compressed sparse-row (CSR) matrix format

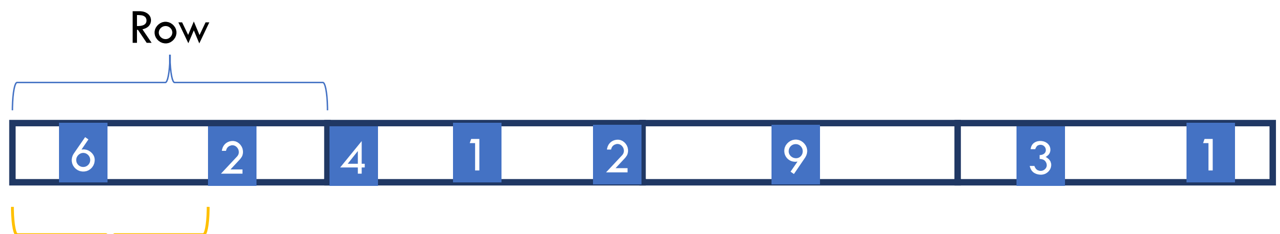
- Dense matrix:

- CSR matrix:



Compressed sparse-row (CSR) matrix format

- Dense matrix:



- CSR matrix:

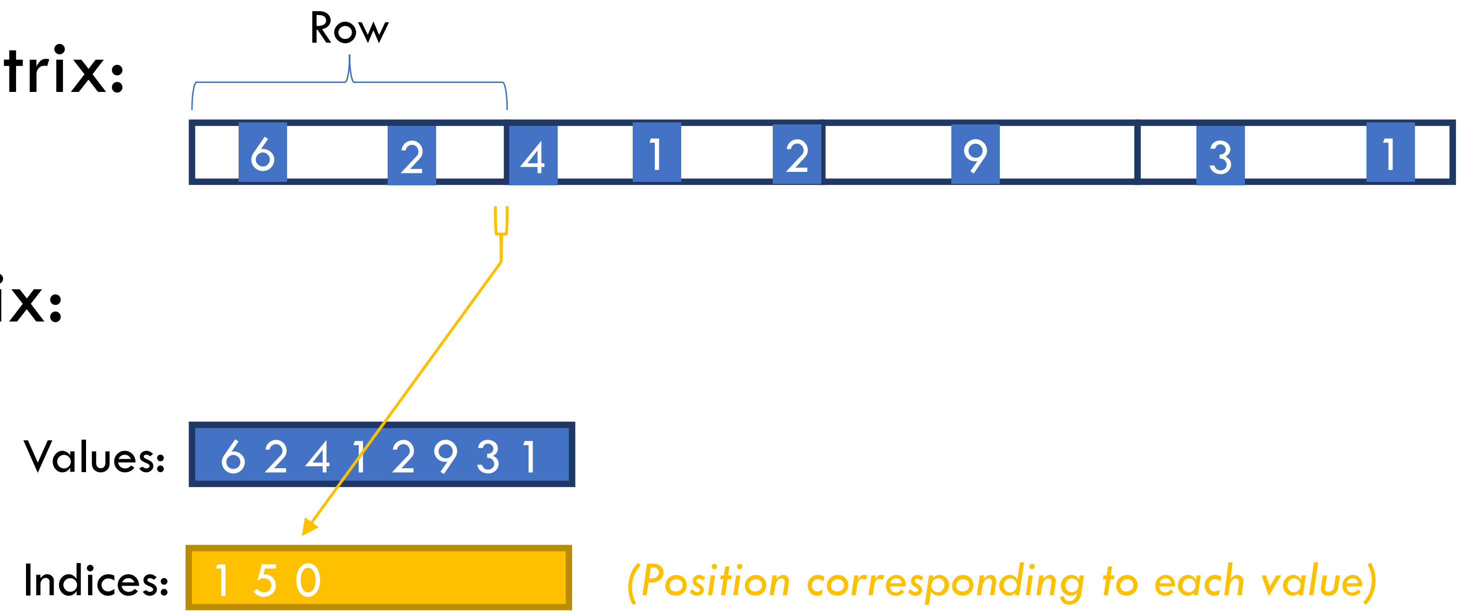
Values: 6 2 4 1 2 9 3 1

Indices: 1 5

(Position corresponding to each value)

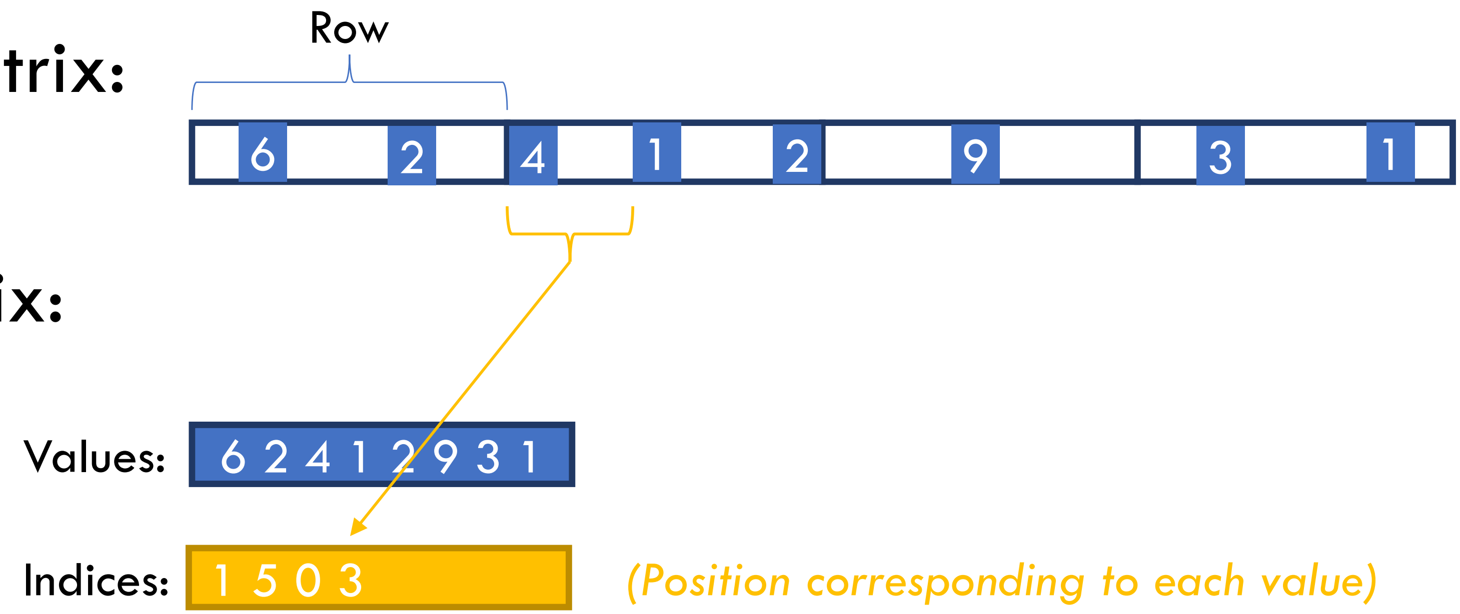
Compressed sparse-row (CSR) matrix format

- Dense matrix:
- CSR matrix:



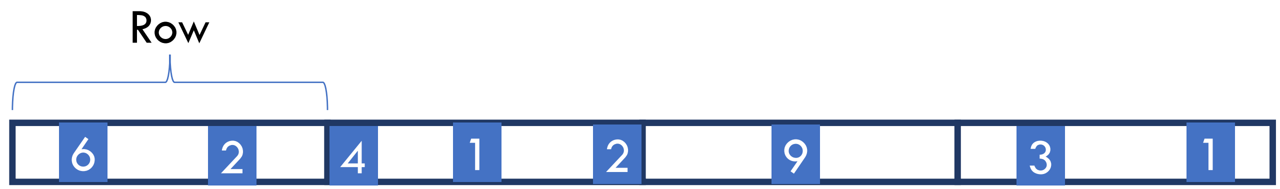
Compressed sparse-row (CSR) matrix format

- Dense matrix:
- CSR matrix:



Compressed sparse-row (CSR) matrix format

- Dense matrix:



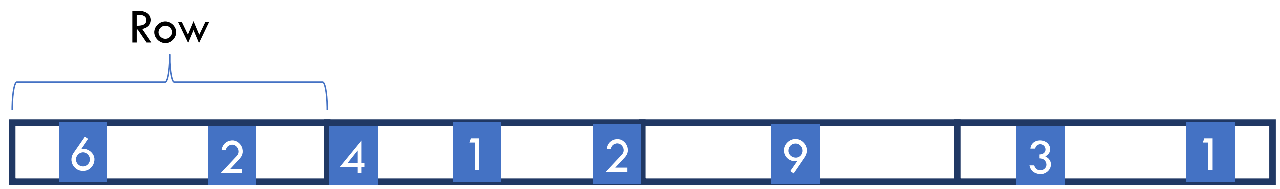
- CSR matrix:

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

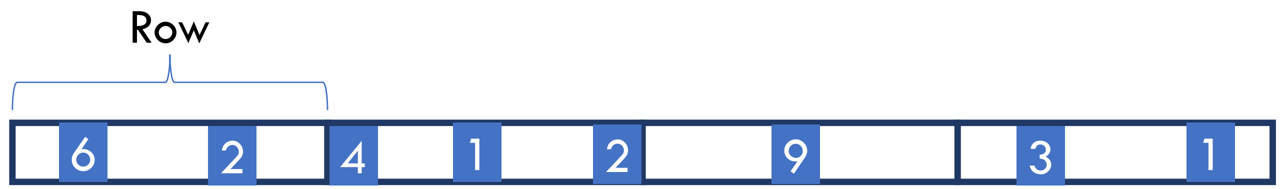
Indices: 1 5 0 3 7 4 1 6

Offsets:

(Where each row starts)

Compressed sparse-row (CSR) matrix format

- Dense matrix:



- CSR matrix:

Values: 6 2 4 1 2 9 3 1

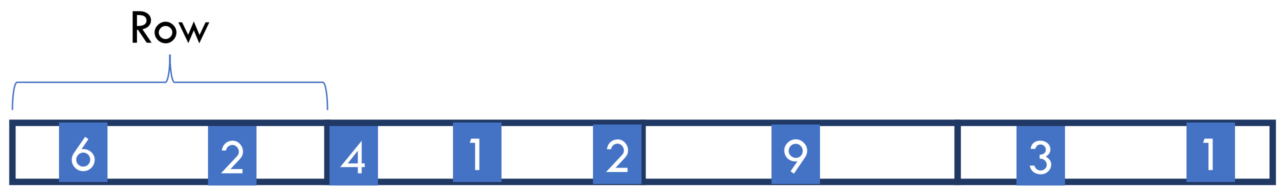
Indices: 1 5 0 3 7 4 1 6

Offsets: 0

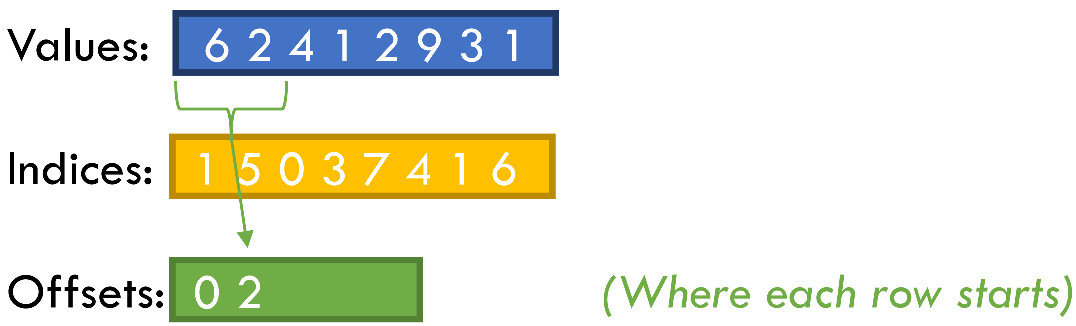
(Where each row starts)

Compressed sparse-row (CSR) matrix format

- Dense matrix:

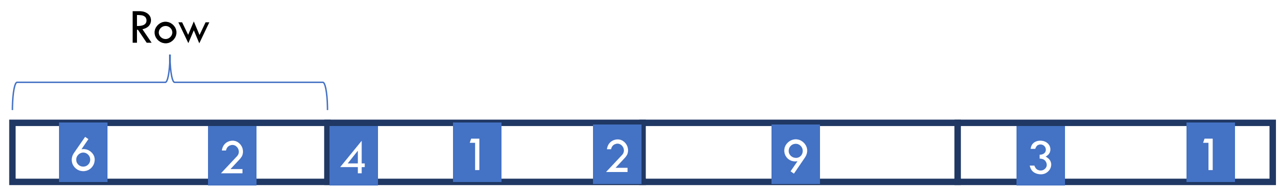


- CSR matrix:

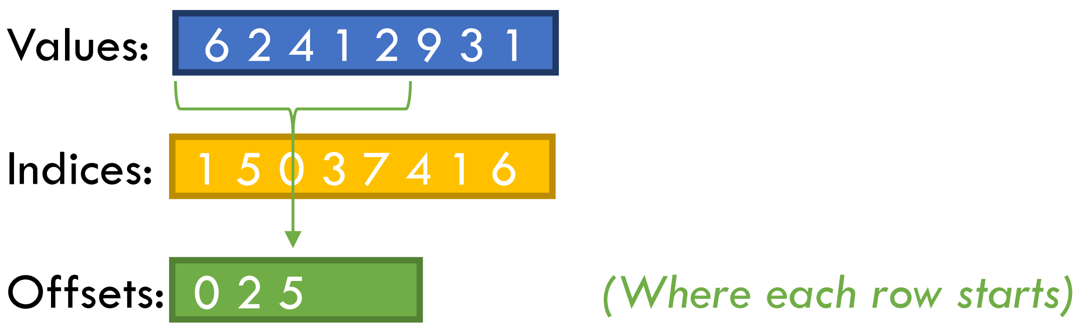


Compressed sparse-row (CSR) matrix format

- Dense matrix:

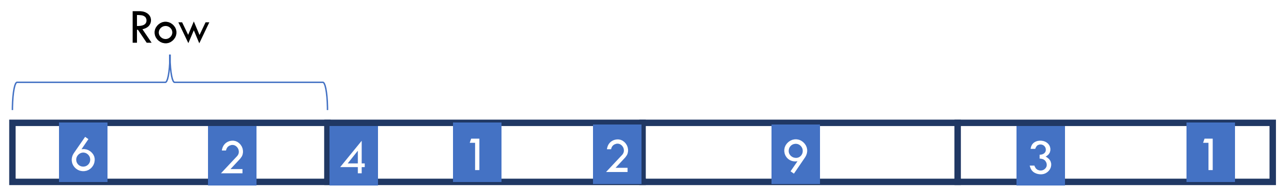


- CSR matrix:

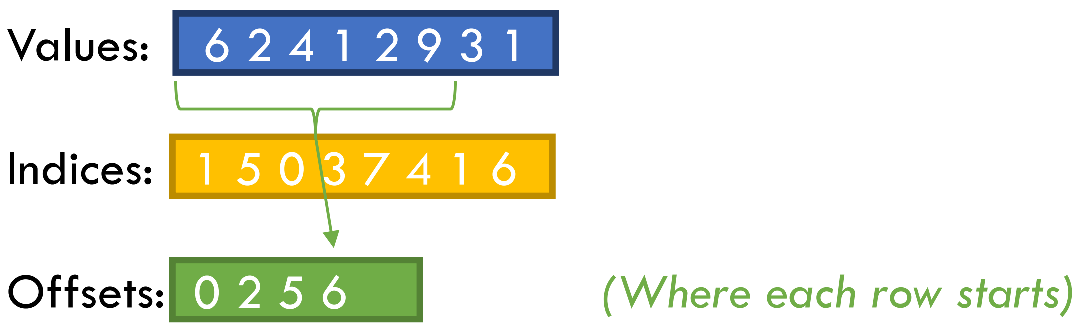


Compressed sparse-row (CSR) matrix format

- Dense matrix:

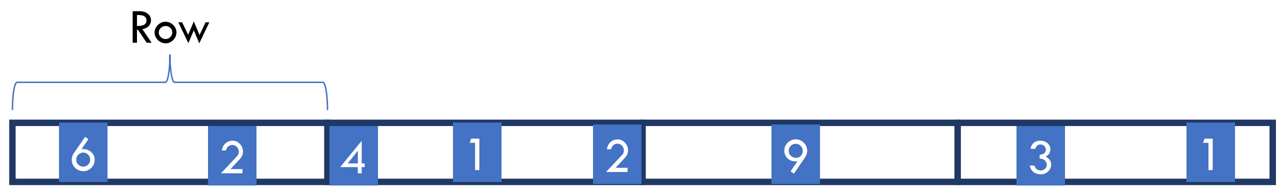


- CSR matrix:



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

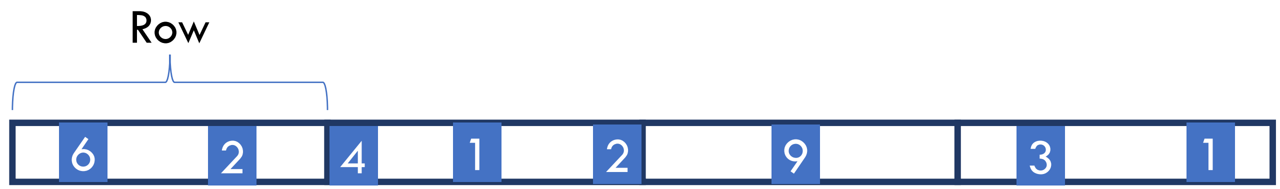
Offsets: 0 2 5 6 8

(Where each row starts)

← Dummy row…explained momentarily

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

```
data_t rvp_csr_seq(csr_t *m, vec_t *x, index_t r) {
    index_t idxmin = m->rowstart[r];
    index_t idxmax = m->rowstart[r+1];
    data_t val = 0.0;
    for (index_t idx = idxmin; idx < idxmax; idx++) {
        index_t c = m->cindex[idx];
        data_t mval = m->value[idx];
        data_t xval = x->value[c];
        val += mval * xval;
    }
    return val;
}
```

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

Iterate over nonzero values in row

```
/* 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 sparse mat-vec

- Uniform Matrix: 16384 x 16384 (65,536 nonzero entries)
  - Each row contains exactly nnz/nrow = 4 nonzero elements
  - Sequential: 2.45 GF
  - Parallel Rows: 13.87 GF (5.66 X)
  - Parallel Columns: 0.01 GF (Oops)
    - Only 4 nonzero elements / row

Benchmarking sparse mat-vec

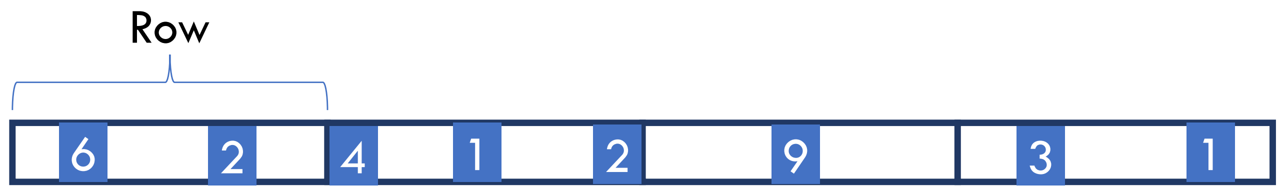
- Skewed Matrix: 16384 x 16384 (65,536 nonzero entries)
  - All nonzeros in first nnz/nrow = 4 rows
  - Sequential: 1.56 GF
  - Parallel Rows: 2.07 GF (1.33 X)
  - Parallel Columns: 0.11 GF (Oops, but better than before!)
    - Still too fine-grained

A "Data-Oriented" Strategy

- Run in parallel over all nonzero entries
  - Have each product update the appropriate row value

Compressed sparse-row (CSR) matrix format #2

- Dense matrix:



- CSR matrix:

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

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

Row Indices: 0 0 1 1 1 2 3 3 *(Row corresponding to each value)*

Data-oriented matrix-vector multiplication (atomic)

```
void full_mvp_csr_atomic(csr_t *m, vec_t *x, vec_t *y) {
    index_t nnz = m->nnz;
    zero_vector(y);
    #pragma omp parallel for
    for (index_t idx = 0; idx < nnz; idx++) {
        data_t mval = m->value[idx];
        index_t r = m->rindex[idx];
        index_t c = m->cindex[idx];
        data_t xval = x->value[c];
        data_t prod = mval * xval;

        #pragma omp atomic
        y->value[r] += prod;
    }
}
```

Partition all nonzero data into blocks

Each thread accumulates partial products for a block

Must use atomic addition to avoid races

- Require atomic updating of each value of y

Benchmarking sparse mat-vec

- Skewed Matrix: 16384 x 16384 (65,536 nonzero entries)
  - All nonzeros in first nnz/nrow = 4 rows
  - Sequential: 1.56 GF
  - Parallel Rows: 2.07 GF (1.33 X)
  - Parallel Columns: 0.11 GF (Oops)
    - Still too fine-grained
  - Data par, atomic 0.05 GF (Oops)
    - Atomic updating is expensive!

Data-oriented matrix-vector multiplication (separate accums)

Strategy (T = number of threads)

- Have T separate vectors
- Parallel over nonzero data:
  - Each thread zeros its vector
  - Each thread accumulates results in own vector
- Parallel over rows:
  - Sum vector values for each row
- Properties
  - No need for synchronization
  - Extra space and work

Data-oriented matrix-vector multiplication (separate accums)

```
void full_mvp_csr_basic(csr_t *m, vec_t *x, vec_t *y) {
    index_t nrow = m->nrow;
    index_t nnz = m->nnz;
    #pragma omp parallel
    {
        index_t tid = omp_get_thread_num();
        index_t tcount = omp_get_num_threads();
        vec_t *svec = scratch_vector[tid];
        zero_vector(svec);
        #pragma omp for
        for (index_t idx = 0; idx < nnz; idx++) {
            data_t mval = m->value[idx];
            index_t r = m->rindex[idx];
            index_t c = m->cindex[idx];
            data_t xval = x->value[c];
            data_t prod = mval * xval;
            svec->value[r] += prod;
        }
        #pragma omp for
        for (index_t r = 0; r < nrow; r++) {
            data_t val = 0.0;
            for (index_t t = 0; t < tcount; t++)
                val += scratch_vector[t]->value[r];
            y->value[r] = val;
        }
    }
}
```

Scratch vectors allocated at startup

Partition all nonzero data into blocks

Each thread accumulates partial products for block in separate vector

Recruit threads to sum values in the T different vectors

Benchmarking sparse mat-vec

- Skewed Matrix: 16384 x 16384 (65,536 nonzero entries)
  - All nonzeros in first nnz/nrow = 4 rows
  - Sequential: 1.56 GF
  - Parallel Rows: 2.07 GF (1.33 X)
  - Parallel Columns: 0.11 GF (Oops)
    - Still too fine-grained
  - Data par, atomic 0.05 GF (Oops)
    - Atomic updating is expensive!
  - Data par, sep. 3.65 GF (2.34 X)

Data-oriented matrix-vector multiplication (separate accums)

Observation:

- Accumulating in memory is more expensive than in registers

```
val += prod;                 // Fast
svec->value[r] += prod;  // Slow
```

- Data will have long runs with same row
  - Accumulate in register until row changes

Data-oriented matrix-vector multiplication (register accum)

```
index_t tid = omp_get_thread_num();
index_t tcount = omp_get_num_threads();
vec_t *svec = scratch_vector[tid];
zero_vector(svec);
data_t val = 0.0;
index_t last_r = 0;
#pragma omp for nowait
for (index_t idx = 0; idx < nnz; idx++) {
    data_t mval = m->value[idx];
    index_t r = m->rindex[idx];
    index_t c = m->cindex[idx];
    data_t xval = x->value[c];
    data_t prod = mval * xval;
    if (r == last_r) {
        val += prod;
    } else {
        svec->value[last_r] = val;
        last_r = r;
        val = prod;
    }
}
svec->value[last_r] = val;
#pragma omp barrier
```

Eliminate implicit barrier, since we're inserting explicit one

Partition all nonzero data into blocks

Each thread accumulates partial products in register

Store value to separate vector when change rows

Must store final row value

Explicit barrier synch required

Benchmarking sparse mat-vec

- Skewed Matrix: 16384 x 16384 (65,536 nonzero entries)
  - All nonzeros in first nnz/nrow = 4 rows
  - Sequential: 1.56 GF
  - Parallel Rows: 2.07 GF (1.33 X)
  - Parallel Columns: 0.11 GF (Oops)
    - Still too fine-grained
  - Data par, atomic 0.05 GF (Oops)
    - Atomic updating is expensive!
  - Data par, sep. 3.65 GF (2.34 X)
  - Data par, reg acc 4.64 GF (2.97 X)

Another use for accumulating in registers

- Combine register updating with atomic updating
  - Accumulate values in register
  - When write to memory, do so by atomic addition to row in y

Data-oriented matrix-vector multiplication (register accum, atomic updates)

```
void full_mvp_csr_opt_atomic(csr_t *m, vec_t *x, vec_t *y) {
    index_t nnz = m->nnz;
    zero_vector(y);
    #pragma omp parallel
    {
        data_t val = 0.0;
        index_t last_r = 0;
        #pragma omp for nowait
        for (index_t idx = 0; idx < nnz; idx++) {
            data_t mval = m->value[idx];
            index_t r = m->rindex[idx];
            index_t c = m->cindex[idx];
            data_t xval = x->value[c];
            data_t prod = mval * xval;
            if (r == last_r) {
                val += prod;
            } else {
                #pragma omp atomic
                y->value[last_r] += val;
                last_r = r;
                val = prod;
            }
        }
        #pragma omp atomic
        y->value[last_r] += val;
    }
}
```

Need to explicitly zero-out destination vector

Eliminate implicit barrier, since implicit one at end of omp parallel

Partition all nonzero data into blocks

Each thread accumulates partial products in register

Atomically add value to destination vector when change rows

Must add final row value

Benchmarking sparse mat-vec

- Skewed Matrix: 16384 x 16384 (65,536 nonzero entries)
  - All nonzeros in first nnz/nrow = 4 rows
  - Sequential: 1.56 GF
  - Parallel Rows: 2.07 GF (1.33 X)
  - Parallel Columns: 0.11 GF (Oops)
  - Still too fine-grained
  - Data par, atomic 0.05 GF (Oops)
  - Atomic updating is expensive!
  - Data par, sep. 3.65 GF (2.34 X)
  - Data par, reg acc 4.64 GF (2.97 X)
  - Data par, reg atom 9.99 GF (6.40 X)

Benchmarking sparse mat-vec

- Uniform Matrix: 16384 x 16384 (65,536 nonzero entries)
  - nnz/nrow = 4 nonzero entries/row
  - Sequential: 2.45 GF
  - Parallel Rows: 13.87 GF (5.66 X)
  - Parallel Columns: 0.01 GF (Oops)
  - Still too fine-grained
  - Data par, atomic 1.76 GF (Oops)
  - Atomic updating is expensive!
  - Data par, sep. 5.46 GF (2.29 X)
  - Data par, reg acc 5.79 GF (2.36 X)
  - Data par, reg atom 5.06 GF (2.07 X)

Some Observations

- Parallel performance more sensitive to data characteristics than sequential
  - Sequential 1.56–2.48 GF
  - Parallel 5.11–15.43 GF
- Easy to get parallelism out of highly structured data
  - Dense matrices
  - Sparse but regular
- But, if data sparse & irregular, need to find technique that is effective
- Need to try different approaches

Common Mistake #1

```
void mvp_dense_mps_impl(dense_t *m, vec_t *x, vec_t *y, rvp_dense_t rp_fun)
{
    index_t nrow = m->nrow;
    index_t t, tcount, delta, rstart, rend;
    #pragma omp parallel
    {
        // Following code executed by each thread
        t = omp_get_thread_num();
        tcount = omp_get_num_threads();
        delta = (nrow+tcount-1)/tcount;
        rstart = t * delta;
        rend = (t+1) * delta;
        if (rend > nrow) rend = nrow;
        for (index_t r = rstart; r < rend; r++) {
            y->value[r] = rp_fun(m, x, r);
        }
    }
}
```

Variables declared outside scope of omp parallel are global to all threads

- Variables outside of parallel are global
- Either wrong answers or poor performance

Common Mistake #2

```
data_t rvp_dense_mpr(dense_t *m, vec_t *x, index_t r) {
    index_t nrow = m->nrow;
    index_t idx = r*nrow;
    data_t val = 0.0;

    #pragma omp parallel for reduction(+:val)

    for (index_t c = 0; c < nrow; c++) {
        data_t mval = m->value[idx++];
        data_t xval = x->value[c];
        val += mval * xval;
    }
    return val;
}
```

Sequential version stepped through matrix values sequentially

But, that's not true for parallel version

- Low-level optimization can often introduce sequential dependency

Common Mistake #3

```
void full_mvp_csr_allocate(csr_t *m, vec_t *x, vec_t *y) {
    index_t nrow = m->nrow;
    index_t nnz = m->nnz;
    // Allocate new scratch vectors
    vec_t *scratch_vector[MAXTHREAD];
    #pragma omp parallel
    {
        index_t t = omp_get_thread_num();
        index_t tcount = omp_get_num_threads();
        scratch_vector[t] = new_vector(nrow);
    . . .
```

Scratch vectors allocated every time multiplication performed

- Allocate all data structures beforehand
  - Typical computation uses them repeatedly

Relation to Assignment 3

- Graphs
  - 28,800 nodes
  - 171,400–286,780 edges
  - Degrees 5–4,899
  - Similar to sparse, irregular matrix
- Properties
  - Cannot assume FP arithmetic is associative
    - Limits combining strategies
  - Integer addition is associate
    - Counting rats