15-418/618, Spring 2017
Recitation Week 7

Randal E. Bryant

3 March, 2017

1 Logistics

All files can be found in:

`/afs/cs.cmu.edu/academic/class/15418-s17/recitations/recw7`

2 Sparse matrix times dense vector

All code in subdirectory `mvmul`.

Data structures and declarations

In file `matrix.h`.

Base data types:

```
// Underlying data type for representing matrix indices.
typedef unsigned index_t;
// Type of matrix data
typedef float data_t;
```

Vectors:

```
// Dense representation of vector
typedef struct {
    index_t length;
    // Array of values
```

```
    data_t *value;
} vec_t;
```

Compressed Sparse Row (CSR) Representation of matrix. Note introduction of `rindex`. Indicates row number for each nonzero matrix element.

```
// CSR representation of square matrix
typedef struct {
    index_t nrow;       // Number of rows (= number of columns)
    index_t nnz;        // Number of nonzero elements
    data_t *value;      // Nonzero matrix values, row-major order [nnz]
    index_t *cindex;    // Column index for each nonzero entry   [nnz]
    index_t *rindex;    // Row index for each nonzero entry      [nnz]
    index_t *rowstart;  // Offset of each row                    [nrow+1]
} csr_t;
```

Row-oriented multiplication

In file `mvmul.cpp`.

Using static parallelism over rows

```
// Using OMP static parallel over rows
void mul_csr_mprow(csr_t *m, vec_t *x, vec_t *y) {
    index_t nrow = m->nrow;
    #pragma omp parallel for schedule(static)
    for (index_t r = 0; r < nrow; r++) {
        index_t idxmin = m->rowstart[r];
        index_t idxmax = m->rowstart[r+1];
        float 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;
        }
        y->value[r] = val;
    }
}
```

Measured on `latedays.andrew.cmu.edu`:

- Xeon E5-2620
- 2.40 GHz

- 15MB L3 cache
- 12 cores, each with 2-way hyperthreading

Test for matrix with N rows, for $N = 10^3$ and $N = 10^4$. All matrices have 10% density. The number of nonzero entries is $M = 0.1 \cdot N^2$, i.e., either $M = 10^5$ or $M = 10^7$.

Random data distributed among matrix rows by two different conventions:

Uniform (U): Each row has (almost) the same number of elements

Skewed (S): All data packed into as few rows as possible

Performanced expressed in Gigaflops. Matrix with N rows and M non-zero elements requires $2M$ floating-point operations (either 2×10^5 or 2×10^7.)

Method	$N = 10^3$		$N = 10^4$	
	U	S	U	S
Sequential, row-oriented	2.2	1.9	1.6	1.7
Parallel, row-oriented	4.3	0.7	4.3	1.3

Data-oriented multiplication, updating memory

Idea: compute in parallel over the nonzero data entries, regardless of which rows they are in. Simplest version performs atomic additions to the result vector.

```
// Parallel over elements, updates to memory
void mul_csr_data_mps(csr_t *m, vec_t *x, vec_t *y) {
    clear_vector(y);
    #pragma omp parallel for schedule(static)
    for (index_t idx = 0; idx < m->nnz; idx++) {
        index_t r = m->rindex[idx];
        index_t c = m->cindex[idx];
        data_t xval = x->value[c];
        data_t mval = m->value[idx];
        #pragma omp atomic
        y->value[r] += mval * xval;
    }
}
```

Method	$N = 10^3$		$N = 10^4$	
	U	S	U	S
Sequential, row-oriented	2.2	1.9	1.6	1.7
Parallel, row-oriented	4.3	0.7	4.3	1.3
Sequential, data-oriented. Update memory	0.7	0.7	0.7	0.7
Parallel, data-oriented. Update memory	1.9	0.1	2.3	2.1

Questions:

1. Why is the sequential data-oriented code slower than row-oriented code?
2. Why is the parallel performance so poor?
3. What does this tell you about the performance of atomic operations?

Data-oriented multiplication, updating registers

Can take advantage of fact that values of `rindex` will be monotonic. Use variable `last_r` to track when transition from one row to the next.

```
// Sequential over elements, updates to registers
void mul_csr_rdata_seq(csr_t *m, vec_t *x, vec_t *y) {
    clear_vector(y);
    index_t last_r = 0;
    data_t val = 0.0;
    for (index_t idx = 0; idx < m->nnz; idx++) {
        index_t r = m->rindex[idx];
        if (r != last_r) {
            if (val != 0.0)
                y->value[last_r] += val;
            last_r = r;
            val = 0.0;
        }
        index_t c = m->cindex[idx];
        data_t xval = x->value[c];
        data_t mval = m->value[idx];
        val += mval * xval;
    }
    if (val != 0.0)
        y->value[last_r] += val;
}
```

Questions:

1. Function `clear_vector` sets elements of vector to zero. It uses `memset`. Why is this needed?

2. Is the test `val != 0.0` necessary? What would be the effect of removing it?

3. Why is the final update required?

4. What would happen if attempt to parallelize the loop with an OpenMP pragma?

   (a) Would variable `val` be thread-local or global?

   (b) Would variable `last_r` be thread-local or global?

   (c) Would variable `r` be thread-local or global?

   (d) If the matrix has 1600 nonzero entries, and there are 16 threads:

      i. What value would thread i initially assign to `idx`?

      ii. What value would thread i initially assign to `r`?

      iii. What would happen with the test `r != last_r`?

   (e) What would be the overall effect of the function?

Understanding OpenMP parallelism:

- Declaring code block to be `parallel` indicates that program should split into multiple threads, each running the code block independently, except when other forms of synchronization are specified.

- Any variable declared within block is thread local.

- Declaring a for loop to be parallelized with static scheduling means that the different threads will divide the loop indices (almost) evenly.

- Declaring a particular read+write operation to be atomic means that only one thread may perform the operation at a time.

```
// Parallel over elements, updates to registers
// Shared destination vector
void mul_csr_rdata_mps(csr_t *m, vec_t *x, vec_t *y) {
    clear_vector(y);
    #pragma omp parallel
    {
        index_t last_r = 0;   // Private to thread
        data_t val = 0.0;     // Private to thread
        #pragma omp for schedule(static)
        for (index_t idx = 0; idx < m->nnz; idx++) {
```

```
            index_t r = m->rindex[idx];
            if (r != last_r) {
                if (val != 0.0)
                    #pragma omp atomic
                    y->value[last_r] += val;
                last_r = r;
                val = 0.0;
            }
            index_t c = m->cindex[idx];
            data_t xval = x->value[c];
            data_t mval = m->value[idx];
            val += mval * xval;
        }
        if (val != 0.0)
            #pragma omp atomic
            y->value[last_r] += val;
    }
}
```

Questions:

1. What would be the effect of removing the test `val != 0.0`?
2. How many threads will update element $r > 0$ of the result vector?
3. Would variable `r` be thread-local or global?
4. If the matrix has 1600 nonzero entries, and there are 16 threads:
   (a) What value would thread i initially assign to `idx`?
   (b) What value would thread i initially assign to `r`?
   (c) What would happen with the test `r != last_r`?
5. What would be the overall effect of the function?

Method	$N = 10^3$		$N = 10^4$	
	U	S	U	S
Sequential, row-oriented	2.2	1.9	1.6	1.7
Parallel, row-oriented	4.3	0.7	4.3	1.3
Sequential, data-oriented. Update memory	0.7	0.7	0.7	0.7
Parallel, data-oriented. Update memory	1.9	0.1	2.3	2.1
Sequential, data-oriented. Update register	1.7	1.8	1.5	1.7
Parallel, data-oriented. Update register	5.8	4.4	3.5	2.0

Questions:

1. How does the sequential performance compare to the other two sequential versions? Explain.
2. How sensitive is this version to skewed data?
3. Why does the performance drop off for large values of N?

Avoiding memory contention

Use thread-local vectors to accumulate result values. Track range of rows values computed by each thread to reduce number of updates.

```
// Parallel over elements, updates to registers
// Separate destination vector
void mul_csr_rldata_mps(csr_t *m, vec_t *x, vec_t *y) {
    clear_vector(y);
    #pragma omp parallel
    {
        index_t nrow = m->nrow;
        index_t last_r = 0;          // Private to thread
        data_t val = 0.0;            // Private to thread
        data_t local_y[nrow];        // Private to thread
        index_t min_r = 0;           // Private to thread
        bool first = true;           // Private to thread
        memset((void *) local_y, 0, nrow * sizeof(data_t));
        #pragma omp for schedule(static) nowait
        for (index_t idx = 0; idx < m->nnz; idx++) {
            index_t r = m->rindex[idx];
            if (first)
                min_r = last_r = r;
            first = false;
            if (r != last_r) {
                local_y[last_r] += val;
                last_r = r;
                val = 0.0;
            }
            index_t c = m->cindex[idx];
            data_t xval = x->value[c];
            data_t mval = m->value[idx];
            val += mval * xval;
        }
        local_y[last_r] += val;
        // Combine local y values
```

```
        for (index_t r = min_r; r <= last_r; r++) {
            #pragma omp atomic
            y->value[r] += local_y[r];
        }
    }
}
```

Questions

1. Where are the arrays `local_y` allocated?
2. How many calls will the function make to `memset`?
3. How will each thread assign a distinct value to `min_r`?
4. Why is it possible to specify that the for loop can be parallelized with option `nowait`?
5. What would happen if the final loop ranged from `0` to `nrow`?

Method	$N = 10^3$		$N = 10^4$	
	U	S	U	S
Sequential, row-oriented	2.2	1.9	1.6	1.7
Parallel, row-oriented	4.3	0.7	4.3	1.3
Sequential, data-oriented. Update memory	0.7	0.7	0.7	0.7
Parallel, data-oriented. Update memory	1.9	0.1	2.3	2.1
Sequential, data-oriented. Update register	1.7	1.8	1.5	1.7
Parallel, data-oriented. Update register	5.8	4.4	3.5	2.0
Parallel, data-oriented. Local copy of y	6.8	4.9	3.6	2.5

Further improvements

Imagine a scenario where the program must perform many multiplications for a given matrix. This would occur, for example, when computing an iterative solution to a linear system.

1. What data structure could be used for the local values of `y` that would eliminate the need for allocating them on each call?
2. How could the program avoid having to clear out all of the vectors with each call?
3. How could the final summations be performed without requiring any atomic operations?

3 Radix Sorting

All code in subdirectory `radixsort`.

- Sort set of 64-bit values
- Algorithm
  - Make 8 passes, from least significant byte (byte 0) to most (byte 7).
  - On each pass, perform stable sort using byte i as key.
  - Implement each pass with bucket sort.
- Implementation of one pass
  - Bucket has counter for each of the 256 possible key values
  - Read through all data and count number of keys for each bucket
  - Compute starting offsets for each key value
  - Read through all data and place in appropriate location in destination.
- Use two different arrays, alternating between source and destination.

Exercise: Use radix sort to sort the following words in 3 passes:

Input	Pass 1	Pass 2	Pass 3
`pig`			
`pat`			
`dog`			
`cat`			
`boy`			

Declarations

From `rsort.h`

```c
typedef unsigned long data_t;
typedef unsigned index_t;

// How many bits does each pass use
#define BASE_BITS 8
// What is radix of sort
#define BASE (1 << BASE_BITS)
// Mask of all 1's over BASE_BITS
#define MASK (BASE-1)
// Extract sorting key from data
#define DIGITS(v, shift) (((v) >> (shift)) & MASK)
```

Sequential code

```c
// Radix sorting, sequential.
void seq_radix_sort(index_t N, data_t *indata,
                    data_t *outdata, data_t *scratchdata) {
    data_t *src = indata;
    // Assume even number of steps, so by toggling between
    // outdata and scratchdata, result will end up in outdata
    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) {
        memset(count, 0, BASE*sizeof(index_t));
        // Accumulate count for each key value
        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];
        }
        // Find new src & dest
```

```
            src = dest;
            dest = (dest == outdata) ? scratchdata : outdata;
        }
}
```

Questions:

1. What is the role of array `scratchdata`?
2. What would be the values of `count` and `offset` when performing the first pass for the exercise example?
3. What kind of cache performance would be seen for the data reads (array `src`) and writes (array `dest`)?

Parallel version

This version is derived from code made available online by Haichuan Wang, based on code he wrote for a class at UIUC. It uses lots of clever tricks.

```
// Variation of code due to Haichuan Wang, UIUC
void full_par_radix_sort(index_t N, data_t *indata,
                         data_t *outdata, data_t *scratchdata) {
    data_t *src = indata;
    // Assume even number of steps, so by toggling between
    // outdata and scratchdata, result will end up in outdata
    data_t *dest = scratchdata;
    index_t total_digits = sizeof(data_t) * 8;
    index_t count[BASE];
    index_t offset[BASE];

    for(index_t shift = 0; shift < total_digits; shift+=BASE_BITS) {
        memset(count, 0, BASE*sizeof(index_t));
        #pragma omp parallel
        {
            // Per-thread counts and offsets
            index_t local_count[BASE] = {0};
            index_t local_offset[BASE];
            #pragma omp for schedule(static) nowait
            // Gather counts on per-thread basis
            for(index_t i = 0; i < N; i++){
                data_t key = DIGITS(src[i], shift);
                local_count[key]++;
            }
```

```
            // Compute global counts based on local ones
            // Critical faster than each addition being atomic
            #pragma omp critical
            for(index_t b = 0; b < BASE; b++) {
                count[b] += local_count[b];
            }
            #pragma omp barrier
            // Compute global offsets
            #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();
            // Compute local offsets
            // Enforce serialization by triggering each thread in sequence
            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];
            }
        }
        // Find new src & dest
        src = dest;
        dest = (dest == outdata) ? scratchdata : outdata;
    }
}
```

Questions:

1. Why are the updates to `count` placed in critical section (rather than either unsynchronized, or atomic)?

2. What is the purpose of the `barrier` pragma?
3. What is the effect of the `single` pragma?
4. What happens in the loop over thread ids?
    (a) In what order will the copies of `local_offset` be computed?
    (b) Why couldn't the computation of `local_offset` be done using a simple critical section?
    (c) Why are the values of `offset` updated?

Performance

Express in megawords of data sorted per second.

Method	$N = 10^6$	$N = 10^7$
Library quicksort	28.1	22.1
Sequential radix	31.7	26.1
Parallel radix	330.0	67.9

Questions:

1. Why does the performance of quicksort fall off faster than radix sort for large values of N?
2. What limits the parallel performance for large values of N?