Assignment 1

Jason Medcoff

January 31, 2018

1 Matrix Construction

We will begin by considering the actions necessary to construct the $N \times N$ integer matrix. As we want to fill the matrix with random integers up to some limit, it makes sense to begin by thinking about how to divide this task. We'll start with a naive function to fill our matrix with the desired values.

```
void fill_matrix(int arr[N][N], int N, int M) {
  srand(1234);
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      arr[i][j] = rand() % M;
    }
  }
}
```

Next, in order to parallelize, we consider the nested loop. We know C is row-major, so we would like to parallelize on the outer loop. `THREADS` and `SCHED` from now on denote macros for the number of threads to use and schedule to use, respectively.

```
void fill_matrix_p1(int arr[N][N], int N, int M) {
  srand(1234);
  int n_threads;
  #pragma omp parallel shared(n_threads) num_threads(THREADS)
  {
    #pragma omp single
      n_threads = omp_get_num_threads();
    int tid = omp_get_thread_num();
    int start = tid*N/n_threads;
    int end = (tid+1)*N/n_threads;
    int i, j;
    for (i=start; i<end; i++) {
      for (j=0; j<N; j++) {
arr[i][j] = rand() % M;
```

```
      }
    }
  }
}
```

For another variant, we can utilize the `for` construct.

```
void fill_matrix_p2(int arr[N][N], int N, int M) {
  srand(1234);
  #pragma omp parallel for num_threads(THREADS) schedule(SCHED)
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      arr[i][j] = rand() % M;
    }
  }
}
```

When constructing the matrix, there is no reduction of any sort taking place, e.g. sums or maximums. Therefore, there is no `p3` version with a reduction clause.

2 Finding the Maximum

Next, we would like to find the maximum value in the matrix. Intuitively, this is where the omp `reduction` clause will shine, but we still begin with a serial version.

```
int max_in_matrix(int arr[N][N], int N) {
  int max = arr[0][0];
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      if (arr[i][j] > max)
max = arr[i][j];
    }
  }
  return max;
}
```

The next version assigns work explicitly among each thread.

```
int max_in_matrix_p1(int arr[N][N], int N) {
  int max = arr[0][0];
  int n_threads;
  #pragma omp parallel shared(n_threads) num_threads(THREADS)
  {
    #pragma omp single
```

```
      n_threads = omp_get_num_threads();
    int tid = omp_get_thread_num();
    int start = tid*N/n_threads;
    int end = (tid+1)*N/n_threads;
    for (int i=start; i<end; i++) {
      for (int j=0; j<N; j++) {
if (arr[i][j] > max) {
  #pragma omp critical
  max = arr[i][j];
      }
    }
  }
  }
  #pragma omp barrier
  return max;
}
```

Next, an intelligent version can be constructed by using a `parallel for`.

```
int max_in_matrix_p2(int arr[N][N], int N) {
  int max = arr[0][0];
  #pragma omp parallel for num_threads(THREADS) schedule(SCHED)
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      if (arr[i][j] > max) {
#pragma omp critical
max = arr[i][j];
      }
    }
  }
  return max;
}
```

Finally, reduction is used to abstract away the task of computing the maximum between threads.

```
int max_in_matrix_p3(int arr[N][N], int N) {
  int max = arr[0][0];
  #pragma omp parallel for reduction(max:max) num_threads(THREADS) schedule(SCHED)
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      if (arr[i][j] > max)
max = arr[i][j];
    }
```

```
  }
  return max;
}
```

3 Histograms

Each histogram that we construct will be a structure containing the values for every bin in the histogram. We will begin by constructing an array, and iterating through the matrix, incrementing the appropriate bin in the array for each matrix entry. We can easily see that for elements up to `M`, we want ten bins `bin_0`, ..., `bin_9` such that `bin_i` represents the number of occurrences of numbers in the range bounded by $i * M/10$ on the bottom, and $(i + 1) * M/10$ on top. This task benefits from parallelism upon first glance, but care must be taken with incrementation to avoid bad writes.

Again, we begin with a serial implementation. Iterating over the elements of the matrix, we loop over the bins and increment the appropriate bin, then break.

```
void make_histogram(int hist[], int arr[N][N], int N, int M) {
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      for (int k=0; k<10; k++) {
if (k*M/10 <= arr[i][j] && arr[i][j] < (k+1)*M/10) {
  hist[k]++;
  break;
}
      }
    }
  }
}
```

Next, a parallel version is implemented with manual task assignment.

```
void make_histogram_p1(int hist[], int arr[N][N], int N, int M) {
  int n_threads;
  #pragma omp parallel shared(n_threads) num_threads(THREADS)
  {
    #pragma omp single
      n_threads = omp_get_num_threads();
    int tid = omp_get_thread_num();
    int start = tid*N/n_threads;
    int end = (tid+1)*N/n_threads;
    for (int i=start; i<end; i++) {
      for (int j=0; j<N; j++) {
for (int k=0; k<10; k++) {
  if (k*M/10 <= arr[i][j] && arr[i][j] < (k+1)*M/10) {
```

```
    #pragma omp critical
    hist[k]++;
    break;
  }
}
      }
    }
  }
}
```

Here, we write a version with the `for` construct.

```
void make_histogram_p2(int hist[], int arr[N][N], int N, int M) {
  #pragma omp parallel for num_threads(THREADS) schedule(SCHED)
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      for (int k=0; k<10; k++) {
if (k*M/10 <= arr[i][j] && arr[i][j] < (k+1)*M/10) {
  #pragma omp critical
  hist[k]++;
  break;
}
      }
    }
  }
}
```

And again, with a reduction.

```
void make_histogram_p3(int hist[], int arr[N][N], int N, int M) {
  #pragma omp parallel for reduction(+:hist[:10]) num_threads(THREADS) schedule(SCHED)
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      for (int k=0; k<10; k++) {
if (k*M/10 <= arr[i][j] && arr[i][j] < (k+1)*M/10) {
  hist[k]++;
  break;
}
      }
    }
  }
}
```

Trial runs to construct matrices, find maximums, and build histograms can now be performed. Out of a few tests, we find that the most common maximums are unsurprisingly 98 and 99, while a sample histogram follows.

Range	[0,10)	[10,20)	[20,30)	[30,40)	[40,50)
Occurrences	11	6	8	17	6

Continued	[50,60)	[60,70)	[70,80)	[80,90)	[90,100)
...	11	17	10	9	5

4 Experiment

We now want to test the performance of different schedules, chunk sizes, and thread availability on different matrix sizes and number generation bounds. Namely, we will first test the plain serial implementation. Next, we will test the `p1` version with manual task assignment on two thread sizes. Then, we will test the `p2` and `p3` versions – each on two thread sizes – with static, dynamic with chunk size 2, dynamic with chunk size 5, and guided schedules.

Each of the described tests will be performed with $N \in \{10, 10^2, 10^4\}$ and $M \in \{100, 1000\}$. We begin by borrowing the timing code from an in class exercise.

```
#include <sys/timeb.h>
double read_timer() {
    struct timeb tm;
    ftime(&tm);
    return (double) tm.time + (double) tm.millitm / 1000.0;
}
```

The `main` function containing the test code follows. Note that the code computes an average runtime of 10 executions. `N` and `M` are declared as constant integers to simplify testing.

```
// batch test: average of 10 runs
int main(void) {
  int (*A)[N] = malloc(sizeof(int[N][N]));
  int max;
  int hist[10];
  for (int k=0; k<10; k++) {
  hist[k] = 0; }
  double sumtime = 0;
  double start;

  for (int i=0; i<10; i++) {
    start = read_timer();
    fill_matrix(A, N, M);
    max = max_in_matrix(A, N); // replace with others too
    make_histogram(hist, A, N, M);
    sumtime += read_timer() - start;
```

```
  }
  free(A);
  double avg_elapsed_ms = sumtime*100;
  printf("N: %d, M: %d, t (ms) = %f\n", N, M, avg_elapsed_ms);
  return 0;
}
```

The first table represents the four algorithms plotted against N with M fixed at 100 on a static schedule with 4 threads. We can see clear improvement in the parallel algorithms over the serial version. Namely, the `reduction` algorithm is more than twice as fast.

	N=10	N=100	N=10000
serial	0.1	4.39	31106.2
p1	0.2	2.79	24000.4
p2	0.19	2.89	23847.2
p3	0.1	2.20	13829.3

The second table has the same structure as table 1, but with M fixed at 1000. The numbers are for the most part similar to above.

	N=10	N=100	N=10000
serial	0.1	4.31	38313.3
p1	0.09	3.20	24028.1
p2	0.1	3.10	23894.1
p3	0.107	2.11	13772.4

The third table gives the three parallel algorithms plotted against the number of threads used, either 4 or 8, with N fixed at 10^4 and M fixed at 1000. Surprisingly, more threads does not universally yield a faster execution time, but this may be due to experimental error such as background tasks or too few threads to notice a substantial difference.

	4 threads	8 threads
p1	24028.1	27158.0
p2	23894.1	26884.2
p3	13772.4	13716.4

The final table gives the results of different schedule types, relevant only to the algorithms using `for` constructs. N and M are fixed as above, with the number of threads fixed at 8. The difference in runtime is not so great here, most likely due to the fact that the workload on different rows of the matrix are about the same.

static	chunk 2	chunk 5	guided
26884.2	26907.9	26709.7	26624.6
13716.4	13642.3	13610.9	13713.2

5 Conclusion

Generally, the parallel algorithms outperformed the serial algorithm. Experimenting with different schedules and implementations gave insight into how they affected the running time. Reduction clauses seem to work well, at least in dense data situations. Interesting further work on the experiment might consider sparse matrices, i.e. where elements are random, and their distribution is random but based on a sparseness factor.