Assignment 1

Jason Medcoff

January 30, 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.

```
void fill_matrix_p1(int arr[N][N], int N, int M) {
  srand(1234);
  int n_threads;
  #pragma omp parallel shared(n_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
  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)
  {
    #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;
    printf("\n\n");
    for (int i=start; i<end; i++) {
      printf("\n");
      for (int j=0; j<N; j++) {
printf("%d\t", arr[i][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
  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)
  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(hist[], 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(hist[], arr[N][N], int N, int M) {
  int n_threads;
  #pragma omp parallel shared(n_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;
  }
}
      }
    }
  }
}

void make_histogram_p2(hist[], arr[N][N], int N, int M) {
  #pragma omp parallel for
  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;
}
      }
    }
  }
}
```

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 $M \in \{100, 1000\}$.