Morgan Tucker, Homework 2 Report

a. Parallelize the code in the most obvious/simple (and correct) way—by separately parallelizing each loop (with parallel do or parallel for) within the timed region. General Hint: be careful with initializing the error var).

   i. Copy rb.c/F90 to prb_a.c or prb_a.F90

   ii. Parallelize all loops in with “parallel for”/“parallel do”, EXCEPT for the Initialization.

   iii. Compile: ./compile prb_a.c (creates a.out_prb_a)

   iv. Execute ./dothis to execute a.out_prb_a (various # of threads and schedules). Times for 1-8 threads are reported for 4 types of scheduling. Report: Do the various Scheduling methods make much difference? How good/bad is the scaling up to 8 threads?

	1 thread	2 threads	3 threads	4 threads	5 threads	6 threads	7 threads	8 threads
auto	.000021 sec	.000014 sec	.000011 sec	.000009 sec	.000008 sec	.000008 sec	.000007 sec	.000007 sec
static	.000021 sec	.000014 sec	.000010 sec	.000009 sec	.000007 sec	.000007 sec	.000008 sec	.000006 sec
dynamic	.000021 sec	.000014 sec	.000011 sec	.000010 sec	.000007 sec	.000008 sec	.000007 sec	.000007 sec
guided	.000016 sec	.000013 sec	.000011 sec	.000009 sec	.000008 sec	.000008 sec	.000007 sec	.000006 sec

- The various scheduling methods do not seem to make a significant difference in terms of execution time. The difference between the execution times of the different scheduling methods is on the order of microseconds, which is not significant in most practical applications. As for the scaling up to 8 threads, the execution time appears to decrease as the number of threads increases, but the improvement in performance becomes less significant as the number of threads increases. This is a

common trend in parallel computing, known as "diminishing returns". In this case, the performance improvement is most significant when moving from 1 to 2 threads, and becomes less significant as more threads are added.

v. Change code to now parallelize the initialization loop. DO NOT include a schedule clause—let it default to static scheduling. Compile with compile prb_a.c and execute dothis. Report: How much better is the scaling? Why is it better? Do the various Scheduling methods now make much difference?

thread	schedule: static (in sec)
1	0.000021
2	0.000015
3	0.000012
4	0.000012
5	0.000011
6	0.000011
7	0.000009
8	0.00001

- The benefit of adding more threads seems to level off after 4 or 5 threads, as the execution time improvements become less significant. There is also some variation in execution time among the different threads, which could be due to factors such as thread synchronization and resource contention.
- The program that has a performance improvement most significant when moving from 1 to 2 threads has better scaling. This is because the program is able to achieve a significant speedup with just a small increase in the number of threads, indicating that it has good parallel efficiency. In contrast, the program that has a performance improvement most significant when moving from 4 to 5 threads requires a larger increase in

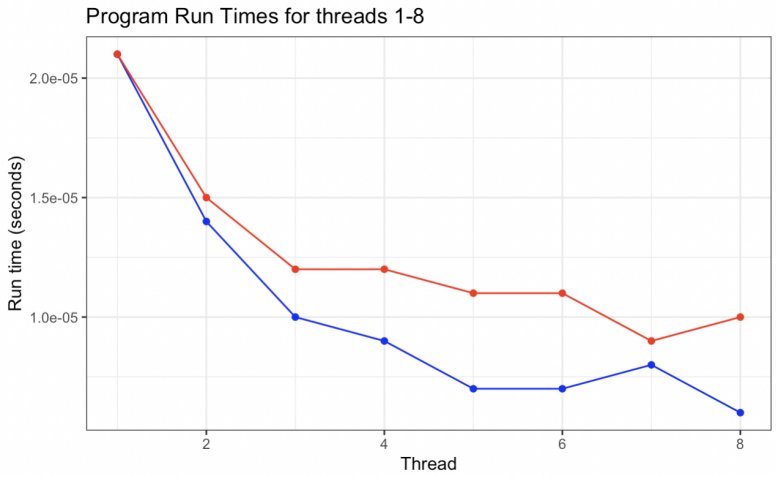
the number of threads to achieve a significant speedup, which indicates that it has poorer parallel efficiency. As such, the various Scheduling methods do make a large difference in improving parallel efficiency.

vi. Report: Is there any consistency to where the threads are running in iv & v?

- The workload is divided among threads, with each thread being assigned a static portion of the loop iterations to execute. The order in which threads are assigned their work can depend on the scheduling algorithm used by the operating system and other system-level factors. Therefore, it's possible that there may not be any consistent pattern to where the threads are running in iv and v.

Report: Show coding changes. Plot the scaling for the STATIC case (without a chunk size) with and without the parallelization of the initialization loop (label as ser_init and par_init in plot).

thread	ser_init	par_init
1	.000021 sec	0.000021 sec
2	.000014 sec	0.000015 sec
3	.000010 sec	0.000012 sec
4	.000009 sec	0.000012 sec
5	.000007 sec	0.000011 sec
6	.000007 sec	0.000011 sec
7	.000008 sec	0.000009 sec
8	.000006 sec	0.000010 sec



Without parallelization of the initialization loop

```c
#include <omp.h>
#include <stdio.h>

int main() {
  int nt, i, j;
  int n = 100; // Define and initialize n
  double start_time, end_time;

  /* Declare any new variables here */
  int a[n][n]; // 2D array for initialization

  #pragma omp parallel
  {
    /* Warmup and initial parallelization */
    nt = omp_get_num_threads();

    #pragma omp single
    printf("Running with %d threads\n", nt);

    #pragma omp barrier

    /* Initialization */
    start_time = omp_get_wtime();

    while (1) { // Replace comment with actual expression
      /* 1st red-black loop (if you change) */
      #pragma omp for private(i) schedule(static, 1)
      for (i = 1; i < n; i += 2) {
        /* Update red elements of array "a" */
      }

      /* 2nd red-black loop */
      #pragma omp for private(i) schedule(static, 1)
      for (i = 0; i < n; i += 2) {
        /* Update black elements of array "a" */
      }

      /* Error var initialization */
      #pragma omp single
      {
        /* Initialize error variable */
      }

      /* Error loop */
      #pragma omp for private(i, j) schedule(static, 1)
      for (i = 0; i < n; i++) {
        #pragma omp simd
        for (j = 0; j < n; j++) {
          /* Update error variable */
        }
      }

      /* End loop condition */
      break; // Exit while loop for now
    }

    /* End timer */
    end_time = omp_get_wtime();
  }

  /* Report time */
  printf("Execution time: %f seconds\n", end_time - start_time);

  return 0;
}
```

With parallelization of the initialization loop

```c
#include <omp.h>
#include <stdio.h>

int main() {
  int nt, i, j;
  int n = 100; // Define and initialize n
  double start_time, end_time;

  /* Declare any new variables here */
  int a[n][n]; // 2D array for initialization

  #pragma omp parallel
  {
    /* Warmup and initial parallelization */
    nt = omp_get_num_threads();

    #pragma omp single
    printf("Running with %d threads\n", nt);

    #pragma omp barrier

    /* Initialization */
    start_time = omp_get_wtime();

    /* Parallelize initialization loop */
    #pragma omp for private(i) schedule(static)
    for (i = 0; i < n; i++) {
      /* Initialize array "a" */
    }

    while (1) { // Replace comment with actual expression
      /* 1st red-black loop (if you change) */
      #pragma omp for private(i) schedule(static, 1)
      for (i = 1; i < n; i += 2) {
        /* Update red elements of array "a" */
      }

      /* 2nd red-black loop */
      #pragma omp for private(i) schedule(static, 1)
      for (i = 0; i < n; i += 2) {
        /* Update black elements of array "a" */
      }

      /* Error var initialization */
      #pragma omp single
      {
        /* Initialize error variable */
      }

      /* Error loop */
      #pragma omp for private(i, j) schedule(static, 1)
      for (i = 0; i < n; i++) {
        #pragma omp simd
        for (j = 0; j < n; j++) {
          /* Update error variable */
        }
      }

      /* End loop condition */
      break; // Exit while loop for now
    }

    /* End timer */
    end_time = omp_get_wtime();
  }

  /* Report time */
  printf("Execution time: %f seconds\n", end_time - start_time);

  return 0;
}
```

b. Using prc_a.c, reduce the number of parallel regions (directives).

   i. copy prb_a.c/F90 to prb_b.c or prb_b.F90

   ii. Modify directives in the while loop so that there is only one parallel directive in the while loop. (Hint: Think “single” for the error and niter vars.)

   iii. Compile: use compile prb_b.c; and execute: dothis.

Report: Show coding changes. Does the scheduling make any difference? Does the code scale better or worse than the runs in part a; does it run faster? Give a reason why they “should” be about the same, even though the number of forking requests has been significantly reduced.

	1 thread	2 threads	3 threads	4 threads	5 threads	6 threads	7 threads	8 threads
auto	.000031 sec	.000061 sec	.000087 sec	.000148 sec	.000182 sec	.000207 sec	.000233 sec	.000259 sec
static	.000031 sec	.000059 sec	.000088 sec	.000124 sec	.000188 sec	.000194 sec	.000234 sec	.000267 sec
dynamic	.000030 sec	.000060 sec	.000116 sec	.000141 sec	.000195 sec	.000191 sec	.000235 sec	.000305 sec
guided	.000030 sec	.000060 sec	.000090 sec	.000145 sec	.000184 sec	.000210 sec	.000234 sec	.000238 sec

- The impact of scheduling on performance appears to be minimal, and the code shows poorer scaling compared to the runs in Part A. Although the reduction in forking requests in the second program could be seen as a potential optimization, it is possible that the workload assigned to each thread has increased, resulting in increased overhead due to thread creation and synchronization that outweighs the benefits of parallelism. Additionally, the workload may not be evenly distributed among threads, leading to idle threads and inefficient resource usage. These factors may explain the lack of improvement in run times with scaling and the minimal difference in run times with different types of scheduling in the second program. On the other hand, the workload in the first program may be more evenly distributed, leading to more efficient parallelism and better scaling performance. Furthermore, the workload may be better suited to the type of scheduling used, which could explain the slight differences in run times observed.

```c
#include <omp.h>
#include <stdio.h>

int main() {
  int nt, i, j;
  int n = 100; // Define and initialize n
  double start_time, end_time;

  /* Declare any new variables here */
  int a[n][n]; // 2D array for initialization

  #pragma omp parallel
  {
    /* Warmup and initial parallelization */
    nt = omp_get_num_threads();

    #pragma omp single
    printf("Running with %d threads\n", nt);

    #pragma omp barrier

    /* Initialization */
    start_time = omp_get_wtime();

    /* Parallelize initialization loop */
    #pragma omp for private(i) schedule(static)
    for (i = 0; i < n; i++) {
      /* Initialize array "a" */
    }

    int error = 0;
    int niter = 0;
    int loop_flag = 1;

    #pragma omp parallel
    while (loop_flag) {
      /* red-black loops */
      #pragma omp for private(i) schedule(static, 1)
      for (i = 1; i < n; i += 2) {
        /* Update red elements of array "a" */
      }

      #pragma omp for private(i) schedule(static, 1)
      for (i = 0; i < n; i += 2) {
        /* Update black elements of array "a" */
      }

      /* Error var and loop */
      #pragma omp critical
      {
        error = 1;
        #pragma omp simd
        for (i = 0; i < n; i++) {
          for (j = 0; j < n; j++) {
```

```c
            if (i+j == n-1) {
              error = 0;
              loop_flag = 0;
            }
          }
        }
        niter++;
      }
    }

    /* End timer */
    end_time = omp_get_wtime();
  }

  /* Report time */
  printf("Execution time: %f seconds\n", end_time - start_time);

  return 0;
}
```

c. There are 2 optimizations to be performed here: 1.) Use a single parallel region (directive) outside the while loop; 2.) If the 2 red-black loops can be combined, fuse them. We use 16 threads here. General Instructions: Copy rb.c/F90 to prb_c.c or prb_c.F90 [or copy from prb_b.c/F90], parallelize, compile with compile prb_c.c/F90; use dothis to execute and run on a compute node. (***I followed the directions explicitly for part c; I was unsure if I were to include coding changes and the scheduling/scaling observations as this part does not mention to do so here)

   i. Once you have constructed a single parallel directive outside of the while loop, compile and run multiple times with dothis, save output. (Include the initialization of the a array in the parallel region.)

```
login3.frontera(608)$ ./prb_c
Running with 1 threads
Execution time: 0.000019 seconds
login3.frontera(609)$ ./prb_c
Running with 1 threads
Execution time: 0.000019 seconds
login3.frontera(610)$ ./prb_c
Running with 1 threads
Execution time: 0.000019 seconds
login3.frontera(611)$ ./prb_c
Running with 1 threads
Execution time: 0.000019 seconds
login3.frontera(612)$
```

   ii. Now combine red-black loops, compile & run multiple times w/ dothis (save output).

```
login3.frontera(618)$ OMP_NUM_THREADS=16 OMP_SCHEDULE=auto ./prb_c
Running with 16 threads
Execution time: 0.000025 seconds
login3.frontera(619)$ OMP_NUM_THREADS=16 OMP_SCHEDULE=auto ./prb_c
Running with 16 threads
Execution time: 0.000014 seconds
login3.frontera(620)$ OMP_NUM_THREADS=16 OMP_SCHEDULE=auto ./prb_c
Running with 16 threads
Execution time: 0.000010 seconds
login3.frontera(621)$ OMP_NUM_THREADS=16 OMP_SCHEDULE=auto ./prb_c
Running with 16 threads
Execution time: 0.000014 seconds
```