Prog 4 will help you gain a deeper understanding of multiprocessor microarchitecture. Prog 4 builds on top of Prog 3. The goal is to improve the maximum speedup for *func1* in Prog3.

There are many ways to divide the iterations of a loop across different processors. One way is to evenly and finely interleave adjacent iterations across threads (e.g., run iteration 0 on core 0, iteration 1 on core 1, iteration 2 on core 0, inter 3 on core 1, etc.). Another way is to evenly assign coarse contiguous chunks of iterations to each thread (e.g., run iterations 0-1023 on core 0, iterations 1024-2047 on core 1, iterations 2048-3071 on core 0, iterations 3072-4096 on core 1, etc.). Yet another way is to unevenly, but dynamically, assign iterations to different threads so that cores running faster can compute more iterations. OpenMP compiler directives can be used to specify how to divide up the iterations of a loop.

Use your IBM VM to measure the execution time of *func1* under four different policies of dividing up its loop iterations - (1) Even and fine-grained (i.e., interleave adjacent iterations across cores) (2) Even and coarse-grained (i.e., contiguous 1024 iterations per core) (3) Dynamic and fine-grained (4) Dynamic and coarse-grained. Like Prog 3, report execution time as the average across 10 trials.

Part A) Create four versions of *func1*, one for each of the four policies above. Measure and report speedup over single-core execution under each of the four policies. **Use the maximum of 28 threads** for all measurements. Which policy provides the highest speedup? What is the speedup over the second fastest policy?

Part B) The skeleton code provides a *func3*, which is slightly different from *func3.* Repeat Part A) for *func3*. For *func3*, which policy provides the highest speedup? What is the speedup over the second fastest policy?

Part C) Based on what you learned in class and what you see in Part A and Part B, explain why the best policy in Part A is so much faster than all other policies.

What to turn in:

- Fill in the skeleton program with your new code and turn in the completed program.
- The C code should also print your answers and explanations so that the grader can get all the answers to Part A, Part B, and Part C by running the C code.

Advice:

- When compiling your measurement program, turn off all compiler optimizations so that the binary executable behaves as closely as possible to the original C/C++ program.

OpenMP Code Example on How to Divide Iterations across Processors:

The following demonstrates the "#pragma omp parallel for" directive to execute the different iterations of a loop under different threads.

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

int main(int argc, char* argv[])
{
  omp_set_num_threads(4);
  printf("Num threads: %d\n", omp_get_max_threads());

  #pragma omp parallel for schedule(static, 1)
  for (int i = 0 ; i < 8 ; i++) {
    int thread_id = omp_get_thread_num();
    printf("Thread %d executes iteration %d\n", thread_id, i);
  }

  #pragma omp parallel for schedule(dynamic, 4)
  for (int i = 0 ; i < 8 ; i++) {
    int thread_id = omp_get_thread_num();
    printf("Thread %d executes iteration %d\n", thread_id, i);
  }
  return 0;
}
```

How to compile:

$ gcc openmp_helloworld.c -o omp -fopenmp

How to run:

$ ./omp

Output:

Static scheduling

Thread 0 executes iteration 0
Thread 3 executes iteration 3
Thread 3 executes iteration 7
Thread 2 executes iteration 2

Thread 2 executes iteration 6
Thread 1 executes iteration 1
Thread 1 executes iteration 5
Thread 0 executes iteration 4
Dynamic scheduling
Thread 0 executes iteration 0
Thread 0 executes iteration 1
Thread 0 executes iteration 2
Thread 0 executes iteration 3
Thread 1 executes iteration 4
Thread 1 executes iteration 5
Thread 1 executes iteration 6
Thread 1 executes iteration 7