Assignment 1 Report

Jean Knubel, Tsogt Baigalmaa

1 Algorithm description

1.a Program phases

pi.c

(Phase1) Call to the *calculate_pi* function starts with a serial phase where the passed arguments are checked, variables are declared/initialized and the number of threads are set. **(Phase2)**It is immediately followed by a parallel section that initializes a random number generator, statically schedules *samples* number of loops in which a random pair x and y are generated in order to increment the private variable *pcount* when required, and finally, atomically increases the total *count*. **(Phase3)**The function ends with a serial phase that does simple arithmetic operations to estimate *pi* and returns.

The following arguments hold under the assumption that there are at least as many processors as the number of threads; The initialization of the random number generator was parallelized because it is faster for threads to have their own generators rather than having to share a single one. The for loop was parallelized since the random number generations are independent of each other and can be performed in any level of parallelization. The update of the total *count* was paralellized in a critical section because coupled with *nowait*, threads can finish their job and update the global variable atomically without having to wait for the others to finish.

integral.c

1.b Performance-critical operations

pi.c

(Phase1): *omp_set_num_threads*? comparison (store load?)
(Phase2): float multiplication (how is the rand num generated?)
(Phase3): float division

integral.c

1.c Asymptotic execution time

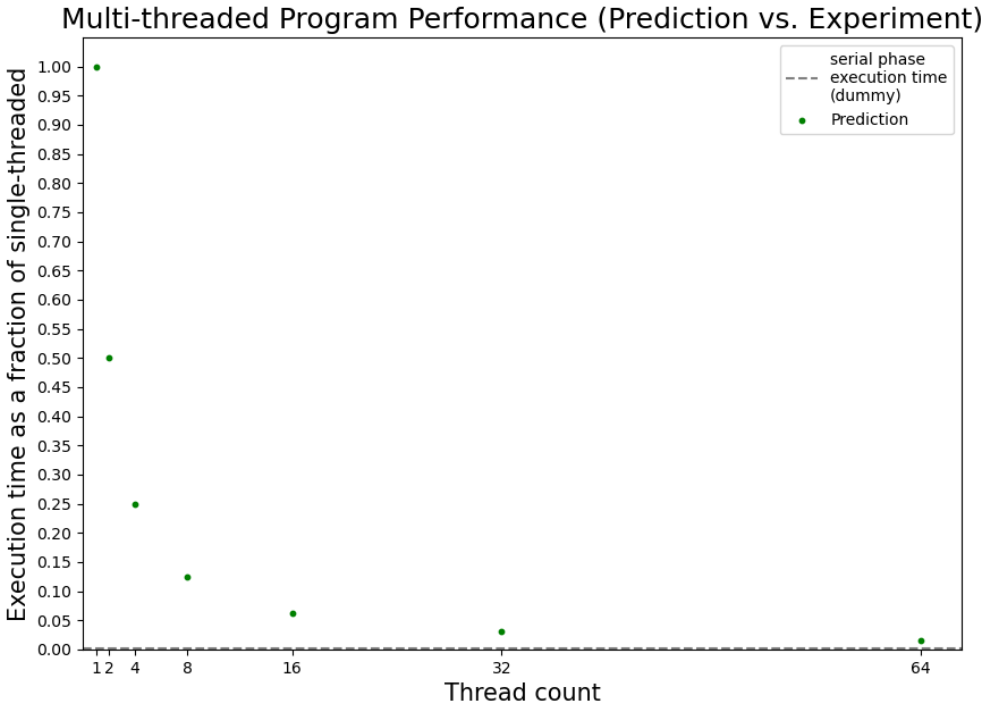
pi.c

(Phase1): N/A
(Phase2): *num_threads*, *samples*
(Phase3): N/A
$T(num\_threads, samples) \approx 2C_{comp} + (2C_{add} + 2C_{mult} + 2C_{comp})\frac{samples}{num\_threads} + C_{arith}$
$\Leftrightarrow T(num\_threads, samples) = O(\frac{samples}{num\_threads})$

integral.c

1.d Execution time vs. Number of threads

pi.c



integral.c

2 Prediction vs. Experiment

3 Discussion

4 OpenMP vs. pthreads

Programming using pthreads may require a bit more effort because, unlike OpenMP, we have to pass the desired data structures to the routines as an argument and shared variables are duplicated and passed to each one of them in order to avoid race conditions(if we don't wanna share variables). Since OpenMP gives us useful features like scheduling and critical sections, with pthreads one has to manually schedule the threads and adopt synchronization primitives such as mutual exclusion. Although, OpenMP allows us to personalize the thread behaviours using the thread ids, its semantics are designed for similar operations on homogeneous data, hence pthreads may be suitable when different tasks are to be performed in different settings using thread attributes and exclusive routines.