Pijus Kadisius

Alec Webb

CS 499

Professor Kauffman

05/02/16

Homework 4: Shared Memory Password Cracker

**OpenMP Parallelization**

We check the environment variable to determine number of processes. Load all dictionaries and passwords from a file. Compute the maximum length of temporary buffer required by all possible passwords. Loop through each password in the password file and attempt to crack it. Call try\_crackomp(..) within every iteration of the loop. Within the try\_crackomp() use “#pragma omp parallel shared(found)” to ensure that the, **int** found, can be safely read/ written by all processes. We also use “#pragma omp for” to parallelize with the right number processes. Each thread has its own buffer, which upon finding the solution copies the data into the global buffer and informs the other threads to stop doing work.

**PThread Parallelization**

We check the environmental variable for number of threads. Spawn a number of threads specified by the user. Create a struct to hold data for passing required parameters into the pthread function. We create a buffer for each individual thread to work with. Calculate how many iterations will be executed per thread. Within the parallel “int try\_crackpthread(…) divide the loop into portions and assign the portions to each thread, communicate between threads when the password is cracked in order to avoid extra work by way of the global variable pfound, and like the omp version copies the solution to the global buffer.

**time-crack.sh output:**

Running with data files pass-files/encrypted-5-2-from-100.txt dict-files/english-100.txt dict-files/english-100.txt

Serial code

real 9.54

user 9.12

sys 0.00

OMP nthreads 2

real 5.14

user 9.81

sys 0.00

OMP nthreads 4

real 2.08

user 7.78

sys 0.00

OMP nthreads 6

real 1.39

user 7.55

sys 0.00

OMP nthreads 8

real 1.01

user 6.93

sys 0.00

PTHREADS nthreads 2

real 4.94

user 9.52

sys 0.00

PTHREADS nthreads 4

real 2.05

user 7.44

sys 0.00

PTHREADS nthreads 6

real 1.58

user 8.52

sys 0.00

PTHREADS nthreads 8

real 1.34

user 9.26

sys 0.01

**Timing analysis:**

OpenMP parallelization was most effective when using 2 and 4 threads. Running the program with 2 threads made it run twice as fast as the serial version. Increasing the number of threads from 2 to 4 also made the software run twice as fast again. Further increasing the number of processes was also effective.

PThreads parallelization provided results similar to those of the OpenMP version. The program spawning two threads achieved approximately twice the speed up compared to the serial version. Increasing the number of threads to 4 cut the execution time in approximately a half again. Increasing the number of threads produced some speedup but not as much as in the first two cases.

In conclusion, speedup is achieved in all cases. We see an almost proportional speedup related to the number of processors introduced of the form: Time\_serial/num\_procs = Time\_parallel