Andrew Hecht

Noel Negusse

Programming For Performance

Professor Pingali

Assignment 6

Note: The sequential program has a runtime of 1583574812 nanoseconds

**Why does the sequential code provide an estimate for pi?**

The derivative of arcsin(x) is 1 / sqrt ( 1 – x2), and so integrating 1 / sqrt ( 1 – x2) gives you arcsin(x). We also know that arcsin(0.5) = pi/6, and arcsin(0) = 0. So by solving the integration from 0 to 0.5 of 1 / sqrt ( 1 – x2), we get an answer of pi/6. To just solve for pi, we can simply integrate from 0 to 0.5 of 6 / sqrt ( 1 – x2). The sequential code is performing this integration. For each step, we are calculating the area of the rectangle with a width of step and height of the function: 6 / sqrt ( 1 – x2).

Integrating in the range [0.0,0.5] gives more accurate results than integrating in the range [0.0,1.0) because with [0.0,0.5], we can integrate up to and including the x-value of 0.5. Using an interval of [0.0,1.0) does not calculate the integral for x = 1. This will yield less accurate results, because arcsin(1) = pi/2, but arcsin(0.999999999) is not exactly equal to pi/2.

True Sharing:

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | Running Time (nanoseconds) | Speedup | Value of Pi |
| 1 | 2786669240 | 0.568267948 |  |
| 2 | 10489426885 | 0.150968669 |  |
| 4 | 26701658551 | 0.059306234 |  |
| 8 | 19033891192 | 0.083197639 | "3.14159265138599597122" |

Atomic Instruction:

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | Running Time (nanoseconds) | Speedup | Value of Pi |
| 1 | 4539733475 | 0.348825503 |  |
| 2 | 9411049805 | 0.168267605 |  |
| 4 | 12201878349 | 0.129781233 |  |
| 8 | 14705023356 | 0.107689378 | "3.14159265138601817569" |

Removing mutexes, and using atomic instructions instead, decreases the runtime for 2, 4, and 8 threads. Similarly, by using atomic instructions, we see an improvement in speedups. For example, running on 4 threads, the atomic instruction yields a speedup of 0.12978. For true-sharing on 4 threads, this speedup is 0.0593. By using atomic instructions, we eliminate the need for mutex locks. We see quicker runtimes because in the true sharing example, each thread busy-waits trying to acquire a lock. Only after acquiring the lock, can it go find and update the global variable. Instead, with atomic instructions, the waiting happens on the variable itself. So, once no other thread is touching that variable, it just updates the value.

Global Array (False-Sharing):

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | Running Time (nanoseconds) | Speedup | Value of Pi |
| 1 | 1647966876 | 0.96092636 |  |
| 2 | 2564958534 | 0.617388075 |  |
| 4 | 2827194844 | 0.560122276 |  |
| 8 | 2737298091 | 0.578517487 | "3.14159265138563936759" |

Global Array (Eliminating Both True-Sharing and False-Sharing):

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | Running Time (nanoseconds) | Speedup | Value of Pi |
| 1 | 1597302804 | 0.991405517 |  |
| 2 | 799347676 | 1.981083901 |  |
| 4 | 399854229 | 3.960380301 |  |
| 8 | 199875656 | 7.922799823 | "3.14159265138563936759" |

Overlaying each part on a single Graph

What we’ve seen from these results, is that neither true nor false sharing provides a tremendous benefit for runtime. For the first run, using true-sharing, each thread updates a single global variable. That means that no two threads can update this variable at the same time, so instead, each thread busy waits while another thread does a little work. With true-sharing, you don’t get much benefit from extra threads because only the thread that currently holds the mutex lock is able to do work. We do see a faster runtime with 8 threads than 4 however. We attribute this to the fact that with 8 threads, there is always a thread ready to take the lock immediately when another releases it. With 4 threads, there may be time spent where no thread is doing work.

With false-sharing, each thread updates it’s own slot in a global array. This is much faster than true-sharing, because now multiple threads can do work at the same time. Every thread is able to do their own work sequentially, without worrying about other threads touching their slot. We still don’t see great speedups with false-sharing however. This is because we are not able to take advantage of caches as much as we’d like. A thread may update its slot, but by the time that same thread tries to add the next part of its work to that slot, the address may have been evicted from the cache, since other threads would have been updating their own slots in the meantime.

By using local variables, and eliminating both true and false-sharing, we can get around this issue. The local variable will get put on each thread’s stack, and so every time a thread goes to update this variable, it is always still there. With this run, we see significant decreases in runtime and substantial speedups. In fact, running on 8 threads is about 8 times faster than the simple sequential code’s runtime, with 4 threads being 4 times faster, and 2 threads, twice as fast. Eliminating true and false-sharing was really the only way to truly take advantage of multithreading, and this is evident with the speedup graph, where this run separates itself from the others.

**SSSP – Parallel Bellman-Ford**

Output for the four SSSP graphs run on 8-threads can be found in ‘results.out’

Thread speed ups can be observed from the serial.out, threads1.out, threads2.out, threads4.out and threads8.out files.

Question) Do you observe good speedups for rmat-style graphs? How about road networks?

The best speed ups are observed in the road networks. The high node, medium degree nature of the road networks are more easily parallelized via our algorithms compared to the medium node, high degree sets shown in the rmat-style graphs. When assigning nodes to individual threads, work can be more easily distributed when the total edges assigned are uniform instead of sporadic like in rmat15, where more than half of the edges are clustered from the 1st vertex.