CMSC416: Introduction to Parallel Computing

Topic: Parallel algorithms
Date: March 5, 2024

Announcements:

- The assignment 2 due date is extended to March 8
- Either work with calling context tree or call graph
- Can use from_hpctoolkit in hatchet to ingest data
- Don't use absolute value when finding differences
- Quiz: MPI runnable on shared memory, when you run on Zaratan you use 128 cores on a shared memory node
- Scribe notes added to webpage, may have errors, when scribing use complete sentences. Finally, submit notes in the correct format.

Parallel Algorithms:

The clause firstprivate gives the value in master to all threads, lastprivate gives the value from the last iteration of the for loop to master. The reduction clause reduces values across all private copies
Question: why can't we just write to a shared variable? That might result in data races

The schedule clause determines how to assign the iterations. The static type means that iterations are divided as evenly as possible, dynamic means we assign chunks to every thread, allowing threads to request more work when their chunk finishes. Guided is similar to dynamic, but decreases chunk size as time goes on. Auto lets the compiler schedule. Instead of changing the OpenMP pragma, we can set the OMP_SCHEDULE environment variable. This allows us to change the schedule at runtime instead of hardcoding (similar to OMP_NUM_THREADS).

Program: convert program that calculates value of pi from serial to parallel
Using OpenMP, we add a pragma to parallelize the for loop. When we do this, we have to think about how the variables will be used. We will have to account for the sum being incremented without issues by using a reduction. We also want to keep x private so that it doesn't get overwritten. The loop index i is private by default.

We add this line before the for loop:
#pragma omp parallel for firstprivate(h) private(x) reduction(+: sum)

Question: can we declare x inside the for loop to make it private? Probably not.
We only read from h, so we do not need the firstprivate for h.
Clauses that you put on variables don't help with reading them faster.
Parallel region pragma:

All threads execute the structured block, and the number of threads can be specified just like parallel for. You usually need to manually tell the compiler what threads should execute what part of the program. Parallel for is the typical way to use OpenMP.

Synchronization:

Concurrent access to shared data may result in inconsistencies, so we need to use mutual exclusion to avoid that. We should only use this sparingly to keep the program efficient.
We can use the critical directive, which specifies that the codeblock is only executed by one thread at a time. (#pragma omp critical [(name)])
The atomic directive specifies a specific expression should be run atomically (mainly writes to a memory location).

GPGPU:
OpenMP can take advantage of GPUs in its newer versions. The big difference of GPUs is that you have a large number of low power cores (slower individually). You have L1, L2 cache and DRAM just like in CPU.

OpenMP on GPU:

The keyword target creates the threads on the GPU. The teams distribute keyword creates a team of worker threads and distributes work among them. The memory is copied to the GPU side (?). Most of the clauses transfer over from the CPU clauses.

Aside: parallel programming is used in many different problems, not just scientific computing.
Matrix Multiplication:

In matrix multiplication, you have 3 2D arrays: two input and one output. Let's name the inputs A and B, and the output C. we do pairwise multiplications of different cells in the input arrays and sum into the output array. Let's say A is an M x L array, and B is an L x N array. We need the Ls to match to be able to do multiplication. To compute the cell i,j in C, we will take the ith row in A and the jth column in B, multiply them pairwise, then sum up the multiplications into one number and put it in the cell.

The serial algorithm has a triple for loop which loops through M, N, and L. The calculation happens one one line inside the loops. If we have large arrays (10,000 x 10,000), the calculation time will increase and the arrays may be too large to fit in memory. Because of how you store the arrays (row major or column major), one array will have more cache misses than the other.

One solution, Blocking:

We create smaller blocks that fit in the cache, which leads to cache hits. Let's say we have 128 rows, we create 32 row blocks in A, same for B. We bring one block into memory at a time, compute on that block, then sum across blocks.
If you choose the block sizes properly, we can avoid having to bring data in every time we use a new row/column. This is much more noticeable in B where we otherwise would have to bring data in for every calculation as we multiply down the column of B.
We can do this in sequential, but we can also do this in parallel.
We treat the blocks as cells: As usual, we multiply rows and columns of blocks in A and B> we matrix multiply the blocks, then sum them up to become the result block in C.
In code, we add 3 more nested for loops to determine the blocks.

Parallel Matrix Multiply:

We store A and B in a distributed manner, and we communicate to get the right sub matrices.
Each process computes a portion of C.