CMSC416: Introduction to Parallel Computing

Topic: OpenMP and Parallel Algorithms
Date: March 5, 2024

1 OpenMP

1.1 Lastprivate clause

Question: In the below loop, can you make val a global variable and get the expected behavior?

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
     val += i;
}
printf("%d\n", val);
```

Answer: Sadly, no. Using a global variable in this context would constitute a data race, as there are multiple threads reading from the same value and at least one is writing. Though it may give the correct answer on some runs, the use of a global (or shared) variable in this context would make the program unsafe for parallel execution.

1.2 Loop Scheduling

Loop scheduling describes the strategy that OpenMP takes to divide iterations of a loop amongst worker threads. The default schedule seeks to balance the work between threads as evenly as possible; however, the user can specify different strategies with a schedule clause.

The clause is of the form: `schedule (type[, chunk])`

The type can have the values of **static, dynamic, guided,** or **runtime**

Chunks define a contiguous block of indices assigned to a thread. A chunk size of 1 would have each adjacent index assigned to a different thread.

Static scheduling evenly divides chunks between the threads. If a thread finishes work before other threads, it will wait for the others to finish. This schedule works well in cases where iterations of the loop take a constant amount of time across indices.

Dynamic scheduling takes the next chunk to process from an internal work queue, with the default chunk size equal to one. For those familiar, this execution model is similar to the async channel model that the Go programming language uses. This scheduling is useful when iterations of the loop can take a variable amount of time.

Guided scheduling works similarly to dynamic, but the chunk size decreases as threads pull from the work queue. This seeks to help with load imbalances between iterations.

Auto leaves the schedule up to the compiler.

Runtime reads the schedule from the OMP_SCHEDULE environment variable. This allows the user to select the schedule at runtime without needing to recompile the program, which can take a large amount of time for complex, large projects.

1.3 Calculating Pi with OpenMP

```
int main(int argc, char *argv[])
{
      …

      n = 10000;
      h = 1.0 / (double) n;
      sum = 0.0;

      #pragma omp parallel for firstprivate(h) private(x) reduction(+:
      sum)
      for (i = 1; i <= n; i += 1) {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x * x));
      }

      pi = h * sum;
      …
}
```

To parallelize this loop, we simply needed to add the `pragma` statement shown above. h and `x` are declared private for correctness. h most likely does not need to be declared `firstprivate` as it is not being written to inside the loop, and it is already shared within the `pragma`. The `private(x)` could have also been removed by declaring the variable inside the loop, making it local to the block. In general, OpenMP tries to keep the same semantics as the sequential

program, which extends to making locally scoped variables private. The `reduction` is then the only clause necessary, as `sum` is being read from and written to by multiple threads.

1.4 Parallel Region

A **parallel region** can be parallelized using OpenMP using `#pragma omp parallel [clause [clause] … ]`. The parallel region sets the number of threads in the same way that they are for the parallel for. If placed above an expression (like a function call), it will parallelize that single line. If placed above a code block, delineated by curly braces, it will parallelize the entire block.

Warning: It is on the programmer to make sure that the code block does not have race conditions and is thread safe.

1.5 Synchronization

If all data is local to the parallel region, then no synchronization is needed. This goes for a region that only reads from variables as well. Sadly, these cases cover few useful programs and synchronization is necessary.

Note: Synchronization should be used as sparingly as possible, as that section becomes sequential. If possible, prefer lighter weight synchronization mechanisms like atomics over heavier mutexes and critical regions. Remember Amdahl's Law! We are performance limited by the fraction of our program that is sequential.

Atomic is a construct that specifies that the memory location should be updated atomically at the next expression. This controls access to a memory location in a way that does not allow for data races. See https://www.openmp.org/spec-html/5.1/openmpsu105.html for details. It is restricted to a single expression, as opposed to a code block. If multiple lines need to be accessed synchronously, please use the critical region or library locks.

Critical regions create a sequential region nested inside a parallel region. They can span multiple lines, and are created with a pragma statement. As with other `pragma`s, a single expression can be affected as well. A single line critical region is useful for a variable that cannot be updated atomically. The semantics are similar to the use of the `synchronized` keyword in Java.

Locks are a programmatic way of locking a region of code. It acts similarly to a **critical region,** except it is called by calling a pair of set and unset functions, as opposed to using a `pragma`. See https://www.openmp.org/spec-html/5.0/openmpse31.html for more information.

1.6 GPGPU

General purpose graphics processing units are the paradigm of using GPUs for general computation instead of the fixed function graphics pipelines they were originally designed for. GPGPU programming was first done by leveraging shaders in ways they were not originally intended. This way of programming was often challenging, as the semantics of your problem had to be warped to the semantics of graphics programming. With the introduction of compute platforms such as CUDA, and more recently HIP/ROCm, non-graphics applications could leverage GPUs without adopting a graphics rasterization model.

GPUs are separate daughter boards in most computers (though some are integrated into the CPU package in what is known as integrated graphics). While CPUs have between 1 and 128 cores on most architectures, GPUs have thousands of cores. This trade off is possible because GPU cores have fewer total instructions and are clocked at lower speeds. The overall cache structure is different from CPUs, with small groups of threads sharing a cache and an L2 cache separating thread block caches from the DRAM. It is critical to note that GPUs have separate memory than the system memory that CPU uses (though there are some unified architectures where GPU and CPU share the same memory, such as on many game consoles or Apple's custom silicon). This means that programmers need to manage the transfer of data between these two forms of memory. This transfer is often the slowest process in GPGPU programming and should be avoided as much as possible.

1.7 OpenMP on GPUs

With newer versions of OpenMP, the parallelization can be performed on the GPU. The pragma is of the form `#pragma omp target teams distribute parallel for`. The **teams distribute** construct creates a team of worker threads that process the work. The **target** specifies that the parallel execution should be handled on an accelerator. It is up to the programmer to handle memory transfers between the CPU and GPU memory.

2 Parallel Algorithms

2.1 Matrix Multiplication

A common operation in scientific computing (and machine learning) is the matrix multiply. Frequently this is done on matrices that are larger than system memory and need to be parallelized by necessity. For examples of libraries that perform this operation, check out https://eigen.tuxfamily.org/index.php?title=Main_Page, https://bitbucket.org/blaze-lib/blaze/src/master/, https://arma.sourceforge.net/, https://www.netlib.org/blas/, https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html#gs.5zyov7, https://icl.utk.edu/magma/, and https://developer.nvidia.com/cublas.

In matrix multiplication, the rows of the first matrix are dotted with the columns of the second matrix. This produces a single element in the final matrix. Due to this, the number of columns of the first matrix must equal the number of rows of the second matrix, or else the operation is not possible. The resulting matrix then has the same number of rows as the first matrix and the same number of columns as the second matrix. Under this definition, the matrices need not be square.

2.2 Blocking for Matrix Multiplication

Even before parallelizing the multiplication, the sequential algorithm can be updated to better utilize the cache. This is especially useful for the second matrix, as its elements are accessed in column order. In C/C++ matrices are stored in row major order, so to iterate columns the entire length of the row needs to be skipped to access the next element. For larger matrices, this can cause many cache misses as the row cannot fit in memory. To alleviate this problem, elements are stored in blocks that are sized based on the cache. Each block should be able to fit entirely in cache so that matrix multiplication has a higher percentage of cache hits. This can greatly improve the speed of the operation.