CMSC416 Instructor Notes Spring 2024
Topics: Shared Memory and OpenMP

1 Introduction to OpenMP

OpenMP is a parallel programming model designed to work with C, C++, and Fortran, enabling developers to write applications that can run on multiple cores or threads simultaneously. Unlike MPI (Message Passing Interface), which is used for distributed memory systems, OpenMP is used for shared memory systems where threads can access the same memory space. This model simplifies parallel programming, especially for applications that involve loops and arrays, by using compiler directives, also known as pragmas, and library routines.

The model is based on the fork-join principle, where the master thread forks a specified number of slave threads to execute a parallel section of the code, and these threads are joined back into the master thread upon completion. The main components of OpenMP are pragmas, which are compiler directives used to specify parallel regions, loops, and synchronization among threads.

1.1 Key Concepts

Shared Memory Model: OpenMP operates on the principle that all threads spawned by the application have access to a shared memory space, making data sharing between threads straightforward. However, this also necessitates careful management of data access to prevent race conditions.

Compiler Directives (Pragmas): OpenMP makes use of pragmas to instruct the compiler on how to parallelize the code. These directives are prepended with '#pragma omp' and inform the compiler about the parallel regions within the code.

Thread Management: The runtime environment manages the creation, execution, and termination of threads. A master thread spawns worker threads at the beginning of a parallel region, and these threads are joined back at the end of the parallel region.

1.2 Getting Started with OpenMP

Including OpenMP in Code: To use OpenMP, include the header file 'omp.h' in your C or C++ program.

Compiling OpenMP Programs: Use the '-fopenmp' flag with GCC to enable OpenMP. Different compilers might use different flags.

1.3 Setting the Number of Threads

Setting the Number of Threads: The number of threads can be set either by using the 'OMP_NUM_THREADS' environment variable before execution or programmatically within the code using 'omp_set_num_threads(int)'.

To query the number of available cores on a system, the function 'omp_get_num_procs()' can be used, enabling dynamic adjustment of the number of threads based on the system's capabilities.

1.4 OpenMP Pragmas and Constructs

- The basic structure of an OpenMP pragma is '#pragma omp ¡construct¿ [¡clause¿,...]'.

`parallel`: The '#pragma omp parallel' directive is used to parallelize the immediately following block of code. Every thread executes the block independently.

`parallel for`: Specifically used to parallelize loops, '#pragma omp parallel for' divides the iterations of the loop across the available threads. This is particularly useful for loops that iterate over arrays or perform repetitive tasks that do not depend on the order of execution.

Clauses are optional parameters that modify the behavior of the pragma, such as setting the number of threads or specifying how variables are shared among threads.

2 Shared vs. Private Variables

In OpenMP, most variables in a parallel region are shared by default, meaning all threads can access and modify them. Certain variables, like loop indices and variables declared within a parallel region, are

private by default, ensuring each thread has its own copy. To prevent race conditions and ensure correct program behavior, developers need to carefully manage shared and private variables.

2.1 Data sharing clauses

Private Clause: Specifies variables that should be private to each thread.

First Private Clause: Similar to private, but initializes each thread's copy of the variable with the value from the master thread before entering the parallel region.

Last Private Clause: Ensures the value of a variable from the last iteration of a loop is copied back to the master thread after the parallel region.

Reduction Clause: Allows performing a reduction operation (such as sum or max) across variables from all threads, consolidating them into a single value.

3 Advanced Topics

Synchronization: To manage access to shared resources and prevent race conditions, OpenMP provides mechanisms like barriers, critical sections, and atomic operations.

Reduction: When aggregating results from multiple threads (e.g., summing values), OpenMP's reduction clause simplifies the process by handling the aggregation in a thread-safe manner.

3.1 Loop Scheduling in OpenMP

OpenMP provides several scheduling strategies for distributing loop iterations among threads, including static, dynamic, guided, and auto.

Static scheduling divides iterations into fixed-size chunks assigned to threads upfront, while dynamic scheduling allows threads to request new chunks of iterations as they finish their current work, offering better load balancing for uneven workloads.

Guided Schedule: Similar to dynamic but adjusts the chunk size over time to balance workloads more effectively.

Auto Schedule: Leaves the decision of scheduling to the compiler/runtime, which can be a good default option.

The choice of scheduling strategy can significantly impact the performance and efficiency of parallel loops, especially in cases where iterations have varying computational costs.

3.2 OpenMP for GPU Acceleration

OpenMP 4.0+ supports directives for GPU acceleration, enabling code to run on GPU cores. This requires specific clauses like 'target' to offload computation to the GPU.