CMSC416 Notes

Date:2/29 — Assignment 2 Overview and OpenMP

Assignment Information

Step 1: Collect Performance Data

- **Using LULESH MPI Code**:
    - (LULESH) MPI code is a simulation code that models hydrodynamics. For this assignment, it serves as the basis for performance measurement.
    - **HPC ToolKit Usage**: Utilize this toolkit for detailed performance analysis. It's designed to analyze the behavior of applications executing on high-performance computing systems.
    - **Execution Guidelines**:
        - Perform the operation over 10 iterations to ensure statistical relevance of the performance data collected.
        - The specific instruction to replace ./exe with ./lulesh2.0 for command-line purposes
        - **Batch Script vs. Login Node**: The use of a batch script for mpirun suggests the need for scheduled execution on the HPC's compute nodes, whereas other preparatory or follow-up tasks can be handled directly on the login node, which is generally used for lighter tasks and job management.

Step 2: Python Analysis with Hatchet

- **Hatchet Analysis**:
    - Hatchet is a Python-based tool for analyzing parallel and performance-oriented computations. It allows for the exploration of hierarchical performance data from different profiling tools.
    - This step focuses on analyzing the collected performance data to understand the computational efficiency, identify bottlenecks, and explore opportunities for optimization.
    - The analysis is not constrained to the Zaratan system

Lecture Content

Fork-Join Parallelism (FJP)

- **Conceptual Foundation**: FJP is a pivotal model in parallel computing, where a main thread (the master) spawns a series of parallel threads (fork) to perform segments of computation in parallel, before joining back to continue as a single thread.

OpenMP vs. MPI

- **Message Passing Interface (MPI)**:
    - Designed for high-performance computing on distributed systems. MPI allows processes to communicate with one another by sending and receiving messages. Each process operates in its own memory space, making explicit communication necessary for sharing data.
- **OpenMP (Open Multi-Processing)**:
    - A more straightforward approach for shared-memory architectures. It utilizes directives, or pragmas, to allow developers to parallelize code sections easily. The model assumes shared memory, where threads can access common data structures, but also emphasizes the importance of managing data scope (private, shared) to prevent race conditions and ensure correctness.

Detailed Clauses and Concepts in OpenMP

- **Private, Default, Firstprivate, and Lastprivate Clauses**:
    - These clauses are critical for managing variable scope and lifecycle across the threads in a parallel region. They dictate how variables are initialized, shared, and updated, ensuring that parallel computations are both efficient and correct.
    - **Reduction Clause**: Particularly important for operations that aggregate or combine results from multiple threads (e.g., sum, max). This clause simplifies the process of reducing values across threads while ensuring thread safety.

Primer:

We have private variables in the loop to avoid random increments (mess with the loop guard) . This is to ensure that the loop index variable is private for each thread

Along same vain, stack variables in function calls from parallel regions are also private to each thread (thread-private)

Clauses are things you can add after pragma (custom way to dictate variable thread-privacy).

Private clause: each thread has own copy of variable (uninitialized) and unavailable to master thread (based on FJP) post fork [code on slide 19]

Default Clause: data sharing attributes for which would be implicitly determined otherwise [code on slide 20]

Firstprivate clause: each thread private copy to value of master thread copy upon parallel execution (eliminates the con from private clause, when you try accessing variable from master thread in parallel regions) [code on slide 21]

Lastprivate clause: writes value belonging to thread the executed last iteration of loop to master copy (determined sequentially) → parallel region to master (done via the last iteration, so a potential load imbalance doesn't really affect) [code on slide 22]

Reduction clause: reduce values across private copies of a variable (possible race condition —> because read and write operations are done using a single variable across all threads (non atomic))

> Based on certain operations [defined on slide 23] we can do operations across all copies of the variable that we have.
>
> Make every variable in the reduction clause private (each thread has its own values) then it condenses these values into a final value at the end.

Loop Scheduling in OpenMP

- **Ensuring Balanced Workload**:
  - The different scheduling strategies (static, dynamic, guided, and runtime) are essential for optimizing the performance of parallel loops. They address how work is divided among threads, aiming to minimize idle time and balance the workload efficiently.
  - **Static Scheduling**: Best for loops with iterations that take approximately the same time to execute.
  - **Dynamic Scheduling**: Useful when iterations vary significantly in execution time, as they allow for work to be dynamically reassigned to idle threads.

Loop scheduling → assignment to different worker threads (tries to schedule balance)
Can have Unequal balance → won't be load balanced and visa versa applies
Popular types of schedule Type (static, dynamic, guided, runtime)

> Static : divide iterations as evenly as possible (similar to default)
> Dynamic: assign a chunk size block to each thread → When thread is finished, it retrieves the next block from internal work queue

Chunk is a parameter too because it can start off by slowly assigning minimal iterations to threads and then fill up as it comes back (interleaving threads)
For static cases: it's done based on thread id. For dynamic case: it's done based on how fast a thread does their work (helps for load balancing)

— Guided and Runtime for next class