OpenMP

Lecture 10
March 18, 2025

To Dos

Reading for next time

Program 4 presentations

How Do We Know We Can Parallelize Code?

OpenMP

- API for specifying parallelism for shared memory programming
- Runtime system and compiler decide which threads do what
- Allows incremental conversion of sequential program to parallel program
- Preprocessor directives based (i.e., `#pragma`)

Compiling OpenMP programs

- Consists of library of functions and macros so include
    - `#include <omp.h>`

- Compile flag: `-fopenmp`

Basic Parallel Code Block

- Structured block of code:
  - One point of entry and one point of exit (although calls to `exit()` are allowed)
- `#pragma omp parallel`
  - Specifies structured block of code should be run in parallel according to number of threads runtime system indicates
- `#pragma omp parallel num_threads (4)`
  - `num_threads` clause modifies directive to specify number of threads
- Creates team of threads with implicit barrier at end
  - Each thread uses rank to specify what it should be doing
  - `omp_get_thread_num(void)`
  - `omp_get_num_threads(void)`

What About Shared Data?

- `#pragma omp critical`
  - Creates critical section around structured block of code

How accessible are variables within a parallel block?

- Variables declared in the parallel block are *private*
- Variables declared outside of the parallel block are *shared*

But we can explicitly specify access...

Reduction Variables

- Reduction operator - associative binary operator (e.g., + or *)
- Reduction - computation that repeatedly applies reduction operator to sequence of operands to get single result
- Reduction variable - place where intermediate values of reduction are stored
- reduction clause [+, *, -, &, |, ^, &&, ||]
  - `reduction (operator: variable)`
  - e.g., `reduction(+: global_result)`
- Each thread has private reduction variable and OpenMP adds a critical section where private reduction variables combined together
  - these private variables initialized with identity value for operator

Parallel `for` Code Block

- Creates team of threads to execute structured block that is a `for` loop

```
#pragma omp parallel for num_threads(4)
    for( i = 1; i < n; i++)
        result += i;
```

- Runtime system divides loop iterations among threads, typically using block partitioning as default scheduler
- Default variable scope: private

How accessible are variables within a parallel block (addendum)?

We can explicitly specify access...

- `#pragma omp parallel for (4) private(data,...)`
  - All variables in private clause' parentheses are private, with each thread having its own copy
- `#pragma omp parallel for (4) shared(data,...)`
  - All variables in private clause' parentheses are shared
- `#pragma omp parallel for (4) default(none)`
  - No variable have default access, so each much be specified for private/shared

Thread pools

```
#pragma omp parallel for (4) private(data,...)//create threads
        // do some work
        #pragma omp for  // use threads created above for parallel execution
        for(i = 1 ; i < n; i++) {
            ….
        }
```

Scheduling Parallel Blocks

- Assigning loop iterations to threads is called scheduling
- Default scheduling for `parallel` directive is block partitioning
- Scheduling for `parallel for` and `for` directives can be specified with schedule clause

  `schedule(<type>) [, <chunksize>] )`

  `type: static, dynamic` or `guided, auto, runtime`

  `chunksize:` positive integer representing number of iterations in block to be executed serially (not for auto or runtime)

- Runtime overhead associated with using `schedule` clause
  - none < static < dynamic < guided

How to Get Different Types of Partitioning with `schedule` clause

- block partitioning
- cyclic partitioning
- block-cyclic

Schedule types: `static`

- `chunksize` iterations assigned in round-robin fashion
- default `chunksize` typically iterations / thread
- good if time of iterations changes linearly as loop executes

Schedule types: `dynamic` and `guided`

`dynamic`

- Allocations done in `chunksize` quantities
- Each thread initially gets one `chunksize.` Must ask for next `chunksize` set when it completes
- Default `chunksize` is 1
- Good if iterations do unpredictable amounts of work
- Overhead associated with asking runtime system for work

`guided`

- Like dynamic, but allocation size decreases as chunks are completed
- Typically allocations are ½ of remaining number of iterations
- If `chunksize`specified, allocation size decreases down to `chunksize`

Schedule types: `runtime`

- Runtime system uses environment variable `OMP_SCHEDULE` to determine `type` of schedule
  - e.g., `OMP_SCHEDULE="static,2"`