Date 14th March 2024 **Quiz-2** Thursday

→ Shared Memory Programming :- Lecture 8 :-

1.) physically : processors in a computer share access to the same RAM

2.) virtual : threads running on the processors interact with one another thru shared vars in the common addr space of a single proc

=> performance improvements to serial code is easier with multi-threading than with message-passing.

→ MP reqs code to be redesigned

→ MT allows incremental parallelism

- Clusters are made up of "multiple processors per compute node"
- Processors interact & synchronize with each other thru shared vars

OpenMP with MPI improves performance at 2 levels : shared & distributed

→ Fork / Join :-

→ MT prog is most common shared-mem programming method

serial code begins exe => the proc is master S or only executing S => || portion, master S can fork more $SSSS$ => || portion finish, $SSSS$ join & Master S continues executing serial code

Shared-Memory	Message-Passing
• One active thread at start & end of prog	• all processes remain active thru out exe of program
• Num of active S inside prog changes dynamically process of converting a seq prog to a parallel prog a lil bit at a time	• Seq to Parallel transformation requires major effort transformation done in one giant step rather than many tiny steps
• Supports incremental parallelism	• No incremental parallelism

openMP: standard method for shared-memory prog | OpenMP consists of compiler directives, lib func, env vars
MPI: standard for distributed memory prog
→ Codes are portable
Date → Performance is good

pragma: pragmatic information

```
# pragma omp parallel for
for(i=0; i<N; i++)
    a[i] = b[i]+c[i]
```

① Compiler should know total num of iterations b4 executing the prog

② Body of for-loop must not allow premature exits (e.g break etc)
x return, exit, goto not allowed

③ loops with 'continue' statement are allowed

① Shared Variable: same addr in execution context of every thread
② Private Variable: diff addr in execution context of every thread

∴ thread can NOT access priv variables of another thread

∴ C prog express data-parallel operations as for loops

Lecture 9 :-

∴ Compiler takes care of generating code that fork/join threads & allocates iterations to threads

Clauses :-

1.) Private: directs compiler to make one or more variables private
pragma omp parallel for private(j)

2.) first Private: create private variables having initial values identical to the value of the variable controlled by the Master Thread as the loop is entered.

3.) last Private: used to copy back to the master thread's copy of the var, the priv copy of the variable from the thread that executed the last iteration.

4.) Reduction: takes care of storing partial results in private variables & combining partial results after the loop reduction(+:variable)

5.) if-clause: conditional statement, decides at run-time whether the loop should be executed in parallel or not

Date

6.) Scheduling : scheduling the loops means dividing num of iterations b/w the processes.

- a chunk is a contiguous range of iterations
  - → increasing chunk size reduces scheduling overhead & may increase cache hit rate (due to operations on contiguous mem loc)
  - → decreasing chunk size allows finer balancing of workloads

7.) Static Scheduling :

1.) if chunk-size given : splits iteration space into equal chunks & assigns them to threads in a round robin fashion

2.) if no chunk-size : iteration space is split into as many chunks as there are threads

$\frac{\text{num of iterations}}{\text{total threads}}$ & one chunk is assigned to each thread

- decision about work division is done before code execution
- lower scheduling overhead but can cause load-imbalance if all processors are not of same compute-capability

8.) Dynamic Scheduling :

1.) iteration space is partitioned into chunks given by chunk-size
2.) every thread is assigned single chunk
3.) decision for remaining iteration chunks is done on run-time.
4.) Chunk is assigned to threads as they become idle (takes care of temporal imbalances resulting from static scheduling)

⇒ When you use dynamic sched without specifying chunk size, OpenMP assigns one iterations to each thread initially. As threads complete assigned iterations, they request & are assigned new iterations. No single thread remains idle while others have work to do.

Date 30/3/24 mid-2 sat

Lecture-9 :- (continued)

→ Guided Scheduling :-

- Scheduling algo dynamically adjusts the size of the chunks based on the num of remaining iterations & the execution progress.
- At the beginning → larger chunks of iterations.
- As the parallel loop progresses, threads complete their assigned chunks, guided scheduling gradually decreases the size of the chunks.
- This adaptive beh helps in achieving better load balancing among threads esp as the workload becomes more uneven or as some iterations take longer to execute than others.
- Guided Scheduling allows you to specify a minimum chunk size ('c'). Once the chunk size decreases to this min value, it remains constant for subsequent iterations.
  (this helps in preventing chunk size from becoming too smol, which could lead to increased overhead)

1.) exploits initial parallelism by assigning bigger chunks in start

2.) decreased chunk-size exploits fine-grained load balancing

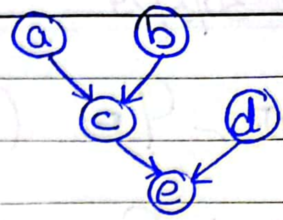
=> No Wait Clause :

→ avoids implicit barrier (every thread waits after executing for loop, waits for the rest of threads)

→ thread can easily move to next after completing assigned tasks/iterations

Date

⇒ functional / task parallelism :-



- # pragma omp sections [clause list]

→ if code is based on diff segments or sections

→ can execute a, b, d parallelly

```
# pragma omp parallel sections (creates a team of threads which execute
{                                 the sections in the region parallelly)
    # pragma omp section
        v = a()
    # pragma omp section
        w = b()
    # pragma omp section
        x = d()
}
```

Omp parallel sections generates it's own team of threads

Omp sections uses existing team of threads & distributes section among the threads

if multiple sections pragmas are inside one parallel block, may reduce fork/join costs

Basic Lib functions :-

1.) omp_get_num_procs

num of CPUs/cores in machine

2.) omp_get_num_threads

num of active threads, call from || region

3.) omp_get_max_threads

value of env variable omp_num_threads

can be called outside of parallel region

4.) omp_get_thread_num

value of thread id

5.) omp_set_num_threads

omp_set_dynamic(0) disable dynamic

omp_set_num_threads(4)

- Controlling Num of Threads & Processes :-

1.) omp_set_num_threads

2.) omp_get_num_threads

3.) omp_get_max_threads

4.) omp_get_thread_num

5.) omp_get_num_procs

6.) omp_in_parallel

- Controlling & Monitoring | Creation :-

1.) omp_set_dynamic

2.) omp_get_dynamic

3.) omp_set_nested

4.) omp_get_nested

Date

Lecture 10 :-

=> Synochronization in OpenMp:-

1.) Barrier directive :- all threads in a team wait until others have caught up, 3 then release (opposite of No Wait 3 is already implemented: default)

2.) Single Directive :- a structured block executed by single thread in parallel region (mostly print statements) implicit barrier

3.) Master Directive:- specialization of single: only master thread will execute structured block no implicit barrier

4.) Atomic Directive :- specifies that the single memory location update should be performed as an atomic operation (single operation w/o interuption from other threads)

5.) Critical Section :- code segment that has a shared var 3 needs to be executed as an atomic action. Only one process can execute critical section at a time

=> Environment Variables

1.) omp_num_threads
2.) omp_dynamic
3.) omp_schedule
4.) omp_nested

1.) Omp_Dynamic :-