Parallel Computing

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

Lecture 7:

- Shared-Memory Programming with OpenMP
  - Functional Parallelism
  - Synchronization Constructs
  - Thread-Local Storage

OpenMP: Sections Construct

The sections construct provides **functional parallelism** by enabling different threads to carry out different tasks:

- It permits us to specify several different code regions, each of which is represented by a section construct and is executed by one thread in the team.

- It consists of two different directives as follows:

```
#pragma omp sections [clause[[,] clause]...]
    {
    [#pragma omp section ]
        structured block
    [#pragma omp section
        structured block ]
    ...
    }
```

OpenMP: Sections Construct

- The code block of each section must be a structured block, i.e. one entry and one exit.
- At runtime, one thread executes one section at a time, and one section will be executed exactly once.
- If there are fewer threads than sections, some or all threads must execute multiple sections.
- If there are more threads than sections, the remaining threads will be idle.

OpenMP: Sections Construct

```
#pragma omp parallel
{

    #pragma omp single
    std::cout << "The number of threads : " << omp_get_num_threads() << std::endl;

    #pragma omp sections
    {
        #pragma omp section
        f();
        #pragma omp section
        g();
        #pragma omp section
        h();
    }
}
```

If three or more threads are available, one thread might invoke *f()*, a second one might invoke *g()* and a third one might invoke *h()* whereas the remaining are idle, waiting at the implicit barrier.

OpenMP: Synchronization

Recall that we need to:

- ❑ synchronize actions on shared variables
- ❑ ensure the correct ordering of reads and write to shared variables
- ❑ protect concurrent updates to shared variables
  - ➢ Updates to variables are not atomic by default.

OpenMP: Barrier Construct

The OpenMP directive used to implement an explicit barrier in OpenMP is

```
#pragma omp barrier
```

- ❑ No threads cannot progress past a barrier construct until all threads in the team have arrived.
  - ➢ The all-or-none Principle !!!
- ❑ Either all threads in the same team or none must encounter a barrier.
  - ➢ Otherwise, this will result in a deadlock.
- ❑ This is one of the very few OpenMP directives that do not have an associated block of code.

OpenMP: Critical Section

In OpenMP, a critical section can be denoted by the critical section construct which provides a means to ensure that multiple threads do not attempt to update shared data simultaneously.

#pragma omp critical *[(name)]* *structured block*

- An optional name can be given to a critical construct and this name is global.
- When a thread encounters a critical section, it waits until no thread is executing inside the critical section with the same name.

OpenMP: Critical Section

```
#pragma omp parallel default(none) shared(totalHits) private(seedVal)
{
    seedVal = omp_get_thread_num();//Each thread seeds with its own thread id.

    #pragma omp for
    for(int i=0 ; i<NUMITERATIONS ; i++)
    {
        double  x = (double) rand_r(&seedVal) / (double) RAND_MAX;
        double  y = (double) rand_r(&seedVal) / (double) RAND_MAX;

        double result = std::sqrt((x*x) + (y*y));

        if(result<1.0){
            #pragma omp critical(update_total_hits)
            {//critical section
                totalHits += 1.0;//check if the generated value is inside a unit circle.
            }
        }

    }
}
```

In this snippet, the program calculates the value of π using Monte Carlo simulation.

- Updates to the shared variable *totalHits* are protected by the critical pragma *update_total_hits*.
- However, speedup is poor with this approach.
- It is more efficient to use the reduction clause.

OpenMP: Atomic Construct

The atomic directive is used to protect a single update to a shared variable:

- It applies to only a single statement, instead of a code block.

- The syntax of the atomic directive is

```
#pragma omp atomic
        statement
```

where *statement* is a single arithmetic statement.

OpenMP: Single Construct

The single construct is a work-sharing construct that specifies that the associated code block should be executed by one thread only.

- There is always an implicit barrier at the end.
- The other threads wait at the barrier until the code block has been executed by the chosen thread.
- The implicit barrier at the end of the construct can be removed with the *nowait* clause.
- The syntax of the single directive is

#pragma omp single *[clause[[,] clause]...]*	*structured block*

OpenMP: Single Construct

- It is up to the OpenMP runtime which thread in the team to execute the code associated with a single construct.

- This construct can be useful when:
    - if one thread in the team is an I/O thread.

    - there is a setup part such as memory allocation and deallocation where only one thread must be responsible for.

OpenMP: Single Construct

private$(list)$
firstprivate$(list)$
copyprivate$(list)$
nowait

Clauses supported by the single construct

OpenMP: CopyPrivate

The *copyprivate* clause provides a mechanism for broadcasting the value of a private variable from one thread to the other threads in the team:

- The typical use for this clause is to have one thread initialize or read private data that is subsequently used by the other threads as well.
- After the single construct has ended, but before the threads have left the associated barrier, the values of the private variables specified by the *copyprivate* clause are copied to the other threads.
- The standard prohibits the use of this clause in combination with the *nowait* clause.

OpenMP: CopyPrivate

```
int x = 1;
int y = 2;

#pragma omp parallel default(none) private(x) shared(y,std::cout)
{
    #pragma omp single copyprivate(x)
    {
        x = 10;
        y = y + 2*x;
    }


    #pragma omp critical
    std::cout << "Thread " << omp_get_thread_num() << " has x = " << x << " and y = " << y << std::endl;
}
```

OpenMP: Master Construct

The master construct is a synchronization construct.

- ❑ The syntax of the master directive.

```
#pragma omp master
        structured block
```

- ❑ Only the master thread in the team executes the associated code block.

- ❑ It differs from the single construct in that there is no implicit barrier at the end.

OpenMP: Master Construct

```
#pragma omp parallel
{
    #pragma omp master
    {
        std::cout << "Paralel Region #1 : I'm Thread " << omp_get_thread_num() << std::endl;
    }
}

#pragma omp parallel
{
    if(omp_get_thread_num()==0)

         std::cout << "Parallel Region #2 : I'm Thread " << omp_get_thread_num() << std::endl;
}
```

The master construct can be implemented with an if-statement as shown in the code snippet above.

OpenMP: ThreadPrivate Clause

By default, global data is shared.

- However, in some situations, we may need or would prefer to have private data that persists throughout the computation.
- This is where the *threadprivate* clause comes in handy.
- The effect of the *threadprivate* clause is that the named global-lifetime objects are replicated so that each thread has its own copy.

OpenMP: ThreadPrivate Clause

- That is, each threads gets a private or a local copy of the specified global variables.
- The syntax is

```
#pragma omp threadprivate (list)
```

- Restriction is that
  - The dynamic threads mechanism must be turned off with *omp_set_dynamic(false)* so that the number of threads across different parallel regions remains constant.

OpenMP: Nested Parallelism

If a thread in a team executing a parallel region encounters another parallel region, it creates a new team and becomes the master thread of that new team.

- This is generally referred to in OpenMP as "**nested parallelism**".

By default, this feature is disabled, meaning that the thread that encounters a nested parallel region will not form a new team, i.e., the encountered parallel region is inactive.

- The *omp_get_nested()* routine can be used to test if nested parallelism is enabled.

OpenMP: Nested Parallelism

```
if(omp_get_nested())

  std::cout << "Nested Parallelism is ON." << std::endl;

else

  std::cout << "Nested Parallelism is OFF." << std::endl;

#pragma omp parallel
{
  #pragma omp critical
  std::cout << "Thread " << omp_get_thread_num() << " executes the outer parallel region." << std::endl;

  #pragma omp parallel
  {

      #pragma omp critical
      std::cout << "\tThread " << omp_get_thread_num() << " executes the inner parallel region." << std::endl;

  }/*--- End of Inner Parallel Region ---*/


}/*--- End of Outer Parallel Region ---*/
```

OpenMP: Nested Parallelism

Executing the code snippet on the previous slide might give you output that looks similar to the following output:

```
Nested Parallelism is OFF.
Thread 3 executes the outer parallel region.
Thread 0 executes the outer parallel region.
        Thread 0 executes the inner parallel region.
        Thread 0 executes the inner parallel region.
Thread 1 executes the outer parallel region.
        Thread 0 executes the inner parallel region.
Thread 2 executes the outer parallel region.
        Thread 0 executes the inner parallel region.
```

To enable nested parallelism, the *OMP_NESTED* environment variable needs to be set to true:

- ❑ This *OMP_NESTED* environment variable is currently deprecated by the standard.
- ❑ In a future version of OpenMP, you might need to use a different mechanism for enabling and disabling nested parallelism.

OpenMP: Nested Parallelism

```
Nested Parallelism is ON.
Thread 3 executes the outer parallel region.
Thread 0 executes the outer parallel region.
Thread 2 executes the outer parallel region.
        Thread 0 executes the inner parallel region.
        Thread 2 executes the inner parallel region.
        Thread 1 executes the inner parallel region.
        Thread 3 executes the inner parallel region.
Thread 1 executes the outer parallel region.
        Thread 0 executes the inner parallel region.
        Thread 1 executes the inner parallel region.
        Thread 2 executes the inner parallel region.
        Thread 3 executes the inner parallel region.
        Thread 3 executes the inner parallel region.
        Thread 1 executes the inner parallel region.
        Thread 0 executes the inner parallel region.
        Thread 2 executes the inner parallel region.
        Thread 2 executes the inner parallel region.
        Thread 1 executes the inner parallel region.
        Thread 3 executes the inner parallel region.
        Thread 0 executes the inner parallel region.
```

After enabling nested parallelism by setting the *OMP_NESTED* environment variable to true.

OpenMP: Collapse Clause

A short loop (one with a small number of iterations) has a downside in a serial program:

- ❑ The loop distribution overhead might dominate the performance if there is no sufficient work to do per loop iteration.
- ❑ A short loop might not only limit the number of threads that may be used, but also leads to load imbalance if the number of iterations is not a multiple of the number of threads.

For perfectly nested rectangular loops, we can parallelize multiple loops in the loop nest using the *collapse* clause by mering or "collapsing" the multiple loops in the loop nest into one single and longer loop.

References

[1] *Barbara Chapman, Gabriele Jost, and Ruud van der Pas. 2007. Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press.*

[2] *Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. 2001. Parallel programming in OpenMP. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.*