OpenMP - 2

CSE 625

Fall 2022

Computer Science and Engineering

Computer University of Louisville

Contents

- Synchronization Constructs
- OpenMP Applications

Sources and References

[1] Official OpenMP Website

http://www.openmp.org

[2] Microsoft OpenMP Functions Reference

https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-functions?view=vs-2019

[3] VS 2019 OpenMPExamples Project

Contents

- Synchronization Constructs
- OpenMP Applications

Why Synchronization?

- Consider a simple example where two threads on two different processors are both trying to increment a variable x at the same time (assume x is initially 0):

THREAD 1:

```
increment(x)
{
    x = x + 1;
}
```

THREAD 1:

```
10  LOAD A, (x address)
20  ADD A, 1
30  STORE A, (x address)
```

THREAD 2:

```
increment(x)
{
    x = x + 1;
}
```

THREAD 2:

```
10  LOAD A, (x address)
20  ADD A, 1
30  STORE A, (x address)
```

One possible execution sequence:

Thread 1 loads the value of x into register A.
Thread 2 loads the value of x into register A.
Thread 1 adds 1 to register A
Thread 2 adds 1 to register A
Thread 1 stores register A at location x
Thread 2 stores register A at location x

The resultant value of x will be 1, not 2 as it should be.

Synchronization Construct master

- Syntax

```
#pragma omp master
structured_block
```

- The *master* directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code.

- There is no implied barrier associated with this directive.

- It is illegal to branch into or out of master block.

Synchronization Constructs – master Example

```
#include <iostream>
#include <omp.h>

void masterDemo()
{
   const int n = 9;
   int i, a, b[n];

   for (i = 0; i < n; i++)
       b[i] = -1;
```

Synchronization Construct master Example –continued

```
    #pragma omp parallel shared(a,b) private(i)
    {
        #pragma omp master
        {
           a = 10;
           printf("Master construct is executed by thread %d\n",
              omp_get_thread_num());
        }
        #pragma omp barrier

        #pragma omp for
        for (i = 0; i < n; i++)
                b[i] = a;
     } /*-- End of parallel region --*/

     printf("After the parallel region:\n");
     for (i = 0; i < n; i++)
        printf("b[%d] = %d\n", i, b[i]);

}
```

Synchronization Construct critical

- Syntax

```
#pragma omp critical [ name ]
structured_block
```

- The *critical* directive specifies a region of code that must be executed by only one thread at a time.

- The optional name enables multiple different CRITICAL regions to exist:
  - Names act as global identifiers. Different CRITICAL regions with the same name are treated as the same region.
  - All CRITICAL sections which are unnamed, are treated as the same section.

Synchronization Construct critical Example

```
void criticalDemo()
{
   const int n = 25;
   int i;
   int sum = 0, TID, a[n];
   int ref = (n-1)*n/2;
   int sumLocal;

   for (i=0; i<n; i++)
       a[i] = i;

   #pragma omp parallel
   {
      #pragma omp single
         printf("Number of threads is %d\n", omp_get_num_threads());
   }
   printf("Value of sum prior to parallel region: %d\n",sum);
```

Synchronization Construct critical Example- continued

```
   #pragma omp parallel default(none) shared(n,a,sum) \
           private(TID,sumLocal)
   {
      TID = omp_get_thread_num();
      sumLocal = 0;
      #pragma omp for
        for (i=0; i<n; i++)
          sumLocal += a[i];
      #pragma omp critical (update_sum)
      {
        sum += sumLocal;
        printf("TID=%d: sumLocal = %d sum = %d\n",TID,sumLocal,sum);
      }
   } /*-- End of parallel region --*/
   printf("Value of sum after parallel region: %d\n",sum);
   printf("Check results: sum = %d (should be %d)\n", sum,ref);
} /* end of criticalDemo() */
```

Synchronization Construct barrier

- Syntax

```
#pragma omp barrier  newline
```

- The barrier directive synchronizes all threads in the team.

- When a barrier directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

Synchronization Construct barrier Example

```
#void barrierDemo ()
{
   const int n =  10;
   int TID, i, a[n], b[n], ref[n];

   printf("Sleep and barrier test:\n\n");
   #pragma omp parallel private(TID)
   {
      TID = omp_get_thread_num();

      if ( TID < omp_get_num_threads()/2 ) Sleep(3000);
      print_time(TID,"before");
      #pragma omp barrier
      print_time(TID,"after ");
   } /*-- End of parallel region --*/
```

Synchronization Construct barrier Example -continued

```
// Another barrier test
printf("\n\nAnother barrier test:\n\n");
for (i=0; i<n; i++)
{
    b[i] = 2*(i+1);
    ref[i] = i + b[i];
}
#pragma omp parallel private(i) shared(n, a, b)
{
   #pragma omp for schedule(dynamic,1) nowait
   for (i=0; i<n; i++)
       a[i] = i;
   #pragma omp barrier
   #pragma omp for schedule(dynamic,1) nowait
   for (i=0; i<n; i++)
       a[i] += b[i];
}
}
```

Synchronization Construct atomic

- Syntax

```
#pragma omp atomic
    statement_expression
```

- The atomic directive specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it. In essence, this directive provides a mini-CRITICAL section.

- The directive applies only to a single, immediately following statement.

Synchronization Construct atomic Example

```
void atomicDemo()
{
   int ic, i, n = 7;
   ic = 0;

   #pragma omp parallel for shared(ic,n) private(i)
   for (i = 0; i < n; i++)
   {
      #pragma omp atomic
         ic += 1;
   }
   printf("Counter = %d\n",ic);

}
```

Synchronization Construct flush

- Syntax

```
#pragma omp flush (list)
```

- The flush directive identifies a synchronization point at which the implementation must provide a consistent view of memory. Thread-visible variables are written back to memory at this point.

- The optional list contains a list of named variables that will be flushed in order to avoid flushing all variables. For pointers in the list, note that the pointer itself is flushed, not the object it points to.

- Implementations must ensure any prior modifications to thread-visible variables are visible to all threads after this point; ie. compilers must restore values from registers to memory, hardware might need to flush write buffers, etc.

Synchronization Construct ordered

- Syntax

```
#pragma omp for ordered [clauses...]
    (loop region)

#pragma omp ordered
   structured_block
   (end of loop region)
```

- The ordered directive specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor.

- Threads will need to wait before executing their chunk of iterations if previous iterations haven't completed yet.

Synchronization Construct ordered - continued

- Used within a for loop with an ORDERED clause
- The ORDERED directive provides a way to "fine tune" where ordering is to be applied within a loop. Otherwise, it is not required.
- An ORDERED directive can only appear in the dynamic extent of the following directives:

  `for` or `parallel for`

- Only one thread is allowed in an ordered section at any time.
- A loop which contains an ORDERED directive, must be a loop with an ORDERED clause.

Synchronization Construct ordered Example

```
void orderedDemo ()
{
   const int n = 9;
   int i, TID, a[n];
   for (i=0; i<n; i++)
       a[i] = i;
   #pragma omp parallel for default(none) ordered schedule(runtime) \
        private(i,TID) shared(n,a)
   for (i=0; i<n; i++)
   {
       TID = omp_get_thread_num();
       printf("Thread %d updates a[%d]\n",TID,i);
       a[i] += i;

       #pragma omp ordered
       {printf("Thread %d prints value of a[%d] = %d\n",TID,i,a[i]);}
   }  /*-- End of parallel for --*/
 }
```

Contents

- OpenMP Concepts
- Parallel Construct
- Work-Sharing Constructs
- Synchronization Constructs
- OpenMP Applications

OpenMP Programming Considerations

- Correctness considerations

  In addition to sequential computing correctness, shared memory parallel programming introduces another dimention of correctness issues:

  - Data race conditions
  - Dependency on the number of concurrently executing threads

- Performance considerations

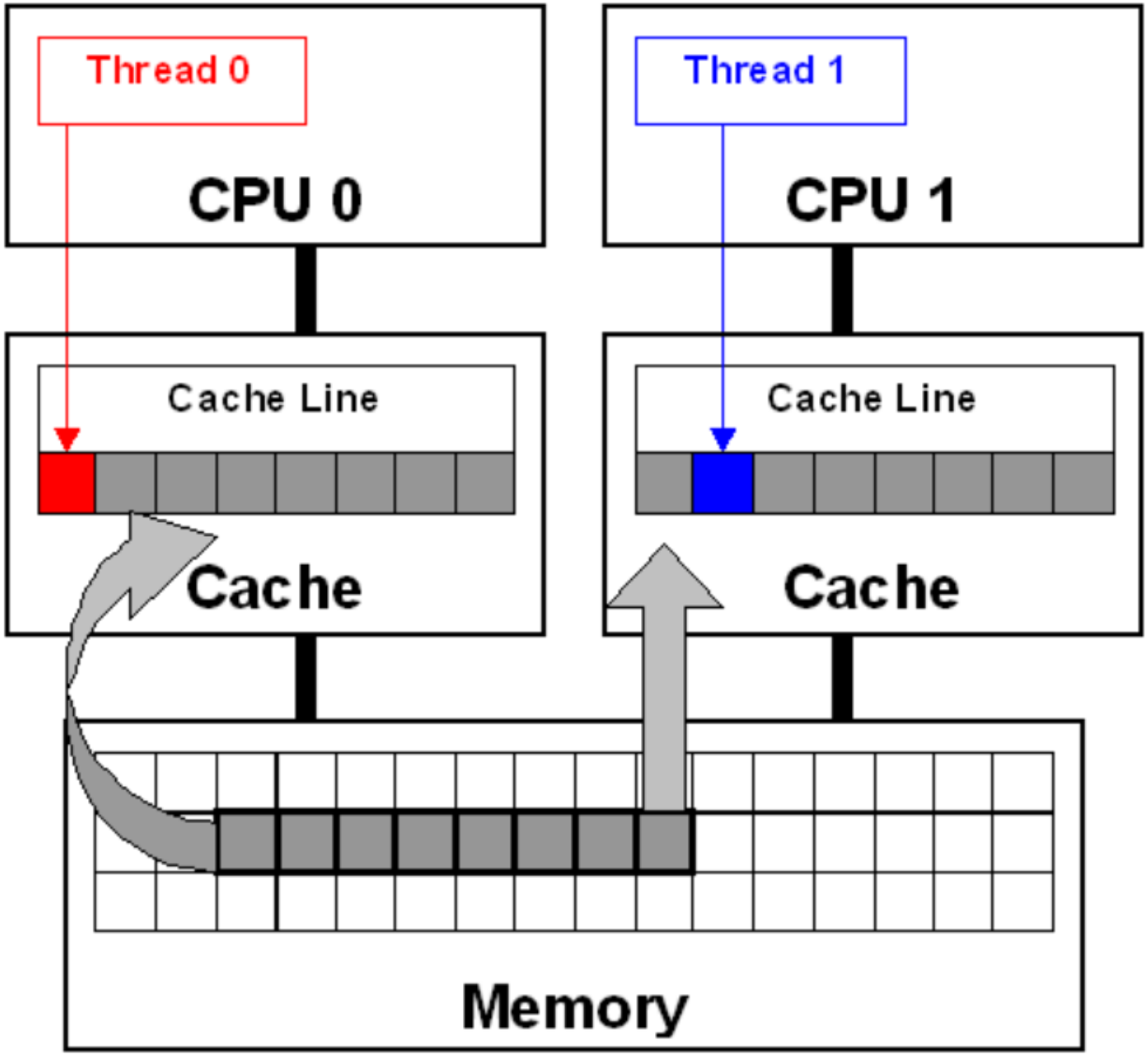
  - Amdahl's law
  - Costs of synchronization constructs
  - Memory wall
    - Cache coherence
    - False sharing, etc.

OpenMP False Sharing Problem

- The memory system must guarantee cache coherence. False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces an update, which hurts performance.

- This circumstance is called *false sharing* because each thread is not actually sharing access to the same variable. Access to the same variable, or true sharing, would require programmatic synchronization constructs to ensure ordered data access.

OpenMP False Sharing Problem - continued



OpenMP False Sharing Problem - continued

```
double sum=0.0, sum_local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)
{
    int me = omp_get_thread_num();
    sum_local[me] = 0.0;
    #pragma omp for
    for (i = 0; i < N; i++)
        sum_local[me] += x[i] * y[i];
    #pragma omp atomic
        sum += sum_local[me];
}
```

- There is a potential for false sharing on array `sum_local`. This array is dimensioned according to the number of threads and is small enough to fit in a single cache line. When executed in parallel, the threads modify different, but adjacent, elements of `sum_local` (the source line shown in red), which invalidates the cache line for all processors.

OpenMP C Runtime Functions (MSDN)

Function	Description
omp_destroy_lock	Uninitializes a lock.
omp_destroy_nest_lock	Uninitializes a nestable lock.
omp_get_dynamic	Returns a value that indicates if the number of threads available in subsequent parallel region can be adjusted by the run time.
omp_get_max_threads	Returns an integer that is equal to or greater than the number of threads that would be available if a parallel region without num_threads were defined at that point in the code.
omp_get_nested	Returns a value that indicates if nested parallelism is enabled.
omp_get_num_procs	Returns the number of processors that are available when the function is called.
omp_get_num_threads	Returns the number of threads in the parallel region.
omp_get_thread_num	Returns the thread number of the thread executing within its thread team.
omp_get_wtick	Returns the number of seconds between processor clock ticks.
omp_get_wtime	Returns a value in seconds of the time elapsed from some point.
omp_in_parallel	Returns nonzero if called from within a parallel region.
omp_init_lock	Initializes a simple lock.
omp_init_nest_lock	Initializes a lock.
omp_set_dynamic	Indicates that the number of threads available in subsequent parallel region can be adjusted by the run time.
omp_set_lock	Blocks thread execution until a lock is available.
omp_set_nest_lock	Blocks thread execution until a lock is available.
omp_set_nested	Enables nested parallelism.
omp_set_num_threads	Sets the number of threads in subsequent parallel regions, unless overridden by a num_threads clause.
omp_test_lock	Attempts to set a lock but does not block thread execution.
omp_test_nest_lock	Attempts to set a nestable lock but does not block thread execution.
omp_unset_lock	Releases a lock.
omp_unset_nest_lock	Releases a nestable lock.

OpenMP Applications

- sinx (Project 2)

  OpenMP with/without AVX

- Matrix multiplication

- Fractal (i.e. Mandelbrot set)

- Monte Carlo simulations

  Random number generators
  pi estimation
  Mandelbrot set area estimation

- Reductions / Prefix sum (Prefix scan)

OpenMP Applications - continued

- Counting primes
- Graph search problems

  Dijkstra's algorithm

- Solving differential equations numerically