Parallel and Distributed Computing
CS3006 (BCS-6C/6D)
Lecture 10

Instructor: Dr. Syed Mohammad Irteza

Assistant Professor, Department of Computer Science, FAST

23 February, 2023

Previous Lecture

- Intro to OpenMP
- Shared and private variables
- Loop worksharing – OpenMP `for` construct
- Example: Parallel Pi
- Synchronization – `critical` section
- Scheduling – static and dynamic (chunk-size)
- OpenMP Reduction (all-to-one abstraction)

OpenMP reduction clause

- reduction (op : list)
- Inside a parallel work a local copy of each list variable is made
- List variable is initialized depending on the “op”

  (e.g. 0 for “+”)
- Compiler updates each local copy
- Local copies are reduced into a single value and combined with the original global value

```
double ave=0.0, A[MAX]; int i;
#pragma omp parallel for reduction (+:ave)
for (i=0; i< MAX; i++) { ave + = A[i]; }
ave = ave/MAX;
```

Reduction Clause

- Reductions are so common that OpenMP provides support for them
- We may add a reduction clause to **`parallel for`** pragma
- We need to specify a reduction operation and a reduction variable
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop
- The reduction clause has this syntax: **`reduction (<op> :<variable>)`**

- **Operators**
  - \+ Sum
  - * Product
  - & Bitwise AND
  - | Bitwise OR
  - ^ Bitwise exclusive OR
  - && Logical AND
  - || Logical OR

Serial Pi Program

```
static long num_steps = 100000;
double step;
void main () {
  int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  for (i=0; i< num_steps; i++){
    x = (i+0.5) * step;
    sum = sum + 4.0/(1.0+x*x);
  }
    pi = step * sum;
}
```

Pi program using reduction clause

```
static long num_steps = 100000;
double step;
void main () {
  int i; double pi, sum = 0.0;
  step = 1.0/(double) num_steps;
#pragma omp parallel {
  double x;
#pragma omp for reduction(+:sum)
  for (i=0; i< num_steps; i++){
    x = (i+0.5) * step;
    sum = sum + 4.0/(1.0+x*x);
  }
}
  pi = step * sum;
}
```

Atomic

```
int ic, i, n;
ic = 0;
#pragma omp parallel shared(n, ic) private(i)
  for (i=0; i<n; i++)
  {
      #pragma omp atomic
      ic = ic + bigfunc();
  }
```

- `ic` is a counter. The atomic construct ensures that no updates are lost when multiple threads are updating a counter value
- The atomic construct does not prevent multiple threads from executing the function bigfunc() at the same time
- Supported functions: +, *, -, /, &, ^, |, <<, >>

Source: https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf

Atomic -- Examples

```
#pragma omp atomic update
      k += n*mass; // k is updated atomically

#pragma omp atomic read
      tmp = c; // c is read atomically

#pragma omp atomic write
      count = n*m; // count is written atomically

#pragma omp atomic capture
      { d = v; v += n; } // atomically update v, but capture original value in d

#pragma omp atomic capture
      o = ++c; // atomically update c, then capture that value
```

Source : http://www.physics.ntua.gr/~konstant/HetCluster/intel12.1/compiler_c/main_cls/cref_cls/common/cppref_pragma_omp_atomic.htm

Sections Constructs

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        (void) funcA();

        #pragma omp section
        (void) funcB();
    } /*-- End of sections block --*/
} /*-- End of parallel region --*/
```

Example: Sections / section

```
#include <omp.h>
#define N 1000
main () {
int i; float a[N], b[N], c[N], d[N];

/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }

#pragma omp parallel shared(a,b,c,d) private(i)
  {
  #pragma omp sections nowait
    {
    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];

    }  /* end of sections */
  }  /* end of parallel section */
}
```

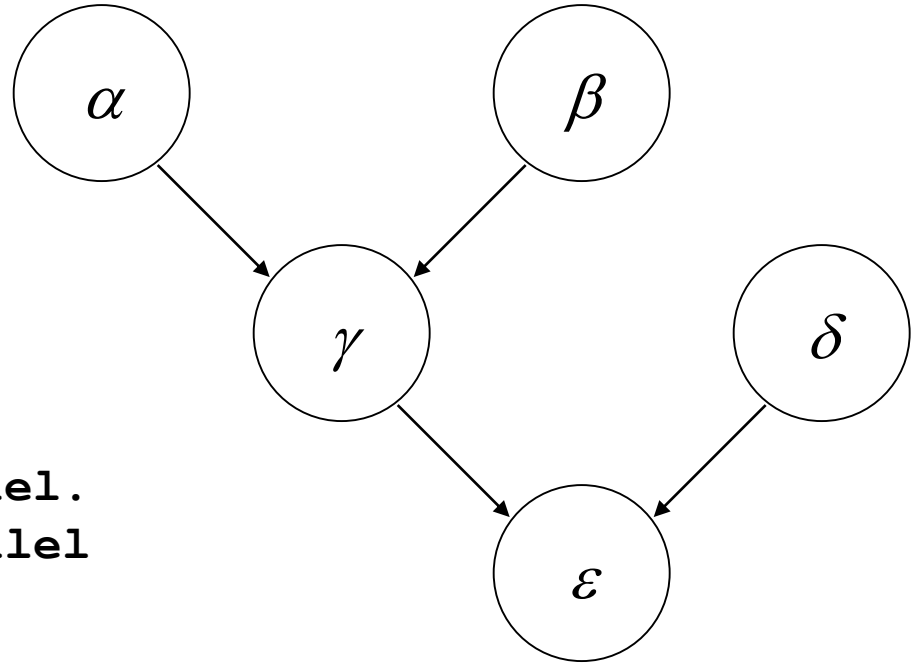
Source: https://hpc-tutorials.llnl.gov/openmp/sections_directive/
Note: There is an implied barrier at the end (slide 2: https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/tasks.1pp.pdf)

Functional Parallelism

Another approach

```
v = alpha();
W = beta();
x = gamma(v,w);
y = delta();
printf("%6.2f\n",epsilon(x,y));
```

- **Execute alpha and beta in parallel.**
- **Execute gamma and delta in parallel**



sections pragma

```
#pragma omp parallel num_threads(2)
    {
       #pragma omp sections
       {
         #pragma omp section //optional
                 v = alpha();
         #pragma omp section
            w = beta();
       } // here an implicit barrier exists

              #pragma omp sections
       {
            x = gamma(v, w);
         #pragma omp section
            y = delta();
       }
    }
    printf ("%6.2f\n", epsilon(x,y));
```

omp sections pragma

- Appears inside a parallel block of code
- This pragma distributes enclosed sections among the threads in the team

- The difference between *omp parallel sections* and *omp sections* is that,
  - ***Omp parallel sections*** generate its own team of threads
  - While simple ***omp sections*** pragma uses existing team of threads and distributes section among the threads
- If multiple sections pragmas are inside one parallel block, this may reduce fork/join costs

Extra Reading on `sections`

- Use of

```
#pragma omp sections
```

- instead of

```
#pragma omp parallel sections
```

- will lead to:
  - https://610yilingliu.github.io/2020/07/17/OpenMPtasksection/

Combined constructs

Full version	Combined construct

#pragma omp parallel
{
#pragma omp for
for-loop
}
	#pragma omp parallel for
	for-loop

#pragma omp parallel
{
#pragma omp sections
{
[#pragma omp section]
structured block
[#pragma omp section
structured block]
...
}
}
	#pragma omp parallel sections
	{
	[#pragma omp section]
	structured block
	[#pragma omp section
	structured block]
	...
	}

Single Constructs

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

/* A barrier is automatically inserted here */

      #pragma omp for
      for (i=0; i<n; i++) b[i] = a;

} /*-- End of parallel region --*/
```

Example -- `single`

```
// omp_single.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main() {
   #pragma omp parallel num_threads(2)
   {
      #pragma omp single
      // Only a single thread can read the input.
      printf_s("read input\n");

      // Multiple threads in the team compute the results.
      printf_s("compute results\n");

      #pragma omp single
      // Only a single thread can write the output.
      printf_s("write output\n");
    }
}
```

Output:

read input
compute results
compute results
write output

Source:
https://learn.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170

omp barrier

- Each thread that encounters this pragma must wait until all threads in the team have arrived. After the last thread of the team arrives, all threads are released and may continue execution of the enclosing parallel region.
- The code for the enclosing parallel region must be arranged so that either all or none of the threads encounter the pragma.

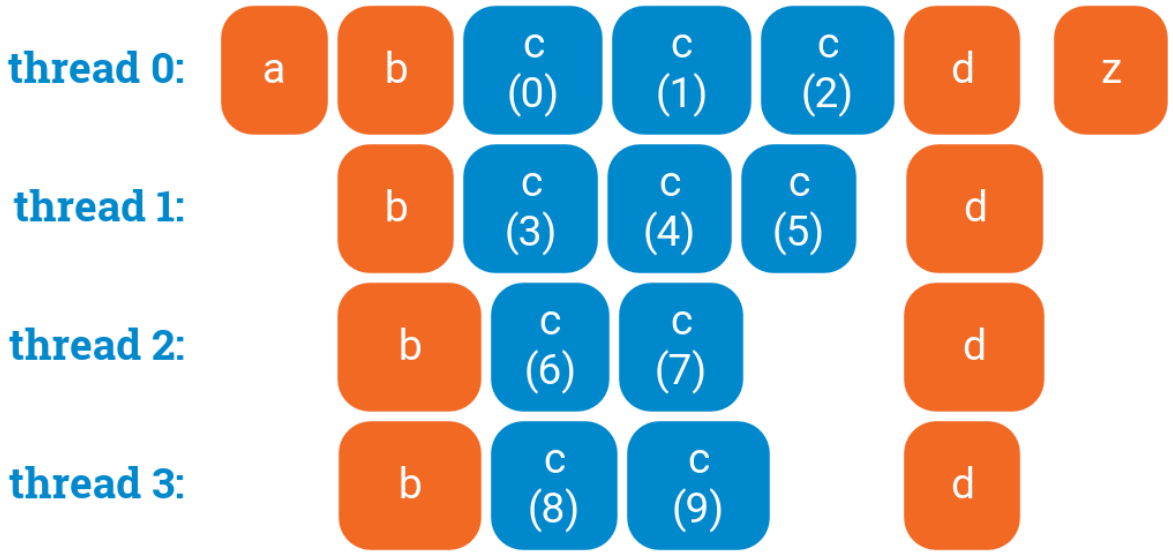
This example demonstrates how to use this pragma to ensure that all threads have executed the 1st loop before executing the 2nd loop:
source: http://portal.nacad.ufrj.br/online/intel/compiler_c/common/core/GUID-FD888904-4AB9-4E25-800D-6084117607B4.htm

```
#include <omp.h>
void work1(int k) {
  // large amount of work
}
void work2(int k) {
  // large amount of work that must all happen after work1 is finished
}

int main() {
  int n=1000000;
  #pragma omp parallel private(i) shared(n) {
    #pragma omp for
    for (i=0; i<n; i++)
      work1(i);

    #pragma omp barrier
    #pragma omp for
    for (i=0; i<n; i++)
      work2(i);
  }
 return 0;
}
```

OpenMP `nowait` clause

- When you use a `parallel` region, OpenMP will automatically wait for all threads to finish before execution continues.
- There is also a synchronization point after each `omp for` loop; here no thread will execute `d()` until all threads are done with the loop:

```
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```

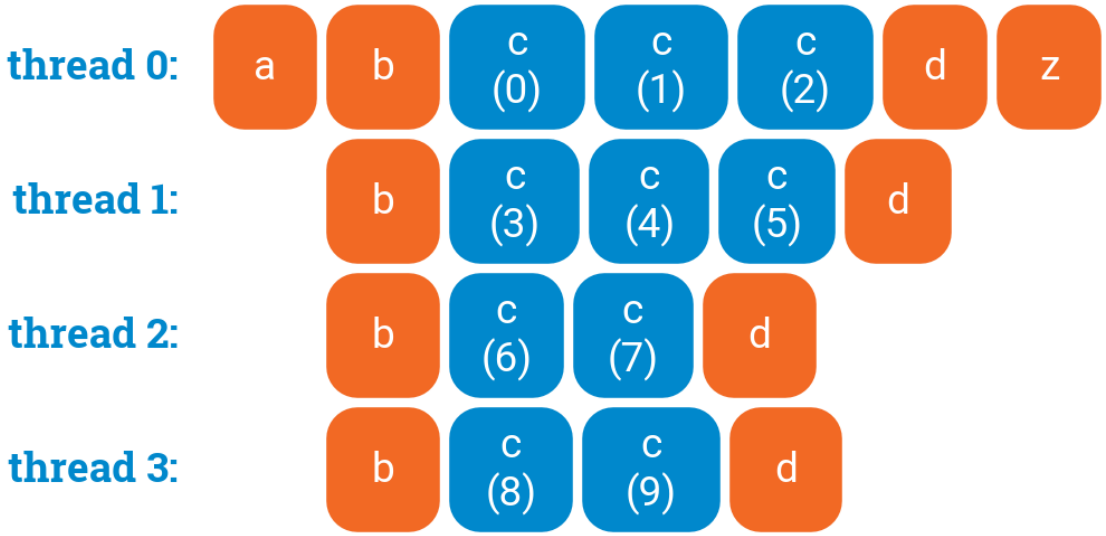


Source: https://ppc.cs.aalto.fi/ch3/nowait/

OpenMP **nowait** clause

- However, if you do not need synchronization after the loop, you can disable it with `nowait`:

```
a();
#pragma omp parallel
{
    b();
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```



Source: https://ppc.cs.aalto.fi/ch3/nowait/

References

1. Kumar, V., Grama, A., Gupta, A., & Karypis, G. (1994). *Introduction to parallel computing* (Vol. 110). Redwood City, CA: Benjamin/Cummings.
2. Quinn, M. J. Parallel Programming in C with MPI and OpenMP, (2003).
3. https://www.cse.iitk.ac.in/users/swarnendu/courses/autumn2019-cs698l/OpenMP.pdf
4. https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf