LAU
Lebanese American University

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Shared Parallel Programming Using OpenMP

Instructor: Haidar M. Harmanani

Spring 2021

OpenMP
Enabling HPC since 1997

Why OpenMP?

- Thread libraries are hard to use
  - Pthreads have many library calls for initialization, synchronization, thread creation, condition variables, etc.
  - Programmer must code with multiple threads in mind
- Synchronization between threads introduces a new dimension of program correctness

Why OpenMP ?

- OpenMP is a parallel programming model for Shared-Memory machines
  - All threads have access to a shared main memory
  - Each thread may have private data.
- Parallelism is expressed explicitly by the programmer.
- Using the *worksharing* constructs, the work can be distributed among the threads of a team.

Why OpenMP ?

```
int main() {

    printf( "Hello, World!\n" );

    return 0;
  }
```

Why OpenMP ? Hello World Pthread Version

```
int main() {
   pthread_attr_t attr;
   pthread_t threads[16];
   int tn;

   pthread_attr_init(&attr);

    for(tn=0; tn<16; tn++) {
      pthread_create(&threads[tn], &attr, SayHello, NULL);
   }

   for (tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
   }
   return 0;
 }

void* SayHello(void *foo) {
   printf( "Hello, world!\n" );
   return NULL;
 }
```

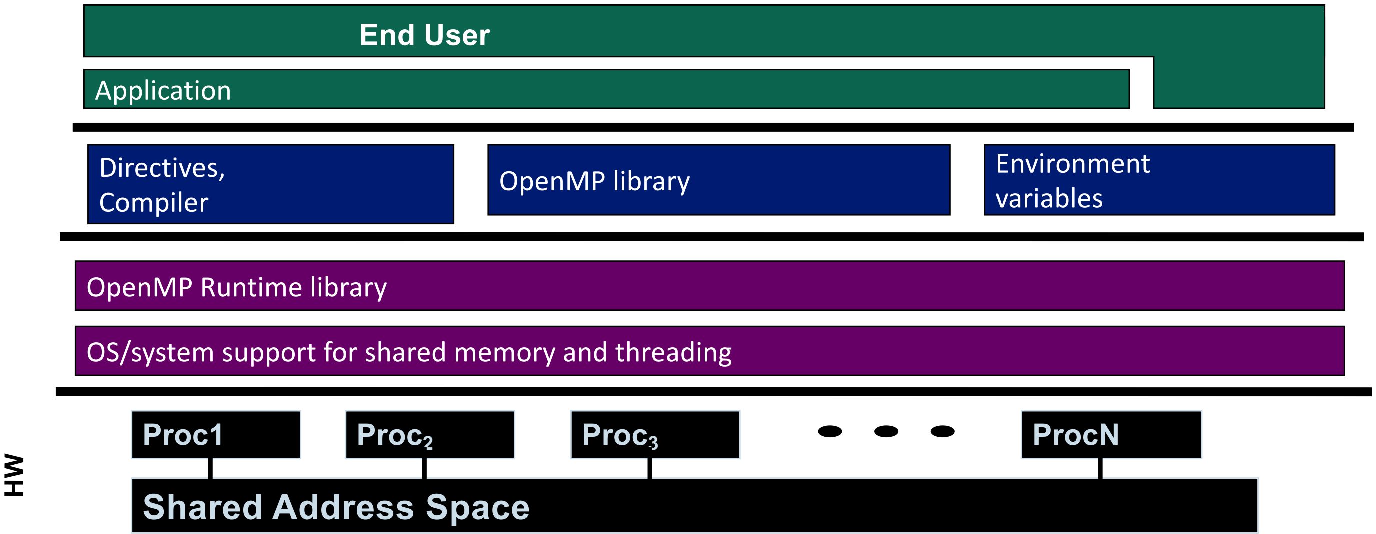
Why OpenMP ? Hello World Pthread Version

```
int main()
{
    omp_set_num_threads(16);

    // Do this part in parallel
    #pragma omp parallel
    {
      printf( "Hello, World!\n" );
    }

    return 0;
}
```

OpenMP: Solution Stack



End User

Application

Directives, Compiler

OpenMP library

Environment variables

OpenMP Runtime library

OS/system support for shared memory and threading

Proc1

Proc$_2$

Proc$_3$

● ● ●

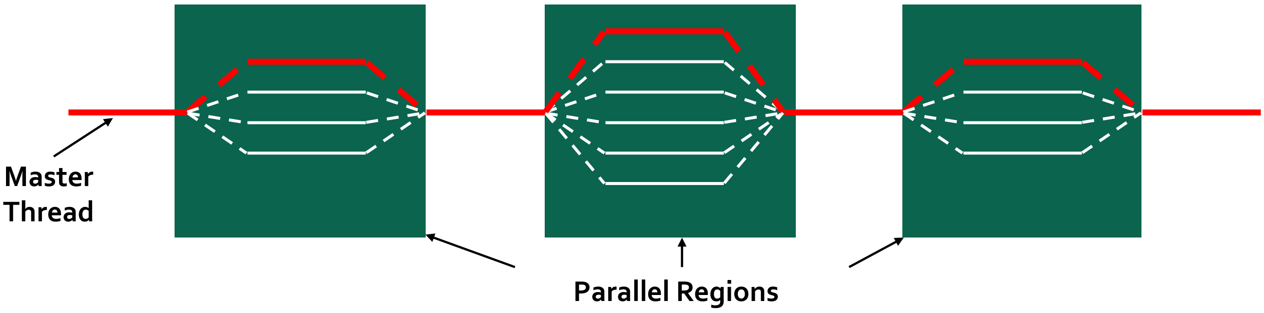
ProcN

Shared Address Space

HW

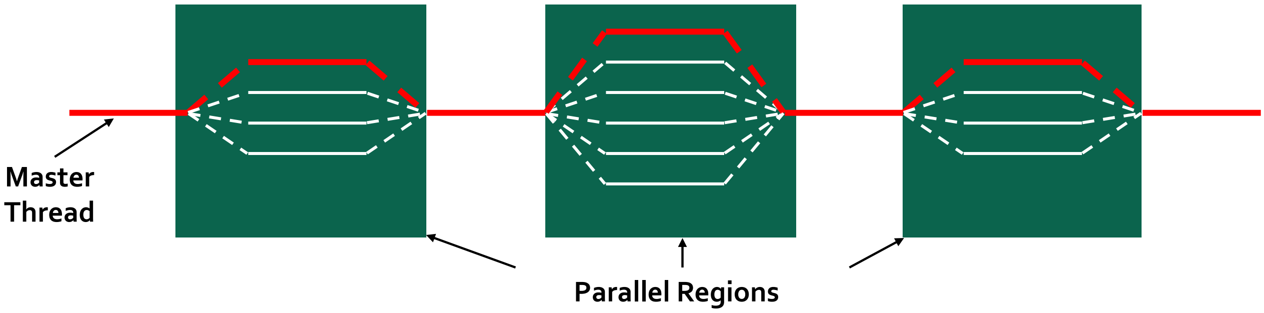
OpenMP Execution Model

- OpenMP uses a fork join methodology to implement parallelism
  - Master thread spawns a team of threads as needed
- Parallel directive creates a team of threads with a specified block of code executed by the multiple threads in parallel.
  - The exact number of threads in the team determined by one of several ways.



OpenMP Execution Model

- Worker threads are spawned at Parallel Regions, together with the Master they form the Team of threads.
- In between Parallel Regions the Worker threads are put to sleep.
- The `OpenMP` Runtime takes care of all thread management work



Getting Started with OpenMP

OpenMP Syntax

- OpenMP constructs are compiler directives or pragmas
  - For C and C++, the pragmas take the form:

```
#pragma omp directive-name [clause[ [,] clause] ... ] new-line
```

Hello Worlds

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

int main() {
    #pragma omp parallel
    {
       int i;
       int ID = omp_get_thread_num();

        printf("Hello World\n");
        for(i=0;i<6;i++)
            printf("Iter:%d, %d\n”,i, ID);
    }
    printf("GoodBye World\n");
}
```

Switches for compiling and linking:

```
gcc –fopenmp filename
```

Begin Parallel region

Runtime library function to return a thread ID.

End Parallel region

#pragma omp parallel

- This pragma will execute in parallel what's next : next line, next loop, next block of code between brackets.
- But the parallel keyword alone won't distribute the workload on different threads. For that we'll see the constructs for, task, section.
- parallel will execute the same thing several times in parallel.

#pragma omp parallel

-
  ```
  printf(“before\n”);
  #pragma omp parallel
  printf(“parallel\n”);
  printf(“after\n”);
  ```
- If you compile as usual (without the -openmp flag), will return :

```
before
parallel
after
```

- A pragma is not regular code and require a special flag to be used by the compiler.

#pragma omp parallel

-
  ```
  printf(“before\n”);
  #pragma omp parallel
  printf(“parallel\n”);
  printf(“after\n”);
  ```
- If compile with the -openmp flag, will return on a quad-core machine:

```
before
parallel
parallel
parallel
parallel
after
```

#pragma omp parallel

-
  ```
  #pragma omp parallel
  {
     printf(“start ”);
     printf(“end ”);
  }
  ```
  Will return on a dual-core machine:
  `start start end end`
  or (depending on various conditions)
  `start end start end`
- Parallel execution does NOT implicate anything about the order of execution.

Structured Blocks

- Most `OpenMP` constructs apply to structured blocks
  - Exactly one entry point at the top
  - Exactly one exit point at the bottom
  - Branching in or out is not allowed
  - Terminating the program is allowed (abort / exit)

OMP Parallel Regions

Structured Blocks

- A parallel region consists of a structured block of code
- A structured block of code is a code fragment with a single point of entry into the block at the top of the block, and one exit to the block at the bottom – AND no breaks out of the block.

Structured Blocks: Example

- Create threads in `OpenMP` using the "omp parallel" pragma.

```
#pragma omp parallel
{
  int id = omp_get_thread_num();

more: res[id] = do_big_job (id);

  if (conv (res[id]) goto more;
}
printf (“All done\n”);
```

A structured block

```
if (go_now()) goto more;
#pragma omp parallel
{
  int id = omp_get_thread_num();
more:  res[id] = do_big_job(id);
  if (conv (res[id]) goto done;
  goto more;
}
done: if (!really_done()) goto more;
```

Not a structured block

OpenMP: Parallel Regions

- Example: create a 4 parallel thread regions

Each thread redundantly executes the code within the structured block

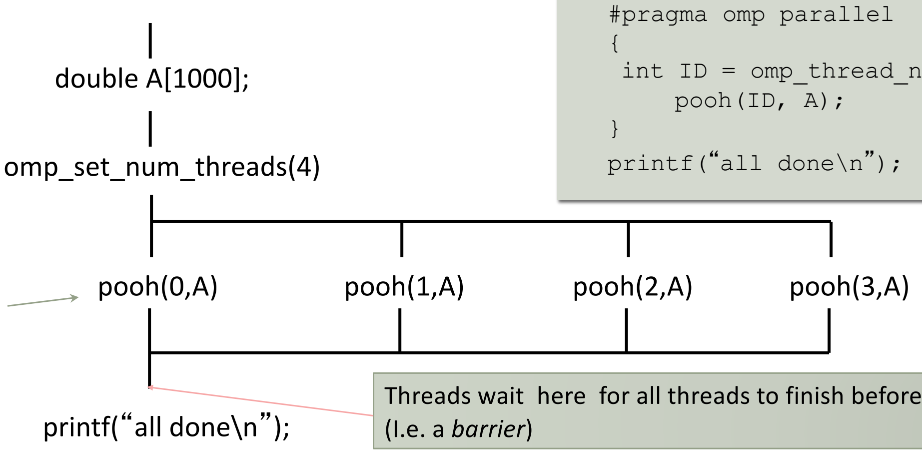
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_thread_num();
        pooh(ID,A);
}
```

Each thread calls pooh(ID) for ID = 0 to 3

OpenMP: Parallel Regions

Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
 int ID = omp_thread_num();
     pooh(ID, A);
}
printf("all done\n");
```



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (I.e. a *barrier*)

single, master and wait clauses

#pragma omp single

- **`single`** will execute the next block of code once by the **first available thread**.
-
  ```
  #pragma omp parallel
  {
     printf(“start ”);
    #pragma omp single
     printf(“end ”);
  }
  ```
  Will return on a dual-core machine:
  ```
  start start end
  ```

#pragma omp master

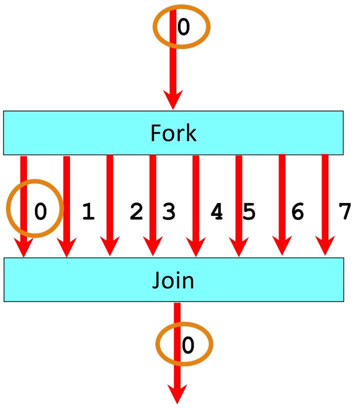
- single will execute the next block of code once by the **`master`** thread.

-
  ```
  #pragma omp parallel
  {
     printf("start ");
    #pragma omp master
     printf("end ");
  }
  ```

  Will return on a dual-core machine:
  `start start end`

#pragma omp master

- Master thread has ID 0
  - Only thread that exists in sequential regions
  - Depending on implementation, may have special purpose inside parallel regions
  - Some special directives affect only the master thread (like master)



#pragma omp … nowait

- **`nowait`** will prevent the join phase at the end of a parallel region (an implicit barrier) from blocking execution.

-
  ```
  #pragma omp parallel for nowait
  for (int i=0;i<n;i++) {
     printf("i");
  }
  #pragma omp parallel for
  for (int j=0;j<m;j++) {
     printf("j");
  }
  ```
  Could print `iiiiiijijiijjjjjjjjjj`

---

for
worksharing construct

data decomposition

for worksharing construct

- The `omp parallel for` construct starts a parallel region by creating an optimal number of threads and maintaining a queue of iterations to execute and distribute them to the threads as needed.
- When all the iterations are executed, the parallel region will end and the code will go back to serial execution.

```
#pragma omp parallel for
for (i=0 ; i<N ; i++) {
     printf(“loop %d\n”,i);
}
```

---

for worksharing construct

- `parallel for` is a simple and flexible way to implement data decomposition:
  - Keep a simple loop structure
  - All the queue management is done automatically
  - Worksharing is handled automatically
- Iterations will not execute in a specific order or show the same behavior on different software or hardware environments
- If some variables are defined before the parallel region, they are shared between threads.
  - Sharing to read is safe but sharing and writing leads to parallel bugs. We'll see how to share variables safely.

task
worksharing construct

flexible task decomposition

#pragma omp task

- Allows parallelization of irregular problems
- unbounded loops
- recursive algorithms
- producer/consumer
- A task is composed of :
- Code to execute
- Data environment
- Internal control variables (ICV)

#pragma omp task

```
#pragma omp parallel
{
 #pragma omp single private(p)
 {
   while (p) {
     #pragma omp task
     processwork(p);
     p = p->next; # not in the task
   }
 } # end of the single region
} # end of the parallel region
```

---

synchronizations

- Tasks are guaranteed to be complete :
- At thread or task barriers
- At the directive :
  `#pragma omp barrier`
- At the directive :
  `#pragma omp taskwait`

barrier

```
#pragma omp parallel
{
        #pragma omp task
        foo();
        #pragma omp barrier

        #pragma omp single
        {
                #pragma omp task
                bar();
        }
}
```

task : Multiple foo() tasks created, one for each thread in the parallel region.

barrier : All foo() tasks are guaranteed to be completed here.

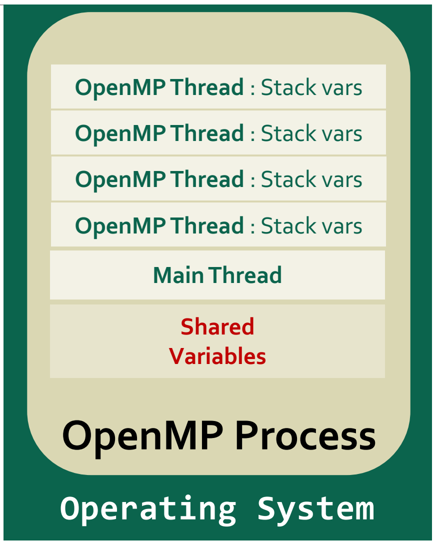
task-single : one bar() task is created because there is only 1 thread running with the single keyword.

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Sharing Variables in OpenMP

OpenMP Shared-Memory Model

- OpenMP worker threads and the master thread share the same process and variables.
- If variable scope includes the parallel region, it is shared by default
  - All the threads will read and write to the same memory location.



OpenMP Thread : Stack vars

OpenMP Thread : Stack vars

OpenMP Thread : Stack vars

OpenMP Thread : Stack vars

Main Thread

Shared Variables

OpenMP Process

`Operating System`

Scoping Rules

- Not everything is shared…
  - Examples of implicitly determined private variables:
    - Stack (local) variables in functions called from parallel regions are PRIVATE
    - Automatic variables **within** a statement block are PRIVATE
    - Loop **iteration variables** are private
    - Implicitly **declared private** variables within tasks will be treated as `firstprivate`
- Shared clause can be used to make items explicitly shared
  - Global variables are shared by default among tasks
    - File scope variables, namespace scope variables, static variables, variables with const-qualified type having no mutable member are shared, Static variables which are declared in a scope inside the construct are shared

Global Data

- Global data are shared and require special care
- A problem may arise in case multiple threads access the same memory section simultaneously:
  - Read-only data is no problem
  - Updates have to be checked for race conditions
  - It is the programmer's responsibility to deal with this situation
- In general one can do the following:
  - Split the global data into a part that is accessed in serial parts only and a part that is accessed in parallel
    - Manually create thread private copies of the latter
    - Use the thread ID to access these private copies

Shared by default

```
float x, y;
int i;
#pragma omp parallel for
for(i=0; i<N; i++) {
   x = a[i]; y = b[i];
   c[i] = x + y;
}
```

- This code is executing correctly in serial but may give a different results in parallel. Why?

Problem 1 : Race condition

- A race condition is nondeterministic behavior caused by the times at which two or more threads access a shared variable.
- Let's suppose we have 2 threads executing :
  `x = a[i]; y = b[i];`
  `c[i] = x + y;`
- If a thread can execute the two lines without having the other thread changing variables x and y, good
  - Not guaranteed.
- If the two threads have a mixed execution, the result `c` will be wrong.
- Race conditions may or may not be visible depending on various experimental conditions (number of cores, other software running, luck, ...)

Problem 2 : Corruption

- Independently from race conditions, writing to the same object or memory location from different threads without protection is risky.
  - Example : Different threads write to the serial output (console) at the same time.
  - If you are lucky, messages will intercalate nicely.
  - If you are not, the output may become garbled as bits of information representing the output text will be mixed together.

Problem 3 : Initialization

- If you use local copies instead of global variables to prevent race conditions and corruption, the last problem is initialization.
- Local variables created by the OpenMP layer may or may not be initialized, or initialized differently

---

Solutions

- Recode to prevent sharing
  - Explicit declarations using data scope clauses
  - Synchronization (more about this one later)

Data Scope Clauses

- `shared`
  - Declares variables in its list to be shared among all threads in the team
- `private`
  - Reproduce the variable for each task
    - Variables are un-initialized;
    - Any value external to the parallel region is undefined
- `firstprivate`
  - Combines the behavior of the **`private`** clause with automatic initialization of the variables in its list
- lastprivate
  - Combines the behavior of the **`private`** clause with a copy from the last loop iteration or section to the original variable object
- More about data scope clauses later
  - `Reduction,` `copyin` and `copyprivate`

Solution 1: Explicitly Change the Scope

```
 float x, y;
int i;
#pragma omp parallel for
for(i=0; i<N; i++) {
      x = a[i]; y = b[i];
      c[i] = x + y;
}
```

Before *: variables defined with global scope from the master thread, shared between threads.*

```
int i;
#pragma omp parallel for
for(i=0; i<N; i++) {
      float x, y;
      x = a[i]; y = b[i];
      c[i] = x + y;
}
```

After *: local variables defined locally. Nothing shared.*

Efficient and safe.

Solution 2: forcing serial execution of the critical block

```
float x, y;
int i;
#pragma omp parallel for
for(i=0; i<N; i++) {
      x = a[i]; y = b[i];
      c[i] = x + y;
}
```

Before : variables defined with global scope from the master thread, shared between threads.

```
float x, y;
int i;
#pragma omp parallel for
for(i=0; i<N; i++) {
#pragma omp critical
      x = a[i]; y = b[i]; c[i] = x + y;
}
```

After : same thing, but serial execution forced.

Safe but not scaling.

Solution 3: atomic

- Instead of protecting an entire block of code, is it enough to protect write accesses to a single shared variable only ?
- If yes, use atomic it will be a lot faster than critical :
  - atomic is like a mini critical section for a variable.

```
#pragma omp parallel for shared(sum)
for(i=0; i<N; i++) {
#pragma omp atomic
   sum += a[i] * b[i];
}
```

Solution 4: Changing the Scope Using the `private` Clause

```
 float x, y;
int i;
#pragma omp parallel for
for(i=0; i<N; i++) {
     x = a[i]; y = b[i];
     c[i] = x + y;
}
```

Before : global variables shared between threads.

```
float x, y;
int i;
#pragma omp parallel for private (x,y)
for(i=0; i<N; i++) {
     x = a[i]; y = b[i];
     c[i] = x + y;
}
```

After : local copies of global variables.
Nothing shared.

Efficient and safe.

Solution 4: Changing the Scope Using the `private` Clause

- The private clause reproduces the variable for each task
  - Variables are un-initialized;
  - C++ object is default constructed
  - Any value external to the parallel region is undefined

```
void* work(float* c, int N) {
  float x, y;
  int i;

     #pragma omp parallel for private(x,y)
     for(i=0; i<N; i++) {
       x = a[i]; y = b[i];
       c[i] = x + y;
     }
}
```