

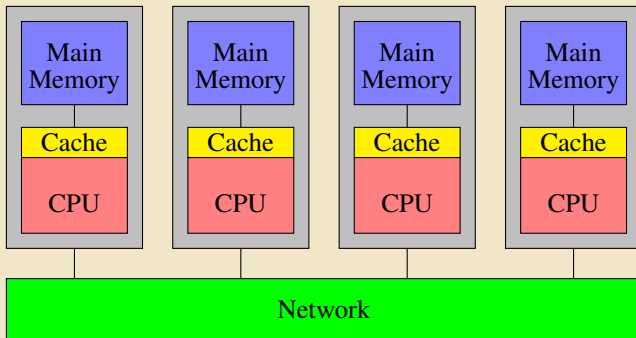
Introduction to OpenMP
2021 HPC Workshop: Parallel Programming

Alexander B. Pacheco

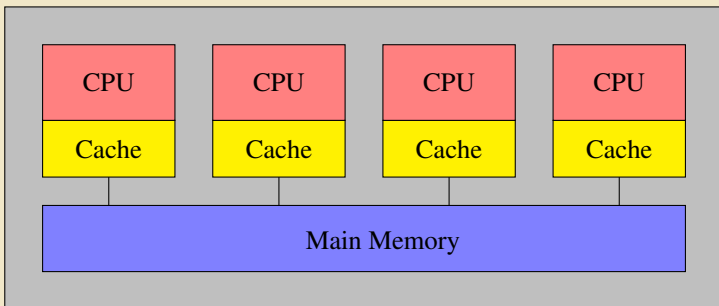
Research Computing

July 13 - 15, 2021

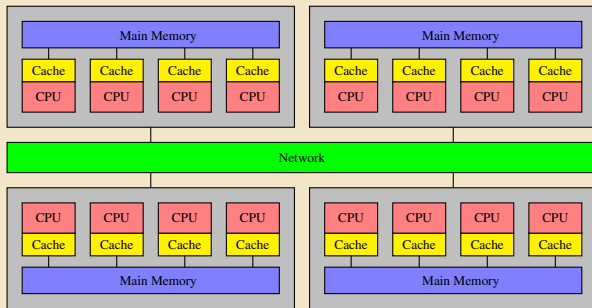
- Each process has its own address space
 - Data is local to each process
- Data sharing is achieved via explicit message passing
- Example
 - MPI



- All threads can access the global memory space.
- Data sharing achieved via writing to/reading from the same memory location
- Example
 - OpenMP
 - Pthreads



- The shared memory model is most commonly represented by Symmetric Multi-Processing (SMP) systems
 - Identical processors
 - Equal access time to memory
- Large shared memory systems are rare, clusters of SMP nodes are popular.



Shared Memory

- Pros
 - Global address space is user friendly
 - Data sharing is fast
- Cons
 - Lack of scalability
 - Data conflict issues

Distributed Memory

- Pros
 - Memory scalable with number of processors
 - Easier and cheaper to build
- Cons
 - Difficult load balancing
 - Data sharing is slow

Compiler Flags for Automatic Parallelization

- **GCC** -floop-parallelize-all
- **Intel** -parallel
- **NVHPC** -Mconcur

When to consider using OpenMP?

- The compiler may not be able to do the parallelization
 - 1 A loop is not parallelized
 - The data dependency analysis is not able to determine whether it is safe to parallelize or not
 - 2 The granularity is not high enough
 - The compiler lacks information to parallelize at the highest possible level

- OpenMP is an Application Program Interface (API) for thread based parallelism; Supports Fortran, C and C++
- Uses a fork-join execution model
- OpenMP structures are built with program directives, runtime libraries and environment variables
- OpenMP has been the industry standard for shared memory programming since 1997
 - Permanent members of the OpenMP Architecture Review Board: AMD, Cray, Fujitsu, HP, IBM, Intel, Microsoft, NEC, PGI, SGI, Sun
- OpenMP 4.0 was released in June 2014

- **Standardization**

- Provide a standard among a variety of shared memory architectures/platforms
- Jointly defined and endorsed by a group of major computer hardware and software vendors

- **Lean and Mean**

- Establish a simple and limited set of directives for programming shared memory machines.
- Significant parallelism can be implemented by using just 3 or 4 directives.

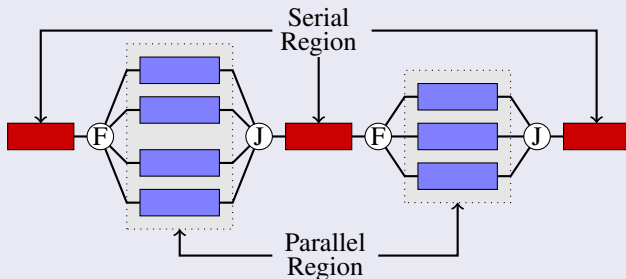
- **Ease to use**

- Serial programs can be parallelized by adding compiler directives
- Allows for incremental parallelization - a serial program evolves into a parallel program by parallelizing different sections incrementally

- **Portability**

- Standard among many shared memory platforms
- Implemented in major compiler suites

- Parallelism is achieved by generating multiple threads that run in parallel
 - A fork (F) is when a single thread is made into multiple, concurrently executing threads
 - A join (J) is when the concurrently executing threads synchronize back into a single thread
- OpenMP programs essentially consist of a series of forks and joins.



- Program directives
 - Syntax
 - C/C++: `#pragma omp <directive> [clause]`
 - Fortran: `!$omp <directive> [clause]`
 - Parallel regions
 - Parallel loops
 - Synchronization
 - Data Structure
 - ...
- Runtime library routines
- Environment variables

- Fortran: case insensitive
 - Add: `use omp_lib` or `include "omp_lib.h"`
 - Usage: **Sentinel directive [clauses]**
 - Fortran 77
 - **Sentinel** could be: `!$omp`, `*$omp`, `c$omp` and must begin in first column
 - Fortran 90/95/2003
 - **Sentinel**: `!$omp`
 - End of parallel region is signified by the end sentinel statement: **!\$omp end directive [clauses]**
- C/C++: case sensitive
 - Add `#include <omp.h>`
 - Usage: **#pragma omp directive [clauses] newline**

- Parallel Directive
 - **parallel**
- Worksharing Constructs
 - Fortran: **do, workshare**
 - C/C++: **for**
 - Fortran/C/C++: **sections**
- Synchronization
 - **master, single, ordered, flush, atomic**

- `if(scalar_expression)`
- `private(list)`, `shared(list)`
- `firstprivate(list)`, `lastprivate(list)`
- `reduction(operator:list)`
- `schedule(method[,chunk_size])`
- `nowait`
- `num_thread(num)`
- `threadprivate(list)`, `copyin(list)`
- `ordered`
- `more . . .`

- Number of Threads: `omp_{set,get}_num_threads`
- Thread ID: `omp_get_thread_num`
- Scheduling: `omp_{set,get}_dynamic`
- Nested Parallelism: `omp_in_parallel`
- Locking: `omp_{init,set,unset}_lock`
- Wallclock Timer: `omp_get_wtime`
- more . . .

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_STACKSIZE
- OMP_DYNAMIC
- OMP_NESTED
- OMP_WAIT_POLICY
- more . . .

- The **parallel** directive forms a team of threads for parallel execution.
- Each thread executes the block of code within the OpenMP Parallel region.

C

```
#include <stdio.h>

int main() {

#pragma omp parallel
{
    printf("Hello world\n");
}
}
```

Fortran

```
program hello

    implicit none

    !$omp parallel
    print *, 'Hello World'
    !$omp end parallel

end program hello
```


Compiling: compiler options code

- The OpenMP compile flag varies based on the compiler

GNU: `-fopenmp`

Intel: `-qopenmp`

NVHPC: `-mp`

```
[alp514.sol](1032): gfortran -fopenmp -o helloc hello.c
```

```
[alp514.sol](1033): ifort -qopenmp -o hellof hello.f90
```

Running : Need to specify number of openmp threads to run code on

```
[alp514.hawk-b624](1001): OMP_NUM_THREADS=4 ./helloc
```

```
Hello world
```

```
Hello world
```

```
Hello world
```

```
Hello world
```

```
[alp514.hawk-b624](1002): export OMP_NUM_THREADS=2
```

```
[alp514.hawk-b624](1003): ./hellof
```

```
Hello World
```

```
Hello World
```

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
 - Evaluation of the IF clause
 - Setting of the NUM_THREADS clause
 - Use of the `omp_set_num_threads()` library function
 - Setting of the OMP_NUM_THREADS environment variable
 - Implementation default
- Threads are numbered from 0 (master thread) to N-1

OpenMP include file

```
#include <omp.h> ←  
#include <stdio.h>  
int main () {  
    #pragma omp parallel  
    { ←  
        printf("Hello from thread %d out of %d  
            threads\n",omp_get_thread_num() ←  
            omp_get_num_threads()←);  
    } ←  
    return 0;  
}
```

Parallel region starts here

Runtime library functions

Parallel region ends here

```
Hello from thread 0 out of 4 threads  
Hello from thread 3 out of 4 threads  
Hello from thread 1 out of 4 threads  
Hello from thread 2 out of 4 threads
```

```
program hello

  implicit none
  integer :: omp_get_thread_num, omp_get_num_threads

  !$omp parallel
  print '(a,i3,a,i3,a)', 'Hello from thread',
    &  omp_get_thread_num(), &
    ' out of ' omp_get_num_threads(), ' threads'
  !$omp end parallel
end program hello
```

Parallel region starts here

Runtime library functions

Parallel region ends here

```
Hello from thread 0 out of 4 threads
Hello from thread 2 out of 4 threads
Hello from thread 1 out of 4 threads
Hello from thread 3 out of 4 threads
```

- Write a “hello world” program with OpenMP where
 - 1 If the thread id is odd, then print a message "Hello world from thread x, I'm odd!"
 - 2 If the thread id is even, then print a message "Hello world from thread x, I'm even!"
- Running the example interactively, say you compiled the code as `helloc`

```
srunc -p hawkgpu -n 6 -t 30 -A hpc2021_prog_083121 --reservation=lts_165 --pty /bin/bash
export OMP_NUM_THREADS=6
./helloc
```

- Run from the head node:

```
export OMP_NUM_THREADS=6
srunc -p hawkgpu -n 6 -t 30 -A hpc2021_prog_083121 --reservation=lts_165 ./helloc
```

- Alternate way to run from the head node:

```
srunc -p hawkgpu -n 1 -c 6 -t 30 -A hpc2021_prog_083121 --reservation=lts_165 --export=OMP_NUM_THREADS=6 ./helloc
```

C/C++

```
#include <omp.h>
#include <stdio.h>
int main() {
    int id;
    #pragma omp parallel private(id)
    {
        id = omp_get_thread_num();
        if (id%2==1)
            printf("Hello world from thread %d, I am odd\n", id
                );
        else
            printf("Hello world from thread %d, I am even\n",
                id);
    }
}
```

```
[alp514.hawk-b624](1003): icc -o hello -qopenmp hello.c
[alp514.hawk-b624](1004): icc -o helloc -qopenmp hello.c
[alp514.hawk-b624](1005): OMP_NUM_THREADS=4 ./helloc
Hello world from thread 0, I am even
Hello world from thread 2, I am even
Hello world from thread 1, I am odd
Hello world from thread 3, I am odd
```

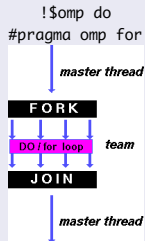
Fortran

```
program hello
    use omp_lib
    implicit none
    integer i
    !$omp parallel private(i)
    i = omp_get_thread_num()
    if (mod(i,2).eq.1) then
        print *, 'Hello from thread',i,', I am odd!'
    else
        print *, 'Hello from thread',i,', I am even!'
    endif
    !$omp end parallel
end program hello
```

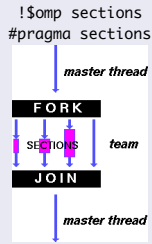
```
[alp514.hawk-b624](1006): ifort -qopenmp -o hellof hello.f90
[alp514.hawk-b624](1007): export OMP_NUM_THREADS=4
[alp514.hawk-b624](1008): ./helfof
Hello from thread      0 , I am even!
Hello from thread      2 , I am even!
Hello from thread      1 , I am odd!
Hello from thread      3 , I am odd!
```

- We need to share work among threads to achieve parallelism

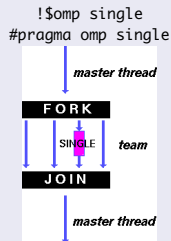
Loops



Sections



Single



- The parallel and work sharing directive can be combined as
 - `!$omp parallel do`
 - `#pragma omp parallel sections`

Example: Parallel Loops

C/C++

```
#include <omp.h>

int main() {
    int i = 0, n = 100, a[100];
    #pragma omp parallel for
    for (i = 0; i < n ; i++) {
        a[i] = (i+1) * (i+2) ;
    }
}
```

Fortran

```
program paralleldo

    implicit none
    integer :: i, n, a(100)

    i = 0
    n = 100
    !$omp parallel
    !$omp do
    do i = 1, n
        a(i) = i * (i+1)
    end do
    !$omp end do
    !$omp end parallel
end program paralleldo
```


- OpenMP provides different methods to divide iterations among threads, indicated by the schedule clause
 - Syntax: `schedule (<method>, [chunk size])`
- Methods include
 - **Static**: the default schedule; divide iterations into chunks according to size, then distribute chunks to each thread in a round-robin manner.
 - **Dynamic**: each thread grabs a chunk of iterations, then requests another chunk upon completion of the current one, until all iterations are executed.
 - **Guided**: similar to **Dynamic**; the only difference is that the chunk size starts large and shrinks to size eventually.

4 threads, 100 iterations

Schedule	Iterations mapped onto thread			
	0	1	2	3
Static	1-25	26-50	51-75	76-100
Static,20	1-20, 81-100	21-40	41-60	61-80
Dynamic	1,...	2,...	3,...	4,...
Dynamic,10	1 - 10, ...	11 - 20, ...	21 - 30, ...	31 - 40, ...

Schedule	When to Use
Static	Even and predictable workload per iteration; scheduling may be done at compilation time, least work at runtime.
Dynamic	Highly variable and unpredictable workload per iteration; most work at runtime
Guided	Special case of dynamic scheduling; compromise between load balancing and scheduling overhead at runtime

- Gives a different block to each thread

C/C++

```
#pragma omp parallel
{
#pragma omp sections
{
#pragma omp section
    some_calculation();
#pragma omp section
    some_more_calculation();
#pragma omp section
    yet_some_more_calculation();
}
}
```

Fortran

```
!$omp parallel
!$omp sections
!$omp section
call some_calculation
!$omp section
call some_more_calculation
!$omp section
call yet_some_more_calculation
!$omp end sections
!$omp end parallel
```

- `Shared(list)`
 - Specifies the variables that are shared among all threads
- `Private(list)`
 - Creates a local copy of the specified variables for each thread
 - the value is uninitialized!
- `Default(shared|private|none)`
 - Defines the default scope of variables
 - **C/C++ API does not have** `default(private)`
- Most variables are shared by default
 - A few exceptions: iteration variables; stack variables in subroutines; automatic variables within a statement block.

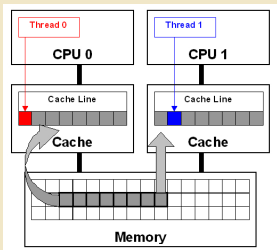
- SAXPY is a common operation in computations with vector processors included as part of the BLAS routines

$$y \leftarrow \alpha x + y$$

- SAXPY is a combination of scalar multiplication and vector addition
- Parallelize the code in the exercise/saxpy folder
- Calculate the speedup with respect to serial code.

Threads	C		Fortran	
	Timing (s)	Speed Up	Timing (s)	Speed Up
1	0.513491	1.00	0.504534	1.00
2	0.264634	1.94	0.300650	1.68
3	0.177902	2.89	0.234661	2.15
4	0.135248	3.80	0.150547	3.35
5	0.109646	4.68	0.120734	4.18
6	0.087660	5.86	0.100535	5.02
12	0.056454	9.10	0.050300	10.03
24	0.048442	10.60	0.026623	18.95
48	0.026348	19.49	0.025263	19.97

- Array elements that are in the same cache line can lead to false sharing.
 - The system handles cache coherence on a cache line basis, not on a byte or word basis.
 - Each update of a single element could invalidate the entire cache line.



```
!$omp parallel
myid = omp_get_thread_num()
nthreads = omp_get_numthreads()
do i = myid+1, n , nthreads
    a(i) = some_function(i)
end do
!$omp end parallel
```


- Multiple threads try to write to the same memory location at the same time.
 - Indeterministic results
- Inappropriate scope of variable can cause indeterministic results too.
- When having indeterministic results, set the number of threads to 1 to check
 - If problem persists: scope problem
 - If problem is solved: race condition

```
!$omp parallel do
do i = 1, n
  if (a(i) > max) then
    max = a(i)
  end if
end do
!$omp end parallel do
```

- “Stop sign” where every thread waits until all threads arrive.
- Purpose: protect access to shared data.
- Syntax:
 - Fortran: `!$omp barrier`
 - C/C++: `#pragma omp barrier`
- A barrier is implied at the end of every parallel region
 - Use the `nowait` clause to turn it off
- Synchronizations are costly so their usage should be minimized.

Critical

Only one thread at a time can enter a critical region

```
!$omp parallel do
do i = 1, n
  b = some_function(i)
  !$omp critical
  call some_routine(b,x)
end do
!$omp end parallel do
```

Atomic

Only one thread at a time can update a memory location

```
!$omp parallel do
do i = 1, n
  b = some_function(i)
  !$omp atomic
  x = x + b
end do
!$omp end parallel do
```

- Not initialized at the beginning of parallel region.
- After parallel region
 - Not defined in OpenMP 2.x
 - 0 in OpenMP 3.x

```
void wrong()
{
  int tmp = 0;
  #pragma omp for private( tmp )
  for (int j = 0; j < 100; ++j)
    tmp += j
  printf("%d\n", tmp)
}
```

tmp not initialized here


OpenMP 2.5: tmp undefined

OpenMP 3.0: tmp is 0


- Firstprivate
 - Initialize each private copy with the corresponding value from the master thread
- Lastprivate
 - Allows the value of a private variable to be passed to the shared variable outside the parallel region

```
void wrong()
{
  int tmp = 0;
  #pragma omp for firstprivate( tmp ) lastprivate(tmp)
  for (int j = 0; j < 100; ++j)
    tmp += j
  printf("%d\n", tmp)
}
```

tmp initialized as 0



The value of tmp is the value when j=99



Exercise: Calculate pi by Numerical Integration

- We know that

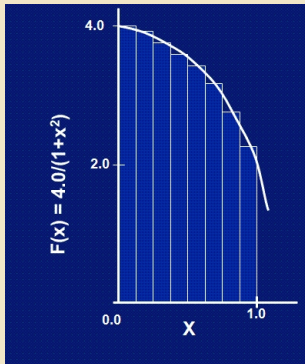
$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

- So numerically, we can approximate pi as the sum of a number of rectangles

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Meadows et al, A “hands-on” introduction to OpenMP, SC09

- Parallelize the code in the exercise/calc_pi folder



- The reduction clause allows accumulative operations on the value of variables.
- Syntax: reduction (operator:variable list)
- A private copy of each variable which appears in reduction is created as if the private clause is specified.
- Operators
 - 1 Arithmetic
 - 2 Bitwise
 - 3 Logical

Example: Reduction

C/C++

```
#include <omp.h>
int main() {
    int i, n = 100, sum , a[100], b[100];
    for (i = 0; i < n; i++) {
        a[i] = i;
        b[i] = 1;
    }
    sum = 0;
    #pragma omp parallel for reduction
    (+:sum)
    for (i = 0; i < n ; i++) {
        sum += a[i] * b[i];
    }
}
```

Fortran

```
program reduction

    implicit none
    integer :: i, n, sum , a(100), b(100)

    n = 100 ; b = 1; sum = 0
    do i = 1 , n
        a(i) = i
    end do
    !$omp parallel do reduction(+:sum)
    do i = 1, n
        sum = sum + a(i) * b(i)
    end do
    !$omp end parallel do

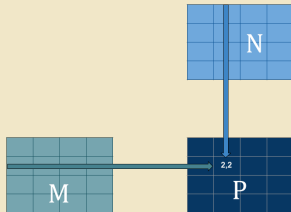
end program reduction
```


Exercise 3: pi calculation with reduction

- Redo exercise 2 with reduction

Exercise: Matrix Multiplication

- Most Computational code involve matrix operations such as matrix multiplication.
- Consider a matrix **C** of two matrices **A** and **B**:
Element i,j of **C** is the dot product of the i^{th} row of **A** and j^{th} column of **B**



- Parallelize the code in the exercise/matmul folder
- Calculate the speedup with respect to serial code.

- **Modify/query the number of threads**
 - `omp_set_num_threads()`, `omp_get_num_threads()`,
`omp_get_thread_num()`, `omp_get_max_threads()`
- **Query the number of processors**
 - `omp_num_procs()`
- **Query whether or not you are in an active parallel region**
 - `omp_in_parallel()`
- **Control the behavior of dynamic threads**
 - `omp_set_dynamic()`, `omp_get_dynamic()`

- `OMP_NUM_THREADS`: set default number of threads to use.
- `OMP_SCHEDULE`: control how iterations are scheduled for parallel loops.

- <https://www.openmp.org/>
- OpenMP API 5.1 Specification
- OpenMP 4.5 Reference Guide Fortran
- OpenMP 4.5 Reference Guide C/C++
- Using OpenMP Portable Shared Memory Parallel Programming - Barbara Chapman, Gabriele Jost and Ruud van der Pas
- <https://www.openmp.org/resources/tutorials-articles/>
- <https://www.openmp.org/resources/openmp-books/>
- <http://en.wikipedia.org/wiki/OpenMP>
- <https://hpc.llnl.gov/tuts/openMP>
- <https://www.hpc-training.org/xsede/moodle>