OpenMP – Shared Memory Parallelization

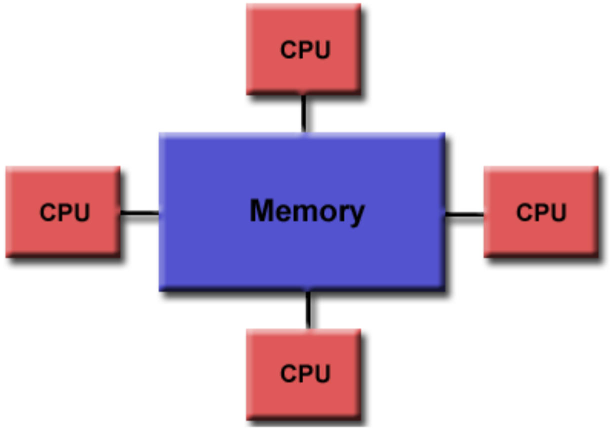
Ramakrishnan Kannan

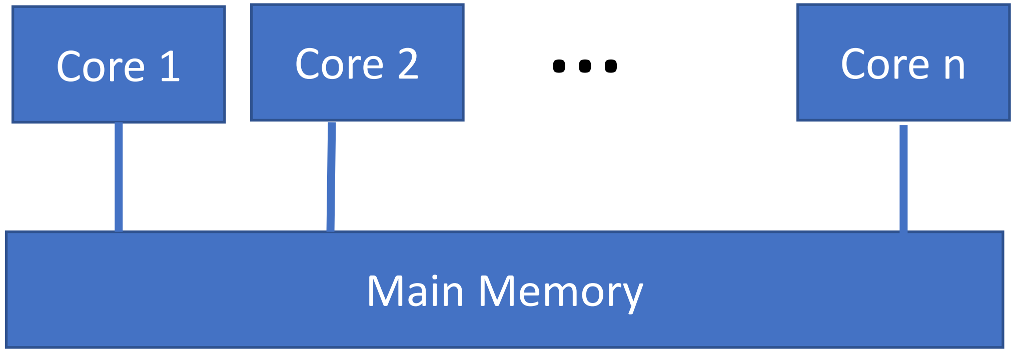
Shruthi Shivakumar

Motivated out of Alexander B. Pacheco slides, SC’22 Parallel Computing 101 tutorial, OLCF Tutorial

Shared Memory - Introduction

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

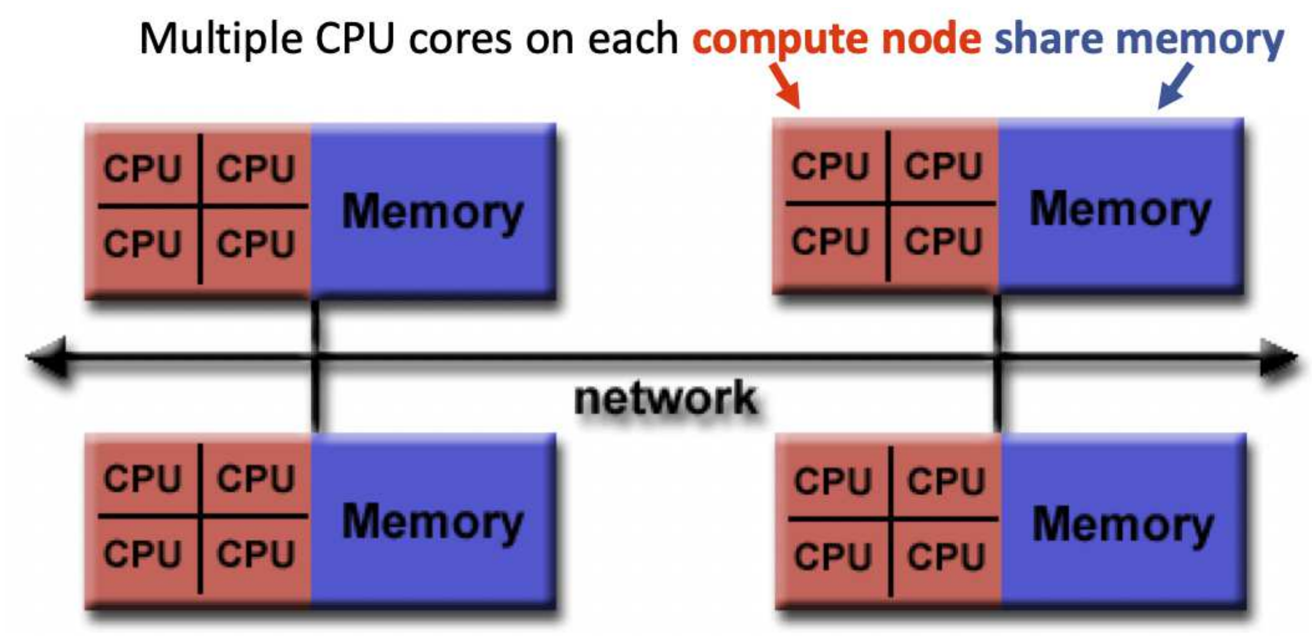




Example systems

1. Laptops
2. Cellphones
3. Extreme threaded machines

Distributed systems with Shared memory



- The shared memory model is most commonly represented by Symmetric Multi-Processing (SMP) systems
    - Identical processors
    - Equal access time to memory
- Non-local data can be sent across the network to other CPUs
- Large shared memory systems are rare, clusters of SMP nodes are popular

Shared vs Distributed Memory

- 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

Shared Memory Parallelization

- Shared memory (SM) machines have always been important in high performance computing.
    - All processors can directly access all the memory in the system (though access time can be different).
    - This greatly reduces communication latency.
    - However, synchronization errors can be quite subtle.

Parallelization Techniques: OpenMP

- First Introduced in 1997
- Latest OpenMP specification is 5.2 (Nov 2021)
- OpenMP is an Application Program Interface (API): directs multi-threaded shared memory parallelism
- Explicit Programming Model – Compiler interprets parallel constructs
- Based on a combination of compiler directives, library routines and environment variables.
- OpenMP uses the fork-join model of parallel execution.
- https://www.openmp.org

Goals of OpenMP

- Standardization – different architectures, compilers and hardware platforms
- Lean – Limited set of compiler directives – 4-6
- Ease of Use
- Portability – across different programming languages Fortran/C++

Three building blocks

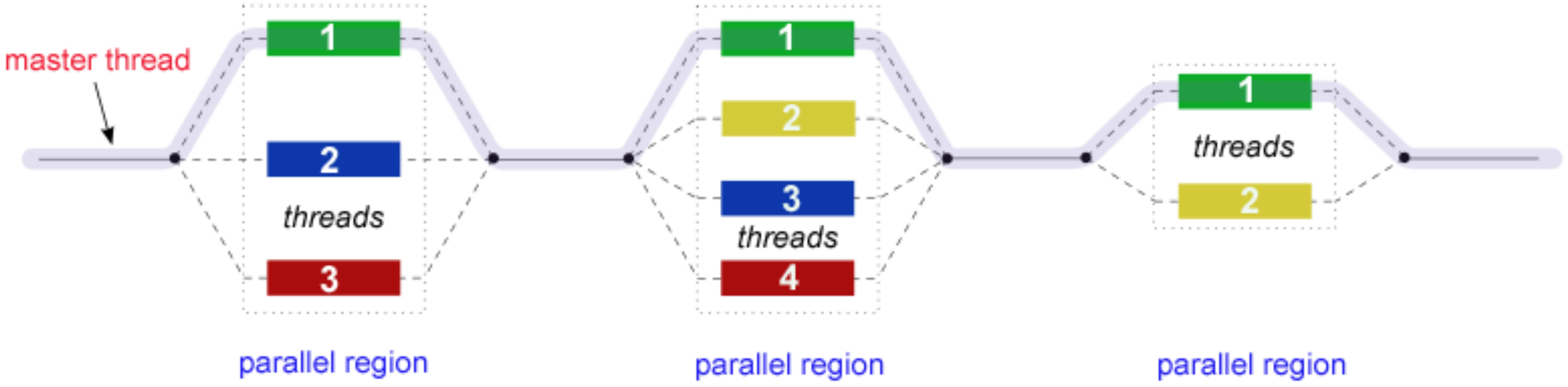
- Compiler Directives
  - private(list), shared(list)
  - firstprivate(list), lastprivate(list)
  - reduction(operator:list)
  - schedule(method[,chunk_size])
  - nowait
  - if(scalar_expression)
  - num_thread(num)
  - threadprivate(list), copyin(list)
  - Ordered
- Runtime Libraries/APIs
  - omp_set/get_num_threads, omp_get_thread_num, omp_{set,get}_dynamic, omp_in_parallel, omp_get_wtime
- Environment variables
  - OMP_NUM_THREADS, OMP_SCHEDULE, OMP_STACKSIZE, OMP_DYNAMIC, OMP_NESTED, OMP_WAIT_POLICY

Fork-Join Model

- OpenMP programs begin as a single process: the master thread.

- The master thread executes sequentially until the first parallel region construct is encountered.

- FORK: the master thread then creates a team of parallel threads.
  - The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
- The number of parallel regions and the threads that comprise them are arbitrary.



OpenMP Notation and Parallel Loops

- C/C++: case sensitive
  - Add #include <omp.h>
  - Usage: #pragma omp directive [clauses] newline
  - Use the flag –fopenmp during compilation

Parallel Directive

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

```
#include <stdio.h>
#include <omp.h>
int main()
{
    #pragma omp parallel
    {
        printf("Hello world\n");
    }
}
```

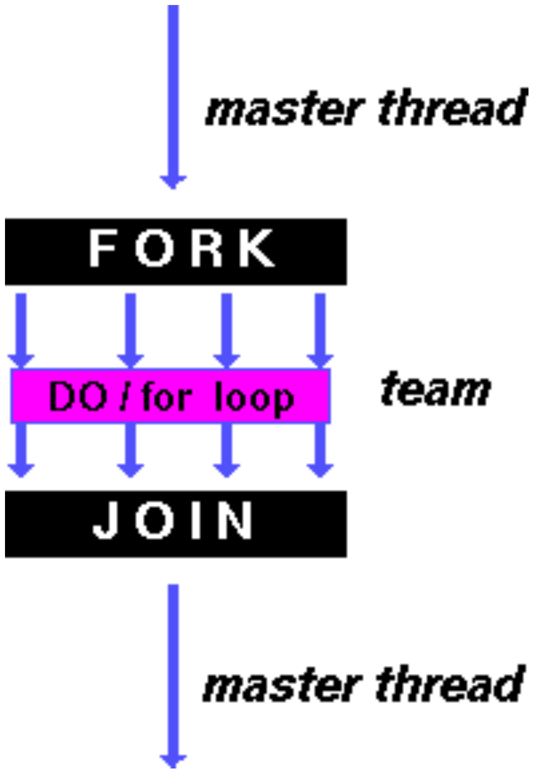
Helloworld Example

```
#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;
}
```

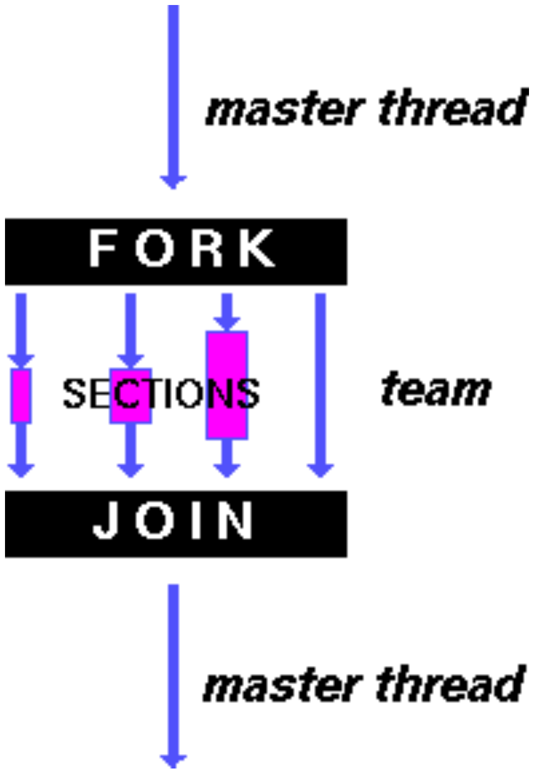
Private variable

```
#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);
    }
}
```

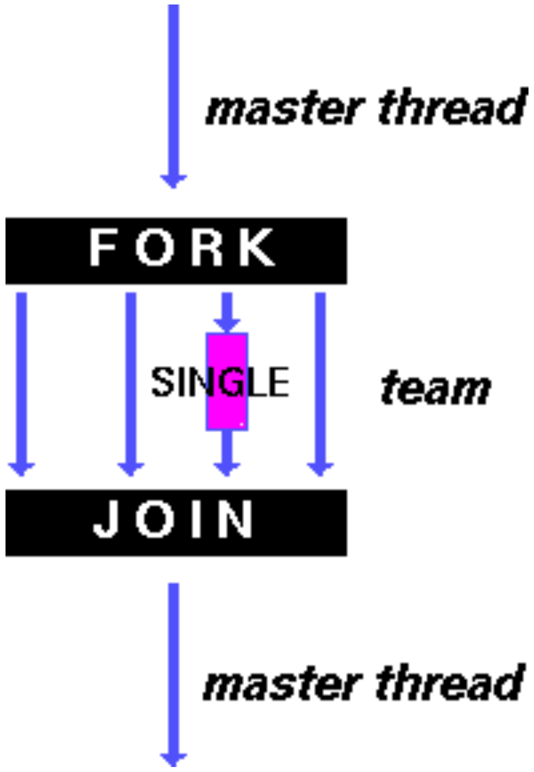
Workshare vs Sections vs Single



DO / for shares iterations of a loop across the team. Represents a type of “data parallelism”.



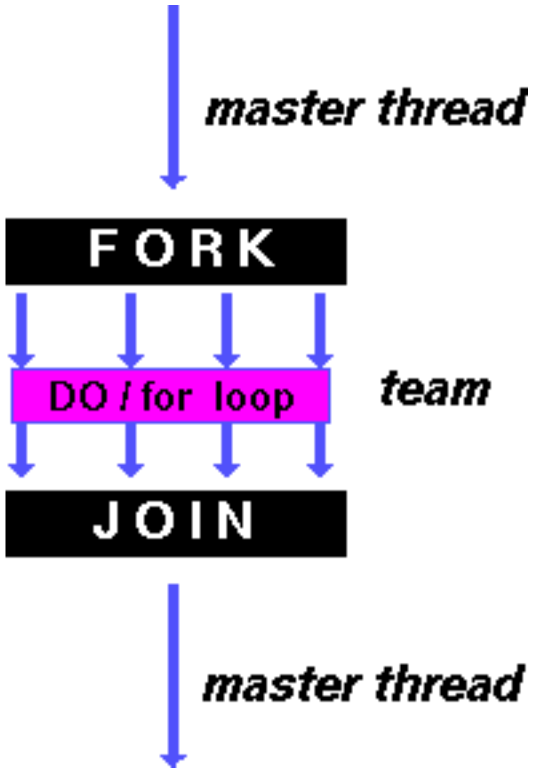
SECTIONS breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of “functional parallelism”.



SINGLE serializes a section of code

Parallel For

```
#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) ;
        }
}
```



DO / for shares iterations of a loop across the team. Represents a type of “data parallelism”.

Saxpy example

- Linear combination of two float arrays
- y = ax + y, where x and y are arrays of same length, and a is a scalar.

$$
\begin{array}{ccccc}
a \cdot & \boxed{x_1} & + & \boxed{y_1} & = & \boxed{a \cdot x_1 + y_1} \\
a \cdot & \boxed{x_2} & + & \boxed{y_2} & = & \boxed{a \cdot x_2 + y_2} \\
 & \vdots & & \vdots & & \vdots \\
a \cdot & \boxed{x_d} & + & \boxed{y_d} & = & \boxed{a \cdot x_d + y_d}
\end{array}
$$

```
for (i = 0; i < n; i++)
{
        y[i] = a*x[i] + y[i];
}
```

Sequential Code

```
#pragma omp parallel for
for (i = 0; i < n; i++)
{
y[i] = a*x[i] + y[i];
}
```

openmp Code

Load Balancing

- 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 interations 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

Load Balancing - II

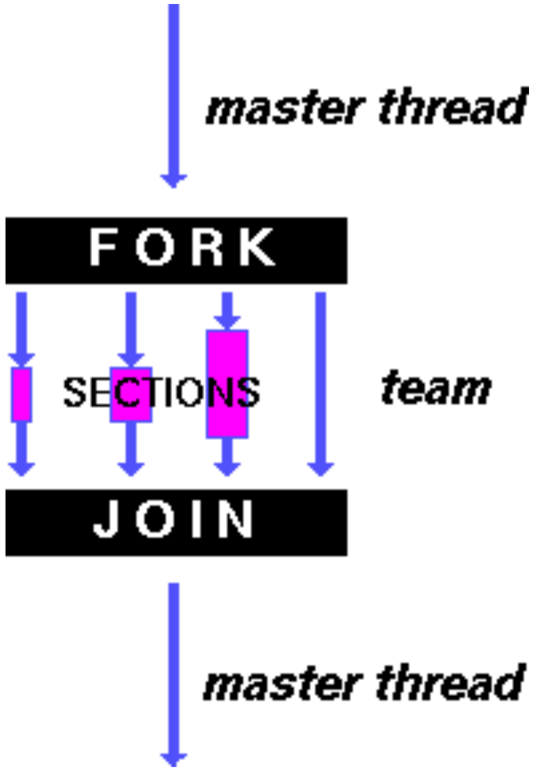
4 threads, 100 iterations

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`	$1, \cdots$	$2, \cdots$	$3, \cdots$	$4, \cdots$
`Dynamic,10`	$1-10, \cdots$	$11-20, \cdots$	$21-30, \cdots$	$31-40, \cdots$

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

Worksharing

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



SECTIONS breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of “functional parallelism”.

Variables scope

- 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 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.

Synchronization : Critical and Atomic

- Critical: Only one thread at a time can enter a critical region

```
#include
main()
{
int x;
x = 0;
#pragma omp parallel shared(x)
  {
  #pragma omp critical
  x = x + 1;
  }  /* end of parallel section */
}
```

- Atomic: Only one thread at a time can update a memory location

```
#include
main()
{
int x;
x = 0;
#pragma omp parallel shared(x)
  {
  #pragma omp atomic
  x = x + 1;
  }  /* end of parallel section */
}
```

Special Cases – First and last private

- 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

tmp initialized as 0

```
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 )
}
```

The value of tmp is the value when j=99

Reduction

- 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
  - Arithmetic
  - Bitwise
  - Logical

```
int main()
{
   int i, n;
   n = 10000;
   float a[n], b[n];
   double result, sequential_result;
   /* Some initializations */
   result = 0.0;
   for (i = 0; i < n; i++)
   {
      a[i] = i * 1.0;
      b[i] = i * 2.0;
   }
#pragma omp parallel for default(shared) private(i) \
   schedule(static) reduction(+ : result)
   for (i = 0; i < n; i++)
      result = result + (a[i] * b[i]);
   printf("Final result= %f\n", result);
   return 0;
}
```