Indian Institute of Information Technology Sri city

1

High Performance Computing

Dr.Bheemappa H
IIIT, Sricity
bheemappa.h@iiit.in

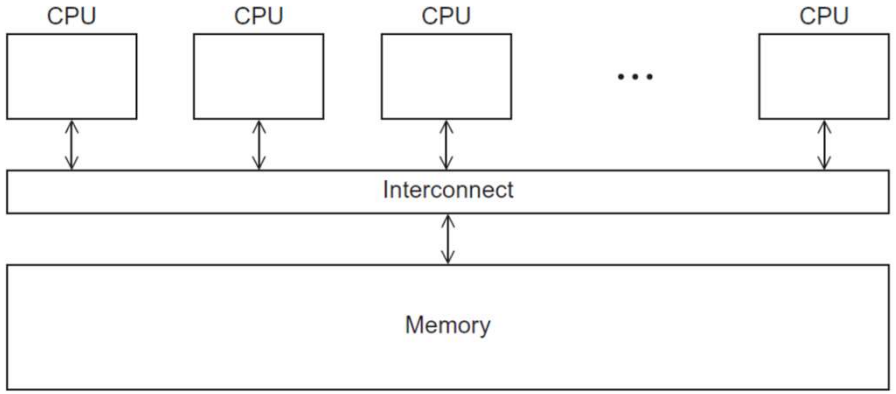
M4: Shared Memory Programming with OpenMP

- Shared Memory Programming with OpenMP: A Standard for Directive Based Parallel Programming. (6-Hrs)
  - Writing programs that use OpenMP.
  - Using OpenMP to parallelize many serial for loops with only small changes to the source code.
  - Task parallelism.
  - Explicit thread synchronization.
  - Standard problems in shared-memory programming

Writing programs that use OpenMP.

- An API for shared-memory parallel programming.
  - MP = multiprocessing
  - Designed for systems in which each thread or process can potentially have access to all available memory.
- System is viewed as a collection of cores or CPU's, all of which have access to main memory.

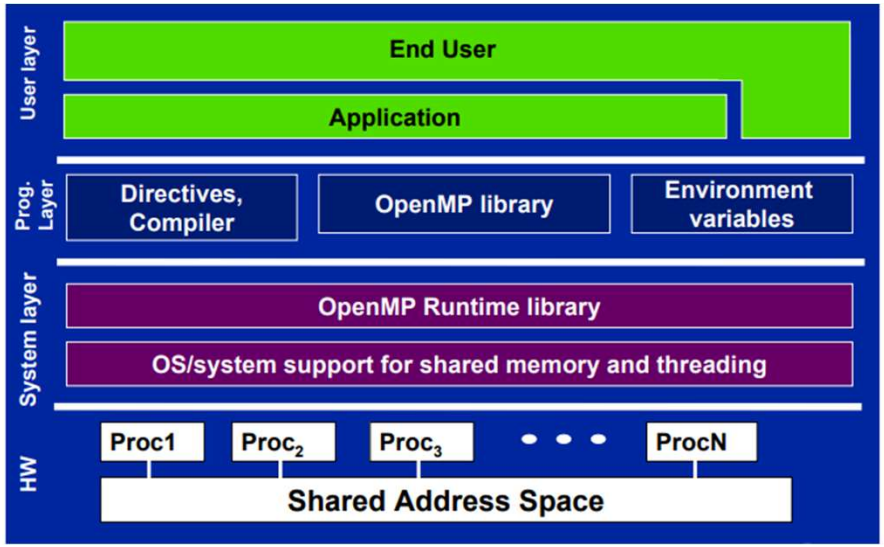
•Shared Memory Systems
Conceptual model
Real Hardware model



- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
  - Unintended sharing of data causes race conditions
  - To control race conditions: – Use synchronization to protect data conflicts.

OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.
  - #pragma omp construct [clause [clause]…]
- Function prototypes and types in the file: #include<omp.h>.
- constructs apply to a “structured block”




```
#pragma omp parallel num_threads(4)
```

Request a certain number of threads

OpenMp pragmas

pragma omp parallel

Most basic parallel directive

pragma omp parallel num_threads (thread_count)

It allows the programmer to specify the number of threads that should execute the following block

Implicit barrier : thread that has completed the block of code will wait for all the other threads in the team to complete the block

Pragmas

- **Pragmas**
  - Special preprocessor instructions.
  - Typically added to a system to allow behaviors that aren't part of the basic C specification.
  - Compilers that don't support the pragmas ignore them

```
#pragma omp parallel
{
  //Parallel region code
}
```

```
#pragma
```

```
gcc −g −Wall −fopenmp −o omp  hello omp  hello . C
/ omp  hello 4
```

```
export OMP_NUM_THREADS=5
```

Exercise 1:


```c
#include<stdio.h>


int main( int ac, char **av)
{
                //a and shared between all thread (race cond)
                #pragma omp parallel
                {
                int a;
                int b;
                a=omp_get_num_threads();
                b=omp_get_thread_num();
                printf("Total No of Threads %d",a);
                printf("Hello World!!! with thread ID %d \n ",b);
                }

                return 0;

}
```

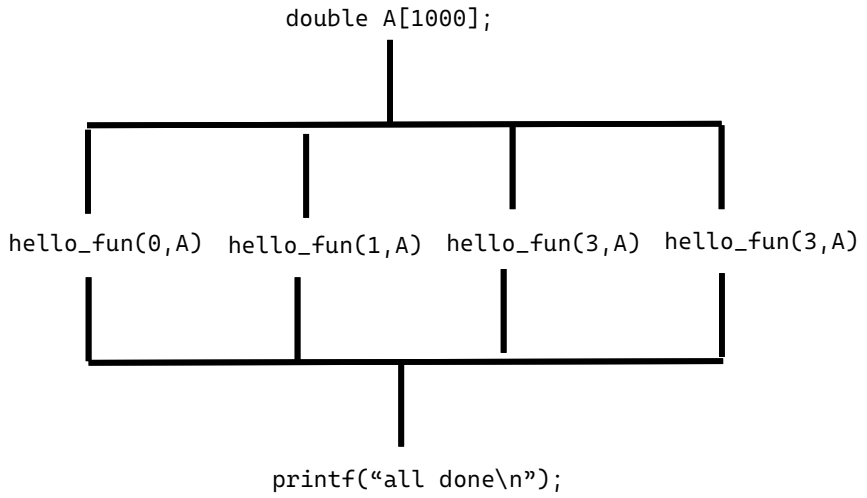
original thread and the new threads — is called a team, the original thread is called the master, and the additional threads ar called slaves.

```
Total No of Threads 8
Total No of Threads 8
Total No of Threads 8
Hello World!!! with thread ID 1
 Total No of Threads 8
Hello World!!! with thread ID 7
 Hello World!!! with thread ID 0
 Hello World!!! with thread ID 4
 Total No of Threads 8
Total No of Threads 8
Hello World!!! with thread ID 6
 Hello World!!! with thread ID 2
 Total No of Threads 8
Hello World!!! with thread ID 3
 Total No of Threads 8
Hello World!!! with thread ID 5
```

```
Total No of Threads 8
Total No of Threads 8
Hello World!!! with thread ID 5
 Total No of Threads 8
Total No of Threads 8
Hello World!!! with thread ID 7
 Hello World!!! with thread ID 4
 Total No of Threads 8
Total No of Threads 8
Hello World!!! with thread ID 3
 Hello World!!! with thread ID 2
 Hello World!!! with thread ID 0
 Total No of Threads 8
Hello World!!! with thread ID 1
 Total No of Threads 8
Hello World!!! with thread ID 6
```

Thread Creation: Parallel Regions example


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



1. passing the number of threads at the command line

Parallel construct


```
#pragma omp parallel [clause [, clause] ...]
  structured-block
```

The following clauses apply:

- `if`
- `num_threads`
- `shared, private, firstprivate, default`
- `reduction`
- `copyin`
- `proc_bind`

Parallel construct

- num_threads clause
  - Specifies how many threads should execute the region
  - Syntax: “num_threads (*scalar-logical-expression*)”
- if clause
  - Conditional parallel execution
  - Avoid parallelization overhead if little work to be parallelized
  - Syntax: “if (scalar-logical-expression)”
  - If the logical expression evaluates to true: execute in parallel

Example:

```
int main( ... )
{
    [...]
    #pragma omp parallel num_threads (nths)
    {
        [...]
    }
    [...]
}
```

```
int main( ... )
{
    [...]
    #pragma omp parallel if (n > 1000)
    {
        [...]
    }
    [...]
}
```

Parallel construct

- **Shared clause :**
  - **Syntax: shared (item-list).**
  - Each thread can freely read and modify its value
- **Private Clause**
  - Specifies data that will be replicated so that each thread has a local copy
    - **private clause (*item-list*)**

firstprivate and **lastprivate** are just special cases of private. The first one results in bringing in values from the outside context into the parallel region while the second one transfers values from the parallel region to the outside context

Parallel construct

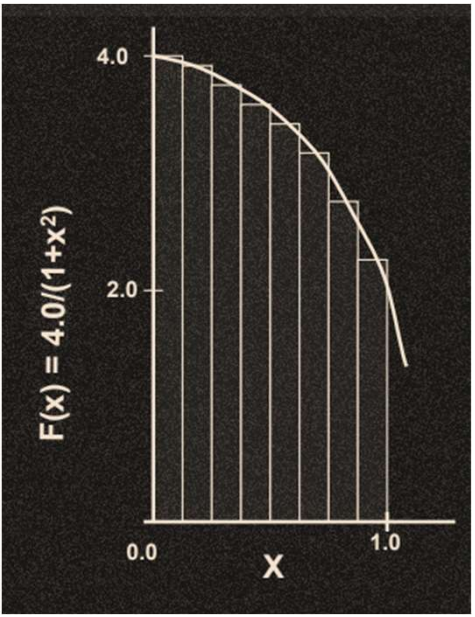
- Shared-memory programming model
- Variables are shared by default.
  - **Shared**:
    - All variables visible upon entry of the construct
    - Static variables
  - Private:
    - Variables declared within a parallel region
    - (Stack) variables in functions called from within a parallel region
- General rules for data-sharing clauses
  - Clauses default, **private, shared, firstprivate** allow changing the default behavior
  - Syntax : keyword and a comma-separated list. For instance: private(a,b).

```
int N = 10;
int main( void )
{
                double array[N];
                #pragma omp parallel
                {
                int i, myid;
                double thread_array[N];
                for ( i = 0; i < N; i++ )
                thread_array[i] = myid * array[i];
                function( thread_array );
                }
}
double function( double arg )
{
                static int cnt;
                double local_array[N];
                [...]
}
```

Numerical Integration Example

Mathematically, we know that:

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



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

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;
}
```

Homework

1. Create a parallel version of the pi program using a parallel construct. ???

```
#include <stdio.h>
#include <stdlib.h>
#define NUM_STEPS 10000
int main( void )
{
int i;
double sum = 0.0, pi, x_i;
double step = 1.0/NUM_STEPS;
for ( i = 0; i < NUM_STEPS; i++ ) {
x_i = (i + 0.5) * step;
sum = sum + 4.0 / (1.0 + x_i * x_i);
}
pi = sum * step;
printf("Pi: %.15e\n", pi);
return 0;
}
```

Hints:

#pragma omp parallel

num_threads(...)

omp_set_num_threads(...)

omp_get_num_threads(...)

omp_get_thread_num(...)

Challenges:

split iterations of the loop among threads

create an accumulator for each thread to hold partial sums, which can later be combined to generate the global sum

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

int main (void)
{
    int i = 10;

    #pragma omp parallel private(i)
    {
        printf("thread %d: i = %d\n", omp_get_thread_num(), i);
        i = 1000 + omp_get_thread_num();
    }

    printf("i = %d\n", i);

    return 0;
}
```

```
thread 0: i = 0
thread 3: i = 32717
thread 1: i = 32717
thread 2: i = 1
i = 10

(another run of the same program)

thread 2: i = 1
thread 1: i = 1
thread 0: i = 0
thread 3: i = 32657
i = 10
```

When I is made firstprivate

```
thread 2: i = 10
thread 0: i = 10
thread 3: i = 10
thread 1: i = 10
i = 10
```

Working with loops

- Find compute intensive loops
- Make the loop iterations independent
- Place the appropriate OpenMP directive and test

Syntax:

```
#pragma omp for [clause [, clause] ...]
   for-loop
```

Example:

```
#pragma omp parallel
{
  #pragma omp for
  for (i = 0; i < n; i++)
    [...]
}
```

Sequential:

```
int i;
for (i = 0; i < N; i++)
  z[i] = alpha * x[i] + y[i];
```

Parallel region:

```
#pragma omp parallel
{
  int i, id, nths, istart, iend;
  id = omp_get_thread_num();
  nths = omp_get_num_threads();
  istart = id * N / nths;
  iend = (id+1) * N / nths;
  if (id == nths-1) iend = N;
  for (i = istart; i < iend; i++)
    z[i] = alpha * x[i] + y[i];
}
```

Avoid False Sharing

- In the presence of multiple threads that share data, there are a number of sharing effects that may affect performance. One such sharing pattern is *false sharing*.
- False sharing is likely to significantly impact performance under the following conditions:
- 1. Shared data is modified by multiple threads.
- 2. The access pattern is such that multiple threads modify the same cache line(s).
- 3. These modifications occur in rapid succession