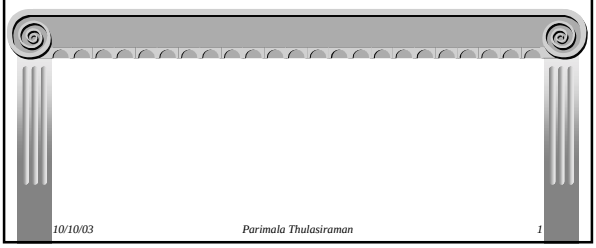
OpenMP



What is it

- A Parallel Programming model for shared and distributed shared memory multiprocessors.
- SGI
- Standard for parallelizing applications
- www.openmp.org

OpenMP

- Compiler directives to describe parallelism in source code + library functions
- C/C++ or Fortran
- Why compiler directives
  - Easier to write portable code
    - Ignored by compiler not supporting OpenMP
  - Compiler based optimizations
- !$omp <directive> & (fortran)
- #pragma omp … (C)

Fork/join Model

- The standard view of parallelism – *fork/join* parallelism
- When the program begins execution, it has one active master thread.
  - It executes the sequential portion of the algorithm.
  - Where parallelism is required, more threads are forked (created) by the master thread
  - At the end of the parallel code, threads die or suspended.
  - Flow goes back to the master thread--Join

Key Difference –SMP vs. Message passing Model

- In Message passing, all processes remain active throughout the execution of the program—Heavy Weight threads
- In SMP, threads change dynamically—Light Weight threads
  - Supports incremental parallelization: transforms sequential code into parallel program one block at a time.

Parallel Constructs –CONTROLLING PARALLELISM

1) Divide work among parallel threads – (loop level parallelism)– iterations of a loop are divided among the threads
2) A directive to create multiple threads that execute concurrently with each other. – Parallel Region

Loop Level Parallelism

What is the syntax?

#pragma omp parallel for [clause[..]clause[..]

for index = first; test_expr; increment_expr{

Body of loop

}

Parallel ***for*** Loop

- for (I = first; I < size; I+=prime) marked[I] = 1;
- There is no indication of any dependencies.
- How to convert into parallel?
  - In OpenMP, simply indicate to the compiler that the iterations of a *for* loop may be executed in parallel.
  - The compiler takes care of generating the code that forks/joins threads and
  - **Schedules the iterations: allocates iterations to threads**

parallel for **Pragma**

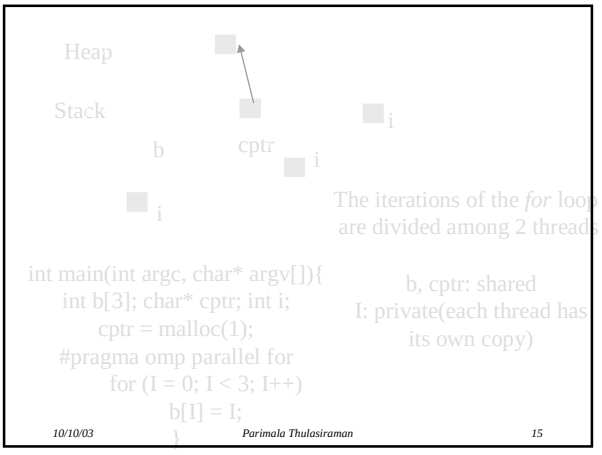
- Pragma: Compiler directive in C or C++
  - Pragmatic information
  - A way to communicate information to the compiler
  - Helps the compiler to optimize the program
  - Begins with the character #:
    - # pragma omp <rest of pragma>
    - # pragma omp parallel for
      - – Putting this right before the for loop lets the compiler to parallelize the loop
    - #pragma omp parallel for
      - – for (I = first; I < size; I+=prime) marked[I] =1;
  - for loop must not exit prematurely (break, return, exit)

What is really happening?

- Every thread has its own *execution context*:an address space containing all of the variables a thread may access.
- It may contain
  - *shared variable*
    - Same address space in the execution context of the thread
  - or *private variable* access.
    - Different address space in the execution context of every thread.—it can access its own variable but not the others.

parallel for

- Variable by default are shared
- However, *loop index is private*.



Number of threads

- How does the run time system know how many threads to create?
- OMP_NUM_THREADS
- On the command line before running the program, you can set it as :
  - setenv OMP_NUM_THREADS 4
    - 4 threads are created
  - Or you can set the number of threads inside the program: omp_set_num_threads(t)

Number of processors

- If you want to set the number of threads = number of processors:
  - int omp_get_num_procs(void)

```
int t;
t = omp_get_num_procs();
omp_set_num_threads(t); /* this maybe called at multiple points in the program—you can tailor the level of parallelism*/
```


Declaring Private Variables

- private <variables>
- Directive tells the compiler to allocate a private copy of the variable to each thread executing the block of code the pragma precedes.


```
#pragma omp parallel for private(j,x,y)
for (i = 0; i< m; i++){
    for (j = 1; j< n; j++) {
        x = i/m;
        y= j/n;}
        }
```

Each thread works through *n* values of *j* for Each iteration of the *i* loop.

Each *i* loop executes the *j* loop sequentially

The private copies of j are accessible only inside the loop.

In this case

- Even if j had a previously assigned value before entering the for loop, none of the threads can access the value.
- Whatever value j was assigned to during the parallel execution, when it returns, shared j is not affected (if there is a shared j).
- Default: the value of the private variable is undefined when the parallel construct is entered and is also undefined when it is exited.

firstprivate clause

- Sometimes we want a private variable to inherit the value of the shared variable.

```
x[0] = complex_function();/* initialized outside*/
for (i = 0; i < n; i++){
    for (j = 1; j < 4; j++)
        x[j] = g(i, x[j-1]);
    answer[i] = x[1]- x[3];
}
```

Assume g has no side effects.We Want to execute the outer loop in Parallel—make x private

We want each thread's private value of x to inherit x[0] assigned by the master thread

firstprivate(<variable list>)

Correct way

```
x[0] = complex_function();
#pragma omp parallel for private(j) firstprivate(x)
for (i = 0; i < n; i++){
    for (j = 1; j < 4; j++)
        x[j] = g(i, x[j-1]);
    answer[i] = x[1]- x[3];
}
```

- Note: variable in firstprivate list are initialized once per thread (NOT once per iteration)
- If a thread executes multiple iterations of the parallel loop and modifies the values of one of these variables in the iteration, subsequent iterations referencing the variable will get the modified value not the original.

lastprivate clause

- *lastprivate* write back to the master's copy the value contained in the private copy belonging to thread that executed the last iteration of the loop

```
c1=2; c2 = 0;
#pragma omp for firstprivate(c1) lastprivate(c2)
        for (int i= 0; i < 10; i++) {
                c2 = c1+i}//after loop c2=11 ; c1 = 2
```


At the last iteration x[3] gets assigned

```
for (i = 0; i < n; i++) {
    x[0] = 1.0;
    for (j = 1; j < 4; j++)
        x[j] =  x[j-1]*(i+1);
    sum_of_powers[i] =
      x[0]+x[1]+x[2]+x[3];
}
n_cubed = x[3];
```

```
# pragma omp parallel for
    private(j) lastprivate(x)
```

Critical Sections

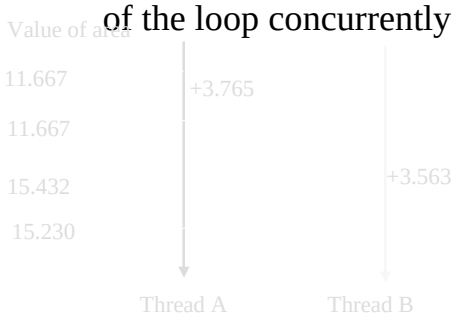
```
double area, pi, x;
int i,n;
area = 0.0;
for (i = 0; i <n; i++){
    x = (i+0.5)/n;
    area +=4.0/(1.0+x*x);
}
pi = area/n;
```

- Each iteration is not independent of each other
- Every iteration reads and updates *area*.

Critical Sections

```
#pragma omp parallel for private(x)
double area, pi, x;
int i,n;
area = 0.0;
for (i = 0; i <n; i++){
    x = (i+0.5)/n;
    area +=4.0/(1.0+x*x);/* Race condition: nondeterministic
      behaviour when multiple threads access a shared
      variable*/
}
pi = area/n;
```


2 threads executing the iterations of the loop concurrently



So

- The assignment statement that reads and updates area must be put in a **critical section.**

  A portion of the code only one thread at a time may execute.

critical pragma

- #pragma omp critical;
- This pragma directs the compiler to enforce mutual exclusion among the threads trying to execute the block of code.

Critical Sections

```
#pragma omp parallel for private(x)
double area, pi, x;
int i,n;
area = 0.0;
for (i = 0; i <n; i++){
    x = (i+0.5)/n;
    #pragma omp critical
    area +=4.0/(1.0+x*x);/* Race condition: nondeterministic
       behaviour when multiple threads access a shared
       variable*/
}
pi = area/n;
```


LLP

- Easy to express parallelism
- Limitations
  - Non-loop applications not parallelizable this way
  - Fork-join paradigm: all threads must wait for the slowest thread to finish.
  - Negative impact on performance and scalability