# OpenMP : A Portable Solution for Threading

→ The OpenMP standard was formulated in 1997 as an API for writing portable, multithreaded applications.

→ It started as a fortran-based standard but later grew to include C and C++.

→ The current version is OpenMP version 2.5.

→ The OpenMP programming model provides a platform-independent set of compiler pragmas, directives, function calls, and environment variables that explicity instruct compiler how and where to use parallelism in the application.

→ The power and simplicity of OpenMP can be demonstrated by looking an example —

```
#pragma omp parallel for
        for (i=0; i<numPixels; i++)
        {
            pGrayScaleBitmap[i]=(unsigned BYTE)
                (pRGBBitmap[i].red * 0.299 +
                 pRGBBitmap[i].green * 0.587 +
                 pRGBBitmap[i].blue * 0.114);
        }
```

→ Above loop converts each 32-bit RGB pixel in an array into an 8-bit grayscale pixel.

→ The one pragma, which is inserted before the loop, is all that is needed for parallel execution under OpenMP.

→ The example uses Work-sharing
↓ the general term that OpenMP uses to describe distributing work across threads.

→ When work-sharing is used with the for construct, the iterations of the loop are distributed among multiple threads.

5 restrictions on which loops can be threaded by OpenMP version 2.5. →

1. The loop variable must of type signed integer. Unsigned integer will not work (this restriction going to be removed in version 3.0)
2. Comparision operation must be in the form loop-variable <, <=, >, or >= loop-invariant integer.
3. If comparison operation is < or <= / > or >=, the loop variable must ~~be~~ increment/decrement on every iteration.
4. The loop must be a single entry and single exit loop, meaning no jumps from the inside of the loop to outside or outside to the inside.
5. The third expression or inc. portion of the loop must be either integer addition or integer substration and by a loop-invariant value.

## ~~Challenges~~ in Threading a Loop

1. Loop-carried Dependence → Even if loop meets all five loop criteria and the compiler threaded the loop, it may still not work correctly,

given the existence of date dependencies that the compiler ignores due the presence of OpenMP ~~prag~~ pragmas.

→ The theory of date dependence imposes 2 requirements that must be met for a statement $S_2$ and to be data dependent on statement $S_1$.

(1) There must exist possible execution path such that $S_1$ and $S_2$ both reference the same memory location L.

(2) The execution of $S_1$ that references L occurs before the execution of $S_2$ that reference L.

→ In order for $S_2$ to depend upon $S_1$, it is necessary for some execution of $S_1$ to write to a memory locatin L that is later read by an execution of $S_2$. This is also called flow dependence.

→ Other dependencies exist when 2 statements write the same memory location L, called an output dependence or a read occurs before a write, called an anti-dependence.

This pattern can occur in on of 2 ways —

(1) $S_1$ can reference the memory location L on one iteration of a loop; on a subsequent iteration $S_2$ can reference the same memory location, L.

→ example of loop-carried dependence.

(2) $S_1$ & $S_2$ can reference the same mem. location, L on the same loop iteration but with $S_1$ preceding $S_2$ during execution of loop iteration.

→ Loop -independ~~ate~~ dependence.

Diagramatically, Different cases of loop-carried Dependences.

1 ≤ d ≤ n → dependence distance

| | iteration K | iteration K+d |
|---|---|---|
| | Loop-carried flow dependence | |
| Statement S1 | write L | — |
| S2 | | read L |
| | Loop-carried anti-dependence | |
| S1 | read L | |
| S2 | | write L |
| | Loop-carried output-dependence | |
| S1 | write L | |
| S2 | | write L |

② Data-race Conditions →

→ could be due to output dependences, in which multiple threads attempt to update the same mem. location, or variable, after threading.

→ In such situation, the code needs to be modified via privatization or synchronized using mechanisms like mutexes.

For eg → we can add private(x) clause to the parallel for pragme to eliminate data-race condition on variable x for any loop.

③ Managing Shared & Private Data →

→ In multithreaded program, understanding which data is shared and which is private becomes extremely important.

→ OpenMP makes this through a set of clauses like shared, private and default.

## (4) Loop Scheduling & Partitioning :⇒

→ schedule ( kind [, chunksize]) clause provide loop scheduling information, so that compiler & runtime library can better partition & distribute the iterations of the loop across the threads, and therefore the cores, for optimal load balancing.

→ Four scheduling schemes —

(1) static (2) dynamic (3) runtime (4) guided.

static ↓ default with no chunk size

dynamic ↓ chunks are handled with First come First serve scheme

→ For guided scheduling, partitioning of a loop is done based on formule ⇒ $\pi_K = \left\lceil \frac{\beta_K}{2N} \right\rceil$

$\beta_0$ = no. of loop iteration

N = no. of threads

$\pi_K$ = size of Kth chunk

## (5) Effective use of Reductions :⇒

→ In large applications, we often see the reduction operation inside hot loops.

→ Loops that reduce ~~the~~ a collection of values to a single value are fairly common.

→ reduction clause is used

e.g.

```
sum = 0;
#pragma omp parallel for reduction (+ : sum)
  for(k=0 ; k<100; k++) {
     sum = sum + fun(k);
  }
```

(+ → operator)

| operator | Initialization value |
|---|---|
| + | 0 |
| - | 0 |
| * | 1 |
| & | ~0 |
| \| | 0 |
| ^ | 0 |
| && | ! |
| \|\| | |

→ For each variable specified, a private copy is created, one for each thread, as if the private clause is used.

## 6 Minimizing Threading Overhead

→ Using OpenMP, we can parallelise loops, regions and sections or straight-line code blocks.

→ OpenMP employs the simple fork-join execution model, it allows the compiler & run-time library to compile & run OpenMP programs efficiently with lower threading overhead.

## 7 Work-sharing Sections

→ Since very few programs are comprised only of loops, additional constructs are used to handle non-loop code. A work-sharing is one such construct.

eg

```
#pragma omp parallel
{
  #pragma omp for
  for( K=0; K< m; K++) {
      x= f1(K) +f2(K);
  }
  #pragma omp sections private(y,z)
  {
    #pragma omp section
        { y= sectionA(x); f7(y); }
    #pragma omp section
        { z= sectionB(x); f8(z); }
  }
}
```

# Performance-oriented Programming

→ OpenMP provides a set of important pragmas & runtime functions that enable thread synchronization & related actions to facilitate correct parallel programming.

## (1) Using Barrier & Nowait

→ Barriers are a form of synchronization method that OpenMP employs to synchronize threads.

→ Threads will wait at a barrier until all the threads in ||l region have reached the same point.

→ barrier can be removed with nowait clause

## (2) Interleaving single-thread and Multi-thread Execution

→ In real world applications, a program may consist of both serial & parallel code segments due to various reasons such as data dependence constraints & I/O operations.

→ A need to execute something only once by only one thread will certainly be required within a parallel region, because we are making these regions as large as possible to reduce overhead.

→ So, OpenMP provides a way

## (2) Data Copy-in and Copy-out

→ firstprivate, lastprivate, copyin, copyprivate clauses are present in OpenMP to accomplish the data copy-in and copy-out operations.

→ When we parallelize a program, we would normally deal with how to copy in the initial value of a private variable to initialize its private copy for each thread.

→ We would also copy out the value of the private variable computed in the last iteration/section to its original variable for the master thread at the end of of parallel region.

## (4) Protecting Updates of Shared variables ⇒

→ The critical and atomic pragmas are supported by the OpenMP standard for us to protect the updating of shared variables for avoiding data-race conditions.

→ The code block enclosed by a critical section and an atomic pragma are areas of code that may be entered only when no other thread is executing in them.

eg

```
#pragma omp critical
{
  if (max < new_value) max = new_value
}
```

→ In practice, named critical sections are used when more than one thread is competing for more than one critical resource.

for eg:

```
#pragma omp critical (maxvalue)
{
  if (max < new_value) max = new_value
}
```

→ With named critical sections, applications can have multiple critical sections and threads can be in more than one critical section at a time.

→ Entering nesting critical sections runs the risk of deadlock.

atomic pragma :→ directs the compiler to generate code to ensure that the specific memory storage is updated atomically.

For eg

```
int main()
  { float y[1000];
    int k, idx[1000];
    #pragma omp parallel for shared(y, idx)
     for(K=0; K<8000; K++)
       {
         idx[K] = K%1000;
         y[idx(K)] = 8.0;
       }
    #pragma omp parallel for shared (y, idx)
     for (K=0; K<8000; K++)
     { #pragma omp atomic
         y[idx[K]]+ = 8.0 *(K%1000);
     }
     return 0;
  }
```

Here, advantage of using the atomic pragma is that it allows update of two different elements of array 'y' to occur in parallel.

If a critical section were used instead, than all updates to elements of array 'y' would be executed serially, but not in guaranteed order.

→ Thus, whenever it is possible we should use the atomic pragma before using the critical section in order to avoid data-race conditions on statements that update a shared memory location.

# Intel Taskqueuing Extension to OpenMP

→ allows a programmer to parallelize control structures such as recursive function, dynamic-tree search and pointer-chasing while loops

→ parallel taskq pragme directs the compiler to generate code to create a team of threads and an environment for while loop to enqueue the units of work specified by the enclosed task pragme.

→ captureprivate clause ensures that a private copy of the link pointer 'p' is captured at the time each task is being enqueued.

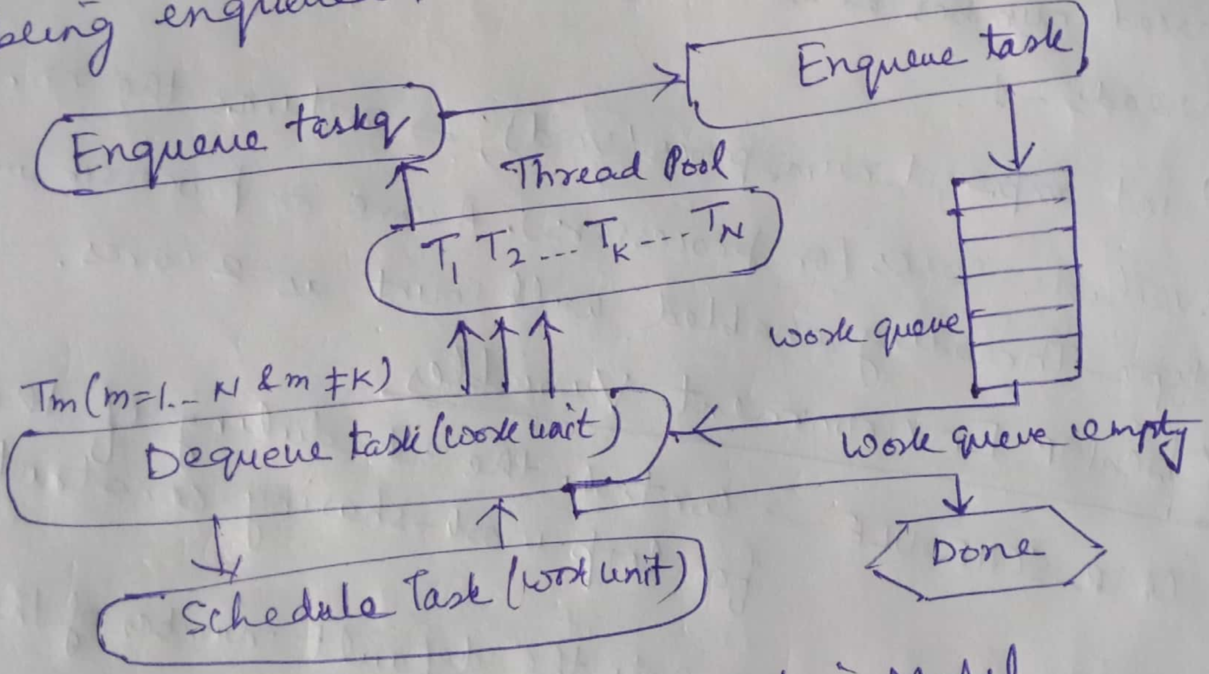

Fig : Taskqueuing Execution Model

# OpenMP Library Functions

① int omp_get_num_threads(void). → returns the no. of threads currently in use. If called outside a parallel region, this function will return 1.

② int omp_set_num_threads(int NumThreads); → This function sets the no. of threads that will be used when entering a parallel section. It overrides OMP_NUM_THREADS environment variable.

③ int omp_get_thread_num(void); → returns the current thread number between 0 (master thread) and total no. of threads - 1.

④ int omp_get_num_procs(void); → returns the no. of available cores (or processors). A core or processor with Hyper-Threading enabled will count as 2 cores.

## OpenMP Environment Variables

① OMP_SCHEDULE → controls the scheduling of the for-loop work-sharing construct.

② OMP_NUM_THREADS → Sets the default number of threads.