Intro to OpenMP

Subodh Sharma

Indian Institute of Technology Delhi

An API Standard

- A directive-based method to
  - invoke parallel computations on share-memory multiprocessors
- Specified for C/C++ and Fortran.
- Represents fork-join model of parallelism
- Jointly developed by all major h/w and s/w vendors:
  - HP, Fujistsu, AMD, IBM, Intel, Nvidia, Cray, TI, Oracle ..

Motivations?

- To primary parallelize regular loops.
  - Such as those in matrix multiplication
- Now supports task-parallelism too
  - Inspired from Cilk, TBBs, X10, Chapel
- Latest version has support for:
  - Accelerators
  - Atomics
  - Thread affinity
  - User defined reduction

How to Compile OpenMP

- >= gcc4.2 supports OpenMP 3.0
  – gcc –fopenmp example.c

- To change the number of threads:
    - setenv OMP_NUM_THREADS 4 (tcsh) or export OMP_NUM_THREADS=4(bash)
    - can also be provided in the code it self.

```c
            /*Introducing omp parallel */

#include <omp.h> // required header to write OpenMP code
#include <stdio.h>

int main (){

}
```

```c
/*Introducing omp parallel */

#include <omp.h> // required header to write OpenMP code
#include <stdio.h>

int main (){

// sentinel directive_name [clause[,clause]…]

#pragma omp parallel /* compiler directive */
{
   int tid = omp_get_thread_num(); /* Library call */
   printf(“hello world %d \n”, tid);

} // implicit barrier synchronization here!

}
```

```
Hello world 0
Hello world 2
Hello world 1
Hello world 3
```

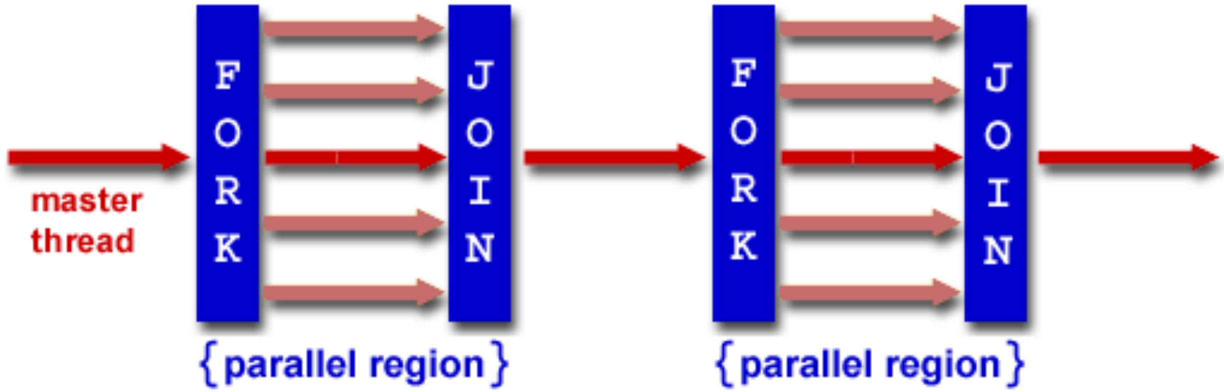
Why 4 threads? Ans: Default – can be changed by suitably setting the environment var: OMP_NUM_THREADS

```c
/*Introducing omp parallel num_threads(x) */

#include <omp.h> // required header to write OpenMP code
#include <stdio.h>

int main (){

#pragma omp parallel num_threads (2)
{
    int tid = omp_get_thread_num();
    printf("hello world %d \n", tid);

} // implicit barrier synchronization here!

}
```

Hello world 0
Hello world 1

Execution Model

[courtesy: xyuan, FSU]



- Execution model host-centric – host device offloads targets to target devices
- Threads can't migrate from one device to other
- A team is created when a parallel region is encountered

Execution Model

- **parallel** region creates tasks and map to threads
- implicit barrier at the end of **parallel** region
- Only the master resumes beyond **parallel** region
- Threads in a **team** executes worksharing tasks cooperatively

Memory Model

- OpenMP API provides a relaxed-consistency, shared-memory model
- Threads is allowed to have its own *temporary view* of memory.
  - cache, local storage (thread-private mem), registers
- Data-sharing attribute: **shared, private**
  - For each **private** var new

Memory Model

- a thread's temporary view of memory is not required to be consistent with memory at all times
  - **Solution:** **Flush** operation
- **Flush:** Strong, Rel, Acq
  - Strong: thread writes multiple times between two strong flush ops, guarantees the last write to be in memory.

```c
/*Introducing omp lib functions */

#include <omp.h> // required header to write OpenMP code
#include <stdio.h>

int main (){

#pragma omp parallel num_threads (4)
{
    int numt = omp_get_num_threads();
    int tid = omp_get_thread_num();
    printf(“hello world %d of %d \n”, tid, numt);

} // implicit barrier synchronization here!

}
```

Hello world 0 of 4
Hello world 2 of 4
Hello world 3 of 4
Hello world 1 of 4

```
/*Exposing data race*/

int main (){

int numt, tid ;

#pragma omp parallel num_threads (4)
{
   numt = omp_get_num_threads();
   tid = omp_get_thread_num();
   printf(“hello world %d of %d \n”, tid, numt);

} // implicit barrier synchronization here!

}
```

Hello world 0 of 4
Hello world 2 of 4
Hello world 3 of 4
Hello world 1 of 4

Data race not exposed!

```
/*Exposing data race*/

int main (){

int numt, tid ;

#pragma omp parallel num_threads (4)
{
    numt = omp_get_num_threads();
    tid = omp_get_thread_num();sleep(1);
    printf("hello world %d of %d \n", tid, numt);

} // implicit barrier synchronization here!

}
```

Hello world 3 of 4
Hello world 3 of 4
Hello world 3 of 4
Hello world 3 of 4

```
/*Fixing data race*/

int main (){

int numt, tid ;

#pragma omp parallel num_threads (4) shared (numt) private(tid)
{
    numt = omp_get_num_threads();
    tid = omp_get_thread_num();sleep(1);
    printf("hello world %d of %d \n", tid, numt);

} // implicit barrier synchronization here!

}
```

Hello world 1 of 4
Hello world 0 of 4
Hello world 2 of 4
Hello world 3 of 4

tid becomes a thread local variable – put on thread stack!

/*Thread local var storage*/

```
int main (){

int numt, tid ;

printf("hello world %d of %d: A(numt) = %x, A(tid) = %x  \n", tid, numt);

#pragma omp parallel shared (numt) private(tid)
{
  numt = omp_get_num_threads();
  tid = omp_get_thread_num();
  printf("hello world %d of %d: A(numt) = %x, A(tid) = %x  \n", tid, numt,
&numt, &tid);

}

}
```

tid for Master thread gets duplicated

from MASTER: A(numt) = 726fd678, A(tid) = 726fd67c

Hello world 0 of 4: A(numt) = 726fd678, A(tid) = 726fd64c

Hello world 3 of 4: A(numt) = 726fd678, A(tid) = fa87ae5c

Hello world 1 of 4: A(numt) = 726fd678, A(tid) = fb91ce5c

Hello world 2 of 4: A(numt) = 726fd678, A(tid) = fb0cbe5c

```
/*only one thread populating numt*/

int main (){

int numt, tid ;

numt = omp_get_num_threads();

#pragma omp parallel shared (numt) private(tid)
{
  tid = omp_get_thread_num();
  printf("hello world %d of %d\n", tid, numt);

}

}
```

Wrong move – threads are not even created yet!!

Hello world 3 of 1
Hello world 2 of 1
Hello world 0 of 1
Hello world 1 of 1

/*only one thread populating numt*/

```
int main (){

int numt, tid ;

#pragma omp parallel private(tid) {

tid = omp_get_thread_num();

if (tid == 0)
  numt = omp_get_num_threads();

}


#pragma omp parallel shared (numt) private(tid)
{
  tid = omp_get_thread_num();
  printf("hello world %d of %d\n", tid, numt);

}

}
```

computing tid multiple times

Hello world 0 of 4
Hello world 1 of 4
Hello world 3 of 4
Hello world 2 of 4

/*only one thread populating numt & use of barrier*/

```
int main (){

int numt, tid ;

#pragma omp parallel shared (numt) private(tid)
{
  tid = omp_get_thread_num();

   if(tid == 0) numt = omp_get_num_threads();

#pragma omp barrier

  printf("hello world %d of %d\n", tid, numt);

}

}
```

Hello world 0 of 4
Hello world 1 of 4
Hello world 3 of 4
Hello world 2 of 4

/*threadprivate(vars) – buggy usage*/

```
int tvar;

#pragma omp threadprivate (tvar)

int main (){

int numt, tvar = 10;

printf("From MASTER: A(tid) = %x \n", &tid);

#pragma omp parallel shared (numt) private(tid)
{
  tid = omp_get_thread_num();

 numt = omp_get_num_threads();

 printf("hello world %d of %d: tvar = %d, A(tvar) = %x \n", tid, numt, +
+tvar, &tvar);

}
}
```

Hello world 0 of 4: tvar = 11, A(tvar) = 880d0828
Hello world 2 of 4: tvar = 12, A(tvar) = 880d0828
Hello world 3 of 4: tvar = 14, A(tvar) = 880d0828
Hello world 1 of 4: tvar = 13, A(tvar) = 880d0828

/*threadprivate(vars) – copyin usage*/

```
int tvar;

#pragma omp threadprivate (tvar)

int main (){

int numt;
tvar = 10;

printf("From MASTER: A(tid) = %x \n", &tid);

#pragma omp parallel shared (numt) private(tid) copyin(tvar)
{
  tid = omp_get_thread_num();

 numt = omp_get_num_threads();

 printf("hello world %d of %d: tvar = %d, A(tvar) = %x \n", tid, numt, +
+tvar, &tvar);

}
}
```

copies the value of master's threadprivate var

Hello world 0 of 4: tvar = 11, A(tvar) = 880d0828
Hello world 2 of 4: tvar = 11, A(tvar) = 6f1d27b8
Hello world 3 of 4: tvar = 11, A(tvar) = 6d73e6f8
Hello world 1 of 4: tvar = 11, A(tvar) = 6df8f6f8

/*private(vars) – firstprivateusage*/

```
int main (){

int numt;
int tvar = 10;

printf("From MASTER: A(tid) = %x \n", &tid);

#pragma omp parallel shared (numt) private(tid) firstprivate(tvar)
{
  tid = omp_get_thread_num();

 numt = omp_get_num_threads();

 printf("hello world %d of %d: tvar = %d, A(tvar) = %x \n", tid, numt, +
+tvar, &tvar);

}
}
```

initialize the value with prior values available before the region

Hello world 0 of 4: tvar = 11, A(tvar) = 880d0828
Hello world 2 of 4: tvar = 11, A(tvar) = 6f1d27b8
Hello world 3 of 4: tvar = 11, A(tvar) = 6d73e6f8
Hello world 1 of 4: tvar = 11, A(tvar) = 6df8f6f8

```c
/*only one thread populating numt & use of thread single*/

int main (){

int numt, tid ;

#pragma omp parallel shared (numt) private(tid)
{
  tid = omp_get_thread_num();

#pragma omp single
{
   numt = omp_get_num_threads();
} // implicit barrier

  printf("hello world %d of %d\n", tid, numt);

}

}
```

Hello world 0 of 4
Hello world 1 of 4
Hello world 3 of 4
Hello world 2 of 4

```c
/*use of copyprivate with single directive*/

int main (){

int numt, tid, a ;

#pragma omp parallel shared (numt) private(tid, a)
{
  tid = omp_get_thread_num();

#pragma omp single copyprivate(a)
{
   numt = omp_get_num_threads();
   a = tid;
} // implicit barrier

  printf("hello world %d of %d\n: a = %d", tid, numt, a);

}

}
```

broadcasts a thread private var value to others

Hello world 0 of 4:a = 0
Hello world 1 of 4:a = 0
Hello world 3 of 4:a = 0
Hello world 2 of 4:a = 0

```
/*use of thread single with nowait*/

int main (){

int numt, tid ;

#pragma omp parallel shared (numt) private(tid)
{
  tid = omp_get_thread_num();

#pragma omp single nowait
{
   numt = omp_get_num_threads();
}

  printf("hello world %d of %d\n", tid, numt);

}

}
```

No implicit barrier

Hello world 0 of 4
Hello world 1 of 4
Hello world 3 of 4
Hello world 2 of 0

```
/*use of thread single with only master thread*/

int main (){

int numt, tid ;

#pragma omp parallel shared (numt) private(tid)
{
  tid = omp_get_thread_num();

#pragma omp master
{
   numt = omp_get_num_threads();
}   // no implicit barrier!

  printf("hello world %d of %d\n", tid, numt);

}

}
```

Only master thread executes

Hello world 0 of 4
Hello world 1 of 4
Hello world 3 of 4
Hello world 2 of 0

Computing Pi via Integral of 4.0/(1.0 + x*x)

```
/*Synchronization in OpenMP*/

/*Barriers*/
int main (){

int numt, tid ;

#pragma omp parallel shared (numt) private(tid)
{
  tid = omp_get_thread_num();

   if(tid == 0) numt = omp_get_num_threads();

#pragma omp barrier

  printf("hello world %d of %d\n", tid, numt);

}

}
```

```
/*Synchronization in OpenMP*/

/*Locks*/

omp_init_lock(omp_lock_t *)
   - Nestable locks are declared with the type
omp_nest_lock_t
omp_set_lock() -- acquires lock
omp_unset_lock() – releases lock
omp_destroy_lock() – free memory
omp_test_lock () – set the lock if available, else
return w/o blocking


while(!flag) {
   flag = omp_test_lock();

}
```

/*Synchronization in OpenMP*/

```
/*Ordered Construct*/

int main() {

int i;

# pragma omp parallel for ordered
{
  for (i=0; i < 5; i++)
      foo();
}
}
```

```
/*Synchronization in OpenMP*/

/*Critical Construct*/
sum = 0;
# pragma omp parallel shared (n,a,sum) private(tid,
sumLocal)
{
    tid = omp_get_thread_num();
    sumLocal = 0;

    #pragma omp for
      for(i=0;i<n;i++)
          sumLocal += a[i];

   #pragma omp critical (update_sum) // name of the block
      sum += sumLocal;
}
}
```

/*Synchronization in OpenMP*/

```
/*atomic Construct*/
sum = 0;
# pragma omp parallel shared (n,a,sum) private(tid,
sumLocal)
{
    tid = omp_get_thread_num();
    sumLocal = 0;

    #pragma omp for
      for(i=0;i<n;i++)
          sumLocal += a[i];

   #pragma omp atomic
      sum += sumLocal + foo();
}
}
```

/*use of Loop work sharing*/

```
#pragma omp parallel
{
#pragma omp for
  for (i = 0; i < N; i++)
   do_stuff();  // a[i] += b[i]
} // implicit barrier synchronization here!
```

- Limited to loops where iterations are a-priori known!
- The loop iteration variable is treated as thread local by the compiler
- iteration-to-threads mapping compiler defined!
- Loops must be free from loop-carried dependencies!

- **clauses supported**: private, firstprivate, reduction, schedule, nowait, ..

/*use of Loop work sharing*/

```
#pragma omp parallel for
{
  for (i = 0; i < N; i++)
    do_stuff();  // a[i] += b[i]
}
```

Succinct form when there is only a loop to be parallelized in "omp parallel" region

/*cyclic distribution of the same workload*/

```
#pragma omp parallel
{
   int id, Nthrds, istart, iend;
   id = omp_get_thread_num();
   Nthrds = omp_get_num_threads();
   // figure out istart, iend
   for (i = istart; i < iend; i++)
    do_stuff();  // a[i] += b[i]
}
```

```
/*use of Loop work sharing with schedule clause*/

#pragma omp parallel for schedule (static, chunk-size)
{
  for (i = 0; i < N; i++)
    do_stuff();  // a[i] += b[i]
}
```

- granularity of workload-per-thread is chunk-size
  - chunk-size – is a continuous non-empty subset of iteration space.
- Least overhead; chunks are assigned to threads in round-robin manner in the order of the thread IDs.
  - The last thread may get smaller number of iterations
- each thread gets at most 1 chunk if the chunk-size is unspecified

/*use of Loop work sharing with schedule clause*/

```
#pragma omp parallel for schedule (dynamic, chunk-size)
{
  for (i = 0; i < N; i++)
   do_stuff();  // a[i] += b[i]
}
```

- iterations are assigned to threads in the team of chunks
  - Each chunk = chunk-size many iterations
- once threads finish with their allotted chunk they request for more
- When no chunk-size specified – the size defaults to 1.
- Mostly used when the workload is unpredictable and poorly balanced

/*use of Loop work sharing with schedule clause*/

```
#pragma omp parallel for schedule (guided, chunk-size)
{
  for (i = 0; i < N; i++)
    do_stuff();  // a[i] += b[i]
}
```

- Similar to dynamic scheduling, except that chunks start decreasing over time
  - For chunk-size = **1** --- each chunk size will be proportional to (number of unassigned iterations)/(number of threads in the team)
  - For chunk-size = **k** ---size of the chunk determined as before but with the restriction that no chunk will have less than k iterations (except possibly for the chunk that contains the last iteration)
  - When no chunk-size is specified, it defaults to 1.

```
/*use of Loop work sharing with schedule clause*/

#pragma omp parallel for schedule (runtime)
{
  for (i = 0; i < N; i++)
   do_stuff();  // a[i] += b[i]
}
```

- the decision regarding scheduling is deferred until run time, and the schedule and chunk size are taken from the run-sched-var ICV (Internal Control Variable).

```
/*Reduction Clause*/

#pragma omp parallel for
{
  for (i = 0; i < N; i++)
    sum += a[i];
}
```

Can we do it this way?

Always a good debugging practice

```
/*Reduction Clause*/

#pragma omp parallel for default(none) reduction(+:sum)
{
  for (i = 0; i < N; i++)
    sum += a[i];
}
```

```
/*Reduction Clause*/

#pragma omp parallel for default(none) shared(a,N)
reduction(+:sum)
{
  for (i = 0; i < N; i++)
    sum += a[i];
}

/*Reduction Clause explicit implementation*/

#pragma omp parallel default(none) shared(a,N,sum)
{
  int sumLocal = 0;
#pragma omp for
  for (i = 0; i < N; i++)
    sumLocal += a[i];
#pragma omp critical
  sum += sumLocal
}
```

/*Reduction Clause*/

Operators and initial values supported by reduction clause

<table>
<tr><th>Operator</th><th>Initialization value</th></tr>
<tr><td>+</td><td>0</td></tr>
<tr><td>*</td><td>1</td></tr>
<tr><td>-</td><td>0</td></tr>
<tr><td>&</td><td>1</td></tr>
<tr><td>|</td><td>0</td></tr>
<tr><td>&&</td><td>1</td></tr>
<tr><td>||</td><td>0</td></tr>
</table>

```c
/*Parallel for Collapse*/

void sub(float *a)
{
 int i, j, k;
 #pragma omp for collapse(2) private(i, k, j)
 for (k=kl; k<=ku; k+=ks)
     for (j=jl; j<=ju; j+=js)
         for (i=il; i<=iu; i+=is)
           bar(a,i,j,k);
}
```

The iterations for the **k** and the **j** loops are collapsed into one loop – the iteration space is then divided according to the **schedule** clause.

/*use of Section work sharing*/

```
#pragma omp parallel{
    #pragma omp sections {
        #pragma omp section
           func_x();
        #pragma omp section
           func_y();
 } // sections region has an implicit barrier
}
```

- each thread executes the region within a section
- each section is executed only once
- if only one thread is there? What is the order of execution of regions?
- if more than two threads?
- Assignment of code blocks to threads is implementation-dependent

```
/*OpenMP Tasking Constructs*/

#pragma omp parallel
{
    #pragma omp task
      Compute1();
    #pragma omp task
      Compute2();
}
```

Order of execution of tasks is undefined!

- Binding: a **task** binds to the innermost enclosing **parallel** region
- Task synchronization: either by #pragma omp barrier or by #pragma omp taskwait
  - taskwait: explicit wait on the completion of child tasks

```
/*OpenMP Task Dependencies*/

 #pragma omp parallel
{
     #pragma omp task depend (OUT:x)
       x = Compute1();
     #pragma omp task depend (IN:x)
         Compute2(x);
 }
```

- **In Dependency:** generated task will depend on all previously generated sibling tasks with **out** or **inout** clauses
- **Out & Inout Dependency**: generated task will depend on all previously generated sibling tasks with **in, out** and **inout** clauses

```c
/*OpenMP Task Scheduling*/
void Foo(omp_lock_t *lock, int n){

    for (int i=0 ; i<n ; i++) {
        #pragma omp task
        {
            something_useful();
            while (!omp_test_lock(lock)){
            # pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
        }
}
```

- **taskyield:** the current task can be suspended in favor of the execution of a different task.
  - to avoid deadlocks.

```
/*Advanced OpenMP features - flush*/

#pragma omp flush

// 1. OpenMP standard specifies that all shared mem
modifications are available to all threads at synchronization
points

//2. Flush forces an update in between sync points

//3. It does not synchronize the actions of different threads

//4. Compiler can re-order flush operation – thus, it may not
execute exactly at the position relative to other operations
as specified by the programmer.
```

/*Performance Optimization Tips*/

1. Understand memory access patterns and perform loop interchange, if necessary

```
for (j=0; j<n; j++) {// Better soln: interchange the loops
        for (i=0; i<m; i++) {
         sum += a[i][j];
    }}
```

2. *Loop unrolling* is a powerful technique to avoid loop-overheads

```
for (i=1; i<n; i++) {
    a[i]= b[i] + 1;
    c[i] = a[i] + a[i-1] + b[i-1];
    /* transformed in to:
    for (i=1; i<n; i+=2) {
    a[i]= b[i] + 1;
    c[i] = a[i] + a[i-1] + b[i-1];
   a[i+1]= b[i+1] + 1;
    c[i+1] = a[i+1] + a[i] + b[i];
    */  }
```

/*Performance Optimization Tips*/

3. Loop with complex access patterns

```
for (j=0; j<n; j++) {    // loop tiling: exercise!
    for (i=0; i<m; i++) {

        a[i][j+1]= a[i+1][j] + 1;

}}
```

a[2][2] is computed in iter j=1, i=2 and used to assign a new value to a[1][3] in iteration j=2, i=1

4. Loop fusion/fission: merge/split two loops to create a bigger/smaller loop to increase cache usage.

```
Eg: for (i=0; i<n; i++) { // loop fission
        c[i]= exp(i/n);
        for (j=0; j<m; j++) {
            a[j][i] = b[j][i] + d[j]*e[i];
        }
    }
```

/*Performance Optimization Tips*/

5. Optimize use of barriers;

```
#pragma omp parallel  ... {

    #pragma omp for {
     for (i=0; i<n; i++)
         a[i] += b[i];
    }
    #pragma omp for {
      for (i=0; i<n; i++)
         c[i] += d[i];
    }
    #pragma omp for reduction (+:sum)
    for (i=0; i<n; i++)
         sum += a[i] + c[i];
}
```

6. Avoid large critical sections, use atomics where you can

/*Synchronization*/

barrier, critical, atomic, lock

/*Work Sharing*/

for, sections, single, master

/*Clauses to control Work Sharing*/

shared, private, firstprivate, copyin, default, nowait, schedule, reduction

/*Additional Library routines, environment variables*/