Shared Memory Programming with OpenMP

Salvatore Filippone

salvatore.filippone@uniroma2.it

OpenMP

Overview

- Introduction and basic features;
- Threading model;
- OpenMP API

Useful links:

- `http://openmp.org`
- `https://computing.llnl.gov/tutorials/openMP/`

Introduction: Thread Programming

We need specific programming constructs to define parallel computations. We shall use (sequential) C as a starting point.

- In this lecture, we investigate extensions to C that allow the programmer to express parallel activity in the thread- based, or data-sharing, style.
- There are two main approaches:
  1. Extensions that allow the programmer to create and manage threads explicitly *pthreads*
  2. Extensions that deal with threads and their management rather more implicitly. *OpenMP*

Introduction: Thread Programming

Different Levels of Thread-based Programming

- None of these schemes is fully implicit (i.e. fully automatic); full autoparallelisation of C (or any other serial) programs is beyond the present state-of-the-art. Instead, different schemes offer increasing amounts of high-level assistance for the creation and management of parallel threads.
- Pthreads and/or Win32 Parallel Threads Library
  ⇒ "Bare-metal" approach, i.e. the programmer is responsible for everything except the thread call implementation;
- OpenMP API
  ⇒ More functionality is provided, e.g. at the loop level; the programmer is presented with a simpler picture, but the scope for losing performance through naivety increases.

Implicit Thread Control: OpenMP

OpenMP is the de-facto standard API (Application Program Interface) for writing shared memory parallel applications in Fortran, C and C++. It was first conceived in 1997.

OpenMP consists of

- Compiler directives,
- Run time routines,
- Environment variables.

The specification is maintained by the OpenMP Architecture Review Board (`www.openmp.org`).

- Latest Version: 5.0, released November 2018; implementations under way, partial support available in GCC 9;
- Version 4.5, released in November 2015;
- Reference material at `https://www.openmp.org/resources/refguides/`

Advantages of OpenMP

- Good performance and scalability,
  ⇒ provided that you do it right. . .
- De-facto and mature standard.
- Portability: OpenMP is supported by many compilers;
- Simplifies the writing of multi-threaded programs in Fortran, C and C++;
- Allows a sequential program to be parallelized incrementally.
- Well suited for multicore architectures.
- Can be nested inside MPI programs on distributed memory machines (hybrid parallelization)

OpenMP Overview

How do threads interact?

OpenMP is a multi-threading, shared address model.

- It uses a Fork-Join model of parallelism to co-ordinate threads.
  $\Rightarrow$ A program runs in sequential mode using only the master thread until it reaches a section of the program that requires concurrent execution, when it creates multiple threads to perform the required functionality.
- Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  Use synchronization to protect data conflicts,

  but remember that synchronization are expensive.

OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.
  $\Rightarrow$ Appear as comments to non-OpenMP compilers.
- Most OpenMP constructs apply to a `block` (defined by braces in C, by an `end` statement in Fortran)
- a block has one or more statements with one point of entry at the top and one point of exit at the bottom.
  $\Rightarrow$ e.g. a loop nest or just a set of statements.
  $\Rightarrow$ No branches into, or out of, a block.

OpenMP core syntax

C Directives

- Format is case sensitive and is of the form:

```
#pragma omp directive-name [clause ...]
{  <block> }
```

- *#pragma* indicates to the compiler that the statement that follows is not part of the standard C language;
- `omp` indicates that the remainder of the statement is an OpenMP construct;
- Directive-name specifies a particular OpenMP directive/construct, and must appear after the sentinel and before any clauses;
- Clauses are optional, can be in any order, and are repeated as necessary, unless otherwise restricted.

OpenMP core syntax

PARALLEL Directive

The parallel directive defines a block of code that will be executed by a team of multiple threads. This is the fundamental OpenMP parallel construct. Threads are created at the beginning and destroyed at the end of the block.

```
  #pragma omp parallel [clause ...]
      if (scalar_logical_expression)
      private (list)
      shared (list)
      firstprivate (list)
      default (private|firstprivate|shared|none)
      reduction (operator: list)
      copyin (list)
      num_threads (scalar-integer-expression)
 {
    structured block
 }
```

OpenMP core syntax

Hello world

```
#pragma omp parallel private(tid)
  {
    /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    /* Only master thread does this */
    if (tid == 0)  {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  }
  /* All threads join master thread and terminate */
```

OpenMP core syntax

Shared Clause

- The shared clause states that the named variables should be available to all threads in the team during the extent of the parallel region.
- A shared variable exists in only one memory location and all threads can read or write to that address.

OpenMP core syntax

Shared Clause

```
  int x = 0;
#pragma omp parallel shared(x)
  {
    x++;
    printf("x: %d\n", x);
  }
```

The output from the program for a team of two threads would (with an extremely high probability) be either:

```
x:1
x:2
```

or

```
x:2
x:1
```

Warning:

this is unsafe! (we'll see how to do it properly).

OpenMP core syntax

Private Clause

- The private clause ensures that each thread gets its own instance of the specified variable(s).
- A new object of the same type is declared once for each thread.
- Changes in a private variable by one thread do not affect the value of that same variable being used by another thread.
- Cannot pass values into or out of a parallel region when using the private clause.

OpenMP core syntax

Private Clause

```
  int x = 10;
#pragma omp parallel private(x)
  {
    x = 0;
    x++;
    printf("x: %d\n", x);
  }
  printf("x: %d\n", x);
```

The output from the program for a team of two threads would be:

```
x:1
x:1
x:10
```

OpenMP: Work-Sharing Constructs

The for Directive

The for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team of threads. This assumes a parallel region has already been initiated, otherwise it executes serially on a single processor.

```
#pragma omp for [clause ...]
             schedule (type [,chunk])
             ordered
             private (list)
             firstprivate(list)
             lastprivate(list)
             shared (list)
             linear(list:step)
             reduction (operator: list)
             collapse (n)
             nowait
         for_loop
```

OpenMP: Work-Sharing Constructs

The for Directive

The loop is subject to restrictions:

- The loop variable must be integer;
- The increment must be constant;
- There must be a specified loop bound

i.e. *it must be possible to predict the exact number of iterations*

In other words, the `for` must follow the semantics of the Matlab `for` which in turns follows the semantics of the Fortran `do`

OpenMP: Work-Sharing Constructs

Schedule Clause

The schedule clause describes how iterations of the loop are divided amongst the threads in the team.

STATIC	loop iterations are divided into pieces of size chunk and statically assigned to threads.
DYNAMIC	loop iterations are divided into pieces of size chunk and dynamically scheduled amongst the threads.
GUIDED	uses “guided self scheduling” (GSS).
RUNTIME	as specified by the environment variable `OMP_SCHEDULE`.

Ordered Clause

The ordered clause specifies that the iterations of part of the loop (marked with an `ordered` construct) must be executed as they would be in a serial program.

Environment variables

OMP_NUM_THREADS how many threads should be started

OMP_SCHEDULE Applies only to parallel for directives which have their schedule clause set to `RUNTIME`; determines how iterations of the loop are scheduled, e.g.
`export OMP_SCHEDULE="guided, 4"` or
`export OMP_SCHEDULE="dynamic"`

OMP_DYNAMIC Whether the number of threads can be adjusted at runtime

OMP_NESTED Whether nested parallelism is allowed.

OpenMP: Work-Sharing Constructs

```
  /* Some initializations */
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
  {
#pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
  }
  /* end of parallel section */
```

By default, thread execution is synchronised at the end of a for by an implicit barrier, but not in this case `nowait`.

OpenMP: Work-Sharing Constructs

Sections Directive

The SECTIONS directive is a non-iterative work-sharing construct.

- The enclosed section(s) of code are to be divided among the threads in the team.
- Independent SECTION directives are nested within a SECTIONS directive. Each SECTION is executed once by a thread in the team.
- Different sections may be executed by different threads.
- It is possible for a thread to execute more than one section if it is quick enough and the implementation permits such.

Work-Sharing Constructs

Sections Directive

```
#pragma omp sections [clause ...]
           private (list)
           firstprivate (list)
           lastprivate (list)
           reduction (operator: list)
           nowait
{
#pragma omp section
  {
  structured_block
  }
#pragma omp section
  {
  structured_block
  }
}
```

Work-Sharing Constructs

Combined Parallel Work-Sharing Constructs

OpenMP provides two combined parallel/work-sharing directives:

- PARALLEL for
- PARALLEL SECTIONS

For the most part, these directives behave identically to an individual PARALLEL directive being immediately followed by a separate work-sharing directive. Most of the rules, clauses and restrictions that apply to both directives are in effect. See the OpenMP API for full details.

Work-Sharing Constructs

Parallel for

```
#pragma omp parallel for \
   shared(a,b,c,chunk) private(i) \
   schedule(static,chunk)
   for (i=0; i < n; i++)
    c[i] = a[i] + b[i];
```

Synchronisation Constructs

OpenMP provides a variety of Synchronization Constructs that control how the execution of each thread proceeds relative to other threads of the team:

MASTER directive: region executed by the master thread only:

```
#pragma omp master
{
structured block
}
```

CRITICAL directive: region executed by only one thread at a time; if one thread is executing the critical region, any other thread has to wait if it reaches the critical region.

```
#pragma omp critical
{
structured block
}
```

Synchronisation Construct Example

Consider the simple dot program code:

```
float dot=0.0
#pragma omp parallel shared(n,x,y,dot) private(tid,mydot)
  {
    tid = omp_get_thread_num();
#pragma omp for
    for (i=0; i<n; i++)
        dot += x[i]*y[i];
  }
```

In this version, the results will not be correct (or if they do, it's purely by chance).

Synchronisation Construct Example

One correct way to implement a dot product:

```
float dot=0.0
#pragma omp parallel shared(n,x,y,dot) private(tid,mydot)
  {
    tid = omp_get_thread_num();
    float mydot = 0.0;
#pragma omp for
    for (i=0; i<n; i++)
      mydot += x[i]*y[i];
#pragma omp critical (update_dot)
    {
      dot += mydot;
      printf("thread %d: mydot %f  dot %d\n",
             tid,mydot,dot);
    }
  }
```

The `critical` ensures proper result.

Clauses: reductions

The same example can be implemented even better with the `reduction` clause:

```
float dot=0.0
#pragma omp parallel for shared(n,x,y) \
     reduction(+: dot)
  for (i=0; i<n; i++)
      dot += x[i]*y[i];
```

This leaves the exact sequence of sums to the compiler:

- There exist much better algorithms than the simple sequential sum (cfr collective operations in MPI);
- With floating-point arithmetic, results will not be exactly identical (which applies to the `critical` example as well).

Synchronisation Constructs

BARRIER directive: synchronises all the threads in a team. When a `BARRIER` directive is reached, a thread will wait until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

```
#pragma omp barrier
```

TASKWAIT directive: wait on completion of child tasks since beginning of current task.

```
#pragma omp taskwait
```

ATOMIC directive: specifies that a memory location must be updated atomically, rather than letting multiple threads attempt to write to it.

```
#pragma omp atomic
```

LOCKS & ATOMIC

OpenMP provides an interface to acquire locks on certain regions.

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);
```

The ATOMIC directive

```
#pragma omp atomic
   i = i + 1;
```

Other OpenMP clauses

The IF clause

Allows control of parallelization to avoid excessive overhead

```
#pragma omp parallel if (n>5) shared(n) \
        private(tid)
  {
    tid = omp_get_thread_num();
#pragma omp single
    {
      printf("Value of n: %d\n",n);
      printf("Number of threads %d\n",
             omp_get_num_threads());
    }
    printf("Hello from thread %d\n",tid);
  }
```

There exist benchmarks to measure the various overheads.

Other OpenMP clauses

The NUM_THREADS clause

Allows control of number of threads, overriding the environment

```
  omp_set_num_threads(4);
#pragma omp parallel if (n>5) shared(n) \
    num_threads(n) private(tid)
  {
    tid = omp_get_thread_num();
#pragma omp single
    {
      printf("Value of n: %d\n",n);
      printf("Number of threads %d\n",
             omp_get_num_threads());
    }
    printf("Hello from thread %d\n",tid);
  }
```

Other OpenMP clauses

FIRSTPRIVATE

Take initial value of a private variable from the global variable with same name.

```
indx = 4;
#pragma omp parallel firstprivate(indx) \
        private(i,tid) shared(n,x)
{
  tid = omp_get_thread_num();
  indx += n*tid;
  for (i=indx; i<indx+n;  i++)
     x[i] += tid;
}
```

What does this code do?

Other OpenMP clauses

LASTPRIVATE	Ensures that the shared variable at the end will have the same value as if the code had been executed sequentially
DEFAULT	can take values `none`, `shared` or (only in Fortran) `private`. Using `none` forces to explicitly declare all variables;
ORDERED	Must be used if there is an `ORDERED` construct (rarely needed, expensive);
COPYPRIVATE	Provides a mean to broadcast data defined in a `single` region to all threads

Run-Time Library Routines

The OpenMP standard defines an API for library calls that perform a variety of functions, including:

- Query the number of threads/processors, and set the number of threads to use.
- Manipulate general purpose locking routines: initiate, destroy, set and unset a lock associated with a lock variable.
- Portable wall clock timing routines.
- Set execution environment functions: nested parallelism, dynamic adjustment of threads.

You can hide calls to run-time behind

```
#ifdef _OPENMP
```

Run-Time Library Routines

OMP_GET_NUM_THREADS Returns the number of threads that are currently executing the parallel region from which it is called.

```
#include <omp.h>
int omp_get_num_threads(void)
```

OMP_GET_THREAD_NUM Returns the thread number of the thread, within the team, making this call. This number will be between 0 and the value returned by `omp_get_num_threads` minus 1. The master thread of the team is labelled thread 0

```
#include <omp.h>
int omp_get_thread_num(void)
```

Dynamic threads

The number of threads *may* be adjusted at runtime.

omp_get_dynamic() Returns whether dynamic threads are enabled;

omp_set_dynamic(int) Sets the value

OMP_DYNAMIC Environment variable

OMP_SET_NUM_THREADS Sets the number of threads that will be used in the next parallel region. `num_threads` must be a positive integer.

```
#include <omp.h>
void omp_set_num_threads(int num_threads)
```

Other OpenMP Functionality

Tasks support (since OpenMP 3.0)

- The directives we have seen so far generally create relatively static "units of work" that get executed by a team of threads, where the work is based on regular data structures (e.g. arrays)
  $\Rightarrow$ e.g. a parallel do directive with static/chunk scheduling.
- OpenMP tasks supports the creation and scheduling of more arbitrary "units of work", defined as `tasks`.
- Tasking may be more suitable where less regular data structures are being used (e.g. linked lists or general graphs).
- For further OpenMP functionality, see the web links and the Summary card.

Tasking

The TASK constructs allows the programmer to define units of work that are irregular and dynamic (determined at runtime based on data contents), thereby allowing greater flexibility.

Tree traversal

```
struct node{
  struct node *left, *right;
}
extern void process (struct node *);
void traverse(stct node *p){
  if (p->left)
#pragma omp task // p is firstprivate
    traverse(p->left);
  if (p->right)
#pragma omp task // p is firstprivate
    traverse(p->rigth);
process(p);
}
```

Performance issues

Many factors concur to determine what level of performance your code will attain:

- Single thread performance and memory access; also, false sharing;
- Sequential work fraction;
- OpenMP construct overheads;
- Load imbalance;
- Explicit synchronization costs

Performance issues

Useful hints

- Optimize number of barriers (use `nowait` when possible)
- Avoid `ordered`;
- Make critical regions as small as possible;
- Maximize size of parallel regions;
- Avoid parallel regions in inner loops;
- Avoid false sharing;

Summary

High level abstractions like OpenMP make programming for shared memory systems easier, but they provide greater opportunity for performance to be lost, due to the unforeseen actions by the compiler or the run-time system.