Basics on Shared-Memory Parallel Programming

Riccardo Rossi

CIMNE, UPC, Barcelona

July 7, 2009

"What is OpenMP?"

Definition

It is an *An Application Program Interface* (API) that may be used to explicitly direct multi-threaded, shared memory parallelism

It is based on:

1. Compiler Directives

Jointly defined and endorsed by a group of major computer hardware and software vendors

www.openmp.org

"What is OpenMP?"

Definition

It is an *An Application Program Interface* (API) that may be used to explicitly direct multi-threaded, shared memory parallelism

It is based on:

1. Compiler Directives
2. Runtime Library Routines

Jointly defined and endorsed by a group of major computer hardware and software vendors

www.openmp.org

"What is OpenMP?"

Definition

It is an *An Application Program Interface* (API) that may be used to explicitly direct multi-threaded, shared memory parallelism

It is based on:

1. Compiler Directives
2. Runtime Library Routines
3. Environment Variables

Jointly defined and endorsed by a group of major computer hardware and software vendors

www.openmp.org

"What OpenMP is not?"

- Meant for distributed memory parallel systems (by itself)
- Necessarily implemented identically by all vendors
- Required to check for data dependencies, data conflicts, race conditions, or deadlocks
- ...

"Main Idea of OpenMP"

Definition

Programming model based on **relaxed memory consistency**

Each thread is allowed (temporarily) to have a slightly different view of the memory.
memory is syncronized only at some points in the program, as prescribed **by users** through directives.

IMPLICATION:

1. Allows writing very efficient programs

"Main Idea of OpenMP"

Definition

Programming model based on **relaxed memory consistency**

Each thread is allowed (temporarily) to have a slightly different view of the memory.
memory is syncronized only at some points in the program, as prescribed **by users** through directives.

IMPLICATION:

1. Allows writing very efficient programs
2. the order in which instructions are executed is non-deterministic –¿ it is **VERY difficult** to reproduce bugs in a multithreaded program

"Main Idea of OpenMP"

Definition

Programming model based on **relaxed memory consistency**

Each thread is allowed (temporarily) to have a slightly different view of the memory.
memory is syncronized only at some points in the program, as prescribed **by users** through directives.

IMPLICATION:

1. Allows writing very efficient programs
2. the order in which instructions are executed is non-deterministic –¿ it is **VERY difficult** to reproduce bugs in a multithreaded program
3. (very) difficult to debug

"Getting Started"

Compilers;

- gcc 4.2,4.3,4.4. OpenMP 3.0 from 4.4. Compilation command *-fopenmp*
- intel compiler 9,10,11. OpenMP 3.0 in release 11. Compilation command *-openmp*
- msvc 2005-2008

Environment variables:
To set the number of threads (to 4 in this case):

```
export OMP_NUM_THREADS=4
```

timing routines

Sequential timing routines **DO NOT WORK** in conjunction with openmp
To do timing omp functions are needed (pragmas are not enough)

```
#import "omp.h"
```

and then time as follows

```
double start = omp_get_wtime();
do_something();
double stop = omp_get_wtime();
std::cout << "time: " << stop - start;
```

"an embarassingly parallel problem"

Let's consider the sum of three vectors of size N:

$a = b + c$

This can be written in c++ as:

```
for(int i=0; i<N;i++)
 {
    a[i] = b[i] + c[i];
 }
```

Each of the elements of "a" can be computed independently of the others.

parallelizing means here "finding a way" of splitting the work between the different CPUs

"parallelizing a for loop"

A way is needed to tell to the compiler to put the loop in parall.
achieved through a "pragma"

```
int N = A.size();
int i;
#pragma omp parallel for
  private(i), firstprivate(N)
for(i=0; i<N;i++)
{
  a[i] = b[i] + c[i];
}
```

This instructs the compiler to divide the work between the different threads.

why "pragmas"

Definition

PRAGMA: A message written into the source code that tells the compiler to compile the program in some fashion that differs from the default method. Ignored if not recognized.

What happens if the code is to be compiled "without openmp"? All "pragmas will be ignored and a serial version will be compiled

Parallel and serial version of library can coexist!

essential "details"

The example of loop shown is not "just an example".
In OpenMP 2.0 and 2.5 the loop counter "i" **has to be an "int"**. no other type is allowed!!!
in particular neither unsigned int nor std::size_t is valid!

OpenMP 3.0 (finally!!!) removes this requirement (more on this later)

private and shared variables

let's observe again the loop

```
int N = A.size();
int i;
#pragma omp parallel for
  private(i), firstprivate(N)
for(i=0; i<N;i++)
{
  a[i] = b[i] + c[i];
}
```

the variable "i" and "N" are defined outside of the parallel region. They are needed by all of the threads.
in particular:

- "i" will updated (written!) at every iteration
- "N" will be READ at every iteration

private and shared variables

"i" has to assume different values depending on the thread!

- we need to update "i" separately on each thread
- the initial value (outside of the for) of "i" is NOT of interest
- the final value of "i" is unimportant

a statement is needed to tell the compiler that a copy of "i" is needed on each thread

private(i)

tells that each thread will play with a private copy of the variable

private and shared variables

"N" has the same value on different threads!

- a common value would be sufficient
- we may wish each thread to have a local copy of N for optimization purposes. In this case the the initial value of "N" is important
- as before the final value of "i" is unimportant

OPTION 1 - a common copy of N for all threads:
In this case the value is "shared". We should specify shared(N)
OPTION 2 - a private copy of N:
We want it private and we want it initialized with the value at the beginning of the loop. firstprivate(N)

private and shared variables

"Classifying" the variables is essential in FORTRAN. For the cases in which many variables are to be specified the possibility exists of specifying a default assumption
for example

```
default(private)
```

the alternatives are

- default(shared)
- default(none)

an interesting option is **default(none)** as it obliges to specify all of the variables that will be touched in one loop.

variable scoping (how C/C++ makes your life easier)

The "life span" of a variable is defined by its "scope" which is well defined already by the C standard.
The implication is that **the specification of the "type" of the variables is largely unnecessary in C/C++**, for example the loop shown before could be written as

```
int N = A.size();
#pragma omp parallel for
for(int i=0; i<N;i++)
  a[i] = b[i] + c[i];
```

- "i" is defined in the parallel part $\Longrightarrow$ it is private to each thread
- "N" is defined in the serial part $\Longrightarrow$ it is common (shared) between the threads

OPTION 1 is automatic!

Parallel Regions

The parallelization capabilities of OpenMP are completed by the "parallel" directive, which ensures that one portion of the code (not necessarily a loop) is executed in parallel

```
#pragma omp parallel
{
  all in here is in parallel
}
```

full control on what happens inside a parallel region is given by a bunch of other directives (see documentation for a better description)

- master
- single
- section $\longrightarrow$ allows to prescribe one portion of code to be executed by a given thread

All good ... but ... where is the problem?

Now we want to write a simple parallel statement that prints out a greeting and the number of threads. We need:

- a way to query the thread number of a given processor
- a way to make different threads to execute a same portion of code in parallel

This is as simple as

```
#pragma omp parallel
{
  int my_id = omp_get_thread_num();
  std::cout << "ID=" << my_id << std::endl;
}
```

All good ... but ... where is the problem?

we run on 2 threads and we look at the result:

FIRST RUN:

```
ID=1
ID=0
```

SECOND RUN:

```
ID=0
ID=1
```

THIRD ATTEMPT:

```
IID=D=0
1
```

what happend? The effect of a "cout" (or printf, write, etc.) is in the sequence:

- write data onto an auxilary variable
- output such auxiliary variable to the screen

An error in OpenMP??

The order at which the different threads are run is not specified

The different treads are run in any order, so in particular they may (write) access to the same variable in the "natural" order ... or in any possible combination

This is a crucial feature of OpenMP and of all the parallel programming models

"text" variables are no exception

Having two threads to write on the same "double" var leads to **unpredictable** (read WRONG) results

What happens on reading?

Threads also read the memory in random order. The only problem is that only one can access the same memory at time. **Performance issue**

Race conditions

In a parallel context threads compete (race) to access to data. The OpenMP standard provides the tools to manage the interaction of the threads ... not a single solution hence ... **The Art of Parallel Programming**.

In the following:

- Motivate the OpenMP choice
- Discuss a little the load balancing (scheduling)
- Go to next section and discuss the syncronization of data

Motivate the OpenMP choice

Basic Observations:

1. In a perfect parallel problem we have many "threads" running independenly one from the other. **As long as they do not interact the parallelism is not spoiled**
2. Independent threads may take different times to execute different tasks
3. If we oblige threads to start in parallel than we will have to oblige them to finish in parallel ... and have threads to wait without doing anything.

SOLUTION: do not specify order of execution of threads. Oblige the user to specify it to achieve the desidered result!

Scheduling

Let's define the (admittedly stupid) following function:

```
double f(int i)
{
  if(i<10) return pow(2.0,i);
  else return 2.0 + double(i);
  //second option is much faster than the first
}
```

and now consider the following simple parallel loop

```
std::vector<double> temp(20);
#pragma omp parallel for
for(int i=0; i<20; i++)
{
  temp[i] = f(i);
}
```

Scheduling

The naive approach would assign the operations 1-10 to the first thread and the remaining ones to a second thread **INEFFICIENT!** the first thread would have to run all of the expensive operations!

Scheduling

The OpenMP provides different strategies for dividing (scheduling) the different operations between the different threads

Scheduling

- **STATIC** Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.
- **DYNAMIC** Loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.
- **GUIDED** (see documentation)
- **RUNTIME** (decision deferred at runtime)
- **AUTO** (new option) decision left to the compiler/system

it is possible to specify a CHUNKSIZE, basically the number of operations to be processed together

computing the norm of a vector

Let's consider now we want to compute the norm of a vector. The scalar code would loop something like

```
std::vector<double> a(100000);
double norm = 0.0;
for(int i=0; i<a.size(); i++)
  norm = norm + a[i]*a[i];
norm = sqrt(norm);
```

until now we learnt how to parallelize the loop, however we are not sure on how to obtain the final norm.

computing the norm of a vector

makign use of what we know we could write

```
std::vector<double> a(100000);
//create an auxiliary vector
 int available_threads = omp_get_max_threads();
 std::vector<double> aux(available_threads);
 for(int i=0; i<available_threads; i++)
       aux[i] = 0.0;
//each thread computes the sum of the squares in
#pragma omp parallel for
 for(int i=0; i<a.size(); i++)
 {  int thread_id = omp_get_thread_num();
  aux[thread_id] = aux[thread_id] + a[i]*a[i];}
//sum up contributions with a non-parallel loop
double norm = 0.0;
for(int i=0;i<available_threads;i++) norm norm+au
norm = sqrt(norm);
```

long and inefficient!!!

computing the norm of a vector

Much better than this in OpenMP! $\rightarrow$ **reduction** clause

```
std::vector<double> a(100000);
double norm = 0.0;
#pragma omp parallel for reduction(+,norm)
for(int i=0; i<a.size(); i++)
  norm = norm + a[i]*a[i];

norm = sqrt(norm);
```

The only modification to the standard parallel for is the statement reduction(+,norm)
which says that "norm" is the variable to be reduced and that the reducing operation is "+"

REMARK:

This is the first time we need all of the thread to "syncronize" to provide a common output

the problem of the syncronization

We are already aware that the sycronization between the different threads may lead to "troubles".

Syncronization:

OpenMP provides different constructs to manage the syncronization efficiently

- **atomic** directive
- **critical** directive
- "locks" (semaphores)

"critical" directive

Definition

The **critical** directive ensures that the code included in the area controlled by the critical statement will be guaranteed to be run by a single thread at a time

Imagine for example we want to count the number of threads (of course there exist better ways of doing it)

```
int x = 0;
//lets start a parallel region
#pragma omp parallel
  //but now let's oblige all the threads
  //to update X without race condition
  #pragma omp critical
  x += 1;
```

extremely inefficient(scalar code would be far faster) however ... it works! syncronization is dealt with correctly

"atomic" directive

Definition

The **atomic** directive specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it. In essence, this directive provides a mini-CRITICAL section with restrictions on the type of operations that can be performed.

ATOMIC has more restrictions than critical (see documentation for exact limits of use) ... however it is much faster in the sense that it has a much lower overhead

"lock" directives

Definition

The system of "locks" provide more flexibility in controlling the syncronization. Essentially once a lock is "set" any other thread trying to access to the locked code will have to wait until the process that locked it has finished

- Before usage locks HAVE to be initialized and deinitialized after
- a lock need to be "set" and "unset"

A realistic example

Let:

- **A,b** be a system matrix and vector (read large matrices)
- **LHS,RHS** a local matrix and vector as computed by ComputeSystemContributions()
- **EqId** a small array with the global ids to which the local terms have to be assembled

we want to write a credible FE assembly procedure (code not compilable for space reasons)

FE Assembly "scalar version"

The scalar version may look something like

```
for(int el = 0; el<nels; el++)
{
  ComputeSystemContributions(el,LHS,RHS)
  ComputeIds(EqId)

  Assemble(LHS,RHS,EqId,A,b)
}
```

FE Assembly using critical

A first parallel version

```
#pragma omp parallel for
        private(EqId,LHS,RHS)
for(int el = 0; el<nels; el++)
{
  ComputeSystemContributions(el,LHS,RHS)
  ComputeIds(EqId)

  #pragma omp critical
  {
    Assemble(LHS,RHS,EqId,A,b)
  }
}
```

works efficiently as long as the time in constructing LHS and RHS is very large compared to the time for writing on A

FE Assembly using atomic

As an alternative we could use the "atomic" directive redefining the Assemble

```
#pragma omp parallel for
        private(EqId,LHS,RHS)
for(int el = 0; el<nels; el++)
{
  ComputeSystemContributions(el,LHS,RHS)
  ComputeIds(EqId)

  AtomicAssemble(LHS,RHS,EqId,A,b)

}
```

FE Assembly using atomic

As an alternative we could use the "atomic" directive redefining the Assemble

```
void AtomicAssemble(LHS,RHS,EqId,A,b)
{
  for(int I=0; I<LHS.size1(); I++)
  {
    #pragma omp atomic
    b[EqId[I]] += RHS[I];

    for(int J=0; J<LHS.size2(); J++)
    {
      #pragma omp atomic
        A(EqId[I],EqId[J]) += LHS(I,J);
    }
  }
}
```

atomic is called MANY times ... efficient?

FE Assembly using locks


```
//construct and initialize an array of locks
std::vector< omp_lock_t > lock_array(A.size1());
int A_size = A.size1();
for(int i = 0; i<A_size; i++)
    omp_init_lock(&lock_array[i]);

//perform the parallel assembly loop
#pragma omp parallel for
      private(EqId,LHS,RHS)
for(int el = 0; el<nels; el++)
{  ComputeSystemContributions(el,LHS,RHS)
  ComputeIds(EqId)
  LockedAssemble(LHS,RHS,EqId,A,b,lock_array)}

//deinitialize the locks
for(int i = 0; i<A_size; i++)
  omp_destroy_lock(&lock_array[i]);
```

FE Assembly using atomic


```
void LockedAssemble(...,lock_array)
{ for(int I=0; I<LHS.size1(); I++)
  {
    int assembly_row = EqId[I];
    omp_set_lock(&lock_array[assembly_row]);

    //perform assembly
    b[EqId[I]] += RHS[I];
    for(int J=0; J<LHS.size2(); J++)
       A(EqId[I],EqId[J]) += LHS(I,J);

    omp_unset_lock(&lock_array[assembly_row]);
  }
}
```

the only check is that two threads do not access to the same line ... should be efficient...

OpenMP today

OpenMP 3.0 was released in 2008 and is currently supported only on the newest compilers ... this is a problem for portability
OpenMP 2.5 can now be found on most modern compilers

Do we need the last standard?

- OpenMP 2.5 very mature for Fortran but with a few very important limitations for C++
- OpenMP 3.0 is thought to bring C++ at full speed! targeted at C++

News in OpenMP 3.0

1. omp loops can use types different from "int"
2. support for std::iterators in parallel loops
3. standardize support for threadprivate variables (including class statics)
4. introduce "tasking"

NUMBER 3 is crucial

threadprivate class variables (or statics) were a showstopper for old openmp (standard was not complete and implementations diverged)

ThreadPrivate variables

It is now possible for a class to have a **threadprivate static member** (for example an auxiliary work array). This is fundamental in order to make the class threadsafe
Showstopper in OpenMP 2.5 as at the moment of writing the omp loops no information is available on the internal structure of the objects which are known only through their interface (and work arrays are important for efficiency)

Tasking

It is a new feature. Basically it is possible to define "tasks" to be assigned to threads. Tasks are thought so to make possible for example their recursive execution

The application to the construction of a Quick Sort can be found at:

http://wikis.sun.com/display/openmp/Tasks+vs+Nested+Parallel+Regions

Alternatives to OpenMP

- Directly dealing with threads ... why???
- Intel TBB (Threading Building Blocks) ... very advanced scheduling strategies. PORTABLE

As a Hint ... Use OpenMP But **use libraries!!**
Have a look at "omptl" (just google it) provides a parallel version of the standard library