An introduction to
The openMP programming environment

- Introduction: the openMP model
- Basic syntax
- A few examples
- See also the following for many resources:
  http://openmp.org
Threads and the openMP model

- openMP implements the Fork-Join model
- Supports data parallelism
- Easy to parallelize loops or parallel sections of codes

Pros: Arguably the simplest approach to parallel programming.
Cons: Limited to SMPs [Shared memory computers]
The openMP approach

Use C or Fortran and add directives / pragmas

* Indicate parallel loops,

* Parallel regions of code, ..

Plus a few library routines [e.g., \texttt{OMP\_GET\_THREAD\_NUM()}]

Intrinsically designed for Shared Memory SMP machines.

Portable – supported by all High Performance computer vendors

– see:

\url{http://openmp.org}

and implemented in GNU compilers (gcc,..).
Directives/ pragmas:

In C: 
```
#pragma omp ... directives
```

In Fortran: 
```
!$OMP ... directives
```

```c
#pragma omp parallel
{
  ...
  // structured block
  ...
}
```

```c
int i;
#pragma omp parallel for
for (i=0; i<n; i++) {
  y[i] += x[i];
  ...
}
```

These notes will illustrate only a few directives

See [http://openmp.org/](http://openmp.org/) for additional details:

- Reference guide for a quick overview
- Specifications [a pdf file] for details
Basic functions

- `omp_get_thread_num()` - get thread number
- `omp_set_num_threads(nthreads)` - set # of threads
- `omp_get_num_threads()` - get number of threads used

Example:

```c
#include <omp.h>
int omp_get_thread_num();
int main(){
    #pragma omp parallel
    {
        printf("Thread number : %d\n", omp_get_thread_num());
    }
}
```
1. Compile with `gcc -o test.ex -fopenmp test.c`

2. Set number of threads with environment variable:

   `setenv OMP_NUM_THREADS 4`

3. Run

   `.test.ex`
   
   Thread number: 0
   Thread number: 3
   Thread number: 2
   Thread number: 1
Hello World in openMP: pragma parallel

Compile and run this other version of the previous example

```c
#include <stdio.h>
#include <stdlib.h>
int main () {
    int i;
    int omp_get_thread_num();
    printf("Entering parallel threads: \n");
    #pragma omp parallel
    {
        i = omp_get_thread_num();
        printf(" -->> Hello from thread : %d \n",i);
    }
    printf(" <<<-- Out of threads \n");
}
```
Hello World in openMP: pragma parallel for

```c
#include <stdio.h>
#include <stdlib.h>

int main () {
    int i, p;
    int omp_get_thread_num();
    printf(" Entering parallel threads: \n");
    #pragma omp parallel for
    for (i =0; i < 12; i ++) {
        p = omp_get_thread_num();
        printf(" -->> Hello from thread : %d \n", p);
    }
    printf(" <<<-- Out of threads \n");
}
```
compile and run:

```bash
gcc -fopenmp omp_hello.c
```

- Can set the number of threads from environment variable...
  ```bash
  setenv OMP_NUM_THREADS 4
  ```

- ... or in the code with the command
  ```c
  omp_set_num_threads(nthreads)
  ```

- This freezes the number of threads [takes precedence over environment variable OMP_NUM_THREADS]

What is the difference between the two examples?
Scoping of variables

- Variables can be shared among threads as in
  
  ```c
  #pragma omp parallel shared(var1, var2, ...)
  ```

- Beware of racing between variables.. [no guaranteed order of modifications]

What can happen if several threads write to the same shared variable? See situation in following example.

```c
#define N_MAX 10000
int main() {
  int i;
  double fx, fsum=0.0;
  #pragma omp parallel for
  for (i = 1; i <= N_MAX; i++) {
    fx = (double)i;
    fsum += fx;
  }
  printf("-- sum %f \n", fsum);
}
```

Program race.c:
**Private variables**

Variables can be private – local copies of variables made for each thread – Note: when copies are made they are *not* initialized

```
#pragma omp private(var1, var2, ...)
```

- Can set default for scoping of variable by

```
#pragma omp default(DEF)
```

where DEF == one of private, shared, or none.

- If no default is set, and there is no explicit clause for scoping, variables are assumed to be shared
Example: Dot-Product

```c
omp_set_num_threads(nt); // nt = number of threads
    m = n/nt; // assumes n divisible by nt (!)
#pragma omp parallel for private(t, i1, i2, i)
for (it = 0; it < nt; it++) {
    i1 = it*m;
    i2 = i1+m;
    if (i2 > n) i2 = n;
    t = 0.0;
    for ( i = i1; i < i2; i++ )
        t += x[i]*y[i];
    tt[it] = t;
}
t = 0.0;
for (it = 0; it < nt; it++)
    t += tt[it];
```
Critical sections

Solutions to race conditions: critical sections which permit a code fragment to be executed by one thread only

```
#pragma omp critical [name]
{
    ... structured block ... 
}
```

Go back to program race.c seen earlier.

Here is how it can be corrected:

a) First declare fx as private.

b) Then summation should be critical. [loss of parallelism]
Program race_cor.c:

```c
#define N_MAX 10000
int main() {
    int i;
    double fx, fsum;
    #pragma omp parallel for private(fx)
    for (i = 1; i <= N_MAX; i++)
        {fx = (double)i ;
        #pragma omp critical
        fsum += fx;}
    printf("-- sum %f \n", fsum);
}
```

Better solution: Reduction operation
Reduction operations

A reduction does a global operation (e.g. a sum) on an array down to one single result. For example $a = \sum_{i=0}^{n-1} x_i$ or $a = \max_{i=0}^{n-1} x_i$, ..

Clause syntax: `reduction(<op >: variable)`

ope = +, x, min, max, ..
Example: Dot product computation seen earlier

```c
omp_set_num_threads(nt);
...
t = 0.0;
#pragma omp parallel for reduction(+:t)
for (i = 0; i<n; i++)
t +=x[i]*y[i];
```

- Private copy of $t$ (in clause) is created for each thread.
- At the end of reduction, reduction operation $+$ (in clause) is applied to private variable $t$ (in clause) –
- Result of this reduction written to ‘master’ thread (shared variable) $t$
Sections

- Each section executed by one thread
- Cannot branch into and out of block of sections

```c
#pragma omp sections
{
  // section 1
  #pragma omp
  {
    block
  }
  // section 2
  #pragma omp
  {
    block
  }
  ...
}
```
Consider the example

```c
#pragma omp parallel for
for (i=0; i<n; i++) {
    /* --- cost of function varies a lot with i */
    x[i] = some_function(i);
}
```

Result: poor load balancing. Solution schedule work dynamically.

```
schedule (type [,chunk])
type is one of static, dynamic, guided or runtime
```
runtime = set by an environment variable

`setenv OMP_SCHEDULE `‘type,chunk’` command.

```c
#pragma omp parallel for schedule(dynamic)
for (i=0; i<n; i++) {
    /* cost of function varies a lot with i */
    x[i] = some_function(i);
}
```
A few runtime functions

- `omp_get_num_threads()` returns current number of threads
  Note: this is always one in a sequential section.

- `omp_set_num_threads(int)` sets number of threads in code

2 other methods for this: environment variable (seen before), and
clause `[#pragma omp parallel num_threads(3)]`

- `omp_get_num_procs()` returns current number of processors available