Introduction to shared memory parallel programming with OpenMP

Paschalis Korosoglou (support@grid.auth.gr)
User and Application Support
Scientific Computing Center @ AUTH
Overview

- Shared vs Distributed memory models
- Why OpenMP
- How OpenMP works
- Basic examples
- How to execute the executable
Hardware considerations
Introduction to OpenMP

- API extension to C/C++ and Fortran languages
  - Most compilers support OpenMP
    - GNU, IBM, Intel, PGI, PathScale, Open64 ...
  - Extensively used for writing programs for shared memory architectures over the past decade
  - Thread (process) communication is implicit and uses variables pointing to shared memory locations; this is in contrast with MPI which uses explicit messages passed among each process
Threaded parallel programming (openMP)

- openMP is based on a fork - join model
  - Master - worker threads

Use of directives and pragmas within source code
Approach towards parallelism

- From serial to parallel with OpenMP
Memory issues

- Threads have access to the same address space
  - Communication is implicit
- Programmer needs to define
  - local data
  - shared data
Yet another hello world example

```fortran
PROGRAM HELLO

INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS,
       OMP_GET_THREAD_NUM

! Fork a team of threads giving them their own copies of variable:
$OMP PARALLEL PRIVATE(TID)

! Obtain thread number
TID = OMP_GET_THREAD_NUM()
PRINT *, 'Hello World from thread = ', TID

! Only master thread does this
IF (TID .EQ. 0) THEN
   NTHREADS = OMP_GET_NUM_THREADS()
   PRINT *, 'Number of threads = ', NTHREADS
END IF

! All threads join master thread and disband
$OMP END PARALLEL
END
```

```c
#include <iostream>
#include <omp.h>

using namespace std;

int main(int argc, char* argv[]) {
    #pragma omp parallel
    {
        // Your parallel code here
        printf("Hello World! This is thread %d out of %d\n", omp_get_thread_num(), omp_get_num_threads());
    }
    return 0;
}
```
## Common API calls

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int omp_get_num_threads()</td>
<td>Returns the number of threads in the concurrent team</td>
</tr>
<tr>
<td>int omp_get_thread_num()</td>
<td>Returns the id of the thread inside the team</td>
</tr>
<tr>
<td>int omp_get_num_procs()</td>
<td>Returns the number of processors in the machine</td>
</tr>
<tr>
<td>int omp_get_max_threads()</td>
<td>Returns maximum number of threads that will be used in the next parallel region</td>
</tr>
<tr>
<td>double omp_get_wtime()</td>
<td>Returns the number of seconds since a time in the past</td>
</tr>
<tr>
<td>bool omp_in_parallel()</td>
<td>1 if in parallel region, 0 otherwise</td>
</tr>
</tbody>
</table>
```cpp
#include <iostream>
#include <omp.h>

using namespace std;

int main(int argc, char* argv[]) 
{
    double start = omp_get_wtime();
    if( !omp_in_parallel() )
    {
        printf("Number of processors is: %d\n", omp_get_num_procs());
        printf("Number of max threads is: %d\n", omp_get_max_threads());
    }
    sleep(1);
    double end = omp_get_wtime();
    printf("start = %.16g\nend = %.16g\ndiff = %.16g\n", start, end, end - start);
    return 0;
}
```
Data scoping

- For each parallel region the data environment is constructed through a number of clauses
  - shared (variable is common among threads)
  - private (variable inside the construct is a new variable)
  - firstprivate (variable is new but initialized to its original value)
  - default (used to set overall defaults for construct)
  - lastprivate (variable’s last value is copied outside construct)
  - reduction (variable’s value is reduced at the end)
A few examples

```c
int x=1;
#pragma omp parallel shared(x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);

int x=1;
#pragma omp parallel private(x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);

int x=1;
#pragma omp parallel firstprivate(x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```

x = 3
x = 2
x = 3

or

x = 2
x = 3
x = 3

Will print anything and then x=1

x = 2
x = 2
x = 1
Synchronization

- OpenMP provides several synchronization mechanisms
  - barrier (synchronizes all threads inside the team)
  - master (only the master thread will execute the block)
  - critical (only one thread at a time will execute)
  - atomic (same as critical but for one memory location)
Synchronization examples

foo(3), foo(3)  

int x=1;  
#pragma omp parallel num_threads(2)  
{  
#pragma omp master  
{  
x++;  
}  
foo(x);  
}  

foo(2), foo(2)  or  foo(1), foo(2)

int x=1;  
#pragma omp parallel num_threads(2)  
{  
#pragma omp barrier  
foo(x);  
}

foo(2), foo(3)  

int x=1;  
#pragma omp parallel num_threads(2)  
{  
#pragma omp critical  
{  
x++;  
foo(x);  
}  
}
Data parallelism

- Worksharing constructs
  - Threads cooperate in doing some work
  - Thread identifiers are not used explicitly
  - Most common use case is loop worksharing
  - Worksharing constructs may not be nested
- DO/for directives are used in order to determine a parallel loop region
The for loop directive

```c
#pragma omp for [clauses]
for (iexpr ; test ; incr)
```

- Where clauses may be
  - private, firstprivate, lastprivate
  - Reduction
  - Schedule
  - Nowait

- Loop iterations must be independent
- Can be merged with parallel constructs
- Default data sharing attribute is shared
```
int i, j;
#pragma omp parallel
#pragma omp for private(j)
for (i=0; i<N; i++)
{
    for (j=0; j<N; j++)
    {
        m[i][j] = f(i, j);
    }
}
```

- **j must be declared private explicitly**
- **i is privatized automatically**
- **Implicit synchronization point at the end of for loop**
The schedule clause

- Schedule clause may be used to determine the distribution of computational work among threads
  - static, chunk; The loop is equally divided among pieces of size chunk which are evenly distributed among threads in a round robin fashion
  - dynamic, chunk; The loop is equally divided among pieces of size chunk which are distributed for execution dynamically to threads. If no chunk is specified chunk=1
  - guided; similar to dynamic with the variation that chunk size is reduced as threads grab iterations

- Configurable globally via OMP_SCHEDULE
  - i.e. setenv OMP_SCHEDULE "dynamic,4"
```cpp
#include <iostream>
#include <omp.h>

using namespace std;

int main(int argc, char **argv)
{
    int n = atoi(argv[1]);
    double *x, *y;

    x = new double [n]; for(int i=0; i<n; i++) x[i] = (double) (i+2);
    y = new double [n]; for(int i=0; i<n; i++) y[i] = (double) (i*3);

    double start = omp_get_wtime();
    #pragma omp parallel for
    for (int i=0; i<n; i++) x[i] = x[i] + y[i];
    double end = omp_get_wtime();
    printf("diff = %.16g\n", end - start);

    return 0;
}
```
Reduction clause

- Useful in the case one variable’s value is accumulated within a loop

- Using the reduction clause
  - A private copy per thread is created and initialized
  - At the end of the region the compiler safely updates the shared variable
  - Operators may be +, *, -, /, &, ^, |, &&, ||
# Reduction clause example

```cpp
#include <iostream>
#include <cmath>
#include <vector>
#include <omp.h>

using namespace std;

int main(int argc, char* argv[]) {
    // declarations
    int i, N=atoi(argv[1]);
    vector <double> A(N);
    double s;
    // calculations
    #pragma omp parallel for shared(A,N) private(i)
    for(i=0; i<N; i++)
    {
        A[i] = pow(cos((double) i),2)/3.0 - 1.0/sqrt((double) (i+1));
    }
    #pragma omp parallel for shared(A,N) private(i) reduction(+:s)
    for(i=0; i<N; i++)
    {
        s += A[i];
    }
    cout << "Total sum is s = " << s << endl;
    return 0;
}
```
Mixing MPI and OpenMP

- Hybrid architectures
  - Clusters on SMPs
  - HPC Platforms
    - IBM BlueGene (i.e. Jugene)
    - IBM P6 (i.e. Huygens)
  - Good starting point
    - Mapping of MPI on nodes (interconnection layer)
    - Multithreading with OpenMP inside SMPs