Parallel Programming: OpenMP

Xianyi Zeng
xzeng@utep.edu

Department of Mathematical Sciences
The University of Texas at El Paso. November 10, 2016.

OpenMP: Open Multi-Processing

- An *Application Program Interface* (API) for parallelism with:
  - Shared memory.
  - Multiple threads.
- Supports C/C++ and Fortran.
- Three components:
  - Compiler directives.
  - Runtime library routines.
  - Environmental variables.
- Incremental programming style:
  - Start with a serial program.
  - Insert compiler directives to enable parallelism.
- Support both fine-grain and coarse-grain parallelism.

OpenMP: Open Multi-Processing

- Accomplish parallelism exclusively through the use of threads.
- Use compiler directives to denote parallel regions: fork/join model.
- Allow nested parallel regions.
- Specifies nothing about parallel I/O.
- Relaxed-consistency: threads do not maintain exact consistency with real memory.
- To build with gcc/g++: Need to turn on the OpenMP support by adding the flag -fopenmp:

```
gcc -fopenmp -o ...
```

OpenMP: Compiler Directives

- Syntax:

```
#pragma omp directive-name [clause, ...]
```

- The compiler directive applies to one succeeding statement – use block to enclose multiple statements.
- The compiler directive is case sensitive.
- Examples:
  - `#pragma omp parallel`
    The code is to be execude by multiple threads in parallel.
  - `#pragma omp for`
    The work in a for loop to be divided among threads.
  - `#pragma omp parallel for`
    Shortcut combining the previous two.
  - `#pragma omp master`
    Only the master thread should execute the code.
  - `#pragma omp critical`
    Create a critical section.
  - `#pragma omp barrier`
    All threads pause at the barrier, until all threads execute the barrier.

OpenMP: Run-time Library Routines

- Include "omp.h", and these routines work similarly as C/C++ functions.
- Purposes:
  - Setting and querying the number of threads.
  - Quering a thread's unique identifier (thread ID).
  - Quering the thread team size.
  - Quering if in a parallel region, and at what level.
  - Setting, initializing and terminating locks.
  - Quering wall clock time.
  - Etc.
- Example 1: Get the total number of threads:

```
int Nthreads = omg_get_num_threads();
```

- Example 2: Get the thread ID for the current thread:

```
int tid = omg_get_thread_num();
```

OpenMP: Environmental Variables

- Work as other Unix environmental variables – use `export` to allow them to be passed to the executed program (process).
- All 13 environmental variable names are uppercase, and the values assigned to them are **not** case-sensitive.
- A few of them:
  - `OMP_NUM_THREADS`
    The default number of threads during the parallel region.
  - `OMP_THREAD_LIMIT`
    The number of OpenMP threads to use for the whole program.
  - `OMP_NESTED`
    Enable/disable nested parallelism.
  - `OMP_MAX_ACTIVE_LEVELS`
    The maximum number of nested active parallel regions.
  - `OMP_PROC_BIND`
    Enable/disable threads binding to processors.
  - `OMP_STACKSIZE`
    Set the size of the stack for non-master treads (default is about 2MB).
  - Etc.

Example 1: Hello World

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

int main( int argc, char** argv ) {
#pragma omp parallel
  {
    int nthreads = omp_get_num_threads();
    int tid = omp_get_thread_num();
    printf( "Hellow world from thread %d ( out of %d ).\n",
            tid, nthreads );
  }

  return 0;
}
```

Output example:

```
Hellow world from thread 1 ( out of 4 ).
Hellow world from thread 0 ( out of 4 ).
Hellow world from thread 2 ( out of 4 ).
Hellow world from thread 3 ( out of 4 ).
```

Example 2: Data parallelism

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

int main( int argc, char** argv ) {
  int a[12], b[12], c[12];

  for ( int i = 0; i < 12; i++ ) {a[i] = 11*i; b[i] = -i; }

  for ( int i = 0; i < 12; i++ ) printf( "%3d ", a[i] );
  printf( "\n" );

  for ( int i = 0; i < 12; i++ ) printf( "%3d ", b[i] );
  printf( "\n" );

#pragma omp parallel
  {
#pragma omp for
    for ( int i = 0; i < 12; i++ ) c[i] = a[i] + b[i];
  }

  for ( int i = 0; i < 12; i++ ) printf( "%3d ", c[i] );
  printf( "\n" );

  return 0;
}
```

One can combine the two directives into one:

```
#pragma omp parallel for
```

Example 3: Intermediate Variables

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main( int argc, char** argv ) {
  double height[12], width[12], weight[12];
  double area, density = 1.0;

  for ( int i = 0; i < 12; i++ ) height[i] = width[i] = 1.0*i;

#pragma omp parallel for
  for ( int i = 0; i < 12; i++ ) {
    area = height[i] * width[i]; sleep( 0.2 );
    weight[i] = area * density; }

  for ( int i = 0; i < 12; i++ ) printf( "%3.1lf ", weight[i] );
  printf( "\n" );

  return 0;
}
```

Solution: use clause to make `area` private.

Data Sharing Clauses

- `shared(list)`
  Default with the exception of the loop indices (C/C++) and locally declared variables.
- `private(list)`
  Make the variables in the list private to each thread.
- `firstprivate(list)`
  Initialize the privates with the value from the master thread.
- `lastprivate(list)`
  Copy out the last thread value into the master thread copy.
- `default(list)`
  Change the default type to some of the others.
- `threadprivate(list)`
  A `private` variable exists on the thread stack and only for the duration of the parallel region. A `threadprivate` variable, however, may exist on the heap and can exist across regions.

Example 4: Private Variables

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main( int argc, char** argv ) {
  int a[10];
  for ( int i = 0; i < 10; i++ ) a[i] = 0;
  int i, j = 10;

#pragma omp parallel for lastprivate(i) firstprivate(j)
  for ( i = 0; i < 10; i+=3 ) {
    j = j + i;
    a[i] = i*j;
  }

  printf( " Last index: %d; Value of j: %d.\n", i, j );
  for ( int i = 0; i < 10; i++ ) printf( " %2d ", a[i] );
  printf( "\n" );

  return 0;
}
```

`lastprivate` is only valid with the directives `for` and `section`.

Example 5: Sections

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main( int argc, char** argv ) {
  int a[10], b[10], c[10], d[10];
  for ( int i = 0; i < 10; i++ ) a[i] = b[i] = i;

#pragma omp parallel sections
  {
#pragma omp section
    for ( int i = 0; i < 10; i++ ) {
      printf( "Executing sum at index %d by tread %d.\n",
          i, omp_get_thread_num() );
      c[i] = a[i] + b[i]; sleep( 1 );}

#pragma omp section
    for ( int i = 0; i < 10; i++ ) {
      printf( "Executing product at index %d by thread %d.\n",
          i, omp_get_thread_num() );
      d[i] = a[i] * b[i]; sleep( 1 );}
  }

  return 0;
}
```

Each `section` is executed by one thread.

Example 6: Nested For Loops

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main( int argc, char** argv ) {
  int m[3][4], i, j;

  for ( i = 0; i < 3; i++ )
    for ( j = 0; j < 4; j++ ) m[i][j] = 0;

#pragma omp parallel for
  for ( i = 0; i < 3; i++ ) {
    sleep( 0.2 );
    for ( j = 0; j < 4; j++ ) {
      m[i][j] = i + j; sleep( 0.2 ); }
  }

  for ( int i = 0; i < 3; i++ ) {
    for ( int j = 0; j < 4; j++ )
      printf( "%2d\t", m[i][j] );
    printf( "\n" );
  }

  return 0;
}
```

- For C/C++, `j` is shared by default!
- For `Fortran`, however, both `i` and `j` are private by default.
- Correction 1: make `j` private by using

  ```
  for ( int j = 0; j < 4; j++ )
  ```

  i.e., the style supported by the C99 standard.
- Correction 2: explicitly declare `private(j)` in the compiler directive.

Example 7:An Old Friend

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main( int argc, char** argv ) {
  int a[10000], b[10000];
  for ( int i = 0; i < 10000; i++ ) {
    a[i] = i+1; b[i] = 1;}

  int sum = 0;

#pragma omp parallel for
  for ( int i = 0; i < 10000; i++ )
    sum += a[i] * b[i];

  printf( "Parallel inner product: %d.\n", sum );

  return 0;
}
```

Inconsistent and unpredictable result due to data race.

Example 7:An Old Friend

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

omp_lock_t my_lock;

int main( int argc, char** argv ) {
  int a[10000], b[10000];
  for ( int i = 0; i < 10000; i++ ) {
    a[i] = i+1; b[i] = 1;}

  int sum = 0;
  omp_init_lock(&my_lock);

#pragma omp parallel for
  for ( int i = 0; i < 10000; i++ ) {
    int loc_prod = a[i] * b[i];
    omp_set_lock(&my_lock);
    sum += loc_prod;
    omp_unset_lock(&my_lock);
  }

  omp_destroy_lock(&my_lock);
  printf( "Parallel inner product: %d.\n", sum );

  return 0;
}
```

Use the run-time lock/unlock routines to avoid data race.

Example 7:An Old Friend

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main( int argc, char** argv ) {
  int a[10000], b[10000];
  for ( int i = 0; i < 10000; i++ ) {
    a[i] = i+1; b[i] = 1;}

  int sum = 0;

#pragma omp parallel for
  for ( int i = 0; i < 10000; i++ ) {
    int loc_prod = a[i] * b[i];

#pragma omp critical
    sum += loc_prod;

  }

  printf( "Parallel inner product: %d.\n", sum );

  return 0;
}
```

Use the compiler directive to indicate a critical section.

Example 7:An Old Friend

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main( int argc, char** argv ) {
  int a[10000], b[10000];
  for ( int i = 0; i < 10000; i++ ) {
    a[i] = i+1; b[i] = 1;}

  int sum = 0;

#pragma omp parallel for reduction(+:sum)
  for ( int i = 0; i < 10000; i++ )
    sum += a[i] * b[i];

  printf( "Parallel inner product: %d.\n", sum );

  return 0;
}
```

Use the `reduction` clause.

Reduction Operators

- `+`
  Compute the sum, initialize with zero.
- `max`
  Compute the maximum, initialize with the least number possible.
- `min`
  Compute the minimum, initilize with the largest number possible.
- Bit operations, logical operations, etc.

Example 8: Prime Number Counter

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

int main( int argc, char** argv ) {
  int n = 200000;
  int not_primes = 0, i, j;

#pragma omp parallel for private(j) \
            reduction(+: not_primes)
  for ( i = 2; i <= n; i++ ) {
    for ( j = 2; j < i; j++ )
      if ( i % j == 0 ) {
        not_primes++;
        break;
      }
  }

  printf( "Primes: %d.\n", n - not_primes );
  return 0;
}
```

Resources

- The LLNL tutorial on OpenMP

  https://computing.llnl.gov/tutorials/openMP/
- The XSEDE workshop.