Some More on OpenMP

09/04/2020

Some content from Bill Gropp, University of Illinois

OpenMP Can Break Your Program

- Placing pragmas before loops can be harmful
  - Can make program incorrect
    - If you accidentally share variables that should not be shared, you can create race conditions
  - Can make program slow
    - Don’t want to decompose control processing
    - Want to decompose *data* processing

OpenMP Can Break Your Program

A really simple stencil example

```
int i, j;
double sum = 0, diff = 0;
for (int i = 1; i < (N – 1); i++)
    for (int j = 1; j < (N – 1); j++) {
   sum = old[i-1][j] + old[i+1][j] + old[i][j-1] + old[i][j+1];
   new[i][j] = sum/4.0;
   diff = MAX(diff, new[i][j] – old[i][j]);
```

General Approach to Parallel Code

- Find regions that only one thread can be in at a time (“critical”)
- Find regions of code that can only be executed by one thread (“single” or “master” pragmas)
- Declare statements (not blocks) that must be atomic, potentially with hardware support
- Declare variables with thread-specific semantics (“private”, “reduction”)

Shared Variables

- The default state of any variable declared before the #pragma omp parallel statement
- Stored in main memory
- Accessed and updated by each thread
- Can result in race conditions and false sharing

Private Variables

- Each thread holds a copy of the entire object
- For instance, if a double* is private, all threads hold the entire array and copy all data in the array
- Very expensive to copy and can run out of memory for large objects
- Only want to use private variables for small objects (or single variables like ints, doubles)
- If not necessary to use private, use shared variables and find other ways to avoid false sharing :
  - local variables inside for loops
  - reduction operations
  - split up loops to avoid shared cache lines

Back to Race Conditions

- Let's go back to finding a maximum (or minimum) from a list

```
for (i=0; i<n; i++) {
   if (x[i] > maxval) {
      maxval = x[i];
      maxloc = i; }
}
```

Using locks

- Around each race condition:
  omp_set_lock(&min_val_lock);
  omp_unset_lock(&min_val_lock);

- Will fix race conditions, correct code

- Expensive, serializes code (only one thread can execute at a time), but adds additional cost of locking and unlocking

Updating same value

- In case of finding min/max in list, all threads may be accessing and changing the same values — Race Condition
- Several OpenMP methods for this:
  - **#pragma omp atomic :** only one thread can execute following statement (not block) at a time
  - **#pragma omp critical** : only one thread can execute following block of code at a time
- Atomic may be faster, depends on hardware

Atomic Operations

- Can read, write, and perform updates with only a single thread at a time
- Expensive, serializes code, but avoids more expensive locks

```
for(i = 0; i < 10; i++)
  {
    #pragma omp atomic read
    temp[i] = x[f(i)];

    #pragma omp atomic write
    x[i] = temp[i]*2;

    #pragma omp atomic update
    x[i] *= 2;
  }
```

Critical Section

- Have entire block of code performed by only a single thread at a time
- Still serializes code, can be more expensive than atomic operations, but more versatile

```
#pragma omp critical
for(i = 0; i < 10; i++)
{
    temp[i] = x[f(i)];
    x[i] = temp[i]*2;
    x[i] *= 2;
}
```

Re-Structure the Program to Avoid Race Conditions

- To find maximum, search for local maximum and have each thread save local maximum.
- Step through local maximums with a single thread to find max of all threads
  - #pragma omp master
  - End #pragma omp parallel and step through results after parallel block of code

Master Region

- Why use master region? Can access OpenMP routines

```
#pragma omp parallel
{
    #pragma omp master
    {
        int n_threads = omp_get_num_threads();
        …
    }
}
```

Reductions

- Programmer indicates variable used for reduction and type of reduction
- #pragma omp parallel for reduction(<type>, <variable>)
- Many different types, including
  - Arithmetic reductions : +, *, -, min, max
  - Logical reductions :
    - `In C:  &  &&  |  ||  ^`
    - `In Fortran : .and. .or. .eqv. .neqv. .iand. .ior. .ieor.`

Loop Scheduling

- The method #pragma omp parallel for splits up the for loop across threads
  - Splitting determined by OpenMP
  - Programmer does not need to change the loop
- Three choices for loop scheduling:
  - Static : cheapest, determined at compile time
  - Dynamic : determined at runtime
  - Guided : special instance of dynamic that attempts to reduce overhead

Loop Scheduling

- Schedule Routine:
  ```
  #pragma omp parallel for schedule(kind[,chunksize])
  for (int i…)
  ```

- Example:
  ```
  #pragma omp parallel for schedule(guided, 100)
  for (i = 0; i<n; i++)
      c[i] = a[i]
  ```

Static Loop Scheduling

```
schedule(static):
****************
                ****************
                                ****************
                                                ****************
```

```
schedule(static, 4):
****            ****            ****            ****
    ****            ****            ****            ****
        ****            ****            ****            ****
            ****            ****            ****            ****
```

```
schedule(static, 8):
********                        ********
        ********                        ********
                ********                        ********
                        ********                        ********
```

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Dynamic Loop Scheduling

```
schedule(dynamic):
*   ** **  * * *  *      *  *    **   *  *  * *       *  *   *
  *       *     *    * *     * *   *    *        * *   *    *
 *       *    *     * *   *   *     *  *       *  *  *  *  *   *
   *  *     *    * *    *  *    *    *    ** *  *   *     *   *
```

```
schedule(dynamic, 1):
    *    *     *        *   *    * *  *  *         *  * *  * *
*  *  *   * *     *  * * *    * *      *   ***  *   *         *
 *   *  *  *  *    ** *    *      *  *  * *   *  *   *   *
  *    *     *  **        *  * *    *          *  *    *  * *  *
```

```
schedule(dynamic, 4):
            ****                    ****                    ****
****            ****    ****            ****        ****
    ****            ****    ****            ****        ****
        ****                    ****            ****
```

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Guided Loop Scheduling

```
schedule(guided):
                          *********                           *
                ************                        *******   ***
                                        *******                   *
*****************                               *****        **     *
```

```
schedule(guided, 2):
                ************                        ****      **
                                        *******           ***     **
                          *********
*****************                               *****        **     **
```

```
schedule(guided, 4):
                                        *******
                ************                        ****     ****
                          *********
*****************                               *****     ****     ***
```

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Runtime Loop Scheduling

- Set environment variable OMP_SCHEDULE to a schedule type
- Or use the function omp_set_schedule()
- Use runtime schedule:

```
#pragma omp parallel for schedule(runtime,<chunksize>)
```

- The type of loop schedule can then be changed after program is compiled

STREAM Benchmark

- Supports OpenMP, i.e. copy loops like:

```
#pragma omp parallel for
for (j = 0; j < STREAM_ARRAY_SIZE; j++)
    c[j] = a[j]

Run with:
export OMP_NUM_THREADS=<num_threads>
./stream
```

How does this perform? How is the loop scheduled? We will explore this further in class.