OpenMP - IV

Diego Fabregat-Traver and Prof. Paolo Bientinesi

HPAC, RWTH Aachen
fabregat@aices.rwth-aachen.de

WS15/16

OpenMP

References

- Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press, 2007.

  B. Chapman, G. Jost, R. van der Pas.

Today's agenda

- Solve homework from last week
- Complete core OpenMP features
- Clarify pending questions
- Pitfalls and common mistakes
- Performance problems
- Tasks, memory consistency, and flush

Synchronization (cont.)

Synchronization - Locks

- Low level constructs for mutual exclusion
- Flexible, allow greater control over how to do synchronization than critical/atomic

- Lock variables of type `omp_lock_t`
- Related routines:
  - `omp_init_lock`, `omp_destroy_lock`
  - `omp_set_lock`, `omp_unset_lock`
  - `omp_test_lock`

Synchronization - Locks

```
int i, ctr = 0;
omp_lock_t l;

omp_init_lock(&l);
#pragma omp parallel for
for( i = 0; i < 1000; i++ )
{
   omp_set_lock(&l);
   ctr = ctr + 1;
   omp_unset_lock(&l);
}
omp_destroy_lock(&l);
```

Exercise (histogram.c)

- Parallelize the `for` loop in the following code. Use an array of locks to fine-grained mutual exclusion on each individual entry of the `hist` array.

```
int main( int argc, char *argv[] )
{
    int i, nsamples, nbuckets;
    int *hist, *data;

    // Allocate arrays
    [...]
    // Initialize histogram and data
    [...]

    // Fill in the histogram
    for ( i = 0; i < nsamples; i++ )
        hist[ data[i] ] += 1;

    // Free resources
    [...]

    return 0;
}
```

Hints:

- `omp_init_lock`, `omp_destroy_lock`
- `omp_set_lock`, `omp_unset_lock`

Synchronization - Locks - `omp_test_lock`

- The following code allows the threads to perform some other useful work while waiting for access to the critical region.

```
omp_lock_t l;
omp_init_lock(&l);

#pragma omp parallel
{
  [...]
  while (!omp_test_lock(&l))
  {
    do_some_other_work();
  }
  do_critical_work()
  omp_unset_lock(&l);
  [...]
}
omp_destroy_lock(&l);
```

Master and Single

Master and Single

- The `master` and `single` constructs guarantee that code in a parallel region is executed by one single thread
- Initialization of data, I/O, ...

- Both constructs apply to a structured block
- Differences:
  - The `master` construct guarantees the block is executed only by the master thread
    The `single` construct guarantees the block is executed by any one and only one thread
  - A `single` construct has an implicit barrier at the end
    A `master` construct does not
  - Efficiency may differ. Application and implementation dependent.

Master

06.master.c

```
#pragma omp parallel num_threads(2)
{
  int id = omp_get_thread_num();
  #pragma omp master
  {
    printf("[%d] Executed only by the master thread\n", id);
  }
  printf("[%d] Executed by all threads\n", id);
}
```

Single

06.single.c

```
#pragma omp parallel num_threads(2)
{
  int id = omp_get_thread_num();
  #pragma omp single
  {
    printf("[%d] Executed by only one thread\n", id);
  }
  printf("[%d] Executed by all threads\n", id);
}
```

Exercise

- Parallelize the body of the *while* loop in the following code. Use the `for` construct to parallelize the *for* loop. Use the `master/single` construct to serialize the portions of the code that should be executed by only one thread. Produce two versions of your code.

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

#define N 1000

int main( void ) {
    int iter = 0;
    float buffer_in[N*N], buffer_out[N*N];
    while (1) {
        iter++;
        printf( "Timestep %d\n", iter );
        read_input( buffer_in );
        for (i = 0; i < N; i++)
           process_signal( &buffer_in[i*N], &buffer_out[i*N] );
        write_output( buffer_out );
    }
}
```

Hints:

- `#pragma omp parallel`
- `#pragma omp master`
- `#pragma omp single`
- Challenge:
  - Pay special attention to the subtle differences between `master` and `single`

Sections

Sections

- The `sections` construct allows different threads to carry out different tasks (task parallelism)
- Non-iterative worksharing construct
- It consists of a pair of directives: `sections` and `section`
- `sections` indicates the start of the construct (enclosing the region with the multiple tasks)
- `section` marks each different section/task (a structured block)

Sections

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    computation_x()
    #pragma omp section
    computation_y()
    #pragma omp section
    computation_z()
  }
}
```

- Think of a queue of executable blocks of code (each individual `section`)
- Theads grab one block at a time from the queue and execute it, until no more blocks remain
- Each block is executed only once
- See `09.sections.c`

Exercise

- Consider the following sequence of operations. Assuming the only dependencies among the operations are given by their input and output variables, write a piece of OpenMP code that exposes as much parallelism as possible using the `sections` and `section` constructs.

```
mx = mean( x );
my = mean( y );
fout = f( mx, my );
gout = g( mx, my );
final = summary( fout, gout );
```

Hints:

- `#pragma omp parallel`
- `#pragma omp sections`
- `#pragma omp section`
- Hint:
  - You may need multiple `sections` regions.

Sections and nested parallelism

```
int main( void )
{
  [...]
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      parallelizable()
      #pragma omp section
      sequential()
    }
  }
  [...]
}
```

```
int parallelizable( void )
{
  [...]
  #pragma omp parallel
  {
    #pragma omp for
    for( i = 0; i < HUGE; i++ )
    {
        [...]
    }
  }
  [...]
}
```

Recap

Randomness pi montecarlo (HW 8.1)

Sequence of random numbers:

```
[0]: 0.000000
[1]: 0.000985
[2]: 0.041631
[3]: 0.176643
[4]: 0.364602
[5]: 0.091331
[6]: 0.092298
[7]: 0.487217
[8]: 0.526750
[9]: 0.454433
```

Randomness pi montecarlo (HW 8.1)

Sequence of random numbers:

```
[0]: 0.000000
[1]: 0.000985
[2]: 0.041631
[3]: 0.176643
[4]: 0.364602
[5]: 0.091331
[6]: 0.092298
[7]: 0.487217
[8]: 0.526750
[9]: 0.454433
```

Sequential run:

1. (0.000000, 0.000985)
2. (0.041631, 0.176643)
3. (0.364602, 0.091331)
4. ...

Randomness pi montecarlo (HW 8.1)

Sequence of random numbers:

```
[0]: 0.000000
[1]: 0.000985
[2]: 0.041631
[3]: 0.176643
[4]: 0.364602
[5]: 0.091331
[6]: 0.092298
[7]: 0.487217
[8]: 0.526750
[9]: 0.454433
```

Sequential run:

1. (0.000000, 0.000985)
2. (0.041631, 0.176643)
3. (0.364602, 0.091331)
4. ...

Possible parallel run:

- Thread 1
  1. (,)
  2. (,)
  3. ...
- Thread 2
  1. (,)
  2. (,)
  3. ...

Randomness pi montecarlo (HW 8.1)

Sequence of random numbers:

```
[0]: 0.000000
[1]: 0.000985
[2]: 0.041631
[3]: 0.176643
[4]: 0.364602
[5]: 0.091331
[6]: 0.092298
[7]: 0.487217
[8]: 0.526750
[9]: 0.454433
```

Sequential run:

1. (0.000000, 0.000985)
2. (0.041631, 0.176643)
3. (0.364602, 0.091331)
4. ...

Possible parallel run:

- Thread 1
  1. (0.00000,)
  2. (,)
  3. ...
- Thread 2
  1. (,)
  2. (,)
  3. ...

Randomness pi montecarlo (HW 8.1)

Sequence of random numbers:

```
[0]: 0.000000
[1]: 0.000985
[2]: 0.041631
[3]: 0.176643
[4]: 0.364602
[5]: 0.091331
[6]: 0.092298
[7]: 0.487217
[8]: 0.526750
[9]: 0.454433
```

Sequential run:

1. (0.000000, 0.000985)
2. (0.041631, 0.176643)
3. (0.364602, 0.091331)
4. ...

Possible parallel run:

- Thread 1
  1. (0.00000,)
  2. (,)
  3. ...
- Thread 2
  1. (0.00098,)
  2. (,)
  3. ...

Randomness pi montecarlo (HW 8.1)

Sequence of random numbers:

```
[0]: 0.000000
[1]: 0.000985
[2]: 0.041631
[3]: 0.176643
[4]: 0.364602
[5]: 0.091331
[6]: 0.092298
[7]: 0.487217
[8]: 0.526750
[9]: 0.454433
```

Sequential run:

1. (0.000000, 0.000985)
2. (0.041631, 0.176643)
3. (0.364602, 0.091331)
4. ...

Possible parallel run:

- Thread 1
  1. (0.00000,)
  2. (,)
  3. ...
- Thread 2
  1. (0.00098, 0.04163)
  2. (,)
  3. ...

Randomness pi montecarlo (HW 8.1)

Sequence of random numbers:

```
[0]: 0.000000
[1]: 0.000985
[2]: 0.041631
[3]: 0.176643
[4]: 0.364602
[5]: 0.091331
[6]: 0.092298
[7]: 0.487217
[8]: 0.526750
[9]: 0.454433
```

Sequential run:

1. (0.000000, 0.000985)
2. (0.041631, 0.176643)
3. (0.364602, 0.091331)
4. ...

Possible parallel run:

- Thread 1
  1. (0.00000,)
  2. (,)
  3. ...
- Thread 2
  1. (0.00098, 0.04163)
  2. (0.17664,)
  3. ...

Randomness pi montecarlo (HW 8.1)

Sequence of random numbers:

```
[0]: 0.000000
[1]: 0.000985
[2]: 0.041631
[3]: 0.176643
[4]: 0.364602
[5]: 0.091331
[6]: 0.092298
[7]: 0.487217
[8]: 0.526750
[9]: 0.454433
```

Sequential run:

1. (0.000000, 0.000985)
2. (0.041631, 0.176643)
3. (0.364602, 0.091331)
4. ...

Possible parallel run:

- Thread 1
  1. (0.00000, 0.36460)
  2. (,)
  3. ...
- Thread 2
  1. (0.00098, 0.04163)
  2. (0.17664,)
  3. ...

Randomness pi montecarlo (HW 8.1)

Sequence of random numbers:

```
[0]: 0.000000
[1]: 0.000985
[2]: 0.041631
[3]: 0.176643
[4]: 0.364602
[5]: 0.091331
[6]: 0.092298
[7]: 0.487217
[8]: 0.526750
[9]: 0.454433
```

Sequential run:

1. (0.000000, 0.000985)
2. (0.041631, 0.176643)
3. (0.364602, 0.091331)
4. ...

Possible parallel run:

- Thread 1
  1. (0.00000, 0.36460)
  2. (0.09133,)
  3. ...
- Thread 2
  1. (0.00098, 0.04163)
  2. (0.17664,)
  3. ...

Randomness pi montecarlo (HW 8.1)

Sequence of random numbers:

```
[0]: 0.000000
[1]: 0.000985
[2]: 0.041631
[3]: 0.176643
[4]: 0.364602
[5]: 0.091331
[6]: 0.092298
[7]: 0.487217
[8]: 0.526750
[9]: 0.454433
```

Sequential run:

1. (0.000000, 0.000985)
2. (0.041631, 0.176643)
3. (0.364602, 0.091331)
4. ...

Possible parallel run:

- Thread 1
  1. (0.00000, 0.36460)
  2. (0.09133,)
  3. ...
- Thread 2
  1. (0.00098, 0.04163)
  2. (0.17664, 0.09229)
  3. ...

Randomness pi montecarlo (HW 8.1)

Sequence of random numbers:

```
[0]: 0.000000
[1]: 0.000985
[2]: 0.041631
[3]: 0.176643
[4]: 0.364602
[5]: 0.091331
[6]: 0.092298
[7]: 0.487217
[8]: 0.526750
[9]: 0.454433
```

Sequential run:

1. (0.000000, 0.000985)
2. (0.041631, 0.176643)
3. (0.364602, 0.091331)
4. ...

Possible parallel run:

- Thread 1
  1. (0.00000, 0.36460)
  2. (0.09133, 0.48721)
  3. ...
- Thread 2
  1. (0.00098, 0.04163)
  2. (0.17664, 0.09229)
  3. ...

Parallelization of different loops (HW 8.2)

```
#pragma omp parallel
{
  for (i = 0; i < m; i++)
    #pragma omp for
    for (j = 0; j < n; j++)
      [...]
}
```

VS

```
  for (i = 0; i < m; i++)
    #pragma omp parallel for
    for (j = 0; j < n; j++)
      [...]
```

Parallelization of different loops (HW 8.2)

- Computation is distributed in the same way
- Top version incurs in much less synchronization overhead

Version	nths	Time (s)	Speedup	Time (s)	Speedup
Sequential	1	0.43	1.00	56.67	1.00
Par. outermost loop	8	0.22	1.92	25.93	2.19
Par. innermost (top)	8	1.06	0.41	44.07	1.29
Par. innermost (bot)	8	0.49	0.88	29.33	1.93

Table: Left: $n = 100$; right: $n = 1000$.

Reductions

```
double sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for ( int i = 0; i < NUM_STEPS; i++ ) {
  sum += f(i) // accumulate to sum
}
```

```
double shared_sum = 0.0;
#pragma omp parallel
{
  double local_sum = 0.0; // Initializer (0.0 for sum of doubles)
  #pragma omp for // Same computation on the local accumulator
  for ( int i = 0; i < NUM_STEPS; i++ ) {
    local_sum += f(i) // accumulate to your local_sum
  }
  #pragma omp critical // reduce partial sums using the
  shared_sum += local_sum; // combiner function (+ in this case)
}
```

Summary

- We have seen the core of what most people uses from OpenMP
- `#pragma omp parallel`
  - Indicates a portion of code to be parallelized
  - Creates team of threads
  - Replicates work: distribute manually or via worksharing constructs
  - Clauses to control data-sharing attributes, number of threads, etc.
- `#pragma omp for`
  - Work-sharing construct
  - Distributes iterations among threads
  - Clauses to control data-sharing attributes, load balancing, etc.

Summary

- Synchronization and mutual exclusion constructs and tools
  - `barrier`, `critical`, `atomic`
  - `lock`
  - `master`, `single`
- `sections`, `section`
  - Work-sharing construct
  - Task parallelism
- Supporting library routines and environment variables
  - `void omp_set_num_threads( int n )`
  - `int omp_get_num_threads( void )`
  - `int omp_get_thread_num( void )`
  - `OMP_SCHEDULE`, `OMP_NUM_THREADS`, ...

Pitfalls and common mistakes

Pitfalls

- Race conditions
- Deadlocks

Pitfalls - Race conditions

Some common sources of problems

- Loop-carried dependencies
- Missing `private`
- Missing `critical`/`atomic`
- No implicit barrier after `master`
- Incorrect use of `nowait`

Pitfalls - Loop construct and nowait

Different loop regions with the same schedule and iteration count, even inside the same parallel region, may distribute iterations among threads differently.

```
// INCORRECT
#pragma omp parallel
{
  #pragma omp for nowait
  for (i = 0; i < n; i++)
    a[i] = a[i] + i;
  #pragma omp for
  for (i = 0; i < n; i++)
    b[i] = 2 * a[i];
}
```

Pitfalls - Loop construct and nowait

Exception:

- Loops belong to the same parallel region
- Both loops use `static` scheduling
- Both loops run for the same number of iterations
- Both specify the same `chunk_size` or do not specify it at all
- (none is associated with the SIMD construct)

If all of the above holds, then, and only then, can we assume that the same iteration numbers are assigned to the same threads in both loops.

Pitfalls - Loop construct and nowait

```
  // CORRECT :)
  #pragma omp parallel
  {
    #pragma omp for nowait schedule(static)
    for (i = 0; i < n; i++)
      a[i] = a[i] + i;
    #pragma omp for schedule(static)
    for (i = 0; i < n; i++)
      b[i] = 2 * a[i];
}
```

Pitfalls - Deadlocks

From the OpenMP 4.0 specifications (Section 2.7):

- *Each worksharing region must be encountered by all threads in a team or by none at all, unless cancellation has been requested for the innermost enclosing parallel region*.
- *The sequence of worksharing regions and barrier regions encountered must be the same for every thread in a team.*

Otherwise, the behavior is unspecified, leading to deadlocks or unexpected behavior.

- See `10.worksharing-order.c`

Pitfalls - Deadlocks

From the OpenMP 4.0 specifications (Section 2.12.3):

- *Each barrier region must be encountered by all threads in a team or by none at all , unless cancellation has been requested for the innermost enclosing parallel region.*
- *The sequence of worksharing regions and barrier regions encountered must be the same for every thread in a team.*

In general...

- One has to double check many such restrictions in the specifications:
  - Barrier not allowed in a worksharing construct, critical section, master, ...
  - Master not allowed in a worksharing construct

Performance issues

Performance issues - Examples

- False sharing
- Named critical sections

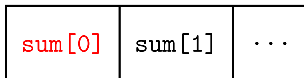
False sharing

```
 double local_sum[NUM_THREADS];
 double step = 1.0/NUM_STEPS, sum, pi;

 // init local_sum
 #pragma omp parallel num_threads(NUM_THREADS)
 {
   int id = omp_get_thread_num();
   int nths = omp_get_num_threads();
   double x_i;

   for ( i = id; i < NUM_STEPS; i += nths )
   {
     x_i = (i + 0.5) * step;
     local_sum[id] = local_sum[id] + 4.0 / (1.0 + x_i * x_i);
   }
}

sum = 0.0;
for ( i = 0; i < NUM_THREADS; i++ )
  sum += local_sum[ i ];
pi = sum * step;
```

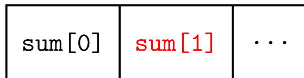
False sharing

# threads	Time (s)
1	1.97
2	2.88
4	3.54

Core 0



Core 1



- Independent elements (`sum[0]` and `sum[1]`) lie in the same cache line. Each write by one thread/core will invalidate the other's line. This increases traffic, idle time, and thus scalability.
- When promoting scalars to arrays to store values local to each thread in separate variables (addresses), the array elements will share cache lines. A possible solution is to pad the arrays.

False sharing

```
#DEFINE PAD 8
int main( void )
{
  double local_sum[NUM_THREADS][PAD];
  // other declarations
  for ( int i = 0; i < NUM_THREADS; i++ )
    local_sum[ i ][0] = 0.0;
  #pragma omp parallel num_threads(NUM_THREADS)
  {
    [...]
    for ( i = id; i < NUM_STEPS; i += nths ) {
      x_i = (i + 0.5) * step;
      local_sum[id][0] = local_sum[id][0] + 4.0 / (1.0 + x_i * x_i);
    }
  }
  sum = 0.0;
  for ( i = 0; i < NUM_THREADS; i++ )
    sum += local_sum[ i ][0];
  pi = sum * step;
  [...]
  return 0;
}
```

False sharing

# threads	False Sh.	Padding	Critical	Reduction
1	1.97	1.97	2.33	1.97
2	2.88	0.98	29.84	0.98
4	3.54	0.49	62.70	0.49

Named critical regions

```
#pragma omp critical [(name)]
```

- *All critical regions without a name are considered to have the same unspecified name*.
- *A thread waits at the beginning of a critical region until no thread is executing a critical region with the* **same name**.

Named critical regions

- An unnamed critical section provide exclusive access with respect to all unnamed critical sections in the entire program
- Sometimes, different critical regions protect the access to different data
- These cases may lead to reduced parallelism/scalability
- Named critical sections help solving this problem

Named critical regions

```
int min = INT_MAX,
    max = INT_MIN;
#pragma omp parallel for
for ( i = 0; i < n; i++ )
{
    if ( a[i] < min )
    {
        #pragma omp critical (min_lock)
        if ( a[i] < min )
            min = a[i];
    }
    if ( a[i] > max )
    {
        #pragma omp critical (max_lock)
        if ( a[i] > max )
            max = a[i];
    }
}
```