OpenMP: Open Multi-Processing

Chris Kauffman

Last Updated:
Tue Mar 29 09:44:29 AM CDT 2022

Logistics

Today

OpenMP for shared memory machines

Reading

- Grama 7.10 (OpenMP)
- OpenMP Tutorial at Laurence Livermore National Labs

Next Lecture

- A bit more OpenMP
- Java Threads

Next Week

Thu: Mini-Exam 4

OpenMP: High-level Shared Memory Parallelism

- OpenMP = Open Multi-Processing
- A standard, implemented by various folks, compiler-makers
- Targeted at shared memory machines: multiple processing elements sharing memory
- Specify parallelism in code with
  - Some function calls: *which thread number am I?*
  - Directives: *do this loop using multiple threads/processors*
- Can orient program to work without need of additional processors - direct serial execution
- OpenMP targets multiple processors, new relative OpenACC which targets “accelerators” like GPUs with same ideas
- The *easiest* parallelism you’ll likely get in C / C++ / Fortran

#pragma in C

> *The '#pragma' directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself.*
> – *GCC Manual*

- Similar in to Java's annotations (`@Override`)
- Indicate meta-info about about code

```
printf("Normal execution\n");

#pragma do something special below
normal_code(x,y,z);
```

- Several other pragmas supported by `gcc` including
  - `once`: include a header file once only
  - `poison`: if a poisoned identifier is used, cause an error
  - `dependency`: warn if another file is newer than this one

OpenMP Basics

```
#pragma omp parallel
single_parallel_line();

#pragma omp parallel
{
  parallel_block();
  with_multiple(statements);
  done_in_parallel();
}
```

- Pragmas indicate a single line or block should be done in parallel.
- Examine `omp_basics.c`

Compiler Support for OpenMP

- Most other modern compilers have support for OpenMP
- GCC, CLang/LLVM, Intel C/C++ Compiler, MS Visual Studio, Portland Group / NVidia tools - all support OpenMP in various ways
- GCC supports OpenMP with appropriate options

```
>> gcc omp_basics.c             # no parallelism
>> gcc omp_basics.c -fopenmp    # enable parallelism
```

- OpenMP was introduced in the mid 90's and has expanded and added features which are available depending on platform

GCC Version	4.2	4.4	4.7	4.9	6.0	9.0
OpenMP Version	2.5	3.0	3.1	4.0	4.5	5.0

Hints at OpenMP Implementation

- OpenMP ≈ coarse-grained parallelism
- PThreads ≈ fine-grained parallelism
- From libGOMP Documentation (OMP library in GCC)

```
OMP CODE
  #pragma omp parallel
  {
    body;
  }
BECOMES
  void subfunction (void *data){
    use data;
    body;
  }
  setup data;
  GOMP_parallel_start (subfunction, &data, num_threads);
  subfunction (&data);
  GOMP_parallel_end ();
```

Not exactly a source transformation, but OpenMP can leverage many existing pieces of Posix Threads libraries.

Grama Sample Translation: OpenMP → PThreads

```
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

Sample OpenMP program

```
int a, b;
main() {
    // serial segment

    for (i = 0; i < 8; i++)
        pthread_create (......., internal_thread_fn_name, ...);

    for (i = 0; i < 8; i++)
        pthread_join (.......);

    // rest of serial segment

}

void *internal_thread_fn_name (void *packaged_argument) [
    int a;

    // parallel segment
}
```

Code inserted by the OpenMP compiler

Corresponding Pthreads translation

Figure 7.4 A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

OpenMP Thread Identification

- OpenMP divides computation into *threads*
- Nearly identical model to PThreads approach BUT not always implemented via PThreads (`icc` may use Intel Thread Building Blocks)
- Threads execute concurrently / in parallel, can have private data, shared data
- OpenMP provides basic id / environment functions for threads and synchronization constructs

```c
#pragma omp parallel
{
  int thread_id = omp_get_thread_num();
  int num_threads = omp_get_num_threads();
  int work_per_thread = total_work / num_threads;
  ...;
}
```

Specifying Number of Threads

```
#pragma omp parallel                // Default # threads based on system config
{
  run_with_max_num_threads();
}

if (argc > 1) {                     // Number of threads based on command line
  omp_set_num_threads( atoi(argv[1]) );
}
#pragma omp parallel
{
  run_with_current_num_threads();
}

#pragma omp parallel num_threads(2) // Number of threads as part of pragma
{
  run_with_two_threads();
}

int NT = 4;                         // Number of threads from program variable
#pragma omp parallel num_threads(NT)
{
  run_with_four_threads();
}

>> OMP_NUM_THREADS=4 ./a.out       // Set default via environment variable
```

Tricky Memory Issues Abound

Program Fragment

```c
// omp_shared_variables.c

int id_shared=-1;
int numThreads=0;

#pragma omp parallel
{
  id_shared = omp_get_thread_num();
  numThreads = omp_get_num_threads();
  printf("A: Hey from thread %d / %d\n",
         id_shared, numThreads);
}

printf("\n");

#pragma omp parallel
{
  int id_private = omp_get_thread_num();
  numThreads = omp_get_num_threads();
  printf("D: Hey from thread %d / %d\n",
         id_private, numThreads);
}
```

Possible Output

```
A: Hello from thread 2 of 4
A: Hello from thread 3 of 4
A: Hello from thread 0 of 4
A: Hello from thread 0 of 4

D: Hello from thread 1 of 4
D: Hello from thread 3 of 4
D: Hello from thread 0 of 4
D: Hello from thread 2 of 4
```

Lessons

- OpenMP Threads share memory just like PThreads including heap, globals, any stack vars in master thread
- Threads share any stack variables NOT in parallel blocks
- Thread variables are **private** if declared inside parallel blocks
- Pragmas can be used to create private copies of otherwise shared variables
- Take care with shared variables: easy to accidentally share variables as programming language scope does not make sharing as obvious

Exercise: Pi Calc via OpenMP

Examine:
`https://cs.umn.edu/~kauffman/5451/picalc_omp_reduction.c`

Questions

- Contrast the structure of the program with PThreads version
- How is the number of threads used to run determined?
- What is the business with `reduction(+: total_hits)`?
- Can variables like `points_per_thread` be moved out of the parallel block?
- Do you expect speedup for this computation?

Answers: Pi Calc via OpenMP

- Contrast the structure of the program with PThreads version
  Shorter and sweeter, no need for auxiliary function, casting, loops to create/join threads.
- How is the number of threads used to run determined?
  From the command line and set via the function `omp_set_num_threads()`
- What is the business with `reduction(+: total_hits)`?
  Performs a reduction on shared variable `total_hits`: correct results + performance; more in a moment...
- Can variables like `points_per_thread` be moved out of the parallel block?
  `points_per_thread` and `num_threads` can be shared; `thread_id` and `state` should NOT be shared.
- Do you expect speedup for this computation?
  Yes - get nearly linear speedup and correct results with less effort than PThreads version.

Exercise: Placement of Variables vs Runtime

Analyze these two examples and explain the timing difference

```
// (A) picalc_omp_reduction.c
#pragma omp parallel ...
{
  unsigned int state =
    123456789 * thread_id;
  ...
  double x =
    ((double) rand_r(&state))...

TIMING
>> time a.out 75000000 4
npoints: 75000000
hits:    58910475
pi_est:  3.141892

real    0m0.291s
user    0m1.125s
sys     0m0.004s
```

```
// (B) picalc_omp_rand_contention.c:
unsigned int state =
  123456789;
#pragma omp parallel...
{
  ...
  double x =
    ((double) rand_r(&state))...

TIMING
>> time -p a.out 75000000 4
npoints: 75000000
hits:    58910901
pi_est:  3.141915

real    0m1.200s
user    0m4.285s
sys     0m0.001s
```

Answers: Placement of Variables vs Runtime

- (A) `picalc_omp_reduction.c` places the `state` variable within the `parallel` region - becomes **thread private**
- (B) `picalc_omp_rand_contention.c` places it `state` outside so it is a **shared variable** among threads
- Each call to `rand_r()` must alter `state` so there is memory contention around it

Note on `rand()`

- `rand_r()` is **reentrant** and **thread-safe**
  - When programming in multi-threaded contexts look for these qualities
  - *Note:* When calling `rand_r()` in multiple threads with the *same* `state` variable, likely to lose "randomness"
- `rand()` is another matter...
  - Generates random numbers a la `int r = rand();`
  - Uses a "hidden" global variable to track generator state
  - For many moons, was NOT thread safe
  - Most Linux / GLIBC implementations are thread safe, but...
  - Likely use a mutex to protect the state variable slowing things down considerably...

```
>> time ./picalc_omp_rand 75000000 1
...
real    0m1.439s
...

>> time ./picalc_omp_rand 75000000 4
...
real    1m3.403s
...
```

Reductions in OpenMP

`omp_picalc.c` used a `reduction()` clause

```
//                operation --+  +-- variable
//                            V  V
#pragma omp parallel reduction(+: total_hits)
{
  ...;
  total_hits++;
}
```

- Shared var `total_hits` is updated "properly" and reasonably efficiently
  - May exploit the fact that addition is transitive - can be done in any order
  - Likely to introduce a private version of reduction variable for each thread then reduce over threads at the end
  - Alternatively may utilize a mutex or hardware atomic ops
- Most other arithmetic ops available
- Statement of **policy** rather than **mechanism**

OpenMP Atomic Pragmas

```
#pragma omp parallel
{
  ...;
  #pragma omp atomic
  total_hits++;
}
```

- Use atomic hardware instruction available
- Restricted to single operations, usually arithmetic
- No hardware support → compilation problem

```
#pragma omp atomic
printf("woot"); // compile error
```

Alternative: Critical Block

```
#pragma omp parallel
{
  ...;
  #pragma omp critical
  {
    total_hits++;
  }
}
```

- Not restricted to hardware supported ops
- Uses locks to restrict access to a single thread

Reduction vs. Atomic vs. Critical

- `omp_picalc_alt.c` has commented out versions of for each of reduction, atomic, and critical
- Examine timing differences between the three choices

```
lila [openmp-code]% gcc omp_picalc_alt.c -fopenmp
lila [openmp-code]% time -p a.out 100000000 4
npoints: 100000000
hits:    78541717
pi_est:  3.141669

real ??? - Elapsed (wall) time
user ??? - Total user cpu time
sys  ??? - Total system time
```

Time	Threads	real	user	sys
Serial	1	1.80	1.80	0.00
Reduction	4	0.52	2.00	0.00
Atomic	4	2.62	9.98	0.00
Critical	4	9.02	34.46	0.00

Exercise: No Reduction for You

```c
int total_hits=0;
#pragma omp parallel reduction(+: total_hits)
{
  int num_threads = omp_get_num_threads();
  int thread_id = omp_get_thread_num();
  int points_per_thread = npoints / num_threads;
  unsigned int state = 123456789 * thread_id;
  for (int i = 0; i < points_per_thread; i++) {
    double x = ((double) rand_r(&state)) / ((double) RAND_MAX);
    double y = ((double) rand_r(&state)) / ((double) RAND_MAX);
    if (x*x + y*y <~ 1.0){
      total_hits++;
    }
  }
}
```

- Alter `picalc` to NOT use `reduction` clause
- Use alternative like `atomic` or `critical`
- **Goal:** achieve same/better speed as `reduction` version

Answers: No Reduction for You

```c
// picalc_omp_atomic.c:
#pragma omp parallel
{
  int num_threads = omp_get_num_threads();
  int thread_id = omp_get_thread_num();
  int points_per_thread = npoints / num_threads;
  int my_hits = 0;              // private count
  unsigned int state = 123456789 * thread_id;
  int i;
  for (i = 0; i < points_per_thread; i++) {
    double x = ((double) rand_r(&state)) / ((double) RAND_MAX);
    double y = ((double) rand_r(&state)) / ((double) RAND_MAX);
    if (x*x + y*y <= 1.0){
      my_hits++;
    }
  }
  #pragma omp atomic
  total_hits += my_hits;      // lock total_hits before updating
}
```

Thread Variable Declarations

Pragmas can specify that variables are either shared or private. See `omp_private_variables.c`

```
tid = -1;
// #pragma omp parallel
#pragma omp parallel shared(tid)
{
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);
}

tid = -1;
#pragma omp parallel private(tid)
{
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);
}
```

Also available

- `shared` which is mostly redundant
- `firstprivate` guarantees initialization with shared value
- All of these are subsumed by lexical scoping in C

Parallel Loops

```
#pragma omp parallel for
for (int i = 0; i < 16; i++) {
  int id = omp_get_thread_num();
  printf("Thread %d doing iter %d\n",
         id, i);
}
```

```
OUTPUT
Thread 0 doing iter 0
Thread 0 doing iter 1
Thread 0 doing iter 2
Thread 0 doing iter 3
Thread 2 doing iter 8
Thread 2 doing iter 9
Thread 2 doing iter 10
Thread 2 doing iter 11
Thread 1 doing iter 4
Thread 1 doing iter 5
...
```

- OpenMP supports parallelism for independent loop iterations
- Note variable `i` is declared in loop scope
- Iterations **automatically divided** between threads in a blocked fashion
- **Assumption**: Loop Iterations are independent

Exercise: OpenMP Matrix Vector Multiply

```
// matvec_serial.c: Matrix/vector multiply demo
for(i=0; i<rows; i++){
  for(j=0; j<cols; j++){
    result[i] += matrix[i][j] * vector[j];
  }
}
```

- Describe 3 ways one might parallelize this operation
- Write OpenMP `#pragmas` for each
- Note: `reduction` on an array variables varies based on OpenMP version

Answers: OpenMP Matrix Vector Multiply

```
// Outer for loop multiplication
#pragma omp parallel for
for(int i=0; i<rows; i++){
  for(int j=0; j<cols; j++){
    result[i] += matrix[i][j] * vector[j];
  }
}

// Inner for loop multiplication: reduction
// on result[i] added in recent OpenMP
for(int i=0; i<rows; i++){
  #pragma omp parallel  for reduction(+:result[i])
  for(int j=0; j<cols; j++){
    result[i] += matrix[i][j] * vector[j];
  }
}

// Outer and Inner for loop multiplication
#pragma omp parallel for
for(int i=0; i<rows; i++){
  #pragma omp parallel for reduction(+:result[i])
  for(int j=0; j<cols; j++){
    result[i] += matrix[i][j] * vector[j];
  }
}
```

Timing Differences

Circa 2017

```
# Desktop
>> gcc omp_matvec_timing.c -fopenmp

# SKINNY
>> a.out 20000 10000
Outer :   0.2851
Inner :   0.2022
Both  :   0.2191

# FAT
> a.out 10000 20000
Outer :   0.2486
Inner :   0.1911
Both  :   0.2118

> export OMP_NESTED=true
> a.out 20000 10000
Outer :   0.2967
Inner :   0.2027
Both  :   1.1783
```

Today

```
# Laptop
>> gcc matvec_omp.c -O3 -fopenmp

# SKINNY
>> ./a.out 20000 10000
Outer   :   0.1568
Inner   :   0.1888
Both    :   0.1515

# FAT
>> ./a.out 10000 20000
Outer   :   0.1490
Inner   :   0.1869
Both    :   0.1484

>> export OMP_MAX_ACTIVE_LEVELS=2
>> ./a.out 20000 10000
Outer   :   0.1559
Inner   :   0.1935
Both    :   3.5133
```

Nested Parallelism Control

- By default nested parallelism is disabled for most GCC versions
- Like other aspects of OpenMP, can control nested parallelism via function calls like

```
// Old but Deprecated
omp_set_nested(1); // ON
omp_set_nested(0); // OFF
// NEW
omp_set_max_active_levels(2);
```

- Can also be specified via environment variables

```
export OMP_NESTED=true            # deprecated
export OMP_NESTED=false           # deprecated
export OMP_MAX_ACTIVE_LEVELS=2    # NEW method
export OMP_NUM_THREADS=4
```

- Env. Vars are handy for experimentation
- Other Features such as loop scheduling are controllable via directives, function calls, or environment variables

Syntax Note

```
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i < REPS; i++) {
    int id = omp_get_thread_num();
    printf("Thread %d did iter %d\n",
           id, i);
  }
}
printf("\n");

// ABOVE AND BELOW IDENTICAL

#pragma omp parallel for
for (int i = 0; i < REPS; i++) {
  int id = omp_get_thread_num();
  printf("Thread %d did iter %d\n",
         id, i);
}
printf("\n");
```

- Directives for OpenMP can be separate or coalesced
- Code on top and bottom are parallelized the same way
- In top code, removing first #pragma removes parallelism

Loop Scheduling - 4 Types

Static

- So far only done static scheduling with fixed size chunks
- Threads get fixed size chunks in rotating fashion
- Great if each iteration has same work load

Dynamic

- Threads get fixed chunks but when done, request another chunk
- Incurs more overhead but balances uneven load better

Guided

- Hybrid between static/dynamic, start with each thread taking a “big” chunk
- When a thread finishes, requests a “smaller” chunk, next request is smaller

Runtime

- Environment variables (`OMP_SCHEDULE`) used to select one of the others
- Flexible but requires user awareness

Basic Loop Scheduling

```c
// omp_loop_scheduling.c, assumes OMP_NUM_THREADS=4
const int REPS = 16;

#pragma omp parallel for schedule(static)
for (int i = 0; i < REPS; i++) { // thr 0: 0-3, thr 1: 4-7
  ...                             // thr 2: 8-11,thr 4: 12-15
}

#pragma omp parallel for schedule(static,2)
for (int i = 0; i < REPS; i++) { // thr 0: 0,1,8,9   thr 1: 2,3,10,11
  ...                             // thr 2: 4,5,12,13 thr 3: 6,7,14,15
}

#pragma omp parallel for schedule(dynamic,2)
for (int i = 0; i < REPS; i++) { // varies, all start with 2 iters
  ...                             // request more as completed
}

#pragma omp parallel for schedule(guided)
for (int i = 0; i < REPS; i++) {
  ...                            // varies, start with large chunks
}                                // request smaller chunks

#pragma omp parallel for schedule(runtime)
for (int i = 0; i < REPS; i++) {
  ...                            // controlled via environment var
}                                // ex: OMP_SCHEDULE=static
```

Code for Loop Scheduling

- `omp_loop_scheduling.c` demonstrates loops of each kind with printing
- `omp_guided_schedule.c` longer loop to demonstrate iteration scheduling during Guided execution

Exercise: Spell Checking

- Consider a spell checking problem
- Look up each word in a document in a dictionary to determine correct spelling
- If document word is not in the dictionary, report a misspelling

```
// fragment from spellcheck_omp.c
for (int i=0; i < document->word_count; i++) {
  int result =
    linear_search(dictionary, document->words[i]);
  if(result == -1){
    misspelled++;
  }
}
```

Questions

1. Parallelize the “outer” loop over words or the “inner” loop that is `linear_search()`
2. Which type of loop schedule seems to make the most sense? Static? Dynamic? Guided?

Answers: Spell Checking

1. Parallelize the "outer" loop over words or the "inner" loop that is `linear_search()`

   For a large number of words, outer "word" loop makes more sense than inner loop : induces less thread statup overhead. For a small number of words, may be more worthwhile to parallelize inner loop.

2. Which type of loop schedule seems to make the most sense? Static? Dynamic? Guided?

   Dynamic or Guided makes more sense. Especially with `linear_search()`, expect that some checks will take longer than others which means a Static schedule may lead to some threads with much more work than others.

Example Runs on Spellcheck w/ Word Loop Parallelized

```
>> time OMP_SCHEDULE=static spellcheck_omp ...
threads = 8
misspelled: 0
Thread 0 work: 110803941
Thread 1 work: 332426710
Thread 2 work: 554049479
Thread 3 work: 775672248
Thread 4 work: 997295017
Thread 5 work: 1218917786
Thread 6 work: 1440540555
Thread 7 work: 1662044229
Total work: 7091749965

real    0m12.110s
user    0m53.495s
sys     0m0.008s
```

```
>> time OMP_SCHEDULE=dynamic spellcheck_omp ...
threads = 8
misspelled: 0
Thread 0 work: 851351653
Thread 1 work: 887921206
Thread 2 work: 908569538
Thread 3 work: 893075776
Thread 4 work: 882219930
Thread 5 work: 873179476
Thread 6 work: 904986970
Thread 7 work: 890445416
Total work: 7091749965

real    0m7.877s
user    1m0.578s
sys     0m0.011s
```

```
>> time OMP_SCHEDULE=guided spellcheck_omp ...
threads = 8
misspelled: 0
Thread 0 work: 901203843
Thread 1 work: 892041145
Thread 2 work: 897067217
Thread 3 work: 895931158
Thread 4 work: 850295834
Thread 5 work: 892967175
Thread 6 work: 896993276
Thread 7 work: 865250317
Total work: 7091749965

real    0m8.853s
user    1m9.492s
sys     0m0.031s
```

```
>> time OMP_SCHEDULE=static,1 spellcheck_omp ...
threads = 8
misspelled: 0
Thread 0 work: 886431528
Thread 1 work: 886446415
Thread 2 work: 886461302
Thread 3 work: 886476189
Thread 4 work: 886491076
Thread 5 work: 886505963
Thread 6 work: 886520850
Thread 7 work: 886416642
Total work: 7091749965

real    0m7.665s
user    1m0.295s
sys     0m0.011s
```

Notes on Spellcheck

- Pure static scheduling does not balance the work well
- Dynamic / Guided gives reasonable performance improvement over pure Static scheduling
- Specific instance of

  ```
  >> spellcheck_omp english-words.txt english-words.txt
  ```

  allows for block-cyclic distribution for 0-overhead fair distribution of work
- Most problems where work distribution is unknown benefit from dynamic or guided scheduling

Sections: Non-loopy Parallelism

- Independent code can be “sectioned” with threads taking different sections.
- Good to parallelize distinct independent execution paths
- See `omp_sections.c`

```
#pragma omp sections
{
  #pragma omp section
  {
    printf("Thread %d computing d[]\n",
           omp_get_thread_num());
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];
  }

  #pragma omp section
  printf("Thread %d chillin' out\n",
         omp_get_thread_num());
}
```

Locks in OpenMP

- Implicit parallelism/synchronization is awesome but...
- Occasionally need more fine-grained control
- Lock facilities provided to enable mutual exclusion
- Each of these have analogues in PThreads we will discuss later

```
void omp_init_lock(omp_lock_t *lock);    // create
void omp_destroy_lock(omp_lock_t *lock); // destroy
void omp_set_lock(omp_lock_t *lock);     // wait to obtain
void omp_unset_lock(omp_lock_t *lock);   // release
int omp_test_lock(omp_lock_t *lock);     // check, don't wait
```

OpenMP Thread Pools

- By default, OpenMP + GCC makes use of **thread pools**
- Once a thread is started, it remains active, associated with the running process
- Tradeoff is
  1. Thread startup overhead is reduced after the first parallel block
  2. System load is constant for an OpenMP program: finishing a parallel block does NOT release OS resources for threads
- Generally this is favorable for most HPC applications

Experiment with `omp_thread_pool.c` to see this.