OpenMP: Open Multi-Processing

Chris Kauffman

Last Updated:
Mon Nov 8 09:33:13 AM CST 2021

Logistics

Today

- More on PThreads
- OpenMP for shared memory machines

Reading

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

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 pragmas supported by `gcc` including `poison` and `dependency`

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 `openmp_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/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

```
#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: Hello from thread %d of %d\n",
         id_shared, numThreads);
}

printf("\n");

#pragma omp parallel
{
  int id_private = omp_get_thread_num();
  numThreads = omp_get_num_threads();
  printf("B: Hello from thread %d of %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

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

Lessons

- OpenMp Threads share heap/globals just like PThreads
- Threads share any stack variables NOT in parallel blocks
- Thread variables are **private** if declared inside parallel blocks
- 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
- 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?

Exercise: Placement of Variables vs Runtime

Analyze these two examples and explain the timing difference between them

```
// (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

```
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;
  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){
      total_hits++;
    }
  }
}
```

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

Answers: No Reduction for You

```
// 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) RANI
    double y = ((double) rand_r(&state)) / ((double) RANI
    if (x*x + y*y <= 1.0){
      my_hits++;
    }
  }
  #pragma omp atomic
  total_hits += my_hits;       // lock total_hits before
```

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: cannot perform `reduction` on an array variable

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

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

```
val>> gcc matvec_omp.c -O3 -fopenmp
val>> ./a.out 2000 1000 4
Outer :   0.0332
Inner :  13.6268
Inner2:   9.1982
Both  :   0.0067
```

- Problem size much reduced on laptop
- Performance of INNER variants is abysmal
- Worth investigating further
- Starting threads induces overhead: cut down on this to improve performance
- Outer loop parallelism tends to be better in these cases

Why the performance difference for Inner/Both?

Nested parallelism turned off

No Reduction

```
# SKINNY
> gcc omp_matvec_timing.c -fopenmp
> a.out 20000 10000
Outer :   0.3143
Inner :   0.8805
Both  :   0.4444

# FAT
> a.out 10000 20000
Outer :   0.2481
Inner :   0.8038
Both  :   0.3750
```

With Reduction

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

Nested Parallelism is Not the Default

```
> gcc omp_matvec_printing.c -fopenmp
> a.out 10000 20000
#threads = 4 (outer)
#threads = 4 (inner)
#threads = 4 (both outer)
#threads = 1 (both inner)
Outer :   0.2547
Inner :   0.8186
Both  :   0.3735

> export OMP_NESTED=TRUE
> a.out 10000 20000
#threads = 4 (outer)
#threads = 4 (inner)
#threads = 4 (both outer)
#threads = 4 (both inner)
Outer :   0.2904
Inner :   0.8297
Both  :   0.8660
```

- Aspects of OpenMP can be controlled via function calls

  ```
  omp_set_nested(1); // ON
  omp_set_nested(0); // OFF
  ```

- Can also be specified via environment variables

  ```
  export OMP_NESTED=TRUE
  export OMP_NESTED=OFF
  export OMP_NUM_THREADS=4
  ```

- Env. Vars are handy for experimentation
- 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 used to select one of the others
- Flexible but requires user awareness: *What’s an environment variable?*

Code for Loop Scheduling

Basic Codes

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

Attempts to Get Dynamic/Guided Scheduling to Shine

- `omp_collatz.c` looping to determine step counts in Collatz sequences
- `omp_spellcheck.c` simulates spell checking with linear search for words
- In both cases Static scheduling appears to work just as well for large inputs

Thread Variable Declarations

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

```
tid = -1;
#pragma omp parallel
{
  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

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
```