OpenMP: Open Multi-Processing

Chris Kauffman

Last Updated:
Wed Nov 3 08:19:22 AM CDT 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
- 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

  ```c
  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
  - M$ Visual C++
  - Intel C/C++ compiler
- 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 ≈ high-level parallelism
- PThreads ≈ lower-level 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

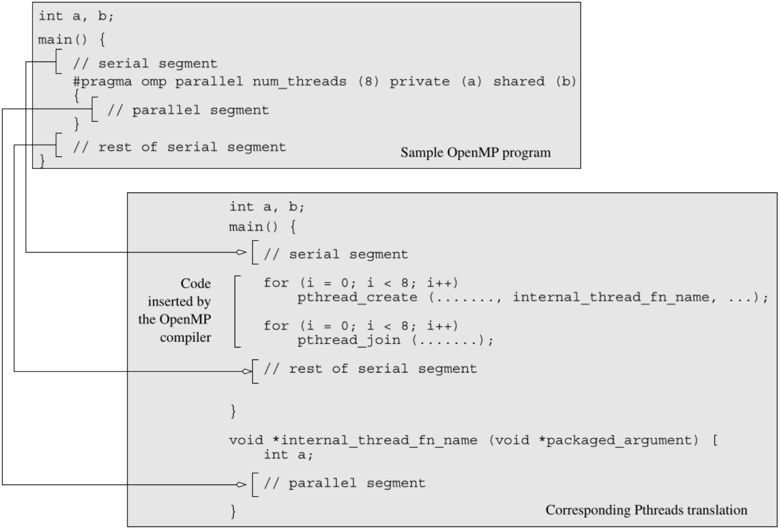

Figure 7.4 A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

OpenMP Thread Identification

- OpenMP divides computation into *threads*
- Nearly identical model to PThreads approach BUT not always implemented via PThreads
- 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

- Threads share heap
- Threads share any stack variables not in parallel blocks
- Thread variables are private if declared inside parallel blocks
- Take care with shared variables

Exercise: Pi Calc via OpenMP

Consider: `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?

rand() vs rand_r(long *state)

- rand() generates random integers on each invocation
- How can a function can return a different value on each call?
- rand_r(x) does the same thing but takes a parameter
- What is that parameter?
- What's the difference between these two?

Explore variant pi_calc_rand_r which exclusively uses rand_r()'s capabilities.

Comparing usage of rand_r()

What looks interesting to you about these two results.

omp_picalc.c

```
#pragma omp parallel ...
{
  unsigned int seed =
    123456789 * thread_id;
  ...
  double x =
    ((double) rand_r(&seed))...
```

TIMING

```
> gcc omp_picalc.c -fopenmp
> time -p a.out 100000000
npoints: 100000000
hits:    78541717
pi_est:  3.141669
real 0.52
user 2.00
sys 0.00
```

omp_picalc_rand_r.c:

```
unsigned int seed =
  123456789;
#pragma omp parallel...
{
  ...
  double x =
    ((double) rand_r(&seed))...
```

TIMING

```
> gcc omp_picalc_rand_r.c -fopenmp
> time -p a.out 100000000
npoints: 100000000
hits:    77951102
pi_est:  3.118044
real 3.05
user 11.77
sys 0.01
```

Note on `rand()`

- Not sure if `rand()` is or is thread-safe
- Conflicting info in manual, likely that this is a system dependent property
- Be careful

  The function rand() is not reentrant, since it uses hidden state that is modified on each call. This might just be the seed value to be used by the next call, or it might be something more elaborate. In order to get reproducible behavior in a threaded application, this state must be made explicit; this can be done using the reentrant function $rand_r$*().*

Interface	Attribute	Value
rand(), rand_r(), srand()	Thread safety	MT-Safe

Reduction

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

```
#pragma omp parallel reduction(+: total_hits)
{
  ...;
  total_hits++;
}
```

- Guarantees shared var `total_hits` is updated properly by all procs,
- As efficient as possible with an increment
- May exploit the fact that addition is transitive - can be done in any order
- Most arithmetic ops available

Alternative: Atomic

```
#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

```c
#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 seed = 123456789 * thread_id;
  int i;
  for (i = 0; i < points_per_thread; i++) {
    double x = ((double) rand_r(&seed)) / ((double) RAND_MAX);
    double y = ((double) rand_r(&seed)) / ((double) RAND_MAX);
    if (x*x + y*y <= 1.0){
      total_hits++;
    }
  }
}
```

- Reformulate `picalc` to NOT use `reduction` clause, use `atomic` or `critical` sections instead
- **Constraint:** must have same speed as the original `reduction` version
- Hint: draw on your experience from distributed MPI days

Parallel Loops

```c
#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

Exercise: OpenMP Matrix Vector Multiply

- Handout `matvec.c`: serial code to generate a matrix, vector and multiply
- Parallelize this with OpenMP
- Consider which `#pragma` to use
- Annotate with any problem spots

Available at:
`https://cs.gmu.edu/~kauffman/cs499/matvec.c`

Original Timing Differences

```
// OUTER
#pragma omp parallel for
for(int i=0; i<rows; i++){
  for(int j=0; j<cols; j++){
    res[i] += mat[i][j] * vec[j];
  }
}
// INNER
for(int i=0; i<rows; i++){
  #pragma omp parallel for
  for(int j=0; j<cols; j++){
    res[i] += mat[i][j] * vec[j];
  }
}
// BOTH
#pragma omp parallel for
for(int i=0; i<rows; i++){
  #pragma omp parallel for
  for(int j=0; j<cols; j++){
    res[i] += mat[i][j] * vec[j];
  }
}
```

```
# 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
```

Consider the timing differences between each of these three and explain the differences at least between

- OUTER SKINNY vs OUTER FAT
- INNER SKINNY vs INNER FAT
- OUTER vs INNER on both FAT and SKINNY

Updated Timing Differences

```c
// OUTER
#pragma omp parallel for
for(int i=0; i<rows; i++){
  for(int j=0; j<cols; j++){
    res[i] += mat[i][j] * vec[j];
  }
}
// INNER: with reduction
for(int i=0; i<rows; i++){
  double sum = 0.0;
  #pragma omp parallel \
    reduction(+:sum)
  {
    #pragma omp for
    for(int j=0; j<cols; j++){
      sum += mat[i][j] * vec[j];
    }
  }
  result[i] = sum;
}
```

```
# 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
```

Reduction was missing in the old version

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

```c
#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

Exercise: Looking Forward To HW 4

- Likely do a password cracking exercise
- Given an securely hashed password file
- **Describe** means to decrypt password(s) in this file
- How might one parallelize the work and why
- Does static or dynamic scheduling seem more appropriate?

id	username	hashed password
1	admin	645E2A7B0C1F4D45EF859725386B605D
2	pumpkin22	614B1F421A1F52727FF72A13CAC74F56
3	johndoe	8598500975B68DD9F2616A2B1A471F4E
4	alexa45	14BC2B3E56370B1FF4B8EFFC5DA13226
5	guy	7BB9FE4E6292A5D7CCD749755BC6B593
6	maryjane	8598500975B68DD9F2616A2B1A471F4E
7	dudson123	614B1F421A1F52727FF72A13CAC74F56

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`

```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
```