Lecture 6: OpenMP
Control vs. Simplicity Tradeoff

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
int main() {

    // Do this part in parallel

    printf( "Hello, World!\n" );

    return 0;
}
Small and Easy Motivation

```c
int main() {

    omp_set_num_threads(16);

    // Do this part in parallel
    #pragma omp parallel
    {
        printf( "Hello, World!\n" );
    }

    return 0;
}
```
Small and Easy Motivation

```c
void* SayHello(void *foo) {
    printf( "Hello, world!\n" );
    return NULL;
}

int main() {
    pthread_attr_t attr;
    pthread_t threads[16];
    int tn;
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    for(tn=0; tn<16; tn++) {
        pthread_create(&threads[tn], &attr, SayHello, NULL);
    }
    for(tn=0; tn<16; tn++) {
        pthread_join(threads[tn], NULL);
    }
    return 0;
}
```

The Pthreads version!
OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence.

OpenMP is a small API that hides cumbersome threading calls with simpler directives.
Interesting Insights About OpenMP

These insights are coming from HPC folks though!

Source: www.sdsc.edu/~allans/cs260/lectures/OpenMP.ppt
OpenMP

• An **API** for shared-memory parallel programming.

• Designed for systems in which each **thread** can potentially have access to all available memory.

• System is viewed as a collection of cores or CPU's, all of which have access to main memory → shared memory architecture
A shared memory system
Pragmas

- Special preprocessor instructions.
- Typically added to a system to allow behaviors that aren’t part of the basic C specification.
- Compilers that don’t support the pragmas ignore them.

#pragma
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

    #pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /* Hello */
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

./omp_hello 4

running with 4 threads

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

possible outcomes

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
OpenMP Parallel Programming

1. Start with a parallelizable algorithm
   • loop-level parallelism is necessary
2. Implement serially
3. Test and Debug
4. Annotate the code with parallelization (and synchronization) directives
   • Hope for linear speedup
5. Test and Debug
OpenMP In a Nutshell

• Works with C/C++, and Fortran
  – example: gcc -fopenmp

• Consists of:
  – compiler directives
  – library routines
  – environment variables

• Strengths: portability, simplicity, and scalability

• Weakness: less control to the programmer
Goals of OpenMP

- Standardization among platforms/architectures
- Easy of use
- Portability
All OpenMP programs begin with a single thread: **master thread** (ID = 0)

**FORK:** the master thread then creates a team of parallel **threads**.

**JOIN:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate.

OpenMP uses the fork-join model of parallel execution.
int main() {
    // serial region
    printf("Hello...");
    // parallel region
    #pragma omp parallel
    {
        printf("World");
    }
    // serial again
    printf("!");
}

We didn’t use `omp_set_num_threads()`, what will be the output?
Isn’t Nested Parallelism Interesting?
Important!

- The following are implementation dependent:
  - Nested parallelism
  - Dynamically alter number of threads
- It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multithreaded program.
- Threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time. The programmer is responsible for insuring that the variable is FLUSHed by all threads as needed.
What do you think is the output of the following code?

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

int main()
{
    int i, sum = 0;

    #pragma omp parallel for
    for(i = 0; i < 4; i++)
        printf("sum = %d\n", sum++);
}
```

one run:
sum = 0
sum = 1
sum = 2
sum = 2

another run:
sum = 1
sum = 2
sum = 0
sum = 2

yet another run:
sum = 0
sum = 1
sum = 2
sum = 3

OpenMP will not check dependencies among loop iterations!
#pragma omp parallel for

• Forks a team of threads to execute the following structured block.
• The structured block following the parallel for directive must be a for loop.
• The system parallelizes the for loop by dividing the iterations of the loop among the threads.
\[
h = (b-a)/n;
\]
\[
approx = (f(a) + f(b))/2.0;
\]
\[
\textbf{for} \ (i = 1; \ i \leq n-1; \ i++)
\]
\[
\quad \approx \ += \ f(a + i*h);
\]
\[
\approx = h*\approx;
\]
Legal forms for parallelizable for statements

Number of iterations MUST be known prior to the loop execution.

There can be a call to exit in the loop body.

OpnMP won’t parallelize while loops or do-while loops.
Caveats

• The variable **index** must have integer or pointer type (e.g., it can’t be a float).

• The expressions **start**, **end**, and **incr** must have a compatible type. For example, if index is a pointer, then **incr** must have integer type.
Caveats

• The expressions \texttt{start}, \texttt{end}, and \texttt{incr} must not change during execution of the loop.

• During execution of the loop, the variable \texttt{index} can only be modified by the “increment expression” in the for statement.
Scope

• In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.

• In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a parallel block.
Scope in OpenMP

• A variable that can be accessed by all the threads in the team has shared scope.

• A variable that can only be accessed by a single thread has private scope.

• The default scope for variables declared before a parallel block is shared.
```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

    #pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
}
/* Hello */
```
Directives

- Case sensitive
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

```
#pragma omp  [clause, clause, ...]
```
OpenMP

Directives  Runtime Libraries  Envir. Variables

#pragma omp [clause, clause, ...]

Example:
#pragma omp parallel [clause ...] newline
if (scalar_expression)
private (list)
shared (list) - Specify variables that will be shared and private
default (shared | none)
firstprivate (list)
reduction (operator: list)
copyin (list)
um_threads (integer-expression)

structured_block
#pragma omp parallel

- **if (scalar exp)**: the program code executes in parallel only if the scalar expression represented by exp evaluates to a non-zero value at runtime.
- **private (list)**: Declares the scope of the data variables in list to be private to each thread. Data variables in list are separated by commas.
- **firstprivate (list)**: Declares the scope of the data variables in list to be private to each thread. Each new private object is initialized with the value of the original variable as if there was an implied declaration within the statement block.
- **num_threads (int_exp)**: The value of int_exp is an integer expression that specifies the number of threads to use for the parallel region. If dynamic adjustment of the number of threads is also enabled, then int_exp specifies the maximum number of threads to be used.
- **shared (list)**: Declares the scope of the data variables in list to be shared across all threads.
- **default (shared | none)**: Defines the default data scope of variables in each thread.
#pragma omp parallel

- **copyin (list):** For each data variable specified in list, the value of the data variable in the master thread is copied to the thread-private copies at the beginning of the parallel region.

- **reduction (operator: list)** Performs a reduction on all scalar variables in list using the specified operator.
  - A private copy of each variable in list is created for each thread.
  - At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator.
  - The result is placed back into the original value of the shared reduction variable.
#include <omp.h>

main () {
    int nthreads, tid;

    /* Fork a team of threads with each thread having a private tid variable */
    #pragma omp parallel private(tid) {
        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }

    /* All threads join master thread and terminate */
}
#include <omp.h>

main () {
    int nthreads, tid;

    /* Fork a team of threads with each thread having a private tid variable */
    #pragma omp parallel private(tid) {
        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }

    /* All threads join master thread and terminate */
}
How Many Threads?

- Setting of the `NUM_THREADS` clause
- Use of the `omp_set_num_threads()` library function
- Setting of the `OMP_NUM_THREADS` environment variable
- Implementation default - usually the number of cores.
- Threads are numbered from 0 (master thread) to N-1
How Many Threads?

Fork

Join
Dividing Work Among Threads

- **FORK**
  - **DO** / for loop
  - **JOIN**
  - Team
  - Master thread

- **FORK**
  - **SECTIONS**
  - **JOIN**
  - Team
  - Master thread

- **FORK**
  - **SINGLE**
  - **JOIN**
  - Team
  - Master thread
Dividing Work Among Threads

- Number of iterations must be known in advance
- No gotos into or out of the loop.
#pragma omp for schedule (type [, chunk])

- **schedule clause** determines how loop iterations are divided among threads
  - **static** divides iterations statically among threads
    - Each thread receives [chunk] iterations, rounding as necessary to account for all iterations
    - Default [chunk] is \( \text{ceil}( \# \text{iterations} / \# \text{threads} ) \)
  - **dynamic** allocates [chunk] iterations per thread, allocating an additional [chunk] iterations when a thread finishes
    - Forms a logical work queue, consisting of all loop iterations
    - Default [chunk] is 1
  - **guided** allocates dynamically, but [chunk] is exponentially reduced with each allocation
Keep in mind:

- There is an overhead in using the schedule directive.
- The overhead is higher in dynamic than static schedules.
- The overhead of guided is the greatest of all three.
- So: if we get satisfactory performance without schedule then don’t use schedule.
Rules of thumb:

• If each iteration requires roughly the same amount of computation → default is best

• If the cost of each iteration increases/decreases linearly as the loop executes → static with small chunksize

• If the cost cannot be determined → you need to try several schedules
#pragma omp parallel private(f)
{
    f=7;
    #pragma omp for
    for (i=0; i<20; i++)
        a[i] = b[i] + f * (i+1);
} /* omp end parallel */
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000

main () {
    int i, chunk;
    float a[N], b[N], c[N];

    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;

    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
    } /* end of parallel section */
}
Dividing Work Among Threads

```
#pragma omp sections [clause ...]
{
    #pragma omp section
    structured_block

    #pragma omp section
    structured_block

} ← Implicit barrier here, unless you use nowait.
```
```c
#pragma omp parallel
{
#pragma omp sections
  {{
    a=...;
    b=...;
  }}
#pragma omp sections
  {
    c=...;
    d=...;
  }
#pragma omp sections
  {
    e=...;
    f=...;
  }
#pragma omp sections
  {
    g=...;
    h=...;
  }
} /*omp end sections*/
} /*omp end parallel*/
```
Dividing Work Among Threads

Specifies that the enclosed code is to be executed by only one thread in the team.

```
#pragma omp single [clause ...]
structured_block
```
Example About Consistency Model

**Code:**

*Initially A = Flag = 0*

```
P1
A = 23;
Flag = 1;
```

```
P2
while (Flag != 1) {};
... = A;
```

**Possible execution sequence on each processor:**

```
P1
Write A 23
Write Flag 1

P2
Read Flag //get 0
......
Read Flag //get 1
Read A //what do you get?
```

*Do You see the problem?*
### Example About Consistency Model

**Code:**

Initially $A = \text{Flag} = 0$

```
P1
A = 23;
flush;
Flag = 1;
```

```
P2
while (Flag != 1) {;
... = A;
}
```

**Execution:**

- P1 writes data into $A$
- Flush waits till write to $A$ is completed
- P1 then writes data to Flag
- Therefore, if P2 sees Flag = 1, it is guaranteed that it will read the correct value of $A$ even if memory operations in P1 before flush and memory operations after flush are reordered by the hardware or compiler.
What Does OpenMP Say Here?

```c
#pragma omp flush (list)
```

- Thread-visible variables are written back to memory at this point.
- A bit complicated scenario arises if two threads execute the flush at the same time with common variables between them.
- If you do not specify a list, then all variables will be flushed.
Synchronization: Critical Directive

Enclosed code
- executed by all threads, but
- restricted to only one thread at a time
  • C/C++:
    #pragma omp critical [( name )]
    structured-block

  • A thread waits at the beginning of a critical region until no other thread in the team is executing a critical region with the same name.
cnt = 0;
f=7;
#pragma omp parallel
{
#pragma omp for
    for (i=0; i<20; i++) {
        if (b[i] == 0) {
#pragma omp critical
            cnt ++;
        } /* endif */
        a[i] = b[i] + f * (i+1);
    } /* end for */
} /*omp end parallel */
Synchronization: Other Directives

- OpenMP Synchronization
  - OpenMP Critical Sections
    - No explicit locks
  - Barrier directives
  - Explicit Lock functions
    - may require flush directive
  - Single-thread regions within parallel regions
    - master, single directives

```c
#pragma omp critical
{
    /* Critical code here */
}

#pragma omp barrier

omp_set_lock( lock l );
/* Code goes here */
omp_unset_lock( lock l );

#pragma omp single
{
    /* Only executed once */
}
```
Directives

Runtime Libraries

Envir. Variables

**omp_get_thread_num()**
- The number of threads remains unchanged for a parallel region
- The above function returns an integer
- Different parallel regions may have different number of threads.

**omp_get_num_threads()**
- Returns, as integer, the number of threads in the current parallel region.
Directives

Are we in a parallel region?
OMP_IN_PARALLEL()

How many processors in the system?
OMP_GET_NUM_PROCS()
Environment variables allow the end-user to control the parallel code.

- All environment variable names are uppercase.

Example:

**OMP_NUM_THREADS**
Sets the maximum number of threads to use during execution.
For example: `setenv OMP_NUM_THREADS 8`
What OpenMP Does NOT Do

- **Not Automatic parallelization**
  - User explicitly specifies parallel execution
  - Compiler does not ignore user directives even if wrong

- **Not meant for distributed memory parallel systems**

- **Not necessarily implemented identically by all vendors**

- **Not Guaranteed to make the most efficient use of shared memory**
Questions: Pthreads Vs OpenMP

• When will you use Pthreads and when will you use OpenMP?
• Writing the same program in Pthreads and OpenMP, which one do you think will have better performance? Scalability?
• If you are using OpenMP, do you have any freedom to help the underlying hardware?
Microbenchmark: Ocean

• Simulates ocean temperature gradients via successive-approximation
  – Operates on a 2D grid of floating point values

• “Embarrassingly” Parallel
  – Each thread operates in a rectangular region
  – Inter-thread communication occurs only on region boundaries
  – Very little synchronization (barrier-only)

• Easy to write in OpenMP!
Microbenchmark: Ocean

```c
#pragma omp parallel for \
shared(ocean, x_dim, y_dim) private(x, y)
for( t=0; t < t_steps; t++) {
    for( x=0; x < x_dim; x++) {
        for( y=0; y < y_dim; y++) {
            ocean[x][y] = /* avg of neighbors */
        }
    }
} // Implicit Barrier Synchronization

temp_ocean = ocean;
ocean = other_ocean;
other_ocean = temp_ocean;
```
Microbenchmark: Ocean

- **ocean_dynamic** - Traverses entire ocean, row-by-row, assigning row iterations to threads with dynamic scheduling.

- **ocean_static** – Traverses entire ocean, row-by-row, assigning row iterations to threads with static scheduling.

- **ocean_squares** – Each thread traverses a square-shaped section of the ocean. Loop-level scheduling not used—loop bounds for each thread are determined explicitly.

- **ocean_pthreads** – Each thread traverses a square-shaped section of the ocean. Loop bounds for each thread are determined explicitly.
Microbenchmark: Ocean
Microbenchmark: Ocean

Normalized Speedup, Ocean 258x258

Threaded Strategy

- Dynamic
- Static
- Squares
- Pthreads

Thread Count
- 1p
- 2p
- 4p
- 8p
- 16p
Question

Can we parallelize the following loop? If yes, do it. If not, why not?

\[
a[0] = 0;
\]

\[
\text{for( } i = 1; i < n; i++ \) \quad a[i] = a[i-1] + i;
\]
Conclusions

• OpenMP is an easier way than Pthreads for multithreaded programming

• OpenMP depends on compiler directives, runtime library, and environment variable.

• May aspects of OpenMP are still implementation dependent, so you need to be careful!