

16-20 OpenMP

Lecture 16 OpenMP - I

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

int main() {

    omp_set_num_threads(16);

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

    return 0;
}
```

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

OpenMP

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

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

System is viewed as a collection of cores, all of which have access to the same main memory → **shared memory architecture**

Most of OpenMP is based on Pragmas

#pragma

Special **preprocessor** instructions, specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself.

How to Compile an OpenMP program?

```
gcc -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 pragmas

```
# pragma omp parallel
```

Most basic parallel directive.

The number of threads that run the following structured block of code is determined by the run-time system if the programmer does not specify a number of threads.

clause

Definition: text that modifies the pragma directive.

The `num_threads` clause can be added to a parallel directive. It allows the programmer to specify the number of threads that should execute the following block.

```
# pragma omp parallel num_threads ( thread_count )
```

Some terminology

In OpenMP parlance the collection of threads executing the parallel block — the original thread and the new threads — is called a **team**, the original thread is called the **master**, and the additional threads are called **slaves**.

Mutual exclusion

```
# pragma omp critical
```

```
global_result += my_result ;
```

only one thread can execute the following structured block at a time

```
#pragma omp atomic — critical for one line
```

#pragma omp for nowait —

```
void compute_histogram_mt4(char *page, int page_size,
                           int *histogram, int num_buckets){
1:   int num_threads = omp_get_max_threads();
2:   #pragma omp parallel
3:   {
4:     __declspec (align(64)) int local_histogram[num_threads][num_buckets];
5:     int tid = omp_get_thread_num();
6:     #pragma omp for
7:     for(int i = 0; i < page_size; i++){
8:         char read_character = page[i];
9:         local_histogram[tid][read_character]++;
10:    }
11:
12:    #pragma omp for
13:    for(int i = 0; i < num_buckets; i++){
14:        for(int t = 0; t < num_threads; t++)
15:            histogram[i] += local_histogram[t][i];
16:    }
17: }
```

Speed is
3.60 seconds.

What Can We Learn from the Previous Example?

Atomic operations

- They are expensive
- Yet, they are fundamental building blocks.

Synchronization:

- correctness vs performance loss
- Rich interaction of hardware-software tradeoffs

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

What we learned so far

- `#include <omp.h>`
- `gcc -fopenmp ...`
- `omp_set_num_threads(x);`
- `omp_get_thread_num();`
- `omp_get_num_threads();`
- `#pragma omp parallel [num_threads(x)]`
- `#pragma omp atomic`
- `#pragma omp single`

Conclusions

OpenMP is a standard for programming shared-memory systems.

OpenMP uses both special functions and preprocessor directives called pragmas.

OpenMP programs start multiple threads rather than multiple processes.

Many OpenMP directives can be modified by clauses.

Lecture 17 OpenMP - II

Scope

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

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.

Example

```

    global_result = 0.0;
    # pragma omp parallel num_threads(thread_count)
    {
    #     pragma omp critical
        global_result += Local_trap(double a, double b, int n);
    }

```

We force the threads to execute sequentially → slower

We can avoid this problem by:

1. **declaring a private variable inside the parallel block**
2. **moving the critical section after the function call**

```

    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
        double my_result = 0.0;  /* private */

        my_result += Local_trap(double a, double b, int n);
#       pragma omp critical
        global_result += my_result;
    }

```

Can we do better?


Reduction operators

A **reduction** is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result (we have seen that in MPI!).

All of the intermediate results of the operation should be stored in the same variable: the **reduction variable**.

A reduction clause can be added to a parallel directive.

```
reduction(<operator>: <variable list>)
```



+, *, -, &, |, ^, &&, ||

And the code becomes:

```

    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count) \
        reduction(+: global_result)
    global_result += Local_trap(double a, double b, int n);

```

#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**.

"Scope" rules

The default scope for variables declared before a parallel block is shared.

The default scope of loop index is private.

You can override the above rules using `default()`, `shared()`, and `private()` clauses.

OpenMP compilers don't check for dependences among iterations in a parallelized loop.

A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP.

The default clause

`default(none)`

With this clause the compiler will require that we specify the scope of each variable in the block and that has been declared outside the block.

Conclusions

To make the best use of OpenMP try to have programs with for-loops.

Scope is an error prone concept here. So be careful!

- It may be a good idea to specify the scope of each variable in the parallel (or parallel for) block

Lecture 18 OpenMP - III

More about loops in OpenMP — Odd-Even Sort

```
# pragma omp parallel num_threads(thread_count) \
    default(none) shared(a, n) private(i, tmp, phase)
for (phase = 0; phase < n; phase++) {
    if (phase % 2 == 0)
#        pragma omp for
        for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
                tmp = a[i-1];
                a[i-1] = a[i];
                a[i] = tmp;
            }
        }
    else
#        pragma omp for
        for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
                tmp = a[i+1];
                a[i+1] = a[i];
                a[i] = tmp;
            }
        }
    }
}
```

for directive does not fork any threads. But uses whatever threads that have been forked before in the enclosing parallel block.

Scheduling loops

cyclic assignment

Thread	Iterations
0	$0, n/t, 2n/t, \dots$
1	$1, n/t + 1, 2n/t + 1, \dots$
\vdots	\vdots
$t - 1$	$t - 1, n/t + t - 1, 2n/t + t - 1, \dots$

`schedule (type [, chunksize])`

Type can be:

- **static**: the iterations can be assigned to the threads before the loop is executed.
- **dynamic** or **guided**: the iterations are assigned to the threads while the loop is executing.
- **auto**: the compiler and/or the run-time system determine the schedule.
- **runtime**: the schedule is determined at runtime.

The chunksize is a positive integer.

The Static Schedule Type

Example: twelve iterations, $0, 1, \dots, 11$, and three threads

<code>schedule(static, 1)</code>	<code>schedule(static, 2)</code>
Thread 0: 0, 3, 6, 9	Thread 0: 0, 1, 6, 7
Thread 1: 1, 4, 7, 10	Thread 1: 2, 3, 8, 9
Thread 2: 2, 5, 8, 11	Thread 2: 4, 5, 10, 11

```

schedule(static, 4)
Thread 0: 0, 1, 2, 3
Thread 1: 4, 5, 6, 7
Thread 2: 8, 9, 10, 11

```

The Dynamic Schedule Type

The iterations are broken up into chunks of **chunksize** consecutive iterations.

Each thread executes a chunk, and when a thread finishes a chunk, it **requests another one from the runtime system**.

This continues until all the iterations are completed.

The chunksize can be omitted. When it is omitted, a **default chunksize of 1** is used.

The Guided Schedule Type

Each thread also executes a chunk, and when a thread finishes a chunk, it requests another one.

As chunks are completed **the size of the new chunks decreases**.

If no chunksize is specified, the size of the chunks decreases down to 1.

If chunksize is specified, it decreases down to chunksize, with the exception that the very last chunk can be smaller than chunksize.

The Runtime Schedule Type

The system uses the environment variable **OMP_SCHEDULE** to determine at run-time how to schedule the loop.

The OMP_SCHEDULE environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.

Example: `export OMP_SCHEDULE = "static,1"`

Rules of thumb

There is an overhead in using the schedule directive: guided > dynamic > static

If we get satisfactory performance without schedule then don't use schedule.

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

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

If the cost cannot be determined → `schedule(runtime)` and try different options with OMP_SCHEDULE

Conclusions

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

The main concept to parallelize a program with OpenMP is how to have independent for-loops.

Lecture 19 OpenMP - IV

Producers and consumers

Queues: Enqueue (from tail), Dequeue (from head)

Producer threads might "produce" requests for data.

Consumer threads might "consume" the request by finding or generating the requested data.

Example of Usage: Message-Passing

Each thread could have a shared message queue, and when one thread wants to “send a message” to another thread, it could enqueue the message in the destination thread’s queue.

A thread could receive a message by dequeuing the message at the head of its message queue.

Each thread executes the following:

Each thread executes the following:

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {  
    Send_msg();  
    Try_receive();  
}  
  
while (!Done())  
    Try_receive();
```

Send_msg()

```
    mesg = random();  
    dest = random() % thread_count;  
    # pragma omp critical  
    Enqueue(queue, dest, my_rank, mesg);
```

Try_receive()

```
    if (queue_size == 0) return;  
    else if (queue_size == 1)  
    #    pragma omp critical  
        Dequeue(queue, &src, &mesg);  
    else  
        Dequeue(queue, &src, &mesg);  
    Print_message(src, mesg);
```

Termination Detection

```

queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count)
    return TRUE;
else
    return FALSE;

```

each thread increments this after completing its for loop



Startup

When the program begins execution, a single thread, the master thread, will get command line arguments and allocate an **array of message queues**: one for each thread.

This array needs to be **shared among the threads**.

Each thread allocates its queue in the array.

One or more threads may finish allocating their queues before some other threads. We need an explicit barrier so that when a thread encounters the barrier, it blocks until all the threads in the team have reached the barrier.

- # pragma omp barrier

The Atomic Directive

pragma omp atomic

Higher performance than critical

It can only protect critical sections that consist of **a single C assignment statement**.

The statement must have one of the following forms:

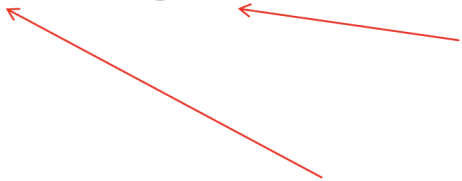
```

x <op>= <expression>;
x++;
++x;
x--;
--x;

```

+, *, -, /, &, ^, |, <<, or >>

Must not reference X



Critical Sections

pragma omp critical(name)

OpenMP provides the option of adding a name to a critical directive:

When we do this, two blocks protected with critical directives with **different names** can be **executed simultaneously**.

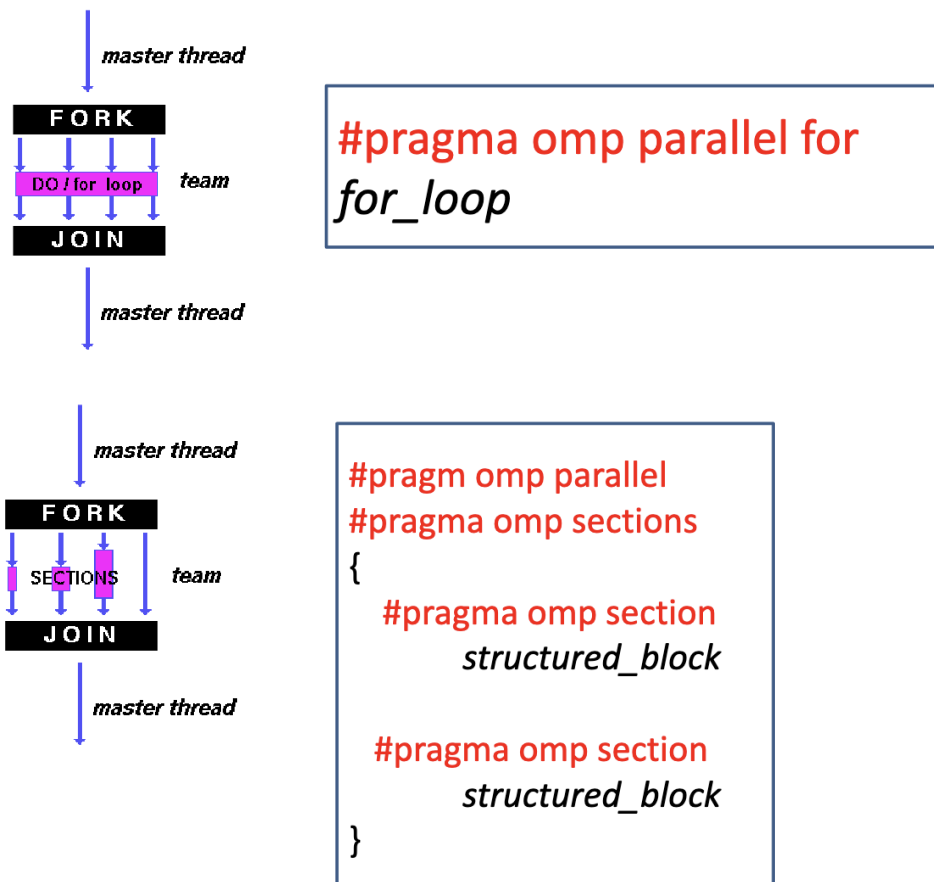
However, the **names are set during compilation**, and we may want a different critical section for each thread's queue.

Locks

A lock consists of a data structure and functions that allow the programmer to explicitly **enforce mutual exclusion** in a critical section.

```
void omp_init_lock(omp_lock_t * lock_p);  
void omp_set_lock(omp_lock_t * lock_p);  
void omp_unset_lock(omp_lock_t * lock_p);  
void omp_destroy_lock(omp_lock_t * lock_p);  
  
q_p = msg_queues[dest]  
omp_set_lock(&q_p->lock);
```

Dividing Work Among Threads



Tasks

Feature added to version 3.0 of OpenMP

A task is: an independent unit of work

A thread is assigned to perform a task.

Tasks example:

```
#pragma omp parallel {  
  
    #pragma omp single {  
        node *p = head_of_list;  
        while (p) {  
            #pragma omp task private(p)  
            process(p);  
            p = p->next;  
        } // end while  
    } //end pragma single  
} // end pragma parallel
```

Threads start executing tasks at that point.

Implicit barrier

Task Synchronization

#pragma omp barrier

#pragma omp taskwait

- explicitly waits on the completion of **child tasks**

Conclusions

We have seen three mechanisms to enforce mutual exclusion: atomic, critical, and locks

- atomic is fastest but with limitations (one line only)
- critical can name sections but at compile time
- locks are slowest but sometimes are the only option

Lecture 20 OpenMP - V

5 main causes of poor performance:

- Sequential code
 - Amdahl's law: Limits performance.
 - In OpenMP, all code outside of parallel regions and inside MASTER, SINGLE and CRITICAL directives is sequential. This code should be as small as possible.
- Communication
 - On Shared memory machines, **communication = increased memory access costs**.
 - Unlike message passing, communication is spread throughout the program. Much harder to analyze and monitor.

- Caches and coherency
- False sharing
- Data affinity
- Load imbalance
 - Worth experimenting with different scheduling options
- Synchronisation
 - Barriers can be very expensive
 - Avoid barriers via:
 - Careful use of the **NOWAIT** clause.
 - Parallelize at the outermost level possible.
 - May require re-ordering of loops /indices.
 - Choice of CRITICAL / ATOMIC / lock routines may impact performance.
- Compiler (non-)optimization.
 - Sometimes the addition of parallel directives can inhibit the compiler from performing sequential optimizations.
 - Symptoms: 1-thread parallel code has longer execution than sequential code.
 - Can sometimes be cured by making shared data private, or local to a routine.

Performance Tuning: Timing the OpenMP Performance

`$time ./a.out`

The “real”, “user”, and “system” times are then printed after the program has finished execution.

```
$ time .program.exe
real    5.4 Elapsed time
user    3.2
sys     1.0 } CPU time
```

A common cause for the difference between the wall-clock time of 5.4 seconds and the CPU time is a processor sharing too high a load on the system.

If sufficient processors are available, your elapsed time should be less than the CPU time.

The `omp_get_wtime()` function provided by OpenMP is useful for measuring the elapsed time of blocks of source code.

Performance Tuning: Avoid Parallel Regions in Inner Loop

Move parallel regions out of the innermost loops.

By doing this, the parallel construct overheads are minimized.

Hybrid OpenMP and MPI

– Pure MPI Pro:

- Portable to distributed and shared memory machines.
- Scales beyond one node
- No race condition problem

– Pure MPI Con:

- Difficult to develop and debug
- High latency, low bandwidth
- Explicit communication
- Large granularity
- Difficult load balancing

– Pure OpenMP Pro:

- Easy to implement parallelism
- Low latency, high bandwidth
- Implicit Communication
- Dynamic load balancing

– Pure OpenMP Con:

- Only on shared memory machines
- Scale within one node
- Possible race condition problem
- No specific thread order

Hybrid MPI/OpenMP paradigm is the **software trend** for clusters of SMP architectures, supercomputers (although GPUs are usually also used here),

Elegant in concept and architecture: using **MPI across nodes** and **OpenMP within nodes**. Good usage of shared memory system resource (memory, latency, and bandwidth).

Avoids the extra communication overhead with MPI within node.

OpenMP adds fine **granularity** and allows **increased** and/or **dynamic load balancing**.

Some problems have two-level parallelism naturally.

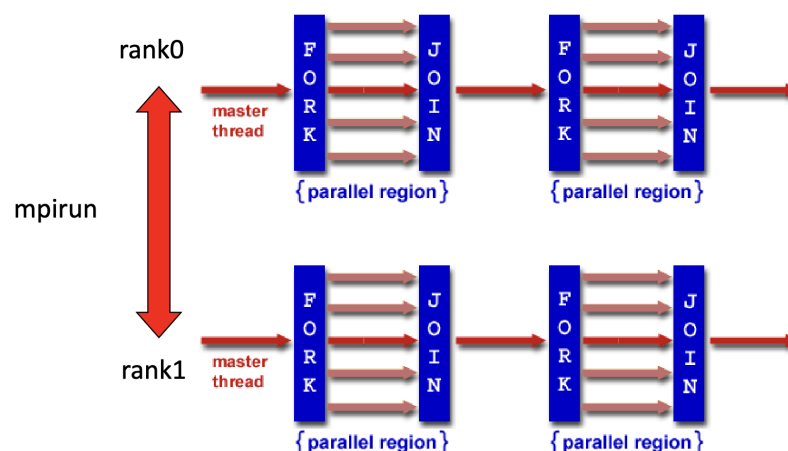
Could have better scalability than both pure MPI and pure OpenMP.

Hybrid Parallelization Strategies

From sequential code: decompose with MPI first, then add OpenMP.

Simplest and least error-prone way is:

- Use MPI outside parallel region.
- Allow only master thread to communicate between MPI tasks.



```

#include <stdio.h>
#include <mpi.h>
#include <omp.h>

int main(int argc, char *argv[]) {
    int numprocs, rank, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int iam = 0, np = 1;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(processor_name, &namelen);

    #pragma omp parallel default(shared) private(iam, np)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        printf("Hello from thread %d out of %d from process %d out of %d on %s\n",
              iam, np, rank, numprocs, processor_name);
    }

    MPI_Finalize();
}

```

Compile and Execution

mpicc -fopenmp mixhello.c

mpiexec -n x ./a.out

Conclusions

Always keep in mind the 5 reasons of poor performance

If:

- you have a machine with several nodes
- The problem at hand has two levels of parallelism

Then:

- consider hybrid OpenMP + MPI