
OpenMP

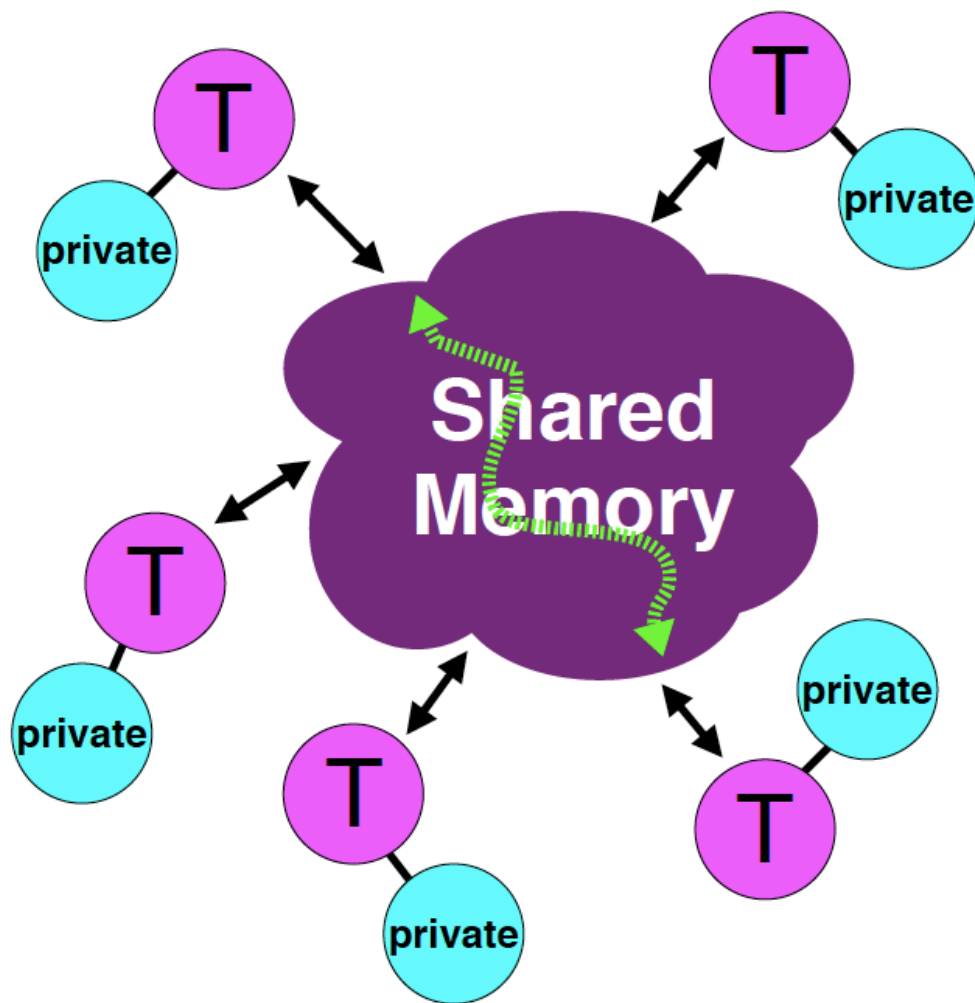
Open Multi-Processing

Salvatore Orlando

OpenMP

- OpenMP is a *portable directive-based API* that can be used with FORTRAN, C, and C++ for programming shared address space machines
 - the standard was formulated in 1997
- OpenMP provides the programmer with a *higher level of abstraction* than pthreads
- OpenMP supports:
 - Parallel execution
 - Parallel loops
 - Critical sections
 - Barriers
 - Access to a set of run-time environment variables
- There is no data distribution directive:
 - Data is in shared memory !! (as for pthreads)
- Obviates the need for explicitly setting up mutexes, condition variables, etc.
- OpenMP directives in C and C++ are based on the *#pragma* compiler directives.

OpenMP Memory Model



- ✓ *All threads have access to the same, globally shared, memory*
- ✓ *Data can be shared or private*
- ✓ *Shared data is accessible by all threads*
- ✓ *Private data can only be accessed by the thread that owns it*
- ✓ *Data transfer is transparent to the programmer*
- ✓ *Synchronization takes place, but it is mostly implicit*

OpenMP directives

- A directive consists of a directive name followed by clauses:

```
#pragma omp directive [clause list]
```

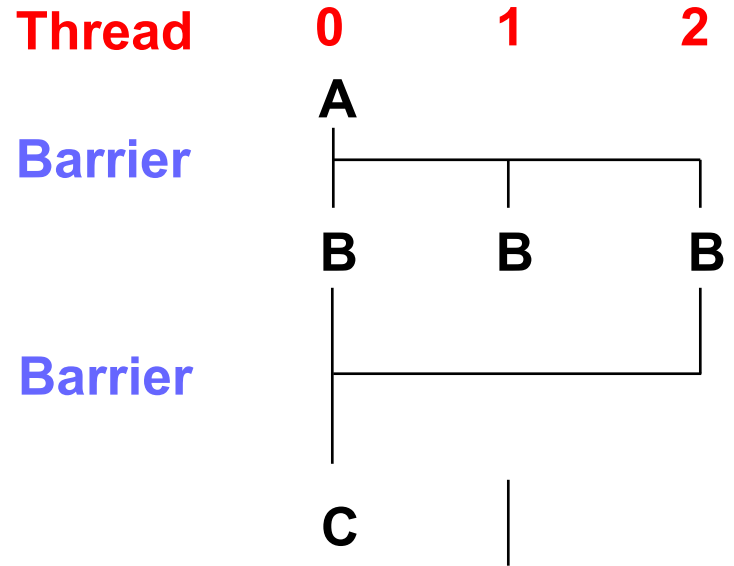
- OpenMP programs execute serially until they encounter the parallel directive, which creates a ***team of threads***.

```
#pragma omp parallel
{
    /* structured block */
}
```

- The main thread that encounters the parallel directive becomes the ***master*** of ***this team of threads*** and is assigned the ***thread id 0*** within the group

Execution model

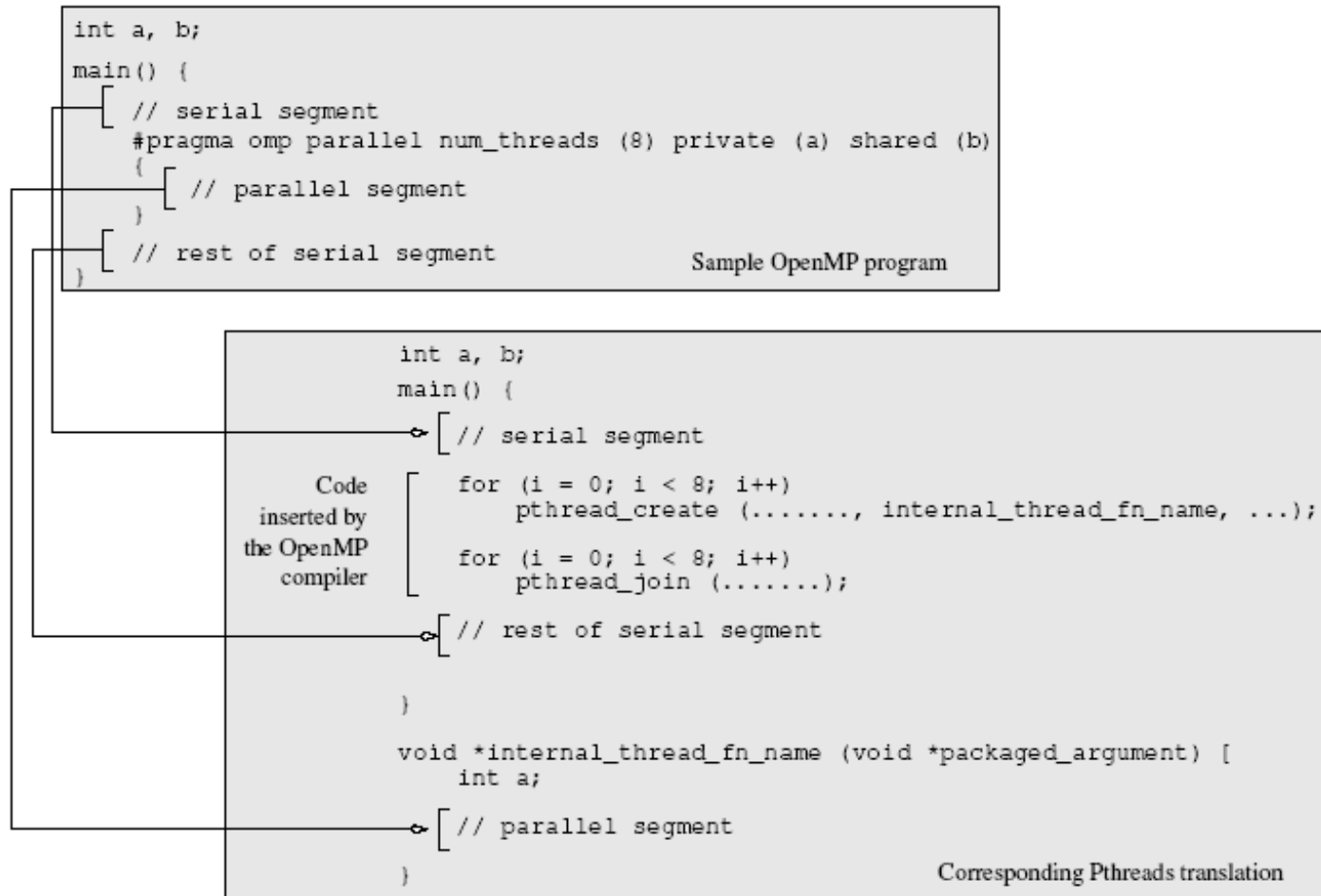
```
int main () {  
    // BLOCK A  
    ...  
    #pragma omp parallel  
    { // BLOCK B  
        cout << "Hello World!"  
          << endl;  
    }  
    // BLOCK C  
    ...  
}
```



A given number of threads execute the same body code (SPMD)

A thread can ask its own ID by using specific OMP primitives

OpenMP to threads



- A sample OpenMP program, along with its pthreads translation that might be performed by an OpenMP compiler.

Basic clauses

- **Degree of Concurrency:** the clause `num_threads(integer expression)` specifies the number of threads to create. The same can be decided with an environment variable
- **Conditional Parallelization:** the clause `if (scalar expression)` determines whether the parallel construct results in creation of threads
 - If the expression evaluates to false, the only one existing thread executes the following instruction block.

```
bool parallel = true;
#pragma omp parallel if(parallel) num_threads(10)
{
    cout << "Hello World!" << endl;
}
```

Variable sharing clauses

- **private (x)** : each thread has his own copy of variable x; variable x is not initialized
- **shared (x)** : every thread accesses the same memory location
- **firstprivate (x)** : each thread has his own copy of variable x; variable x is initialized with the current value of x before the various threads start
- **default (shared/none)** : affects all the variables not specified in other clauses. default(none) can be used to check whether you consider all the variables

Example

Source:

```
bool parallel = true;
int a=1, b=1, c=1, d=1;
#pragma omp parallel if(parallel) \
    num_threads(10) \
    private(a) shared(b) \
    firstprivate(c)
{
    cout << "Hello World!" << endl;
    a++;
    b++;
    c++;
    d++;
}

cout << "a: " << a << endl;
cout << "b: " << b << endl;
cout << "c: " << c << endl;
cout << "d: " << d << endl;
```

Output:

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
a: 1
b: 11
c: 1
d: 11
```

Reduction Clause

- The **reduction** clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.
 - `reduction (operator: variable list)`
- The variables in the list are implicitly specified as being private to threads
 - At the beginning of the parallel block, a private copy is made of the variable and pre-initialized to a certain value
 - At the end of the parallel block, the private copy is atomically merged into the shared variable using the defined operator
- The operator must be associative and can be one of:
`+`, `*`, `-`, `&`, `|`, `^`, `&&`, and `||`

Example

Source:

```
bool parallel = true;
int a=1, b=1, c=1, d=1;
#pragma omp parallel if(parallel) \
    num_threads(10) \
    reduction(*:a)
{
    cout << "Hello World!" << endl;
    a++;
    b++;
    c++;
    d++;
}

cout << "a: " << a << endl;
cout << "b: " << b << endl;
cout << "c: " << c << endl;
cout << "d: " << d << endl;
```

Output:

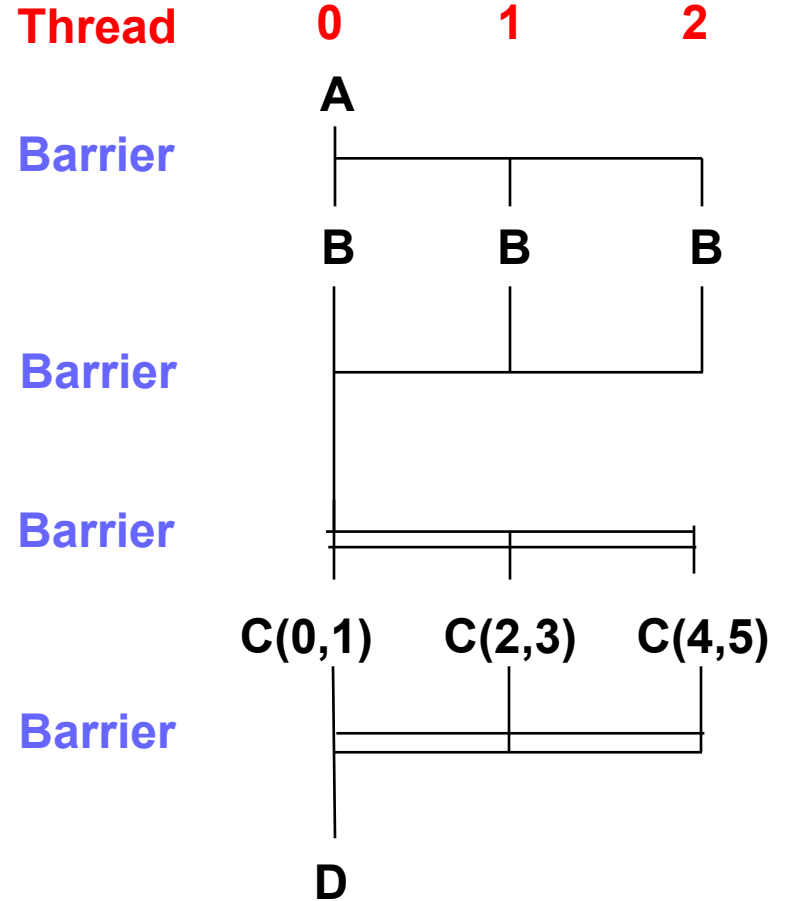
```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
a: 1024
b: 11
c: 11
d: 11
```

Specifying tasks

- The `parallel` directive can be used in conjunction with other directives to specify concurrency across iterations and tasks
- OpenMP provides two directives `for` and `sections` - to specify concurrent iterations and tasks.
- The `for` directive is used to split parallel the iterations of the subsequent loop among the available threads.
 - `lastprivate (x)` : has the same effect of `private(x)`, but the last running thread will store its value of `x`.
 - A barrier occurs at the end of the `for` execution block
- The `section` directive is used to explicitly identify pieces of code that can be run in parallel by the available threads

Execution model

```
int main () {  
    // BLOCK A  
    ...  
    #pragma omp parallel  
    { // BLOCK B  
        cout << "Hello World!"  
          << endl;  
    }  
    #pragma omp parallel  
    {  
        #pragma omp for \  
          schedule(static,2)  
        for (int i=0; i<6; i++) {  
            // BLOCK C  
        }  
    }  
    // BLOCK D  
    ...  
}
```



Directive for

Source:

```
int add = 0;
#pragma omp parallel reduction(+:add)
{
    #pragma omp for
    for (int i=0; i<10; i++) {
        add += i*i;
    }
}
cout << "squares sum: " << add << endl;
```

Output:

squares sum: 285

Assigning for iterations to threads

- The **schedule (policy[,param])** clause of the `for` directive deals with the assignment of iterations to threads.
- **schedule(static)**: the loop is statically split into chunks and each chunk is statically assigned to a thread.
 - Still we don't know in which order chunks will be executed
- **schedule(dynamic)**: the loop is statically split into chunks and each thread asks for the next thread to be executed.
 - We don't know each thread how many chunks will process
 - Useful to balance the load
- **schedule(guided)**: the chunk size is decrease exponentially in time.
 - Long tasks are assigned soon to minimize any (synchronization) overhead
 - Short tasks are assigned at the end to avoid idle threads
- **param**: specifies the chunk size (number of iterations to assign); in the guided case, it specifies the minimum chunk size

nowait and ordered directive

- Clause **nowait**: when the end of the for is not the end of the parallel work to be executed, it is desirable to avoid the barrier at the end of the execution block.
- directive **ordered**: when within a for execution block we want to force a piece of code to be executed according to the natural order of the for-loop

Restrictions to the for directive

- **For loops must not have break statements.**
- **Loop control variables must be integers.**
- **The initialization expression of the control variable must be an integer.**
- **The logical expression must be one of $<$, \leq , $>$, \geq .**
- **The increment expression must have integer increments and decrements.**

Sections

- OpenMP supports non-iterative parallel task assignment using the **sections** directive
 - task parallelism

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
            taskC();
        }
        #pragma omp section
        {
            taskD();
        }
    }
}
```

Synchronization directives

- **barrier**: all the threads wait for each other to reach the barrier
- **single [nowait]**: The following structured block enclosed is executed by only one thread in the team. Threads in the team that are not executing the SINGLE block wait at the end of the block unless NOWAIT is specified.
- **master**: Only the master thread of the team executes the block enclosed by this directive. The other threads skip this block and continue. There is no implied barrier on entry to or exit from the master construct.

Synchronization directives

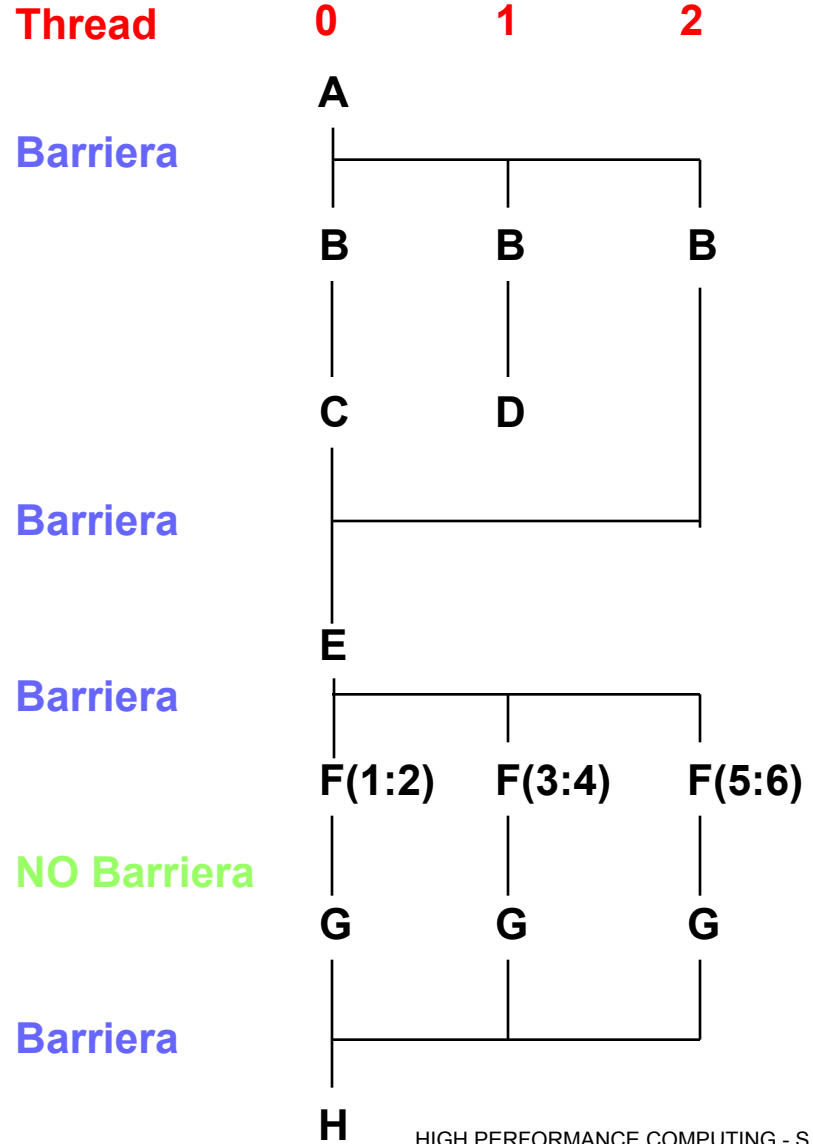
- **atomic**: assures atomicity of simple expression statements, otherwise you need to exploit the *critical* directive:
 - $x \text{ binop} = \text{expr}, x++, ++x, x--, --x$
- **critical [(name)]**: restricts access to the structured block to only one thread at a time. The optional name argument identifies the critical region. No two threads can enter a critical section with the same name.

Esecution model: parallel, section, for, single

```

program main
  A
  parallel
    B
    psections
      section
        C
      section
        D
    end psections
  psingle
    E
  end psingle
  pfor i=1,6
    F(i)
  end pdo no wait
  G
end parallel
H
end

```



Synchronization library functions

- Mutual Exclusion

```
void omp_init_lock (omp_lock_t *lock);  
void omp_destroy_lock (omp_lock_t *lock);  
void omp_set_lock (omp_lock_t *lock);  
void omp_unset_lock (omp_lock_t *lock);  
int omp_test_lock (omp_lock_t *lock);
```

Synchronization issues

- OpenMP synchronizes shared variables, but:
 - Compile may unpredictably optimize the code (using register variables)
 - Open MP uses a minimum effort approach on synchronization
- The **flush (var list)** directive can be used to synchronize a set of shared variables among the threads in the team
- OpenMP does flushing on:
 - barrier; parallel - upon entry and exit; critical - upon entry and exit; ordered - upon entry and exit; for - upon exit; sections - upon exit; single - upon exit

Environment variables

- **OMP_NUM_THREADS:** This environment variable specifies the default number of threads created upon entering a parallel region.
- **OMP_SET_DYNAMIC:** Determines if the number of threads can be dynamically changed.
- **OMP_NESTED:** Turns on nested parallelism.
- **OMP_SCHEDULE:** Specifies the scheduling algorithm used for loops not explicitly assigned a scheduling algorithm

Other Library functions

```
/* thread and processor count */
void omp_set_num_threads (int num_threads);
int omp_get_num_threads ();
int omp_get_max_threads ();
int omp_get_thread_num ();
int omp_get_num_procs ();
int omp_in_parallel();

/* controlling and monitoring thread creation */
void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();
```

OpenMP: example

```
#include <omp.h>
#include <stdio.h>
int main()
{
    #pragma omp parallel
    printf("Hello from thread %d, nthreads %d\n",
        omp_get_thread_num(), omp_get_num_threads());
}
```

foo.c

```
$> gcc -fopenmp foo.c -o foo
```

i686-apple-darwin10-gcc-4.2.1

```
$> export OMP_NUM_THREADS=2
```

```
$> ./foo
```

```
Hello from thread 0, nthreads 2
```

```
Hello from thread 1, nthreads 2
```

OpenMP: example

foo.c

```
#include <omp.h>
#include <stdio.h>
#define n 5
int a[n], b[n], c[n];
int main() {
    int i;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        printf("iteration %d, thread %d\n", i, omp_get_thread_num());
        c[i] = a[i] + b[i];
    }
}
```

```
$> gcc -fopenmp foo.c -o foo
```

```
$> export OMP_NUM_THREADS=3
```

```
$> ./foo
```

```
$> iteration 2, thread 1
```

```
$> iteration 3, thread 1
```

```
$> iteration 4, thread 2
```

```
$> iteration 0, thread 0
```

```
$> iteration 1, thread 0
```

OpenMP: mandelbrot

mandel.cc

```
#include <complex>
#include <cstdio>
typedef std::complex<double> complex;

int MandelbrotCalculate(complex c, int maxiter) {
    // iterates  $z = z^2 + c$  until  $|z| \geq 2$  or maxiter is reached,
    // returns the number of iterations.
    complex z = c;
    int n=0;
    for(; n<maxiter; ++n) {
        if(std::abs(z) >= 2.0)
            break;
        z = z*z + c;
    }
    return n;
}
```

OpenMP: mandelbrot

mandel.cc

```
int main() {
    const int width = 78, height = 44, num_pixels = width*height;
    const complex center(-.7, 0), span(2.7, -(4/3.0)*2.7*height/width);
    const complex begin = center-span/2.0, end = center+span/2.0;
    const int maxiter = 100000;
    #pragma omp parallel for ordered schedule(dynamic)
    for(int pix=0; pix<num_pixels; ++pix) {
        const int x = pix%width, y = pix/width;
        complex c = begin + complex(x * span.real() / (width +1.0),
            y * span.imag() / (height+1.0));
        int n = MandelbrotCalculate(c, maxiter);
        if (n == maxiter)
            n = 0;
        #pragma omp ordered {
            char c = ' ';
            if(n > 0) {
                static const char charset[] = ".,c8M@jawrpog0QEPGJ";
                c = charset[n % (sizeof(charset)-1)];
            }
            std::putchar(c);
            if(x+1 == width) std::puts("|");
        }
    }
}
```

Dynamic scheduling: the execution order is unpredictable (threads asks the OpenMP runtime library for an iteration number)

It is possible to force that certain events within the loop happen in a predicted order, using the **ordered clause**.

Example computing Pi

```
void computePi(int n_points, int n_threads) {
    cout << "Starting..." << endl;
    timeval start;
    gettimeofday(&start, NULL);

    double d = 1.0/(double)n_points;
    double pi = 0.0, x;

    #pragma omp parallel for \
        num_threads(n_threads) \
        reduction(+:pi) \
        private(x) \
        firstprivate(d)
    for (int i=1; i<=n_points; i++) {
        x = (i-0.5)*d;
        pi += 4.0/(1.0+x*x);
    }

    pi *= d;

    timeval end;
    gettimeofday(&end, NULL);
    double elapsed = end.tv_sec+end.tv_usec/1000000.0
        -start.tv_sec-start.tv_usec/1000000.0;

    cout << "PI = " << pi << endl;
    cout << "elapsed time : " << elapsed << " secs." << endl;
}
```

Further details

- **Short**
 - <https://computing.llnl.gov/tutorials/openMP/>
 - <http://bisqwit.iki.fi/story/howto/openmp/>
- **Long:**
 - http://www.filibeto.org/sun/lib/development/studio_8/817-0933.pdf