



CME 213
SPRING 2017

Eric Darve

OPENMP

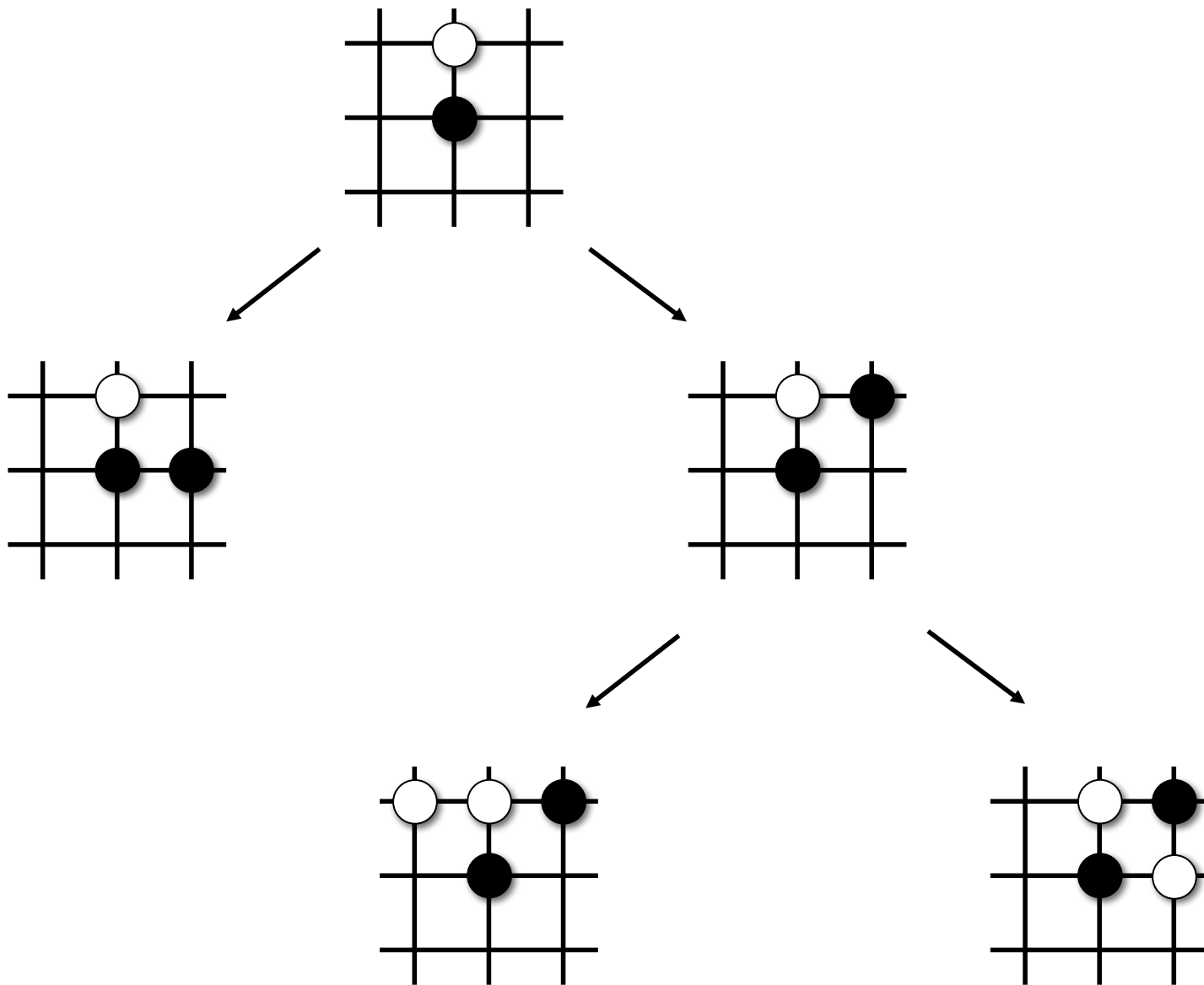
- Standard multicore API for scientific computing
- Based on fork-join model: fork many threads, join and resume sequential thread
- Uses pragma: `#pragma omp parallel`
- Shared/private variables
- Parallel for loop: scheduling (static, dynamic, guided)
- Sections

TASKING CONSTRUCTS



TASKS

- Sections are convenient in static cases where we know ahead of time (at compilation) how many concurrent operations need to be executed.
- However, there are situations when we need to generate parallel work at run time.
- Example:
 - Parallel traversal of a tree.
 - Parallel traversal of a linked list.
- Tasks provide a more **dynamic and flexible** way to handle concurrent work.



TREE . CC

- Assume we have a binary tree that we wish to traverse.
- We go through each node and execute some operations.
- Tree is not full, e.g., some of the child nodes may be missing.
- See `tree.cc`

```
#pragma omp parallel
#pragma omp single
    // Only a single thread should execute this
    Traverse(root);

// Pre-order traversal
void Traverse(struct Node* curr_node) {
    // Pre-order = visit then call Traverse()
    Visit(curr_node);

    if(curr_node->left)
#pragma omp task
        Traverse(curr_node->left);

    if(curr_node->right)
#pragma omp task
        Traverse(curr_node->right);
}
```

Does the function return before the tasks complete?

Yes

No

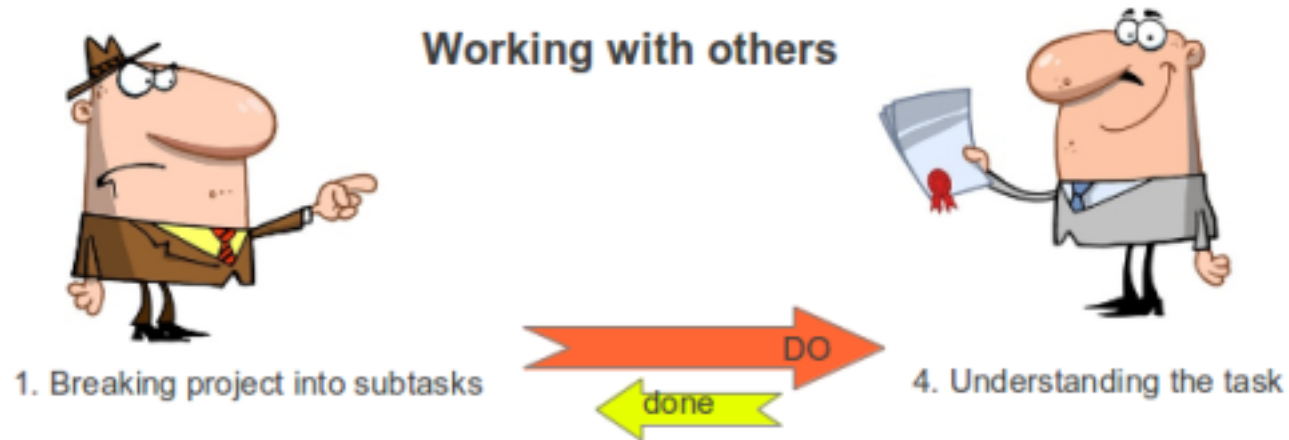
Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

0

Total Results: 0

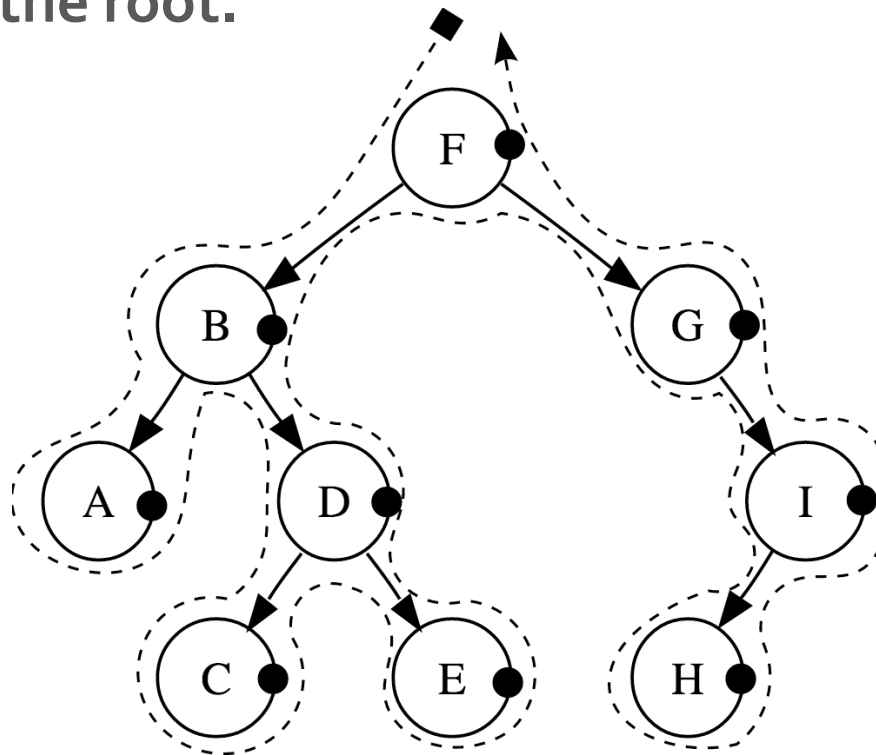
TASK



- When a thread encounters a **task** construct, a task is generated for the associated structured block.
- The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task.
- **task** should be called from within a parallel region for the different specified tasks to be executed in parallel.
- The tasks will be executed in no specified order because there are no synchronization directives.

POST-ORDER TRAVERSAL

- Let us now assume that we want to traverse the tree using a post-order:
 1. Traverse the left subtree.
 2. Traverse the right subtree.
 3. Visit the root.



POST-ORDER TRAVERSAL

- Assume we cannot visit the root until the left and right subtrees have been visited.
- For example, we are counting some quantity associated with each tree node.
- **In that case, a synchronization point is needed.**
- The thread needs to wait until both subtrees have been visited.

`tree_postorder.cc`

```

int PostOrderTraverse(struct Node* curr_node) {
    int left = 0;
    int right = 0;

    if(curr_node->left)
#pragma omp task shared(left)
        left = PostOrderTraverse(curr_node->left);

    if(curr_node->right)
#pragma omp task shared(right)
        right = PostOrderTraverse(curr_node->right);

#pragma omp taskwait
    curr_node->data = left + right;
    // Number of children nodes
    Visit(curr_node);

    return 1 + left + right;
}

```

- Default attribute for task constructs is **firstprivate**.
- **firstprivate**: private but value is initialized with the value that the corresponding original item has when the construct is encountered.

If left/right are shared, is there a potential race condition on the write?

Yes

No

Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

0

Total Results: 0

TECHNICAL EXPLANATION ON TASKWAIT

- OpenMP defines the concept of child task. A child task of a piece of code is a task generated by a directive `#pragma omp task` found in that piece of code.
- For example, in the previous code
`PostOrderTraverse(curr_node->left)`
and
`PostOrderTraverse(curr_node->right)`
are child tasks of the enclosing region.
- **taskwait** specifies a wait on the completion of the child tasks of the current task.
- Note that **taskwait** requires to wait for completion of the child tasks, but not completion of all descendant tasks (e.g., child tasks of child tasks).

PROCESSING ENTRIES IN A LIST

- This is another classical example cases of using tasks.
- We want to process entries in a list in parallel.
- A parallel for is not possible because there is no index we can use to iterate over the list entries.
- Tasks can be used for that purpose.

See `list.cc`



```

void IncrementListItems(Node* head) {
#pragma omp parallel
{
#pragma omp single
{
    Node* curr_node = head;
    while(curr_node) {
#pragma omp task
        {
            // curr_node is firstprivate by default
            Visit(curr_node);
        }
        curr_node = curr_node->next;
    }
}
}

void Visit(Node* curr_node) {
    /* Add 1 to data */
    ++(curr_node->data);
}

```


When is curr_node initialized inside the task?

At this line:
curr_node...

When the task is
run


When the task is
created

Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

0

Total Results: 0

The background of the slide features a large, light gray watermark of the Stanford University seal. The seal is circular, with a diamond-patterned border. Inside the border, the text "LELAND STANFORD JUNIOR UNIVERSITY" is written in a circular path. Below this, the German phrase "DIE LUFT DER FREIHEIT" is visible. In the center of the seal is a detailed illustration of a redwood tree standing on a rocky outcrop. At the bottom of the seal, the year "1891" is inscribed.

OPENMP MASTER AND SYNCHRONIZATION CONSTRUCTS

SYNCHRONIZATION CONSTRUCTS

- **Several constructs are available.**
- **We won't go into all the details.**
- **Check the reading material for more information.**
- **From most to least common.**

REDUCTION

- Consider the code in `reduction.cc`
- There is a race condition on `average`.
- The reduction clause does two things:
 - Prevent a race condition when updating **average**.
 - Improved efficiency: local reduction followed by a single final reduction across all thread values.

```
#pragma omp parallel for reduction (+:sum)
    for(int i = 0; i < size; i++) {
        sum += a[i];
    }

    assert(sum == (size*(size-1))/2);
    cout << "All tests passed\n";
    return 0;
}
```

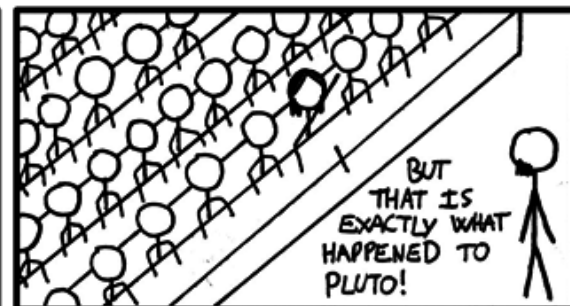
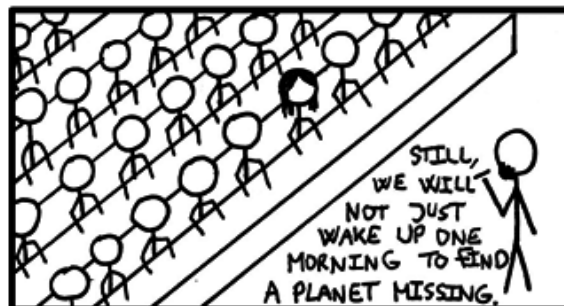
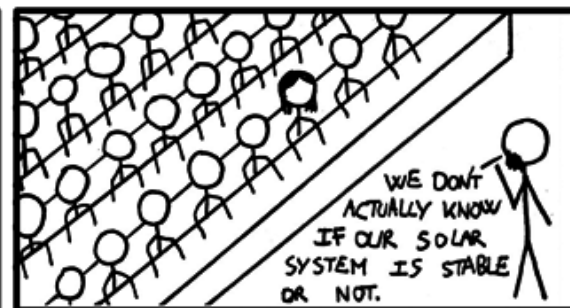
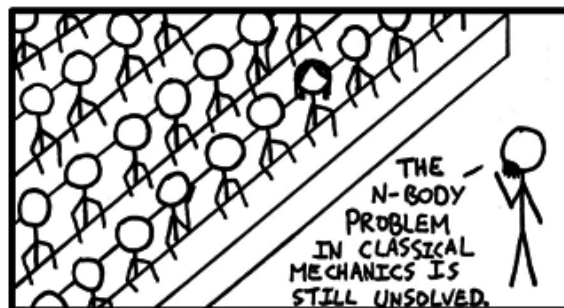
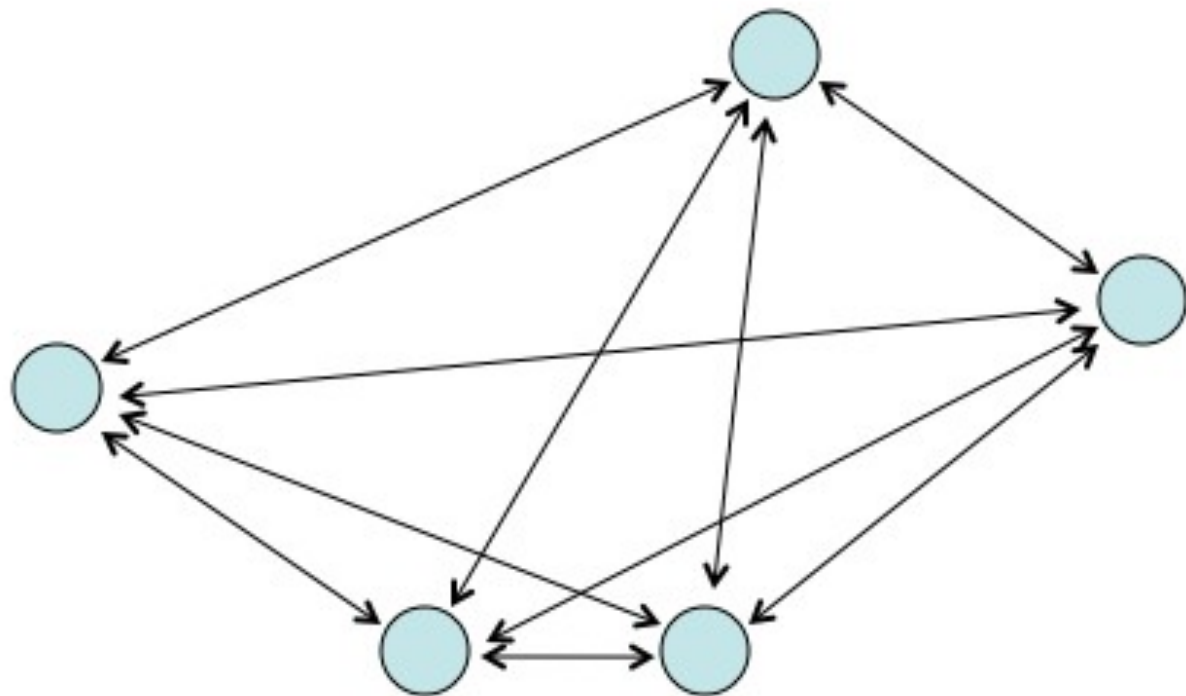
TECHNICAL DEFINITION

reduction (op: list)

- **op** can be: $\{+, -, *, \&, ^, |, \&\&, ||\}$
- For each of the variables in **list**, a private copy is created for each thread.
- The private copies are initialized to the neutral element (zero) of the operation **op** and can be updated by the owning thread.
- At the end of the region, the local values of the reduction variables are combined according to the operator **op** and the result of the reduction is written into the original shared variable.

ATOMIC

- This is a situation similar to the previous case.
- However, this is generalized to any kind of updates of a shared variable.
- When all threads try to read and write to a shared variable, a race condition ensues.
- Atomic guarantees the correctness of the final result.
- See `atomic.cc`
- Atomic: atomic exclusion, but only applies to the update of a memory location, e.g., the force in our example.



```
#pragma omp parallel for
    for(int i=0; i<n; ++i)
        for(int j=i+1; j<n; ++j) {
            const float x_ = x[i] - x[j];
            const float f_ = force(x_);
#pragma omp atomic
            f[i] += f_;
#pragma omp atomic
            f[j] -= f_;
        }
```


atomic and reduction more or less the same thing

No, atomic is more general

Yes, because they correspond to a variable update

No, their impact on performance is different

Yes, because a

ction can always be

written using atomic

Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

0

Total Results: 0

CRITICAL

A further extension where a piece of code need to be executed only by a one thread at a time. `critical.cc`

```
set<int> m;
#pragma omp parallel
{
    const int id = omp_get_thread_num();
    const int n_threads = omp_get_num_threads();

    for(int i(id); i < n; i += n_threads) {
        // A different way to write a parallel for loop
        bool is_prime = LongCalculation(i);

        // Comment the line below to get a bug
        #pragma omp critical
        if(is_prime) {
            Consume(i, m);    /* Save this prime */
        }
    }
}
```

SINGLE

#pragma omp single

Definition: structured block is executed by only one thread.

- See **single.cc**
- See also example using tasks.

```
#pragma omp parallel
{
    BigCalculationOne();
    #pragma omp single
    {
        ApplyBoundaryConditions();
    }
    BigCalculationTwo();
}
```

ORDERED

```
#pragma omp for ordered schedule(dynamic)
    for(unsigned n=0; n<size; ++n) {
        files[n].compress();

#pragma omp ordered
        send(files[n]);
    }
```

MANDELBROT FRACTAL SET

- Iteration:

$$z_{n+1} = z_n^2 + c$$

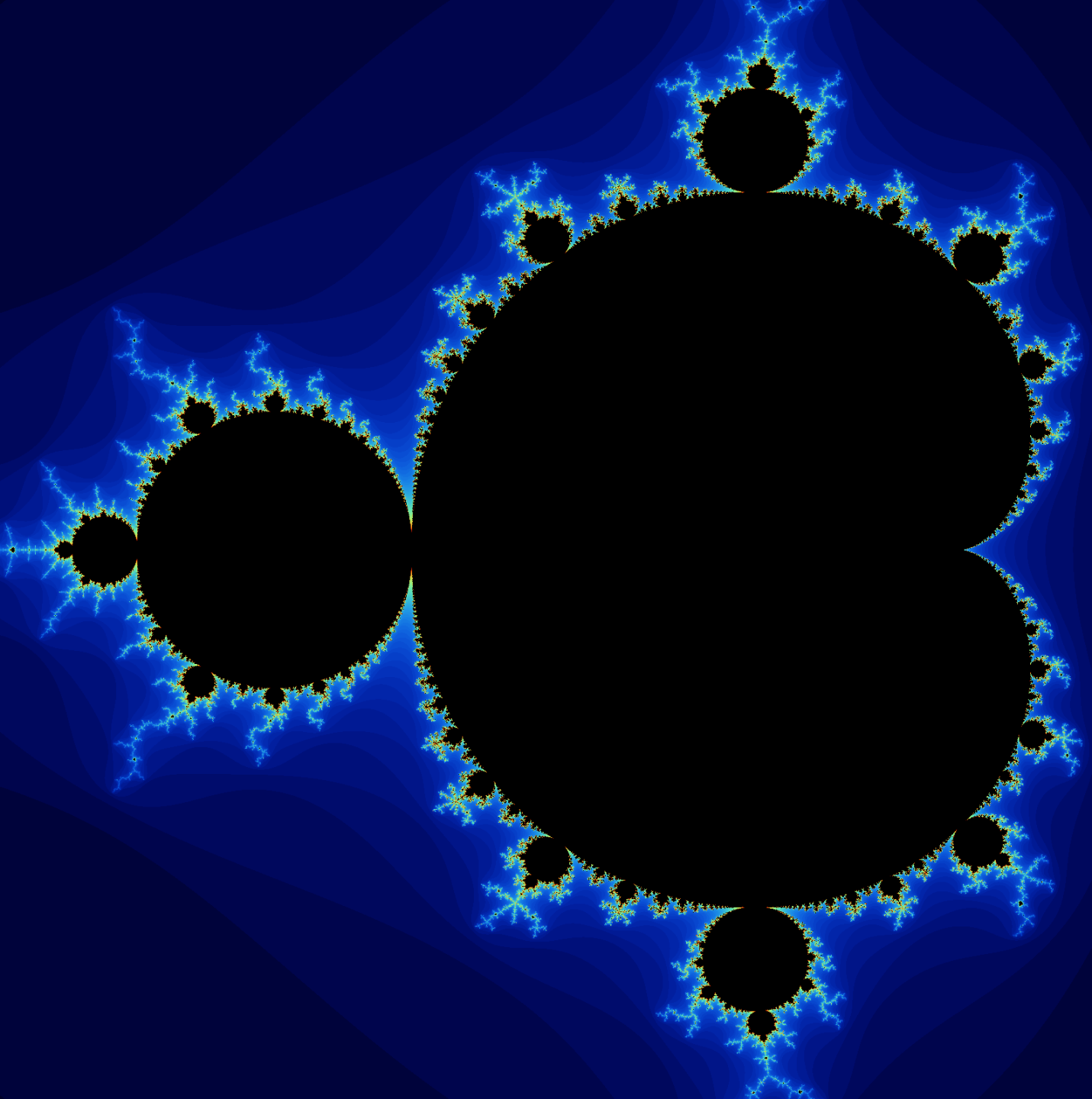
- Start iteration from 0.
- Complex number c is in the Mandelbrot set M if the sequence z_n remains bounded.
- For example, $c = 1$ gives
0 1 2 5 26 ...
and diverges. Not in M .
- $c = -1$ gives
0 -1 0 -1 0 -1 ...
and remains bounded. c is in M .

NUMBER OF ITERATIONS

We calculate the number of iterations until z_n becomes greater than 2:

```
unsigned MandelbrotCalculate(complex c) {  
    const unsigned iter_max = 100;  
    unsigned iter = 0;  
  
    complex z(0);  
    while (abs(z) <= 2.0 && iter < iter_max) {  
        z = z * z + c;  
        iter++;  
    }  
  
    return (iter==iter_max) ? 0 : iter;  
}
```

This is based on $c \in M \iff \limsup_{n \rightarrow \infty} |z_n| \leq 2$



Color
based on
iter

OMP CODE FOR MANDELBROT

```
#pragma omp parallel for ordered schedule(dynamic)
    for(int k = 0; k < size; ++k) {
        const int i = k % width;
        const int j = k / width;
        complex c = begin
        + complex(i * span.real() / (width - 1.0),
                  j * span.imag() / (height - 1.0));
        unsigned n = MandelbrotCalculate(c);
        colors[k] = (int) n;

#pragma omp ordered
        {
            /* Print an ASCII character corresponding to the
               number of iterations. */
            PrintCharacter(i, j, n, width);
        }
    }
```


ASCII output

es ordered make the loop execute sequential

Yes

No

Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

0

Total Results: 0

OTHER CONSTRUCTS

- There are several other constructs that can be useful in special cases.
- Short list:
 1. Master
 2. Barrier
 3. Locks
 4. Flush
 5. `double omp_get_wtime(void);`
 6. and many others...