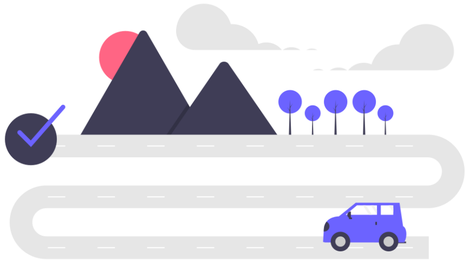
CME 213, ME 339 Spring 2023
Introduction to parallel computing using MPI, openMP, and CUDA

Eric Darve, ICME, Stanford

"Optimism is an occupational hazard of programming; feedback is the treatment." (Kent Beck)

Stanford University

Recap



- How to use the icme-gpu cluster.
- OpenMP: facilitates writing multi-threaded code
- Relies on directives like `#pragma omp parallel`
- Fork-join model
- Parallel for loops

Loop scheduling

- Loop scheduling is important for performance.
- Some fine-tuning is often required.
- A parallel for loop often has a barrier at the end.
- At the barrier, the other members of the team must wait for the last thread to arrive.
- To minimize this wait time, shared work should be distributed so that all threads arrive at the barrier at about the same time.
- The schedule clause allows optimizing this execution.

General syntax

```
#pragma omp for schedule(kind,chunk_size)
```

`kind = static, dynamic, guided`

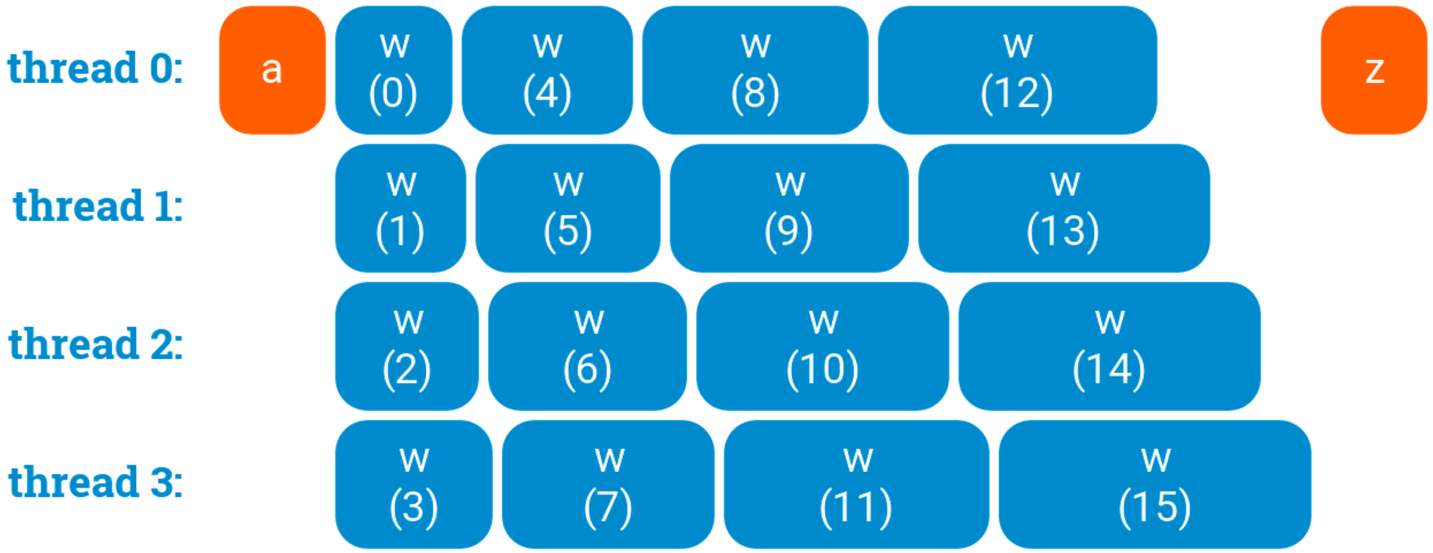
`chunk_size` the exact meaning varies depending on `kind` but generally this is the number of iterations in a given chunk assigned to a thread.

`static`

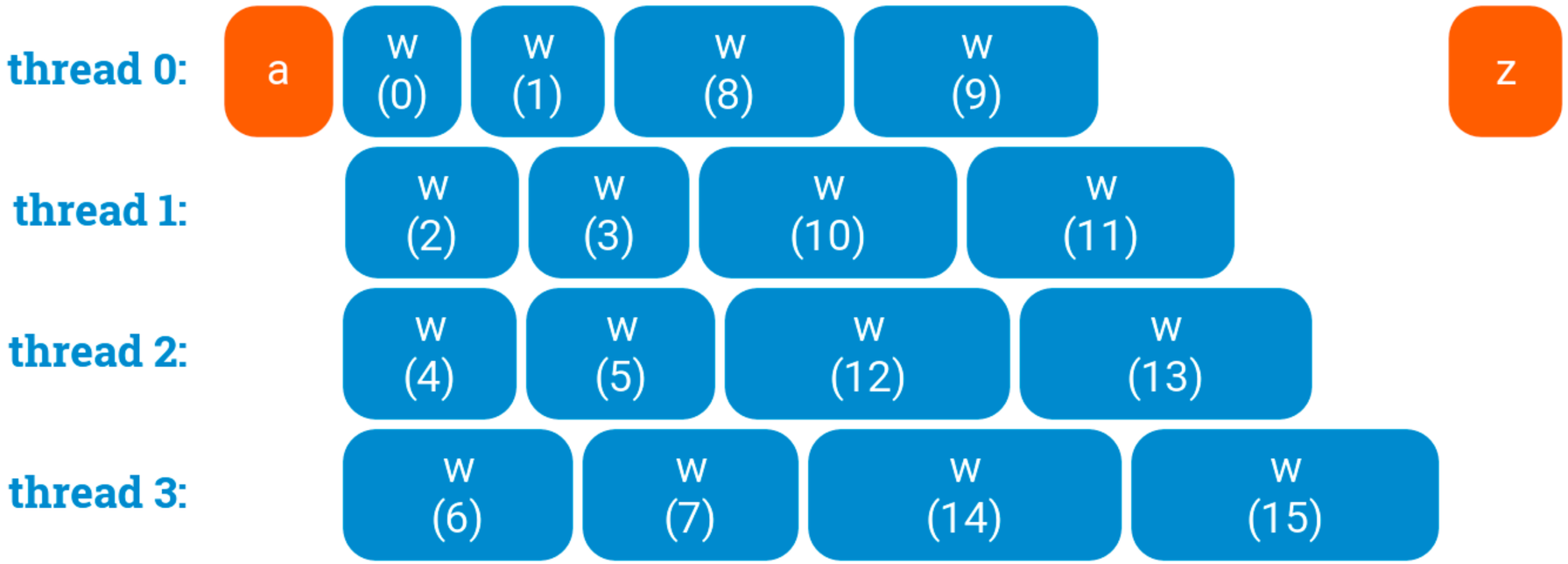
- This is the simplest.
- chunk size is fixed (and equal `chunk_size`).
- The chunk assignment is done using a round-robin assignment.
- Pro: low-overhead
- Con: will not work well if the runtime of each iteration varies significantly.

Example

#pragma omp parallel for schedule(static,1)



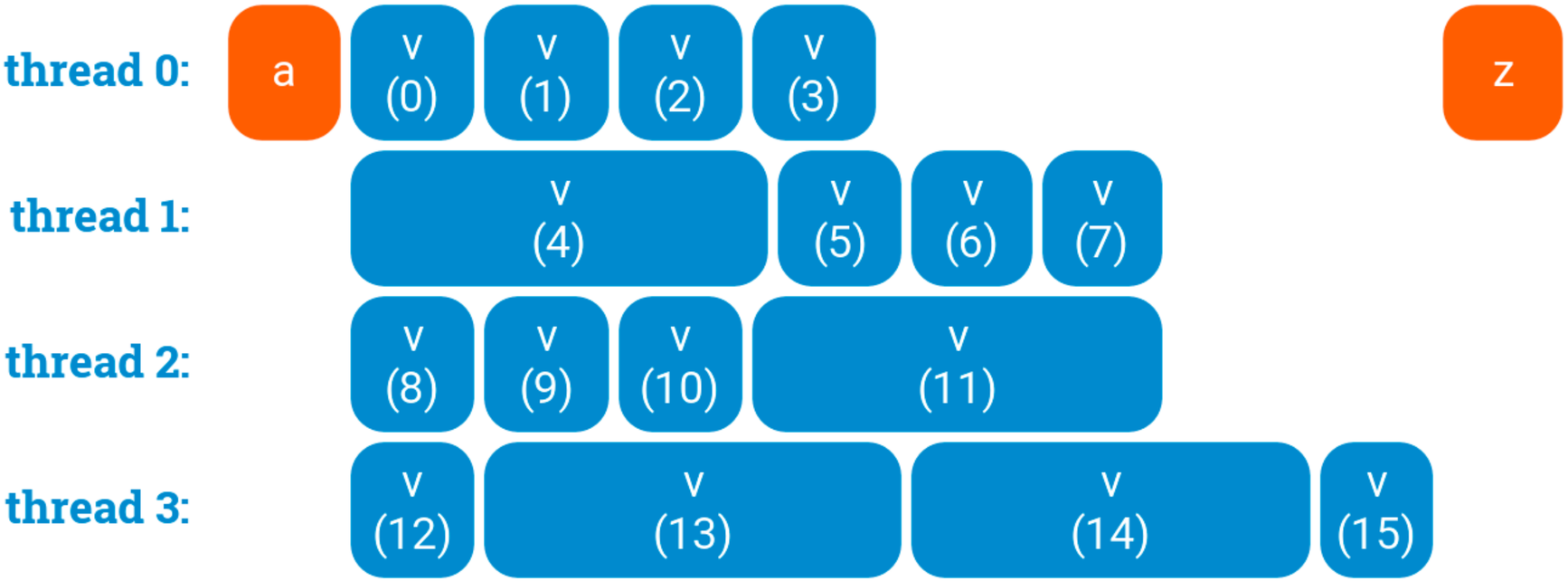
#pragma omp parallel for schedule(static,2)



dynamic

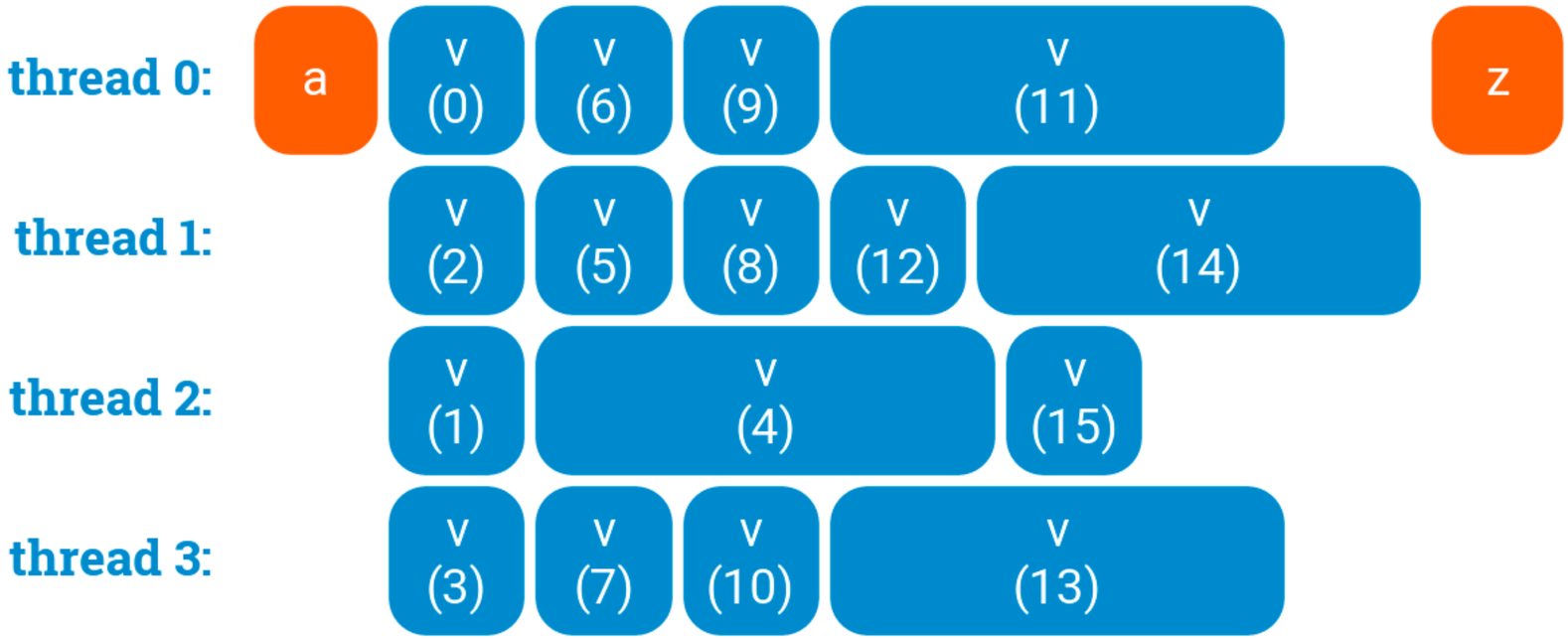
- Each thread executes a chunk.
- Then, requests another chunk until none remain.
- Pro: adapts to iterations with varying execution times.
- Con: overhead is more significant.

Example of uneven iteration runtime



Execution with `dynamic`

#pragma omp parallel for schedule(dynamic,1)



`guided` schedule

- The guided schedule is appropriate for the case in which the threads may arrive at varying times at a for construct with each iteration requiring about the same amount of work.

- For chunk size k, a typical implementation will assign
  ```
  q = ceiling(n/p)
  ```
  iterations to the first available thread, set n to the larger of n−q and p∗k, and repeat until all iterations are assigned.

- Pro: accommodates threads that arrive at varying times.

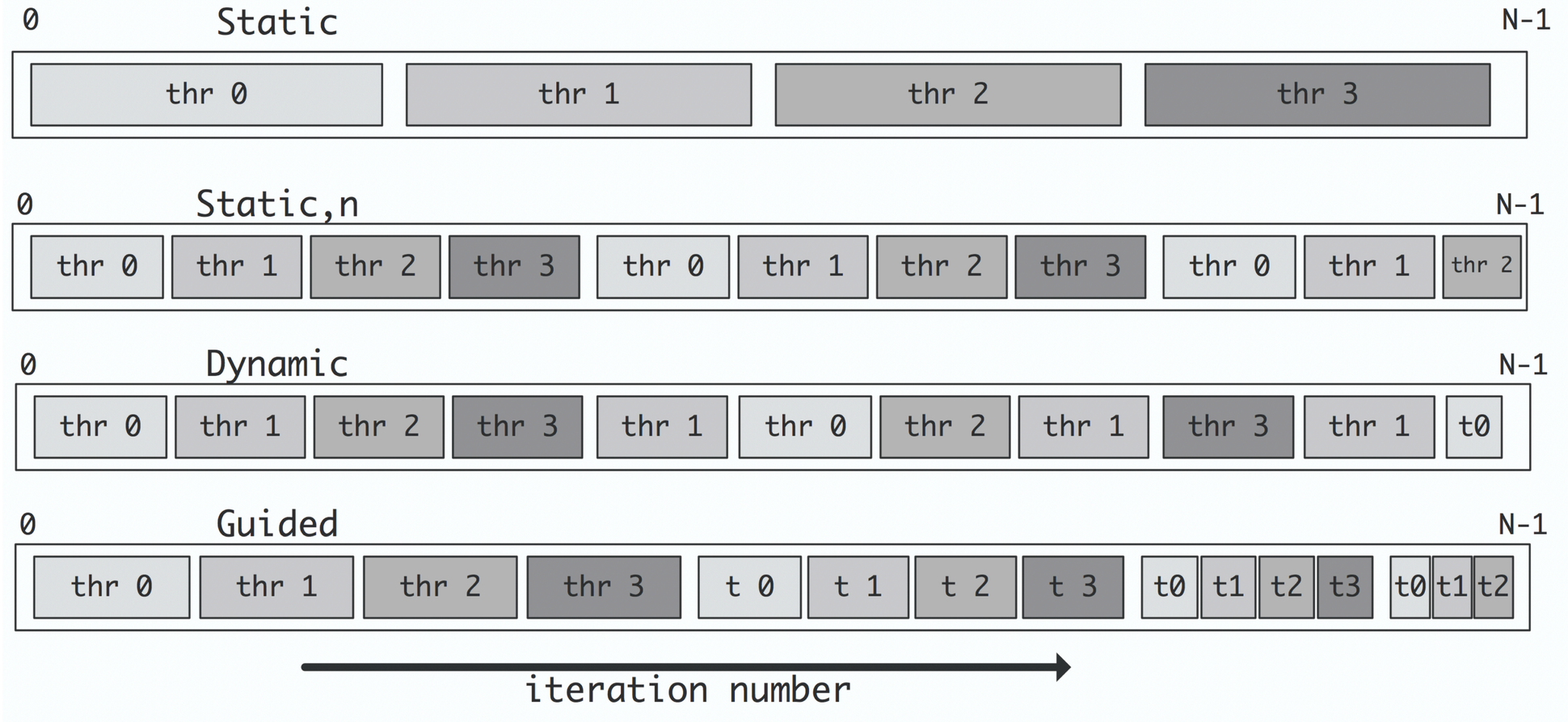
- Con: scheduling overhead; performance is case dependent; compare with `dynamic`

Properties of `guided`

- Like dynamic, the guided schedule guarantees that no thread waits at the barrier longer than it takes another thread to execute its final iteration.

- Among such schedules, the guided schedule is characterized by the property that it requires the **fewest synchronizations.**

Summary graphics



nowait

Avoid potentially unnecessary synchronization points at the end of a for loop.

```
#pragma omp parallel
  {
#pragma omp for nowait
    for (int i = 1; i < n; i++) b[i] = (a[i] + a[i - 1]) / 2.0;
#pragma omp for nowait
    for (int i = 0; i < m; i++) y[i] = sqrt(z[i]);
  }
```

collapse

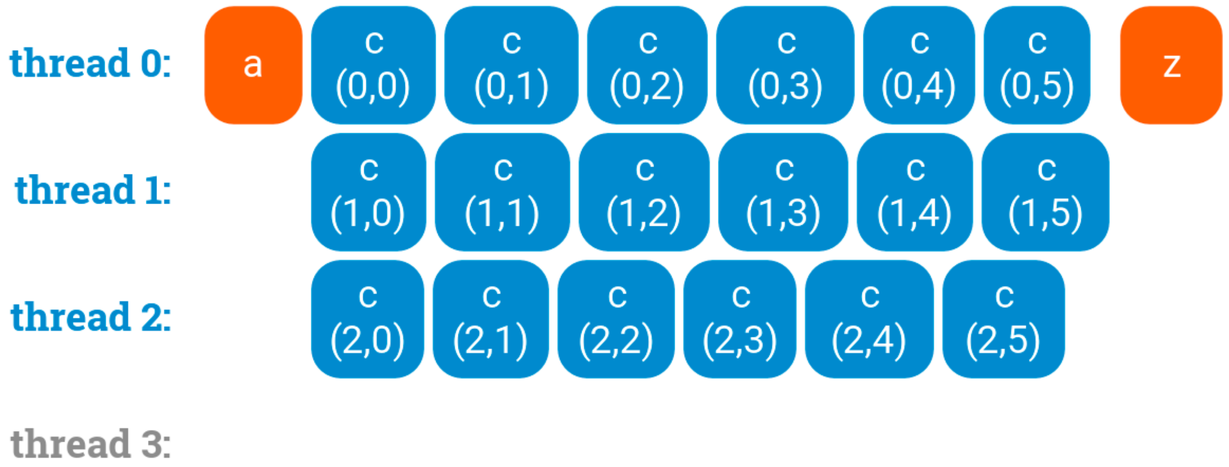
```
#pragma omp for collapse(2) private(i, k, j)
  for (int k = kl; k <= ku; k += ks)
    for (int j = jl; j <= ju; j += js)
      for (int i = il; i <= iu; i += is) {
        bar(a, i, j, k);
      }
```

Allows to parallelize multiple loops in a nest.

This is important when the loops are too short to be efficiently parallelized.

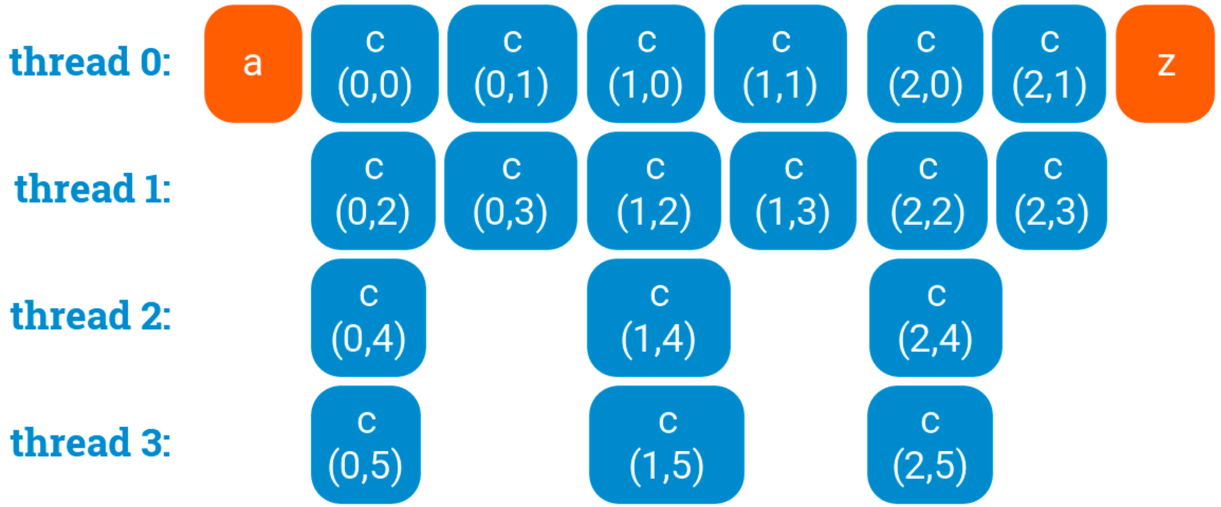
Example: short outer loop

```
  a();
#pragma omp parallel for
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 6; ++j) {
      c(i, j);
    }
  }
  z();
```



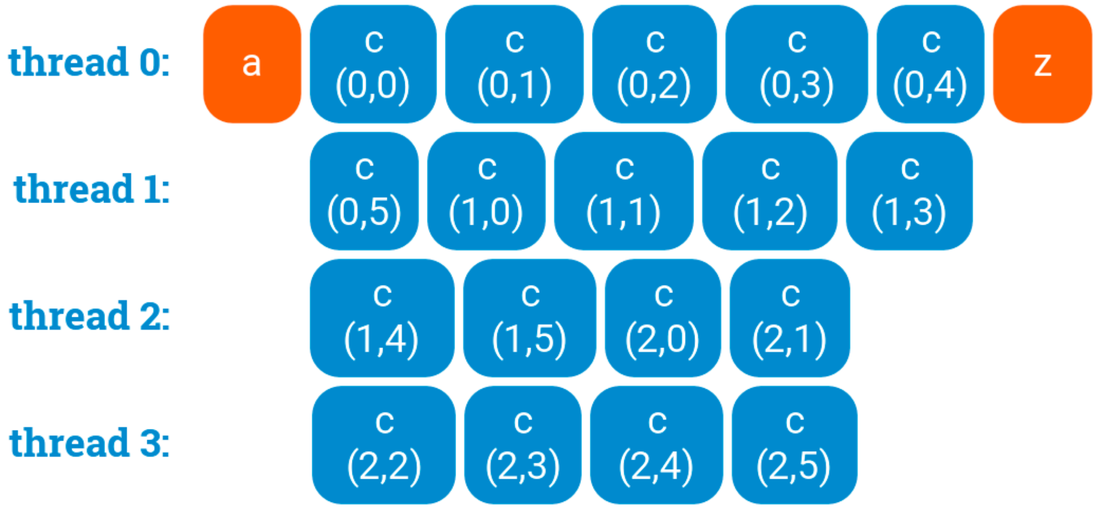
Example: inner loop; greater overhead

```
  a();
  for (int i = 0; i < 3; ++i) {
#pragma omp parallel for
    for (int j = 0; j < 6; ++j) {
      c(i, j);
    }
  }
  z();
```



With `collapse`

```
  a();
#pragma omp parallel for collapse(2)
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 6; ++j) {
      c(i, j);
    }
  }
  z();
```



Conditions for `collapse`

- The loops must form a rectangular iteration space and the bounds and stride of each loop must be invariant over all the loops.

- The loops must be perfectly nested; that is, there is no intervening code nor any OpenMP pragma between the loops which are collapsed.

OpenMP data sharing clause

- Recall in C++ threads:
  - Variables passed as argument to a thread are shared
  - Variables inside the function that a thread is executing are private to that thread
- OpenMP makes some reasonable default choices.
- But they can be changed using `shared` and `private`.
- See `shared_private.cpp`

Example of variable shared by default

```
  int shared_int = -1;

#pragma omp parallel
  {
    // shared_int is shared
    int tid = omp_get_thread_num();
    printf("Thread ID %2d | shared_int = %d\n", tid,
    shared_int);
    assert(shared_int == -1);
  }
```

Variable is made private inside the block

```
  int is_private = -2;

#pragma omp parallel private(is_private)
  {
    int tid = omp_get_thread_num();
    int rand_tid = rand();
    is_private = rand_tid;
    printf("Thread ID %2d | is_private = %d\n", tid, is_private);
    assert(is_private == rand_tid);
  }
```

Output

```
darve@icme-gpu1:~/2023/openMP$ srun -c 16 -p CME ./shared_private_openmp
Running main() from /home/darve/googletest-1.13.0/googletest/src/gtest_main.cc
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from sharing_clause
[ RUN      ] sharing_clause.shared
Thread ID  0 | shared_int = -1
Thread ID  2 | shared_int = -1
Thread ID  1 | shared_int = -1
Thread ID  3 | shared_int = -1
[       OK ] sharing_clause.shared (0 ms)
[ RUN      ] sharing_clause.private
Thread ID  3 | is_private = 1804289383
Thread ID  0 | is_private = 846930886
Thread ID  2 | is_private = 1681692777
Thread ID  1 | is_private = 1714636915
Main thread | is_private = -2
[       OK ] sharing_clause.private (0 ms)
[----------] 2 tests from sharing_clause (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 2 tests.
darve@icme-gpu1:~/2023/openMP$
```

Data-Sharing Attribute Clauses

- Most common:
  - `shared(list)`
  - `private(list)`
- Less common: `firstprivate, lastprivate, linear, …`

- **firstprivate:** declares one or more list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.

- **lastprivate:** declares one or more list items to be private, and causes the corresponding original list item to be updated after the end of the region.

- **linear:** see p. 323 of specifications; variable is private and has a linear relationship with respect to the iteration space of a loop associated with the construct.

More complicated demo example

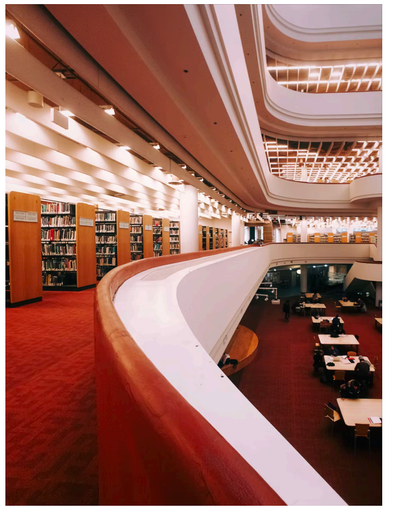
See `last_private.cpp`

`private, shared, firstprivate, lastprivate, copyin, threadprivate`

Summary

private	Specifies that each thread should have its own instance of a variable.
firstprivate	Specifies that each thread should have its own instance of a variable, and that the variable should be **initialized using the value before the parallel construct.**
lastprivate	Specifies that the enclosing context's version of the variable is **set equal to the private version of whichever thread executes the final for loop iteration.**
shared	Specifies that the variable should be shared among all threads.
reduction	Specifies that one or more variables that are private to each thread are the subject of a **reduction operation at the end of the parallel region.**
copyin	Allows threads to access the main thread's value, for a threadprivate variable.

References



- OpenMP Specifications: https://www.openmp.org/specifications/
- OpenMP examples: https://www.openmp.org/wp-content/uploads/openmp-examples-5.1.pdf
- Reference guides: https://www.openmp.org/resources/refguides/

Data affinity and NUMA

- We previously discussed the NUMA architecture.

- Example on `icme-gpu1; $ srun -c 1 -p CME lscpu`

```
Thread(s) per core:                  2
Core(s) per socket:                  8
L1d cache:                           256 KiB (8 instances)
L1i cache:                           256 KiB (8 instances)
L2 cache:                            4 MiB (8 instances)
L3 cache:                            128 MiB (8 instances)
NUMA node(s):                        4
NUMA node0 CPU(s):                   0,1,8,9
NUMA node1 CPU(s):                   2,3,10,11
NUMA node2 CPU(s):                   4,5,12,13
NUMA node3 CPU(s):                   6,7,14,15
Model name:                          AMD EPYC 7262 8-Core Processor
```

numactl information

```
darve@icme-gpu1:~/numactl/numactl-master$ srun -c 16 -p CME ./numactl -H
available: 4 nodes (0-3)
node 0 cpus: 0 1 8 9
node 0 size: 32070 MB
node 0 free: 26576 MB
node 1 cpus: 2 3 10 11
node 1 size: 32220 MB
node 1 free: 29175 MB
node 2 cpus: 4 5 12 13
node 2 size: 32254 MB
node 2 free: 31200 MB
node 3 cpus: 6 7 14 15
node 3 size: 32241 MB
node 3 free: 30907 MB
node distances:
node   0   1   2   3
  0:  10  12  12  12
  1:  12  10  12  12
  2:  12  12  10  12
  3:  12  12  12  10
darve@icme-gpu1:~/numactl/numactl-master$
```

NUMA performance

The memory latency distance between a node and itself is normalized to 10 (1.0x).

First touch policy

- The First Touch Placement Policy allocates the data page in the memory closest to the thread accessing this page for the first time.
- What does this mean for OpenMP programs?

First-touch optimization

How to leverage the first-touch policy?

Use the same parallelization strategy for the data initialization and computation parts

Example code with first-touch optimization

```
#pragma omp parallel for
    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j)
        {
            mat_a[i * size + j] = MatA(i, j);
            mat_b[i * size + j] = MatB(i, j);
        }

#pragma omp parallel for
    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j)
        {
            mat_c[i * size + j] = mat_a[i * size + j] * mat_b[i * size + j];
        }
```

First-touch

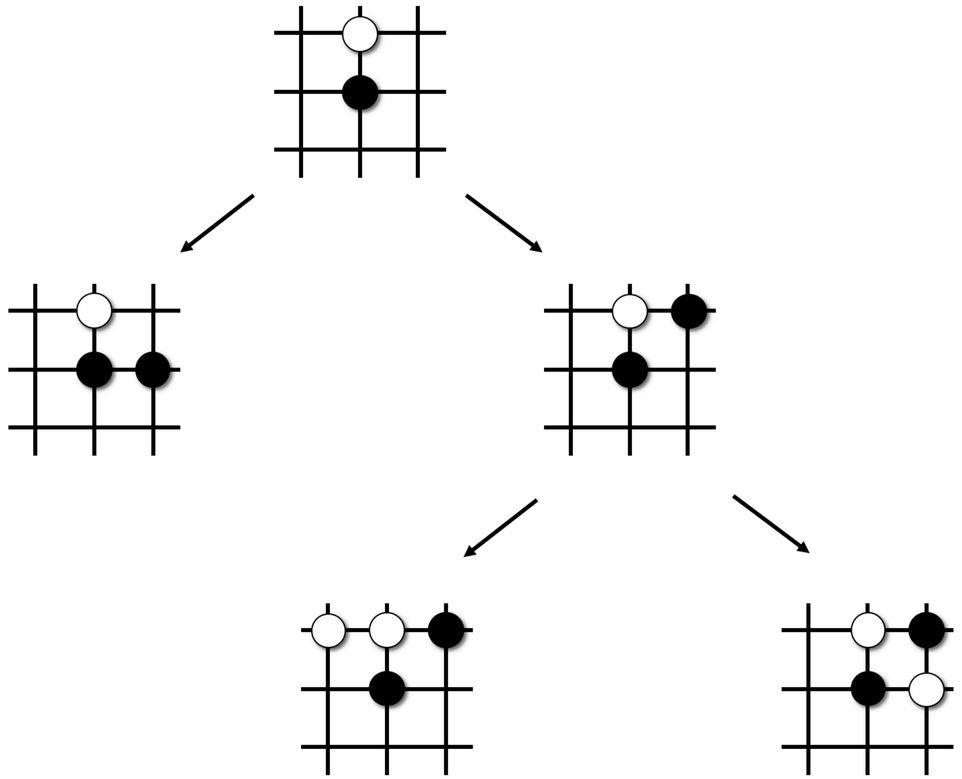
Low latency

OpenMP tasks

#pragma omp task

- Many situations require a more flexible way of expressing parallelism
- Example: tree traversal
- Let’s play a game of go.
- How would you describe the different ways to play the game?

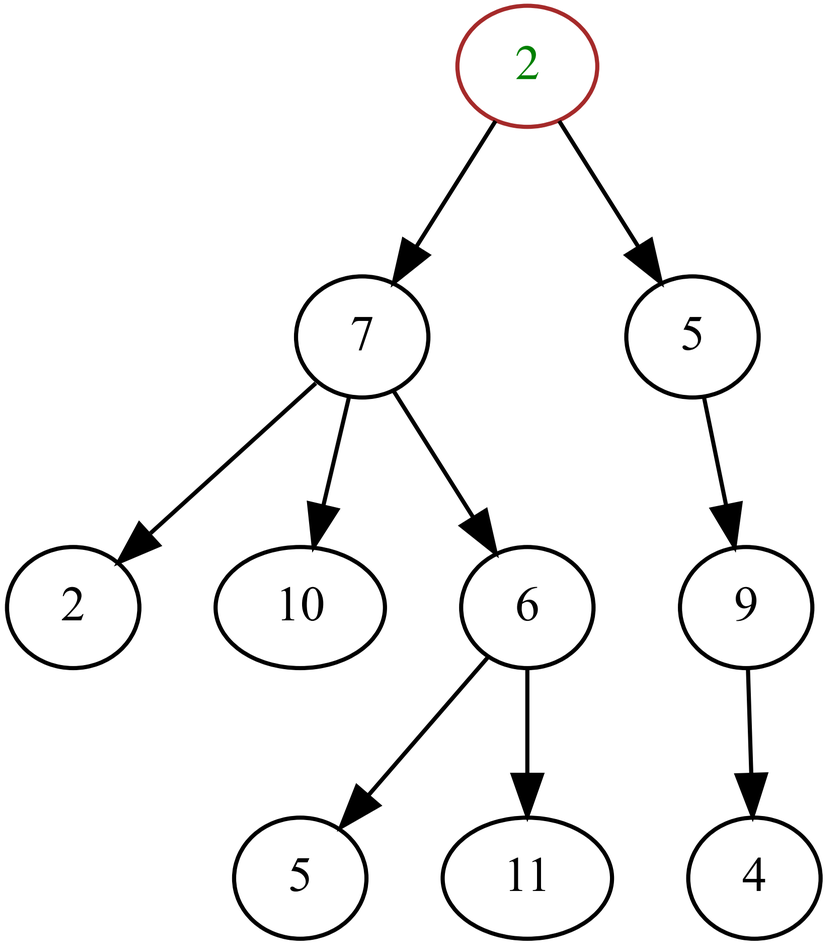
Playing Go



- Playing a game of go can be described using a tree structure.
- Traversing the tree and doing some computations is a key step in many algorithms.

Tree traversal

- Go through each node and execute some operation
- But, the tree is not full, e.g., number of child nodes varies
- As a result, the calculation is very unstructured.
- Work is discovered dynamically as the tree is traversed.
- A `for` loop is insufficient for this type of calculations.



omp tasks

- See `tree.cpp`
- Create tasks on the fly as we encounter new work to do.

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

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

Creating the parallel region

- We create a parallel region to have a pool of threads ready to do work.
- But the block of code is only executed by a single thread.

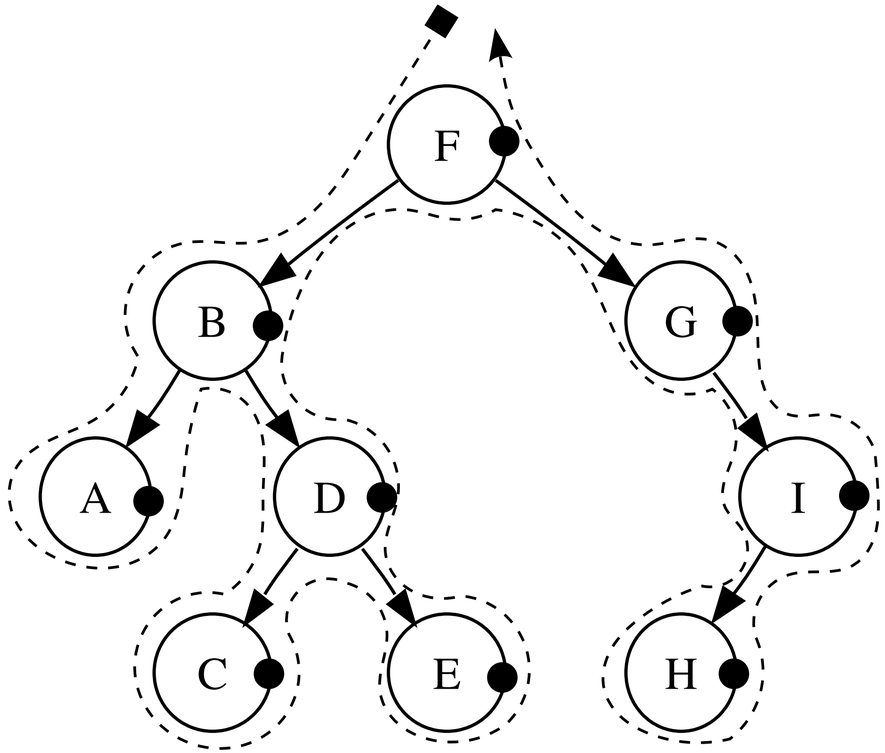
```
#pragma omp parallel
#pragma omp single
  // Only a single thread should execute this
  Traverse(root);
```

Task execution

- The encountering thread may immediately execute the task, or defer its execution.
- Any thread in the team may be assigned the task.
- This is managed by the OpenMP library.

Post-order traversal

- Let's now assume that we traverse the tree and based on the result of the children node, we do some calculation.
- This requires a synchronization. We have to wait for the children tasks to complete before we can proceed.



taskwait

- See `tree_postorder.cpp`
- `taskwait` is required to make sure left and right are up to date.

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

  if (curr_node->left)
#pragma omp task shared(left)
    left = PostOrderTraverse(curr_node->left);
  // Default attribute for task constructs is firstprivate
  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;
}
```

Synchronization constructs

- `taskwait` : specifies a wait on the completion of child tasks of the current task.
- `taskgroup` : specifies a wait on completion of child tasks of the current task and their descendent tasks.
- `barrier` : specifies an explicit barrier at the point at which the construct appears. All threads of the team that is executing the binding parallel region must execute the barrier region and complete execution of all explicit tasks bound to this parallel region before any are allowed to continue execution beyond the barrier.

Data sharing for tasks

- tasks are different from parallel `for` regions.
- In `for` loops, thread wait at the end of the loop.
- For tasks, the main thread just continues the execution and a worker thread takes care of the task.
- Default `shared` attribute is usually not the expected attribute.
- For tasks, the default attribute rules are different.

Default for tasks

- In a task generating construct, if no default clause is present, a variable that in the enclosing context is determined to be `shared` by all implicit tasks bound to the current team is `shared`.

- In a task generating construct, if no default clause is present, a variable for which the data-sharing attribute is not determined by the rule above is `firstprivate`.

`firstprivate`

- The `firstprivate` clause declares one or more list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.
- In this case, however, we want the variables `left` and `right` to be shared.
- A `shared` clause must be used.

```
  int left = 0, right = 0;

  if (curr_node->left)
#pragma omp task shared(left)
    left = PostOrderTraverse(curr_node->left);
  // Default attribute for task constructs is firstprivate
  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);
```

car race!

See `car_race.cpp`

Can you explain what each directive is doing?

What are the possible outputs for each case?

Is the output deterministic or undetermined?

What parts are deterministic, and what parts are undetermined?

Processing entries in a list using tasks

- We want to read and modify entries in a linked list using OpenMP tasks.
- Each entry can be updated independently of the others.
- See `list.cpp`

```
#pragma omp parallel
#pragma omp single
  {
    Node *curr_node = head;

    while (curr_node) {
      printf("Main thread. %p\n", (void *)curr_node);
#pragma omp task
      {
        // curr_node is firstprivate by default
        Wait();
        int tid = omp_get_thread_num();
        Visit(curr_node);
        printf("Task @%2d: node %p data %d\n", tid, (void *)curr_node,
               curr_node->data);
      }
      curr_node = curr_node->next;
    }
  }
```

- The variable `curr_node` is `firstprivate` by default.
- This is the correct data sharing attribute.

Recent additions to tasks

- Task priority.
- You may wish to assign higher priority to certain tasks.
- This is important in situations where for example, a task takes much longer than other tasks. In that case, we want OpenMP to assign a thread to that task as soon as possible.
- The priority clause specifies a hint for the task execution order. Among all tasks ready to be executed, higher priority tasks (those with a higher numerical priority-value) are recommended to execute before lower priority ones.

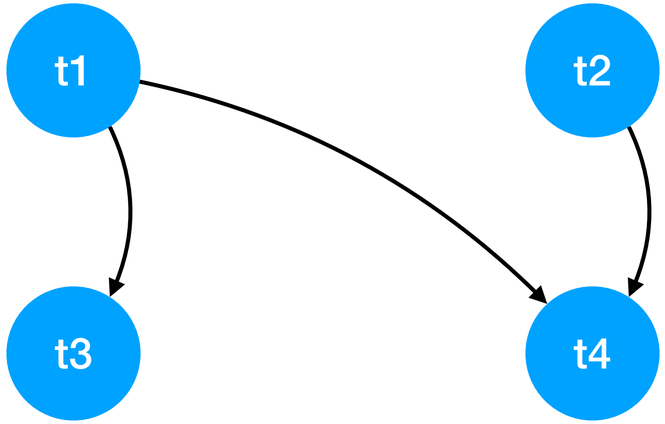
```
  for (i = 0; i < N; i++)
#pragma omp task priority(i)
    compute_array(&array[i * M], M);
```

What is the expected order of execution of the tasks?

`taskwait` and `depend`

- Although `taskwait` allows to synchronize tasks, it is not suitable when a fine-grained synchonization is required.

- For example: t3 can start as soon as t1 finishes; t4 needs to wait for t1 and t2.



depend

- This type of fine-grained dependency can be expressed using the `depend` clause.
- `depend(dep-type: x)`
- `dep-type` is one of
  - `in, out, inout, mutexinoutset`
  - `x` : variable or array section.

List of dependencies

dep-type	waits on	waits on	waits on	mutually exclusive
in		out/inout	mutexinoutset	
out/inout	in	out/inout	mutexinoutset	
mutexinoutset	in	out/inout		mutexinoutset

Example

```
  int x = 1;
#pragma omp parallel
#pragma omp single
  {
#pragma omp task shared(x) depend(in : x)
    printf("x = %d\n", x);
#pragma omp task shared(x) depend(out : x)
    x = 2;
  }
```

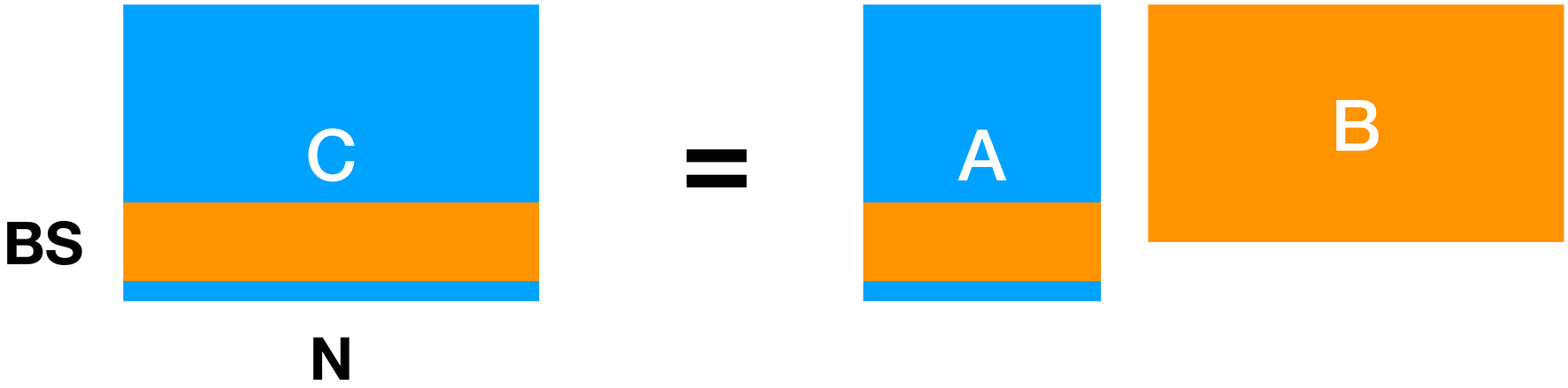
Always prints x = 1

```
  int d;
#pragma omp parallel
#pragma omp single
  {
    int a, b, c;
#pragma omp task depend(out : c) shared(c)
    c = 1; /* Task T1 */
#pragma omp task depend(out : a) shared(a)
    a = 2; /* Task T2 */
#pragma omp task depend(out : b) shared(b)
    b = 3; /* Task T3 */
#pragma omp task depend(in : a) depend(mutexinoutset : c) shared(a, c)
    c += a; /* Task T4 */
#pragma omp task depend(in : b) depend(mutexinoutset : c) shared(b, c)
    c += b; /* Task T5 */
#pragma omp task depend(in : c) shared(c)
    d = c; /* Task T6 */
  }
  printf("%d\n", d);
```

1. Does t4 wait on t2?
2. Does t5 wait on t3?
3. Does t5 wait on t4?
4. Does t5 wait on t1?
5. Can t4 and t5 execute at the same time?
6. Can t6 execute at the same time as t4?
7. What is the output of this program?

Matrix-matrix product

```
#pragma omp parallel
#pragma omp single
  for (int i = 0; i < N; i += BS) {
// i is firstprivate by default
#pragma omp task depend(in: A[i * N:BS * N], B) depend(inout: C[i * N:BS * N])
    for (int ii = i; ii < i + BS; ii++)
      for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
          C[ii * N + j] += A[ii * N + k] * B[k * N + j];
  }
```



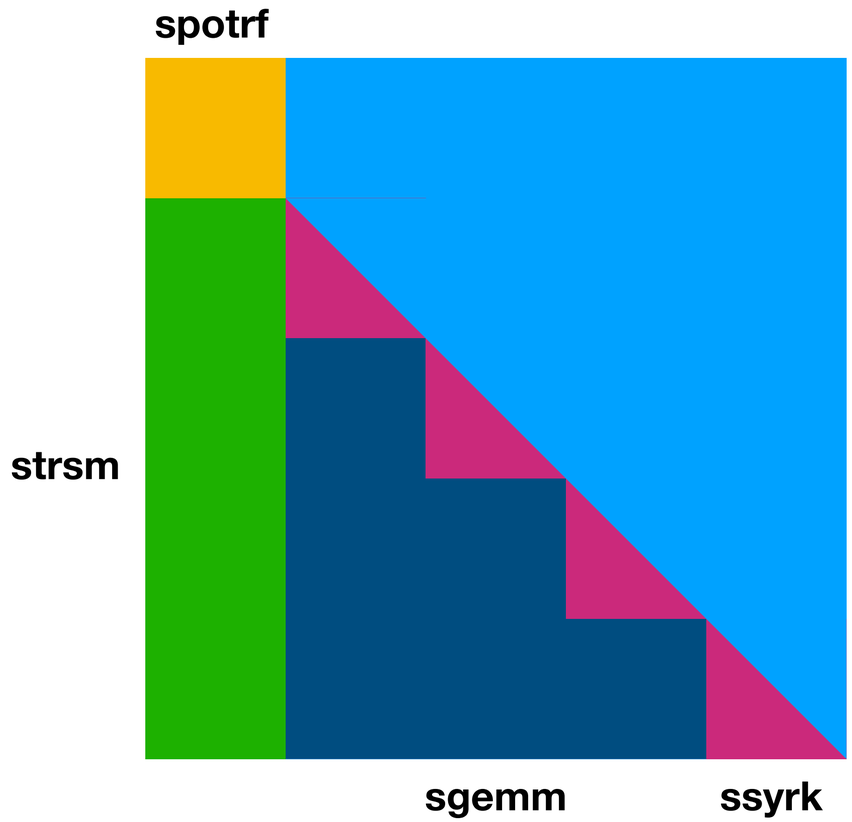
`depend` syntax

- `depend(in: A[i * N:BS * N])`
- Specifies the entries in A for which there is an `in` dependency.
- Syntax:

```
A[lower-bound : length : stride]
```

Cholesky algorithm

```
  for (int k = 0; k < NB; k++) {
#pragma omp task depend(inout : A[k][k])
    spotrf(A[k][k]);
    for (int i = k + 1; i < NB; i++)
#pragma omp task depend(in : A[k][k]) depend(inout : A[k][i])
      strsm(A[k][k], A[k][i]);
    // update trailing submatrix
    for (int i = k + 1; i < NB; i++) {
      for (int j = k + 1; j < i; j++)
#pragma omp task depend(in : A[k][i], A[k][j]) depend(inout : A[j][i])
        sgemm(A[k][i], A[k][j], A[j][i]);
#pragma omp task depend(in : A[k][i]) depend(inout : A[i][i])
      // diagonal block update
      ssyrk(A[k][i], A[i][i]);
    }
  }
```



OpenMP synchronization constructs

- Some race conditions have a simple pattern and can be resolved using openMP directives.
- Learning these patterns is important to make sure the code is correct and to get good performance.

Reduction

- This is the most common data race condition.

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

Reduction optimization

- Although += is a race condition, there is a strategy to generate parallel code.
- Have each thread compute a local sum, private to each thread.
- Then do an efficient reduction over partial results.
- OpenMP does not analyze the content of the loop. It relies on the user to correctly specifies the reduction operator.
- The reduction operator is used to:
  - Correctly initialize sum
  - Correctly combine the partial results after the `for` loop is complete.

Initializers and combiners

Identifier	Initializer	Combiner
`+`	`omp_priv = 0`	`omp_out += omp_in`
`-`	`omp_priv = 0`	`omp_out += omp_in`
`*`	`omp_priv = 1`	`omp_out *= omp_in`
`&`	`omp_priv = ~0`	`omp_out &= omp_in`
`|`	`omp_priv = 0`	`omp_out |= omp_in`
`^`	`omp_priv = 0`	`omp_out ^= omp_in`
`&&`	`omp_priv = 1`	`omp_out = omp_in && omp_out`
`||`	`omp_priv = 0`	`omp_out = omp_in || omp_out`
`max`	`omp_priv =` *Least representable number in the reduction list item type*	`omp_out = omp_in > omp_out ? omp_in : omp_out`
`min`	`omp_priv =` *Largest representable number in the reduction list item type*	`omp_out = omp_in < omp_out ? omp_in : omp_out`

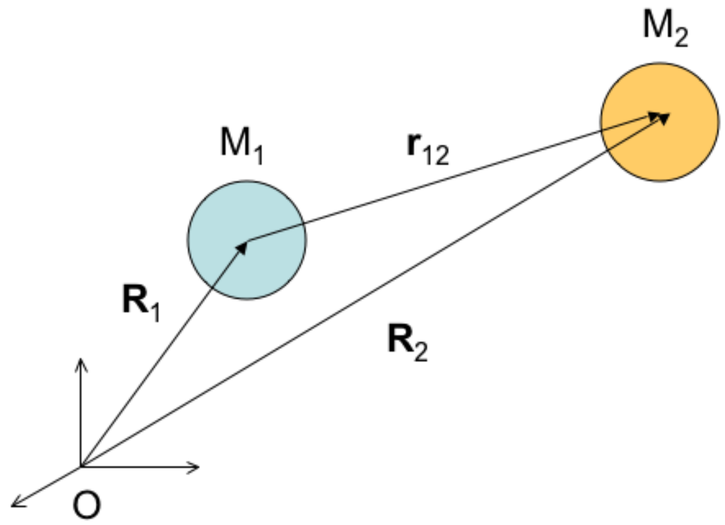
Example: entropy calculation

See `entropy.cpp`

Atomic operations

- There are cases where a similar += operation need to be computed but `reduction` does not apply.
- In that case, we need to use an `atomic` clause.
- Atomic operations should be: `+, *, -, /, &, ^, |, <<, or >>`
- But it is not as efficient as the `reduction` clause.
- The atomic construct ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values.

`atomic` example



N-body calculation.

Summation of forces leads to a race condition that can be resolved with an `atomic` clause.

See `atomic.cpp`

`atomic` is required both for `i` and `j` in order to guarantee a correct result.

```
#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_;
    }
```

`critical`

- Similar to mutex, there might be regions of code that can be excuted by **a single thread at a time** only.
- `critical` : restricts execution of the associated structured block to a single thread at a time.
- See `critical.cpp`

Example: insertion in set

```
  set<int> m;
#pragma omp parallel for
  for (int i = 2; i <= n; ++i) {
    bool is_prime = is_prime_test(i);
#pragma omp critical
    if (is_prime) m.insert(i);
    // Save this prime; this requires a critical clause.
    // Only a single thread can execute this line at a time.
  }
```

Insertion leads to a race condition that needs to be protected using `critical`.

Performance considerations: highlight

- Use **scheduling** clause for for loops.
- Prefer: reduction > atomic > critical.
- Use **`nowait`** whenever possible.
- Avoid creating parallel regions one after the other if they can safely be merged.
- **Memory access** is key for performance. **Data placement** matters for CC-NUMA processors.
- See Mastering OpenMP Performance.

Other topics (not covered):

```
affinity, target, simd, locks
```

5.0 OpenMP specification

5.2 is much less readable.