What is OpenMP?

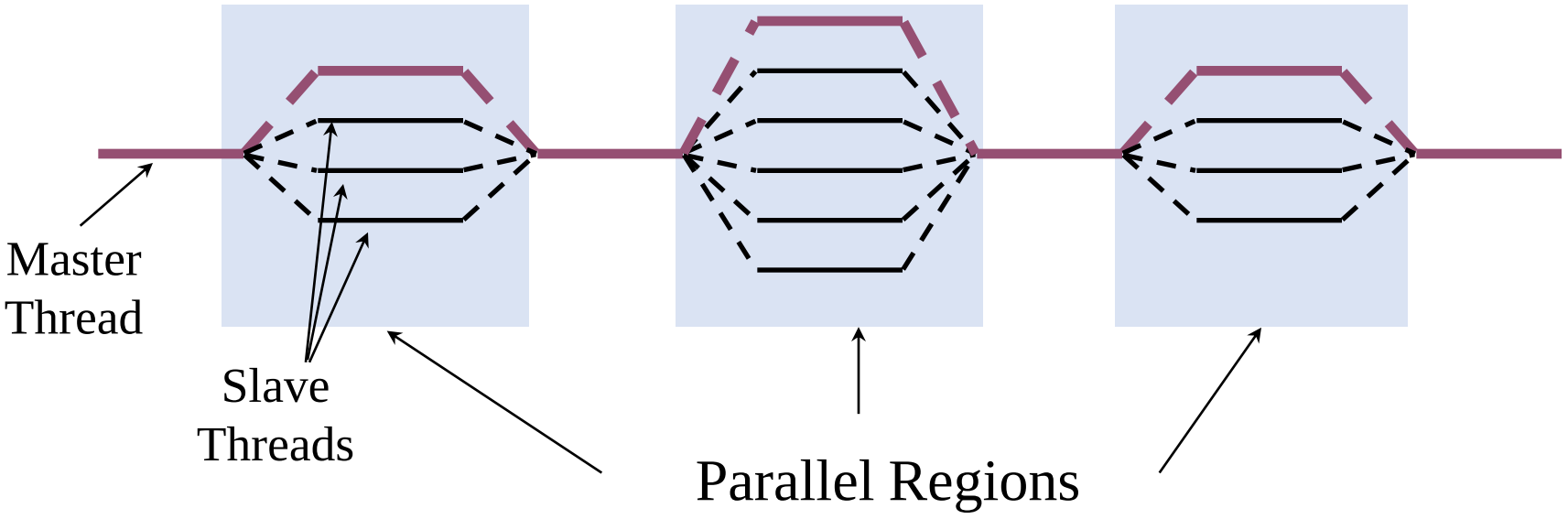
- **Parallel Computing** gives you more performance to throw at your problems.
  - Parallel computing is when a program uses concurrency to either:
    - Decrease the runtime for the solution to a problem.
    - Increase the size of the problem that can be solved
- **OpenMP** provides a **standard** for shared memory programming for scientific applications.
  - Has specific support for scientific application needs (*unlike Pthreads*).
  - Rapidly gaining acceptance among vendors and developers.
  - See *http://www.openmp.org* for more info.

OpenMP API Overview

- OpenMP - portable shared memory parallelism
  - An API for Writing Multithreaded Applications
    - API is a **set** of:
      - 1) compiler directives inserted in the source program
      - in addition to some 2) library functions and 3) environment variables
- OpenMP: Programming Model
  - **Fork-Join** Parallelism:
    - Master thread spawns/generates a team of threads as needed.
    - Parallelism is added incrementally:
      - i.e. the **sequential** program **evolves** into a **parallel** program

API Semantics

- Master thread executes sequential code.
- Master and slaves execute parallel code.
- **Note**:
  - very similar to fork-join semantics of *Pthreads* create/join primitives.



How is OpenMP typically used?

- OpenMP is usually used to parallelize loops:
  - Find your most **time-consuming loops**.
  - Split them up between threads.

Split-up this loop between multiple threads

```
void main()
{
  double Res[1000];

  for( int i=0;i<1000;i++ ) {
       do_huge_comp(Res[i]);
  }
}
```

Sequential Program

```
void main()
{
  double Res[1000];
  #pragma omp parallel for
  for( int i=0;i<1000;i++ ) {
     do_huge_comp(Res[i]);
  }
}
```

Parallel Program

OpenMP: How do threads interact?

- **OpenMP** is a **shared memory** model.
  - Threads communicate by sharing variables.
- Unintended sharing of data can lead to race conditions:
  - **race condition**: when the ***program's outcome*** changes as the threads are scheduled differently.
- To control race conditions:
  - Use **synchronization** to protect data conflicts.
- **Synchronization** is **expensive** so:
  - Often, we intend to change/control how data is stored to minimize the need for synchronization.

OpenMP Directives

- OpenMP implementation
  - Compiler directives, Library, and Environment, Unlike Pthreads (*purely a library*).

- Parallelization directives:
  - **`parallel region`**
  - **`parallel for`**
- Data environment directives:
  - **`shared`**, **`private`**, **`threadprivate`**, **`reduction`**, etc.
- Synchronization directives:
  - **`barrier`**, **`critical`**, **`atomic`**

- General Rules about Directives
  - They always apply to the *next statement* (or block of statements), which must be a structured block.

```
#pragma omp …
    Statement
#pragma omp …
{
    statement1;
    statement2;
    statement3;
}
```

OpenMP: Contents

- OpenMP's **constructs** fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
  - Some Advanced Features

OpenMP Parallel Region

```
#pragma omp parallel
```

- A number of threads are spawned/created at the entry.
- Each thread executes the *same code* (SPMD model).
- The master thread *waits all other threads at the end*. (join)
- Very similar to a number of *create/join's* with the **same function** in *Pthreads*.

OpenMP: Parallel Regions

- You create threads in OpenMP with the "`omp parallel`" pragma.
- For example, to create a 4 thread Parallel Region:

Each thread *redundantly* executes the code within the structured block

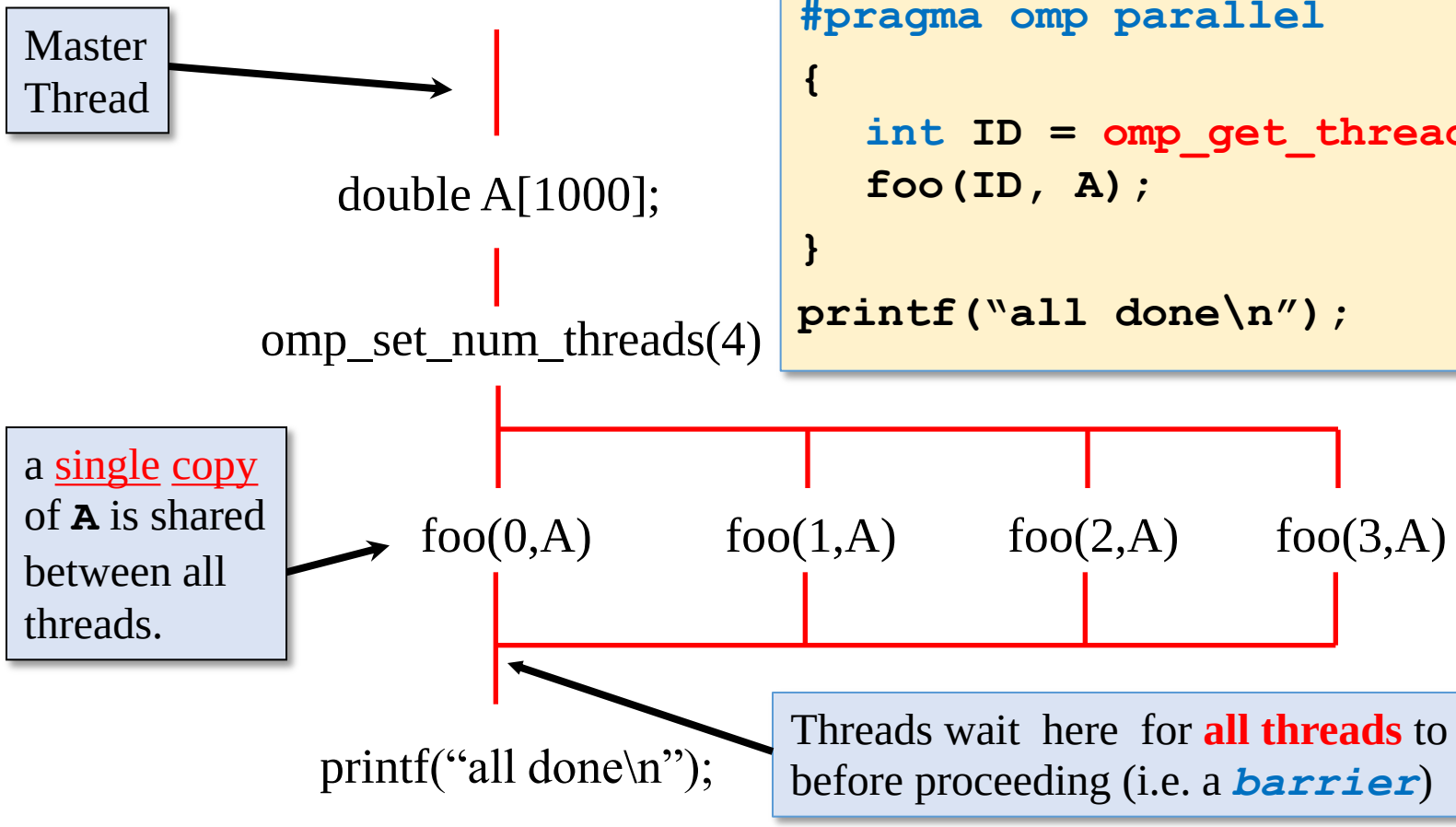
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_thread_num();
    foo(ID,A);
}
```

Each thread calls `foo(ID,A)` for ID = 0 to 3

OpenMP: Parallel Regions

- Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   foo(ID, A);
}
printf("all done\n");
```



#pragma omp parallel [clause list] Typical clauses in [clause list]

• Conditional parallelization

– if (scalar expression)

• Determine whether the parallel construct creates threads

• Degree of concurrency

– num_threads (integer expresson)

• number of threads to create

• Date Scoping

– private (variable list) • Specifies variables local to each thread

– firstprivate (variable list) • Similar to the private • Private variables are initialized to variable value before the parallel directive

– shared (variable list) • Specifies variables that are shared among all the threads

– default (data scoping specifier) • Default data scoping specifier may be shared or none

Example: #pragma omp parallel if (is_parallel == 1) num_threads(8) shared (var_b) private (var_a) firstprivate (var_c) default (none) { /* structured block */ }

- if (is_parallel == 1) num_threads(8) – If the value of the variable is_parallel is one, create 8 threads
- shared (var_b) – Each thread shares a single copy of variable b
- private (var_a) firstprivate (var_c) – Each thread gets private copies of variable var_a and var_c – Each private copy of var_c is initialized with the value of var_c in main thread when the parallel directive is encountered
- default (none) – Default state of a variable is specified as none (rather than shared) – Singals error if not all variables are specified as shared or private

OpenMP: Contents

- OpenMP's **constructs** fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
  - Some Advanced Features

OpenMP: Work-Sharing Constructs

- The "**for**" Work-Sharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
#pragma omp for
for ( I=0; I<N; I++ )
{
      NEAT_STUFF(I);
}
```

```
#pragma omp parallel for
for ( I=0; I<N; I++ )
{
      NEAT_STUFF(I);
}
```

… OR …

By default, there is a **barrier** at the end of the "**omp for**".

Use the "**nowait**" clause to **turn off (***disable it***)** the barrier.

Work Sharing Constructs

A motivating example

Sequential code

```
for(i=0;i<N;i++) { a[i]=a[i] + b[i]; }
```

OpenMP parallel region

```
#pragma omp parallel
{
  int id, i, Nthrds, istart, iend;
  id = omp_get_thread_num();
  Nthrds = omp_get_num_threads();
  istart = id * N / Nthrds;
  iend = (id+1) * N / Nthrds;
  for(i=istart; i<iend; i++){
     a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a work-sharing for-construct

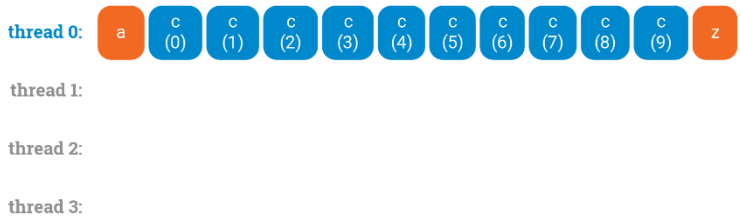
```
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

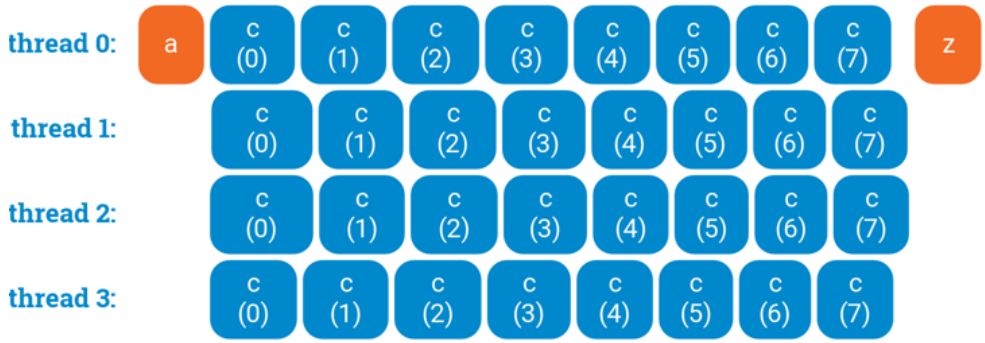
OpenMP Directive: parallel for

```
a();

for(int i=0; i<10; ++i)
{
        c(i);
}
z();
```

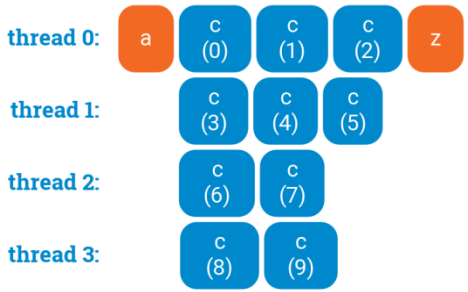
```
omp_set_num_threads(4);
a();
#pragma omp parallel
for( int i = 0; i < 8; ++i)
{
     c(i);
}
z();
```





OpenMP Directive: parallel for

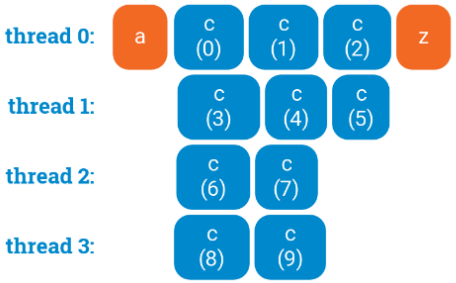
```
omp_set_num_threads(4);
a();
#pragma omp parallel
{
    #pragma omp for
    for( int i= 0; i<10; ++i)
    {
        c(i);
    }
}
z();
```




```
omp_set_num_threads(4);
a();

#pragma omp parallel for
for( int i=0; i<10; ++i)
{
        c(i);
}

z();
```



It is just a shorthand

Work Sharing Directives

- Always occur within a parallel region directive
- Two principal:
  - parallel for
  - parallel section

OpenMP Parallel For
-- Each thread executes a subset of the iterations
-- All threads wait at the end of the parallel for

```
#pragma omp parallel
#pragma omp for
for( … ) { … }
```

is equivalent to

```
#pragma omp parallel for
for ( … ) { … }
```

Note that: Same for `parallel sections`

```
#pragma omp parallel
{
  #pragma omp sections
  {
      ……
      #pragma omp section
      { …… }
      #pragma omp section
      { …… }
       ……
  }
}
```

Example: Matrix Multiply

Sequential Approach

```
for( i=0; i<n; i++ )
   for( j=0; j<n; j++ ) {
       c[i][j] = 0.0;
       for( k=0; k<n; k++ )
            c[i][j] += a[i][k] * b[k][j];
   }
```

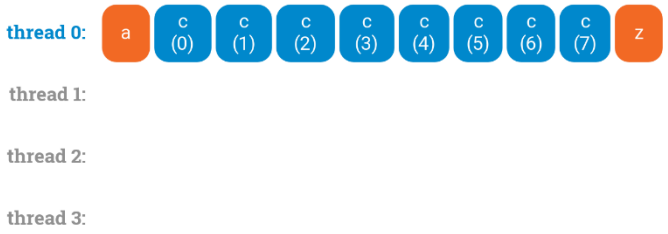
OpenMP Based Parallel Approach

```
#pragma omp parallel for
for( i=0; i<n; i++ )
   for( j=0; j<n; j++ ) {
       c[i][j] = 0.0;
       for( k=0; k<n; k++ )
            c[i][j] += a[i][k] * b[k][j];
   }
```

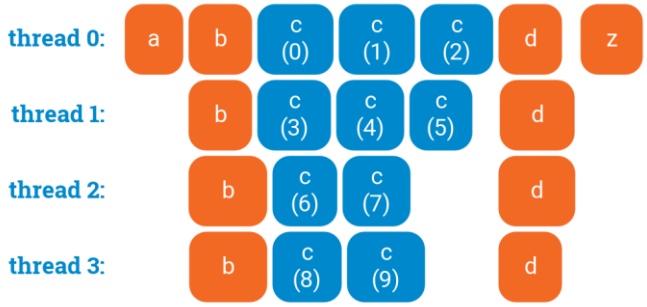
OpenMP Directive: parallel for

Common mistakes in the use `omp parallel or omp for`

```
a();
#pragma omp for
for (int i=0; i<8; ++i)
{
    c(i);
}
z();
```




```
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i=0; i<10; ++i) {
        c(i);
    }
    d();
}
z();
```



OpenMP **parallel for**

Waiting / No Waiting

Multiple Work Sharing Directives

May occur within a **single parallel region**

All threads **wait at the end** of the **first for**.

```
#pragma omp parallel
{
        ……
    #pragma omp for
    for( ; ; ) { … }
        ……
    #pragma omp for
    for( ; ; ) { … }
        ……
}
```

The `nowait` Qualifier

Threads proceed to second **for** without waiting.

```
#pragma omp parallel
{
    #pragma omp for nowait
    for( ; ; ) { … }
    #pragma omp for
    for( ; ; ) { … }
}
```

Note the Difference between ...

```
#pragma omp parallel
{
    #pragma omp for
    for( ; ; ) { … }
    foo();
    #pragma omp for
    for( ; ; ) { … }
}
```

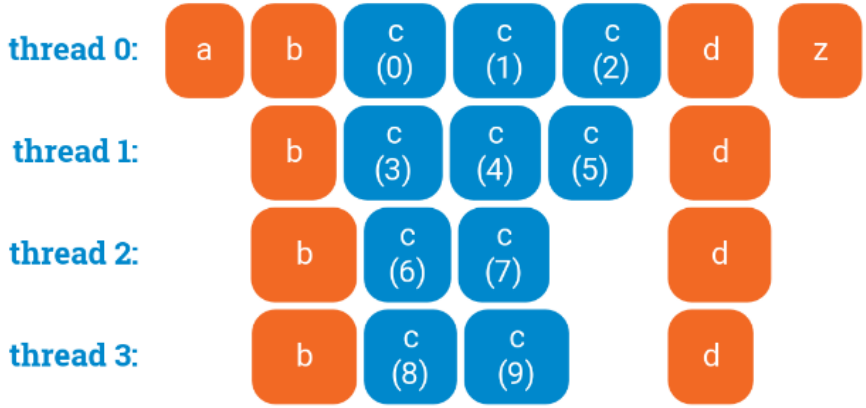
… and ...

```
#pragma omp parallel for
for( ; ; ) { … }
foo();
#pragma omp parallel for
for( ; ; ) { … }
```

OpenMP parallel for loops: waiting

- In a `parallel` region, OpenMP will automatically wait for all threads to finish before execution continues.
- There is also a *synchronization point* after each `omp for` loop;
  - here no thread will execute `d()` until all threads are done with the loop

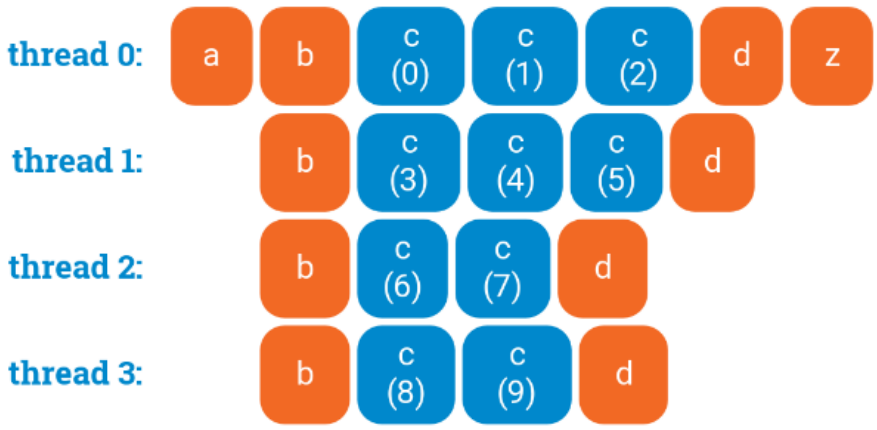
```
a();
#pragma omp parallel
{
  b();
  #pragma omp for
  for (int i=0; i<10; ++i)
  {
       c(i);
  }
  d();
}
z();
```



OpenMP parallel for loops: waiting

- However, if you do not need synchronization after the loop, you can disable it with nowait:

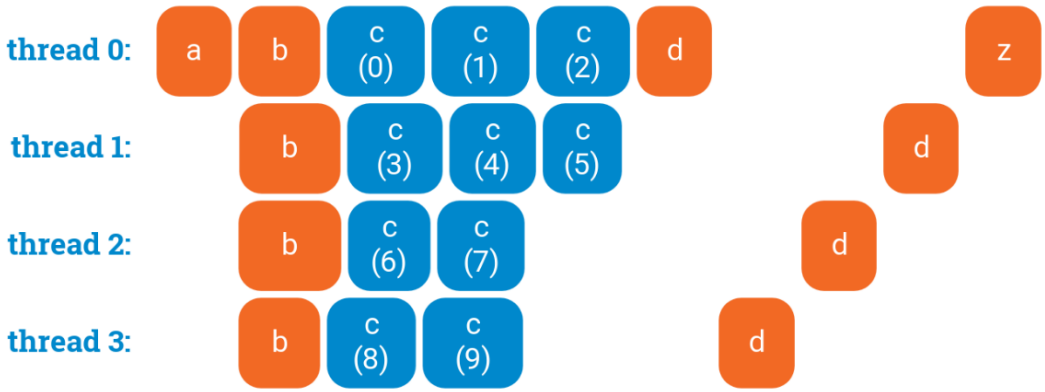
```
a();
#pragma omp parallel
{
  b();
  #pragma omp for nowait
  for (int i=0; i<10; ++i)
  {
       c(i);
  }
  d();
}
z();
```



OpenMP parallel for loops:
Interaction with critical sections

If you need a critical section after a loop, note that normally OpenMP will first wait for all threads to finish their loop iterations before letting any of the threads to enter a critical section:

```
a();
#pragma omp parallel
{
  b();
  #pragma omp for
  for (int i=0; i<10; ++i)
  {
     c(i);
  }
  #pragma omp critical
  {
    d();
  }
}
z();
```

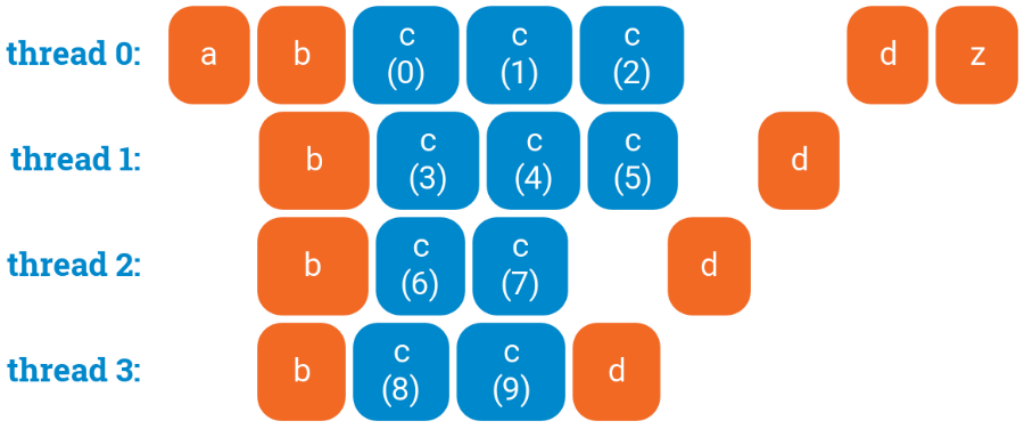


OpenMP parallel for loops:
Interaction with critical sections

You can disable waiting, so that some threads can start doing post-processing early. This would make sense if, e.g., **d()** updates some global data structure based on what the thread computed in its own part of the parallel for loop:

```
a();
#pragma omp parallel
{
  b();
  #pragma omp for nowait
  for (int i=0; i<10; ++i)
  {
     c(i);
  }
  #pragma omp critical
  {
    d();
  }
}
z();
```

Notice



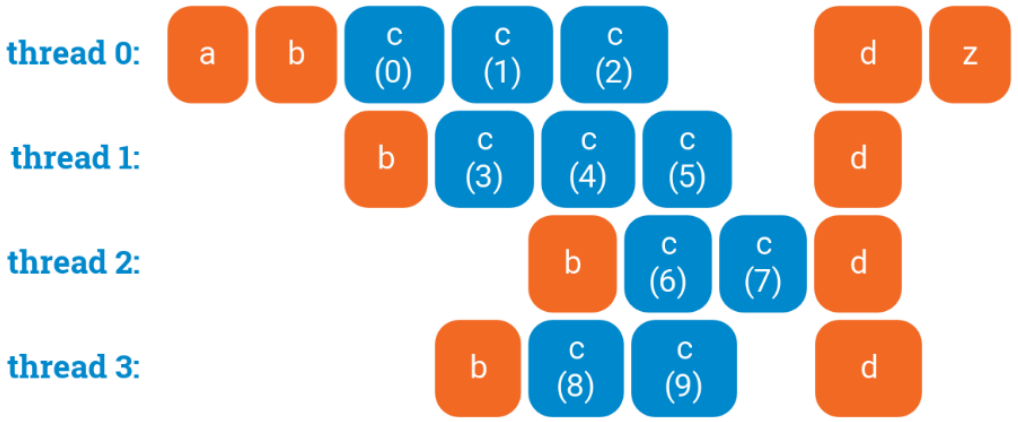
OpenMP parallel for loops:

No waiting before a loop

Now, note that there is no synchronization point *before the loop starts*.

If threads reach the for loop at different times, they can start their own part of the work as soon as they are there, without waiting for the other threads:

```
a();
#pragma omp parallel
{
  #pragma omp critical
  {
    b();
  }
  #pragma omp for
  for (int i=0; i<10; ++i)
  {
    c(i);
  }
  d();
}
z();
```



References

- OpenMP topic: Loop parallelism
  - https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html
- A “Hands-on” Introduction to OpenMP
  - https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf
- OpenMP in a nutshell
  - http://www.bowdoin.edu/~ltoma/teaching/cs3225-GIS/fall17/Lectures/openmp.html
- #pragma omp parallel (IBM)
  - https://www.ibm.com/support/knowledgecenter/SSGH3R_13.1.3/com.ibm.xlcpp1313.aix.doc/compiler_ref/prag_omp_parallel.html
- OpenMP Directives (Microsoft)
  - https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=vs-2019
- Guide into OpenMP: Easy multithreading programming for C++
  - https://bisqwit.iki.fi/story/howto/openmp/