HOWTOCFD23

Introduction to OpenMP

Instructor: Niranjan S. Ghaisas

OpenMP

- Standard for shared-memory parallel programming
- Anybody can implement these standards: gcc, intel, PGI, …

- Detailed Specifications:
  - https://www.openmp.org/

- Extensions to high-level languages (e.g. C, Fortran, C++)
- High-level ‘wrapper’ to an internal implementation of threads, e.g. POSIX threads

- Practically: add a set of compiler directives to a serial code

OpenMP

- Standard for shared-memory parallel programming
- Anybody can implement these standards: gcc, intel, PGI, …

- Extensions to high-level languages (e.g. C, Fortran, C++)

- History:

OpenMP Version	Year of Release	GNU C Support	Intel C Support
3.0	2008	gcc 4.4.0 +	icc 12.0+
4.0	2013	gcc 4.9.1 +	icc 15.0 +
4.5	2015	gcc 7.1 +	icc 17.0 +
5.0	2019	gcc 9.1 +	icc 19.1 +

OpenMP

- Threads:
  - A strand of execution within a *process*
  - Process: collection of instructions + memory (stack and heap)
  - Threads: parts of the process with same heap memory but individual stack memory
  - A process can have one or multiple threads

- Compiler Directives:
  - Special instructions to compiler (sometimes the preprocessor, e.g. #define, #include)
  - Specific directive used in OpenMP is

    #pragma omp parallel

OpenMP: Workflow

```
int main(){
    // C code starts here

    int i;
    ……
    {
      // one thread working
everywhere
          ……
    }
……
    return 0;
}
```

Program Execution

```
int main(){
    // C code starts here

    int i;
    …..
    #pragma omp parallel {
        // multiple threads
working here
          …..
    }
…...
    return 0;
}
```

OpenMP: Workflow

```c
int main(){
    // C code starts here

    int i;
    ……
    {
      // one thread working everywhere
          ……
    }
……
    return 0;
}
```

- gcc myprog.c
- ./a.out

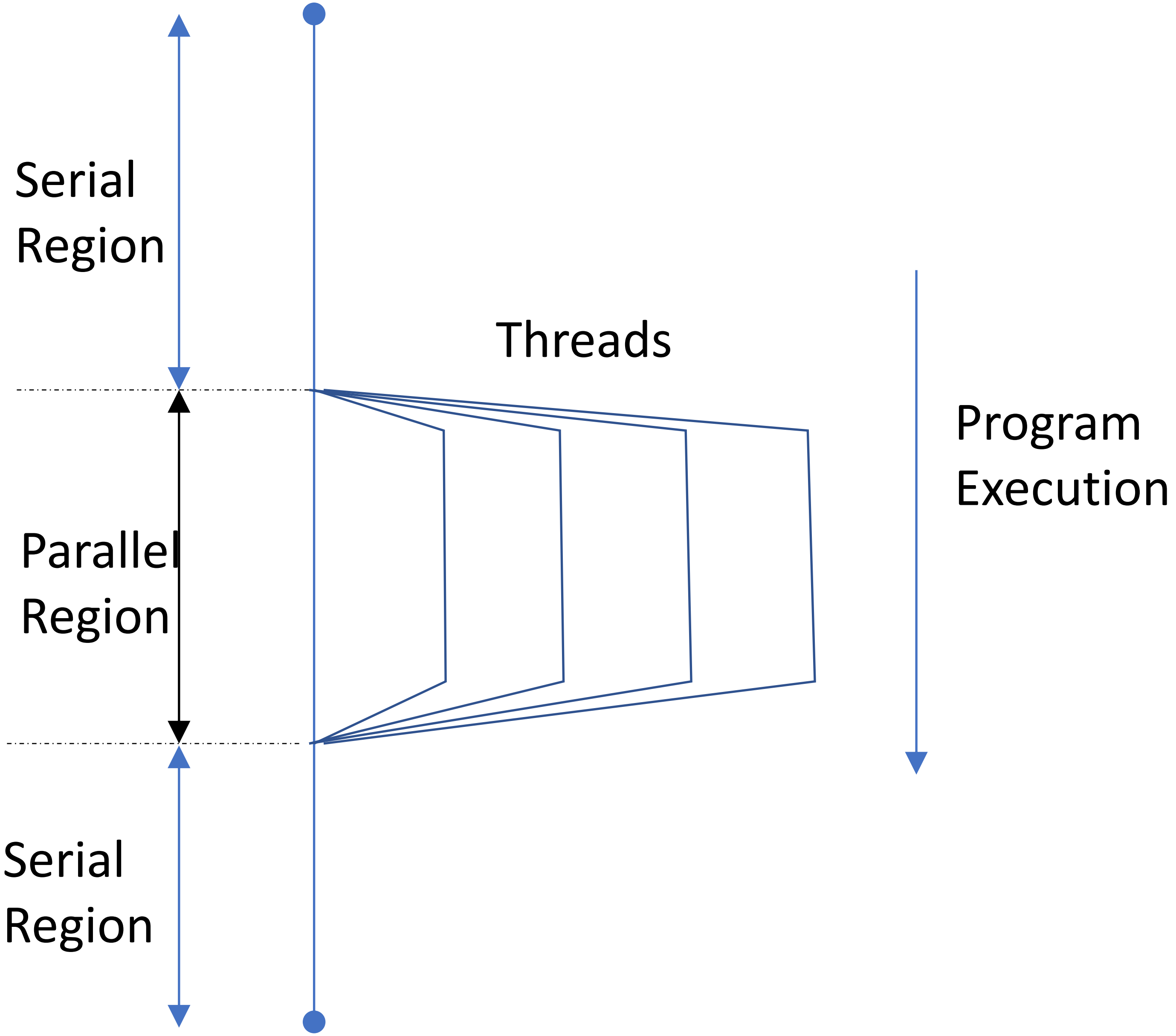
Program Execution

```c
int main(){
    // C code starts here

    int i;
    …..
    #pragma omp parallel {
        // multiple threads working here
          …..
    }
…..
    return 0;
}
```

- gcc –fopenmp myprog.c
- ./a.out

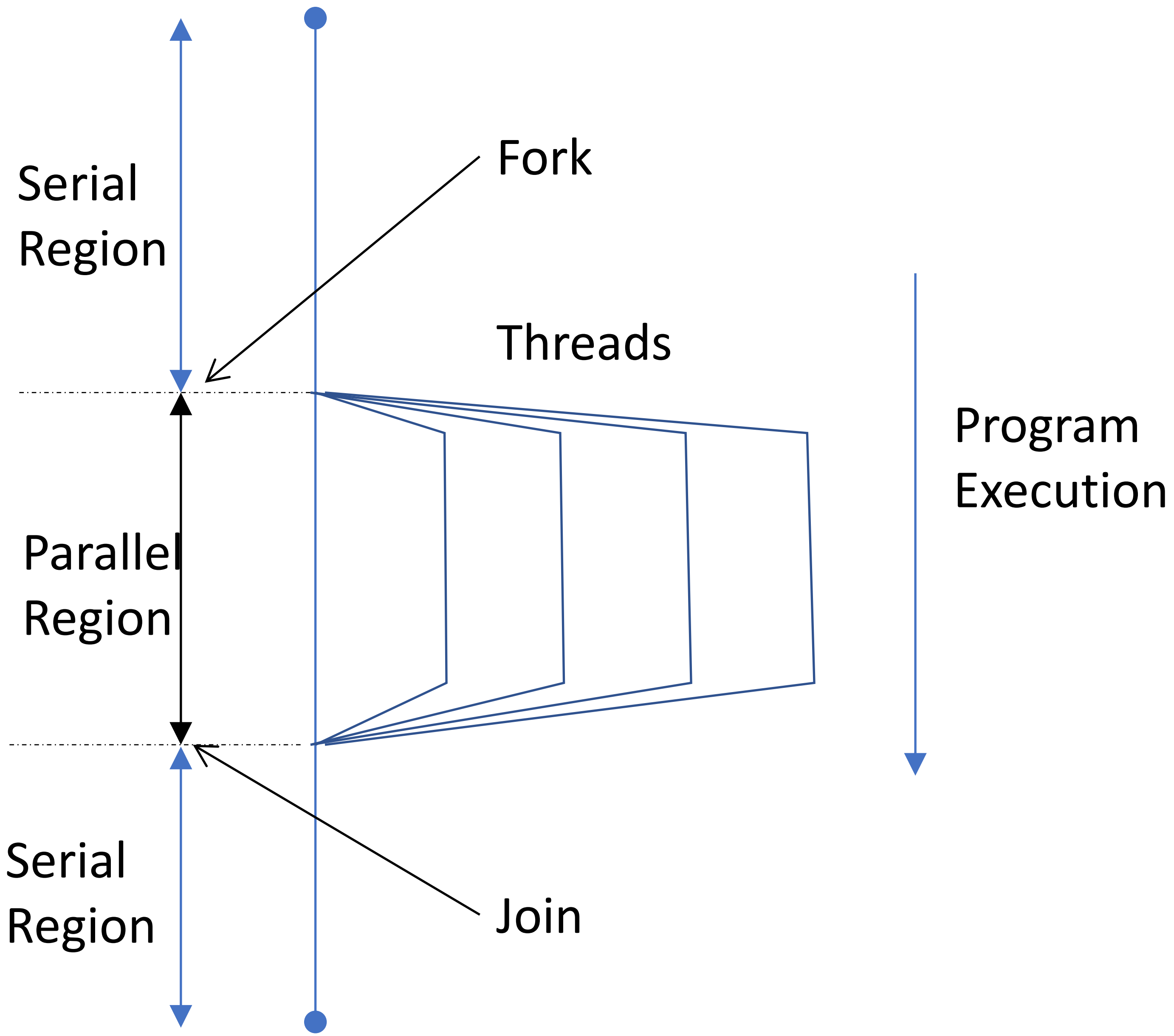
OpenMP: Workflow

```
int main(){
    // C code starts here


    int i;
    …..
    #pragma omp parallel {
        // multiple threads
working here
            …..
    }
…...
    return 0;
}
```



OpenMP: Fork/Join Model

```
int main(){
    // C code starts here

    int i;
    …..
    #pragma omp parallel {
        // multiple threads
working here
            …..
    }
…...
    return 0;
}
```



OpenMP: Construct and Clauses

```
int main(){
    // C code starts here


    int i;
    …..
    #pragma omp parallel {
        // multiple threads
working here
            …..
    }
…...
    return 0;
}
```

- #pragma omp construct [clause(options)]
  - construct: main instruction that applies to the following block of code
  - clause(options): optional, some default values assumed if not specified explicitly

OpenMP: `parallel` construct

```
int main(){
    // C code starts here


    int i;
    …..
    #pragma omp parallel {
        // multiple threads
working here
            …..
    }
…...
    return 0;
}
```

- #pragma omp construct [clause(options)]
    - construct: main instruction that applies to the following block of code
    - clause(options): optional, some default values assumed if not specified explicitly

Number of Threads

- Three ways of setting number of threads:

- Environment variable OMP_NUM_THREADS
- Library Routine: omp_set_num_threads()
- num_threads clause

Linux:

- gcc –fopenmp myprog.c
- export OMP_NUM_THREADS = 8
- ./a.out

Number of Threads

- Three ways of setting number of threads:

- Environment variable OMP_NUM_THREADS
- Library Routine: omp_set_num_threads()
- num_threads clause

```
int main(){
    // C code starts here
    int p = 8;
    omp_set_num_threads(p);
    … // master thread only
    #pragma omp parallel {
        // p threads working here
          …
    }
    … // master thread only
    return 0;
}
```

Number of Threads

- Three ways of setting number of threads:

- Environment variable OMP_NUM_THREADS
- Library Routine: omp_set_num_threads()
- num_threads clause

```
int main(){
    // C code starts here

    int p = 8;
    …
    #pragma omp parallel num_threads (p)
{
        // multiple threads working here
          …
    }
…
    return 0;
}
```

Number of Threads

- Three ways of setting number of threads:

- Environment variable OMP_NUM_THREADS
- Library Routine: omp_set_num_threads()
- num_threads clause

- Hello world example

Number of Threads

- Three ways of setting number of threads:

- Environment variable OMP_NUM_THREADS
- Library Routine: omp_set_num_threads()
- num_threads clause

- Hello world example

- Question: If more than one of the above is present?
  - Experiment for yourself and find out.

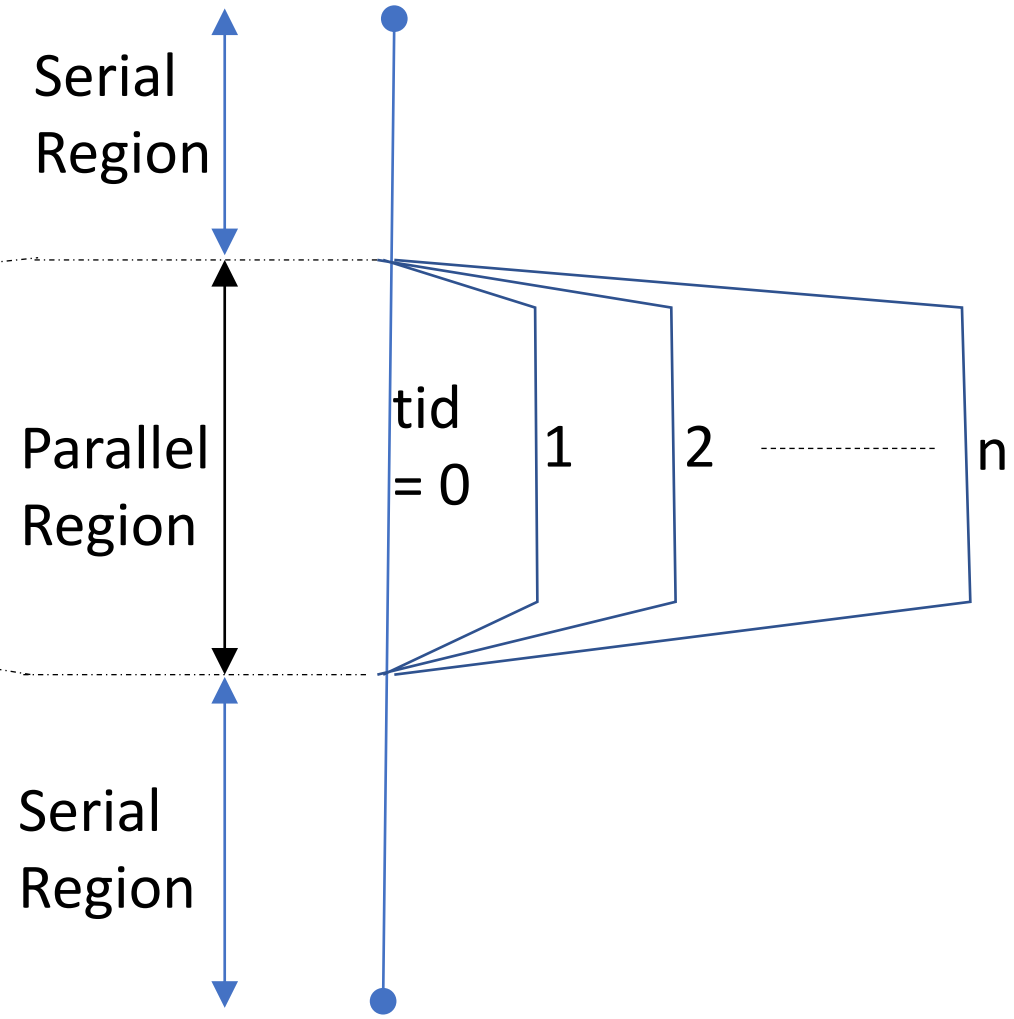
- Question: How many threads can you run?
  - Depends on the “system”. Effectively, no limit.

- Question: How many *should* I run?
  - Always experiment. Thumb rule: number of cores

Distinguishing Between Threads

```
int main(){
  int n = 8;
  omp_set_num_threads(n);
  … // master thread only

  #pragma omp parallel {
    int tid = omp_get_thread_num();
    printf(“Hello world %d”, tid);
  }
  … // master thread only
  return 0;
}
```

- Library Routine: omp_get_num_threads()



OpenMP Basics Summary/Overview

Purpose	construct/clause
Create threads	• `parallel` construct
Distribute work	• `for` construct • `sections` construct
Variable scoping	• `shared`, `private`, `firstprivate`, `lastprivate`, `reduction`, `collapse`, … clauses
Synchronization	• `critical`, `atomic`, `barrier`, `nowait` constructs
Functions/environment variables	• `omp_set_num_threads`, `omp_get_thread_num` • `OMP_NUM_THREADS`, `OMP_SCHEDULE`

OpenMP Parallelism: Other Languages

- Exactly analogous support for Fortran (replace #pragma with !$OMP)
- Some functionality supported in Matlab as well (e.g. parfor)

- Python: the picture is complicated
  - Python operates under ‘GIL’ (Global Interpreter Lock); only one thread can be active at a time
  - any compiled code (C/Fortran/…) called from within Python can be OpenMP-parallel

OpenMP Basics Summary/Overview

Purpose	construct/clause
Create threads	• `parallel` construct
Distribute work	• `for` construct • `sections` construct
Variable scoping	• `shared`, `private`, `firstprivate`, `lastprivate`, `reduction`, `collapse`, … clauses
Synchronization	• `critical`, `atomic`, `barrier`, `nowait` constructs
Functions/environment variables	• `omp_set_num_threads`, `omp_get_thread_num` • `OMP_NUM_THREADS`, `OMP_SCHEDULE`

OpenMP Basics Summary/Overview

Purpose	construct/clause
Create threads	• `parallel` construct
Distribute work	• `for` construct • `sections` construct
Variable scoping	• `shared`, `private`, `firstprivate`, `lastprivate`, `reduction`, `collapse`, … clauses
Synchronization	• `critical`, `atomic`, `barrier`, `nowait` constructs
Functions/environment variables	• `omp_set_num_threads`, `omp_get_thread_num` • `OMP_NUM_THREADS`, `OMP_SCHEDULE`

Sharing Work Between Threads

- Two ways:

- for
- sections

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

Split loop iterations among available threads

Wait until all threads arrive here (implicit barrier)

Sharing Work Between Threads

- Two ways:

- for
- sections

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

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

Sharing Work Between Threads

- Two ways:

- for
- sections

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

What happens if you miss the `for`?

Sharing Work Between Threads

- Two ways:

- for
- sections

```
…
#pragma omp parallel {
  #pragma omp sections {
    #pragma section {
      // task 1      }
    #pragma section {
      // task 2      }
    ……
    #pragma section {
      // task n      }
  }
}
…
```

Assign one section to one thread

Wait until all threads arrive here (implicit barrier)

Sharing Work Between Threads

- Two ways:
- for
- sections

```
…
#pragma omp parallel {
  #pragma omp sections {
    #pragma section {
      // task 1      }
    #pragma section {
      // task 2      }
    ……
    #pragma section {
      // task n      }
  }
}
…
```

- Question: Which thread gets which section?
- Question: What if num_threads > num_sections?
- Question: What if num_threads < num_sections?

Sharing Work Between Threads

- Two ways:

- for
- sections

```
#pragma omp parallel {
  …
  #pragma omp sections {
    #pragma section {
      // task 1      }
    #pragma section {
      // task 2      }
    ……
    #pragma section {
      // task n      }
  }
  …
}
```

- Question: Which thread gets which section?
  - First come first served
- Question: What if num_threads > num_sections?
  - Latecomers remain idle
- Question: What if num_threads < num_sections?
  - Round robin assignment of sections as threads become available

schedule Clause

- Two ways:

- for
- sections

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

Split loop iterations among available threads

- Question: How can we control the ‘split’?
- Answer: schedule clause

schedule Clause

```
…
#pragma omp parallel {
  #pragma omp for schedule(kind, chunksize)
  for(i=0; i<N; i++)
  {
      a[i] = b[i] + c[i];
      d[i] = do_work(i, a[i]);
  }
}
…
```

- kind can be:
  - static
  - dynamic
  - guided
  - runtime
- chunksize is optional

`schedule` Clause

```
…
#pragma omp parallel {
  #pragma omp for schedule(kind, chunksize)
  for(i=0; i<N; i++)
  {
      a[i] = b[i] + c[i];
      d[i] = do_work(i, a[i]);
  }
}
…
```

- `kind` can be:
  - static
  - dynamic
  - guided
  - runtime
- `chunksize` is optional
- Round-robin assignment of chunks
- Default `chunksize` = N/num_threads (roughly)

`schedule` Clause

```
…
#pragma omp parallel {
  #pragma omp for schedule(kind, chunksize)
  for(i=0; i<N; i++)
  {
      a[i] = b[i] + c[i];
      d[i] = do_work(i, a[i]);
  }
}
…
```

- `kind` can be:
  - static
  - dynamic
  - guided
  - runtime
- `chunksize` is optional
- Chunks assigned as threads become available
- Default `chunksize` = 1

`schedule` Clause

```
…
#pragma omp parallel {
  #pragma omp for schedule(kind, chunksize)
  for(i=0; i<N; i++)
  {
      a[i] = b[i] + c[i];
      d[i] = do_work(i, a[i]);
  }
}
…
```

- `kind` can be:
  - static
  - dynamic
  - guided
  - runtime
- `chunksize` is optional
- dynamic with dynamic size of chunks (proportional to remaining work).
- `chunksize` sets the minimum size except for the last assignment

`schedule` Clause

```
…
#pragma omp parallel {
  #pragma omp for schedule(kind, chunksize)
  for(i=0; i<N; i++)
  {
      a[i] = b[i] + c[i];
      d[i] = do_work(i, a[i]);
  }
}
…
```

- kind can be:
  - static
  - dynamic
  - guided
  - runtime
- chunksize is optional
- kind and chunksize set at runtime. Details vary by implementation (gnu vs intel vs …)

`schedule` Clause

```
…
#pragma omp parallel {
  #pragma omp for schedule(kind, chunksize)
  for(i=0; i<N; i++)
  {
      a[i] = b[i] + c[i];
      d[i] = do_work(i, a[i]);
  }
}
…
```

`schedule` example

- `kind` can be:
  - static
  - dynamic
  - guided
- `chunksize` is optional
- When to use what?
  - Simple, predictable, roughly equal work for each iteration -> static
  - Unpredictable, highly variable work in different iterations -> dynamic, guided

OpenMP Basics Summary/Overview

Purpose	construct/clause
Create threads	• parallel construct
Distribute work	• for construct • sections construct
Variable scoping	• shared, private, firstprivate, lastprivate, reduction, collapse, … clauses
Synchronization	• critical, atomic, barrier, nowait constructs
Functions/environment variables	• omp_set_num_threads, omp_get_thread_num • OMP_NUM_THREADS, OMP_SCHEDULE

Variable Sharing

```
int main(){
  int p = 8, i;
  omp_set_num_threads(p);
  … // master thread only

  #pragma omp parallel {
    int tid = omp_get_thread_num();
    printf("Hello world %d", tid);
  }
  … // master thread only
  return 0;
}
```

vs

```
int main(){
  int p = 8, i, tid;
  omp_set_num_threads(p);
  … // master thread only

  #pragma omp parallel {
    tid = omp_get_thread_num();
    printf("Hello world %d", tid);
  }
  … // master thread only
  return 0;
}
```

Variable Sharing

```c
int main(){
  int p = 8, i;
  omp_set_num_threads(p);
  … // master thread only

  #pragma omp parallel {
    int tid = omp_get_thread_num();
    printf("Hello world %d", tid);
  }
  … // master thread only
  return 0;
}
```

✓

tid is private to each thread (stack)

vs

```c
int main(){
  int p = 8, i, tid;
  omp_set_num_threads(p);
  … // master thread only

  #pragma omp parallel {
    tid = omp_get_thread_num();
    printf("Hello world %d", tid);
  }
  … // master thread only
  return 0;
}
```

✗

tid is shared among all threads (heap)

Variable Sharing

```
int main(){
  int p = 8, i, N = 64;
  omp_set_num_threads(p);

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

vs

```
int main(){
  int p = 8, i, N = 64, k;
  omp_set_num_threads(p);

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

Variable Sharing

```
int main(){
  int p = 8, i, N = 64;
  omp_set_num_threads(p);

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

✓

vs

```
int main(){
  int p = 8, i, N = 64, k;
  omp_set_num_threads(p);

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

✗

i is private to each thread (stack)

k is shared among all threads (heap)

Variable Sharing Rules

- Variables that are shared by default:
  - Allocated on heap
  - Declared outside the scope of the parallel construct
  - `static` or `constant` variables

- Variables that are private by default:
  - Declared within the scope of a `parallel` construct
  - Index variable in a `parallel for` construct

- Not paying attention to variable scoping is the biggest cause for “data race” or ”race conditions”
- Debugging these issues is, very often, the most time-consuming part of OpenMP programming!

Variable Sharing: Overriding the Defaults

```
int main(){
  int p = 8, tid;
  omp_set_num_threads(p);
  … // master thread only

  #pragma omp parallel {
    tid = omp_get_thread_num();
    printf("Hello world %d", tid);
  }
  … // master thread only
  return 0;
}
```

✘

tid is shared among all threads (heap)

Variable Sharing: Overriding the Defaults

```
int main(){
  int p = 8, tid;
  omp_set_num_threads(p);
  … // master thread only

  #pragma omp parallel private(tid){
    tid = omp_get_thread_num();
    printf("Hello world %d", tid);
  }
  … // master thread only
  return 0;
}
```

tid is private to each thread (stack)

vs

```
int main(){
  int p = 8, tid;
  omp_set_num_threads(p);
  … // master thread only

  #pragma omp parallel {
    tid = omp_get_thread_num();
    printf("Hello world %d", tid);
  }
  … // master thread only
  return 0;
}
```

tid is shared among all threads (heap)

Variable Sharing: Overriding the Defaults

```
int main(){
  int p = 8, tid;
  omp_set_num_threads(p);
  … // master thread only

  #pragma omp parallel private(tid){
    tid = omp_get_thread_num();
    printf(“Hello world %d”, tid);
  }
  … // master thread only
  return 0;
}
```

tid is private to each thread (stack)

vs

- p-1 additional instances of tid are created (copies on stack of each thread)
- May or may not be initialized (implementation dependent, OpenMP standard does not specify initialization)

Variable Sharing: Overriding the Defaults

```
int main(){
  int p = 8, tid;
  omp_set_num_threads(p);
  … // master thread only

  #pragma omp parallel private(tid){
    tid = omp_get_thread_num();
    printf("Hello world %d", tid);
  }
  printf("Outside parallel %d", tid);
  return 0;
}
```

tid is private to each threads (stack)

vs

- p-1 additional instances of `tid` are created (copies on stack)
- May or may not be initialized (implementation dependent, OpenMP standard does not specify initialization)

- What happens at the end of the `parallel` construct?
- Only one instance (master) remains

Variable Sharing: Overriding the Defaults

```
int main(){
  int p = 8, tid;
  omp_set_num_threads(p);
  … // master thread only

  #pragma omp parallel private(tid){
    tid = omp_get_thread_num();
    printf(“Hello world %d”, tid);
  }
  printf(“Outside parallel %d”, tid);
  return 0;
}
```

tid is private to each threads (stack)

vs

- p-1 additional instances of tid are created (copies on stack)
- May or may not be initialized (implementation dependent, OpenMP standard does not specify initialization)

- What happens at the end of the parallel construct?
- Only one instance (master) remains

- Ways to control initialization and termination of parallel construct

`parallel` construct clauses, options (partial)

```
…
#pragma omp parallel
  private (var1, var2, var3, var4, …)
  firstprivate (var1, var2, …)
  reduction(operator:list)
  {
    // parallel block
  }
…
```

- Control for initialization:
  - firstprivate: initialize value of var for all threads to that before `parallel`
- Control at termination:
  - reduction: apply `operator` (e.g. sum, min, max,…) on each variable in the `list`. Value of variable after the `parallel` construct is determined by this operation

`parallel` construct clauses, options (complete)

```
…
#pragma omp parallel
  num_threads(p)
  private (var1, var2, var3, var4, …)
  firstprivate (var1, var2, …)
  reduction(operator:list)
  if (boolean)
  default(shared OR none)
  shared (var5, var6)
  copyin (var1, var3, …)
  proc_bind(…)
  allocate(…)
{ // parallel block }
…
```

- `if`: whether to execute in parallel
- `default`: for variables declared within scope
- `shared`: force variables to be shared across threads
- `copyin`: initialize variables to values existing prior to parallel (similar to `firstprivate` here)

- `proc_bind`, `allocate`: look up yourself

`sections` construct clauses, options (complete)

```
#pragma omp parallel
#pragma omp sections
  private (var1, var2, var3, var4, …)
  firstprivate (var1, var2, …)
  lastprivate (var3, var4, …)
  reduction(operator:list)
  nowait
  allocate(…)
{ // parallel sections block }
…
```

- `private`: variable specific to each thread
- `firstprivate`: assign value of var1, var2 from before parallel region at the start of `sections`
- `lastprivate`: assign value from the last `section` to var3, var4 at the end of `sections`
- `reduction`: apply operator (e.g. sum, min, max,…) on each variable in the `list`. Value of variable after the `sections` construct is determined by this operation
- `nowait`: coming up
- `allocate`: look up yourself

`for` construct clauses, options (complete)

```
#pragma omp parallel
#pragma omp for
  private (var1, var2, var3, var4, …)
  firstprivate (var1, var2, …)
  lastprivate (var3, var4, …)
  reduction(operator:list)
  schedule (kind, chunk_size)
  linear (var5:2, list[:linear-step])
  collapse(n)
  ordered [(n)]
  order (concurrent)
  nowait
  allocate(…)
{ // parallel block }
…
```

- `lastprivate`: assign value from the last iteration to variable at the end of `for`
- `linear`: increment var5 by 2 at every iteration; retain value at end
- `collapse`: nested for loops, n loops are combined and then distributed
- `nowait`: coming up
- `ordered, order, allocate`: look up yourself

`parallel/for/sections` construct examples

- `private, firstprivate, example`
- `reduction example`

OpenMP Basics Summary/Overview

Purpose	construct/clause
Create threads	• `parallel` construct
Distribute work	• `for` construct • `sections` construct
Variable scoping	• `shared`, `private`, `firstprivate`, `lastprivate`, `reduction`, `collapse`, … clauses
Synchronization	• `critical`, `atomic`, `barrier`, `nowait`, `single`, `master` constructs
Functions/environment variables	• `omp_set_num_threads`, `omp_get_thread_num` • `OMP_NUM_THREADS`, `OMP_SCHEDULE`

Synchronization

- No synchronization between threads unless explicitly required by the code (MIMD model)
- (Implicit) Barrier/`nowait`
- `single`/`master`
- `critical`/`atomic`
- `ordered`
- locks

```
#pragma omp parallel {
  …… // parallel work
  …… // parallel work
  …… // parallel work
  …… // synchronization
  …… // parallel work
  …… // parallel work
 …… // synchronization
}
```

Synchronization: barrier

```
#pragma omp parallel {
   ......
   #pragma omp barrier
   ......
   ......
   #pragma omp barrier
   ......
   ......
   ......
}
...
```

Fork

Threads wait until all threads have reached here

Threads wait until all threads have reached here

Join

Synchronization (implicit barrier)

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

Fork

Implicit Barrier

Implicit Barrier

Join

(De-) synchronization: **nowait**

```
#pragma omp parallel {
  ......
  #pragma omp for
  for(i=0; i<N; i++) {
    a[i] = b[i] + c[i];
  }
  ......
  #pragma omp for
  for(i=0; i<2*N; i++) {
    d[i] = e[i] + f[i];
  }
  ......
}
```

Fork

Implicit Barrier

Implicit Barrier

Join (Barrier)

```
#pragma omp parallel {
  ......
  #pragma omp for nowait
  for(i=0; i<N; i++) {
    a[i] = b[i] + c[i];
  }
  ......
  #pragma omp for
  for(i=0; i<2*N; i++) {
    d[i] = e[i] + f[i];
  }
  ......
}
```

(De-) synchronization: **nowait**

```
#pragma omp parallel {        <- Fork
  ......
  #pragma omp for
  for(i=0; i<N; i++) {
    a[i] = b[i] + c[i];
  }                           <- Implicit Barrier
  ......
  #pragma omp for
  for(i=0; i<2*N; i++) {
    d[i] = e[i] + f[i];
  }                           <- Implicit Barrier
  ......
}                             <- Join (Barrier)
```

```
#pragma omp parallel {
  ......
  #pragma omp for nowait
  for(i=0; i<N; i++) {
    a[i] = b[i] + c[i];
  }
  ......
  #pragma omp for nowait
  for(i=0; i<2*N; i++) {
    d[i] = e[i] + f[i];
  }
  ......
}
```

(De-) synchronization: `single/master`

```
#pragma omp parallel {
  ……
  #pragma omp for
  for(i=0; i<N; i++) {
    a[i] = b[i] + c[i];
  }
  #pragma omp single
    printf("Finished loop\n");
  ……
  ……
}
```

First thread to reach single executes; others proceed to the next line

```
#pragma omp parallel {
  ……
  #pragma omp for
  for(i=0; i<N; i++) {
    a[i] = b[i] + c[i];
  }
  #pragma omp master
    printf("Finished loop\n");
  ……
  ……
}
```

Only master executes; others proceed to the next line

Synchronization: `critical/atomic`

```
#pragma omp parallel
private(lsum) shared (sum) {
  #pragma omp single
    sum = 0.0;
  lsum = 0.0;
  #pragma omp for
  for(i=0; i<N; i++)
    lsum += a[i];
  #pragma omp critical
     sum += lsum;
  ……
}
```

- Only one thread executes at a time.
- No particular order is enforced

Synchronization: `critical/atomic`

```
#pragma omp parallel
private(lsum) shared (sum) {
  #pragma omp single
    sum = 0.0;
  lsum = 0.0;
  #pragma omp for
  for(i=0; i<N; i++)
    lsum += a[i];
  #pragma omp critical
     sum += lsum;
  ......
}
```

- Only one thread executes at a time.
- No particular order is enforced

```
#pragma omp parallel
private(lsum) shared (sum) {
  #pragma omp single
    sum = 0.0;
  lsum = 0.0;
  #pragma omp for
  for(i=0; i<N; i++)
    lsum += a[i];
  #pragma omp atomic
     sum += lsum;
  ......
}
```

Synchronization: `critical/atomic` differences

```
#pragma omp parallel {
  ……
  #pragma omp critical
    sum += lsum;
  ……
  ……
 #pragma omp critical {
    …… // Any operation
    …… // Any operation
  }
  ……
}
```

Critical:

One or more operations. No restrictions

```
#pragma omp parallel {
  ……
  #pragma omp atomic
    sum += lsum;   ✓
  ……
  ……
 #pragma omp atomic {   ✗
    ……
    ……
  }
  ……
}
```

Atomic:

Only one operation; Only specific operations (assignment)

Synchronization: `critical/atomic`

```
#pragma omp parallel {
  ……
  #pragma omp critical
    sum += lsum;
  ……
  ……
 #pragma omp critical {
    if(current_val > max_val)
      max_val = current_val;
  }
  ……
}
```

- If more than one critical section occurs, they are mutually exclusive
  - Only one thread at a time across all critical sections
- Synchronization is being forced across critical sections; can affect performance

Synchronization: `critical/atomic`

```
#pragma omp parallel {
  ……
  #pragma omp critical (summ)
    sum += lsum;
  ……
  ……
 #pragma omp critical (maxx) {
    if(current_val > max_val)
      max_val = current_val;
  }
  ……
}
```

- If more than one critical section occurs, they are mutually exclusive
  - Only one thread at a time across all unnamed critical sections
- Synchronization is being forced across critical sections; can affect performance
- Remedy: named critical sections
  - Named critical sections are not mutually exclusive

Synchronization: `ordered`

```
#pragma omp parallel {
   ......
   #pragma omp for ordered
   for(i=0; i<N; i++) {
      ......
      ......
      #pragma omp ordered
         printf("Var value on %d = 
%f\n", tid, val);
   }
   ......
}
```

- #pragma omp construct [clause(options)]
- ordered clause: force a block in a for loop to be executed as if it were serial
- ordered construct: indicates that a ordered clause appears somewhere in this construct

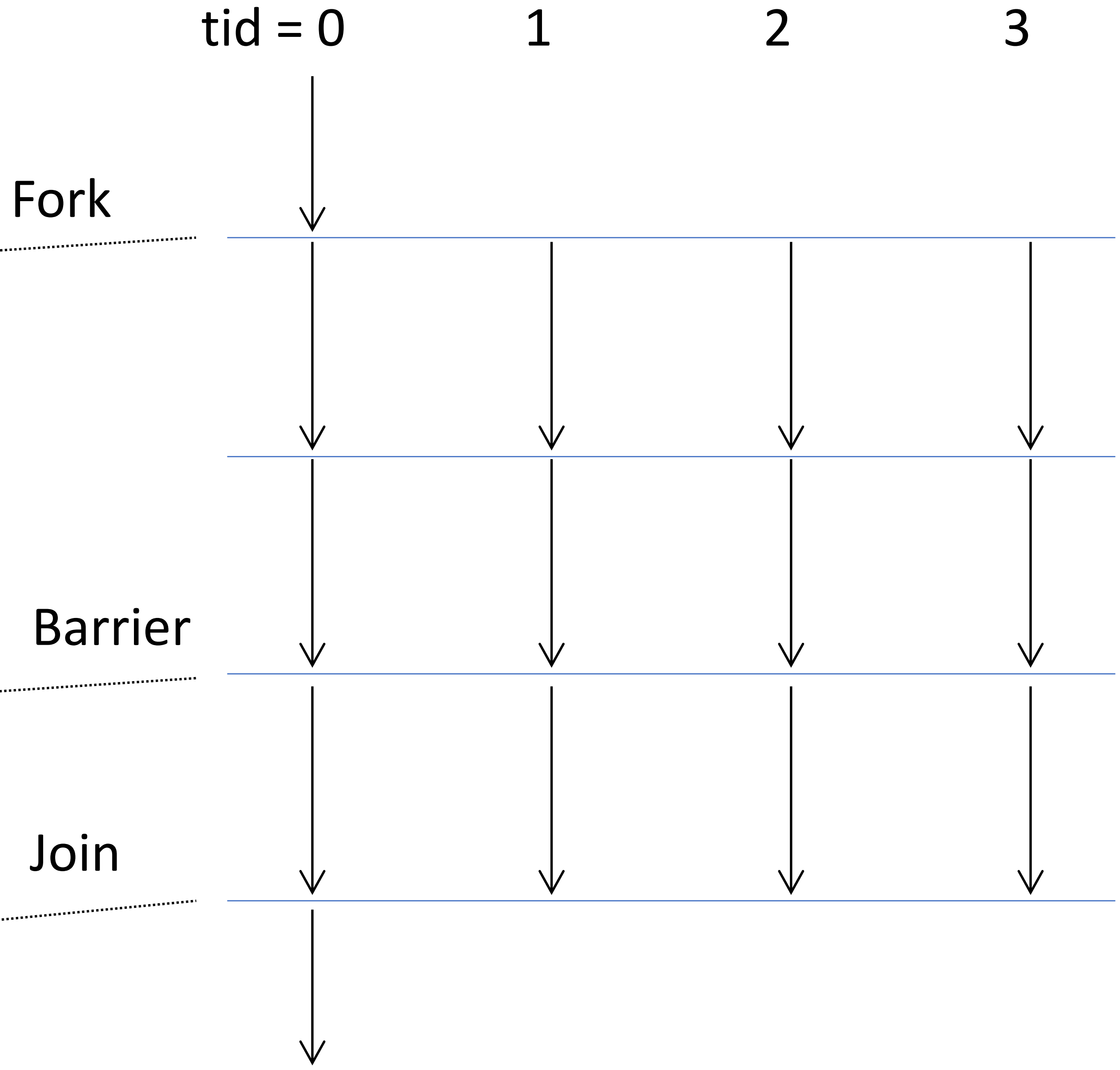
OpenMP Basics Summary/Overview

Purpose	construct/clause
Create threads	• `parallel` construct
Distribute work	• `for` construct • `sections` construct
Variable scoping	• `shared`, `private`, `firstprivate`, `lastprivate`, `reduction`, `collapse`, … clauses
Synchronization	• `critical`, `atomic`, `barrier`, `nowait` constructs
Functions/environment variables	• `omp_set_num_threads`, `omp_get_thread_num` • `OMP_NUM_THREADS`, `OMP_SCHEDULE`

More Examples

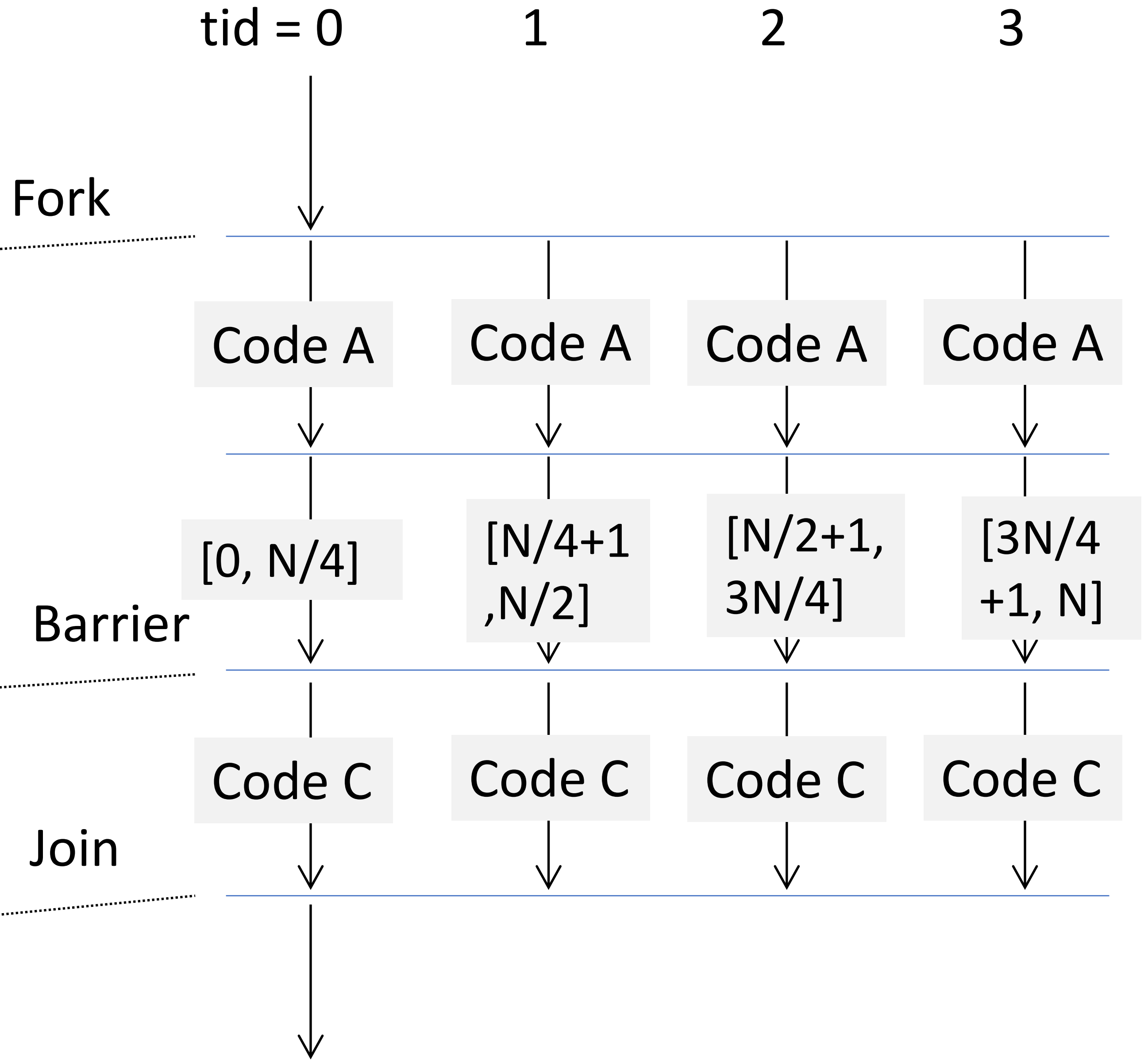
More Examples

```
……
#pragma omp parallel {
  …… //{code A}

  #pragma omp for
  for(i=0; i<N; i++) {
    …… //{code B}
   }

  …… // {code C}
}
……
```



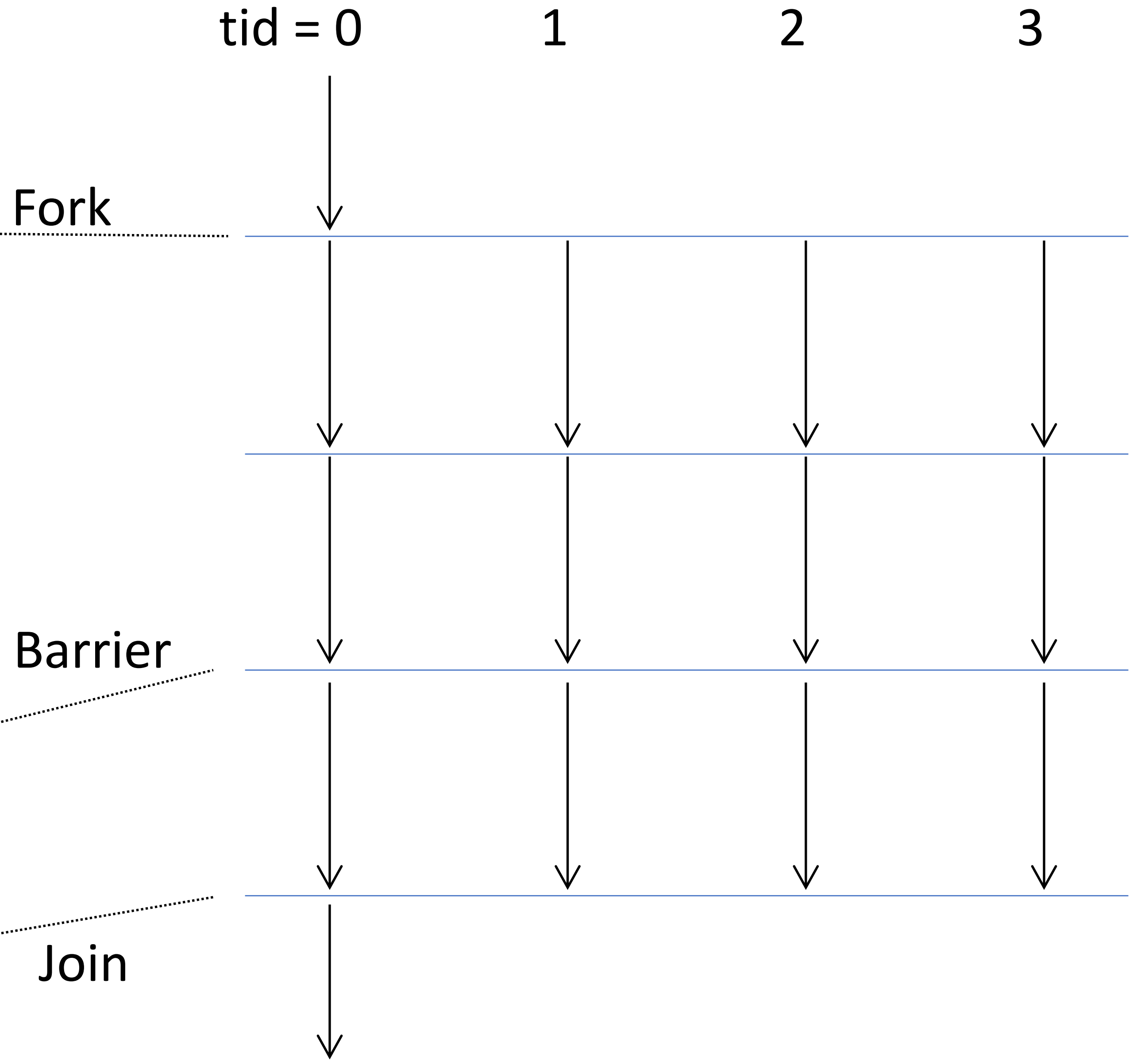
More Examples

```
……
#pragma omp parallel {
  …… //{code A}

  #pragma omp for
  for(i=0; i<N; i++) {
    …… //{code B}
   }

  …… // {code C}
}
……
```



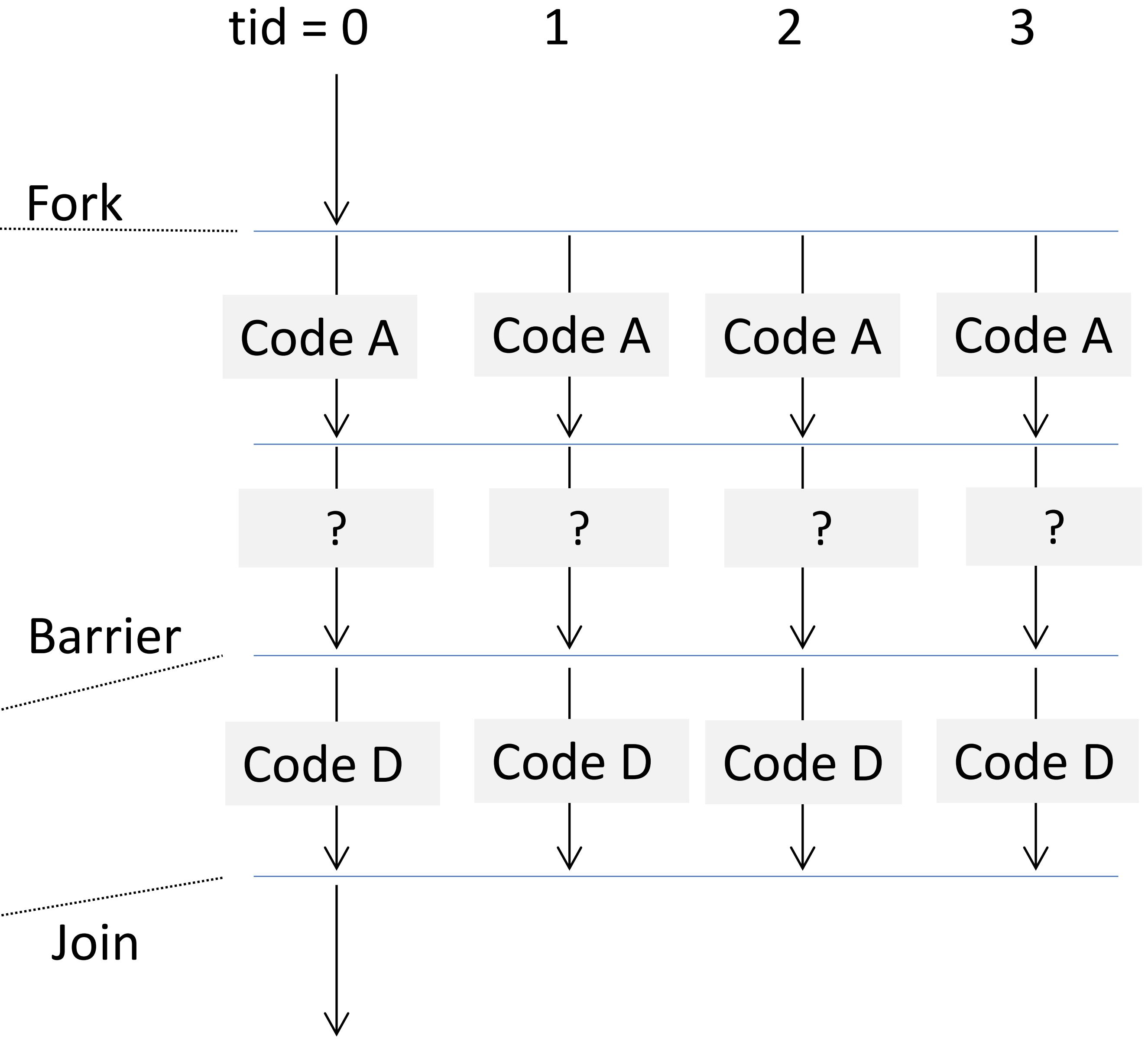
More Examples

```
……
#pragma omp parallel {
  …… //{code A}
  #pragma omp sections {
    #pragma section {
      …… //{code B}
    }
    #pragma section {
      …… //{code C}
    }
  }
  …… // {code D}
}
……
```



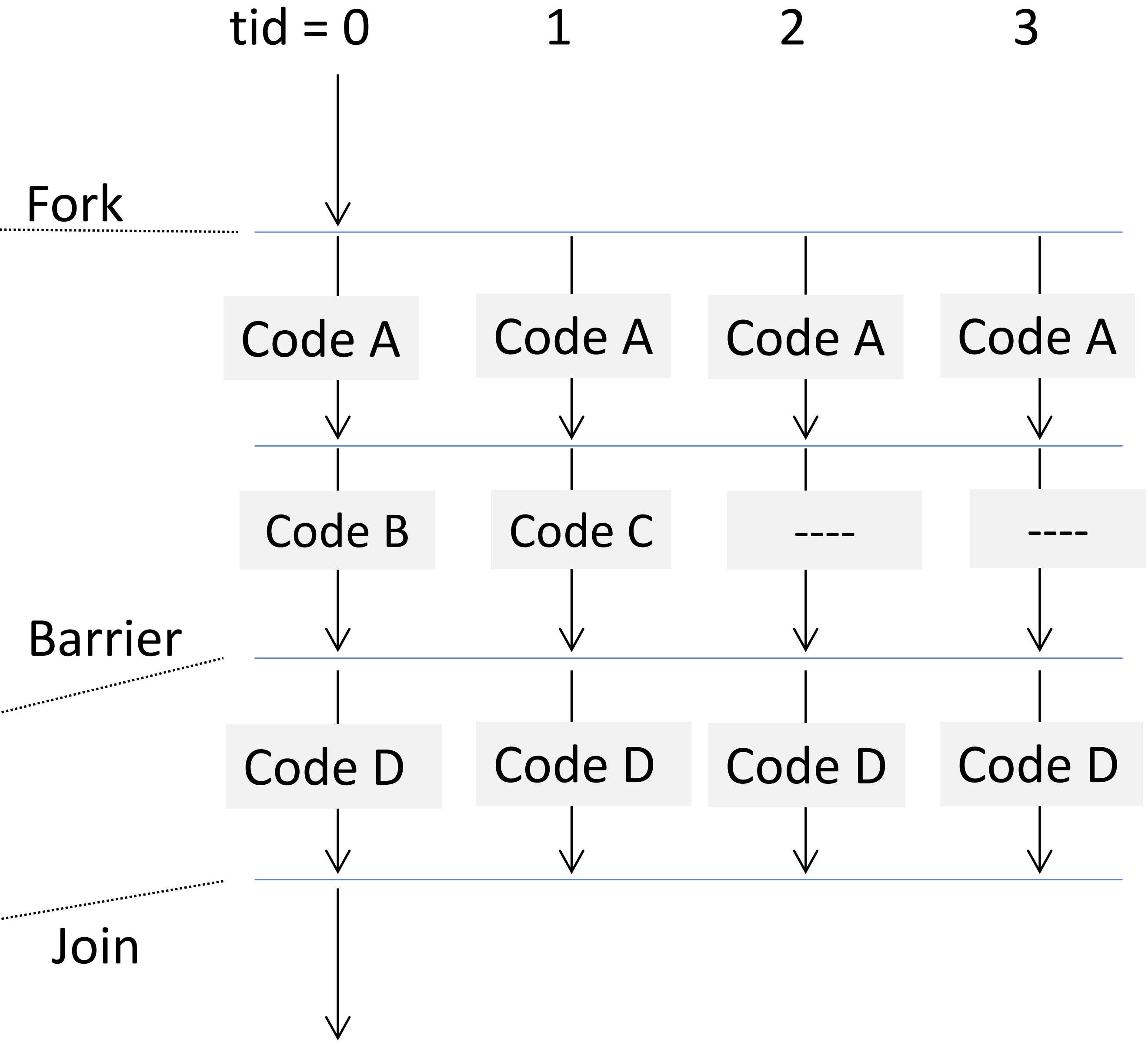
More Examples

```
……
#pragma omp parallel {
  …… //{code A}
  #pragma omp sections {
    #pragma section {
      …… //{code B}
    }
    #pragma section {
      …… //{code C}
    }
  }
  …… // {code D}
}
……
```



More Examples

```
……
#pragma omp parallel {
  …… //{code A}
  #pragma omp sections {
    #pragma section {
      …… //{code B}
    }
    #pragma section {
      …… //{code C}
    }
  }
  …… // {code D}
}
……
```



Example: Find the bug

```
/* Declare and initialize a, b, c, chunk, … */


#pragma omp parallel for shared(a,b,c,chunk)
private(i,tid) schedule(static,chunk) {
    tid = omp_get_thread_num();
    for (i=0; i < N; i++) {
      c[i] = a[i] + b[i];
      printf("tid= %d i= %d c[i]= %f\n", tid,
i, c[i]);
    }
  }  /* end of parallel for */
```

- Compile-time or runtime bug?

Example: Find the bug

```
/* Declare and initialize a, b, c, chunk, … */


#pragma omp parallel for shared(a,b,c,chunk)
private(i,tid) schedule(static,chunk) {
    tid = omp_get_thread_num();
    for (i=0; i < N; i++) {
      c[i] = a[i] + b[i];
      printf("tid= %d i= %d c[i]= %f\n", tid,
i, c[i]);
    }
  }  /* end of parallel for */
```

- Compilation error
- First line in `parallel for` must be the `for` loop

Example: Find the bug

```
#pragma omp parallel {
  ……
  #pragma omp for nowait
    for(i=0; i<N; i++) {
      a[i] = b[i] + c[i];
    }
  …… // No barriers
  #pragma omp for schedule (dynamic)
    for(i=0; i<2*N; i++) {
      c[i] = e[i] + f[i];
    }
  ……
}
```

- Compile-time or runtime?

Example: Find the bug

```
#pragma omp parallel {
  ……
  #pragma omp for nowait
    for(i=0; i<N; i++) {
      a[i] = b[i] + c[i];
    }
  …… // No barriers
  #pragma omp for schedule (dynamic)
    for(i=0; i<N; i++) {
      c[i] = e[i] + f[i];
    }
  ……
}
```

- Runtime error (will not segfault)
- Answer may be wrong (race)