OpenMP Examples (1A)

-
-

Young Won Lim
10/22/20

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice and Octave.

Installation

STEP 1: Check the GCC version of the compiler
gcc –version

STEP 2: Configuring OpenMP
echo | cpp -fopenmp -dM |grep -i open
sudo apt install libomp-dev

STEP 3: Setting the number of threads
export OMP_NUM_THREADS=8

https://www.geeksforgeeks.org/openmp-introduction-with-installation-guide/

Parallel regions

```
// OpenMP header
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    int nthreads, tid;

    // Begin of parallel region
    #pragma omp parallel private(nthreads, tid)
    {
        // Getting thread number
        tid = omp_get_thread_num();
        printf("Welcome to GFG from thread = %d\n", tid);
        if (tid == 0) {
            // Only master thread does this
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }
}
```

https://www.geeksforgeeks.org/openmp-introduction-with-installation-guide/

Private variables

```
#include <omp.h>

main(int argc, char *argv[]) {

    int nthreads, tid;

    /* Fork a team of threads with each thread having a private tid variable */
    #pragma omp parallel private(tid)
    {

        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

    }  /* All threads join master thread and terminate */

}
```

https://computing.llnl.gov/tutorials/openMP/#Compiling

OpenMP Code Structure

```
#include <omp.h>

main ()  {
    int var1, var2, var3;
    Serial code
    …

    Beginning of parallel region. Fork a team of threads.
    Specify variable scoping

        #pragma omp parallel private(var1, var2) shared(var3)
        {

            Parallel region executed by all threads
            Other OpenMP directives
            Run-time Library calls
            All threads join master thread and disband

        }

    Resume serial code
    ...
}
```

https://computing.llnl.gov/tutorials/openMP/

OpenMP Directives

#pragma omp parallel [**clause** ...] newline
                if (scalar_expression)
                private (list)
                shared (list)
                default (shared | none)
                firstprivate (list)
                reduction (operator: list)
                copyin (list)
                num_threads (integer-expression)

structured_block

https://computing.llnl.gov/tutorials/openMP/

OpenMP Directives

Directive name

A valid OpenMP directive.

Must appear after the pragma and before any clauses.

[clause, …]

Optional.

Clauses can be in any order, and repeated as necessary

unless otherwise restricted.

Newline

Required.

Precedes the structured block

which is enclosed by this directive.

https://computing.llnl.gov/tutorials/openMP/

Installation

Compile:

gcc -fopenmp test.c

Execute:

./a.out

https://www.geeksforgeeks.org/openmp-introduction-with-installation-guide/

Number of cores

grep processor /proc/cpuinfo | wc -l

sysconf(_SC_NPROCESSORS_CONF)
sysconf(_SC_NPROCESSORS_ONLN)

grep -c ^processor /proc/cpuinfo

grep -c ^cpu /proc/stat # subtract 1 from the result

https://stackoverflow.com/questions/150355/programmatically-find-the-number-of-cores-on-a-machine

OpenMP API Overview

The OpenMP 3.1 API is comprised of three distinct components:

- **Compiler Directives**
- **Runtime Library Routines**
- **Environment Variables**

https://computing.llnl.gov/tutorials/openMP/#API

Compiler Directives

- Spawning a <u>parallel</u> <u>region</u>
- Dividing <u>blocks</u> of code among threads
- Distributing <u>loop</u> <u>iterations</u> between threads
- <u>Serializing</u> sections of code
- <u>Synchronization</u> of work among threads

https://computing.llnl.gov/tutorials/openMP/#API

Runtime Library Routines

- Setting and querying the <u>number</u> of <u>threads</u>
- Querying a thread's unique identifier (<u>thread ID</u>),
  a thread's <u>ancestor's</u> <u>identifier</u>, the thread <u>team</u> <u>size</u>
- Setting and querying the <u>dynamic</u> <u>threads</u> feature
- Querying if in a <u>parallel</u> <u>region</u>, and at what <u>level</u>
- Setting and querying <u>nested</u> <u>parallelism</u>
- Setting, initializing and terminating <u>locks</u> and <u>nested</u> <u>locks</u>
- Querying <u>wall</u> <u>clock</u> <u>time</u> and <u>resolution</u>

https://computing.llnl.gov/tutorials/openMP/#API

Environment Variables

- Setting the <u>number</u> of <u>threads</u>
- Specifying how <u>loop</u> <u>iterations</u> are divided
- Binding <u>threads</u> to <u>processors</u>
- Enabling/disabling <u>nested</u> <u>parallelism</u>;
  setting the <u>maximum</u> <u>levels</u> of nested parallelism
- Enabling/disabling <u>dynamic</u> <u>threads</u>
- Setting <u>thread</u> <u>stack</u> <u>size</u>
- Setting <u>thread</u> <u>wait</u> <u>policy</u>

https://computing.llnl.gov/tutorials/openMP/#API

Examples

Compiler Directive Examples

#pragma omp **parallel**
#pragma omp **parallel private**(partial_Sum) **shared**(total_Sum)
#pragma omp **parallel private**(thread_id)
#pragma omp **barrier**
#pragma omp **for**
#pragma omp **critical**

Runtime Library Routine Examples

omp_get_thread_num();
omp_get_max_threads();

https://stackoverflow.com/questions/150355/programmatically-find-the-number-of-cores-on-a-machine

Hello

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv) {

    printf("Hello from process: %d\n", omp_get_thread_num());

    return 0;
}
```

```
// only one thread giving us a Hello statement
// must use the #pragma omp parallel { … } directive
// for multiple threads
```

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Hello

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int thread_id;

    #pragma omp parallel
    {
        printf("Hello from process: %d\n", omp_get_thread_num());
    }
    return 0;
}
```

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Private clauses

The PRIVATE clause declares variables in its list
to be private to each thread.

- A new object of the same type is declared once
  for each thread in the team
- All references to the original object are replaced with
  references to the new object
- Should be assumed to be uninitialized for each thread

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Shared clauses

The SHARED clause declares variables in its list t
o be shared among all threads in the team.

A shared variable exists in only one memory location and
all threads can read or write to that address

It is the programmer's responsibility to ensure that
multiple threads properly access SHARED variables
(such as via CRITICAL sections)

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Shared clauses

Variables that are created and assigned
inside of a parallel section of code will be
inherently be **private**

variables created outside of parallel sections
will be inherently **public**.

Hello

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int thread_id;

    #pragma omp parallel private(thread_id)
    {
        thread_id = omp_get_thread_num();
        printf("Hello from process: %d\n", thread_id );
    }

    return 0;
}
```

// create a separate instance of thread_id for each task.

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Barrier and critical directives

#pragma omp **barrier**

The barrier directive <u>stops</u> all processes
for proceeding to the next line of code
<u>until</u> <u>all processes</u> have reached the barrier.
This allows a programmer
to **synchronize** sequences in the parallel process.

#pragma omp **critical** { … }

A critical directive ensures that
<u>a line of code</u> is only run <u>by one process</u> at a time,
ensuring **thread safety** in the body of code.

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Barrier (1)

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    //define loop iterator variable outside parallel region
    int i;
    int thread_id;

    #pragma omp parallel
    {
        thread_id = omp_get_thread_num();

        //create the loop to have each thread print hello.
        for(i = 0; i < omp_get_max_threads(); i++){
            printf("Hello from process: %d\n", thread_id);
        }
    }
    return 0;
}
```

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Barrier (2)

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int i;
    int thread_id;

    #pragma omp parallel
    {
        thread_id = omp_get_thread_num();

        for(i = 0; i < omp_get_max_threads(); i++){
            if(i == thread_ID){
                printf("Hello from process: %d\n", thread_id);
            }
        }
    }
    return 0;
}
```

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

Barrier (3)

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int i;
    int thread_id;

    #pragma omp parallel
    {
        thread_id = omp_get_thread_num();

        for( int i = 0; i < omp_get_max_threads(); i++){
            if(i == omp_get_thread_num()){
                printf("Hello from process: %d\n", thread_id);
            }
            #pragma omp barrier
        }
    }
    return 0;
}
```

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

OMP for

OpenMP's power comes from
easily splitting a larger task into multiple smaller tasks.
Work-sharing directives allow for simple and effective **splitting**
of normally serial tasks into fast parallel sections of code.

The directive omp for divides a normally serial for loop into a parallel task.

#pragma omp for { … }

https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html#parallel-hello-world-program

OMP for

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int partial_Sum, total_Sum;

    #pragma omp parallel private(partial_Sum) shared(total_Sum)
    {
        partial_Sum = 0;
        total_Sum = 0;

        #pragma omp for
        {
            for(int i = 1; i <= 1000; i++){
                partial_Sum += i;
            }
        }

        //Create thread safe region.
        #pragma omp critical
        {
            //add each threads partial sum to the total sum
            total_Sum += partial_Sum;
        }
    }

    printf("Total Sum: %d\n", total_Sum);
    return 0;
}
```

Data Sharing Rules – Implicit Rules

```
int n = 10;                    // shared
int a = 7;                     // shared
```

```
#pragma omp parallel for
for (int i = 0; i < n; i++)        // i private
{
    int b = a + i;                 // b private

    ...
}
```

http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html

Data Sharing Rules – Explicit Rules

```
#pragma omp parallel for shared(n, a)
for (int i = 0; i < n; i++)
{
    int b = a+ i;

    ...
}
```

```
#pragma omp parallel for shared(n, a) private(b)
for (int i = 0; i < n; i++)
{
    b = a + i;

    ...
}
```

http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html

Data Sharing Rules – Explicit Rules

int p = 0;
// the value of p is 0

```
#pragma omp parallel private(p)
{
    // the value of p is undefined
    p = omp_get_thread_num();
    // the value of p is defined
    ...
}
```
// the value of p is undefined

```
#pragma omp parallel
{
    int p = omp_get_thread_num();
    ...
}
```

http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html

Data Sharing Rules – Default(Shared)

int **a**, **b**, **c**, **n**;

...

```
#pragma omp parallel for default(shared)
for (int i = 0; i < n; i++)
{
    // using a, b, c
}
```

http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html

Data Sharing Rules – Default(none)

```
int n = 10;
std::vector<int> vector(n);
int a = 10;
```

```
#pragma omp parallel for default(none) shared(n, vector)
for (int i = 0; i < n; i++)
{
    vector[i] = i * a;
}
```

```
error: 'a' not specified in enclosing parallel
        vector[i] = i * a;
                       ^
error: enclosing parallel
    #pragma omp parallel for default(none) shared(n, vector)
        ^
```

Data Sharing Rules – Default(none)

```
int n = 10;
std::vector<int> vector(n);
int a = 10;
```

```
#pragma omp parallel for default(none) shared(n, vector, a)
for (int i = 0; i < n; i++)
{
    vector[i] = i * a;
}
```

http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html

Data Sharing Rules – Default(none)

The default context of a variable is determined by the following rules:

- **static** variables – **shared**.
- **auto** variables in a **parallel** region – **private**
- **dynamically allocated** objects – **shared**.
- **heap allocated** variables – **shared**.
  there can be <u>only one</u> **<u>shared heap</u>**.
- all variables defined <u>outside</u> a **parallel** construct
- **– shared** in a **parallel** region
- **loop iteration** variables are **private** <u>within their loops</u>.
  the value of the iteration variable after the **loop**
  is the same as if the **loop** were run sequentially.
- memory allocated within a **parallel** loop
  by the **alloca** function
  <u>persists</u> <u>only</u> for the duration of <u>one iteration</u>,
  and is **private** for each thread.

alloca()

NAME
    alloca - allocate memory that is <u>automatically freed</u>

SYNOPSIS
    #include <alloca.h>

    void *alloca(size_t size);

DESCRIPTION
    The alloca() function allocates size bytes of space in the stack
    frame of the caller. This <u>temporary</u> space is <u>automatically</u> <u>freed</u>
    when the function that called **alloca**() <u>returns</u> to its caller.

RETURN VALUE
    The **alloca**() function returns a pointer to the beginning of the
    allocated space. If the allocation causes stack overflow, program
    behavior is undefined.

https://man7.org/linux/man-pages/man3/alloca.3.html

OpenMP Hello World (1A)
    35
    Young Won Lim
10/22/20

Data Sharing Rules – Default(none)

```
int E1;                         /* shared static                    */

void main (argvc,...) {         /* argvc is shared                  */
   int i;                       /* shared automatic                 */

void *p = malloc(...);          /* memory allocated by malloc       */
                                /* is accessible by all threads (shared) */
                                /* and cannot be privatized         */
```

Data Sharing Rules – Default(none)

```
void main (argvc,...) {              // argvc is shared
  int i;      void *p = malloc(...);

  #pragma omp parallel firstprivate (p)
  {
    int b;                           // private automatic
    static int s;                    // shared static

    #pragma omp for
    for (i =0;...) {
      b = 1;                         // b is still private here !
      foo (i);                       // i is private here because it is an iteration variable
    }
    #pragma omp parallel
    {
      b = 1;                         // b is shared here because it
    }                                // is another parallel region
  }
}
```

Data Sharing Rules – Default(none)

```
int E2;                    /* shared static */

void foo (int x) {         /* x is private for the parallel */
                           /* region it was called from     */

  int c;                   /* c is private for the same reason */
... }
```

Data Sharing Rules – Default(none)

The **private** clause declares the variables in the list to be
private to each thread in a team.

The **firstprivate** clause provides a superset of the functionality
provided by the private clause.
The private variable is <u>initialized</u> by the original value of the variable
when the parallel construct is encountered.

The **lastprivate** clause provides a superset of the functionality
provided by the private clause.
The private variable is <u>updated</u> after the end of the parallel construct.

Data Sharing Rules – Default(none)

The **shared** clause declares the variables in the list to be
shared among all the threads in a team.
All threads within a team access the same storage area for shared variables.

The **reduction** clause performs a reduction on the scalar variables
that appear in the list, with a specified operator.

The **default** clause allows the user
to affect the data-sharing attribute of the variables appeared in the parallel construct.

https://www.ibm.com/support/knowledgecenter/SSLTBW_2.4.0/com.ibm.zos.v2r4.cbcpx01/cuppvars.htm

Nested Parallelism (1)

```
void fun1()
{
    for (int i=0; i<80; i++)

        ...
}


main()
{
  #pragma omp parallel
  {
      #pragma omp for
      for (int i=0; i<100; i++)

          ...


      #pragma omp for
      for (int i=0; i<10; i++)
          fun1();
  }
}
```

the 2nd loop in **main**
   can only be distributed to **10** threads

80 loop iterations in **fun1**
   which will be called **10** times in **main** loop.

total **800** iterations in **fun1** and the **main** loop

This gives much more parallelism potential
   if parallelism can be added in both levels.

https://software.intel.com/content/www/us/en/develop/articles/exploit-nested-parallelism-with-openmp-tasking-model.html

Nested Parallelism (2)

```
void fun1()
{
    #pragma omp parallel for
    for (int i=0; i<80; i++)
        ...
}


main
{
    #Pragma omp parallel
    {
        #pragma omp for
        for (int i=0; i<100; i++)
            …

        #pragma omp for
        for (int i=0; i<10; i++)
            fun1();
    }
}
```

may either have <u>insufficient threads</u> for the 1st main loop as it has <u>larger loop count</u>, or

create exploded number of threads for the 2nd main loop when OMP_NESTED=TRUE.

The simple solution is to <u>split</u> the parallel region in main and create separate ones for each loop with a distinct thread number specified.

https://software.intel.com/content/www/us/en/develop/articles/exploit-nested-parallelism-with-openmp-tasking-model.html

Nested Parallelism (3)

```
void fun1()
{
    #pragma omp taskloop
    for (int I = 0; i<80; i++)
        ...
}

main
{
    #pragma omp parallel
    {
        #pragma omp for
        for (int i=0; i<100; i++)
            ...

        #pragma omp for
        for (int i=0; i<10; i++)
            fun1();
    }
}
```

don't have to worry about the thread number changes
in 1st and 2nd main loops.

Even though you still have a small amount of (10) threads
allocated for 2nd main loop,
the rest available threads will be able
to be distributed through omp **taskloop** in fun1.

https://software.intel.com/content/www/us/en/develop/articles/exploit-nested-parallelism-with-openmp-tasking-model.html

Tasking

- Tasking was introduced in OpenMP 3.0
- Until then it was <u>impossible</u> to efficiently and easily implement **certain types of parallelism**
- the initial functionality was very **simple** by design
- note that tasks can be **nested**

https://www.openmp.org//wp-content/uploads/sc13.tasking.ruud.pdf

Tasking

Developer
- Use a **pragma** to specify where the tasks are
- Assume that all tasks can be executed <u>independently</u>

OpenMP runtime system
- when a thread encounters a **task** construct,
  a new task is generated
- the **moment** of **execution** of the task
  is up to the **runtime system**
- execution can either be **immediate** or **delayed**
- **completion** of a task can be enforced
  through **task synchronization**

https://www.openmp.org//wp-content/uploads/sc13.tasking.ruud.pdf

Tasking

The task pragma can be used to explicitly define a task.

Use the task pragma when you want to identify a block of code
to be executed in parallel with the code outside the task region.

The task pragma can be useful for parallelizing irregular algorithms
such as pointer chasing or recursive algorithms.

The task directive takes effect only if you specify the SMP compiler option.

Tasking

Tasks are a mechanism that OpenMP uses under the cover:
if you specify something as being parallel,
OpenMP will create a `block of work': a section of code
plus the data environment in which it occurred.
This block is set aside for execution at some later point.

The thread that executes this code segment creates a task,
which will later be executed, probably by a different thread.
The exact timing of the execution of the task is up to a task scheduler ,
which operates invisible to the user.

```
#include <stdlib.h>
#include <stdio.h>
int main(intargc, char *argv[])
{
    printf("A ");
    printf("race ");
    printf("car ");
    printf("\n");
    return(0);
}
```

```
$ cc -fast hello.c
$ ./a.out
A race car
$
```

```
#include <stdlib.h>
#include <stdio.h>
int main(intargc, char *argv[])
{
    #pragma omp parallel  {
        printf("A ");
        printf("race ");
        printf("car ");
    }
    printf("\n");
    return(0);
}
```

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2 $
./a.out
A race car A race car        or
  "A A race race car car"   or
  "A race A car race car"   or
  "A race A race car car"
```

https://www.openmp.org//wp-content/uploads/sc13.tasking.ruud.pdf

Task example (2)

```c
#include <stdlib.h>
#include <stdio.h>
int main(intargc, char *argv[])
{
    #pragma omp parallel   {
        #pragma omp single   {
            printf("A ");
            printf("race ");
            printf("car ");
        }
    }
    printf("\n");
    return(0);
}
```

```
$ cc -xopenmp –fast hello.c
$ export OMP_NUM_THREADS=2 $
./a.out
A race car
```

```c
#include <stdlib.h>
#include <stdio.h>
int main(intargc, char *argv[])
{
    #pragma omp parallel   {
        #pragma omp single   {
            printf("A ");
            #pragma omp task   {  printf("race ");}
            #pragma omp task   {  printf("car "); }
        }
    }
    printf("\n");
    return(0);
}
```

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out        A race car
$ ./a.out        A race car
$ ./a.out        A car race
$
```

https://www.openmp.org//wp-content/uploads/sc13.tasking.ruud.pdf

Task example (3)

```
#include <stdlib.h>
#include <stdio.h>
int main(intargc, char *argv[])
{
   #pragma omp parallel   {
      #pragma omp single   {
         printf("A ");
         #pragma omp task   {  printf("race ");}
         #pragma omp task   {  printf("car "); }
         printf("is fun to watch ");
      }
   }
   printf("\n");
   return(0);
}


$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out       A is fun to watch race car
$ ./a.out       A is fun to watch race car
$ ./a.out       A is fun to watch car race
$
```

https://www.openmp.org//wp-content/uploads/sc13.tasking.ruud.pdf

```
#include <stdlib.h>
#include <stdio.h>
int main(intargc, char *argv[])
{
   #pragma omp parallel   {
      #pragma omp single   {
         printf("A ");
         #pragma omp task   {  printf("race "); }
         #pragma omp task   {  printf("car "); }
         #pragma omp taskwait {  printf("is fun to watch "); }
      }
   }
   printf("\n");
   return(0);
}


$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out       A race car is fun to watch
$ ./a.out       A race car is fun to watch
$ ./a.out       A car race is fun to watch
$
```

taskloop

The **taskloop** pragma is used to specify
that the iterations of one or more associated loops
are executed in parallel using OpenMP tasks.
 The iterations are <u>distributed across tasks</u> that are
created by the construct and scheduled to be executed.

The taskloop construct generates as many as 20 tasks.
The iterations of the for loop are distributed among the tasks
generated for the **taskloop** construct.

```
#pragma omp parallel
#pragma omp single                          // only one process performs taskloop
#pragma omp taskloop num_tasks(20)
  for (i=0; i<N; i++) {
    arr[i] = i*i;
  }
```

taskloop

#pragma omp **parallel**
#pragma omp **single**
#pragma omp **taskloop** **num_tasks**(20)
  for (i=0; i<N; i++) {
    arr[i] = i*i;
}
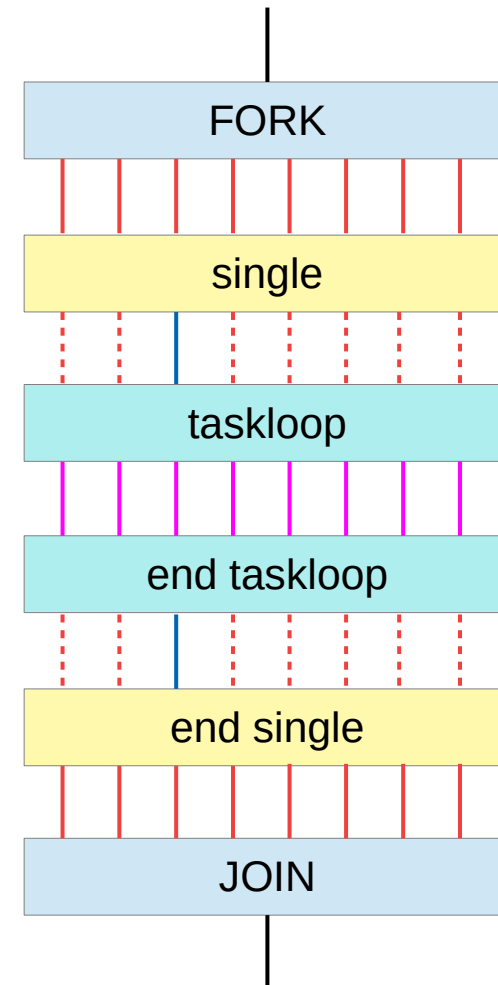
taskwait

Completion of a subset of all explicit tasks bound to
a given parallel region may be specified
through the use of the **taskwait** directive.

The **taskwait** directive specifies a wait
on the completion of child tasks generated
since the beginning of the current (implicit or explicit) task.

Note that the taskwait directive specifies a wait
on the completion of direct children tasks, not all descendant tasks.

https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

Tasking example

```c
#include <stdio.h>
#include <omp.h>
int fib(int n)
{
  int i, j;
  if (n<2)
    return n;
  else
   {
      #pragma omp task shared(i) firstprivate(n)
      i=fib(n-1);

      #pragma omp task shared(j) firstprivate(n)
      j=fib(n-2);

      #pragma omp taskwait
      return i+j;
   }
}
```

```c
int main()
{
  int n = 10;

  omp_set_dynamic(0);
  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
}
```

```
% CC -xopenmp -xO3 task_example.cc
% a.out
fib(10) = 55
```

https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

Tasking example

The following C/C++ program illustrates
how the OpenMP task and taskwait directives
can be used to compute Fibonacci numbers recursively.

In the example, the parallel directive denotes
a parallel region which will be executed by four threads.
In the parallel construct, the single directive is used
to indicate that only one of the threads
will execute the print statement that calls fib(n).

Tasking example

The call to fib(n) generates two tasks,
indicated by the task directive.
One of the tasks computes fib(n-1) and
the other computes fib(n-2),
and the return values are added together
to produce the value returned by fib(n).
Each of the calls to fib(n-1) and fib(n-2)
will in turn generate two tasks.
Tasks will be recursively generated
until the argument passed to fib() is less than 2.

https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

Tasking example

The taskwait directive ensures that
the two tasks generated in an invocation of fib()
are completed (that is. the tasks compute i and j)
before that invocation of fib() returns.

Note that although only one thread executes the single directive
and hence the call to fib(n), all four threads will participate
in executing the tasks gener

Single

The **single** construct specifies that
the associated structured block is
executed by only one of the threads in the team
(not necessarily the master thread),
in the context of its **implicit task**.

The other threads in the team,
which do not execute the block,
wait at an **implicit barrier**
at the end of the single construct
unless a **nowait** clause is specified.

Single

denotes block of code
to be <u>executed</u> by only one thread

- first thread to arrive is <u>chosen</u>
- **implicit barrier** <u>at end</u>

```
#pragma omp parallel
{
  a();
  #pragma omp single
  {
    b();
  } // threads wait here for single
  c();
}
```

chosen

a(0)	a(1)	c(0)
a(2)	a(3)	c(1)
a(4)	b	c(2)
a(8)		c(3)

Master

Denotes block of code to be executed only by the master thread
No **implicit barrier** <u>at end</u>

```
#pragma omp parallel
{
  a();
  #pragma omp master
  {    // if not master skip to next stmtp
      b();
  }
  c();
|
```
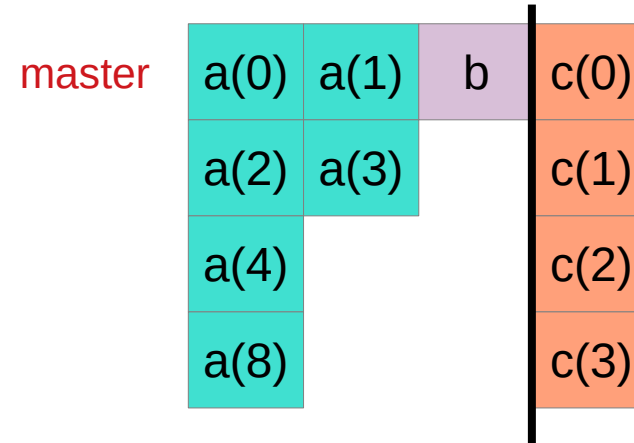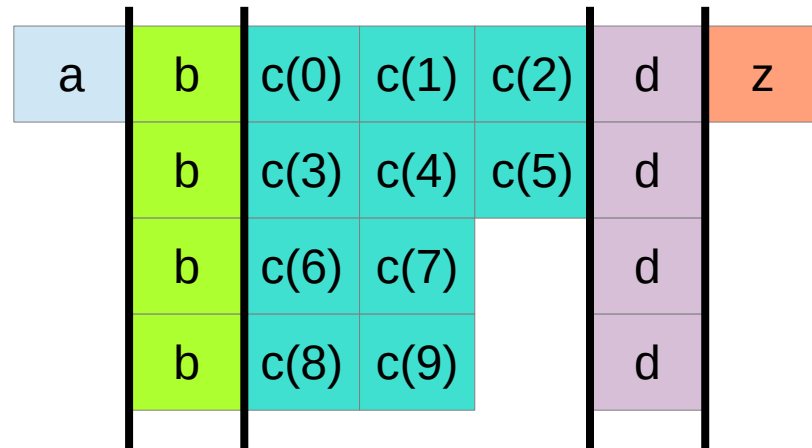
master	a(0)	a(1)	b	c(0)
	a(2)	a(3)		c(1)
	a(4)			c(2)
	a(8)			c(3)

Nowait (1)

In an omp parallel region, automatically wait for all threads to finish
In an omp for loop, a synchronization point after the end of the loop

```
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```
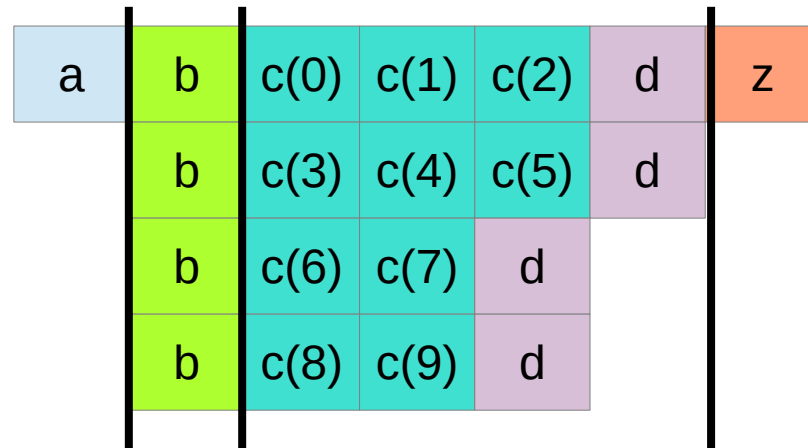
Nowait (2)

no thread will execute d() until all threads are done with the loop:
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();
```


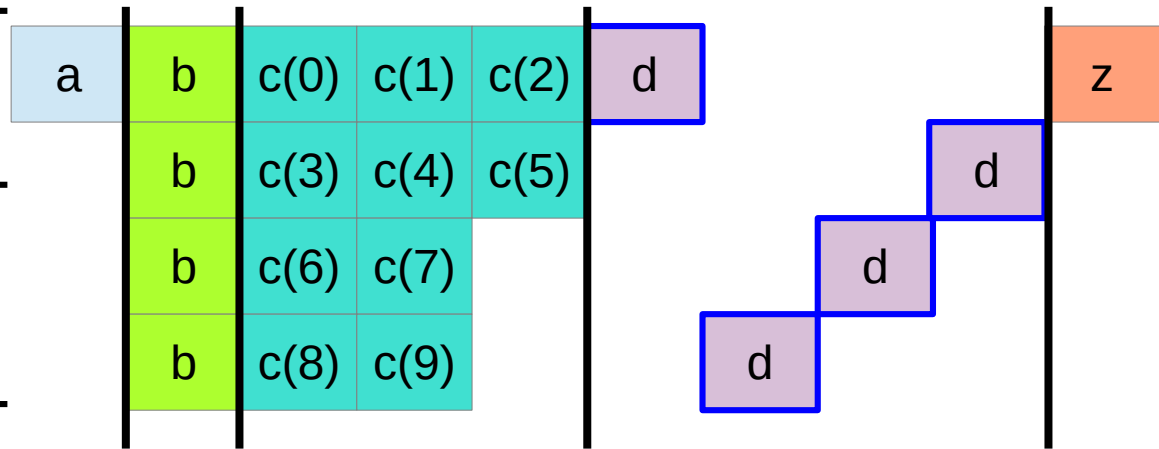
http://ppc.cs.aalto.fi/ch3/nowait/

Nowait (3)

for a critical section after a loop,
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();
```


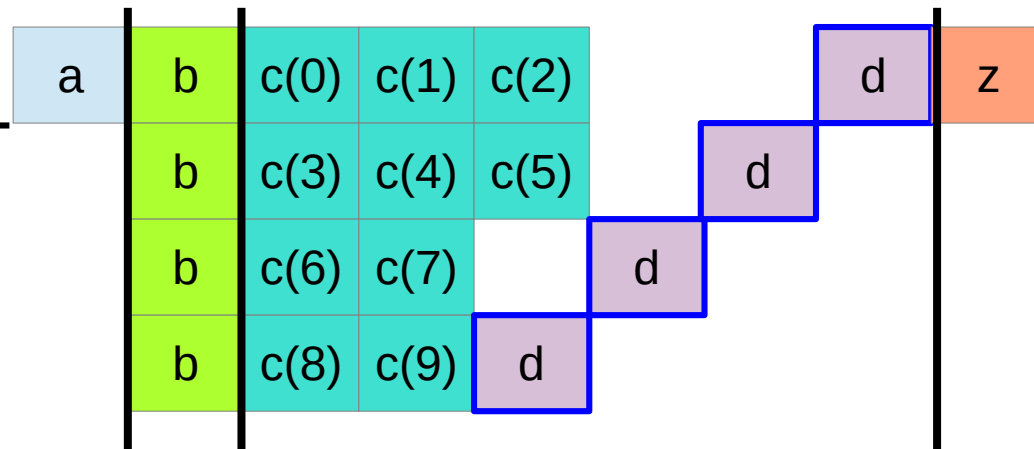http://ppc.cs.aalto.fi/ch3/nowait/

Nowait (4)

disable this waiting, so that some threads can start doing postprocessing 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();
```

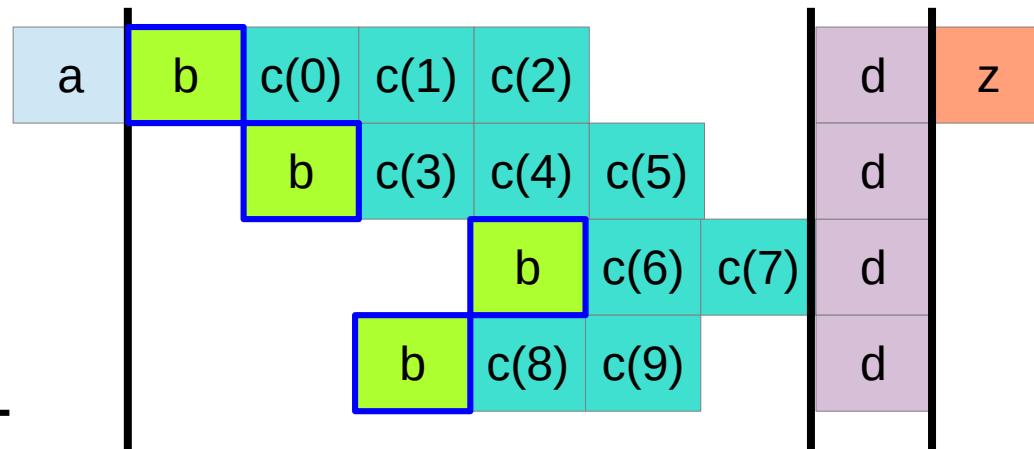| a	b	c(0)	c(1)	c(2)				d	z
	b	c(3)	c(4)	c(5)			d		
	b	c(6)	c(7)			d			
	b	c(8)	c(9)	d					

http://ppc.cs.aalto.fi/ch3/nowait/

Nowait (5)

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();
```

a	b	c(0)	c(1)	c(2)		d	z
		b	c(3)	c(4)	c(5)		d
			b	c(6)	c(7)		d
		b	c(8)	c(9)			d

http://ppc.cs.aalto.fi/ch3/nowait/

Implicit task (1)

In addition to **explicit tasks** specified using the **task** directive,
the OpenMP specification version 3.0 introduces
the notion of **implicit tasks**.

An **implicit task** is a task generated by the **implicit parallel region**,
or generated when a **parallel construct** is encountered during execution.

The **code** for each **implicit task** is the code inside the **parallel construct**.

Each **implicit task** is assigned to a different **thread** in the **team** and is **tied**;

that is, an **implicit task** is always executed from beginning to end
by the **thread** to which it is initially assigned.

https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

Implicit task (2)

All **implicit tasks** generated
    when a **parallel construct** is encountered
are guaranteed to be <u>complete</u>
    when the **master thread** exits the **implicit barrier**
    at the end of the parallel region.

On the other hand,
all **explicit tasks** generated within a **parallel region**
are guaranteed to be <u>complete</u>
    on <u>exit</u> from the <u>next</u> **implicit** or **explicit barrier**
    within the parallel region.

https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

Implicit task (3)

When an **if clause** is present on a **task construct**
and the value of the scalar-expression evaluates to false,
the thread that encounters the task must immediately execute the task.

The **if clause** can be used to avoid the overhead of
generating many finely grained tasks and
placing them in the conceptual pool.

Implicit barrier

Implicit BarriersSeveral OpenMP* constructs have implicit barriers

- parallel
- for
- single

Unnecessary barriers hurt performance

- Waiting threads accomplish no work!

Waiting threads accomplish no work!
Suppress implicit barriers, when safe, with the nowait

#pragma omp

1. pragmas for defining **parallel regions**
in which work is done by threads in parallel (**#pragma omp parallel**).
Most of the OpenMP directives either statically or dynamically bind
to an enclosing parallel region

2. pragmas for defining how work is **distributed** or **shared**
across the threads in a parallel region
(**#pragma omp sections**, **#pragma omp for**, **#pragma omp single**, **#pragma omp task**).

3. pragmas for **controlling synchronization** among threads
(**#pragma omp atomic**, **#pragma omp master**, **#pragma omp barrier**,
#pragma omp critical, **#pragma omp flush**, **#pragma omp ordered**) .

4. pragmas for defining the **scope** of **data visibility**
across parallel regions within the same thread
(**#pragma omp threadprivate**).

5. pragmas for **synchronization**
(**#pragma omp taskwait**, **#pragma omp barrier**)

https://www.ibm.com/support/knowledgecenter/SSLTBW_2.4.0/com.ibm.zos.v2r4.cbcpx01/cuppovrv2.htm

#pragma omp

The #pragma omp pragmas generally appear
immediately before the section of code to which they apply.

The following code defines a parallel region
in which iterations of a for loop can run in parallel:

```
#pragma omp parallel
{
  #pragma omp for
    for (i=0; i<n; i++)
      ...
}
```

Young Won Lim
10/22/20

#pragma omp

The following example defines a parallel region
in which two or more non-iterative sections of program code
can run in parallel:

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
      structured_block_1
          ...
    #pragma omp section
      structured_block_2
          ...
        ....
  }
}
```

Sections, section

The omp **section** directive is <u>optional</u>
for the <u>first</u> program <u>code segment</u>
inside the omp **sections** directive.

Following segments <u>must</u> be preceded
by an omp **section** directive.

All omp section directives must appear
within the lexical construct of
the program source code segment
associated with the omp sections directive.

When program execution reaches a omp **sections** directive,
program segments defined by the following omp **section** directive
are <u>distributed for parallel execution</u> among available threads.

A <u>barrier</u> is <u>implicitly defined</u> at the <u>end</u> of the larger program region
associated with the omp sections directive unless the **nowait** clause is specified.

Sections

Parallel SectionsIndependent sections of code can execute concurrently

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();

    #pragma omp section
    phase2();

    #pragma omp section
    phase3();
}
```

Single (1)

```
int main()
{
    int salaries1 = 0;
    int salaries2 = 0;

    for (int employee = 0; employee < 25000; employee++)
    {
        salaries1 += fetchTheSalary(employee, Co::Company1);
    }

    std::cout << "Salaries1: " << salaries1 << std::endl;

    for (int employee = 0; employee < 25000; employee++)
    {
        salaries2 += fetchTheSalary(employee, Co::Company2);
    }

    std::cout << "Salaries2: " << salaries2 << std::endl;

    return 0;
}
```

http://jakascorner.com/blog/2016/06/omp-single.html
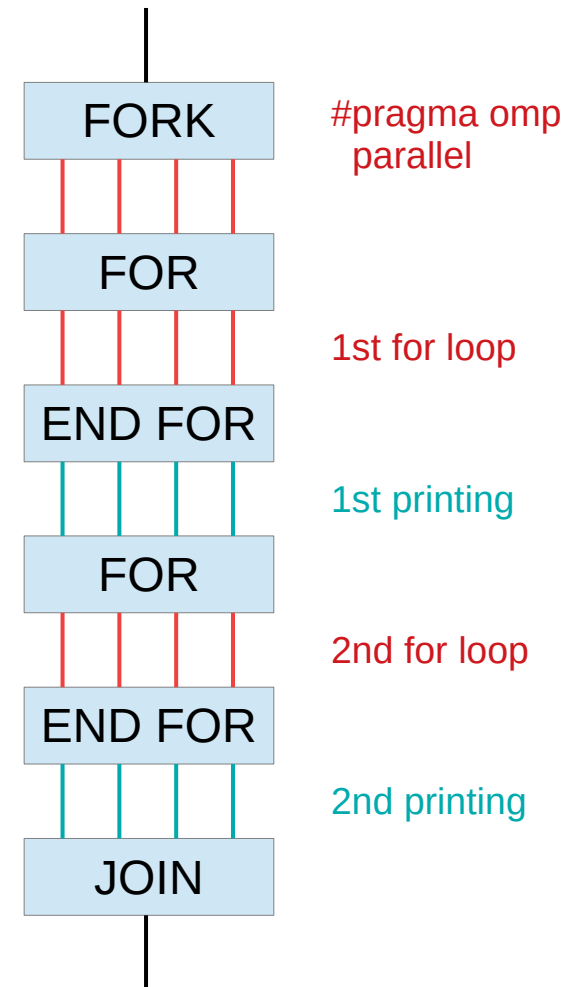
Young Won Lim
10/22/20

Single (2)

```cpp
int salaries1 = 0;
int salaries2 = 0;

#pragma omp parallel shared(salaries1, salaries2)
{
    #pragma omp for reduction(+: salaries1)
    for (int employee = 0; employee < 25000; employee++)
    {
        salaries1 += fetchTheSalary(employee, Co::Company1);
    }

    std::cout << "Salaries1: " << salaries1 << std::endl;

    #pragma omp for reduction(+: salaries2)
    for (int employee = 0; employee < 25000; employee++)
    {
        salaries2 += fetchTheSalary(employee, Co::Company2);
    }

    std::cout << "Salaries2: " << salaries2 << std::endl;
}
```

FORK — #pragma omp parallel

FOR — 1st for loop

END FOR — 1st printing

FOR — 2nd for loop

END FOR — 2nd printing

JOIN

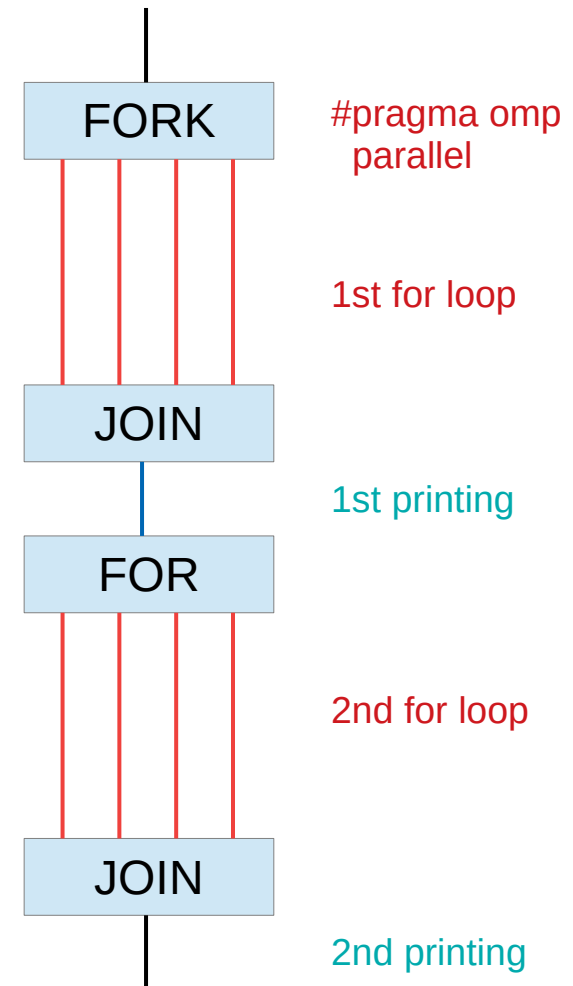http://jakascorner.com/blog/2016/06/omp-single.html

Single (v1)

```
#pragma omp parallel for reduction(+: salaries1)
for (int employee = 0; employee < 25000; employee++)
{
    salaries1 += fetchTheSalary(employee, Co::Company1);
}

std::cout << "Salaries1: " << salaries1 << std::endl;

#pragma omp parallel for reduction(+: salaries2)
for (int employee = 0; employee < 25000; employee++)
{
    salaries2 += fetchTheSalary(employee, Co::Company2);
}

std::cout << "Salaries2: " << salaries2 << std::endl;
```

FORK — #pragma omp parallel

1st for loop

JOIN

1st printing

FOR

2nd for loop

JOIN

2nd printing

http://jakascorner.com/blog/2016/06/omp-single.html
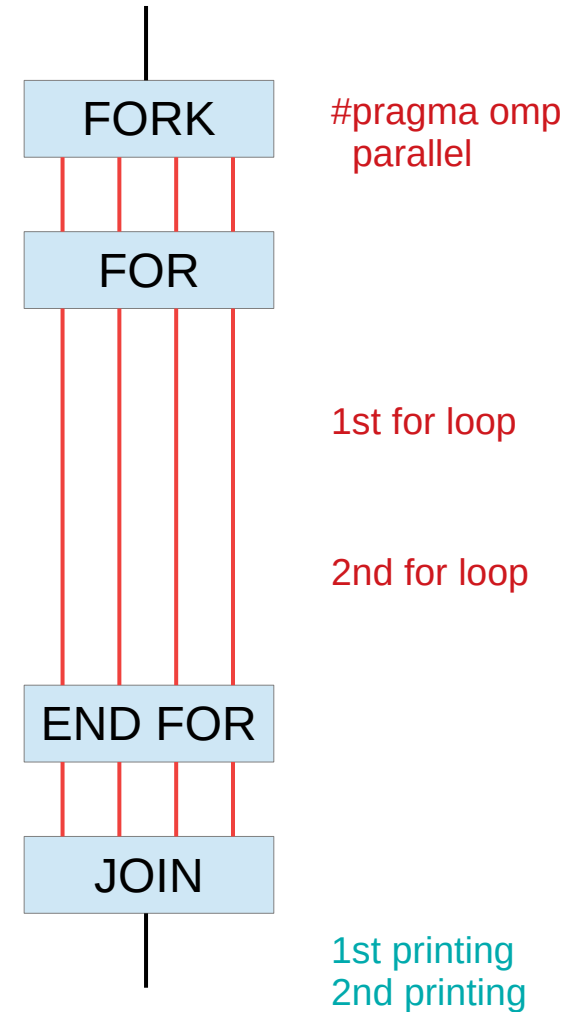
Single (v2)

```
#pragma omp parallel for reduction(+: salaries1, salaries2)
for (int employee = 0; employee < 25000; employee++)
{
    salaries1 += fetchTheSalary(employee, Co::Company1);
    salaries2 += fetchTheSalary(employee, Co::Company2);
}
```

```
std::cout << "Salaries1: " << salaries1 << "\n"
       << "Salaries2: " << salaries2 << std::endl;
```
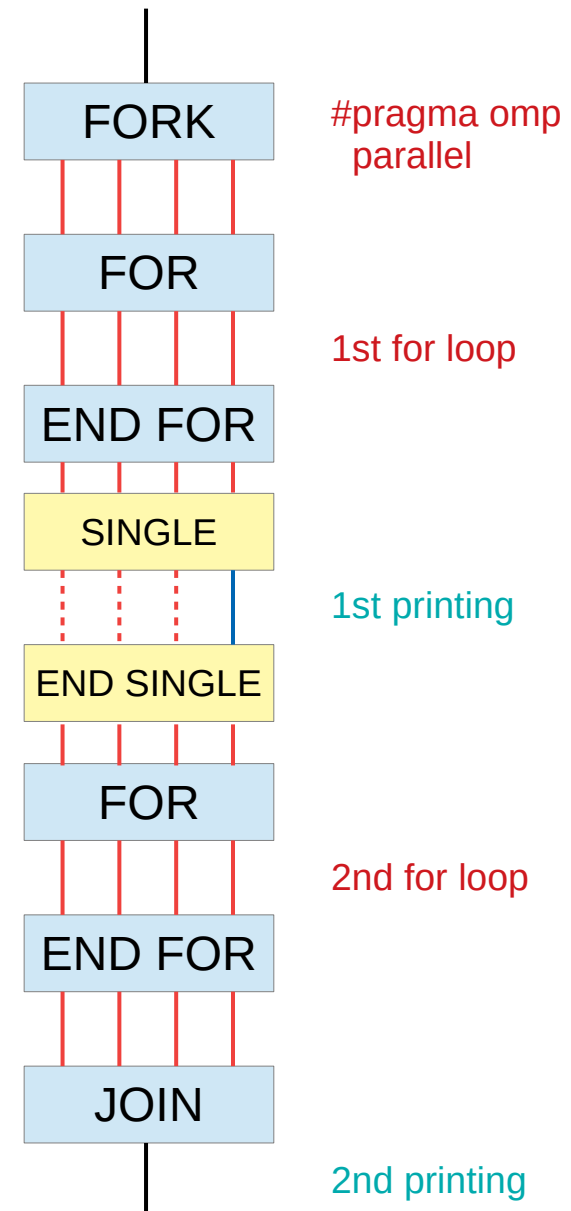
FORK — #pragma omp parallel

FOR

1st for loop

2nd for loop

END FOR

JOIN

1st printing
2nd printing

http://jakascorner.com/blog/2016/06/omp-single.html

Single (v3)

```cpp
#pragma omp parallel shared(salaries1, salaries2)
{
    #pragma omp for reduction(+: salaries1)
    for (int employee = 0; employee < 25000; employee++)
    {
        salaries1 += fetchTheSalary(employee, Co::Company1);
    }

    #pragma omp single
    {
        std::cout << "Salaries1: " << salaries1 << std::endl;
    }

    #pragma omp for reduction(+: salaries2)
    for (int employee = 0; employee < 25000; employee++)
    {
        salaries2 += fetchTheSalary(employee, Co::Company2);
    }
}

std::cout << "Salaries2: " << salaries2 << std::endl;
```

http://jakascorner.com/blog/2016/06/omp-single.html

FORK — #pragma omp parallel

FOR

END FOR — 1st for loop

SINGLE

1st printing

END SINGLE

FOR

END FOR — 2nd for loop

JOIN

2nd printing

taskloop

```
Int main (int argc, char* argv[])
{
    ***

  #pragma omp parallel
  {
    #pragrma omp single
    {
      fib(input);
    }
  }
    ***

}
```

```
Int fib(int n)
{
  if (n < 2) return n;
  int x, y;

  #pragma omp task shared(x)
  {
    x = fib(n-1);
  }
  #pragma omp task shared(y)
  {
    y = fib(n-2);
  }
  #pragma omp taskwait;
  {
    return x+y;
  }
}
```

https://pop-coe.eu/sites/default/files/pop_files/pop-webinar-openmptasking.pdf

References

[1]   en.wikipedia.org
[2]   M Harris, http://beowulf.lcs.mit.edu/18.337-2008/lectslides/scan.pdf