Multicore & GPU Programming: OpenMP basics

Raymond Namyst, Pierre-André Wacrenier

Dept. of Computer Science

University of Bordeaux, France

https://gforgeron.gitlab.io/it224/

The OpenMP standard (www.openmp.org)

- Parallel Programming Interface designed for shared-memory multiprocessor machines
  - Language extensions to C, C++ and Fortran

- Incremental parallelization
  - `#pragma omp` *`directive`*
  - Less intrusive than adding calls to libraries (e.g. POSIX threads)
  - Pragmas can be ignored to easily switch back to the original sequential code
    - Hmm, really?

The OpenMP standard (www.openmp.org)

- Incremental parallelization
  - Pragmas are like “On my honor, I swear that this code is parallel”
    - Compiler will trust you! (no check)
  - `#pragma omp` *`directive clause clause`* …
    - The more you say, the more performance you can get (hopefully)
  - Seems like a piece of cake, uh?
- The OpenMP standard keeps evolving
  - Architecture Review Board (Intel, IBM, AMD, Microsoft, Oracle, etc.)

Our first “Hello World” program

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

int main ()
{
#pragma omp parallel
  printf ("Hello world!\n");
  printf ("Bye!\n");

  return EXIT_SUCCESS;
}
```

```
[my-machine] make
gcc -Wall  hello.c  -o hello
[my-machine] ./hello
Hello world!
Bye!
```

Our first “Hello World” program

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

int main ()
{
#pragma omp parallel
  printf ("Hello world!\n");
  printf ("Bye!\n");

  return EXIT_SUCCESS;
}
```

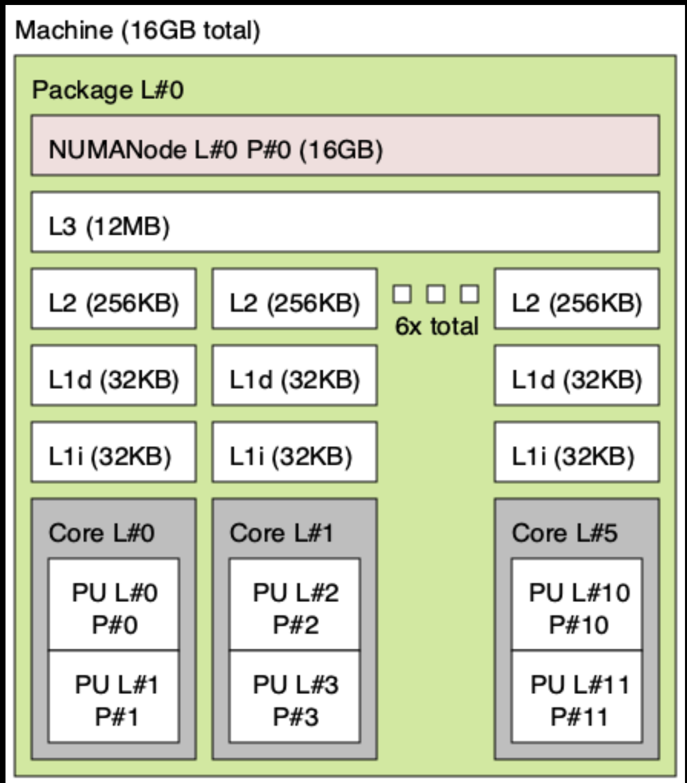
```
[my-machine] make
gcc -Wall  -fopenmp hello.c  -o hello
[my-machine] ./hello
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Bye!
```

Our first “Hello World” program

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

int main ()
{
#pragma omp parallel
  printf ("Hello world!\n");
  printf ("Bye!\n");

  return EXIT_SUCCESS;
}
```

```
[my-machine] make
gcc -Wall  -fopenmp hello.c  -o hello
[my-machine] ./hello  | cat -n
    1  Hello world!
    2  Hello world!
    3  Hello world!
    4  Hello world!
    5  Hello world!
    6  Hello world!
    7  Hello world!
    8  Hello world!
    9  Hello world!
   10  Hello world!
   11  Hello world!
   12  Hello world!
   13  Bye!
```

Our first “Hello World” program



Output of the “lstopo” command on my-machine

```
[my-machine] make
gcc -Wall  -fopenmp hello.c   -o hello
[my-machine] ./hello  | cat -n
     1  Hello world!
     2  Hello world!
     3  Hello world!
     4  Hello world!
     5  Hello world!
     6  Hello world!
     7  Hello world!
     8  Hello world!
     9  Hello world!
    10  Hello world!
    11  Hello world!
    12  Hello world!
    13  Bye!
```

Our first “Hello World” program

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

int main ()
{
#pragma omp parallel
  printf ("Hello world!\n");
  printf ("Bye!\n");

  return EXIT_SUCCESS;
}
```

```
[my-machine] make
gcc -Wall  -fopenmp hello.c   -o hello
[my-machine] OMP_NUM_THREADS=4 ./hello | cat -n
     1  Hello world!
     2  Hello world!
     3  Hello world!
     4  Hello world!
     5  Bye!
```

Our first “Hello World” program

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

int main ()
{
#pragma omp parallel num_threads(6)
  printf ("Hello world!\n");
  printf ("Bye!\n");

  return EXIT_SUCCESS;
}
```

```
[my-machine] make
gcc -Wall  -fopenmp hello.c   -o hello
[my-machine] ./hello | cat -n
   1  Hello world!
   2  Hello world!
   3  Hello world!
   4  Hello world!
   5  Hello world!
   6  Hello world!
   7  Bye!
```

Our first "Hello World" program

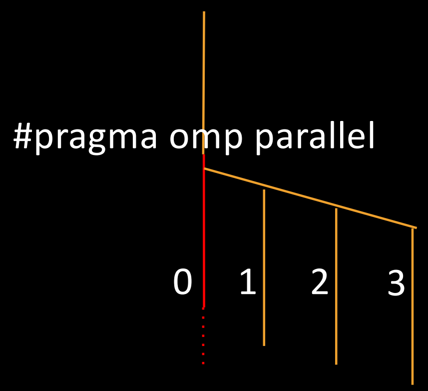
```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main ()
{
#pragma omp parallel num_threads(6)
  printf ("Hello world!\n");
  printf ("Bye!\n");

  return EXIT_SUCCESS;
}
```

Usually not a good idea

```
[my-machine] make
gcc -Wall  -fopenmp hello.c   -o hello
[my-machine] ./hello | cat -n
   1  Hello world!
   2  Hello world!
   3  Hello world!
   4  Hello world!
   5  Hello world!
   6  Hello world!
   7  Bye!
```

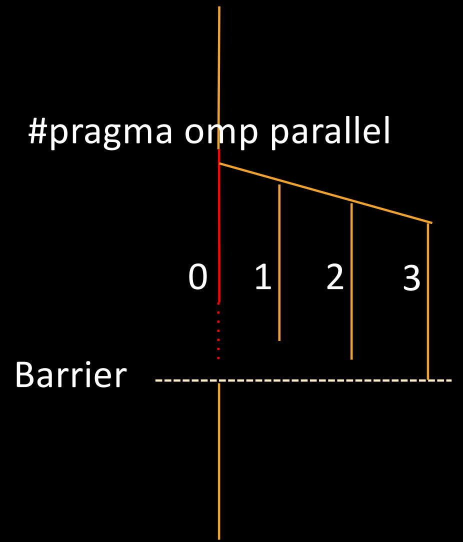
Fork-Join parallelism

- A single thread initially executes the main function
- When it reaches a “parallel” directive
  - A team of threads is created
  - The initial thread is part of the team (and is the master)
  - Each thread executes the parallel region



Fork-Join parallelism

- At the end of the parallel region
  - All threads enter a synchronization barrier (*rendez-vous*)
  - When all threads have reached the barrier, all threads but the master are freed
  - The master thread can then continue executing code beyond the region



How to introduce divergence?

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

int main ()
{
#pragma omp parallel
  printf ("Hello from %d!\n", omp_get_thread_num());
  printf ("Bye!\n");

  return EXIT_SUCCESS;
}
```

```
[my-machine] make
gcc -Wall  -fopenmp hello.c   -o hello
[my-machine] OMP_NUM_THREADS=4 ./hello
Hello from 0!
Hello from 3!
Hello from 1!
Hello from 2!
Bye!
```

How to introduce divergence?

```
int main()
{
#pragma omp parallel
  {
    switch (omp_get_thread_num())
    {
      case 0:
        f(); break;
      case 1:
        g(); break;
      ...
    }
  }
  return EXIT_SUCCESS;
}
```

- Not a sound solution
  - Parallelism does not always depend on the number of OpenMP threads!
- Our program is definitely not an *incremental* evolution of a sequential one any more…

Loop parallelism

```
int main ()
{


    for (int i = 0; i < 10; i++)
      f (i);

  return EXIT_SUCCESS;
}
```

- We assume that f(i) calls can be performed in parallel

Loop parallelism

```
int main ()
{
#pragma omp parallel
  {
    for (int i = 0; i < 10; i++)
      f (i);
  }
  return EXIT_SUCCESS;
}
```

- We assume that f(i) calls can be performed in parallel
- In the current code
  - f(0) is executed by all threads
  - So are f(1), f(2), …

Loop parallelism

```
int main ()
{
#pragma omp parallel
  {
    for (int i = 0; i < 10; i++)
      f (i);
  }
  return EXIT_SUCCESS;
}
```

- We assume that f(i) calls can be performed in parallel
- In the current code
  - f(0) is executed by all threads
  - So are f(1), f(2), …
- We'd like to distribute the iteration range to the thread!

Loop parallelism

```
int main ()
{
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < 10; i++)
      f (i);
  }
  return EXIT_SUCCESS;
}
```

Distribute iteration range

- We assume that f(i) calls can be performed in parallel
- In the current code
  - f(0) is executed by all threads
  - So are f(1), f(2), …
- We'd like to distribute the iteration range to the thread!

Loop parallelism

```
int main ()
{
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < 10; i++)
      printf("f(%d) computed by %d\n",
             i, omp_get_thread_num());
  }
  return EXIT_SUCCESS;
}
```

```
[my-machine] OMP_NUM_THREADS=4 ./loop
f(0) computed by 0
f(1) computed by 0
f(8) computed by 3
f(9) computed by 3
f(6) computed by 2
f(7) computed by 2
f(2) computed by 0
f(3) computed by 1
f(4) computed by 1
f(5) computed by 1
```

Loop parallelism

```
int main ()
{
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < 10; i++)
      printf("f(%d) computed by %d\n",
             i, omp_get_thread_num());
  }
  return EXIT_SUCCESS;
}
```

- By default (with gcc), the iteration range is splitted in chunks
  - Each thread was assigned one chunk of contiguous iterations
  - That is: static partitioning

Loop parallelism

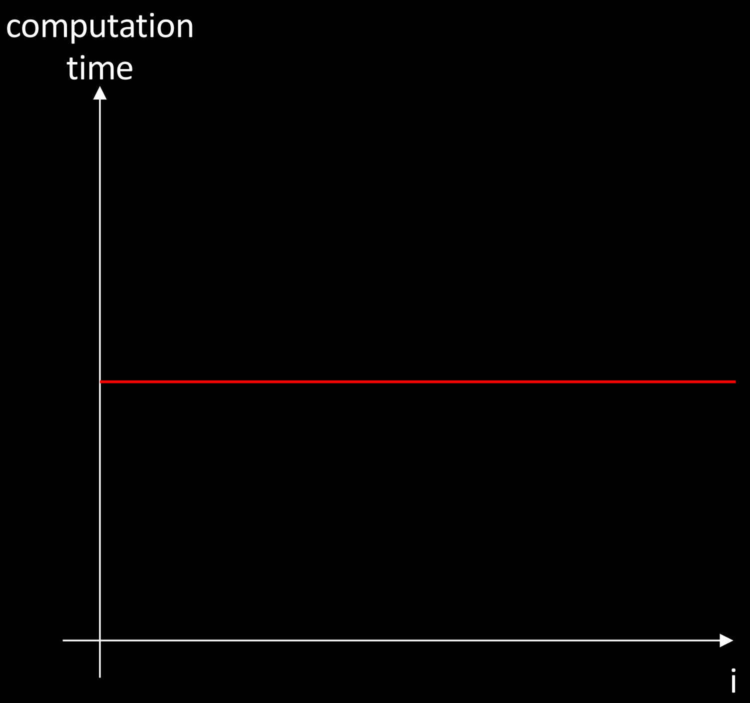
```
int main ()
{
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < 10; i++)
      printf("f(%d) computed by %d\n",
             i, omp_get_thread_num());
  }
  return EXIT_SUCCESS;
}
```

- By default (with gcc), the iteration range is splitted in chunks
  - Each thread was assigned one chunk of contiguous iterations
  - That is: static partitioning
- Side note: an implicit barrier takes place at the end of the loop

Parallelizing computations

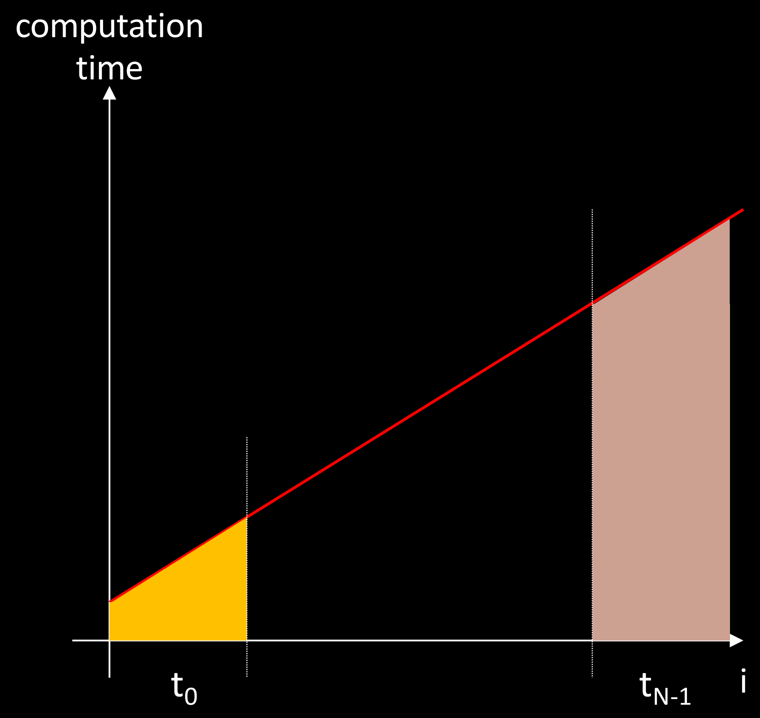
- How good is a static *block* distribution?
  - OK if the computation time of f(i) is constant
    - I.e. does not depend on the value of i

```
#pragma omp for schedule (static)
  for (int i = 0; i < 10; i++)
    f (i);
```



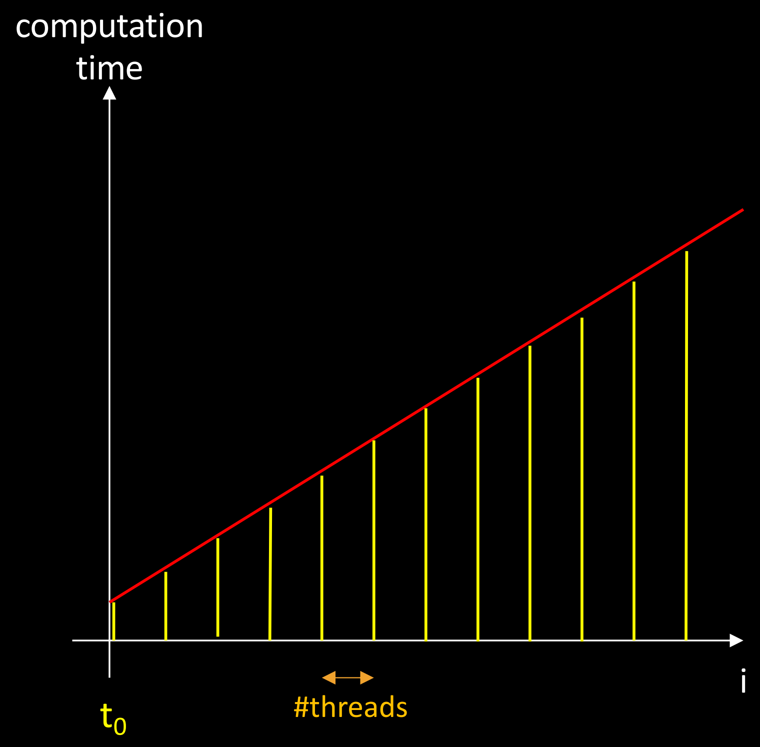
Parallelizing computations

- What if the computation time is linearly increasing?
  - Our block distribution is no longer relevant
    - Well, using a mirror block distribution assigning two blocks per thread would work…
- What kind of distribution should we use?



Parallelizing computations

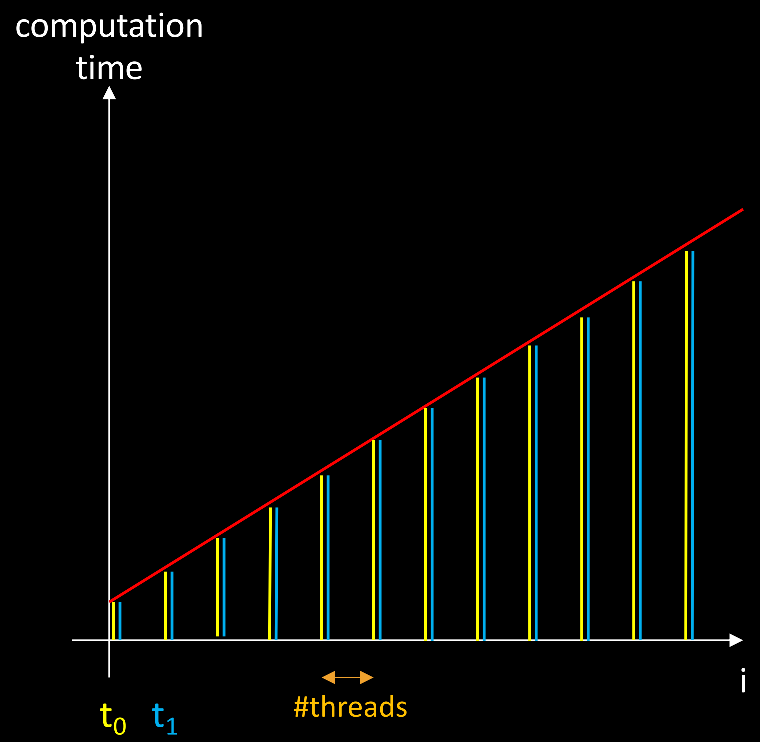
- What if the computation time is linearly increasing?
  - A cyclic distribution of indexes would be a good option



Parallelizing computations

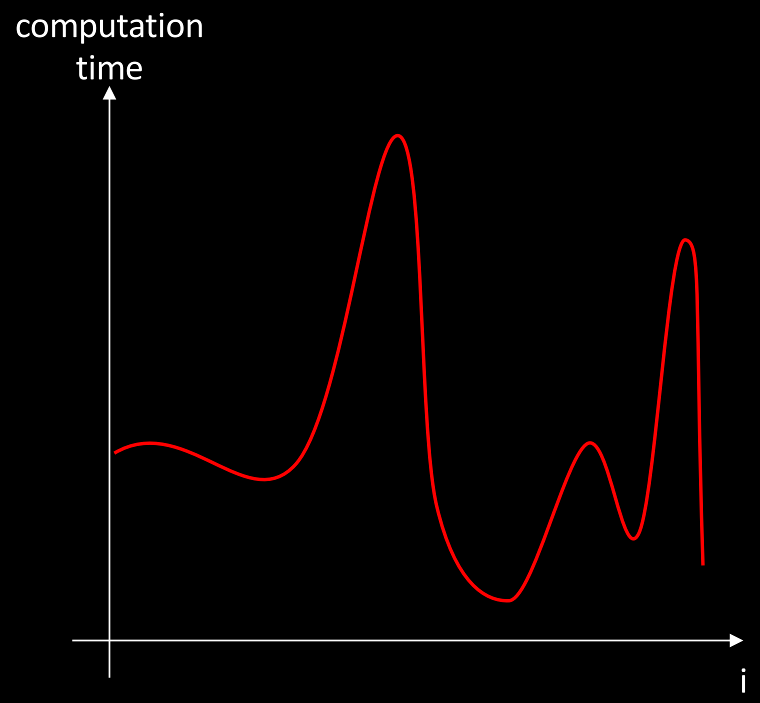
- What if the computation time is linearly increasing?
  - A cyclic distribution of indexes would be a good option

```
#pragma omp for schedule (static, 1)
  for (int i = 0; i < 10; i++)
    f (i);
```



Parallelizing computations

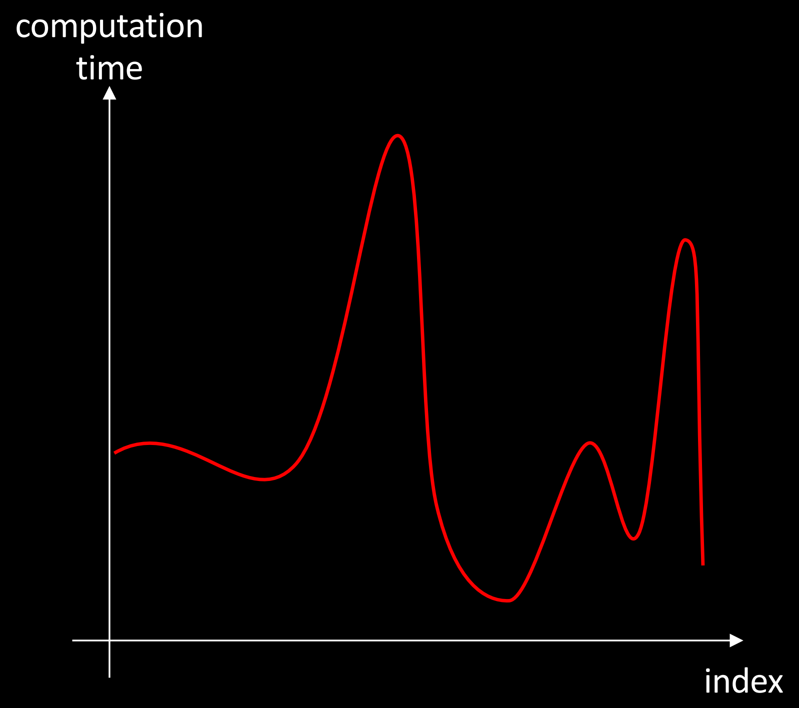
- What if the computation time is unpredictable?
  - Even the cyclic strategy may fail



Parallelizing computations

- What if the computation time is unpredictable?
  - Dynamic strategy
    - Distribute indexes in a greedy manner

```
#pragma omp for schedule (dynamic)
  for (int i = 0; i < 10; i++)
    f (i);
```



Fixing loop scheduling at run time

```
int main ()
{
#pragma omp parallel
  {
#pragma omp for schedule (runtime)
    for (int i = 0; i < 10; i++)
      printf("f(%d) computed by %d\n",
             i, omp_get_thread_num());
  }
  return EXIT_SUCCESS;
}
```

```
[my-machine] OMP_SCHEDULE=dynamic ./loop
f(0) computed by 0
f(2) computed by 1
f(3) computed by 1
f(4) computed by 1
f(5) computed by 1
f(6) computed by 1
f(7) computed by 1
f(8) computed by 1
f(1) computed by 0
f(9) computed by 2
```

Collapsing nested loops

```
int main ()
{
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 4; j++)
        f (i, j);
  }
  return EXIT_SUCCESS;
}
```

- Problem
  - We only distribute 3 i-values to threads
    - Then each threads executed the j-loop sequentially

Collapsing nested loops

```
int main ()
{
#pragma omp parallel
  {
    for (int i = 0; i < 3; i++)
#pragma omp for
      for (int j = 0; j < 4; j++)
        f (i, j);
  }
  return EXIT_SUCCESS;
}
```

- Problem
  - We only distribute 3 i-values to threads
    - Then each threads executed the j-loop sequentially
  - Moving `#pragma omp for` between i-loop and j-loop doesn't help that much

Collapsing nested loops

```
int main ()
{
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 4; j++)
        f (i, j);
  }
  return EXIT_SUCCESS;
}
```

- Ideally, we'd like to perform all the f() calls in parallel on a 12-core machine

Collapsing nested loops

```
int main ()
{
#pragma omp parallel
  {
#pragma omp for collapse (2)
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 4; j++)
        f (i, j);
  }
  return EXIT_SUCCESS;
}
```

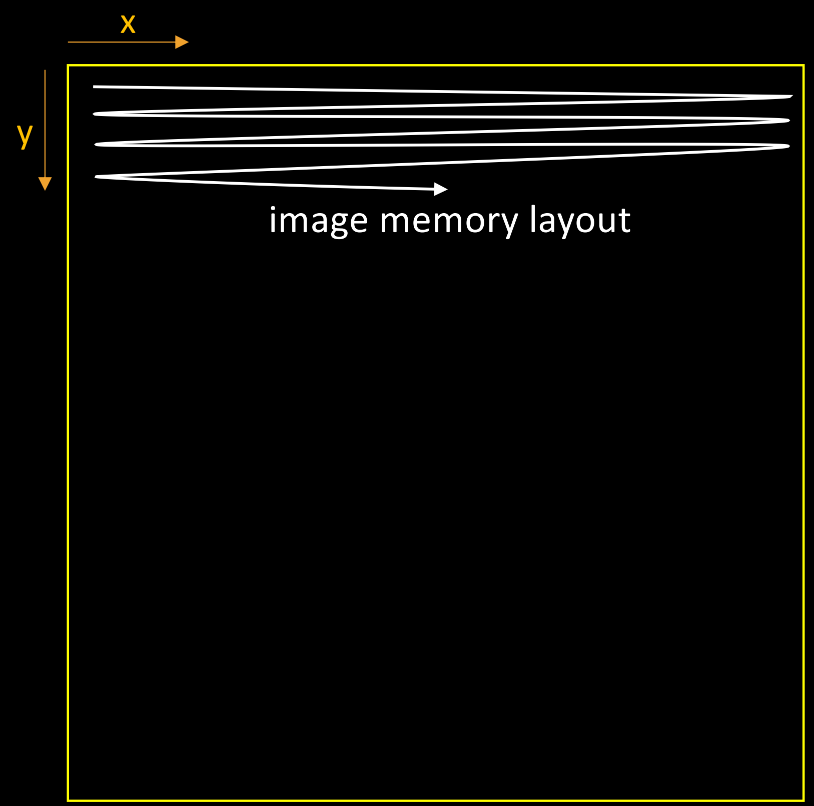
Merge two loops

- Ideally, we'd like to perform all the f() calls in parallel on a 12-core machine
- The collapse clause distributes all possible (i, j) pairs to threads
  - Can be used in conjunction with schedule (*policy*)

Our first EasyPAP kernel

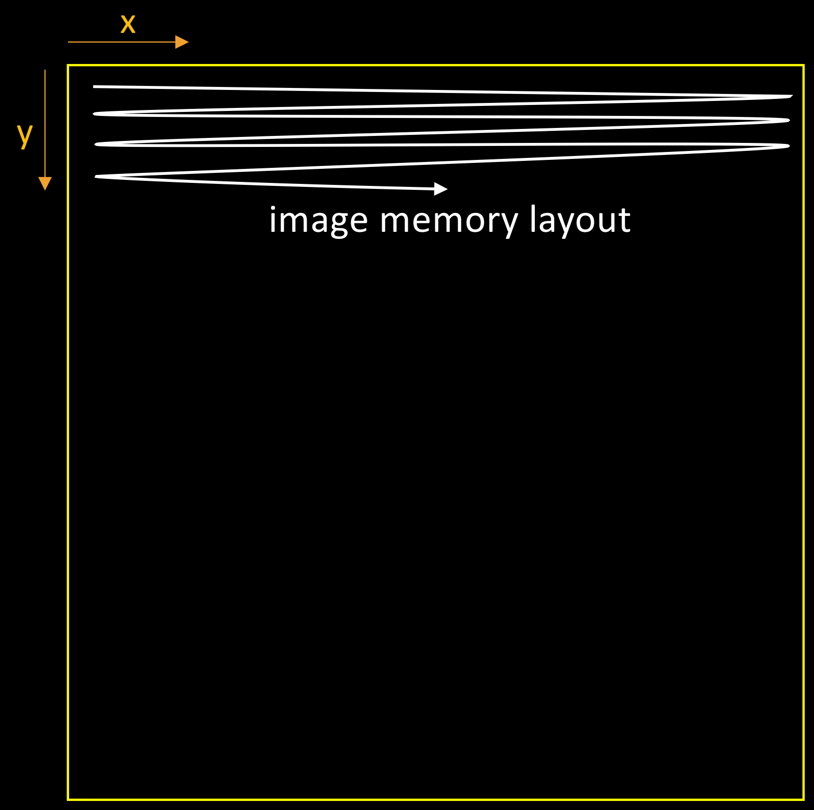
- EasyPAP is a parallel programming framework
  - Design parallel 2D kernels
  - Observe computations in real time
  - Analyze traces post mortem

- Computations work on a DIM x DIM array of pixels
  - unsigned image[DIM][DIM];
    - (pixels format: RGBA8888)
      - image[y][x] = 0xFF0000FF;



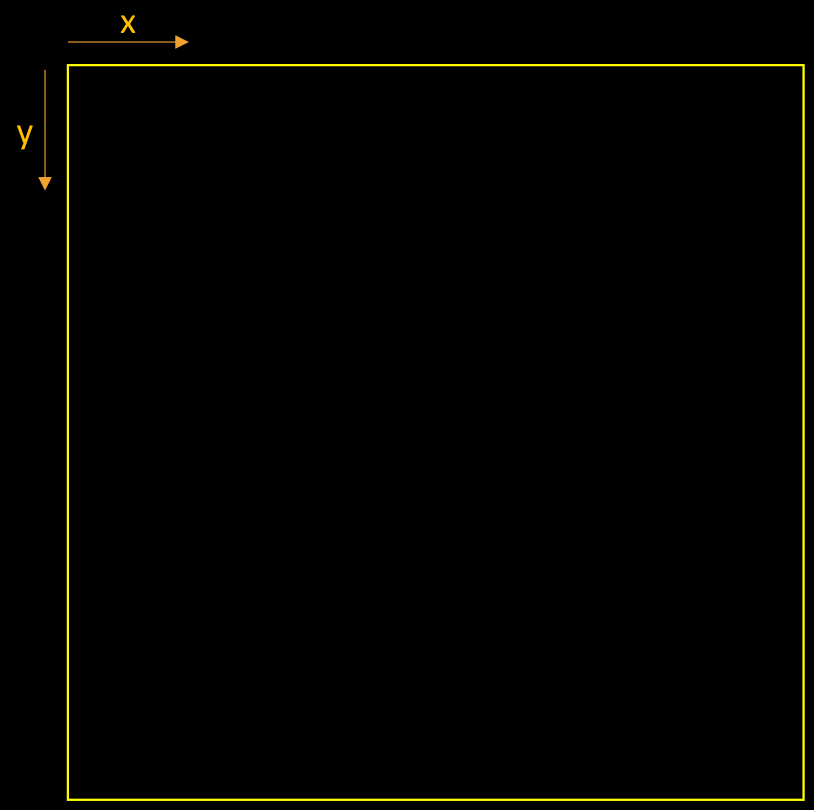
Our first EasyPAP kernel

```
for (unsigned it = 1; it <= nb_iter; it++) {

  for (int i = 0; i < DIM; i++)
    for (int j = 0; j < DIM; j++)
      cur_img (i, j) = compute_color (i, j);

  rotate (); // Slightly increase base angle
}
```



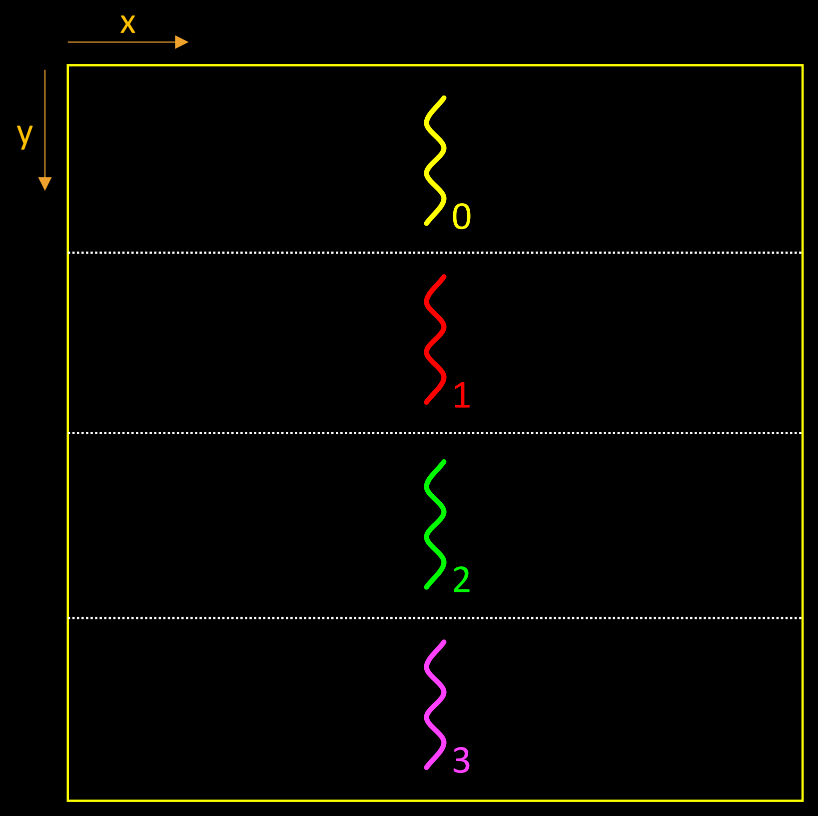
A first, straightforward OpenMP variant

```
for (unsigned it = 1; it <= nb_iter; it++) {
#pragma omp parallel for schedule(static)
  for (int i = 0; i < DIM; i++)
    for (int j = 0; j < DIM; j++)
      cur_img (i, j) = compute_color (i, j);

  rotate (); // Slightly increase base angle
}
```



A first, straightforward OpenMP variant

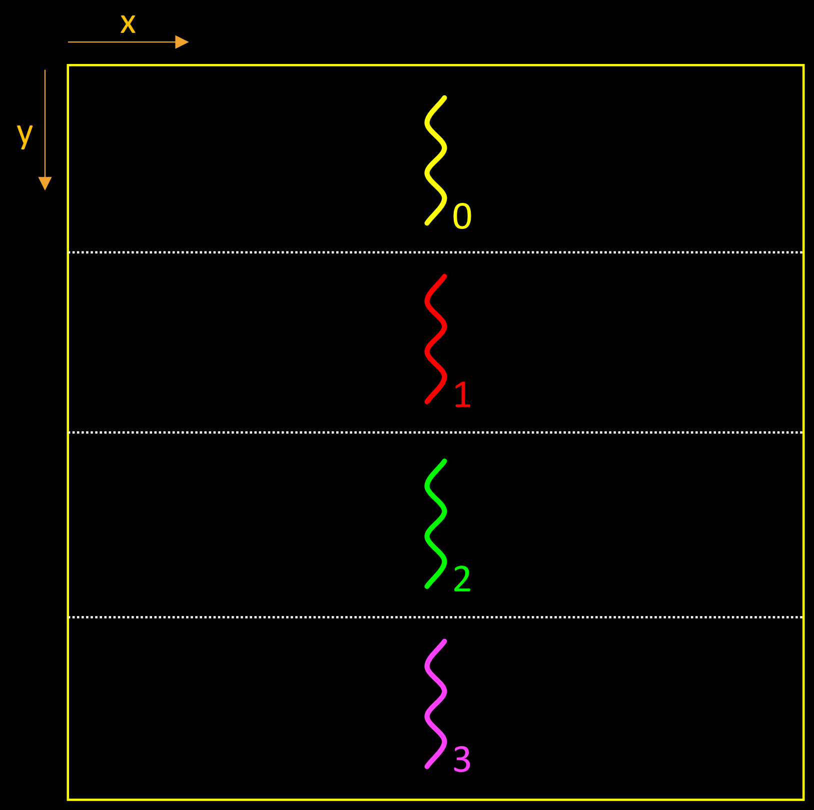
```
for (unsigned it = 1; it <= nb_iter; it++) {
#pragma omp parallel for schedule(static)
  for (int i = 0; i < DIM; i++)
    for (int j = 0; j < DIM; j++)
      cur_img (i, j) = compute_color (i, j);

  rotate (); // Slightly increase base angle
}
```



A first, straightforward OpenMP variant

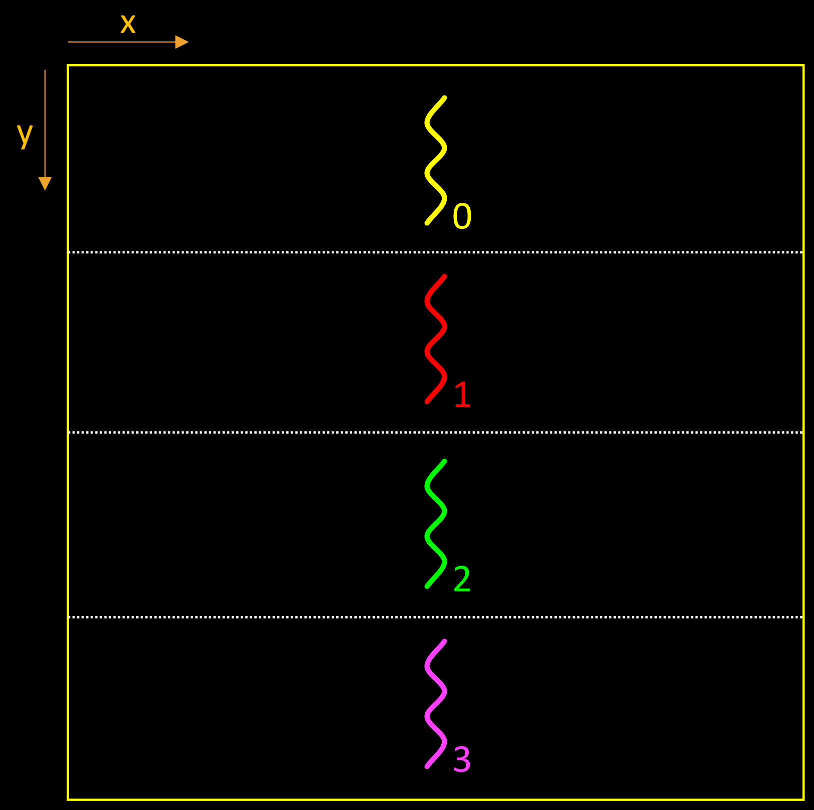
```
for (unsigned it = 1; it <= nb_iter; it++) {
#pragma omp parallel for schedule(static)
  for (int i = 0; i < DIM; i++)
    for (int j = 0; j < DIM; j++)
      cur_img (i, j) = compute_color (i, j);

  rotate (); // Slightly increase base angle
}
```

Note: we fork/join threads at every iteration…



A first, straightforward OpenMP variant

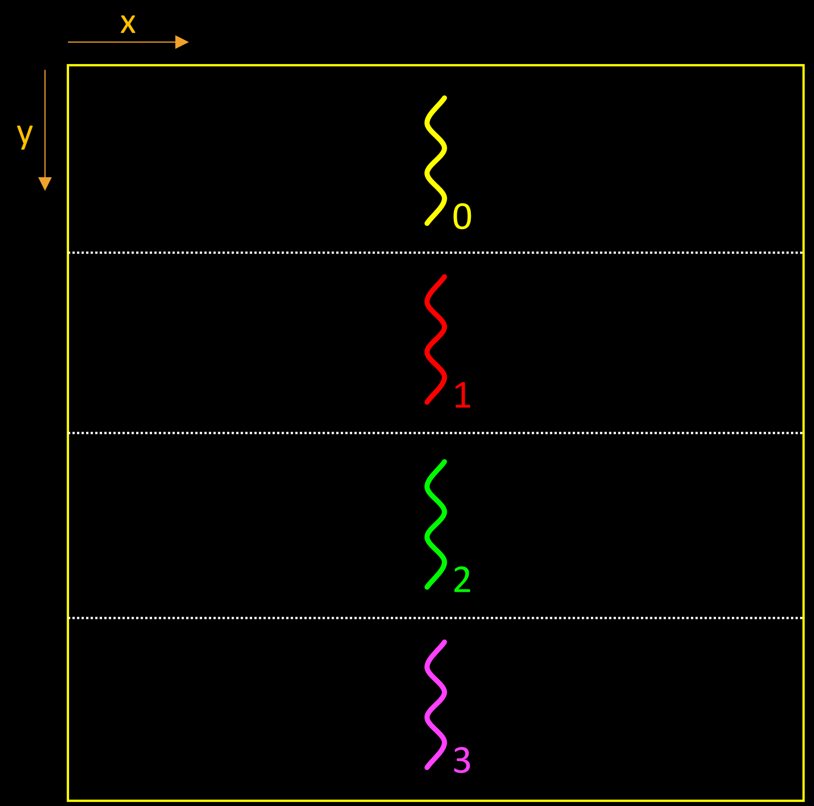
```
#pragma omp parallel
for (unsigned it = 1; it <= nb_iter; it++) {
#pragma omp for schedule(static)
  for (int i = 0; i < DIM; i++)
    for (int j = 0; j < DIM; j++)
      cur_img (i, j) = compute_color (i, j);

  rotate (); // Slightly increase base angle
}
```



A first, straightforward OpenMP variant

```
#pragma omp parallel
for (unsigned it = 1; it <= nb_iter; it++) {
#pragma omp for schedule(static)
  for (int i = 0; i < DIM; i++)
    for (int j = 0; j < DIM; j++)
      cur_img (i, j) = compute_color (i, j);
#pragma omp single
  rotate (); // Slightly increase base angle
}
```

Only one thread should perform this call

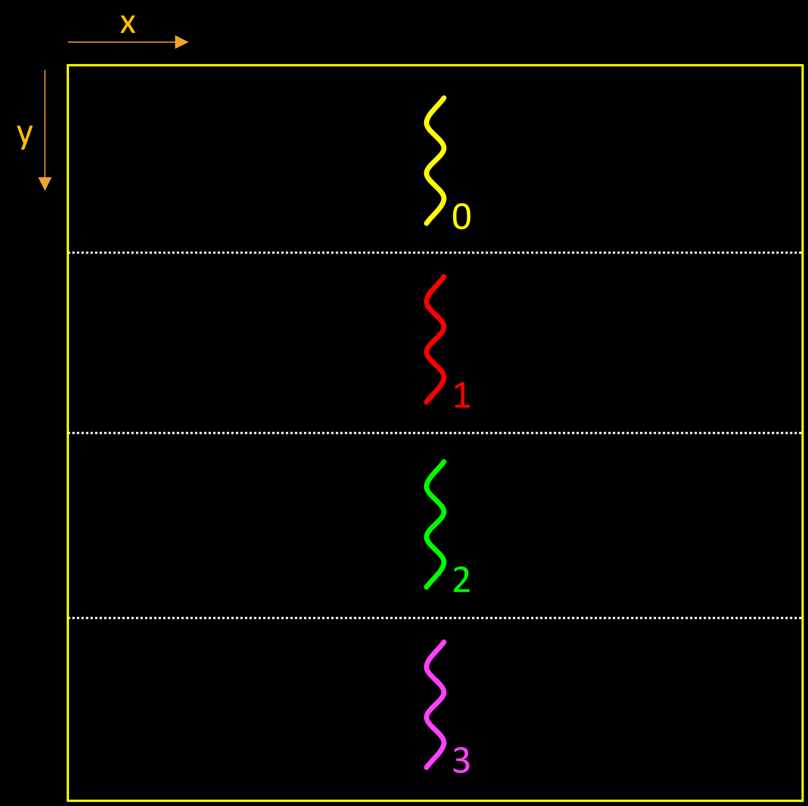


See `single.c`

A first, straightforward OpenMP variant

```
#pragma omp parallel
for (unsigned it = 1; it <= nb_iter; it++) {
#pragma omp for schedule(static)
  for (int i = 0; i < DIM; i++)
    for (int j = 0; j < DIM; j++)
      cur_img (i, j) = compute_color (i, j);
#pragma omp single
  rotate (); // Slightly increase base angle
}
```

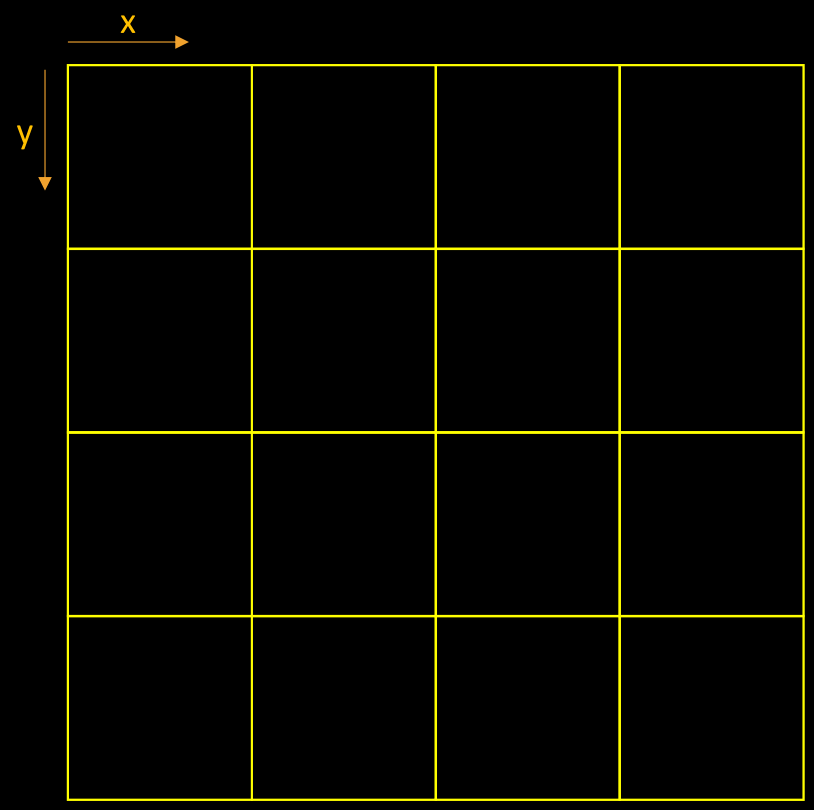
Implicit barriers



Tiling

- Tiled computation scheme

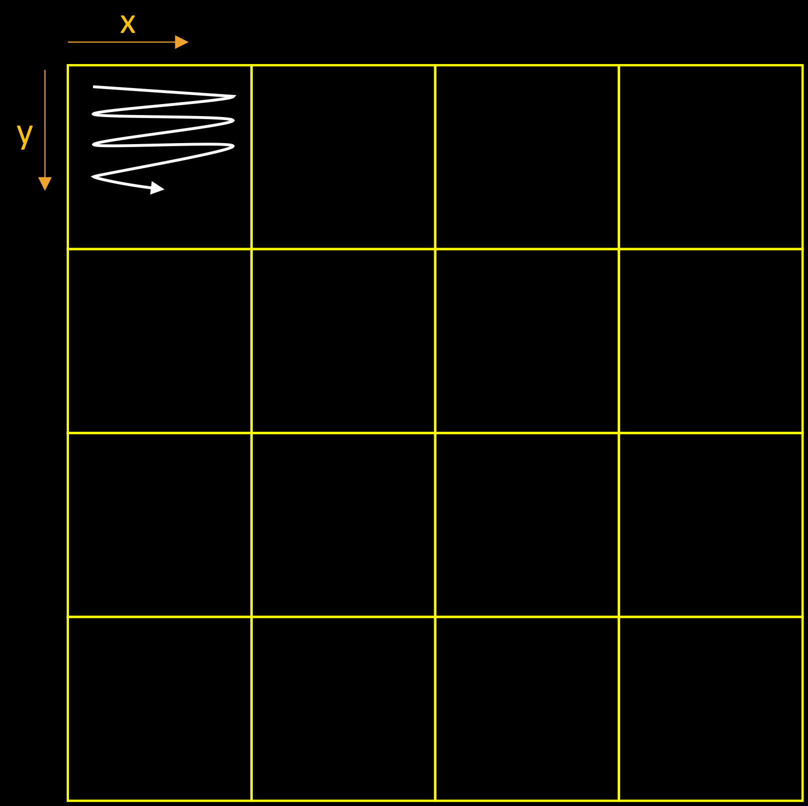
```
void do_tile (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      cur_img (i, j) = compute_color (i, j);
}

for (int y = 0; y < DIM; y += TILE_SIZE)
  for (int x = 0; x < DIM; x += TILE_SIZE)
    do_tile (x, y, TILE_SIZE, TILE_SIZE);
```



Tiling

- Tiled computation scheme

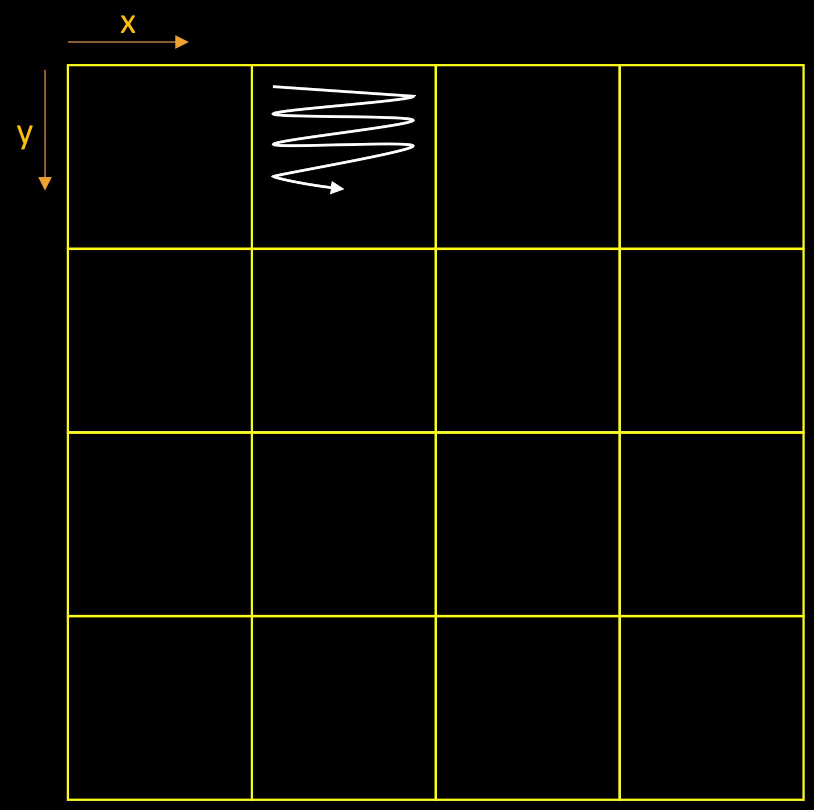
```
void do_tile (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      cur_img (i, j) = compute_color (i, j);
}

for (int y = 0; y < DIM; y += TILE_SIZE)
  for (int x = 0; x < DIM; x += TILE_SIZE)
    do_tile (x, y, TILE_SIZE, TILE_SIZE);
```



Tiling

- Tiled computation scheme

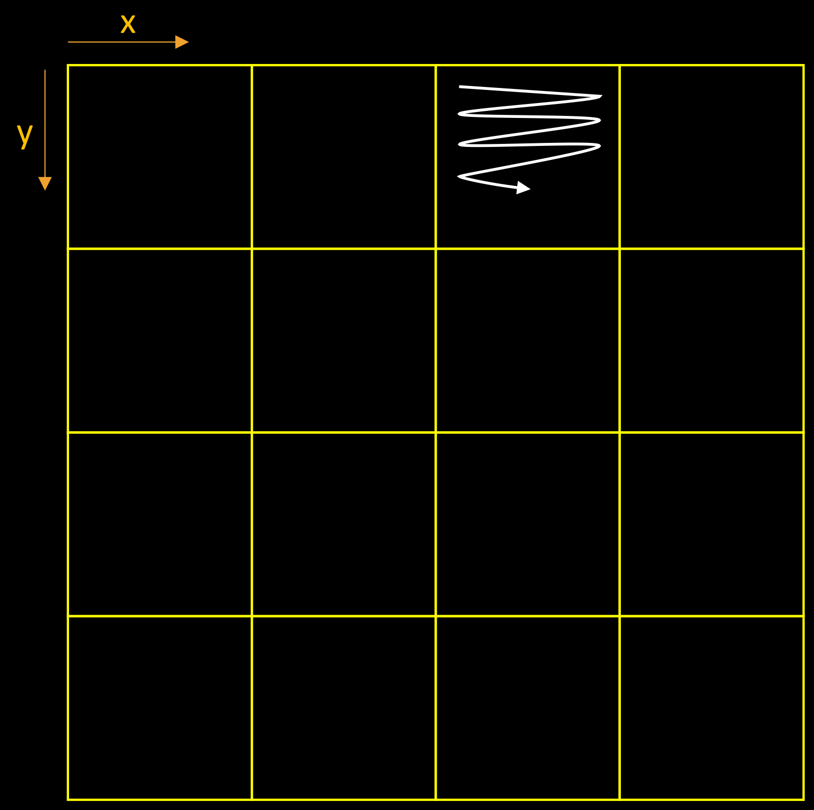
```
void do_tile (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      cur_img (i, j) = compute_color (i, j);
}

for (int y = 0; y < DIM; y += TILE_SIZE)
  for (int x = 0; x < DIM; x += TILE_SIZE)
    do_tile (x, y, TILE_SIZE, TILE_SIZE);
```



Tiling

- Tiled computation scheme

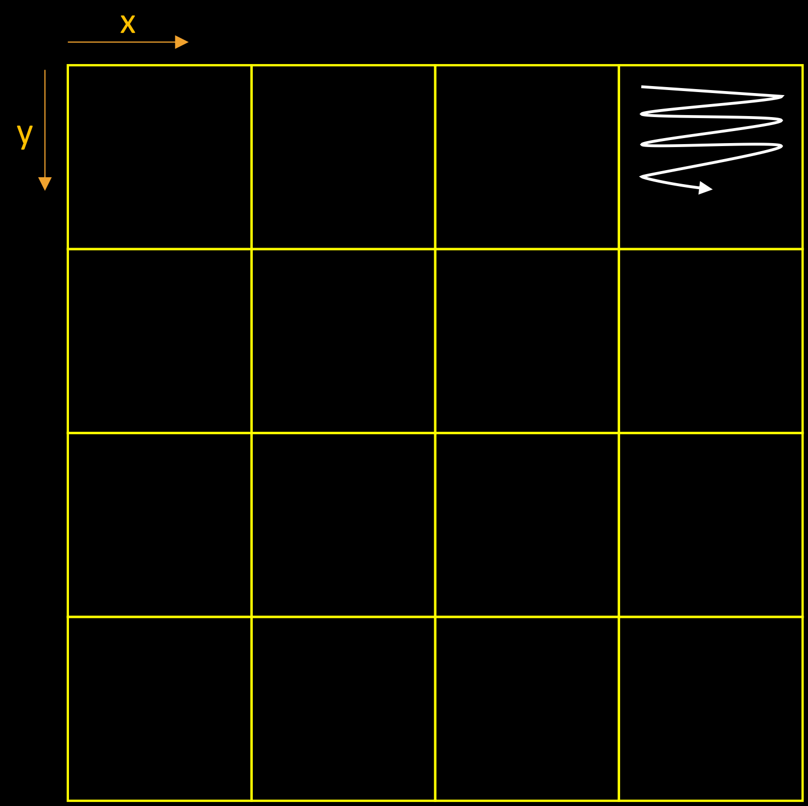
```
void do_tile (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      cur_img (i, j) = compute_color (i, j);
}

for (int y = 0; y < DIM; y += TILE_SIZE)
  for (int x = 0; x < DIM; x += TILE_SIZE)
    do_tile (x, y, TILE_SIZE, TILE_SIZE);
```



Tiling

- Tiled computation scheme

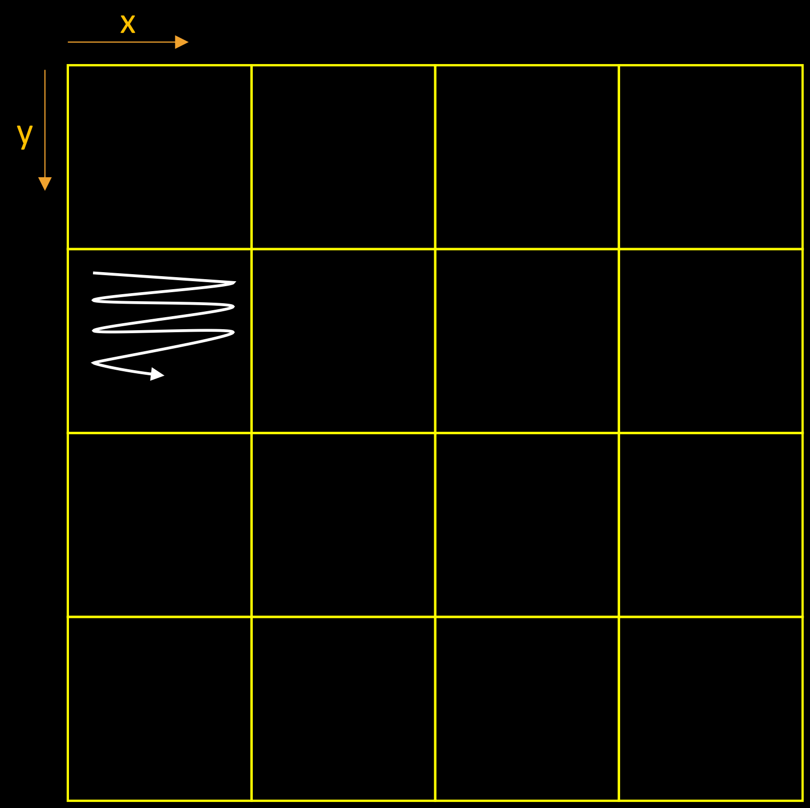
```
void do_tile (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      cur_img (i, j) = compute_color (i, j);
}

for (int y = 0; y < DIM; y += TILE_SIZE)
  for (int x = 0; x < DIM; x += TILE_SIZE)
    do_tile (x, y, TILE_SIZE, TILE_SIZE);
```



Tiling

- Tiled computation scheme

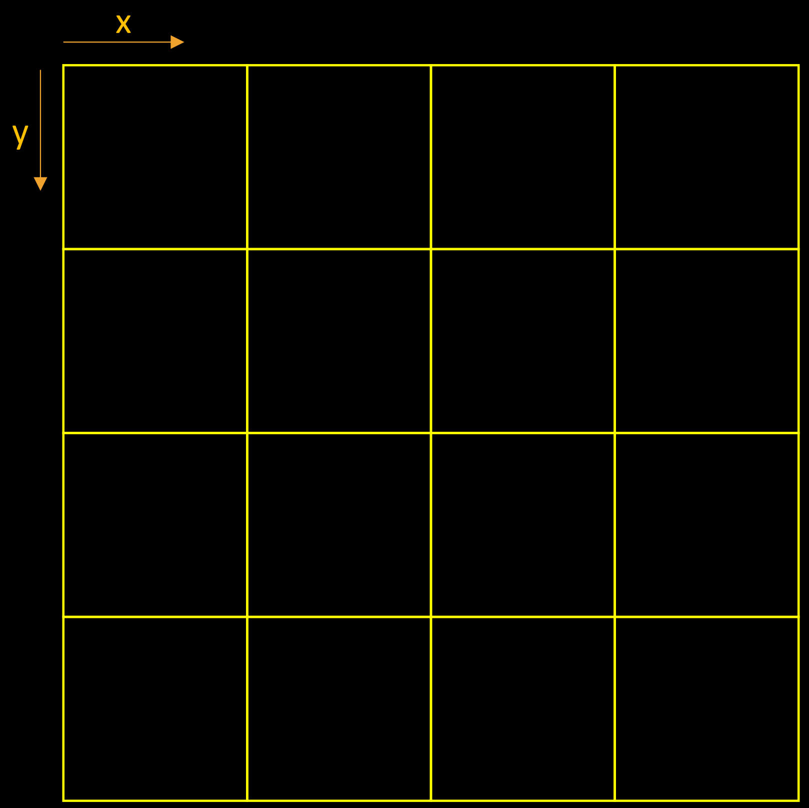
```
void do_tile (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      cur_img (i, j) = compute_color (i, j);
}

for (int y = 0; y < DIM; y += TILE_SIZE)
  for (int x = 0; x < DIM; x += TILE_SIZE)
    do_tile (x, y, TILE_SIZE, TILE_SIZE);
```



Tiling

- Tiled computation scheme

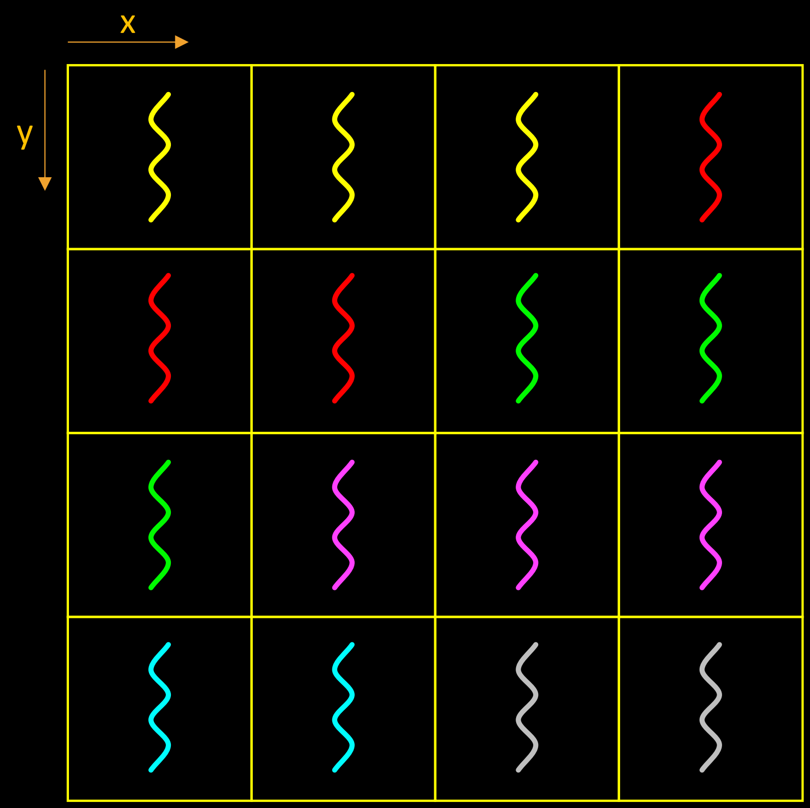
```
void do_tile (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      cur_img (i, j) = compute_color (i, j);
}
#pragma omp parallel for collapse (2)
for (int y = 0; y < DIM; y += TILE_SIZE)
  for (int x = 0; x < DIM; x += TILE_SIZE)
    do_tile (x, y, TILE_SIZE, TILE_SIZE);
```



Tiling

- Tiled computation scheme

```
void do_tile (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      cur_img (i, j) = compute_color (i, j);
}
#pragma omp parallel for collapse (2) schedule(static)
for (int y = 0; y < DIM; y += TILE_SIZE)
  for (int x = 0; x < DIM; x += TILE_SIZE)
    do_tile (x, y, TILE_SIZE, TILE_SIZE);
```



Tile distribution with OMP_NUM_THREADS=6

Data sharing attributes

- Regarding variables used in an OpenMP construct… but declared outside
  - OpenMP defines a number of default rules
    - E.g. *"in a parallel construct, variables are shared"*

```
{
  int k = 5;

#pragma omp parallel
  {
    k = omp_get_thread_num ();
    printf ("Thread %d sees k == %d!\n",
            omp_get_thread_num (), k);
  }

}
```

See `shared.c`

Data sharing attributes

- Several attributes allow to explicitly specify data sharing rules
  - Shared
  - Private
  - Firstprivate
    - ≈ private + copy initial value

```
{
  int k = 5;

#pragma omp parallel private (k)
  {
    k = omp_get_thread_num ();
    printf ("Thread %d sees k == %d!\n",
            omp_get_thread_num (), k);
  }

}
```

See `private.c`

Data sharing attributes

- Several attributes allow to explicitly specify data sharing rules
  - Shared
  - Private
  - Firstprivate
    - ≈ private + copy initial value

```
{
  int k = 5;

#pragma omp parallel firstprivate (k)
  {
    while (k--)
      printf ("Thread %d sees k == %d!\n",
              omp_get_thread_num (), k);
  }

}
```

See `loop.c`

Concurrent data accesses

- Surprising outputs?

```
[my-machine] OMP_NUM_THREADS=12 ./inc
```

```
{
  int k = 5;

#pragma omp parallel shared (k)
  {
    k++;
    printf ("Thread %d sees k == %d!\n",
            omp_get_thread_num (), k);
  }

}
```

See `inc.c`

Concurrent data accesses

- Surprising outputs?

```
[my-machine] OMP_NUM_THREADS=12 ./inc
Thread 1 sees k == 8!
Thread 5 sees k == 10!
Thread 3 sees k == 10!
Thread 2 sees k == 10!
Thread 7 sees k == 9!
Thread 6 sees k == 11!
Thread 9 sees k == 12!
Thread 8 sees k == 13!
Thread 0 sees k == 14!
Thread 4 sees k == 15!
Thread 11 sees k == 16!
Thread 10 sees k == 17!
```

```
{
  int k = 5;

#pragma omp parallel shared (k)
  {
    k++;
    printf ("Thread %d sees k == %d!\n",
            omp_get_thread_num (), k);
  }

}
```

See inc.c

Concurrent data accesses

- Surprising outputs?

```
[my-machine] OMP_NUM_THREADS=12 ./inc
Thread 1 sees k == 7!
Thread 3 sees k == 9!
Thread 5 sees k == 7!
Thread 6 sees k == 10!
Thread 9 sees k == 11!
Thread 2 sees k == 12!
Thread 10 sees k == 13!
Thread 7 sees k == 14!
Thread 8 sees k == 10!
Thread 4 sees k == 8!
Thread 0 sees k == 11!
Thread 11 sees k == 12!
```

```c
{
  int k = 5;

#pragma omp parallel shared (k)
  {
    k++;
    printf ("Thread %d sees k == %d!\n",
            omp_get_thread_num (), k);
  }

}
```

See `inc.c`

Concurrent data accesses

- Surprising outputs?

```
[my-machine] OMP_NUM_THREADS=12 ./inc
Thread 1 sees k == 7!
Thread 3 sees k == 9!
Thread 5 sees k == 7!
Thread 6 sees k == 10!
Thread 9 sees k == 11!
Thread 2 sees k == 12!
Thread 10 sees k == 13!
Thread 7 sees k == 14!
Thread 8 sees k == 10!
Thread 4 sees k == 8!
Thread 0 sees k == 11!
Thread 11 sees k == 12!
```

Ouch! Max value is 14!

```
{
  int k = 5;

#pragma omp parallel shared (k)
  {
    k++;
    printf ("Thread %d sees k == %d!\n",
            omp_get_thread_num (), k);
  }

}
```

See `inc.c`

Race conditions

```
volatile int n = 0;
```

```
for (int i = 0; i < 100; i++)
  n++;
```

```
for (int i = 0; i < 100; i++)
  n++;
```

join

```
printf ("n = %d\n", n);
```

$$n = 200\ ?$$

Race conditions

```c
volatile int n = 0;
```

```c
for (int i = 0; i < 100; i++)
  n++;
```

```c
for (int i = 0; i < 100; i++)
  n++;
```

join

```c
printf ("n = %d\n", n);
```

$n \in [100,200]$?

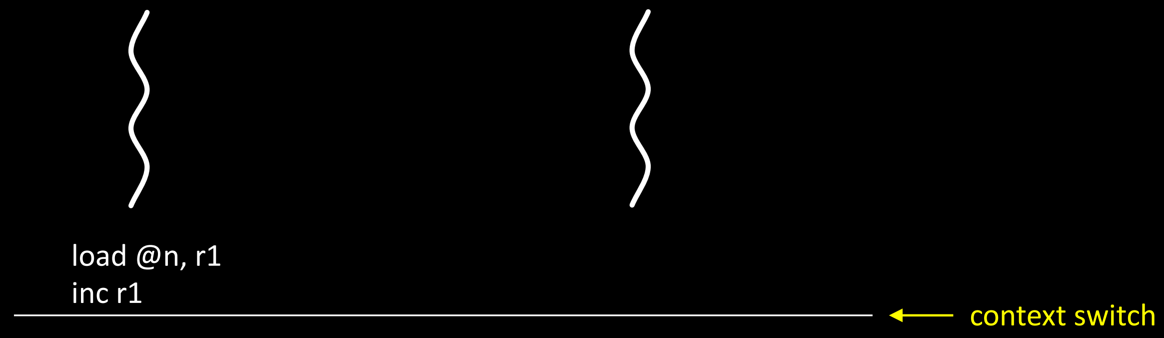
Possible scenario

```
          load @n, r1      ; load from memory
n++ ⇔     inc r1           ; increment register
          store r1, @n     ; store in memory
```

n : 0

Possible scenario

n++ ⇔ load @n, r1 ; load from memory
inc r1 ; increment register
store r1, @n ; store in memory



n : 0

Possible scenario

```
         load @n, r1      ; load from memory
n++ ⇔    inc r1           ; increment register
         store r1, @n     ; store in memory
```

n : ~~0~~ 99

```
load @n, r1
inc r1
---------------------------------------
                      load @n, r1
                      inc r1
                      store r1, @n      } 99x
                      ...
---------------------------------------
```

Possible scenario

n++ ⇔ load @n, r1 ; load from memory
inc r1 ; increment register
store r1, @n ; store in memory

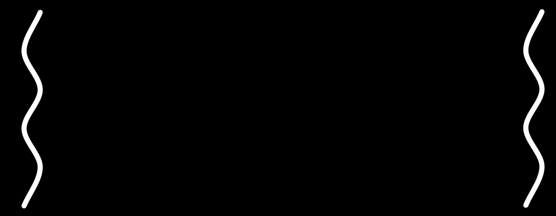
```
load @n, r1   ; load from memory
inc r1        ; increment register
store r1, @n  ; store in memory
```

n : ~~0~~ ~~99~~ 1

Thread 1	Thread 2	
load @n, r1 inc r1		
	load @n, r1 inc r1 store r1, @n …	99x
store r1, @n		

Possible scenario

n++ ⇔
```
load @n, r1       ; load from memory
inc r1            ; increment register
store r1, @n      ; store in memory
```



load @n, r1
inc r1

---

load @n, r1
inc r1
store r1, @n
…
} 99x

---

store r1, @n

---

load @n, r1
inc r1

---

n : ~~0 99~~ 1

Possible scenario

n++ ⇔

```
load @n, r1      ; load from memory
inc r1           ; increment register
store r1, @n     ; store in memory
```

n : ~~0 99 1~~ 100

Thread 1	Thread 2	
load @n, r1 inc r1		
	load @n, r1 inc r1 store r1, @n …	99x
store r1, @n		
	load @n, r1 inc r1	
99x: load @n, r1 inc r1 store r1, @n …		

Possible scenario

n++ ⇔ load @n, r1 ; load from memory
inc r1 ; increment register
store r1, @n ; store in memory

n : ~~0 99 1 100~~ 2

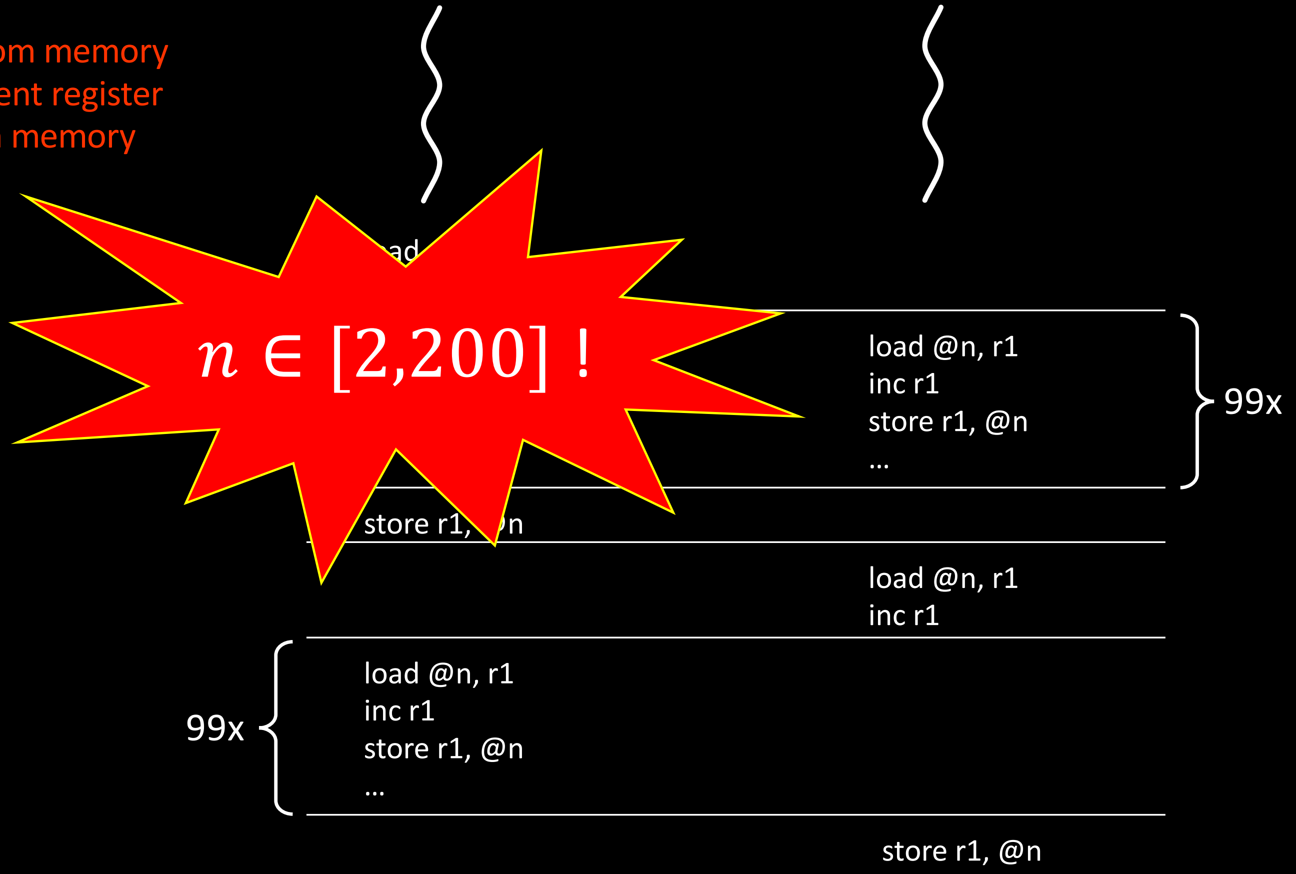
Thread 1	Thread 2	
load @n, r1 inc r1		
	load @n, r1 inc r1 store r1, @n …	99x
store r1, @n		
	load @n, r1 inc r1	
99x: load @n, r1 inc r1 store r1, @n …		
	store r1, @n	

Possible scenario

n++ ⇔

```
load @n, r1      ; load from memory
inc r1           ; increment register
store r1, @n     ; store in memory
```

n : ~~0 99 1 100~~ 2



Race conditions

- Even the simple ++ operator is not an *atomic* operation
  - So we must prevent multiple threads to execute this operation concurrently!

- To do so, we need synchronization tools

```
#pragma omp critical
{
  n++; // critical section
}
```

Critical vs Atomic sections

- The implementation of `critical` uses OS locks underneath
  - E.g. mutexes
  - Quite expensive to protect a simple `n++` operation!

- OpenMP can use atomic hardware instructions (instead of software locks) in a few cases
  - ++, --, *=, …

```
#pragma omp atomic
  n++;
```

Critical vs Atomic sections

- More complex operations are possible

```
#pragma omp atomic capture
  v = n++;
```

- Atomic instructions incur some overhead compared to their regular variant
  - Hardware lock of cache lines
    - There’s no such thing as a free lunch ☹

- Use atomic whenever possible, fallback to critical otherwise

Additional resources available on http://gforgeron.gitlab.io/it224/