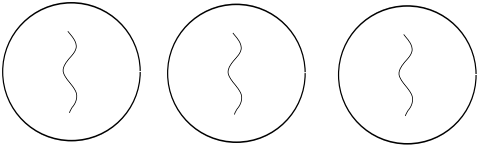
Introduction to OpenMP

November 14, 2023

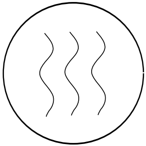
Prologue: Thread VS Process

Process : "control flow" $+$ "memory space"
Thread : "control flow"

Specific to each process	Specific to each thread
Memory (Heap)	Instruction pointer
Global variables	Registers
Open Files	Stack (local variables)
child process, signals...	CPU state



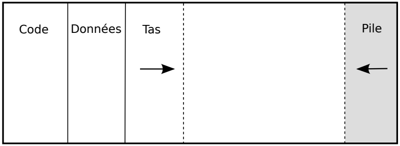
Multi-processes



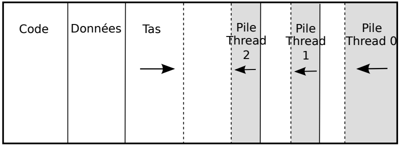
multi-threads

Prologue: Thread VS Process (continued)

Single-thread process:



multi-thread process:



All threads have access to the process' memory space

- **Shared** variables (between several threads)
  - *global* variables ("data segment")
  - Dynamically allocated variables with shared pointer
- **private** variables ("owned" by a thread)
  - Local variables (on the stack)
  - Dynamically allocated variables with private pointer

OpenMP

- Easier to use than MPI
- Preserve the original sequential code
- Code is easier to understand and maintain
- Allows progressive parallelization
- Shared memory machines only (e.g. ONE server)
- Works better on specific code patterns (loop nests, ...)

MPI

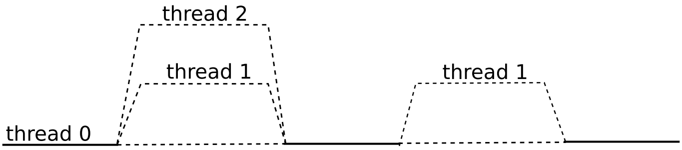
- Designed for **distributed memory** machines
- Also works fine on shared-memory machines
- Separated memory spaces (no conflicts)
- Algorithmic modifications often necessary
- Harder to use
- Performance depends on the network

OpenMP: timeline

- 1997, a consortium of industrials and academics adopt OpenMP (*Open Multi Processing*) as a **standard**. Fortran, C and C++ interface
- Version 2.5 (2000, `gcc 4.2`) : *lean and mean* (for loops)
- Version 3.0 (2008, `gcc 4.4`) : new *task* concept
- Version 3.1 (2008, `gcc 4.7`) : better tasks, `atomic`
- Version 4.0 (2013, `gcc 4.9`) : SIMD and *devices* (GPU, ...)
- Version 4.5 (2013, `gcc 6`) : better SIMD, more GPU, ...
- Version 5.1 (2020, `gcc 10`) : `atomic compare` + ...
  - v5.1 of the spec is much harder to read than v4.5
- Version 5.2 (2021) : clean up the spec...
- Version 6.0 (2024) : ???

Principle

- A single process runs on a machine.
- The corresponding thread is the "master thread" (number 0)
- It occasionally spawns other threads to run parallel computation, then waits for them to complete (*fork and join* model)



- Declaring parallel sections in the code is done using **OpenMP directives**
- Specific **memory model** (shared / private variables)

Using OpenMP

- **Compile-time** directives (`#pragma` in C)
  - Interpreted by OpenMP-aware compilers
  - Silently ignored otherwise
  - Tell the compiler to generate parallel code
  - `gcc`: `-fopenmp` option
- Must **link** against OpenMP run-time library
  - `gcc`: `-fopenmp` option
- At **run-time**: environment variables allow some control over OpenMP

OpenMP in One Slide

```c
// sequential prologue
#pragma omp parallel for
for (int i = 0 ; i < n ; i++) {
  /*
   * All the iterations of this loop can
   * be executed in parallel
   */
}
// sequential epilogue
```

```
gcc -fopenmp prog_omp.c -o prog_omp
```

- "normal" sequential program until `#pragma omp`
- A **team of threads** is created
- The iterations of the loop are **distributed** between them
- **Barrier** at the end of the loop
- "normal" sequential program then

Conditional Compilation and Run-Time OpenMP Functions

Conditional compilation

```
#ifdef _OPENMP
    // Code included only if the compiler supports OpenMP
    // With gcc, only if the -fopenmp option has been given
#endif
```

OpenMP run-time functions

With `#include <omp.h>`

- Enable a SPMD style of programming (as in MPI)
- `omp_get_num_threads()`
- `omp_get_thread_num()`
- `omp_set_num_threads()`
- ...

Hello world

Program

```c
#ifdef _OPENMP
#include <omp.h>
#endif

int main()
{
  #pragma omp parallel
  {
    #ifdef _OPENMP
    printf("Hello world, thread %d/%d\n",
          omp_get_thread_num(),
          omp_get_num_threads());
    #else
    printf("Hello world\n");
    #endif
  }
}
```

Execution

```
$ gcc hello.c -o hello
$ ./hello
Hello world
$ gcc -fopenmp hello.c \
 -o hello
$ export OMP_NUM_THREADS=4
$ ./hello
Hello world, thread 0/4
Hello world, thread 3/4
Hello world, thread 1/4
Hello world, thread 2/4
```

OpenMP Directives

`#pragma omp` *directive [clause[[,]clause]...]*

Barrier (synchronization) at the end by default

Using directives

- Jumping out of a parallel section is forbidden (~~goto~~, ~~setjmp/longjmp~~, ...)
- One directive per `#pragma omp`
- Case-sensitive
- Directives ⊆ { `parallel`, `for`, `sections`, `section`, `single`, `master`, `critical`, `barrier`, `atomic`, `flush`, `ordered`, `threadprivate`, `...` }

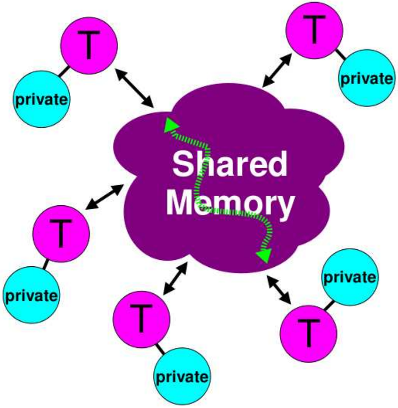
OpenMP Memory Model

Reminder

Variable : identifier denoting a memory address

Variables present in the original sequential code can be declared as *shared* or *private* with OpenMP.

- **Shared**: all threads access the memory address of the original variable
- **Private**: each thread "owns" a copy of the original variable (all located at different memory addresses)



(after *An Overview of OpenMP 3.0*, R. van der Pas)

OpenMP Memory Model (continued)

- Variables declared **before** a parallel region are **shared by default**
- Their status can be modified using clauses in OpenMP directives
  - `private`, `shared`, `firstprivate`, `lastprivate`, `default(shared)`, `default(none)`, `reduction`, `copyin`
- Local variables of each thread are private (cf. infra)

`parallel` Directive

```
#pragma omp parallel [clause[[, ]clause]...]
structured block
```

- Assemble a thread team (creation/recycling)
- **All** threads run the *structured block*.

Associated clauses

- `if(cond)`: `cond == False` → no threads
  - E.g., don't use all of the machine for small problem
- `private` (*var_list*), `firstprivate` (*var_list*)
- `reduction` (cf. infra)
- `num_threads`(*int*): force the size of the thread team

```c
int main()
{
    ...
    initialization();
    #pragma omp parallel ...
    {
        parallel_computation();
    }
    post_processing();
}
```

How to choose the number of threads

By decreasing order of priority:

Compile-time	: #pragma omp parallel num_threads(16)
Run-time	: *omp_set_num_threads(4)*
Environment variable	: export OMP_NUM_THREADS=4

Predefined Variables are Shared by Default

Program

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

int main()
{
  int c=0;

  #pragma omp parallel
  {
    c++;
    printf("c=%d thread %d\n",
        c, omp_get_thread_num());
  }
}
```

Execution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
c=1 thread 3
c=2 thread 0
c=3 thread 1
c=4 thread 2
```

Beware of Conflicts!

Program

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

int main()
{
  int c = 0;

  #pragma omp parallel
  {
    for (int i=0; i<100000; i++)
        c++;
    printf("c=%d thread %d\n",
        c, omp_get_thread_num());
  }
}
```

Execution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
c=100000 thread 0
c=200000 thread 3
c=270620 thread 2
c=286162 thread 1
```

`private` Clause

`private` variable:

- Each thread owns a (private) local copy
- Not initialized

BUG :

Program

```c
int main()
{
  int a = 100;
  #pragma omp parallel private(a)
  {
   /* This "a" is not the same
      as before */
    a = a + 10;
    printf("a=%d\n", a);
  }
  printf("After a=%d\n", a);
}
```

Execution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
a=-1208433038
a=-22
a=-22
a=-22
After a=100
```

firstprivate Clause

firstprivate Variables;

- Each thread owns a (private) local copy
- Initialized with the preexisting value

Program

```
int main()
{
int a = 100;
#pragma omp parallel \
                firstprivate(a)
{
  a = a + 10;          // idem...
  printf("a=%d\n", a);
}
printf("After a=%d\n", a);
}
```

Execution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
a=110
a=110
a=110
a=110
After a=100
```

Local Variables

- All local variables in functions called from a parallel section are owned by the corresponding thread (on their stacks)
- Same goes for local variables declared inside the *structured block*

Program

```c
void func()
{
  int a = 10;
  a += omp_get_thread_num();
  printf("a=%d\n", a);
}

int main()
{
  #pragma omp parallel
  func();
}
```

Execution

```
$ export OMP_NUM_THREADS=4
$ ./test2
10
11
12
13
```

Local Variables: My Opinion

Complex (avoid)

```
int main()
{
  int a;
  #pragma omp parallel private(a)
  {
    a = ...;
    ...
  }
}

/**********************************/

int main()
{
  int a;
  #pragma omp parallel firstprivate(a)
  {

    ... a ...
  }
}
```

Simple (better)

```
int main()
{

  #pragma omp parallel
  {
    int a = ...;
    ...
  }
}

/**********************/

int main()
{
  int a;
  #pragma omp parallel
  {
    int b = a;
    ... b ...
  }
}
```

Reminders / Details

`#pragma omp parallel`

- A **thread team** is created
- The **encountering thread** belongs to it (**master**)
- All threads of the team execute the *structured block*
- Identifiers (“ranks”) 0 (master), 1, 2, ..., #threads - 1
  - `omp_get_num_threads(), omp_get_thread_num()`
- **Barrier** at the end
- Encountering thread then resumes sequential execution

`for` Directive

```
#pragma omp for [clause[[, ]clause]...]
⟨ for loop ⟩
```

Worksharing directive

Threads of the team cooperate and **divide** the work between them

Associated clauses :

- `private` (*variable_list*), `firstprivate` (*variable_list*), `lastprivate` (*variable_list*)
- `reduction`(*operator: variable_list*)
- `ordered`
- `collapse`(*n*)
- `schedule`(*type*, *size*)
- `nowait`

`for` Directive (continued)

Canonical loop form

`for` (*init_expr* ; *cond* ; *increment*)

- Integer iteration variable
- Loop counter updated with `++`, `--`, `+=`, `-=`, `var=var+inc`, `var=inc+var`, `var=var-inc`
- integer increment
- Condition: `<`, `>`, `<=`, `>=`. Bound is a fixed expression
- No early exit (`break`, `return`, `exit`)
- `continue` is allowed

Consequences of the `for` directive:

- Implicit barrier at the end of the loop (except if `nowait`)
- No barrier at the beginning
- **The iteration variable is private**

Example

```
int main()
{
  int t[100];
  #pragma omp parallel
  {
    #pragma omp for
    for (int i = 0; i < 100; i++)
      t[i] = i;
  }
}
```

With 4 threads, the first one may **for instance** compute the $t[i]$ from 0 to 24, the second from 25 to 49, ...

Short Form of the `for` directive

`#pragma omp parallel for` *[clause[[,]clause]...]*
for loop

Admits all clauses of `parallel` and `for`, except `nowait`.

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

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

The reduction Clause

Program

```c
int main()
{
  int a[4][4], s=0;
  for (int i = 0; i < 4; i++)
      for (int j = 0; j < 4; j++)
        a[i][j] = i * 4 + j;
  #pragma omp parallel for reduction(+:s)
  for (int i = 0 ; i < 4 ; i++) {
      for (int j = 0; j < 4; j++)
          s += a[i][j];
      printf("PAR=%d : i=%3d  s=%d\n",
           omp_get_thread_num(), i, s);
     }
  }
  printf("SEQ s=%d\n", s);
}
```

Execution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
PAR=0 : i=  0  s=6
PAR=1 : i=  1  s=22
PAR=2 : i=  2  s=38
PAR=3 : i=  3  s=54
SEQ s=120
```

The `reduction` Clause (continued)

- Operators : `+`, `-`, `*` , `&`, `|`, `^`, `&&`, `||`, `min`, `max`
- Can define your own

Authorized on `omp for`, `omp parallel`, ...

```
int main()
{
  int m = 0;
  #pragma omp parallel reduction(max:m)
  {
    int tid = omg_get_thread_num();
    m = f(tid);
  }
  ...
}
```

New Features in `reduction`

- `reduction` on *arrays*
- ... or array *slices*
- Starting with OpenMP 4.5
  - November 2015, gcc ≥ 6.1

```c
double *A = malloc(n * sizeof(*A));
...
#pragma omp parallel reduction(+:A[0:n])
{
    // Each thread own its own copy of A[0:n]
    for (int i = 0; i < n; i++) {
        A[i]= ....
    }
    ....
}
// A[i] contains the sum of the
// private A[i]'s of all threads
```

Load Balancing in Loops

`omp for` admits load-balancing clauses: `schedule` and `nowait`

- `nowait` clause:
  Removes the automatic barrier at the end of the loop
- `schedule(mode, chunk_size)` clause:
  4 modes: `static`, `dynamic`, `guided`, `runtime`

Implementation-dependent if not specified (static on gcc)

schedule Example

```c
#define MAX 10
int main()
{
  int a[MAX];
  #pragma omp parallel
  {
      int imax = 0;
      int imin = MAX;
      #pragma omp for schedule(static)
      for (int i = 0; i < MAX; i++) {
          imin = (i < imin) ? i : imin;
          imax = (i > imax) ? i : imax;
          a[i] = 1;
          sleep(1); /* simulate computation */
          printf("%3d:%3d\n",omp_get_thread_num(), i);
      }
      printf("T%d imin=%d imax=%d\n", omp_get_thread_num(), imin, imax);
  }
}
```

`static` and `dynamic` Clauses

`schedule(...)`

static

```
  1:  3
  3:  8
  2:  6
  0:  0
  1:  4
  3:  9
  2:  7
  0:  1
  1:  5
  0:  2
T1 imin=3 imax=5
T0 imin=0 imax=2
T2 imin=6 imax=7
T3 imin=8 imax=9
```

static, 2

```
  1:  2
  3:  6
  2:  4
  0:  0
  1:  3
  3:  7
  0:  1
  2:  5
  0:  8
  0:  9
T0 imin=0 imax=9
T1 imin=2 imax=3
T3 imin=6 imax=7
T2 imin=4 imax=5
```

dynamic, 2

```
  1:  4
  0:  0
  2:  6
  3:  2
  1:  5
  2:  7
  0:  1
  3:  3
  1:  8
  1:  9
T1 imin=4 imax=9
T3 imin=2 imax=3
T2 imin=6 imax=7
T0 imin=0 imax=1
```

Schedule

- `schedule(static)`: block distribution
- `schedule(static, n)`: block-cyclic distribution.
  - block size `n`
- `schedule(dynamic, n)`: chunks of `n` iterations are affected to available threads ($n = 1$ by default).
- `guided`: like `dynamic` but the chunk size is proportional to the number of remaining iterations
- `auto`: the OpenMP runtime does its magic
- `runtime`: choice is deferred until runtime
  Example : `export OMP_SCHEDULE="static,1"`

`single` Directive

`#pragma omp single` *directive [clause[[,]clause]...]*
structured block

Goal

Sequential portion inside a parallel region, *i.e.* a code chunk executed by a **single** thread

- It can by *any* thread
- Implicit barrier at the end of `single`
- `nowait` and `copyprivate` are incompatible clauses

Possible clauses:

- `private` (*variable_list*), `firstprivate` (*variable_list*), `copyprivate` (*variable_list*)
- `nowait`

single Example: Prefix-Sum

```c
int s = 0;
for (int i = 0; i < n; i++) {
    int tmp = A[i];
    A[i] = s;
    s += tmp;
}
```

```c
#pragma omp parallel
{
    int t = omp_get_thread_num();
    S[t] = 0;
    #pragma omp for schedule(static)
    for (int i = 0; i < n; i++)
        S[t] += A[i];
    #pragma omp single
    {
        int p = omp_get_num_threads();
        int s = 0;
        for (int i = 0; i < p; i++) {
            int tmp = S[i];
            S[i] = s;
            s += tmp;
        }
    }
    int s = S[t];
    #pragma omp for schedule(static)
    for (int i = 0; i < n; i++) {
        int tmp = A[i];
        A[i] = s;
        s += tmp;
    }
}
```

Synchronizations in OpenMP

Several options:

- Barriers
- `atomic` and `critical`
- Locks via OpenMP runtime functions (not covered):
  `omp_init_lock()`
  `omp_{set,test}_lock()`
  `omp_unset_lock()`
  `omp_destroy_lock()`

`barrier` Directive

```
#pragma barrier
```

All threads must *enter* the barrier before any of the threads continue execution beyond the barrier

Problems with C syntax

```c
if (n != 0)
    #pragma omp barrier   // syntactically incorrect
```

```c
if (n != 0) {
    #pragma omp barrier   // OK
}
```

`critical` Directive

```
#pragma omp critical [name]
structured block
```

- Only a single thread may enter the *structured block* simultaneously
- "**critical section**"
- An incoming thread has to wait while another thread executes the *structured block*
- The *name* allows to distinguish distinct critical sections.

critical Example

Adding an Item to a Linked List

```
struct item_t {
  void *data;
  struct item_t *next;
};
struct item_t *list;                    /* global variable */

void append(void *payload)
{
   struct item_t *new_item = malloc(sizeof(*new_item));
   new_item->data = payload;
   #pragma omp critical
   {
      new_item->next = list;
      list = new_item;
   }
}
```

`atomic` Directive

```
#pragma omp atomic [ read | write | update ]
```
atomic-update

- The *atomic-update* is atomic (cannot be interrupted)
- *atomic-update* of the type:
  - `v = x;` or `x = expr;`
  - `x += expr;` or `x = x + expr;`
    - Works with `+, *, -, /, &, &&, ^, |, ||, >>, <<`
    - *expr* must not reference *x*.
- Only reading/writing/updating *x* is atomic.
- Evaluation of *expr* is not

Program

```c
#include <omp.h>

int main()
{
  int c = 0;
  #pragma omp parallel
  {
    for (int i = 0; i < 100000; i++) {
       #pragma omp atomic
       c++;
    }
    printf("c=%d thread %d\n",
      c, omp_get_thread_num());
  }
}
```

Execution

```
$ export OMP_NUM_THREADS=4
$ ./a.out
c=100000 thread 0
c=294308 thread 2
c=394308 thread 3
c=400000 thread 1
```

`atomic capture`

Since OpenMP 3.0 (2008)

```
#pragma omp atomic capture
structured block
```

- Capture = saving the previous value ("capture") + update
- With the `capture` clause, *structured block* can be:

```
{v = x; x = expr;}
{v = x; x += expr;}
{x += expr; v = x;}
{v = x; x = x + expr;}
{v = x; x = expr + x;}
{x = x + expr; v = x;}
{x = expr + x; v = x;}
```

```
{v = x; x++;}
{v = x; ++x;}
{++x; v = x;}
{x++; v = x;}
```

atomic capture Example

Appending an Item to a Preallocated Array

```c
void *A[];
int n = 0;                /* #items in the array */

void atomic_append(void *payload)
{
   int i;
   #pragma omp atomic capture
   {
      i = n;         /* equivalent to: i = n++ */
      n += 1;
   }
   A[i] = payload;
}
```

atomic compare

Since OpenMP 5.1 (needs gcc 12.2, August 2022!)

“Compare-and-Swap”

```
#pragma omp atomic compare
if (x == old) x = new;
```

“Compare-and-Swap” with capture

```
#pragma omp atomic compare capture
{ if (x == old) { x = new; }; else { cap = x; } }
```

“Compare-and-Swap”, most general form

```
#pragma omp atomic compare capture
{ ok = x==old; if (ok) { x = new; } else { cap = x; } }
```

- All CPUs have *hardware* support for this operation

atomic compare Example

Adding an Item to a Linked List

```c
struct item_t {
    void *data;
    struct item_t *next;
}
struct item_t *list;                    /* global variable */

void atomic_append(void *payload)
{
    struct item_t *new_item =  malloc(sizeof(*new_item));
    new_item->data = payload;
    bool ok = 0;
    while (!ok) {
        new_item->next = list;
        #pragma omp atomic compare capture
        {
            ok = (list == new_item->next);
            if (ok)
                list = new_item;
        }
}
```

"Transactions" Using *Compare-And-Swap*

It is possible to do almost anything using *Compare-And-Swap*!

Idiom: *Compare-And-Swap Loop*

1. [Begin.] $x_{old} \leftarrow x$
2. [Work.] Compute an updated x_{new}
   - This may read x...
   - ... and x may be modified by another thread
3. [Commit.] `ok = (x == x_old); if (ok) x = x_new;`
4. [Repeat.] If not OK, go back to 1.

atomic compare Example

Insertion Into a Hash Table With Linear Probing

Hash Table With Linear Probing

```
void insert(void *H, void *item)
{
    int i = hash_function(item);          // hash
    while (H[i] != EMPTY)                 // locate empty slot
        i = (i + 1) % HASHTABLE_SIZE;
    H[i] = item;                          // insert
}
```

Thread-safe version

```
void atomic_insert(void *H, void *item)
{
    int i = hash_function(item);
    book ok = 0;
    while (!ok) {
        #pragma omp atomic compare capture
        { ok = H[i] == EMPTY; if (ok) H[i] = item; }
        i = (i + 1) % HASHTABLE_SIZE;
    }
}
```

Differences Between `atomic` and `critical`

`atomic`:

- Intended to update variables
- Depends on hardware support
  - "Atomic" CPU instructions (compare-and-swap or ll/sc)
  - Implemented with locks as a last resort
- Lighter overhead *a priori*
- To be preferred if possible

`critical`:

- Intended to encompass a larger portion of code
- Implemented using locks
- Heavier overhead

atomic VS critical: Tricky Example

Appending an Item to a Dynamic Array with resizing

```
void *A[];
int n = 0;                          /* #items inserted in A[] */
size = 0;                           /* allocated size of A */

void append(void *payload)
{
   #pragma omp critical
   {
      if (n >= size) {              /* array too small? Enlarge it */
         size = 2 * size + 1;
         A = realloc(A, size);      /* change A */
      }
      A[n] = payload;               /* finally, insert */
      n += 1;
   }
}
```

Without locks / critical?

- Possible, but quite difficult
- Nice challenge! Try it at home!

Lock-Free Algorithms

Definition

A function is **lock-free** if, when it is called by several threads at the same time, at least one of the invocations finishes in a finite number of steps (whatever the others do, even if they block).

$\Rightarrow$ if a function acquires a lock or enters a critical section, it cannot be *lock-free*.

Challenge

Design a *lock-free* data structure to hold a set of N integers. API:

```
void setup(int N);      /* initialise S = {0, 1, ..., N-1} */
bool remove(int i);     /* Remove i from S. Return 1 if i was in S */
int extract_min();      /* let k = min(S) in remove(k) */
```

Golden rule of multi-thread programming

ALL potentially conflicting accesses* to **shared** variables **MUST** be protected (`atomic`, `critical`, ...).

* at least one of them is a write

The Same Example Again

Program

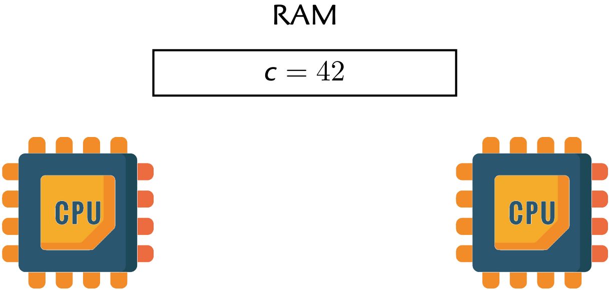
```
#include <stdio.h>

int main()
{
  int c = 0;
  #pragma omp parallel for
  for (int i = 0; i < 100000; i++) {
      c++;
  }
  printf("c=%d\n", c);
}
```

Execution

```
$ ./a.out
c=15074
```

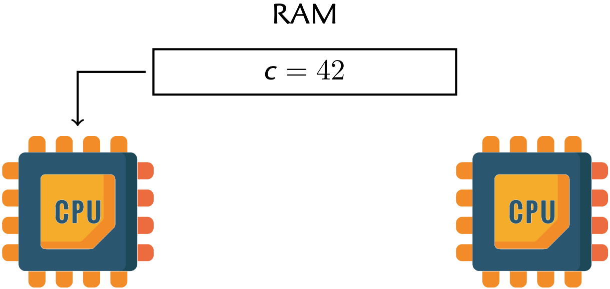
Interleaved Increments

Load-Store Hardware Architectures



Interleaved Increments

Load-Store Hardware Architectures



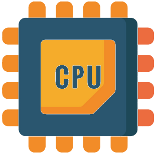
Interleaved Increments

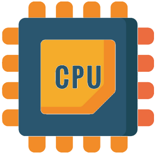
Load-Store Hardware Architectures

RAM

$c = 42$

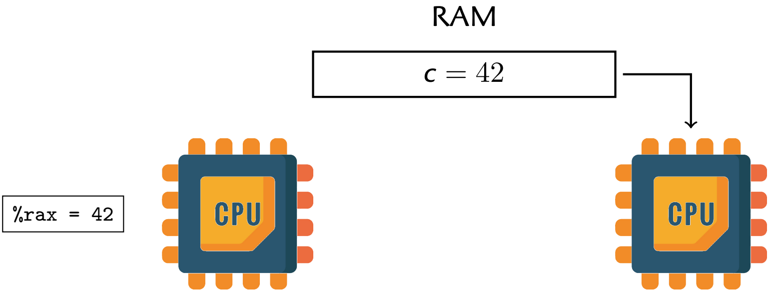
`%rax = 42`





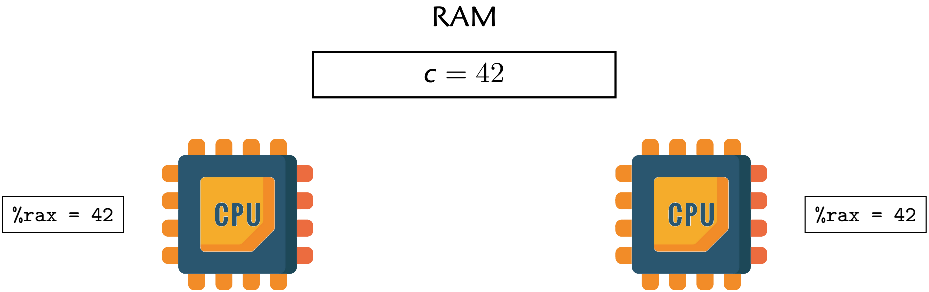
Interleaved Increments

Load-Store Hardware Architectures



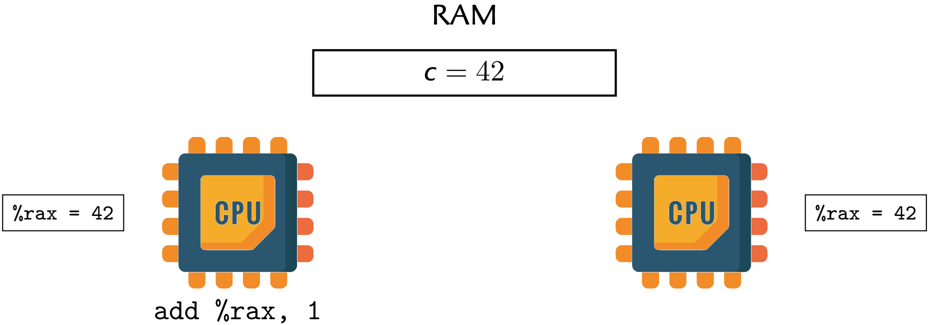
Interleaved Increments

Load-Store Hardware Architectures



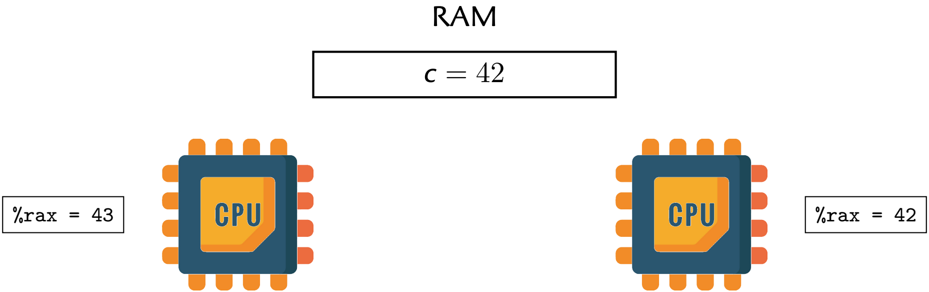
Interleaved Increments

Load-Store Hardware Architectures



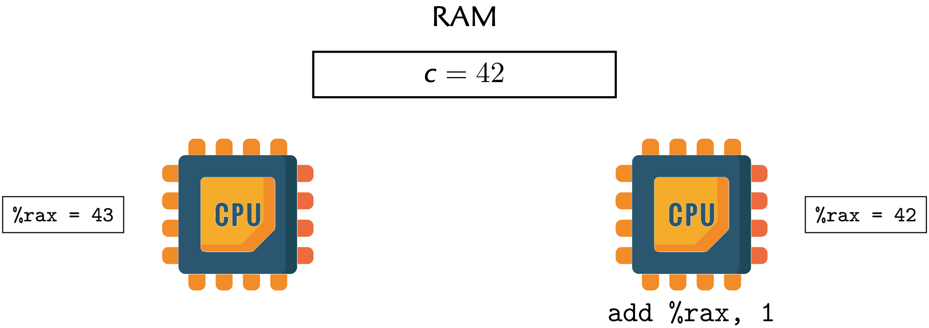
Interleaved Increments

Load-Store Hardware Architectures



Interleaved Increments

Load-Store Hardware Architectures



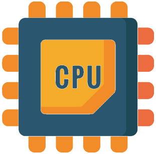
Interleaved Increments

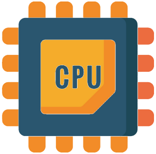
Load-Store Hardware Architectures

RAM

$c = 42$

`%rax = 43`

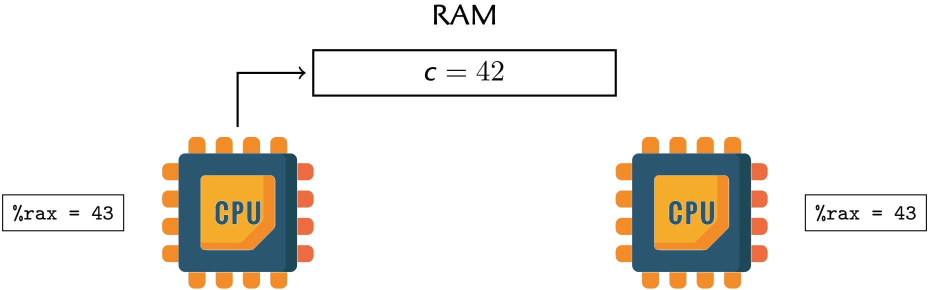




`%rax = 43`

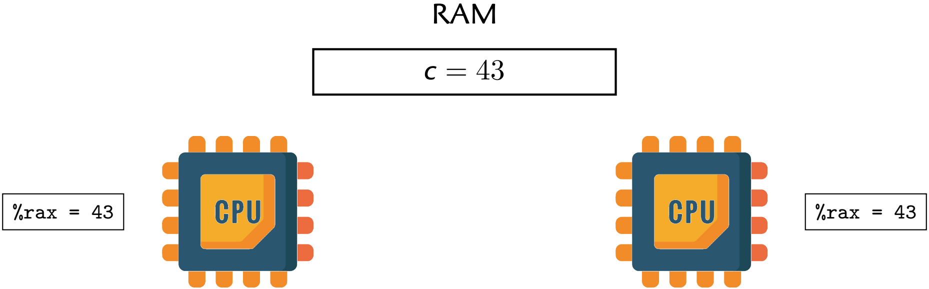
Interleaved Increments

Load-Store Hardware Architectures



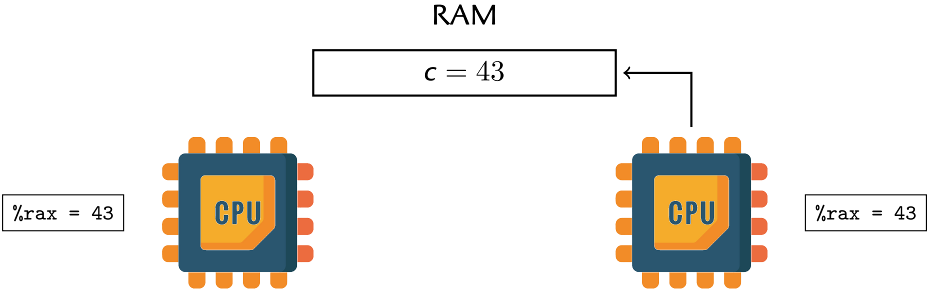
Interleaved Increments

Load-Store Hardware Architectures



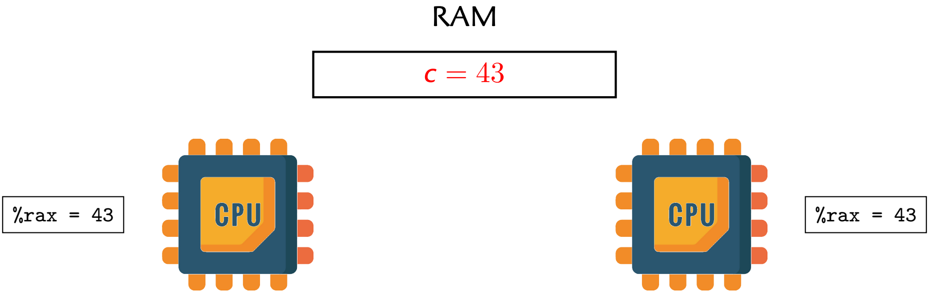
Interleaved Increments

Load-Store Hardware Architectures



Interleaved Increments

Load-Store Hardware Architectures



Bad

```c
#include <stdio.h>

int main()
{
  int c = 0;
  #pragma omp parallel for
  for (int i = 0; i < 100000; i++) {

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

Good

```c
#include <stdio.h>

int main()
{
  int c = 0;
  #pragma omp parallel for
  for (int i = 0; i < 100000; i++) {
    #pragma omp atomic update
    c++;
  }
  printf("c=%d\n", c);
}
```

Load-store architectures:

- `x++`, `x += 10`, etc. are **not atomic**
- Not even when performed by a single CPU instruction

Tasks

Since OpenMP 3.0 (2008)

French speakers: point d'orthographe

Tache
: marque, salissure, souillure (*stain*). *Une tache d'encre*

Tâche
: travail à exécuter (*task*). *Une tâche ardue*

Tasks

Since OpenMP 3.0 (2008)

French speakers: point d'orthographe

Tache	marque, salissure, souillure (*stain*). *Une tache d'encre*
Tâche	travail à exécuter (*task*). *Une tâche ardue*

Reminder: `#pragma omp parallel`

- A **thread team** is assembled
- ...
- An **implicit task** is scheduled on each thread
- It is **tied** (cannot migrate to another thread)

task = code + associated data ("closure"), executed by a thread

`omp task` Directive

```
#pragma omp task [clause], [clause], ...
structured block
```

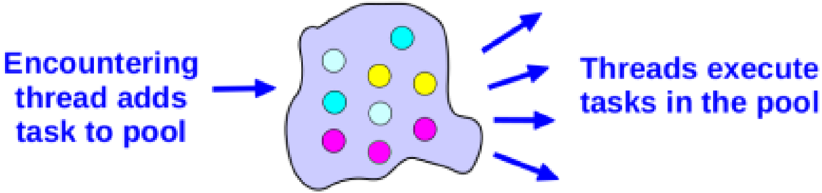
- The encoutering thread creates an **explicit task**
- Execution can be immediate or deferred
- Execution by one of the threads of the team... if available

`omp task` Directive

```
#pragma omp task  [clause], [clause], ...
structured block
```

- The encoutering thread creates an **explicit task**
- Execution can be immediate or deferred
- Execution by one of the threads of the team... if available
- A thread may suspend a task and start/resume another one...
- ... but only at a *task scheduling point*

"Task Pool"



Developer specifies tasks in application
Run-time system executes tasks

(An Overview of OpenMP 3.0, R. van der Pas, IWOMP2009)

Synchronizing Tasks

Fork/Join Model

"Normal" barrier

- Implicit: at the end of a parallel region, of `omp for`, ...
- Explicit: #**pragma omp barrier**

Guarantee: All tasks created by threads in the team are completed after the barrier

Task barrier: #**pragma omp taskwait**

- The current task waits for completion of its child tasks
- Only direct children, not descendant

Example: Iterating Over a Linked List

```c
struct item_t {
  void *data;
  struct item_t *next;
};
struct item_t *list;
```

Sequential

```c
struct item_t *e = list;


while (e != NULL) {

  process(e->data);
  e = e->next;
}
```

With tasks

```c
struct item_t *e = list;
#pragma omp parallel
#pragma omp single
while (e != NULL) {
  #pragma omp task
  process(e->data);
  e = e->next;
}
```

Example: Visiting the Nodes of a Binary Tree

```
struct tree_t {
  ...
  struct tree_t *left, *right;
};
struct tree_t *root;
```

Sequential

```
void walk(struct tree_t *t)
{
  ...
  if (t->left)

      walk(t->left);
  if (t->right)

      walk(t->right);
}


walk(root);
```

With tasks

```
void walk(struct tree_t *t)
{
  ...
  if (t->left)
      #pragma omp task
      walk(t->left);
  if (t->right)
      #pragma omp task
      walk(t->right);
}
#pragma omp parallel
#pragma omp single
walk(root);
```

`omp task` Directive (continued)

```
#pragma omp task [clause], [clause], ...
structured block
```

Associated clauses:

- `private` (*variable_list*), `firstprivate` (*variable_list*), `shared` (*variable_list*)
- `default(shared | none)`
- `untied`
- `depend`(*dependance-type*: *list*)
- `if` (*expression*)
- ...

Tasks: Variable Scoping

- Most useful data attribute with tasks: **firstprivate**

⇝ By default on all variables...

- ...**except** if they are already **shared**
  - Global cariables
  - Declared before the parallel section
  - Explicitly tagged as `shared`

Be careful with shared variables on the stack

```
void f()
{
    int i = 3;
    #pragma omp task shared(i)
    printf("%d\n", i);
}

#pragma omp parallel
#pragma omp single
f();
```

Tasks: When is shared Necessary?

```
struct tree_t {
  ...
  struct tree_t *left, *right;
};
struct tree_t *root;

/* Return \# nodes in the tree. */
int size(struct tree_t *t)
{
  int s_left = 0, s_right = 0;
  if (t->left)
      #pragma omp task shared(s_left)
      s_left = size(t->left);
  if (t->right)
      #pragma omp task shared(s_right)
      s_right = size(t->right);
  #pragma omp taskwait
  return 1 + s_left + s_right;
}
#pragma omp parallel
#pragma omp single
printf("%d\n", size(root));
```

Tasks: granularity

Creating a task has a non-trivial cost

Don't create microscopic tasks

- **if** *Clause* from the `omp task` directive
  - E.g. `#pragma omp task if(prof < PROF_MAX)`
  - The task is created anyway...
  - ...but executed immediately by the encountering thread
- `if` *instruction*:

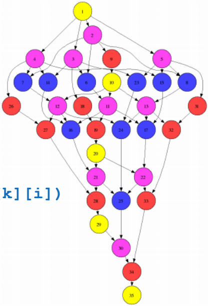
```
if (prof < PROF_MAX) {
  #pragma omp task
  stuff(...);
} else {
  stuff(...);
}
```

$\rightarrow$ to be preferred

Tasks: dependances

Example (Christian Terboven)

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



* image from BSC

Nested Parallelism

- A `parallel` directive inside a `parallel` directive
- In most cases, this is a no-op.
  - No more “available” threads to work on the nested parallel region: they are all in the parent parallel region.
- It is possible to force the creation of a new thread teams (cf. spec)

My opinion

- **STRICTLY USELESS** to do it voluntarily
- Could happen if a parallel program invoked a parallel library inside a parallel region

OpenMP Runtime Library

Some common functions (with `#include <omp.h>`):

- `void omp_set_num_threads(int num_thread)`: sets the number of threads to use in subsequent parallel regions
- `int omp_get_num_threads(void)`: return the number of threads in the current team
- `int omp_get_max_threads(void)`: upper bound on #threads in new teams
- `int omp_get_thread_num(void)`
- `int omp_get_num_procs(void)`
- `int omp_in_parallel(void)`: return true if we are inside a parallel region
- `double omp_get_wtime(void)`

Environment Variables

- `OMP_NUM_THREADS`
- `OMP_SCHEDULE`:
  `export OMP_SCHEDULE="static,4"`
  `export OMP_SCHEDULE="dynamic"`
- ...

Cheat Sheet

Most common

- The *must*: `#pragma omp parallel for`
- `atomic` and `critical` directives
- `schedule` and `reduction` clauses

Less common: SPMD mode

- Parallel region with explicit `#pragma omp for` inside
- `omp_get_thread_num()` and `omp_get_num_threads()`
- `barrier` or `single` directives

Rare (reserved for the most delicate cases...)

Everything else!