Any Other Bussiness

Luca Tornatore - I.N.A.F.

"Foundation of HPC - basic" course

DATA SCIENCE &
SCIENTIFIC COMPUTING

2022-2023 @ Università di Trieste

Outline

Sparse and different topics.
Either concepts and notions that were preparatory for the course or on-the-spot in-depth details that were aftermath of Q&A

Table of contents

Expressing performance

| Expressing performance

In these slides we introduce a metrics to estimate the performance of a code in exploiting some of the CPU's resources.

Specifically, we will focus on *(i)* how "fast" a code is and *(ii)* how well a code exploits the instruction-level parallelism capability of a CPU.

The metrics we will introduce are best-suited for those code sections that performs loops, which indeed are a large and significant fraction of scientific codes in general.

The "performance" intended differently, for instance as energy-to-solution or memory imprint, needs some different metrics and in some cases some dedicated measures.

Expressing performance

As you know, a CPU does not work "continuosly". At the opposite, its activity is regulated by an internal clock whose pace is of the order of billions ticks per seconds.
The typical CPU frequency is between 2 and 4 GHz, meaning that the time taken by a "clock cycle" is of the order of 0.5 - 0.35 *ns*.
We will refer to this time span as "a cycle".

Moreover, on the purpose if energy saving the CPUs have throttling capabilities, meaning that the clock frequency is not fixed but may be adapted to the worload. The larger the workload, the higher the clock and vice-versa.

| Expressing performance

Due to the variability of the CPU's clock, measuring the "wall-clock time-to-solution" is not always the best, or at least the only, metrics to be collected for the purpose of evaluating how some code snippet actually behave.

Quantifying the number of clock cycles spent on a code section is it may be even more informative than knowing how many seconds it recquired to execute.

Focusing on code sections that repeat a block of instructions over an array(*), which are very common and often represent the most computationally intense hotspots, that easily translates in a *cycles-per-element* (CPE) metrics.

| Expressing performance

In fact, we would be interested in knowing how much efficient our code is *per-element* rather than *per-iteration* since our implementation may be able to process more elements per iteration.

Conversely, in the case we wanted to estimate how well we are exploiting the super-scalar capability of the CPU, assessing how many i*nstructions-per-cycle* (IPC) are executed would be the adecquate metrics to look at.

We will see in the next lectures how to collect these sophisticated metrics in practice. As for now, the focus is on clarifying how the performance must be expressed and measured.

Expressing performance

Still, measuring the "execution time" of a code is a fundamental metrics that we should gather, at least for a first assessment.

Basically, you have access to 3 different types of "time".

1. "wall-clock" time
   Basically the same time you can get from the wall-clock in this room.
   It is a measure of the "absolute time".
   In POSIX systems, it is the amount of the number of seconds elapsed since the start of the Unix epoch at 1 January 1970 00:00:00 UT.

2. "system-time"
   The amount of time that the whole system spent executing your code.
   It may include I/O, system calls, etc.

3. "process user-time"
   The amount of time spent by CPU executing your code's instructions, strictly speaking.

Expressing p

Still, measuring the "execution time" of a code is a fundamental metrics that we should gather, at least for a first assessment.

Basically, you have access

1. "wall-clock" time
   Basically the same tim
   It is a measure of the
   In POSIX systems, it i
   since the start of the
2. "system-time"
   The amount of time t
   It may include I/O, sy
3. "process user-time"
   The amount of time s
   strictly speaking.

POSIX systems

POSIX is an ensemble of IEEE standards meant to define a standard environment and a standard API for applications, as well as the applications' expected behaviour.

It enlarges the C API, for instance, the CLI utilities, the shell language and many things.

All ⋆NiX systems are POSIX systems.
Other compliant systems are

- AIX *(IBM)*
- OSX *(Apple)*
- HP-UX *(HP)*
- Solaris *(Oracle)*

Expressing performance

You can measure all the quoted times :

- *Outside* your code
  → you measure the whole code execution
  you ask the OS to measure the time your code took to execute time:
  - using the `time` command (see `man time`)
  - using `perf` profiler
  - .. discover other ways on your system

- *Inside* your code
  → you can measure separate code's section
  you access system functions to access system's counter

- ☐ What time do we need ?
  Real, User, System, …
- ☐ What precision do we need ?
  1s, 1ms, 1us, 1ns
- ☐ What wrap-around time do we need ?
- ☐ Do we need a *monotonic* clock ?
- ☐ Do we need a *portable* function call ?

Expressing performance

Baseline: you call the correct system function right before and after the code snippet you're interested in, and calculate the difference (yes, you're including the time function's overhead).

- **`gettimeofday(..)`** returns the wall-clock time with μs precision
  Data are given in a `timeval` structure:

```
struct timeval {time_t      tv_sec;     // seconds
                suseconds_t tv_usec; };  // microseconds
```

- **`clock_t clock()`** returns the user-time + system-time with μs precision.
  Results must be divided by `CLOCKS_PER_SEC`

- **`int clock_gettime(clockid_t clk_id, struct timespec ..)`**

  `CLOCK_REAL_TIME` system-wide realtime clock;
  `CLOCK_MONOTONIC` monotonic time
  `CLOCK_PROCCES_CPUTIME` High-resolution per-process timing
  `CLOCK_THREAD_CPUTIME_ID` high-precision per-thread timing
  Resolution is 1 ns

```
struct timespec { time_t   tv_sec;       /* seconds */
                  long      tv_nsec;};  /* nanoseconds */
```

Expressing performance

Baseline: you call the correct system function right before and after the code snippet you're interested in, and calculate the difference (yes, you're including the time function's overhead).

- **int getrusage(int** who**, struct rusage ***usage**)**

  `RUSAGE_SELF` process + all threads
  `RUSAGE_CHILDREN` all the children hierarchy
  `RUSAGE_THREAD` calling thread

```
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long   ru_maxrss;        /* maximum resident set size */
    long   ru_ixrss;         /* integral shared memory size */
    long   ru_idrss;         /* integral unshared data size */
    long   ru_isrss;         /* integral unshared stack size */
    long   ru_minflt;        /* page reclaims (soft page faults) */
    long   ru_majflt;        /* page faults (hard page faults) */
    long   ru_nswap;         /* swaps */
    long   ru_inblock;       /* block input operations */
    long   ru_oublock;       /* block output operations */
    long   ru_msgsnd;        /* IPC messages sent */
    long   ru_msgrcv;        /* IPC messages received */
    long   ru_nsignals;      /* signals received */
    long   ru_nvcsw;         /* voluntary context switches */
    long   ru_nivcsw;        /* involuntary context switches */
};
```

Expressing performance

A possibility on a POSIX system is:

```
#define CPU_TIME (clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &ts ),\
                             (double)ts.tv_sec +       \
                             (double)ts.tv_nsec * 1e-9)

....

Tstart = CPU_TIME ;

// your code segment here

Time = CPU_TIME - Tstart;
```

Accumulate data on performance

Independently of what metrics you are accumulating, dealing with only 1 measure is not a good estimate: computer systems are really complicate ones and lots of things are going on continuosly, above all on modern architectures, and you may observe significant variations in your metrics from a run to another run.

As such, you must procede "*statistically*", i.e. by accumulating several measure and modelling the measure and system overhead.

For instance, acquiring the cycles' number or the time requires itself a number of cycles; a loop as an amount of inherent overhead. And so on.

Quite often, averaging over a "sufficient" number of runs and subtracting the known overhead is sufficient to get an good-enough estimate.

Accumulate data on performance

examples in pseudo-language:

```
double t_overhead;
timing_overhead = get_time();
for ( int i = 0; i < LOTS_OF_ITER; i++ )
   double this_time = get_time();
t_overhead = (get_time() - t_overhead) /
              LOTS_OF_ITER;
```

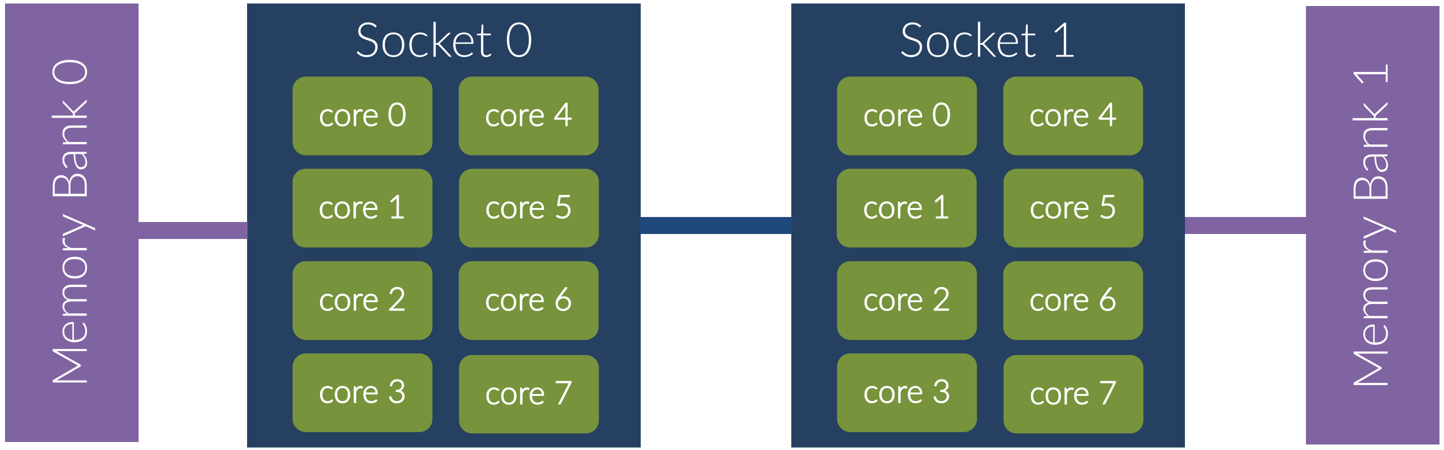
```
double timing = get_time();
  ...
  block of code you want to characterize
  ...
timing = get_time() - timing - t_overhead;
```

```
double timing = 0;
double stddev;
for( int i = 0; i < MANY_ITER; i++ )
  {
     ...
     double time0 = get_time();
     block_to_be_measured
     double time1 = get_time();
     double elapsed = time1-time0;
     timing += elapsed - t_overhead;
     stddev += elapsed * elapsed;
     ...
}

timing = timing/MANY_ITER;
stddev = sqrt( stddev/MANY_ITER - timing*timing);
```

Accumulate data on performance

Inside a socket there are many cores, from 4-8 in commodity CPUs found on laptops or desktop computer up to ~64 in high-end CPUs for servers and computational nodes



The O.S. can *migrate* you code's threads from one core to another and also from one socket to another. Whether the data are also migrated depends on the adopted policy.

Accumulate data on performance

Then, running a program without ***binding*** it to a specific core and a specific memory banck may result in a non-optimal behviour.

So, when you are interested in profiling a code, you must be sure that it will not be migrated.
You can ask that to the O.S. by using

```
taskset
numactl
```

just have a look to the related man pages for all the details

Accumulate data on performance

`numactl -H`	exposes the topology of the node

`numactl --cpunodebin=`*`n`*
`numactl -C `*`n`*
	bind the execution to core *n*

`numactl --membind=`*`n`*
`numactl -m `*`n`*
	bind the memory to memory bank associated with core *n*

`numactl -C +`*`list`*	bind the execution to the *relative* cores listed in *list*. *example:* `numactl -C +0,2,4 prog.x` *will execute* `prog.x` *on the cores 0,2 and 4 of the cpuset given to the job.*

Studying the scalability

What is the "scalability"

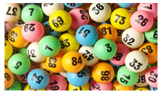
You will be often requested to prove the "scalability" of a code, for instance when you apply for computational time at a facility.

The "scalability" is the behaviour of the code as:

1. you increase the number of computational resources at a fixed problem size; this is called **strong scalability**.
2. you increase the nummber of computational resources scaling up proportionally the problem size, i.e. the workload for each computational resource is constant;
   this is called **weak scalability**.

Note: "computational resources" may be either cores (physical cores, not logical cores) or - increasingly so - entire nodes.

Parallel generation of pseudo-random numbers

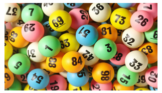


| Don’t use std generators

The generation of "true" pseudo-random number in parallel is an active field of research and a difficult issue.

So, you're of course not required to solve that in the cases you need to generate random sequences in parallel, but just to pay attention: **using the standard routines you have used so far when you are in multi-threading is an incorrect approach because they rely on a status register that is shared.**

It may still be adequate when you are in distributed-memory, provided that the initial seed differs from task to task.

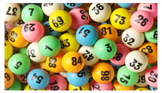


How to generate / 1

You may want to use `rand_r()`

```
#pragma omp parallel
{
    int myid   = omp_get_thread_num();
    int myseed = function_of_myid( myid );
    int random_number = rand_r();
}
```

Although it satisfies only elementary statistical properties, it would be enough for many simple usages.



| How to generate / 2

For more sophisticated series, use the `drand48_r()` family which are *only* GNU extensions:

```
#pragma omp parallel
{
    int myid   = omp_get_thread_num();
    struct drand48_data myrng;
    // here you initialize the 3 entries of mystatus
    // with some function_of_myid( myid );
    long int myseed = FUNCTION_THAT_GET_UNIQUE_SEED_per_THREADID(myid);
    srand48_r( myseed, &myrng );
    double random_number;
    drand48_r(&myrng, &random_number);
 }
```

THE C PROGRAMMING LANGUAGE

Some sparse topic on C

Outline

Pointers

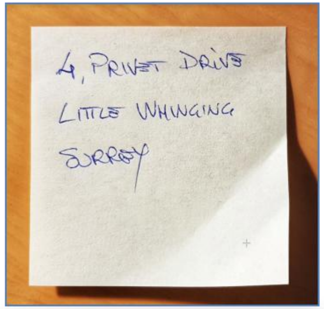
Although it is extraordinarly simple, this topic is usually a terrifying one for most people.
Although it's plenty of good primaries on the web, and I encourage you to search for them, I think that some simple words are in order.

Pointers

As first, let's start with a very simple concept.
I guess you have a special physical place, however you love to imagine it, that you call "home".

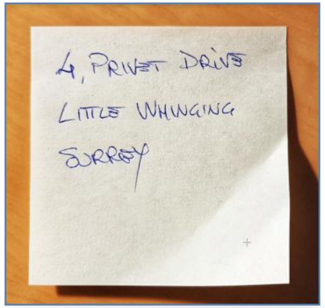
Let's suppose you are boring normal people, and that your place has an *address:*

4, Privet Drive, Little Whinging, Surrey



You appreciate the fact that this address needs some memory storage to be kept; in my case, a simple sticky note.

Pointers

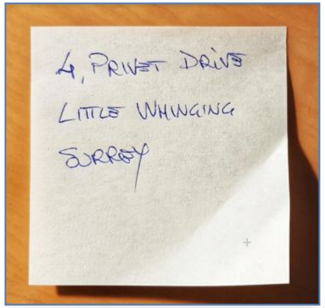


4, Privet Drive, Little Whinging, Surrey

You appreciate the fact that this address needs some memory storage to be kept; in my case, as we said, a simple stick note.

You also appreciate that there is a clear difference between the string written on the note and your actual home (try to inhabit my note if don't see it).

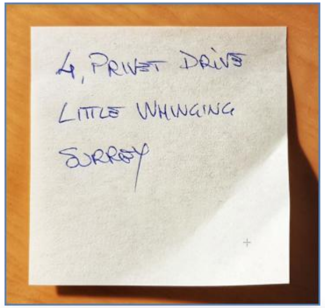
Pointers



So, my note *points* to your home, and occupy some well-defined physical space for the purpose (the sticky note sheet), but it is *not* your home whose physical occupancy does not depend on my note (it's hard to know from the note whether it's a castle or a roulotte).

Conversely, your home is somewhere else, in a well-defined place that is reachable - let us say addressed - by using my note.

Pointers



If you change something in your house, the address on my note is still valid and can be useful to reach you.

Nobody noticed that you renewed your bathroom.

Pointers

12, Grimmaud Place, London

If you move, and your home has now a different address, I need to know it to get there and invite myself for dinner.
To save your new address, I can still use the same sticky note sheet, i.e. the same physical storage of the same size.

The physical location has changed, but I can still use the same sticky note to reach you.

Pointers

All in all, then:

- a ***pointer*** is a variable, i.e. a memory location of fixed size (8B in 64bits systems) which contains an *address*, specifically a memory address and not your place's one.
That address is the *starting point* of a memory area.
So, a pointer can point to an integer (4B), a double (8B) an array of 1G items. Whatever stays in memory has some location where it is stored and that location can be pointed to by a pointer variable.

- ***de-referenceing a pointer*** means to get to the pointer variable, i.e. at the memory location that the variable occupies, to read those bytes acquiring the address and then to get to that memory address

Pointers

- a *pointer* is declared as

```
        type *ptr_variable_name;
```

examples:

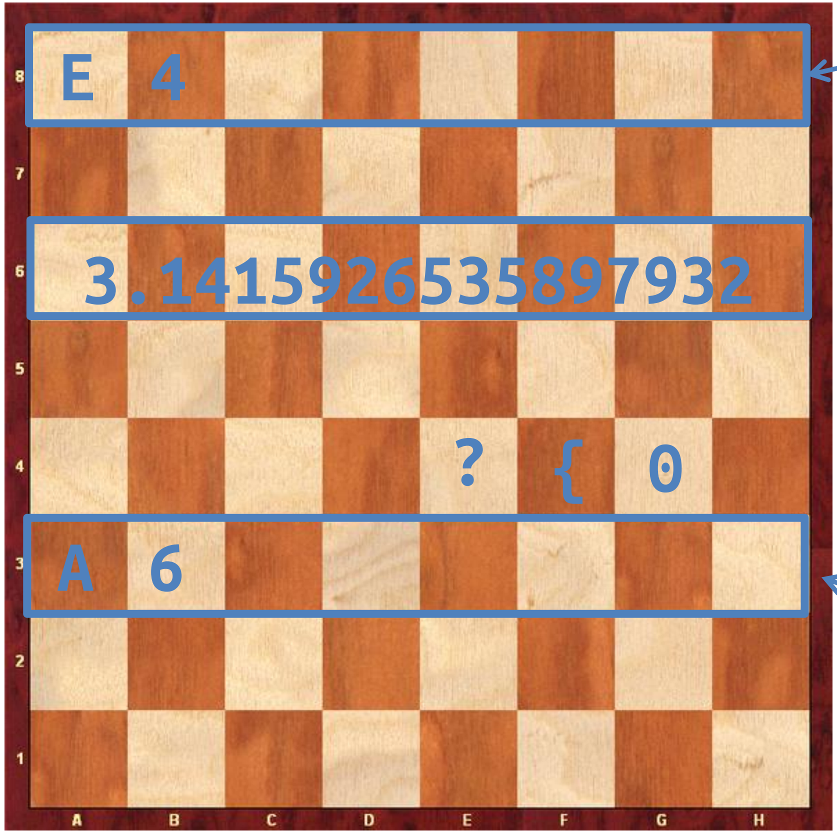
`char *c;`	points to a char
`double *d;`	points to a double
`struct who_knows *w;`	points to a struct `who_knows`

you assign a value to a pointer variable by assignment:
c = 0x123456; c = &my_preferred_letter;

you read the address it points to by de-referencing:
*c is actually the content of the byte pointed by c, not the c's value

note that &c is the c's own address.

Pointers



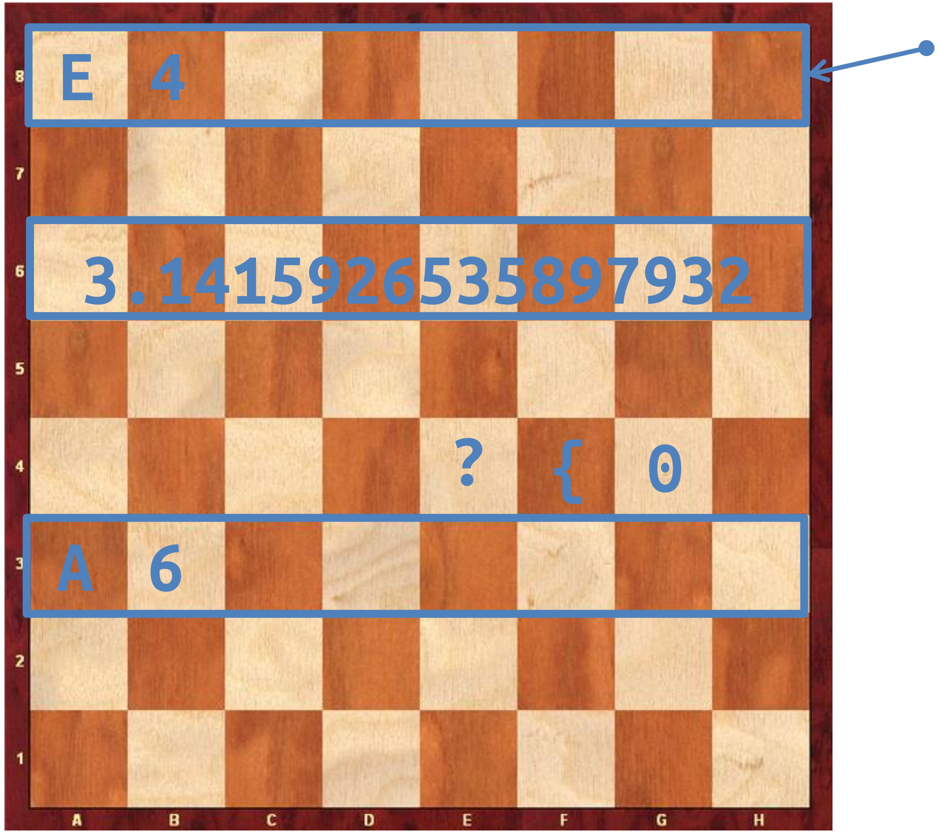
`char *c;` points to memory location E4. It resides at memory location A8, and lasts 8 bytes until H8, so the c memory address, &c, is A8.

the memory bytes from A6 to H6 are interpreted as a double if you access them starting from A6 by de-referencing the pointer **`d`** by `*d`;

the memory byte at E4 is interpreted as a char if you access it by de-referencing the pointer `c` by `*c`;

`double *d;` points to memory location A6. It resides at memory location A3, and lasts 8 bytes until H3, so the d memory address, &d, is A3.
Pointers always occupy 8 bytes.

Pointers' arithmetic



as we've seen, **c** points to E4, which contains a byte that interpreted as a character reads as "?".

c is a normal variable (actually, I remind you: it is 8B that we interpret as an address) and as such can be used in arithmetic operations.

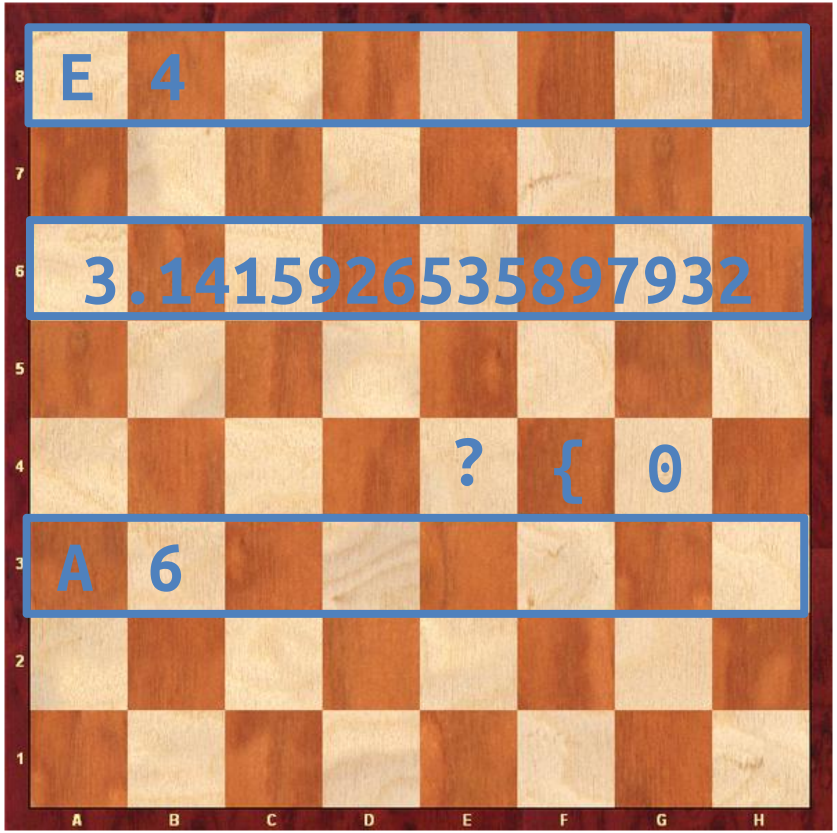
For instance, what **c+1** is ?
The operation +1 increases the *value* of **c** by 1; as a result, **c** will point to F4 (*), i.e. to "{".

Instead, since ***c** means "read the value at the location **c**", **(*c)++** increases the value pointed by **c**. In other words, in E4 we'll have ?+1.

What happens if you increase by 1 the value of **d**, the pointer to double ? will it point to A7?

(*) let's imagine che in our chessboard-memory the addresses increases along the rows, so as A1, B1, C1,.. H1, A2, B2,..H2, A3, etc.

Pointers' arithmetic



What happens if you increase by 1 the value of **d**, the pointer to double ? will it point to A7 ?

The answer is no, because of the pointers arithmetic: when we increase by 1 a pointer, its value - i.e. the address it points to - is increased by the byte-size of the type it points to.

If a pointer points to `char`, since `sizeof(char)` is 1, the increase is 1.
If a pointer points to a `double`, since `sizeof(double)` is 8, the increase is 8bytes, or 1 in `double` units.
If a pointer points to an `int`, the increase is 4 bytes and so on.

```
double *array = malloc(sizeof(double)*N);
double *end = &array[N-1];
double *ptr = array;
```

these 2 loops are equivalent

```
for ( int i = 0; i < N; i++ )
   S += array[i];

for ( ; ptr <= end; ptr++ )
   S += *ptr;
```

Pointers zoo

what this strange creature is ?

```
char **monster;
```

Pointers zoo

what this strange creature is ?

```
char **monster;
```

Let's reason by steps, from right to left (as you should read a C declaration)

1) `monster` → we declare a variable monster
2) `*monster` → it is a pointer
3) `**monster` → it points to a pointer (remind, a pointer is just a variable and it can be pointed)
4) `char **monster` → the pointed pointer points to a `char`

so `*monster` is a pointer which points to a `char`.
Good to know. But what does it mean?

Pointers zoo

Before fully understand this, we should stress a fact: a pointer points to a precise memory location *but you are not limited to access only that location.*

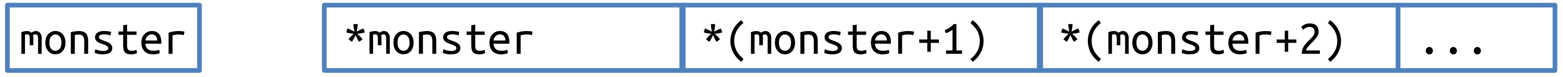
Actually, pointers are the way in C you address dynamically-allocated array:

```
double *array = malloc( sizeof(double) * N );
```

you have allocated room for `N` double entries and the location at the beginning of that memory region is stored in `array`.
Hence, you can access the *ith* element both by `*(array + i)` or by `array[i].`

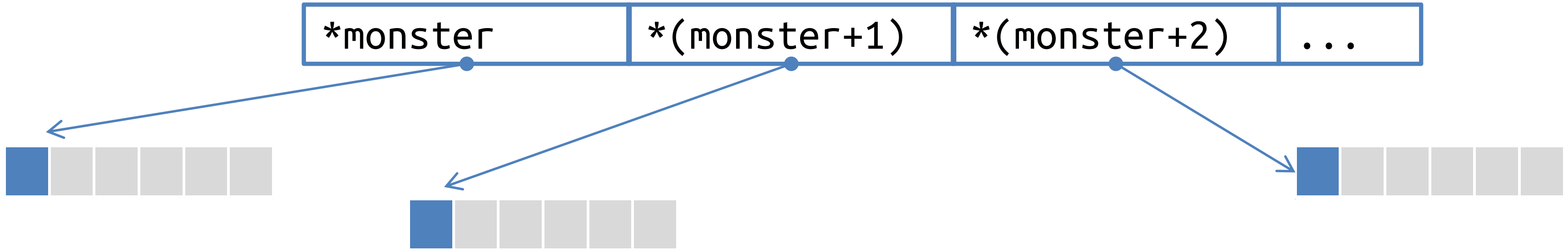
Pointers zoo

Then, it happens that since **`monster`** points to a pointer, also **`monster+1`** points to a pointer (8bytes away because a pointer is 8B long), and so on:



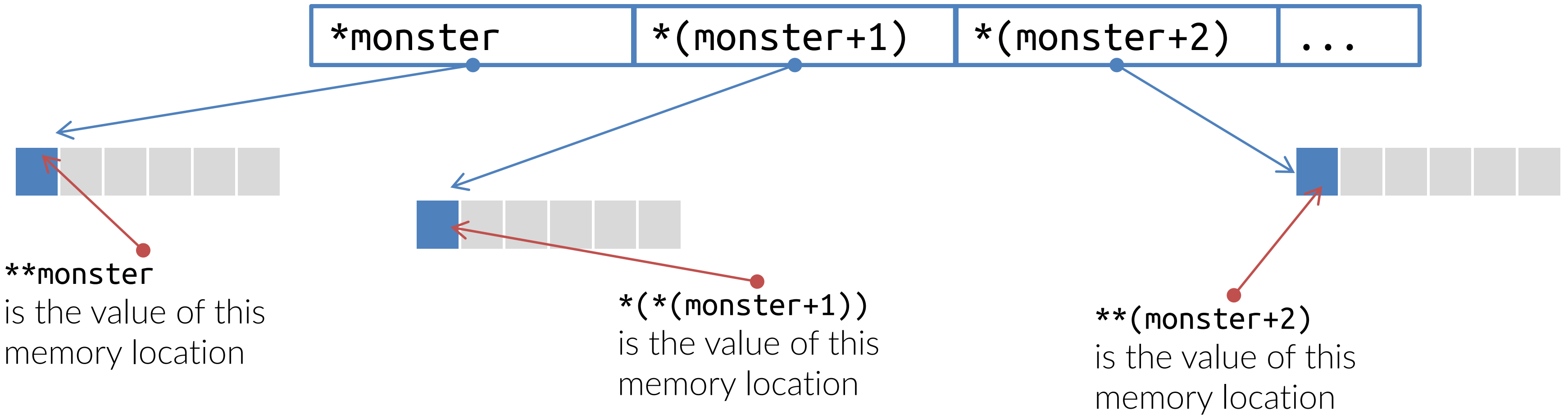
note that *monster+n is *very* different than *(monster+n)

`*(monster+n)` are all interpretable (formally they are since we are referencing them through **`monster`**) as pointers to **`char`**:



Pointers zoo

`**(monster+n)` are the value at the byte pointed by `*(monster+n)`



`**monster`
is the value of this memory location

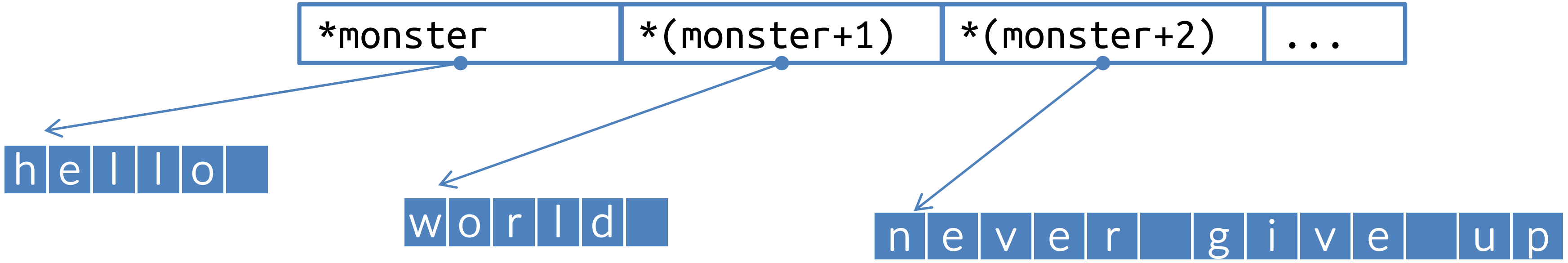
`*(*(monster+1))`
is the value of this memory location

`**(monster+2)`
is the value of this memory location

`*(*(monster+n)+j)` is the *jth* byte after `*(monster+n)` which actually starts to look like a string..

Pointers zoo

`**(monster+n)` are the value at the byte pointed by `*(monster+n)`



then

`**monster` = ‘h’, `*((*monster)+1)` =’e’, `*((*monster)+2)` =’l’, …

`**(monster+1)` = ‘w’, `*(*(monster+1)+1)` =’o’, `*(*(monster+1)+2)`=’r’, …

or, in other words,

`*monster` = “hello”, `*(monster+1)` = “world”, `*(monster+2)` = “never give up”

Pointers zoo

That is actually how you access the command-line's arguments, for instance:

```
int main (int argc, char **argv)
  {
     ... let's see the worked example, arguments.c
  }
```

How to exit from an OpenMP parallel region

OpenMP $\geq$ 4.0 **has** a cancellation construct that causes a loop (or other constructs, see below) to exit before its natural end.

> *NOTE : `gcc` up to v.9.3 does not yet support this construct. Recent enough `icc`, `pgi` and `clang` compilers do.*

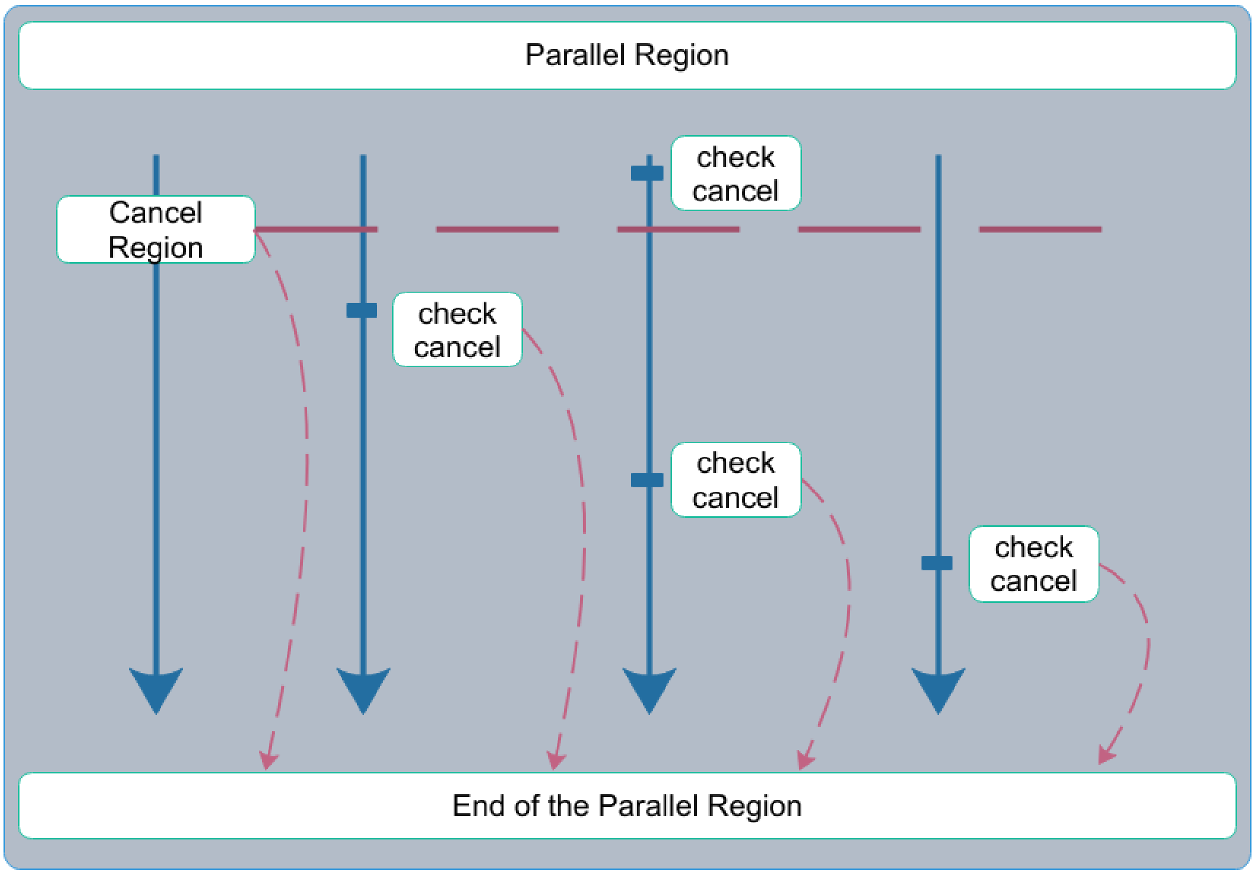
Once a thread meets the exit condition, it breaks issuing a cancel request to all other threads. The other threads are checking whether such a request has been issued, and, if so, they also exit the loop.

For the cancellation directive to work, The environmental variable `OMP_CANCELLATION` must be set to `true`:

```
export OMP_CANCELLATION=TRUE
```

In the cancellation model, there are 2 fundamental point:

1. the `cancel` construct, at which a thread breaks the loop
2. the `cancellation` point, at which a thread checks whether there is a request for cancellation



The cancellation construct exists for the following parallel regions:

- `parallel`
- `for`
- `sections`
- `taskgroup`

Example:

```
#pragma omp for
for ( i = 0; i < N; i++ )
  {
    do_some_work( ... );

    if ( i == key )
      {
        #pragma omp cancel for
      }
    #pragma omp cancellation point for
  }
```

You find this code above in a more comprehensive and commented example in the assignment directory, `cancel_construct.c`.