Scalable BIBOP garbage collection for parallel functional programs on multi-core machines

Luca Saiu `<saiu@lipn.univ-paris13.fr>`

Équipe LCR, LIPN, Université Paris 13
GNU Project

Eventually, parallel machines have come.

But we’re still unable to program them.
Many hard, open problems. What's the right solution?

- Automatic parallelization
- Structured parallel programming (skeletons)?
  - ...with automatic dynamic reconfiguration?
- Data-flow?

We don’t know yet. Anyway, we need **high-level** tools and languages.

High-level languages depend on ***GC***.

Requirements

- Scalable and fast allocation
- Scalable and fast collection
  - Stop-the-world, non-incremental

Throughput,
not latency

- Ease of interfacing with C and compiler-generated assembly code
  - Non-moving is easiest (mark-sweep)
  - No safe points (less easy, but worthwhile (?))
  - ...not necessarily a `malloc()` drop-in replacement
- Make a good use of modern memory hierarchies
  - *Memory density* [more to come]

We depend on hardware performance models

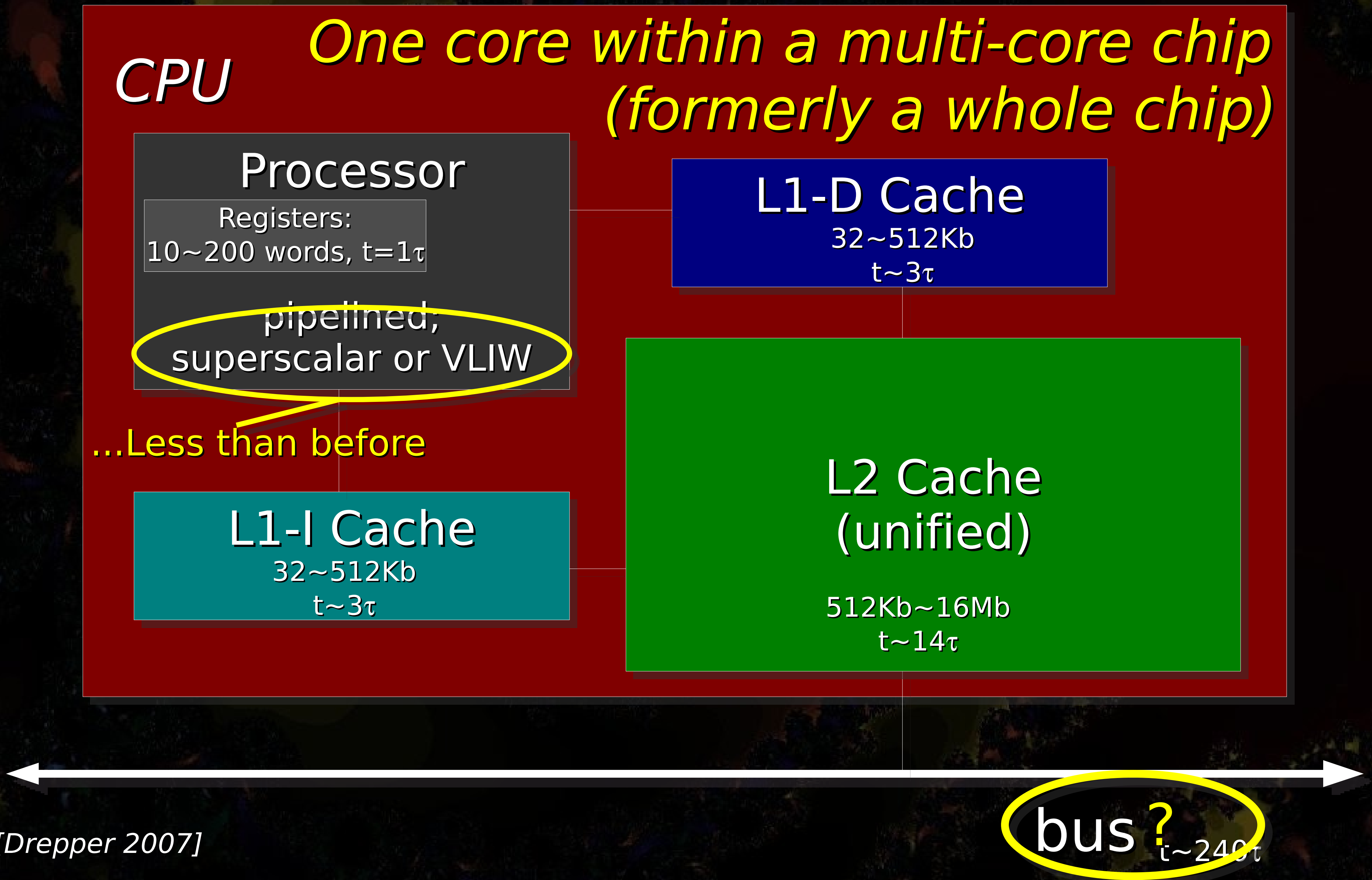
Here comes a very short summary of recent memory architectures.

If you are interested in more details, see:

[Drepper 2007]

Ulrich Drepper: *What every programmer should know about memory*. Technical Report. RedHat, November 2007, 114 pages.

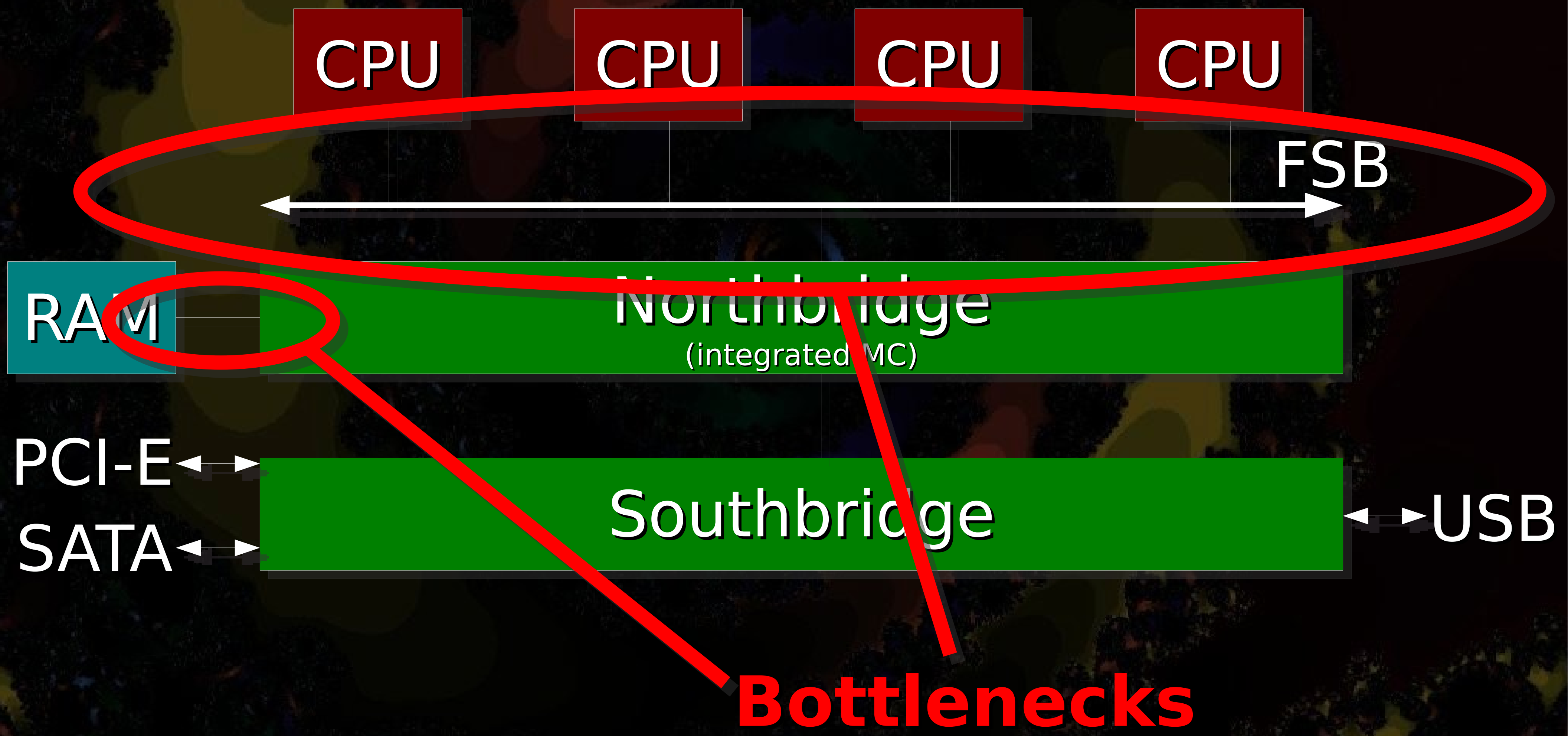
A simple modern CPU [core]



[Drepper 2007]

Multi-cores are SMPs (1)

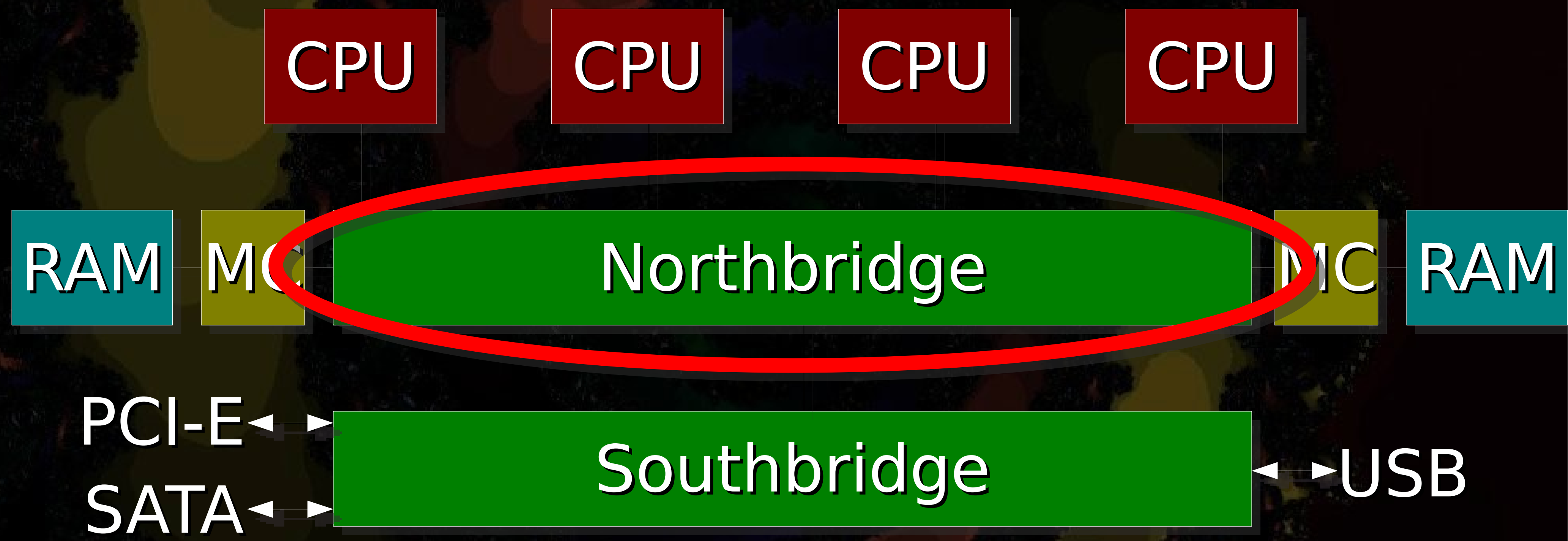
Multi-cores as SMPs: “commodity” architecture



[Drepper 2007]

Multi-cores are SMPs (2)

Multi-cores as SMPs: a more expensive solution



Several external MCs, no FSB: modern NorthBridges tend to have high bandwidth

[Drepper 2007]

Are mult

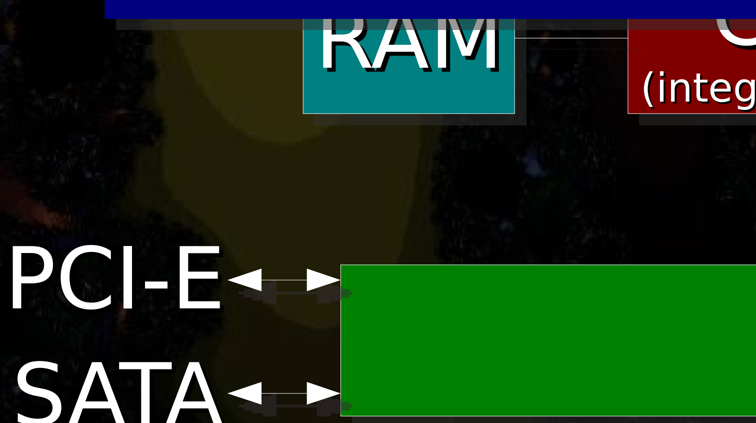
This is **very hard** to efficiently exploit.

Multi-cores as NU

Pure NUMAs are off-topic today

e message passing r clusters) among ocesses when then NUMA effect is important.

No shared heap: parallel non-distributed GC becomes irrelevant...



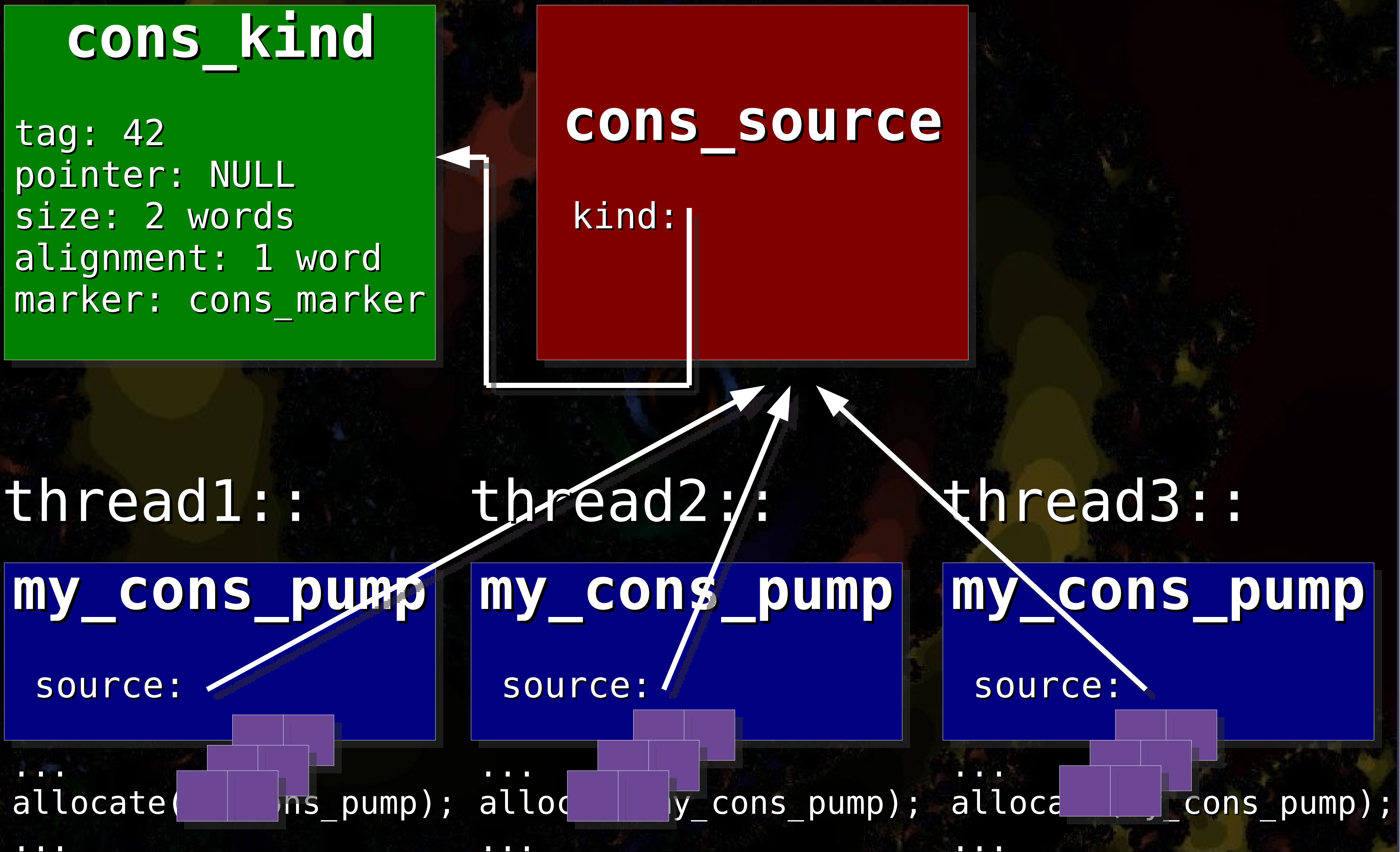
[Drepper 2007]

The “NUMA effect” is more pronounced with longer distances between pairs of CPUs

User-level architecture (1)

- **Kinds**: “shapes” of groups of objects (size, alignment), metadata (tag and pointer)
- **Sources**: global, “inexhaustible streams” of objects of one given kind
- **Pumps**: thread-local allocators which make one new object per request

User-level architecture (2)



What does not fit in the picture

- **Kindless objects**, particularly objects whose size is only known at creation time.

[More on this later]

- **Other parts of the interface**: explicit collection, collection disabling, tuning.

Canonical and unenlightening.

Essential user API

```
/* A tracer is a pointer to a function taking a pointer
   as its parameter and returning nothing: */
typedef void (*epsilongc_tracer_t)(epsilongc_word_t);

/* Create a kind: */
epsilongc_kind_t
epsilongc_make_kind(const size_t object_size_in_words,
                    const epsilongc_unsigned_integer_t
                          pointers_per_object_in_the_worst_case,
                    const size_t object_alignment_in_words,
                    const epsilongc_metadatum_tag_t tag,
                    const epsilongc_metadatum_pointer_t pointer,
                    const epsilongc_tracer_t tracer);

/* Create a source from a kind: */
epsilongc_source_t epsilongc_make_source(epsilongc_kind_t kind);

/* Initialize a (thread-local) pump from a source: */
void epsilongc_initialize_pump(epsilongc_pump_t pump,
                               epsilongc_source_t source);

/* Finalize a pump before exiting the thread: */
void epsilongc_finalize_pump(epsilongc_pump_t pump);

/* Allocate a kinded object from a thread-local pump: */
epsilongc_word_t
epsilongc_allocate_from(epsilongc_pump_t pump);
```

```
/* Lookup metadata: */
epsilongc_tag_t
epsilongc_object_to_tag(const epsilongc_word_t object);

epsilongc_metadatum_pointer_t
epsilongc_object_to_metadatum_pointer(const epsilongc_word_t
                                      object);

epsilongc_integer_t
epsilongc_object_to_size_in_words(const epsilongc_word_t
                                  object);

/* Allocate kindless objects: */
epsilongc_word_t
epsilongc_allocate_words_conservative(const epsilongc_integer_t
                                      size_in_words);
epsilongc_word_t
epsilongc_allocate_words_leaf(const epsilongc_integer_t
                              size_in_words);
epsilongc_word_t
epsilongc_allocate_bytes_conservative(const epsilongc_integer_t
                                      size_in_bytes);
epsilongc_word_t
epsilongc_allocate_bytes_leaf(const epsilongc_integer_t
                              size_in_bytes);
```

That's it.

BIBOP

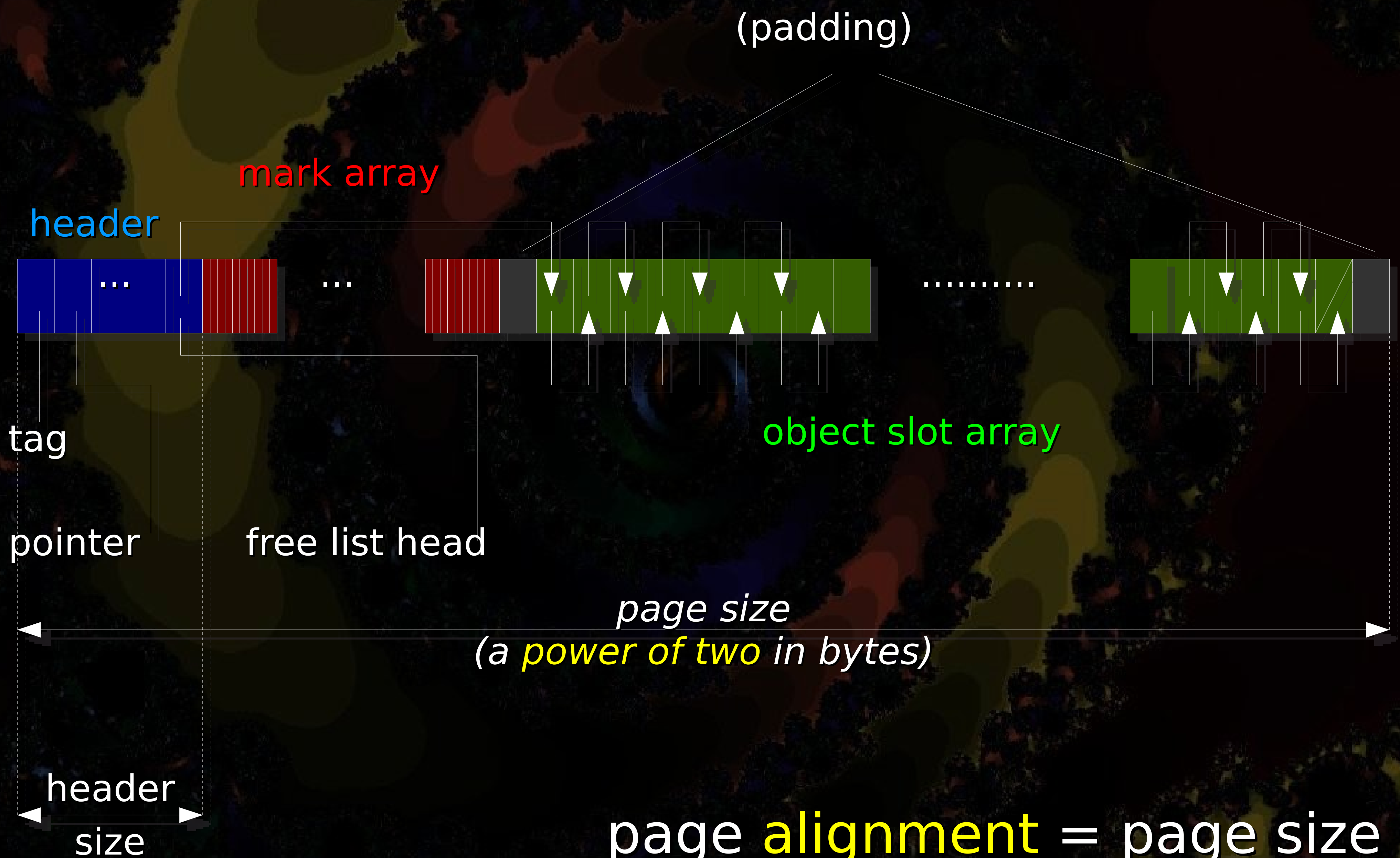
- Segregate objects into "pages" of fixed size and alignment: each page only holds objects of one kind.

  (not necessarily for my definition of "kind")

- Store kind identification and (some) metadata in a table, the "Big Bag Of Pages"

- Idea and first implementation by Steele, 1977: MacLisp on the PDP-10
  - Lots of variants since then, including the one by Boehm...
  - ...my version is similar to [Kriegel 1993]

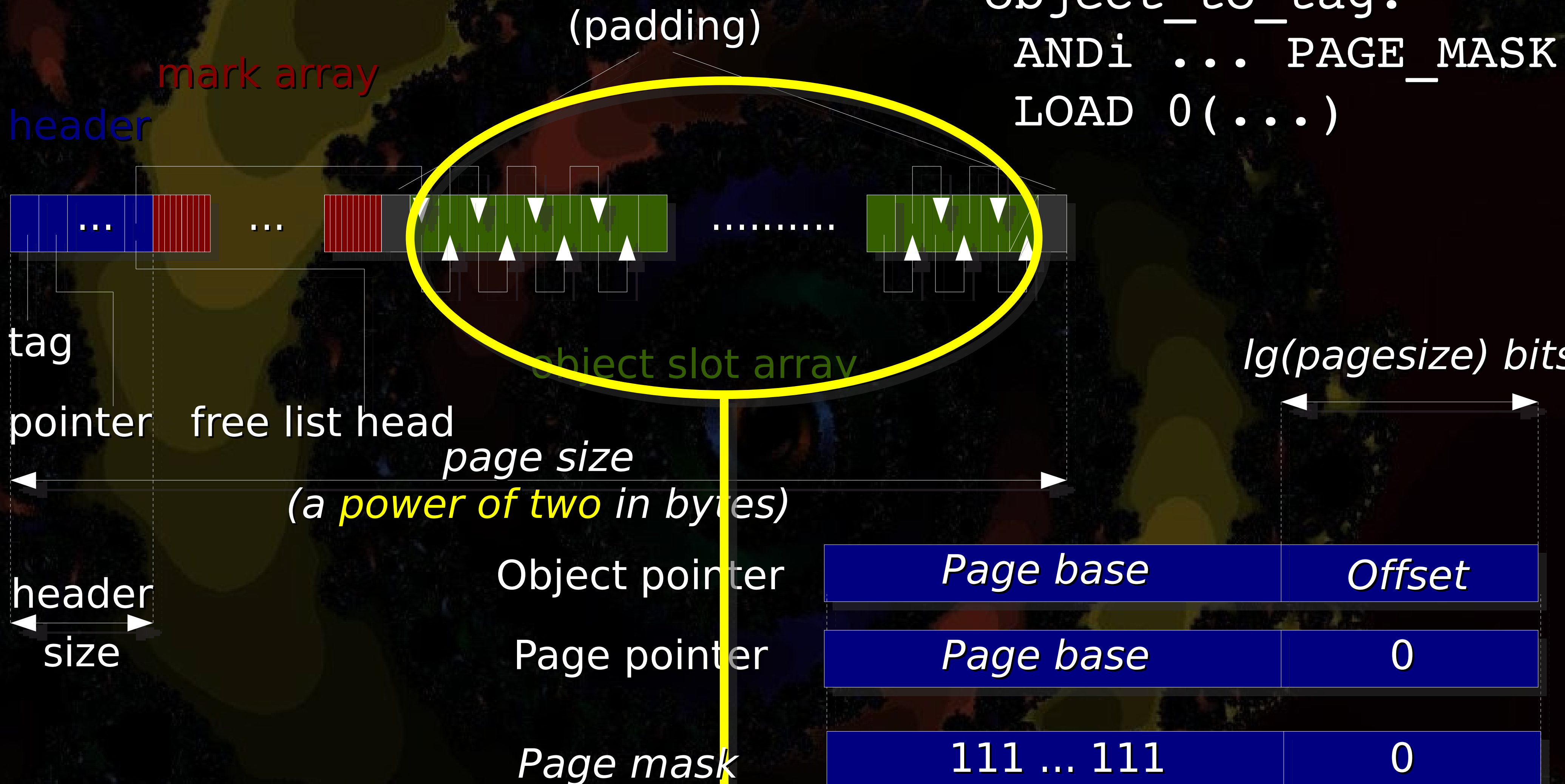
Structure of an (empty) page as in epsilongc



page alignment = page size

What' s the advantage

```
object_to_tag:
 ANDi ... PAGE_MASK
 LOAD 0(...)
```




```
object_to_pointer:
 ANDi ... PAGE_MASK
 LOAD 4(...)
```

Only payload here (no headers)

word size

BIBOP for parallel machines

- Why is BIBOP a good idea for parallel machines?

- Why store metadata in page headers, with modern cache architectures?
  - Boehm **did that** for the first versions of his collector (~1989), **but then changed** according to [Jones-Lins 1996]

Page primitive operations (for mark-sweep)

- Page creation and destruction
- Page sweeping
- Page refurbishing

Take in account cache (and OS) effects for each operation

- ~~Allocation from a page?~~
  - Not really: we have good reasons to do this with a pump [in a minute]

Implementation of user-level structures

- **Kinds** are trivial records
  - They just pre-compute at creation time some data (particularly offsets which will be needed by all the pages of the kind)
- **Sources** contain lists of pages not currently “held” by any thread
  - According to the sweeping policy we may need different lists for full pages [in a minute]
- **Pumps** contain a reference to a “held” page (or NULL) and “cache” important data. They are tread-local!

Object allocation from a pump

This is performance-critical: let's have a look at the source code

Parallel marking...

- ...is not so hard
- It might need some atomic memory intrinsics (depending on how mark arrays are implemented)
  - It's **very** disruptive for the cache
- Find pointers conservatively only in the roots
  - By default C stack, registers, user-registered buffers

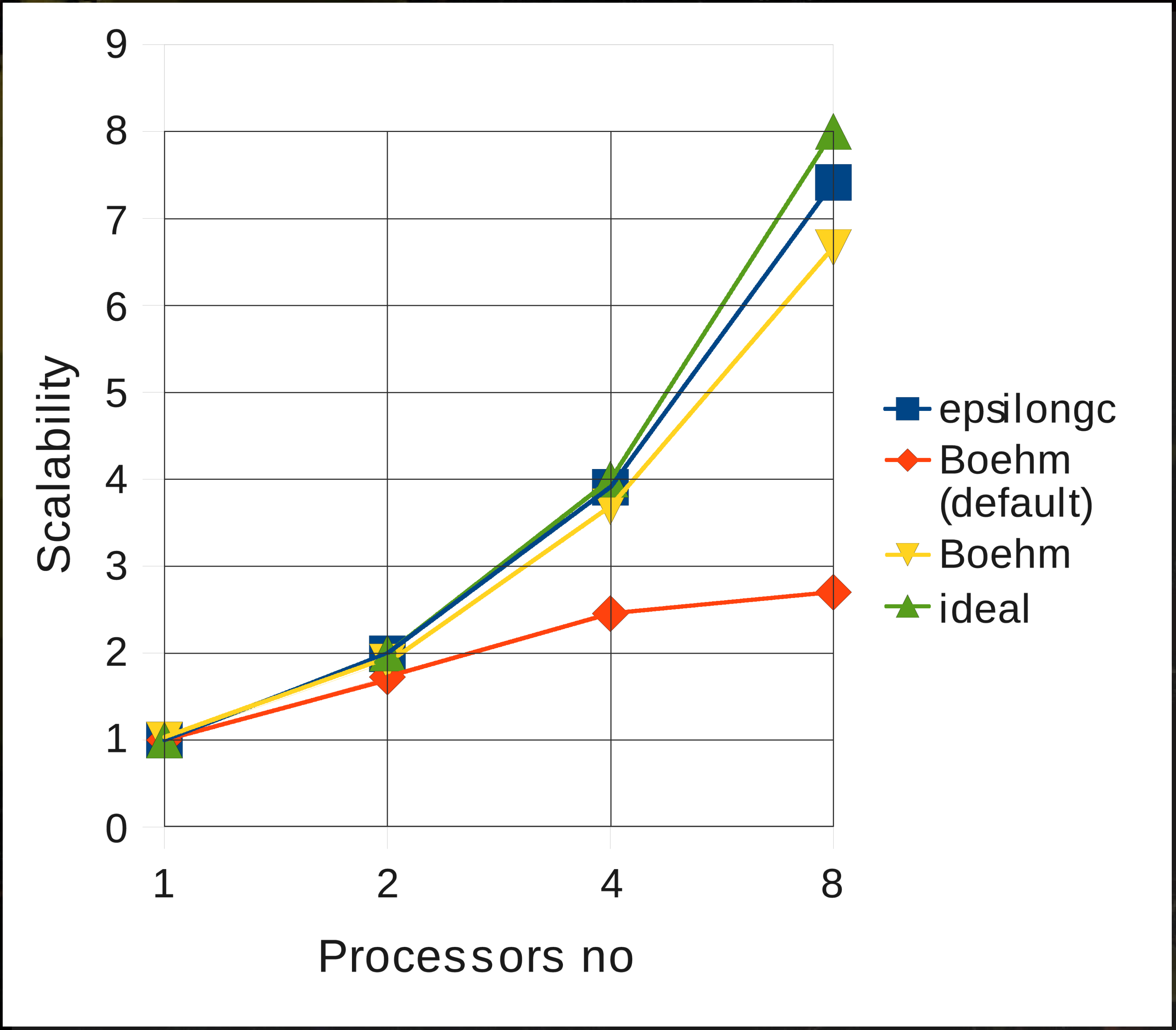
`setjmp()` or `getcontext()`

Parallel sweeping...

- ...is easy
- But it completely trashes L1 and L2 for all CPUs
  - *On-demand sweeping* instead can even serve as pre-allocation “prefetching” [like Boehm]
  - It's even better if we do it backwards

- Each page free list is ordered by element address (good for *locality* and *automatic hardware prefetching* for mostly empty pages)

Scalability (total completion time)



Memory density

Given a kind k of objects with alignment a_k and size s_k, I define the effective size e_k needed to store each object, and the corresponding *memory density* d_k, the number of objects representable per word, as:

$$e_k \triangleq a_k \cdot \left\lceil \frac{s_k}{a_k} \right\rceil \qquad\qquad d_k \triangleq \frac{1}{e_k}$$

Memory density is an index of the number of objects representable per cache line.
Per-object headers count as part of the size.

Memory density should be maximized.

- Sounds reasonable...
- But it's **not** yet experimentally confirmed

Implementation

- **C**
- Autoconf options, lots of `#ifdefs`.
- Macros, `__attribute()`s, inlining hacks
- **~5,000 LoC**, heavily commented
  - Surprisingly easy to understand for being such a low-level, inherently concurrent software.

Distributed as a sub-project of *epsilon*, part of the GNU Project. **GPLv3 or later**

Portability

- The usual “reasonable” assumptions on C.
- **TLS**: uses `__thread`
- **Processor-agnostic**: *endianness*, *word size*, *stack growth direction*...
- **Dependencies**: (maybe) GNU libc, (currently) POSIX threads, Unix signals, (at compile time) GCC.

- Performance-critical functions are **easy to re-implement in assembly** as compiler intrinsics (probably less than 50 instructions, total).

For more information

http://www.gnu.org/software/epsilon

http://www-lipn.univ-paris13.fr/~saiu

saiu@lipn.univ-paris13.fr

Thanks