



Memory-Level Parallelism

Dependent and independent accesses | Practical 2

Ömer Gölcük — Sabancı University

Course lead: Prof. Dr. Özcan Öztürk

Following a pointer chain

The next node's address is stored in the current node.



`p = p->next;`

Within one chain, each load needs the preceding pointer value.

Each node stores one successor pointer in a 64-byte aligned slot.

Dependency concept after [1]; diagram original. Node numbers are labels, not addresses or a measured trace.

The two traversal kernels

Serial: one ring at a time

```
for (int c = 0; c < 4; ++c) {  
    Node* p = start[c];  
    for (std::size_t i = 0;  
        i < steps; i += 4) {  
        p = p->next;  
        p = p->next;  
        p = p->next;  
        p = p->next;  
    }  
    finish[c] = p;  
}
```

Interleaved: four pointer states

```
Node* p0 = start[0];  
Node* p1 = start[1];  
Node* p2 = start[2];  
Node* p3 = start[3];  
for (std::size_t i = 0;  
    i < steps; ++i) {  
    p0 = p0->next;  
    p1 = p1->next;  
    p2 = p2->next;  
    p3 = p3->next;  
}
```

steps = nodes per ring (a multiple of 4). Both loop bodies contain four loads. Interleaved endpoint stores are omitted here.

The same graph, the same amount of work

Four rings in one randomized node pool. Only the schedule changes.

Serial $A_0 \rightarrow A_1 \rightarrow A_2 \rightarrow \dots$ then $B_0 \rightarrow B_1 \rightarrow B_2 \rightarrow \dots$

Interleaved $A_0 \rightarrow B_0 \rightarrow C_0 \rightarrow D_0$ then $A_1 \rightarrow B_1 \rightarrow C_1 \rightarrow D_1$

Example configuration 64 MiB pool; four rings; eight passes

Work in each mode $1,048,576 \text{ nodes} \times 8 = 8,388,608 \text{ accesses}$

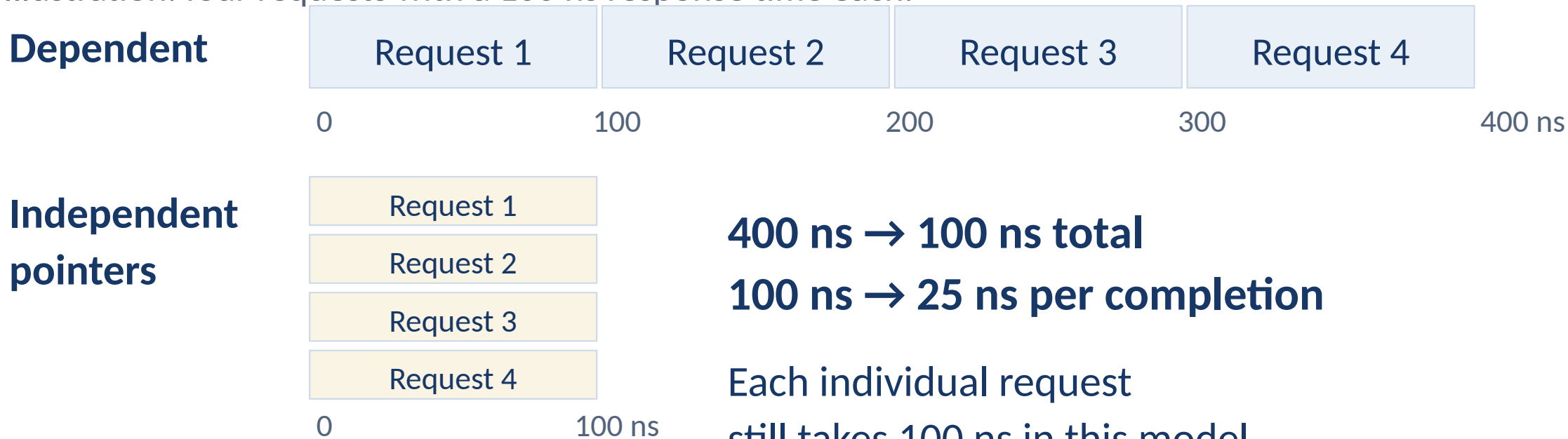
Same allocated nodes and links. One thread—not four threads.

Original experiment: four disjoint equal rings; one complete visit to every node per pass in either schedule.

Overlap changes throughput

Memory-level parallelism: overlap between memory requests.

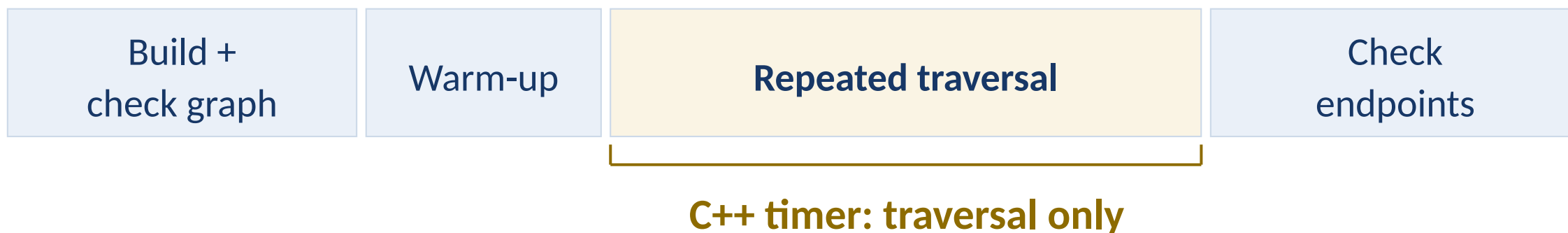
Illustration: four requests with a 100 ns response time each.



Overlap model after [1]; diagram original. Idealized 100 ns responses; no issue overhead or contention. Not measured data.

A fair measurement

Keep the graph, total work, build and CPU placement fixed.



Each mode gets its own warm-up; paired order is randomized.

perf with :u includes user-space setup, warm-up and checks.

Use separate single-mode commands for the two profiles.

[2-4] Compiler barriers preserve work but are not CPU fences. External perf collection remains whole-command.

Reading the timing comparison

Traversal time

Elapsed time for the repeated traversal.

Both modes complete the same number of link traversals.

Cost per access

Traversal time $\div$ completed accesses.

This is inverse throughput—not individual-request latency.

Throughput

Completed accesses per second.

This is not a measurement of DRAM bytes per second.

Paired speedup

Serial time $\div$ interleaved time in the same run.

A ratio above 1 means that interleaving finished faster.

Use repeated, unprofiled paired runs. Plot all observations and medians; no predetermined speedup is assumed.

Using perf to interpret the result

perf stat	Count selected events for one mode at a time. Whole-command IPC uses matching instructions and cycles.
perf report	Locate chase_serial or chase_interleaved. Check whether graph construction still dominates the profile.
perf annotate	Inspect the load dependencies in source and assembly. Serial reuses one pointer; interleaved has four independent states.

A timing improvement supports a schedule benefit; it does not establish lower DRAM latency or an exact hardware MLP value.

[1-4] Definitions and collection scope matter. Generic cache-misses does not directly measure MLP or DRAM traffic.

References

[\[1\] Chou, Fahs & Abraham — ISCA 2004](#)

Microarchitecture Optimizations for Exploiting Memory-Level Parallelism.

[\[2\] Linux perf tutorial](#)

Event counting, user-space filters and sampled profiles.

[\[3\] perf-stat\(1\) manual page](#)

Collection scope; see also perf-record(1), perf-report(1) and perf-annotate(1).

[\[4\] GCC documentation — Extended Asm](#)

Compiler barriers and their distinction from processor fences.

Teaching example: Ömer Gölcük. Original code and diagrams; full commands and documentation links accompany the lecture.

Thank you!



**Co-funded by
the European Union**



EuroHPC
Joint Undertaking



EuroCC 3 has received funding from the [European High-Performance Computing Joint Undertaking \(JU\)](#) under Grant Agreement No. 101306701. The JU receives support from the [European Union](#)'s Digital Europe Programme and Germany, Albania, Austria, Belgium, Bosnia and Herzegovina, Bulgaria, Croatia, Cyprus, Czechia, Denmark, Estonia, Finland, France, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, Montenegro, the Netherlands, North Macedonia, Norway, Poland, Portugal, Romania, Serbia, Slovakia, Slovenia, Spain, Sweden, Türkiye, and Kosovo.

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or EuroHPC Joint Undertaking. Neither the European Union nor the EuroHPC Joint Undertaking can be held responsible for them.