EURO Greece

....

OpenMP
A gentle introduction to OpenMP

Iakovos Panourgias ipanourgias@grnet.gr
GRNET
January 2026

Agenda

- Why OpenMP?
  - Simplicity and Portability
  - Avoid low-level threading (e.g., pthreads)
- Introduction
  - Shared Memory Systems
  - Threaded Programming Model
  - Thread Sharing Data Example
  - Synchronisation
  - Parallel Loops
  - Synchronisation Example
- OpenMP Basics
  - Parallel Regions
  - Data Sharing
  - Work Sharing
  - Reductions
  - Synchronisation
  - Utilities
- Summary
  - Tips and Tricks
  - Performance
  - Resources
- Wrap-up

Why OpenMP?

- Simplicity and Portability
- Avoid low-level threading (e.g., pthreads)

Simplicity and Portability

- A de-facto standard API to write shared memory parallel applications in C, C++ and Fortran
- Compiler directives
- Runtime routines
- Environment variables
- Very mature (around since 1997)
- Version 6.0 has been released (11/2024)

Simplicity and Portability

- Portable: supported by GNU, Intel, NVIDIA/PGI, SUN Studio, others
- Portable: supported on many (all important ones) Hardware platforms
- Allows incremental parallelisation (see next slide)
- You can revert to your serial application with a flick of a switch (either set OPENMP NUM THREADS to 1 or don’t link with libopenmp)
- Supports tasks
- Supports other architectures (offloading to accelerators)
- Can be used with MPI for hybrid parallelism

Incremental Approach

Parallelism added incrementally until performance goals are met:

- Start with a serial application
- Benchmark
- Identify hotspots
- Parallelise that section
- Test
- Repeat

Introduction

- OpenMP stands for Open Multi-Processing.
- It is an API for shared-memory parallel programming.
- Supports C, C++, and Fortran.

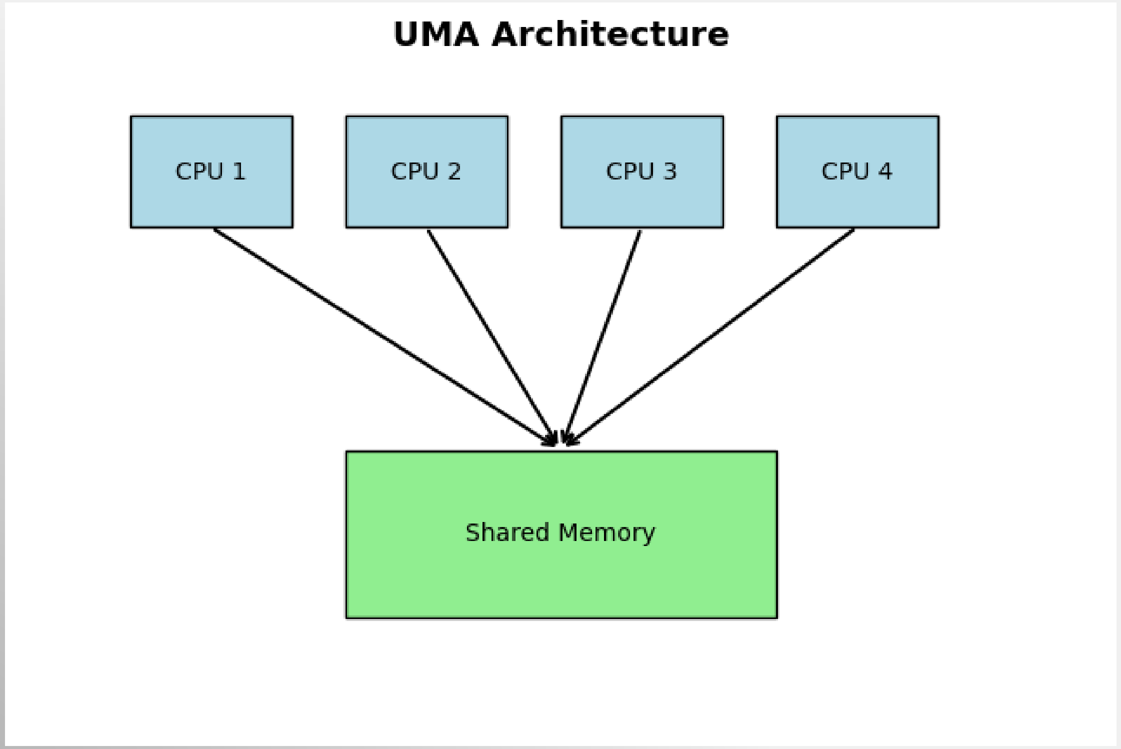
C/C++

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
{
a[i] = b[i] + c[i];
}
```

Fortran

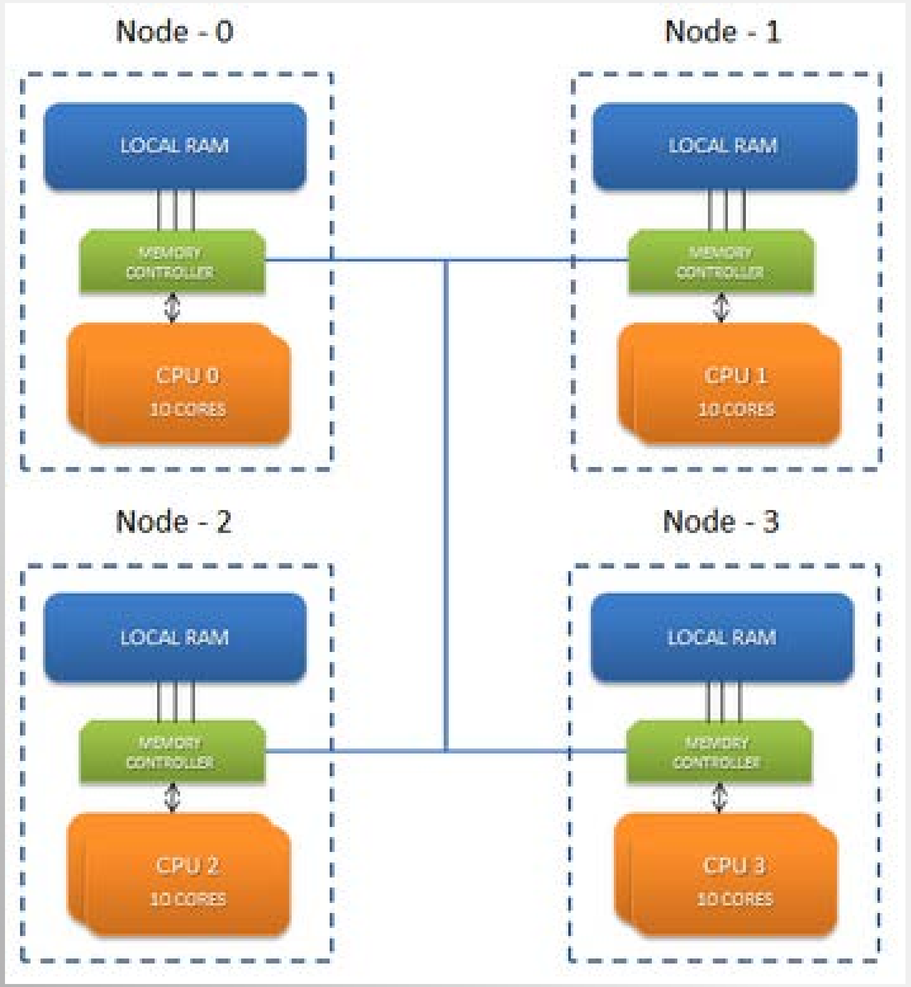
```
!$omp parallel do
do i = 1, N
   a(i) = b(i) + c(i)
end do
!$omp end
parallel do
```

Uniform Memory Access



- All CPUs share the same memory.
- Equal access time for all processors.
- Simpler programming model.
- Common in small-scale systems like desktops or basic servers.

Non-Uniform Memory Access

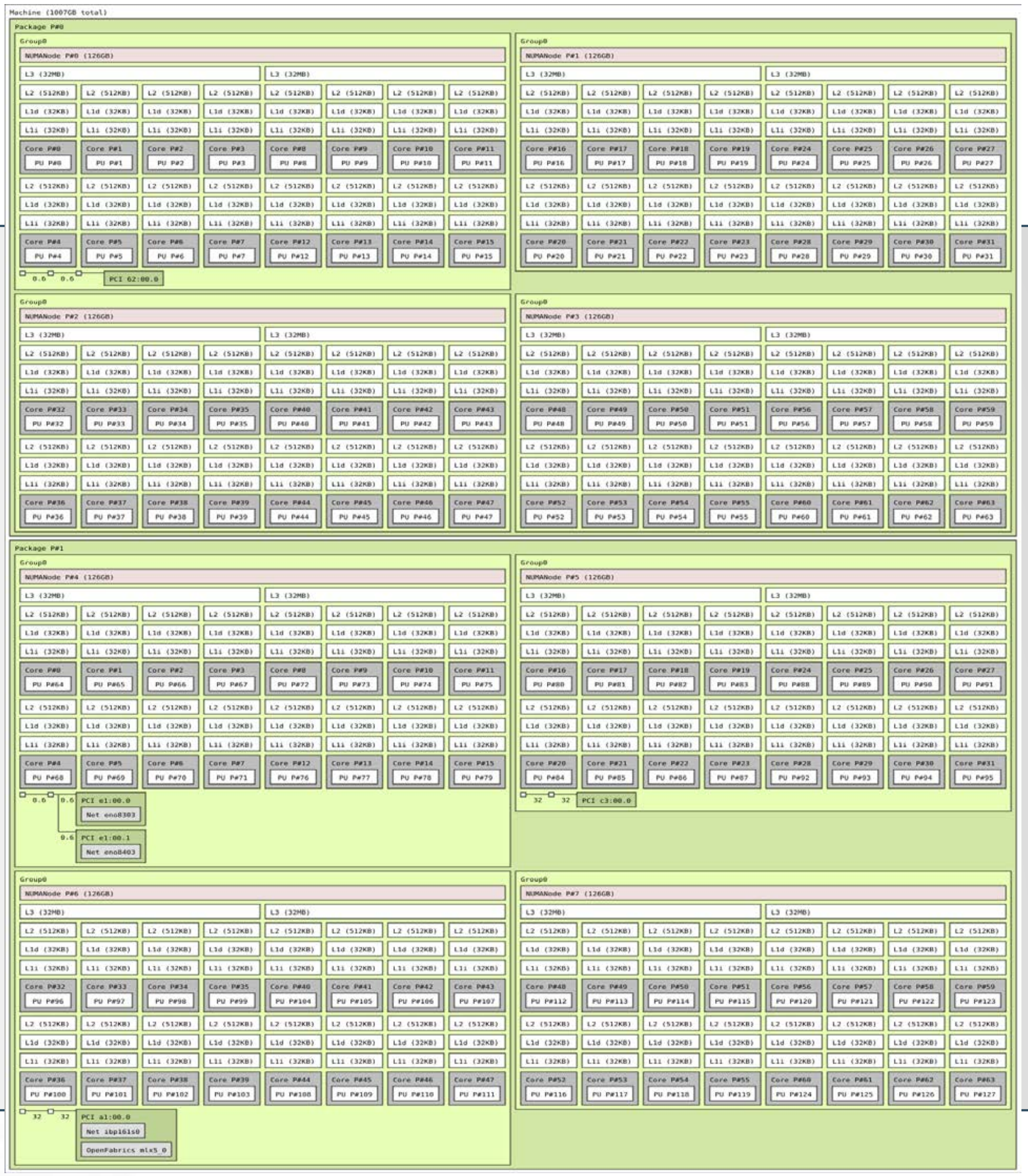


- Each processor has its own local memory, but can also access other CPUs' memory.
- Faster access to local memory, slower to remote memory.
- More complex but scales better for large systems.
- Common in modern HPC nodes and multi-socket servers.

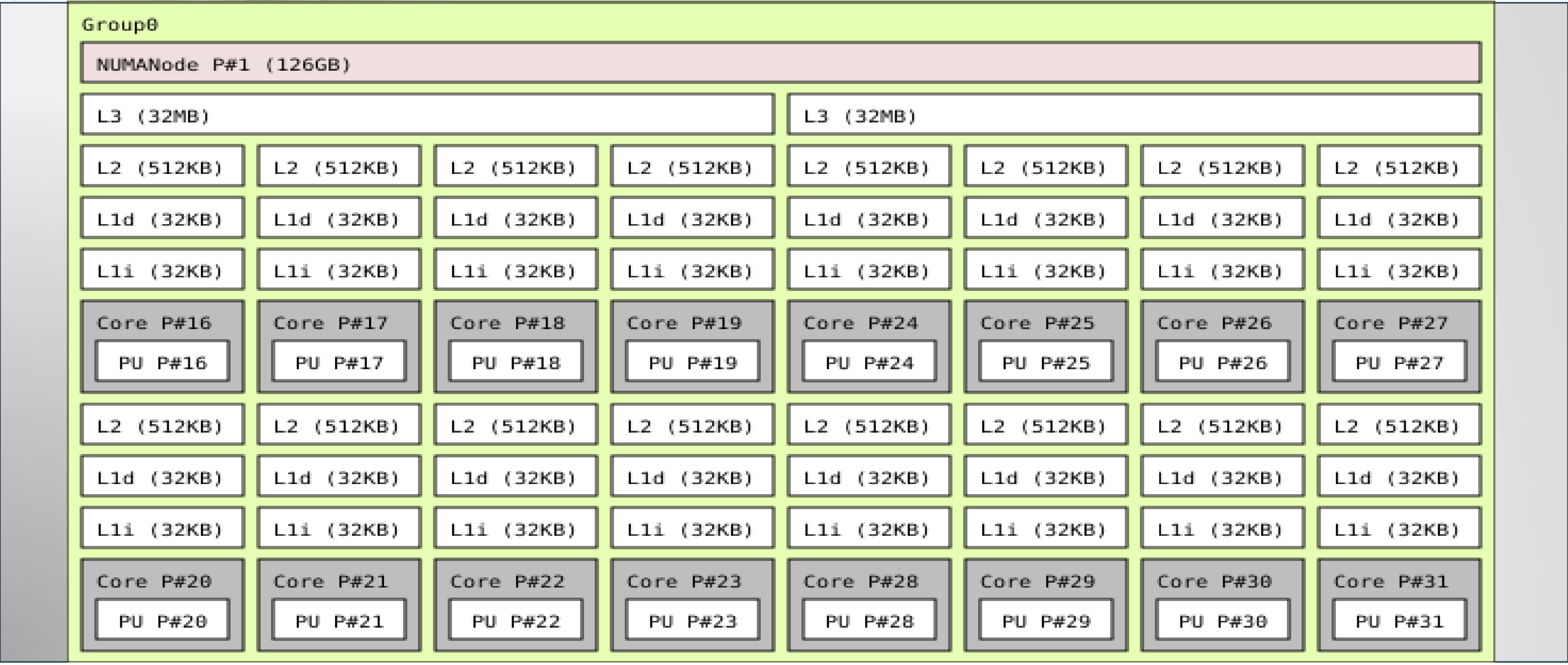
Reality

This is how a Real System looks like. (ARIS expansion)

- Multiple levels of Cache(s)
- Memory could be split/fragmented across CPUs/Sockets
- Sharing of hardware resources across CPUs/Sockets (L3 caches)



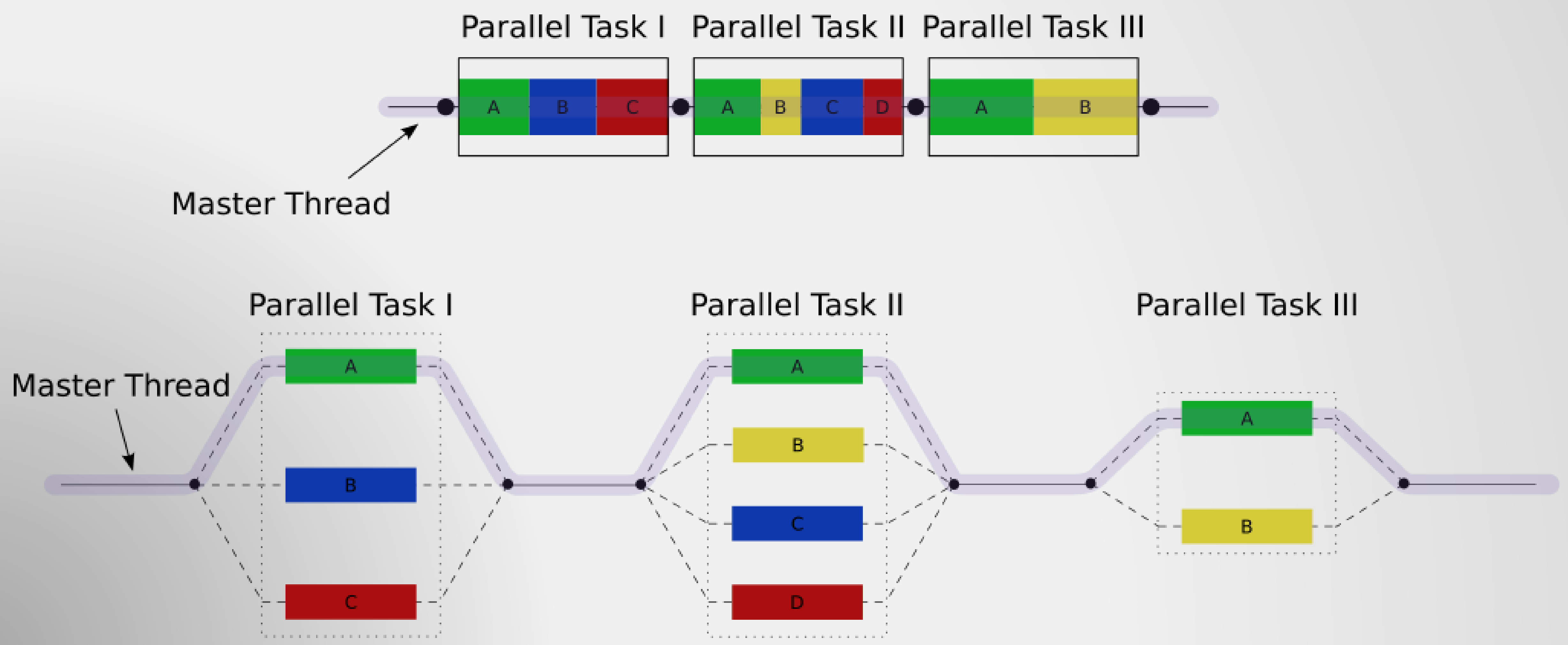
Reality / Closeup



Threaded Programming Model

- OpenMP uses threads.
- A thread is the smallest unit of processing that can be scheduled by an operating system.
- Threads are very light-weight processes; but threads can share memory with other threads.
- However, each Thread has it’s own stack and stack pointers.
- Threads are part of a process. Without a process, threads do not exist.
- Threads can access Shared Data.
- Threads can have Private Data.
- Threads are unique and have their own state (regardless of the other threads).

Fork - Join Model



Hello World C/C++

```
#include "omp.h"
#include <stdio.h>
int main ()
{
#pragma omp parallel
    {
    printf("Hello from process: %d \n", omp_get_thread_num ());
    }
}
```

```
gcc -g3 -O2 -Wall -Wextra 01.c -o 01. exe -fopenmp
```

Hello World Fortran

```
PROGRAM Parallel_Hello_World
USE OMP_LIB
!$OMP PARALLEL
PRINT *, "Hello from process: ", OMP_GET_THREAD_NUM ()
!$OMP END PARALLEL
END
```

```
gfortran -g3 -O2 -Wall -Wextra 01. f90 -o 01_f.exe -fopenmp
```

DEMO: Hello OpenMP

```
# compile
gcc -O2 -fopenmp hello_omp.c -o hello_omp


# run
./hello_omp
OMP_NUM_THREADS=8  ./hello_omp
OMP_NUM_THREADS=12 ./hello_omp
```

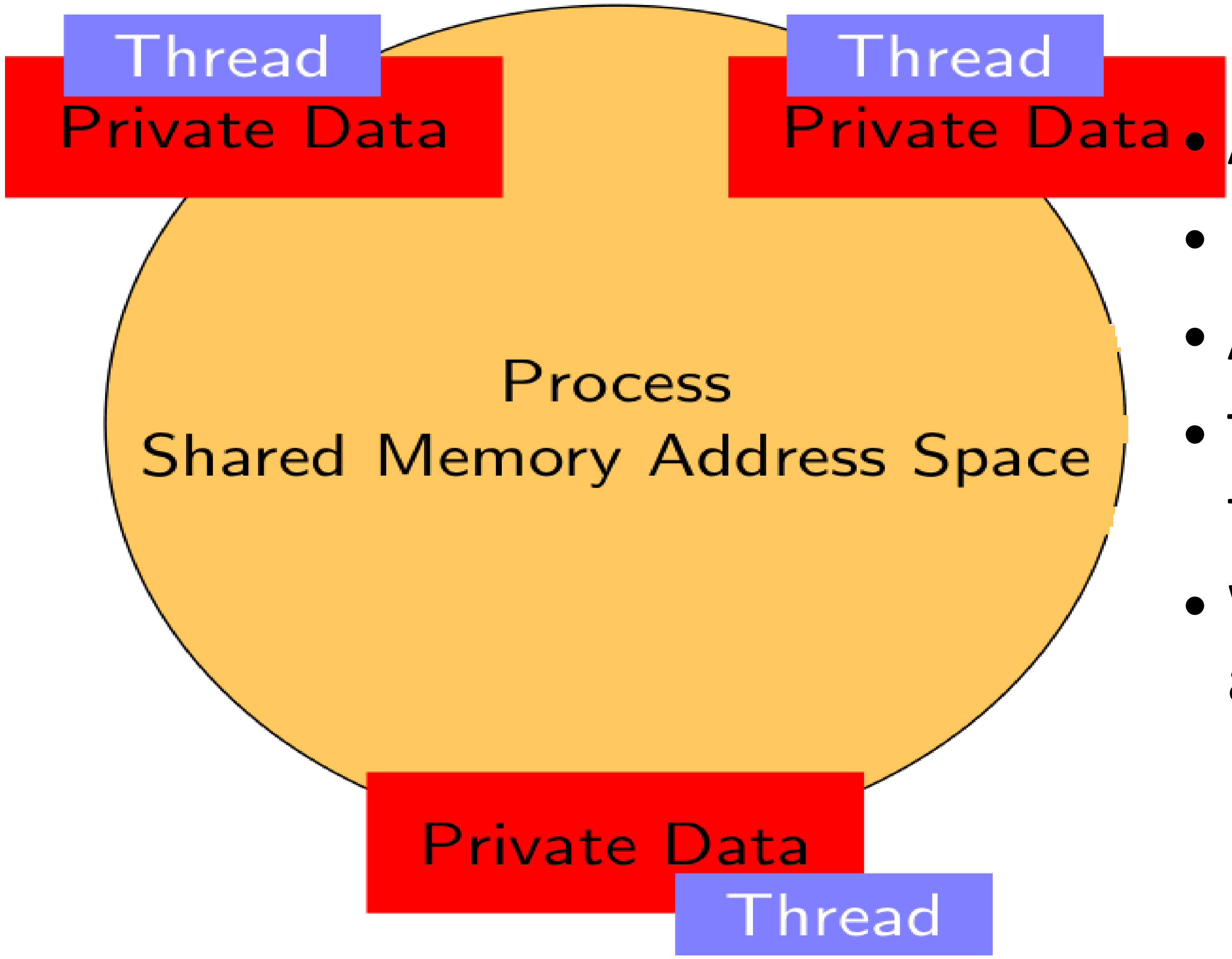
DEMO: Hello OpenMP

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

int main(void)
{
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int nt  = omp_get_num_threads();

        #pragma omp single
        printf("parallel region: num_threads=%d\n", nt);

        if (tid < 8)
            printf("  hello from thread %d\n", tid);
    }

    return 0;
}
```

Shared Data Model

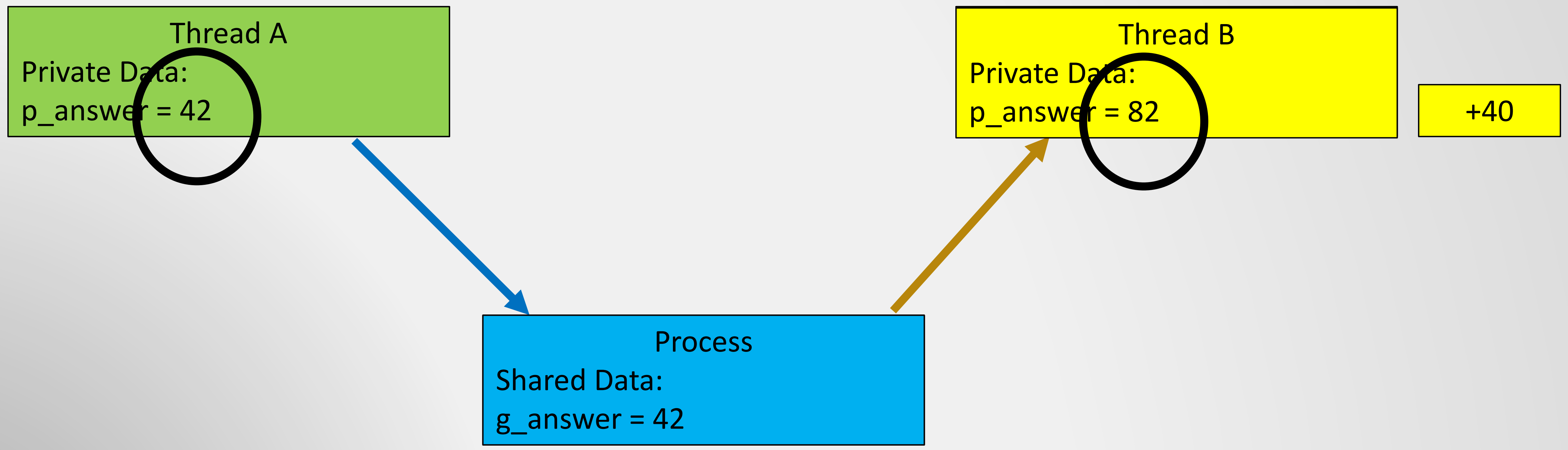
- Threads can (for non-trivial problems they **must**) exchange data.
- Shared Data constructs are used to exchange data.
  - Thread A writes to a shared variable.
  - Thread B reads the shared variable.
- No implicit synchronisation
- No implicit barriers (on entry)
- No implicit checks

Thread Sharing Data



- A single Process; with many Threads
- Each Thread has it's own Private Data
- All Threads share Shared Data
- The OS is responsible for scheduling the Process/Threads
- Without synchronisation, things fall apart

Thread Sharing Data Example



Synchronisation

Simple Example, yet:

- We don’t control the Threads (unless we block them) [**DO NOT**]
- The OS controls thread execution and scheduling
- Threads run at their own pace
- Even in this very simple example, we need to guard that Thread B doesn’t read an uninitialised value from g_answer.
- Even though it takes one line of source code to modify a value; in reality this is a multi-step process (at least three instructions are issued).
- If multiple threads try to modify a shared variable at the same time ...

Synchronisation Example

Simple Example:

- We want to add 8 to 0.
- This is a computationally expensive problem; so we will use OpenMP.
- Since we are adding 8; we will use 8 Threads.
- Each Thread will add 1.
- At the end we will print the result.
- We are expecting to dramatically improve the performance of our application.

Synchronisation Example Single Threaded (C/C++)

```
#include <stdio.h>
#define NUM_TIMES 8
int main ()
{
    int VALUE = 0;
    for (int i = 0; i < NUM_TIMES; ++i)
    {
    VALUE += 1;
    }
    printf("Result :%d \n", VALUE);
}
```

Synchronisation Example Single Threaded (Fortran)

```
PROGRAM Training
Implicit none

integer :: NUM_TIMES = 8, VALUE = 0, n

do n = 1, NUM_TIMES
        VALUE = VALUE + 1
end do
PRINT *, "Result: ", VALUE

END PROGRAM Training
```

Synchronisation Example Single Threaded

```
$ gcc -g3 -O2 -Wall -Wextra -fsanitize=address ,undefined 02.c -o 02. exe
$ ./02. exe
Result :8
```

```
$ gfortran -g3 -O2 -Wall -Wextra -fsanitize=address ,undefined 02. f90 -o
02_f.exe
$ ./02 _f.exe
Result:                    8
```

Parallel Loops

- In most applications; runtime is spent in loops.
- Loops are the first place that we will want to parallelise.
- If a loop is independent then it can be trivially parallelised.
- A loop from 0-99 can be run on one thread running all iterations; or 4 threads can run iterations 0-24/25-49/50-74/75-99 (OpenMP can use other schemes to divide iterations across threads).

Synchronisation Example Single Threaded (C/C++)

```
#include <stdio.h>
#include "omp.h"    // <=================
#define    NUM_TIMES 8
int main ()
{
     int VALUE = 0;

     #pragma omp  parallel for   // <=================
     for (int i = 0; i < NUM_TIMES; ++i)
     {
          VALUE += 1;
     }
     printf("Result :%d \n", VALUE);
}
```

Synchronisation Example Single Threaded (Fortran)

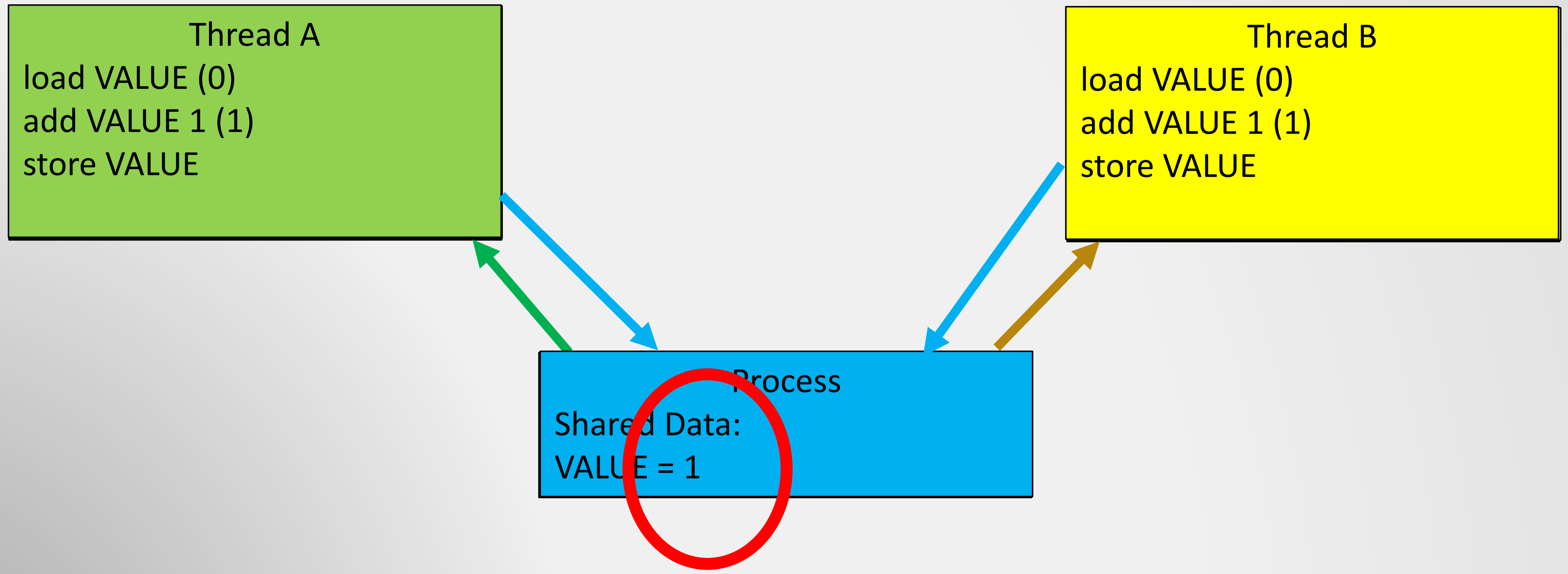
```
PROGRAM Training
USE OMP_LIB  !<=================
Implicit none
integer :: NUM_TIMES = 8, VALUE = 0, n
!$OMP PARALLEL DO !<=================
    do n = 1, NUM_TIMES
        VALUE = VALUE + 1
    end do
    PRINT *, "Result: ", VALUE
END PROGRAM Training
```

Synchronisation Example Single Threaded

```
$ gcc -g3 -O2 -Wall -Wextra -fsanitize=address ,undefined 02.c -o 02. exe -fopenmp
$ ./02. exe
Result: X


$ gfortran -g3 -O2 -Wall -Wextra -fsanitize=address , undefined 02. f90 -o 02_f.exe -fopenmp
$ ./02 _f.exe
Result:                 X
```

Synchronisation Example

Time: 0, 1, 2, 3, 4, 5, 6, 7



Race Condition

- If you run the example code (in Linux or WSL) you will see that the result is not always 8.
- Why?? Sometimes the threads run one after another. Other times they overlap.
- If they overlap and at least one (in our case all of them) writes/updates a shared value; we can have a race condition.
- Race conditions are hard to debug
- Sometimes they can be detected by run-time debuggers

DEMO: Race condition

```
# compile
gcc -O2 -fopenmp race_sum.c -o race_sum


# run
OMP_NUM_THREADS=8 ./race_sum 90000000 0
OMP_NUM_THREADS=8 ./race_sum 90000000 1
OMP_NUM_THREADS=8 ./race_sum 90000000 2


# Expected: mode 0 WRONG; mode 1 OK but slower; mode 2 OK and usually fastest.
```

DEMO: Race condition

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

int main(int argc, char **argv)
{
    long long N = (argc > 1) ? atoll(argv[1]) : 200000000LL; // 2e8
    int mode = (argc > 2) ? atoi(argv[2]) : 0;              // 0=race, 1=atomic, 2=reduction
    long long sum = 0;

    double t0 = omp_get_wtime();

    if (mode == 2) {
        #pragma omp parallel for reduction(+:sum)
        for (long long i = 0; i < N; i++) sum += 1;
    } else if (mode == 1) {
        #pragma omp parallel for
        for (long long i = 0; i < N; i++) {
            #pragma omp atomic
            sum += 1;
        }
    } else {
        #pragma omp parallel for
        for (long long i = 0; i < N; i++) sum += 1; // INTENTIONAL RACE
    }

    double t1 = omp_get_wtime();

    int threads = 1;
    #pragma omp parallel
    #pragma omp single
    threads = omp_get_num_threads();

    printf("threads=%d  N=%lld  mode=%d (0=race,1=atomic,2=reduction)\n", threads, N, mode);
    printf("sum=%lld  expected=%lld  %s\n", sum, N, (sum == N ? "OK" : "WRONG"));
    printf("time=%.3f s\n", t1 - t0);

    return 0;
}
```

OpenMP Basics

- Parallel Regions
- Data sharing & scoping: shared, private, firstprivate, lastprivate, default(shared or none)
- Worksharing: for / do, sections, schedule
- Reductions & loop shaping: reduction, collapse
- Synchronization: barrier, critical, atomic, ordered (loop order), master / masked
- Utilities: omp get num threads(), omp get thread num(), timing via omp get wtime()
- Good practices

Parallel Regions

- This one is simple. Any code inside an OMP pragma is executed by all threads
- Don’t assume that OpenMP will handle special cases for you
- If you have code that writes data to the file system inside an OMP pragma; then you will write the data OMP NUM THREAD times.
- Same for functions. A function called from inside an OMP pragma will be called OMP NUM THREAD times.

Parallel Regions

C/C++
```
#pragma omp parallel
{
   CODE
}
```

Fortran
```
!$omp parallel
  CODE
!$omp end parallel
```

C/C++
```
#pragma omp parallel
{
   FUNCTION <== NUM_THREAD
}
```

Fortran
```
!$omp parallel
  FUNCTION <== NUM_THREAD
!$omp end parallel
```

Parallel Regions

- Parallel Regions can also use clauses
- #pragma omp parallel num threads(4) : Use 4 threads
- #pragma omp parallel for if(condition) : Only start extra threads if condition is TRUE. For example, if the size of an array is less than 10000 run serially. If not, start X number of Threads

Data Sharing

The most important clauses in any Parallel Region are:

- default(shared | none)
- private(list)
- firstprivate(list)
- shared(list)

Data Sharing

Reminder about variables (single variables, arrays, etc):

- SHARED: All threads see the same copy (careful when reading/writing)
- PRIVATE: Every thread sees it’s own copy (careful when you want to exchange data).

Data Sharing

SHARED and PRIVATE variables important caveats:

- When entering any PARALLEL section, PRIVATE variables are uninitialised.
- When exiting any PARALLEL section (even when you have another one starting immediately after) private copies/variables are lost.
- Any variable declared inside a PARALLEL section is PRIVATE.
- Always, always, always use **DEFAULT(none)**. It will save you time and money.

Data Sharing

SHARED and PRIVATE variables important caveats:

- By default, global and static variables are SHARED, and loop indices are private.
- firstprivate: Like PRIVATE, but each thread’s copy is initialized with the value of the original variable at entry
- lastprivate: Like private, but after the parallel region finishes, the variable in the original context is updated with the value from the last iteration (or section) in the region

Data Sharing

```
int a, b=10;
#pragma omp parallel private(a) firstprivate(b)
{
    // 'a' is uninitialized private per thread;
    // 'b' is private per thread each initialized to 10.
}
```

```
integer :: a, b
b = 10
!$omp parallel private(a) firstprivate(b)
    ! 'a' is uninitialized privateper thread
    ! 'b' is private per thread each initialized to 10
!$omp end parallel
```

DEMO: lastprivate

```
# compile
gcc -O2 -fopenmp lastprivate_demo.c -o lastprivate_demo


# run
OMP_NUM_THREADS=8  ./lastprivate_demo 100000 0   # shared "last" (wrong / nondeterministic)
OMP_NUM_THREADS=8  ./lastprivate_demo 100000 1   # lastprivate (correct)

# Expected: mode 0 WRONG; mode 1 OK but slower; mode 2 OK and usually fastest.
```

DEMO: lastprivate

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

int main(int argc, char **argv)
{
    long long N = (argc > 1) ? atoll(argv[1]) : 10000000LL;
    int mode    = (argc > 2) ? atoi(argv[2]) : 0; // 0=shared last (race), 1=lastprivate

    long long last = -1;

    if (mode == 1) {
        #pragma omp parallel for default(none) shared(N) lastprivate(last)
        for (long long i = 0; i < N; i++) {
            last = i; // OpenMP will assign the "logical last iteration" value to last
        }
    } else {
        #pragma omp parallel for
        for (long long i = 0; i < N; i++) {
            last = i; // INTENTIONAL RACE: many threads write "last"
        }
    }

    printf("N=%lld  mode=%d (0=shared race, 1=lastprivate)\n", N, mode);
    printf("last=%lld  expected=%lld  %s\n", last, N-1, (last == N-1 ? "OK" : "WRONG / NONDETERMINISTIC"));

    return 0;
}
```

Worksharing

- Loops (for / do)
- Schedules
- Sections
- Single
- Master

Loops (for / do) [C/C++]

- Loops (for / do): Divides loop iterations among threads in a parallel region

```
int N = 1000
#pragma omp parallel for
for (int i = 0; i < N; i++)
{
            a[i] = b[i] + c[i];
}
```

Schedules

Loops (for / do) support the following schedule types:

- static: assigns fixed chunks in advance
- dynamic: hands out chunks on the fly as threads finish
- guided: starts with large chunks that shrink over time to balance workload.
- auto: the compiler/runtime decides.
- runtime: use OMP_SCHEDULE environment variable or a call to set it.
- chunk: we can define the size of the chunks.

Schedules

- If we know exactly what is going on; static is the fastest.
- If we don’t know; but we know that there is variability, use dynamic. But your data locality (if important) will disappear.
- If we know nothing; then start with guided. More overhead.
- We can also set the chunk size to help the compiler.

Loops (for / do) [C/C++]

Use a dynamic schedule. Each thread will get 4 iterations. The first thread that finishes, will get the next 4; until the end of the loop.

```
int N = 1000
#pragma omp parallel for schedule(dynamic ,4)
for (int i = 0; i < N; i++)
{
		a[i] = b[i] + c[i];
}
```

DEMO: scheduling

```
# compile
gcc -O2 -fopenmp schedule_demo.c -o schedule_demo


# run
OMP_NUM_THREADS=8  ./schedule_demo 40000000 0 0   # static, chunk=1
OMP_NUM_THREADS=8  ./schedule_demo 40000000 1 1   # dynamic, chunk=1
OMP_NUM_THREADS=8  ./schedule_demo 40000000 2 1   # guided,  chunk=1
```

DEMO: scheduling

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

static inline void burn(int iters)
{
    volatile double x = 0.0;
    for (int k = 0; k < iters; k++) x = x * 1.0000001 + 1.0;
}

int main(int argc, char **argv)
{
    long long N = (argc > 1) ? atoll(argv[1]) : 20000000LL; // many iterations
    int mode    = (argc > 2) ? atoi(argv[2]) : 0;           // 0=static,1=dynamic,2=guided
    int chunk   = (argc > 3) ? atoi(argv[3]) : 0;           // 0 means "default" for static

    omp_sched_t kind = omp_sched_static;
    const char *kname = "static";
    if (mode == 1) { kind = omp_sched_dynamic; kname = "dynamic"; }
    if (mode == 2) { kind = omp_sched_guided;  kname = "guided";  }

    // heavy tail: last 10% is heavier
    long long tail = (9LL * N) / 10LL;
    int light_iters = 40;
    int heavy_iters = 400;  // 10x heavier but still "small" so overhead differences show up

    omp_set_schedule(kind, chunk);

    double t0 = omp_get_wtime();

    #pragma omp parallel for schedule(runtime)
    for (long long i = 0; i < N; i++) {
        burn(i < tail ? light_iters : heavy_iters);
    }

    double t1 = omp_get_wtime();

    int threads = 1;
    #pragma omp parallel
    #pragma omp single
    threads = omp_get_num_threads();

    printf("threads=%d  N=%lld  schedule=%s  chunk=%d  tail=last10%%  time=%.3f s\n",
           threads, N, kname, chunk, t1 - t0);

    return 0;
}
```

Sections

- Not all code runs in loops.
- Allows different code blocks to run in parallel.
- Each section block is executed by one thread.
- When all sections complete, threads synchronize (unless nowait is used).

Sections [C/C++]

The two functions will run on two different threads. There is a block at the end of the parallel section; waiting for both threads to finish.

```
#pragma omp parallel sections
{
        #pragma omp section something_interesting_A ();

        #pragma omp section something_interesting_B ();
}
```

Sections

- Pure Sections only allow running 1 Thread per section.
- If somethinginterestingA() needs 10x more CPU; then you are out of luck
- (until the advanced Session which talks about Tasks and Nested Parallelism)

Single

- Single: Ensures a block is executed by only one thread (unspecified which)
- All other threads wait at an implicit barrier at the end of the single region
- Unless nowait is specified.
- You can still use PRIVATE and FIRSTPRIVATE clauses.

Single

```
#pragma omp single
{
        something_only_for_one ();
}
```

```
!$omp single
call something_only_for_one ()
!$omp end single
```

Master

- The Master directive specifies that the enclosed code block is executed only by the master thread (thread 0).
- Unlike single, it has no implicit barrier at entry or exit, so other threads skip it and continue without waiting.
- It’s often used for setup, I/O, or coordination work that must be done once without pausing the other threads.

Reduction operations

- (various) Reduction operations
- Collapse

Reduction operations

- Performs a parallel reduction on a variable across threads.
- Each thread gets a private copy of var
- Updates it with
- At the end all copies are combined (reduced) into a single result.
- Common operators include +, *, max:, min:, etc.
- Why use a reduction?? **Performance!**

Reduction operations [C/C++]

```
int sum = 0;
#pragma omp parallel for reduction (+: sum)
for(int i=0; i<N; i++)
{
        sum += data[i];
}
```

Collapse

- When you use an omp parallel do/for you only parallelise the outer loop.
- With COLLAPSE we can tell the OpenMP library to treat the next 2, 3, 4 loops as one big loop and divide it to the available threads.

Collapse [C/C++]

In this example OpenMP will create 3 threads to parallelise the outer loop. This is obviously inefficient (in a machine with 128 cores; 125 will sit IDLE).

```
int i, j;
#pragma omp parallel for
for (i = 1; i <= 3; i++) {
        for (j = 1; j <= 400; j++) {
                printf("Thread %d: i=%d, j=%d\n", omp_get_thread_num (), i, j);
        }
}
```

Collapse [C/C++]

In this example OpenMP will divide the 1200 iterations to all available threads. Better usage of the 128 cores.

```
int i, j;
#pragma omp parallel for collapse (2)
for (i = 1; i <= 3; i++) {
        for (j = 1; j <= 400; j++) {
                printf("Thread %d: i=%d, j=%d\n", omp_get_thread_num (), i, j);
        }
}
```

DEMO: collapse

```
# compile
gcc -O2 -fopenmp collapse_demo.c -o collapse_demo


# run
OMP_NUM_THREADS=8  ./collapse_demo 4 8000000000 0   # no collapse
OMP_NUM_THREADS=8  ./collapse_demo 4 8000000000 1   # collapse(2)

# mode 1 (collapse) uses threads better and is faster when outer loop is small.
```

DEMO: collapse

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

static inline void burn(long long iters)
{
    volatile double x = 0.0;
    for (long long k = 0; k < iters; k++) x = x * 1.0000001 + 1.0;
}

int main(int argc, char **argv)
{
    int I = (argc > 1) ? atoi(argv[1]) : 4;            // small outer loop
    long long J = (argc > 2) ? atoll(argv[2]) : 200000000LL; // big inner work
    int mode = (argc > 3) ? atoi(argv[3]) : 0;          // 0=no collapse, 1=collapse(2)

    double t0 = omp_get_wtime();

    if (mode == 1) {
        #pragma omp parallel for collapse(2) schedule(static)
        for (int i = 0; i < I; i++) {
            for (long long j = 0; j < J; j++) {
                if ((j & 0x3FFFFF) == 0) burn(2000); // sparse burn to keep runtime reasonable
            }
        }
    } else {
        #pragma omp parallel for schedule(static)
        for (int i = 0; i < I; i++) {
            for (long long j = 0; j < J; j++) {
                if ((j & 0x3FFFFF) == 0) burn(2000);
            }
        }
    }

    double t1 = omp_get_wtime();

    int threads = 1;
    #pragma omp parallel
    #pragma omp single
    threads = omp_get_num_threads();

    printf("threads=%d  I=%d  J=%lld  mode=%d (0=no collapse,1=collapse)  time=%.3f s\n",
           threads, I, J, mode, t1 - t0);

    return 0;
}
```

Synchronisation

- barrier: makes all threads in a team wait until every thread has reached the barrier.
- critical: ensures the enclosed code block is executed by only one thread at a time.
- atomic: makes a simple memory update (e.g. count++) atomic at the hardware level.
- lock: creates user managed lock(s)
- ordered: Enforces the enclosed block to execute in sequential loop order.
- master: Specifies that only the master thread (thread 0) of the team executes the block.
- masked (OpenMP 5.0+): Runs a block on a subset of threads. Without a filter, it behaves like master (only thread 0 executes).

Barrier

#pragma omp barrier

- All threads stop at this point until every thread in the team reaches it.
- It ensures that no thread proceeds past the barrier before the others
- have caught up.
- Useful for synchronizing phases of work across threads.

Barrier [C/C++]

The second function will only be called once all threads have finished executing the first one.

```
#pragma omp parallel
{
        calculate_critical_mass ();

        #pragma omp barrier
        simulate_explosion ();
}
```

Critical

#pragma omp critical [(name)]

- Only one thread at a time can execute the block of code marked critical.
- Threads wait their turn to enter, preventing race conditions on shared data.
- This is straightforward but can become a bottleneck if the protected code is slow.
- Use this to protect updates to shared data when atomic cannot be used.

Critical [C/C++]

Sum will be updated correctly. DO NOT DO THIS. USE A REDUCTION!!

```
#pragma omp parallel for
for ( int i = 0; i < Ni; i++ ) {
        #pragma omp critical
        sum += array[i];
}
```

Atomic

#pragma omp atomic

- Protects a single memory update so it happens without interference from other threads.
- It’s lighter weight than critical because it’s limited to simple operations (e.g., increments, sums).
- Ideal for fine-grained synchronization on shared variables

Atomic [C/C++]

Sum will be updated correctly. DO NOT DO THIS. USE A REDUCTION!!

```
#pragma omp parallel for
for ( int i = 0; i < Ni; i++ ) {
        #pragma omp atomic
        sum += array[i];
}
```

Critical vs Atomic

- Atomic uses hardware instructions
- Atomic does not use lock/unlock on entering/exiting the line of code
- Lower overhead

Lock

- You explicitly create and control a lock object with omp init lock(), omp set lock(), and omp unset lock().
- Locks give you more control over when and where mutual exclusion happens.
- They can protect complex or multiple code regions, but require careful pairing of set/unset to avoid deadlocks.
- Same functionality as a mutex/semaphore.

Lock [C/C++]

```
omp_lock_t myLock;
(void) omp_init_lock (& myLock);
#pragma omp parallel
{
    (void) omp_set_lock (& myLock); // acquire lock
    important_function ();
    (void) omp_unset_lock (& myLock); // release lock
} // End of parallel region
(void) omp_destroy_lock (& myLock);
```

Reality Check: Locks, Critical, Atomic

- Critical sections serialise execution. They **destroy** scalability.
- Locks serialise execution. They **destroy** scalability.
- Atomic uses hardware instructions: CAS instructions.
- CAS instructions still need to fetch data (sometimes from another NUMA node)
- CAS: Compare-and-swap
- Fetching needs time.
- Atomic sooner (or later) **destroy** scalability.
- It’s better to re-think/re-write the algorithm to allow **parallelisation**.

DEMO: critical/atomic/locks

```
# compile
gcc -O2 -fopenmp compound_update.c -o compound_update

# run
OMP_NUM_THREADS=8  ./compound_update 1000 4000000 0 # wrong
OMP_NUM_THREADS=8  ./compound_update 1000 4000000 1 # wrong (atomic can't protect two ops)
OMP_NUM_THREADS=8  ./compound_update 1000 4000000 2 # correct
OMP_NUM_THREADS=8  ./compound_update 1000 4000000 3 # correct

# 0=race, 1=atomics-only (still wrong), 2=critical, 3=lock
```

DEMO: critical/atomic/locks

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

static inline void tiny_delay(void)
{
    // Encourage interleaving; cheap and portable
    #pragma omp flush
}

int main(int argc, char **argv)
{
    long long tokens0 = (argc > 1) ? atoll(argv[1]) : 1000;     // initial tokens
    long long tries   = (argc > 2) ? atoll(argv[2]) : 200000;   // attempts per thread
    int mode          = (argc > 3) ? atoi(argv[3]) : 0;
    // mode: 0=race, 1=atomics-only (still wrong), 2=critical, 3=lock

    volatile long long tokens = tokens0;
    volatile long long winners = 0;

    omp_lock_t lock;
    if (mode == 3) omp_init_lock(&lock);

    int T = 1;
    #pragma omp parallel
    #pragma omp single
    T = omp_get_num_threads();

    double t0 = omp_get_wtime();

    // Start together to maximize contention
    #pragma omp parallel
    {
        #pragma omp barrier

        for (long long i = 0; i < tries; i++) {

            if (mode == 2) {
                #pragma omp critical
                {
                    if (tokens > 0) { tokens--; winners++; }
                }
            }
            else if (mode == 3) {
                omp_set_lock(&lock);
                if (tokens > 0) { tokens--; winners++; }
                omp_unset_lock(&lock);
            }
            else if (mode == 1) {
                // Atomics on the pieces do NOT make the whole "check-then-act" atomic.
                long long t;
                #pragma omp atomic read
                t = tokens;

                if (t > 0) {
                    tiny_delay(); // widen race window

                    #pragma omp atomic update
                    tokens--;

                    #pragma omp atomic update
                    winners++;
                }
            }
            else {
                // mode 0: pure race
                if (tokens > 0) {
                    tiny_delay();
                    tokens--;
                    winners++;
                }
            }
        }
    }

    double t1 = omp_get_wtime();

    if (mode == 3) omp_destroy_lock(&lock);

    const char *name =
        (mode==0) ? "race" :
        (mode==1) ? "atomics-only (NOT atomic transaction)" :
        (mode==2) ? "critical" : "lock";

    printf("threads=%d  tokens0=%lld  tries/thread=%lld  mode=%d (%s)\n",
           T, tokens0, tries, mode, name);

    printf("tokens=%lld  winners=%lld  (expect: winners <= tokens0 and tokens >= 0)\n",
           (long long)tokens, (long long)winners);

    if (winners > tokens0 || tokens < 0)
        printf("RESULT: WRONG (compound check-then-act broke)\n");
    else
        printf("RESULT: OK\n");
    printf("time=%.3f s\n", t1 - t0);

    return 0;
}
```

Ordered

- For use inside a parallel for, it forces the enclosed code to run in loop-iteration order.
- Only one thread executes the ordered block at a time, and in the correct sequence.
- Handy when most work is parallel but a small part must happen in strict order.

Ordered [C/C++]

```
#pragma omp for ordered
for (i=0; i<n; i++) {
    #pragma omp ordered
    work(i);
}
```

Master

#pragma omp master

- The enclosed code is executed only by the master thread (thread 0).
- Unlike single, it has no implicit barrier before or after, so other threads skip it and keep going.
- Useful for setup, teardown, or I/O handled by a single designated thread.
- Or for MPI operations. Usually, the MPI library requires the Master thread to make MPI calls.

Masked

#pragma omp masked

- Similar to master, but lets you specify which thread(s) should execute the region via a filter expression.
- For example, filter(omp get thread num()==1) would make only thread 1 run the block.
- Other threads skip the block and can continue without waiting.
- Gives more flexibility than master for designating work to a specific thread in the team

Utilities

- omp_get_num_threads() : How many threads we are using
- Careful; will always return 1 if run **outside** a parallel section
- omp_get_thread_num() : Returns our Thread number

Summary

- Tips and Tricks
- Performance
- Resources

Tips and Tricks

- Creating and starting a Parallel section takes time.
- The optimised code must be worth it (it must runs for at least for a couple of seconds).
- If the optimised code doesn’t always run long enough all the time; use the IF clause.
- NOWAIT can help; however, it can cause also cause race conditions.
- CHUNKSIZE is important. It should be tuned. It can be changed at RUNTIME (which is always helpful).

Tips and Tricks

- MASTER, SINGLE and MASKED are useful. However, MASTER is the one with lowest overhead. SINGLE and MASKED require some synchronisation.
- However, if you don’t know beforehand which THREAD will reach the directive first; best to use SINGLE. Most of the time this will be quicker than waiting for the MASTER thread to arrive.

Tips and Tricks

- **Never**, **ever**, **ever** use default(shared).
- **Never, ever, ever** have a parallel section without a default (most implementations default to shared).

Tips and Tricks

- I found the section of my code that takes up most of the runtime ...
- ... but it’s a FOR/DO loop with hundreds lines of code!!!
- How do I set the PRIVATE and SHARED variables??
- You said to never use default(shared)!!

Tips and Tricks

- You need to put that code into a function.
- Use the SCOPE attributes of C/C++ and Fortran and pass everything as an argument to the function.
- Everything else can be a local variable inside the function.
- Test.
- If everything works; start adding PARALLEL sections.

Tips and Tricks

- Sometimes; you will need to refactor your code. Either because it’s on the verge of the standards or because it has already fallen off and it compiles/works with a specific compiler and runs on a specific hardware.

Tips and Tricks

Handy and useful environment variables:

- OMP_WAIT_POLICY=active : Tell the OpenMP runtime to spin IDLE threads; instead of putting them to sleep.
- OMP_DYNAMIC=false : Tell the OpenMP runtime to allocate exactly the number of threads you asked for.
- OMP_PROC_BIND=true : Stop threads from migrating and destroying data locality.

Tips and Tricks

- Modern debuggers support OpenMP (gdb, DDT, TotalView).
- For Race Conditions you need other tools.
- SUN Studio, Intel Inspector (or the Intel Studio), Valgrind DRD, Clang ThreadSanitizer

Tips and Tricks

Timers:

- Don’t use clock() or other system timers. They don’t work well with OpenMP.
- ”srun time myapplication.exe” will give you the correct overall duration of your application.
- Use omp_get_time(). It works perfectly with OpenMP.

Performance

I’ve gone through your excellent training and optimised my code using OpenMP. I got a 2% speedup!!

- Sequential Code
- Synchronisation
- Scheduling / Idle Threads
- Communication
- Hardware resources

Performance: Sequential Code

- Anything that is not parallelised (outside PARALLEL section, inside MASTER, SINGLE sections) runs sequentially.
- Amdahl's law states that the maximum speedup is limited by the unparallelised code.
- If exactly 50% of the work can be parallelized, the best possible speedup is 2 times.
- If 95% of the work can be parallelized, the best possible speedup is 20 times.

Performance: Sequential Code

Solutions:

- Try to parallelise everything :-)
- If you can’t, then understand the limitations of your code.

Performance: Synchronisation

- Whenever we synchronise; there is implicit communication between the Threads.
- Communication uses hardware resources (next slides); uses CPU time; sometimes destroys data locality.
- CRITICAL, ATOMIC, LOCKS are points where we lose performance.

Performance: Synchronisation

Solutions:

- Minimise barriers. You may need to refactor your code.
- Use NOWAIT; but be careful.
- Use ATOMIC rather than CRITICAL or LOCKS. But know that every
- time you use it; you lose performance.

Performance: Scheduling / Idle Threads

- More often than not (always); some threads finish first and wait for the others.
- When they wait; they are IDLE. They are not doing anything helpful.
- When multiple Threads reach a CRITICAL section; they start to wait.
- Even worse, they have to talk to each other to pass through the CRITICAL section one at a time.
- Not only they wait and do nothing; they are actually using Hardware Resource to talk to each other!!!

Performance: Scheduling / Idle Threads

The wrong scheduling type or chunksize can have very bad effects.

```
#pragma omp parallel for schedule(dynamic , 1)
for(int i=0; i < 50000000; i++) {
            CODE
}
```

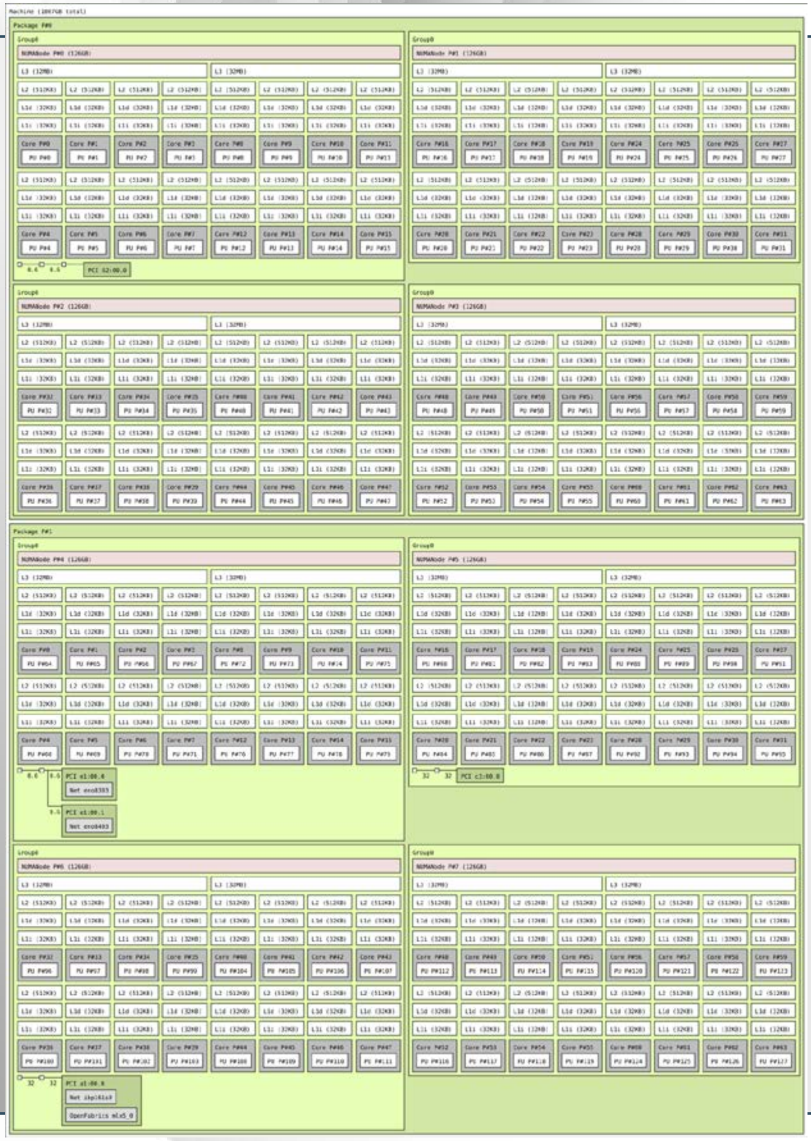
The OpenMP runtime will not be very happy about our choice of scheduler and chunksize.

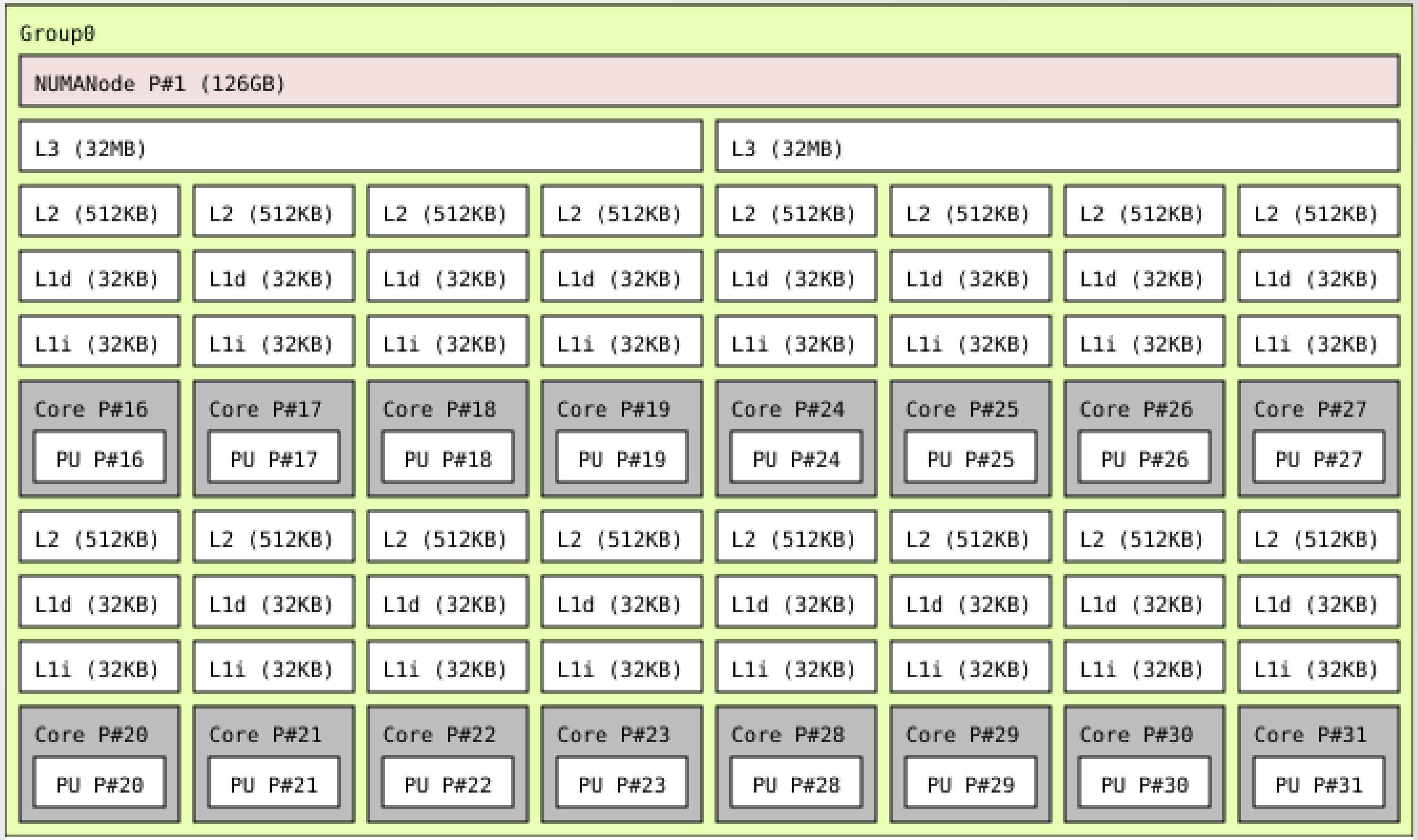
Performance: Scheduling / Idle Threads

Solutions:

- Use sensible SCHEDULE parameters. Experiment with different schedulers.
- Use sensible CHUNKSIZE values. Experiment with different values.
- Too big CHUNKSIZE and you risk some THREADS finishing early and wait.
- Too small CHUNKSIZE and you will cause scheduling overheads and synchronisation bottlenecks in the OpenMP runtime.
- Ideally, we want all threads to finish at the same time.

Performance: Communication





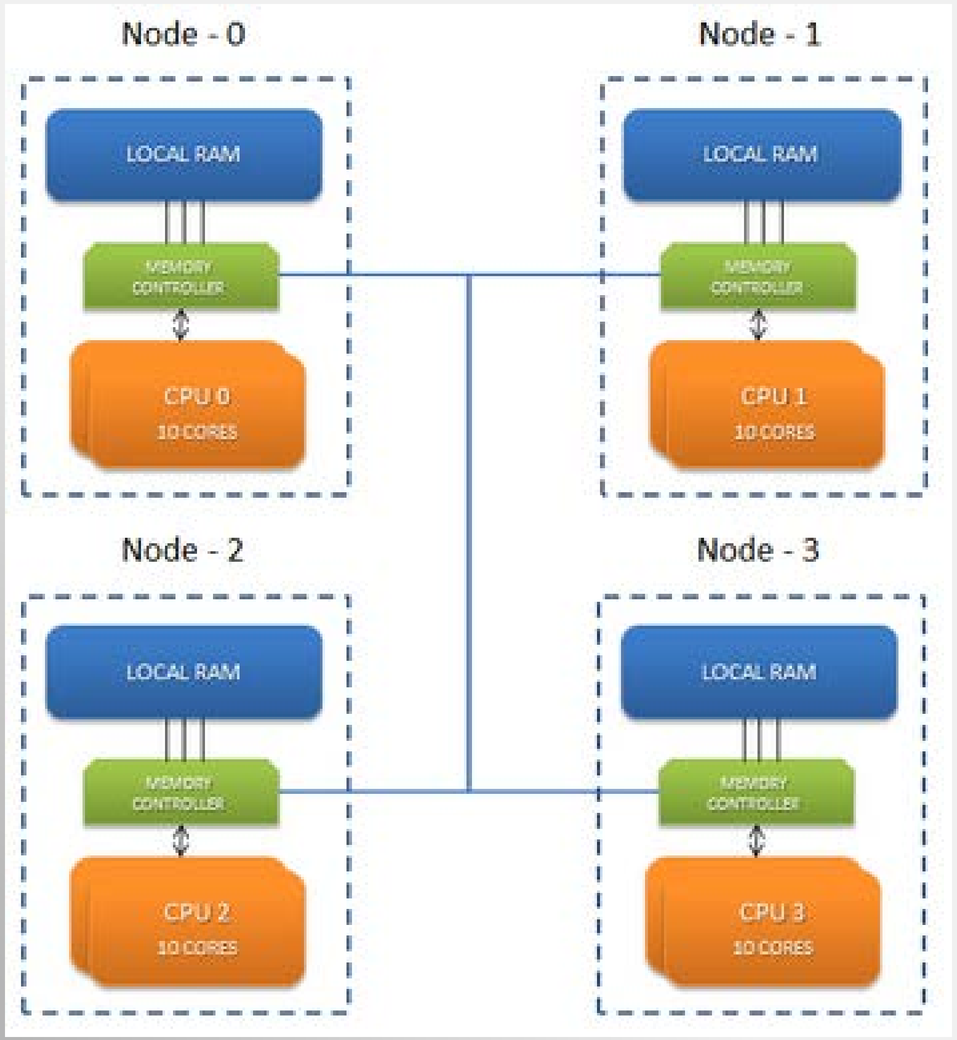
Performance: Communication

- Accessing DATA from Memory is really slow (compared to **CACHE**).
- Really slow means hundreds to thousand times slower.
- If we need to access memory from a remote NUMA node; that’s even worse.
- And that’s only about reading.
- Every READ (and WRITE) operation goes through the Cache Coherency Mechanism.

Performance: Communication

- When we write (modify) data; things become really expensive. When a THREAD (running on CPU#1) writes some data:
- The Cache Coherency fetches the data (from wherever) and brings it to the local CACHE (L3, L2 or L1).
- **All** other copies of that memory are invalidated on all the other CACHEs of all the other CPUs (to avoid reading stale data).
- If another Thread on another CPU wants to read the same data; the Cache Coherency Mechanism needs to fetch and propagate that Memory location to the local CACHE (L3, L2 or L1) of that CPU.
- Imagine if our Threads write to some memory that other Threads need to read. Say goodbye to any performance improvements.

Performance: Cache Coherency Mechanism



Performance: Communication

Solutions (kind of):

- We need to use Data Affinity!
- This means that we should (as much as possible) access (READ/WRITE) the same data with the same THREAD
- Try to use large contiguous memory areas (don’t update single INT8)
- We can then use CACHEd data (ten to hundred times faster than local memory)
- Use OMP_PROC_BIND=true to help pin the threads.

Performance: Communication

More problems with NUMA (most systems):

- Most Operating Systems employ the first touch policy.
- The first touch policy essentially tells the Memory Subsystem to place memory closest to the THREAD (or PROCESS) that asked for the memory.
- That makes sense for sequential applications. The memory is now in the local NUMA node and hopefully in the CACHE as well.
- It doesn’t work for parallel applications.
- Especially if initialisation is done by the MASTER thread. Everything is now in the MASTER thread NUMA node; and everyone else have to wait for the data to trickle across.

Performance: Communication

Solutions (kind of):

- For OpenMP (and parallel applications in general); it’s better to initialise in a parallel section.
- Try to keep the data local. Try not to change the loop iteration forcing data to migrate to other CPUs (or even worse other NUMA nodes).
- Use numactl (on Linux) to change the NUMA policy.

Performance: Communication

More problems:

- Memory is written in 4KB and 16KB pages.
- Some (most) systems use huge pages (2MB, 4MB, 1GB).
- Operating Systems like larger pages because they have to handle less pages overall.
- Less TLB (Translation Lookaside Buffer) misses, mean higher memory access.
- However, if our stride is more than the page size; we will go through the huge pages like a knife through butter.

Performance: Communication

We write an int8 at i*2MB+1byte :

- The TLB updates the mapping of the virtual page to the physical memory frame.
- The CPU fetches the 64 bytes cache line containing the target byte into L1 (read-for-ownership), modifies 1 byte, and later evicts the dirty line.
- For a 1 byte update; we get a 64 bytes read traffic and 64 bytes write traffic (assuming that each CACHE line size is 64 bytes).
- If the same CACHE line was in any other CPU; it needs to get invalidated.
- Since we never use this line again; we don’t get the benefits of the CACHE system.
- Soon we start incurring extra costs; because the TLB misses will start to increase.
- Imagine this scenario with tens or hundreds of THREADs running at the same time.

Performance: Communication

- This problem even has a name: False Sharing.
- We discussed how CACHEs, CACHE line size Memory and the Cache Coherency Mechanism work.
- If we have THREADs updating a single int8, the whole CACHE line get’s invalidated.
- What if we have a SHARED array of int8 counters and each THREAD updates it’s own index.

```
counter_array[ omp_get_thread_num () ]++;
```

Performance: Communication

- The previous code looks innocent enough.
- However, each update of an index; will invalidate the surrounding 64 bytes.
- The surrounding 64 bytes are 64 entries which will need to start their journey around the NUMA node(s).
- Invalidating all copies. And fetching the new value from the THREAD that did the original update.
- The next THREAD comes along and updates it’s copy (which has been just fetched from who knows where).
- Once the update takes place, all the other copies will get invalidated again.
- Say a big goodbye to your memory bandwidth and to performance.

DEMO: false sharing

```
# compile
gcc -O2 -fopenmp false_sharing.c -o false_sharing

# run
OMP_NUM_THREADS=8  ./false_sharing 800000000 0
OMP_NUM_THREADS=8  ./false_sharing 800000000 1

# 0=false sharing (stride=1), 1=no false sharing (stride=16)
```

DEMO: false sharing

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

int main(int argc, char **argv)
{
    long long iters = (argc > 1) ? atoll(argv[1]) : 400000000LL;
    int mode        = (argc > 2) ? atoi(argv[2]) : 0; // 0=stride1, 1=padded

    int T = omp_get_max_threads();
    int stride = (mode == 1) ? 16 : 1; // 16*8=128B spacing -> usually avoids same cache line

    long long *base = (long long*)calloc((size_t)T * (size_t)stride, sizeof(long long));
    if (!base) { perror("calloc"); return 1; }

    // Treat as volatile to force memory traffic (prevents "register-only" optimization)
    volatile long long *a = (volatile long long*)base;

    double t0 = omp_get_wtime();

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        volatile long long *p = &a[tid * stride];

        for (long long i = 0; i < iters; i++) {
            (*p)++;
        }
    }

    double t1 = omp_get_wtime();

    long long total = 0;
    for (int t = 0; t < T; t++) total += base[t * stride];

    printf("threads=%d  iters=%lld  mode=%d (0=stride1,1=padded stride=%d)  total=%lld  time=%.3f s\n",
           T, iters, mode, stride, total, t1 - t0);

    free(base);
    return 0;
}
```

Performance: Hardware resources

Memory bandwidth / Caches:

- As we have seen; memory bandwidth (which is limited) can be exhausted really easily.
- False sharing.
- Not using Data Locality.
- Not using Data Affinity.
- Most systems share a level of CACHE (usually L3)
- Using multithreaded applications, CPUs fight for the same resource. Sometimes, using the whole resource for themselves.

Performance: Hardware resources

- CPU instructions and instructions CACHEs are not infinite.
- For compute intensive codes; they can run out and THREADs have to wait.
- On the other hand, when THREADs wait for memory resources; even compute intensive codes can run happily.
- Don't oversubscribe (unless you are testing).
- Using more THREADs than COREs destroys data locality and is really slow.
- Using more THREADs than COREs is a nice test for your implementation. Sometimes, it makes hard to detect race conditions more prominent.

Resources

- OpenMP: https://openmp.org
- OpenMP: https://www.openmp.org/resources/openmp-presentations/
- Various HPC centers and documentation (EPCC, HLRS, LLNL, others)

EURO Greece

Thanks!

Co-funded by the European Union

EuroHPC Joint Undertaking

Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Türkiye, Republic of North Macedonia, Iceland, Montenegro, Serbia under grant agreement No 101101903.