NTNU – Trondheim
Norwegian University of
Science and Technology

Atomicity in OpenMP

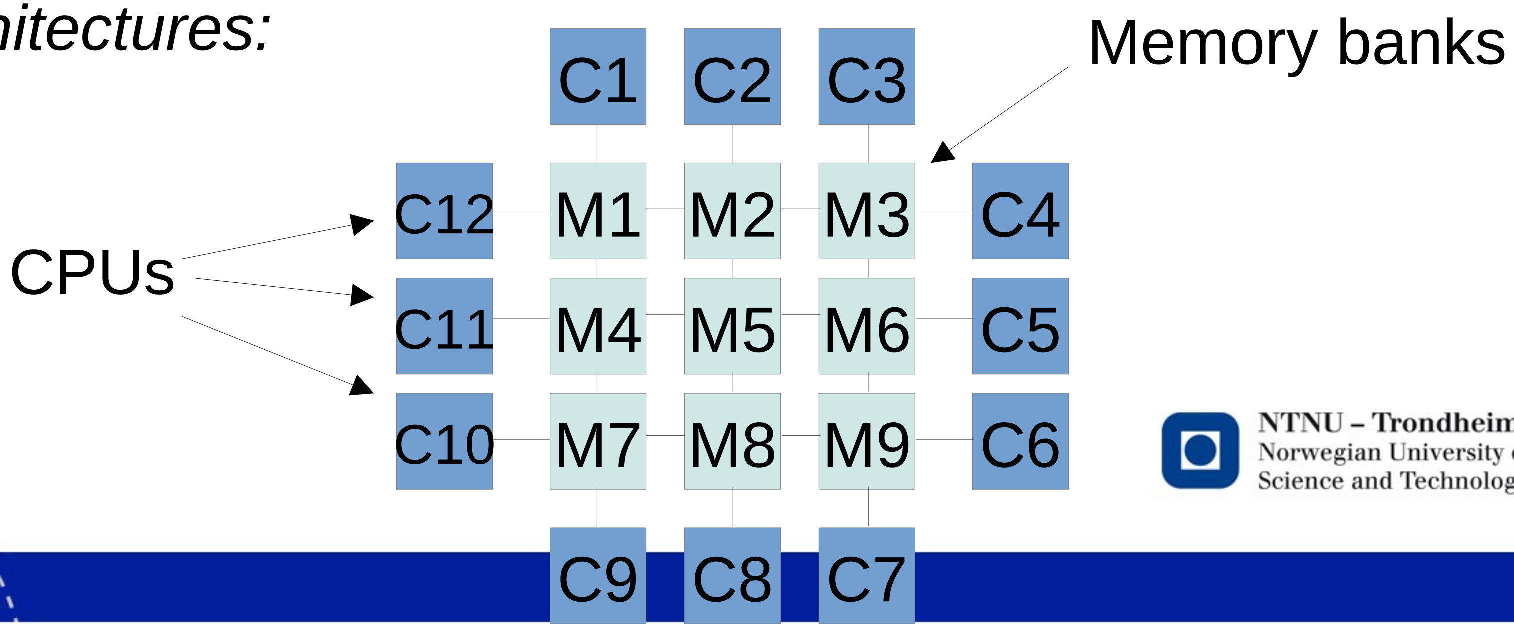
Jan.Christian.Meyer@ntnu.no

Atomic operations

- We’ve pointed out several times that a relaxed order of memory operations causes problems
  - In simultaneous attempts to write the same location
  - In our home-made attempt at protecting a critical region
- Efficient implementation of a mutex requires some architectural support
  - In the form of *atomic operations*
  - From Greek “*atomos*”, meaning “indivisible”
  - Some instructions are hardwired to complete without interruption

A brief history of shared memory

- In days of yore, there was
  - only 1 processing core on a chip
  - comparable clock rates for cpu and memory bus
  - no cache memory
- In parallel computing, this gave us *dancehall architectures:*



Properties of the dancehall architectures

- All memory is shared by every CPU
- Any CPU can read/write to any memory bank at the same speed
  - Uniform Memory Access (UMA)

    ...and hence,
- Any CPU can contact any other at the same cost
  - just like any partner can invite any other to dance in a dancehall
- Another name is *Symmetric MultiProcessor* (SMP*)
  - *"Symmetric"* because everything costs the same everywhere

* **NB:** this abbreviation means something else now

This came with race conditions

- To solve it, the interconnect fabric+cpu design supported atomic operations such as

  Test-and-set
  - Check if a value is 0, set it to 1 if it isn't, return result to the CPU
  - Great for spin-locking

  Fetch-and-increment
  - Increase the number in memory, return what it was before to the CPU
  - Great for obtaining ticket numbers in a queue, for instance

  Fetch-and-add
  - Fetch-and-increment with arbitrary sized increment

  Compare-and-swap
  - Check if a value is equal to an expected value, exchange it for a number from the CPU if it is, and return whether or not it succeeded

Fetch-and-phi operations

- Together, these are called *fetch-and-phi* ops
- If they otherwise cost the same, some of these operations are more powerful than others
    - *Compare-and-swap* admits more general synchronization algorithms than *test-and-set* in the same # of ops
- For almost two decades, it was held that support for better fetch-and-phi operations meant you had a better supercomputer

Dawn of the 21st century

- As the memory wall emerged, access to closer memory banks grew faster than access to remote memory banks
  - Non-Uniform Memory Access (NUMA)
- Caches try to bridge the performance gap, but they only work for 1 processor and make it worse for the rest
  - cache-coherent NUMA (ccNUMA)
- SMP went from meaning “Symmetric MultiProcessor” to “Shared Memory Processor”
  - Because they’re similar, but memory access isn’t symmetric anymore
- Multi-core laptops are technically small SMPs
  - The name is no longer highly fashionable
  - You can still come across it, though

Alternate atomic solutions #1

Lock access to the cache line targeted by an instruction

- The interconnect fabric knows which CPUs have a copy of the cache line
- If there’s more than 1, invalidate all other copies, and lock access to the memory bank with the value in it as well

- This is what we get with Intel family & its compatible competitors
  - They carry legacy from CISC design philosophy

Alternate atomic solutions #2

Load Linked/Store Conditional

- LL is an instruction that fetches a value into a register, and temporarily tags the memory bank it came from
- While the value is in registers, it can be manipulated using the CPUs entire instruction set
- The matching SC instruction tries to write the result back to the tagged memory bank, and returns whether or not it succeeded
- If it fails, the value isn't stored, because someone else altered it in the meantime
- The program gets to know about the failure, and can decide what to do

- This comes from the MIPS line of processor designs
  - Explicit Load/Store instructions is more of a RISC way to handle things

Alternate atomic solutions #3

Atomic Reservoir Memory

- Separate memory banks are wired directly to the processor, and bypass all caching mechanisms
- Slower, but all read/write operations are atomic
- O/S supports separate malloc/free functions that only get blocks of memory from this subsystem

- This comes from the Stanford DASH line of SMP systems
  - Not fashionable in 2023, but you never know when an old idea will put in a new appearance again

Read/modify/write instructions in x86_64

- If you know x86_64 assembly, you'll be familiar with the fact that it has a CISC style instruction set
  - Large set of complicated operations with many addressing modes
  - Many of these instructions require multiple CPU cycles to complete
- Some operations include an entire read/modify/write cycle in one single instruction, such as
  - incq (%rax) ← increment value at addr. in register rax
  - addq $14,(%rax) ← add 14 to value at addr. In register rax
  - xchgq %rbx,(%rax) ← swap value at addr. (%rax) with reg. Rbx

Atomic ops in x86_64

- Such instructions can be made atomic by prefixing them with ‘lock’ in the assembly code

lock incq (%rax)	← Atomically increment nr. at (%rax)
lock addq $14,(%rax)	← Atomically add 14 to nr. at (%rax)
lock xchg %rbx,(%rax)	← Atomically swap %rbx for (%rax)

- This makes them run a bit slower
- The effect of “lock” is to grant exclusive access to either
  - the cache line with the memory value in it (if no other core has a copy), or
  - the entire memory bus, if necessary

  for the duration of the instruction

  (Solution #1 of the variants we mentioned)

Atomic ops in GCC

- GCC has a set of built-in functions that aren’t directly part of C (or any other) language, with names like

  __atomic_test_and_set

  __atomic_fetch_add

  __atomic_compare_exchange

  ...and so on
- These mirror the fetch-and-phi ops of olden times
- They’re actually there because they are used to implement the atomics defined in C++ since 2011
- You can call them yourself, if you like
  - They’re probably supported by clang too, but I haven’t looked

The nasty part of all this

- At the CPU architecture / assembly instruction level, atomics differ from design to design
  - ...and not everyone likes to mix assembly with their high-level source code
- At the O/S compiler level, atomics aren't standardized
  - The builtin functions of GCC are just a design decision that the GCC people invented
  - It's popular to be GCC-compatible, but it's not mandatory
- This is not good for writing portable code

Atomic ops in OpenMP

- In the name of portability, OpenMP assumes that your architecture has *some* range of atomic instructions that can be used on these statements:

```
x++          ++x        x--        --x
x += (expr)     x = x + (expr)           x = (expr) + x
x -= (expr)     x = x - (expr)           x = (expr) - x
x *= (expr)     x = x * (expr)           x = (expr) * x
x /= (expr)     x = x / (expr)           x = (expr) / x
x &= (expr)     x = x & (expr)           x = (expr) & x
x ^= (expr)     x = x ^ (expr)           x = (expr) ^ x
x |= (expr)     x = x | (expr)           x = (expr) | x
x <<= (expr)    x = x << (expr)          x = (expr) << x
x >>= (expr)    x = x >> (expr)          x = (expr) >> x
```

The atomic directive

- If you want to make an expression like that atomic, just prefix it like so:

```
#pragma omp atomic
x += my_local_value
```

- We can apply this to our pi example from last time
- Implementation in today's example code archive, pi_atomic_openmp.c
  - Note that the lock is gone, along with its initialization and destruction

Bigger critical sections

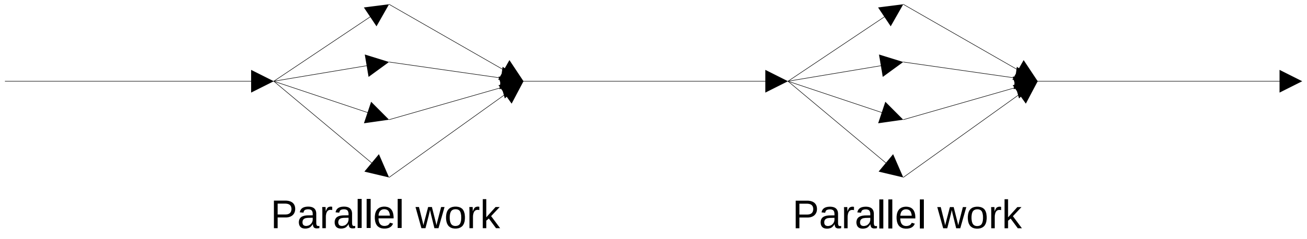
- The statements that can be made atomic are all quite short and sweet
  - Their protection mechanism is expected to be a single instruction
- If we have a longer bit of code to protect, we already know how to do it with a lock
- We can make OpenMP generate the lock too

```
#pragma omp critical
{
    /* Only one thread will come in here at a time */
}
```

- Almost redundant example code: pi_critical_openmp.c

Last of the mutual exclusion

- As we've noted, OpenMP threads can be spawned and joined many times throughout a program
- Execution typically runs in bursts of parallelism:



- One of these threads spawns the others, and lives on afterwards
- In OpenMP terminology
  - The collective is called a *team*
  - The spawning/joining thread is called the team's *master*

The master directive

- Inside a parallel region, you can label a block like this

```
#pragma omp parallel
{
    /* Lots of threads run here */
    #pragma omp master
    {
        /* Only the master thread will come in here */
    }
}
```

A final pi example

- The same program is implemented again in pi_master_openmp.c, using the obvious mechanism
- The structure is a little different
  - omp_get_max_threads() obtains the thread count outside of a parallel region, and sizes up an array with an entry per thread
  - All the worker-threads put their partial pi estimates in that array
  - There’s a barrier to make sure that everyone’s work is finished
  - The master section adds up the final global sum
- The example is a little contrived
  - For *this* problem, it would be easier to shut down the threads and do the sum afterwards
  - Still, you can see the principle at work

Footnote: This version is also quite slow, we will get back to the reason later