NTNU – Trondheim
Norwegian University of
Science and Technology

OpenMP

Guest lecture for TDT4200

Jan Christian Meyer 24.09.2024

Pthreads reminder

- We start pthreads by giving them a function call to work on in the background
- When there's nothing left to do in the foreground, we wait for them to finish
- The threads have their own private values in the local variables of their function call
- The threads share the same copies of global variables, open files, *etc.*
  - We can coordinate access using *mutexes* (locks)

A quick example

- I have prepared a short demonstration
  - It’s in the code archive, under ‘01_hello_pthreads/’
- It starts 100 threads, each of them
  - starts with a brief, random wait (to upset the starting order)
  - takes a number from a global counter
  - increments the global counter
  - reports the number
- It uses a lock (mutex) to ensure that each thread gets a different number
  - If you remove the lock, there will be duplicates every now and again

Things to do with pthreads

- Start
  - This requires a function call that takes a (void *) argument, and returns a (void *) result
- Stop
  - This happens when a thread's function call returns
- Synchronize using explicit locks
  - This requires the program to declare and manipulate each lock
- Send (and wait for) signals
  - If we attach a pthread_cond_t to a lock, threads that are waiting in line to lock it can be activated by the one that holds the lock
- Wait for everyone to arrive at a barrier
  - This is technically an optional pthreads feature, but it's often supported
  - When it's not, we can make barriers using locks+signals

Pthreads code is verbose

- Every time we start a thread, it has to be
  - Declared
  - Initialized
  - Waited for
- Every time we want mutual exclusion, we need to
  - Declare a lock
  - Initialize it
  - Make sure that it's used correctly
  - Destroy it
- Every time we want a signal, we need to
  - Declare a condition variable
  - Initialize it
  - …
- It's a lot of typing

Pthreads are not (inherently) safe

- They just let you start several concurrent calls
  - If those calls write to the same locations in memory, they'll just do it in whatever order they are scheduled
  - This is not necessarily the same every time you run the program
- All synchronization is up to the programmer
  - Locks don't actually force threads to stay away from the section of code they protect
  - The threads must actively check the state of the lock and respect it in order to avoid race conditions
- This philosophy carries through to OpenMP
  - If you say something should be done in parallel when it should not, you will get wrong answers and no help from the compiler

OpenMP
(Open Multi-Processing)

- OpenMP offers a less typing-intensive way to manage pthreads
  - OpenMP is not actually *required* to use pthreads behind the scenes, but practically every implementation does it
- It also offers a few other things, but thread control is a good place to start

  ...and enough to cover in TDT4200

What’s in a name?

- We’ve heard about MPI
  - It’s a specification for some function calls
  - Anyone can make their own implementation, so many exist
  - There’s *mpich*, *IntelMPI*, *SpectrumMPI*, *mvapich*, …
  - The idea is that you can switch variants without changing your code
- OpenMP is also a general specification with different implementations
  - There’s *gomp, llvm/openmp, IOMP, xlc-openmp,* …
  - Various flavors have come and gone since 1997
- In 2004, someone decided to call their MPI variant...
  *(*sigh*)* … ”OpenMPI”
  - I wish they would have chosen another name, but nobody asked me
  - It’s easy to confuse the names, but please try not to, they are entirely different things
  - **OpenMP** is an interface for parallel programming with threads
    OpenMPI is one of many MPI alternatives, for parallel programming with processes

OpenMP vs. pthreads

- If you want to, you can do the same things with OpenMP as you can with pthreads
- In today's code archive, '02_hello_openmp/' contains an OpenMP version of the same program as we used to demonstrate pthreads
- It's almost exactly the same, except that
  - The lock type is called omp_lock_t, and its init/destroy functions also have names with 'omp' in them
  - The lock/unlock functions are similarly renamed
  - The start/stop loops are gone – *what's this?*
    #pragma omp parallel num_threads(100)

Before we proceed

- I just said that you can do the same things with OpenMP as with pthreads, but it's **not 100%** true:
  - You can't make threads go to sleep and then signal them to wake up again
  - The idea with OpenMP threads is that they will all have something to do for as long as they last, so there's no suspend/resume equivalent of the pthreads' condition variables
  - OpenMP threads are really easy to start and stop, so we can just stop unemployed threads, and start some new ones when there is extra work again

Returning to our topic

- Instead of a 100-iteration loop that starts one thread per iteration, (and another that waits for them), we now have this:

```
#pragma omp parallel num_threads(100)
hello_world();
```

- The same thing happens when the program runs, it's just shorter to write
- The compiler's preprocessor finds the #pragma directive, and inserts code to
  - spawn 100 threads
  - make each thread call 'hello_world'
  - wait for them all to finish

…before the result is passed on to the compiler and treated as usual

Changes in the Makefile

- There's another "small" difference:
  - The pthreads implementation has -pthread among the compiler flags
  - The OpenMP implementation has -fopenmp in the same place
- The difference is actually quite profound
  - The first one tells the compiler to find the pthreads library, and link it up with all the phtread function calls that we put in the source code
  - The second one tells the preprocessor to find these "#pragma omp" things in the source code, and use them to edit in a bunch of pthreads code that we didn't explicitly write

So, what is this #pragma?

- Generally speaking,
  - #pragma is a way for compiler authors to extend the programming language with features that may or may not be supported by other compilers
- If you put #pragma directives in your code that the compiler doesn't understand, it will just ignore them
- They can request anything, you can literally write
  - #pragma make me a sandwich

  in your program, and the code will work just fine
  - ...but you will only get a sandwich if you have a compiler that knows how to make one
- Try it at home!

The OpenMP solution

- Directives that begin with **#pragma omp** suggest that you hope your compiler is prepared to create some pthread code for you
- Pretty much all of the compilers used in HPC know how to do this, so it's OK to expect some automatic parallelization to happen
- If you wrote your code without *requiring* it to be parallelized, it will still work with compilers that don't understand OpenMP directives
  - It will just run a lot slower

The anatomy of a directive

- All the OpenMP directives go

  #pragma omp <directive> [clauses]

This says we're writing OpenMP

This says what kind of work we want done to the code

These are optional parts that adjust how the directive acts

- In the example, 'parallel' is the directive
  – It says that we want to start a bunch of threads
- num_threads(#) is a clause
  – If you don't specify it, you'll get one thread per core in the machine by default

Private and shared variables

- We mentioned that pthreads have private variables inside the scope of the function they call

```
void *threaded_function ( void *args ) {
    int my_value = rand();      // <--- only this thread has this value
}
```

- This is because function calls allocate local variables in *stack memory*, and each thread has its own stack
- Function calls are actually a kind of special case for this, stack allocation happens more often than one might think

Basic blocks

- When you put some statements between { curly braces } in C, you get an opportunity to declare variables that only last as long as the (basic) block
- A basic block is a statement, so you can put it almost anywhere
- Witness:

```
int main() {
    int a = 64, b = 32, c;
    {
        int a = 42;        // override the a we had (i.e. push a different a on top of stack)
        c = a – b;  // use the closest definition of a (i.e. the topmost one on the stack)
    }
    printf ( “%d\n”, c );        // <--- this prints 10, even though a is 64 again now
}
```

NTNU – Trondheim
Norwegian University of Science and Technology

Basic blocks with names

- Functions in C are pretty much just top-level basic blocks that have been decorated with a name
  - They also feature some local variables that can get values from outside (arguments)
  - They let you extract one local value and keep it for later (return value)
  - Otherwise, they're not much more than a bit of stack memory where we keep block-local declarations
- OpenMP uses this fact to let you spawn threads that last as long as a basic block
  - There's an example in the code archive, under 03_hello_basic_block

When there isn't a basic block

- In the second example, we have

```
#pragma omp parallel
hello_world();
```

- This is (practically) equivalent to

```
#pragma omp parallel
{
    hello_world();
}
```

- OpenMP directives apply to the next statement

  ...but as we've seen, the next statement can be a block

NTNU – Trondheim
Norwegian University of
Science and Technology

Identity numbers and such

- Inside a parallel region, each thread can obtain its index using

  omp_get_thread_num()

  and find out how many active threads there are with

  omp_get_num_threads()

- Outside of parallel regions, these will respond with 0 and 1
  - If you want to know how many threads will be launched without actually starting them, there is

    omp_get_max_threads()

Now we can do All The Things™

- As with MPI, all we really need is
  - An ID number
  - Size of the collective
  - A way to communicate (threads can use shared data + locks)

  …and then all kinds of parallel things can happen
- The rest of OpenMP is window dressing to make stuff easier to write
  - It's really practical, though - hardly anyone needs to do the thread-id and locking thing explicitly, I just wanted to show that it's there
  - Next lecture, we'll talk about the *worksharing directives*