

Introduction to OpenMP



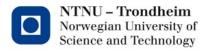
Open Multi-Processing

- OpenMP is almost as storied as MPI
 - Version 1.0 was published in 1997
- It occupies a similar position as a de facto standard tool for scientific number crunching
- It targets shared-memory systems
 - MPI's unit of parallelism is the process
 - OpenMP's unit of parallelism is the thread
- You can think of it as a more convenient way to handle pthreads
 - It's not obliged to be implemented using pthreads, but it very often is
 - It doesn't only encompass threads, but that is by far its most common use



What's in a name?

- Many moons ago, a major MPI implementation called LAM/MPI merged with two less prominent ones
- A new name was required after the merger, so they settled on...
 "OpenMPI"
 - *sigh*
 - I think they should have chosen a different name, but here we are
- OpenMPI is one out of many MPI implementations
- OpenMP is an entirely different programming model, with its own specifications document
 - Various implementations of this interface emerge and disappear...
 - So it goes.



Parts of the puzzle

As we've seen

- MPI is a separate 3rd party library of functions that you install in addition to your compiler
- Pthreads are provided by the operating system, the function declarations come with the compiler

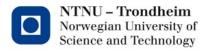
OpenMP is a little bit of both

- Its core is a set of language extensions that must be supported by the compiler
- It also has a runtime library of functions that you can call to inspect the state of what the compiler has generated



Language extensions?

- Yes; C has a standardized way to do nonstandard things, so to speak
- The #pragma directive can be followed by some text that the compiler will discover during its initial scan of the program code
 - If it understands the text, it can insert appropriate code to replace the directive with
 - If it doesn't understand the text, the compiler is free to discard it
- This way, compilers can support optional features in the code that
 - Work when you use a compiler that supports them
 - Don't break the program even if you use a compiler that doesn't support them



A hypothetical example

#pragma can ask for literally anything:

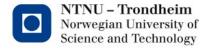
```
#include <stdio.h>
int main() {
    printf ( "Hello, world!\n" );
    #pragma play me a song
    return 0;
}
```

Norwegian University of Science and Technology

- You can compile this code without issue (try it at home)
 - My compiler only makes the usual hello-world binary without any special effects
 - It still reads the command
 - It just doesn't know what to do with it, and throws it away
- Given a compiler that supported it, this directive could produce a musical executable

A more practical use

- Pthreads code is tediously repetitive
- We have to do the same things over and over:
 - Declare, initialize, use, and destroy a mutex for every thing that needs protection
 - Declare, initialize, use, and destroy a cond for every signal
 - Declare, initialize, use, and destroy an object for every barrier
 - Simple sets of operations make for lots of repetitive typing
- Since the code is practically the same over and over, we might as well make the compiler generate it
- It can figure out what to generate from a tiny language embedded in well-placed #pragma directives
- That's OpenMPs mechanism of choice



Stack contexts

- We've covered how a function call encapsulates a local set of values on the call stack
 - That's the connection between function calls and pthread creation
- Other local scopes also contain stack contexts
- Consider this program fragment:

```
int main ()
{
   int a = 42, b = 32, c = 0;
   {
     int a = 64;
     c = a - b;
   }
   printf ( "a = %d, b = %d, c = %d\n", a, b, c );
   return 0;
}
```



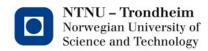
The output is "a = 42, b = 32, c = 32"

What's going on?

- An open { /* basic block */ } establishes a local stack context
 - Just like a function call, except that it doesn't have arguments and return value
- A basic block can appear wherever a statement can
 - That's how we make if-branches and loop bodies (and function bodies, for that matter)

```
int main ()
{
  int a = 42, b = 32, c = 0;
  {
    int a = 64;
    c = a - b;
  }
  printf ( "a = %d, b = %d, c = %d\n", a, b, c );
  return 0;
}
```

Basic block acting as a statement

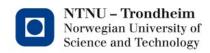


- Even when they don't have names and arguments, basic blocks let you declare variables that live only inside the block
- That is a stack context at work:

```
int main ()
{
    int a = 42, b = 32, c = 0;
    {
        int a = 64;
        c = a - b;
    }
    printf ( "a = %d, b = %d, c = %d\n", a, b, c );
    return 0;
}
```

Stack state

c=0 b=32 a=42



- After a few more steps, another stack context has been started
- We now have two variables called 'a'
 - The most recent one is near the top of the stack in the scope of the most recently opened block
 - The other one sits in the stack space of the enclosing block

```
int main ()
{

Execution is here

int a = 42, b = 32, c = 0;

{

int a = 64;

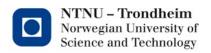
c = a - b;
}

printf ("a = %d, b = %d, c = %d\n", a, b, c);

return 0;
```

Stack state

a=64 c=0 b=32 a=42



- When the time comes to evaluate this expression
 - The nearest declaration of a is used
 - The current block's context doesn't contain b and c, so they are tracked down in the enclosing scope
 - (If they hadn't been there, the next thing would be to check if they were declared globally)

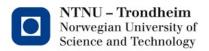
```
Execution is here
```

```
int main ()
{
    int a = 42, b = 32, c = 0;
    {
        int a = 64;
        c = a - b;
    }
    printf ( "a = %d, b = %d, c = %d\n", a, b, c );
    return 0;
}
```

Stack state

$$c=32$$

$$a = 42$$



- When the block ends, its local context is deleted from the stack
 - the "old" value of a becomes the topmost one in our stack context again
 - Hence, c is 32 even though a-b is 10 now
 - We temporarily created a stack context with a different local variable in

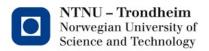
```
int main ()
               int a = 42, b = 32, c = 0;
                  int a = 64:
                  c = a - b;
Execution
               printf ( "a = %d, b = %d, c = %d\n", a, b, c );
               return 0:
```

Stack state

c = 32

b = 32

a = 42



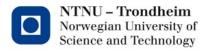
is here

Stack contexts can be threads

- We might as well leave it to the compiler to write the thread spawning and joining logic
- There's a program called 'hello_openmp' in today's example archive
- Notice that the Makefile has added the flag
 -fopenmp

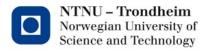
to the C compiler's command line

- This enables OpenMP using gcc and clang
- Using icc, the flag is -qopenmp
- Using MSVC I don't know what it is, but it's something (read the manual)



We have the magic ingredients again

- Armed with a thread count and a thread id#, we can solve all the embarrasingly parallel problems again
 - Pick a task based on the id#
 - Handle it
- OpenMP has a far richer set of concepts and tools than this
 - So far, it's definitely the least amount of typing to make a hello world example parallel, though



How many threads do we get?

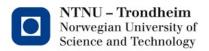
- By default, OpenMP assumes that you want one thread per core that your O/S recognizes
- You can adjust it without recompiling the program
 - if you set the environment variable
 OMP_NUM_THREADS
 in your shell, OpenMP will look it up there
- You can also hard-code it into the program
 - #pragma omp parallel num_threads(4)
 will always spawn 4 threads, overriding both your system information and
 the environment variable
 - There's rarely a good reason to do this, though



We can do locking

(just like pthreads)

- The example program 'pi_mutex_openmp.c' is (functionally) identical to last lecture's 'pi_mutex_fast' example
 - Computes local estimates per thread
 - Uses a mutex data structure to avoid race conditions for a global value
- There are smoother ways to do this in OpenMP
 - Don't take it as a wonderful implementation strategy
 - I just wanted to demonstrate that OpenMP code can act precisely like pthread code



We can do barriers

(just like pthreads)

- The example program 'pi_barrier_openmp.c' is (functionally) identical to last lecture's 'pi_barrier' example
 - Repeats computation 10 times
 - Synchronizes between repetitions, to avoid race conditions when resetting the global value
- There are smoother ways to do this in OpenMP as well
 - Don't take it as a wonderful implementation strategy
 - I just wanted to demonstrate... oh, you get the point



We can't do pthread_cond_t

- Inter-thread signals aren't a thing in OpenMP
- OpenMP threads aren't supposed to be sleeping, they're supposed to be computing something
 - The constructs contain lots of busy-waiting, critical sections are expected to be as short as possible
 - Oversubscribe thread counts at your own peril
- If you want to yield CPU cores, just shut down the threads instead
 - They're very easy to bring back again

(There <u>is</u> actually a different technique as well, but we'll get back to it later)



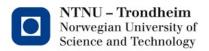
How safe is this stuff?

- It is a little easier to write correct OpenMP code than it is to write correct pthreads code
 - That's mostly because it requires you to consider fewer details at a time, though
- The "gentleman's agreement" philosophy still applies
 - OpenMP makes threads when you tell it to
 - If you treat a shared variable as if it were private, OpenMP will take you at your word
 - If you say that something should be parallelized when it should not, you will get programs that compute wrong answers



Our reason to do this

- Today's examples are really written in a pretty clunky style
 - It is actually quite rare to need the thread id# and count for anything in OpenMP
 - I just wanted to show you that they are there, so as to demonstrate that the correspondence to pthreads lurks just below the surface
- That's kind of why we covered pthread programming in the first place
 - Like assembly code, it's not very common to need explicit pthread code
 - Like assembly code, it's good to know what's going on even if you don't type it out by hand



Going forward

Next time, we'll start on the rich library of OpenMP abstractions

