OpenMP and Rust

Contents of Lecture 7

- OpenMP
- Rust

Parallel execution of software

- Ideally optimizing compilers would be able to parallelize source code.
- From our example of the preflow-push algorithm, I think it's impossible to write such a compiler.
- Instead of writing new sequential programs we can for example use
  - Java/C/C++ threads, or
  - Scala actors.
- What about all existing codes?

OpenMP for C/C++ and FORTRAN

- Another option is tool support for manual parallelization:
  - Programmer annotates the source code and guarantees the validity of parallelization of a loop.
  - Tool support: generating parallel code for a loop
- GCC supports the OpenMP standard for this approach.
- Include `<omp.h>` and annotate e.g. as:

```
#pragma omp parallel
#pragma omp for
for (i = 0; i < n; ++i) {
        /* ... */
}
```

- Compile with `gcc -fopenmp`

The main advantage with OpenMP

- We don't want to rewrite millions of C/C++ and FORTRAN codes from scratch.
- Using a new and relatively untested language may be a big risk.
- Untested = less than a decade of community experience and tool support
- Support from only one company may also be problematic
- With OpenMP we can parallelize our applications **incrementally**.
- We can focus on one for-loop at a time.

Origin of OpenMP

- All supercomputer companies had their own compiler directives to support this ”semi-automatic” parallelization.
- When SGI and Cray (one of the three Cray companies) merged they needed to define a common set of compiler directives.
- Kuck and Associates, a compiler company, and the U.S. ASCI (Accelerated Strategic Computing Initiative) joined SGI and Cray.
- In 1997 IBM, DEC, HP and others were invited to join the group now called OpenMP.
- In 1997 the specification for OpenMP 1.0 for FORTRAN was released.
- Next year the specification for C/C++ was released.
- The current version is OpenMP 5.2 and was published 2021.

OpenMP components

1. Compiler directives. `#pragma` in C/C++.
2. A runtime library
3. Environment variables, like `OMP_NUM_THREADS`.

Barriers

- A barrier is a synchronization primitive which makes all threads reaching the barrier wait for the last.
- Similar to Pthreads barriers and not a ”memory barrier” in the sense of a C11 memory fence
- This barrier needs a lock, a counter, and knowing the number of threads.
- When the last thread has reached the barrier, all threads can proceed and continue after the barrier.

#pragma omp parallel

- A **structured block of code** is either
  - a compound statement, i.e. a block enclosed in braces, or
  - a for-loop.
- The pragma `omp parallel` is used before a structured block of code and specifies all threads should execute all of that block.
- Note that this is typically not what we want in a for-loop, see below.
- A default number of threads is used, which can be changed with the environment variable `OMP_NUM_THREADS`, which can be larger than the number of processors in the machine.
- This pragma creates an implicit barrier after the structured block.

An example

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

int main(void)
{
        #pragma omp parallel
        {
                int     tid; // thread id.

                tid = omp_get_thread_num();
                printf("hello world from thread %d\n", tid);
        }

        return 0;
}
```

- Since tid is declared in the compound statement, it becomes private.
- `omp_get_thread_num()` returns an id starting with zero.

OpenMP and Pthreads

- The OpenMP runtime library creates the threads it needs using Pthreads on Linux.
- After a parallel block, the threads wait for their next work and are not destroyed in between.
- This model of parallelism is called **fork-join** and only the main thread executes the sequential code.
- It's possible to nest parallel regions.

Parallel for-loops

- In addition to the `#pragma omp parallel` you must also specify `#pragma omp for` before the loop.
- Without the second pragma each thread will execute all iterations.
- Note that it's the programmer's responsibility to check that there are no data dependences between loop iterations.

Two OpenMP functions

- To specify in the program how many threads you want, use

```
omp_set_num_threads(nthread);
```

- To measure elapsed wall clock time in seconds, use

```
double  start, end;

start = omp_get_wtime();
/* work. */
end = omp_get_wtime();
```

For loop scheduling

- There are three ways to schedule for-loops:
- `schedule(static)`
  - The iterations are assigned statically in contiguous blocks of iterations.
  - Static scheduling has the least overhead, obviously, but may suffer from poor load imbalance, e.g. in an LU-decomposition.
- `schedule(dynamic)` or `schedule(dynamic, size)`
  - The default size is one iteration.
  - A thread is assigned size contigouos iterations at a time.
- `schedule(guided)` or `schedule(guided, size)`
  - The default size is one iteration.
  - With a size, a thread never (except possibly at the end) is assigned fewer than size contigouos iterations at a time.
  - The number of iterations assigned to a thread is proportional to the number of unassigned iterations and the number of threads.
- We can set the scheduling with the environment variable `OMP_SCHEDULE` which must be one of above three but without the size.

An Example

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

#define N (1024)

float a[N][N];
float b[N][N];
float c[N][N];

int main(void)
{
        int     i;
        int     j;
        int     k;

        #pragma omp parallel private(i,j,k)
        #pragma omp for schedule(static, N/omp_get_num_procs())
        for (i = 0; i < N; ++i)
                for (k = 0; k < N; ++k)
                        for (j = 0; j < N; ++j)
                                a[i][j] += b[i][k] * c[k][j];


        return 0;
}
```

- We need private i, j and k since they are declared before the pragma.
- If a function is called in a parallel region, all its local variables become private.

Parallel tasks

```
#pragma omp sections
{
        #pragma omp section
        {
                work_a();
        }

        #pragma omp section
        {
                work_b();
        }

}
```

- Each section is executed in parallel.

Reductions

- By a **reduction** is meant computing a scalar value from an array such as a sum.
- The loop has a data dependence on the `sum` variable.
- How can we parallelize it anyway?

```
float   a[N];
float   sum;
int     i;

for (sum = i = 0; i < N; ++i)
        sum += a[i];
```

OpenMP reductions

- By introducing a sum variable private to each thread, and letting each thread compute a partial sum, we can parallelize the reduction:

```
float   a[N];
float   sum;
int     i;

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

- We can write the pragmas on one line if we wish:

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

- There are reductions for: `+ - * & | ^ && ||` with suitable start values such as 1 for `*` and `~0` for `&`.

Critical sections

- A critical sections is created as in:

```
#pragma omp critical
{
        point->x += dx;
        point->y += dy;
}
```

Atomic update

- When one variable should be updated atomically, we can use:

```
#pragma omp atomic
count += 1;
```

Explicit barriers

- Recall there is an implicit barrier at the end of a parallel region.
- To create a barrier explicitly, we can use:

```
#pragma omp barrier
```

Work for one thread

- Recall only the main executes the sequential code between parallel regions.
- If we wish only the main should execute some code in a parallel region, we can use

```
#pragma omp master
```

- If it doesn't matter which thread performs the work, we can instead use

```
#pragma omp single
```

- There is a difference between the two above constructs: an implicit barrier is created after a `single` directive.

Locks

- OpenMP supports two kinds of locks: plain locks and recursive locks.
- Recall a thread can lock a recursive lock it already owns without blocking for ever.
- Recursive locks are called nested locks in OpenMP.
- The lock functions are `omp_init_lock`, `omp_set_lock`, `omp_unset_lock`, `omp_test_lock` and `omp_destroy_lock`, and `omp_nest_init_lock`, `omp_nest_set_lock`, `omp_nest_unset_lock`, `omp_nest_test_lock` and `omp_nest_destroy_lock`

OpenMP memory consistency model

- Weak ordering is the consistency model for OpenMP.
- The required synchronization instructions are inserted implicitly with the above introduded directives.
- A for loop can be created without an implicit barrier using `nowait` and in that case `#pragma omp flush` makes caches consistent.
- A list of variables to write back can be specified:
  `#pragma omp flush(a,b,c)`

Open source compiler and company support for OpenMP

- Non-profit consortium OpenMP architecture review board `openmp.org`
- Both GNU and Clang compilers (Clang only C/C++)
- Absoft, AMD, ARM, Cray, HP, Fujitsu, IBM, Intel, Microsoft, Nvidia, NEC, Oracle, Pathscale, Portland Group, Red Hat, Texas Instruments,

Rust

- Hello world in Rust
- Object ownership for single-threaded programs
- Message passing
- Threads
- Shared memory objects in multi-threaded programs

Hello, world

```
fn main()
{
        println!("hello, world");
}
```

- Save in `a.rs` and type `rustc a.rs && ./a`
- Or in `src/main.rs` and type `cargo run`
- With cargo you need a file `Cargo.toml` with some definitions, e.g.:

```
[package]

name = "preflow"
version = "0.0.1"
authors = [ "Jonas Skeppstedt" ]

[dependencies]
text_io = "0.1.8"
```

More printing

```
fn main()
{
        let s = String::from("world");
        println!("hello, {} {}", s, "again");
}
```

- Creates a string from the heap (Java new and C malloc)
- `{}` takes the next parameter
- Output is `hello, world again`
- The memory for an object can be deallocated with the function `drop`
- The `drop` function is called automatically when reaching the `}`

Java

```
class a {

        public static void main(String[] args)
        {
                String s = new String("hello, world");
                String t = s;
                System.out.println(t);
        }
}
```

- Only one string object
- Garbage collection takes care of the memory for the string object, of course

C

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

int main()
{
        char*   s;
        int     n;

        n = 1 + strlen("hello");
        s = malloc(n);
        strcpy(s, "hello");
        printf("%s\n", s);
        free(s);
}
```

An error in C

```
int main()
{
        char*   s;
        char*   t;
        int     n;

        n = 1 + strlen("hello");
        s = malloc(n);
        strcpy(s, "hello");

        t = s;
        printf("%s\n", t);
        free(s);
        free(t); // disaster will follow
}
```

Rust heap objects

- Java's garbage collection can be slow
- It can be bad if it occurs at the wrong moment (e.g. when emergency landing an airplane)
- The C explicit allocation and deallocation is fine if you are careful
- Rust has no garbage collection like Java but strict rules about how pointers can be used
- A purpose with Rust is to be fast systems programming language without the headaches of C (*their* interpretation)
- Or, (my interpretation) a new Gulag without the freedom of C
- It is interesting to study, though, and has many nice ideas

Karl Rikte's MSc thesis from 2018

https://lup.lub.lu.se/student-papers/search/publication/8938297

Rust upholds the safety and zero-cost claims. Using Rust has been found to aid in achieving an improved, shorter, more expressive architecture. The learning curve is a bit steep, but productivity has been found to be high once learned. Tooling support is mature, but IDEs are not yet full featured.

Moving objects

```
fn main()
{
        let s = String::from("world");
        let t = s;

        println!("hello, {}", t);

}
```

- Similar to the Java program and no complaints
- The variable s becomes useless however
- The string object has been **moved** to t which owns it from the =
- Only one owner at a time!

Using a moved object

```
fn main()
{
        let s = String::from("world");
        let t = s;
        println!("hello, {}", s);
}
```

```
error[E0382]: borrow of moved value: ‘s‘
 --> a.rs:5:31
3 |     let s = String::from("world");
  | - move occurs because ‘s‘ has type ‘std::string::String‘,
      which does not implement the ‘Copy‘ trait
4 |     let t = s;
  | - value moved here
5 |         println!("hello, {}", s);
  | - value borrowed here after   ^ move
```

Clone

```
fn main()
{
        let s = String::from("world");
        let t = s.clone();
        println!("hello, {}", s);
}
```

- This works but gets complaint about unused variable t

Function call

```
fn f(t: String) { }

fn main()
{
        let s = String::from("world");
        f(s);
        println!("hello, {}", s);
}
```

- Also invalid since the call also moves the string

Returning a value

```
fn f(s: String) -> String
{
        s
}

fn main()
{
        let s = String::from("world");
        s = f(s);
        println!("hello, {}", s);
}
```

- Ownership can be returned from a function
- Still wrong though: cannot assign to s twice (but see `mut` below)

Solved

```
fn f(s: String) -> String
{
        s
}

fn main()
{
        let s = String::from("world");
        let t = f(s);
        println!("hello, {}", t);
}
```

- Now correct
- But we may want to modify s instead of introducing t

References

```
fn f(s: &String, t: &String, u: &String)
{
}

fn main()
{
        let s = String::from("world");
        f(&s, &s, &s);
        println!("hello, {}", s);
}
```

- Safe to give away references to s (even multiple)
- Keep ownership using &
- f is not allowed to modify s

Mutable object reference

```
fn f(s: &mut String)
{
        s.push_str(" with some added text");
}

fn main()
{
        let mut s = String::from("world");
        f(&mut s);
        println!("hello, {}", s);
}
```

- Declare mutable to allow modification
- Only one reference can borrow an object at a time when mutable

Spawn

```
use std::thread;

fn main() {
        let h = thread::spawn(|| {
                println!("thread");
        });

        h.join().unwrap();
        println!("main");
}
```

- Create a thread with `spawn`
- Wait for it with `join`
- `unwrap` means controlled exit if something is wrong

Message

```
use std::thread;
use std::sync::mpsc; // multi-producer single-consumer

fn main() {
        let (tx, rx) = mpsc::channel();

        thread::spawn(move || {
                let val = String::from("hi");
                tx.send(val).unwrap();
        });

        let received = rx.recv().unwrap();
        println!("got {}", received);
}
```

- The send moves `val` from sender to receiver
- Note the `move` near `spawn` — see next slide

Send moves objects

- Since the new thread accesses data created in the main thread, it needs to own that data
- With `move` all data accessed by the new thread is moved to it
- If main also would try to use `tx` we get a compiler error:

  `error[E0382]: borrow of moved value: ‘tx‘`

Mutex

```
use std::sync::Mutex;

fn main() {
        let m = Mutex::new(129);

        let mut val = m.lock().unwrap();
        *val = 124;

        println!("{:?}", m);
}
```

- The mutex protects an integer with value 129
- The variable `val` is a mutable reference to that integer
- Use `:?` to print the mutex
- What is printed?

Output

```
Mutex { data: <locked> }
```

Unlocked

```
use std::sync::Mutex;

fn main() {
        let m = Mutex::new(129);

        {
                let mut val = m.lock().unwrap();
                *val = 124;
        }

        println!("{:?}", m);
}
```

- The mutex is unlocked automatically when `val` goes out of scope
- Now it prints:

  `Mutex { data: 124 }`

First attempt

```
use std::sync::Mutex;
use std::thread;

fn main() {
        let m = Mutex::new(129);

        thread::spawn(move || {
                let mut val = m.lock().unwrap();
                *val = 124;
        });

        println!("{:?}", m);
}
```

- No: `main` has given away the mutex

Atomic reference counters

```
use std::sync::{Mutex,Arc};
use std::thread;

fn main() {
        let m = Arc::new(Mutex::new(129));
        let c = Arc::clone(&m);

        let h = thread::spawn(move || {
                let mut val = c.lock().unwrap();
                *val = 124;
        });

        h.join().unwrap();
        println!("{:?}", m);
}
```

- Arc = atomic reference counter

Keeping track of threads

```
use std::sync::{Mutex,Arc};
use std::thread;

fn main() {
        let m = Arc::new(Mutex::new(100));
        let mut a = vec![];

        for _ in 0..2 {
                let c = Arc::clone(&m);
                let h = thread::spawn(move || {
                        let mut val = c.lock().unwrap();
                        *val = *val + 1;
                });
                a.push(h);
        }

        for h in a {
                h.join().unwrap();
        }

        println!("{:?}", m);
}
```

- A reference counter is a ”smart pointer” (from C++) which knows how many pointers point to some data
- The Arc is an atomic reference counter for the same mutex