Nettverksprogrammering
Atomic typer, parallellisering og prosesser

Ole Christian Eidheim

Institutt for datateknologi og informatikk,
NTNU

15. januar 2024

Oversikt

- **Atomic-typer**
- CPU/GPU parallellisering
- Prosesser

Atomic-typer
- trenger ikke å bli beskyttet av mutex

Ulempe: bare enkle datatyper som int og float kan gjøres atomic.

```
#include <atomic>
#include <iostream>
#include <thread>

using namespace std;

int main() {
  atomic<int> sum(0);

  thread t1([&sum]() {
    for (int c = 0; c < 1000; c++)
      sum++;
  });
  thread t2([&sum]() {
    for (int c = 0; c < 1000; c++)
      sum++;
  });
  t1.join();
  t2.join();

  cout << sum << endl; // Output: 2000
}
```

Atomic-typer
- Handtering av tilstand i flere tråder

```
#include <iostream>
#include <thread>

using namespace std;

enum class State { sitting, standing_up, standing };

int main() {
  atomic<State> state(State::sitting);

  thread([&state] { /* Draw animation frames based on state */ });

  while (true) { // Handle input
    if (/* keypress */) {
      // Stand up if sitting:
      auto expected = State::sitting;
      if (state.compare_exchange_strong(expected, State::standing_up)) {
        // Standing up, play squeaky chair sound
      }
    }
  }
}
```

Legg merke til den spesielle funksjonen *compare_exchange_strong()*, og at vi slipper å bruke mutex selv om vi både leser og skriver til state.

Mer om atomic-typer - referansetelling

- En form for *garbage collection* der et objekt blir frigjort når det ikke lenger blir brukt
- Kan være nyttig i trådprogrammering der en ikke vet i hvilken tråd et objekt brukes for siste gang

Eksempler:

- C++: std::shared_ptr
- Rust: std::sync::Arc (Atomically Reference Counted)

```cpp
#include <iostream>
#include <thread>

using namespace std;

int main() {
  thread t;
  {
    std::shared_ptr<int> ref_counted(new int(42));
    t = thread([ref_counted] {    // ref_counted is copied to thread
      this_thread::sleep_for(1s); // Wait 1 second
      cout << "value from thread: " << *ref_counted << endl;
      cout << "count from thread: " << ref_counted.use_count() << endl;
      // The last ref_counted object is destroyed at end of thread,
      // and its int value is then freed from memory
    });
    cout << "value: " << *ref_counted << endl;
    cout << "count: " << ref_counted.use_count() << endl;
    // One ref_counted object is destroyed at end of scope,
    // and its use_count is reduced by 1
  }
  t.join();
}
// Output:
// value: 42
// count: 2
// value from thread: 42
// count from thread: 1
```

Mer om atomic-typer
- referansetelling implementasjon, men ikke trådsikker

```cpp
#include <iostream>
using namespace std;

class RefCountedInt {
public:
  class Object {
  public:
    int *ptr;
    int count;
    Object(int *ptr_) : ptr(ptr_), count(1) {}
    ~Object() { delete ptr; }
  };
  Object *object;

  RefCountedInt(int *ptr) : object(new Object(ptr)) {
    cout << "constructor" << endl;
  }
  ~RefCountedInt() {
    cout << "destructor" << endl;
    object->count--;
    if (object->count == 0) {
      cout << "deleting object" << endl;
      delete object;
    }
  }
  RefCountedInt(const RefCountedInt &other) {
    cout << "copy constructor" << endl;
    object = other.object;
    object->count++;
  }
};
```

```cpp
void f(RefCountedInt ref_counted) {
  cout << "count: " << ref_counted.object->count << endl;
}

int main() {
  RefCountedInt ref_counted(new int(42));
  auto a = ref_counted;
  f(a);
}
// Output:
// constructor
// copy constructor
// copy constructor
// count: 3
// destructor
// destructor
// destructor
// deleting object
```

Ikke så farlig om du ikke forstår all koden i venstre kolonne, men se forskjellene på neste slide.

Mer om atomic-typer
- trådsikker referansetelling implementasjon med atomic

```cpp
#include <iostream>
using namespace std;

class RefCountedInt {
public:
  class Object {
  public:
    int *ptr;
    atomic<int> count;
    Object(int *ptr_) : ptr(ptr_), count(1) {}
    ~Object() { delete ptr; }
  };
  Object *object;

  RefCountedInt(int *ptr) : object(new Object(ptr)) {
    cout << "constructor" << endl;
  }
  ~RefCountedInt() {
    cout << "destructor" << endl;
    auto previous_count = object->count.fetch_sub(1);
    if (previous_count == 1) {
      cout << "deleting object" << endl;
      delete object;
    }
  }
  RefCountedInt(const RefCountedInt &other) {
    cout << "copy constructor" << endl;
    object = other.object;
    object->count++;
  }
};
```

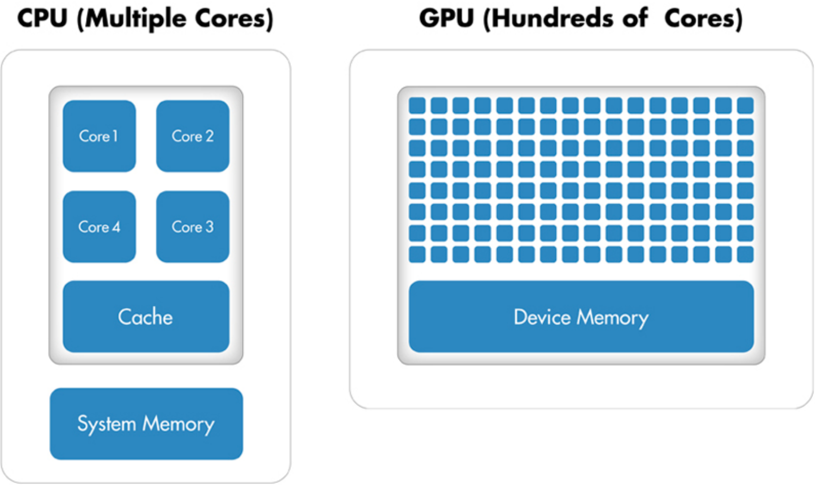
```cpp
void f(RefCountedInt ref_counted) {
  cout << "count: " << ref_counted.object->count << endl;
}

int main() {
  RefCountedInt ref_counted(new int(42));
  auto a = ref_counted;
  f(a);
}
// Output:
// constructor
// copy constructor
// copy constructor
// count: 3
// destructor
// destructor
// destructor
// deleting object
```

Legg merke til den spesielle funksjonen *fetch_sub*().

Oversikt

- Atomic-typer
- **CPU/GPU parallellisering**
- Prosesser

Parallellisering CPU vs GPU



CPU parallellisering
- skal parallellisere dette

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
  vector<int> a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  vector<int> b = {0, 1, 2, 0, 1, 2, 0, 1, 2, 0};
  vector<int> c(10);

  for (int i = 0; i < 10; i++) {
    c[i] = a[i] + b[i];
  }

  // c: 0 2 4 3 5 7 6 8 10 9
}
```

CPU parallellisering
- manuell tungvint løsning, kun CPU

```cpp
#include <iostream>
#include <thread>
#include <vector>

using namespace std;

int main() {
  vector<int> a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  vector<int> b = {0, 1, 2, 0, 1, 2, 0, 1, 2, 0};
  vector<int> c(10);

  vector<thread> threads;
  for (int thread_number = 0; thread_number < 5; thread_number++) {
    threads.emplace_back([thread_number, &a, &b, &c] {
      for (int i = thread_number * 2; i <= thread_number * 2 + 1; i++)
        c[i] = a[i] + b[i];
    });
  }

  for (auto &t : threads)
    t.join();

  // c: 0 2 4 3 5 7 6 8 10 9
}
```

Suboptimal CPU parallellisering - OpenMP (Open Multi-Processing)

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
  vector<int> a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  vector<int> b = {0, 1, 2, 0, 1, 2, 0, 1, 2, 0};
  vector<int> c(10);

#pragma omp parallel for
  for (int i = 0; i < 10; i++) {
    c[i] = a[i] + b[i];
  }

  // c: 0 2 4 3 5 7 6 8 10 9
}
// Compile with g++ and add the flag -fopenmp
```

CPU parallellisering
- std::algorithm før c++17 ingen parallellisering

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main() {
  vector<int> a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  vector<int> b = {0, 1, 2, 0, 1, 2, 0, 1, 2, 0};
  vector<int> c(10);

  transform(a.begin(), a.end(), b.begin(), c.begin(),
            [](int a_element, int b_element) {
    return a_element + b_element;
  });

  // c: 0 2 4 3 5 7 6 8 10 9
}
```

CPU(/fremtidig GPU?) parallellisering - std::algorithm c++17

```
#include <algorithm>
#include <execution>
#include <iostream>
#include <vector>

using namespace std;

int main() {
  vector<int> a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  vector<int> b = {0, 1, 2, 0, 1, 2, 0, 1, 2, 0};
  vector<int> c(10);

  transform(execution::par, a.begin(), a.end(), b.begin(), c.begin(),
            [](int a_element, int b_element) {
    return a_element + b_element;
  });

  // c: 0 2 4 3 5 7 6 8 10 9
}
// Compile using a newer g++ version with the flags: -ltbb -std=c++17
```

GPU parallellisering
- Komplisert OpenCL (Open Computing Language) kode

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int b[10] = {0, 1, 2, 0, 1, 2, 0, 1, 2, 0};
int c[10];

string kernel_code =
    "void kernel simple_add(global const int* a, global const int* b, global int* c) {"
    "   c[get_global_id(0)] = a[get_global_id(0)] + b[get_global_id(0)];"
    "}";

cl::Program program(/*her velges kernel_code og OpenCL parametere*/);
cl::Kernel kernel_add = cl::Kernel(program, "simple_add");

cl::CommandQueue queue(/*her settes OpenCL parametere*/);
cl::Buffer device_a(/*OpenCL parameter*/, CL_MEM_READ_WRITE, sizeof(int) * 10);
cl::Buffer device_b(/*OpenCL parameter*/, CL_MEM_READ_WRITE, sizeof(int) * 10);
cl::Buffer device_c(/*OpenCL parameter*/, CL_MEM_READ_WRITE, sizeof(int) * 10);
queue.enqueueWriteBuffer(device_a, CL_TRUE, 0, sizeof(int) * 10, a);
queue.enqueueWriteBuffer(device_b, CL_TRUE, 0, sizeof(int) * 10, b);
kernel_add.setArg(0, device_a);
kernel_add.setArg(1, device_b);
kernel_add.setArg(2, device_c);

//Programmet kjores på den valgte enheten (feks GPU):
queue.enqueueNDRangeKernel(kernel_add, cl::NullRange, cl::NDRange(10), cl::NullRange);
queue.finish();
queue.enqueueReadBuffer(device_c, CL_TRUE, 0, sizeof(int) * 10, c);

// c: 0 2 4 3 5 7 6 8 10 9
```

GPU parallellisering
- Boost.Compute (se eksempelet her)

```
#include <boost/compute/algorithm/transform.hpp>
#include <boost/compute/container/vector.hpp>
#include <iostream>

using namespace std;
namespace compute = boost::compute;

int main() {
  auto device = compute::system::default_device();
  compute::context context(device);
  compute::command_queue queue(context, device);

  vector<int> a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  vector<int> b = {0, 1, 2, 0, 1, 2, 0, 1, 2, 0};
  vector<int> c(10);

  compute::vector<int> device_a(a.size(), context);
  compute::vector<int> device_b(b.size(), context);
  compute::copy(a.begin(), a.end(), device_a.begin(), queue);
  compute::copy(b.begin(), b.end(), device_b.begin(), queue);

  compute::vector<int> device_c(c.size(), context);
  compute::transform(device_a.begin(), device_a.end(),
                     device_b.begin(), device_c.begin(), compute::plus<int>(), queue);
  compute::copy(device_c.begin(), device_c.end(), c.begin(), queue);

  // c: 0 2 4 3 5 7 6 8 10 9
}
```

Andre CPU/GPU parallelliseringsbiblioteker

- ArrayFire
  - Startet i 2014
  - C++ bibliotek
  - Støtter CUDA, OpenCL og CPU
  - Kan enkelt installeres med for eksempel pacman (Arch Linux / Manjaro) eller brew (MacOS)
- Kompute
  - Startet i 2020
  - C++ bibliotek
  - Støtter CUDA, OpenCL og Vulkan.
- 2021: Mangler dessverre enda slike biblioteker i Rust

Oversikt

- Atomic-typer
- CPU/GPU parallellisering
- **Prosesser**

Prosesser

- Tråder
  - Kjører i delt minneområde
    - Programmeringsfeil kan føre til at en tråd får tilgang til minneområdet til en annen tråd
  - Krasj i en tråd krasjer hele programmet
- Prosesser
  - Kjører i seperate minneområder
    - Sikrere mot programmeringsfeil, men kommunikasjon mellom prosesser er mer ressurskrevende
  - En krasj vil ikke påvirke andre prosesser

Prosesser, lese data fra prosess - eksempler med tiny-process-library

```cpp
#include "process.hpp"
#include <iostream>

using namespace std;
using namespace TinyProcessLib;

int main() {
  Process process("echo Hello World", {},
                  [](const char *bytes, size_t n) {
                    cout << string(bytes, n); // Output: Hello World
                  });

  cout << process.get_exit_status() << endl;  // Output: 0
}
```

Prosesser, lese data fra prosess - eksempler med tiny-process-library

```cpp
#include "process.hpp"
#include <iostream>

using namespace std;
using namespace TinyProcessLib;

int main() {
  Process process("cat nonexistent_file", {},
                  [](const char *bytes, size_t n) {
                    cout << string(bytes, n); // No output
                  }, [](const char *bytes, size_t n) {
                    // Output: nonexistent_file: No such file or directory
                    cout << string(bytes, n);
                  });

  cout << process.get_exit_status() << endl; // Output: 1
}
```

Prosesser, lese data fra *inline* prosess - eksempler med tiny-process-library

```cpp
#include "process.hpp"
#include <iostream>

using namespace std;
using namespace TinyProcessLib;

int main() {
  Process process([] { // Does not work on Windows
                       // where an executable file is needed
    cout << "Hello" << endl;
    cerr << "World" << endl;
    exit(10);
  }, [](const char *bytes, size_t n) {
    cout << string(bytes, n); // Output: Hello
  }, [](const char *bytes, size_t n) {
    cout << string(bytes, n); // Output: World
  });

  cout << process.get_exit_status() << endl; // Output: 10
}
```

Prosesser, *inline* prosess
- eksempler med tiny-process-library

Hva skjer her?

```cpp
#include "process.hpp"
#include <iostream>

using namespace std;
using namespace TinyProcessLib;

int main() {
  int a = 42;

  Process process([&a] { // Does not work on Windows
                         // where an executable file is needed
    a++;
    cout << a << endl;
    exit(0);
  }, [](const char *data, size_t n) {
    cout << string(data, n); // Output: 43
  });

  cout << process.get_exit_status() << endl; // Output: 0

  cout << a << endl; // Output 42
}
```

Prosesser, skrive til og lese data fra prosess - eksempler med tiny-process-library

```cpp
#include "process.hpp"
#include <iostream>

using namespace std;
using namespace TinyProcessLib;

int main() {
  Process process("cat", {},
                  [](const char *bytes, size_t n) {
                    cout << string(bytes, n); // Output: Hello World
                  }, nullptr /* no stderr */, true /* open stdin */);

  process.write("Hello World\n");
  process.close_stdin();

  cout << process.get_exit_status() << endl; // Output: 0
}
```