

Multithreading C++11. 3-р хэсэг

F.CS306 ПАРАЛЛЕЛ ПРОГРАММЧЛАЛ – Лекц 9

Г.ГАНБАТ
ganbatg@must.edu.mn

Хичээлийн агуулга

- ❖ Lock-Free програмчлал (Atomic, Compare-and-Swap)
- ❖ Work-Sharing Thread pool
- ❖ Parallel Graph Search

Lock-Free програмчлал

- ❖ **Lock бол удаан:** Thread pool-рүү их хэмжээний жижиг ажлууд дуудвал lock-д суурилсан хуваарилалтын логикоос шалтгаалан программыг бүхэлд нь цуваа ажиллуулж CPU ачааллыг гүйцэд ашиглахгүй байх нөхцөл үүснэ.
- ❖ Даалгавар нь Thread pool-ийн хуваарилалтын логикоос илүү тооцоолоход хугацаа зарцуулвал динамик хуваарилалтын утга алдагадана
- ❖ Lock-оос шалтгаалсан CPU дутуу ашиглалтыг багасгах практик арга бол алгоритмыг mutex-д хамааралгүйгээр дахин боловсруулах юм. Гэвч, уралдааны нөхцлөөс ангид байхын тулд дахин бичих нь үргэлж боломжтой байдаггүй.
 - ❖ Давталтын хувьд бүхэл тоог нэгээр нэмэгдүүлэх нь маш хөнгөн үйлдэл бөгөөд ийм энгийн зааварт үнэтэй lock ашиглах нь үндэслэлгүй байна.

Atomic Counting

- ❖ C++ 11 *atomic* өгөгдлийн төрөл: Хугацаа үрэх lock
эзэмшихгүйгээр параллел орчныг аюулгүй удирдана.
- ❖ CPU болон CUDA-enabled GPU гэх мэт орчин үеийн
боловсруулалтын нэгжүүд нь 32 ба 64 битийн бүхэл тоон *atomic*
тооцоололд зориулсан техник хангамийн түшний үр ашигтай
зааварчилгаатай.
- ❖ Мөн, x86_64 CPU нь 8, 16 эсвэл 128 битийн өгөгдлийн төрлүүдэд
atomic үйлдлийг дэмждэг.
- ❖ Эдгээр тоног төхөөрөмжийн зааварчилгаа нь нэг хувьсагчийн
нэмэгдүүлэх/багасгах *atomic*, эсвэл хоёр хувьсагчийг
тасалдалгүйгээр солилцох *atomic* боломжийг олгодог.

Mutex VS Atomic

```
auto lock_count =
    [&] (volatile uint64_t* counter,
          const auto& id) -> void {

    for (uint64_t i = id; i < num_iters; i += num_threads) {
        std::lock_guard<std::mutex> lock_guard(mutex);
        (*counter)++;
    }
};

auto atomic_count =
    [&] (volatile std::atomic<uint64_t>* counter,
          const auto& id) -> void {

    for (uint64_t i = id; i < num_iters; i += num_threads)
        (*counter)++;
};
```

} уралдааны нөхцлөөс сэргийлж тоолуурыг түгжигдсэн хүрээнд удирддаг уламжлалт арга.

std :: atomic<uint64_t> гэсэн *atomic* өгөгдлийн төрлийг ашиглан параллел нэгээр нэмэх үйллдлийг гүйцэтгэдэг арга

Compile

- ❖ Кодыг C ++ 14 дэмждэг compiler-aap -latomic ашиглан хөрвүүлнэ:

```
g++ -O2 -std=c++14 -pthread -latomic \atomic_count.cpp -o atomic_count
```

- ❖ Дээрх жишээг Xeon E5-2683 v4 CPU-дээр 10 thread ашиглан ажиллуулахад 7 дахин үр ашигтай байсан

Xeon E5-2683 v4 CPU:10 thread

#elapsed time (mutex_multithreaded) :	16.0775s
#elapsed time (atomic_multithreaded) :	2.25695s
100000000 100000000	

Өгөгдлийн atomic төрлүүд

- ❖ C ++ 11 нь 8, 16, 32, 64 битийн суурь тоон төрлийг дэмждэг.
- ❖ *std :: atomic <T>* -ээр 24, 32, 48, 128 гэх мэт урттай бүтэц тодорхойлж болно. Ижил API-ийн хүрээнд уралдааны нөхцөлгүй нэгдсэн хандлагатай.
 - ❖ Суурь бус бүтцийн хувьд техник хангамж дэмжихгүй учир Lock ашиглана.
- ❖ Байгуулагч функцийг зөвшөөрдөггүй. Гишүүн өгөгдөл хэрэглэх бол объектийг *std::atomic* болгохоосоо өмнө нэмэлт гишүүн функцийн хэрэгжүүлнэ.
- ❖ *Atomic* бүтцийг тасалдал хэрэглэлгүй ачаалах, хадгалах боломжтой зэрэгцээ байтууд гэж үзнэ.

Atomic параллел үйлдэл

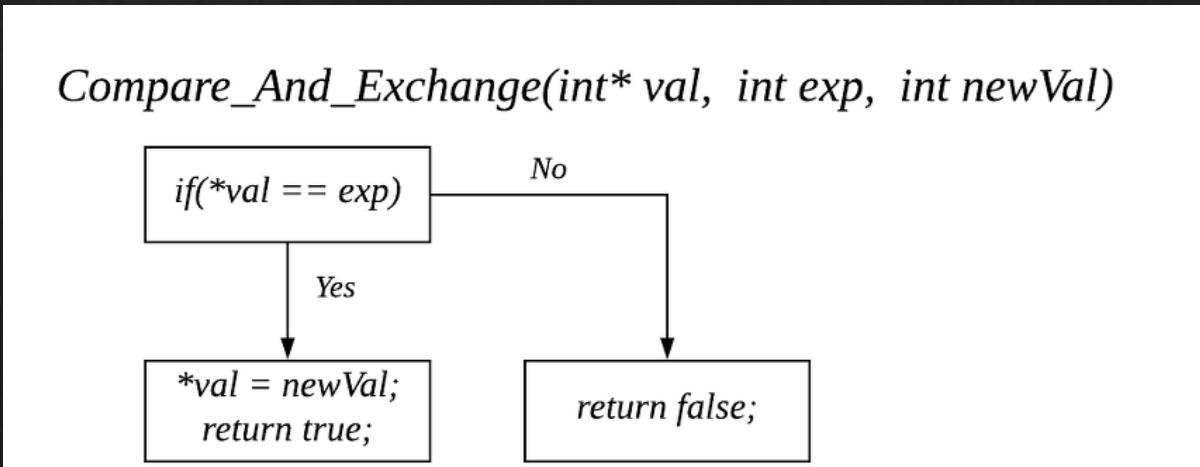
- ❖ *atomic++*; нь ачаалах, нэмэгдүүлэх, хадгалах ажлыг нэг үйлдлээр гүйцэтгэнэ.
 - ❖ Бүхэл тоо: *operator++/operator--*,
 - ❖ Бит: *operator+=, operator-=, operator&=, operator|=, operator^=*
- ❖ C ++ 11-ын *atomic* төрлүүд нь *compare-and-swap (CAS)* үйлдэл хийх боломжтой. Жнь:

atomic.compare_exchange_weak(T& expected, T desired):

1. *Atomic*-т хадгалагдсан *expected* утгын хаяг
 2. *atomic.load() == expected* нөхцлийг хангах *atomic-t* хадгалах утга
- ❖ CAS үйлдэл зөхөн давталтан дотор хэрэгжинэ.
 - ❖ Thread нь *atomic.compare_exchange_strong* функц хэрэглэн *atomic*-ийг дангаараа удирдах тохиолдолд зөрчиж болно.
 - ❖ *Atomic* бүтцүүд дээр *atomic++*; төрлийн үйлдлүүдийг CAS давталт ашиглан хэрэгжүүлнэ.

CAS үйлдлийн алхамууд

- ❖ CAS нь дараах гурван үе шатыг *atomic* хэлбэрээр тасалдалгүйгээр гүйцэтгэдэг:
 1. Θгөгдсөн *expected* утгыг *atomic* дахь утгатай жишигээшнэ.
 2. Хэрэв өгөгдсөн *expected* болон *atomic* дахь утгууд ижил бол *atomic*-д өгөгдсөн *desired* утгыг олгоно, үгүй бол *expected* утгыг *atomic* дахь утгаар солино.
 3. 2-р алхам амжилттай бол үнэн, үгүй бол худал утгаана.



Дарааллаас max утгыг олох

```
// WARNING: this closure produces incorrect results
auto false_max =
    [&] (volatile std::atomic<uint64_t>* counter,
         const auto& id) -> void {

    for (uint64_t i = id; i < num_iters; i += num_threads)
        if(i > *counter)
            *counter = i;
};

// Using a compare and swap-loop for correct results
auto correct_max =
    [&] (volatile std::atomic<uint64_t>* counter,
         const auto& id) -> void {

    for (uint64_t i = id; i < num_iters; i += num_threads) {
        auto previous = counter->load();
        while (previous < i &&
               !counter->compare_exchange_weak(previous, i)) {}
    }
};
```

- ❖ *false_max*: Нөхцөл шалгах, atomic-д хадгалах нь хоёр бие даасан үйлдэл учир atomic үйлдэл гүйцэтгэгдэхээс өмнө энэ утгыг олон thread уншчих боломжтой. Тиймээс 999,999,999 гэсэн үр дүн гарахгүй байх боломжтой

ABA асуудал

- ❖ 0-ийг 1 болгох atomic үгүйсгэл тооцоолоход гарч болох :
 1. Thread 0 нь 0 утгатай atomic уншиж, 1 болгон өөрчлөхийг оролдох ба CAS хараахан гүйцэтгэгдээгүй байна.
 2. Thread 1 нь 0 утгатай atomic уншиж, 1 болгон өөрчилнө.
 3. Thread 1 нь 1 утгатай atomic уншиж, 0 болгон өөрчилнө.
 4. Atomic нь 0 *expected* төлөвтэй тул thread 0 нь CAS-ийг гүйцэтгэнэ.
- ❖ 4-р алхамд, Thread 0 нь төлөв 1 ба 3-р алхамын алинд өөрчлөгдсөнийг ялгаж чадахгүй. 4-р алхмын CAS нь 2 ба 3-р алхамд хийгдсэн өөрчлөлтүүдийг мэдэхгүй. Иймээс синхрончлох зорилгоор энэ төлвийг ашиглаж болохгүй. (Lock-based аргаар thread 1 нь lock эзэмшиж амжихгүй учраас 1 ба 4-р алхмыг тасалдалгүйгээр гүйцэтгэнэ.)

ABA асуудлын шийдэл

- ❖ Олон thread lock эзэмших гэж оролдох үед thread бүр тасалдлын дараа л эзэмшинэ. Тиймээс thread бүр өгөгдөл дээр өөрийн өөрчлөлтийг бүрэн хийж чадна.
- ❖ Төлөв хэдэн удаа өөрчлөгдсөнийг *atomic* тоолдог зориулалтын хувьсагч оруулж өгснөөр энэ асуудлыг шийдэж чадна.
- ❖ Хэрэгтэй өгөгдөл болон харгалзах тоолуурыг нэг *atomic* өгөгдлийн төрөлд багтаах ёстой.
 - ❖ Бүхэл тооны эхний битээс хэрэгтэй өгөгдлийг байрлуулж, Үлдсэн битүүдийг тоолоход ашиглаж болно.
- ❖ Динамик массив, зэрэгцээ hash map зэрэг lock-free өгөгдлийн бүтцийн ашигтай загварт тулгамдсан асуудал болж байна.

Work-sharing Thread Pool

- ❖ *Нөөц ашиглалтыг сайжруулахын тулд:*
 - ❖ Идэвхтэй thread-ын тоо c багтаамж (хуваалтын хэмжээ) -аас бага байхад даалгавар гүйцэтгэж буй thread бүр (эхэндээ нэг) ажлын ачааллыг бусад сул (idle) thread-уудтай хуваалцдана.
- ❖ Хоёртын модны рекурсив тойролт ашиглан ажлыг *thread*-үүд дунд динамикаар хуваарилах:
 - ❖ Parallel tree traversal. Модны оройгоос эхлэн сул thread үлдэхгүй болтол рекурсивээр давтаж зангилаа бүр дээр ажлыг хагаслан хуваана.
 - ❖ Ажлаа хуваарилах оролдлого бүртээ даалгавруудын тоог авч нэгж хувьсагчид авч Pool-ийн багтаамжтай харьцуулна. Lock эсвэл *atomic* хэрэглэнэ.
 - ❖ Thread pool-ийн төгсгөлд синхрончлох зорилгоор өөр нэг condition variable хэрэгтэй.

Parallel tree traversal.

```
ThreadPool TP(8);
void waste_cycles(uint64_t num_cycles) {
    volatile uint64_t counter = 0;
    for (uint64_t i = 0; i < num_cycles; i++) counter++;
}

void traverse(uint64_t node, uint64_t num_nodes) {
    if (node < num_nodes) {
        waste_cycles(1<<15); // ямар нэг ажил гүйцэтгэнэ.

        // боломжит сүл thread-ийг ашиглан зүүн мөчрийг
        // ажилуулахаар оролдоно.
        TP.spawn(traverse, 2*node+1, num_nodes);

        //баруун мөчрийг дараалуулан ажилуулна.
        traverse(2*node+2, num_nodes);
    }
}

int main() {
    TP.spawn(traverse, 0, 1<<20);
    TP.wait_and_stop();
}
```

Parallel graph search

- ❖ Онолын хувьд шугаман хурдсэлт хүрч чадах хамгийн сайн хурд юм. Гэсэн ч заримдаа сайн кэшлэлтийн чадавхиас үүдэлтэй хэт *superlinear* хурдсалт үзүүлж болно
 - ❖ хэрэв бид асар том матрицын локал хуулбарууд дээр ажилладаг бол боловсруулсан өгөгдлийн хангалттай сайн locality хийх учраас cache hits илүү их болно.
 - ❖ dual socket CPU: нэг CPU-нээс кэшлэлт хоёр дахин их.
- ❖ branch-and-bound алгоритмууд нь HW архитектурын давуу талаас хамаarahгүй зохиомол *superlinear* хурдсалтын боломжтой.
 - ❖ The Traveling Salesman Problem, Binary Knapsack Problem

Баярлалаа.