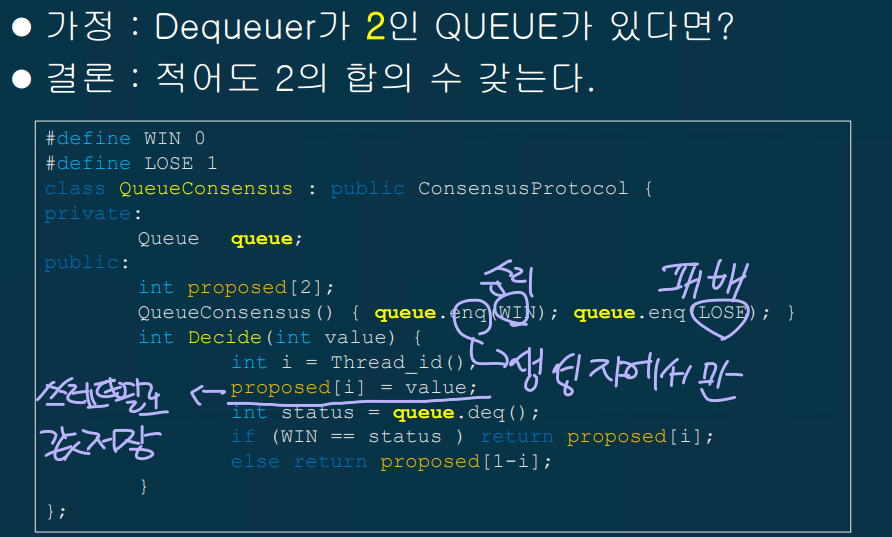
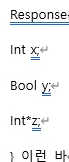
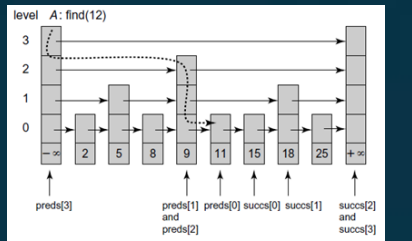
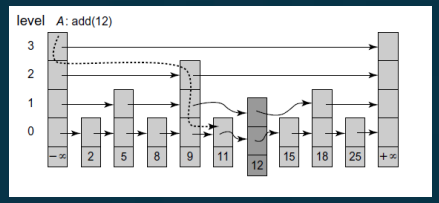
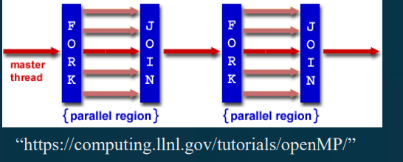
* 배경 이론
  + 합의
    - 새로운 동기화 연산을 제공하는 가상 객체
    - 동기화 연산: decide
      * N개 스레드가 호출해도 문제없어야 함
      * 여러 번이 아니라 스레드당 한번 이하로 만 호출함
      * 모든 호출에 대해 같은 값 반환, 전달된 값중 하나가 반환
      * 가장 먼저 리턴한 스레드의 값으로 통일하면 된다,wait free만족하면 두번째도 상관없음
    - 모든 스레드가 같은결론, wait-free
    - 여러 경쟁 스레드중 하나를 선택할 때 주로 사용
  + 합의 수
    - 합의 객체를 갖고 멀티스레드 알고리즘을 구현했을 때 n 스레드 합의 문제중 최대 의 n을 말한다,
    - 최대 n이 존재 하지않으면 무한하다라고 함
    - 0,1 : 있으나마나
    - 2: 2개쓰레드 , 3개불가
    - 무한대:가장 파워풀
  + Decide 알고리즘의 실행
    - 동시 실행 ,return 값 결정
    - 공유 메모리 read write
    - 실행 순서에 따라 공유 메모리에 저장되는output값이 달라짐
    - 같은 순서라면 항상 같은 값이 나옴
    - 모든 실행 경로는 이진트리로 나타낼 수 있음
    - 이진트리는 무한대가아니라 크기가 정해짐(wait free)
    - CAS를 사용하면 이진트리 결과가 같다(2개 쓰레드)
    - 3개쓰레드는 3진트리가 될것임
    - 이동이 아니면 결과에 영향을 미치지 않음
    - 초기 상태 : 아무런 이동이 없는 경우
    - 최종 상태 : 모든 스레드가 이동을 마친 상태’
    - 일가(univalent)상태 : 앞으로 어떠한 이동을 하더라도 결정값의 변화가 없다
    - 이가(bivalent)상태 : 최종 결정 값이 결정되지 않음
    - 임계(critical)상태: 현재 상태가 이가임, 다음 이동으로 무조건 일가 상태가됨
  + 증명
    - 명제 : atomic 메모리로 2쓰레드 합의 문제를 해결
    - 임계상태 반드시 존재
    - 하나만 read
      * read인지 알수 없다
    - 둘다 쓰기인데 다른 공유 메모리
      * 누가먼저 실행했는지 알수 없다
      * 똑 같은 메모리여야함
    - 둘다 쓰기인데 같은 공유 메모리
      * A의 입장에서 덮어썼는지 구분할 수 없다
    - 모든 경우가 모순이다, 따라서 2스레드 합의가 불가능하다
    - 결론: atomic 메모리의 read/write 만으로 두개 스레드 합의 non blocking 구현 불가능
    - 무엇이 필요한가? Queue객체가 필요함
    - 가상의 Queue 가정
      * 2개 쓰레드에서 동시에 deque햇을 때 atomic하고 wait free 하게 동작
      * 
      * 합의 수는 적어도 2
    - Atomic 메모리 는 합의 수1, 이것만으로는 무대기 구현 작성 할 수 없다
    - N thread fifo queue 합의수는 2, 3개스레드는 될까?
      * 둘다 deq
        + C는 누가 먼저 deq인지 구별 불가
      * Deq enq
        + 실행 순서 관계읪이 똑같다 - > 구별불가
      * 둘다 enq
        + A,b
        + B,a
        + deque를해야 판단이가능하다, 꺼내야 알수있음, 근데 그 결과를 다른 스레드에게 전달할 방법이 없다
        + 공유메모리에는 결과가 적혀저있지않다 쓸시간도없이 멈추기때문에
    - 따라서 3개쓰레드는 불가능, 합의 수는 2
  + 합의 수가 1인건 = 공유 메모리 읽고 쓰기
  + 합의 수 2인 것 큐
  + 3인건 다중 대입 객체
    - 배열로 구성, 복수의 원소를 atomic하게 변경
    - Write 대신 assign사용 (배열크기 n 바꾸는 값 m개)
    - I ,i+1에 값을 씀
    - Read는 인덱스를 받아 i번째값 반환
    - One write multi read인 스냅샷(합의수 1)과 반대이다
    - 읽을때 배열 전체 값을 원자적으로 읽음
    - 2스레드 2진합의 풀수있다 먼저 실행된 스레드 값을 핪의 값으로 하면되는데 다른 스레드가 이걸 판단할 수있다
    - 텍스트, 거리, 실외, 표지판이(가) 표시된 사진

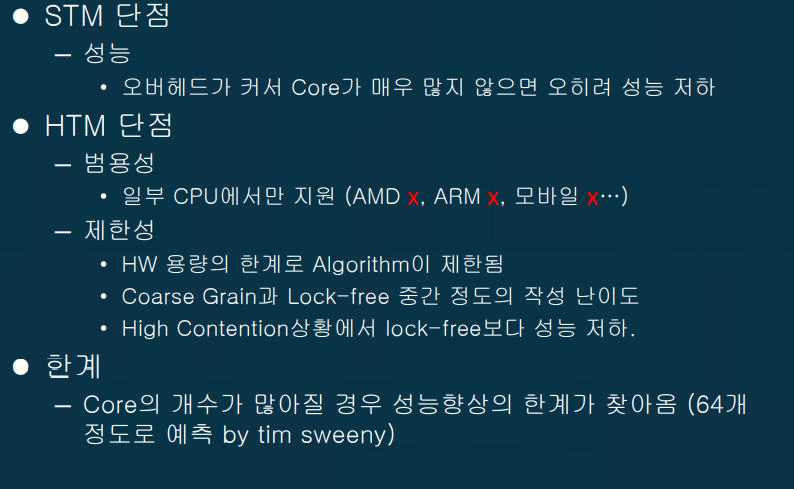
      자동 생성된 설명
    - 원자적인 (n,n(n+1)/2)- 대입객체는 최소 n(>1)의 합의수 가짐
    - 대입 객체 hw 지원시 합의 문제 무대기 해결, 그러나 오버헤드가 너무커서 못쓴다
    - 대안은 RMW연산 (+=) Read Modify Write
      * GetAndSet
      * GetAndIncrement
      * GetAndAdd
      * CAS
      * Get() / 항등함수
      * 항등함수가아닌 rmw는 명백하지 않다 고함
    - 명백하지 않은 rmw는합의수최소 2, rmw가 INIT을 읽으면 내가 이긴 것, 못읽으면 진 것, 하드웨어 도움이 필요
    - Common2 Rmw 연산
      * 연산 순서를 바꾸거나 한 함수가 다른 함수를 덮어쓰는 경우를 말함
      * 합의 수 2 따라서 최근 프로세스는 제거되는 추세
      * getAndSet: 덮어씀
      * 3스레드 경우 승자 패자를 알 수 있어야하는데 Common2 연산은 교환 덮어쓰기가 가능해서 판단이 불가능해서 합의 수가 2가 되는것임
      * 따라서 Common2가 아닌 RMW가 필요함
    - CAS 연산
      * CAS는 무한대의 합의 수를 갖음
      * CAS 실패하면 누가 이겼는지 알 수있다
      * N개의 쓰레드에서 논블러킹 자료구조를 구현할 수 있는가?
  + ??
    - 어떤알고리즘을 wait free롤 만들 수 있는가?
    - 적은 합의 수를 갖는 객체로 큰 합의 수를 갖는 객체를 구현할 수 없음, 모든 무대기 동기화 객체는 합의 수 능력 차이가 있음
    - 증명을 통해서 불가능한 시도를 하는걸 미연에 방지할 수 있음(원자적 메모리를 가지고 4개 스레드 무대기 병렬 큐를 작성)
    - 모든 자료구조에 무대기 동기화가 가능? 가능하다
      * 만능 객체 : 어떠한 객체든 무대기 병렬 객체로 변환시킴
    - 만능
      * 클래스 C 객체들과 원자적 메모리로 모든 개개체를 무대기 구현으로 변환하는 것이 가능하다면 클래스 C는 만능이다.
      * 모든 객체의 메소드가 다양하기 때문에 변형,변환 시켜야함 (파라미터, 리턴값을 통합)
    - 무잠금 만능
      * 순차 객체 A가 있고 이를 n 스레드에서 무대기 구현하려하면 A는 결정적이여야 함
      * 모든 객체의 초기상태는 같고 똑 같은 인풋을 넣었을 때 항상 같은 값을 넣어야함
      * 같은 입력 값을 동일 순서로 입력 시 항상 같은 결과를 얻는 조건을 이용해 따로 저장한다 ( = 로그,log)
      * 순차 객체 A – 병렬화 하고자하는 객체를 감싼 객체, 호출 메소드를 apply 하나로 통일텍스트이(가) 표시된 사진

        자동 생성된 설명
      * Invocation - 호출하고자 하는 원래 객체의 메소드와 그 입력 값을 갖는 객체, 이 객체는 메소드를 지시하는 아이디가 있고 모든 파라미터 조합을 다 넣어서 통합함
      * Response – 여로 메소드들의 결과 타입을 압축한 객체 이런 느낌
    - Log
      * Log는 node들의 리스트이다
      * Consensus는 합의 객체로 데이터 레이스가 없도록 해준다
    - Log를 락프리로 관리하는 것이 핵심
    - 무잠금 만능의 구현
      * 텍스트, 표지판이(가) 표시된 사진

        자동 생성된 설명
    - 노드마다 합의 객체를 갖고 있다, 합의 객체로 한 노드의 넥스트는 어떤 스레드에서 유일함을 보장함
    - 순차객체는 매호출마다 새로생성되고 다른 스레드는 절대 그 순차객체를 호출하지 않는다
    - 무대기 만능
      * 무잠금 만능보다 좀 더 복잡하다
      * 합의할 때 자기대신 우선순위가 높은 다른 스레드 노드를 합의(Helping)
      * 모든 스레드가 같은 헬프를 해줘야함 -> Node의 Seqnumber & MAX\_Thread를 help함
    - 쓰레드 a,b가 동시에 노드 a를 추가할 수 있는가
      * 같은 자리는 덮어쓰기라상관없고
      * head[]가 정해 지는 시점에서 a.seq != 0 – anounce.seq나 help.seq 검사를 통과할 수가 없다.
      * Head[i] = announce[i]는 문제없나? Decide가 알아서 하므로 문제없음
* 병렬 알고리즘 – QUEUE
  + 풀, 큐, ㅇ
  + 풀
    - 리스트는 Set 객체
    - Queue와 stack은 pool객체
    - Pool 객체
      * 같은 아이템의 복수 존재를 허용
      * Contains 메소드를 항상 제공하지 않음 -> 비효율적이기 때문
      * Get()과 Set()을 제공 (remove,add)
      * 보통 생산자-소비자 문제의 버퍼로 사용 ?? (하나의 스레드는 만들고 하나의 스레드는 꺼내간다)
  + 풀의 종류
    - 길이제한
      * 있다 – 제한 큐 (구현하기 쉬움,생산자 소비자 간격 제한)
      * 없다 – 무제한 큐(이거를 구현해보자)
    - 메소드의 성질
      * 완전(total): 비어있는 풀에서 get할 때 실패 반환(이걸 구현해보자)
      * 부분적(partial): 비어있는 풀에서 get할 때 다른 누군가가 set할때까지 기다림(이건 블로킹이다)
      * 동기적: 버퍼의 크기가 0임 enque가없으면 deque를 못함 -> 다른 스레드의 메소드 호출의 중첩이 필요로함, 랑데부라고한다 (당연히 블로킹)
  + 큐
    - 무제한 완전 큐
    - Coarse Grain
      * Locking을 사용해 구현해보기
      * Head pointer에서 deq
      * Tail pointer에서 enq
      * 초기형태는 비어있는 큐
      * 무잠금, CAS로 구현해보기
      * CAS가 성공하고 tail이 전진하지 않으면 enqueue 실패하므로 모든 다른 스레드가 기다려서 블로킹이되버림
      * 전진 시에도 CAS를 사용
      * Dequeue는 값을 CAS전에 읽어야함 CAS성공하면 제대로된 값이다
      * Tail이 전진되지 않은상태에서 enqueue 가 들어오는문제
        + Tail 전진을 dequeue 전에 보조한다
      * 또 하나 주의할 것은 tail과 head 비교할때임 volatile 붙여라
      * 그럭저럭 돌아가나 오동작이 발생한다
      * Delete를 안하면 안죽음
    - ABA 문제
      * CAS를 속인다, A가 B가 됬다 A가됨
      * 굉장히 골치아픈 문제
      * Version number를 붙여서 함께비교한다 (=스탬프)
      * LL SC 명령을 쓰면 변경여부를 검사하기 때문에 ABA를 해결 가능, 그러나 최신 CPU가아니면 명령어가 없다
      * EBR, Hazard Pointer를 사용하면 해결가능
    - Time Stamp Version
      * 128bit CAS
* 병렬 알고리즘 – STACK
  + 연결리스트로 구성, top필드가 첫노드를 가리킴
  + 스택이 비엇으면 nullptr
  + Push, pop
  + 성긴 동기화 스택
  + LF Stack
    - CAS호출 순서로 하나씩 진행되어 순차 병목현상이 나타날수 있음 병렬성이없다
    - ABA문제 Queue보다 생길확률 큼 -> stamp를 쓰거나해야함
    - CAS 동기화문제
      * 경쟁이 심하면 CAS실패로 인한 성능 저하가 커진다
  + Back Off
    - 경쟁이 심하면 경쟁을 줄이자
    - CAS가 실패했을 경우 적절한 기간 동안 실행을 멈췄다가 재개 (경쟁 빡세니까 좀 쉬었다가하자)
    - 적절한 기간
      * 처음에 짧게, 실패할때 마다 점점 길게, 첫 시도에 성공 시 짧게
      * Thread마다 기간을 다르게(같으면 다같이 충돌한다)
    - 간단
    - 병렬성 감소
    - 시스템 콜을 쓰지않는 경우
      * RDTSC CPU명령어 사용
      * eax에 dec명령어 사용하여 loop를 돌린다
      * asm을 사용하기 때문에 x64로 돌릴수 없는 단점이 있다
    - 코어 수가 많고 쓰레드가 많아서 CAS가 실패하는 경우가 많아지는 경우 높은 성능향상을 얻을 수 있다
  + 소거
    - 스택에 넣지 말고 직접 data를 주고 받자
    - 주고받는 별도의 객체를 복수로 준비
    - Push pop이 동시에 실행되면 연산이취소되고 스택은 접근하지 않음
    - Elimination Array
      * 임의(오버헤드가 없다) 원소를 선택, Array 크기는 가변
      * 상위 2비트를 state를 저장하고 나머지를 v를 저장해서 2^62보다 작은 값만 저장가능
    - ABA문제
      * 해당 알고리즘은 aba 문제를 일으키지 않음
      * 교환대상이 바뀐 것 뿐이고 값도 그대로여서 아무 문제가 없음
    - 소거된 연산은 절대로 스택에 접근하지 않기 때문에 LockFreeStack 에서의 경쟁이 줄어든다
    - LF\_STACK보다 성능은 좋지않다, 최적화가 필요함
* SKIP\_LSIT
  + 순차 스킵 리스트
    - Find
      * 언제나 시작은 head, 높은 레벨 포인터부터 검색, 한 레벨 검색이 끝나면 다음 레벨 검색 시작 맨아래 레벨 도달 시 종료
      * 
    - Add
      * 랜덤한 높이의 노드 생성, 검색한 위치에 추가,pred[] curr[]필요
      * 
    - 병행 스킵리스트
      * Mark Field를 사용한 Lazy방식
      * 여기서 스킵리스트는 n개를 연결시켜야 추가가 되었다고 판단하기 때문에 fullylinked 변수를 사용한다
      * Std::recursive\_mutex 필요, 하나의 노드를 여러 번 잠글 수 있다, 락 횟수만큼 언락해야 됨
    - LF\_SKIPLIST
      * CAS사용
      * 물리적 제거는 Find에서
      * 삭제시 노드의 모든 next에 표시함
      * Add : 추가되는 시점은 0층이 연결되었는가로 판별(fullylinked는 사용불가), find시 0층이 연결되었는가를 여부를 true false로 리턴함
      * Remove: 최하층 만 빼고 표시를 남김, 전부 표시했으면 최하층의 next 참조에 표시, 이게 성공하면 find로 물리적 제거함
* 병렬 라이브러리
  + C++11
    - 멀티스레드 프로그래밍 API의 표준화
    - 스레드에 람다로 넘길수 있지만 람다를 쓰는 것은 기능이 작다는 것이고 그러면 굳이 멀티스레드 프로그래밍을 할 이유가 없다
    - Lock\_guard, 락이 되고, 블록을 빠져나가면 자동 언락됨, 객체이기 때문에 블록을 빠져나가면서 소멸자에서 언락을 해주는 방식이다.
    - 텍스트, 스크린샷, 실외이(가) 표시된 사진

      자동 생성된 설명
    - 운영체제 호출이라서 다루지않은 명령어가 많다,오버헤드가 크거나 블록킹구현임
  + OpenMP
    - 병렬프로그램을 가능하게 해주는 API
    - 컴파일러 디렉티브, 함수, 변수로 구성
    - 분산 메모리에서는 사용할 수 없다, 다른 컴퓨터에서 실행될수 없음
    - Data Race, Dead Lock,Data Dependency 검사등은 프로그래머가 해야함
    - 병렬화를 어떻게 어느부분을 할지를 프로그래머가 지정해야함
    - Visual studio는 프로퍼티에서 언어탭에서 설정할 수 있다
    - Fork-Join 모델
    - 컴파일러 디렉티브에 의존
    - Nesting 가능 (이중으로 병렬화)
    - 동적 쓰레드 할당
    - 메모리 일관성은 보장하지않음,FLUSH(atomic\_defence\_fence)사용
    - #pragma omp, 디렉티브 뒤에는 방드시 Block이와야함
    - #pragma omp critical로 data race를 막을 수 있다
    - 작업 : 병렬성을 지정하는 프로그램 단위
    - 작업 분배 지정
      * Do/For : 루프를 여러 쓰레드가 나누어 수행, data dependency를 검사하지 않음
      * SECTIONS: 블록으로 나누어진 작업들을 여러 쓰레드가 나누어 수행
      * SINGLE: 쓰레드 하나가 전담 수행
    - #pragma omp atomic  
      (atomic\_int와 유사)
    - Openmp로 쓴다고 해서 딱히 장점이 없다, 멀티스레드프로그래밍 기존문제를 그대로 갖고있다.
  + Intel Thread Building Block (TBB)
    - CPU판매를 위해 만듬, 최적화가 정말 잘 되어 있어 많이 사용한다.
    - Task 관리 기능 포함
    - Intel CPU에서 동작하며 최적화됨(AMD는 보증하진 않지만 잘 돌아갈 것임)
    - 최근 OneAPI 프로젝트에 통합됨
    - C++ 11과 겹치는 내용이 많으므로 유의해야한다
    - TBB의 기능들
      * Loop Parallelizer:   
        #pragma를 사용하지 않고 고유의 함수를 사용
        + 변형이 필요함
        + 루프의 범위를 지정할 수 있어야함
        + Operator를 등록해야함
        + TBB용 클래스로 변환(blocked\_range만큼만 실행하는 객체)
        + Parallel\_for를 호출 -> C++11이 더간단한 인터페이스를 제공한다
        + 텍스트이(가) 표시된 사진

          자동 생성된 설명
      * Containers:  
        STL과 유사한 형태의 멀티 쓰레드 넌블러킹 컨테이너를 제공함(LockFree)
        + Concurrent 시리즈
      * concurrent unordered map
        + erase가 thread\_safe 하지 않음
      * concurrent\_hash\_map
        + find,insert,remove제공
        + 사용법이 좀 특이해서 위에 보다 호환성이 떨어짐
        + 모든 자료 접근은 accessor를 통해 이루어짐 (일종의 스마트 포인터,shared\_ptr)
        + 읽기만하면 const\_accessor를 사용하는 것이 좋다
      * Concurrent\_vector
        + Push\_back,grow\_by,gro\_to\_at\_least,size
        + Clear는 불가능
        + pointer연산은 불가능하다
        + 원소를 읽을 때 생성중일 수 있으므로 생성완료를 학인하도록 프로그래밍해야함
      * Concurrent\_queue
        + Push,try\_pop
        + Try\_pop제공 이유는 멀티스레드에서 안전하게 pop 할 수 없기 때문
      * Mutual Exclusion (지원에서 제거)
        + C+11에 있는mutex를 사용하도록 변경
        + 굳이 tbb것을 사용할 이유는 없다
        + C++17 scoped\_lock과 tbb scoped\_lock은다르니 주의
        + RWlocking 지원,읽기만하는 락 구분하는것 -> 성능향상,c++14에서 추가됨  
          오버헤드가 크기 때문에 Read비중비 클 때 성능향상이 있다
        + Scalable: busy waiting을 없애 cpu낭비를 막음, 운영체제호출로 오버헤드큼
        + Fair: 도착한 순서대로 락을 얻음
        + Recursive: 같은 스레드는 락을 다중으로 얻음
        + Long wait: 오래기다리는 경우,   
          yield – 같은 프로세스 의 다른 스레드 실행  
          block – 깨워 줄 때 까지 멈춤
        + Shared\_lcok
      * 메모리 일관성 지시(지원에서 제거)
      * 메모리 할당자:  
        멀티스레드에서 new delete 시에 제대로된 성능이 나오는가  
        -> 멀티스레드에서 고성능을 내는 메모리 할당자가 있다
        + 기본 동적 메모리 할당 함수들은 intel TBB의 할당 함수들로 자동적으로 대체함
        + Window - #include “tbb/tbbmalloc\_proxy.h”
        + 성능비교시 똑같다
      * Task 스케줄링
        + 게임은 쓸일이 별로없음
        + 작업을 여러 개로 Task로 나누어서 병렬로 처리
  + CUDA
    - Computer unified Device Architecture
    - 대규모 병렬 처리를 GPU에서 수행
    - GPU가 CPU보다 몇백배 빠름
    - NVIDIA 하드웨어만 지원
    - 대안
      * DirectCompute: DirectX의 일부분 GPU 벤더에 상관없이 동작
      * OpenCL: Apple이 Mac OSX에 구현하고 공개,AMD에서 잘동작
    - 게임업계에서는 잘 쓰지않음, 렌더링이 느려진다
    - 1. 비디오메모리에 데이터를 카피함  
      2.GPU가 실행할 프로그램을 올림  
      3.실행함  
      4.결과를 꺼낸다
    - 데이터를 올리는데 오버헤드가 크다, 따라서 데이터를 올리는것보다 계산하는 양이 압도적으로 많아야 성능향상이 이루어진다
    - \_\_global\_\_ 를 붙이면 GPU에서 실행된다
    - 직관적이다
    - OpenCL
      * Nuget이나 window설치파일을다운받아 실행
  + GPGPU
    - 장점
      * CPU의 몇 십 배 속도가능(계산량이 많으면)
    - 단점
      * 반대로 계산량이 적으면 속도가 CPU보다 느리다
      * 낮은 IO 및 직렬 계산속도(CPU, GPU병목현상)
      * 싱글 스레드면 훨씬 느리다
      * 적은 메모리
  + 지금까지
    - 블러킹
      * 직관적이고 의도대로 잘동작함
      * 병렬성이 없다 -> 성능 개선여지가 없음
      * 의도하지 않은 멈춤 현상( 우선순위역전, 컨보잉)
      * 데드락에 빠지기 쉽다.
      * 성능 때문에 복수 lock 사용, 주석에 의존
    - 논 블러킹
      * HW도움으로 wait -free CAS연산 사용
      * lock으로인한 멈춤 현상 회피가능
      * 알고리즘이나 자료구조 설계가 매우 어렵다
      * CAS 근본적인 문제
        + 연산단위가 Word
        + 여러 개 Word 변경을 원자적으로 할 수 있으면 알고리즘 구현이 쉬워짐
    - MultiCAS?
      * 그런것은 존재하지않음
      * singleCas를 사용해서 구현해야함, 그러나 어렵다
    - 락프리의 한계
      * 확장성이 떨어짐, 기존 메소드를 전부 수정해야함
      * 메소드의 연속동작을 atomic하게 구현하는것이 어려움
    - 자료구조의 정확성
      * 락프리 알고리즘의 정확성을 증명하는 것은 매우 어려움
  + 트랜잭션
    - 지금까지 단점을 보완하기 위한 프로그래밍 모델
    - 하나의 스레드가 실행하는 일련의 프로그램 블록 => 트랜잭션으로 정의
    - 각각의 트랜잭션은 Atomic하다
    - 투기적(speculative) 실행(안되면말고)
    - 하나의 트랜잭션에 속하는 모든 메모리 연산은 임시적(tentative) 실행
    - 트랜잭션의 실행이 끝난 후 동기화 충돌 검사 (혹시나 같은 메모리를 읽고 썻나 검사)
    - Commit: 충돌없으면 임시적 실행 영구화
    - Abort: 충돌 있으면 모든 실행 무효화
    - 트랜잭션 사이의 중첩(트랜잭션안의 트랜잭션), Chiled 트랜잭션의 Abort가 Parent를 Abort시키지 않음
    - Atomic{} : 하나의 트랜잭션 구간을 정의, 이 블록을 트랜잭션으로 돌려라
  + 트랜잭션 메모리의 구현
    - 소프트웨어 트랜잭션 메모리: Tboost.STM, Intel STM Compiler, SXM, STM
    - 구현시 문제점
      * 충돌 이후에도 트랜잭션이 계속실행됨
      * 틀린값을 읽어서 무한루프나 예외상황이 발생함
      * 모든 메모리 접근을 API를 통해서 해야하므로 성능이 좋지 않다
    - 하드웨어 트랜잭션 메모리
      * 캐시 일관성 프로토콜을 수정해서 구현
      * Intel x86(Haswell 부터)
      * 캐시 태그에 transaction bit 추가
      * 트랜잭션 캐시 라인이 invalidate 되면 캐시라인을 메모리에 쓰지않고 폐기함, CPU 실행도 abort함, 옛날 상태로 되돌리는것
      * Haswell의 XTM
        + 복수개 메모리에 대한 트랜잭션 허용(멀티 카스가능)
        + cpu에서 트랜잭션 실패시 복구 제공
        + visual studio 2012, update 2부터 지원
        + \_xbegin(), \_xend()
        + 게으른동기화 이용
      * 지원하지 않는 CPU는 프로그램이 죽는다
      * 락프리와는 비슷한 성능이나, 구현은 더 쉽다
      * 지원 CPU에 영향을 받아서 게임 클라이언트에는 적용못함, 서버에서 적용한다.
      * 텍스트이(가) 표시된 사진

        자동 생성된 설명
    - 정리
      * 
    - Haswell HTM 한계
      * 모든 알고리즘에 적용 불가
      * HW용량 한계, 알고리즘의 맞춤형 수정 필요
    - AMD는 ASF 아직도 proposal단계
    - Intel은 아직 많이 쓰이진 않음
  + 새로운 언어
    - 지금까지는 성능 향상에 한계가 있다, C스타일 언어를 사용하기 때문이다
    - 함수형 언어를 사용하자, 메모리공유가 없고 side effect없다, 프로그램 자체가 자연스러운 병렬성이 내장됨
    - 모든변수가 불변임, 불변 변수는 data race를 발생시키지 않음
    - 함수(입력, 출력) = 함수’(입력, 중간 값) && 함수’’(중간 값, 출력)
    - Go
      * 구글에서 c와비슷하게 만듬
      * 언어에서 멀티스레드 지원, 소프트웨어 스레드임, 개수 제한이 없다
      * Data race존재, mutex 필요 고속통신으로 해결
    - 하스켈
      * 순수 함수형 언어
      * 익히기 어려움
    - Erlang
      * 분산처리언어
      * 롤 채팅 ,콜옵 서버에서 사용
    - 함수형언어 문제
      * 생산성
        + 전혀 다른 스타일의 프로그래밍 (진입장벽)
        + 익숙해지면 오히려 생산성이 올라간
      * 성능
        + 구조체를 함수에 전달할 때 포인터가 아니라 내용을 전달하는 것과 비슷한 오버헤드.
        + 너무 자잘한 병렬화
      * IO 문제
        + IO는 순서대로 행해져야 하는데 함수형 언어에서 IO Operation의 순서를 정해주는 것이 비효율적이다.