

NUMA



산업용병렬처리특론

정내훈

2020년도 1학기

한국산업기술대학교 스마트팩토리융합학과

목차

- 과제 리뷰
- NUMA 소개
- NUMA Aware Programming
- 4차 과제

3주차 과제

● Hazard Pointer + SkipList를 구현하시오

- 제출 E-Class에 제출 (재택 수업 기간 동안만, 정상 수업 시 Perforce에 제출 예정)
 - 소스코드, 성능 벤치마크 결과(초당 Operation수, 메모리 사용량), 성능 분석
 - HP_set.cpp (실습 프로그램) 참조.
- 기한
 - 4월 7일 화요일 오후 1시까지.

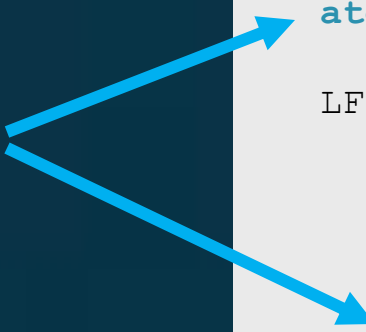
3주차 과제 결과

	SKIPLIST + Hazard Pointer
최재용	Fail (Crash) => Good
강수빈	X => Fail (Crash)
성소윤	Fail (Crash) => Good

성소윤

● Reference Counter 도입

- 자신을 가리키는 링크가 모두 없어졌을 때 Retire



```
class LFSKNode
{
public:
    int key;
    LFSKNode* next[MAX_LEVEL];
    int topLevel;
    atomic_uint32_t refCnt{ 0 };

    LFSKNode(int x, int height) {
        key = x;
        for (int i = 0; i < MAX_LEVEL; i++)
            next[i] = AtomicMarkableReference(NULL, false);
        topLevel = height;
        refCnt.store(topLevel + 1, memory_order_relaxed);
    }
    ...
};
```

성소윤

● Hazard Pointer 자료 구조

- 레벨별로 따로 저장.

- 의미가 있나? 검색 루프가 길어지고, Cache에 좋지 않을 듯

```
atomic<LFSKNode*>* HPpool[MAX_THREADS][MAX_LEVEL][MAX_HP];
```

- Atomic 포인터의 포인터를 저장

- Cache Thrashing 방지용 => 성능비교 필요(scan 호출 주기와의 상관관계?)
- Memory Reordering으로 부터 안전하지 않을 텐데?
 - 접근 시 memory_order 지정

```
LFSKNode* curr = NULL;
```

```
...
```

```
HPpool[tid][level][cur_idx]->store(curr, memory_order_release);
```

성소윤

● 성능

```
c:\depot\Projects\Lecture\TMT\HW1\SSY\Release\Project1.exe
First 20 entries are : 0(0), 2(2), 3(1), 6(0), 7(0), 9(1), 10(0), 11(0), 13(2), 15(0), 21(1), 22(2), 24(0), 27(1), 28(1), 30(0), 32(0), 33(1), 35(0), 36(2),
1 Threads, Time = 1070 ms
First 20 entries are : 3(1), 5(0), 9(0), 10(0), 12(0), 13(0), 16(4), 18(2), 19(2), 23(0), 24(0), 26(2), 29(2), 31(0), 33(0), 38(0), 43(1), 45(0), 46(1), 47(0),
2 Threads, Time = 795 ms
First 20 entries are : 1(1), 6(3), 7(1), 8(0), 11(1), 14(1), 15(0), 16(4), 17(1), 18(0), 19(0), 20(0), 21(1), 22(1), 23(0), 25(0), 31(0), 32(0), 34(1), 36(2),
4 Threads, Time = 553 ms
First 20 entries are : 0(1), 3(1), 5(0), 8(0), 10(1), 11(0), 12(2), 13(0), 14(0), 16(0), 17(2), 18(0), 19(0), 20(0), 23(2), 24(1), 25(1), 26(0), 30(1), 31(2),
8 Threads, Time = 414 ms
First 20 entries are : 0(1), 1(0), 2(1), 3(2), 7(0), 8(0), 9(0), 10(0), 11(0), 15(0), 16(2), 20(0), 21(1), 22(1), 26(1), 30(1), 31(1), 32(0), 33(1), 38(1),
16 Threads, Time = 400 ms
계속하려면 아무 키나 누르십시오 . . .
```

최재용

- 역시 Reference Counter 사용해서 skiplist에서 완전히 빠졌을 때 retire

```
class LFSKNode
{
public:
    int key;
    LFSKNode* next[MAX_LEVEL];
    int topLevel;
    atomic_uint ref_count;
    ...
}
```


최재용

● Hazard Pointer 관리

- Smart Pointer로 관리 HazardPtrGuard
 - method에서 return하는 순간 자동으로 해제.
- local_hps로 탐색한 후 결과를 level_pred_hps와 level_succ_hps에 저장

```
HazardPtrList<LFSKNode> hp_list;  
using HP = HazardPtrGuard<LFSKNode>;  
thread_local array<HP, MAX_LEVEL> level_pred_hps;  
thread_local array<HP, MAX_LEVEL> level_succ_hps;  
thread_local array<HP, 3> local_hps;
```

최재용

● Hazard Pointer 관리

- Sliding을 막기 위해 swap사용
 - pointer를 swap하는 것이 아니고 pointer의 pointer를 swap하므로 hp_list에 들어있는 값은 swap되지 않음.
- 메소드 종료시 hazard pointer를 초기화 하지 않음.
 - correctness에는 문제가 없으나. 살짝 메모리 Leak,

최재용

● 성능

```

Microsoft Visual Studio 디버그 콘솔
First 20 entries are : 0(0), 2(2), 3(1), 6(0), 7(0), 9(1), 10(0), 11(0), 13(2), 15(0), 21(1), 22(2), 24(0), 27(1), 28(1), 30(0), 32(0), 33(1), 35(0), 36(2),
1 Threads, Time = 1463 ms
First 20 entries are : 3(1), 5(0), 9(0), 10(0), 12(0), 13(0), 16(4), 18(2), 19(2), 23(0), 24(0), 26(2), 29(2), 31(0), 33(0), 38(0), 43(1), 45(0), 46(1), 47(0),
2 Threads, Time = 966 ms
First 20 entries are : 1(1), 6(3), 7(1), 8(0), 11(1), 14(1), 15(0), 16(4), 17(1), 18(0), 19(0), 20(0), 21(1), 22(1), 23(0), 25(0), 31(0), 32(0), 34(1), 36(2),
4 Threads, Time = 779 ms
First 20 entries are : 0(1), 3(1), 5(0), 8(0), 10(1), 11(0), 12(2), 13(0), 14(0), 16(0), 17(0), 18(0), 19(0), 20(0), 23(2), 24(1), 25(1), 26(0), 30(1), 31(2),
8 Threads, Time = 760 ms
First 20 entries are : 0(1), 1(0), 2(1), 3(2), 7(0), 8(0), 9(0), 10(0), 11(0), 15(0), 16(2), 20(1), 21(1), 22(1), 26(1), 30(1), 31(1), 32(0), 33(1), 38(1),
16 Threads, Time = 1236 ms
First 20 entries are : 0(1), 1(1), 7(0), 8(0), 9(0), 10(0), 11(0), 13(0), 14(0), 16(0), 18(5), 19(1), 22(0), 24(0), 25(0), 26(1), 28(1), 29(0), 31(0), 32(0),
32 Threads, Time = 2144 ms

C:\wdepot\Projects\Lecture\TMT\HW1\CJY\Release\CJY.exe(프로세스 21460개)이(가) 종료되었습니다(코드: 0개).
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] -> [디버깅] > [디버깅이 중지되면 자동으로 콘솔 닫기]를 사용하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요 ...
  
```

평가

● 성능 : Skiplist + EBR

```
C:\depot\Projects\Lecture\TMT\HW1\SSY\Release\Project1.exe
First 20 entries are : 0(0), 2(2), 3(1), 6(0), 7(0), 9(1), 10(0), 11(0), 13(2), 15(0), 21(1), 22(2), 24(0), 27(1), 28(1), 30(0), 32(0), 33(1), 35(0), 36(2),
1 Threads, Time = 760 ms
First 20 entries are : 3(1), 5(0), 9(0), 10(0), 12(0), 13(0), 16(4), 18(2), 19(2), 23(0), 24(0), 26(2), 29(2), 31(0), 33(0), 38(0), 43(1), 45(0), 46(1), 47(0),
2 Threads, Time = 511 ms
First 20 entries are : 1(1), 6(3), 7(1), 8(0), 11(1), 14(1), 15(0), 16(4), 17(1), 18(0), 19(0), 20(0), 21(1), 22(1), 23(0), 25(0), 31(0), 32(0), 34(1), 36(2),
4 Threads, Time = 332 ms
First 20 entries are : 0(1), 3(1), 5(0), 8(0), 10(0), 11(0), 12(2), 13(0), 14(0), 16(0), 17(0), 18(0), 19(0), 20(0), 23(2), 24(1), 25(1), 26(0), 30(1), 31(2),
8 Threads, Time = 196 ms
First 20 entries are : 0(1), 1(0), 2(1), 3(2), 7(0), 8(0), 9(0), 10(0), 11(0), 15(0), 16(2), 20(7), 21(1), 22(1), 26(1), 30(1), 31(1), 32(0), 33(1), 38(1),
16 Threads, Time = 452 ms
계속하려면 아무 키나 누르십시오 . . .
```

평가

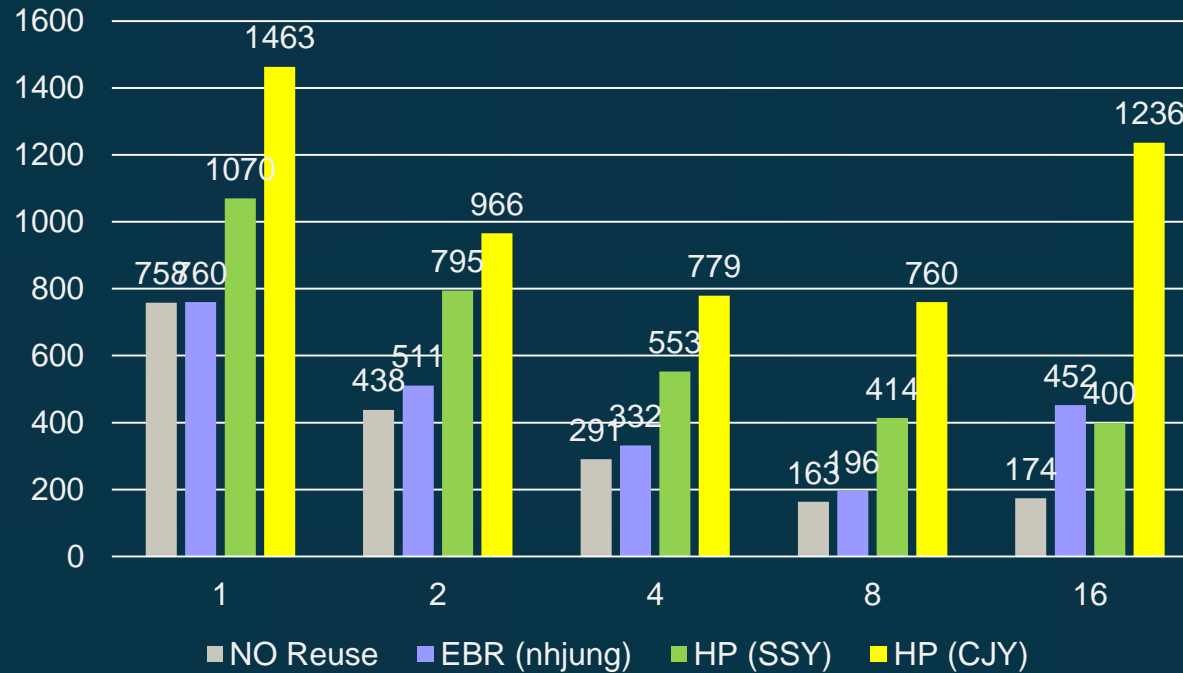
● 성능 : Skiplist + NoReuse

```
c:\depot\Graduate\2020-TMT\LFSKIPLIST\Release\LFSKIPLIST.exe
First 20 entries are : 0(0), 2(2), 3(1), 6(0), 7(0), 9(1), 10(0), 11(0), 13(2), 15(0), 21(1), 22(2), 24(0), 27(1), 28(1), 30(0), 32(0), 33(1), 35(0), 36(2),
1 Threads, Time = 758 ms
First 20 entries are : 3(1), 5(0), 9(0), 10(0), 12(0), 13(0), 16(4), 18(2), 19(2), 23(0), 24(0), 26(2), 29(2), 31(0), 33(0), 38(0), 43(1), 45(0), 46(1), 47(0),
2 Threads, Time = 438 ms
First 20 entries are : 1(1), 6(0), 7(1), 8(0), 11(1), 14(1), 15(0), 16(4), 17(1), 18(0), 19(0), 20(0), 21(1), 22(1), 23(0), 25(0), 31(0), 32(0), 34(1), 36(2),
4 Threads, Time = 291 ms
First 20 entries are : 0(1), 3(1), 5(0), 8(0), 10(1), 11(0), 12(2), 13(0), 14(0), 16(0), 17(2), 18(0), 19(0), 20(0), 23(2), 24(1), 25(1), 26(0), 30(1), 31(2),
8 Threads, Time = 163 ms
First 20 entries are : 0(1), 1(0), 2(1), 3(2), 7(0), 8(0), 9(0), 10(0), 11(0), 15(0), 16(2), 20(7), 21(1), 22(1), 26(1), 30(1), 31(1), 32(0), 33(1), 38(1),
16 Threads, Time = 174 ms
계속하려면 아무 키나 누르십시오 . . .
```

평가

● 성능

SKIPLIST의 성능



Hazard Pointer + SkipList

● 구현 Tip

- Retire Point를 잘 잡아야 한다.

- 전역 변수로 접근이 불가능한 point를 확실히 잡아야 한다.
- reference counter가 단순/무식/확실
- EBR과 많이 비슷하다.

- Sliding에 주의 해야 한다.

- Hazard Pointer끼리 값을 move하는 경우 scan()에서 빠뜨릴 수 있다.
- Move가 atomic인 것과는 아무 관계가 없다. Scan()이 atomic이 아니다.

Hazard Pointer + SkipList

● 총평

- 어렵지는 않다.
- 하지만 쉽지도 않다.
 - 특히 pointer들이 여러 노드들을 이동하는 자료 구조
- 같은 Hazard Ptr이라도 최적화 여부에 따라 성능차이가 꽤 난다.
- 성능은 별로다
 - EBR이 훨씬 좋다.
 - Hazard Ptr를 써야할 충분한 이유가 없으면 EBR을 사용할 듯.

목차

- 과제 리뷰
- NUMA 소개
- NUMA Aware Programming
- 4차 과제

NUMA

- Non Uniform Memory Access
- 메모리에 접근할 때 Physical Address에 따라 접근시간이 달라 질 수 있는 메모리 구조
- 반대말은 UMA
- COMA(Cache Only Memory Access) 같은 것도 있음.
- 1990년대에 상용화, PC에서는 AMD의 Opteron에 처음으로 도입 되었음(2001).

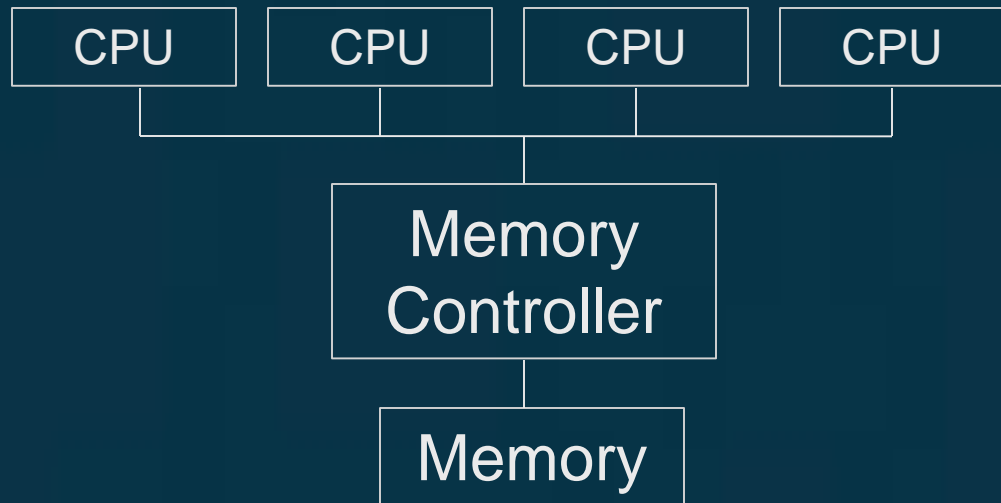
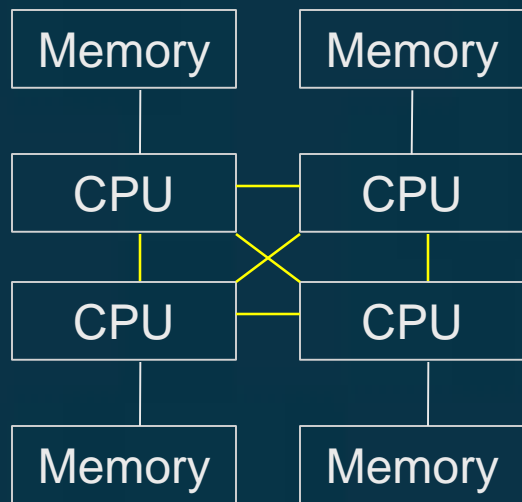
NUMA

- 일반적인 구조
 - 멀티 CPU + SMP + CPU마다 Local Memory
 - CPU들은 고속의 HW기반 네트워크로 연결되어 있다.
 - Cache Coherence를 지원한다.
- 프로그래밍 방식
 - 기존 UMA 프로그램이 아무 문제 없이 작동
 - 성능에서만 약간의 문제가 있을 수 있음.

NUMA

- 메모리 연결

NUMA



OLD UMA

CPU의 개수가 많아질 수록
UMA 방식에서는 성능에 한계

NUMA

● 장점

- 중앙 집중형 메모리 구조에 비해 메모리 Bandwidth가 늘어난다. => 멀티 CPU 컴퓨터에서 Memory Access Bottleneck이 해소된다.
 - 예) 2003년 NCSoft에서 L2를 실행할 때 몇배의 성능 차.

● 단점

- 성능향상을 위한 프로그램 복잡도 증가
 - pinning과 NUMA friendly 메모리 관리 필요.

NUMA

● 프로그래밍

- Pinning : 각각의 thread를 특정 CPU에 강제 할당. (특정 Core set에 할당하는 방식)

- 하지 않으면 복불복 스케줄러에 의해 여러 core에서 돌아가며 실행
- 원인 : 컴퓨터에는 많은 다른 프로세스들이 존재, 수시로 걸려오는 HW 인터럽트
- Pinning을 하면 Cache Miss감소, But Bottleneck 가능

- 메모리 할당

- 메모리 요청시 Local Memory를 할당. 쓰레드 별로 별도의 memory pool이 존재해야 함.

NUMA

● 프로그래밍

- Pinning : 각각의 thread를 특정 CPU에 강제 할당. (특정 Core set에 할당하는 방식)

- 하지 않으면 복불복 스케줄러에 의해 여러 core에서 돌아가며 실행
- 원인 : 컴퓨터에는 많은 다른 프로세스들이 존재, 수시로 걸려오는 HW 인터럽트
- Pinning을 하면 Cache Miss감소, But Bottleneck 가능

- 메모리 할당

- 메모리 요청시 Local Memory를 할당. 쓰레드 별로 별도의 memory pool이 존재해야 함.

NUMA

	Machine A	Machine B	Machine C
Processor Family	Opteron 8380	Opteron 6164	Opteron 6174
Processor Speed	2.5GHz	1.7GHz	2.2GHz
Number of NUMA nodes	4	4	8
Local Memory Access Latency (cycles)	261	175	151
Remote Memory Access Latency (cycles)	286 (1 hop) 377 (2 hops)	247	232 (1 hop) 337 (2 hops)
Memory controller bandwidth	7580MB/s	5850MB/s	6080MB/s
Interconnect technology	HT1 No HT Assist	HT3 HT Assist	HT3 HT Assist
Interconnect bandwidth	2850MB/s (16-bit link)	4810MB/s (16-bit link)	4590MB/s (16-bit link)
		2100MB/s (8-bit link)	3830MB/s (16-bit link, 8 bits used)
			2550MB/s (8-bit link)

Table 1.2 – Main characteristics of the machines used in the evaluations.

NUMA

- 프로그래밍 최적화
 - 가능한 Remote Memory의 접근을 줄일 것

목차

- 과제 리뷰
- NUMA 소개
- **NUMA Aware Programming**
- 4차 과제

NUMA Aware

● Basic Idea

- Remote Memory Access를 줄인다.
- => Local Memory에서 모든 것을 해결한다.
- => Local Memory에 Clone을 저장한다.
 - 같은 CPU에 있는 Core들은 하나의 Clone을 공유한다.
- => Clone은 Universal Constructor를 사용한다.

Universal Constructor

- 만능기계

- 모든 알고리즘 (자료 구조 + Method)를 멀티쓰레드 Lock-Free(또는 Wait-Free)로 만들어주는 막강한 알고리즘
- 교재에 잘 나와있고, 고급멀티쓰레드 프로그래밍 과목에서 구현해 보았음.

Universal Constructor

- 만능기계

- 모든 알고리즘 (자료 구조 + Method)를 멀티쓰레드 Lock-Free(또는 Wait-Free)로 만들어주는 막강한 알고리즘
- 교재에 잘 나와있고, 고급멀티쓰레드 프로그래밍 과목에서 구현해 보았음.

Universal Constructor

```
class LFUniversal {
private:
    Node *head[N],  Node tail;

public:
    LFUniversal() {
        tail.seq = 1;
        for (int i=0;i<N;++i) head[i] = &tail;
    }
    Response apply(Invocation invoc) {
        int  i = Thread_id();
        Node prefer = Node(invoc);
        while (prefer.seq == 0) {
            Node *before = tail.max(head);
            Node *after = before->decideNext->decide(&prefer);
            before->next = after; after->seq = before->seq + 1;
            head[i] = after;
        }
        SeqObject myObject;
        Node *current = tail.next;
        while (current != &prefer) {
            myObject.apply(current->invoc);
            current = current->next;
        }
        return myObject.apply(current->invoc);
    } };
```

Universal Constructor

● 문제점

- 너무 느리다.

쓰레드 갯수	1	2	4	8	16	32	64
무잠금 만능	374900	196600	169700	112000	74200	52500	41300
무대기 만능	364000	196400	121900	113600	57700	59900	44800
EnterCritical	232	822	1160	1765	1914	4803	7665

XEON, E5-4620, 2.2GHz, 4CPU (32 core), stl::queue, 값 ms

- 알고리즘의 문제 보다는 구현의 문제

- 반드시 처음부터 invocation을 적용해야 하는가?
- Read Only Method를 저장할 필요가 있는가?

4주차 과제

- Universal Constructor를 사용하여 Lock-Free Skiplist를 구현하시오.
 - 제출 E-Class에 제출
 - 최적화 기법들을 적용하시오
 - Invocation호출 최소화, Read Only Invocation제거
 - 소스코드, 성능 벤치마크 결과(초당 Operation수, 메모리 사용량), 구현기법 설명, 성능 분석
 - sklist.cpp (실습 프로그램) 참조
 - 기한
 - 4월 15일 수요일 오후 1시까지.