병렬컴퓨팅을 위한 Lock-Free 고유 식별자 할당

손범준, 안기영*
한남대학교,*한남대학교

sbj1229@gmail.com, *kya@hnu.kr

Lock-free unique identifier allocation for parallel computing

Bum Jun Son, Ki Yung Ahn*
Hannam Univ., *Hannam Univ.

요 약

분산 및 멀티프로세스 환경에서는 다양한 고유 식별자 할당 방식이 존재한다. 그러나 이러한 방법들은 하나의 프로세스 안에서 멀티코어를 활용하는 병렬컴퓨팅 환경에서는 불필요한 정보를 포함하고 있어 효과적이지 못하며, 기존의 순차적 프로그래밍에서 일반적으로 사용하던 방식들은 순차적 의존성으로 인해 병렬처리를 통한 성능 향상을 저해한다. 본 논문은 이러한 환경에서 나머지 연산을 이용해 순차적 의존성을 제거하여 Lock 이 필요하지 않은 더 효과적인 고유 식별자 할당 방식을 연구하였다. 벤치마크를 통해 연구한 방식과 Lock 을 사용하는 방식의 구현을 비교하여 우리가 연구한 Lock-Free 방식이 멀티코어를 활용한 병렬화로 성능을 개선하는 데 적합함을 확인하였다.

Ⅰ. 서론

이 연구는 한 프로세스 내에서의 멀티코어 병렬처리에 적합한 고유 식별자 할당 방식에 대한 것이다. 분산 및 멀티 프로세스 환경에서는 UUID[1]를 비롯해 다양한 용도에 맞게 제안되고 구현된 고유 식별자 할당 방식이 이미 여러가지 있는데, 그런 고유 식별자 생성에는 하드웨어 정보 및 프로세스 번호 등도 활용된다. 그러나 한 프로세스 안으로 한정된 상황에서 그런 정보는 불필요하며, 더 가볍고 단순한 방식의 고유 식별자 할당이 적합하다.

많은 시간이 걸리는 큰 작업을 여러 개의 작은 작업으로 분할할 수 있고 서로 의존성이 없다면 병렬처리를 통해 성능 향상을 기대할 수 있다. 그런데 핵심 알고리즘의 분할에는 의존성이 없더라도 로그를 남기거나 트랜잭션 번호를 부여하는 등의 부가적 기능이 순차적 의존성이 있다면 기대했던 만큼의 성능 향상이 이루어지지 못한다. 이런 부가적 기능에서 종종 필요한 요소가 바로 고유 식별자 할당이다. 따라서 병렬 컴퓨팅에 적합한 방식으로 고유 식별자를 할당할 수 있다면 부가적 기능 구현에서 병목 없이 병렬화를 통한 원활한 성능을 향상을 기대할 수 있다. 실제로 이전 연구[2]에서는 각 병렬 작업마다 특정 구간의 연속된 정수를 서로 겹치지 않도록 미리 배정해주는 방식으로 (예컨대, 1~999, 1000~1999, 2000~2999 같은 식으로) 고유 식별자를 할당하였고 이를 멀티코어 병렬 컴퓨팅 환경에서 벤치마크하여 성능이 개선됨을 확인하였다. 여기서는 이전 연구보다 더 유연하게 멀티코어 병렬 프로그래밍에 활용가능한 병렬 고유 식별자 생성 방법을 연구한다.

Ⅱ. 본론

서론에서 언급한 서로 겹치지 않는 연속된 구간 할당 방식도 분명 성능 개선 효과는 있지만 각 병렬 작업마다 필요한 고유 식별자의 개수를 미리 예상할 수 있어야 한다는 단점이 있다. 고유 식별자 개수 예상이 어렵거나 가능하더라도 그것을 파악하는 데 복잡한 계산이 필요한 경우에는 효과적이지 못한 방법이다. 우리는 각 병렬 작업에서 필요한 고유 식별자 개수를 신경 쓰지 않고 Lock 없이 병렬 작업에 고유 식별자를 할당하는 방법을 제시하고 이를 구현하여 Lock 을 활용한 방식과 성능을 비교하는 간단한 벤치마크를 수행하였다.

병렬화를 고려하지 않은 순차적 프로그램이라면 전역 상태인 카운터를 두어 순차적으로 1 씩 증가시키는 간단한 방식으로 고유 식별자를 생성할 수 있다. 하지만 이 방식 그대로 병렬화하려면 여러 스레드가 동시에 이 카운터에 접근하려는 경쟁상황(race condition)이 발생 가능하다. 물론 카운터에 접근할 때마다 Lock 을 걸면 경쟁상황은 막을 수 있으나 핵심 알고리즘 진행 때문이 아닌 단지 고유 식별자 하나를 생성하기 위해 스레드 사이에 동기화를 유발하는 부담을 추가한다. 빈번한 Lock 의 사용은 병렬화의 병목이 되어 성능을 저해할 가능성이 높다.

2.1. 나머지 연산 Lock-Free 고유 식별자 생성 방식

우리가 고안한 방식은 그림 1 처럼 여러 갈래의 병렬 작업에서 나머지 연산(modular arithmetic)으로 서로 겹치지 않게 고유 식별자를 생성한다. 병렬로 분기하기 전에 1 씩 증가시키는 방식(…, k-2, k-1, k)으로 고유 식별자를 생성하다가 세 갈래의 병렬 작업으로 분기하면 각각 k, k+1, k+2 에서부터 3 씩 증가시키며 생성하고, 다시 한 갈래로 합쳐질 때는 각 병렬 작업의 마지막 상태의 최대값(max(x,y,z))으로부터 다시 1 씩 증가시키며 고유 식별자를 생성한다. 참고로 (k, i)의 의미는 다음 번에 생성할 고유 식별자가 k 이며 i 씩

증가시키며 생성하라는 뜻이다. 일반적으로 (k, i)인 상태에서 n 갈래의 병렬 작업으로 분기한다면 (k,i*n), (k+1,i*n), …, (k+n-1,i*n)과 같이 나머지 연산을 통해 병렬 작업 도중 동기화 없이 서로 겹치지 않게 고유 식별자를 병렬적으로 생성해 나갈 수 있다.

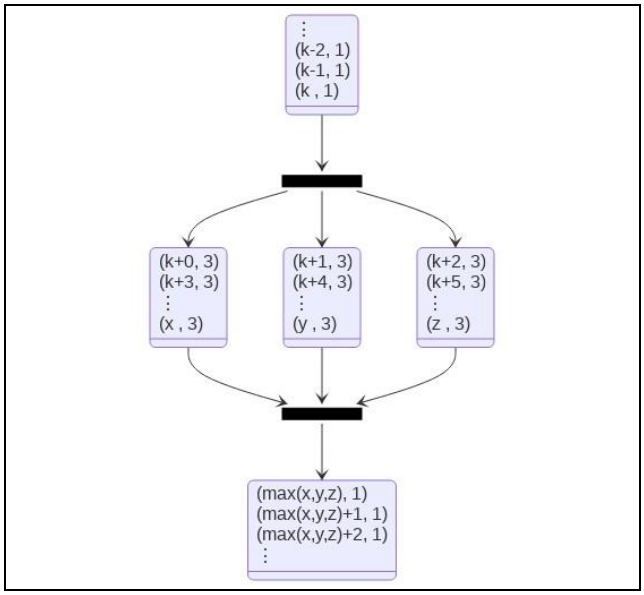

그림 1. 나머지 연산 활용 고유 식별자 할당 (세 갈래로 나뉘어 병렬처리 후 다시 한 갈래로 합쳐 진행하는 경우)

2.2. 구현 및 벤치마크

멀티코어 병렬처리를 잘 지원하는 언어인 Haskell 로 구현하였으며, 고유 식별자 생성 구현 및 벤치마크에 사용된 소스코드는 온라인에 공개되어 있다.[1] Haskell 은 Lock 을 걸며 동기화를 직접 관리하는 전통적 멀티스레드 프로그래밍을 비롯해 Lock 을 사용하지 않는 결정적 병렬성(deterministic parallelism) 등 다양한 병렬 프로그래밍 형식을 지원하고 있어 [3-5] 서로 다른 병렬처리 방식을 구현하여 벤치마크하기 용이하다. 좀더 구체적으로는 나머지 연산 기반의 Lock-Free 방식은 결정적 병렬성을 지원하는 parallel 라이브러리를 활용해 구현하였으며, Lock 을 사용하며 1 씩 증가시키는 방식은 base 라이브러리의 MVar 를 활용해 구현하였다.

벤치마크는 다른 작업 내용 없이 고유 식별자를 반복적으로 일정 개수 생성하는 작업을 그림 1 처럼 병렬적으로 1 개에서부터 8 개까지 늘려 가는 방식으로 실행하였다. 6 코어 12 하이퍼스레드를 지원하는 i7-9750H CPU 가 탑재된 컴퓨터에서 Lock-Free 방식은 각 병렬 작업마다 이백만개씩, Lock 방식은 각 병렬 작업마다 십만개씩 생성하는 벤치마크를 수행하였으며 그 결과가 그림 2 와 그림 3 에 나타나 있다. 참고로 Lock 방식의 벤치마크의 경우 병렬 작업 개수가 1 개일 때에도 Lock 을 사용한다. Lock-Free 방식이 단독 작업에서 5 초 이내에 백만 개 단위를 생성하는 데 비해 Lock 방식은 Lock 을 관리하는 오버헤드 때문에 단독 작업에서 5 초대에 십만 개를 생성하기도 버겁다. 또한 가용 스레드 개수를 늘렸을 때 성능의 차이가 심화됨을 그림 2 와 그림 3 을 대조하면 확인할 수 있다.

[1] https://github.com/hnu-pl/par-uniq-id

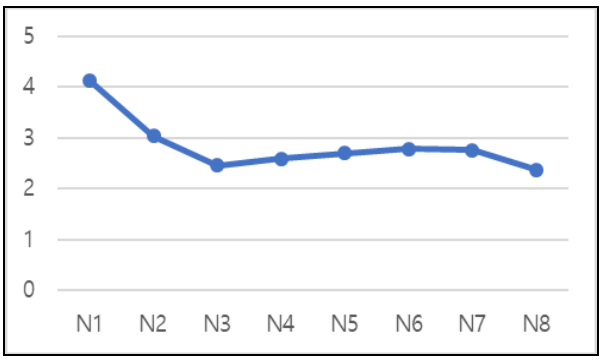

그림 2. Lock-Free 고유 식별자 생성 병렬화 벤치마크 (각 병렬 작업마다 2,000,000 개 생성)

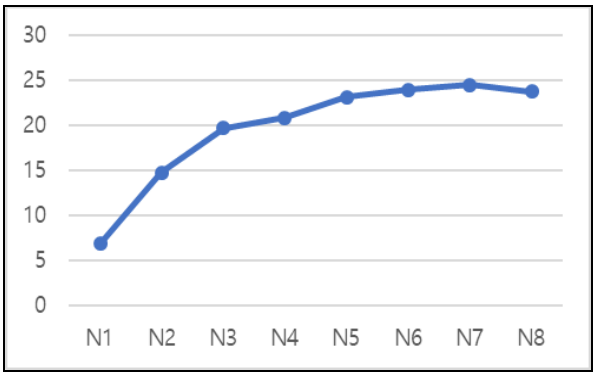

그림 3. Lock 활용 고유 식별자 생성 병렬화 벤치마크 (각 병렬 작업마다 100,000 개 생성)

Ⅲ. 결론

나머지 연산(modular arithmetic)을 활용한 고유 식별자를 할당 방식은 다음과 같은 장점이 있어 다양한 방식의 병렬 프로그램에 적용할 수 있다. (1) Lock 이 필요하지 않기 때문에 Lock 으로 인한 병목현상을 피할 수 있고, (2) 병렬 작업이 요구하는 식별자 개수를 미리 파악할 필요가 없으며, (3) 분기된 병렬 작업 내에서 재분기하는 경우에도 유연하게 활용 가능하다. 벤치마크를 통해 Lock 이 배제된 결정적 병렬화 방식으로 구성된 병렬 프로그램에 활용해도 성능에 방해가 되지 않는다는 것을 벤치마크로 확인하였다. 본문의 벤치마크를 요약하면 다른 실제적 작업내용 없이 고유 식별자 생성만 병렬로 반복할 경우 우리가 고안한 Lock-Free 방식과 Lock 을 사용한 방식을 대조하여 벤치마크했다. 가용 스레드 개수를 1~8 로 늘려나가면 Lock-Free 방식은 수행 시간이 단축되는 경향을 보이는 반면 Lock 을 사용하는 방식은 오버헤드가 오히려 늘어나며 수행 시간이 증가하는 경향을 보였다.

이 연구 결과를 통해 얻은 통찰은 기존에 병렬화를 고려하지 않고 작성된 순차적 고유 식별자 생성에 기반한 라이브러리의 활용은 병렬화에 있어 병목이 될 수 있다는 점이다. 따라서 병렬화로 성능 개선을 도모한다면 고유 식별자를 활용하는 로직을 포함한 라이브러리를 면밀하게 검토하여 병렬화에 병목을 일으키지 않는 방식으로 라이브러리로 대체하거나 라이브러리 내부 구현을 병렬화에 적합한 방식으로 개선할 필요가 있다는 것이다.

향후 연구로는 고유 식별자를 활용하여 구현하는 알고리듬[6-7]을 병렬화하여 실제적 작업내용이 포함된 벤치마크로 기존의 알고리즘 병렬화에 활용했을 때 실제적인 성능 개선이 얼마나 가능한지 탐구하고자 한다.

ACKNOWLEDGMENT

본 연구는 한남대학교의 2021 미래한남사업의 연구비 지원을 받아 수행되었습니다.

참 고 문 헌

[1] Paul J. Leach, Michael Mealling, and Rich Salz. 2005. “A Universally Unique IDentifier (UUID) URN Namespace”, RFC, Vol. 4122, pp. 1-32, July 2005.

[2] Ki Yung Ahn. 2021. “Deterministic Parallelism for Symbolic Execution Programs based on a Name-Freshness Monad Library”, Journal of the Korea Society of Computer and Information, vol. 26, no. 2, Feb 2021 pp. 1-9.

[3] Simon Peyton Jones, Andrew D. Gordon, and Sigbjorn Finne. 1996. “Concurrent Haskell”, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PoPL). 1996.

[4] Simon Marlow, Simon L. Peyton Jones, and Satnam Singh. 2009. “Runtime Support for Multicore Haskell”, 14th ACM SIGPLAN International Conference on Functional Programming, pp. 65-78, 2009.

[5] Simon Marlow, Parallel and Concurrent Programming in Haskell, Oreilly & Associates Inc(2013)

[6] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986.“Hygienic Macro Expansion”, Symposium on LISP and Functional Programming, pp. 151-161, August 1986.

[7] Ross Horne, Ki Yung Ahn, Shang-wei Lin, and Alwen Tiu. 2018. “LICS '18: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science”, July 2018, pp. 26- 35

Appendix A. 나머지 연산 Lock-Free 방식 소스코드

```
import Control.Parallel.Strategies
import Control.Monad.Trans.State.Strict

newId :: Monad m => StateT (Int, Int) m Int
newId = do { (k,i) <- get; put (k+i,i); return k }

runParallel :: [State (Int, Int) a] -> State (Int, Int) [a]
runParallel ms = do
    (k,i) <- get
    let n = length ms
    let ps = zipWith ($) [flip runState (k+r,i*n) | r<-[0..n-1]] ms
             `using` parList rpar
    let (as,ss) = unzip ps
    put (maximum (map fst ss), i)
    return (as `using` parList rseq)

bench1 n m = flip runState (0,1) . runParallel $
    replicate n (foldl1 (>>) $ replicate m newId)
```

Appendix B. 공유 메모리 카운터 Lock 방식 소스코드

```
import qualified Contorl.Monad.Parallel as MP

newIdMVar :: MVar Int -> IO Int
newIdMVar c = modifyMVar c (\n -> return (n+1,n))

bench2 n m = do
    c <- newMVar (0 :: Int)
    let new = newIdMVar c
    MP.replicateM n (foldl1 (>>) $ replicate m new)
```