JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 27 No. 4, pp. 1-8, April 2022
https://doi.org/10.9708/jksci.2022.27.04.001

Lock-free unique identifier allocation for parallel macro expansion

Bum-Jun Son*, Ki Yung Ahn*

*Master's Candidate, Dept. of Computer Engineering, Hannam University, Daejeon, Korea
*Assistant Professor, Dept. of Computer Engineering, Hannam University, Daejeon, Korea

[Abstract]

In this paper, we propose a more effective unique identifier allocation method for macro expansion in a single-process multicore parallel computing environment that does not require locks. Our key idea for such an allocation method is to remove sequential dependencies using the remainder operation. We confirmed that our lock-free method is suitable for improving the performance of parallel macro expansion through the following benchmark: we patched an existing library, which is based on a sequential unique identifier allocation, with our proposed method, and compared the performances of the same program but using two different versions of the library, before and after the patch.

▸ **Key words**: Unique Identifier, Parallel Processing, Sequential Dependence, Name Binding, Haskell

[요 약]

이 논문에서는 싱글 프로세스 멀티코어 환경의 매크로 확장에서 Lock이 필요하지 않은 더 효과적인 고유식별자 할당 방식을 제안한다. 이 할당 방식의 핵심 아이디어는 나머지 연산을 이용해 순차적 의존성을 제거하는 것이다. 우리가 고안한 방식이 멀티코어 병렬 환경에서 매크로 확장의 성능 개선에 적합함을 확인하기 위해, 기존에 순차적 방식의 고유식별자 생성으로 구현된 라이브러리를 우리가 고안한 방식으로 변경하여 변경 전의 버전과 후의 버전의 라이브러리로 작성된 같은 프로그램의 성능을 비교하는 벤치마크를 수행하였다.

▸ **주제어**: 고유식별자, 병렬처리, 순차적 의존성, 이름 바인딩, 하스켈

• First Author: Bum-Jun Son, Corresponding Author: Ki Yung Ahn
*Bum-Jun Son (sbj1229@gmail.com), Dept. of Computer Engineering, Hannam University
*Ki Yung Ahn (kya@hnu.kr), Dept. of Computer Engineering, Hannam University
• Received: 2021. 12. 23, Revised: 2022. 03. 25, Accepted: 2022. 04. 07.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

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

많은 시간이 걸리는 큰 작업을 여러 개의 작은 작업으로 분할 가능하고 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 고유식별자 할당이다. 따라서 병렬 컴퓨팅에 적합한 방식으로 고유식별자를 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) 고유식별자를 할당하여 이를 멀티코어 병렬 컴퓨팅 환경에서 벤치마크하여 성능개선을 확인하였다. 해당 연구[2]도 성능 개선 효과가 있지만 각 병렬 작업마다 필요한 고유식별자의 개수를 어느 정도 예상할 수 있어야 한다는 한계가 있다. 필요한 고유식별자 개수의 예상이 불가능하거나 가능하더라도 복잡한 분석/계산이 필요한 매크로 확장과 같은 경우에는 효과적이지 못한 방식이다.

여기서는 이전 연구보다 더 유연하게 멀티코어 병렬 프로그래밍에 활용 가능한 병렬 고유식별자 생성 방법을 제안하고 이를 실제 매크로 확장 구현에 활용되는 라이브러리에 반영해 성능 변화를 확인해 보았다. 우리는 각 병렬 작업에서 필요한 고유식별자 개수를 신경 쓰지 않고 Lock 없이 병렬 작업에 고유식별자를 할당하는 방법을 제시하고 이를 구현에 반영해 변경하였으며 (III절), 병렬성이 없는 순차적 예시 코드를 새로운 방식의 변경을 적용하기 전/후의 성능을 비교해 기존에 작성된 순차적 코드 성능 저하 여부를 가늠해 보는 벤치마크 및 병렬화 적용 시 성능 향상을 확인하는 벤치마크를 수행하였다 (IV절).

II. Background and Related work

1. Functional Programming and Parallelism

메모리를 공유하는 전통적 멀티스레드 병렬 프로그래밍의 가장 큰 난관이 바로 한 스레드가 특정 메모리 내용의 변경 시점을 다른 스레드가 미처 파악하지 못해 발생하는 경쟁상황(race condition)이라는 동시성 문제다. 함수형 프로그래밍은 부수효과(side-effect)가 없는 순수 함수(pure function)와 기존의 내용을 덮어쓰지 않고 필요한 부분만 새롭게 구성함으로써 데이터 참조가 시간에 따라 변하지 않는 불변 자료구조(persistent data-structure)를 주로 활용하는 프로그래밍 패러다임이므로, 경쟁상황을 애초에 발생시킬 여지가 적어 병렬 프로그래밍에 유리하다고 알려져 있다. 이와 관련해 국내에서도 함수형 프로그래밍 언어인 하스켈(Haskell)의 병렬/분산처리에 관한 다수의 관련 연구가 진행된 바 있다 [3,4].

하지만 함수형 프로그래밍이 병렬화에 유리함은 처음부터 병렬성을 고려해 병렬화 라이브러리 외에도 다른 모든 구성요소를 작성하면 그렇다는 원론적 가능성이며, 실제 사례를 분석해 보면 함수형 코드도 항상 쉽사리 병렬화로 성능이 개선된다는 보장은 없다. 예를 들어 순수 함수형 언어인 하스켈은 모나드(monad)로 부수효과를 관리해 순수한 계산과 혼동을 막고 있는데, 많은 모나드 기반 라이브러리들이 논리적 필연성이 아닌 (기존 순차적 알고리듬에서 검증된 방식이라) 편의상 순차성에 의존해 구현하고 있다. 함수형 프로그래밍의 장점을 다른 곳에서 잘 살렸다 하더라도 해당 라이브러리를 사용한 부분에서 병렬화로 성능을 높이는 데 있어 병목이 될 수 있다.

우리의 연구는 기존 하스켈 라이브러리에서 불필요한 순차성을 제거하여 병렬화에 적합하게 수정하되, 병렬성을 활용하지 않는 기존 싱글 스레드 코드의 성능도 희생되지 않도록 유지하는 것을 목표로 하고 있다. 우리가 집중하는 고유식별자 문제의 경우, 전역 카운터를 1씩 증가시키는 순차적 방식 그대로를 병렬화하면 여러 스레드가 동시에 카운터에 접근하는 경쟁상황이 빈발한다. 카운터 접근에 Lock을 걸어 경쟁상황을 막더라도 핵심 알고리즘 진행이 아니라 단지 고유식별자를 생성할 때마다 생기는 동기화 부담은 병렬화의 성능을 저해는 병목이 될 가능성이 크다. 따라서 Lock을 사용하지 않는 방식으로 각 스레드가 생성하는 고유식별자가 겹치지 않도록 보장하는 방식이 필요하다.

2. Unbound-generics library

Unbound-generics는 하스켈에서 프로그래밍 언어의 문법과 같은 나무(tree)구조의 재귀적 데이터 타입을 정의할 때, 변수 유효범위 개념을 다루기 적합한 *이름 바인딩* (name binding) 구조를 유연하게 명세할 수 있도록 돕는 라이브러리로, 정의된 데이터 타입으로부터 자유변수 추출 및 포획없는 치환(capture-free substitution) 등의 편의 기능까지 자동으로 유도해 주는 편리한 범용적(generic) 라이브러리다. 참고로 unbound-generics는 원래 unbound라는 이름으로 학계에서 발표된 구현[5]을 GHC에 내장으로 채택된 제네릭 프로그래밍 방식을 기반으로 포팅한 구현이다. 해당 라이브러리를 이용하면 람다 계산법의 문법을 아래와 같이 정의할 수 있다.

```
type Nm = Name Expr
data Expr = Var Nm
          | Lam (Bind Var Expr)
          | App Expr Expr
          deriving (Show, Generic, Typeable)

instance Alpha Expr
instance Subst Expr Expr where
  isvar (Var x) = Just (SubstName x)
  isvar _       = Nothing
```

이렇게 작성된 기본적인 람다계산법 문법은 자유 변수의 계산 및 바인딩 작업을 자동으로 구성하고, 이름을 치환해주는 subst 함수의 정의를 자동으로 유도한다. 이렇게 정의된 문법에서 ((λx.x) x) 는 (App (Lam (bind x (Var x)) x))로 나타낸다. ((λx. x) x)에서 자유로운 이름 x를 y로 치환하라는 subst x (Var y) (App (Lam (bind x (Var x)) x))를 실행한 결과는 람다계산법의 이름 범위를 따르는 규칙에 맞게 ((λx. x) y)를 나타내는 (App (Lam (bind x (Var x)) y))가 된다.

FIg. 1처럼 기본 명세에 새로운 문법 요소를 추가하더라도 다른 부분을 변경 없이 subst와 같은 유틸리티 함수가 변경된 문법 정의에 맞게 자동으로 유도된다. subst 함수처럼 기본적이고 기계적인 정의지만 매번 직접 작성하기 번거로운 이름 관리와 관련된 유틸리티 함수를 문법 정의가 수정될 때마다 그에 맞는 함수 정의로 자동으로 유도해 주는 것이 unbound-generics의 편리한 점이다. 따라서 unbound-generics로 이름의 범위를 명세하며 Fig. 1처럼 문법을 정의하면 subst와 같은 유틸리티 함수를 활용해 심볼릭 실행기를 작성하고 유지보수하는 데 좋다.

```
type Nm = Name Expr  -- Expr을 나타내는 이름

data Expr
  = Var Nm                  -    -- x
  | Lit Integer                  -- n
  | Lam (Bind Nm Expr)           -- (λx. e)
  | App Expr Expr                -- (e1 e2)
  | Add Expr Expr                -- e1 + e2
  | AddP Expr Expr               -- e1 |+| e2
  | If Expr Expr Expr  -- if e then e1 else e0
  deriving (Show, Generic, Typeable)
```

Fig. 1. Syntax of the λ-calculus extended with addition (+), parallel addition (|+|), and conditional expressions, defined as a Haskell datatype using the unbound-generics library.

현재 통용되는 구현은 III절에 소개할 Fig.3처럼 순차적 의존성이 있는 방식이므로, 우리가 제안한 병렬화에 적합한 방식에 따라 그 구현을 수정해 보고 벤치마크를 통해 성능 개선을 확인하고자 한다.

3. Related work

스레드를 직접 관리하는 전통적인 비결정적 병렬 프로그래밍 모델에서도 잠금(lock)이나 원자적 연산(atomic operation)에 의존하지 않고 스레드 관리와는 별개의 층위에 시스템 혹은 언어 런타임에서 제공하는 실무적으로 검증된 기능을 활용하는 고유식별자를 할당하는 방법도 활용되고 있다. 예컨대, 하스켈의 unique 라이브러리[6]는 길이 0인 배열에 대한 메모리 할당을 요청해 할당된 배열 메모리의 주소를 고유생성자로 활용한다. 메모리 할당도 그 내부 구현에서 원론적으로는 잠금이나 원자적 연산에 해당하는 상호배제 기능을 활용하더라도 스레드 관리와는 별개의 계층에서 구현되므로 직접적으로 스레드 스케줄링에 영향을 미치지는 않아 스레드 수준의 상호배제 연산보다 훨씬 효율이 좋을 것이다. 하지만 동작 원리가 명확히 드러나지 않는 외부 로직에 의존하면 디버깅을 위한 정보 출력에 불리하며 생성된 고유식별자만으로 대략적인 고유식별자 생성 개수 등의 경향을 판단하기 힘들다는 단점이 있다. 또한 멀티스레드 활용 여부와 관계없이 시스템 및 컴파일 환경에 따라 할당되는 개별 고유식별자의 값이 달라지는 비결정적 요소가 있다는 것도 단점이다. 따라서 이런 방식의 경우 고유식별자끼리 서로 겹치지 않는다는 성질만 이용하고 최종 결과에는 개별 고유식별자로 인한 영향을 완전히 배제하는 경우에만 결정적 알고리듬을 구성할 수 있다. 따라서 시스템의 특성을 기반으로 하는 방식

의 고유식별자 생성은 결정적 병렬 프로그램을 작성하는데 있어서는 일반적으로 불리하다.

결정적 병렬성(deterministic parallelism)에 관한 연구의 필요성은 실용적 활용성을 중시하는 USENIX 학회에서 병렬성에 대한 주요 주제 탐구를 목적으로 하는 2006년 첫 HotPar 워크샵 논문에서도 제기되었다. Bochino Jr. 외 3인[7]은 전통적인 병렬 프로그래밍 모델의 스레드 스케줄링으로 인한 비결정성은 의도치 않은 동작과 해결하기 어려운 버그 발생의 요인이 되므로 결정적 병렬 프로그래밍 모델이 기본이 되고 비결정성은 필요한 경우에만 예외적으로 활용하는 방식의 접근이 일반화되어야 함을 강조하며 당시까지 시도된 다양한 접근방법을 소개하였다.

함수형 프로그래밍은 프로그램 진행 중 상태 변화에 의존을 최소화하여 스레드 간의 공유 상태를 관리할 여지가 적다는 점이 병렬화에 유리하다. 공유 상태가 없으면 상호배제 연산을 사용할 이유도 없어 효과적인 병렬화가 가능하다. 병렬화되지 않는 부분은 논리적 의존성 혹은 데이터 흐름의 의존성이 발생하는 경우뿐이다. 이러한 장점을 살려 하스켈 컴파일러 GHC에서는 전통적인 비결정적 스레드 조작 방식보다 고수준으로 추상화되어 결정적 병렬 프로그램을 작성하기 적합한 다양한 병렬 API를 제공하고 있다 [8, 9, 10] 우리의 이번 연구는 그중에서도 특정 프로그램식의 결과값을 병렬 계산을 요청하는 가장 기본적인 Eval 모나드를 활용하는 병렬화에 대한 벤치마크를 수행한 것이다.

함수형 프로그래밍에 한정하지 않고 최근 관련 주제에 관한 연구 동향을 알 수 있는 사례를 몇 가지 소개한다.

결정적/비결정적 병렬성과는 다른 각도에서 명시적/암시적 병렬성을 분류하기도 한다. 전통적으로 스레드와 상호배제를 관리하는 방식은 명시적인 비결정적 병렬성을 다루며, 우리 연구에서 활용하는 방식은 어떤 부분의 병렬화를 요청할지 소스코드에 드러나므로 명시적인 결정적 병렬성을 다룬다. 임의의 프로그램을 일반적으로 어떻게 병렬화하는 것이 효과적인지 판단하기는 대단히 어렵기 때문에 아무 제약 없이 암시적 병렬성은 실현하기 어렵다. 하지만 머신러닝과 같이 병렬화하기 좋은 특정 알고리듬을 활용하는 경우를 한정하면 그에 대한 암시적 병렬화를 분산 컴퓨팅 환경에서도 확장성 있게(scalable) 적용하는 연구 결과가 발표된 바 있다 [11].

전통적인 스레드와 상호배제 모델의 멀티스레드 활용이나 GPU 등을 활용한 병렬처리는 OpenMP, OpenCL 등과 같이 다양한 언어와 플랫폼에서 활용 가능한 API 혹은 프레임워크를 활용해 이식성 있는 프로그램 작성이 어느 정도 가능하다. 이보다 더 추상적인 고수준의 병렬 프로그래밍을 하기 위해 프로그래밍 패러다임이나 병렬화 모델에 따라 다양한 접근방식이 시도되고 있다. 하스켈과 같은 함수형 프로그래밍 언어에서 결정적 병렬성을 지원하는 방식들도 그러한 시도에 속한다고 볼 수 있는데, 특정 프로그래밍 언어나 패러다임을 기반이 아닌 더 범용적으로 활용 가능한 고수준의 병렬화 모델을 제공하려는 연구도 최근 발표되고 있다. Lohstroh 외 3인[12]은 고전적 '액터' 개념을 적절히 제약해 비결정성이 관리되는 '리액터'라는 개념을 바탕으로 멀티코어 머신과 분산 환경을 아우르는 결정적 병렬성 모델을 제안함으로써 하나의 공통된 어휘로 결정적 병렬 프로그래밍에 접근하는 야심찬 시도이다. Swalens 외 2인[13]은 이보다 좀 더 실용적인 접근으로 다양한 언어 및 플랫폼에서 활용되는 future, transaction, actor 등의 여러 가지 동시성 관리 혹은 동기화 모델을 Chocola라는 명확하고 엄밀하게 정의된 언어로 정의함으로써 두 가지 이상의 동기화 모델을 조합하여 사용하더라도 각각의 모델에서 보장하는 성질이 서로 충돌하지 않는 조합성을 보장하는 것을 목표로 하고 있다.

III. Lock-Free Unique Identifier using Modular arithmetic

1. Unique Identifiers for deterministic parallelism

우리가 고안한 방식은 Fig 1.처럼 여러 갈래의 병렬 작업에서 나머지 연산(modular arithmetic)으로 서로 겹치지 않게 고유식별자를 생성한다. 병렬로 분기하기 전에 1씩 증가시키는 방식(…, k-2, k-1, k)으로 고유식별자를 생성하다가 세 갈래의 병렬 작업으로 분기하면 각각 k, k+1, k+2에서부터 3씩 증가시키며 생성하고, 다시 한 갈래로 합쳐질 때는 각 병렬 작업의 마지막 상태의 최대값(max(x,y,z))으로부터 다시 1씩 증가시키며 고유식별자를 생성한다. 참고로 (k, i)의 의미는 다음번에 생성할 고유식별자가 k이며 i씩 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 고유식별자를 병렬적으로 생성해 나갈 수 있다.

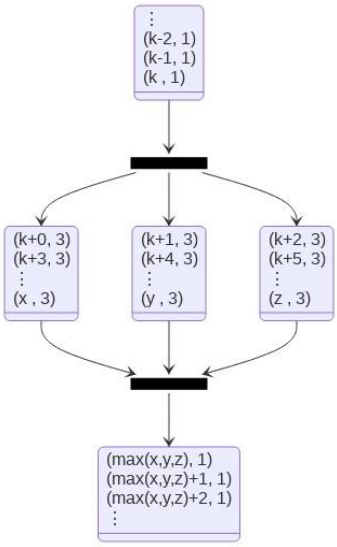

Fig. 2. Illustration of our lock-free unique udenifier generation based on modular arithmetic

또한 여러 스레드로 분기가 일어난 뒤 각 스레드의 계산 결과를 종합하여 하나로 합쳐지는 경우 분기하기 전과 마찬가지로 1씩만 증가하는 고유식별자가 생성되므로 사용되는 정수의 지나친 낭비를 막고 있다.

2. Changes to the Unbound-generics library

기존 Unbound-generics 라이브러리 중 이름 생성을 위한 fresh 함수는 Fig. 3과 같이 정의되어 있는데, 동작을 살펴보면 상태값에 단순히 +1 연산을 적용하는 방식으로 동작한다. 그러나 이러한 동작은 순차적으로 적용하는 방식이므로 병렬화를 하여도 순차적 의존성으로 인해 경쟁상태가 발생한다. 따라서 우리는 III-1 절의 아이디어를 이 라이브러리에 적용시켜 아래 Fig. 4와 같이 병렬 작업의 수에 따라 변경시킬 수 있도록 수정하였다. 우선 값을 저장하는 상태 St를 단일 값 k뿐만이 아니라 작업 수인 i까지 나타낼 수 있도록 (k,i) 형태로 변경하고, 이름 생성을 위한 fresh 함수에서 현재 값인 k만 가져오는 것이 아닌 현재 값 k에 병렬 작업 수인 i를 추가로 가져와 4행에서 (k+i, i) 연산을 진행하여 병렬 작업 중에도 고유식별자가 순차적 의존성에 영향을 받지 않고 생성될 수 있도록 하였다. 순차적 의존성을 해결하여 경쟁상황 없이 동작할 수 있도록 수정하였다.

```
1. instance Monad m =>
                Fresh (FreshMT m) where
2.     fresh (Fn s _) = FreshMT $ do
3.         n <- St.get
4.         St.put $! n + 1
5.         return $ (Fn s n)
6.     fresh nm@(Bn {}) = return nm
```

Fig. 3. Original unbound-generic library code using sequential increment

```
1. instance Monad m =>
                Fresh (FreshMT m) where
2.     fresh (Fn s _) = FreshMT $ do
3.         (k, i) <- St.get
4.         St.put $! ((,) $! (k + i)) $! i
5.         return $ (Fn s k)
6.     fresh nm@(Bn {}) = return nm
```

Fig. 4. Modified unbound-generics library code using modular arithmetic

IV. Performanc Benchmark

이번 절에서는 앞 절에서 설명한 unbound-generic 라이브러리 변경사항과 관련된 기존 싱글스레드 프로그램의 성능 저하 여부 및 매크로 확장 병렬화의 성능 개선 여부를 알아보기 위한 벤치마크 결과를 보고한다. 벤치마크 환경은 i7-9750H CPU(6코어 12스레드), 16G RAM 컴퓨터의 WSL2 가상화 환경의 Ubuntu 20.04 LTS에서 GHC 8.6.5가 설치된 환경에서 진행하였으며, 실행시간의 측정은 GHC 런타임에서 제공하는 -s 옵션을 활용해 프로그램 종료 직후 표준 출력으로 보고되는 내용을 기반으로 기록하였다.

1. Baseline comparison against sequential unique identifier generation

우선 III-2절에서 설명한 unbound-generic 구현에서 순차적 의존성을 해결한 변경사항이 싱글 스레드 프로그램의 성능에 어떠한 영향을 미치는지 알아보았다. 벤치마크 예시로 활용한 프로그램의 내용은 unbound-generics를 이용해 반복적으로 새로운 이름 생성을 순차적으로 반복함으로써 고유식별자 생성을 천만 회 이상 단위로 반복하는 단순한 작업이다.

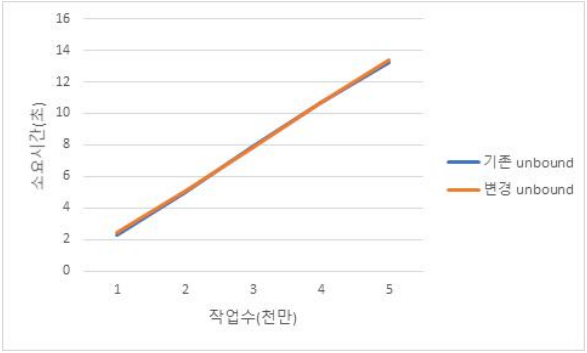

Fig. 5. Single thread execution benchmark, comparing the performance before and after changes to the unbound-generics code (see Fig. 3 and 4)

Fig. 5를 보면 기존 대비 우리가 변경한 라이브러리의 실행 소요 시간이 거의 동일한 것을 볼 수 있다. 따라서 변경사항이 기존의 순차적 실행환경에서 성능에 악영향을 거의 미치지 않고 실행 가능하다는 것을 확인할 수 있다.

```
1. eval 0 (Plus e1 e2)  = do
2.       Const n1 <- eval 0 e1
3.       Const n2 <- eval 0 e2
4.       return $ Const (n1 + n2)
5. eval k (Plus e1 e2)  = do
6.       e1' <- eval k e1
7.       e2' <- eval k e2
8.       return $ Plus e1' e2'
```

Fig. 6. Code snippet of the eval function (sequential version) for Plus operation

```
1. ev runM 0 (Plus e1 e2)  = do
2.     [Const n1, Const n2] <- runParT runM
                        $ ev runM 0 <$> [e1, e2]
3.     return $ Const (n1 + n2)
4. ev runM k (Plus e1 e2)  = do
5.     [e1', e2'] <- runParT runM
                        $ ev runM k <$> [e1, e2]
6.     return $ Plus e1' e2'
```

Fig. 7. Code snippet of the ev function (parallel version) for Plus operation

2. Case study: Parallel macro-expansion

이제는 병렬화를 가정하고 만들어진 프로그램에 대하여 병렬화 성능에 대한 차이를 알아보고자 한다. 이를 위해 unbound-generic 라이브러리를 이용하여 간단한 매크로 확장 시스템을 멀티 스테이지 프로그래밍[10] 기반으로 작성하고, 성능 평가를 진행하였다. 작성한 매크로 확장 시스템은 아래와 같은 형태이다.

Fig. 6, 7은 각각 병렬화를 고려하지 않은 멀티스테이지 함수 eval과 병렬화를 위해 수정된 버전의 ev의 정의 중 덧셈(Plus)식을 처리하는 부분만 발췌한 내용이다. 0단계에서는 실제 값의 계산이 이루어지고, k단계에서는 소스코드 데이터의 형태로 확장이 이루어진다. 이러한 eval과 ev 함수로 $x^0 + x^1 + \cdots + x^y$ 형태의 거듭제곱의 합으로 이루어진 다항식 계산을 벤치마크하였다. 이러한 다항식을 구성하기 위해 효율적인 거듭제곱 소스코드를 생성하는데 활용한 재귀적 매크로 함수 exp의 정의를 람다식과 멀티스테이지 프로그래밍 연산자 〈 〉와 ~를 이용해 아래와 같은 슈도코드로 나타낼 수 있다.

```
exp = \x y ->
        if (y < 1) then 〈 1 〉
        else if (evn y)
                then 〈 (\v -> v*v) ~(exp x (y/2)) 〉
                else 〈 x * ~(exp x (y + -1)) 〉
```

위에서 사용된 멀티스테이지 연산자 〈e〉는 그 안의 식 e의 단계를 하나 높이며 반대로 ~e는 하나 낮춘다. 직관적 의미는 〈e〉는 소스코드 e를 문법분석 단계까지만 처리된 추상문법트리(AST) 데이터로 유지하여 즉시 계산하지 않는다. 반대로 ~〈e〉는 0단계에서는 코드 데이터인 〈e〉를 컴파일해 식 e의 값 계산을 강제하되 1단계 이상에서는 소스코드를 확장하는 데 그친다. 이를 이용하면 항상 고정된 식으로 계산하는 대신 입력에 따라 더 효율적인 소스코드를 생성해 계산할 수 있다. 위의 정의에서는 x^{2n}형태의 짝수 거듭제곱의 경우 x^n을 계산한 결과값 v를 자기 자신과 곱하여 계산 과정에서 곱셈의 횟수를 줄이고 있다. 우리가 수행한 벤치마크는 exp의 입력 중 x=2로 고정하고 y=0~2000까지의 합을 구하는 식을 eval과 ev을 통해 계산했을 때 허용하는 최대 CPU 코어 수에 따른 수행 시간을 비교한 것으로, 그 결과 그래프는 Fig. 8과 같다.

eval 함수의 경우 병렬화가 전제되지 않은 코드이기 때문에 스레드 수가 늘어날수록 불필요한 스레드 관리의 오버헤드로 인해 실행시간이 늘어나는 반면, ev함수의 경우에는 스레드가 늘어나며 효과적인 병렬화로 실행시간이 감소한다. 스레드 수가 아주 적을 때는 더 이상 병렬화할 코어 자원이 없는데도 일어나는 병렬화 요청을 거절해야 하는 오버헤드로 인해 ev가 eval보다 실행시간이 길지만 코어 수를 늘림에 따라 곧 역전된다.

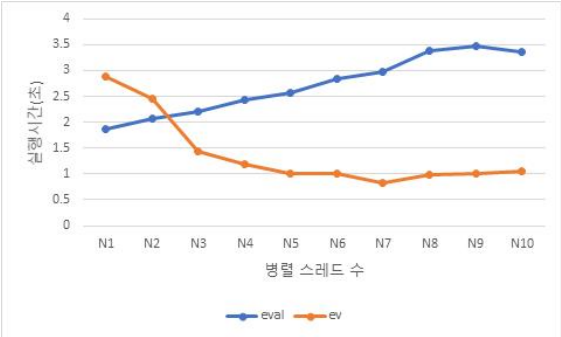

Fig. 8. eval vs. ev benchmark expanding exponentiation in the expression 1+x+x^2+...+x^2000$1 + x^1 + x^2 + \cdots + x^{2000}$

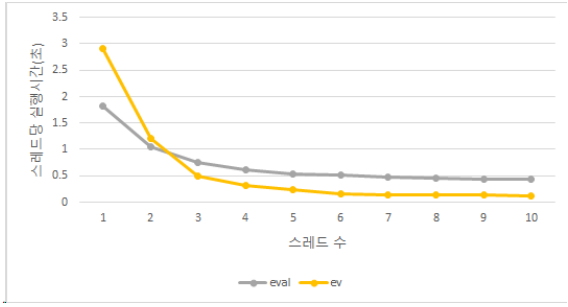

Fig. 8-2. eval vs. ev benchmark expanding exponentiation in the expression $1 + x^1 + x^2 + \cdots + x^{2000}$

참고로 Fig. 8은 매크로 확장과 실제로 덧셈과 곱셈 연산이 수행되는 시간을 함께 벤치마크한 결과인데, 대부분의 시간이 매크로 확장에 소요된다. 즉, Fig. 8에서 병렬화로 개선되는 효과가 그래프로 나타나는 대부분이 매크로 확장의 성능 개선에 대한 변화를 반영한다. 이를 확인하기 위한 보조적 벤치마크(Fig. 9)를 수행하였다.

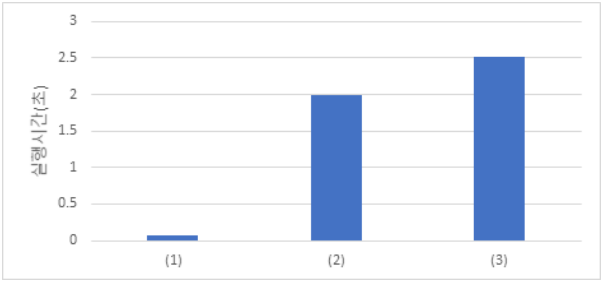

Fig. 9. benchmark macro expansion in the expression $1 + x^1 + x^2 + \cdots + x^{2000}$

Fig. 9는 싱글 스레드 eval로 x^0에서 x^{2000}까지의 거듭제곱을 더하는 프로그램의 세 가지 버전을 실행한 시간에 대한 그래프이다. (1)은 미리 exp에 해당하는 매크로 확장을 완료한 상태로 덧셈과 곱셈 계산만 하는 데 걸리는 시간, (3)은 매크로 확장 없이 단순하게 x^n을 n번의 곱셈으로 계산하는 시간을 나타내며, 가운데의 (2)는 대조군으로 Fig. 8의 eval의 N1에 해당한다. (1)의 결과는 매크로 확장에 비해 순수한 연산에 걸리는 시간의 비중이 매우 낮음을 확인할 수 있다. 하지만 매크로 확장으로 효율적인 거듭제곱 공식으로 변환하지 않고 순진무구(naive)한 알고리듬으로 강행하면 (3)의 결과와 같이 비효율적이므로 여러 번 수행할 연산식이면 효율적 알고리듬으로 한번 매크로 확장을 해 놓고 수행하는 것이 효과적이라는 것도 추가로 확인할 수 있다.

V. Conclusion and Future work

나머지 연산(modular arithmetic)을 활용한 고유식별자를 할당 방식은 다음과 같은 장점이 있어 다양한 방식의 병렬 프로그램에 적용할 수 있다. (1) Lock이 필요하지 않기 때문에 Lock으로 인한 병목현상을 피할 수 있고, (2) 병렬 작업이 요구하는 식별자 개수를 미리 파악할 필요가 없으며, (3) 분기된 병렬 작업 내에서 재분기하는 경우에도 유연하게 활용가능하다. 이 논문에서 재귀적 매크로 확장을 활용하는 벤치마크를 통해 이러한 장점을 확인하였다.

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

함수형 프로그래밍 언어의 라이브러리도 순차성에 의존하는 구현이 병렬화에 문제가 될 수 있다. 하지만 잘 작성된 순수 함수형 라이브러리의 경우 부수효과 등을 잘 관리하는 함수형 패러다임의 관례에 따라 순차성에 의존하는 부분이 잘 모듈화되어 있을 가능성이 크다. 따라서 적절한 병렬화 방식만 찾는다면 구현에서 순차성을 관리하는 부분만 제한적으로 변경함으로써 이 연구의 사례로 개선한 unbound-generics와 같이 기존 싱글 스레드 코드의 성능에 거의 영향을 미치지 않으면서 병렬화에 적합한 구현으로 개선가능한 경우가 많을 것으로 생각된다.

이후 연구로는 프로그램식의 일부를 병렬화하는 방식 외에 데이터 의존성의 특정 분기점을 기반으로 병렬화하는 데이터 흐름 병렬화 방식에도 이 연구에서 제안한 고유생성자 할당이 효과적인지 벤치마크를 통해 확인하고자

한다. 이후 다양한 사례를 통해 안정적인 성능이 확인되면 특정 라이브러리의 내부 소스코드를 수정해 벤치마크하는 것에 그치지 않고 재사용할 수 있도록 패키징된 라이브러리로 구성하여 배포하는 것도 향후 연구 목표 중 하나다.

ACKNOWLEDGEMENT

This work was supported by 2019 Hannam University Research Fund

REFERENCES

[1] Paul J. Leach, Michael Mealling, and Rich Salz. “A Universally Unique IDentifier (UUID) URN Namespace”, DOI : 10.17487/RFC4122

[2] K.Y. Ahn, “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, pp. 1-9, Feb 2021. DOI : 10.9708/jksci.2021.26.02.001

[3] H. Kim, H. An, S. Byun, and G. Woo. “Tuning the Performance of Haskell Parallel Programs Using GC-Tune,” *KIISE Transactions on Computing Practices*, 23(8), pp. 459-465, August 2017. DOI : 10.5626/KTCP.2017.23.8.459

[4] Y. Kim, J. Cheon, M Kim, I. Wang, S. Byun, G. Woo. "A Shared-Memory based Haskell Parallel Programming Model for the Manycore Environments", *Proceedings of the Korean Information Science Society Conference*, pp. 1554-1556, Korea, June 2021. DOI : 10.5626/KTCP.2017.23.8.459

[5] S. Weirich, B. A. Yorgey, and Tim Sheard. “Binders unbound,” *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming*, ICFP 2011, Tokyo, Japan, September 19-21, 2011, pp. 333-345, ACM, 2011. DOI: 10.1145/2034574.2034818

[6] E. A. Kmett (maintainer), “unique: Fully concurrent unique identifiers”, ver. 0.0.1, 2021-01-04 (revised 2021-11-10), URL: https://hackage.haskell.org/package/unique, accessed 2021-11-25

[7] R. L. Bocchino, V. S. Adve, S. V. Adve, and M. Snir. “Parallel programming must be deterministic by default”, *HotPar'09: Proceedings of the First USENIX conference on Hot topics in parallelism*, March 2009, pp 4.

[8] S. Marlow, P. Maier, H.W. Loidl, M. K. Aswad, and P. Trinder, “Seq no more: better strategies for parallel Haskell”, *ACM SIGPLAN Notices* 45(11), November 2010, pp. 91–102, DOI : 10.1145/2088456.1863535

[9] S. Marlow, R. Newton, and S. Peyton Jones, “A monad for deterministic parallelism”, *Haskell '11: Proceedings of the 4th ACM symposium on Haskell*, September 2011, pp. 71–82, DOI : 10.1145/2034675.2034685

[10] N. Krauter, P. Raaf, P. Braam, R. Salkhordeh, S. Erdweg, and A. Brinkmann, “Persistent software transactional memory in Haskell”, *Proceedings of the ACM on Programming Languages*, Vol. 5, Issue ICFP, August 2021, Article No.: 63, pp. 1–29, DOI : 10.1145/3473568

[11] M. Bauer, W. Lee, E. Slaughter, Z. Jia, M. D. Renzo, M. Papadakis, G. Shipman, P. McCormick, M. Garland, and A. Aiken. “Scaling implicit parallelism via dynamic control replication“, *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 105-118, Feb 2021. DOI : 10.1145/3437801.3441587

[12] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee. “Toward a Lingua Franca for Deterministic Concurrent Systems”, *ACM Trans. Embed. Comput. Syst.*, Vol. 20, No. 4, pp. 1-27, June 2021. DOI : 10.1145/3448128

[13] J. Swalens, J. D. Koster, and W. D. Meuter, “Chocola: Composable Concurrency Language“, *ACM Trans. Program. Lang. Syst.*, Vol. 42, No. 4, pp. 1-56, Feb 2021. DOI : 10.1145/3427201

Authors

Bum-Jun Son received the B.S. degrees in Hannam University, Korea, in 2020 He is currently pursuing the M.S degree in Computer Engineering, Hannam University, Korea

languages, type systems, and parallel computing

Ki Yung Ahn joined the faculty of the Department of Computer Engineering at Hannam University, Daejeon, Korea, in 2018. His research interests are parallelism, functional programming, programming

languages, type systems, and cyroptographic protocols.