OpenMP를 활용한 LSH DRBG 병렬 최적 구현

권혁동* 안규황* 서화정*†

*한성대학교 대학원 정보시스템공학과

hdgwon@naver.com tiger9212@gmail.com hwajeong84@gmail.com

LSH DRBG parallel optimization using OpenMP

Hyeok-Dong Kwon* Kyu-Hwang An* Hwa-Jeong seo*†

*Information system engineering, Hansung University of Graduate school.

요 약

결정론적 난수 발생기(Deterministic Random Bit Generator, DRBG)는 미국 표준 기술 연구소(National Institute of Standards and Technology, NIST)에서 권고하는 결정론적 난수 발생 알고리즘으로 입력에 따라 의사난수를 생성하는 알고리즘이다. DRBG의 내부에는 의사난수함수를 사용하는데, 아직까지 표준으로 제정 및 구현된 적 없는 LSH DRBG를 구현하며, 더 나아가 DRBG의 규격 별로 또는 DRBG의 함수 중 유도함수와 내부출력생성함수에서 반복적으로 호출 되는 의사난수함수를 OpenMP를 통해 병렬화 처리를 적용하여 최적화를 하고자 한다.

I. 서론

해시 함수 입력 값의 길이와 무관하게 고정된 길이의 임의 값을 출력하는 함수로 이를 활용한 알고리즘 중 하나가 난수 발생기이다. 난수 발생기는 무작위 값을 제작하지만 입력에 따라 출력 값이 고정된 결정론적 성질을 가지며 내부의 의사난수함수를 교체하는 것으로 출력을 변경할 수 있다.

일반적인 프로그램은 코드의 상단부터 절차적으로 진행된다. 그러나 코드의 중간 결과물이 서로 독립적이라면 각 출력물을 대기하지 않고 병렬적으로 동작하는 구조가 효율적일 수 있다. 이와 관련된 예시가 fig. 1으로, 동일한 함수 func1, func2를 호출한다고 가정하면 일반적인 프로그램은 func1이 종료되어야 func2가 실행될 수 있다. 하지만 병렬화를 적용한 프로그램의 경우에는 func1과 func2가 동시에 동작하기에 전체 동작 시간이 줄어들게 된다.

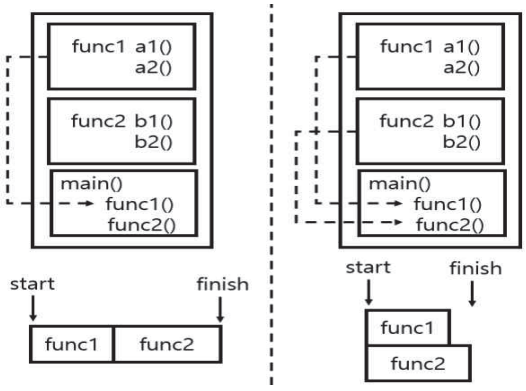

Fig. 1. 좌) 일반적인 프로그램의 구조
우) 병렬화가 적용된 프로그램의 구조

본 논문에서 사용할 결정론적 난수 발생기(deterministic random bit generator, DRBG)는 LSH를 사용하여 총 6개 규격을 지원하기에 각 규격 출력에 병렬화 시도를 한다. 또한 동작 중에는 의사난수함수를 반복적으로 호출하는 과정이 있으므로 내부 동작에도 병렬화를 적용하여 최적화를 구현한다.

II. 관련 연구 동향

2.1. 결정론적 난수 발생기(Deterministic Random Bit Generator)

일반적으로 난수는 예측 불가능한 무작위의 수를 의미하나 실제로는 컴퓨터에서 완전 난수를 생성하기 어렵기 때문에 입력 값에 따라 출력이 정해진 결정론적으로 구성된 난수 발생기를 의미한다.

일반적으로 내부에 적용하는 의사난수함수의 종류에 따라 해시기반 DRBG와 HMAC기반 DRBG로 나뉘지만 둘의 내부 구조는 다르며 해시기반 DRBG가 병렬화 기법을 적용하기 유리하기 때문에 본 논문에서 언급하는 모든 DRBG는 해시기반 DRBG임을 명시한다.

2.2. LSH(Lightweight Secure Hash)[2]

LSH는 국가보안기술연구소(National Security Research Institute, NSR)에서 2014년에 개발한 암호학적 해시 함수로 wide-pipe merkle Damgård 구조가 적용되었다. 출력 규격으로는 224, 256, 384, 512비트를 제공한다. 성능면에서 SHA2, SHA3에 비해 약 2배 이상 성능이 뛰어남을 보였으며 현재 국내 TTA 표준으로 제정되어 있다.

2.3. SSE(Steaming Simd Extensions)[3]

SSE는 Intel사에서 제공하는 병렬화 기법 중 하나로 1999년에 출시된 펜티엄3부터 사용 가능하며, SIMD(Single Instruction Multiple Data)에 속한다. SIMD는 하나의 명령어로 절차지향적으로 수행될 데이터를 한 번에 처리할 수 있기에 전체적인 동작속도를 향상시킨다.

2.4. AVX(Advanced Vector Extension)[4]

AVX는 2008년에 발표된 SIMD 명령어 집합으로 기존 16바이트에서 증가한 32바이트를 한 번에 계산할 수 있다. 특히 AVX는 부동소수점 연산이 뛰어나기 때문에 이와 관련된 모델링, 실험 등의 분야에서 유용하게 활용할 수 있다.

III. 제안 기법

구현한 LSH DRBG에 병렬화를 적용하는 방법에는 데이터 병렬화와 태스크 병렬화 두 가지 방법이 있으며 본 장에서는 각각의 적용 방법을 서술한다.

3.1. 데이터 병렬화(Data Parallelism)

LSH는 총 6개의 규격을 지원하며 프로그램 구조상 후행 규격은 선행 규격의 출력을 대기하게 된다. 따라서 출력을 대기하는 시간이 소요된다. 이러한 구성은 대기 시간이 누적되어 작업 시간을 증가를 초래한다. 하지만 DRBG의 각 출력 값은 서로 독립적이기 때문에 선행 규격의 완료 여부와 상관없이 작업을 진행해도 무방하다. 단, 각 규격별로 작업 시간이 동일하지는 않기에 프로그램이 종료되기 위해서는 모든 규격이 완료되기를 기다려야 한다. 이것을 적용한 모델이 fig. 2로 각 규격별로 병렬적으로 동작하며 모든 규격이 종료되기를 대기하는 시간이 새롭게 포함된다. 새롭게 생긴 대기 시간은 기존의 누적되는 대기 시간에 비해 짧을 것으로 예상되기에 동작 소요 시간을 월등히 절감할 것으로 기대할 수 있다.

Algorithm 1

1. read test vectors
2. loop count = number of hash output types
3. operate OpenMP to for loop
4. For i = 1 to loop count
   - 4.1 DRBG using hash[i] type
   - 4.2 write DRBG result
   - 4.3 waiting until other processing finished
5. DRBG finished

Fig. 2. 데이터 병렬화가 적용된 LSH DRBG

3.2. 태스크 병렬화(Task Parallelism)

DRBG의 구조 중에서 유도함수와 내부출력생성함수는 내부에서 반복적인 해시 연산을 수행하며 반복 산정 기준은 함수마다 다르다. 유도함수의 반복 횟수 산정에는 규격별로 정해진 상수 값만이 사용되기 때문에 2회 또는 3회로 고정된다. 반면 내부출력생성함수의 반복 산정

기준은 출력 길이에 비례하기 때문에 내부출력 생성함수의 반복 횟수는 출력 길이를 변경하는 것으로 조절이 가능하다. 두 함수 자체는 독립적이며 또한 내부의 해시 연산 작업도 서로 영향을 주지 않기 때문에 병렬화 적용이 가능하며 fig. 3, fig. 4와 같은 모습을 가진다.

Algorithm 2

1.1 If hash bits is 224 or 256
  seed bits = 440
1.2 If hash bits is 384 or 512
  seed bits = 888
2. len = ceil (seed bits / hash bits)
3. counter = 0x01
4. input = counter || seed bits || msg from parameter
5. operate OpenMP to for loop
6. For i = 1 to len
  6.1 output[i] = Hash(input)
  6.2 counter + = 1
  6.3 refresh input value
  6.4 waiting until other processing finished
7. final output = (output[1] || … || output[n]) mod $2^{\text{seed bits}}$

Fig. 3. 태스크 병렬화가 적용된 유도함수

Algorithm 3

1. len = ceil (output bits from parameter / hash bits)
2.1 If hash bits is 224 or 256
  seed bits = 440
2.2 If hash bits is 384 or 512
  seed bits = 888
3. data = state V from parameter
4. operate OpenMP to for loop
5. For i = 1 to len
  5.1 output[i] = Hash(data)
  5.2 data = (data + 1) mod $2^{\text{seed bits}}$
  5.3 waiting until other processing finished
6. final output = (output[1] || … || output[n]) mod $2^{\text{output bits}}$

Fig. 4. 태스크 병렬화가 적용된 내부출력생성함수

IV. 성능평가

본 장에서는 데이터 병렬화와 태스크 병렬화에 따른 성능 변화를 비교할 것이며 DRBG 구현 및 작업 환경은 다음 table. 1과 같다.

Table. 1. 작업 환경

OS	Windows 10 Pro
Processor	Intel Core i7-8550U CPU 1.8GHz
Compiler	MinGW
IDE	Eclipse Photon (4.8.0)
Language	C

테스트에 사용된 데이터와 DRBG 구현 작업물은 깃허브[1]에서 열람이 가능하다.

4.1 데이터 병렬화 성능 평가

각 규격별 동작에 대해 병렬화를 적용하여 6개 규격이 동시에 출력을 진행한다. 6개 규격 출력이 완료된 시점이 1회이며 규격별로 실험 환경은 테스트벡터 60종을 1000회에 반복해서 출력하게 설정하며 예측내성 지원모델로 동작한다. 대조군은 병렬화를 적용하지 않고 동일하게 동작하며 실험 결과는 table. 2와 같다.

Table. 2. 데이터 병렬화 비교

	평균수행시간(ms)	cpb
대조군	73	491
병렬화	27	181

대조군의 수행 시간을 CPB(Clock cycle Per Bytes) 단위로 환산하면 약 491cpb가 나온다. 반면 병렬화를 적용한 경우에는 약 181cpb의 효율을 보여 약 2.71배의 성능 향상을 보인다. 대조군은 출력 대기 시간이 누적되지만 병렬화 모델은 그렇지 않기에 그만큼 빠른 동작을 보장한다.

4.2 태스크 병렬화 성능 평가

DRBG의 내부 구조상 병렬화가 적용 가능한 두 부분에 병렬화를 적용하여 해시 연산이 동시에 이뤄지게 한다. 4.1절과 동일한 환경에서 진행하며 마찬가지로 모든 규격이 완료된 시점을 1회로 한다. 실험 결과는 table. 3과 같다.

Table. 3. 태스크 병렬화 비교

	평균수행시간(ms)	cpb
대조군	73	491
병렬화	149	1001

대조군은 데이터 병렬화의 대조군과 유사한 결과로 약 491cpb를 보여준다. 반면 병렬화 적

1) https://github.com/korlethean/drbg_with_mp

용 모델은 1001cpb가 도출되며 약 2.04배 가량 부족한 성능을 보인다.

이는 병렬화 과정의 오버헤드로 성능이 저하되는 것으로 판단되었다. 병렬화 과정 중에 오버헤드의 발생은 필연적이기 때문에 병렬화 효율을 높이기 위해서는 반복의 횟수를 증가시켜야 한다. DRBG 구조상 출력 길이 조절로 내부 출력생성함수의 반복 횟수를 늘릴 수 있으며 이에 따라 유도함수 내부는 통상적인 동작으로 수정하고 출력 길이를 점진적으로 늘려가며 실험한 결과는 fig. 5와 같다.

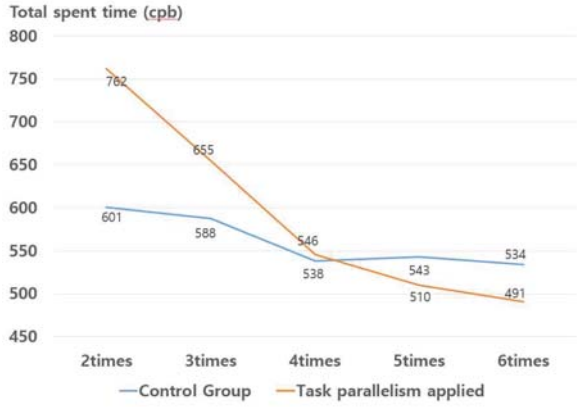

Fig. 5. 출력 길이별 비교 그래프

출력 길이가 기존대비 2, 3배인 경우에는 대조군의 효율이 뛰어나지만 4배의 경우엔 근소한 차이만을 보이며 5배 이후부터 병렬화 모델이 뛰어남을 보인다. 또한 병렬화 모델은 출력 길이가 증가함에도 동작 효율의 변화가 크게 변하지 않음을 확인할 수 있다.

V. 결론

본 논문에서는 DRBG의 구조상에서 병렬화 모델을 적용하여 최적화 구현을 시도하였고 그 성능이 통상적인 DRBG 모델보다 뛰어남을 확인하였다.

데이터 병렬화를 적용할 경우 여러 규격의 DRBG를 동시에 가동할 수 있기에 다수 규격 출력에 유리함을 보인다. 반면 태스크 병렬화의 경우에는 표준 규격 상에서는 효과적이지 못하나 일정 출력 길이 이상에서는 병렬화 적용이 뛰어남을 보였다. 이는 표준과는 다르기 때문에 실제로 사용하기는 어렵지만 이론상으로 DRBG에 태스크 병렬화 적용이 가능함을 보였다.

각종 병렬화 기법은 다양한 영역에서 활용된다. 이때 단지 병렬화 기법을 적용하는 것만으로 성능 향상을 기대하게 되나 본 논문의 실험 결과와 같이 오버헤드로 인해 성능이 저하되는 일이 발생할 수 있다. 따라서 병렬화 기법을 맹목적으로 적용하여 성능 향상을 꾀하려 하지 않으며 통상적인 동작과 저울질 끝에 정확한 적용을 시도하여야만 성능 향상을 이끌어 낼 수 있다.

[참고문헌]

[1] National Institute of Standards and Technology, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators", Available: https://www.nist.gov/publications/recommendation-random-number-generation-using-deterministic-random-bit-generators-2

[2] Telecommunications Technology Association, "Hash function Full structure and compression function of LSH", Available: http://commitee.tta.or.kr/data/standard_view.jsp?nowPage=2&pk_num=TTAK.KO-12.0276&commit_code=TC5.

[3] Srinivas K. Raman, Vladimir Pentkovski, Jagannath Keshava, "Implementing Streaming SIMD Extensions on the Pentium III processor", Available: https://www.computer.org/csdl/mags/mi/2000/04/m4047.pdf

[4] Chris Lomont, "Introduction to Intel® Advanced Vector Extensions", Available: https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf